Engineering Algorithms for Finding Patterns in Biological Data
Jesper Nielsen

PhD Dissertation

Department of Computer Science
Bioinformatics Research Centre
Aarhus University
Denmark
Engineering Algorithms for Finding Patterns in Biological Data

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Jesper Nielsen
July 21, 2011
Abstract

This Ph.D. dissertation presents contributions to a wide range of topics within the field of bioinformatics. In particular it contains work on the computation of molecule similarity, on association mapping, on analysis of molecular biological data, on Hidden Markov Models and on the computation of the quartet distance between trees. The common denominator is the interest in the design of both theoretically and practically fast algorithms.

Related to molecular similarity we show how arranging a set of bitstrings into a tree can significantly speed up searches for bitstrings similar to a query bitstring. Similarity is measured by the Tanimoto coefficient, and the speed-up is achieved by pruning subtrees, if it can be shown that any leaf fingerprint in the subtree will be dissimilar to the query. Furthermore we demonstrate how an inverted index can improve the computation time of a matrix of LINGOsim similarity scores.

Association mapping is a technique based on using large amounts of data on Single Nucleotide Polymorphisms (SNPs) to statistically infer associations between segments of DNA and effects in the host. Within the area of association mapping we develop an efficient file format and software library, called SNPFile. The file format is able to store both large amounts of SNP data and associated metadata, such as ids and affected-status of samples. Thus the file format can both speed-up SNP data access and simplify data management significantly.

On the topic of molecular biological data, we analyze data from an experiment on exosome knockout. The exosome is a complex with a role in RNA degradation. We find that knockout of the exosome stabilize hitherto unknown RNA transcripts upstream active transcription start sites.

With respect to Hidden Markov Models we develop two fast algorithms. We show that formulating the Hidden Markov Model framework using linear algebra allows one to solve the traditional problems using parallel reduction. This approach increases the total amount of work, but can still lead to significant speed-ups on Hidden Markov Models with a small state space. For Hidden Markov Models with a large state space we demonstrate a coarse-to-fine $k$-best Viterbi algorithm that might be used to approximate the forward algorithm.

Finally we present work on the computation of the quartet distance. There are many algorithms for generating trees, and for the systematic study of differences between these algorithms, a way to measure the distance between trees is useful. The quartet distance is such a measure, based on counting dissimilar quartet substructures between two trees. We present a new algorithm for computing the quartet distance and show that it is fast in theory and practice.
Resumé

Denne Ph.d. afhandling præsenterer bidrag til en lang række emner inden for bioinformatikken. Specielt beskriver den arbejde indenfor molekylesammenligning, associationskortlægning, analyse af molekylærbiologisk data, skjulte markov modeller og beregning af kvartetafstanden imellem to træer. Fællesnævneren er interessen i at designe algoritmer der er hurtige, både i teori og praksis.

Relateret til molekylesammenligning viser vi hvordan søgninger efter bitstrenge, lignende en forespørgselsbitstreng, kan gøres signifikant hurtigere ved at ordne bitstrengene i et træ. Lighed er her målt med Tanimoto koefficienten og hastighedsforøgelsen opnåes ved at beskære deltræer, hvis det kan vises at ethvert blad i deltræet er ulig forespørgslen. Desuden demonstreres det hvordan et inverteret indeks kan forbedre beregningstiden af en matrice af LINGOsim lighedsværdier.

Associationskortlægning er en teknik baseret på at bruge store mængder af data om enkeltnukleotidpolymerfier (SNPer) til statistisk at udlede sammenhænge imellem segmenter af DNA og effekter i en organisme. Inden for associationskortlægningen udvikler vi et effektivt filformat og software bibliotek, kaldet SNPFile. Filformatet er i stand til at gemme både store mængder af SNP data og tilhørende metadata, såsom id og status på prøverne. På den måde kan filformatet både øge hastigheden af tilgang til SNP data og simplificere datahåndtering betydeligt.

Angående molekylærbiologisk data analyserer vi data fra et eksperiment om knockout af exosomet. Exosomet er et kompleks med en rolle i nedbrydning af RNA. Vi finder at knockout af exosomet stabiliserer hidtil ukendte RNA transskriptioner opstrøm aktive transskriptionsstartsteder.

I forhold til skjulte markov modeller udvikler vi to hurtige algoritmer. Vi viser at en omformulering af skjulte markov modeller til lineær algebra tillader en at løse de traditionelle problemer via parallel reduktion. Denne tilgang øger den samlede arbejdshyde, men kan stadigvæk føre til signifikante hastighedsforøgelser på skjulte markov modeller med et lille tilstandsrum. For skjulte markov modeller med et stort tilstandsrum demonstrerer vi en grov-til-fin k-bedste Viterbi algoritme, der muligvis kan bruges som en tilnærmelse til forward algoritmen.

Til sidst præsenterer vi arbejde omkring beregning af kvartetafstanden. Der findes mange algoritmer til at generere træer og for at studere disse systematisk er det praktisk at have et mål for afstanden imellem to træer. Kvartetafstanden er sådan et mål, baseret på en optælling af antallet af kvartet delstrukturen der
er forskellige imellem de to træer. Vi præsenterer en ny algoritme til at beregne kvartetafstanden og viser at den er hurtig i teori og praksis.
Acknowledgments

First I thank my adviser Christian N.S. Pedersen for all the work he has done for me, and all the help he has given me during my studies at the Bioinformatics Research Centre.

Besides my adviser I thank my other co-authors and colleagues who have worked with me and made me become productive as a Ph.D. student. At the Bioinformatics Research Centre these include Thomas Mailund, Mikkel Heide Schierup, Thomas Greve Kristensen and Andreas Sand Pedersen. At the department of Molecular Biology these are Pascal Preker and Torben Heick Jensen.

I also thank my other colleagues at the Bioinformatics Research Centre, whom I might not have worked directly with, but who have nevertheless been a great inspiration and good friends. The closest of these include Asbjørn Tølbøl Brask, Qian Yu, Martin Simonsen, Iwona Siuda and Paula Tataru. They have helped keep me sane and my life interesting.

I thank professor Yun S. Song and his research group at the University of California, Berkeley, for accepting me for my study abroad, and for advising me and working with me while there. I also thank the International House in Berkeley, and in particular all the friends I had there, for making the stay a positive and unforgettable experience.

Finally I thank my parents and sister for supporting me, and for taking a large part in making me who I am today.

Jesper Nielsen,
Aarhus, July 21, 2011.
# Contents

Abstract v

Abstract vii

Acknowledgments ix

I Overview 1

1 Introduction 3

2 Molecular Screening 7
  2.1 Background ................................................. 7
  2.2 Searching for similar molecules .......................... 9
      2.2.1 Related Work ........................................... 9
      2.2.2 Research Contributions ............................... 10
      2.2.3 Future Work ........................................... 14
  2.3 Fast computation of the LINGOsim ....................... 14
      2.3.1 Related Work ........................................... 15
      2.3.2 Research Contributions ............................... 15
      2.3.3 Future Work ........................................... 17

3 Association Mapping 19
  3.1 Background .................................................. 19
  3.2 Related Work ............................................... 21
  3.3 Research Contributions .................................... 22
  3.4 Future Work ............................................... 23

4 Analyzing Molecular Biological Data 25
  4.1 Background .................................................. 25
  4.2 Related Work ............................................... 26
  4.3 Research Contributions .................................... 27

5 Hidden Markov Models 31
  5.1 Background .................................................. 31
      5.1.1 The Forward Algorithm ............................... 32
      5.1.2 The Viterbi Algorithm ............................... 33
10 SNPFile – A software library and file format for large scale
association mapping and population genetics studies 89
10.1 Background ........................................ 91
10.2 Results and Discussion .......................... 92
  10.2.1 Simple and efficient genotype data manipulation . 92
  10.2.2 Framework for arbitrary secondary data .......... 94
  10.2.3 Using SNPFile .................................. 94
10.3 Conclusions ........................................ 95
10.4 Methods ........................................... 95
  10.4.1 File manipulation ............................. 95
  10.4.2 Accessing genotype data ........................ 96
  10.4.3 Accessing secondary data (meta data) .......... 99
  10.4.4 Script access to SNPFiles ...................... 105
10.5 Availability and requirements ...................... 105
10.6 Abbreviations ...................................... 106
10.7 Competing interests ................................ 106
10.8 Authors’ contributions ............................ 106
10.9 Acknowledgements ................................ 106

11 RNA exosome depletion reveals transcription upstream of active human promoters 107
11.1 Acknowledgements ................................ 115

12 RNA polymerase plays both sides: Vivid and bidirectional transcription around and upstream of active promoters 117
12.1 Acknowledgements ................................ 122

13 Algorithms for a parallel implementation of Hidden Markov Models with a small state space 123
13.1 Introduction ...................................... 125
13.2 Methods .......................................... 126
  13.2.1 The parredForward algorithm .................. 127
  13.2.2 The parredViterbi algorithm ................... 130
13.3 Results ........................................... 133
13.4 Conclusion ....................................... 136
13.5 Acknowledgment ................................... 136

14 A Coarse-to-Fine Approach to Computing the $k$-Best Viterbi Paths 137
14.1 Introduction ...................................... 139
14.2 Methods .......................................... 140
  14.2.1 The Viterbi algorithm ......................... 141
  14.2.2 Coarse-to-fine ................................ 141
  14.2.3 $k$-best ..................................... 142
  14.2.4 Coarse-to-fine $k$-best ......................... 143
  14.2.5 Building $T$ ................................... 144
14.3 Results .......................................... 145
15 A sub-cubic time algorithm for computing the quartet distance between two general trees

15.1 Background

15.2 Methods: A sub-cubic time and space algorithm

15.2.1 Preprocessing

15.2.2 Counting shared butterfly topologies

15.2.3 Counting different butterfly topologies

15.2.4 Time analysis

15.2.5 Results

15.3 Conclusion

15.4 Availability

15.5 Competing interests

15.6 Author’s Contributions

15.7 Acknowledgements
Part I

Overview
Chapter 1

Introduction

In this dissertation I will present the work of my Ph.D. studies. All the work was performed within the field of bioinformatics, but spans widely within it. My interests lie mainly with the development of algorithms and their efficient implementation, while the biological questions and concrete data are of somewhat less interest to me. Thus I have worked on topics as diverse as virtual screening, association mapping, molecular transcription, Hidden Markov Models and tree distance.

The world in general and the field of Science in particular are currently spending large amounts of resources on research into biology and biological technology. One of the major motivations for this is human health. We want to design new cures and drugs for human diseases. In drug development, screening is the process of testing large libraries of molecules for activity for some target, for example a disease. Physically testing this large number of molecules is expensive, so virtual screening is becoming an important tool. Virtual screening is the use of computers to predict activity instead of determining it experimentally. One way of making these predictions is to simulate what happens at the atomic scale, but this requires a very large amount of computational resources, therefore it can still be quite expensive, monetary wise. A simpler approach is to assume that similar molecules have similar properties. That way one can predict the properties and activity of a new molecule from similar known molecules. For example, if molecules that show activity for a disease already are known, one could screen for new drugs by finding molecules similar to a known active. My work with virtual screening is described in detail in Chapter 2, and has resulted in the papers [78–80] which are also found in Chapters 8 and 9.

Everything happening inside the body of a human being is, directly or indirectly, affected by the genes of that individual. This includes any diseases and any drugs for those diseases. Thus an important aspect in increasing human health is the study of our genes and how they interact with diseases and drugs. Due to mutations, a gene may exist in several different versions, known as alleles. Different alleles may have different effects on the body and anything in it, and one way of studying these effects is an association study. Association studies uses statistics and large sample size, more than biological knowledge and clever experiment design, to achieve knowledge about the genome. Basically, a large number of individuals have their sets of alleles determined experimentally,
and these alleles are correlated with the physical traits of the individuals. If an allele correlates with a trait we can use the genomic location of the mutation to guide relevant research. That will increase our understanding of that particular trait, and may eventually lead to cures for related diseases. I have worked with association mapping, and the details can be found in Chapter 3. The work has resulted in the paper [99], which is included in Chapter 10.

The effect of studying individual genes may be limited if we do not fully understand how genes work and their role in general. In the classical model a gene is a segment of a chromosome which is transcribed into RNA; has certain sections, called introns, spliced out; and is then translated into an amino acid sequences called a protein. However, it has become evident in the recent years, that reality is not quite so simple. For example RNA that is not translated into protein is often found to play important roles by itself; RNA that is not even a proper part of a gene is found to be transcribed; and we even find that changes in traits can be inherited without changes to the DNA itself. I have worked with a group of molecular biologists, helping them interpret their data from experiments related to the transcription of genes. This is described in Chapter 4, and was published in [104, 105], which are included in Chapters 11 and 12.

If we consider DNA itself it consists of a long sequence of base-pairs. Each base-pair is one of four possibilities, which we denote by $A$, $C$, $G$ and $T$. The process of experimentally determining the concrete sequence of a segment of DNA is called sequencing. Deriving useful information from these, often very long, sequences is one task in bioinformatics. A tool that is very useful for the analysis of sequences is the Hidden Markov Model. In a Hidden Markov Model the sample sequence is assumed to be emitted by an equally long sequence of hidden states. By designing the hidden states to mirror physical states or phenomena, and working backward from the sample input sequence, one can make inferences about the properties of the processes that created the input sequence. For example some researchers make inferences about speciation time, by exploiting incomplete lineage sorting, having the hidden states represent different evolutionary histories, and using an alignment of genomes as input sample sequence [47]. In my studies I have designed new and fast algorithms for working with Hidden Markov Models. These were published in [97, 100] which are available in Chapters 13 and 14.

Evolution plays a very important role in biology and no data in the field should be interpreted without proper consideration of the evolutionary context. The evolutionary relationship between individuals can be modeled as a directed acyclic graph, and stepping back to consider the species-level, it may be adequately approximated by a tree – the so-called tree of life. Considering a given set of species, different sets of data or different methods of interpreting the data may lead to different trees. To study the effects of different data sets or different methods systematically, a measure of the difference or similarity between trees is useful. The right way of defining such a tree distance is not immediately obvious, but several measures have been suggested. The different approaches can generally be split into two groups. One group of approaches is based on counting the number of identical or different substructures among two trees,
while the other group counts the number of operations needed to transform one tree into the other. I have worked on an algorithm for computing the quartet distance between two trees, which is based on counting the number of different quartet substructures between the two trees. An overview of this can be found in Chapter 6, while the details were published in [90,98], and can be read in Chapter 15.

Chronologically, my interest in bioinformatics started with a job as a student programmer. My first paper, which was on association mapping was an extension of the work done in that job. Soon thereafter I worked with the analysis of data related to the transcription of RNA. My recent work, which takes up the majority of this dissertation is work on the computation of molecular similarity, work on algorithms for Hidden Markov Models, and work on the distance between trees. Concretely, the work in this thesis is based on four conference papers and six journal papers. Two of these papers were published both at a conference and in a journal, giving eight distinct papers.


The remainder of this dissertation is organized as follows. You are reading the first part which is an introduction to, and overview of, the subjects I have worked on during the course of my studies. Part two contains the papers I have published. Besides the introduction and a conclusion, part one is organized into one chapter for each subject I have studied. Although there are some exceptions, each of those chapters are roughly organized into: A section on relevant background knowledge, a section on recent competing work, a section outlining my research contributions, and finally a section containing speculations on possible future work in that area.
Chapter 2

Molecular Screening

Developing new medical drugs is expensive. Among the first steps is a screening process, in which several thousands of molecules are synthesized and tested for activity against a given target [132]. This requires a lot of resources and manpower, and is thus very expensive. Therefore it has become common to use computers for predicting the activity of very large libraries of molecules, to identify the most promising leads for further laboratory experiments. This is called virtual screening. Since computer simulations generally require fewer resources than physical experimentation this can lower the cost of medical and biological research significantly.

In this area I have worked on practically fast algorithms for the computation of similarity of molecules, resulting in the papers


2.1 Background

Proteins is a class of macromolecules that play some of the most important roles in nature. Proteins have functions as catalysts, in signaling and in structural roles. A protein consists of a chain of amino acid residues. This chain folds into a more or less rigid structure, and it is this structure that has a biological function. A protein that functions as a catalyst will have a certain place, called the binding site, where other molecules will dock as the protein performs its function. The binding site is usually an indentation or cave in the structure of
Protein Binding site → Docked ligand

Figure 2.1: A ligand docking to a protein. Another ligand may dock with the same protein, if it is sufficiently similar.

The protein. A ligand is a small molecule that docks with a biological molecule, such as a protein, to perform some function.

One way to combat a disease is to find a ligand that will dock with a protein important for that disease, and disrupt its normal function. In general, one will have a list of molecules that are available for manufacturing. The virtual screening mentioned above may be done by simulating the docking between the protein and the ligand on a computer, but that kind of simulation can require a lot of computing time. Instead, one may rely on the idea that similar structure leads to similar properties. Thus, one may predict the properties of a molecule by studying the properties of similar molecules; or if one already know one ligand that will dock with a given protein, for example from another medical drug or a ligand that binds to the molecule in nature, one may find drug candidates by looking for ligands that are similar to the known binder [140].

It is not immediately obvious how to measure the similarity between two molecules. However, some quite simple measures have proven to be surprisingly good when used for virtual screening [92, 131]. For example, one might compute a bitstring encoding representative information about the molecules and use the similarity between the bitstrings as a measure of the similarity between the molecules. Such a bitstring for a molecule is denoted a fingerprint. Of course, there are many ways to compute a fingerprint for a molecule. One general approach is to select a set of features, each of which a molecule may or may not have. Each feature will then correspond to one bit in the fingerprint, and that bit will be set or not, according to whether the given molecule has the feature. Fingerprints of this form will often be quite long, and with many bits

\[ S_T(A, B) = \frac{|A \land B|}{|A \lor B|} , \]

where \( A \land B \) and \( A \lor B \) are the logical and and logical or respectively of \( A \) and \( B \), and \( |A| \) is the number of bits set in \( A \).

Just as there are many ways to compute the distance between two fingerprints there are also many ways to compute the actual fingerprints [32]. One general approach is to select a set of features, each of which a molecule may or may not have. Each feature will then correspond to one bit in the fingerprint, and that bit will be set or not, according to whether the given molecule has the feature. Fingerprints of this form will often be quite long, and with many bits
2.2 Searching for similar molecules

As mentioned above one of the major motivations for molecular similarity is finding molecules for medical drugs. The problem can be formalized as: We are given a database $\mathcal{D}$ of fingerprints of synthesizable molecules, a query molecule $A$, and a minimal similarity $S_{\text{min}}$. The task is then to find all molecules $B \in \mathcal{D}$ where $S_T(A, B) \geq S_{\text{min}}$. Assume $\mathcal{D}$ contains $N$ fingerprints, all of length $n$.

### 2.2.1 Related Work

In [127] it is observed that since $|A \land B| \leq \min(|A|, |B|)$ and $|A \lor B| \geq \max(|A|, |B|)$, we can upper-bound the similarity between $A$ and $B$ by

$$S_T(A, B) = \frac{|A \land B|}{|A \lor B|} \leq \frac{\min(|A|, |B|)}{\max(|A|, |B|)} = S_{\text{count max}}(A, B).$$

We can use this to make queries faster than a linear search by sorting the fingerprints $B \in \mathcal{D}$ into bins depending on their counts of one-bits $|B|$. When a query is performed we can compute which bins has a $S_{\text{count max}}(A, B) \geq S_{\text{min}}$ and only examine the fingerprints in those bins.

In a later paper [14] they suggest a filter based on XOR signatures to improve this pruning even further. First split the fingerprints into $k$ equal-sized

| $A$ | 1 | 0 | 1 | 1 | 0 | 1 | $|A| = 4$ |
|-----|---|---|---|---|---|---|---------|
| $B$ | 1 | 1 | 0 | 1 | 0 | 0 | $|B| = 3$ |
| $A \land B$ | 1 | 0 | 0 | 1 | 0 | 0 | $|A \land B| = 2$ |
| $A \lor B$ | 1 | 1 | 1 | 1 | 0 | 1 | $|A \lor B| = 5$ |

$S_T(A, B) = \frac{2}{5}$

Figure 2.2: The notation used for bitstrings.

Another way to compare ligands was suggested in [130]. Their proposed measure is based on a comparison of the SMILES strings of the molecules. SMILES [137] strings are a standard way to encode the two dimensional structure of a molecule in a one dimensional text string. Any string of length $n \geq q$ will have exactly $n - q + 1$ substrings of length $q$. A substring, of length $q$, of a SMILES string is called a LINGO in [130]. Thus the SMILES string of a ligand may be viewed as a multiset of LINGOs, which they call the LINGO profile of the molecule. The similarity between two ligands they then measure as the Tanimoto coefficient between the LINGO profiles of the ligands, and this measure they name LINGOsim.

set to zero. To use space more efficiently they are often hashed or compressed into shorter fingerprints.
fragments such that
\[ A = A_1 \cdot A_2 \cdots A_k, \quad B = B_1 \cdot B_2 \cdots B_k. \]

Now compute XOR signatures as
\[ a = A_1 \oplus A_2 \oplus \cdots \oplus A_k, \quad b = B_1 \oplus B_2 \oplus \cdots \oplus B_k. \]

The observation is that since we can compute
\[ |A \land B| = \frac{|A| + |B| - |A \oplus B|}{2}, \quad |A \lor B| = \frac{|A| + |B| + |A \oplus B|}{2}, \]
and we can lower-bound the size of the XOR of the fingerprints by the size of the XOR of the signatures
\[ |A \oplus B| \geq |a \oplus b|, \]
we can bound the similarity as
\[ S_T(A, B) = \frac{|A| + |B| - |A \oplus B|}{|A| + |B| + |A \oplus B|} \leq \frac{|A| + |B| - |a \oplus b|}{|A| + |B| + |a \oplus b|} = \frac{S_{\text{max}}(A, B)}{S_{\text{min}}(A, B)}. \]

Since \( a \) and \( b \) are shorter than \( A \) and \( B \), \( a \oplus b \) can be computed faster than \( A \oplus B \).

This is used as a filter, where the fingerprints \( B \in \mathcal{D} \) are still stored in bins depending on \( |B| \), but the signature of each fingerprint is stored with it, and the final \( S_T(A, B) \) is only computed for fingerprints where \( S_{\text{max}}(A, B) \geq S_{\text{min}} \).

Finally \[122\] stores \( \mathcal{D} \) as a trie, although the author of the paper does not seem to be aware of that name. The key observation is that by walking down the trie one can bound the similarity between the query fingerprint \( A \) and any fingerprint \( B \) in a leaf below the current node in the trie. Consider a node at a level \( d \) in a trie. Let \( A_{\text{head}} \) be the first \( d \) bits of the query fingerprint \( A \) and \( A_{\text{tail}} \) be the remaining \( n - d \) bits. Similarly for an arbitrary database fingerprint \( B \) below the node. We may now observe that
\[ |A \land B| \leq |A_{\text{head}} \land B_{\text{head}}| + |A_{\text{tail}}|, \quad |A \lor B| \geq |A_{\text{head}} \lor B_{\text{head}}| + |A_{\text{tail}}|, \]
and hence
\[ S_T(A, B) = \frac{|A \land B|}{|A \lor B|} \leq \frac{|A_{\text{head}} \land B_{\text{head}}| + |A_{\text{tail}}|}{|A_{\text{head}} \lor B_{\text{head}}| + |A_{\text{tail}}|} = S_{\text{trie}}(A, B). \]

Thus, like above, we need only visit the children of the current node if \( S_{\text{trie}}(A, B) \geq S_{\text{min}} \). A trie easily takes up a lot of memory, so in the paper the trie is compressed by collapsing long runs of zero-bits into one node. This works well, because molecular fingerprints tend to be sparse.

### 2.2.2 Research Contributions

We have worked on data structures for similarity queries that are fast in practice. The first is, what we dub, the \( k \)D-grid. Again split all fingerprints into into \( k \) equal-sized fragments such that
\[ A = A_1 \cdot A_2 \cdots A_k, \quad B = B_1 \cdot B_2 \cdots B_k. \]
2.2. Searching for similar molecules

Now, place all database fingerprints $B \in D$ into bins in a $k$ dimensional grid, based on the bit counts of the fragments $|B_i|$. Like above we can compute bounds on the similarity between a query fingerprint $A$ and the database fingerprints in any of these bins:

$$S_T(A, B) = \frac{|A \land B|}{|A \lor B|} \leq \frac{\sum_{i=1}^{k} \min\{|A_i|, |B_i|\}}{\sum_{i=1}^{k} \max\{|A_i|, |B_i|\}} = S_{\text{grid}}(A, B).$$

In practice we implement the grid as a tree with $k$ levels and leaves of degree $\frac{n}{k}$, but with branches without leaf fingerprints pruned. When looking up a query, we walk down the tree and can compute bounds on all sub-branches. Assume we are visiting a node at level $l$ in the tree. The bound is then

$$S_T(A, B) \leq \frac{\sum_{i=1}^{l} \min\{|A_i|, |B_i|\} + \sum_{i=l+1}^{k} |A_i|}{\sum_{i=1}^{l} \max\{|A_i|, |B_i|\} + \sum_{i=l+1}^{k} |A_i|}.$$

Note that if we set $k = 1$ this corresponds to the approach of [127] and if we set $k = n$ this becomes the trie of [122].

Naively one would store the fingerprints in each bin in a simple list. We can do better, however. In the paper we present two alternative data structures for representing the bins. The first is the singlebit tree. The fingerprints for a given bin will be stored in the leaves of a tree, while the internal nodes each store the index of a bit. Fingerprints with the indexed bit clear will be stored in the left subtree of the given node, and fingerprints with the indexed bit set will be stored in the right subtree. Thus it is similar to a trie, except the bits

\[\begin{array}{cccccccccc}
B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 & B_9 & B_{10} \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
|B_1| = 2 & |B_2| = 3 & |B_3| = 1
\end{array}\]
in the bitstring can be examined in any arbitrary order, instead of left-to-right, as they are in a trie. Also, since we know what bucket the singlebit tree is sitting in we have information about the number of set bits for all fingerprints in the entire tree, which allows us to derive tighter bounds than those of [122]. Let $M_{ij}$ be the count of positions where $A$ has an $i$-bit and $B$ has a $j$-bit. For example $M_{10}$ is the number of positions where $A$ has a one and $B$ has a zero. Walking down a singlebit tree we will obtain partial knowledge of $M_{ij}$, as we compare the bits in the nodes of the tree with those in $A$. Let the number of positions we have knowledge about, and where $A$ has an $i$-bit and $B$ has a $j$-bit
2.2. Searching for similar molecules

Figure 2.6: Example of a multibit tree. The black squares denote the match bits, while the gray squares denote bits that are match bits further up the tree.

be $m_{ij}$ and the unknown difference $m_{ij}$ and $M_{ij}$ be $u_{ij}$, 

$$M_{ij} = m_{ij} + u_{ij}.$$ 

Now we can bound the Tanimoto coefficient by

$$S_T(A, B) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}} \leq \frac{\min\{|A| - m_{10}, |B| - m_{01}\}}{\max\{|A| + m_{01}, |B| + m_{10}\}},$$

and only visit subtrees where the leaves may be sufficiently similar to the query fingerprint.

Improving upon this we also suggested the multibit tree. The multibit tree is similar to the singlebit tree, but stores several bits in each internal node instead of only one. This means that we can no longer split the children of a node based on whether they have a one or a zero, thus the semantics needs to change somewhat. For each node in the tree store a list of bit positions, along with a boolean value. These bits we call the *match bits*. The match bits of a node are exactly those bits for which all the children of the node have the same value and that is not a match bit further up the tree. Walking down a multibit tree we again gain partial knowledge about the leaves of the tree and exactly the same bound as that of the singlebit tree may be used.

How best to build the singlebit and multibit trees is not obvious. The algorithm we used in our implementation is to split the dataset recursively into smaller and smaller subsets. For each set of fingerprints we choose the bit that splits the tree into two subsets that are maximally close to having the same size. The set is then split into two subsets based on whether that bit is set or not for each fingerprint. The reasoning is to attempt to obtain a tree that is as well-balanced as possible. Theoretically it is not clear that this is the right way to build the trees, but in practice it seems to perform well.

Our experimental implementation can be downloaded from http://www.birc.au.dk/~tgk/TanimotoQuery and the paper on this work can be found in chapter 8.
Chapter 2. Molecular Screening

2.2.3 Future Work

This project may be continued in several directions. Most obviously the way the trees are built might be improved. As mentioned above the motivation for the current algorithm is to build well-balanced trees, but what we actually should be doing is to minimize the number of nodes visited in a query, and it is not obvious that the current algorithm does that in any way. Another area for improvement is memory usage. Currently the data structures use a lot of memory. To store very large molecule databases it might be relevant to create an I/O efficient implementation that stores the data structures on disk in a way that can be processed efficiently without reading the entire structure into memory. Also, in our current implementation the kD-grid is actually a tree. It might be possible to create one data structure that generalizes both the multibit tree and the kD-grid, like the kD-grid generalizes the approaches of [127] and [122].

2.3 Fast computation of the LINGOsim

The LINGOsim [130] similarity measure has a certain attractiveness because it only relies on the SMILES description of the molecule. Above the LINGOsim is described as working with length \( q \) substrings of SMILES strings, but technically

Figure 2.7: Running time versus database size for our algorithms, the previous best algorithm and a naive linear search. For each kD-grid the \( k \) which gave the best results were chosen.
the SMILES strings need to be lightly modified. In the remainder of this text we are simply going to use the term SMILES strings to include these slightly modified ones. Nevertheless, in the case that the database of molecules already stores the structure of the molecules as a SMILES string very little further computation or storage is needed. Furthermore LINGOsim has proven to be competitive with more computationally expensive methods for predicting ligand properties, despite its simplicity [59]. Although the LINGOsim already is faster to compute than most other methods, growing databases of molecules is a motivation for faster algorithms.

### 2.3.1 Related Work

The paper of [59] primarily focus on evaluating the merits of the LINGOsim measure, but also suggest a fast algorithm, using a finite state machine. Given a query SMILES string \( A \) and a database \( D \), they suggest building a finite state machine from \( A \) to be able to quickly compare it against any other SMILES string. Remember that the LINGOsim similarity between two molecules is the Tanimoto coefficient of the two multisets of all substrings, of some length \( q \), of the SMILES strings of the two molecules. The paper is scarce on details, but the algorithm starts by building a trie from all length \( q \) substrings of \( A \). This trie is converted into a finite state machine, where states correspond to length \( q \) strings, and substrings in \( A \) are accept states. Running this state machine against another SMILES string, the size of the intersection can quickly be found, and from this the size of the union is also easy to find since

\[
|A| + |B| = |A \cap B| + |A \cup B|.
\]

With the sizes of the intersection and union in hand, the LINGOsim is easy to obtain.

A parallel algorithm is suggested in [63]. Their first observation is that since a character in a computer normally uses eight bits and both [130] and [59] show that the optimal length of LINGOs is \( q = 4 \), a LINGO can be stored in a 32-bit computer word. For each molecule they explicitly store a sorted list of all LINGOs in the SMILES string of the molecule, along with a count of how many times each LINGO occur in the molecule. This allows the intersection size of LINGOs between two molecules to be computed by iterating over the two LINGO lists simultaneously, similar to the merge of a merge-sort. As above the intersection size is enough to compute the LINGOsim. Parallelization is achieved by processing several molecules at the same time and the paper presents implementations for both CPUs and GPUs. They name their algorithm SIML for Single-Instruction Multiple-LINGO.

### 2.3.2 Research Contributions

We suggest using an inverted index to compute the LINGO intersection size between a query SMILES string \( A \) and the entire database \( D \) quickly. First a preprocessing step is necessary, where each LINGO in the database \( D \) is given an integer number, such that the first occurrence of a given LINGO in a
Chapter 2. Molecular Screening

(a) Example simplified SMILES string

\[ S = c0ccccc0L \]

(b) LINGOs

<table>
<thead>
<tr>
<th>LINGO</th>
<th>freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0cc</td>
<td>1</td>
</tr>
<tr>
<td>0ccc</td>
<td>1</td>
</tr>
<tr>
<td>cccc</td>
<td>2</td>
</tr>
<tr>
<td>ccc0</td>
<td>1</td>
</tr>
<tr>
<td>cc0L</td>
<td>1</td>
</tr>
</tbody>
</table>

(c) LINGO ids

<table>
<thead>
<tr>
<th>LINGO id</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0cc</td>
</tr>
<tr>
<td>0ccc</td>
</tr>
<tr>
<td>cccc</td>
</tr>
<tr>
<td>ccc0</td>
</tr>
<tr>
<td>cc0L</td>
</tr>
</tbody>
</table>

(d) Inverted indices data structure

```
  30
 /|
/ |
  S
```

```
  42
 /|
/ |
  S
```

```
  54
 /|
/ |
  S
```

Figure 2.8: From SMILES strings to inverted index. (a) SMILES string simplified for LINGOsim. (b) LINGOs of example SMILES string. (c) The LINGOs are given ids, with multiple occurrences given unique ids. (d) A reference to the SMILES string \( S \) is stored for all the ids of the LINGOs in \( S \).

SMILES string is given a unique id, the second occurrence another one, and so on. This step reduces the problem from multisets of LINGOs to ordinary sets of ids. Next we store the database in an inverted index [75], that is, instead of storing a list of LINGOs or ids for each database SMILES string, we store a list of SMILES strings for each LINGO id. Now we can compute the LINGO intersection size for the entire database the following way: Create a counter for each database SMILES string. For each LINGO in the query \( A \) increase the counter of all database SMILES strings found on the list of that LINGO. When all LINGOs has been processed the counters contain the LINGO intersection sizes, from which the LINGOsim can be computed. This is fast because we only need to visit relevant LINGOs in the database, as opposed to the above methods that always query the entire database. Like above we can parallelize this by processing several molecules at the same time.

For benchmarking we use a method similar to that of [63], computing the LINGOsim similarity of all pairs of fingerprints in the database. We compare our implementation against SIML [63] and the commercial OpenEye implementation.

The implementation used in our experiments can be downloaded from http://www.birc.au.dk/~tgk/ii. The paper describing this work in detail can be found in chapter 9.
2.3. Fast computation of the LINGOsim

Fold speedup over OpenEye.

Number of fingerprints.

1 CPU core

4 CPU cores

\(2^{14}\)

\(2^{15}\)

\(2^{14}\)

\(2^{15}\)

OpenEye

SIML [63]

Our method

Figure 2.9: Comparison of our implementation against that of OpenEye and SIML, for one and four CPU cores.

2.3.3 Future Work

I do not find the preprocessing into unique ids very satisfying. I suggest this can be avoided by creating a hybrid algorithm between our inverted indexes and the finite state machine of [59]. Instead of building a finite state machine over the query string \(A\), build one over the database \(D\). A cursory glance at a small SMILES database counts 28 different characters, some of which cannot appear in a LINGO, since those are build over slightly modified SMILES strings. For \(q = 4\) this gives only 614656 different possible LINGOs, making it feasible to build a state machine over the entire LINGO space. Annotate each of these states with a list of the database SMILES strings containing the corresponding LINGO and a count of how many times. Now a query can be performed by running the query SMILES string \(A\) through the state machine and for each state increasing a counter for all relevant database SMILES strings.
Chapter 3

Association Mapping

Many diseases have some genetic component. Even if the disease is not hereditary in itself, there may exist some hereditary traits that makes an individual more or less likely to contract the disease. To determine the heritability of a given disease one may study families affected by the disease. Once a disease has been determined to have a hereditary component it is highly relevant to find out exactly which genetic variations that result in the change in disease risk. This information will point researchers to parts of the genome that are related to the disease, and may thus further our understanding of the disease and how to cure it. Association mapping uses large amounts of data and correlations between genomic features to predict associations between expressed traits and regions of the genome. My contributions to this area are published as


3.1 Background

In biology a polymorphism is a genomic feature of which there exists several different forms in a given population. The different forms are called different alleles. In general we are interested in humans, who are diploid, which means we have two of each chromosome. For any genomic feature we say that an individual is homozygotic if it has two copies of the same allele, and heterozygotic if it has two different alleles. Two genomic features are in linkage disequilibrium if the probability of seeing them together is different from what one would expect, from their frequencies in the population, if they were paired completely at random. Humans inherit one of their two copies from each parent. These copies are not necessarily exact copies of one of the parents chromosomes, because a recombination may have happened. A recombination is an event that mixes the two copies of the same chromosome and results in two hybrids instead.

A Single Nucleotide Polymorphism (SNP) is a point in the genome, a locus, where there is a single nucleotide that is polymorphic. What makes SNPs interesting is that thousands of them can be genotyped cheaply and efficiently at the same time, using DNA microarrays.
Chapter 3. Association Mapping

Double stranded break

Figure 3.1: Meiotic recombination is a process during cell division where genetic material is exchanged between two homologue chromosomes. If a double-stranded break occurs the cell has mechanisms that will attempt to repair this, and the result may be a recombination. If the dots represent mutations we see how the two mutations, $b$ and $c$, that were previously sitting on two different chromosome now share one.

Figure 3.2: Assume the dots mark mutations in the above phylogeny. If we know that an individual has the $a$ mutation we gain information about the state of the other mutations. Concretely we will know that the probability of the individual having the $b$ mutation is greater than zero and that the individual will not have the $c$ mutation. In the real world recombination will slowly erode this information.

A Whole Genome Association Study is a study in which a set of SNPs covering the entire genome is selected and genotyped in a large number of individuals. This data is then studied to find out which parts of the genome affect which traits. Usually a particular trait, for example a disease, is selected beforehand and the test individuals are selected so that half of them will carry the trait and the other half will not. In general it is not expected that the genotyped SNPs themselves are affecting the studied trait directly, but they are likely to be in linkage disequilibrium with genomic features that are, and hence it may be expected that some SNPs will still be statistically correlated with a trait. For all of the above see [37, 38, 84].

When the data is collected one needs to examine the data for correlations. In principle one could simply consider the SNPs one-by-one, and test if there is a statistically significant difference between the size of the groups of affected and unaffected individuals, for each state of the SNP. This approach is problematic because the effects we see are usually quite small. Even if there is a difference at a given SNP the difference in size between groups is small, and at the same time there are hundreds of thousands of SNPs. This means one will get a lot of false positives unless one corrects for multiple testing by placing strict requirements on the tests. However, that would significantly reduce the power of the test.
## 3.2. Related Work

A well-known toolset for association analysis is PLINK [106]. PLINK is able to perform many of the most simple and common operations and statistics for association studies. Although the implemented methods tend to be basic the toolset is designed to be fast and to handle large data sets efficiently. Data for PLINK is stored in two files: A PED file containing the genotypes and information about each individual, and a MAP file containing information about

<table>
<thead>
<tr>
<th>SNP 1</th>
<th>SNP 2</th>
<th>SNP 3</th>
<th>SNP 4</th>
<th>SNP 5</th>
<th>SNP 6</th>
<th>SNP 7</th>
<th>SNP 8</th>
<th>SNP 9</th>
<th>SNP 10</th>
<th>...</th>
<th>SNP 500,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>AA</td>
<td>CG</td>
<td>GT</td>
<td>AA</td>
<td>TT</td>
<td>GG</td>
<td>CC</td>
<td>TT</td>
<td>CG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 10,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 10,001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 10,002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 10,003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 10,004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sample 20,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3: Example of what association mapping data might look like. Notice for example, SNP 6. It seems that the G allele is more common among the affected than the unaffected. A proper test is needed to determine whether the difference is significant.

Therefore more complicated tests, that examine several SNPs at the same time, have been proposed, for example [24, 25, 67, 88, 102, 109, 128, 129, 145]. Usually whole genome association data sets consist of data from hundreds of thousands of SNPs genotyped in thousands of individuals. Besides the actual SNP data one normally has metadata describing the SNPs and individuals. For example one might store an id, age, gender, geographical region and the status of various biological traits for each individual; and also an id, along with chromosome and chromosomal position for each SNP. Often this data is stored in simple text files, sometimes with the different kinds of data scattered over several different files in different formats, for example [106]. Text files can take up excessive amounts of space, and be slow to parse, while large amounts of data spread over several heterogeneous file formats can be a significant data management challenge.

### 3.2.1 Unaffected (Healthy)

<table>
<thead>
<tr>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>...</th>
<th>Sample 10,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG AA</td>
<td>GG TT</td>
<td>CG AT</td>
<td>CC AT</td>
<td>...</td>
<td>CC TT</td>
</tr>
<tr>
<td>CG CG</td>
<td>CC CC</td>
<td>CG GT</td>
<td>CC AT</td>
<td>...</td>
<td>GG CC</td>
</tr>
<tr>
<td>GT AA</td>
<td>AG CC</td>
<td>GT AA</td>
<td>CC AT</td>
<td>...</td>
<td>AG CC</td>
</tr>
<tr>
<td>AA TT</td>
<td>AG CT</td>
<td>TT CT</td>
<td>AG CT</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>GG CC</td>
<td>TT AG</td>
<td>CC AG</td>
<td>AG CC</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>TT CC</td>
<td>GG CC</td>
<td>TT AG</td>
<td>CC AG</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>CC TG</td>
<td>GG CC</td>
<td>TT AG</td>
<td>CC AG</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>CC TG</td>
<td>GG CC</td>
<td>TT AG</td>
<td>CC AG</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>CG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.2 Affected (Sick)

<table>
<thead>
<tr>
<th>Sample 1</th>
<th>Sample 2</th>
<th>Sample 3</th>
<th>Sample 4</th>
<th>...</th>
<th>Sample 10,003</th>
</tr>
</thead>
<tbody>
<tr>
<td>GG AA</td>
<td>CG AT</td>
<td>CG AT</td>
<td>GG AA</td>
<td>...</td>
<td>GG AA</td>
</tr>
<tr>
<td>CG GG</td>
<td>GG CC</td>
<td>GG CC</td>
<td>GG CC</td>
<td>...</td>
<td>GG TT</td>
</tr>
<tr>
<td>GG CC</td>
<td>TT GG</td>
<td>CC AG</td>
<td>AG CC</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>TT CC</td>
<td>GG CC</td>
<td>TT AG</td>
<td>CC AG</td>
<td>...</td>
<td>AG CT</td>
</tr>
<tr>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Therefore, more complicated tests, that examine several SNPs at the same time, have been proposed, for example [24, 25, 67, 88, 102, 109, 128, 129, 145]. Usually whole genome association data sets consist of data from hundreds of thousands of SNPs genotyped in thousands of individuals. Besides the actual SNP data one normally has metadata describing the SNPs and individuals. For example, one might store an id, age, gender, geographical region and the status of various biological traits for each individual; and also an id, along with chromosome and chromosomal position for each SNP. Often this data is stored in simple text files, sometimes with the different kinds of data scattered over several different files in different formats, for example [106]. Text files can take up excessive amounts of space, and be slow to parse, while large amounts of data spread over several heterogeneous file formats can be a significant data management challenge.
Chapter 3. Association Mapping

the SNPs. Recognizing that parsing the genotype data can consume a large part of the time of a simple analysis, PLINK also supports an efficient binary data format. A binary PED file is called a BED file. To conserve size the data in a BED file is packed so that data for both alleles for one SNP for one individual only use two bits. Two bits give four different combinations, two of these are used to represent the two homozygous states, one represents a heterozygous state, and the final combination represents missing data. This implies some assumptions about the data. First, all the individuals must be diploid, second there can only be two alleles of each SNP, and finally the data must be unphased, so that the two heterozygous states are indistinguishable. When a BED file is created two associated text files are also created: A FAM file, which contain the same information as the PED file, except the genotypes have been removed and a BIM file, which is a MAP file extended with allele information that is lost from the PED file. Thus the BED file format does conserve space and make data access faster, but actually increases the data management burden, instead of lessening it.

3.3 Research Contributions

In connection with the development of other software for association mapping [17, 88], we have developed a file format and related software library for storing and handling SNP data. Both the file format and software library is called SNPFile. The majority of the data in a SNPFile is the genotype data. This is simply stored in a matrix of bytes, with rows for individuals and columns for SNPs. Having an entire byte for each genotype might seem wasteful, but it simplifies access and allows us to store SNPs with multiple alleles or different kinds of error codes for missing data in the cells. Furthermore SNPFile supports phased data, where the two haplotypes in a heterozygotic genotype have been assigned to chromosomes. In that case SNPFile will use two rows, one for each chromosome, for each individual. The genotype data in SNPFile is stored in row-major, or SNP-major order, so that genotypes for one SNP is stored together on the disk. This is because association mapping methods tend to work on a per-SNP basis, and in that case storing data for each SNP close together improves performance, due to caching and prefetching. A major feature of SNPFile is that besides the SNP data a SNPFile also stores arbitrary metadata. The SNPFile library provides simple byte buffers in the files, where any extra information may be stored. On top of these byte buffers there is a serialization framework that supports all the primitive types, the most common containers from the C++ STL, and arbitrary user extensions. Information such as affected status of individuals and SNP genomic positions are stored in this metadata. Thus, when compared to the binary format of PLINK, both formats speed up SNP data access by skipping the parsing step, PLINK uses significantly less spaces by packing its bits, and SNPFile is more flexible in which kinds of SNP data can be stored, and removes a lot of the data management burden by keeping all relevant data in one file.

The SNPFile library can be found on-line at http://www.birc.au.dk/
3.4 Future Work

SNPFile was released in 2008. Since then sequencing of complete genomes has become very much cheaper and is, to some degree, replacing the SNP and microarray technology for larger studies. This makes the SNPFile format somewhat less relevant, however one could create a similar format for sequencing data. Since a SNPFile is simply a matrix of bytes with arbitrary metadata, one could actually store an alignment of sequences in a SNPFile without any modifications to the software library. The largest hurdle to storing sequence data in a SNPFile is probably file size. The human genome is roughly $3 \cdot 10^9$ base pairs long, and would thus take up 3GB of disk space. If an alignment of several of these were to be stored in one single file that would make for a very big file, and it is not obvious that it would be desirable to store this much data in one file. Moreover SNPFile is not designed to handle files of that size, in fact all addressing is done in 32 bits, which means that SNPFile is limited to roughly 4GB of data. Other than file size requirements sequence data is likely to have domain-specific requirements currently not supported by SNPFile, just like SNPFile currently requires one to specify whether the contained data is phased, which does not make sense in a sequence context.
Chapter 4

Analyzing Molecular Biological Data

In molecular biology the classical central dogma is that the way the genome influences the surrounding world is by being *transcribed* into RNA, the RNA is then *translated* into protein, which is what fulfills useful roles in biology. However, it has become evident that reality is not that simple, and in particular that RNA itself can have many functional roles. In cooperation with researchers from molecular biology I have published:


4.1 Background

Proteins are synthesized based on a DNA gene. DNA is double stranded and a given gene resides on only one of these strands. The strand containing the gene is called the *sense* strand, while the opposite strand is called the *antisense* strand. It varies which strand is sense, depending on the gene. In front of the gene on the sense strand sits an area called the *promoter*, which serves to attract the molecules needed for transcription and regulation thereof. A gene is transcribed into messenger RNA, by a protein called *RNA polymerase II*. In eukaryotes, such as humans, some segments of the gene are not used in the final protein product. The segments of a gene that are used are called *exons*, while the remaining segments are called *introns*. Introns are removed from the messenger RNA in a process called splicing. Finally the RNA is translated into a protein, which is a chain of amino acid residues, by a complex called the *ribosome* [37, 95]. See Figure 4.1. We say that a gene is *(expressed)*, when the information in it is used by the cell.

RNA plays many roles in the cell. For example some get translated into protein as stated above; some are functional in their own right; and some are
involved in gene regulation. The cell have several pathways for degrading RNA, and one of the factors in this, which is relevant for this dissertation, is the exosome [93].

RNA interference is a mechanism that down regulates or completely stops the expression of a gene [54]. It is usually based on complementary RNA binding to messenger RNA and guiding other molecules to cleave the messenger RNA, thus preventing it from successfully being translated. RNA interference occurs naturally in the cell, both for the normal gene regulation and in defense against viruses. However, it has also turned out to be an important tool in biological research, as it gives researchers a simple and robust way to turn off the expression of certain genes in experiments.

### 4.2 Related Work

In a paper [142] from 2005, a group of researchers created a yeast strain with an inactivated Rrp6 gene. Rrp6 is normally translated into a protein that is a part of the exosome. The authors of the paper then analyzed the transcribed RNA, the transcriptome, using a DNA microarray. They found a new class of
polymerase II transcripts in areas between genes, which they dubbed CUTs, for Cryptic Unstable Transcripts.

Our work was published in the December 19 2008 issue of Science. That issue has a short perspective named “Gene Expression – Where to start?” [27], and besides our work the issue contained three more papers on similar results [40, 64, 119]. Shortly thereafter, on February 19 2009, Nature published an issue with three related papers [52, 96, 143].

Four of the above papers work with various human cell lines [40, 52, 64, 119]. The paper [119] examine embryonic stem cells; in [64] a new method is developed, that can measure the levels of transcribed RNA and is able to differentiate which of the two DNA strands it is transcribed from; and [40] presents a new method that measures position, amount and orientation of RNA polymerase engaged in transcription. All three of these report very similar data showing that transition does start at previously identified transcription start sites, but that it unexpectedly seems to start in both directions, with the anti-sense transcription relatively quickly stalling. [52] used next-generation sequencing to catalog all small RNA, defined as all RNA of length less than 200 nucleotides, in HeLa and HepG2 cell lines. They found small RNAs both in well-known genes and between them, along with the above-mentioned divergent transcription.

The papers [143] and [96] both work with yeast. In [143] tiling arrays were used to obtain transcriptomes of wild-type and Rrp6 mutant yeast grown under various conditions, while [96] used sequencing to build a complete map of yeast CUTs. Like above, both find that the majority of transcripts originate in gene promoter regions and that many of them are antisense. In general it seems that promoters are bidirectional, and that there may be some other mechanism that determines which transcripts are elongated and translated into proper proteins.

4.3 Research Contributions

To discover more about what is actually transcribed we disabled the exosome of HeLa cells, stabilizing many RNA fragments that would otherwise have been degraded. To measure which transcripts are stabilized by exosome depletion, we used a DNA microarray covering the 1% of the human genome of the ENCODE pilot project [19]. The exosome was depleted by RNA interference knocking out some of the genes coding for the proteins that forms the complex. More concretely experiments were done knocking out two of the catalytic subunits, hRrp40 and hRrp6, and a subunit that forms part of the scaffold of the complex, hRrp44. Knockout of either hRrp40 or hRrp6 alone had little effect, but knockout of hRrp44 or both hRrp40 and hRrp6 at the same time showed a major stabilization of RNA. We interpret this as hRrp40 and hRrp6 being redundant subunits, with either one being enough for normal function, and hRrp44 being an essential part of the structure of the protein.

My primary involvement was the analysis of data from the DNA microarray. Besides the exosome knockout strain we had a control strain treated with enhanced green fluorescent protein. When examining the data we would
Chapter 4. Analyzing Molecular Biological Data

Figure 4.2: When data is aligned on transcription start or termination, data will be biased around the site of interest, because genes always start and end with an exon. The arrows show transcription start sites, while the boxes denote exons.

primarily consider the ratio of RNA expression between the knockout strain and the control. At first we saw a peak in the ratio somewhat upstream of active transcription start sites, and just before the transcription termination sites. We named these PROMPTs and TERMPTs respectively. However, when aligning data at transcription start- and termination sites the data will be biased because genes always start and end with an exon, as shown in Figure 4.2. By filtering out non-exonic data in the regions the TERMPTs disappeared, still leaving the PROMPTs, as seen in Figure 4.3. The PROMPTs peak about 1500nt before the gene, which makes it a separate phenomenon from that described in the Section 4.2 above, since their data peak around 200nt before the transcription start site. To verify our results my coauthors used real-time polymerase chain reaction (qrt-PCR) to measure RNA in the PROMPT region, in a separate experiment.

In an attempt at understanding the reason for, and mechanism behind, the transcription of these, apparently non-functional, DNA we compared our data against publicly available data sets from experiments measuring related properties of the genome. Such experiments could for example measure where polymerase II binds; where transcription factors, which take part in regulating transcription, binds; or histone modifications, which are also known to correlate with transcription. My coauthors selected 64 genes with positions on the
4.3. Research Contributions

Figure 4.3: Exon bias explains the TERMPT, but not the PROMPT. On the left an average of expression around the transcription start site, thus the PROMPT is plotted. On the right an area around the transcription termination site, thus the TERMPT is plotted. The full line show the ratio of stable RNA between our experimental cell line and the control. The dotted line show the ratio of genes having an exon at that position. The vertical line show the transcription start or termination site.

As for the function of PROMPTs we have suggested a few hypotheses. First of all PROMPTs might simply be a by-product of transcription. Second it has been suggested that they might serve as a reservoir of polymerase II, so that they are near the gene when transcription is needed. We also found a couple of cases of non-coding RNA in PROMPT regions, which suggests PROMPTs may be part of the regulatory machinery, but these seem to be exceptional cases. Finally we have also found a correlation between PROMPT activity and CpG pairs. A CpG pair is a cytosine (C) immediately followed by a guanine (G) in the DNA sequence, and these are known to have a high chance of becoming methylated and mutating. Since areas that still have many of these pairs tend to have a more active PROMPTs than those with few CpG pairs, transcription of the PROMPT might be involved in protecting them.

The above work has been published as the paper in Chapter 11, while a review of that and related work has been published as the paper in Chapter 12.

Figure 4.4: The PROMPT compared to Polymerase II (Pol II), TAF1 and E2F1 is shown. TAF1 and E2F1 are transcription factors, which means they are involved in the attraction of or binding of polymerase to the DNA. It seems we can actually measure polymerase II transcribing the PROMPT, but that transcription factors do not promote this. Finally we plotted the activity in the gene against activity in the PROMPT, as seen in Figure 4.5, and found that more active genes also have more active PROMPTs.

genome, so that no other genes were nearby, which we termed our conservative sample. This was done because many genes cluster together, sometimes even overlapping, which makes it impossible to determine which gene a transcript is related to with the used technology. Plotting data from our conservative sample only may thus give a graph with significantly less noise than a graph over data from all genes. In Figure 4.4 the PROMPT compared to Polymerase II (Pol II), TAF1 and E2F1 is shown. TAF1 and E2F1 are transcription factors, which means they are involved in the attraction of or binding of polymerase to the DNA. It seems we can actually measure polymerase II transcribing the PROMPT, but that transcription factors do not promote this. Finally we plotted the activity in the gene against activity in the PROMPT, as seen in Figure 4.5, and found that more active genes also have more active PROMPTs.

As for the function of PROMPTs we have suggested a few hypotheses. First of all PROMPTs might simply be a by-product of transcription. Second it has been suggested that they might serve as a reservoir of polymerase II, so that they are near the gene when transcription is needed. We also found a couple of cases of non-coding RNA in PROMPT regions, which suggests PROMPTs may be part of the regulatory machinery, but these seem to be exceptional cases. Finally we have also found a correlation between PROMPT activity and CpG pairs. A CpG pair is a cytosine (C) immediately followed by a guanine (G) in the DNA sequence, and these are known to have a high chance of becoming methylated and mutating. Since areas that still have many of these pairs tend to have a more active PROMPTs than those with few CpG pairs, transcription of the PROMPT might be involved in protecting them.

The above work has been published as the paper in Chapter 11, while a review of that and related work has been published as the paper in Chapter 12.
Figure 4.4: The PROMPT compared to data from other experiments related to transcription activity, for 64 selected conservative genes only.

Figure 4.5: The activity in the prompt region correlates with the activity in the associated gene. On the left our data is plotted log-transformed. The right graph show the polymerase II mono activity as measured by Michael Snyder (unpublished, data set GEO id GSE6391), thus the amount of RNA being transcribed. The $y$-axis show the measured activity in the 1.5kb before the gene, while the $x$-axis show the activity in the first 1.5kb inside the gene. Values are not shown on the axis, because they are experimental values without well-defined units.
Chapter 5

Hidden Markov Models

A Hidden Markov Model is a simple probabilistic model that defines a distribution over strings. Strings are modeled as being emitted by a sequence of hidden states. Because Hidden Markov Models are relatively easy to understand and computationally efficient algorithms for them are well known, they are widely used, especially in Bioinformatics [9, 45–47, 49, 55, 73, 74, 120, 133, 144] and Speech Recognition [33, 82, 138]. For example [47] uses a Hidden Markov Model to make inferences about incomplete lineage sorting, by having different phylogenies as hidden states and the genomes of extant species as the observed sequences. Similarly [144] has ancestral species as hidden states and genomic data from an extant species as observed sequence to estimate crossover and gene conversion rates.

Although there exists simple and fast algorithms for the most common problems related to Hidden Markov Models it is relevant to find even faster algorithms because they are so widely used. I have designed and implemented algorithms for the parallelization of Hidden Markov Models with a small state space and an approximate algorithm for Hidden Markov Models with a very large state space.


5.1 Background

A Hidden Markov Model has a sequence of hidden states that evolves through time. These hidden states are sampled randomly, with a Markov property: A state depends only on the preceding state. At each time step an observable symbol is emitted, sampled from a distribution determined by the hidden state
at that step. When a Hidden Markov Model is used in practice the hidden states are normally unknown, and we want to infer something about them from the observable symbols. According to Rabiner [107] there are three main problems associated with a Hidden Markov Model. The first problem is to compute the probability of the observed sequence given the model, the second is to infer the hidden states given the model and the observed sequence, and the final one is to infer model parameters from observed sequences.

We will use a notation similar to that of [107]. Let \( S = \{ S_1, S_2, ..., S_N \} \) be a set of \( N \) hidden states. If we run the Hidden Markov Model for \( T \) time steps we will set \( Q = q_1 q_2 ... q_T \) to be the sequence of actual hidden states, with \( q_t \in S \). Similarly let \( V = \{ V_1, V_2, ..., V_M \} \) be an alphabet of observable symbols, and \( O = O_1 O_2 ... O_T, O_t \in V \), the sequence of symbols emitted by the hidden states \( Q \). Let \( \pi = (\pi_1, \pi_2, ..., \pi_N) \) be the distribution of the initial hidden state, \( \pi_i = \mathbb{P} (q_1 = i) ; A = \{ a_{ij} \} \) be the transition matrix, \( a_{ij} = \mathbb{P} (q_t = j \mid q_{t-1} = i) \); and finally \( B = \{ b_j(k) \} \) be the emission matrix, \( b_j(k) = \mathbb{P} (O_t = k \mid q_t = j) \).

All the parameters of the Hidden Markov Model together we will denote by \( \lambda = (A, B, \pi) \). With the above we can compute the probability of a state- and observation sequence, \( Q \) and \( O \) respectively, given a model \( \lambda \) as

\[
\mathbb{P} (O, Q \mid \lambda) = \mathbb{P} (O \mid Q, \lambda) \mathbb{P} (Q \mid \lambda)
\]

\[
= \left( \sum_{t=1}^{T} \mathbb{P} (O_t \mid q_t, \lambda) \right) \left( \mathbb{P} (q_1 \mid \lambda) \sum_{t=2}^{T} \mathbb{P} (q_t \mid q_{t-1}, \lambda) \right)
\]

\[
= \left( \sum_{t=1}^{T} b_{q_t}(O_t) \right) \left( \pi_{q_1} \sum_{t=2}^{T} a_{q_{t-1}q_t} \right). 
\]

### 5.1.1 The Forward Algorithm

The common solution to the first problem mentioned above, is the forward algorithm. We want to compute the probability of the observed sequence \( O \), given the model \( \lambda \). In principle this is done by summing the joint probability over all possible hidden state sequences \( Q \),

\[
\mathbb{P} (O \mid \lambda) = \sum_{Q \in S^T} \mathbb{P} (O, Q \mid \lambda). 
\]

Unfortunately it is infeasible to do this computation directly, since there exists \( \mathcal{O}(N^T) \) possible sequences \( Q \). However, we can compute it reasonably efficiently using dynamic programming. Define \( \alpha_t(i) = \mathbb{P} (O_t O_2 ... O_t, q_t = i \mid \lambda) \). These \( \alpha_t(i) \) can be computed recursively,

\[
\alpha_t(i) = \begin{cases} 
    b_i(O_1) \pi_i & \text{if } t = 1 \\
    b_i(O_t) \sum_{j \in S} a_{ji} \alpha_{t-1}(j) & \text{if } t > 1
\end{cases}
\]

and if the values are cached so that each \( \alpha_t(i) \) is only computed once this can be done in \( \mathcal{O}(N^2T) \) time and \( \mathcal{O}(NT) \) space. The total probability can be computed from the sum over \( \alpha_T(i) \),

\[
\mathbb{P} (O \mid \lambda) = \sum_{i \in S} \mathbb{P} (O, q_T = i \mid \lambda) = \sum_{i \in S} \alpha_T(i). 
\]
5.1.2 The Viterbi Algorithm

The second classical problem is to find a state sequence $Q'$ that best explains the observations $O$, and the classical solution to this is the Viterbi algorithm. Formally we want to compute a $Q'$ that maximizes $P(Q' \mid O, \lambda) \propto P(O, Q' \mid \lambda)$. Similarly to the forward algorithm there are too many possible $Q'$ for it to be feasible to enumerate all of them, but the problem can again be solved using dynamic programming. Define $\delta_t(i)$ to be the likelihood of the maximally likely state sequence from time 1, that ends in state $i$ at time $t$,

$$\delta_t(i) = \max_{q'_1, q'_2, \ldots, q'_{t-1} \in S^{t-1}} \{ P(O_1O_2\ldots O_t, q'_1, q'_2, \ldots, q'_t = i \mid \lambda) \} .$$

The $\delta_t(i)$ can be computed recursively

$$\delta_t(i) = \begin{cases} b_i(O_1) \pi_i & \text{if } t = 1 \\ b_i(O_t) \max_{j \in S} \{ a_{ji} \delta_{t-1}(j) \} & \text{if } t > 1 \end{cases} ,$$

and by caching the results, these values can be computed fast, just like the forward algorithm. The above only computes the value of $P(Q' \mid O, \lambda)$, not the actual state sequence $Q'$. To compute $Q'$, one can do back-tracking through the values computed above

$$q'_t = \begin{cases} \arg\max_{i \in S} \{ a_{i q'_{t+1}} \delta_t(i) \} & \text{if } t < T \\ \arg\max_{i \in S} \{ \delta_T(i) \} & \text{if } t = T \end{cases} .$$

Computing the $\delta_t(i)$ takes time $O(N^2T)$, just like for the forward algorithm, and the backtracking takes time $O(NT)$. The dynamic programming table uses $O(NT)$ space.

5.1.3 Posterior decoding and the Baum-Welch Algorithm

Another way of finding a hidden state sequence explaining the data is to choose the $q'_t$ that maximizes $P(q'_t \mid O, \lambda)$. One needs to be aware that if $Q'$ is found this way then $P(Q' \mid O, \lambda)$ may be low because the individual $q'_t$ are chosen independently. Nevertheless this approach is a good choice for some applications. To compute $P(q'_t \mid O, \lambda)$ we need the backward algorithm. Define $\beta_t(i) = P(O_{t+1}O_{t+2}\ldots O_T \mid q'_t = i, \lambda)$. Without going into details $\beta_t(i)$ can be computed by the backward algorithm, which is similar to the forward and the Viterbi algorithms. Using the forward and the backward algorithms we can compute

$$\gamma_t(i) = P(q'_t \mid O, \lambda) = \frac{P(O, q'_t \mid \lambda)}{P(O \mid \lambda)} = \frac{\alpha_t(i) \beta_t(i)}{P(O \mid \lambda)} ,$$

where $P(O \mid \lambda)$ can be found by the forward algorithm as above.

The Baum-Welch algorithm is an expectation-maximization technique for optimizing $\lambda$ given sample data $O$, and it uses both $\alpha_t(i)$ and $\beta_t(i)$ in the computation of the expectation. The Baum-Welch algorithm optimizes $\lambda$, but in some cases the Hidden Markov Model is based on a physical problem that
imposes certain restrictions on $\lambda$, which the Baum-Welch algorithm will not be aware of. Assume $\lambda$ is a function of some other set of parameters $\theta$ which has a physical interpretation, $\lambda = f(\theta)$. In that case we can instead use the forward algorithm to compute $P(O | \lambda) = P(O | f(\theta))$, and use any suitable optimization algorithm to find the $\theta$ that maximizes $P(O | f(\theta))$. For example [47] uses a modified Newton-Raphson method to estimate speciation time, population size and recombination rate directly; and [144] use an unspecified optimization method to determine crossover rate, gene conversion rate and gene conversion tract length directly.

We will not go into further details regarding the backward algorithm, posterior decoding or the Baum-Welch algorithm, because they are not directly relevant to the work in the rest of this chapter. They are merely presented as important examples of the use of the forward algorithm, and because no text on Hidden Markov Models would be complete without.

5.1.4 Numerical stability

The results of the forward and the Viterbi algorithms are products of probabilities. Since these probabilities are generally smaller than one, the results will tend toward zero exponentially fast. Often the results of these algorithms will be stored on a computer in the IEEE 754 floating point format. This is relevant because IEEE 754 floating point numbers cannot store values arbitrarily close to zero, and if these algorithms were implemented naively the results would often be rounded to zero. For the Viterbi algorithm, a way to handle this is to compute $\log(P(Q | O, \lambda))$ instead. This is easy because the only operations used by the Viterbi algorithm is multiplication and maximum. The recursion simply becomes

$$\log(\delta_t(i)) = \begin{cases} 
\log(b_i(O_1)) + \log(\pi_i) & \text{if } t = 1 \\
\log(b_i(O_t)) + \max_j \{ \log(a_{ji}) + \log(\delta_{t-1}(j)) \} & \text{if } t > 1 
\end{cases},$$

with a similar change to the backtracking procedure.

Handling underflow in the forward algorithm is somewhat more involved. The basic approach is again to compute $\log(P(O | \lambda))$, but the forward algorithm requires summation and multiplication. Although it is technically possible to do summation in logarithmic space it is quite slow. A more elegant approach is to continuously scale the $\alpha_t(i)$ such that they never approach zero, and by storing the scales we can reconstruct $\log(P(O | \lambda))$. We define three new mutually recursive variables: $c_t$ is the scaling factor used at time step $t$. $\alpha'_t(i)$ corresponds to $\alpha_t(i)$, but is based on scaled values, and finally $\hat{\alpha}_t(i)$, which is the rescaled $\alpha'_t(i)$. Formally set

$$c_t = \sum_{i \in S} \alpha'_t(i),$$

$$\alpha'_t(i) = \begin{cases} 
b_i(O_1)\pi_i & \text{if } t = 1 \\
b_i(O_t)\sum_{j \in S} a_{ji}\hat{\alpha}_{t-1}(j) & \text{if } t > 1 
\end{cases},$$

and

$$\hat{\alpha}_t(i) = \frac{\alpha'_t(i)}{c_t}.$$
5.2 Parallelizing small Hidden Markov Models

Using induction it is straightforward to show that

\[ \alpha_t(i) = \left( \prod_{1 \leq t' \leq t} c_{t'} \right) \hat{\alpha}_t(i). \]

Since \( c_t \) is chosen so that \( \sum_{i \in S} \hat{\alpha}_t(i) = 1 \) we can find

\[ \log(\mathbb{P}(O | \lambda)) = \log \left( \sum_{i \in S} \alpha_T(i) \right) = \log \left( \prod_{1 \leq t' \leq T} c_{t'} \right) = \sum_{1 \leq t' \leq T} \log(c_{t'}). \]

5.2 Parallelizing small Hidden Markov Models

Previously the processor in a new computer could be expected to have a significantly higher clock-frequency than the one in an older computer. This meant that all software that ran on the old computer would be expected to run faster on the new machine, without modifications. Recently, however, modern processors do not operate at increasing clock frequencies, but instead add more processing cores, allowing for the parallel execution of several threads. To fully use a modern processor software needs to be rewritten to properly distribute the work between several processor cores.

5.2.1 Related Work

Assume we want to implement a parallel version of the forward algorithm above. To compute \( \alpha_t(i) \) we need to know \( \alpha_{t-1}(j) \) for all \( j \), but not any of \( \alpha_t(i') \), \( i' \neq i \). Thus we can split the set of all hidden states \( V \) between the available processors, and simply compute all \( \alpha_t(i) \), for all \( i \) and a given \( t \), in parallel.

This approach works well if the number of states \( N \) is large, compared to the number of processors, but the method needs to synchronize, that is, wait for all processors to agree they are finished, between each time step \( t \). The time to compute \( \alpha_t(i) \) is \( \mathcal{O}(N) \), so if \( N \) is small, \( \alpha_t(i) \) will be very fast to compute and each processor will only be given few of them to compute for each \( t \). This means that the time between each synchronization will be very short, and since synchronization is relatively slow the majority of the execution time will be spent waiting for synchronization instead of working on the Hidden Markov Model. This is the parallelization scheme used by HMMlib [114].

5.2.2 Research Contributions

Some rewriting of the recursions for computing \( \alpha_t(i) \), will reveal an alternative approach to parallelizing the forward algorithm. We will write the equations using linear algebra. Remember that the transitions \( A \) already are stored in a matrix. We take the initial probabilities \( \pi \) to be a column vector. Define

\[ B_t = \begin{bmatrix} b_1(O_t) \\ b_2(O_t) \\ \vdots \\ b_N(O_t) \end{bmatrix}, \]
the diagonal matrix of the emission probabilities at time $t$, and

$$\alpha_t = \begin{bmatrix} \alpha_t(1) \\ \alpha_t(2) \\ \vdots \\ \alpha_t(N) \end{bmatrix},$$

the column vector of all $\alpha_t(i)$ for a given $t$. We can now compute $\alpha_t$ as

$$\alpha_t = \begin{cases} B_1 \pi & \text{if } t = 1 \\ B_t A^T \alpha_{t-1} & \text{if } t > 1 \end{cases},$$

where $A^T$ is the matrix transpose of $A$. For convenience we also define

$$C_t = \begin{cases} B_1 \pi & \text{if } t = 1 \\ B_t A^T & \text{if } t > 1 \end{cases},$$

so that we get

$$\alpha_t = C_t C_{t-1} \ldots C_2 C_1.$$

If we consider the traditional forward algorithm we will see that it simply computes this matrix product from right to left. The trick is that we do not necessarily need to do it this way, since matrix multiplication is associative. Assuming we have $P$ processors we can simply cut the product into $P$ fragments and let each processor process one fragment. Doing it this way we only need to synchronize once, when all processors have finished their work, and we require relatively little communication between the processors. The processors each need to receive $A$, $B$, $\pi$ and their fragment of $O$, and they only need to return the resulting matrix. Exploiting associativity like this is a well-known technique in the field of parallel computing, and is called reduction. Using the naive algorithms, matrix-matrix multiplication is a factor of $N$ slower than matrix-vector multiplication. Since the traditional algorithm only uses matrix-vector multiplication and we use matrix-matrix multiplication we only expect our algorithm to be fast for Hidden Markov Models with a small state space. Things are complicated a bit because $C_1$ is a vector, thus a part of the work can be done as fast matrix-vector multiplications. Particularly $C_2 C_1$ will also be a vector, as will $C_3 C_2 C_1$, and so on and so forth. All this implies that the processor working on the first fragment of $O$ will work roughly a factor of $N$ times faster than the remaining processors and should thus receive a correspondingly larger fraction of the work. On a modern computer we cannot expect the $P$ processors to run deterministically at a constant speed. Therefore we actually split the workload into several smaller fragments and have the processors consume those greedily. To take maximum advantage of the first processor being fastest we let that one consume the fragments from one end of a queue, while the rest consumes fragments from the other end. For numerical stability it turns out we can reuse the exact same technique as was used in the traditional algorithm, except, in this case it is matrices that are scaled to sum to one, and not vectors.

Surprisingly the above strategy also work with the Viterbi algorithm. First we define a new operator

$$(P \times^m Q)_{ij} = \max_k \{ P_{ik} Q_{kj} \},$$
5.2. Parallelizing small Hidden Markov Models

Figure 5.1: Backtracking in Viterbi can be done through an intermediary matrix. Assume $\delta_{t+1} = M_{t+1} \times m \delta_t$. Then the maximally likely state $q'_t$ for column $\delta_t$ can be found using only $M_{t+1}$, $\delta_t$, and $q'_{t+1}$, which is similar to matrix multiplication, $(P \times Q)_{ij} = \sum_k P_{ik}Q_{kj}$, except summation has been replaced with a maximum. It turns out $\times m$ is associative. As with the forward algorithm above, we can now formulate the $\delta_t(i)$, using linear algebra

$$
\delta_t = \begin{bmatrix}
\delta_t(1) \\
\delta_t(2) \\
... \\
\delta_t(N)
\end{bmatrix},
\delta_t = \begin{cases}
C_1 & \text{if } t = 1 \\
C_t \times m \delta_{t-1} & \text{if } t > 1
\end{cases}.
$$

and due to the associativity of $\times m$ we can compute $\delta_T$ using parallel reduction.

Of course $\delta_T$ is only the last column of the traditional dynamic programming table, and when working with the Viterbi algorithm, we will normally be interested in backtracking to obtain the hidden state sequence $Q'$. Assume the observations were split into $F$ fragments at indices $t_1, t_2, \ldots, t_{F-1}$, and the matrix resulting from processing fragment $i$ is $M_i$ such that

$$
M_1 = C_{t_1} \times m C_{t_1-1} \times m \cdots \times m C_1 \\
M_i = C_{t_i} \times m C_{t_i-1} \times m \cdots \times m C_{t_i-1+1} \\
M_F = C_T \times m C_{T-1} \times m \cdots \times m C_{T_{F-1}+1}
$$

This means we can find any

$$
\delta_{t_i} = M_i \times m M_{i-1} \times m \cdots \times M_1,
$$

and the last state in $Q'$ since

$$
q'_T = \arg\max_{i \in S} \{\delta_T(i)\}.
$$

From $M_F$, $\delta_{T-1}$ and $q'_F$ we can find $q'_{F-1}$. In fact we can find any $q'_t$, given $M_{t+1}$, $\delta_t$ and $q'_{t+1}$ as

$$
q'_t = \arg\max_{j \in S} \{(M_{t+1}) q'_{t+1} j \cdot \delta_t(j)\}.
$$

With all $\delta_t$ and $q'_t$ in hand we can derive the rest of $Q'$ in parallel. A processor is given $\delta_t$ and $q'_{t+1}$ for a given fragment: $\delta_t$ is enough for it to fill out the
Chapter 5. Hidden Markov Models

Figure 5.2: Our method benchmarked against HMMlib and GHMM on a computer with eight hyper-threaded cores. Our implementation used 16 threads, and HMMlib used only eight, because hyper-threading actually decreased its performance. GHMM is inherently single-threaded.

traditional dynamic programming table for that fragment, and \( q'_{t+1} \) allows the processor to do backtracking through that table fragment.

We have implemented this in a C++ library, we call \texttt{parredHmmLib}, and benchmarked it against other Hidden Markov Model implementations. It turns out fast general-purpose libraries for Hidden Markov Models are rare. This is probably because the Hidden Markov Models rarely are used in the simple form presented here, but often have modifications. For example the transition and emission probabilities may depend on the time step \( t \) and external data. Also a given model may have a structure that allows for domain-specific simplifications and optimizations. Since the algorithms are relatively simple it will often make sense to make specialized implementations to allow for modifications and take advantage of optimizations. Nevertheless we have compared our implementation against the high-performance implementation HMMlib [114], and the very general implementation GHMM [118].

Our implementation is available from \texttt{http://www.birc.au.dk/~asand/parredhmmlib}. Further details on this work can be found in the paper presented in Chapter 13.

5.2.3 Future Work

Currently \texttt{parredHmmmLib} only implements the forward and Viterbi algorithm for a homogeneous multiprocessor. This work can be extended in at least two ways. One approach is to extend it to more different architectures. Since there is very little communication between processors one could imagine distributing it over a network of computers. Alternatively it might be a candidate for implementation on a graphics processing unit (GPU). The other way this work could be continued is to implement the posterior-decoding and the Baum-Welch
Both of those algorithms should be easy to parallelize, once the full tables from the forward and the backward algorithms are computed. Unfortunately our current algorithms do not compute those. On a system with shared memory one could employ the same strategy for computing the forward and backward tables as we currently use to compute the tables for Viterbi backtracking. However, if we want to combine the two above extensions, things become more complicated. If we are working on a network or combining the work from both a CPU and a GPU we cannot expect to rely on shared memory, and communicating the entire dynamic programming table between devices is likely to become a bottleneck.

### 5.3 $k$-best Viterbi on large Hidden Markov Models

As mentioned above the execution time of the traditional forward algorithm is $O(N^2T)$. For most practical purposes this is acceptable, but the $N^2$ term can become problematic if the number of distinct hidden states $N$ is very big. Assume that we want to use the forward algorithm to compute $P(O|\lambda)$, where $\lambda = f(\theta)$, to estimate $\theta$ using some optimization algorithm. In a private discussion Michael I. Jordan, Yun S. Song and Junming Yin suggested that a sufficient approximation might be to compute $\sum_{Q} P(O, Q|\lambda)$, where $Q$ is the set of the $k$ most likely state paths. This sum can be computed by an extension to the Viterbi algorithm. The Viterbi algorithm has the same asymptotic execution time as the forward algorithm that we are trying to replace, and the extension that finds the $k$ best paths is even slower, so at first this does not seem like a good idea. However, in the case of a large hidden state space, we can use a coarse-to-fine approach that can make the algorithm significantly faster.

#### 5.3.1 Related Work

Neither applying the coarse-to-fine approach to Viterbi, nor a $k$-best Viterbi is a new idea.

Assume we have a Hidden Markov Model $\lambda$. The idea of the coarse-to-fine approach is to create a simpler or coarser Hidden Markov Model $\lambda'$, and use results from that to guide our search in $\lambda$. $\lambda'$ should have a significantly smaller state space than $\lambda$ and each state in $\lambda'$ should represent several states in $\lambda$. Note that this approach can be repeated recursively, creating a series of ever coarser Hidden Markov Models. The details of how to use the results from $\lambda'$ to guide algorithms on $\lambda$ are less obvious. The naive approach is to run the Viterbi algorithm on $\lambda'$ first, and next run it on $\lambda$, while only considering states corresponding to coarse states visited in $\lambda'$. Another approach is to use posterior decoding to find the likelihood of each state at each time step and restrict $\lambda$ to states represented by states with a sufficiently good likelihood in the coarse model. These approaches have been used by several researchers [30, 31, 58]. While they are simple and fast they do not necessarily give the right result. The coarse Hidden Markov Model must necessarily contain less information than the fine one, and therefore the path found through that, might not be correct, which could exclude important states from consideration in the later
Figure 5.3: The coarse to fine approach of [108]. a) First fine states are clustered into coarser states. b) Next the Viterbi algorithm is run on this reduced state space. c) All states on the path found by the Viterbi algorithm is split into their fine states. d) The Viterbi algorithm is run again on the finer state space. This is repeated until the Viterbi algorithm only find fine states. Of course the coarse states may themselves be clustered recursively into ever coarser states.

step. A better algorithm was suggested in [108]. That algorithm has two requirements. First, all the probabilities in $\lambda'$ must be an upper bound on those of the simple states represented in $\lambda$. Of course that means that, strictly speaking, they will no longer be probabilities, since they are unlikely to sum to one. The Viterbi algorithm will give a sensible result anyway. Secondly, when a path through $\lambda'$ is found, the coarse states visited by that path is split into the finer states, but the remaining coarse states are kept in the model. The Viterbi algorithm must then be run several times until a path containing only fine states is found. This is somewhat slower, since the Viterbi algorithm need to be run many times, but [108] shows that it is guaranteed to find a path that is optimal in the original model $\lambda$.

Work has also been done on a $k$-best Viterbi, particularly in the speech recognition community [31]. My work is primarily based on [68]. In that work four algorithms, numbered 0 through 3 are suggested. I will refer to these algorithms as HC0 through HC3. The HC0 algorithm is the straightforward way to extend the Viterbi algorithm. The observation is that while you store the single best path $\delta_t(i)$ for each state/time step combination in the traditional algorithm, one can locally extend this to store the $k$ best paths for each state/time step combination for a $k$-best Viterbi. You compute the $kN$ possibly paths to a given state/time step, sort them, and only store the $k$ best ones. Their HC1 algorithm is a domain-specific optimization, that does not relate to the work in this dissertation. Their third algorithm HC2 is based on the observation that one does not necessarily have to compute all $kN$ different paths to a given state. If you keep the local best-path lists sorted you can merge them, similar to the merge in merge sort, and stop the merge when $k$ paths have been found, thus significantly reducing the computation time. The HC3 algorithm takes this one step further by exploiting that one will rarely need all $k$ best paths from the source states, thus one can delay the computation of these until they are actually needed. It should be noted that the algorithms HC2 and HC3 have quite different properties. Since the algorithm HC3 computes everything on-demand you do not need to know $k$ before the algorithm is run, but can keep pulling new solutions until some arbitrary condition is satisfied. On the other hand, we are not actually interested in the $k$ actual paths, but only their likelihoods. For HC2 this means we only need to refer to the solutions at time $t-1$ to compute
5.3. k-best Viterbi on large Hidden Markov Models

The $k$-best Viterbi algorithms of [68]. In the HC0 the $k$-best list for a given state/time step is generated by concatenating and sorting lists of all possible paths from the previous time step. In the HC2 algorithm the possible paths from the previous time step are instead merged into the current time step. In the HC3 algorithm the paths are again merged, but this time the previous time step is computed lazily on demand.

The work of Charniak and Johnson [31] needs to be explicitly mentioned, because they also combine a $k$-best approach with coarse-to-fine. However their work is not based on a Hidden Markov Model, but on a Probabilistic Context-free Grammar, and their coarse-to-fine work is not based on that of [108].

5.3.2 Research Contributions

We combine the above mentioned work of [108] and [68] to create two different coarse-to-fine $k$-best algorithms for estimating the forward probability – one algorithm based on the HC2 algorithm and one based on the HC3 algorithm. The algorithms inherently use several degrees of coarseness. In fact a binary tree of the hidden states are always built, with the internal nodes in the tree being states in the coarser Hidden Markov Model. Thus the first and coarsest set of states considered are the immediate children of the root of the tree, and every time an internal node is chosen to be split it is replaced by its children. How to build the state tree optimally is not obvious. We have experimented with several different ways of doing this, including clustering states that are similar according to various measures; building the tree so that states that are likely to be visited a priori are close to the root; and trees that are perfectly balanced, so that no states are far from the root. We would like to build a tree that minimizes the expected number of nodes visited, but have been unable to design a good model of when states are visited. Nevertheless the current clustering algorithm is based on this idea, although it is heuristic in nature. For details on our tree-construction heuristic we refer the reader to Section 14.2.5. One aspect that complicates tree construction is execution speed. Remember we denote the number of hidden states by $N$, the number of observable symbols by $M$, and the length of the input sequence by $T$. The traditional forward
algorithm, which we want to compete with, has an asymptotic execution time of $O(TN^2)$, and is actually quite fast in practice, so we need the tree construction to be significantly faster than that. My current heuristic tree construction implementation has an execution time of $O(MN^2)$, remembering that $M$ is the size of the alphabet of observable symbols.

The transition matrix $A$ has $N^2$ entries. This means that if $A$ is explicitly stored we cannot build the tree faster than $O(N^2)$. However there may be many cases where this is not strictly necessary. For example the initial reason for our research in this area was to apply it to the model in [144]. That work can have a very large state space, but it only has sixteen distinct transition probabilities. For use with the coarse-to-fine framework we can compute which of those sixteen different transition probabilities apply between two coarse states, and take their maximum, in constant time. Similarly we have experimented with using this work in a setting where each hidden state was a rectangle in a two-dimensional euclidean space and the transition probability, that depended only on their relative coordinates, could be computed in constant time.

Combining HC2 with the coarse-to-fine approach is relatively straightforward: We simply run the HC2 algorithm repeatedly on increasingly fine state spaces. Making a coarse-to-fine implementation of the HC3 algorithm is more challenging, due to the laziness of the algorithm. In our current implementation states are split as paths running through them are lazily computed, which is in the spirit of the HC3 algorithm, but does not perform particularly well. We refer to these methods as C2FHC2 and C2FHC3.

The algorithms have been implemented in a C++ library which can be downloaded from http://www.birc.dk/~jn/c2flib. For the $k$-best approach we have compared our implementations against our own implementations of HC2 and HC3. Furthermore we have also benchmarked the speed of the algorithms as an alternative to the forward algorithm for parameter optimization in the work of [144]. Details on this work can be found in Chapter 14.
Table 5.1: Running time of algorithms on the method from [144]. This particular Hidden Markov Model has enough structure that the authors in the original paper managed to make the forward algorithm run in $O(TN)$ which is difficult to compete with.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC2</td>
<td>34669 s</td>
</tr>
<tr>
<td>HC3</td>
<td>8858 s</td>
</tr>
<tr>
<td>C2FHC2</td>
<td>831 s</td>
</tr>
<tr>
<td>C2FHC3</td>
<td>55.3s</td>
</tr>
<tr>
<td>Forward</td>
<td>137 s</td>
</tr>
<tr>
<td>Original method [144]</td>
<td>15.2s</td>
</tr>
</tbody>
</table>

5.3.3 Future Work

The current implementation strictly works with binary trees. Making the algorithm work on general trees would allow trees with internal nodes of higher degree. On one hand this would decrease the height of the trees, so that the $k$-best Viterbi algorithms would have to be run fewer times thus making the algorithm faster, but on the other hand the state space would grow more in each iteration making the following iterations relatively slower, yet I suspect it might lead to an overall faster algorithm. To use trees of higher degree one would of course have to build them first, and most of our current tree-building algorithms inherently build binary trees. One might be able to come up with algorithms for turning binary trees into multifurcating ones: In particular, our currently used algorithm can compute a heuristic expected cost of any subtree independent of degree. I would suggest building a binary tree, and then restructuring that to a multifurcating one by optimizing subtrees locally.

I mostly consider the coarse-to-fine HC2 algorithm to be complete, but it might be possible to improve performance by choosing which states to split in a better way; finding the $k$ best paths in every run is probably not optimal. Either one could search for fewer paths when the model is still very coarse, or one could make paths going through coarse states count as multiple paths.

As for the coarse-to-fine HC3 algorithm, I suspect the performance could be significantly improved. The performance of the coarse-to-fine algorithms depend very much upon choosing the right states to split into children. When a state is split into children the scores for all reachable states at later time steps may change. However, in our current coarse-to-fine HC3 implementation those scores are not updated immediately, due to the laziness of the algorithm. Therefore the algorithm may make poor choice for further states to split, which can hurt performance significantly. I propose that another way to decide which states to split, or a way to propagate information about split states might improve performance. For example one might implement something akin to the current coarse-to-fine HC2 algorithm by alternating between two separate phases of computing the $k$ best paths and splitting states.
Finally one would obviously want to investigate how well a $k$ best Viterbi algorithm actually perform as a replacement for the forward algorithm in parameter optimization. It might be possible to say something about how big $k$ should be as a function of the model size, but since the HC3 algorithm is an on-line algorithm one could also imagine a scheme where one decides $k$ dynamically based on other criterion. For example one could estimate how much of the total probability mass has been computed and stop when a certain fraction of this has been found.
Chapter 6

Comparing trees

Many relationships in data can be described by trees. Phylogenetic trees are of course a well-known example in bioinformatics. Often there are several different algorithms for inferring these trees from data, which do not necessarily yield the same trees. To study the properties of tree-building algorithms systematically a measure of the distance between two trees is useful.

I have worked on a fast algorithm for computing the quartet distance between two general trees. First we submitted a version claimed to run in quadratic time. However, after it was accepted it was found to be flawed. We created a new version, with a somewhat more complicated algorithm, claiming only that it runs in sub-cubic time. Unfortunately it was too late to update the submitted paper. Later a master’s student [78] implemented the algorithm in C++ and we published an extended version as a journal paper including experimental results from his implementations.


6.1 Background

Several measures of the distance between two trees have been developed. One such measure is the symmetric distance, which is also known as the Robinson Foulds distance or partition metric [110,111]. Any edge in a tree partitions the leaves of the tree into two sets. The symmetric distance between two trees is the number of partitions that exist in one tree but not the other. Surprisingly it turns out that this can be computed in time linear in the number of leaves in the trees [43]. Although the symmetric distance is very fast to compute, many pairs of trees have no partitions in common, make the measure useless for trees that are moderately dissimilar [124].
Another measure, and the one that is studied in my work, is the quartet distance. The smallest unrooted tree that has more than one possible topology has four leaves. With four leaves the possible unrooted topologies are exactly the four shown in Figure 6.1. A set of four leaves we call a quartet. Given a tree and a quartet, the tree will induce one of the four possible topologies on the quartet - the topology found by removing all leaves that are not a part of the quartet. The quartet distance between two trees is the number of quartets that does not have the same topology in both trees [51].

The two above distances are related in that they both count the number of different substructures. Another way to define a distance is to count the minimum number of operations needed to transform one tree into the other one, for some appropriate operation. Three such measures are the nearest neighbor interchange [136], the subtree prune and regraft, and the tree bisection and reconnection distances. Consider an internal edge in a binary tree: It will have four other edges connected to it through two vertices. The nearest neighbor interchange operation is to select an edge and swap one of the incident edges with one connected at the other end of the edge. A subtree prune and regraft operation is to select a subtree; prune it, by cutting the edge connecting it to the rest of the tree; and then graft it onto the tree in another location. Finally a tree bisection and reconnection operation is to choose an edge; remove it, thus splitting the tree into two trees; and finally reconnecting the trees by adding a new edge. All three operations are illustrated in Figure 6.2. Notice the nearest neighbor interchange operation corresponds to a subtree prune and regraft operation, restricted so that the subtree must be regrafted at distance of at most one vertex from where it was pruned. Also, the difference between a subtree prune and regraft operation and a tree bisection and reconnection operation is that for the subtree prune and regraft operation one end of the edge connecting the two trees cannot be moved [10]. There are two problems with these transformation based measures. First, the measures do not distinguish how large subtrees they move, so two quite different trees could have a tree bisection and reconnection distance of one. Second, they seem difficult to compute efficiently [10,21,41,42].
6.2 Related Work

Consider the comparison of two unrooted trees $T_1$ and $T_2$, with the same set of labeled leaves. The size of the set of the leaves we will denote by $n$. Let $g$ denote the highest degree of any node in either tree. A quartet is a set of four leaves, and will have exactly one of the four topologies shown in Figure 6.1. A quartet with a butterfly topology, clustering $a$ and $b$ together, and $c$ and $d$ together we will write as $ab|cd$, while one with a star topology will be written as $abcd$. Furthermore a quartet can be placed in exactly one of the five categories shown in Figure 6.3, depending on which topology it has in $T_1$ and $T_2$. We will denote the number of quartets in each category by the symbol shown in Figure 6.3. From these we can compute the total number of butterfly topologies $x_i$ in tree $T_i$ as

$$x_i = s + d + r_i,$$

the number of different quartets as

$$q_d = d + r_1 + r_2,$$

the number of similar quartets as

$$q_s = s + u,$$
Figure 6.3: Given a quartet and two trees the quartet may be classified into one of five categories, depending on whether it is has a star topology or a butterfly topology in either tree, and whether the topologies are the same. For each category we will use the shown symbol to represent the number of quartets in that category.

and finally the total number of quartets as

\[ q = s + d + r_1 + r_2 + u = \binom{n}{4}. \]

The quartet distance is exactly \( q_d \). The algorithms for computing the quartet distance generally computes some subset of these values, from which the remaining, or at least \( q_d \) can be computed. Some comparison algorithms are restricted to binary trees, which are trees where all internal nodes have degree 3. In a binary tree \( r_1 = r_2 = u = 0 \), thus an algorithm for binary trees need only compute \( s \) or \( d \), since in that case

\[ q_d = d = \binom{n}{4} - s. \]

6.2.1 Algorithms based on paths and centers

It seems the earliest work on a fast algorithm for quartet distance is a bachelors thesis [44], in which an \( \mathcal{O}(n^3) \) time and \( \mathcal{O}(n) \) space algorithm is presented for the computation of \( x_1, s, r_1 \) and \( r_2 \). The core of the algorithm is to split all the quartets into sets of quartets \( S_{ki,j} \), for each pair of leaves \((i, j)\), where \( i < j \), in each tree \( T_k \). \( S_{ki,j} \) contains all butterfly quartets of the form \( ij|kl \), and all star quartets of the form \( ijk \), where \( i \) is the minimum leaf in the quartet. This means that each butterfly quartet will be found in exactly one such set, but each star will be found in three of them. The algorithm then computes \( x_1, s, r_1 \) and \( r_2 \), for each pair of corresponding sets \( S_{1i,j} \) and \( S_{2i,j} \). Since there are \( \mathcal{O}(n^2) \) sets this needs to do be computed in linear time. For each tree the
algorithm first finds the path between leaves $i$ and $j$. Next all nodes on the path $1 \ldots n_p$ and all nodes directly connected to one of these nodes $1 \ldots n_p$, are numbered. The observation is that for any quartets of the form $ijkl$; $k$ and $l$ will be in the same subtrees adjacent to the same node on the path between $i$ and $j$, while any quartets of the form $ijkl$ will have $k$ and $l$ in different subtrees, but connected to the same node on the path between $i$ and $j$. When the numbering has been done for both trees the algorithm annotates all leaves $k$, $k > i$, $k \neq j$ with $((a_1, b_1), (a_2, b_2))$, where $(a_i, b_i)$ is the identification of the subtree containing $k$ according to the numbering of the nodes on, and adjacent to, the path between $i$ and $j$. By sorting the leaves primarily by $(a_1, b_1)$ and secondarily by $(a_2, b_2)$, which can be done in linear time using radix sort, all nodes residing in the same subtrees in $T_1$ and $T_2$ can be identified as blocks of the same $((a_1, b_1), (a_2, b_2))$ values. The values of $x_1$, $s$, and $r_1$ can now be computed, for $Sk_{ij}$ combinatorially. Sorting by $(a_2, b_2)$ primarily and $(a_1, b_1)$ secondarily allows the computation of $r_2$.

A somewhat related method is presented in [34]. It counts $q_s$ in time $O(n^3)$ and space $O(n^2)$. The method is based on centers. Select three nodes, $a$, $b$, and $c$. The center of $a$, $b$, and $c$ is the unique node where the paths between them meet. As shown in Section 6.2.2 the sizes of intersections between all subtrees in $T_1$ and $T_2$ can be precomputed and stored in time and space $O(n^2)$. Using these sizes and the center, the paper shows how to count all quartets with those three leaves and the same topology in $T_1$ and $T_2$ in constant time. Since there are $O(n^3)$ triplets, the centers must be found in constant time. This is done by, for each pair of nodes $(a, b)$, finding the path between them, and for all remaining nodes, annotating which node in that path they are connected to. There are $O(n^2)$ pairs of nodes, and finding the path and annotating the nodes can be done in $O(n)$ time. Once the annotation has been performed, the center of $a$, $b$, and $c$ can be found in constant time.

### 6.2.2 Algorithms based on claims of butterfly quartets

Bryant et al. in [26] present an algorithm that computes $s$ in time and space $O(n^2)$, but for binary trees only. An important contribution of that paper is an algorithm for computing the sizes of intersections between all subtrees in $T_1$ and all subtrees of $T_2$ in time and space $O(n^2)$. The algorithm is dynamic programming. The intersection size of two leaves is 1 if they are the same leaf and 0 if they are not; for two internal nodes the size of the intersection is the sum of sizes over all intersections of the children. In a binary tree all quartets have a butterfly topology $abcd$. Consider an internal edge $e$ in a tree. The edge will split the leaves of the tree into two sets $A$ and $B$. For each quartet of leaves $a_1, a_2 \in A$, $b_1, b_2 \in B$, the edge $e$ induces the butterfly quartet $a_1a_2b_1b_2$. Comparing an edge in $T_1$ with an edge in $T_2$ the total number of shared butterfly quartets, induced by both those edges can be found in constant time, using the precomputed tables of tree intersections. However, many quartets will be induced by several edges. The paper claims it avoids double counting by a preprocessing step, where each internal edge claims as many quartets as possible that has not been claimed by any other edge, but from the paper it is not obvious
Chapter 6. Comparing trees

Figure 6.4: An edge can be split into two undirected edges.

Figure 6.5: A directed edge $e$ claims all butterfly quartets of the form $ab|cd$, where $ab$ are in the subtree behind $e$, and $c$ and $d$ are in two different subtrees, among those immediately in front of $e$.

The above work is extended to general trees in [34]. The authors observe that

$$r_1 + r_2 = x_1 + x_2 - 2(s + d),$$

which means that the quartet distance can be found as

$$q_{d} = d + r_1 + r_2 = r_1 + r_2 - 2s - d.$$ 

Basically, the complete quartet distance can be computed without ever considering star topologies. They also note that $r_i$ can be computed by any algorithm for computing $s$, since the total number of butterfly quartets in a tree must be exactly the number of butterfly quartets it shares with itself. Thus one only needs algorithms for computing $s$ and $d$ quickly. The work uses the algorithms for precomputing the subtree intersections and the concepts of quartets induced by edges from above. The problem with multiple counting is solved quite elegantly the following way: Split all edges into two directed edges, as shown in Figure 6.4. Now define that a directed edge $e$ claims all butterfly quartets $ab|cd$, where $a$ and $b$ are in subtrees behind $e$, and $c$ and $d$ are in two different subtrees in front of $e$, as shown in Figure 6.5. That way each butterfly quartet will be claimed exactly two times. Consider two directed edges, $e_1$ and $e_2$ from $T_1$ and $T_2$ respectively. Shared butterfly quartets claimed by $e_1$ and $e_2$ will have the form $ab|cd$ with $a, b$ in subtrees $A_1, A_2$ behind $e_1, e_2$, and $c$ and $d$ in different subtrees $B_1, B_2$ and $C_1, C_2$ in front of them, as shown in Figure 6.6.
6.2. Related Work

Figure 6.6: All shared butterfly quartets claimed by edges $e_1$ and $e_2$ will be of the same form in both trees.

Figure 6.7: All different butterfly quartets claimed by edges $e_1$ and $e_2$ will be of the form $ab|cd$ in $T_1$ and $ac|bd$ in $T_2$.

These can be counted as

$$\frac{1}{2} \left( \frac{|A_1 \cap A_2|}{2} \right) \sum_{B_1 \neq A_1, B_2 \neq A_2} |B_1 \cap B_2| \sum_{C_1 \neq A_1, B_1, C_2 \neq A_2, B_2} |C_1 \cap C_2| . \quad (6.1)$$

Different butterfly topologies will have the form $ab|cd$, with $a,b$ in subtree $A_1$ behind $e_1$, and $c,d$ in subtrees $B_1,C_1$ in front of it, while they will have the form $ac|bd$, with $a,c$ in subtree $A_2$ behind $e_2$ and $b,d$ in different subtrees $B_2,C_2$ in front of it, as shown in Figure 6.7. These can be counted as

$$|A_1 \cap A_2| \sum_{B_1 \neq A_1, B_2 \neq A_2} |A_1 \cap B_2||B_1 \cap A_2| \sum_{C_1 \neq A_1, B_1, C_2 \neq A_2, B_2} |C_1 \cap C_2| . \quad (6.2)$$

If this algorithm was to be implemented naively it would have a running time of $O(g^4n^2)$, due to the time it takes to compute the sums. In [34] this is reduced to $O(g^2n^2)$, basically by expanding all nodes of degree greater than three, to several binary nodes.

Let $i_1$ and $i_2$ be the number of internal nodes in $T_1$ and $T_2$ respectively.

Let the **internal degree** of a node be the number of edges connecting that node
to other internal nodes. Let $id_1$ and $id_2$ be the largest internal degrees of any nodes in $T_1$ and $T_2$ respectively. In [35] an algorithm is presented that runs in time $O(n + i_1i_2 \min\{id_1, id_2\})$, and space $O(n + i_1i_2)$. Two techniques are used to achieve this. First instead of equations 6.1 and 6.2 other, somewhat more complicated equations, where it is not necessary to sum over the leaves, are used. Second, assume you need to compute the double sum

$$\sum_i f(i) \sum_{j \neq i} g(j) . \tag{6.3}$$

Naively the computation of this would take time $O(n^2)$, where $n$ is the number of possible values of $i$ and $j$. However, it can also be done efficiently by precomputing

$$S = \sum_j g(j) ,$$

which reduces 6.3 to

$$\sum_i f(i) (S - g(i)) .$$

This computation, including the precomputation, only takes time $O(n)$. Using this trick repeatedly they bring the time for counting shared butterfly topologies between two nodes with internal degrees $id_1$ and $id_2$ down to $O(id_1id_2)$, and the time for counting different butterfly topologies down to $O(id_1id_2 \min\{id_1, id_2\})$. Summing over all pairs of nodes this gives total running times, of $O(i_1i_2)$ and $O(id_1id_2 \min\{id_1, id_2\})$ respectively. The paper also demonstrates how to formulate the algorithm for computing intersections of leaf sets such that it runs in time $O(n + i_1i_2)$.

### 6.2.3 Algorithms based on colorings

In [23] an algorithm is presented that can compute $s$ for two binary trees in time $O(n \log n)$. Their algorithm is based on colorings. An internal node $v$ in $T_1$ claims all quartets claimed by any directed edge incident to that node. Since the tree is binary $v$ will have exactly three subtrees attached to it. We say that the leaves are colored according to $v$, if and only if, the leaves are colored, such that each of the three subtrees has its own color. Furthermore we say that a quartet is compatible with a certain coloring if two of the leaves have their own color, and the remaining two share a third one. This means the quartets compatible with a coloring according to $v$ are exactly the quartets claimed by $v$. For each node $v$ in $T_1$ the algorithm colors the leaves according to that node, and count how many quartets in $T_2$ are compatible with that coloring – these are exactly the quartets claimed by $v$ that are also quartets in $T_2$. Summing these gives $s$. To do this efficiently they present a clever data structure which allows fast update of colorings and retrieval of the number of compatible quartets. The algorithm is extended to general trees running in time $O(g^d n \log n)$ in [126].
6.3 Research Contributions

Our work is on a fast algorithm for computing the quartet distance between general trees. The algorithm is a claim-based approach like [26, 34, 35]. It iterates over all \(O(n^2)\) pairs of edges from the two trees and counts the number of shared and different butterfly quartets, claimed by each pair. Applying the trick of precomputed sums, used on equation 6.3, several times, we can compute equations 6.1 and 6.2 in constant time. For the number of shared butterfly quartets \(s\), this works out well, as the needed precomputed sums can be found in \(O(n^2)\), which is hidden by the \(O(n^2)\) it takes to iterate over all pairs of edges. Things are somewhat more complicated for the different quartets \((r_1 + r_2)\). Among the required precomputations are tables of the form

\[
J[i, j] = \sum_{k=1}^{d_1} I[i, k] I[j, k] ,
\]

where \(i, j = 1, \ldots, d_1\), with \(d_1\) and \(d_2\) as degrees of internal nodes in the trees. If computed naively this would result in a running time of \(O(n^3)\). The above may, however be recognized as matrix multiplication, and fast algorithms for the multiplication of square matrices have been developed. Assume an algorithm for multiplying two \(N \times N\) matrices in time \(O(N^\omega)\) is available. We can then compute 6.4 in three different ways. We could use the naive matrix multiplication algorithm and get \(O(d_1^2 d_2)\); pad \(I\) to become square and use the advanced matrix multiplication algorithm for a time of \(O(\max\{d_1, d_2\}^\omega)\); or we could compute a symmetric set of tables instead and use the naive algorithm again to get a time of \(O(d_1 d_2^2)\). Thus the precomputations for a pair nodes can be done in time

\[
O(\min\{d_1^2 d_2, d_1 d_2^2, \max\{d_1, d_2\}^\omega\}) .
\]

Let \(\alpha = \frac{\omega - 1}{2}\). By choosing matrix multiplication algorithm based on whether \(d_1 < d_2\); and whether \(d_1 < d_2^2\) or symmetrically, the total running time for the preprocessing for counting different quartets becomes \(O(n^{2+\alpha})\). This means the time for preprocessing dominates the running time of the algorithm.

The Coppersmith-Winograd algorithm [39] can multiply two square matrices in time \(O(N^{2.376})\) which would make our algorithm run in \(O(n^{2.688})\), making it the fastest algorithm for general trees, that is not dependent on the degree of the inner nodes.

A master’s student [76] has implemented our algorithm using a BLAS implementation for matrix multiplication. In particular the one from the Mac OS X vecLib framework was used in our experiments. The student furthermore implemented the \(O(n^3)\) algorithm from [34] and a naive algorithm based on listing all quartets. Those algorithm were chosen because they apply to general trees and their execution time does not depend on the internal degree of the nodes. For experiments four different kinds of trees were generated: Random general trees; random binary trees; trees with a star-topology; and trees with one internal node of degree \(\frac{n}{2}\) and \(\frac{n}{2}\) internal nodes of degree 3. There are ten possible pairings of these types of trees, and we benchmarked all the algorithms on all the possible pairs of trees, on tree sizes from \(n = 10\) up to \(n = 14698\).
Figure 6.8: Execution time of our algorithm plotted against that of [34] and a naive one. In a double-logarithmic plot like this, $n^k$ becomes a straight line. The inlaid lines are not regression lines, but added to help the reader judge the time complexity of the implementations.

The results are shown in Figure 6.8. In most cases our algorithm seems to be closer to $O(n^2)$ than to $O(n^3)$. One series of data points for our algorithm converges toward $O(n^3)$, though. Inspection of the data shows this to be the comparison of star trees against star trees, thus the pair with the nodes of the largest degrees. If the matrix multiplication algorithm has $\omega = 3$, then $\alpha = 1$ and our algorithm runs in time $O(n^3)$, so we interpret this as the matrix multiplication algorithm of the vecLib being asymptotically slow. The two worst and the two best tree pairs are plotted in Figure 6.9, for context.

The paper on this topic can be found in Chapter 15, and the software is available from http://birc.au.dk/software/qdist.

### 6.4 Future Work

Future work would include finding an algorithm for the computation of the quartet distance on general trees that is better than the currently best. I consider the currently best methods to be our $O(n^{2.668})$ time algorithm and the $O(n + i_1 i_2 \min\{id_1, id_2\})$ time algorithm of [35]. The $O(g^9 n \log n)$ time algorithm of [126], obviously also needs to be considered, but I find the $g^9$ term to be excessive.
6.4. Future Work

Figure 6.9: Execution time of our algorithm comparing trees with different topologies. In a double-logarithmic plot like this, \( n^k \) becomes a straight line. The inlaid lines are not regression lines, but added to help the reader judge the time complexity of the implementations.

I suspect it will be possible to find an algorithm that can compute the quartet distance between general trees in time \( O(n^2) \), but even if an algorithm is found that computes a matrix product in time \( O(n^2) \), our algorithm would only run in time \( O(n^{2.5}) \). Remember that our work computes the number of shared butterfly quartets \( s \) in time \( O(n^2) \), while [35] computes it in \( O(n + i_1i_2) \) – thus it is the computation of the different butterfly topologies \( d \) that is the dominating step in both algorithms. Having worked with the equations it seems it is the asymmetry in the different quartets that is the problem: Compare Figure 6.6 to 6.7 or Equation 6.1 to 6.2. Of course an \( O(n^2) \) time algorithm for computing \( d \) would be satisfactory, but that is proving elusive. If we consider the \( O(n^2) \) algorithms we currently have available we can compute \( s, x_1, x_2 \) and \( q \), which you will remember from Section 6.2 as

\[
\begin{align*}
  s &= s \\
  x_1 &= s + d + r_1 \\
  x_2 &= s + d + r_2 \\
  q &= s + d + r_1 + r_2 + u
\end{align*}
\]

This is four linear equations, with five variables, which is not enough equations to give a unique solution. However, it allows us to recognize that it would be sufficient to find an efficient algorithm for the computation of \( u \), which would
allow the computations of the quartet distance as

\[ q - s - u = q_d , \]

or an algorithm for \((r_1 + r_2)\), which could lead to the quartet distance as

\[ \frac{x_1 + x_2 + r_1 + r_2 - 2s}{2} = d + r_1 + r_2 = q_d . \]

The problem with the computation of those is that they require star-quartets to be considered, which means that for each node four edges have to be considered, instead of the three necessary for the comparison of butterfly quartets. On the other hand the computation of \(u\) has some symmetry that the computation of \(d\) does not, which might be useful.
Chapter 7

Summary and Conclusion

In summary I have worked with algorithms in five different areas of bioinformatics. I have worked with the similarity of small molecules, with data for association mapping, with molecular biological data, with Hidden Markov Models, and with tree similarity.

With respect to molecule similarity I have developed three data structures for fast similarity searching, based on the Tanimoto coefficient. The data structures are: the $k$D-grid, the singlebit tree and the multibit tree. They are all different variants of trees and based on computing bounds on the Tanimoto coefficient for all bitstrings in a subtree. Furthermore I have developed an algorithm based on inverted indices for fast computation of LINGOsim molecule similarity. Basically molecules are indexes by their LINGO substrings, instead of the LINGO substrings being indexed by molecule, which allows fast lookup of all molecules containing a given LINGO substring.

For association mapping I have developed a file format and software library called SNPFile. The file format stores both the SNP data and any arbitrary meta-data through a serialization framework. On one hand this file format allows one to handle large amounts of SNP data efficiently, and on the other hand it also relieves the information management burden, because all relevant data is collected into one file.

Related to molecular biological data I have analyzed expression data from exosome knockout cells. When the exosome is knocked out RNA fragments that are otherwise degraded are stabilized, allowing us to observe them. It turns out DNA upstream active transcription start sites is transcribed although it does not have a known function. My co-authors and I term these new transcriptions PROMPTs. PROMPT activity correlates with the activity at the downstream gene, and with activity of other transcription indicators in the area.

I have also developed new algorithms for Hidden Markov Models. In particular I show how formulating Hidden Markov Models as linear algebra leads naturally to algorithms based on parallel reduction. These algorithms require more computational work than the traditional algorithms, but allow for parallelization with very little communications overhead. This can give a significant speed-up on models with a small state space, that can otherwise be difficult to parallelize. I have also developed two coarse-to-fine $k$-best Viterbi algorithms. These algorithms can find the $k$ most likely hidden state paths, without neces-
Chapter 7. Summary and Conclusion

Sarly considering all states at all time steps. This can give a significant speed-up on models with a very large state space, and can even be faster than the traditional algorithm that only finds the single best path. Thus our coarse-to-fine \( k \)-best algorithm might be used as an approximation to the forward algorithm for computing the total likelihood on models with a very large state space.

Finally I have presented a new algorithm for the computation of the quartet distance. The new algorithm runs in time \( O(n^{2+\alpha}) \), where \( \alpha = \frac{\log_2 n - 1}{2} \), if we can multiply two \( N \times N \) matrices in time \( O(N^\omega) \). This makes it the fastest algorithm that is independent of the degree of the nodes in the trees. The algorithm has been implemented and benchmarked against other algorithms that are independent of the degree of the nodes in the trees, and was found to be the fastest one.

In general we want to increase our understanding about biology in general and human biology in particular. We want this, both because biology is interesting in its own right, but also because we want to find cures for human diseases and ailments. My above work on PROMPTs and real molecular biological data of course contributes directly to the understanding human of biology, and my work on SNPFile is directly applicable to research in human diseases. My other work is relevant because the cost of bioinformatic analysis is brought down when they can be performed faster. As analysis become cheaper and faster it becomes feasible to implement them in software used by non-experts, and maybe in an interactive fashion. This moves the analysis and data closer to the researchers and technicians in the laboratory, who use the results, and thus increase the rate at which biological experiments can be performed.
Part II

Papers
Chapter 8

A tree-based method for the rapid screening of chemical fingerprints

The paper A tree-based method for the rapid screening of chemical fingerprints presented in this chapter has been published as a conference paper and in a journal.


The journal paper extends the conference paper by adding additional experimental results. Except for typographical and formatting changes the content of this chapter is equal to the journal paper [78]. An implementation of the data structures presented in this chapter is available at http://www.birc.au.dk/~tgk/TanimotoQuery.
A tree-based method for the rapid screening of chemical fingerprints

Thomas G. Kristensen∗† Jesper Nielsen∗‡
Christian N. S. Pedersen∗§

Abstract

The fingerprint of a molecule is a bitstring based on its structure, constructed such that structurally similar molecules will have similar fingerprints. Molecular fingerprints can be used in an initial phase of drug development for identifying novel drug candidates by screening large databases for molecules with fingerprints similar to a query fingerprint.

In this paper, we present a method which efficiently finds all fingerprints in a database with Tanimoto coefficient to the query fingerprint above a user defined threshold. The method is based on two novel data structures for rapid screening of large databases: the $k$D grid and the Multibit tree. The $k$D grid is based on splitting the fingerprints into $k$ shorter bitstrings and utilising these to compute bounds on the similarity of the complete bitstrings. The Multibit tree uses hierarchical clustering and similarity within each cluster to compute similar bounds. We have implemented our method and tested it on a large real-world data set. Our experiments show that our method yields approximately a three-fold speed-up over previous methods.

Using the novel $k$D grid and Multibit tree significantly reduce the time needed for searching databases of fingerprints. This will allow researchers to (1) perform more searches than previously possible and (2) to easily search large databases.

8.1 Introduction

When developing novel drugs, researchers are faced with the task of selecting a subset of all commercially available molecules for further experiments. There are more than 8 million such molecules available [71], and it is not feasible to perform computationally expensive calculations on each one. Therefore, the need arises for fast screening methods for identifying the molecules that are most likely to have an effect on a given disease. It is often the case that a molecule with some effect is already known, e.g. from an already existing

∗Bioinformatics Research Center (BiRC), Aarhus University, C. F. Møllers Allé 8, DK-8000 Aarhus C, Denmark.
†E-mail: tgk@birc.au.dk.
‡E-mail: jn@birc.au.dk.
§E-mail: cstorm@birc.au.dk.
drug. An obvious initial screening method presents itself, namely to identify the molecules which are similar to this known molecule. To implement this screening method one must decide on a representation of the molecules and a similarity measure between representations of molecules. Several representations and similarity measures have been proposed [57, 83, 139]. We focus on molecular fingerprints. A fingerprint for a given molecule is a bitstring of size $N$ which summarises structural information about the molecule [83]. Fingerprints should be constructed such that if two fingerprints are very similar, so are the molecules which they represent. There are several ways of measuring the similarity between fingerprints [139]. We focus on the Tanimoto coefficient, which is a normalised measure of how many bits two fingerprints share. It is 1.0 when the fingerprints are the same, and strictly smaller than 1.0 when they are not. Molecular fingerprints in combination with the Tanimoto coefficient have been used successfully in previous studies [141].

We focus on the screening problem of finding all fingerprints in a database with Tanimoto coefficient to a query fingerprint above a given threshold, e.g. 0.9. Previous attempts have been made to improve the query time. One approach is to reduce the number of fingerprints in the database for which the Tanimoto coefficient to the query fingerprint has to be computed explicitly. This includes storing the fingerprints in the database in a vector of bins [127], or in a trie like structure [122], such that searching certain bins, or parts of the trie, can be avoided based on an upper-bound on the Tanimoto coefficient between the query fingerprint and all fingerprints in individual bins or subtries. Another approach is to store an XOR summary, i.e. a shorter bitstring, of each fingerprint in the database, and use these as rough upper bounds on the maximal Tanimoto coefficients achievable, before calculating the exact coefficients [14].

In this paper, we present an efficient method for the screening problem, which is based on an extension of an upper bound given in [127] and two novel tree based data structures for storing and retrieving fingerprints. To further reduce the query time we also utilise the XOR summary strategy [14]. We have implemented our method and tested it on a realistic data set. Our experiments clearly demonstrate that it is superior to previous strategies, as it yields a three-fold speed-up over the previous best method.

### 8.2 Methods

A fingerprint is a bitstring of length $N$. Let $A$ and $B$ be bitstrings, and let $|A|$ denote the number of 1-bits in $A$. Let $A \land B$ denote the logical and of $A$ and $B$, that is, $A \land B$ is the bitstring that has 1-bits in exactly those positions where both $A$ and $B$ do. Likewise, let $A \lor B$ denote the logical or of $A$ and $B$, that is, $A \lor B$ is the bitstring that has 1-bits in exactly those positions where either $A$ or $B$ do. With this notation the Tanimoto coefficient becomes:

$$S_T(A, B) = \frac{|A \land B|}{|A \lor B|}.$$

Figure 8.1 shows an example the usage of this notation. In the following, we present a method for finding all fingerprints $B$ in a database of fingerprints
8.2. Methods

with a Tanimoto coefficient above some query-specific threshold $S_{\text{min}}$ to a query fingerprint $A$. The method is based on two novel data structures, the $k$D grid and the Multibit tree, for storing the database of fingerprints.

### 8.2.1 $k$D grid

Swamidass et al. showed in [127] that if $|A|$ and $|B|$ are known, $S_T(A, B)$ can be upper-bounded by

$$S_{\text{max}} = \frac{\min(|A|, |B|)}{\max(|A|, |B|)}.$$  

This bound can be used to speed up the search, by storing the database of fingerprints in $N + 1$ buckets such that bitstring $B$ is stored in the $|B|$th bucket. When searching for bitstrings similar to a query bitstring $A$ it is sufficient to examine the buckets where $S_{\text{max}} \geq S_{\text{min}}$.

We have generalised this strategy. Select a number of dimensions $k$ and split the bitstrings into $k$ equally sized fragments such that

$$A = A_1 \cdot A_2 \cdot \ldots \cdot A_k$$

$$B = B_1 \cdot B_2 \cdot \ldots \cdot B_k,$$

where $X \cdot Y$ is the concatenation of bitstrings $X$ and $Y$.

The values $|A_1|, |A_2|, \ldots, |A_k|$ and $|B_1|, |B_2|, \ldots, |B_k|$ can be used to obtain a tighter bound than $S_{\text{max}}$. Let $N_i$ be the length of $A_i$ and $B_i$. The $k$D grid is a $k$-dimensional cube of size $(N_1 + 1) \times (N_2 + 1) \times \ldots \times (N_k + 1)$. Each grid point is a bucket and the fingerprint $B$ is stored in the bucket at coordinates $(n_1, n_2, \ldots, n_k)$, where $n_i = |B_i|$. An example of such a grid is illustrated in Fig. 8.2. By comparing the partial coordinates $(n_1, n_2, \ldots, n_i)$ of a given bucket to $|A_1|, |A_2|, \ldots, |A_i|$, where $i \leq k$, it is possible to upper-bound the Tanimoto coefficient between $A$ and every $B$ in that bucket. By looking at the partial coordinates $(n_1, n_2, \ldots, n_{i-1})$, we can use this to quickly identify those partial coordinates $(n_1, n_2, \ldots, n_i)$ that may contain fingerprints $B$ with a Tanimoto coefficient above $S_{\text{min}}$.

Assume the algorithm is visiting a partial coordinate at level $i$ in the data structure. The indices $n_1, n_2, \ldots, n_{i-1}$ are known, but we need to compute which

\[
\begin{array}{c|c|c}
A & 1 & 0 & 1 & 1 & 0 & 1 & |A| = 4 \\
B & 1 & 1 & 0 & 1 & 0 & 0 & |B| = 3 \\
A \wedge B & 1 & 0 & 0 & 1 & 0 & 0 & |A \wedge B| = 2 \\
A \lor B & 1 & 1 & 1 & 1 & 0 & 1 & |A \lor B| = 5 \\
S_T(A, B) = \frac{2}{5} \\
\end{array}
\]
Figure 8.2: Example of a kD-grid with $k = 3$. $B$ is split into smaller substrings and the count of 1-bits in each determines where in $B$ is placed in the grid. The small inner cube shows the placement of $B$.

$n_i$ to visit at this level. The entries to be visited further down the data structure $n_{i+1}, ..., n_k$ are, of course, unknown at this point. A bound can be calculated in the following manner.

$$S_T(A, B) = \frac{|A \wedge B|}{|A \lor B|}$$

$$= \frac{\sum_{j=1}^{k} |A_j \wedge B_j|}{\sum_{j=1}^{k} |A_j \lor B_j|}$$

$$\leq \frac{\sum_{j=1}^{k} \min(|A_j|, n_j)}{\sum_{j=1}^{k} \max(|A_j|, n_j)}$$

$$= \frac{\sum_{j=1}^{i-1} \min(|A_j|, n_j) + \min(|A_i|, n_i) + \sum_{j=i+1}^{k} \min(|A_j|, n_j)}{\sum_{j=1}^{i-1} \max(|A_j|, n_j) + \max(|A_i|, n_i) + \sum_{j=i+1}^{k} \max(|A_j|, n_j)}$$

$$\leq \frac{\sum_{j=1}^{i-1} \min(|A_j|, n_j) + \min(|A_i|, n_i) + \sum_{j=i+1}^{k} |A_j|}{\sum_{j=1}^{i-1} \max(|A_j|, n_j) + \max(|A_i|, n_i) + \sum_{j=i+1}^{k} |A_j|}$$

$$= S_{\text{grid max}}$$

The $n_i$s to visit lie in an interval and it is thus sufficient to compute the upper and lower indices of this interval, $n_u$ and $n_l$ respectively. Setting $S_{\text{min}} = S_{\text{grid max}}$. 

$$B: \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ n_1 = 2 & n_2 = 3 & n_3 = 1 \end{bmatrix}$$
isolating \( n_i \) and ensuring that the result is an integer in the range \( 0 \ldots N_i \) gives:

\[
\begin{align*}
  n_l &= \max \left( \left\lceil S_{\min}(A_i^{\max} + |A_i| + A_i^{|j|}) \right\rceil, 0 \right) \\
  n_u &= \min \left( \left\lfloor A_i^{\min} + |A_i| + A_i^{|j|} - S_{\min}(A_i^{\max} + A_i^{|j|}) \right\rfloor, N_i \right)
\end{align*}
\]

where \( A_i^{\min} = \sum_{j=1}^{i-1} \min(|A_j|, n_j) \) is a bound on the number of 1-bits in the logical \( \land \) in the first part of the bitstrings. \( A_i^{\max} = \sum_{j=1}^{i-1} \max(|A_j|, n_j) \) is a bound for the logical \( \lor \) in the first part of the bitstrings. Similarly, \( A_i^{|j|} = \sum_{j=i+1}^{k} |A_j| \) is a bound on the last part.

Note that in the case where \( k = 1 \) this datastructure simply becomes the list presented by Swamidass et al. [127], and in the case where \( k = N \) the datastructure becomes the binary trie presented by Smellie [122].

We have implemented the \( k \)D grid as a list of lists, where any list containing no fingerprints is omitted. See Fig. 8.3 for an example of a 4D grid containing four bitstrings. The fingerprints stored in a single bucket in the \( k \)D grid can be organised in a number of ways. The most naive approach is to store them in a simple list which has to be searched linearly. We propose to store them in tree structures as explained below.

8.2.2 Singlebit tree

The Singlebit tree is a binary tree which stores the fingerprints of a single bucket from a \( k \)D grid. At each node in the tree a position in the bitstring is chosen. All fingerprints with a zero at that position are stored in the left subtree while all those with a one are stored in the right subtree. This division
is continued recursively until all the fingerprints in a given node are the same. When searching for a query bitstring $A$ in the tree it now becomes possible, by comparing $A$ to the path from the root of the tree to a given node, to compute an upper bound $S_{\text{max}}$ on $S_T(A, B)$ for every fingerprint $B$ in the subtree of that given node.

Given two bitstring $A$ and $B$ let $M_{ij}$ be the number of positions where $A$ has an $i$ and $B$ has a $j$. There are four possible combinations of $i$ and $j$, namely $M_{00}$, $M_{01}$, $M_{10}$ and $M_{11}$.

The path from the root of a tree to a node defines lower limits $m_{ij}$ on $M_{ij}$ for every fingerprint in the subtree of that node. Let $u_{ij}$ denote the unknown difference between $M_{ij}$ and $m_{ij}$, that is $u_{ij} = M_{ij} - m_{ij}$. Remember that $|B| = \sum_{i=1}^{k} n_k$ is known when processing a given bucket.

By using

\[
\begin{align*}
  u_{10} + u_{11} & = |A| - m_{10} - m_{11} \\
  u_{01} + u_{11} & = |B| - m_{01} - m_{11} \\
  u_{11} & \leq \min(u_{01} + u_{11}, u_{10} + u_{11}) \\
  u_{01} + u_{10} + u_{11} & \geq \max(u_{01} + u_{11}, u_{10} + u_{11})
\end{align*}
\]

an upper bound on the Tanimoto coefficient of any fingerprint $B$ in the subtree can then be calculated as

\[
S_T(A, B) = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}
\]

\[
= \frac{m_{11} + u_{11}}{m_{01} + u_{01} + m_{10} + u_{10} + m_{11} + u_{11}}
\]

\[
\leq \frac{m_{01} + m_{10} + m_{11} + \max(u_{01} + u_{11}, u_{10} + u_{11})}{m_{01} + m_{10} + m_{11} + \min(|A| - m_{10}, |B| - m_{01})}
\]

\[
= \frac{m_{01} + m_{10} + \max(|A| - m_{10}, |B| - m_{01})}{m_{01} + m_{10} + \max(|A| - m_{10}, |B| - m_{01})}
\]

\[
= S_{\text{max}}^{\text{single}}.
\]

When building the tree data structure it is not immediately obvious how best to choose which bit positions to split the data on, at a given node. The implemented approach is to go through all the children of the node and choose the bit which best splits them into two parts of equal size, in the hope that this creates a well-balanced tree. It should be noted that the tree structure that gives the best search time is not necessarily a well-balanced tree. Figure 8.4 shows an example of a Singlebit tree.

The Singlebit tree can also be used to store all the fingerprints in the database without a $kD$ grid. In this case, however, $|B|$ is no longer available and thus the $S_{\text{max}}^{\text{single}}$ bound cannot be used. A less tight bound can be formulated, but experiments, not included in this paper, indicate that this is a poor strategy.
8.2. Methods

Figure 8.4: Example of a Singlebit tree. The black squares mark the bits chosen for the given node, while the grey squares mark bits chosen at an ancestor. The grey triangles represent subtrees omitted to keep this example simple. Assume we are searching for the bitstring $A$ in the example. When examining the node marked by the arrow we have the knowledge shown in $B'$ about all children of that node. Comparing $A$ against $B'$ gives us $m_{00} = 0$, $m_{01} = 0$, $m_{10} = 1$ and $m_{11} = 2$. Thus $S_{\text{single max}} = \frac{5}{4}$. Indeed we find that $S_T(A, B) = \frac{3}{7}$ and $S_T(A, B') = \frac{4}{6}$.

8.2.3 Multibit tree

The experiments in Sec. 8.3 unfortunately show that using the $k$D grid combined with Singlebit trees decreases performance compared to using the $k$D grid and simple lists. The fingerprints used in our experiments have a length of 1024 bits. In our experiments no Singlebit tree was observed to contain more than 40,000 fingerprints. This implies that the expected height of the Singlebit trees is no more than 15 (as we aim for balanced trees cf. above). Consequently, the algorithm will only obtain information about 15 out of 1024 bits before reaching the fingerprints. A strategy for obtaining more information is to store a list of bit positions, along with an annotation of whether each bit is zero or one, in each node. The bits in this list are called the match-bits.

The Multibit tree is an extension of the Singlebit tree, where we no longer demand that all children of a given node are split according to the value of a single bit. In fact we only demand that the data is arranged in some binary tree. The match-bits of a given node are computed as all bits that are not a match-bit in any ancestor and for which all fingerprints in the leaves of the node have the same value. Note that a node could easily have no match-bits. When searching through the Multibit tree, the query bitstring $A$ is compared to the match-bits of each visited node and $m_{00}$, $m_{01}$, $m_{10}$ and $m_{11}$ are updated.
8.3 Experiments

We have implemented the $kD$ grid and the Single- and Multibit tree in Java. The implementation along with all test data is available at

http://www.birc.au.dk/~tgk/TanimotoQuery/.
8.4. Results

Using these implementations, we have constructed several search methods corresponding to the different combinations of the data structures. We have examined the $k$D grid for $k = 1, 2, 3$ and $4$, where the fingerprints in the buckets are stored in a simple list, a Singlebit tree or a Multibit tree. For purposes of comparison, we have implemented a linear search strategy, that simply examines all fingerprints in the database. We have also implemented the strategy of “pruning using the bit-bound approach first, followed by pruning using the difference of the number of 1-bits in the XOR-compressed vectors, followed by pruning using the XOR approach” from [14]. This strategy will hereafter simply be known as Baldi. A trick of comparing the XOR-folded bitstrings [14] immediately before computing the true Tanimoto coefficient, is used in all our strategies to improve performance. The length of the XOR summary is set to 128, as suggested in [14]. An experiment, not included in this paper, confirmed that this is indeed the optimal size of the XOR fingerprint. We have chosen to reimplement related methods in order to make an unbiased comparision of the running times independent of programming language differences.

The methods are tested on a real-world data set by downloading version 8 of the ZINC database [71], consisting of roughly 8.5 million commercially available molecules. Note that only 2 million of the molecules have actually been used, due to memory constraints. The distribution of one-bits is presented in Fig. 8.6, where it can be seen there are many buckets in the 1D grid that will be empty.

The experiments were performed on an Intel Core 2 Duo running at 2.5GHz and with 2GB of RAM. Fingerprints were generated using the CDK fingerprint generator [125] which has a standard fingerprint size $N$ of 1024. One molecule timed out and did not generate a fingerprint. We have performed our tests on different sizes of the data set, from 100,000 to 2,000,000 fingerprints in 100,000 increments. For each data set size, the entire data structure is created. Next, the first 100 fingerprints in the database are used for queries. We measure the query time and the space consumption.

Figure 8.6: Distribution of the number of bits set in the 1024 bit CDK fingerprints from the ZINC database.
Chapter 8. A Tree-based method for screening of fingerprints

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>line type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singlebit tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multibit tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) (b) (c)

Figure 8.7: Different strategies tested with $k = 1, \ldots, 4$. Each experiment is performed 100 times, and the average query time is presented. All experiments are performed with a $S_{\text{min}}$ of 0.9. The three graphs (a) – (c) show the performance of the three bucket types for the different values of $k$. The best $k$ for each method is presented in graph (d) along with the simple linear search results and Baldi.

8.4 Results

Figure 8.7 shows the average query time for the different strategies and different values of $k$ plotted against the database size. We note that the Multibit tree in a 1D grid is best for all sizes. Surprisingly, the simple list, for an appropriately high value of $k$, is faster than the Singlebit tree, yet slower than the Multibit tree. This is probably due to the fact that the Singlebit trees are too small to contain sufficient information for an efficient pruning: the entire tree is traversed, which is slower than traversing the corresponding list implementation. All three approaches (List, Singlebit- and Multibit trees) are clearly superior to the Baldi approach, which in turn is better than a simple linear search (with the XOR folding trick).
8.4. Results

Figure 8.8: Experiments with simple lists for $k = 1, \ldots, 10$. Each test is performed 100 times, and the average query time is presented. All experiments are performed with a $S_{\text{min}}$ of 0.9. Missing data points are from runs with insufficient memory.

From Fig. 8.7a we notice that the List strategy seems to become faster for increasing $k$. This trend is further investigated in Fig. 8.8, which indicates that a $k$ of three or four seems optimal. As $k$ grows the grid becomes larger and more time consuming to traverse while the lists in the buckets become shorter. For sufficiently large values of $k$, the time spent pruning buckets exceeds the time visiting buckets containing superfluous fingerprints. The Singlebit tree data in Fig. 8.7b indicates that the optimal value of $k$ is three. It seems the trees become too small to contain enough information for an efficient pruning, when $k$ reaches four. In Fig. 8.7c we see the Multibit tree. Again, a too large $k$ will actually slow down the data structure. This can be explained with arguments similar to those for the Singlebit tree. Surprisingly, it seems a $k$ as low as one is optimal.

Figure 8.9 shows the memory usage per fingerprint as a function of the number of loaded fingerprints. The first thing we note is that the Multibit tree uses significantly more memory than the other strategies. This is due to the need to store a variable number of match-bits in each node. The second thing to note is the space usage for different $k$’s. In the worst case, where all buckets contain fingerprints, the memory consumption per fingerprint, for the grid alone, becomes $O \left( \frac{1}{n} \left( \frac{N}{k} \right)^k \right)$, where $n$ is the number of fingerprints in the database. Thus we are not surprised by our actual results.

Figure 8.10 shows the search time as a function of the Tanimoto threshold. In general we note that the simpler and more naive data structures performs better for a low Tanimoto threshold. This is due to the fact that, for a low
Chapter 8. A Tree-based method for screening of fingerprints

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>line type</td>
<td>-</td>
<td>- - -</td>
<td>- - -</td>
<td>- - - - -</td>
</tr>
<tr>
<td>List</td>
<td>Single bit tree</td>
<td>Multibit tree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.9: The memory consumption of the data structure for different strategies tested with $k = 1, \ldots, 4$. The three graphs (a) – (c) show the performance of the three bucket types for the different values of $k$. The $k$ yielding the fastest query time for each method is presented in graph (d) along with the simple linear search results and Baldi.

Tanimoto threshold a large part of the entire database will be returned. In these cases very little pruning can be done, and it is faster to run through a simple list than to traverse a tree and compare bits at each node. Of course we should remember that we are interested in performing searches for similar molecules, which means large Tanimoto thresholds.

The reason why linear search is not constant time for a constant data set is that, while it will always visit all fingerprints, the time for visiting a given fingerprint is not constant due to the XOR folding trick.

The running times of the different methods depend on the number of Tanimoto coefficients between pairs of bitstrings that must be calculated explicitly. This number depends on the method and not on the programming language in which the method is implemented, and is thus an implementation independent
8.5 Conclusion

In this paper we have presented a method for finding all fingerprints in a database with Tanimoto coefficient to a query fingerprint above a user defined
Chapter 8. A Tree-based method for screening of fingerprints

Our method is based on a generalisation of the bounds developed in [127] to multiple dimensions. Our generalisation results in a tighter bound, and experiments indicate that this results in a performance increase. Furthermore, we have examined the possibility of utilising trees as secondary data structures in the buckets. Again, our experiments clearly demonstrate that this leads to a significant performance increase.

Our methods allow researchers to search larger databases faster than previously possible. The use of larger databases should increase the likelihood of finding relevant matches. The faster query times decreases the effort and time needed to do a search. This allow more searches to be done, either for more molecules or with different thresholds $S_{\text{min}}$ on the Tanimoto coefficient. Both of these features increase the usefulness of fingerprint based searches for the researcher in the laboratory.

Our method is currently limited by the rather larger memory consumption of the Multibit tree. Another implementation might remedy this situation somewhat. Otherwise we suggest an I/O efficient implementation where the tree is kept on disk.

To increase the speed of our method further we are aware of two approaches. Firstly, the best way to construct the Multibit trees remain uninvestigated. Secondly, a tighter coupling between the Multibit tree and the $k$D grid would

Figure 8.11: The fraction of the database for which the Tanimoto coefficient is calculated explicitly, measured for different number of fingerprints. The Tanimoto threshold is kept at 0.9.
8.6 Competing interests

The authors declare that they have no competing interests.

8.7 Authors contributions

The project was initiated by TGK, who also came up with the SingleBit tree. JN invented the kD grid and the Multibit tree. All datastructures were implemented, refined and benchmarked by JN and TGK. TGK, JN and CNSP wrote the article. CNSP furthermore functioned in an advisory role.
Chapter 9

Using Inverted Indices for Accelerating LINGO Calculations

The paper *Using Inverted Indices for Accelerating LINGO Calculations* presented in this chapter was published in a journal.


Except for typographical and formatting changes the content of this chapter is equal to [80]. An implementation of the data structures and methods presented in this chapter is available at [http://www.birc.au.dk/~tgk/ii](http://www.birc.au.dk/~tgk/ii).
9.1. Introduction

A common task in drug discovery is the computational analysis of chemical compounds which can take the form of e.g. predicting numerical properties such as the logP value or performing screening studies in which new drug candidates are sought from a large database of available molecules. Both problems are often managed by representing molecules as either graphs or 3D structures. The number of available molecules is increasing rapidly: the ZINC database contains more than 13 million molecules [71] and the GDB-13, containing all synthesizable molecules up to size 13, contains 970 million chemical compounds [20]. Therefore, novel vector models such as fingerprints and numerical descriptors have been proposed and tested as predictors and screening tools. Several studies have examined the effectiveness of these methods [92,131] and other studies...
Chapter 9. Using Inverted Indices for Accelerating LINGO Calculations

\[ S = c1cccc1cL \]
(a) Example SMILES

\[ S' = c0cccc0cL \]
(b) Simplified string

<table>
<thead>
<tr>
<th>LINGO</th>
<th>freq.</th>
<th>LINGO</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0cc</td>
<td>1</td>
<td>c0cc</td>
<td>54</td>
</tr>
<tr>
<td>0ccc</td>
<td>1</td>
<td>0ccc</td>
<td>22</td>
</tr>
<tr>
<td>cccc</td>
<td>2</td>
<td>cccc</td>
<td>30</td>
</tr>
<tr>
<td>ccc0</td>
<td>1</td>
<td>cccc'</td>
<td>42</td>
</tr>
<tr>
<td>ccc0L</td>
<td>1</td>
<td>ccc0</td>
<td>7</td>
</tr>
<tr>
<td>cccL</td>
<td>1</td>
<td>cc0L</td>
<td>101</td>
</tr>
</tbody>
</table>
(c) LINGOs

(d) Verbose rep.

(e) Inverted indices rep.

Figure 9.1: Example of generating LINGO multisets in the verbose and inverted indices representation. The inverted index lists contain references to the original SMILES string \( S \).

have already examined the acceleration of queries into molecular databases represented as vectors \([14, 76, 77, 122, 127]\). If a database is stored as canonical SMILES strings, both fingerprints and numerical descriptors require an explicit model of the chemical structure as a graph or as a 3D structure to be constructed. The LINGO representation \([130]\) avoids this problem by generating a representation of a molecule directly from canonical SMILES strings \([137]\) of the molecules.

Given a canonical SMILES string for a molecule the set of LINGOs in this study is generated by replacing all ring closure numbers with zero and all occurrences of Br and Cl with R and L respectively. The LINGOs are defined as all substrings of size \( q \) in the resulting string.

9.1 (a, b, c) illustrates the generation of the LINGOs for a small molecule with \( q = 4 \), as this value of \( q \) is suggested as optimal by two previous empirical studies \([59, 130]\). In the article introducing LINGOs the model was used to predict ADME (absorption, distribution, metabolism and excretion) properties with an \( r^2 \) correlation of 0.93 to experimentally observed values of logP \([130]\). The study furthermore compared the intermolecular similarity between bioisosteric molecules with that of randomly sampled pairs of molecules using the LINGOsim measure, finding a large discriminatory power.
9.2 Previous work

A multiset is a set where elements are allowed to occur multiple times. Given two multisets of LINGOs the LINGOsim is defined as the Tanimoto coefficient between the sets. If \( A = \{ 'cccc', 'c0cc', 'cccc', 'cccc' \} \) and \( B = \{ 'cScc', 'cccc', 'cccc' \} \) then

\[
\text{LINGOsim}(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|\{ 'cccc', 'cccc' \}|}{|\{ 'cccc', 'c0cc', 'cccc', 'cccc', 'cScc' \}|} = \frac{2}{5}
\]

Grant et al. used the LINGOsim as a ranking tool in a screening study in which it performed comparable to Daylight fingerprints [59]. Whereas most previous work concerning fingerprints and numerical descriptors has been focused on accelerating queries into large databases [14, 76, 122, 127] most of the work concerning LINGOs has been focused on the calculation of LINGOsim between pairs of molecules. The first fast LINGOsim method was based on constructing a finite state machine (FSM) based on the transformed SMILES string in 9.1 (b). In the initial study of this FSM it was described as being constructed as repeatedly inserting LINGOs into a trie [59], which can be done in linear time [8].

A less complicated method called SIML is based on encoding the frequency table from 9.1 (c) as two word-arrays: one containing the codes for the LINGOs and one for encoding their frequency [63]. One study suggests that a size of four is optimal [59], and as every ASCII character can be encoded in 8 bit, a LINGO of size four can be encoded in 32 bits which is precisely a word on common hardware. If the lists are sorted by LINGO code the LINGOsim can be calculated by a parallel run through two pairs of lists which is linear in the number of distinct LINGOs in the two strings. The SIML study used SIMD instructions and GPU hardware and compared its performance to that of a commercial implementation by OpenEye with speedups in the order of a factor of 80. It is, however, limited to a \( q \) of four or less on 32-bit hardware and requires a very fast GPU to obtain a significant speedup.

9.3 Verbose representation

A problem with the SIML encoding scheme is that it is bound to both \( q \) and the word size of the underlying hardware. Instead of encoding each character of the LINGO, we propose generating ids for each LINGO in the SMILES strings as they are observed when reading the strings. Multiple occurrences of the same LINGO in one SMILES string is given different ids as illustrated in 9.1 (d). Generating these ids is done by using a trie as in the FSM method, storing ids in the leaves. Inserting LINGOs in the trie is done in linear time and the entire set of SMILES strings is inserted in linear time. If each id is stored in one word the verbose representation can store more different LINGOs than the SIML encoding in the same word size. This is because non-occurring LINGOs are not stored, in contrast to SIML which also encodes impossible LINGOs such as 'h43f' or 'c(\[]S'. The SIML representation uses two arrays for representing
Table 9.1: Frequencies of size four LINGOs in the Maybridge and ZINC data.

<table>
<thead>
<tr>
<th>Times LINGOs repeated</th>
<th>Maybridge</th>
<th>ZINC avg. per molecule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26.353</td>
<td>42.793</td>
</tr>
<tr>
<td>2</td>
<td>2.434</td>
<td>8.391</td>
</tr>
<tr>
<td>3</td>
<td>0.159</td>
<td>3.352</td>
</tr>
<tr>
<td>more than 4</td>
<td>0.000</td>
<td>9.001</td>
</tr>
</tbody>
</table>

Table 9.2: The average word consumption of size four LINGO multisets in the SIML and verbose representation on the Maybridge and ZINC data. The ZINC data is generated using CDK which adds explicit hydrogen information, which accounts for them being larger than the Maybridge data.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Maybridge</th>
<th>ZINC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIML rep.</td>
<td>64.4</td>
<td>127.1</td>
</tr>
<tr>
<td>Verbose rep.</td>
<td>31.8</td>
<td>155.4</td>
</tr>
</tbody>
</table>

A LINGO multiset but the multisets can be represented more verbosely with only one array with multiple occurrences of the same LINGO represented as multiple different ids, as illustrated with the two 'cccc' LINGOs in 9.1 (d). This verbose representation has the downside that it will potentially take up more space: the SMILES string 'cccccccc' would take up two words in SIML but five in the verbose representation.

However, analysing the test data from Maybridge presented in the Experiments section reveals that the average SMILES length is 35.2 (yielding 64.4 words in SIML) whereas the average number of ids in the verbose representation is only 31.8 as demonstrated in 9.2. The ZINC data has slightly longer SMILES strings with more repetitions (9.1) and the SIML representation is slightly shorter than the verbose representation. If 4 byte integers are used for representing the ids the verbose representation takes up less than four times the memory of the original SMILES strings.

The presented verbose representation can be interpreted as sparse fingerprints in which case the LINGOsim is the Tanimoto coefficient between fingerprints. This implies that all data structures developed for effectively performing queries into fingerprint databases [14,76,77,122,127] can be used on LINGOsim queries into databases of LINGO multisets.

### 9.4 Inverted indices method

This paper proposes calculating the LINGOsim between a target LINGO multiset and a database of LINGO multisets by storing the database as inverted indices. Inverted indices are also used to handle the T-occurrence problem.
9.4. Inverted indices method

\( x_1 : 0, 1, 2, 3 \quad x_2 : 0, 2, 3, 4 \)
\( x_3 : 2, 4, 5 \quad x_4 : 1, 3, 4 \)

(a) Input verbose representation.

\[
\begin{array}{cccccc}
I_0 & I_1 & I_2 & I_3 & I_4 & I_5 \\
x_1 & x_1 & x_1 & x_1 & x_2 & x_3 \\
x_2 & x_4 & x_2 & x_2 & x_3 & x_4 \\
x_3 & x_4 & x_4 & x_4 & & \\
\end{array}
\]

(b) Inverted indices datastructure.

\( x : 2, 4, 5 \)

(c) Query verbose representation.

\[
\begin{array}{cccc}
x_1 & x_2 & x_3 & x_4 \\
1 & 2 & 3 & 1 \\
\end{array}
\]

(d) The \( C \) counting vector.

Figure 9.2: Example of the data structure and algorithm. The input (a) is used to build an inverted indices table (b), which lists all molecules associated with a given id. To perform a query with a molecule \( x \) (c) the counting vector \( C \) is created by running through list \( I_2, I_4 \) and \( I_5 \) in the inverted indices table and increasing the entries corresponding to the encountered molecules (d). From this vector the similarities can be computed.

From string algorithms: given a query string and a database of strings, identify all database strings that share more than \( T \) substrings of size \( q \) with the query [85, 116]. The idea is to store the LINGO multisets as a vector where each cell represents one of the LINGO ids from the verbose representation. A database of \( n \) multisets \( x_1, \ldots, x_n \) can be stored as a vector \( I \) where each cell \( I_k \) stores a list of all the multisets containing \( k \), as illustrated in 9.1 (e). The LINGOsim between a multiset \( x_i \) and every other multiset \( x_j \) in the database can be calculated by first calculating the intersection sizes between \( x_i \) and every \( x_j \). To do this, let \( C \) be a counting vector of length \( n \) initialised with all zeros and let \( x_i \) contain the \( m \) ids \( x_{i,1}, \ldots, x_{i,m} \). Next, traverse the inverted indices lists \( I_{x_{i,1}}, \ldots, I_{x_{i,m}} \) and increment the counter \( C_{x_j} \) every time \( x_j \) is observed. An example of this is shown in 9.2. Afterwards \( C_{x_j} \) will contain \( |x_i \cap x_j| \) from which the LINGOsim can be calculated by using \( |x_i \cup x_j| = |x_i| + |x_j| - |x_i \cap x_j| \). Note that this strategy only works if the verbose representation contains unique ids for multiple occurrences of LINGOs within the same multisets as they would otherwise be counted multiple times when the inverted indices lists are traversed. The inverted indices data structure is not limited to LINGO multisets but can also be used to calculate the Tanimoto coefficients for general fingerprints.

The inverted indices are constructed by first identifying the largest id in the
data set. This id is used to find the size of $I$ so that this can be allocated. Next, run through all the LINGO multisets $x_i$ and insert them into the lists $I_{x_i,1}, \ldots, I_{x_i,m}$. All this takes linear time and takes up memory linear in the size of the verbose representation.

### 9.5 Experiments

Experiments were performed on data from Maybridge, taken from the SIML study and on data generated using the Chemistry Development Kit [125] canonical SMILES generator on molecules from the ZINC database version 8, subset 10 [71]. The SIML data contains 4,096 SMILES strings with an average length of 35.2 and the ZINC data contains the first 65,536 SMILES strings from ZINC with an average length of 158.4. LINGOs were generated for $q = 4$ as in previous studies using the method from the original paper [130]. Three methods were evaluated: the LINGOsim calculator from OpenEye [6] (OE), the SIML implementation without GPU support and a C implementation of our inverted indices method (IIM). The OpenEye LINGOsim calculator was chosen because it is used elsewhere in the literature [130]. It is not FSM based. Each method was measured as in the SIML study, that is on the time it takes to fill out a similarity matrix for a set of LINGO multisets, not including the time to load the LINGOs into memory or outputting the similarity matrix to disk or screen. As the matrix is symmetric, it is only necessary to compute half the entries. However, the matrix in our experiments is too big to fit in memory, and it is therefore streamed, which means that the old values cannot be read and every entry has to be computed explicitly by all three methods. The resulting matrix can be used for performing statistical calculations, clustering or similar data analysis.

All experiments were performed on an Intel Core 2 Quad Q9450 2.66 GHz machine with 4 GB of RAM running Ubuntu version 9.04 with GCC 4.3.3 using the -O3 optimisation flag. Presented results are from a single experiment.

### 9.6 Results

As presented in 9.3, re-running the OE and SIML implementations on the Maybridge benchmark data yields results very similar to those in the original study. The data in the first column is taken from Table 2 and Table 3 in the SIML article [63] – the GPU measurements are not from the same machine as the first four rows. As can be seen from 9.3 the IIM method is faster than all the tested methods, even those using multiple cores and GPUs. For one core the speedup compared to the OpenEye implementation is of a factor of 37 and drops to 31 when the implementations use all four cores. Compared to the SIML implementation the speedup is of a factor of 13 when both implementations are run on one core and 11 when the implementations are allowed to use all four cores. In all cases the IIM method outperforms the previous methods. Surprisingly, the one core version is faster than all but the GPU implementations.
Table 9.3: Results from experiments replicated from Haque et al. [63] on the Maybridge data containing 4096 SMILES strings, along with the results of using the IIM method. The GPU running times were not replicated as we did not have access to the same hardware.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Haque et al. [63] time (ms)</th>
<th>Replicated time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OE</td>
<td>15 060</td>
<td>14 860</td>
</tr>
<tr>
<td>SIML</td>
<td>5 460</td>
<td>5 421</td>
</tr>
<tr>
<td>OE 4 cores</td>
<td>3 880</td>
<td>3 860</td>
</tr>
<tr>
<td>SIML 4 cores</td>
<td>1 420</td>
<td>1 435</td>
</tr>
<tr>
<td>SIML GPU (GeForce GTS 250)</td>
<td>270</td>
<td>–</td>
</tr>
<tr>
<td>SIML GPU (Tesla T10)</td>
<td>215</td>
<td>–</td>
</tr>
<tr>
<td>IIM</td>
<td>–</td>
<td>407</td>
</tr>
<tr>
<td>IIM 4 cores</td>
<td>–</td>
<td>125</td>
</tr>
</tbody>
</table>

Table 9.4: Timing results in seconds from running the implementations on the CDK generated SMILES strings on the ZINC database.

<table>
<thead>
<tr>
<th>SMILES</th>
<th>OE 1 core</th>
<th>OE 4 cores</th>
<th>SIML 1 core</th>
<th>SIML 4 cores</th>
<th>IIM 1 core</th>
<th>IIM 4 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 384</td>
<td>1 493</td>
<td>378</td>
<td>176</td>
<td>44</td>
<td>67</td>
<td>17</td>
</tr>
<tr>
<td>32 768</td>
<td>5 928</td>
<td>1 502</td>
<td>703</td>
<td>178</td>
<td>270</td>
<td>72</td>
</tr>
<tr>
<td>65 536</td>
<td>23 791</td>
<td>6 022</td>
<td>2 812</td>
<td>718</td>
<td>1 093</td>
<td>294</td>
</tr>
</tbody>
</table>

Running the implementations on the ZINC data set yields the running times presented in 9.4. There is a large difference in the length of the SMILES strings between the two data sets and the observations from the ZINC set is therefore a bit slower than that of the Maybridge set and gives rise to different observations in speedup. For one core the speedup between OE and the IIM method drops to a steady factor of 22 across the tested database sizes and the speedup between the SIML implementation and the IIM method drops to a factor of 2.6. The speedups remain the same when using all four cores.

Experiments not presented here were performed to examine if the data size or the data structure is responsible for the decline in speedup. Experimental data was generated by transforming the Maybridge data, extending the lines using concatenation to match the length of the lines in the ZINC data. The ZINC lines were shortened to match those of the Maybridge data. The experiments revealed that both factors contributed equally to the deterioration of the speedup.

There is also a slight difference in the achieved speedup on four cores when going from the Maybridge to the ZINC data. On the Maybridge data, the IIM achieves 81% of linear speedup, while 93% of the achievable speedup is gained on the ZINC data (using all 65 536 SMILES strings). The previously mentioned
experiments revealed that exactly half of the improvement was a result of the ZINC data containing longer SMILES strings. The other half was a result of the structure of the ZINC data, which contains explicit hydrogen, yielding an increased number of shared LINGOs.

The presented tables do not include time to parse the SMILES files. Initial studies revealed that parsing files containing SMILES strings and converting them to the verbose representation takes less than 20% of the total running time on the Maybridge data, and less than 1% on the ZINC data. Converting this data to the inverted indices representation is included in the presented execution times, but accounts for less than 1% of the time spent in the algorithm.

9.7 Conclusion

The number of available molecules is ever increasing and new methods are needed to handle the large chemical databases. This paper presents a reduction from LINGO multisets to sparse fingerprints making it possible to implement methods for performing rapid queries in molecular databases with the LINGOsim similarity measure by using the Tanimoto coefficient in fingerprint databases.

This paper also presents the inverted indices method for storing LINGO multisets along with a method for rapidly calculating the similarity matrix for such a collection. The presented algorithm has been implemented and tested on standard hardware and was observed to be more efficient than other current methods, outperforming them in all tests. The SIML method tested against in this study was designed for a $q$ of four whereas the verbose representation is independent of $q$. The presented method makes it possible to analyse very large data sets without the need for GPUs or other types of specialised hardware. The tested implementation along with the test data is available at http://birc.au.dk/~tgk/ii.

There are two interesting directions for future research, namely statistical analysis of very large data sets using the inverted indices method and acceleration of queries into large data bases by using the reduction to sparse fingerprint presented in this paper.

9.8 Acknowledgements

The authors would like to thank Imran Haque for making the SIML implementation along with the SIML test data available and Open Eye for an academic license to the OEChem TK.
Chapter 10

SNPFile – A software library and file format for large scale association mapping and population genetics studies

The paper *SNPFile – A software library and file format for large scale association mapping and population genetics studies* presented in this chapter was published in a journal.


Except for typographical and formatting changes the content of this chapter is equal to [99]. The software library presented in this chapter is available at http://www.birc.dk/~mailund/SNPFile.
SNPFile – A software library and file format for large scale association mapping and population genetics studies

Jesper Nielsen∗†‡ Thomas Mailund†§

Abstract

High-throughput genotyping technology has enabled cost effective typing of thousands of individuals in hundred of thousands of markers for use in genome wide studies. This vast improvement in data acquisition technology makes it an informatics challenge to efficiently store and manipulate the data. While spreadsheets and flat text files were adequate solutions earlier, the increased data size mandates more efficient solutions.

We describe a new binary file format for SNP data, together with a software library for file manipulation. The file format stores genotype data together with any kind of additional data, using a flexible serialisation mechanism. The format is designed to be IO efficient for the access patterns of most multi-locus analysis methods.

The new file format has been very useful for our own studies where it has significantly reduced the informatics burden in keeping track of various secondary data, and where the memory and IO efficiency has greatly simplified analysis runs. A main limitation with the file format is that it is only supported by the very limited set of analysis tools developed in our own lab. This is somewhat alleviated by a scripting interfaces that makes it easy to write converters to and from the format.

10.1 Background

High-throughput genotyping technology has enabled cost effective typing of thousands of individuals in hundred of thousands of markers for use in genome wide studies [15], in particular genome disease association studies [11,12,48,61,117,123].

There are currently no standard file format for storing such genotype data, and most major analysis tools define their own textual input and output formats. Only a few tools supports several input formats, and often several conversion scripts needs to be implemented in a study. These file formats of analysis tools usually only represent a restricted set of the data collected for the study – only the data necessary for the computations provided by the program – so

∗Bioinformatics Research Center, University of Aarhus, Denmark
†Department of Computer Science, University of Aarhus, Denmark
‡E-mail: jn@daimi.au.dk.
§E-mail: mailund@birc.au.dk.
a study either needs a secondary format for storing all data, with converter programs for import/export to analysis tools, or need several files for storing various types of data.

While spreadsheets and plain text files were adequate, if not optimal, solutions earlier, the increased data size mandates more efficient solutions. While plain text files formats have the advantage that they are human readable and can be edited in any text editor to correct mistakes, they have two major disadvantages: i) they are less space efficient than binary formats, often significantly so, and ii) text formats need to be parsed by tools before the data is analysed, a time consuming task when dealing with massive data sets.

Here we describe a new binary file format, SNPFile, for storing SNP data and a software library for manipulating such files. The file format stores genotype data together with any kind of additional data, using a flexible serialisation mechanism. Data is memory mapped as needed so even very large data sets can be manipulated with moderate RAM requirements. The representation is optimised for accessing nearby markers together, and thus cache and disk efficient for the access patterns of most multi-locus analysis methods.

We have extended the suite of association mapping tools developed in our group [1], including both single marker methods [7] and multi locus methods [2, 5] and now successfully use it in our own studies.

10.2 Results and Discussion

We have developed a new file format and C++ library for manipulating SNP genotype data and arbitrary secondary data. The design allows us to store all genotype and secondary data in a single file, using a flexible serialisation framework. The genotype data representation is designed to be memory and IO efficient for the access patterns typical for multi-marker association mapping methods.

10.2.1 Simple and efficient genotype data manipulation

The primary data in a SNPFile is genotype data, represented as a matrix with one or two rows per individual (depending on whether the phase of the genotypes is know or unknown) and one column per marker. A matrix representation for the primary data is a simple abstraction that makes it relatively easy to implement most analyses.

The actual implementation consists of a small hierarchy of classes for representing the data, depending on the size of the data and the usage pattern. Small matrices can efficiently be stored in RAM while for larger matrices we provide file storage. The abstraction for accessing the data is the same whether the actual data is stored in RAM or on disk.

Although the programming abstraction for RAM based and file based matrices are the same, the time performance can differ significantly between accessing RAM and file data. By representing the file based matrices in a “column by column” order on the disk (see figure 10.1) we have optimised the code for
Figure 10.1: If your program only accesses a few columns at a time they will cluster nicely in virtual memory and it will be easy for the operating system to keep only the needed pages in physical memory. This means you can handle very big SNPFiles while not using very much actual memory. Furthermore, if your program only access columns ordered left-to-right and from top to bottom, the file will simply be accessed from the beginning to the end. This is what the entire computer, both hardware and software, is optimized for. Thus it should be very fast. If you read a row from the matrix, however, you will access a lot of pages in the file, only use a very small part of each and the operating system will waste a lot of time reading data that will not be used, since it operates on entire pages.

Table 10.1: Running time in seconds, for Blossoc using text IO and SNPFile, as a function as the number of individuals.

<table>
<thead>
<tr>
<th>No. Individuals</th>
<th>Text IO</th>
<th>SNPFile</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>1200</td>
<td>867</td>
</tr>
<tr>
<td>1000</td>
<td>1893</td>
<td>1671</td>
</tr>
</tbody>
</table>
Chapter 10. SNPFile

10.2.2 Framework for arbitrary secondary data

Depending on the analysis of the data, various secondary data is needed, such as phenotypes, co-variates etc. Most file formats support only a small fraction of the types of secondary data of interest in a study, since they only focus on the types of analysis intended when the file format was designed. A consequence is that data is often kept in several independent files, with ample risks for accidental inconsistencies between files.

To avoid such problems we have designed a flexible framework for secondary data into SNPFile. Through a serialisation framework, any C++ type can be stored in a SNPFile and accessed through a text key. Built-in types and STL containers are directly supported, and user-defined types can be supported by writing serialisation and de-serialisation methods. This can be done either through template methods in the user-defined types, or non-intrusively through global or name-space functions.

10.2.3 Using SNPFile

Most multi-marker analysis methods can efficiently represent the genotype data in matrix form, with one or two rows (columns) per individual and with a column (row) per marker. For such methods, implementing them using SNPFile is straightforward. With the framework for storing arbitrary C++ data types, porting applications to use SNPFile is usually a simple matter of changing the IO routines to read the relevant secondary data through this framework, and then using SNPFiles matrix classes instead of those used before.
10.3. Conclusions

We have ported our existing association mapping software \([1, 2, 5, 7]\) – both single marker and multi-marker methods – to work on the new file format. Since these tools already represented genotype data as matrices, porting them was a simple task, taking from a few hours to a day or two. We are currently successfully using the updated tools in our own studies, where the format has greatly alleviated the informatics problems in data management and completely eliminated the need for cutting data into windows for analysis, when the full data cannot fit in RAM.

10.3 Conclusions

The size of data that can cost efficiently be collected for population genetics studies – and especially disease mapping studies – has increased immensely the last few years, and this has lead to an informatics challenge in how to efficiently store and manipulate this data together with any secondary data collected for the study.

The file format we have described enables us to store all relevant data – primary and secondary – in a single file. The primary data is stored as a matrix, with a memory layout that makes it IO efficient to manipulate the data on disk, avoiding having to keep large data sets in RAM. The secondary data is stored using a flexible serialisation framework that allows any C++ data type to be stored together with the primary data.

The format has been very useful for our own studies where it has significantly reduced the informatics burden in keeping track of various secondary data, and where the memory and IO efficiency has greatly simplified analysis runs. A main limitation with the file format is that it is only supported by the very limited set of analysis tools developed in our own lab. Through scripting interfaces to the file format, we hope to alleviate this in the future.

A different binary file format for massive genotype data is available in the PLINK project \([16]\). The purpose of the binary format there is also achieving better CPU and memory performance. Where their format differs from ours is mainly in the treatment of secondary data. In the PLINK project, secondary data such as co-variates requires separate files from the genotype data. In contrast, we have designed our format such that we can store arbitrary secondary data together with the primary data in the same files.

10.4 Methods

A SNPFile stores primary data as a matrix as well as any kind of secondary data, e.g. individuals phenotypes, marker names and positions, ethnicity of individuals or co-variates for disease studies.

10.4.1 File manipulation

SNPFiles are accessed through a class of the same name. Instances of the class SNPFile represents a SNPFile on disk, and arguments to its constructor
determine read, write and creation semantics of the file. The constructor for
SNPFile looks like this:

```cpp
SNPFile(const std::string &filename,
    bool allowWriting = false,
    bool createFile = true,
    int mode = 00644);
```

where `filename` specifies the name of the file on disk, `allowWriting` determines if the file should be opened read-only or in read-write mode, `createFile` specifies if the file should be created on disk if it does not already exist, and `mode` specifies the access file permissions. A usage example could look like this:

```cpp
#include <snpfile/snpfile.hh>
using namespace BiRC::SNPFile;

int main()
{
    SNPFile readOnly("/dir1/file1.snp");
    SNPFile readWrite("/dir2/file2.snp", true);

    // Do computations with files
    readOnly.close();
    readWrite.close();

    return 0;
}
```

The read/write semantics of a SNPFile is carried over to the methods and
classes for accessing data, in the form of separate interfaces to mutable and
immutable matrices, providing a compile time check for correct access to the
data. The exception to this design is the SNPFile class itself: the access to
SNPFile objects is checked at runtime, with exceptions thrown in case of incor-
rect access. The reason for this is to permit write access permission to change
at runtime for interactive applications.

10.4.2 Accessing genotype data

The primary data in a SNPFile is genotype data. We represent this as a matrix
with each cell containing a genotype or allele. The matrix has one row for data
with unknown phase and two rows for data with known phase. The matrix has
one column per typed marker. The cells contain a `signed char` representing
the genotype, where by convention, we use -9 to indicate missing values, use 0
and 1 for homozygote genotypes and 2 for heterzygote genotypes. Other values
are reserved for future use.

The library contains a small hierarchy of matrix classes for representing
genotype data, together with two handler classes providing the matrix interface
to the data representation classes. The matrix data representation hierarchy,
rooted in the abstract class \texttt{MatrixData}, implements the memory management strategies, including allocation, deallocation and resizing. The handler classes, \texttt{ImmutableMatrix} and \texttt{Matrix}, provides the interface for accessing and, in the case of \texttt{Matrix}, modifying the matrices. Splitting the matrix classes in two responsibilities, data representation and data access, combines flexibility in representation with efficient data access. We get a flexible design for representing data both in RAM and on disk with no virtual function overhead when accessing the data.

\subsection*{Accessing matrices}

Access to matrix data is through one of the classes \texttt{ImmutableMatrix} and \texttt{Matrix}. \texttt{ImmutableMatrix} represents a read-only matrix, and \texttt{Matrix} represents a read-write matrix. The later is derived from the former, but allows entries in the matrix to be updated. A usage example, calculating the genotype frequencies for all markers in a matrix, is shown below:

```
#include <iostream>
#include <snpfile/matrix.hh>
using namespace BiRC::SNPFile;

void genotypeFrequencies(ImmutableMatrix &m)
{
    for (int j = 0; j < m.noCols(); ++j) {
        int counts[] = {0,0,0};
        int total = 0;
        for (int i = 0; i < m.noRows(); ++i) {
            if (m(i,j) < 0) continue; //missing
            if (m(i,j) > 2) continue; //error
            ++counts[m(i,j)];
            ++total;
        }
        if (total > 0) {
            std::cout << counts[0]/total << ' ';
            std::cout << counts[1]/total << ' ';
            std::cout << counts[2]/total << std::endl;
        } else {
            std::cout << "nan nan nan"
            << std::endl;
        }
    }
}
```

Both handler classes can only be instantiated when assigned a \texttt{MatrixData} instance. They do not represent the data but only provide interfaces to it.
Matrix representation

The MatrixData class is abstract and provides the bridge between data access (the ImmutableMatrix and Matrix classes) and memory management. The actual data management is implemented in sub-classes of MatrixData. The library provides two data representations, one for representing matrix data in RAM and one for representing data on disk (the later actually implemented as two different classes), but application programmers can provide their own as needed.

In our design we have considered the data representations implementation details, so the actual class representations cannot be accessed through the library interface. Instead, instances of the classes can be created through factory methods.

Small matrices, representing small windows of the data, are often used as part of a larger computation, and such matrices are most efficiently stored in RAM. The ArrayMatrixData class is provided for this. The factory method for creating instances of this class returns a Matrix handler since read-only RAM based matrices are of little use. This handler can, of course, always be cast to a ImmutableMatrix handler if a read-only interface is needed for later processing.

Matrices stored on disk are handled by the two classes: ReadOnlyFileMatrixData and ReadWriteFileMatrixData. Both are constructed with a reference to a SNPFile object. The factory method for ReadOnlyFileMatrixData returns a ImmutableMatrix instance while the factory method for ReadWriteFileMatrixData returns a Matrix. It is a runtime error to create a ReadWriteFileMatrixData object with a reference to a SNPFile object opened as read-only. Compile time checks ensure that the access patterns to matrices, after their instantiation, is correct.

An example of accessing a matrix on a read-only file, for calculating the genotype using the function defined above, is shown below:

```cpp
#include <iostream>
#include <snpfile/snpfile.hh>
#include <snpfile/matrix.hh>
#include <snpfile/file_matrix.hh>
using namespace BiRC::SNPFile;

void genotypeFrequencies(ImmutableMatrix &m)
{
    // see above for implementation
    ...
}

int main()
{
    SNPFile file("filename.snp");
    ImmutableMatrix m = newReadOnlyFileMatrix(file);

    genotypeFrequencies(m);
}
10.4. Methods

IO efficiency for large genotype data sets

One of the motivations for SNPFiles is efficient management of large datasets; frequently datasets too large to keep in the computer’s main memory. We achieve this by keeping file matrices (classes ReadOnlyFileMatrixData andReadWriteFileMatrixData) on disk, rather than loading them into RAM, and then use memory mapping to access the matrices. By representing the matrices on disk in a way that matches common usage we can rely on the operating system of the computer to make sure the right parts of the file are read into memory and flushed back to disk in an efficient way. This essentially means representing the matrices column-wise since most multi locus methods access neighbouring markers together but less frequently distant markers together.

Matrix views

For computations on sub-matrices, where the matrix data is not modified, it is inefficient to copy the data. It is often also inconvenient to design the methods to keep track of relevant indices for sub-matrices, especially with recursive methods that modify the views – e.g. split rows based on genotypes or sort columns with respect to marker position as in the Blossoc method [88].

The MatrixView class provides a solution to this problem by wrapping MatrixData objects and modifying matrix indices so a cell index in a view is redirected to the corresponding cell index in the matrix. This design allow transparent rearrangement of rows and columns, and extraction of arbitrary sub-matrices, with very little computational overhead.

10.4.3 Accessing secondary data (meta data)

SNPFiles can store arbitrary secondary data, or meta data, associated to the primary data. Meta data access is handled through three template methods:

```cpp
template<typename T>
void getMetadata(const MetadataAccessor &acc,
                 const std::string &key,
                 T &dest);

template<typename T>
T fetchMetadata(const MetadataAccessor &acc,
                const std::string &key);

template<typename T>
void setMetadata(MetadataAccessor &acc,
                 const std::string &key,
                 T &dest);
```
const T &src);

The first parameter for all tree functions is of type MetadataAccessor. A MetadataAccessor is a container capable of storing meta data, which in practise is usually a SNPFile object. The second parameter is a key used to identify the data. Since we can store arbitrary meta data, keys are used to identify the various data.

The functions are templates parameterized with the type of the meta data. In principle, any C++ type can be used as meta data, but the template functions needs to know how to serialise data of the type. For serialisation, we use a framework similar to the the Boost serialisation framework [4], but one that is binary compatible across different platforms and different versions of the C++ STL. The framework can immediately serialise all primitive types, such as int or double, and the most common STL types, such as map<> or vector<>.

Adding meta data to a SNPFile is done using setMetaData as below:

```cpp
#include <snpfile/metadata_access.hh>
#include <snpfile/snpfile.hh>

#include <iterator>
#include <fstream>

using namespace BiRC::SNPFile;
using namespace std;

namespace {
    // command line options...
    bool binaryPhenotypes;
    // ... more options ...
}

int main(int argc, char *argv[])
{
    // ... option parsing ...

    SNPFile file("someFile.snp", true);
    ifstream input("input.txt");

    if (binaryPhenotypes) {
        // read a sequence of binary phenotypes
        // into a boolean vector
        vector<bool> phenotypes;
        copy(istream_iterator<bool>(input),
             istream_iterator<bool>(),
             back_inserter(phenotypes));

        // store a flag indicating the
        // phenotypes are binary
```
setMetadata(file, "binary phenotypes?", true);

// store the phenotypes as well
setMetadata(file, "phenotypes", phenotypes);

} else {
// read a sequence of quantitative
// phenotypes into a double vector
vector<double> phenotypes;
copy(istream_iterator<double>(input),
     istream_iterator<double>(),
     back_inserter(phenotypes));

// store a flag indicating the
// phenotypes are *not* binary
setMetadata(file, "binary phenotypes?", false);

// store the phenotypes
setMetadata(file, "phenotypes", phenotypes);
}

file.close();

return 0;
}

where the example code reads some phenotype data from a text file – either
binary traits or continous traits depending on command line options – and
stores the data, together with a flag, in the SNPFile. Reading the data back
from a file is done through getMetadata or fetchMetaData:

#include <snpfile/metadata_access.hh>
#include <snpfile/snpfile.hh>
using namespace BiRC::SNPFile;
using namespace std;

int main()
{
SNPFile file("someFile.snp");

if (fetchMetadata<bool>(file, "binary phenotypes?")) {
    vector<bool> phenotypes;
    getMetadata(file, "phenotypes", phenotypes);

    // ... analyse data ...
}
else {
    vector<double> phenotypes;
}
getMetadata(file, "phenotypes", phenotypes);

    // ... analyse data ... 
}

file.close();

    return 0;
}

where fetchMetadata is just syntactic sugar around getMetaData so we can access data without necessarily declaring a variable for it (as in the if statement above). For complex data, such as maps, getMetaData is more efficient than fetchMetadata.

Serialising user-defined types

User defined classes cannot immediately be serialised, but it is possible to extend the serialisation framework with arbitrary types in two ways: by implementing member functions in the class or struct to be serialised, or by implementing free functions in the namespace of the class. The former can be used for classes the application programmer is free to modify, while the later can be used when that is not an option.

The simplest way to add serialisation through member functions it to add a template method named serialize that can be used for both serialisation and de-serialisation, depending on its template instantiation. Alternatively, an overloaded serialize function can be used to handle serialisation and de-serialisation differently. A class that implements serialisation with member functions can then be exported to the SNPFile serialisation framework using the macro BIRC_SNPF_FILE_INTRUSIVE_SERIALIZATION.

For non-intrusive serialisation, the framework can use a serialize template function in the namespace of the class to be serialised and the macro BIRC_SNPF_FILE_NONINTRUSIVE_SERIALIZATION instead. If different methods are needed for serialisation and de-serialisation, one can implement methods load and save in a specialised SerializationTrait template. For details on this, we refer to the library documentation.

The example below illustrates serialisation of user-defined types. The example shows how, in a hypothetical study where a SNPFile combines individuals from different previous studies and from different populations, we can store population and study information from the previous studies, and associate this to each individual. The example defines three new types: IndivData for data associated with each genotyped individual, StudyData for data associated with each previous study where the genotype data is obtained from, and PopulationData for data associated with populations. For IndivData we use the member function approach to serialisation and for the other two classes we use the free function version.

#include <snpfile/metadata_access.hh>
10.4. Methods

```cpp
#include <snpfile/snpfile.hh>
using namespace BiRC::SNPFile;

#include <string>
#include <vector>
#include <map>
using namespace std;

struct IndivData {
    string name;
    int studyID;
    int populationID;

    // member interface to
    // meta data serialisation
    template<class Archive>
    void serialize(Archive & ar) {
        ar | name;
        ar | studyID;
        ar | populationID;
    }
};

// macro needed to export the serialisation
// to the SNPFile framework
BIRC_SNPFILE_INTRUSIVE_SERIALIZATION(IndivData);

struct StudyData {
    string someRelevantData;
    string additionalData;
};

struct PopulationData {
    string popData;
};

// non-intrusive support for serialisation
template<class Archive>
void serialize(Archive & ar, StudyData &d) {
    ar | d.someRelevantData;
    ar | d.additionalData;
}

template<class Archive>
void serialize(Archive & ar, PopulationData &d)
```
Chapter 10. SNPFile

```c++
{
    ar |= d.popData;
}

// macros needed to export
// serialisation
BIRC_SNPFILE_NONINTRUSIVE_SERIALIZATION(StudyData);
BIRC_SNPFILE_NONINTRUSIVE_SERIALIZATION(PopulationData);

int main()
{
    // mapping from studyIDs to study data
    map<int,StudyData> studies;

    // mapping from populationIDs to population data
    map<int,PopulationData> populations;

    // data for each individual in the SNPFile, ordered
    // in the same order as the rows in the genotype
    // matrix
    vector<IndivData> individualsData;

    // ... fill in data for the maps and vector...

    SNPFile file("someFile.snp",true);
    setMetadata(file, "study data", studies);
    setMetadata(file, "population data", populations);
    setMetadata(file, "individuals data", individualsData);
    file.close();

    return 0;
}
```

After the serialisation method is specified in this way, `getMetaData`, `setMetadata` and `fetchMetadata` can be used as for any other type. For more information about serialisation of custom types, we refer to the library documentation.

### Meta-data type system

The serialisation mechanism for meta-data requires that the program accessing meta-data knows the type of the data before accessing it – a consequence of using a statically typed language such as C++. Unfortunately, this limits the general usability of the meta-data framework: tools operating on SNPFiles must all agree on the availability and type of meta-data to be able to manipulate it. This requires a protocol that application programs must follow if their programs...
should be able to operate on the same files.

Our initial design did rely on such a meta-data protocol, with agreed-upon types for the data used by our tool suite. Our experience with this convinced us, however, that this approach was less flexible than desired. This lead us to design a system for storing type information together with meta-data in SNPFiles, enabling us to dynamically extract meta-data information – availability and type of meta-data. With this design, programs can probe SNPFiles to get information about meta-data, and users can – when the tools support this – interactively access data to do their analyses.

The meta-data type system is non-intrusive in the sense that it does not affect the interface to storing meta-data described above. Any kind of meta-data can still be serialised into SNPFiles – using the functions above – and type information will automatically be stored together with the data whenever the type of the data is known by the SNPFile library (which includes primitive types and STL containers).

For custom meta-data – where the SNPFile library does not know the type – a mechanism similar to the serialisation framework allows the application program to provide type information to SNPFile. For example, to add support for the three custom types introduced in the example above, we would just add the following lines to our program:

```
BiRC_SNPFNFILE_EXPORT_TYPE(IndivData)
BiRC_SNPFNFILE_EXPORT_TYPE(StudyData)
BiRC_SNPFNFILE_EXPORT_TYPE(PopulationData)
```

The macros used for exporting types to the serialisation framework will by default add type support as well, however, so in our example this is not needed.

In our program above, the type associated with meta-data “individuals data” will then be `std::vector< IndivData >`, the type associated with “population data” will be `std::map< int32_t, PopulationData >` and the data associated with “study data” will be `std::map< int32_t, StudyData >`. Without specifying the type this way, the types would be stored as `std::vector< unknown >` and `std::map< int32_t, unknown >`, respectively.

### 10.5.4 Script access to SNPFiles

For easier manipulation of SNPFile files, we provide a Python extension module. Through this module, the genotype matrices can be manipulated in ways similar to the C++ interface. Most common meta data types can be serialized and manipulated, but due to type differences between Python and C++ there are some limitations in the Python interface, including manipulation of custom data types.

### 10.5 Availability and requirements

**Project name:** SNPFile  
**Project home page:** http://www.daimi.au.dk/~mailund/SNPFile/
Operating system(s): Binary distributions available for Linux. Source code available for all unix-like platforms.

Programming language: C++

Other requirements: The boost library [3].

License: GNU GPL version 2.

Any restrictions to use by non-academics: None, besides those of the GPL license.

10.6 Abbreviations


10.7 Competing interests

The authors declare that they have no competing interests.

10.8 Authors’ contributions

TM conceived of the project. JN and TM both designed the library, while JN did the majority of the implementation. Both authors drafted the manuscript.

10.9 Acknowledgements

TM is funded by the Danish Research Agency, FNU grant 272-05-0283 and FTP grant 274-05-0365 and a Wellcome Trust ‘Value in People Award’.
Chapter 11

RNA exosome depletion reveals transcription upstream of active human promoters

The paper RNA exosome depletion reveals transcription upstream of active human promoters presented in this chapter was published in Science.


Except for typographical and formatting changes the content of this chapter is equal to [104]. Supporting On-line Material is available at http://www.sciencemag.org/content/suppl/2008/12/04/1164096.DC1/Preker.SOM.pdf.
RNA exosome depletion reveals transcription upstream of active human promoters

Pascal Preker∗‡ Jesper Nielsen†‡ Susanne Kammler∗‡§
Søren Lykke-Andersen∗‡ Marianne S. Christensen∗‡
Christophe K. Mapendano∗‡ Mikkel H. Schierup†‡
Torben Heick Jensen∗‡¶

Abstract

Studies have shown that the bulk of eukaryotic genomes is transcribed. Transcriptome maps are frequently updated, but low-abundant transcripts have probably gone unnoticed. To eliminate RNA degradation, we depleted the exonucleolytic RNA exosome from human cells and then subjected the RNA to tiling microarray analysis. This revealed a class of short, polyadenylated and highly unstable RNAs. These promoter upstream transcripts (PROMPTs) are produced ∼0.5 to 2.5 kilobases upstream of active transcription start sites. PROMPT transcription occurs in both sense and antisense directions with respect to the downstream gene. In addition, it requires the presence of the gene promoter and is positively correlated with gene activity. We propose that PROMPT transcription is a common characteristic of RNA polymerase II (RNAPII) transcribed genes with a possible regulatory potential.

Recent high-throughput analyses have revealed that > 90% of all human DNA is transcribed [19]. The vast majority of these transcripts are noncoding, thus challenging the classical definition of what constitutes a gene and, by association, a promoter [56,72,115]. Furthermore, additional short-lived RNAs might have escaped detection. With the aim of identifying such transcripts, we used RNA interference in HeLa cells to deplete hRrp40, a core component of the human 3’ to 5’ exoribonucleolytic exosome, one of the major RNA degradation complexes (fig. S1A). This resulted in a severe processing defect of the known exosome substrate 5S ribosomal RNA (fig. S1B), demonstrating diminished exosome function. Oligo dT-primed, double-stranded cDNA from cells that had been treated with either a control [enhanced green fluorescent protein (eGFP)] or hRrp40 small interfering RNA (siRNA) was hybridized to an encyclopedia of DNA elements (ENCODE) tiling array, which covers a representative ∼1% of
Figure 11.1: PROMPTs are produced immediately upstream of annotated TSSs and are degraded by the RNA exosome. (A) Relative stabilization of RNA from hRrp40 knockdown over control cells, sorted according to annotated genomic features (http://genome.ucsc.edu/cgi-bin/hgTracks) and normalized to the total signal over the entire ENCODE region. (B) PROMPT signature of a 500-kb ENCODE region (ENr323), showing the log2 transformed hRrp40-siRNA/eGFP-siRNA signal ratio (blue track) below the location of annotated genes (red bars) with their orientation of transcription indicated by arrows. The bottom track shows hRrp40-siRNA/eGFP-siRNA signal peaks (see supporting online material). (C) RT-qPCR analysis of 10 representative PROMPT regions. HeLa cells were treated with eGFP siRNA (control) or the experimental samples hRrp40, hRrp6, hRrp44, or both hRrp6 and hRrp44, as indicated. Mean values with standard deviations from at least three experiments are shown as fold increase in RNA levels of experimental over control samples. All data were normalized to an internal control, glyceraldehyde phosphate dehydrogenase (GAPDH) mRNA. For numbering of PROMPTs, see table S4.

the human genome [19]. Comparison of array data to public gene annotations revealed overall stabilization of mRNAs (exons in Fig. 11.1A), as expected. RNA from intronic and intergenic regions were largely unaffected, with the exception of a 1.5-kb region immediately upstream of transcription start sites (TSSs) that was stabilized ~ 1.5-fold on average (Fig. 11.1A). The relative stabilization of RNA expressed from a 500-kb region exemplifies this: Four of the five genes in this region display peaks of stabilized RNA upstream of their annotated promoters (Fig. 11.1B).
To validate these results, we subjected RNA from exosome-depleted versus control cells to oligo dT-primed reverse transcription followed by quantitative polymerase chain reaction (RT-qPCR) analyses of a region upstream of 20 TSSs, all of which confirmed a statistically significant stabilization under hRrp40 knockdown conditions (Fig. 11.1C and fig. S2A). Depletion of an additional exosome component (hRrp46) resulted in similar levels of stabilization, whereas depletion of other factors involved in RNA turnover (hUpf1, hXrn1, hXrn2, hDcp2, PARN) had no effect (fig. S2B), indicating that promoter upstream transcripts (PROMPTs) are exosome-specific targets. Individual depletion of hRrp6 or hRrp44, the catalytically active exosome subunits, resulted in no or only modest stabilization. Depletion of both, however, caused levels of stabilization comparable to that observed upon depletion of hRrp40 (Fig. 11.1C and fig. S2A), suggesting that hRrp6 and hRrp44 act redundantly to degrade PROMPTs. This stabilization of PROMPTs in exosome-depleted cells is reminiscent of that of Saccharomyces cerevisiae cryptic unstable transcripts that, like PROMPTs, are also transcribed from nongenic regions [142].

To overview the average RNA stabilization profile around all 1594 annotated ENCODE TSSs, we aligned array data from the hRrp40 and control knockdown experiments, as well as the ratio of the two, relative to each other (Fig. 11.2A, top). Because of the different levels of stabilization of exonic and intronic RNA (Fig. 11.1A), we only considered data derived from exonic sequences downstream of the TSSs (fig. S3). Moreover, because many genes have multiple TSS clusters (i.e., promoters) that may confound analyses, we also aligned array data from 64 selected genes with only one major TSS cluster (low-complexity genes) (Fig. 11.2A, bottom, and table S1). Both alignments revealed an average RNA stabilization profile over a ∼ 2-kb region upstream of the TSS with a peak around −1 kb (Fig. 11.2A). In control cells, RNA levels are near background, whereas they are greatly elevated upon hRrp40 depletion. RNA levels in the hRrp40-depleted cells drop to background levels nearing the TSS, indicating that stabilized transcripts are distinct from their neighboring mRNAs. Thus, PROMPTs constitute a class of unstable transcripts, and we refer to the PROMPT-encoding DNA as the “PROMPT region.” Short RNAs produced around TSSs have previously been reported, most notably promoter-associated short RNAs, which were on average 0.5 kb on either side of the TSS [72]. These are, however, physically separate from PROMPTs by several hundred base pairs (fig. S4). In contrast, a few verified PROMPT regions show weak signs of transcriptional activity in other data sets, such as scattered cap analysis of gene expression tags (markers of transcription initiation events) [28] and expressed sequence tags unassigned to known genomic features (fig. S5).

We next examined whether PROMPTs were sense or antisense relative to the mRNA produced from the downstream positioned genes. Orientation-specific RT-qPCR performed on RNA from either hRrp40 depleted- or control cells demonstrated that, regardless of directional preference, both sense and antisense transcripts were detectable in PROMPT regions (Fig. 11.2B). In the presence of actinomycin D, which inhibits spurious synthesis of potential second-strand cDNA artifacts [103], this bidirectionality of PROMPTs was still observed (fig. S6). Moreover, both sense and antisense RNAs were stabi-
Figure 11.2: PROMPT expression maps to 0.5 to 2.5 kb (i) upstream of TSSs, (ii) can occur in both orientations, and (iii) requires the gene promoter. (A) Composite RNA profiles upstream of all 1594 (top) or 64 low-complexity (bottom) TSSs. Raw (single-channel) data (smoothened over a 10-bp window) from hRrp40-siRNA treated cells, control (eGFP) siRNA-treated cells, and their ratio are shown as indicated. The left $y$ axis denotes values for raw data, and the right $y$ axis denotes the log$_2$-transformed ratio of the raw data, scaled to center at zero. Positions in base pairs of RNA signals relative to TSSs are shown on the $x$ axes. (B) The sense (blue)/antisense (red) directionality of selected PROMPTs was determined by RT-qPCR with gene-specific primers (∼1 kb upstream the TSS) in either orientation in combination with a T20VN primer that hybridizes to the 3' poly(A) tail. Fold increases relative to the lowest value in control cells (set to 1) are plotted. PROMPTs are ordered such that the one with the highest preference for sense transcription is at the top. (C) Generation of promoter-upstream transcription in nonhuman DNA. Plasmids containing the $\beta$-globin gene under control of a viral promoter (CMV) or its ∆CMV control were transiently transfected into HeLa cells. Both constructs have an insertion of bacteriophage $\lambda$ DNA (red bar) upstream and a strong SV40 poly(A) site (black box) downstream of the $\beta$-globin gene. RNA levels were analyzed by RT-qPCR. Read-through transcription from the $\beta$-globin promoter was measured with the use of two amplicons upstream of the $\lambda$ DNA (“read through”). The “control” amplicon has no complementary sequence in the ∆CMV plasmid. Values on the $y$ axis are percentages of GAPDH mRNA levels. The dashed box in the linear plasmid representation (top, not drawn to scale) encloses the region that is deleted in the ∆CMV construct. Mean values with standard deviations ($n = 3$) are shown.
lized to a similar extent by hRrp40 depletion (Fig. 11.2B), demonstrating that both species are exosome substrates. When aligning array data to the TSSs of PROMPT regions where either sense or antisense RNA production predominates, they displayed patterns similar to the average PROMPT profile (fig. S7). Taken together, these data suggest a complex pattern of RNA polymerase II (RNAPII) activity in either orientation upstream of individual gene promoters. This observation was supported by nonexhaustive rapid amplification of cDNA ends (RACE) analyses of eight PROMPT regions, which often reveals multiple 5' and 3' ends (fig. S8).

To investigate the requirements for transcription upstream of promoters, we transiently transfected HeLa cells with a plasmid containing the β-globin gene under control of the strong cytomegalovirus promoter (pCMV) that is preceded by 2.2 kb of bacteriophage λ DNA (Fig. 11.2C). This resulted in transcript production from the λ DNA, demonstrating that PROMPT-like transcription can be initiated independent of the underlying DNA sequence. Transcripts arising from the λ DNA region cannot be read-through products from transcription around the plasmid because β-globin transcript levels reach background immediately downstream of the transcription termination site. Again, 5'-and 3'-RACE analyses were employed to map some transcription start- and end points, which substantiated the observation of dynamic and complex RNAPII activity in the region (fig. S9). Deletion of the CMV promoter resulted in the concomitant elimination of PROMPT and β-globin gene transcription (Fig. 11.2C and fig. S9). Thus, the generation of transcripts upstream of an active gene appears to depend on the gene promoter.

To further characterize the transcriptional activity and its origin in PROMPT regions, we compared PROMPT patterns to RNAPII occupancy, transcription factor binding, and chromatin modifications using public data sets generated by the ENCODE project (table S2). In two representative examples, the PROMPT region is covered by markers of active transcription, RNAPII and acetylated histone 3 (H3K9ac), whereas the transcription initiation factor TAF1 peaks at the TSS (Fig. 11.3A). The generality of this observation was examined by creating composite profiles of the 64 low-complexity regions encompassing PROMPT and TSS sequences. PROMPTs generally overlap with RNAPII, marks of active chromatin, and DNase hypersensitive sites [22,65], but not with peaks of transcription initiation factors; e.g., TAF1 or E2F1 [18,65] (Fig. 11.3B and fig. S10). Although this reinforces the concept of substantial transcription activity upstream of bona fide genes, the TSS-restricted localization of transcription initiation factors supports our conclusion using CMV/ΔCMV plasmids and argues against the presence of an independent PROMPT promoter.

A link between transcriptional activity in PROMPT and gene regions is further supported by scatter plots showing a strong positive correlation between total average RNAPII chromatin immunoprecipitation (ChIP) signal within the first 1.5 kb up- and downstream of all 1594 ENCODE TSSs (Fig. 11.4A). This relation is also evident from raw RNA expression data from the hRrp40 depletion experiment (Fig. 11.4B). With slopes of up to 0.7, these plots indicate that transcription activity in the PROMPT region is comparable to that in the beginning of the gene.
Figure 11.3: PROMPT regions are actively transcribed. (A) Details of transcript levels from this study compared with previously published ChIP-chip data for PROMPT and 5' regions of two representative genes. Genomic coordinates are shown on top in numbers of base pairs. (B) Composite profiles of RNA stabilization in the PROMPT regions of 64 low-complexity TSSs displayed as in Fig. 11.2A and compared with the indicated data sets.

Figure 11.4: Overall correlation of PROMPT- and gene-expression levels. (Left) Scatter plot of RNAPII distribution as measured by ChIP-chip over all 1594 TSSs in the ENCODE region (data taken from GEO, accession number GSE6391). Data were integrated over 1.5 kb before (y axis, “PROMPT”) and after (x axis, “Gene Start”) each TSS and plotted against each other. The slope of the linear regression is 0.68 with a P value of $\leq 10^{-300}$ (t test, product-moment correlation) and an $r^2$ value of 0.61 (degrees of freedom (df) = 1511). (Right) Scatter plot of single-channel RNA microarray signals from hRrp40 siRNA-treated cells created as above with the exception that, in the gene, only data corresponding to exonic DNA were used to remove exon/intron biases (fig. S3). Statistical values are slope = 0.45, P value < $10^{-137}$, and $r^2 = 0.39$ (df = 1420).

Given their ubiquitous nature, do PROMPTs have a function? A few non-coding RNAs that have been reported to exert regulatory functions are located in potential PROMPT regions [70, 134]. Likewise, a noncoding RNA directly upstream of the sphingosine-kinase1 (SPHK1) gene, which affects the methy-
Acknowledgements

We thank G. Pruijn for antibodies, D. Libri, A. Sandelin, and D. Schubeler for comments on the manuscript, and D. Riishøj and K. Jürgensen for technical assistance. This work was supported by the Danish National Research Foundation and the Danish Natural Science Research Council. Microarray data have been submitted to gene expression omnibus (GEO) at the National Center for Biotechnology Information (http://www.ncbi.nlm.nih.gov/) under accession number GSE12431.
Chapter 12

RNA polymerase plays both sides: Vivid and bidirectional transcription around and upstream of active promoters

The paper RNA polymerase plays both sides: Vivid and bidirectional transcription around and upstream of active promoters presented in this chapter was published in a journal.


Except for typographical and formatting changes the content of this chapter is equal to [105].
RNA polymerase plays both sides: Vivid and bidirectional transcription around and upstream of active promoters

Pascal Preker†‡ Jesper Nielsen†‡ Mikkel H. Schierup†‡
Torben Heick Jensen*‡§

The transcriptome is a term coined to describe the body of RNA transcripts derived from an organism’s genetic material. Recent technical advances have accelerated the cataloguing of transcriptomes of a wide range of species to reveal a common yet puzzling complexity (for reviews see refs. [91,135]). First the majority of genomic regions previously thought to be transcriptionally inert give rise to a variety of RNAs. Second, regions that were expected to be exclusively transcribed in one orientation overlap additional transcripts that are often produced from the opposite strand of the DNA. While specifics of the function and ultimate fate of most of these newly discovered RNAs remain largely unknown, one underlying pattern is emerging from a set of recent publications: an abundance of short transcripts around, or at a limited distance upstream of, the transcription start sites (TSSs) of known genes [40,52,64,96,104,119,143]. In the following, we will discuss the commonalities, differences and implications of these new and surprising findings.

Of the five publications that studied the transcriptomes of different human cell types, four involved high throughput sequencing strategies that allowed the authors to determine the orientation of the transcripts [40,52,64,119]. The fifth study used hybridization to tiling microarrays with the added twist of stabilizing RNAs that would otherwise go unnoticed by incapacitating the major eukaryotic 3'-5' RNA degradation machinery: the RNA exosome [104]. Finally, two publications used a related approach in the yeast Saccharomyces cerevisiae [96,143].

The studies from the Lis and Sharp laboratories [40,119] were conducted on different human cell lines and employed very different experimental approaches, yet came to surprisingly similar results (Fig. 1A, top). While the Sharp laboratory subjected short RNAs to massive parallel sequencing, the Lis laboratory devised a method to catch RNA polymerase II (RNAPII) in the act and then analyzed its associated nascent transcripts. The latter approach yielded in addition to abundance and orientation of transcripts also information on the

*Centre for mRNP Biogenesis and Metabolism, Department of Molecular Biology, C.F. Møllers Alle, bldg. 1130
†Bioinformatics Research Center, C.F. Møllers Alle, bldg. 1110
‡Aarhus University, Denmark
§Corresponding author: thj@mb.au.dk.
density of active polymerases on the DNA template. In both studies a sharp RNA peak was found an average 50 nt downstream of the TSSs of genes. This could be explained by earlier findings that the transcription machinery “stalls” at the transition from a slow initiation- to a faster elongation-phase, or might even abort transcription at this position (e.g. refs. [62, 94]). More surprisingly, however, was the finding that upstream of the TSS of nearly every active gene an antisense RNA peak trailed away in the opposite direction (Fig. 1A, top). Similar observations had been made two years earlier by the Gingeras laboratory [72], albeit at a lesser resolution due to the use of microarrays. Now the same group has confirmed and further expanded these results [52], again in good agreement with the Lis and Sharp laboratories. Using a cleverly designed method to assay the directionality of transcription on a genome wide scale, the group around Kinzler came to the same conclusion, namely that there is a
concentration of antisense tags upstream of many TSSs (Fig. 1B) [64]. For simplicity we will refer to these transcripts collectively as “divergent transcripts”.

Finally, the fifth paper investigated stabilization of RNA following siRNA-mediated knock-down of the RNA exosome within ~1% of the human genome by tiling microarrays [104]. Surprisingly, this revealed a major stabilization of transcripts expressed from a broad region upstream of the majority of active genes, but further away from the TSS than the peaks of antisense RNAs observed in the other studies (Fig. 1A, bottom red graph). Moreover, manual inspection of several of these so-called PROMoter uPstream Transcripts (PROMPTs) revealed that they generally derive from both strands of the DNA template. Despite their physical separation, PROMPTs share many similarities with the divergent transcripts: i) they are covered by RNAPII and other markers of transcription initiation [52, 104, 119]; in contrast, dimethylation of lysine 79 on H3, an elongation marker, is exclusively present on the gene [119]. ii) their peak strength correlates well with the activity of the downstream gene [40, 52, 104, 119], and both are especially prominent at promoters containing CpG islands [40, 104, 119], iii) their ends are ill defined: while at least a fraction of PROMPTs ends in a poly- or at least an oligo- (A) tail, it has not formally been shown whether PROMPTs and/or divergent transcripts possess canonical 5’ cap structures or are generated by processing and then receive a “5’ modification analogous to a cap structure” as suggested by ref. [52] iv) they are of low abundance and high heterogeneity under normal conditions, and their mechanism(s) of termination or the sequences that guide them have so far resisted scrutiny (see below).

These observations are not unique to humans: in S. cerevisiae, Cryptic Unstable Transcripts (CUTs) are disperse, short and short-lived transcripts that were at the time of their discovery equally elusive [142]. In a recent issue of Nature two laboratories present a comprehensive catalog of yeast CUTs, again using exosome-deplete conditions (specifically a deletion of the nuclear Rrp6p component) as a tool to stabilize them [96, 143]. By using tiling microarrays and high-throughput sequencing, respectively, the Steinmetz and Jacquier groups show that CUTs are heterogeneous in size but rarely exceed 300 nt, and that they originate bidirectionally from nucleosome free regions (nfrs) found in intergenic spaces, most notably in close vicinity to the end of genes. In fact, the distribution of sequence tags over the TSS is strikingly similar to that seen in mammals (compare Fig. 1C and 1A, upper panel).

Are divergent transcripts and PROMPTs related? One possibility is that they are intimately involved in the same function(s). Alternatively, the divergent transcripts could be degradation intermediates or end products of exosome activity acting on longer antisense transcripts. Finally, while divergent transcripts and CUTs all argue for the general bidirectionality of promoters, PROMPTs could be the manifestation of a different property of promoters, possibly that they border “loose” chromatin that readily gives access to RNAPII. Interestingly, loose interactions between histones and DNA may be caused by negative supercoiling created by RNAPII advancing in the sense direction [119]. Perhaps this would suffice to allow the enzyme to initiate without directional preference and without the need for a bona fide promoter even at distances as
Chapter 12. RNA polymerase plays both sides

far upstream the TSS as the PROMPT region. In contrast, in the case of divergent transcripts and CUTs, TATA-binding protein (TBP) and other general transcription factors might simply be insufficient to direct RNAPII into the “proper” direction.

Another open question is whether degradation of PROMPTs by the exosome is a consequence or the cause of a failure of the transcription machinery to transition into productive elongation. In other words, could the exosome be part of a mechanism that prevents these transcripts to proceed into the “wrong” direction? Such a role would be akin to the quality control mechanism that occurs near the 3’ end of faulty gene transcripts (e.g. ref. [112]). It also remains unknown how divergent transcripts are terminated. The answers might partially come from yeast where a set of factors involved in transcription termination and degradation of CUTs are known. In addition to Rrp6p, these are the Nrd1p/Nab3p/Sen1p and the so-called TRAMP complexes (for review, see ref. [66]). Recently, it was shown that the phosphorylation status of the carboxy-terminal domain (CTD) of the largest subunit of RNAPII is also important for transcription termination of CUTs [60]. These findings will undoubtedly inspire experiments in human cells.

What, if any, might the biological role of this lavish transcriptional activity be? It might provide a pool of active RNAPII that could be rapidly recruited to the gene proper. Another not mutually exclusive possibility is that transcription upstream of promoters facilitates gene expression by either recruiting chromatin remodeling factors and/or transcription factors through direct interaction or as an indirect result of the act of transcription itself. Aside from these more general roles, PROMPTs might exert specific regulatory function on certain genes. Examples of regulatory RNAs produced in the vicinity of promoters are surfacing in the literature (e.g. ref. [134]), but no common theme has emerged. In one of the new publications the authors show that expression of Promoter-Associated Small RNAs (PASRs) in trans can lead to mild down-regulation of c-MYC gene expression [52], while we found that an increase in the steady-state level of PROMPTs correlates with an increase in the degree of CpG methylation in the cognate promoter [104]. Thus like many intronic sequences some PROMPTs might have been co-opted in different ways to increase genetic flexibility. An extreme of this idea would be that sense PROMPTs might eventually give rise to new 5’ exons as suggested by occasional ESTs that span from the PROMPT region into the downstream gene. Indeed, every new such discovery challenges the early geno-centric view anew.

12.1 Acknowledgements

Research in the authors laboratory is supported by the Danish National Research Foundation, the Danish Natural Science Research Council, the Danish Cancer Society, the Novo Nordic foundation and the European Science Foundation (RNA Quality program). Ebbe Sloth Andersen, Aarhus University, is thanked for suggesting the acronym “PROMPT”.
Chapter 13

Algorithms for a parallel implementation of Hidden Markov Models with a small state space

The paper Algorithms for a parallel implementation of Hidden Markov Models with a small state space presented in this chapter was published at a conference.


Except for typographical and formatting changes the content of this chapter is equal to the journal paper [100]. An implementation of the algorithms presented in this chapter is available at http://www.birc.au.dk/~asand/parredhmmlib.
Algorithms for a parallel implementation of Hidden Markov Models with a small state space

Jesper Nielsen∗† Andreas Sand ∗‡

Abstract

Two of the most important algorithms for Hidden Markov Models are the forward and the Viterbi algorithms. We show how formulating these using linear algebra naturally lends itself to parallelization. Although the obtained algorithms are slow for Hidden Markov Models with large state spaces, they require very little communication between processors, and are fast in practice on models with a small state space.

We have tested our implementation against two other implementations on artificial data and observe a speed-up of roughly a factor of 5 for the forward algorithm and more than 6 for the Viterbi algorithm. We also tested our algorithm in the Coalescent Hidden Markov Model framework, where it gave a significant speed-up.

13.1 Introduction

Hidden Markov models (HMMs) are a class of statistical models for sequential data with an underlying hidden structure. They were first introduced to the field of bioinformatics in the early 1990s [107], and have since then been used in a wide variety of applications - for example gene annotation [49, 87], protein structure modeling [81], sequence alignment [13, 50] and phylogenetic analysis [47, 121]. Because of their computational efficiency, HMMs are one of the few widely used statistical methodologies that are feasible for genome wide analysis, where sequence lengths are in the millions or billions of characters. With data sets of this size, however, analysis time is still often measured in days or weeks. Improving on the performance of HMM analysis is therefore important to keep up with the quickly growing amount of biological sequence data to be analyzed.

In previous work [114] we have parallelized algorithms for HMM analysis to increase performance, by distributing the computations for each state among the available processors. This works well if the number of states is large, but for HMMs with a small number of states, the synchronization overhead makes this approach inefficient.

∗Bioinformatics Research Centre, Aarhus University, Aarhus, Denmark
†E-mail: jn@birc.au.dk.
‡E-mail: asand@birc.au.dk.
In this paper we present an alternative formulation of the forward algorithm and the Viterbi algorithm, parallelizing the workload across the observed sequence, instead of across the state space. This makes it feasible to give each processor a greater chunk of work and reduces communication overhead between the processors to a minimum. We thereby get a very efficient parallelization for HMMs with a small number of states. The algorithms have been implemented in a C++ library, parredhmmlib, that is freely available at http://www.birc.au.dk/~asand/parredhmmlib. Our implementation has been tested on artificial data, and in the Coalescent Hidden Markov Model (CoalHMM) framework [47].

13.2 Methods

An HMM is a probability distribution over a sequence \( O = O_1O_2\ldots O_T \in V^* \), where \( V = \{V_1,V_2,\ldots,V_M\} \) is an alphabet. We can formally define an HMM as consisting of [107]:

- A finite set of (hidden) states \( S = \{S_1,S_2,\ldots,S_N\} \). At any time \( t \), the HMM will be in any of these states, \( q_t = S_i \).
- A vector \( \pi = (\pi_1,\pi_2,\ldots,\pi_N) \) of initial state probabilities, in which \( \pi_i = P(q_1 = S_i) \) is the probability of the model initially being in state \( i \).
- A matrix \( A = \{a_{ij}\}_{i,j=1,2,\ldots,N} \) of transition probabilities, in which \( a_{ij} = P(q_t = S_j | q_{t-1} = S_i) \) is the probability of the transition from state \( S_i \) to state \( S_j \).
- A matrix \( B = \{b_i(j)\}_{i=1,2,\ldots,N} \), where \( b_i(j) = P(O_t = V_j | q_t = S_i) \) is the probability of state \( S_i \) emitting alphabet symbol \( V_j \).

Now using this definition, a data sequence \( O \) of length \( T \) can be generated from an HMM by performing the following procedure:

1. Set \( t := 1 \);

2. Sample the initial state \( q_1 \) according to the probability distribution \( \pi \);

3. Sample the alphabet symbol \( O_t \) from the emission probability distribution \( b_{q_t}(\cdot) \);

4. Set \( t := t + 1 \);

5. If \( t \leq T \) then sample the next state \( q_t \) from the probability distribution \( a_{q_{t-1}} \), and repeat from step 3; otherwise terminate.

An HMM is parameterized by \( \pi, A \) and \( B \), which we will denote by \( \lambda = (\pi,A,B) \).
13.2. Methods

Figure 13.1: The information flow in the forward algorithm. $\alpha_t$ is computed only from $\alpha_{t-1}$ and $C_t$.

### 13.2.1 The parredForward algorithm

One of the traditional algorithms for Hidden Markov Models is the forward algorithm. The forward algorithm computes the likelihood of seeing our data, given our model, $P(O | \lambda)$. First define

$$\alpha_t(i) = P(O_1, O_2, ..., O_t, q_t = S_i | \lambda).$$

If we can compute these $\alpha_t$s efficiently, we can compute $P(O | \lambda) = \sum_i \alpha_T(i)$. Let $\alpha_t$ be the vector of the $\alpha_t(i)$s

$$\alpha_t = \begin{bmatrix} \alpha_t(1) \\ \alpha_t(2) \\ \vdots \\ \alpha_t(N) \end{bmatrix},$$

$B_t$ be a diagonal matrix of the emission probabilities at time $t$

$$B_t = \begin{bmatrix} b_t(O_1) & & \\ & b_t(O_2) & \\ & & \vdots \\ & & b_t(O_N) \end{bmatrix},$$

and

$$C_t = \begin{cases} B_1 \pi & \text{if } t = 1 \\ B_t A^T & \text{otherwise.} \end{cases}$$

We can compute $\alpha_t$ using only $C_t$ and the previous $\alpha_{t-1}$

$$\alpha_t = C_t \alpha_{t-1} = C_tC_{t-1} \cdots C_2C_1,$$

as shown in Figure 13.1. Now, the classical implementation of the forward algorithm computes $\alpha_T$ as

$$(C_T(C_{T-1} \cdots (C_4(C_3(C_2C_1))) \cdots)).$$

Figure 13.2 shows how the $\alpha_t$ vectors are computed one at a time along the sequence, each one being derived from $C_t$ and $\alpha_{t-1}$. If more than one processor is available, one can attempt to parallelize the computation by computing the entries of $\alpha_t$ in parallel, which is the approach taken in HMMlib [114]. However
Chapter 13. A parallel implementation of Hidden Markov Models

Figure 13.2: The traditional forward algorithm, as described by Rabiner [107]. The rectangles represent matrices and vectors. The black lines denote dependencies. The top row is the $C_i$ matrices.

Figure 13.3: Using parallel reduction on the forward algorithm. The rectangles represent matrices and vectors. The black lines denote dependencies, while the horizontal dotted ones denote synchronization. The top row is the $C_i$ matrices.

The processors have to synchronize after each $\alpha_i$ is computed, and if $N$ is small the computation of $\alpha_i$ contains very little work, thus the time spent on synchronization will dominate. We propose to compute the matrix product using parallel reduction. The idea of reduction is to take advantage of the fact that matrix multiplication is associative, thus the terms can be grouped arbitrarily. For example the terms could be grouped into a binary tree

\[(\cdots(C_TC_{T-1})\cdots((C_4C_3)(C_2C_1))\cdots)\].

Figure 13.3 illustrates how the final $\alpha_T$ can be computed by parallel reduction.

Note that not all the $\alpha_i$s are computed, most being replaced by matrices, and that each matrix multiplication requires a synchronization to wait for its source data.

The traditional algorithm requires $T$ matrix-vector multiplications, giving a workload of $O(N^2T)$. In contrast the above algorithm makes use of matrix-matrix multiplications which are somewhat slower. If we assume, for simplicity, that we use the naive $O(N^3)$-time matrix multiplication the workload becomes $O(N^3T)$, thus it will have more actual work to do, but will be able to do a lot of it in parallel, and may actually be faster. If we assume we have one processor dedicated to each matrix multiplication, we get an execution time of $O(N^3 \log T)$ on our new algorithm, which is better than the traditional $O(N^2T)$ for small $N$ and large $T$. 
13.2. Methods

Numerical stability

All our matrices contain probabilities, which are between 0 and 1. This means that our products will tend toward zero exponentially fast. Normally the values will be stored in an IEEE 754 floating-point format. These formats have a limited precision, and if the above was implemented naively the results would quickly underflow and be rounded to zero.

If we can make do with log(P(O | λ)) instead of P(O | λ), we can prevent this underflow by continuously rescaling our matrices, like the way the columns are rescaled in the traditional forward algorithm [107]. We introduce a scaling constant $c_i$ for every matrix multiplication, setting it to the sum of all entries in the resulting matrix. Each $c_i$ is used two times: First we divide each entry in the resulting matrix by it, to keep the values from underflowing, and next we use it to restore the correct result at the end of our computations.

Assume we have $l$ matrix multiplications and $α_C$ is the resulting matrix, scaled to sum to one. Then

$$α_T = \left( \prod_{k=1}^{l} c_k \right) α_C,$$

and we can compute the final likelihood as

$$P(O | λ) = \sum_i α_T(i)$$

$$= \sum_i \left( \prod_{k=1}^{l} c_k \right) α_C(i)$$

$$= \left( \prod_{k=1}^{l} c_k \right) \sum_i α_C(i)$$

$$= \prod_{k=1}^{l} c_k,$$

since $α_C$ sums to one. Taking the logarithm of this, we get

$$\log(P(O | λ)) = \log \left( \prod_{k=1}^{l} c_k \right) = \sum_{k=1}^{l} \log (c_k).$$

Practical implementation

The above assumption that we have one processor dedicated to each multiplication is generally not true. In our implementation we assume the number of processors in a machine to be on the order of 10, and the sequence length $T$ to be on the order of $10^6$ or greater. We have made a number of changes to the above algorithm to make it simple and fast on a real computer.

To distribute the workload over the available processors we split it into a number of blocks. Each block will be processed completely independently and the result of that computation is a matrix representing that block. When all
Chapter 13. A parallel implementation of Hidden Markov Models

Figure 13.4: Our parredForward implementation, as run on a 4-way parallel system. The rectangles represent matrices and vectors. The black lines denote dependencies, the horizontal dotted ones denote synchronization, and the vertical dotted ones show blocks. For each block \( p \) identifies the processor computing it. The top row is the \( C_i \) matrices.

blocks have been processed one processor multiplies the result matrices to get the final result \( \alpha_T \).

Notice that \( C_1 \) is a vector, and not a matrix. The result of multiplying this with another \( C_1 \) is another vector. This is important because matrix-vector multiplication is faster than matrix-matrix multiplication, which means that the first block, containing \( C_1 \), will be processed faster than the others, and that the result is a vector. Once we know the resulting vector from the first block, we can use that to also compute the second block quickly, and so on and so forth. We use this observation by having one processor processing the blocks from the beginning and continuously using the results between them, while the remaining processors consume the blocks from the other end, to retain as much work as possible for the fast algorithm. Notice that our algorithm reduces to the traditional forward algorithm on these first blocks. Figure 13.4 shows our implementation: In this case the sequence has been split into 11 blocks. The first processor \( p = 0 \) simply runs the traditional forward algorithm on the blocks, starting from the right, while the remaining processors \( p = 1, 2, 3 \), in parallel, consumes three blocks at a time from the left. After all the blocks have been processed the threads are joined, and \( \alpha_T \) is found by multiplying the final vector from \( p = 0 \) with the resulting matrices from the other processors.

The algorithm the processor \( p = 0 \) executes has an asymptotic running time of \( \mathcal{O}(N^2T_1) \), while the algorithm executed by the remaining processors has a running time of \( \mathcal{O}(N^3T_2) \). If we assume that the difference in running time is exactly a factor of \( N \), we expect \( T_1 = \frac{TN}{N+P-1} \) and \( T_2 = \frac{T}{N+P-1} \). This gives us an asymptotic running time for our algorithm of \( \mathcal{O}\left(\frac{T^3}{N+P-1}\right) \), which is a factor \( 1 + \frac{P-1}{N} \) better than the traditional algorithm.

There can be only \( M \) different \( C_i \) matrices, besides \( C_1 \). We assume \( M \) is small compared to \( T \), and precompute all the different possible \( C_i \) matrices.

### 13.2.2 The parredViterbi algorithm

Another classical algorithm, as important as the forward algorithm is the Viterbi algorithm. The Viterbi algorithm finds the most likely state sequence
13.2. Methods

Figure 13.5: The information flow in the viterbi algorithm. $\delta_t$ is computed from $\delta_{t-1}$ and $C_t$ similarly to the forward algorithm. When backtracking $q_t$ depends on $q_{t+1}$ and the data that gave rise to that: $C_{t+1}$ and $\delta_t$.

$Q = q_1, q_2, ..., q_T$ given the observed data. This time define

$$\delta_t(i) = \max_{q_1, q_2, ..., q_{t-1}} \mathbb{P}(q_1, q_2, ..., q_t = i, O_1, O_2, ..., O_t | \lambda).$$

Remember that matrix multiplication is defined as:

$$(P \times Q)_{ij} = \sum_k P_{ik} Q_{kj}.$$

Similarly we define

$$(P \times^m Q)_{ij} = \max_k \{P_{ik} Q_{kj}\}.$$

Note that this new operator is associative. We can now compute

$$\delta_t = C_t \times^m \delta_{t-1} = C_t \times^m C_{t-1} \times^m ... \times^m C_2 \times^m C_1.$$

The entry in $\delta_T$ containing the maximal value will correspond to the final state $q_T$ in $Q$, and the value of the entry will be the likelihood of $Q$. The rest of $Q$ can be found by backtracking as sketched in Figure 13.5.
Chapter 13. A parallel implementation of Hidden Markov Models

Traditionally the $\delta$s and $Q$ would be computed linearly, but we can reduce it in parallel in exactly the same way as $\alpha_T$ was in the forward algorithm. Figure 13.6 illustrates how the $\delta_t$ vectors are traditionally computed exactly like the $\alpha_t$ vectors in the forward algorithm, and how $Q$ also is found by a simple scan. Before we show the above formally, we will define some more notation:

$$D_{k:l} = C_l \times^m C_{l-1} \times^m \ldots \times^m C_{k+1} \times^m C_k,$$

for $1 \leq k \leq l \leq T$, and note that

$$\delta_t = D_{1:t} \quad \text{and} \quad D_{m+1:t} \times^m D_{k:m} = D_{k:l}.$$

Assume we have found some $\delta_t$ and its corresponding $q_t$. This is enough information to find all $q_{t-1}, \ldots, q_1$. $\delta_t$ must have been computed by some computation

$$\delta_t = D_{1:t} = D_{k+1:t} \times^m D_{1:k} = D_{k+1:t} \times^m \delta_k,$$

for some $k$ and this allows us to find $q_k$ as the entry in $\delta_k$ that would give rise to $q_t$,

$$q_k = \operatorname{argmax}_j \{ (D_{k+1:t})_{q,t} \delta_k(j) \}.$$

The values $q_{k-1}, \ldots, q_1$ can be found by recursion. For the values $q_{t-1}, \ldots, q_{k+1}$ note that $D_{k+1:t}$ must also be some product $D_{l+1:t} \times^m D_{k+1:l}$. Thus

$$\delta_t = D_{k+1:t} \times^m \delta_k = D_{l+1:t} \times^m D_{k+1:l} \times^m \delta_k = D_{l+1:t} \times^m \delta_l.$$

Using the above $q_t$ can be found, and the entire range $q_{t-1}, \ldots, q_{k+1}$ can be found by recursion. Since we started out showing how to find $q_T$ we can find all of $Q$. 

Figure 13.7: The traditional viterbi algorithm, as described by Rabiner [107]. The rectangles represent matrices and vectors, and the circles the $q_t$ states. The black lines denote dependencies. The top row is the $C_t$ matrices – to minimize clutter most dependencies on these are left out.
13.3. Results

Practical implementation

Note that any $D_{1;i}$ will be a vector, while $D_{i;j}$ is a matrix, that takes up $N$ times as much space, for $i > 1$. For a large $T$ this could use a lot of memory. To conserve memory we store only one $D_{1:t_1}, D_{t_1+1:t_2}, \ldots, D_{t_n+1:T}$, from each block. From these we compute $D_{1:t_1}, D_{1:t_2}, \ldots, D_{1:T}$, and between these we fill out linearly, such that the majority of the matrices we store are of the form $D_{1;i}$ and we do not use significantly more memory than a traditional implementation.

Figure 13.7 depicts our implementation of parredViterbi: The upper half of the figure is similar to the figure for parredForward (Figure 13.4), and indeed $\delta_T$ is found like $\alpha_T$ was. $q_T$ can be found directly from $\delta_T$, and the last state in each preceding block can be found by backtracking through the vectors just computed and the result matrices that gave rise to them. Once these states are found we start another parallel phase: $p = 0$ simply backtracks through the blocks it processed in the first phase, and $p = 1, 2, 3$ executes the traditional Viterbi algorithm on each of the remaining blocks. For $p = 1, 2, 3$ the initial vector for the Viterbi algorithm is based on the result vector $\delta_t$ from the preceding block, and the backtracking is started from the state found in the single threaded phase.

Also note that since we only do multiplication and maximum of scalars, and no addition, numerical stability is much easier to handle – simply do all computations directly in logarithmic space.

13.3 Results

We have implemented the above algorithms in the \textit{parredhmmlib} package. The package is written in C++ and Python bindings are provided. The package is available from \url{http://www.birc.au.dk/~asand/parredhmmlib}. We have compared our implementation to GHMM [118], and our previous work HMMlib [114]. HMMlib is an implementation that takes advantage of all the features of a modern computer, such as SIMD instruction and multiple cores. The individual features of HMMlib can be turned on or off by the user, and we recommend only enabling these features for HMMs with large state spaces. Comparison with HMMlib is especially interesting because HMMlib also use parallelization, but does so at the level of each time-step instead of across them as presented in this paper. GHMM is a straight-forward, but general implementation of the classical HMM algorithms [107], that does not use any kind of parallelization. Our experiments were run on a Mac-Pro with two Intel quad-core Xeon processors (256kB L2 cache, 8MB L3 cache) running at 2.26GHz and 8GB main memory. In total there are eight hyper-threaded cores allowing us to execute up to 16 processes in parallel.

We did three experiments on artificially generated HMMs. All the plotted points are averages over 10 runs. In the first experiment we varied the state space of the HMM, while keeping the sequence length $T$ constant at 366210 and the alphabet size $M$ at 10. We set HMMlib up for running on HMMs with large state spaces to demonstrate what happens when that kind of parallelization is applied to HMMs with small state spaces. The relative short sequence length
A) Time per transition $\frac{\text{time}}{N^2}$.

B) Number of threads versus time.

C) Sequence length versus time.

Figure 13.8: Results from our experiments. In A) we tested running time versus state space size, $M = 10, T = 366210$, parredhmmlib use 16 threads, HMMlib use 8 threads, with SIMD enabled and GHMM is inherently single threaded. B) shows results from our experiment testing running time as a function of number of threads used, with $N = 4, M = 10, T = 10000000$ and HMMlib using 1 thread with SIMD disabled. Finally C) shows results from an experiment testing running time as a function of sequence length, using $N = 4, M = 10$, parredhmmlib with 16 threads, HMMlib and GHMM both running single threaded and HMMlib having SIMD disabled.

$T$ was chosen because HMMlib stores the entire forward table in memory, and as $N$ grows this can take up a very large amount of memory. As explained
earlier we expect parredhmmlib to be fast for small states spaces, and HMMlib to be fast for large state spaces. GHMM is not optimized for neither small nor large state spaces, and thus we do not expect it to be able to compete with parredhmmlib or HMMlib in those cases. Figure 13.8A shows our experimental results, in which the algorithms to a large extend behave as expected. The slowdown of parredhmmlib around $2^8$ states may be because that is the point where the 8MB of L3 cache of the processor is exceeded.

We have also tested how the number of threads influence the running time of our algorithms, shown in Figure 13.8B. We ran the parredForward and parredViterbi on HMMs with 4 states and an alphabet of size 10 and sequences of length $10^7$, varying the number of threads from one to 128. To compare our algorithms to HMMlib, we set HMMlib up to run as fast as possible on the models with small state spaces that we use. That is SIMD optimizations and parallelization was turned off. Curiously we see that our implementations are actually fastest even for one thread. This is probably because we precomputed the $C_i$ matrices, while HMMlib and GHMM do not. We also note that both parredForward and parredViterbi run fastest with 16 threads, which again is what we expected. Using all eight cores of the machine the parredForward gains a speed-up of a factor of 2.9, and using hyper-threading and running on 16 cores a speed-up of a factor of 3.4. For parredViterbi the numbers are 3.1 and 3.2 respectively.

Finally we have tested how the sequence length $L$ affects the execution time. As above we have set the number of states $N$ to four and the alphabet size $M$ to 10. The running time is expected to be linear in the sequence length, which also is what we see in Figure 13.8C. For the forward algorithm we are 5.5 times faster than HMMlib and 4.6 times faster than GHMM. For the Viterbi algorithm those numbers are 6.4 and 6.8 respectively.

We have also merged our method into the CoalHMM framework, where the existing implementation is based on HMMlib. CoalHMM is a framework that uses an HMM parameterized by coalescent theory, to infer changing genealogy along an alignment of DNA sequences [47]. The hidden states represent different genealogies, and the probability of change in genealogy, between two neighboring loci, is computed based on the probability of coalescence and recombination events. The observations are the columns of the alignment. We have benchmarked our method by running the model on an alignment of chromosome 22 of a Bornean and a Sumatran Orangutan [86,89]. The HMM used in this experiment had $N = 10$ states and the length of the alignment was $T = 35 \cdot 10^6$. The analysis using the old implementation took 2.95 hours while the implementation using parredForward took 1.94 hours, giving a speed-up of a factor of 1.52. The relatively meager speedup can be explained by the CoalHMM framework having a quite significant overhead, especially the derivation of the transition matrices from the coalescent takes a long time.
13.4 Conclusion

We have demonstrated how Hidden Markov Models with small state spaces can be parallelized. Although the obtained speed-up is not proportional to the number of processors, our approach actually does provide a significant improvement, as opposed to previous methods that were counter-productive for such HMMs. Speeding up processing of HMMs with a small state space is highly relevant because many HMMs are handcrafted and have small state spaces.

One important aspect of our method is that it requires very little communication between processors, making it a candidate for use on general purpose graphical processing units, or distribution over a network. Other future work would include applying our method to the Baum-Welch parameter estimation and the posterior decoding algorithms.

13.5 Acknowledgment

We thank Thomas Mailund and Anders Egerup Halager for help with, and data for the CoalHMM experiments.
Chapter 14

A Coarse-to-Fine Approach to Computing the $k$-Best Viterbi Paths

The paper *A Coarse-to-Fine Approach to Computing the $k$-Best Viterbi Paths* presented in this chapter has been published as a conference paper.


Except for typographical and formatting changes the content of this chapter is equal [97]. An implementation of the methods presented in this chapter is available at [http://www.birc.dk/~jn/c2flib](http://www.birc.dk/~jn/c2flib).
A Coarse-to-Fine Approach to Computing the $k$-best Viterbi Paths

Jesper Nielsen*

Abstract

The Hidden Markov Model (HMM) is a probabilistic model used widely in the fields of Bioinformatics and Speech Recognition. Efficient algorithms for solving the most common problems are well known, yet they all have a running time that is quadratic in the number of hidden states, which can be problematic for models with very large state spaces. The Viterbi algorithm is used to find the maximum likelihood hidden state sequence, and it has earlier been shown that a coarse-to-fine modification can significantly speed up this algorithm on some models. We propose combining work on a $k$-best version of Viterbi algorithm with the coarse-to-fine framework. This algorithm may be used to approximate the total likelihood of the model, or to evaluate the goodness of the Viterbi path on very large models.

14.1 Introduction

A Hidden Markov Model (HMM) [107] is a probabilistic model, in which there is a series of hidden states evolving through time, each state depending only on the previous state. At each time-step the current hidden state emits an observable symbol, with the hidden state determining the probability of a given observable symbol being emitted. HMMs are used widely in many fields, particularly Bioinformatics [9, 45–47, 55, 73, 74, 120, 133, 144], and Speech Recognition [33, 82, 138].

One of the reasons for the success of the HMM framework is probably the existence of simple and efficient algorithms for solving the most common problems associated with HMMs. The Viterbi algorithm computes the maximum likelihood sequence of hidden states given the model and observed symbols; the forward algorithm computes the total likelihood of the observed symbols given the model; the backward algorithm, used together with the forward algorithm, can give the total likelihood of a given hidden state at a given point in time; and this again can be used by the Baum-Welch algorithm to learn the probabilities in the model from data.

The Viterbi, forward, and backward algorithms are all similar and all have an execution time that is linear in the product of the number of time-steps.

*Bioinformatics Research Centre, Aarhus University, C.F. Møllers Alle 8, DK-8000 Aarhus C, Denmark. E-mail: jn@birc.au.dk.
Chapter 14. A Coarse-to-Fine Approach to $k$-Best Viterbi

and the number of possible transitions between hidden states. For most use cases this is good enough, but if the transition matrix is dense the number of transitions is quadratic in the number of states, and if the number of states is large this can be a problem.

To achieve fast execution of the Viterbi algorithm on HMMs with a large number of hidden states a coarse-to-fine framework has been proposed and used successfully [29, 58, 108]. The idea is to approximate the desired HMM by a series of coarse HMMs with increasingly fewer states, each of the coarse states corresponding to several states in the previous, finer, HMM. Next, the Viterbi algorithm is used to find the maximally likely path through the coarse HMM and replace all states in this path with the states represented by it in the finer HMM. This is repeated until you find a path containing only states from the original HMM. If the coarse HMMs are constructed correctly this will be the exact maximum likelihood path in your original HMM.

Another extension to the Viterbi algorithm is a $k$-best version [68]. Instead of only finding the single most likely path the $k$ most likely ones are found. In this paper we propose to combine the coarse-to-fine technique with the $k$-best Viterbi algorithm, giving a $k$-best Viterbi algorithm that is fast on very large HMMs.

A coarse-to-fine $k$-best Viterbi algorithm has also been proposed in [31], but that article does, strictly speaking, not use HMMs, and the way they use coarse-to-fine means that they only get an approximate solution.

14.2 Methods

We will use a notation similar to that in [107]. Let the set of $N$ distinct hidden states be denoted by $S = \{S_1, S_2, ..., S_N\}$, and let $Q = q_1q_2 \cdots q_T$ be the sequence of $T$ actual hidden states. Such a sequence of states we will also call a path. Similarly let $V = \{V_1, V_2, ..., V_M\}$ be the set of $M$ distinct observable symbols, and $O = O_1O_2 \cdots O_T$ the sequence of $T$ actual observations. Formally, an HMM is a three-tuple $\lambda = (A, B, \pi)$, where $A = \{a_{ij}\}$, $a_{ij} = P(q_t = j \mid q_{t-1} = i)$, with $i, j \in S$, is the transition matrix, $B = \{b_j(o)\}$, $b_j(o) = P(O_t = o \mid q_t = j)$, with $j \in S$ and $o \in V$, is the distribution of observable symbols, and finally $\pi_i = P(q_1 = i)$, for $i \in S$, is the initial distribution vector.

The probability of a given sequence of hidden states $Q$ and observed symbols $O$ is then:

$$P(O, Q \mid \lambda) = P(O \mid Q, \lambda)P(Q \mid \lambda)$$

$$= \left( \prod_{t=1}^{T} P(O_t \mid q_t, \lambda) \right) \left( P(q_1 \mid \lambda) \prod_{t=2}^{T} P(q_t \mid q_{t-1}, \lambda) \right)$$

$$= \left( \prod_{t=1}^{T} b_{q_t}(O_t) \right) \left( \pi_{q_1} \prod_{t=2}^{T} a_{q_{t-1}q_t} \right).$$

In most real-world scenarios we would not know the path of actual hidden states $Q$. We are going to assume that only the observations $O$ and the model parameters $\lambda$ are known, in this article.
14.2. Methods

14.2.1 The Viterbi algorithm

The classical way to estimate $Q$ is to find the maximum likelihood $Q$ using the Viterbi algorithm [107]. We want to maximize

$$P(Q|O, \lambda) \propto P(Q, O|\lambda).$$

This is done by defining $\delta_t(i)$ to be the likelihood of the most likely sequence of states from time 1 to time $t$ and ending in state $i$

$$\delta_t(i) = \max_{q_1, q_2, \ldots, q_{t-1}} \left\{ P(q_1 q_2 \cdots q_{t-1}, q_t = i, O_1 O_2 \cdots O_t | \lambda) \right\},$$

which can be computed efficiently using dynamic programming

$$\delta_t(i) = \begin{cases} \pi_i b_i(O_1) & \text{if } t = 1 \\ \max_{j \in S} \{ \delta_{t-1}(j) a_{ji} b_i(O_t) \} & \text{otherwise} \end{cases}.$$

The above technically only gives rise to the likelihood of the path, but the actual path can be found by backtracking which entries gave rise to the result of each max operation.

The above algorithm has an execution time of $O(N^2 T)$. This is fast enough for many practical purposes, but due to the $N^2$ term the algorithm may be inadequate if $N$ is big.

14.2.2 Coarse-to-fine

In the case of large $N$, a coarse-to-fine approach may be used [108]. Let $T$ be a tree with hidden states $S$ as leaves. Let $R(T)$ be the root of $T$. Finally, let $c(i)$ be the set of immediate children of node $i$, where $i$ is any node in $T$. If $i$ is a leaf in $T$, that is if $i \in S$, we set $c(i) = \{i\}$ for mathematical convenience.

The nodes in $T$ are going to be the hidden states in a new HMM, so we also need to define the probabilities in this new HMM. The probabilities for an internal node in the tree is simply going to be the maximum over the probabilities for all the children

$$a^{T}_{ij} = \begin{cases} a_{ij} & \text{if } i \text{ and } j \text{ are leaves} \\ \max_{i' \in c(i), j' \in c(j)} \{a^{T}_{i'j'}\} & \text{otherwise} \end{cases},$$

$$b^{T}_{j}(k) = \begin{cases} b_{j}(k) & \text{if } j \text{ is a leaf} \\ \max_{j' \in c(j)} \{b^{T}_{j'}(k)\} & \text{otherwise} \end{cases},$$

and

$$\pi^{T}_{i} = \begin{cases} \pi_{i} & \text{if } i \text{ is a leaf} \\ \max_{i' \in c(i)} \{\pi^{T}_{i'}\} & \text{otherwise} \end{cases}.$$

Strictly speaking $\lambda^{T} = (A^{T}, B^{T}, \pi^{T})$ is not an HMM, because $A^{T}$, $B^{T}$, and $\pi^{T}$ no longer define probabilities, since they do not necessarily sum to one. Also, it turns out that we do not actually need to compute the exact max, but that any upper bound will work, though a tighter bound should give a better execution.
time. We need to modify the Viterbi algorithm, such that it allows a different set of states for each time-step, thus let \( viterbi(O, S^{1n}, ..., S^{Tn}, \lambda^T) \) compute \( Q^n = q^n_1 q^n_2 \cdots q^n_T \), the most likely sequence of states, emitting the observed symbols, constrained to \( q^n_t \in S^{tn} \). If that Viterbi algorithm finds a path that only contains states that are leaves in \( T \), we will call it a true solution, since it is also a solution in the original HMM. Otherwise it is an estimate. The algorithm proposed by [108] is to start by setting \( S^1 = c(R(T)) \), repeatedly use the above Viterbi algorithm to find the most likely path \( Q^n \) and replace all states on that path by their children \( S^{t(n+1)} = (S^{tn} \setminus \{q^n_t\}) \cup c(q^n_t) \), until a true solution is found. During the execution, the algorithm can visit several states that are not associated with the final true solution, but [108] shows that once a true solution is found, it will also be the maximally likely path \( Q \) in the original HMM. This runs the Viterbi algorithm several times, but with a very small state space, and may therefore be faster than the original Viterbi algorithm on the full state space. The speed depends very much upon finding the true solution \( Q \) in few iterations, and not spending time visiting states unrelated to \( Q \). How well this succeeds depends on the concrete model, and how \( T \) is built.

### 14.2.3 \( k \)-best

Our work is based on the work of Huang and Chiang. In [68] they suggest four different algorithms for computing \( k \)-best Viterbi paths, numbered zero through three. The first algorithm is too inefficient to warrant our attention, and the second algorithm is an optimization that is not relevant to this work. We are going to use their algorithms two and three, and refer them as HC2 and HC3 respectively. Define \( \delta^k_l(i) \) to be the likelihood of the \( k \)th most likely path from time 1 to time \( t \) ending in state \( i \), thus \( \delta^1_l(i) = \delta_l(i) \). In the original Viterbi algorithm, we create a table of \( \delta_l(i) \) for all \((t, i)\) combinations. The HC2 algorithm simply extends this to storing a list of length \( k \), instead of a single entry in this table. The observation for HC3 is that the majority of the cells will not be involved in all of the \( k \) most likely paths, so we will delay the computation of \( \delta^k_l(i) \) until \( \delta^{k-1}_l(i) \) has actually been used in a path.

To explain how HC3 works in the framework of this article, define \( h_t(i) \) to be a heap associated with state \( i \) at time \( t \). \( h_t(i) \) contains values of the form \( \delta^l_{t-1}(j)a_{ji} \), and is used to determine how get the solution for the next \( \delta^l_t(i) \) from. \( j \) refers to the source state the solution is from, and \( l \) indicates the rank of the solution from \( j \). Obviously, the first column \( t = 1 \) does not have a previous column to get solutions from. For the first column the only path ending at a given state is the path containing only that state. Therefore we set \( \delta^1_1(i) = \pi_i b_i(O_1) \) and do not use heaps for that column. For the remaining columns the heaps are initially built from \( \{\delta^l_{t-1}(j)a_{ji}\}_{j \in S} \), remembering \( \delta^1_{t-1}(j) \) can be found from \( \delta_{t-1}(j) \).

To compute \( \delta^k_l(i) \), define two functions: processTop\((t, i)\) which is a utility function that processes the top of the heap \( h_t(i) \), and getSolution\((t, i, k)\) which actually finds and returns \( \delta^k_l(i) \). processTop\((t, i)\) simply pops the top \( \delta^l_{t-1}(j)a_{ji} \) from \( h_t(i) \), and computes the next \( \delta^{l+1}_t(i) = \delta^{l}_t(j)a_{ji}b_i(O_t) \). If one more solution \( \delta^{l-1}_{t-1}(j) \) from the source state \( j \) exists, it is pushed on to \( h_t(i) \) to replaced
14.2. Methods

the item that was just popped. Such a solution will not exist if there simply are not enough paths from time 1 ending in state $i$, at time $t$. For example $\delta^2_1(i)$ does not exist, and neither does $\delta^k_2(i)$ for $k$ larger than the total number of states $N$.

$getSolution(t, i, k)$ starts by checking if $\delta^k_t(i)$ has already been computed. If it has, then we simply return it. Otherwise it repeatedly calls $processTop$ until $\delta^{k-1}_t(i)$ has been computed. At this point $\delta^k_t(i)$ can be computed by peeking at the top of the heap. We do not call $processTop$, as that would pop the value off the heap and require us to push a new value. This new value might not actually be needed, and by deferring the computation of it we can save a significant amount of work.

To actually get the $k$ best paths for the entire HMM greedily consume and replace the best solution from the last column in the dynamic programming table, corresponding to $t = T$. This can be done efficiently, using a heap similarly to above: Build the heap from $\{\delta^1_T(i)\}_{i \in S}$. Next, repeatedly pop the best solution $\delta^k_T(i)$ off the stack, return it to the user, and insert the next candidate solution $\delta^{k+1}_T(i)$, from the source state $i$, if such a solution exists. Since you do not need to know $k$ before the algorithm is run, but can keep pulling new solutions until any arbitrary condition is satisfied, this is an on-line algorithm.

14.2.4 Coarse-to-fine $k$-best

The contribution of this paper is to combine the above $k$-best algorithms with the coarse-to-fine framework. For the HC2 algorithm this is relatively straightforward. The algorithm is the same as the coarse-to-fine Viterbi algorithm, except we find the $k$ best paths in each iteration, using the HC2 algorithm, and split all states involved in any of these. We refer to this as the C2FHC2 algorithm. One may note that since the states near the root of the tree represent many leaves they may have a significant fraction of the $k$ best paths passing through them. We have experimented with a version that counts the number of possible paths passing through a state and only split the minimal amount of nodes, necessary to find $k$ paths in the final iteration. It performed significantly worse than the naive approach, and therefore it is the naive version that is presented in this paper. There may exist a better strategy for deciding how many nodes to split in each iteration.

Extending the HC3 algorithm to C2FHC3 is more involved. We propose keeping the same basic structure as the HC3 algorithm, but updating $processTop$ and $getSolution$ to also handle the splitting of nodes into its children. Care is needed with the value returned by $getSolution$ as we need to know whether it is an estimate or a true solution. Furthermore we introduce the concept of a level of an estimate which is used in a heuristic to help make sure that the splitting of states is distributed evenly over all the time-steps, so that any local optima are discovered to be such, as early as possible. Define the level of an estimate to be the depth in $T$ of the highest non-simple state on the solution path. We are going to produce estimates of increasing levels, so we will at some point reach the leaves of $T$, and thus have a true solution. Intuitively we start by asking
getSolution to give us a solution at level 0. Next, repeatedly poll for solutions that are at one level deeper than the solution previously returned, thus pushing the path toward the leaves, until a true solution is returned.

Previously we used the \( k \) parameter in the \( \text{getSolution}(t, i, k) \) function and \( \delta_t^k(i) \) to indicate the rank of a solution. We now allow these to take the value of \( \text{est}(m) \), to denote an estimate at level \( m \). Thus \( \text{getSolution}(t, i, \text{est}(m)) \) requests the computation of an estimate at level \( m \), while \( \text{getSolution}(t, i, k) \) still requests the computation of the \( k \)th best true solution. If an estimate at level \( m \) is requested \( \text{getSolution} \) is allowed to return a solution at a deeper level or the first true solution \( \delta_t^1(i) \).

\( \text{processTop} \) still starts by popping the most promising value \( \delta_{t-1}(j)a_{ji}^T \) from \( h_t(i) \). If this is an estimate at some level \( l = \text{est}(m) \), we want to improve this estimate. If the source state \( j \) is already deeper than the level \( m \), or \( j \) cannot be split, because it is a simple state, we obtain this better estimate by calling \( \text{getSolution}(t - 1, j, \text{est}(m + 1)) \) and getting an estimate at a higher level from \( j \). If, instead, \( j \) is a candidate for splitting, a better solution can be obtained by doing that. If \( l \) is not an estimate we can use it to compute the next true solution for this state \( \delta_t^{k+1}(i) = \delta_{t-1}(j)a_{ji}^Tb_{ji}^T(O_t) \), pushing the next solution \( \delta_{t-1}^1(j)a_{j1}^T \) on to \( h_t(i) \), if it exists.

Finally, \( \text{getSolution}(t, i, k) \) also needs to be updated. First it checks whether an acceptable solution has already been computed, remembering that \( \delta_t^1(i) \) is an acceptable solution if \( k \) is an estimate. If no such solution is found \( \text{processTop} \) is called until one is found, possibly on the heap.

The above does leave out the details of the base cases. \( \text{processTop} \) refers to the previous time-step, which is still not well-defined for the first column. As in the original HC3 algorithm we do not use any heap for the first column, set \( \delta_1^1(i) = \pi_i^Tb_i^T(O_1) \), and initialize the remaining heaps from the first estimate of the preceding states \( \{\text{getSolution}(t - 1, j, \text{est}(0))\}_{j \in \mathbb{R}(\mathcal{T})} \).

### 14.2.5 Building \( \mathcal{T} \)

The tree \( \mathcal{T} \) is irrelevant for the correctness of the result, but it can have a very large impact on the execution time of the algorithm. There are many different ways to build \( \mathcal{T} \), but for our experiments we built it bottom-up, based on a cost function \( K(\mathcal{T}) \). Start out with each state \( q \in S \) being a small tree containing only itself as root. Now build the tree by repeatedly joining the two trees giving the cheapest result according to the cost function \( K \) until only one tree remains. The motivation for the cost function \( K \) is to minimize the expected number of states visited. To define the cost function we first find the \textit{a priori} probability of the hidden states \( \mathbb{P}(q) \) from the stationary distribution of the transition matrix. From this we can also find the probability of a subtree as the sum of the probabilities of the children \( \mathbb{P}(\mathcal{T}) = \sum_{\mathcal{T}' \in \mathbb{R}(\mathcal{T})} \mathbb{P}(\mathcal{T}') \), and the probability of an observable symbol \( o \in V \) as \( \mathbb{P}(o) = \sum_{q \in S} \mathbb{P}(o \mid q) \mathbb{P}(q) \).

Furthermore define

\[
R(o \mid \mathcal{T}) = \begin{cases} 
\mathbb{P}(o \mid q) & \text{if } \mathcal{T} \text{ is a single state } q \\
\max_{\mathcal{T}' \in \mathbb{R}(\mathcal{T})} \{R(o \mid \mathcal{T}')\} & \text{otherwise}
\end{cases}
\]
and \( R(T \mid o) = \frac{R(o \mid T)^p(T)}{C(o)} \). Using this we also define

\[
K(T \mid o) = \begin{cases} 
R(T \mid o) & \text{if } T \text{ is a single state } q \\
R(T \mid o) \left(1 + \sum_{T' \in c(R(T))} K(T' \mid o)\right) & \text{otherwise}
\end{cases}
\]

and finally \( K(T) = \sum_{o \in V} K(T \mid o)^p(o) \). If we cache \( R(o \mid T) \) and \( K(T \mid o) \) for the children of \( T \), we can compute \( K(T) \) in time \( O(M) \), and the entire tree can be built in \( O(N^2M) \), using a quad-tree [53], if we ignore the time it takes to find the stationary distribution of the transition matrix. In our implementation that distribution is approximated by multiplying the transition matrix to the initial probability vector 25 times, which is \( O(N^2) \).

14.3 Results

The model parameters \( \lambda \) are important for the tree \( T \) and the running time of the algorithm. Therefore we have experimented with four different ways to generate them. The first is to set the emission probabilities to the uniform distribution, while the transition probabilities have been randomly drawn. Thus the emissions are ignored by the HMM and the most likely path is determined only by the transition matrix. Similarly we have used an HMM where the transition probabilities are the uniform distribution, while the emission probabilities are randomly chosen. This gives an HMM where the most likely path is determined only by the observed sequence. The third parameter set was built randomly based on a tree, with states clustered closely in the tree also resembling each other, to give a random HMM that is guaranteed to have some structure the clustering algorithm can exploit. Finally, we have used a completely random HMM, where both the transition and emission probabilities were randomly drawn.

Only the sum of the likelihood of the found Viterbi paths are computed and timed, thus no backtracking is performed. The shown values include both the time to build \( T \) and the time to run the algorithm, but not the time to read input from disk. All experiments were run on three different HMMs and the lines in the plot follows the averages of them. The experiments were run on a MacPro with two Intel quad-core Xeon processors running at 2.26GHz and with 8GB of main memory. All the benchmarks were run on using our own implementations of the algorithms.

In the experiments shown in Fig. 14.1 we have benchmarked the execution time of the algorithm against the number of states \( N \). The execution time of neither the HC2 nor the HC3 algorithms changes significantly between different HMMs, which was expected. The HC3 method is good for small state spaces, while the HC2 method performs better for larger state spaces. The models without emission probabilities are generally worst-cases for the coarse-to-fine methods and the models without transitions are best-cases. Without transitions it is trivial to find the most likely path and for those models profiling show that the time to build \( T \) dominates for \( N > 2^7 \). Without that time included the coarse-to-fine methods can be several orders of magnitude faster than the non-coarse-to-fine methods. Note how well the C2FHC3 performs for those models,
Chapter 14. A Coarse-to-Fine Approach to k-Best Viterbi

Number of hidden states versus execution time

![Graphs showing execution time vs. number of hidden states for different methods and emission/transitions types.](image)

**Figure 14.1:** Results from experiments testing execution time as a function of state space size. $k = 1000$, $M = 10$ and $T = 100$. Plotted are the running time divided by the number of states squared and the sequence length. Each experiment was repeated on three different models, with the line showing the average.

although it generally is slow. The hierarchical and random models show the methods under more realistic conditions, and we see that all the coarse-to-fine methods perform badly when the state space is small, but that C2FHC2 can be somewhat faster than the competing algorithms when $N$ is sufficiently large.

In Fig. 14.2 we show results from benchmarks of the impact of the $k$ parameter. What we see from these graphs is that the speed of HC2 versus HC3 depends very much on the $k$ parameter. HC3 might take a long time to build the heaps, but getting the next solution is extremely cheap, once they are built. HC2 is more sensitive to a large $k$.

We have also applied our methods to the technique presented in [144]. The topic of that paper is to jointly estimate genetic crossover and gene conversion rates, which they do using a hill-climbing method to find the maximum likelihood parameter set, with the likelihood of a given set of parameters computed using a number of HMMs. In the article the forward method [107] is used to compute the exact $P(Q | \lambda)$, but since we are only interested in the shape of the fitness landscape, and not the actual values, the result from a $k$-best Viterbi algorithm might be a good approximation. We ran our experiment on
data with 40 sequences of length 35 generated by the \textit{ms} program [69]. The method of Yin et al. generates several different HMMs of increasing complexity, with the biggest having a number of states that is cubic in the number of input sequences. The generated HMMs have a structure such that we can build $\mathcal{T}$ in time $O(N)$, using a domain-specific algorithm, and that the transition probabilities can be computed on-demand in time $O(1)$. However, the structure also allows the forward algorithm, that normally runs in time $O(N^2T)$, to be computed in time $O(NT)$. For the timing of the $O(NT)$ forward algorithm the implementation of Yin et al. was used. For comparison we have also implemented a straightforward $O(N^2T)$ forward algorithm.

The results are shown in Table 14.1. We may notice that C2FHC3 is surprisingly fast compared to the previous experiments. The generated HMMs have only 16 distinct transition probabilities, and since there are many thousands of states the most relevant of our previous experiments would likely be the one with uniform transition probabilities. However, in that experiment our methods were limited by the time it took to construct the tree $\mathcal{T}$, and that is less of an issue in this application.
Table 14.1: Running times for algorithms using an HMM to estimate the likelihood of data given parameters on genetic crossover and gene conversion rates. Many different HMMs are generated, but in the worst case the parameters are $N = 65640$, $M = 2$, $T = 35$, and we set $k = 1000$. The method of Yin et al. is based on a modified forward algorithm, using domain-specific knowledge, running in time $O(NT)$.

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>HC2</td>
<td>34669 s</td>
</tr>
<tr>
<td>HC3</td>
<td>8858 s</td>
</tr>
<tr>
<td>C2FHC2</td>
<td>831 s</td>
</tr>
<tr>
<td>C2FHC3</td>
<td>55.3s</td>
</tr>
<tr>
<td>Forward</td>
<td>137 s</td>
</tr>
<tr>
<td>Yin et al.</td>
<td>15.2s</td>
</tr>
</tbody>
</table>

14.4 Conclusion

We have shown how the coarse-to-fine heuristic can be combined with a $k$-best Viterbi algorithm in a way that can achieve a significant speed-up for HMMs with large state spaces in cases where the model has a suitable structure.

Remembering that the total likelihood of the data is the sum of the likelihood of all paths through all hidden states this may be used to approximate the forward algorithm for HMMs where the state space is so large that it is infeasible to run the traditional algorithm. Alternatively it may be used in its own right to find the $k$-best paths or as an extension to the Viterbi algorithm that will also give an informal sense of the variance and reliability of the result.

Building the tree and predicting whether this approach will work well for a given model remains an unsolved problem although the algorithm for building trees suggested in this paper seems to work well in general. The goal of this method is primarily to be faster than quadratic in the number hidden states. However building the transition matrix is quadratic in the number of hidden states, so this method might be especially suitable to models that have enough structure that the tree and all probabilities can be computed efficiently on demand, without storing them.

Source code can be downloaded from www.birc.dk/~jn/c2flib.

14.5 Acknowledgments.

Part of this research was carried out at the University of California, Berkeley. We thank Yun S. Song and Michael I. Jordan for suggesting the problem and Junming Yin for discussions on it.
Chapter 15

A sub-cubic time algorithm for computing the quartet distance between two general trees

The paper presented in this chapter has been published as a conference paper and in a journal. The conference paper was originally submitted and accepted as A Quadratic Time Algorithm for Computing the Quartet Distance between Two General Trees. Unfortunately an error was found in that paper. We derived a new algorithm running in sub-cubic time, but although we notified the conference committee, it was too late for changes.


The journal paper presents the corrected algorithm, along with an implementation and experimental results. Except for typographical and formatting changes the content of this chapter is equal to the journal paper [98]. Additional file 1 can be found online at http://www.almob.org/content/6/1/15/suppl/S1, and additional file 2 at http://www.almob.org/content/6/1/15/suppl/S2. An implementation of the algorithm presented in this chapter is available at http://birc.au.dk/software/qdist.
A sub-cubic time algorithm for computing the quartet distance between two general trees

Jesper Nielsen∗ ‡, Anders K Kristensen† §, Thomas Mailund∗ ¶, Christian N S Pedersen∗ ‥

Abstract

When inferring phylogenetic trees different algorithms may give different trees. To study such effects a measure for the distance between two trees is useful. Quartet distance is one such measure, and is the number of quartet topologies that differ between two trees.

We have derived a new algorithm for computing the quartet distance between a pair of general trees, i.e. trees where inner nodes can have any degree $\geq 3$. The time and space complexity of our algorithm is sub-cubic in the number of leaves and does not depend on the degree of the inner nodes. This makes it the fastest algorithm so far for computing the quartet distance between general trees independent of the degree of the inner nodes.

We have implemented our algorithm and two of the best competitors. Our new algorithm is significantly faster than the competition and seems to run in close to quadratic time in practice.

15.1 Background

The evolutionary relationship between a set of species is conveniently described as a tree, where the leaves represent the species and the inner nodes speciation events. Using different inference methods to infer such trees from biological data, or using different biological data from the same set of species, often yield slightly different trees. To study such differences in a systematic manner, one must be able to quantify differences between evolutionary trees using well-defined and efficient methods. One approach for this is to define a distance measure between trees and compare two trees by computing this distance. Several distance measures have been proposed, e.g. the symmetric difference [110], the nearest-neighbour interchange [136], the subtree transfer distance [10], the...

∗Bioinformatics Research Centre (BiRC), Aarhus University, C. F. Møllers Allé 8, DK-8000 Aarhus C, Denmark
†Department of Computer Science, Aarhus University, Åbogade 34, DK-8200 Aarhus N, Denmark
‡E-mail: jn@birc.au.dk.
§E-mail: anders.kabell.kristensen@gmail.com.
¶E-mail: mailund@birc.au.dk.
‖E-mail: cstorm@birc.au.dk.
Chapter 15. A sub-cubic time algorithm for the quartet distance

Figure 15.1: The four possible quartet topologies of species $a$, $b$, $c$, and $d$. For binary trees, only the butterfly quartets are possible.

Robinson and Foulds distance [111], and the quartet distance [51]. Each distance measure has different properties and reflects different properties of the tree relationship.

For an evolutionary tree, the quartet topology of four species is determined by the minimal topological subtree containing the four species. The four possible quartet topologies of four species are shown in Fig. 15.1. Given two evolutionary trees on the same set of $n$ species, the quartet distance between them is the number of sets of four species for which the quartet topologies differ in the two trees.

Most previous work has focused on comparing binary trees and therefore avoided star quartets. Steel and Penny in [124] developed an algorithm for computing the quartet distance in time $O(n^3)$. Bryant et al. in [26] improved this result with an algorithm that computes the quartet distance in time $O(n^2)$. Brodal et al., in [23], presented the currently best known algorithm that algorithm computes the quartet distance in time $O(n \log n)$.

Recently, we have developed algorithms for computing the quartet distance between two trees of arbitrary degrees, i.e. trees that can contain star quartets. In [34] we developed two algorithms: the first algorithm runs in time $O(n^3)$ and space $O(n^2)$—and is thus independent of the degree of the inner nodes—the second in time $O(n^2d^2)$ and space $O(n^2)$, where $d$ is the maximal degree of inner nodes in the trees—and thus depends on the degree of the nodes. The $O(n^2d^2)$ was later improved to $O(n^2d)$ [36], and by taking an approach similar to the Brodal et al. [23] $O(n \log n)$ we developed a sub-quadratic algorithm in terms of $n$ but at a significant cost in terms of $d$: $O(d^9n \log n)$ [126].

In this paper we develop an $O(n^{2+\alpha})$ algorithm, where $\alpha = \frac{\omega}{2^\omega - 1}$ and $O(n^\omega)$ is the time it takes to multiply two $n \times n$ matrices. Using the Coppersmith-Winograd [39] algorithm, where $\omega = 2.376$, this yields a running time of $O(n^{2.688})$. The running time is thus independent of the degrees of the inner nodes of the input trees, and this is the first sub-cubic time algorithm with this property. Furthermore we have implemented the algorithm, along with
15.2 Methods: A sub-cubic time and space algorithm

The quartet distance between two trees is the number of quartets where the quartet topology differs between the two trees, i.e. the number of quartets where one tree has the star topology and the other a butterfly topology, plus the number of quartets where the trees have a different butterfly topology. As observed in [34], the former—where one tree has the star topology and the other a butterfly topology—can be expressed in terms of the total number of butterflies in the two trees, the number of shared butterflies and the number of different butterflies: For trees $T$ and $T'$, the number of different topologies due to one being a star and the other a quartet, $\text{diff}_S(T, T')$, is given by

$$\text{diff}_S(T, T') = B + B' - 2 \left( \text{shared}_B(T, T') + \text{diff}_B(T, T') \right), \quad (15.1)$$

where $B$ is the number of butterflies in $T$, $B'$ the number of butterflies in $T'$, $\text{shared}_B(T, T')$ the number of quartets with the same butterfly topology in $T$ and $T'$ and $\text{diff}_B(T, T')$ the number of quartets with different butterfly topologies in $T$ and $T'$. Thus the quartet distance between $T$ and $T'$ is given by the expression

$$\text{qdist}(T, T') = B + B' - 2 \text{shared}_B(T, T') - \text{diff}_B(T, T') \ . \quad (15.2)$$

Since, $B = \text{shared}_B(T, T)$ and $B' = \text{shared}_B(T', T')$, an algorithm for computing $\text{shared}_B(T, T')$ and $\text{diff}_B(T, T')$ gives an algorithm for computing the quartet distance between $T$ and $T'$.

Our approach to counting the shared and different quartets is based on directed quartets and claims [23,34]. An (undirected) butterfly quartet topology, $ab|cd$ induces two directed quartet topologies $ab \rightarrow cd$ and $ab \leftarrow cd$, by the orientation of the middle edge of the topology, as shown in Fig. 15.2. There are twice as many directed butterflies as undirected. If $e = (s_e, t_e)$ is a directed edge from $s_e$ to $t_e$ we call $s_e$ the source of $e$, and $t_e$ the target. To each directed quartet, $ab \rightarrow cd$, we can uniquely associate the directed edge, $e$ so

![Figure 15.2: An undirected quartet topology, (a), and the two directed quartet topologies, (b) and (c), it induces.](image-url)
Chapter 15. A sub-cubic time algorithm for the quartet distance

Figure 15.3: A claim $A \xrightarrow{e} (C, D)$. The claim $A \xrightarrow{e} (C, D)$ claims all ordered butterflies $ab \rightarrow cd$ where $a, b \in A$ and $c \in C, d \in D$ where $C$ and $D$ are two different subtrees in front of $e$.

Figure 15.4: A shared butterfly induces two butterflies in each tree, which will give four pairs of claims, however the butterflies will only be identical in two of these pairs, thus a shared butterfly will be counted twice. A different butterfly also induces four pairs of claims, but since we are counting different butterflies all four will be counted. The way we count shared butterflies prevents the two different butterflies induced by the shared (undirected) butterfly from being counted.

that $a$ and $b$ are leaves in the subtree rooted at $s_e$, and $c$ and $d$ are leaves in different subtrees rooted at $t_e$, see Fig. 15.3. We call such a tree substructure, consisting of a directed edge $e$ with a subtree, $A$ behind $e$ and two distinct subtrees, $C$ and $D$, in front of $e$ a claim, written $A \xrightarrow{e} (C, D)$. We say that the edge $e$ claims the directed quartet $ab \rightarrow cd$, and we also say that an edge $e$ claims an undirected quartet $ab|cd$ if it claims one of its directed quartets. Each (undirected) butterfly quartet defines exactly two directed butterfly quartets, and each directed quartet is claimed by exactly one directed edge; considering each claim and implicitly each directed butterfly claimed by the claim, we can examine each directed butterfly in a tree, or each undirected butterfly twice.

The crux of the algorithm is to consider each pair of claims, one from each tree, and for each such pair count the number of shared and different directed butterflies claimed in the two trees. This way each shared butterfly is counted
twice, and each different butterfly is counted four times, as shown in Fig. 15.4. Dividing the counts by two and four, respectively, gives us shared \( B(T, T') \) and \( \text{diff}_B(T, T') \).

### 15.2.1 Preprocessing

Before counting shared and different butterflies, we calculate a number of values in two preprocessing steps. First, we calculate a matrix that for each pair of subtrees \( F \in T \) and \( G \in T' \) stores the number of leaves in both trees, \( |F \cap G| \). This can be achieved in time and space \( O(n^2) \) [26].

Next, for each pair of inner nodes, \( v \in T, v' \in T' \) with sub-trees \( F_i, i = 1, \ldots, d_v \) and \( G_j, j = 1, \ldots, d_{v'} \), respectively, we calculate a matrix, \( I \), such that

\[
I[i, j] = |F_i \cap G_j|
\]

and we calculate vectors of its row and column sums, and the total sum of its entries:

\[
R[i] = \sum_{j=1}^{d_{v'}} I[i, j] \tag{15.3}
\]

\[
C[j] = \sum_{i=1}^{d_v} I[i, j] \tag{15.4}
\]

\[
M = \sum_{i=1}^{d_v} \sum_{j=1}^{d_{v'}} I[i, j] \tag{15.5}
\]

Inspired by the sums (S.3) – (S.6) in Additional file 1 we calculate a matrix \( I' \), vectors of its row and column sums, the total sum of its entries, and some further values

\[
I'[i, j] = I[i, j](M - R[i] - C[j] + I[i, j]) \tag{15.6}
\]

\[
R'[i] = \sum_{j=1}^{d_{v'}} I'[i, j] \tag{15.7}
\]

\[
C'[j] = \sum_{i=1}^{d_v} I'[i, j] \tag{15.8}
\]

\[
M' = \sum_{i=1}^{d_v} \sum_{j=1}^{d_{v'}} I'[i, j] \tag{15.9}
\]

\[
R''[i] = \sum_{j=1}^{d_{v'}} I[i, j](C[j] - I[i, j]) \tag{15.10}
\]

\[
C''[j] = \sum_{i=1}^{d_v} I[i, j](R[i] - I[i, j]) \tag{15.11}
\]
Calculating the values in Eq. (15.16) and (15.17) takes time \( \mathcal{O} \) in Eq. (15.18) and (15.19) takes time \( \mathcal{O} \), depending on which pair is fastest to calculate. We thus calculate either Eq. (15.16) and (15.17), or Eq. (15.18) and (15.19), depending on which pair is fastest to calculate.

\[
R''[i] = \sum_{j=1}^{d_v} I[i,j]^2 \quad (15.13)
\]

\[
C''[j] = \sum_{i=1}^{d_v} I[i,j]^2 \quad (15.14)
\]

Calculating the values in Eq. (15.3) – (15.14) can be done in time \( \mathcal{O}(d_v d_{v'}) \) for each pair of inner nodes \((v, v') \in T \times T'\), giving a total time of \( \mathcal{O}(\sum_{v \in T} \sum_{v' \in T'} d_v d_{v'}) = \mathcal{O}(n^2) \).

Finally, we need to calculate the following values:

\[
I'''[i,j] = \sum_{k=1}^{d_v} \sum_{l=1, l \neq j}^{d_v} I[i,l]I[k,j]I[k,l] \quad (15.15)
\]

which takes time \( \mathcal{O}(d_v^2 d_{v'}) \) for each pair of inner nodes, giving a total time of \( \mathcal{O}(n^4) \), if done naively. However, as we show in section 1 of Additional file 1, the values in Eq. (15.15) can be calculated faster if we precompute either \( I''_I = I I' \) and \( I''_2 = I_2 I \), or \( I''_I = I I' \) and \( I''_2 = I_2 I \), depending on which pair of matrices is fastest to compute, where \( I \) is the \( d_v \times d_{v'} \) matrix defined above.

We thus calculate either Eq. (15.16) and (15.17), or Eq. (15.18) and (15.19), depending on which pair is fastest to calculate.

\[
I''_I[i,k] = \sum_{j=1}^{d_v} I[i,j]I[k,j] \quad (15.16)
\]

\[
I''_2[i,k] = \sum_{j=1}^{d_v} I[k,j]I''_2[i,j] \quad (15.17)
\]

\[
I''_I[j,l] = \sum_{i=1}^{d_v} I[i,j]I[i,l] \quad (15.18)
\]

\[
I''_2[j,l] = \sum_{i=1}^{d_v} I[i,l]I''_2[i,l] \quad (15.19)
\]

Calculating the values in Eq. (15.16) and (15.17) takes time \( \mathcal{O}(\max(d_v, d_{v'})^\omega) \) if padding the matrices to become square and with \( \omega = 2.376 \) if using the Coppersmith-Winograd algorithm [39] for matrix multiplication, or time \( \mathcal{O}(d_v^2 d_{v'}) \) if using naive matrix multiplication. Similarly, calculating the values in Eq. (15.18) and (15.19) takes time \( \mathcal{O}(\max(d_v, d_{v'})^\omega) \) or \( \mathcal{O}(d_v d_{v'}^2) \). Computing either \( I''_I \) and \( I''_2 \), \( I''_I \) and \( I''_2 \), or \( I''_I \) and \( I''_2 \), thus takes time \( \mathcal{O}(\min(\max(d_v, d_{v'})^\omega, d_v^2 d_{v'}, d_v d_{v'}^2)) \).

### 15.2.2 Counting shared butterfly topologies

For each directed pair of inner edges, \( e \in T \) and \( e' \in T' \), see Fig. 15.5, we count the directed butterflies claimed by both \( e \) and \( e' \). These are all on the form \( ab \to cd \), where \( a, b \in F_i \cap G_j \), \( c \in F_k \cap G_l \) and \( d \in F_m \cap G_n \) for some claims,
15.2 Methods: A sub-cubic time and space algorithm

Figure 15.5: A pair of inner edges, $e \in T$, $e' \in T'$, where $F_i \ (G_j)$ is the sub-tree behind $e \ (e')$ and $F_k, k \neq i \ (G_l, l \neq j)$ the remaining subtrees of the node pointed to by $e \ (e')$. Highlighted are two claims, one from each tree.

Figure 15.6: Graphical illustration of the shared quartet expression, eq. (15.20). On the left, the matrix entries summed over are explicitly shown. On the right, the inner sum is implicitly shown. The sum of the greyed entries can be computed in constant time.

$$F_i \xrightarrow{e} (F_k, F_m) \text{ and } G_j \xrightarrow{e'} (G_l, G_n), \text{ of } e \text{ and } e'. \text{ The total number of directed butterflies common for both } e \text{ and } e' \text{ is therefore given by the expression}$$

$$\frac{1}{2} \left( \frac{|F_i \cap G_j|}{2} \right) \sum_{k \neq i \atop l \neq j} |F_k \cap G_l| \sum_{m \neq i, k \atop n \neq j, l} |F_m \cap G_n| \quad (15.20)$$

or the sum of $\frac{1}{2} \left( \frac{I[i,j]}{2} \right) \cdot I[k, l] \cdot I[m, n]$ for all distinct entries in $I$ but fixed $(i, j)$, see Fig. 15.6(a). We divide by two since we count each quartet twice, due to symmetry between the $(k, l)$ and $(m, n)$ pairs.

Notice, however, that the inner sum is simply the total sum of entries in $I$, $M$, except for the rows $i$ and $k$ and columns $j$ and $l$, see Fig. 15.6(b). Using

$$\sum_{m \neq i, k \atop n \neq j, l} |F_m \cap G_n| = M - \sum_{q=i,k \atop r=j,l} R[q] - \sum_{r=j,l} C[r] + \sum_{q=i,k \atop r=j,l} I[q, r] \quad (15.21)$$

and the precomputed values we can, as shown in section 2 of Additional file 1,
Figure 15.7: Graphical illustration of the different quartet expression, eq. (15.23). On the left, the matrix entries summed over are explicitly shown. On the right, the inner sum is implicitly shown. The sum of the greyed entries can be computed in constant time.

rewrite the expression in Eq. (15.20) to

$$\frac{1}{2} \left( I[i,j] \right) \left( M' - R'[i] - C'[j] + I'[i,j] + \right)$$

$$\left( I[i,j] - R[i] - C[j] \right) \left( M - R[i] - C[j] + I[i,j] \right) +$$

$$R''[i] - I[i,j] \left( C[j] - I[i,j] \right) +$$

$$C''[j] - I[i,j] \left( R[j] - I[i,j] \right)$$

which can be computed in time $O(1)$, if the referenced matrices have been precomputed. Thus we can compute all shared directed butterflies in total time $O(n^2)$. Dividing by two, we get the number of shared undirected butterflies.

15.2.3 Counting different butterfly topologies

Counting the number of different butterflies in the two trees is done similar to counting the number of shared butterflies. As before, we consider a pair of inner edges, $e \in T$ and $e' \in T'$. The quartets claimed by both $e$ and $e'$, but with different butterfly topology, are on the form $a \in F_i \cap G_j$, $b \in F_i \cap G_j$, $c \in F_k \cap G_j$ and $d \in F_m \cap G_n$ for some claims $F_i \xrightarrow{e} (F_k, F_m)$ and $G_j \xrightarrow{e'} (G_l, G_n)$. The number of butterflies claimed by both $e$ and $e'$ but with different topology is therefore given by

$$|F_i \cap G_j| \sum_{k \neq i} \sum_{l \neq j} |F_i \cap G_l||F_k \cap G_j| \sum_{m \neq i,k} \sum_{n \neq j,l} |F_m \cap G_n|$$

or the sum of $I[i,j] \cdot I[i,l] \cdot I[k,j] \cdot I[m,n]$ for all distinct entries in $I$ but fixed $(i,j)$, see Fig. 15.7. In this case there is no need to divide by any normalizing constant, since there are no symmetries between $k$ and $m$ or between $l$ and $n$.

As before, the inner sum can be expressed as in Eq. (15.21), and using the precomputed values we can, as shown in section 3 of Additional file 1, rewrite
the expression in Eq. (15.23) as

\[
I[i, j]\left((M - R[i] - C[j] + I[i, j])(R[i] - I[i, j])(C[j] - I[i, j]) + (R[i] - I[i, j])(I[i, j])(R[i] - I[i, j]) - C''[j]) \right) + \\
(R[i] - I[i, j])(I[i, j])(C[j] - I[i, j]) - C''[j]) + \\
(C[j] - I[i, j])(I[i, j])(C[j] - I[i, j]) - R''[i]) + \\
I''[i, j] - I[i, j]I''[i, i] - I[i, j](C''[j] - I[i, j]^2) \right) 
\]

or

\[
I[i, j]\left((M - R[i] - C[j] + I[i, j])(R[i] - I[i, j])(C[j] - I[i, j]) + (R[i] - I[i, j])(I[i, j])(R[i] - I[i, j]) - C''[j]) \right) + \\
(R[i] - I[i, j])(I[i, j])(C[j] - I[i, j]) - C''[j]) + \\
(C[j] - I[i, j])(I[i, j])(C[j] - I[i, j]) - R''[i]) + \\
I''[i, j] - I[i, j]I''[i, j] - I[i, j](R''[i] - I[i, j]^2) \right) 
\]

depending on whether we have precomputed \(I''[i, i]\) and \(I''[i, j]\), or \(I''[j, i]\) and \(I''[j, j]\). We can thus compute Eq. (15.23) in time \(O(1)\) for each pair of inner edges \(e \in T\) and \(e' \in T'\), giving a total time of \(O(n^3)\) to compute different directed, and thus different undirected, butterfly topologies in the two trees.

To get the actual number of different butterflies we have to divide by four.

### 15.2.4 Time analysis

The running time of the algorithm is dominated by the time \(O(\min(\max(d_v, d_{v'})^{2.376}, d_v^2d_{v'}, d_vd_{v'}^2))\) it takes to compute either \(I''[i, i]\) and \(I''[i, j]\), or \(I''[j, i]\) and \(I''[j, j]\), for each pair of nodes \(v \in T\) and \(v' \in T'\). Let \(O(n^\omega)\) be the time it takes to multiply two \(n \times n\) matrices. In section 4 of Additional file 1 we show that the running of our algorithm is \(O(n^{2+\alpha})\), where \(\alpha = \frac{\omega - 1}{2}\). Using the Coppersmith-Winograd algorithm [39] for matrix multiplication, where \(\omega = 2.376\), this yields a running time of \(O(n^{2.688})\).

### 15.2.5 Results

We have implemented our new algorithm and, for comparison, the \(O(n^3)\) and \(O(n^4)\) algorithms [34] for general trees. We chose those algorithm instead of those from [36,126], because the running time of those algorithms are dependent on the degree of the nodes, while a major feature of our new algorithm is that it has a good asymptotical running time independent of the degree of the nodes. For matrix multiplication we link to a BLAS library, and expect that to choose the most efficient algorithm for matrix multiplication. In our experiments the vecLib library from Mac OS X is used. We have run benchmarks with trees with ten leaves up to trees with almost 15,000 leaves. For each size, trees were generated in four different ways: general trees, binary trees, star trees and trees with one node of degree \(\frac{n}{2}\) surrounded by degree 3 nodes. The code that generated the trees is available in Additional file 2. For each of the ten possible combinations of topologies, one pair of trees were randomly generated,
and the time used for the computation of the quartet distance was measured and plotted. Our experiments were run on a Mac-Pro with two Intel quad-core Xeon processors running at 2.26GHz and with 8GB RAM. As seen in Fig. 15.8 the implementation of our new algorithm is significantly faster than the implementations of the competing algorithms, on trees with many leaves. In the worst cases our algorithm approaches $O(n^3)$ which is expected if the BLAS implementation uses the $O(n^3)$ matrix multiplication algorithm. Indeed Fig 15.9 shows that the slowest of our runs are on two star-shaped trees, where we need to multiply two $n \times n$ matrices and where the time-complexity of the matrix multiplication algorithm is most important. However, in most cases our algorithm seems to be close to quadratic execution time, even though it apparently uses an asymptotically slow matrix multiplication algorithm.

15.3 Conclusion

We have derived, implemented and tested a new algorithm for computing the quartet distance. In theory our algorithm has execution time $O(n^{\alpha+2})$, where $\alpha = \frac{\omega - 1}{2}$. With current knowledge of matrix multiplication this is $O(n^{2.688})$. If an algorithm for matrix multiplication in time $O(n^2)$ is found this would make
15.4 Availability

The software is available from http://www.birc.au.dk/software/qdist. It has been tested on Ubuntu Linux and Mac OS X.

15.5 Competing interests

The authors declare that they have no competing interests.
15.6 Author’s Contributions

JN, TM and CP developed the algorithm. AK implemented the algorithm. AK and JN benchmarked and evaluated the algorithm. JN, TM and CP wrote the paper. All authors read and approved the paper.

15.7 Acknowledgements

We are grateful to Chris Christiansen, Martin Randers and Martin S. Stissing for many fruitful discussions about the quartet distance.
Bibliography


169


