Using Manhattan distance and standard deviation for expressed sequence tag clustering

Dane Kennedy
Supervisor: Scott Hazelhurst

October 25, 2010

Abstract

An explosion of genomic data in recent years has necessitated the novel application of old algorithms and the development of new algorithms in order to process and understand this information. One specific type of data comes in the form of expressed sequence tags (ESTs) which have significant biological importance. They do, however, require a lot of work in the dry lab once they have been created in a wet lab before anything meaningful can be made of the data.

ESTs are short fragments of genomic data that represent the active genes in a sequenced sample. The sequencing process is complex and involves the copying, magnification and then splitting up of long complete sequences. The splitting of each copy is random and by looking for overlaps on the shorter sequences we can then begin to re-assemble the original long sequence. The process of finding the overlaps is called clustering and in theory it groups all the ESTs from a single gene together (where data sets in general contain ESTs from many different genes).

Two new heuristics have been developed in this research that aim to speed up the time taken to do the clustering, which is traditionally an all-against-all comparison (and thus quadratic time complexity). The first heuristic uses a sorted word list of all the words in all the sequences to rapidly identify matching pairs of sequences. Further, the standard deviation between the relative position where words occur in the matching sequences is used to determine the quality of the matches – if a pair of sequences has sufficiently many words in common, and the variation in relative positions of those words is sufficiently low it is then tested by the second heuristic.

The second heuristic uses a distance measure called d^1 which identifies pairs of matching sequences in a very similar way to the d^2 distance measure. d^2 is an accepted distance calculation for calculating the relatedness of ESTs, but d^1 is used as a comparison as it runs in linear time and rules out many of the potential matches from the first heuristic that would also be ruled out by the final d^2 comparison.

The results show that with careful selection of clustering parameters a significant speed-up can be made over various competing clustering techniques with little significant impact on clustering quality.

Acknowledgements

Firstly I would like to thank my supervisor, Scott Hazelhurst, for giving me a great project to work on and being patient with me in times of low-productivity (most times). Thanks also to my sponsors at the National Bioinformatics Network for funding my project as well as educating me in the ways of Bioinformatics. Special thanks to my mother and brother for supporting me in my endevour to be the eternal student. Thanks to Deborah for putting up with me and helping out so much in the final stages of this project. And finally I'd like to thanks Wes and Em for always standing by me and helping me to exercise my TABing ability on the water and on my Bruce.

Contents

List of Figures

Chapter 1

Introduction

In the last two decades we have seen the amount of biological information generated by researchers grow exponentially. This data covers the entire field of biology, but in the field of molecular biology we have seen the largest growth. There are many categories where this data falls into – be it genomic, proteomic, systems biology, but this research is concerned with an important large volume of data that comes in individually small packages – the expressed sequence tag.

Expressed sequence tags (*ESTs*) are created when *mRNA* is sequenced. This genetic data gives us direct information on the genes that are active within some cell, tissue, organ or organism at some point during its life. Being able to effectively use this information enables researchers to better understand the cells, tissues, organs and organisms which in turn enables a variety of further research to be done, whether it be drug development [Lizotte-Waniewski *et al.* 2000], novel gene discovery [Verdun *et al.* 1998] or even creating phylogenetic trees [Dunn *et al.* 2008]. This research looks at a method to speed up a process that takes raw sequence data from the wet lab, to meaningful data in the hands of the researcher.

This step is called *EST clustering*. When a given sample's mRNA is sequenced, the original strand is cut up into smaller strands and the end result is a large number of short sequence fragments (the ESTs), that correspond to many/all the different genes that were active when the sample was taken. Unfortunately when this is done there is no way to keep track of which fragment belonged to which mRNA or gene. Fortunately when the sequencing is done there are many copies of each mRNA, each of which is cut in a different place and, by finding regions of similarity between the ESTs, we can then group together the ESTs that came from the original strand (clustering). We can then try and rebuild what the original strand looked like (assembly).

The clustering is a relatively simple, yet time-consuming, step in the process that requires every EST to be compared to every other EST to find the regions of similarity. Computationally this takes quadratic time which makes it unfeasible to do on very large data sets without either parallelising the solution or using some kind of heuristic to quickly narrow down potential matches. The algorithm developed in this research uses two such heuristics, the first of which speeds up the clustering process by rapidly identifying potential matches and avoiding any complex computation on pairs of sequences with little chance of matching. The second heuristic is more fine-grained than the first heuristic and simply aims at narrowing down the list of candidate matches further.

The rest of this chapter gives a slightly more detailed introduction into the problems of EST clustering and the solution that has been developed. A brief summary of the results is presented and finally the structure of the rest of this document is outlined.

1.1 The problem

More formally, the task of EST clustering can be seen in mathematical terms as creating a graph, where nodes represent ESTs and the presence of edges between vertices indicates some overlap, or match, between them. In the model shown in Figure 1.1 we see that from the original unconnected graph we get three sub-graphs created by connected nodes, which are then considered the clusters.

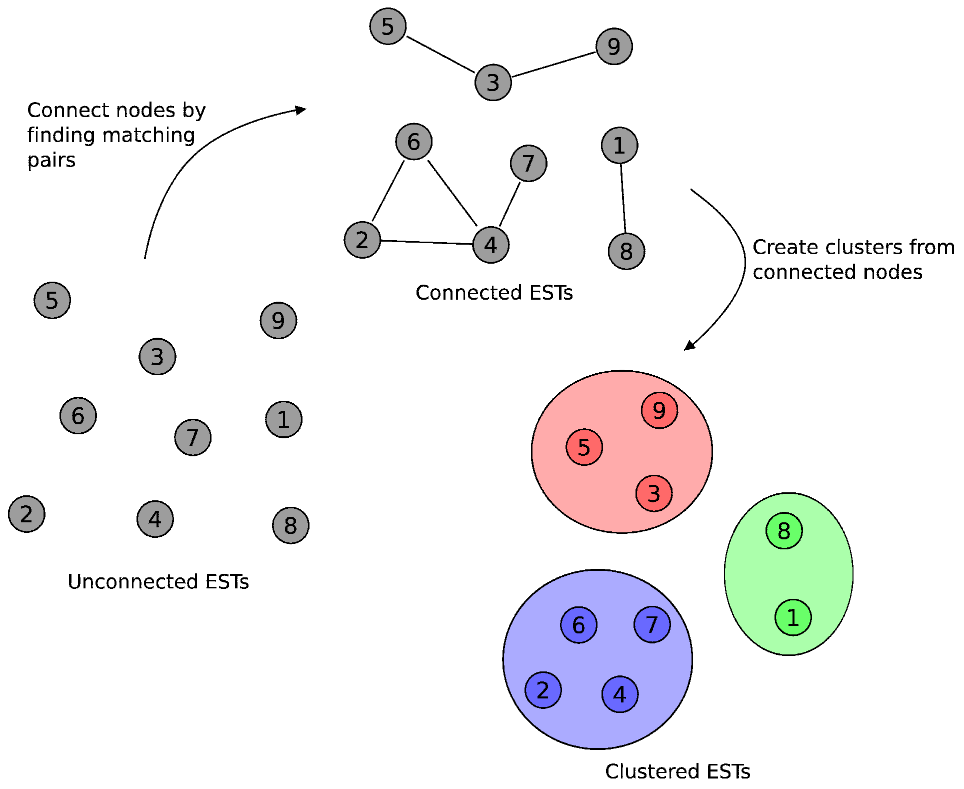

Connect nodes by finding matching pairs
Connected ESTs
Create clusters from connected nodes
Unconnected ESTs
Clustered ESTs

Figure 1.1: Creating clusters by finding the connected components in the graph.

The problem that this research tackles is finding a way to discover a quick and efficient method of identifying which ESTs are connected to which other ESTs. We are not concerned with finding every single edge in the graph, since the final output is simply a list of lists that describe which ESTs belong in a cluster. This does mean that if an EST, α, is found to match another EST, β, which already belongs to a larger cluster, γ, then it is unnecessary to compare α to the rest of the ESTs in γ. However, it is still necessary to compare it to the rest of the ESTs in the data set. In the worst case this means that we need to compare every EST against every other EST – which takes quadratic time (in the number of ESTs) and thus is impractical for very large data sets.

Another complicating factor for EST clustering occurs when the data set is created. EST creation uses a high-throughput sequencing technique that puts an emphasis on quantity rather than quality. As such, the data that is produced tends to have a relatively high error rate of around 2% [Schuler 1997]. The implication of this is that we cannot simply look for exact matches between two sequences, but rather we look for approximate matches. There are two main categories of algorithm for doing this – alignment based and alignment free. Alignment based comparisons generally consider edit distance as the matching criterion – thaey looks at how many substitutions, deletions and insertions are required to convert one sequence into another. Alignment free based comparisons usually consider word frequencies – the occurrence of all the words of some given length are considered and compared in some way. The advantage of alignment based methods is that they use well researched methods of sequence/string comparisons and have been efficiently implemented for many years. Unfortunately they can miss identifying related sequences that have been altered due to shuffling and recombination [Vinga and Almeida 2003]. On the other hand, since alignment free algorithms tend to be based on word frequencies, they are better at identifying sequences which have been altered due to these events.

1.2 The solution

This research has looked at the development of two heuristics which are used to rapidly identify candidate sequences for d^2 based clustering. The d^2 distance measure is an alignment free comparison that was first used by Burke *et al.* [1999] for EST clustering and has since been included in the StackPACK set of tools [Christoffels *et al.* 2001] and *wcd* [Hazelhurst 2003], both of which are often used for EST clustering. The basic procedure is depicted in Figure 1.2 and is described below.

The first heuristic, called sd_heuristic, works by assuming that shorter sub-sequences, or words, within an EST are less likely to occur in non-matching pairs than they are in matching pairs. It takes advantage of this by creating a sorted list of all the words of some given length in all the sequences and then by going through the sequences one-by-one it is simple to check which sequences have common words. This then avoids computationally expensive comparisons between sequences with no, or few, words in common. If the number of common words between sequences exceeds a certain threshold, it then uses a second mechanism to further narrow down the predicted matches. This is done by checking if the relative positions of the matching words in the candidate sequences are similar. The idea behind this is illustrated in Figure 1.3 and says that if the two sequences are similar, then the matching words will occur in a similar order. In order to achieve this a list of the offsets is stored and if the standard deviation of these offsets is less than some value then it is considered a match.

While the first heuristic quickly identifies possible candidate matches, it is not very stringent. The second heuristic, called the d^1 heuristic, narrows down the positive results by calculating the d^1 distance between the sequences. This is more strict than the first heuristic, but less strict than the

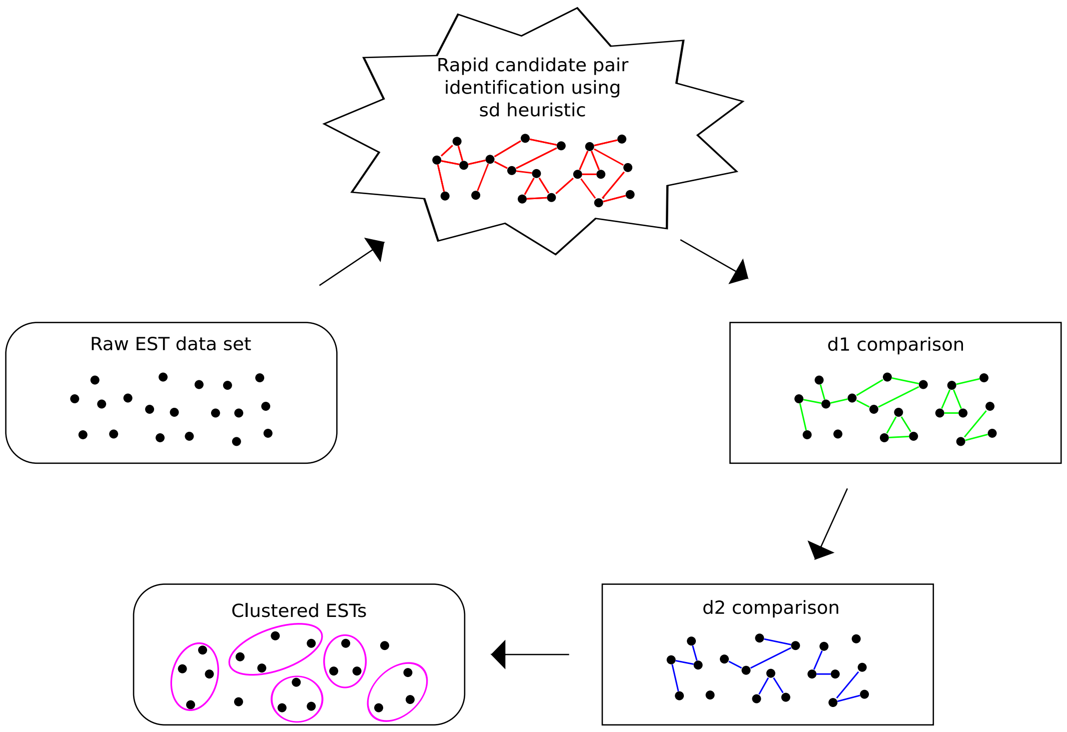


Figure 1.2: The solution for EST clustering in this research takes a data set of EST sequences and rapidly identifies candidate matches using word frequencies as well as positional information in the *sd heuristic*. It then narrows down the candidates using a more stringent sequence comparison called d^1. Finally the matches are checked using the d^2 comparison and if they are sufficiently similar they are clustered together.

final d^2 comparison. The d^1 comparison compares all the windows in one sequence against the entire second sequence and is able to run in linear time. Furthermore it is a good approximation to d^2 since it always predicts the correct positive matches, while still filtering out many of the false positives predicted by the first heuristic.

Results

A number of tests were carried out to analyse the performance of the developed algorithm. They looked at each individual level of the algorithm (the sd heuristic, the d^1 heuristic, overall performance) and considered the accuracy of the clustering as well as the scaling performance. Various data sets were used, including an artificial data set and many real-world data sets. It was found that in most cases the accuracy of the clustering was very good with a Jaccard Index[1] over 95%. Furthermore, both the memory and computational scaling characteristics were found to be sub-quadratic. The memory requirements of the algorithm is an issue and a proposed solution is discussed in Section 5. Computationally it was found that in most of the cases it outperformed the two other clustering

[1]a commonly used measure for comparing EST clusters

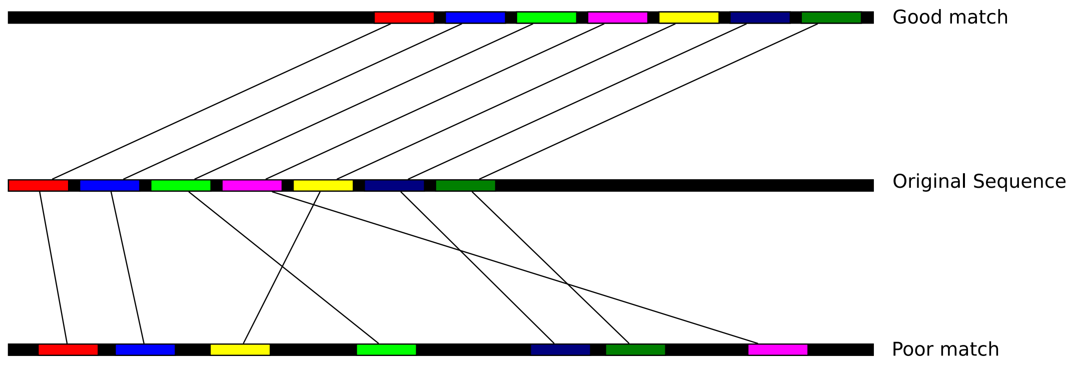


Figure 1.3: This shows an example sequence that has lots of words in common with two other sequences. While they have exactly the same number of common words, in one case it is a poor match since the matches do not occur in the same relative positions, whereas in the other it is a good match since the relative position of the words is consistent.

methods that were considered and parallelisation techniques are proposed in Section 5.

1.3 Contribution of the research

A new method of EST clustering has been developed that takes less time to compute than existing sequential solutions, yet maintains a high standard of quality. This paves the way for dry lab biologists to more rapidly analyse their EST data and allowing for a smoother work-flow. It has demonstrated the usefulness of two new heuristics that rapidly identify candidate matches for EST clustering and could potentially be applied in similar problems.

1.4 Structure of the document

This document is structured as follows:

- Chapter 1 has introduced the subject at a high level and gives a broad outline of what the research covers and how it aims to solve the problem.
- Chapter 2 covers all the background material related to this research. It puts the research in context by relating the biological aspects surrounding ESTs. Competing technologies are compared and contrasted to one another to identify weak and strong points. research hypotheses, and analyses how testing is done from both a theoretical and practical point of view.
- Chapter 3 presents the algorithm and heuristics that were developed and discusses the practical implementation of the algorithm as well as a theoretical analysis of its performance.
- Chapter 4 presents and discusses the empirical results of this research. The results explore the parameter space of the sd_heuristic and shows how the various options affect the overall

clustering. Also discussed is the influence of other factors such as the nature of the data set and the effect of “cleaning” the data before clustering.

- Chapter 5 is a summary of what the research has accomplished and what still remains to be done.

Chapter 2

Background

Bioinformatics, the merger between information science and biology, is a rapidly growing field. This joining of two distinct sciences has come about because of the explosion of information being produced in biological labs – especially those involved in genetic research. The massive amount of data that is output from institutions around the world has created a situation where it is not only impossible to analyse this information by hand, it is becoming increasingly difficult for computers to keep up with the demand. So there is a requirement for more accurate, faster, cheaper ways of analysing the data.

The data that is of interest in this research is expressed sequence tag (or EST) data. ESTs represent small fragments of genes that code for proteins in a cell. These sub-sequences of genes are of interest since they can be used to identify all of the *active* genes in a cell. While each cell has its own complete copy of the DNA, and thus a copy of each gene, it does not need to produce all the possible proteins encoded in it and so only some of the genes are active. Since an individual EST only represents a small portion of a gene they are not of much use on their own, but when they are grouped together then the original gene is easier to identify. The grouping of ESTs together is known as EST clustering and is the process that this research looks at.

There are many methods of clustering EST data, but they are unfortunately all computationally expensive. This research considers an all-against-all approach that reduces the overall time to cluster ESTs. The theoretical worst-case performance of the algorithm is still as bad as any of the existing algorithms but with real world data the performance is often superior.

This chapter begins by introducing much of the background material relevant to this research. It begins by putting the project into its biological context and showing its importance to the field. Finally various clustering algorithms are introduced, with special attention paid to d^2 which is one of the more important algorithms relating to this research.

2.1 Expressed Sequence Tags

Since the discovery of the structure and role of DNA in the 1950s by James D. Watson and Francis Crick, the field of molecular biology has resulted in large amounts of research into the behaviour of DNA, RNA and proteins in a cell. DNA contains all the information necessary for a living cell, or organism, to function and replicate. Much of this information is in the form of genes, which are relatively short regions within the long DNA molecule that most famously code for proteins (the basic functional units within cells).

This Section describes the process of creating proteins and ESTs from the genes within a cell. Firstly the fundamental molecules involved are discussed and then the processes of creating proteins and ESTs are outlined.

2.1.1 From DNA to RNA to protein

It is beyond the scope of this report to give a detailed description of all the molecules and processes involved in protein synthesis so a very simplified outline is given here. For more detailed information a good introduction can be found in Hunter [1993].

The molecules

DNA (or deoxyribonucleic acid) is a polymer made up of pairs of complementary molecules called nucleotides (often referred to as base pairs) that are joined by a hydrogen bond. The nucleotides can be one of four different molecules: adenine; guanine; cytosine; and thymine, where adenine bonds with thymine, and guanine bonds with cytosine. The reasons for complementary strands are: to make the molecule more stable, which is desirable as any changes in the molecule can result in mutations which are, more often than not, harmful to the cell or organism; and for DNA replication during cell division as each strand can be copied once to make two complete copies. An organism's DNA contains anywhere from a few thousand (the virus *Phage X* has 5 386) to many billions (the amoeba *Amoeba dubia* has 670 000 000 000) of nucleotides, and within that there may be anywhere from hundreds to tens of thousands of genes. However, in many organisms, not all DNA codes for genes and there are often large non-coding regions. Even within a gene in the DNA there are coding regions (exons) and non-coding regions (introns) which are ignored during actual protein production. Figure **??** shows the relationshop between a complete strand of DNA, or chromosome, the genes within it and the coding and non-coding regions within the gene.

The second long stranded genetic molecule is RNA (ribonucleic acid). RNA is structurally very similar to DNA except that it is single stranded and Thymine is replaced by Uracil. RNA is transcribed from DNA and represents a single gene, whereas DNA contains many genes.

Proteins are organic compounds comprised of a long chain of amino acids which folds in on itself to create complex structures. Protein is a vital structural component in all organisms and is involved in almost all biological processes, including (but not limited to): acting as enzymes (which catalyse

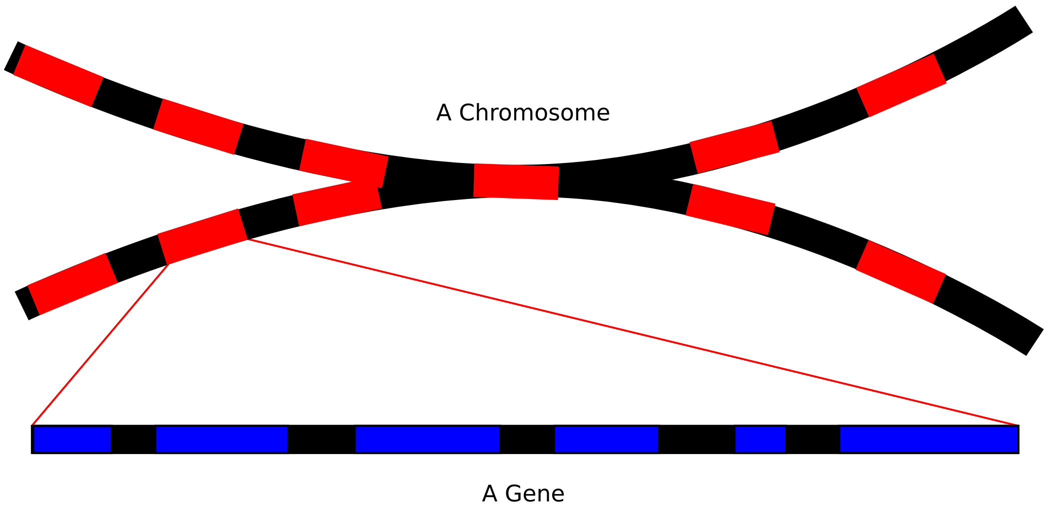


Figure 2.1: This represents a chromosome or a piece of DNA as well as a gene found within the chromosome. The red shows the genes within the chromosome; the blue shows the exons within gene; and the black represents the non-coding parts.

chemical reactions); structural roles (e.g. muscles and connective tissue); involved in the immune system in the form of antibodies. Figure 2.2 shows the relationship between DNA, RNA and protein.

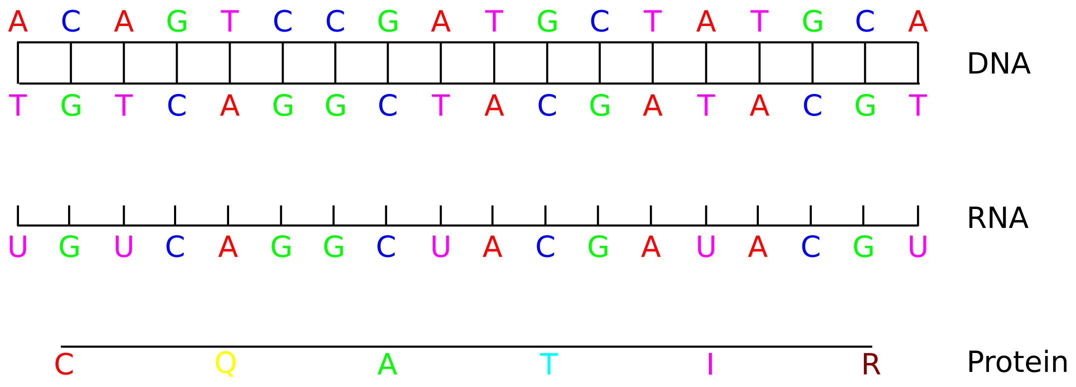

A C A G T C C G A T G C T A T G C A
DNA
T G T C A G G C T A C G A T A C G T
RNA
U G U C A G G C U A C G A U A C G U
C Q A T I R
Protein

Figure 2.2: This figure shows the three polymers considered as well as their constituent parts. First there is DNA made up of nucleotides, then the RNA made up of it's nucleotides (with (U)racil replacing (T)hymine) and finally protein made from it's amino acids.

Protein synthesis

The mechanism for creating proteins is shown in Figure 2.3. The process begins with a gene being transcribed from DNA by RNA polymerase which creates precursor messenger RNA (mRNA). Transcription makes a complementary copy of the DNA and produces all the genetic data in the form of an interrupted gene – a gene containing both introns and exons. Only the exons are necessary for the final protein synthesis so the introns are removed in a process called splicing, with mRNA being the final product. Note that there are alternative splice forms of the mRNA where different regions within a given gene are sliced in or out.

Finally the protein is created by translating the mRNA into a chain of amino acids which folds and gives the protein its final shape. Translation is done by an organelle called a ribosome which takes each successive group of three nucleotides in the mRNA, called codons, finding the associated amino acid and joining it to the chain. Once the amino acid chain is complete, it folds in on itself to form a secondary structure.

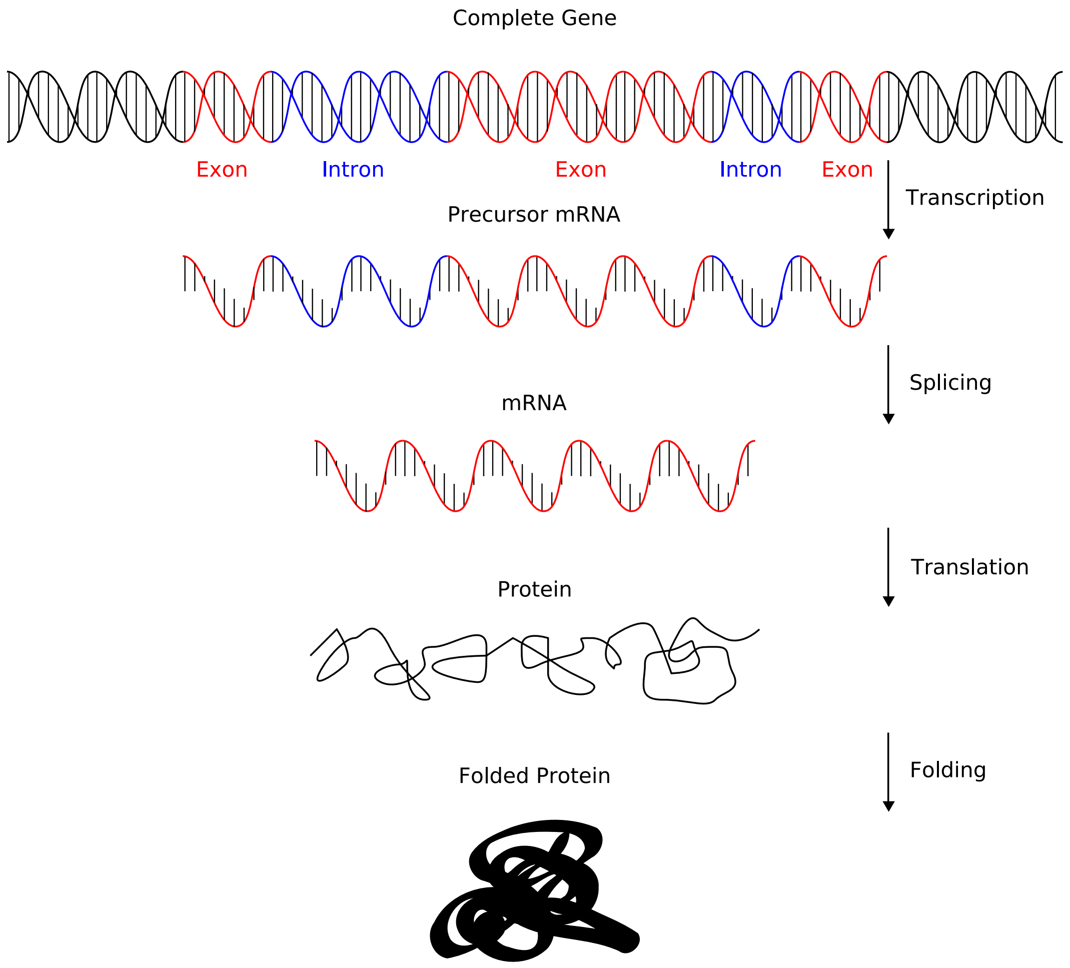


Figure 2.3: This is an illustration of protein synthesis. First a gene is transcribed from the DNA to create precursor mRNA. The introns are then spliced out to produce the mature mRNA. Finally the mRNA is translated into protein which folds to form a secondary structure.

2.1.2 From RNA to cDNA to ESTs

Expressed sequence tags (or ESTs) are the result of sequencing a gene expressed in the form of mRNA. mRNA is unstable and so sequencing is a process that involves extracting the mRNA from the cell and then artificially synthesising a complementary strand of DNA (cDNA) using reverse transcription so that a stable molecule is created. This cDNA sequence is then amplified using a polymerase chain reaction (PCR) and then inserted into a small circular DNA molecule called a plasmid. This plasmid is in turn inserted into a vector organism (such as *E.coli*) which is allowed to grow and multiply – creating copies of: itself; the plasmid; the cDNA clone and ultimately the mRNA sequence [Malde 2004]. This collection of cDNA sequences is known as a cDNA library. Note that mRNA from many different genes has been present from the beginning of the process, and so the cDNA library contains clones from many different genes. The individual cDNA clones can then be extracted out of the plasmid for sequencing since it was inserted in a known place. Random clones are selected from the library and sequenced so that short sequences called ESTs in the range of 100 to 800 nucleotides long are created [Nagaraj *et al.* 2007]. The division of the mRNA is non-deterministic, which means that if the exact same mRNA is sequenced twice, a different set of ESTs will be produced. The process is shown in Figure 2.4.

There are many advantages to using ESTs over regularly sequenced DNA. Firstly an EST shows that the gene is active in a cell. This enables a comparison of gene expression in healthy cells versus unhealthy cells, young cells versus old cells and ultimately this gives valuable information about the healthy cellular life-cycles. Another advantage is that the process of transcription and splicing the DNA into mRNA removes the introns, or non-coding portions of the genes. This is particularly useful as it is then relatively simple to identify the codons and thus identify the sequence of amino acids in a protein. A third advantage is that some ESTs can be used as Sequence Tagged Sites (or STSs). An STS is a relatively short, unique DNA sequence that can be used to identify where in a gene a sequence comes from – a process known as gene mapping.

There are some disadvantages to using EST data. One of the major problems is that the quality of the data tends to be fairly low since each EST is read only once. Sources of errors include contaminants (foreign genetic material in the sample), read errors (random noise, primer interference, or stuttering where a base is repeatedly sequenced when it shouldn't be) and ligation (where unrelated ESTs are bonded together). The other problem is caused by the piecemeal nature of the sequences – rather than having completely sequenced mRNA strands we have many shorter sequences, with no indication of where they come from. Yet another difficulty arises from alternative splicing which occurs when different forms of the mRNA are produced from the precursor mRNA as shown in Figure 2.5. Since the alternative splice forms have different introns and exons, the codons are thus also different meaning that the same gene can code for different proteins.

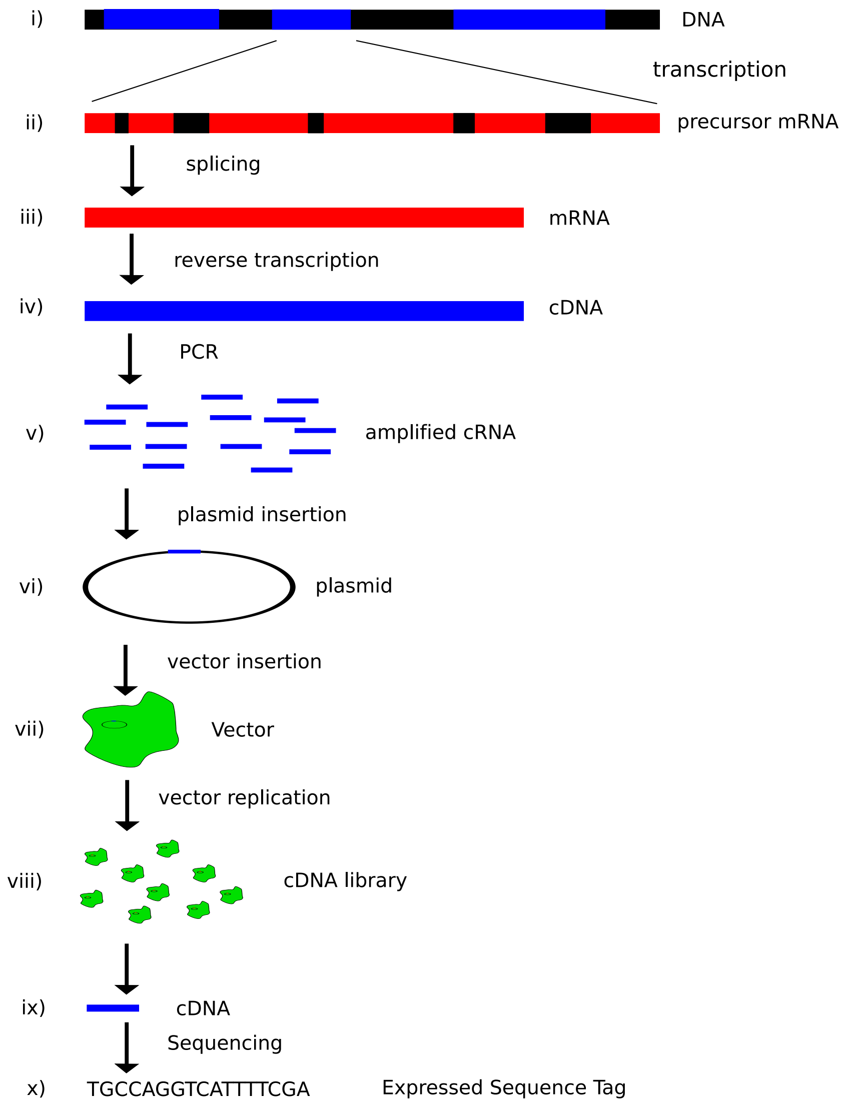


Figure 2.4: This figure shows the process of going from expressing a gene in a cell to the final creation of the expressed sequence tag. Products $i)$ through $iii)$ are natural, while the rest are created artificially.

2.2 EST Clustering

The sequencing step results in many short sub-sequences. The sequences come from many different regions of transcribed DNA and so it is necessary to identify which ESTs belong to the same region before any useful information can be extracted. The process of identifying ESTs from the same gene is called clustering. It is made possible because of the non-deterministic fashion in which the mRNA is split when sequencing. Most genes will be either expressed many times in a given cell or replicated several times in the cloning stage, which means that they will be sequenced several times, resulting in many overlapping ESTs. By looking for these overlaps we can piece the original gene back together as illustrated in Figure 2.6. Note that it is often possible to cluster two ESTs together even if they

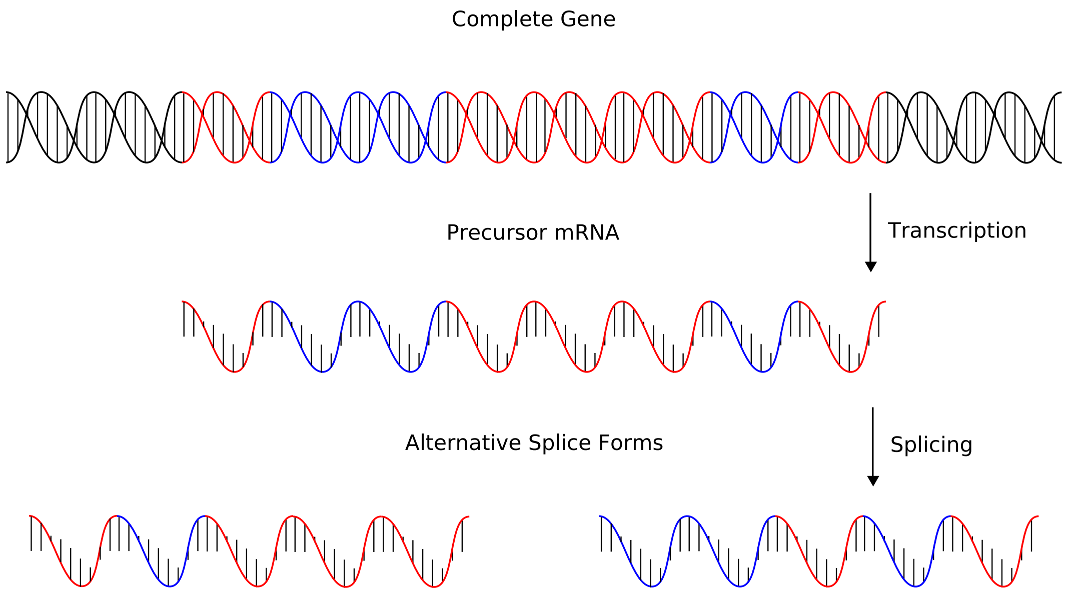


Figure 2.5: This figure illustrates alternative splicing where the precursor mRNA can result in two different mature mRNAs (which in turn code for two different proteins)

share no overlap and this is the case when they have been sequenced from the exact same cDNA clone as (this information is available from the sequencing step).

The process can be looked at as creating a graph where each EST represents a vertex in the graph. If an EST is found to be sufficiently similar to another EST then an edge is created between the two vertices. Clusters are then said to be the connected components in the graph – an example is shown in Figure 2.7. Depending on the method of clustering it might be necessary to check every EST against every other EST. For example in a hierarchical clustering the connected graph is returned which means that every edge in the graph needs to be discovered and thus every EST needs to be compared to every other EST. On the other hand in a non-hierarchical clustering only the connected components are returned, which means that once an EST is found to be in a cluster, it is unnecessary to check it against other ESTs already in the cluster. A comparison of two clusterings of the same data using a hierarchical and a non-hierarchical clustering can be seen in Figure 2.8.

EST clustering has two levels of complexity. On the top level we usually compare every EST against every other EST and this typically results in a complexity $O(n^2)$ where n is the number of sequences. The second level has to do with the sequence comparisons. Depending on the algorithm used, this tends to be between $O(m^2)$ and $O(m)$ where m is the average sequence length. However for accurate sequence comparisons this is usually $O(m^2)$ and the overall complexity is thus $O(m^2n^2)$. An optimization in either level of complexity then results in a performance boost.

More steps can be added into the process and typically clustering programs will be multi-phase algorithms where fast, coarse algorithms are run before the finer-grained solutions. The faster algorithms are used to quickly filter out the obvious mismatches between sequences and then allow faster

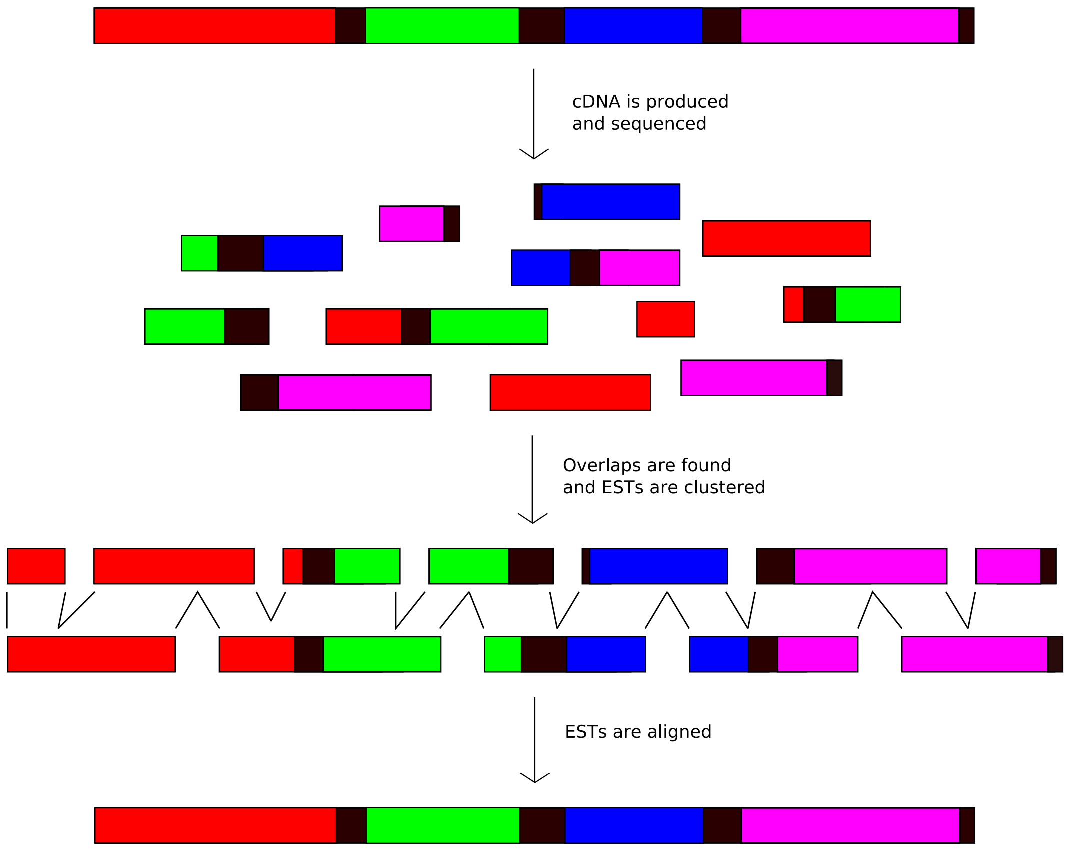


Figure 2.6: This is a simplified version of what happens when mRNA is sequenced, clustered and then aligned. The gene is expressed twice in this case, and thus it is sequenced twice. The two sets of ESTs are produced at the same time and it is not known where each EST comes from. They are then clustered together by looking for overlaps (similarities between a pair of ESTs are indicated as a pair of parallel lines) and then they are aligned and the original gene is found.

algorithms to refine the clustering. An example pre-clustering is shown in Figure 2.9.

There are many methods for sequence comparison, but the general approach is to see if two sequences are sufficiently similar, or have sufficiently similar regions, to be classed together. The reason for looking for "sufficiently similar" regions is that there are typically many errors in the data (as described in Section 2.1.2) which make exact matches unlikely.

Two potential sources of cluster errors are when: ESTs get clustered together that shouldn't be; and ESTs that should be clustered together aren't. An example of when the former error occurs is when ligation has taken place (when two unrelated ESTs have bonded). What can happen then is that the ligated EST overlaps with two separate genes and that one cluster is created instead of two distinct ones. Related ESTs not getting clustered can happen if the data is of low quality. Clustering algorithms usually have parameters that can be set to accommodate for this by making matching more lenient, but of course this also increases the possibility of false positive matches. Statistical tests or human experts can be used to look for these kinds of errors.

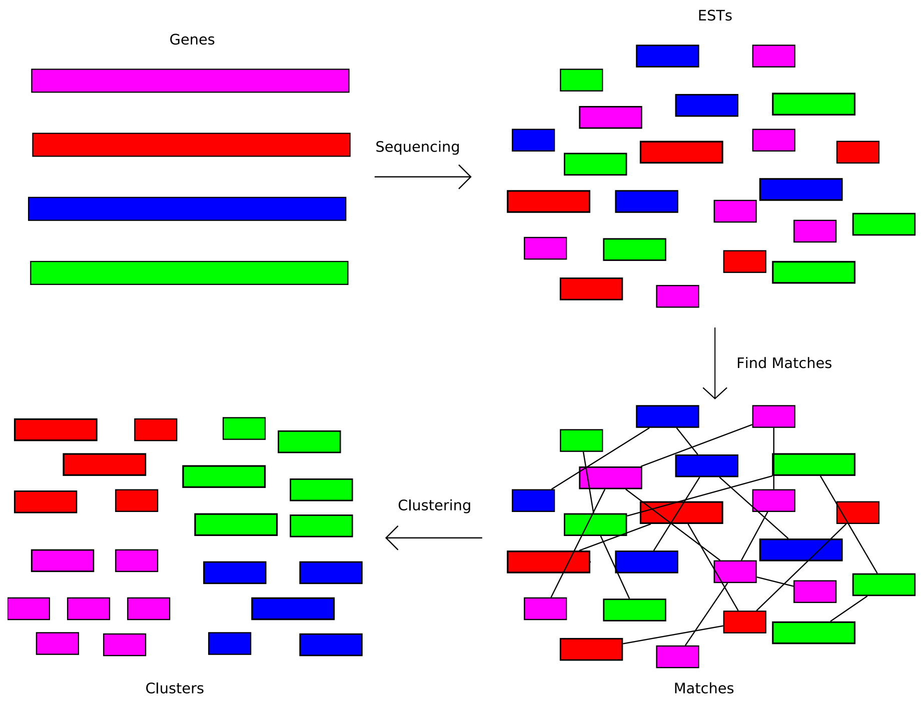


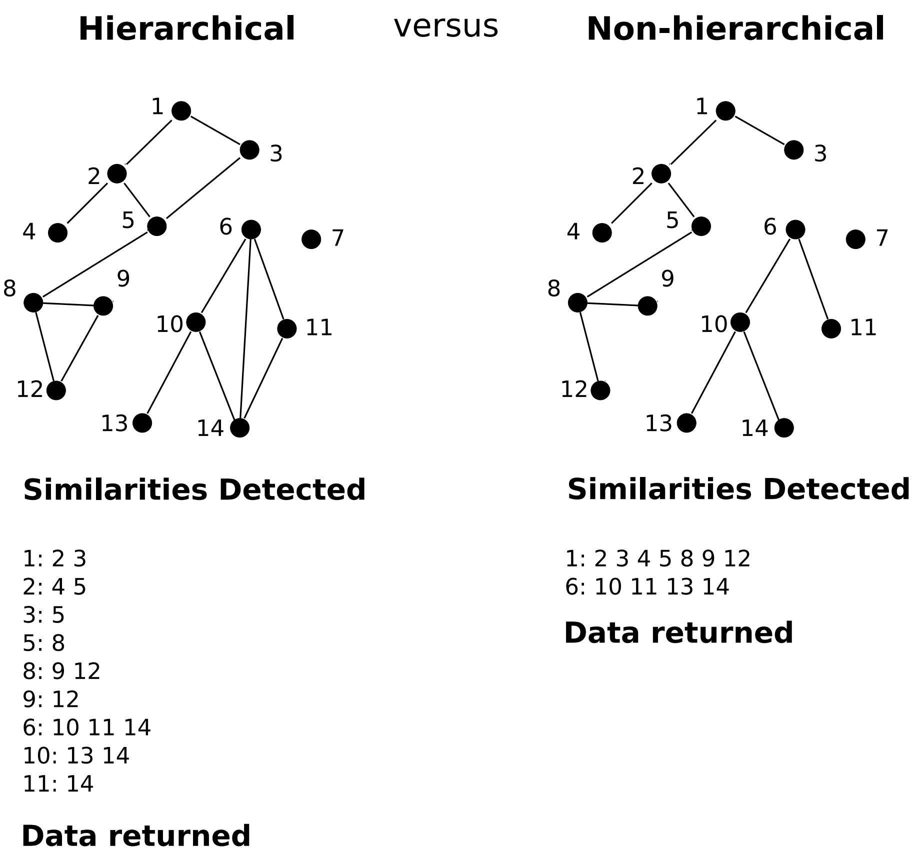
Figure 2.7: An example EST clustering. Firstly the cDNAs are sequenced. Matches are then found between the ESTs. This can be reported as a hierarchical clustering, or the clusters themselves can be returned.

EST Clustering Algorithms

There are many different algorithms used for the sequence comparisons and many different ways to classify them. For most EST clustering algorithms we need to consider two major aspects: first the type of clustering they do; and secondly the way the sequences are compared. Another important aspect to consider for EST clustering is pre-processing of data, which normally involves cleaning the data.

The clustering, or method of grouping ESTs together has several different taxonomies. On one level we can consider whether the clustering is supervised or unsupervised. A supervised clustering considers the EST data set against some other database – usually a genomic database of some organism whose DNA has been sequenced. An unsupervised clustering involves only comparing the ESTs in the data-set against the one another. The advantage of a supervised clustering is that tools such *BLAST* can be used to quickly identify ESTs that belong to the same gene and thus need to be clustered together. Unsupervised clustering is necessary when ESTs have been sequenced from an organism that has not had its genome sequenced. This research deals with an unsupervised EST clustering algorithm.

Furthermore the technique of identifying clusters is another way of classifying the algorithm. Clusters can form hierarchies – that is a graph structure where every node represents an EST and edges between the nodes represent a match between the ESTs. Here we usually start with a graph



```
1: 2 3
2: 4 5
3: 5
5: 8
8: 9 12
9: 12
6: 10 11 14
10: 13 14
11: 14
```

```
1: 2 3 4 5 8 9 12
6: 10 11 13 14
```

Figure 2.8: An example of a hierarchical versus a non-hierarchical clustering. It illustrates the differing amount of information that needs to be discovered and shows that while a hierarchical clustering gives a more complete picture of the similarities between the sequences, it also requires more work.

with no edges and all subsequent matches between sequences are stored as edges in the graph. On the other hand a single linkage algorithm can be used (and is in this research) and this is more like using sets than graphs. Here a cluster is represented only by the ESTs within it, and we are not concerned with the linkage details. In this approach all ESTs are initially considered as clusters on their own (known as singletons). If a pair of sequences match, then the two clusters containing the ESTs are merged.

2.2.1 Sequence Comparison

Sequence comparison looks at how we measure the similarity between two given ESTs. There are two main classes: alignment-based and alignment-free comparisons[1]. Alignment-based algorithms usually look at the edit distance between two sequences – that is the weighted cost of changing the one sequence into the other using the following operations: insertions (adding bases into the sequence); deletions (removing bases from the sequence); and substitutions (changing one base into another).

[1]A comprehensive summary of alignment-free sequence comparisons can be found in Vinga and Almeida [2003].

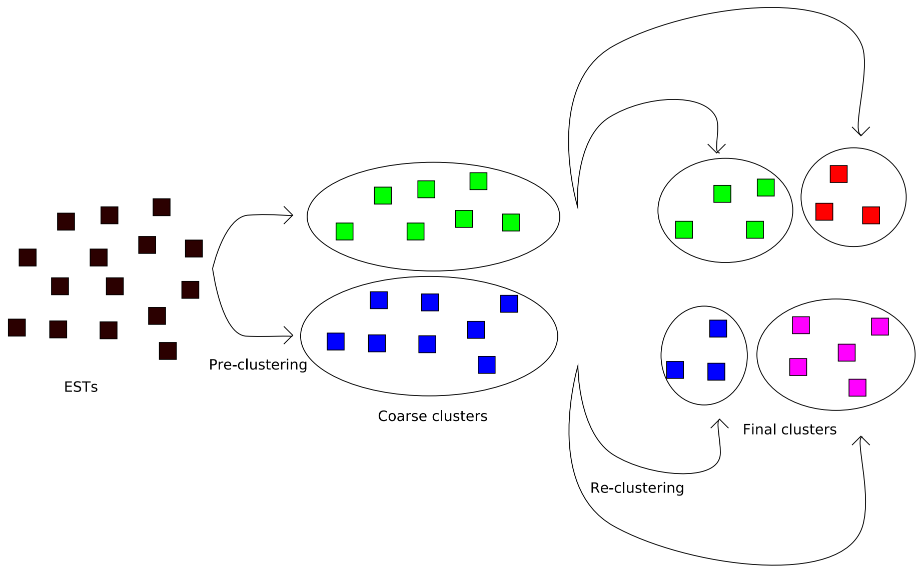


Figure 2.9: ESTs are first pre-clustered with a faster heuristic that quickly generates a coarse clustering. This is then fine tuned by more fine-grained algorithms. ESTs within the pre-clusters do not need to be considered against ESTs not in their cluster.

Using this alignment we can very easily assign a score based on how much the two sequences overlap and how similar the overlaps are. This usually gives good quality clustering, but is computationally expensive ($O(n^2)$).

On the other hand we have alignment-free sequence comparison and this usually takes the form of a comparison of the word-frequencies between two sequences. This is done by counting all the words in each sequence and essentially comparing their histograms. The exact method of comparisons varies and can take many forms including: simply counting the number of words they have in common; finding the sum of the differences in the word counts; finding the sum of the differences squared in the word counts; and many others. Alignment-free comparisons have the advantage that they often have a better computational complexity, but they can also be used to filter out poor quality data. Unfortunately they can also result in two completely unrelated sequences being clustered together if they have similar word counts.

d^2 distance measure

Burke *et al.* [1999] developed a clustering algorithm based on the d^2 distance function. The d^2 distance function gives a similarity score between sequences by comparing the frequencies of words within pairs of sequences . Mathematically this can be represented as:

Given:

The alphabet $\Lambda = \{A, C, G, T\}$
Two sequences of letters from Λ, x and y

Let:

$c_x(w)$ denote the number of times that a word w (also a sequence of letters from Λ) occurs in sequence x

Then:

$d_k^2(x, y) = \sum_{|w_i|=k}(c_x(w_i) - c_y(w_i))^2$ gives the sum of the square of the differences for all of the occurrences of all possible words w_i, of length k, in sequences x and y. So, the smaller the value, the more similar the two sequences are.

More generally:

$d^2(x, y) = \sum_{k=l}^{L} d_k^2(x, y)$ gives the d_k^2 scores for a range of word lengths.

For EST clustering the d^2 score usually refers to a fixed word length k and thus d^2 will, in fact, refer to d_k^2, where k is typically a value between 6 and 8. Also, for EST clustering it is not necessary, or even desirable, to compare entire sequences to each other. So smaller sub-sequences, or windows, are usually compared, with the most similar pair of windows being the most significant result of the sequence comparison. Thus we define:

$\widehat{d_k^2}(x, y, r) = \min\{d^2(u, v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}$, where u and v are subsequences, size r, of sequences x and y respectively ($\sqsubseteq$ denotes a subsequence).

This is the formula that is typically used for d^2 based EST clustering.

Smith-Waterman

The Smith-Waterman algorithm [Smith and Waterman 1981] is a sensitive method for calculating local alignments – that is regions of similarity between sequences. It is a dynamic-programming solution that works by producing a matrix where the elements in the matrix represent the edit-distances between the segments of the two matrices. The edit-distance represents the number of "events" required to convert one sequence into another. Events are operations such as substitutions (where one character is changed into another) as well as additions and deletions (where characters are added or removed from the sequence). An exact match will give an edit distance of zero, and inexact matches are calculated based on the number of events. Penalties for substitutions, additions and deletions vary according to the application.

FASTA

Pearson and Lipman [1988] developed a local alignment tool called *FASTA* that was popular before *BLAST* became the dominant tool. It was an extension of FASTP which was developed by Lipman and Pearson [1985] for doing protein similarity searches. It does a heuristic search for the pattern of word matches between sequences. Conceptually this can be seen as looking for diagonals on a scatter plot where the x-axis represents the words in one sequence, and the y-axis represents the words in the other sequence. A diagonal with a negative gradient represents a matching region, and the length of the line indicates the strength of the match. The algorithm takes into account small regions of dissimilarity by joining diagonals that are close to one another. If two sequences are found to have a sufficiently large region of similarity, then a more accurate alignment can be performed using Smith-Waterman like algorithms.

BLAST

The Basic Local Alignment Search Tool [Altschul *et al.* 1990] is also a local alignment tool. It aims to produce similar alignments to the Smith-Waterman algorithm, but at much lower computational cost. It is a heuristic that does not guarantee the best alignment as Smith-Waterman does, but rather returns an estimated alignment. The method it uses is to find two smaller matching regions in the sequences being compared. This is done via an array of all the possible words of some given length where each entry points to a list of sequences that match that word. Then when looking at the words in a given sequence the matching sequences and positions of those words can be identified. These are then used as a seed whereby the regions of similarity are grown by looking for matching sections to either side of the seeds. The algorithm can take into account short regions of non-matching sequence but once the matching score drops below a certain level the search is ended and the final score computed as the maximum found when growing the seeds.

Needleman and Wunsch

The Needleman-Wunsch algorithm [Needleman and Wuncsh 1970] is similar to the Smith-Waterman algorithm but it performs a global alignment rather than a local alignment – that is it finds the best possible alignment over the entire two sequences rather than just local regions. It was in fact the precursor to the Smith-Waterman algorithm and uses a similar dynamic-programming strategy however the penalties are different which allows one to find the global alignment.

2.2.2 Clustering

In the following subsection we explore various EST clustering techniques and implementations. In terms of this research the most closely related are those that fall under the d^2 *clustering* branch as this research uses the d^2 distance measure as its final sequence comparison.

d^2 clustering

d^2 clustering is based primarily upon the d^2 distance measure discussed on page 17. This distance measure calculates the d^2 score as $d^2_k(x, y, r) = \min\{d^2(u, v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}$ which means we find the minimum d^2 score when all subsequences of length r (typically a window size of 100 is used) are compared between sequences x and y using a word size of k (typically a value of 8 is used). Usually a d^2 score of 40 or less indicates matching sequences.

d2_cluster

The original implementation by Burke *et al.* [1999], called d2_cluster, is undocumented but a naïve implementation would be to calculate the d^2 of every possible pair of sequences. This approach results in a complexity of m^2n^2 where m is the average length of the sequences and n is the total number of sequences. The worst-case performance cannot be improved upon, but certain heuristics and intelligent implementations can make vast improvements to the running time. Examples include *wcd* which is discussed below and Carpenter *et al.* [2002] who present a parallelisation of d2_cluster where the comparisons were distributed on 126 nodes of an SGI Origin 2000 multiprocessor and speedups of around $100x$ were found.

wcd

Hazelhurst *et al.* [2008] implements a set of algorithms for the *wcd* clustering tool that includes an efficient d^2 algorithm and two heuristics. The first heuristic is very fast and compares the algorithm by checking that they have a small number of longer words[2] in common. The second heuristic is similar but looks for more matches while using a shorter word[3].

Once two sequences have passed both heuristics the d^2 score between them is calculated. Hazelhurst [2003] noted that it is unnecessary to recalculate every window's d^2 score from scratch, but rather an adjacent window's score can be calculated by subtracting the amount that the word leaving the window contributes to the d^2 score and then adding in the amount that the new word contributes. This is done in a zig-zag manner that results in the minimum number of calculations being made.

Finally clusters are formed using a disjoint-set data structure which allows non-hierarchical clusters to be formed and members of the clusters to be rapidly identified. The benefit of this is that when considering a pair of sequences it is simple to check if they already belong to the same cluster and thus avoid expensive comparisons (if they do belong to the same cluster).

[2]the current version does an asymmetric comparison of non-overlapping words of length 8
[3]the current version considers all overlapping words of length 6 and requires 65 similarities

Parallelisation of wcd

Parallelisation has been extensively used in optimising d^2 clustering. Ranchod [2005] presents a solution where a cluster of machines are employed. The goal was to organise 100 machines to each handle a small part of the problem, with one machine coordinating them and collecting the results. This was achieved by having the controlling machine broadcast all the data to the machines on the network and tell them which sequences to compare. The computing nodes then did their portion of the clustering and reported the results back to the controller. A super-linear speedup was found where it took 100 machines less than $1/100^{th}$ of the time that a single machine would have taken, which is very surprising since this kind of parallelisation typically shows a sub-linear improvement – often due to the cost of coordination of compute nodes and communication. Ranchod suspects that this is primarily due to fewer cache misses since fewer sequences need to be loaded into any one machine's cache.

FPGA implementation of wcd

FPGAs[4] have been used to implement the d^2 algorithm in Berry [2004]. While the same clustering algorithm is used in both the software and the hardware solutions, the parallel nature of FPGAs allowed for some clever enhancements to be made. Rather than running the 3 stages of the *wcd* algorithm sequentially, it is possible to do them in parallel on the FPGA. The benefit of this is seen primarily in cases when the heuristics pass (indicating a possible match) as the $d2$ calculation has already begun. This research showed that a significant speedup over software solutions was possible most of the time. The software solution was, however, slightly faster on some rare combinations of data. The test samples were divided up into three different cases. The first, and most likely to occur in real data, is where the heuristics fail – indicating a mismatch and no need to calculate the d^2 score. In this case the FPGA solution had a speed-up factor of around 20. The second most commonly occurring case is where the heuristics pass (indicating a possible match), but the d^2 score does not fall below the threshold (indicating a mismatch). This only had a speed-up of around 3.2. The least commonly occurring case is when both the heuristics pass and the d^2 score falls below the threshold (i.e. a match is found). The timing of this varied quite a bit depending on where the matches in the sequences were. In the best case the matches were at the beginning of both sequences. This unlikely case had a speedup factor between 178 and 383. When the matching portions of the sequences were at the end the hardware solution had its worst performance relative to the software solution. This was because the software solution was designed so that it only check the regions where similar word counts occurred in the heuristics. In this case the software implementation had a speedup of approximately 1.5. A significant result was that the functional units of the implementation used a relatively small percentage of the available space on the FPGA, which means that there is the possibility of implementing multiple

[4]An FPGA is a *f*ield *p*rogrammable *g*ate *a*rray which is essentially a customisable chip. While an FPGA typically performs fewer calculations per second than general purpose CPUs they are application specific which means they can be optimised to a given task which often allows for a large amount of parallelisation and efficiency.

functional units on a single FPGA, resulting in a parallelised architecture. A very basic parallelisation is to simply increase the number of d^2 calculations taking place on the FPGA. While this may not scale linearly due to various routing issues on the FPGA, it should still result in good improvements.

PaCE

Kalyanaraman *et al.* [2003] developed a program which does Parallel Clustering of ESTS. They use a generalised suffix trees (GST) in order to find maximal common substrings between ESTs. This is used for rapid candidate pair prediction which has the effect of predicting the most promising pairs early in the process. The tree construction is done at slave nodes and then the result is sorted so that pair prediction based on maximal common substrings can be done. Batches of potential pairs are then sent to slave nodes where they are aligned and the results are then returned to the master node which decides if the clusters containing the pairs should be merged.

HECT

The Hash based EST Clustering Tool developed by Mudhireddy *et al.* [2004] generates a hash table from words within two query sequences – essentially finding all the common words of some fixed length between the sequences. When matches are found the nucleotides to either side of the word are compared and if they match then the nucleotides next to those are compared etc. until some acceptable level of similarity is found. If the pair is found to be similar then the clusters containing the sequences are merged.

CLU

Ptitsyn and Hide [2005] developed single-linkage agglomerative clustering solution that uses a novel all-versus-all word-frequency style sequence comparison. The comparison works by considering all the words in one sequence against all the words in a sliding window in the other sequence. The number of identical words in each window is recorded to produce a vector which then describes the local similarity of the windows to the target sequence as a whole. This vector is then multiplied by a weighting factor which results in single similarity score which can be compared to a pre-defined threshold to determine where the clusters containing the ESTs should be merged.

ESTate

The ESTate program [Slater 2000] is a word frequency based algorithm that is closely related to this research. It uses an all-versus-all algorithm of counting the number of matching words in all the sequences. It uses a novel approach of a virtual finite state machine to calculate the occurrences of all possible words of length k in all the sequences. A similarity score is calculated by summing the products of the word counts between the sequences, i.e. if the word “ATTC” occurred three times in the first sequence and twice in the other, then the contribution of that word to the total is $2 \times 3 = 6$. The scores for all the ESTs are then calculated and then those that have a score above some threshold (in one experiment he uses a threshold of 700) are then subjected to pairwise alignment. If the sequences align well, then they are clustered together. Sequences already in the same cluster do not need to be compared again. There is a priority queue established whereby the sequences with the highest word count scores are sent to the alignment first. This is done so that ESTs with a high probability of being clustered together are done so first, which promotes early cluster formation. The advantage of this is that the total number of pairwise alignments is lessened because there is a greater chance of sequences already belonging to their final clusters.

TGICL

The TIGR Gene Indices Clustering tools is a pipelined set of programs for EST analysis [Pertea *et al.* 2003]. It takes as input a simple FASTA file and starts off by masking regions of low complexity using an undocumented program called *mdust*. The masked sequences are then indexed and an all-against-all similarity search is performed using *mgblast* which is based on *megablast* [Zhang *et al.* 1999]. If the overlap between two sequences is 40 basepairs (default) and there is a 95% identity then the sequences are clustered together. The pipeline then runs *CAP3* [Huang and Madan 1999] to assemble the sequences.

xsact

Malde *et al.* [2003] present an algorithm based on suffix arrays that can be used for pre-clustering. The algorithm is $O(n \log n)$ and uses linear space (an $O(n)$ algorithm is possible, but it uses n^2 space which is impractical for large data-sets). A suffix array for a set of strings is an ordered array of all the suffixes of all the strings. By looking at the suffixes in the list and finding strings where the first k characters match you can say that the original strings (where the suffixes came from) have regions of length k that match exactly. The original sequences can then be grouped together for more accurate clustering later on. With a "good" choice of k there will hopefully be several smaller preclusters to analyse – meaning the algorithm goes from $O(n^2)$ to $O(n_0^2 + n_1^2 + ... + n_j^2)$, where $n_0, n_1, ..., n_j$ are the sizes of the j new preclusters. In the worst case scenario it will still be $O(n^2)$ since all the original sequences will be in the precluster, i.e. $j = 0 \leftrightarrow n_j = n$.

UICluster

The UICluster program developed by Pedretti [2001] uses a greedy algorithm that does not perform an all-versus-all comparison of the ESTs. Rather it starts off with none of the ESTs belonging to a cluster. Then each EST is considered in turn against known clusters, if it does not match any of the known clusters, then it is said to be the start of its own cluster. So the very first EST considered is placed in its own cluster. The second EST is considered against it. If they are found to match, then it is placed in the cluster, if it is not, then it becomes its own cluster. When an EST is considered to membership of a cluster it is only considered against a single representative EST of the cluster.

The representative can be chosen in a number of ways – it can simply be the first EST added to the cluster, or it can be the longest EST added to the cluster, or it can be a virtual representative. The virtual representatives are actually created by the overlaps of the original representative and the ESTs that are clustered with it. If an EST is compared to the virtual representative and found to belong to the EST then any portions that overlap with the ends of the representative are added on so that the representative grows.

The risk of using a greedy algorithm such as this one is a poorer overall clustering – typically it will result in under-clustering as matches will be missed (Figure 2.10). The advantage is that they run quickly, and that the results are relatively good. Just as loose/over-clustered data can be used to speed up clustering, so too can under-clustered data. This can be achieved by telling a clustering solution which ESTs already belong to clusters and don't need to be considered against each other, but do need to be considered against the rest (as opposed to an over-clustering which says that the individual ESTs within the cluster do not need to be considered against ESTs outside of the cluster).

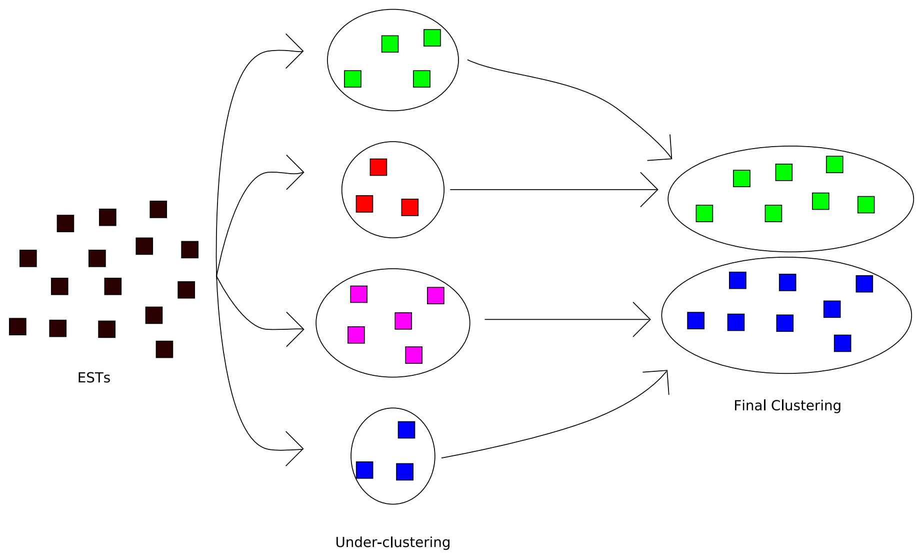


Figure 2.10: In this example the ESTs have been under-clustered, either because of a strict or a greedy algorithm. Once the initial clusters are formed the ESTs only need to be considered against the ESTs in the other three clusters.

Assembly Tools

As noted before, the process of taking the raw EST data from the machine to yielding useful information for a biologist usually involves clustering the data before assembling since this reduces overall computation time. However, it is worth noting that in some cases it is unnecessary to perform clustering before assembly of sequences. This usually happens with small datasets where the cost of trying to align unrelated sequences will not impact too greatly on the overall computation time. There are a large number of tools available ranging from: PHRAP [Green 1999] and CAP3 Huang and Madan

[1999] which are used for assembling shotgun DNA sequences and can take base calling quality into account; to Velvet [Zerbino and Birney 2008] which is usually used on much shorter sequences – usually in the region of 25 to 50 base pairs.

2.2.3 Cleaning the data

It is important to have the best quality data possible when doing EST clustering in order to avoid clustering items together that should not be clustered together. There are many sources of errors – many kinds of read errors caused by technical aspects or from contaminants. The various software solutions listed here aim to remove these two kinds of erroneous data so that the best possible clustering can be performed.

Lucy

Lucy is a tool developed by Chou and Holmes [2001] that removes vector splice sites and contaminants. It has been used extensively at The Institute for Genomic Research (TIGR) and is multi-phase program that looks at removing low quality data by analysing the quality of each base read in a sequence and then filtering out the vector splice sites. It uses some other programs (phred [Ewing *et al.* 1998; Ewing and Green 1998] as well as TraceTuner [Paracel 2000]) to identify the quality of the data. Once low quality data has been identified the splice sites are then trimmed out of the data. This is done using two files: one containing the entire vector sequence so that it can easily be identified in the sequence; and the other contains the upstream and downstream sequences at the splice site. The final step is to remove contaminant data – either from the vector or from unrelated sources.

RepeatMasker

RepeatMasker is a program that screens DNA sequences for interspersed repeats and low complexity DNA sequences [Smit *et al.* 2007; Jerka *et al.* 2005]. It does this by using cross_match which is an efficient implementation of the Smith-Waterman-Gotoh algorithm [Gotoh 1982]. Essentially the algorithm compares the sequences to a curated database of known repeats and if there is a matching region it is masked out.

dust

dust is an undocumented program used to mask low complexity repeats that is widely used as it is part of TGICL's clustering pipeline [Pertea *et al.* 2003]. It looks at each individual sequence in a dataset and locates regions of low complexity where shorter subsequence of between $1 - 4$ nucleotides are repeated. These areas are then masked out. A symmetric DUST implementation has been implemented by Morgulis *et al.* [2006] that aims to make the results more reliable since the original implementation is unsymmetric and masks different areas depending on whether the original sequence or its reverse complement is used.

RBR

Where *mdust* considers repeats on a sequence-by-sequence basis, *RBR* looks at repeats over the entire dataset. Malde *et al.* [2006] does this by first counting the word frequencies over the entire dataset. Then each sequence is taken in turn and the word frequencies are again calculated. If the frequency of a particular word in a sequence is significantly above the frequency of that word in the dataset as a whole, then it is masked out.

2.2.4 Comparison of clustering techniques

A direct comparison of the various sequence comparison, clustering and cleaning tools is very problematic as the tools are often part of larger packages and the output is often not immediately comparable. In terms of sequence comparison the alignment based comparisons are generally more accurate, but slower than their word-frequency base counterparts. Since clustering is usually a precursor to EST assembly where alignment is done anyway, the faster (but still quite accurate) word-frequency based methods were used – specifically using the d^2 distance measure since it accurately identifies matching pairs of sequences and fast related heuristics can be developed.

For clustering purposes a non-hierarchical disjoint-set data structure was chosen for identifiying ESTs belonging to the same cluster. As in *wcd* this means needless comparisons can be avoided as it is quick and simple to check if a pair of ESTs are already in the same cluster. A hierarchical clustering where all matching pairs of sequences is unnecessary since this tends to be computed in the assembly stage of making EST's data useful.

2.3 Summary

EST clustering is an important step in making useful information available to biologists from the wealth of EST data produced in labs. It is a computationally intensive task that has been tackled in a variety of ways ranging from simple all-versus-all exhaustive searches or assemblies, to intelligent searches with time-saving heuristics. With the high-throughput techniques being employed to produce EST data, simple clustering techniques are inadequate since they result in unreasonable computation times and thus ways of rapidly identifying EST clusters is required. For this research we primarily look at rapidly identifying candidate matching pairs of sequences. This is done using an alignment-free comparison which identifies pairs of sequences which will potentially match when using a d^2 sequence comparison. This was chosen since d^2 is widely used and has been found to produce biologically reliable clusters.

Chapter 3

Algorithm

This chapter introduces the algorithm that was developed and implemented for this research. Furthermore a complexity analysis is done in Section 3.2 and a justification of the d^1 heuristic is described in Section 3.3.

3.1 Algorithm description

The algorithm is based on word-frequency similarity in sequences similar to those used in the *wcd* clustering system. There are three major functions used when creating the clusters: the first of which aims to rapidly identify candidate matching sequences (step 1.1); the second function attempts to filter out false positives identified in the first step by using the d^1 heuristic which is similar to the final check (step 1.1); the third function is simply the d^2 comparison to find the final fine-grained clustering (step 1.1). Algorithm 1 shows the basic steps involved and these are then elaborated on in algorithms 2 to 6. A complexity analysis can be found in Section 3.2.

The idea behind the algorithm is to avoid as many unnecessary, costly calculations as possible. This is done in two ways: firstly the outer loop does a comparison (called the *sd_heuristic*) that identifies potentially matching sequences. This is done by going through the sequences one-by-one and looking at the words contained within them, consulting a sorted word list that identifies all the sequences that contain a given word. Every time the source sequence word matches another sequence the match is noted, together with the relative positions of the matches. Since words are checked against a sorted list of matching words, any comparison between sequences that contain no matching words are avoided completely. Note that we may not check every word in the source sequence, but rather only look at every i^{th} word. Once the words in the source sequence have been checked, the list of matches is checked and any sequence that matches more than a given number of times is looked at more closely. Firstly the relative positions of the matches between the sequences are compared and if the standard deviation is found to be small they are looked at with the more fine-grained d^1 heuristic.

The d^1 heuristic, which is justified and explained in more detail in Section 3.3.1 works in much the same way as the d^2 comparison in that it checks the word frequencies between two sequences, but it allows for a linear time comparison. This is achieved by calculating the number of common words between all the windows in one sequence versus the entire other sequence where the parameters are defined by those used in the final d^2 comparison. Even though the one sequence is looked at in its entirety we are still guaranteed that any positive match made by d^2 will be made by d^1 (proof in Section 3.3.1). Because d^1 is similar to d^2 it is hoped that the number of false positives are few (and this is found to mostly be the case with empirical testing discussed in Section 4.2.2). Pairs that match this comparison are then finally checked with a d^2 comparison and if they match they are clustered together.

Algorithm 1: The algorithm

```
1  Create word list (Section 3.1.1)
2  forall ESTs $S$ and their reverse complements $S'$ do
3      Find matching pairs using sd_heuristic (Section 3.1.2)
4      if pair matched using sd_heuristic then
5          Compare pair using $d^1$ heuristic (Section 3.1.3)
6          if pair matched using $d^1$ heuristic then
7              Compare pair using $d^2$ comparison (Section 3.1.4)
8              if pair matched using $d^2$ comparison then
9                  Join clusters containing pair (Section 3.1.5)
10             end
11         end
12     end
13 end
```

There are a number of parameters that are used in the algorithm and they are listed here and explained in the algorithms themselves:

Parameter	**Symbol**
sd word size	ω_{sd}
sd window size	β_{sd}
sd threshold	τ_{sd}
sd skip value	λ_{sd}
d^1 word size	ω_{d^1}
d^1 window size	β_{d^1}
d^1 threshold	τ_{d^1}
d^2 word size	ω_{d^2}
d^2 window size	β_{d^2}
d^2 threshold	τ_{d^2}

3.1.1 Set-up phase

This is a simple step that first involves going through each sequence and counting the number of words of a given size. This allows a table to be created (the *WLT* or word list table) which contains for every word in the data set: the sequence number that the word comes from; and the position in the sequence where the word occurs. This list is then sorted so that: firstly they're in alphabetical order for which word each entry points to; and secondly they're in numerical order.

Algorithm 2: Create word list

```
Count words of length $\omega_{sd}$
Create word list table (WLT)
Sort word list table
```

From an implementation point of view there is only one real issue to be considered and that is that the data is often imperfect. There are often portions of the sequences that are marked as bad or unknown and this can be from either the raw data or from a masking process. There are several ways of dealing with this and it is often done by simply substituting in some random data. In this case it is done with the final d^2 since the heuristics implemented here don't consider regions that are marked as bad and it is unlikely that random substitution will create a match between two otherwise unrelated sequences.

3.1.2 sd_heuristic

This heuristic is an asymmetrical word frequency based comparison with an additional check which looks at the relative position of the words in the sequences to ensure that they occur in similar order. This is done by first looking at every λ_{sd}th word in the source sequence and finding every match by looking in the *WLT*. By only comparing every λ_{sd}th word in the source sequence we can make fewer lookups to the WLT and potentially make far fewer matches. The previous β_{sd} matches between any two given pairs of sequences are stored and the standard deviation of the difference between the positions of those matches is calculated. The reason for calculating the standard deviation is that it is possible for two unrelated sequences to have a large number of words in common as illustrated in Figure 1.3. The standard deviation in this case will be larger than when the matching pairs of words occur in relatively consistent positions. A fixed number for β_{sd} was chosen since there are potentially many matches being made here. Consider a case where there are large regions of low complexity data, say a region of 10 As in both sequences. If we were considering every word of length 6 then we would have 25 matches (since there are 5 separate $AAAAAA$s in each sequence matching to each other). In a large data set this could end up using a large amount of memory. Worse these regions of low complexity like this influence the standard deviation in matching sequences since they can match to many different positions.

If at least β_{sd} matches are found and at some point the standard deviation is less than or equal to τ_{sd} then that pair of sequences passes the heuristic.

Algorithm 3: sd_heuristic

```
foreach $\lambda_{sd}$th word $w$ of size $\omega_{sd}$ in sequence $S_i$ do
    Look in WLT and find all sequences containing $w$
    foreach sequence $S_j$ containing word $w$ do
        Create a list that contains the last $\beta_{sd}$ positions of the word in both sequences
        Calculate the standard deviation ($\sigma_j$) of the difference in those positions
        if $\beta_{sd}$ matches are found AND $\sigma_j \leq \tau_{sd}$ then
            Record match between sequence $S_i$ and sequence $S_j$
        end
    end
end
```

Implementation is done by creating several data structures. The first is a *match array* that has an element for every sequence in the data set. Each element in this array can store β_{sd} potential matches and their positions between itself (S_j) and the sequence S_i. Although it is unlikely that any sequence will match to every other sequence (especially if the data that has been properly cleaned) an array was chosen since it allows quick, indexed access to any potential match. Secondly a *match vector* of indexes is created which stores a list of all sequences S_j that are found to have words in common with S_i. This is done so that the entire match array doesn't need to be looked at to find out which sequences S_i matched to. Also it allows the elements in the match array that have been altered to be reset, rather than having to reset everything for each new S_i. A vector was chosen since it is dynamic and its size will scale to the number of matches found.

3.1.3 d^1 heuristic

The d^1 heuristic is based on calculating a d^1 score between two sequences. As shown in Section 3.3.1 we see that given the correct parameters the d^1 comparison is guaranteed to match anything that d^2 matches. In order to do this we require that the sequences that are being compared are at least as long as the window size used in the d^2 calculation and a d^1 threshold (τ_{d^1}) is calculated based on the d^2 parameters, but is independent of the size of the sequences being compared. In particular the d^1 score that is calculated here compares the windows in one sequence against the whole of the second sequence.

Algorithm 4: d^1 heuristic

```
Create a table $\delta_i$ that records the number of all words of size $\omega_{d^1}$ in sequence $S_i$
foreach window $W_j$ of length $\beta_{d^1}$ in sequence $S_j$ do
    Create a table $\delta_j$ that records the number of all words of size $\omega_{d^1}$ in window $W_j$
    Calculate $score = \sum_{w_k} \min\{\delta_i(w_k), \delta_j(w_k)\}$
    if $score < \tau_{d^1}$ then
        Record match between sequence $S_i$ and sequence $S_j$
    end
end
```

The implementation is again fairly straight forward. Since we find all matches between sequence S_i we need only set up the δ_i table once. Once that has been done the d^1 score of the first window in sequence S_j can be calculated. The window is then shifted all the way to the end of the sequence and it is a simple case of removing the words that are leaving the table from the calculation, and adding those that are entering into the window. τ_{d^1} is calculated from the d^2 thresholds and is equal to:

$$\beta_{d^1} - \omega_{d^2} + 1 - \frac{\tau_{d^2}}{2}$$

3.1.4 d^2 calculation

This is the same used in the *wcd* solution outlined in Section 2.2.2.

3.1.5 Clustering

The final output of the algorithm is a list of clusters, where the cluster itself is simply a list of the ESTs. What this means is that it is unnecessary to know a detailed linkage between the ESTs. Thus the same approach that is used in the *wcd* solution is used and that is to employ a disjoint-set data structure[1]. The data structure is composed of n elements where n is the number of ESTs. Each element then has a pointer that points to its parent and this is initially set to itself. If two ESTs are found to belong to the same cluster then the parent of the topmost parent of one of the ESTs is set to be the topmost parent of the other EST. Thus to find out whether two ESTs belong to the same cluster their parents can simply be traced to their topmost level.

Algorithm 5 shows the recursive algorithm for locating a given node's parent and 6 shows how to join two separate clusters.

[1]also known as union-find, and merge-find data structure

Algorithm 5: FindParent

```
if $Sequence\alpha.parent == alpha$ then
    Return $alpha$
end
else
    Return FindParent(Sequence$\alpha$.parent)
end
```

Algorithm 6: Cluster

```
parentA = FindParent(sequenceA)
parentB = FindParent(sequenceB)
parentA.parent = parentB
```

Tarjan [1975] showed that using some simple path compression techniques can make the time complexity of using the union-find data structure effectively constant for all tractable sizes of that data set.

3.2 Algorithm Analysis

The main outline of the algorithm has already been described as:

We see in algorithm 1 that there is a nested structure where a successful match on one level of the algorithm results in the next level being called. Thus we analyse the algorithm from the outside in. The first step creates a sorted word list and so for n sequences of average length m we have a sorted list of mn words and thus $O(mn \log mn)$.

Next the looping aspect is analysed but this is difficult due to the inability to know how often a success on each level will occur and so we need to assume a naïve worst case for every event. What this means is that both heuristics pass for every pair of sequences, but the d^2 comparison fails (a pass on the d^2 comparison would result in a cluster being formed early on, making subsequent comparisons unneccesary). The analysis is thus very simple – the very first step is creating a sorted list of every word in every sequence, i.e. $O(mn \log(mn))$. Then the loop starts with the sd_heuristic and here every sequence matches every other sequence giving a complexity of $O(mn \log mn + n^2 \times (d^1 complexity + d^2 complexity))$. The $mn \log mn$ comes from every word needing to be searched for in the complete word list and then since matches between every pair are found we have $n^2\times$ the inner loop. The inner loop has the two components which in the case of d^1 is linear in the length of the sequence ($O(m)$) and quadratic in the case of the d^2 comparison ($O(m^2)$).

Thus the final complexity is $O(2 \times mn \log mn + n^2(m + m^2))$ which simplifies to $O(m^2n^2)$. This is, however, a fairly pessimistic prediction as it is unlikely that matches like this would happen in the real world – the empirical analysis shown in Section 4.2.3 shows this to be the case.

The memory requirements are dominated by the sd_heuristic (the d^1 and d^2 steps require a static amount of memory). The first requirement is that of storing the sequences and creating the word list. This scales linearly with the number of sequences and their average size, i.e. $O(mn)$. Then since any one sequence can potentially match every other sequence we need to record this, together with the exact positions of the matches. We only store a fixed number of matches between the two sequences so the memory requirement is $O(\beta_{sd}n)$ where β_{sd} is the window size discussed in chapter 3. In the empirical analysis the memory requirement was found to be greater than $O(n)$ and possible explanations are discussed in Section 4.2.3.

3.3 Why d^1

This Section introduces the fundamental mathematical principal behind the d^1 comparison and justifies its use as a heuristic.

3.3.1 d^1 versus d^2

This Section introduces the d^1 distance measure and compares and contrasts it to the d^2 distance measure defined in Section 2.2.2. If two sequences' similarity is calculated using d^1 and d^2, the d^1 score will always show them to be at least as similar as the d^2 score. The result of this is that clusters based on d^1 will always be valid pre-clusters for a d^2 clustering, i.e. anything that should be clustered by d^2 will be clustered by d^1, but the clusterings may need some further refinement.

Introducing d^1

d^1 is a distance measure which is very similar to d^2 in that it is also based on comparing the word frequencies within sequences. The major difference is that d^1 uses the Manhattan/city block type distance rather than the straight line distance that d^2 uses. Here we introduce a number of subtle variants of the d^1 calculation which look at varying word lengths, and varying the window sizes. After the basic definitions another formulation is given which allows the scores to be calculated based only words that occur in both sequences (or windows). This is important for the algorithm as it allows us to ignore words that are not present in both sequences.

d^1 can formally be defined as:

Given:

The alphabet $\Lambda = \{A, C, G, T\}$

Two sequences of letters from Λ, x and y

Let:

$c_x(w)$ denote the number of times that a word w (also a sequence of letters from Λ) occurs in sequence x

Then:

> $d^1(x,y) = \sum |c_x(w_i) - c_y(w_i)|$ gives the sum of the absolute value of the differences between the counts of all the possible words w_i, in sequences x and y. i.e. if a word appears in x or y then the difference between the counts of that word in each sequence is added to the total.
>
> $d^1_k(x,y) = \sum_{|w_i|=k} |c_x(w_i) - c_y(w_i)|$ gives the sum of the absolute value of the difference between the counts of all the possible words w, of length k, in sequences x and y.
>
> $\widehat{d^1}(x,y,r) = min\{d^1(u,v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}$, where u and v are windows, size r, of sequences x and y respectively. This gives the minimum d^1 score of all the subsequences of size r in sequences x and y.
>
> $\widehat{d^1_k}(x,y,r) = \min\{d^1_k(u,v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}$, where u and v are windows, size r, of sequences x and y respectively, and k is the size of the words. This gives the minimum d^1_k score of all the subsequences of size r in sequences x and y.

Now the standard definition of d^1 gives a result in terms of the differences between word counts. We can, in fact, derive a further function that gives the same score, but the function only needs the number of words that are common to both sequences.

$$d^1(x,y) = \sum |c_x(w_i) - c_y(w_i)| \tag{3.1}$$

$$\text{But } |c_x(w_i) - c_y(w_i)| = \begin{cases} 0 & \text{if } w_i \notin x \text{ or } y \\ c_x(w_i) & \text{if } w_i \in x, \notin y \\ c_y(w_i) & \text{if } w_i \in y, \notin x \\ |c_x(w_i) - c_y(w_i)| & \text{if } w_i \in x \text{ and } y \end{cases}$$

Now we can rewrite **(3.1)** as:

$$d^1(x,y) = \sum_w |c_x(w) - c_y(w)| \tag{3.2}$$

$$= \sum_{w_i \in x,y} |c_x(w_i) - c_y(w_i)| + \sum_{w_j \in x, \notin y} c_x(w_j) \tag{3.3}$$

$$+\sum_{w_k \in y, \notin x} c_y(w_k) \tag{3.4}$$

Now, from **(3.3)** we have:

$$\begin{aligned}\sum_{w_j \in x, \notin y} c_x(w_j) &= \text{WC}(x) - \sum_{w_i \in x, y} c_x(w_i) \\ &\quad (\text{Since } \sum c_x(w) = \text{WC}(x))\end{aligned} \tag{3.5}$$

Where WC(z) is simply a count of the total number of words in z. Similarly, from **(3.4)** we have:

$$\sum_{w_k \in y, \notin x} c_y(w_k) = \text{WC}(y) - \sum_{w_i \in x, y} c_y(w_i) \tag{3.6}$$

Again rewriting $d^1(x, y)$ from **(3.2)** using **(3.5)** and **(3.6)** we have:

$$\begin{aligned}d^1(x, y) &= \sum_w |c_x(w) - c_y(w)| \\ &= \sum_{w_i \in x, y} |c_x(w_i) - c_y(w_i)| \\ &\quad + \text{WC}(x) - \sum_{w_i \in x, y} c_x(w_i) \\ &\quad + \text{WC}(y) - \sum_{w_i \in y, x} c_y(w_i) \\ &= \text{WC}(x) + \text{WC}(y) \\ &\quad + \sum_{w_i \in x, y} \Big[|c_x(w_i) - c_y(w_i)| - c_x(w_i) - c_y(w_i) \Big]\end{aligned} \tag{3.7}$$

Now consider the case where $c_x(w_i) \geq c_y(w_i)$. Then we can say:

$$\begin{aligned}|c_x(w_i) - c_y(w_i)| - c_x(w_i) - c_y(w_i) &= c_x(w_i) - c_y(w_i) - c_x(w_i) - c_y(w_i) \\ &= -2c_y(w_i)\end{aligned} \tag{3.8}$$

Similarly, if $c_y(w_i) \geq c_x(w_i)$ we get:

$$|c_x(w_i) - c_y(w_i)| - c_x(w_i) - c_y(w_i) = -2c_x(w_i) \tag{3.9}$$

Finally using **(3.8)** and **(3.9)** we can rewrite **(3.7)** as:

$$d^1(x, y) = \text{WC}(x) + \text{WC}(y)$$

$$-2 \sum_{w_i \in x,y} \Big[\min\{c_x(w_i), c_y(w_i)\}\Big] \tag{3.10}$$

The relevance of equation **(3.10)** is that we can now find the d^1 scores as a function of: the number of words in the sequence; and the words that are common to both sequences, i.e. we can ignore all the words in the sequences that do not occur in both of them.

Now consider d^1_k. This can simply be rewritten as:

$$\begin{aligned} d^1_k(x,y) &= \mathrm{WC}(x) + \mathrm{WC}(y) \\ &\quad -2 \sum_{w_i \in x,y} \Big[\min\{c_x(w_i), c_y(w_i)\}\Big] \end{aligned} \tag{3.11}$$

The only difference between **(3.10)** and **(3.11)** is that there are fewer words to be counted for $d^1_k(x,y)$ since it is limited to a fixed word size. This is important as the d^2 method of clustering also uses fixed word sizes. Finally consider the formula for $\widehat{d^1_k}$ – that is the formula for calculating the d^1 for the windows of the sequence. This can be rewritten as:

$$\begin{aligned} \widehat{d^1_k}(x,y,r) &= \min\{d^1_k(u,v) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\} \\ &= \min\Bigg\{ \mathrm{WC}(u) + \mathrm{WC}(v) \\ &\quad -2 \sum_{w_i \in u,v} \Big[\min\{c_u(w_i), c_v(w_i)\}\Big] : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r \Bigg\} \end{aligned} \tag{3.12}$$

But, we are looking at $\widehat{d^1_k}(x,y,r)$ which means we are limited to word lengths of k and to window sizes of r, so in this case $\mathrm{WC}(z) = r - k + 1$. Putting this into **(3.12)** gives:

$$\begin{aligned} \widehat{d^1_k}(x,y,r) &= \min\Big\{ 2(r-k+1) \\ &\quad -2 \sum_{w_i \in u,v} \Big[\min\{c_u(w_i), c_v(w_i)\Big] : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}\Big\} \\ &= 2(r-k+1) \\ &\quad +\min\Big\{ -2 \sum_{w_i \in u,v} \Big[\min\{c_u(w_i), c_v(w_i)\Big] : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}\Big\} \end{aligned} \tag{3.13}$$

Comparing d^1 and d^2 scores

In this Section we compare the d^1 and d^2 scores that any given sequence will produce. Further we show that any clustering based on d^1 scores will be a superset of one based on d^2 scores – thus showing that d^1 is valid for pre-clustering data (as any sequences clustered by d^2 will be clustered by d^1). First we show that $d^1(x,y) \le d^2(x,y)$:

$$\begin{aligned}
\text{RTP} \quad & d^1(x,y) \le d^2(x,y) : \\
\text{i.e. RTP} \quad & \sum |c_x(w_i) - c_y(w_i)| \le \sum (c_x(w_i) - c_y(w_i))^2
\end{aligned}$$

$$\begin{aligned}
\text{We know} \quad & |a| \le |a|^2 \qquad \forall a \in \mathbb{Z} && (3.14)\\
\text{Also} \quad & c_x(w) \text{and} c_y(w) \in \mathbb{N}_0 \\
& \Rightarrow (c_x(w) - c_y(w)) \in \mathbb{Z} && (3.15)\\
\text{And} \quad & |a|^2 = a^2 \qquad \forall a \in \mathbb{Z} \\
& \Rightarrow |c_x(w) - c_y(w)|^2 = (c_x(w) - c_y(w))^2 && (3.16)
\end{aligned}$$

$$\begin{aligned}
\textbf{(3.14)}\&\textbf{(3.15)} \quad & \Rightarrow |c_x(w) - c_y(w)| \le |c_x(w) - c_y(w)|^2 && (3.17)\\
\textbf{(3.16)}\&\textbf{(3.17)} \quad & \Rightarrow |c_x(w) - c_y(w)| \le (c_x(w) - c_y(w))^2 && (3.18)\\
\text{Summing}\textbf{(3.18)} \quad & \Rightarrow \sum |c_x(w_i) - c_y(w_i)| \le \sum (c_x(w_i) - c_y(w_i))^2 \qquad \square
\end{aligned}$$

Now since $d^1(x,y) \le d^2(x,y)$, and d^2 clustering works by clustering sequences together whose d^2 scores falls beneath a given threshold, it is clear that any two sequences clustered by d^2 will also be clustered by d^1 (assuming the same threshold). In general however, the clusterings will not be the same as d^1 can cluster sequences that would not be clustered by d^2, we can say that $d^2 cluster \subseteq d^1 cluster$ and so d^1 is valid to use for pre-clustering. Similarly we can show that $d^1_k(x,y) \le d^2_k(x,y)$, $\widehat{d^1}(x,y,r) \le \widehat{d^2}(x,y,r)$, and finally $\widehat{d^1_k}(x,y,r) \le \widehat{d^2_k}(x,y,r)$.

Using d^1 for pre-clustering d^2

Now the standard d^2 score that is used for clustering is the $\widehat{d^2_k}$ formula, so it is natural to then use $\widehat{d^1_k}$ for pre-clustering. As discussed in Section 2.2.2 we know that usually a window size of 100 is used together with a word size of between 6 and 8, with 6 being the default. Using the formula derived in **(3.13)** we can say that:

$$\begin{aligned}
\widehat{d^1_6}(x,y,100) &= 2(r-k+1) + \min[-2 \sum_{w_i \in u,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}] \\
&= 190 + \min[-2 \sum_{w_i \in u,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}]
\end{aligned}$$

Now say in a d^2 cluster we use a threshold of $T = 40$ to decide whether or not two sequences should be clustered together, i.e. if $\widehat{d^2_k}(x,y,r) \le T$ then cluster x and y. Using the same threshold in d^1 means that we cluster when:

$$40 \ \ge \ \widehat{d^1_6}(x,y,100)$$

$$\begin{aligned}
40 &\geq 190 + \min[-2 \sum_{w_i \in u,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}] \\
-150 &\geq \min[-2 \sum_{w_i \in u,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}] \\
-150 &\geq -2 \times \min[\sum_{w_i inu,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}] \\
75 &\leq \min[\sum_{w_i inu,v} \min\{c_u(w_i), c_v(w_i) : u \sqsubseteq x, v \sqsubseteq y, |u| = |v| = r\}] \qquad (3.19)
\end{aligned}$$

What inequality **(3.19)** tells us is that any pair of sequences where we can find two windows of size 100 with at least 75 words in common are to be clustered together and are good candidates for d^2 clustering. Unfortunately the number of calculations to look at individual windows will be very similar to the optimised d^2 algorithm discussed in Section 2.2.2. So now consider $d^1_k(x,y)$ where we don't look at individual windows. Here we have:

$$d^1_k(x,y) \quad = \quad \text{WC}(x) + \text{WC}(y) - 2 \sum_{w_i \in x,y} \min\{c_x(w_i), c_y(w_i)\}$$

Now if we use the same term $2(r-k+1)$ that was used in the $\widehat{d^1_k}$ formulation instead of the WC terms we then get a new formula which we call $d^{1\prime}_k(x,y,r)$:

$$d^{1\prime}_k(x,y,r) \quad = \quad 2(r-k+1) - 2 \sum_{w_i \in x,y} \min\{c_x(w_i), c_y(w_i)\}$$

Using the same thresholds gives the formula:

$$\begin{aligned}
d^{1\prime}_6(x,y,100) &\leq 40 \\
40 &\geq 190 - 2 \sum_{w_i \in x,y} \min\{c_x(w_i), c_y(w_i)\} \\
75 &\leq \sum_{w_i \in x,y} \min\{c_x(w_i), c_y(w_i)\} \qquad (3.20)
\end{aligned}$$

$d^{1\prime}_k$ can be used to pre-cluster $\widehat{d^1_k}$ since the RHS of equation **(3.20)** will always be greater than or equal to that in equation **(3.19)** as the $c_x(w_i)$ and $c_y(w_i)$ terms in $d^{1\prime}_k$ will be greater than or equal to the $c_u(w_i)$ and $c_v(w_i)$ terms in $\widehat{d^1_k}$ as u and v are substrings of x and y respectively. So anything clustered using the $\widehat{d^1_k}$ score will also be clustered by $d^{1\prime}_k$. In english that means that with formula 3.20 we can precluster using d^1 on a whole sequence versus whole sequence using the same thresholds as we would use on a window-by-window and be guaranteed not to miss any matches (assuming the sequences are at least as long as the window size). The implication of this is that we can also compare the windows in one sequence against the whole of the other sequence using d^1 as this can be viewed as comparing a whole set of sequences of the window length against the other sequence. This is how the sequence comparison is implemented in Section 3.1.3.

So now what we have is that $d_k^{1\prime}$ pre-clusters for $\widehat{d_k^1}$, which in turn pre-clusters for $\widehat{d_k^2}$. Thus $d_k^{1\prime}$ pre-clusters for $\widehat{d_k^2}$. Next we develop an algorithm that calculates the $d_k^{1\prime}$ score.

3.4 Summary

To conclude the algorithm developed for this research can be divided into three sub-algorithms, each of which aim to identify pairs of matching sequences so that clusters of related sequences can be identified. Each stage is more refined and aims to rapidly identify potential matching pairs of sequences and rapidly build clusters. The first heuristic aims to find matches between sequences by looking for sequences that have similar words in relatively similar positions. Note that by altering the parameters passed to this heuristic it is possible to make it either very coarse (few matches and/or large standard deviation threshold and/or large skip value), or very fine grained (many matches and/or small standard deviation and/or small skip value). The second stage is closely linked to the third in that it uses the d^1 comparison which is very similar to the d^2 comparison. This has a computational advantage over the popular d^2 comparison, while still giving similar results from an analytical perspective. The final stage is the d^2 comparison which is often used as a similarity measure when doing EST clustering. If two matching sequences are found here they are joined using the disjoint-set data structure which is both computationally and memory efficient and allows for fast joining, or clustering, of sets.

Chapter 4

Results

The empirical results of this research are discussed in this chapter while some theoretical results are discussed in Sections 3.3. The algorithm was implemented in C++ since it is a compiled language and allows for optimisation and a fair comparison to other EST clustering solutions which are often implemented in similar languages. Timing of the algorithm was achieved using code which allows it to count the actual number of clock cycles used during execution. In comparison to a timing which merely looks at the clock, it is more accurate since external processes do not influence the final result. The implementation was evaluated in three main parts: sd_heuristic performance (Section 4.2.1), d^1 heuristic performance (Section 4.2.2) and the overall performance (Section 4.2.3). All testing was done on an Intel server with: two Xeon E5345 cpus clocked at 2.33 gigahertz, each with four cores and four megabytes of L2 cache; and four gigabytes of ram. The code is available at `http://code.google.com/p/sdclust/`

4.1 Experimental tools

While overall performance is the major interest of this study, it is important to look at the individual components to analyse where it succeeds or fails and thus experiments were designed to test not only the overall performance, but also the sd_heuristic's performance and the d^1 heuristic's performance. A wide variety of data sets are used since the performance of the algorithm is dependent on not only the parameters passed to the algorithm, but on the specific nature of the data set itself. This Section gives an introduction to the various data sets and cluster comparison methods that are used.

4.1.1 Experimental data

The experimental data used comes from artificial as well as real-world sources. The artificial data was created using ESTSim [Hazelhurst and Bergheim 2003] and was specifically used to examine the scaling performance of the algorithm as the number of ESTs and their specific characteristics (such as length and error rate) are well controlled. Then most of the real-world data came from Genbank and were chosen largely because there were significant numbers (more than 10 000) of ESTs in the database. Partial data-sets were used in most cases that were judged to be of sufficient size. Looking at the results in Section 4.2.3 we see that run times for the data-sets ranged from a few seconds to many hours and this gives a good idea about the range of performance for various data-set sizes and types.

The data sets are shown in Table 4.1.

4.1.2 Cluster Comparison

In order to evaluate the clustering quality of the algorithm the new clusters are generally compared against a reference clustering that is considered to be correct. There are several different measures to do this, but mostly rely on four individual counts:

- True positives (t_p) indicate a valid matching prediction in the test data.
- False positives (f_p) indicate an invalid matching prediction in the test data.
- True negatives (t_n) indicate a valid non-matching prediction in the test data.
- False negatives (f_n) indicate an invalid non-matching prediction in the test data.

Once these values have been calculated there are a number of ways to gauge the performance of the clustering. Specifically these are:

- Sensitivity $= \frac{\sum t_p}{\sum t_p + \sum f_n}$: This indicates the percentage of the positive matches, in the reference clustering, that were correctly identified. Ideally we want to miss as few as possible of these, thus getting a low f_n count and a value close to 1.
- Specificity $= \frac{\sum t_n}{\sum t_n + \sum f_p}$: This indicates the percentage of correctly identified negative values. A value close to 1 indicates that relatively few f_ps were found and thus not too many false clusterings have been formed.
- False Positive Ratio (FPR) = $\frac{\sum f_p}{\sum t_p + \sum f_p}$
- False Negative Ratio (FNR) = $\frac{\sum f_n}{\sum t_p + \sum f_n}$
- Jaccard Index $= \frac{\sum t_p}{\sum t_p + \sum f_p + \sum f_n}$: This is used to compare the similarity of two sets or clusters. Considering the sets shown in Figure 4.1 it is calculated as the intersection of the test and reference clusters ($\sum t_p$), divided by the union of the test and reference clusters ($\sum t_p + \sum f_p + \sum f_n$). Thus the greater the similarity between the two sets, the more the intersection resembles the union, i.e. the f_p and f_n sums get smaller and the jaccard index gets closer to 1.
- Matthews Correlation Coefficient $= \frac{\sum t_p \times \sum t_n - \sum f_p \times \sum f_n}{\sqrt{(\sum t_p + \sum f_p)(\sum t_p + \sum f_n)(\sum t_n + \sum f_p)(\sum t_n + \sum f_n)}}$: is typically used in machine learning and is a fair overall measure for assessing the quality of a classification.

Data Set	**Num Seqs**	**ave len**	**Description**
Benchmark10000	10 000	360	Human eye-tissue ESTs from [SANBI] that has been used as test data by many other clustering programs and thus is included for comparison purposes (not from genbank).
CSeries	126 719	469	Ten artificial data set created from a cDNA library using ESTSim. All of the ESTs inside the data sets have similar characteristics (length and error rates) and they range in size from around $12,000$ to $126,719$. These properties mean that the data set is ideal for checking the scaling properties of the algorithm.
anopheles gambiae	25 000	548	Mosquito
arabidopsis thaliana	81 000	111	Thale cress ESTs often used as a model organism for genetic studies since it has a small genome and was the first plant to be fully sequenced [Meinke *et al.* 1998].
bos taurus	44 479	624	Cow
caenorhabditis elegans	99 946	380	This nematode was the first sequenced multicellular animal genome [The C. elegans sequencing consortium 1998].
canis familiaris	80 460	593	Dog
drosophila melanogaster	79 999	433	Fruit fly
gallus gallus	44 000	694	Chicken
gasterosteus aculeatus	111 000	987	Three-spined stickleback
homo sapiens	200 009	514	Human
macaca mulatta	58 230	750	Rhesus macaque
mus musculus	46 500	419	House mouse
oryza sativa	54 500	701	Rice
ovis aries	78 500	595	Sheep
pan troglodytes	16 356	472	Chimpanzee
pongo pygmaeus	46 910	541	Bornean orangutan
rattus norvegicus	59 000	743	Brown rat
saccharomyces cerevisiae	34 604	513	Brewer's yeast
sus scrofa	182 500	427	Wild Boar
triticum aestivum	46 917	570	Wheat
xenopus tropicalis	191 500	681	Western clawed frog

Table 4.1: Data sets

Traditionally in EST clustering the way to classify t_ps, f_ps, t_ns and f_ns is shown in Figure 4.1. When we only have information on which ESTs belong to which clusters we say that:

- True positives (t_p) are when ESTs i and j belong to the same cluster in both the test and the reference clustering.
- False positives (f_p) are when ESTs i and j belong to the same cluster in the test clustering, but not in the reference clustering.
- True negatives (t_n) are when ESTs i and j do not belong in the same cluster in either the test clustering or the reference clustering.
- False negatives (f_n) are when ESTs i and j are in the same cluster in the reference clustering, but not in the test clustering.

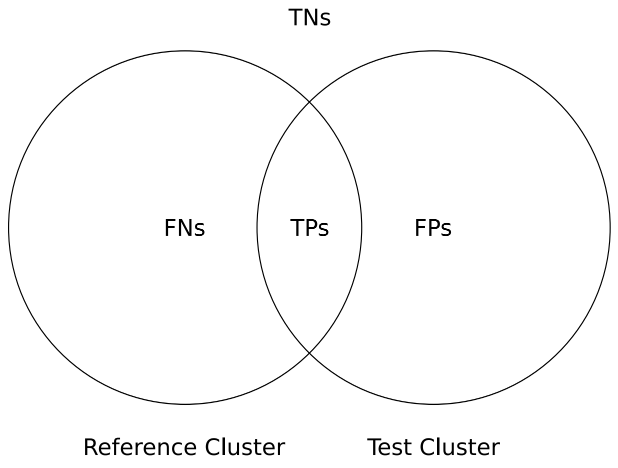


Figure 4.1: The figure shows how to classify pairs of ESTs between the two different clusterings.

For most cases this is the only way of calculating the t_p, t_n, f_p and f_n values of alternate clustering methods. In certain circumstances, however, it can lead to misleading results. Consider the two examples shown in Figures 4.2 and 4.3 which show a reference and test clusterings respectively. Looking at the reference clustering we see that there are two separate clusters. Examining the larger cluster we see that it is composed of 3 groups where there are lots of links between the ESTs, which are then joined by single links. In the test clustering we can see that the clustering missed these single links and so we now have four distinct clusters.

Now when comparing the two clusterings we only consider which clusters the ESTs belong to, and not how they matched the other ESTs. This comparison yields the following values:

- $\sum t_p = 48$
- $\sum t_n = 54$

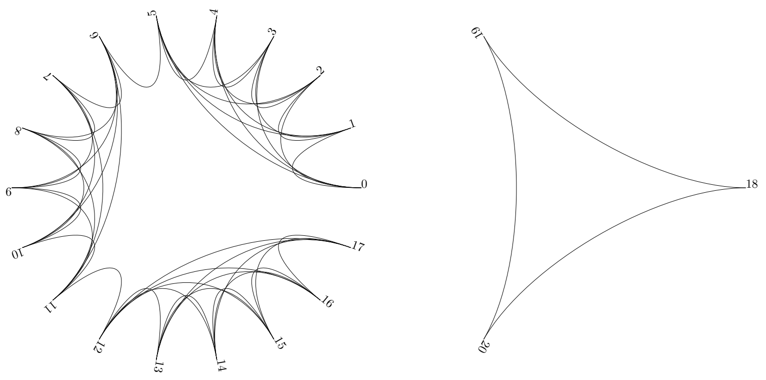


Figure 4.2: Example 1: Reference Cluster

- $\sum f_p = 0$
- $\sum f_n = 108$
- sensitivity = 0.31
- specificity = 1.0
- Jaccard Index = 0.31
- Matthews Correlation Coefficient = 0.32

This would usually lead us to conclude that the clustering is poor. If, instead, the exact matches between the ESTs had been considered such that:

- True positives (t_ps) are when ESTs i and j are found to have an overlap by both the reference clustering program and the test clustering program.
- False positives (f_ps) are when ESTs i and j are found to have an overlap by the test clustering program, but not by the reference clustering program.
- True negatives (t_ns) are when neither the reference clustering program or the test clustering program find overlaps between ESTs i and j.
- False negatives (f_ns) are when ESTs i and j are found to have an overlap by the reference clustering program, but not by the test clustering program.

Recalculating the values shows that:

- $\sum t_p = 48$
- $\sum t_n = 160$
- $\sum f_p = 0$

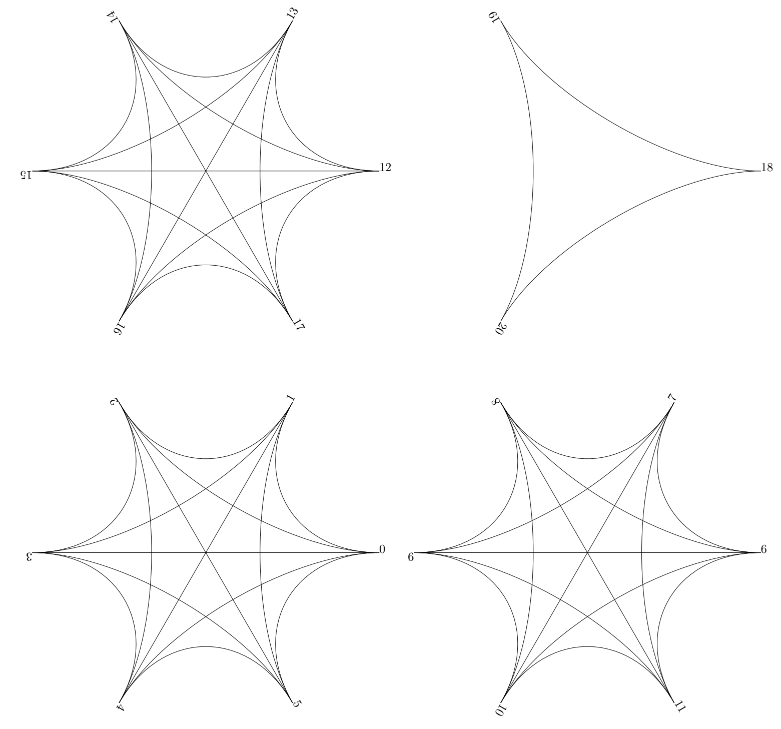


Figure 4.3: Example 1: Test Cluster

- $\sum f_n = 2$
- sensitivity = 0.96
- specificity = 1.0
- Jaccard Index = 0.96
- Matthews Correlation Coefficient = 0.97

The difference in the values is due to there being far fewer false negatives and more true negatives in the new calculation. This leads to what appears to be more confidence in the test clustering. However, Figures 4.4 and 4.5 show a counter example where the new method of calculating the t_ps, t_ns etc. actually leads to having lower confidence in the clustering.

Here the test clustering program produces exactly the same clusters as the reference clustering program, however if we were able to see which ESTs matched with each other in the two different programs, then we would notice that the reference clustering produces clusters which are fully connected (every EST in the cluster matches with every other EST in the cluster), while the test clustering produces a relatively sparse set of matches.

When calculating the different values we find that if we only consider which ESTs belong to which clusters we find:

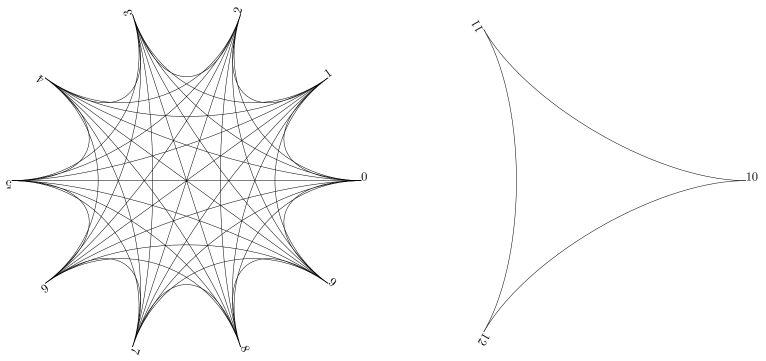


Figure 4.4: Example 2: Reference Cluster

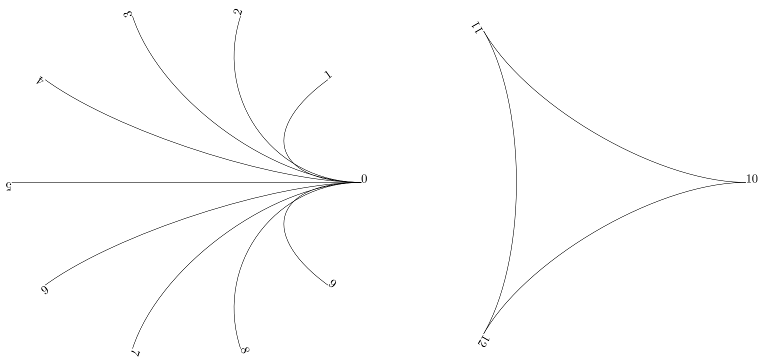


Figure 4.5: Example 2: Test Cluster

- $\sum t_p = 48$
- $\sum t_n = 30$
- $\sum f_p = 0$
- $\sum f_n = 0$
- sensitivity = 1.0
- specificity = 1.0
- Jaccard Index = 1.0
- Matthews Correlation Coefficient = 1.0

Whereas considering the exact matches between the ESTs we find:

- $\sum t_p = 12$
- $\sum t_n = 30$

- $\sum f_p = 0$
- $\sum f_n = 36$
- sensitivity = 0.25
- specificity = 1.0
- Jaccard Index = 0.25
- Matthews Correlation Coefficient = 0.34

So now there are two examples where the different methods of calculating the t_ps, t_ns etc. produce very different results for validating a particular clustering. In this research both have been used. When evaluating the performance of the heuristics, the pairwise matches are considered. When evaluating the overall algorithm performance, the cluster-wise matches are considered.

4.2 Experimental Results

This Section describes the experiments that were done as well as describing and discussing the results. Essentially each step in the overall algorithm is looked at and the parameters analysed and finally the algorithm as a whole is considered.

4.2.1 sd_heuristic Performance

When looking at the accuracy and timing results of the algorithm there are 4 parameters that need to be considered. They are:

1. Word size: This specifies the sizes of words to be considered between any two sequences. A larger word size means that any given word is less likely to occur randomly, so random matches between two unrelated sequences is unlikely. Unfortunately this also means that in noisy data, a match that should occur is more likely to be missed. Conversely with a smaller word size we are more likely to get matches between unrelated sequences and less likely to miss matches between related sequences.

2. Word count threshold: The word count threshold specifies the number of words that two sequences should have in common before considering them a (potential) match. Making a high threshold means that two sequences must have a high number of words in common and so it is relatively unlikely that two unrelated sequences will meet this criterion, but it is also possible that two related sequences will be missed. Conversely with a low threshold we are unlikely to miss two related sequences, but it is more likely that two unrelated sequences will meet the criterion.

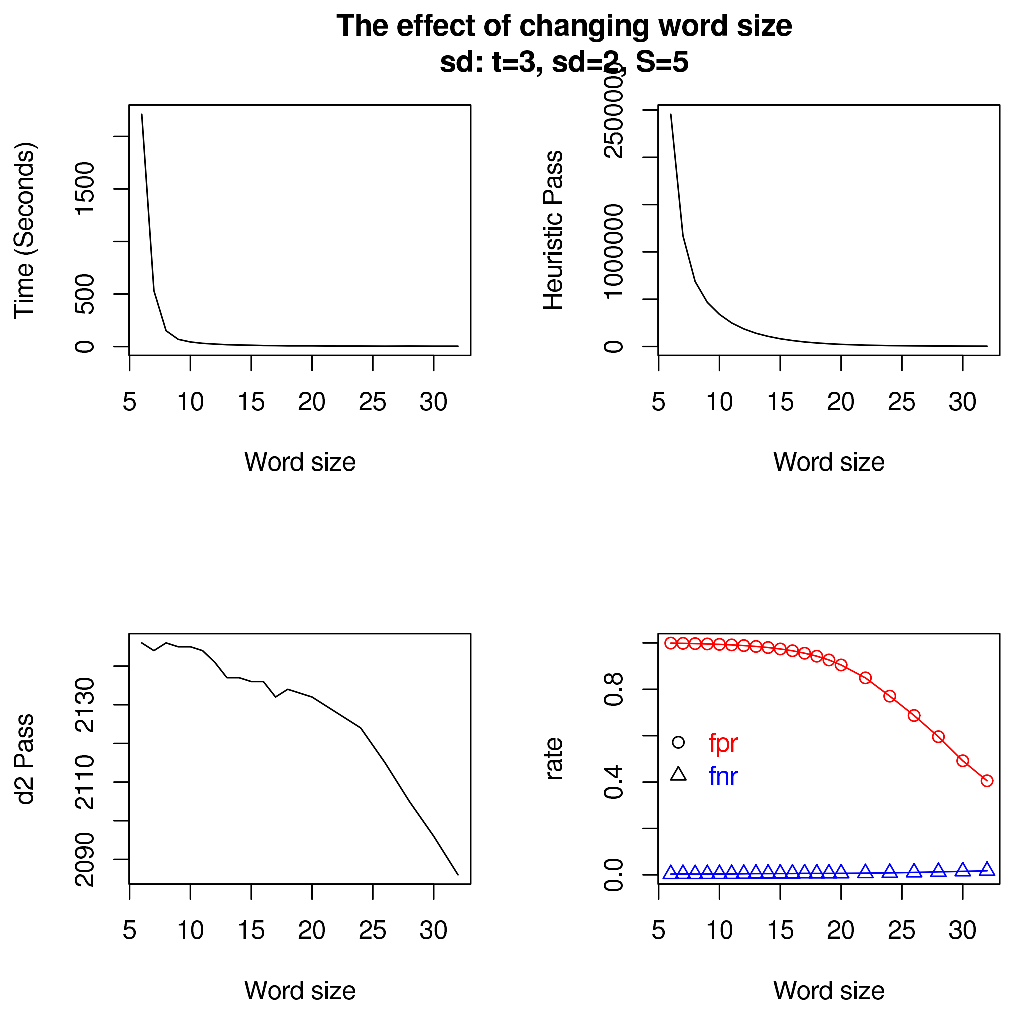


Figure 4.6: Parameter selection – word size

3. Standard deviation threshold: The standard deviation threshold specifies how distributed related words can be across the two sequences being compared. It does this by taking the difference in the positions and then calculating the standard deviation of those values. Two sequences with a low standard deviation will have words that occur in relatively similar positions, i.e. say words α, β, γ and δ occur in positions *1*, *2*, *3* and *4* respectively in sequence *A*, then a related sequence could have them in positions *11*, *12*, *14* and *15*, say, giving a standard deviation of 0.577. An unrelated sequence could have them in positions *11*, *18*, *19* and *12*, say, giving a standard deviation of 4.12.

4. Skip value: The skip value specifies the frequency of words being considered from the source sequences (every word in the target sequence is considered). Having a word spacing of zero

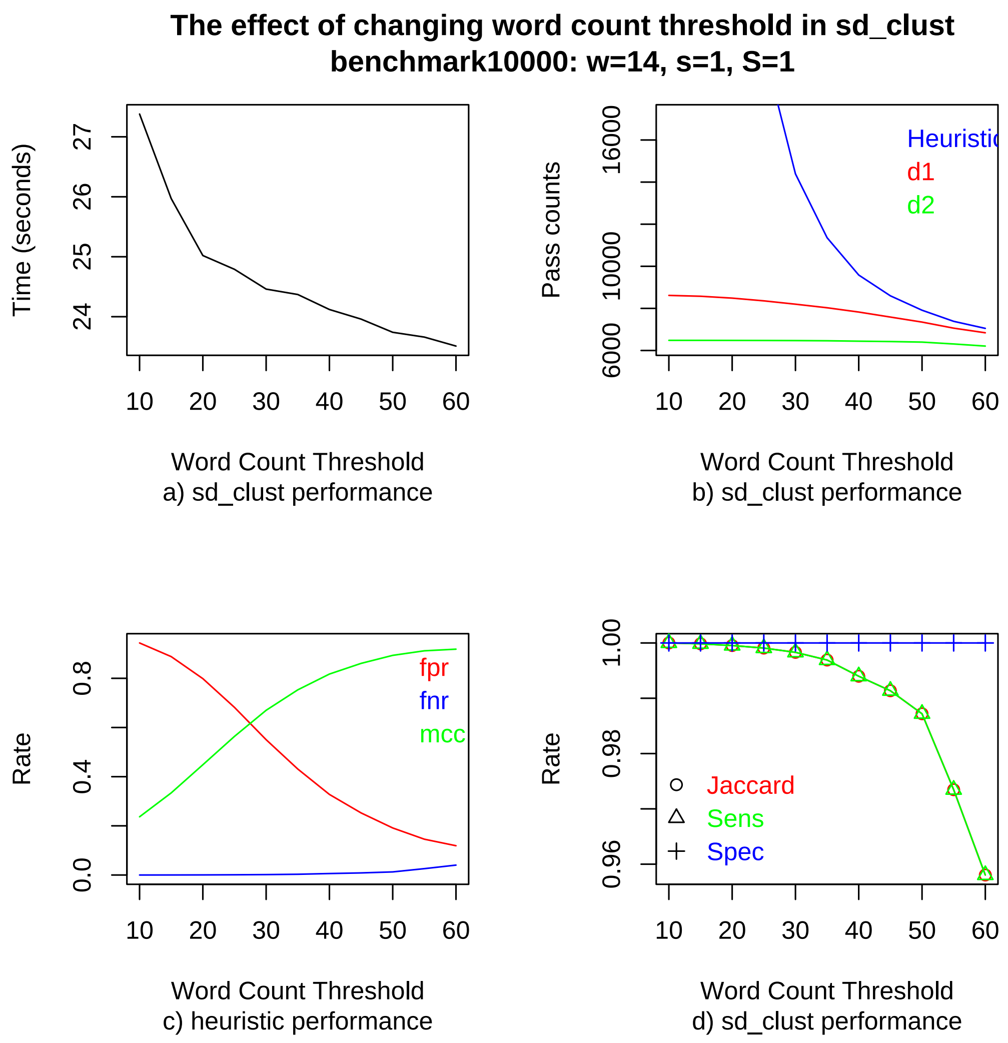


Figure 4.7: Parameter selection – word count threshold

means that every word is considered in the source sequences. A word spacing of one means that every alternate word is considered, a spacing of two means that every third word is considered etc. A small spacing means that it is unlikely that important matching words between sequences are missed, but it also means that it is more likely that matching words between unrelated sequences are found. Conversely a larger spacing means it is less likely to find matching words between unrelated sequences and more likely to miss related words between matching sequences. Of course the word count threshold needs to be adjusted to reflect the change in word spacing since if we are only considering half as many words in the source sequences we expect half as many matches with a matching target sequence.

Each parameter has been varied on its own to see the influence it has on clustering performance. There are four graphs for each parameter which show:

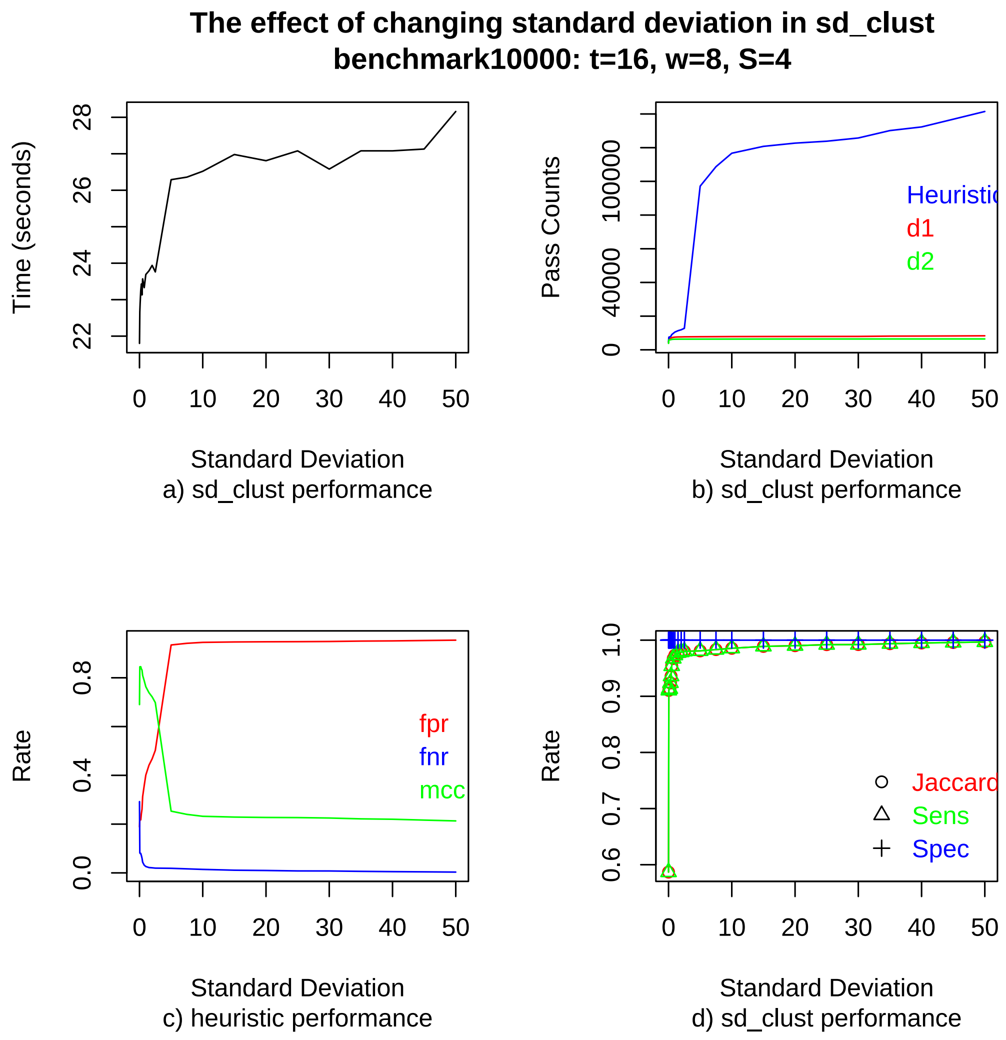


Figure 4.8: Parameter selection – standard deviation threshold

1. The running time

2. The number of times the heuristic, d1, and d2 succeed

3. The heuristic performance in terms of false positive rate, false negative rate, Matthews correlation coefficient

4. The overall clustering performance in terms of sensitivity, specificity and Jaccard index.

Note that the reference cluster was created by doing an all-against-all comparison using d2 scores. And since the sd_clust algorithm also calculates the d2 score it never gets any false positives and so the specificity is always 1 and the Jaccard index is identical to the sensitivity.

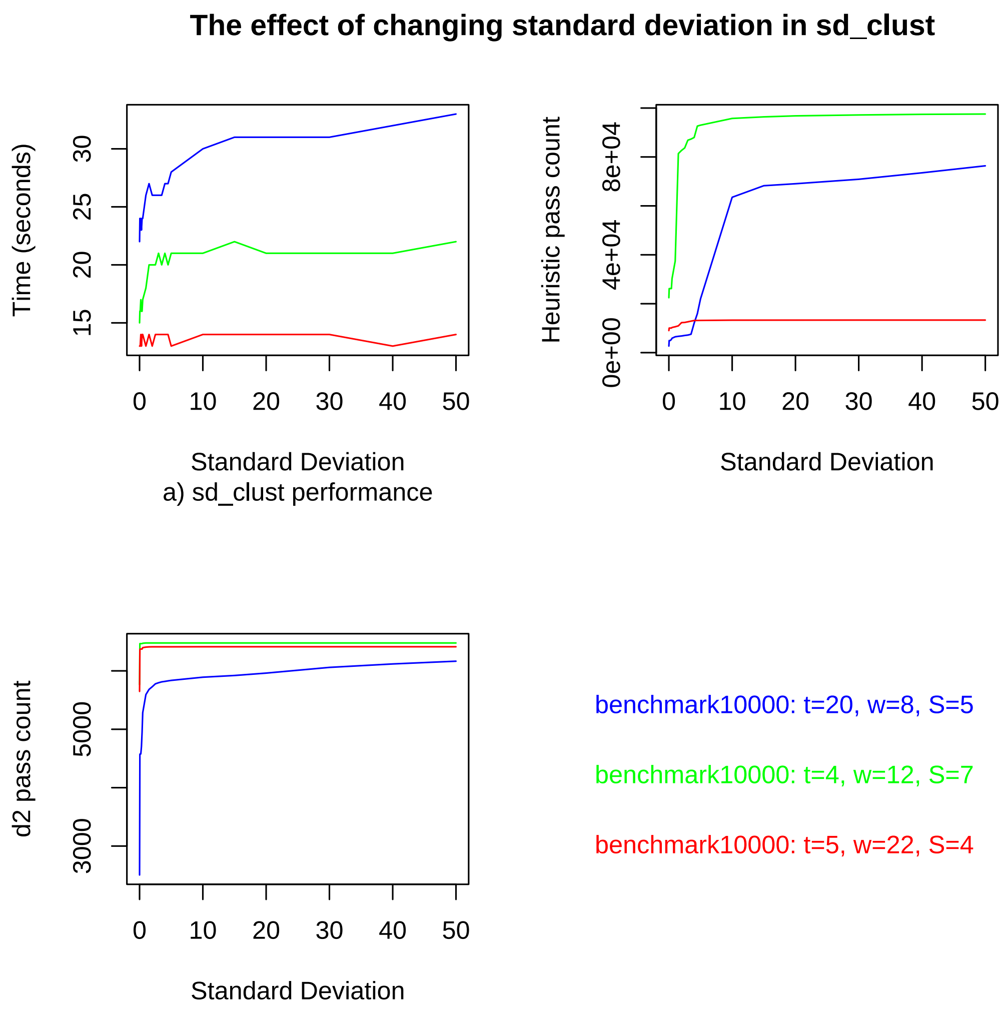


Figure 4.9: Parameter selection – standard deviation threshold

Influence of word size

Word size was found to have a large impact on the performance of the algorithm (Figure 4.6). When fixing the rest of the parameters and increasing the word size from 6 to 32 we find that in general: execution time decreases; the number of false positives decreases; the number of false negatives increases; and the overall clustering quality decreases. Interestingly we see that when we use a small word size of 6 we actually get a poorer quality of clustering than using a word size of 22. This occurs because the smaller word size means that matching words from unrelated parts of matching sequences are found. While this increases the match count, it also increases the standard deviation of the match, which means it sometimes fails to fall below the required threshold. The execution time can be seen to decrease dramatically for word sizes between 6 and 10 after which the speed gains are less sig-

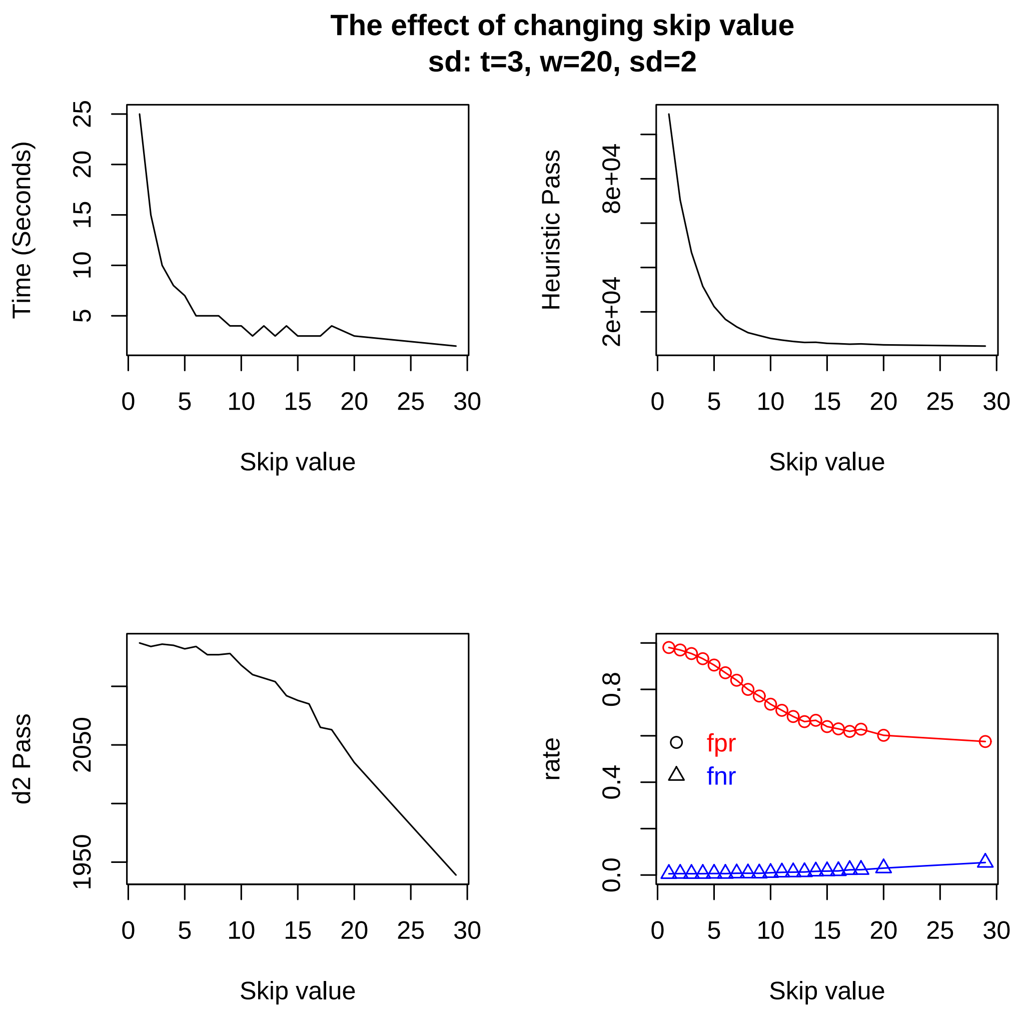


Figure 4.10: Parameter selection – skip value

nificant. Looking specifically at the heuristic performance we see that the larger the word size, the better the performance – with dramatic improvements in the false positive ratio, and a relatively small worsening in the false negative ratio. The overall clustering quality, however, does decrease with the larger word sizes, and this is seen with the decrease in sensitivity and Jaccard index.

Influence of word count threshold

The word count threshold parameter shows a similar pattern to the word size parameter (Figure 4.7). We see that as the word count threshold increases from 10 to 60 the a number of things happen including: execution time decreases; number of false positives decreases; the number of false negatives increases; and overall clustering quality decreases.

Influence of standard deviation threshold

The impact that the standard deviation threshold has on clustering quality appears in Figure 4.8. We see that the larger the threshold, the more pairs of sequences that pass the heuristic. This is, however, dependent on the selection of the other parameters and can be seen in Figure 4.9. What we see here is that with the large word size of 22 we do not get a significantly larger number pairs passing the heuristic, but when a small word size of 8 is chosen the number of pairs matching with the heuristic grows faster and more consistently. The overall clustering quality does not seem to improve dramatically with large standard deviation thresholds, but neither does the clustering time increase dramatically.

Skip Value

Changing the skip value has a very significant effect on clustering quality and time, and the tuning of this parameter is very important. In Figure 4.10 we see that clustering time decreases rapidly with a larger skip value. This is due to two factors: most importantly the number of words being compared drops proportionally to the skip value (half as many comparisons when using a skip value of 2 as compared to using a skip value of 1), and secondly the number of matches between sequences that occur by chance also decreases. Looking at 4.10 *b)* we see that using a skip value of 15 or larger produces almost no false positives. We do however, also start getting more false negatives. Looking at the Matthews correlation coefficient in Figure 4.10 *c)* and the Jaccard index in Figure 4.10 *d)* we see that a good choice for skip value is about 7 (with this set of parameters) which gives a good balance of speed and quality.

4.2.2 d^1 Performance

The quality of the d^1 heuristic was measured by doing an all-versus-all comparison of the sequences in various data sets using d^1 as the only heuristic, and d^2 as the final check. Looking at the $\frac{FP}{TP}$ ratio in Table 4.2 we see that the performance was varied – in the best case it was 0.5% (for every 200 correctly identified matches, there was 1 falsely identified match) and in the worst case it was $2\,154\%$ (for every 2 correctly identified matches, there were 43 incorrectly identified matches). Figure 4.11 shows a box plot of the $\frac{FP}{TP}$ ratio for the raw data, as well as when it has been cleaned (more information on cleaning can be seen in Section 4.2.3).

4.2.3 Overall Performance

Scaling performance

The scaling performance is shown in Figures 4.12, 4.13 and 4.14. Figure 4.12 shows how the performance of sd_clust changes with some arbitrary changes in the parameters (all with similar cluster quality characteristics). Figure 4.13 shows some least-squares fitted curves to evaluate the computational order of the implementation and was found to be between $O(n^{1.58})$ and $O(n^{1.86})$ (for these choices of parameters). Finally Figure 4.14 shows the processing time and peak memory usage for a

Data Set	False Positives	True Positives	$\frac{FP}{TP}$
anopheles gambiae	22 874	236 797	0.097
arabidopsis thaliana	473 137	847 036	0.559
bos taurus	49 011	9 146 383	0.005
caenorhabditis elegans	72 104	1 280 215	0.056
canis familiaris	5 118 387	5 735 435	0.892
drosophila melanogaster	209 992	4 746 030	0.044
gallus gallus	7 132 745	331 155	21.539
gasterosteus aculeatus	5 197 257	10 980 593	0.473
homo sapiens	181 822	63 735 165	0.028
macaca mulatta	2 081 372	6 439 171	0.323
mus musculus	62 481	1 484 072	0.042

Table 4.2: d1 performance

single choice of parameter. The memory usage in this case was found to be $O(n^{1.28})$. The theoretical analysis showed that memory usage should have been $O(n)$ and a possible reason why this didn't occur is that a vector data structure was chosen in the implementation to record when a match between two sequences was found. This was done so that the list of matches could be quickly searched, and then reset, when it was time to do the d^1 comparison. However, with a larger data set, it means that this data structure grows since it is more likely that more pairs of sequences will have a word in common. Compounding this even further is the data structure itself and the fact that its space requirements are dynamically allocated – with a scheme that results in the data structure doubling in size each time it runs out of space. This means that the memory requirements do not scale linearly. While this sounds poor, it does mean that in most cases the memory allocation will be less than if a static allocation is made – since that would require a worst-case scenario memory allocation to be made.

Another interesting feature to note in Figure 4.12 is that total clustering time cannot be predicted solely on a single parameter's value, or on the heuristic's performance. We see that in *b)* that with the largest word size (w = 20) we actually have the most number of false positives predicted, and yet looking at *a)* we see that it performed better (time-wise) than when the smallest word size was used (w = 10) which had the most accurate heuristic. Also, when using the middle word size (w = 16), we had the heuristic's performance falling between the other two, yet having the best computational time values. More work can be done on looking at each parameter's influence on the computational complexity.

Effects of different data sets

Different heuristic parameters do have different degrees of success depending on the data set being tested. Looking at Figure 4.15 we see that in most cases both sets of parameters work very well with a Jaccard index $> 95\%$. In two cases, however, we see that the Jaccard index of the heuristic using the longer words (w = 16) drops dramatically down to 37% and 50% for the *mus musculus* and *gasterosteus aculeatus* data sets respectively. The reason why it performs well on some data sets and not on others is not obvious, however there are a couple of possible explanations.

In the case of *gasterosteus aculeatus* it may be because the average sequence length is very long – about 987 base pairs. The long sequence length means that there is a higher probability that within a pair of matching sequences, matching words are found which do not actually belong to longer regions of similarity. The result of this is that the standard deviation of the matches then increases, meaning that it's less likely to fall under the threshold and thus pass the heuristic test.

In the case of *mus musculus* we can see that only 33 clusters less have been formed. A closer look at the clustering reveals that the "poorly" performing heuristic only gives 633 false negatives out of a total of 1 483 509 positives, when compared to an all-against-all d2 clustering. That's only 0.042% of the total positive matches and if a pair-wise comparison is done, then the Jaccard index is 99.9%. Looking at the histograms in Figure 4.16 we see a possible explanation for this – a large cluster of 4396 ESTs has been divided up into a number smaller clusters (1 910, 995, ...). As described in Section 4.1.2 even though the clusters are in general very similar, when a large cluster is divided up into smaller clusters it can have a large impact on the sensitivity and Jaccard index (when doing a set/cluster-wise comparison).

Effects of cleaning the data

The raw data was run through the *mdust* program which marks all the low-complexity regions with N. Low complexity regions are identified as simple sequence repeats. The default settings were used which meant that the region had to be at least 28 nucleotides long, and that a maximum word size of 3 is used. In other words if a single nucleotide, doublet, or triplet is repeated over a window of 28 or more, then it is masked.

Malde and Jonassen [2008] discuss the value of several cleaning techniques and find that the most effective, in terms of quality of output, is to use masking with species specific repeat libraries. It was beyond the scope of this research to find all the relevent repeat libraries and so the second best alternative of library-less repeat masking was used. Specifically *mdust* as it was readibly available, and widely used since it is included in the *TGICL* pipeline. Additionally *RBR* was used for comparison as it has been shown to give high quality results when clusters were compared to a reference cluster [Malde *et al.* 2006].

Cleaning the data has a significant impact on clustering time and the results can be seen in Table 4.3. The most dramatic improvements can be seen in the data sets which had a large amount of low-complexity, e.g. *bos taurus* had 2% and *gallus gallus* had nearly 4% of their basepairs marked as low-complexity regions after dusting. The low-complexity regions have a large impact on clustering time since they cause a very large number of pairs of ESTs to be matched by the heuristic when, in fact, they are not similar. Looking at *bos taurus* and *gallus gallus* we see that the number of heuristic passes drops from 88 185 534 to 73 754 and from 17 035 937 to 48 601 respectively after dusting.

Data File	Raw		mdust				rbr		
	sd	wcd	time	sd total	wcd total	tgicl	time	sd total	wcd total
anopheles gambiae	69	112	23	50	106	75	152	179	237
arabidopsis thaliana	620	355	11	47	339	148	80	102	406
bos taurus	2 007	3 279	47	135	365	862	370	466	708
caenohabtitis elegans	105	995	64	167	1 088	563	430	542	1 427
canis familiaris	3 102	27 348	80	420	1 088	785	803	991	6 340
drosophila melanogaster	1 240	1 118	58	148	863	513	456	552	1 265
gallus gallus	18 787	20 873	55	149	806	177	520	603	1 391
gasterosteus aculeatus	15 775	42 248	191	13 191	8 352	3 178			
homo sapiens	640	5 575	174	667	5 504	5 233			
macaca mulatta	3 168	8 081	74	281	950	946	684	836	1 687
mus musculus	58	247	32	83	259	243	212	255	437
oryza sativa	58 118	156 467	63	205	1348	336	790	884	2 796
ovis aries	424	1 118	79	239	1 086	464	802	942	1 810
pan troglodytes	18	37	13	26	40	62	66	89	103
pongo pygmaeus	204	1 121	43	148	365	332	326	389	786
rattus norvegicus	252	1 228	75	275	829	989	851	974	1 562
saccharomyces cerevisiae	46	144	30	73	180	238	176	268	397
sus scrofa	6 540	4 437	130	4 599	3 437	36 217	1 551	1 779	5 674
triticum aestivum	103	338	46	147	375	625	331	475	693
xenopus tropicalis	4 715	15 970	225	965	6 832	3 908			

Table 4.3: Time to cluster: This shows the time taken to cluster various data sets before and after they have been cleaned with *mdust* or *rbr*. The *tgicl* algorithm is included next to the *mdust*ed data since it applies mdust as a default when clustering. *rbr* failed to clean the *gasterosteus a.*, *homo s.* and *xenopus t.* data sets and so those results could not be included. The green and red highlighted regions indicate the minumum and maximum clustering times respectively for any given data set

Data Set	Run Time (seconds) $t = 3, w = 14, s = 4, S = 10$	Run Time (seconds) $t = 1, w = 16, s = 4, S = 14$
anopheles gambiae	48	41
arabidopsis thaliana	178	103
bos taurus	1218	505
caenohabtitis elegans	96	93
canis familiaris	2067	1913
drosophila melanogaster	730	646
gallus gallus	6012	5251
gasterosteus aculeatus	7387	7373
homo sapiens	577	550
macaca mulatta	1843	1513
mus musculus	51	46
oryza sativa	35462	33228
ovis aries	301	291
pan troglodytes	16	15
pongo pygmaeus	126	124
rattus norvegicus	190	182
saccharomyces cerevisiae	37	36
sus scrofa	4151	7015
triticum aestivum	78	76
xenopus tropicalis	3122	1197

Table 4.4: Parameter Exploration: This table shows the dramatic, and sometimes unpredictable, effect that varying parameters can have. While both of these sets of parameters gave very similar results in terms of their correctness we see that for the most part the second set results in a shorter run time for the first but in at least one case is actually significantly slower.

Parameter Space Exploration

The brief exploration of the parameter space was made and it was found that the parameters used for the sd_heuristic could be very different to those used for most of this research with large impacts on the clustering time and little effect on the clustering quality. Specifically it was found that setting the word size to 16, a skip value of 14 and a word count threshold of 1 gave very good results. The major implication of this is apparent when looking at the word count threshold of 1, i.e. all we need now is a single match between sequences of a word of length 16. This means that the standard deviation will always be 0 and thus it is a waste to compute this value. Note that this appears to be a sweetspot in terms of the computation – finding few false positives and negatives (d_2 comparisons are still done to filter out the false positives that are found). Looking at Table 4.4 shows some of the results found, but it again becomes clear that parameter selection is dependent on the specific data set being used. For instance in the case of *x. tropicalis* we see that the second set of parameters takes less than 40% of the default parameters, whereas with *s. scrofa* the new parameters actually take about 70% longer.

4.3 Summary

In this Section an empirical analysis of the algorithm has been performed. We showed in Section 3.3 that theoretically the d^1 comparison will always predict the true positives that a d^2 comparison predicts. Theoretical bounds on the computational complexity was shown to be m^2n^2 in Section 3.2, but empirical testing showed this to be sub-quadratic. Absolute timing against several other clustering solutions also found it was quicker in most cases. Parameter selection was found to be important and different parameters were found to perform differently on different data sets.

The theoretical bound on the memory requirement was shown to be $O(m^2n)$, i.e. linear in the number of sequences, but it was actually found to be slightly worse than this. This could be due to the choice of data structure used, however more analysis on this needs to be performed.

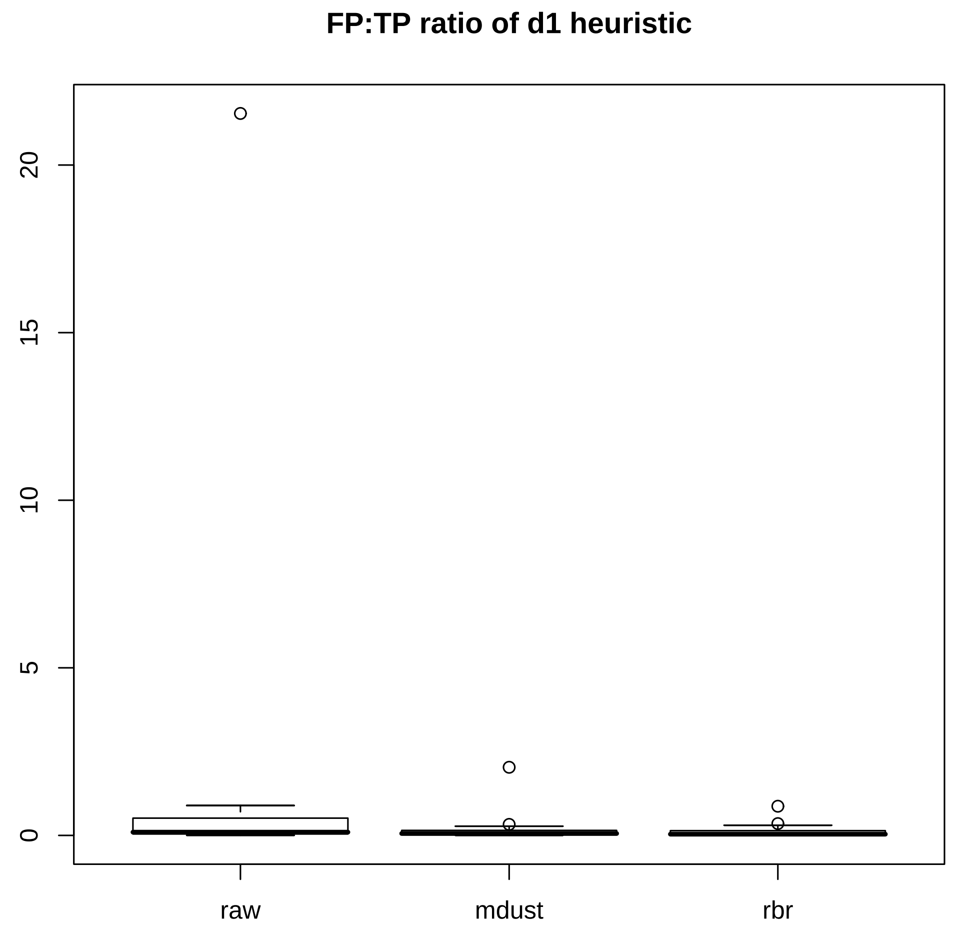


Figure 4.11: This shows a box plot of the FP:TP ratios over all the real world data sets for the raw data as well as the cleaned data. A low FP:TP ratio is good and the figure indicates that most of the data sets had very low FP:TP ratios where even the outliers were relatively low except in the one case in the raw data where there were more than 20 false positives for every true positive.

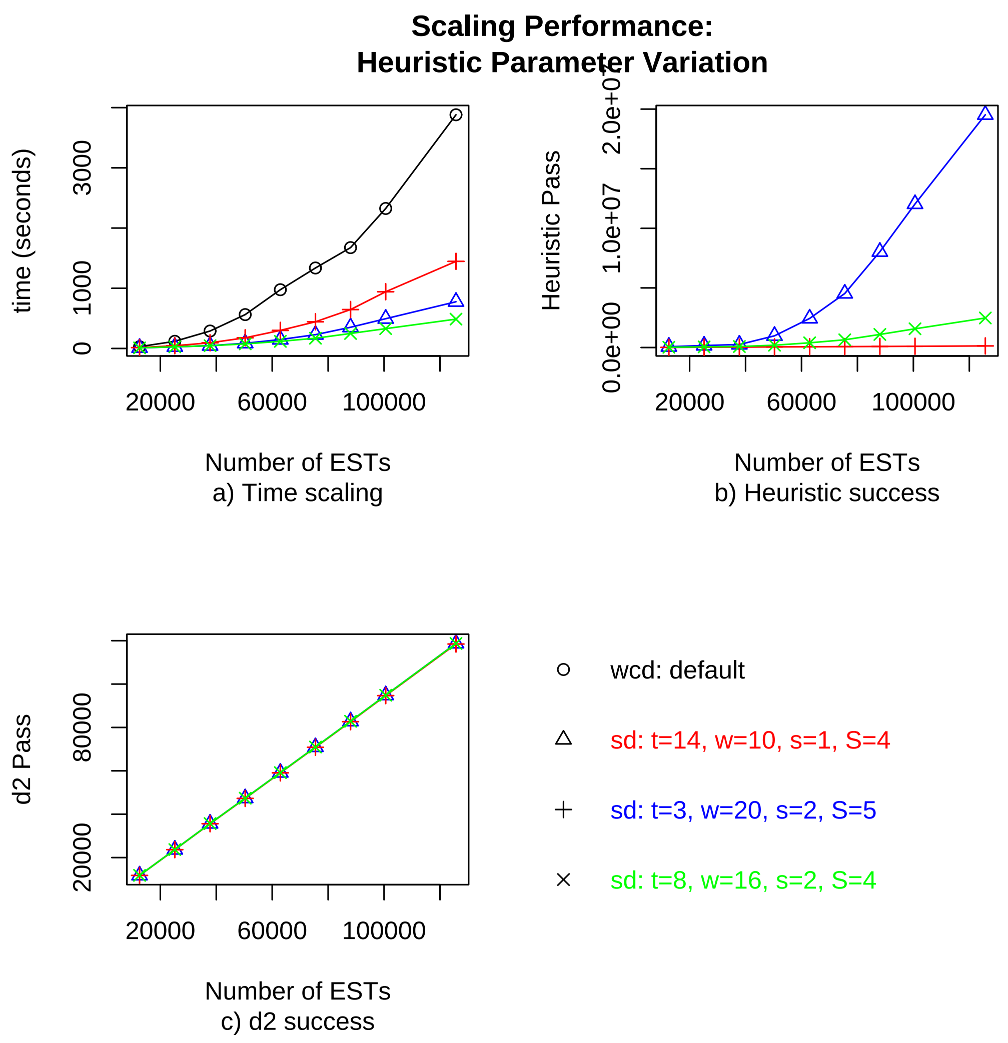


Figure 4.12: Scaling performance – parameter selection: This figure illustrates a number of aspects of how parameter selection effects scaling performance. Firstly it shows how the time scaling varies with parameter selection (more detail in Figure 4.13). Secondly it shows the number of times the heuristic passed for the various parameter selections. Finally it shows the number of times the final *d2* check passed – indicating a similar quality of clustering for all the parameter selections.

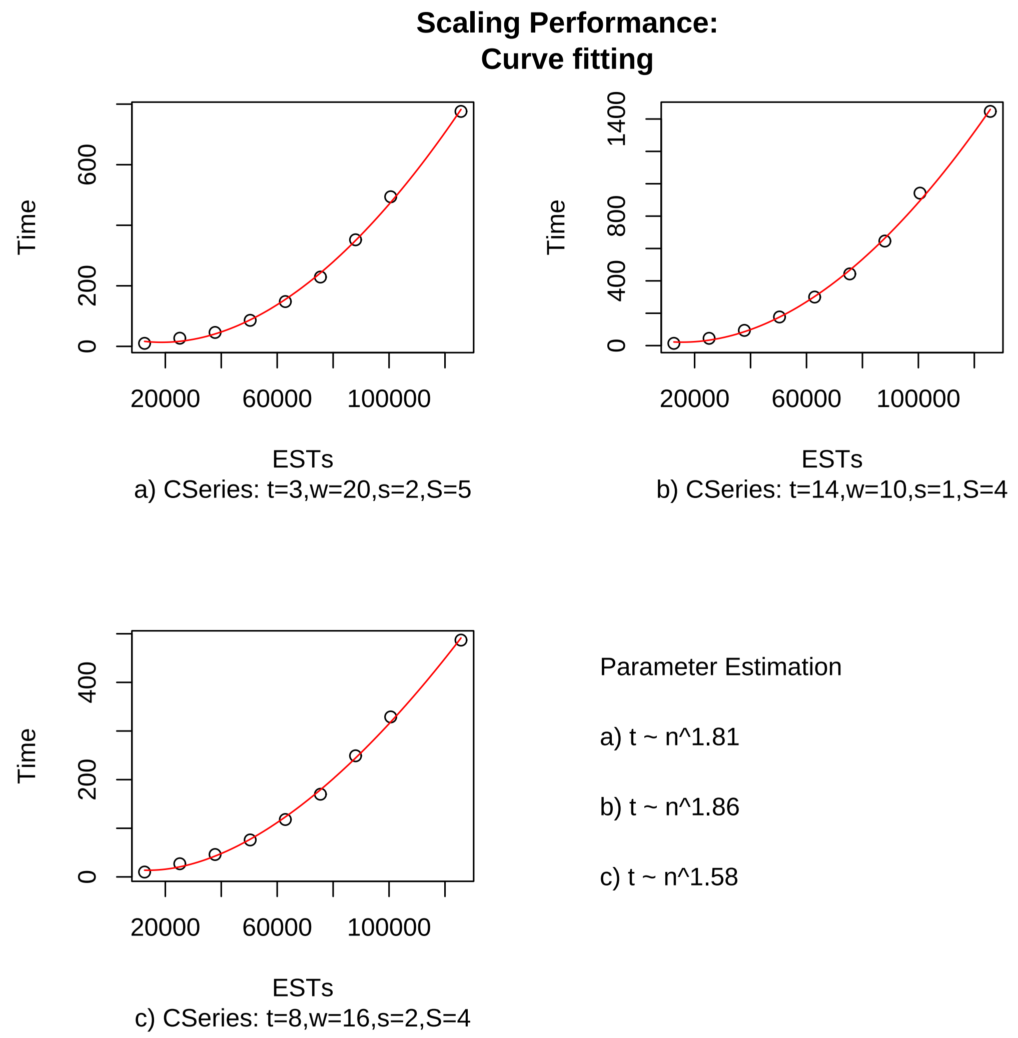


Figure 4.13: Scaling performance – regression: This shows that the scaling performance of *sd_clust* is somewhat dependent on the parameters used. In this case the time scale is between $O(n^{1.58})$ and $O(n^{1.86})$.

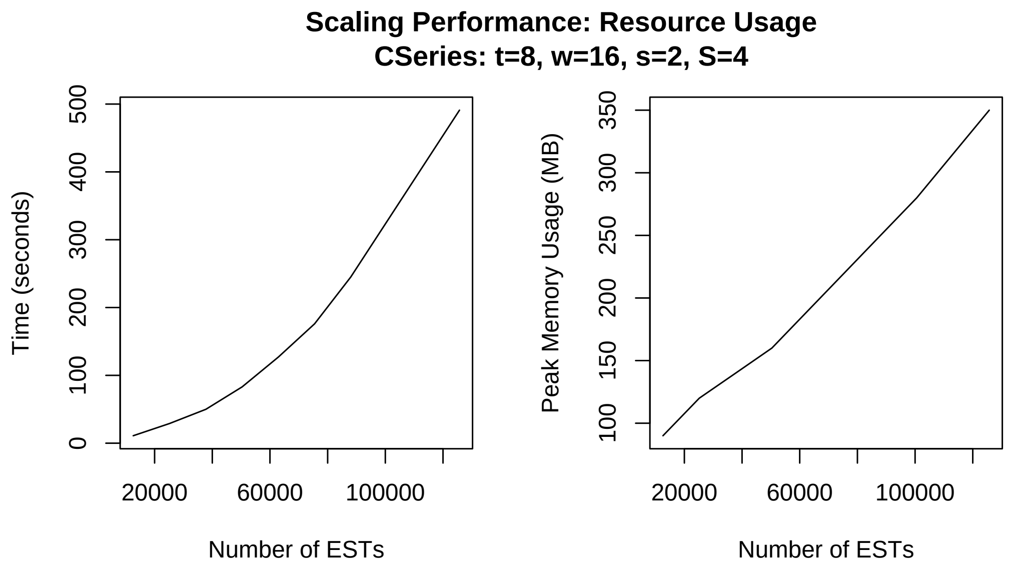


Figure 4.14: Scaling performance – resource usage: This shows that processor usage scales in polynomial time, while the memory usage scales linearly.

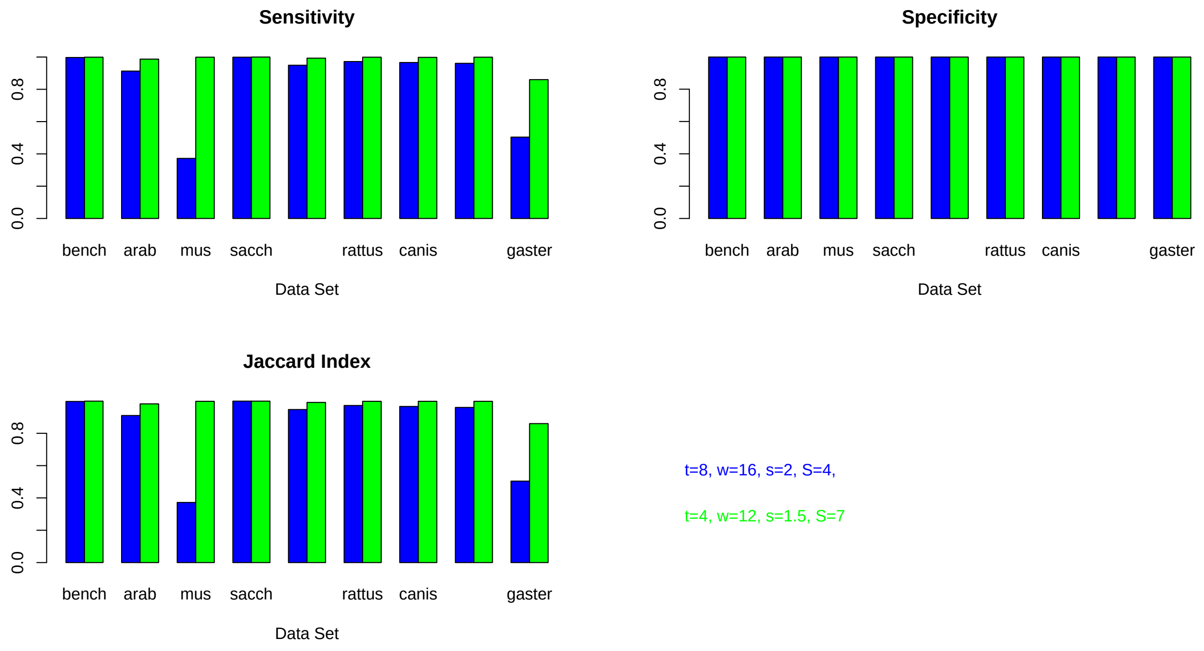


Figure 4.15: Data set variablility: This shows that the success of the heuristic using a given set of parameters is dependent on the data set. A more aggressive set of parameters can work in most cases, but fail in others.

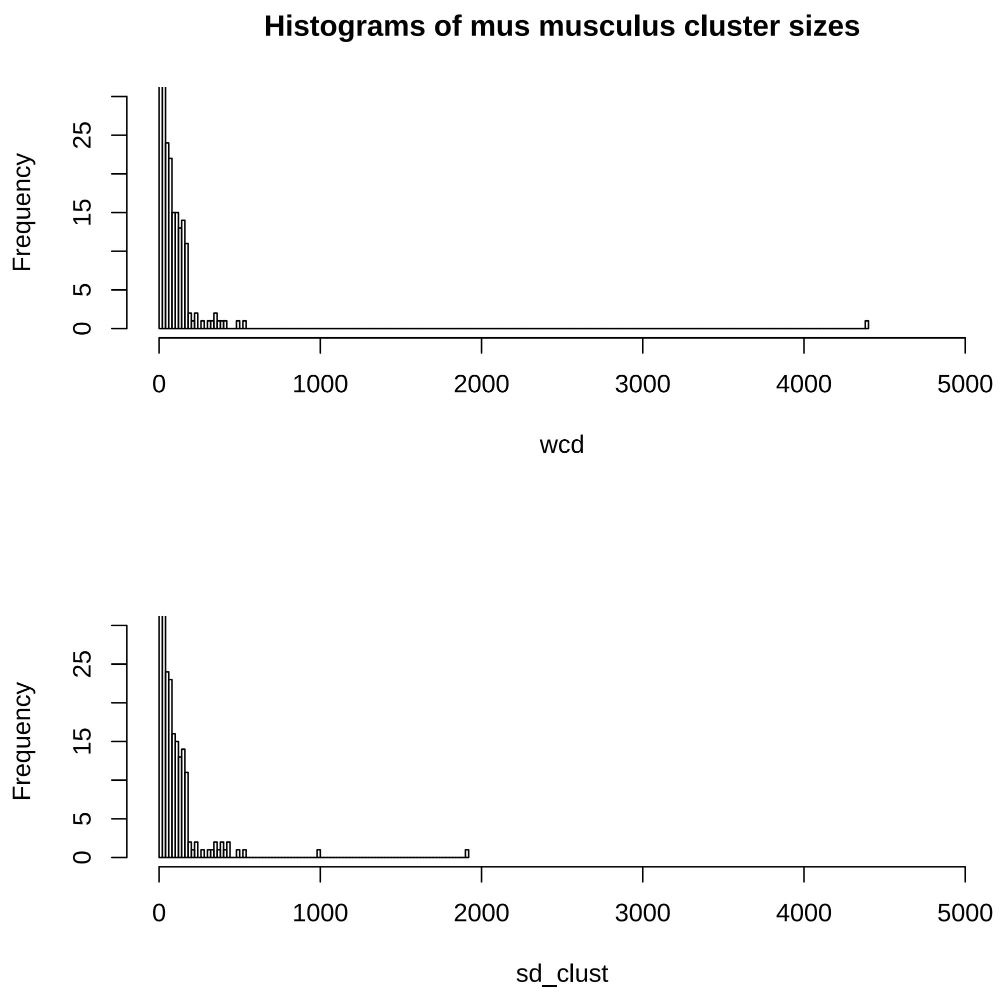


Figure 4.16: Histogram of mus musculus cluster sizes illustrating that *wcd* has combined some of the larger clusters created by *sd_clust* into a larger super-cluster.

Chapter 5

Conclusions

ESTs are an important tool for genetic research, but require significant computation in order to make the data useful. This research has presented an algorithm that takes a set of individual ESTs and creates clusters of them so that similar sequences can be analysed. The algorithm uses a number of word-frequency based techniques to quickly process the information and it was found to be sub-quadratic in computation time. The first heuristic (sd_heuristic) based on finding matching words in similar relative positions was found to be quite sensitive to parameter tuning and to the data-set being used, but in general it performed well – even when conservative values were used. While it was accurate it was ultimately found to be unnecessary as the best results were obtained when the parameters for the heuristic meant that a single match of a word of length 16 was required between two sequences to give the best computational time versus accuracy ratio. By altering the parameters it could be far more accurate (less false positives predicted), however this came at the expense of longer computation which did not give more accurate results after the second heuristic had been run.

The second heuristic was shown to be mathematically correct in that it never produces any false negative predictions when presented with a pair of sequences. Further it was shown to be finer grained than the first heuristic and thus dramatically reduced the number of pairs of sequences that were passed onto the d^2 final sequence comparison.

An empirical analysis showed the computational complexity of the complete algorithm (two heuristics plus the final d^2 comparison) to be sub-quadratic and it was feasible to cluster large data sets (of more than $1\,000\,000$ sequences) on a single modern computer. It was found that clustering time could, in many cases, be significantly reduced by first cleaning the data.

Future Work

The algorithm can be extended in a number of directions. The most obvious of which is to parallelize the solution since the sequence comparisons are independent of one another. Furthermore this is necessary since most modern EST clustering tools have been designed to run in parallel and thus comparing this method against them is unfair. Parallelisation can be done to take advantage of multiple

CPUs in a single machine, as well as multiple computers (possibly with multiple CPUs themselves) in a networked cluster. Both of these ideas should be explored, although they do present some challenges. When distributing the work to a cluster there is always the issue of communication, and when there are many computers to talk to, this can be a limitting factor in achieving speedup. Other factors need to be considered like how much work should be sent to each node at a time so that no nodes are left sitting idle.

Another area that can use more investigation is finding the optimal parameters for the heuristics to use. A fairly conservative set of values was used for many of the tests here that in most cases gave a Jaccard index of 99% or more. If this requirement is relaxed a little – say to 95% then dramatic speedups can be gained. Furthermore it was found that the *sd* heuristic's performance was not consistent with different data sets. In one case it could be fairly accurate – predicting very few false positives, but in other cases it can predict very many false positives. There are two possible areas where work could be done that look at this problem: firstly the program could perform some analysis of the data and then predict a good set of parameters to use; or alternatively a dynamic solution might be possible that tunes the parameters on the fly if they are found to be either too stringent or too relaxed. Either way a more thorough investigation of the parameter space needs to be performed so that guidelines for the parameters can be found.

Another issue that was raised in the research was the memory requirements of the algorithm. There are a number of ideas that can be tested to lower requirements. A simple solution to tackle the space required for the word list is to store fewer words. In its current form every word is stored, but when the algorithm does the sd_heuristic comparison one of the parameters is the skip value which specifies how many words to skip between each word being compared. It would be simple to not skip any words here, but to rather store fewer words in the word list. The added advantage of using less memory here is to have better memory locality – potentially resulting in fewer cache misses and improved overall performance. Another simple technique is to create a far smaller word list in the first place, say only the first k sequences have their words added to the word list and then all the sequences are compared to that. The word list is then deleted and the next k are added. Then all the sequences (apart from the first k sequences, since they have already been compared to everything) are compared to that wordlist. This would mean that any data set that can fit in memory can be analysed given a small enough value of k. Again the memory locality could result in a performance boost.

Finally since the sd_heuristic was not found to be terribly relevant for prediction (since the best results were found for parameters that required a single match between sequences of a word that was 16 characters long) it may be a better idea to consider the standard deviation measure for a final comparison rather than as a heuristic step before a comparison such as d^2. This would require shorter word-lengths, say 6. And a very high *sd* threshold.

References

[Altschul *et al.* 1990] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, May 1990.

[Berry 2004] Michael Berry. *A hardware implementation of the* d^2 *distance function for expressed sequence tag clustering using a single FPGA*, 2004. Honours Research Report, School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa.

[Burke *et al.* 1999] John Burke, Dan Davison, and Winston Hide. d2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9(11):1135–1142, November 1999.

[Carpenter *et al.* 2002] John E. Carpenter, Alan Christoffels, Yael Weinbach, and Winston A. Hide. Assessment of the parallelization approach of d2_cluster for high-performance sequence clustering. *Journal of Computational Chemistry*, 23:755–757, 2002.

[Chou and Holmes 2001] Hui-Hsien Chou and Michael H. Holmes. DNA sequence quality trimming and vector removal. *Bioinformatics*, 17(12):1093–1104, 2001.

[Christoffels *et al.* 2001] A. Christoffels, A. van Gelder, R. Miller, T. Hide, and W. Hide. Stack: sequence tag alignment and consensus knowledgebase. *Nucleic Acids Research*, 29:234–238, 2001.

[Dunn *et al.* 2008] Casey W. Dunn, Andreas Hejnol, David Q. Matus, Kevin Pang, William E. Browne, Stephen A. Smith, Elaine Seaver, Greg W. Rouse, Matthias Obst, Gregory D. Edgecombe, Martin V. Sorensen, Steven H. D. Haddock, Andreas Schmidt-Rhaesa, Akiko Okusu, Reinhardt Mobjerg Kristensen, Ward C. Wheeler, Mark Q. Martindale, and Gonzalo Giribet. Broad phylogenomic sampling improves resolution of the animal tree of life. *Nature Online*, 2008.

[Ewing and Green 1998] Brent Ewing and Phil Green. Base-Calling of Automated Sequencer Traces Using PHRED. ii. Error Probabilities. *Genome Research*, 8(3):186–194, March 1998.

[Ewing *et al.* 1998] Brent Ewing, LaDeana Hillier, Michael C. Wendl, and Phil Green. Base-Calling of Automated Sequencer Traces Using PHRED. i. Accuracy Assessment. *Genome Research*, 8(3):175–185, March 1998.

[Gotoh 1982] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[Green 1999] Phil Green. *Documentation for PHRAP and CROSS_MATCH*, 1999.

[Hazelhurst and Bergheim 2003] Scott Hazelhurst and Anton Bergheim. *ESTSim: A tool for creating benchmarks*, 2003.

[Hazelhurst *et al.* 2008] Scott Hazelhurst, Winston Hide, Zsuzsanna Liptk, Ramon Nogueira, and Richard Starfield. An overview of the wcd EST clustering tool. *Bioinformatics*, 24(13):1542–1546, 2008.

[Hazelhurst 2003] Scott Hazelhurst. *An efficient implementation of the* d^2 *distance function for EST clustering: preliminary investigations*. Technical report, School of Computer Science, University of the Witwatersrand, 2003.

[Huang and Madan 1999] Xiaoqiu Huang and Anup Madan. CAP3 a DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[Hunter 1993] Lawrence Hunter. *Artificial Intelligence and Molecular Biology*, chapter 1. Molecular Biology for Computer Science, pages 1–45. AAAI Press, 1993.

[Jerka *et al.* 2005] J. Jerka, V. Kapitonov, A. Pavlicek, P. Klonowski, O. Kohany, and J. Walichiewicz. Repbase update: a database and an electronic journal. *Cytogenetic and Genome Research*, 110(1-4):462–467, 2005.

[Kalyanaraman *et al.* 2003] Anantharaman Kalyanaraman, Srinivas Aluru, Suresh Kothari, and Volker Brendel. Efficient clustering of large EST data sets on parallel computers. *Nucleic Acids Research*, 31(11):2963–2974, 2003.

[Lipman and Pearson 1985] DJ Lipman and WR Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 1985.

[Lizotte-Waniewski *et al.* 2000] Michelle Lizotte-Waniewski, Wilson Tawe, David B. Guiliano, Wenhong Lu, Jing Liu, Steven A. Williams, and Sara Lustigman. Identification of potential vaccine and drug target candidates by expressed sequence tag analysis and immunoscreening of onchocerca volvulus larval cDNA libraries. *Infection and Immunity*, 68(6):34913501, June 2000.

[Malde and Jonassen 2008] Ketil Malde and Inge Jonassen. Repeats and EST analysis for new organisms. *BMC Genomics*, 9(23), 2008.

[Malde *et al.* 2003] Ketil Malde, Eivind Coward, and Inge Jonassen. Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19:1221–1226, 2003.

[Malde *et al.* 2006] Ketil Malde, Korbinian Schneeberger, Eivind Coward, and Inge Jonassen. RBR: Library-less repeat detection for ESTs. *Bioinformatics*, 22(18):2232–2236, July 2006.

[Malde 2004] K. Malde. *Algorithms for the Analysis of Expressed Sequence Tags*. PhD thesis, Department of Informatics, University of Bergen, 2004.

[Meinke *et al.* 1998] David W. Meinke, J. Michael Cherry, Caroline Dean, Steven D. Rounsley, and Maarten Koornneef. Arabidopsis thaliana: A model plant for genome analysis. *Science*, 282:662–682, October 1998.

[Morgulis *et al.* 2006] Aleksandr Morgulis, E. Michael Gertz, Alejandro A. Schäffer, and Richa Agarwala. A fast and symmetric DUST implementation to mask low-complexity DNA sequences. *Journal of Computational Biology*, 13(5):1028–1040, 2006.

[Mudhireddy *et al.* 2004] R. Mudhireddy, F. Ercal, and R. Frank. Parallel hash-based EST clustering algorithm for gene sequencing. *DNA and Cell Biology*, 23(10):615–623, 2004.

[Nagaraj *et al.* 2007] Shivashankar H. Nagaraj, Robin B. Gasser, and Shoba Ranganathan. A hitchhiker's guide to expressed sequence tag (EST) analysis. *Briefings in Bioinformatics*, 8(1):6–21, January 2007.

[Needleman and Wuncsh 1970] Saul B. Needleman and Christian D. Wuncsh. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[Paracel 2000] Paracel. TRACETUNER, capturing the most information from the latest DNA sequencing systems. 2000.

[Pearson and Lipman 1988] William R Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, April 1988.

[Pedretti 2001] Kevin Thomas Pedretti. *Accurate, parallel clustering of EST (gene) sequences*. Master's thesis, The University of Iowa, 2001.

[Pertea *et al.* 2003] Geo Pertea, Xiaoqiu Huang, Feng Liang, Valentin Antonescu, Razvan Sultana, Svetlana Karamycheva, Yuandan Lee, Joseph White, Foo Cheung, Babak Parvizi Jennifer Tsai, and John Quackenbush. TIGR gene indices clustering tools (TGICL): a software system for fast clustering of large EST datasets. *Bioinformatics*, 19(5):651–652, 2003.

[Ptitsyn and Hide 2005] Andrey Ptitsyn and Winston Hide. CLU: A new algorithm for EST clustering. *BMC Bioinformatics*, 6, 2005.

[Ranchod 2005] Pravesh Ranchod. *Parallelisation of EST Clustering*. Master's thesis, Faculty of Science, University of the Witwatersrand, 2005.

[SANBI] SANBI. *http://www.sanbi.ac.za/benchmarks/benchmark10000.seq.gz*.

[Schuler 1997] Gregory D. Schuler. Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *Journal of Molecular Medicine*, 75(10):694–698, October 1997.

[Slater 2000] Guy St.C. Slater. *Algorithms for the Analysis of Expressed Sequence Tags*. PhD thesis, University of Cambridge, March 2000.

[Smit *et al.* 2007] A Smit, P. Green, G. Glusman, and R. Hubley. RepeatMasker. 2007.

http://www.repeatmasker.org/

[Smith and Waterman 1981] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[Tarjan 1975] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *ACM*, 22(2):215–225, April 1975.

[The C. elegans sequencing consortium 1998] The C. elegans sequencing consortium. Genome sequence of the nematode c. elegans: A platform for investigating biology. *Science*, 282(5396):2012–2018, December 1998.

[Verdun *et al.* 1998] Ramiro E. Verdun, Nelson Di Paolo, Turan P. Urmenyi, Edson Rondinelli, Alberto C. C. Frasch, and Daniel O. Sanchez. Gene discovery through expressed sequence tag sequencing in trypanosoma cruzi. *Infection and Immunity*, 66(11):5393–5398, November 1998.

[Vinga and Almeida 2003] S. Vinga and J. Almeida. Alignment-free sequence comparison – a review. *Bioinformatics*, 19(3):513–524, 2003.

[Zerbino and Birney 2008] D.R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18:821–829, 2008.

[Zhang *et al.* 1999] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1/2):203–214, 1999.