Using Interval Graphs for Solving Map Assembly Problems

by

James R. Randall

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
September 1997

A thesis submitted in conformity with the requirements
of the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © by James R. Randall 1997

Using Interval Graphs for Solving Map Assembly Problems

James R. Randall

Master of Science 1997

Department of Computer Science

University of Toronto

Abstract

This thesis is concerned with graph theory-based algorithms to handle the genetic map assembly problem. These algorithms are designed to build the genetic maps using only graph theoretical properties so as to better understand how much information can be obtained from the graph structure. In particular, several algorithms are developed using ideas from interval graphs and the asteroidal triple-free graphs. The core of the problem is dealing with the noise present in the data. This noise can be split into two different categories: false negatives (edges removed from a graph), and false positives (edges added to a graph). For the case of noise introduced as a result of false negatives, lexicographic breadth-first search is used to produce useful orderings even under heavy noise conditions. Three different algorithms are developed for dealing with false positives. The algorithms can detect the effects of false positives, but none are effective in removing this type of noise.

Acknowledgements

Special thanks to Derek Corneil and Tony Bonner for doing such a great job of supervising me. This thesis would not have been completed if not for their excellent work keeping me on the path.

As well, I would like to thank Lisa White for all the help through the years. I would also like to thank my brother and sister for all of their kind words of encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The fundamental blueprint upon which all of life is based is encoded in a molecule called deoxyribonucleic acid (DNA). These molecules (called chromosomes) are constructed from molecular building blocks called bases. The study of DNA and its components in both humans and other animals is critical to our understanding of how all organisms operate. This study requires a library of easily reproducible fragments of DNA that can be used as the basis for further tests and analysis. To organize such a large library, physical maps need to be constructed. These physical maps associate fragments of DNA with their corresponding position on the genome. The production of these maps can be divided into two phases: data generation, and map assembly. This thesis focuses on the latter problem.

The data used to build the maps is generated by experiments that determine if particular fragments of DNA overlap. Once it is known which DNA fragments overlap, we can then build a physical map that contains an ordered list of the fragments that is consistent with the overlap data (map assembly). This overlap data can be represented by a type of graph known as an interval graph.

The interval graphs are a well-studied family of graphs, and there are efficient algorithms for ordering their vertices such that they correspond directly with a set of intervals on a line. What makes the map-assembly problem difficult for these standard algorithms is the large amount of noise in the data. The noise greatly disrupts the structure of the corresponding graphs, breaking some of the necessary assumptions that these algorithms make.

We attempt to deal with the problem of noise in the data by taking more robust algorithms used in superfamilies of the interval graphs and modifying them to compensate for noise. This noise compensation is done using only graph-theoretical properties – the algorithms do not take advantage of any extra biological information that might be available. Such information is incomplete and not easy to come by, so an algorithm which does not require any such information would reduce the cost of solving the problem.

This thesis is divided into three primary chapters. Chapter two introduces the background necessary to more fully understand the problem. It covers both the biological and graph theory background needed, and describes how the graphs themselves are constructed. Chapter three discusses the genetic data itself. This includes an analysis of the noise that is present in the data and how it affects the structure of the graphs, how the sample genetic data is generated, and how the results of running the algorithms on the data were measured. In chapter four, the algorithms themselves are presented and the results are discussed. Chapter five contains the conclusions that were drawn from this work.

Chapter 2

Background

This chapter contains most of the background material that is used in further chapters of this thesis. How the physical maps are constructed and other biologial issues are discussed in Section 2.1. How the corresponding graphs are constructed is introduced in Section 2.2, as are many important properties of these graphs.

2.1 Genetic Background

Physical maps play an important role in the study of DNA. They have many different uses, such as locating genes and other genomic features, and are used in base sequencing. A good reference for learning more details about the map assembly problem and other related genetic issues is (Cantor, Casey and Spengler, 1992).

One of the first obstacles that must be overcome when generating physical maps is that the DNA fragments that make up the map must be copied. Most tests done on DNA tend to be very destructive to the DNA itself. In addition, all such tests are chemical reactions which require thousands, if not millions, of copies of the molecules involved to work properly. Once there are many copies of the fragments, it is possible to do the tests which will determine the ordering of the fragments.

2.1.1 Yeast Artificial Chromosomes

There are many different methods of copying genetic information, but the process that can handle the largest DNA fragments uses transformed yeast cells as the main method of duplication. This is done by inserting the piece of DNA to be copied into the middle of the yeast's own DNA, and then when the yeast duplicates itself, the inserted DNA is copied as well. Most mammalian chromosomes are too big to fit into a yeast cell, so we have to cut the DNA into smaller bits. This is accomplished using a type of chemical known as an enzyme. Enzymes are protein catalysts that can enhance a wide variety of biochemical reactions depending on the structure of the enzyme. For example, many enzymes cut DNA at every occurence of a particular sequence of bases.

For each piece of DNA that is to be copied we need a yeast cell. A yeast chromosome is cut at a particular spot, and the DNA fragment to be duplicated is then inserted into the cut point and sealed into place using another enzyme called a ligase (Figure 2.1).

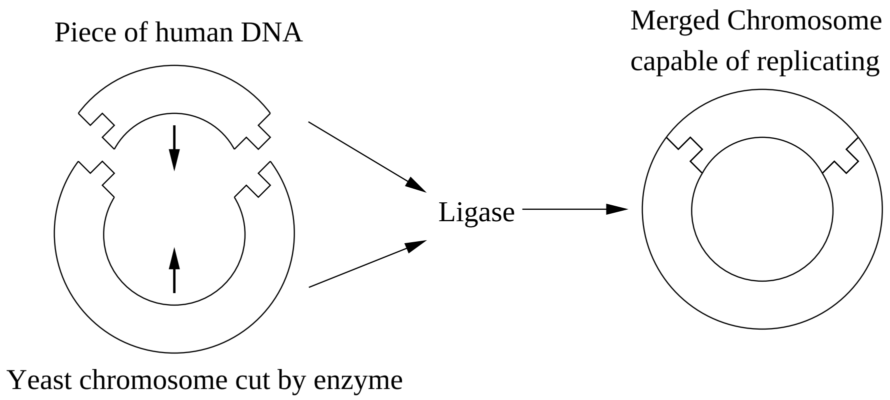

Figure 2.1: Inserting a DNA fragment Into a Yeast Chromosome

The result of this process is a yeast chromosome with the DNA fragment welded into it, so that when the yeast replicates, the DNA fragment will be replicated along with it. To use the DNA fragment after it has been copied, the chromosome that it is embedded in has to be removed from the yeast cell and the DNA fragment cut out of it. The modified chromosome is called a Yeast Artificial Chromosome, or YAC. The term YAC is also frequently used to refer to the DNA fragment itself, since the DNA fragment is the part of the process that researchers are most interested in. The rest of this thesis will use the term in this way.

There are a few interesting ramifications of this process. While it can duplicate large pieces of chromosome quite quickly, it also introduces further complications. As mentioned earlier, the process of cutting the initial chromosome pieces destroys the ordering of the pieces. As well, bits of the original chromosome are discarded and, as a result, are not copied. To compensate for this, additional YACs need to be created that both cover the missing regions and overlap with existing YACs. These can be made by either using a different cutting enzyme on a copy of the original chromosome or by using the same enzyme but controlling the cutting conditions so that the enzyme does not cut the chromosomes at each possible cutting site. The result is a library of easily duplicated YACs that are expected to cover most of the chromosome, and should contain enough overlap between the YACs to reconstruct the original ordering.

2.1.2 Ordering YACs

Ordering the YACs is done primarily by determining which YACs overlap, and thus which YACs are adjacent. Determining YAC overlap may seem to have an obvious solution, as once you have all of the YACs sequenced you could just compare the sequences to determine overlap. Unfortunately sequencing is an extremely expensive operation, and it is not feasible to determine overlap in this way. To solve this problem, extra tests to determine overlap can be done using special regions of the YAC called sequence tagged sites (STSs).

An STS is a small area on the YAC that contains a sequence of bases that are unique in the genome. As a result of this property, if two YACs both contain the same STS they must overlap. Another property that makes these sequences useful, is that they can be duplicated by a process known as the polymerase chain reaction (PCR). To be able to use PCR to duplicate the sequence, it is also necessary to be able to create a set of short sequences called primers, specific to the STS, that are used to initiate the reaction. PCR works by binding the primers to the piece of DNA that matches their bases and then using a special type of enzyme called polymerase that duplicates the sequence of DNA bound by the primers. While doing this, the polymerase also ends up duplicating the primers. As a result, there are two sections of identical DNA along with two sets of primers as well. This means that the reaction can easily be done repeatedly, getting exponential growth in the number of duplications. Since one PCR cycle only takes 5 to 10 minutes to complete, if there is a sequence in the DNA sample that matches the primers, it will not take long for a large number of duplications of it to take place.

In almost all of the data sets available today, there is not yet complete coverage of the chromosome. This means that when the YACs are reassembled, they form contiguous chunks of the chromosome, not the entire chromosome itself. These chunks are called contigs.

2.2 Graph Theory Background

Once we have all of the information from the STS probes, the next step is to use this information to construct an ordering of the STSs. Once we know how the STS probes are ordered, we can get the order of the YACs if we need it by checking which YACs contain the ordered STS probes. This information, abstracted into a graph, can be captured by a type of graph known as an interval graph. Interval graphs were originally inspired by constructing graphs from overlapping intervals on a straight line, a situation which abstractly models the genetic assembly problem well. An interval graph can be constructed by letting the nodes of the graph correspond to the STSs, and putting an edge between two nodes whenever the two STSs both occur in the same YAC (Figure 2.2). A good reference for learning more about interval and other related graphs is (Golumbic, 1980).

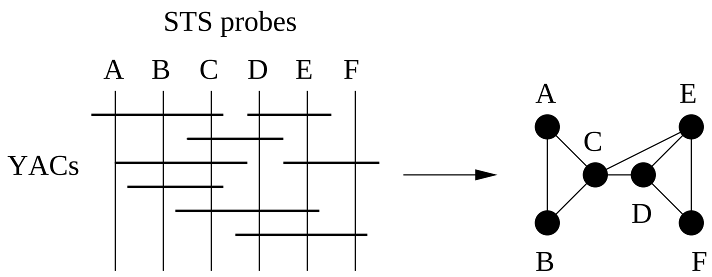

Figure 2.2: Constructing an Interval Graph from YAC/STS Data
The vertical lines represent STS probes, the horizontal lines YACs

Interval graphs have a number of interesting properties. They are chordal (or triangulated), meaning that every cycle of length four or greater has a chord. As well, there are numerous meaningful ways of ordering subcomponents of the interval graphs, as is evidenced by the following two theorems.

Theorem 2.1. (Gilmore and Hoffman, 1964) A graph G is an interval graph if and only if the maximal cliques of G can be linearly ordered such that, for every vertex v of G, the maximal cliques containing x occur consecutively.

Theorem 2.2. (Olariu, 1991) A graph G is an interval graph if and only if its vertices can be ordered so that for all vertices u, v, and w, if u is less than v and v is less than w and if uw is an edge then uv is an edge.

The Olariu ordering described in Theorem 2.2 can be used to create an interval representation for the graph. For every vertex v in a graph, its corresponding interval is $[i, j]$

where i is the position of v in the ordering, and j is the larger of i and the rightmost position of a neighbour of v.

The fact that interval graphs are based on intervals of a straight line would suggest that they should have some kind of linear structure. One potential description for this linearity is contained in asteroidal triple-free graphs (Corneil, Olariu and Stewart, 1997). An asteroidal triple is a set of 3 pair-wise non-adjacent vertices such that between any two there is a path that avoids the neighbourhood of the third (Figure 2.3). It is speculated that asteroidal triples capture the essence of non-linearity in the graph, so a graph without any asteroidal triples (AT-free) should exhibit linear structure. In fact, interval graphs are precisely the intersection of asteroidal triple-free and chordal graphs (Lekkerkerker and Boland, 1962).

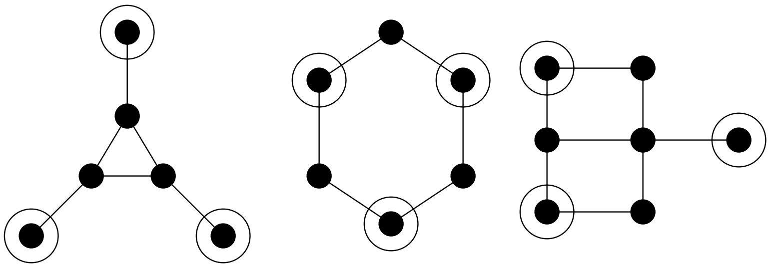

Figure 2.3: Some Asteroidal Triples

Asteroidal triple-free graphs and interval graphs have a wide range of properties and associated algorithms that will be useful later on. Of particular use is an algorithm called lexicographic breadth-first search (LBFS). As the name suggests, it is a variant of the well known breadth-first search (BFS). With BFS, as each vertex v is visited, its neighbours $N(v)$ are added to the queue of vertices to be visited in no special order. In LBFS, a restriction is placed on this order, in that vertices with early visited neighbours are favoured over vertices with recently visited neighbours (Table 2.1). The order in which the vertices are visited gives us an ordering of the vertices which will be very useful later on. Figure 2.4 gives an example of an LBFS ordering on an interval graph.

Some definitions are necessary to understand some of the additional properties that we are going to use. Most of the following definitions and propositions are taken from (Corneil, Olariu and Stewart, 1998a). A vertex v intercepts a path P if v is adjacent to at least one vertex of P. For any u, v in a graph G, $D(u, v)$ is the set of vertices of G that intercept all paths between u and v. Vertices u and v are unrelated with respect to a vertex y if

Algorithm LBFS(G, a)
Input: a connected graph G and a starting vertex a
Output: an ordering σ of the vertices of G
begin
    for all $v \in G$, label(v) $\leftarrow \Lambda$
    $n \leftarrow |G|$; label(a) $\leftarrow n$
    for index $\leftarrow$ 1 to n
        select an unnumbered vertex k with the lexicographically largest label
        $\sigma(k)$ = index
        for all unnumbered v in $N(k)$, append n − index to label(v)
    end for
end

Table 2.1: Pseudocode for the LBFS algorithm

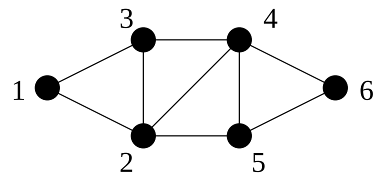

Figure 2.4: An Example of an LBFS ordering on an interval graph

$u \notin D(v, y)$ and $v \notin D(u, y)$. A vertex y of G is admissible if there are no vertices of G unrelated with respect to y. Intuitively, one would expect an admissible vertex to be near the end of a linearly-structured graph. Finally, let y be an admissible vertex in an interval graph G. Vertex y induces a partition of V into sets $Y_0, Y_1, \ldots, Y_p$, $(p \geq 0)$, called slices. These are defined as follows:

- $Y_0 = \{y\}$
- For all $1 \leq i \leq p$, $G_i = G \setminus \cup_{j=0}^{i-1} Y_j$
- For all $1 \leq i \leq p$, let Y_i be the vertices of G_i with the most neighbours in $G \setminus G_i$

And finally, we state three propositions that will be needed later in the thesis.

Proposition 2.3. (Corneil, Olariu and Stewart, 1998a) Let v be an arbitrary vertex of an AT-free graph G. The vertex numbered last by LBFS(v) is admissible in G.

Proposition 2.4. (Corneil, Olariu and Stewart, 1996) For an interval graph G, for every i, j and k with $0 \leq i < j < k \leq p$, if a vertex in Y_i is adjacent to a vertex in Y_k, then it is adjacent to all vertices in Y_j.

Proposition 2.5. (Corneil, Olariu and Stewart, 1996) For an interval graph G, let i and

j be such that $0 \leq i < j \leq p$. In every LBFS(y), the vertices in Y_i are numbered before the vertices in Y_j.

Chapter 3

Data Analysis

This chapter goes into more detail about various aspects of the genetic data. Section 3.1 describes the various types of noise in the data, and the effect that the noise has on the structure of the corresponding graphs. Section 3.2 describes the simulator used to artificially generate data with controlled levels of noise. Finally, section 3.3 outlines several difficulties with measuring how well noise is eliminated from the data, and develops the techniques used for this in the next chapter.

3.1 Errors in the Data

If the genetic data were 100 percent accurate and had no noise, then the problem of ordering the STSs would be much easier. There are several good algorithms, such as the one in (Corneil, Olariu and Stewart, 1998b), which would give a valid interval ordering and do so in linear time. The problem is that the data produced using current techniques contains a very large amount of noise. This noise is bad enough to cause standard interval graph algorithms to fail badly on the given data. The next interesting question is, then, what does the noise do to the underlying graph structure, and can other aspects of graph theory be used to find algorithms that are more reliable in handling noisy data.

The first step towards this goal is to analyse the noise itself. The noise can be broadly grouped into two categories: false positives and false negatives. Once we understand how these two different kinds of noise are produced, we can then start predicting their effect upon graph structure and possibly attempt to come up with better algorithms for dealing with the noise.

3.1.1 False Negatives

A false negative occurs when a test fails to report that a STS is contained in a YAC. There are many different sources of false negatives in the data. In fact, the false negative rate can get as high as 50 percent.

False negatives cause two types of problems for genome map assembly. They disrupt the local structure of the graphs, making local ordering harder, and they thin out the data, breaking up contigs. The second problem is not something that can be solved in this context, but the first problem can be dealt with.

With edges missing due to false negatives, the resulting graphs are no longer interval graphs. Small cycles usually no bigger than length four or five are introduced, and what should be cliques are turned into almost-cliques (Figure 3.1). It is possible for asteroidal triples to be generated, but the graphs are, on a large scale, still very linear.

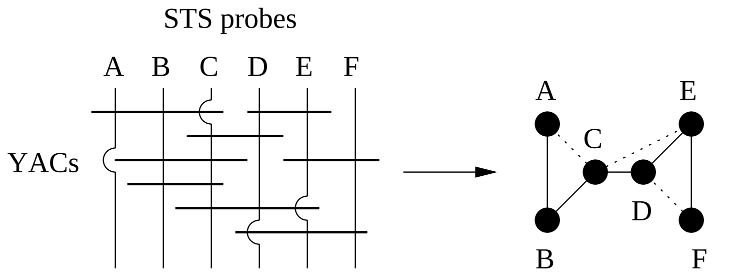

Figure 3.1: A Graph with False Negatives
The "bumps" in the vertical STS lines are the false negatives. The dotted lines in the graph are edges which are removed as a result of the false negatives.

3.1.2 False Positives

A false positive is a data point indicating that two genomic regions are adjacent when really they are not. There are many ways in which such false data can be generated, but the two primary sources in YAC data are from effects called repeat regions and chimerism.

When a DNA sequence appears more than once in a genome, it is said to repeat, and the various regions containing the sequence are called repeat regions (Figure 3.2). These occur naturally, and since they have the same repeated set of DNA, any STS that binds to one will bind to all. This means that widely separated parts of the genome will look as if they are at the same location. Since the repeat regions tend to be small, however, it is rare that more than one STS will hit an individual repeat region.

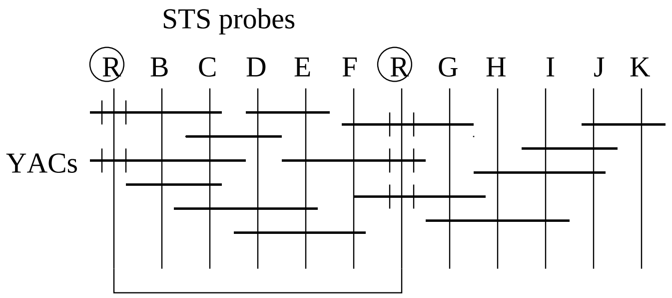

Figure 3.2: An Example of the Effects of Repeat Regions
The STS probe R is contained in a YAC repeat region, and thus adjacent to many more probes than it otherwise would be.

Chimerism occurs when two (or more) YACs appear to be one. There are at least two ways in which this can happen in the lab. The first occurs during the copying process. In this scenario, two or more different pieces of chromosome are unintentionally spliced together into a single continuous strand of DNA. This can occur as the result of the natural repair mechanisms in the DNA. The other way chimerism can arise is when test tubes are contaminated by samples from other test tubes. Both cases lead to several STSs being incorrectly marked as adjacent (Figure 3.3).

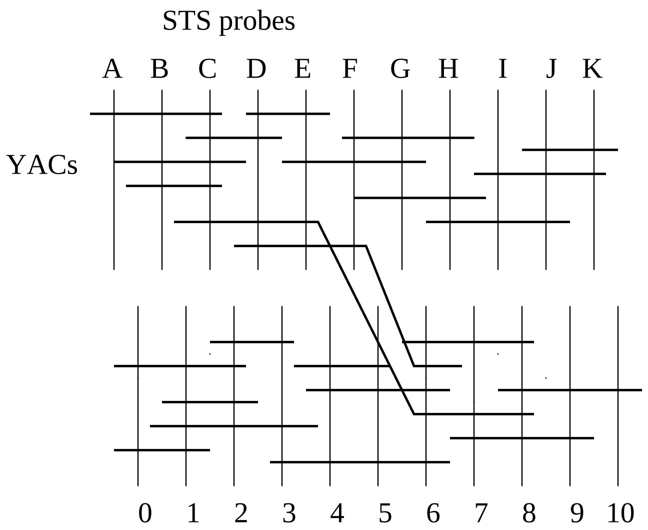

Figure 3.3: An Example of Chimeric YACs
The STS probes C-F appear to be close to probes 6-8 as a result of the two chimeric YACs.

In the raw data, the rate of false positives due to chimerism can commonly be as high as 40 percent, so when calculating if two STSs are adjacent, biologists typically use a so-called double-linkage strategy (Figure 3.4). With this, any *two* STSs must have at least two YACs in common before they are considered adjacent. This dramatically reduces the number of false adjacencies to a more manageable level, as the odds of having two chimeric YACs in the same area of the genome are very low (Arratia, Lander, Tavare and Waterman, 1991). Unfortunately, this strategy tends to force data sets that are thin to become disconnected, but the resulting significant reduction in false positives is almost always worth it.

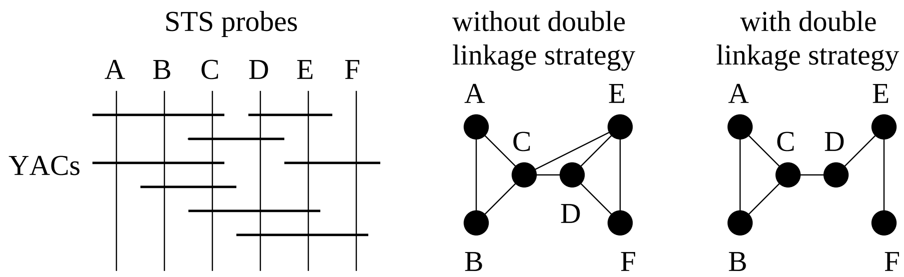

Figure 3.4: The Double Linkage Strategy

False positives in the data greatly disrupt the structure of the corresponding graphs. False positives tend to join up distant sections of the graph, making artificial "branches" in the otherwise linear structure of the graph. The effects on local graph structure are different depending on the type of false positive. Chimeric false positives tend to have several STSs involved on the misleading sequence of DNA, leading to wide branching structures (Figure 3.5). These structures are important enough for us to give them official definitions. A *branch point* is an area where two or more different areas of the genome are connected via false-positive edges. *Branches* are linear sections of a graph which emanate from a branch point. A branch is considered a *good branch* if, relative to an identified branch at the same branch point, it is from the same part of the genome. Similarly, a branch is *bad* if it is from a different part of the genome. It is the elimination of bad branches that will be the focus of our efforts to remove the false positives from the graph.

Since false positives from repeat regions tend to involve only one STS, this tends to create graphs that have a single node or cut point holding together the two separate sections of DNA (Figure 3.6). Other than the number of nodes involved in the branch point, the graph structure induced by false positives from repeat regions is the same as that of false positives from chimerism, i.e. these too result in branches in the graph.

With both kinds of false positives, almost all of the graphs produced will have asteroidal

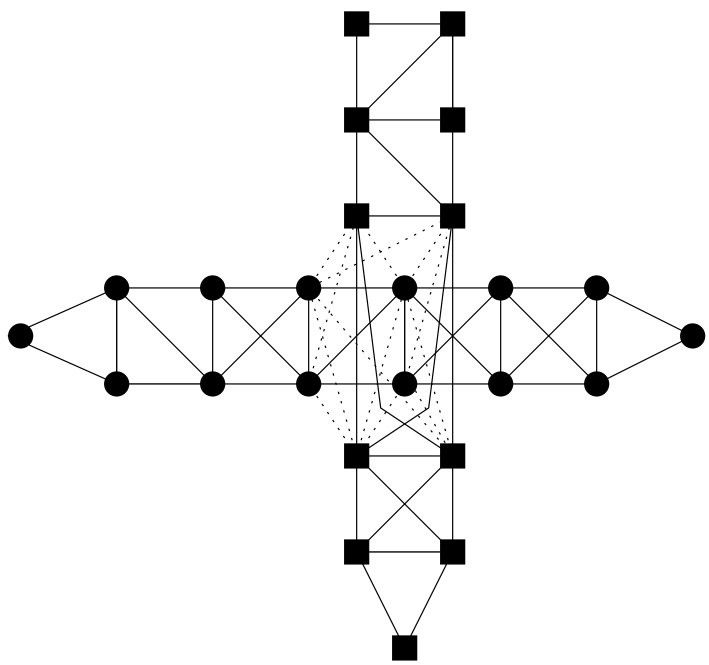

Figure 3.5: Chimeric Graph Structure
The horizontal sections of the graph have been incorrectly linked to the vertical sections of the graph by the dotted false-positive edges. If the horizontal sections of the graph were being ordered, then the vertical portions of the graph would be bad branches, and the horizontal section after the branch point would be a good branch.

triples embedded in them. Some of these asteroidal triples are cycles, from either a single false positive to an area later in the chromosome, or from multiple false positives. Most of the cycles tend to be very large, however.

Another important point is that the local structure is very much like an interval graph. In both Figure 3.5 and Figure 3.6, if you delete the edges due to false positives, thus deleting the bad branches, then the result is an interval graph. Unforunately, if you instead delete the edges that connect the good branch of the graph and keep only one of the bad branches, the result is also an interval graph. It is this point that is the biggest challenge for all of the algorithms which attempt to remove the false positives from the data. Since our algorithms use only graph theoretic information, they rely heavily on the local structure of the graphs as a guide.

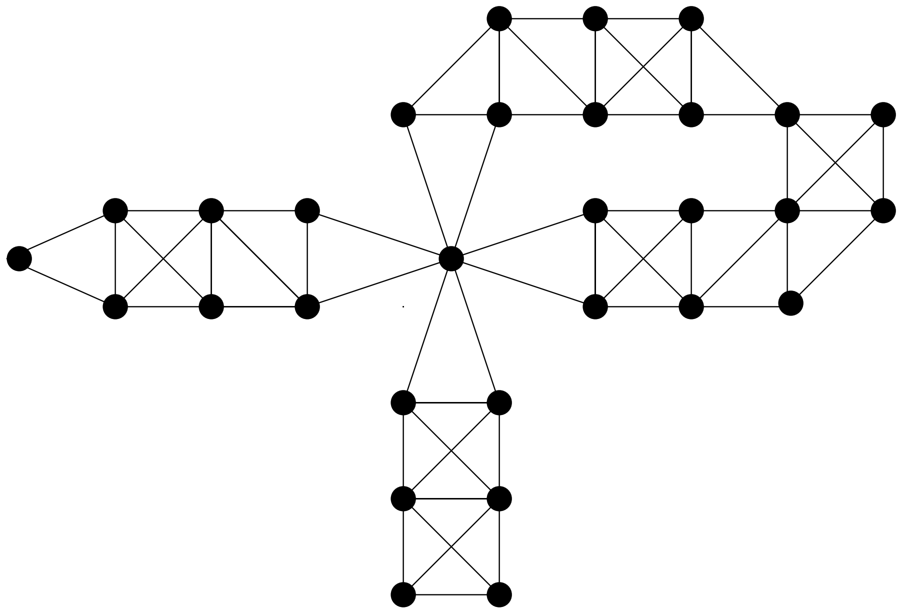

Figure 3.6: Repeat Region Graph Structure

3.2 Modelling noise

If we are to develop algorithms to order STS data, we will need data upon which to test the algorithms. At first glance, it might seem natural to use real genome mapping data. Unfortunately, for most data we do not know the correct ordering, which makes testing the algorithm very problematic. An easier solution from a testing point of view is to use simulated data. This has the additional advantage that we can produce data sets with only certain kinds of noise in them, which is of great use in isolating the effects of particular kinds of noise on the algorithms.

In this thesis, a simulator developed by Lincoln Stein at the Whitehead Institute/MIT Centre for Genome Research and extended by Eric Harley at the University of Toronto was used to generate data sets (Harley and Stein, 1996). This simulator produces STS-style adjacency data by building up a set of chromosomes, and randomly assigning them genetic properties in a way that models those found in real chromosomes. The chromosomes are split up into YACs, noise is added to the data, and STS adjacency data is generated from the result.

There are some assumptions made by this simulator, but in general they are similar to what is thought to be the case in human DNA. In particular, all the tests done in this thesis were done with 24 chromosomes with an average length of 200 million base pairs and a standard deviation of 70 million base pairs. Similarly, about .01 percent of each chromosome was modelled as repeat regions, with those regions having a mean size of 3000 base pairs and a standard deviation of 500 base pairs. The coverage was approximately 6, meaning that there was on average 6 YACs covering each point of the chromosome. Finally, there was an average of 4 STSs per million base pairs with a standard deviation of .3 STSs per million base pairs.

3.3 Measuring results

Not yet touched on is the issue of how to judge the quality of an STS ordering. Ideally a measure should indicate how close a particular assigned order is to the correct one, and it should abstract this into a single number accumulated over a series of tests. It is also important to make sure the length of the generated contigs is taken into account, since an assembly algorithm that makes long contigs is better than an algorithm that makes short ones.

This ideal measure is not an easy thing to come by however. Using the obvious method of mean and standard deviation on the distance from the desired position does not take into account how straight the resulting sequence is. For example, consider the following two sequences: $[8, 1, 2, 3, 4, 5, 6, 7]$, and $[1, 2, 5, 8, 7, 6, 4, 3]$. For the first sequence, the distance of each element from the desired position is 7 for the first node and 1 for the rest, which gives a mean distance of 1.75 and a standard deviation of 1.98. For the second sequence, the mean distance is 2, and the standard deviation is 1.8. So for both sequences, the mean and standard deviation are very similar, yet the first sequence is generally increasing and is arguably much "straighter" than the more random-looking second sequence. Ideally, one needs to take into account both the relative change in value as well as how close the value is to the desired position. In this thesis, rather than invent a new and perhaps statistically dubious measure of this property, a manual inspection of the generated sequences was done to ensure that things were approximately straight. Thus satisfied, mean and standard deviation were used to get a feel for how close the local structure was to being correct. A better and more automated measure of straightness would certainly be a useful addition for future work.

Another problem that affects both the algorithm and the measure of straightness is that there are some cases where it is simply not possible to determine what the correct ordering of the STSs is because there is not enough information present in the data. Take for example the graph in Figure 3.7. Assume that the vertex labelled 1 is the vertex chosen to begin the ordering. In this case, either vertex 2 or vertex 3 could be next in the ordering, and both orderings would be consistent with the data. As a result, it is impossible to tell which of these orderings corresponds to the actual order of the STSs on the genome. In this thesis, one possible order is arbitrarily chosen and the ambiguity is not taken into account when calculating error. For other applications though, a different approach may be necessary.

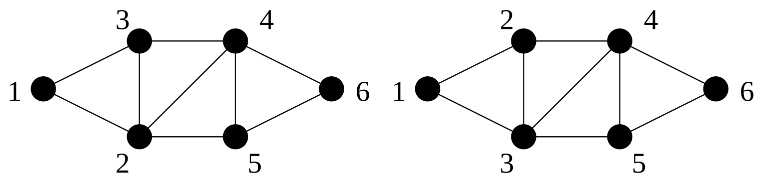

Figure 3.7: A Graph Which Has Multiple Orderings
The numbers correspond to different possible orderings of the vertices.

For algorithms that try to remove false positives by cutting edges out of a graph, counting the number of false positive edges cut relative to the number of false positive edges left in the graph becomes useful. There are two important weaknesses with this measure, however. First, an algorithm that cuts every edge in the graph would get all of the false positives as well, so it is important to make sure that there are not too many good edges being cut along with the bad. Second, it is possible that a bad branch of the graph could be cut without having the actual bad edges removed. If the algorithm left some false positive edges in the graph and cut all connecting edges further down the bad branch, good edges would be cut but the algorithm would have been reasonably successful in that it removed a bad branch.

An ideal measure to tell how well we are eliminating false positives would be one which, for every branch point in the graph, would tell us whether we had cut off bad branches or good branches. For algorithms such as ours, which incrementally add nodes to the ordering, we can approximate such a measure using a state-based heuristic (Figure 3.8 and Table 3.1. The underlying premise is that as the algorithm is traversing along the graph, we can be in two states: on a straight part of the graph, or at a branch point. If we are in the same area of the genome before and after a branch point, we must have followed the good branch, otherwise we followed a bad branch. The heuristic part of the measure comes into play when attempting to determine whether we are at a branch point or on a straight part. If we are at a branch point, the heuristic will start to think it is in a straight part when it sees three consecutive vertices that are very close to each other in the genome. Once in a straight part, it will switch back into thinking it is at a branch point if it sees any vertex that is from a distant part of the genome relative to the previously visited vertex. We can tell where a vertex is on the genome because our data is simulated.

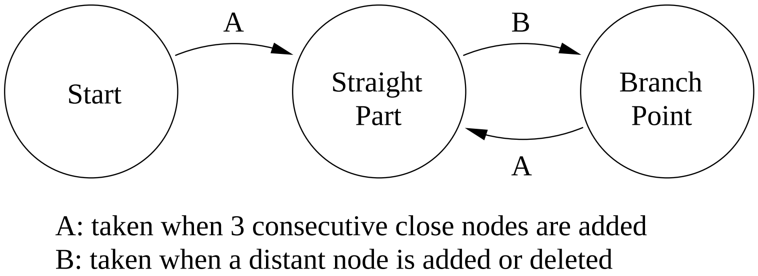

Figure 3.8: State Machine for the Branch-Cut Measuring Heuristic

Note that this measuring heuristic is designed to run synchronously with the graph edge-cutting algorithm, updating its state just after the point at which the algorithm decides to either include a vertex in the ordering or cut the edges connecting a vertex. This heuristic is not foolproof, but it is good enough to give us a general idea of how well our ordering algorithm is faring.

```
Algorithm branch-cut-measure(G)
Input: a vertex x which has been either just been ordered or disconnected
Output: 1 of: good branch taken, bad branch taken, no result
Static Variables:State (initially "start"), Counter (initially 0)
                 Last (initially Λ), Anchor (initially Λ)
begin
      set different to TRUE if x and Last are in different areas of the genome
      if x was disconnected and different isn't TRUE then exit the function.
      if State is "start" then
        if x was disconnected then exit the function
        if different is TRUE then
          Counter = 1;
        else
          Counter = Counter + 1;
          if Counter = 3 then set State to "straight"
        end if
      else if State is "straight" then
        if different is TRUE then
          set State to "branch point"
          set Counter to 1
          set Anchor to Last
        end if
      else (State is "branch point")
        if x was disconnected the exit the function
        if different is TRUE then
          set Counter to 1
        else
          Counter = Counter + 1
          if Counter = 3 then
            if Anchor and Last are in different areas of the genome then
              signal bad branch taken
            else
              signal good branch taken
            end if
            set State to "straight"
          end if
        end if
      end if
      if x was not disconnected then set Last to x
end
```

Table 3.1: Pseudocode for Branch-Cut Measuring Heuristic

Chapter 4

Algorithms and Experimental Results

Now that the data is better understood, it is time to look at algorithms to order the data. Section 4.1 develops an algorithm to deal with false negatives, while Section 4.2 develops three different algorithms to deal with false positives.

In all of the results presented in this chapter, only connected components that had at least 20 vertices were used for generating data. The actual number of such "large" connected components depended heavily on the noise in the data. Data that had 40 percent false negatives and no false positives had only about 300 large connected components, while some of the data used in later tests with 40 percent false positives and no false negatives had over 5000 large connected components.

4.1 False Negatives

Most traditional interval graph recognition algorithms are based on the idea in Theorem 2.1. They determine if a graph is chordal, calculate the maximal cliques, and then try to find a linear ordering of the maximal cliques. False negatives remove edges from these maximal cliques, so the basic structure that these algorithms depend on is disrupted. As a result, such algorithms are very sensitive to false negatives in the data. As well, they do not fail in such a way that an approximate ordering can be easily generated. To solve this problem, we look to AT-free graphs for inspiration. Since AT-free graphs exhibit many of the interesting linear characteristics of interval graphs, algorithms used in manipulating AT-free graphs may be useful in dealing with our noise-damaged interval graphs. In particular, LBFS is a particularly promising algorithm. LBFS is extremely important for AT-free graphs (Corneil, Olariu and Stewart, 1998a), as well as for chordal graph recognition (Rose, Tarjan and Lueker, 1976). Since interval graphs are the intersection of the AT-free and chordal graphs, one would expect LBFS to be important for interval graphs as well. In fact, this recently has been shown to be the case: there is a four-pass LBFS-based algorithm which recognizes and orders interval graphs in linear time (Corneil, Olariu and Stewart, 1998b). LBFS is robust in that it generates an ordering no matter how badly the data is distorted, and if the data is only distorted locally, as with false negatives, then hopefully the LBFS ordering will only be distorted locally. In addition, LBFS is very fast and easy to implement.

One catch in using LBFS is picking the starting point. Just picking an arbitrary starting point can lead to problems, since if the starting point is in the "middle" of the graph, then the ordering generated by LBFS will progress from the middle outwards, which is not at all what we want. To solve this, we need to do two passes with LBFS. The first pass starts at an arbitrary point, and is used solely to find the starting point for the next pass. By Proposition 2.3, if the graph is AT-free the last point visited on the first LBFS is admissible, and thus should make a good starting point. The ordering generated by the second LBFS is then used as the ordering for the vertices (Table 4.1).

Algorithm LBFS-order(G)
Input: a connected graph G
Output: an ordering σ of the vertices of G
begin
    select an arbitrary vertex x
    do LBFS(G, x), retaining final visited vertex y
    do LBFS(G, y), retaining final ordering σ
end

Table 4.1: Pseudocode for LBFS-order algorithm

The results of running this algorithm are summarized in Table 4.2. Each row in the table has an increasing number of edges missing from the graph. An error value was calculated for each vertex, which represents the number of positions the vertex was away from the correct position. The mean and standard deviation values in the table summarize the results. The percentage of vertices that were correctly numbered is shown, and the percentage of vertices that were within 1 place of the correct number is also presented.

Edges Missing	Mean	Stddev	% Correct	% Within 1
0%	0.10	0.42	92.4%	98.9%
10%	0.37	0.82	74.9%	93.3%
20%	0.65	1.06	59.4%	86.2%
30%	0.97	1.31	46.1%	77.1%
40%	1.27	1.53	36.6%	68.7%

Table 4.2: Results of LBFS-order on data with false negatives

In the noise free case, the sequencing is quite accurate, with 92 percent of the vertices sequenced correctly, and 99 percent of them within one position of being correct. As the noise rate is increased, the accuracy decreases (as expected), but it does so gradually. This behaviour confirms our prediction that LBFS is robust in the face of false negatives. Even with 40 percent of the edges missing from the graph, 69 percent of the vertices are within one position of being correct, with the mean and standard deviation of the error in position being only 1.27 and 1.53 respectively. This result is encouraging, and suggests that LBFS may be very useful in dealing with false negatives in genome mapping data.

4.2 False Positives

This type of noise is much harder to deal with than false negatives, primarily because the local structure is very much like that of an interval graph. What this means is that any algorithm which tries to choose a branch based solely on the existence of interval graph structure is not going to have much to go on. Finding an asteroidal triple will give an indication that there is a problem, but it is not immediately obvious that it can detect which edges are the cause of the problem. Such a negative conclusion warrants testing, as it would be very nice to have it proven wrong. To do this we need an algorithm that tries to identify branches due to false positives, and then cut edges connecting them without being confused by false negatives and without cutting too many correct edges. This can be accomplished by taking advantage of some of the linear properties of interval graphs. In particular, propositions 2.4 and 2.5 give us a very effective method for detecting "non-linear" behaviour in the graph. Using LBFS as the basis for the algorithm, the propositions come into play when selecting the next vertex to be numbered. Let y be the vertex selected by LBFS. y has i neighbours that will have already been numbered; in particular, let x be the neighbour that was numbered first. Since the slices in a STS graph are not likely to be too big, the propositions imply that y should be adjacent to most of the numbered vertices that were numbered after x, and thus if the graph is an interval graph, we should have $\sigma(y) \approx \sigma(x) + i$. This gives us an easy basis upon which to decide to cut edges; for any y, it is easy to find the corresponding i and x. All we need then is a parameter τ which roughly controls how tolerant we are of noise, and then we cut any edges connecting a node for which $\sigma(y) > \sigma(x) + i + \tau$ (Table 4.3).

```
Algorithm LBFS-cut(G, τ)
Input: a connected graph G and a tolerance parameter τ
Output: an ordering σ of some of the vertices of G
begin
      select an arbitrary vertex a in G
      do LBFS(G, a), retaining the final visited vertex b
      for all v ∈ G, label(v) ← Λ
      n ← |G|; label(b) ← n; σ(b) ← 1; index ← 2
      while there are labelled, unnumbered vertices
           select an unnumbered vertex y with the lexicographically largest label
           let i be the number of numbered neighbours of y
           let x be the first numbered neighbour vertex of y
           if index > σ(x) + i + τ then
             cut edges connecting numbered vertices to y
             label(y) ← Λ
           else
             σ(y) = index
             for all unnumbered v in N(y), append n − index to label(v)
             index ← index + 1
           end if
      end while
end
```

Table 4.3: Pseudocode for the LBFS-cut algorithm

This algorithm is a good test of the inherent ability of LBFS to order graphs with false positives, since for it to work in this context, it has to order the vertices as correctly as possible for as long as it can. In other words, when it reaches a branch point it should follow the correct branch at first, and then after it has travelled as far as the breadth-first nature of the search will allow, it only then should go back and number vertices on incorrect branches. If it does not do this, then the cutting rule we have developed stands as good a chance of cutting good branches as bad.

The results of running this program are summarized in Table 4.4 and Table 4.5. The first column is the percentage of YACS in the data set that contain false positives. The second and third columns are the results of running the branch-cut measuring heuristic developed in section 3.3 of the previous chapter. The second column contains the number of bad branches kept in the graph, the third contains the number of times that the bad branches were cut away and the good branch kept in the graph. The remaining columns give the number of edges in the graph, the number of edges which connect STSs from distant sections of the genome (called false edges, since they are produced by false positives), the number of edges cut which connect nearby STSs, and finally the number of edges cut which connect STSs from distant areas of the genome.

Error Rate	# Bad Branches	# Good Branches	Total Edges	False Edges	Good Edges Cut	False Edges Cut
0%	0	0	225318	0	0	0
10%	140	82	286578	3133	1575	607
20%	554	342	357470	10715	4800	1141
30%	1718	919	468383	32321	12385	2903
40%	4781	2174	692456	99350	29751	10544

Table 4.4: Results of tolerance 2 LBFS-cut on data with false positives and no false negatives

Error Rate	# Bad Branches	# Good Branches	Total Edges	False Edges	Good Edges Cut	False Edges Cut
0%	0	0	14655	0	622	0
10%	35	37	39110	587	1587	270
20%	162	92	77360	1633	3259	583
30%	621	350	150899	5498	6984	1392
40%	2018	1137	284974	18569	14803	4142

Table 4.5: Results of tolerance 2 LBFS-cut on data with false positives and 40% false negatives

The algorithm is well behaved in that it does not cut any edges unnecessarily when there is no noise, and only eliminates 4% of the edges when the noise consists of 40% false negatives. However, it does no better than random chance in identifying false branches. The algorithm has succeeded partially in that it seems to be correctly finding branch points and trimming away branches without unacceptable collateral damage, but it fails on the primary goal.

One issue that has not been dealt with is the choice of the tolerance parameter τ. The value that was used for the test runs in this thesis was 2. Smaller values caused significantly more edges to be cut without any improvement in false positive identification, and at larger values (4 and up) the algorithm doesn't cut agressively enough, causing many unwanted edges to be included in the ordering.

The poor performance of this algorithm could be attributed to a couple of different things. Our initial analysis could be correct, and there really isn't enough local information in the graphs for the program to deal properly with false positives. On the other hand, it could be that our graphs are just too thin, so that the breadth-first search mechanism is forcing us to travel along the unwanted edges before the cutting rule can reliably kick in. To test this hypothesis, we need a search that will not necessarily behave in a breadth-first manner, but still knows about local structure and has some hope of ordering the nodes properly. Since interval graphs are the intersection of AT-free and chordal graphs, a possible approach is to take an algorithm which is effective on chordal graphs and rely on our AT-free inspired cutting rule to enforce linearity on the graph. This will undoubtedly lead to more good edges being cut, but the hope is that it is close enough for our purposes. Ultimately if it gets the false-positive edges cut without causing too much collateral damage, then the edges are being lost for a good cause.

The algorithm that we will use to replace LBFS is one that chooses the next vertex to be numbered based on having the largest number of neighbours in the already numbered vertices. This is sometimes called MCC, since it picks the vertex with the *maximum cardinality* neighbourhood in its connected *component* (Shier, 1984). This is simple to implement, and while picking the most strongly connected vertex as the next to add is not biologically guaranteed to be the correct, it is a reasonable guess (Table 4.6).

There is a slight change that we can do to MCC that should make it more attractive for our purposes. In the unmodified version of the algorithm ties are broken arbitrarily, but in our version we have a tie-breaking rule. If ties are broken by favouring the element with the highest minimum-numbered neighbour (i.e. largest $\sigma(x)$ in the discussion of LBFS-cut), then this will hopefully encourage the algorithm to follow one branch longer.

Unfortunately, this approach doesn't work very well. Looking at the data in Table 4.7, MCC-cut is actually significantly worse then LBFS-cut. As expected it is more destructive than LBFS-cut, cutting 6.6% of the edges in the noise-free case. When false negatives are

Algorithm MCC-cut(G, τ)
Input: a connected graph G and a tolerance parameter τ
Output: an ordering σ of some of the vertices of G
begin

```
    select an arbitrary vertex a in G
    do LBFS(G, a), retaining the final visited vertex b
    index ← 0; N ← Λ; y ← b
    while y ≠ Λ
        index ← index + 1; σ(y) ← index
        for all neighbours v of y, if v ∉ N, add v to N
        loop
            make a sublist N' of vertices v with most numbered neighbours
            for all v ∈ N', min(v) ← w where w is the first numbered neighbour of v
            select x from N' such that σ(min(x)) is as large as possible
            remove x from N
            let i be the number of numbered neighbours of y
            if index ≤ σ(min(x)) + i + τ then break from inner loop
            cut edges connecting numbered vertices to y
        end loop
    end while
```

end

Table 4.6: Pseudocode for MCC-cut algorithm

added to the picture, MCC-cut cuts 16% of the edges, even with no false positives present! Looking at the branch-cut measuring heuristic, MCC-cut manages to create statistics that are even worse than LBFS-cut's.

Error Rate	# Bad Branches	# Good Branches	Total Edges	False Edges	Good Edges Cut	False Edges Cut
0%	0	0	254377	0	16727	0
10%	219	72	323065	3539	22684	751
20%	1027	269	415202	12531	30477	1640
30%	2846	791	514238	36097	45350	4103
40%	6833	1818	693095	101743	72345	13927

Table 4.7: Results of tolerance 2 MCC-cut on data with false positives and 0% false negatives

Our last attempt at trying to deal with these vexing false positives is to use LBFS and a method which more explicitly attempts to get rid of the unwanted edges, rather than solely relying on ordering properties of the search. The most promising approach is based on an assumption about the relative frequency of YACs with false positives. If we can assume that the number of vertices on a false branch of any branch point is usually less than the number on the good branch, then we have a graph-theoretic basis on which to choose edges to delete. Note that this is by no means guaranteed biologically, but it is at least a plausible guess about what the structure of the data might be.

In this algorithm, LBFS is modified by adding a check at the end of the loop. This check walks through all unnumbered neighbours of the numbered vertices and, considering only the edges between these vertices, groups them into connected components. For noise-free graphs, there will usually be only one connected component. A branch from a false positive, however, would cause the neighbours to become disconnected, at which point we can apply our heuristic and cut the edges leading to the smaller connected component (Table 4.8).

Algorithm CC-cut(G)
Input: a connected graph G
Output: an ordering σ of some of the vertices of G
begin
    select an arbitrary vertex a in G
    do LBFS(G, a), retaining the final visited vertex b
    for all $v \in G$, label(v) $\leftarrow \Lambda$
    $n \leftarrow |G|$; label(b) $\leftarrow n$; $\sigma(b) \leftarrow 1$; index $\leftarrow 2$
    while there are unnumbered, labelled vertices
        select an unnumbered vertex x with the lexicographically largest label
        $\sigma(x) =$ index
        for all unnumbered v in $N(x)$, append $n -$ index to label(v)
        build a list L of connected components using all labelled unnumbered vertices
                    and their unnumbered neighbours
        let m be the size of the largest connected component in L
        for all connected components C in L with size less than m
            for all v in C
                cut edges connected numbered vertices to v
                label(v) $\leftarrow \Lambda$
            end for
        end for
    end while
end

Table 4.8: Pseudocode for CC-cut algorithm

Sadly, this did not work well in practice either. The results of running this program are summarized in Table 4.9. This algorithm is well behaved in much the same way as LBFS-cut was: it does not overly cut up the data, and it degrades well as false negatives are added. However, it does very poorly with the branch-cut measuring heuristic. The most likely reason why this algorithm failed to generate better numbers goes back to our assumption that the number of vertices on the false branch of any branch point is usually less than the number on the good branch.

Error Rate	# Bad Branches	# Good Branches	Total Edges	False Edges	Good Edges Cut	False Edges Cut
0%	0	0	225318	0	0	0
10%	153	43	287796	3191	1622	697
20%	556	165	360437	11064	5534	2093
30%	1614	459	471065	32134	13551	5494
40%	4352	1138	697004	94100	34721	15644

Table 4.9: Results of CC-cut on data with false positives and 0% false negatives

In summary, none of the three algorithms developed here worked very well at eliminating false positives. It seems to be very difficult for this style of algorithm, using only the local structure of the graphs, to distinguish between false-positive edges and their normal counterparts.

Chapter 5

Conclusions

This thesis has produced mixed results. Algorithms such as LBFS work very well in dealing with false negatives; in the face of heavy disruption, the accuracy rate was quite good. False positives were a different story altogether; none of the three types of algorithms that were tried were able to deal successfully with the false positives in the data.

The algorithms presented in Chapter 4 might be useful in solving a restricted subproblem such as false positives from repeat regions only, but the algorithms would have to be modified to take advantage of this. To solve this problem in general, a different approach is needed. It might yet be possible to have an algorithm which can remove false positives using graph theory alone, but given that the local structure around such false positives is so much like an interval graph, it is hard to imagine what that algorithm might be.

A more promising approach might augment one of the approaches tried here to take advantage of extra genetic information. The LBFS-based algorithms are robust and able to handle false negatives. Not only that, while they cannot remove the false positives, there is strong reason to believe that they are very good at identifying the branch points where the false positives occur. If data from other types of genetic experiments could be used to help detect which branch is false, this would ameliorate the primary weakness of these algorithms. This kind of approach, very much like the one developed in (Harley, Bonner and Goodman, 1996), is the type of solution that holds the most promise for dealing with this particularly pernicious type of noise.

References

R. Arratia, E.S. Lander, S. Tavare and M.S. Waterman, Genomic mapping by anchoring random clones: a mathematical analysis, *Genomics*, **11**, (1991), 806-827.

K.S. Booth and G.S. Lueker, Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms, *Journal of Computer and System Sciences*, **13**, (1976), 335-379.

C. Cantor, D. Casey and S. Spengler, Primer on Molecular Genetics, *Department of Energy Human Genome 1991-1992 Program Report*, (June 1992).

G.A. Cheston and D.G. Corneil, Graph property update algorithms and their application to distance matrices, *Informatics*, **20**, (1982), 178-201.

D.G. Corneil, S. Olariu, and L. Stewart, Private communication, (1996).

D.G. Corneil, S. Olariu, and L. Stewart, Asteroidal triple-free graphs, *SIAM Journal of Discrete Mathematics*, **10**, (1997), 399-430.

D.G. Corneil, S. Olariu, and L. Stewart, Linear time algorithms for dominating pairs in asteroidal triple-free graphs, to appear, *SIAM Journal on Computation*, (1998).

D.G. Corneil, S. Olariu, and L. Stewart, The ultimate interval graph recognition algorithm? (extended abstract), to appear, *Proceedings of the Symposium on Discrete Algoritms*, (1998).

P.C. Gilmore and A.J. Hoffman, A characterization of comparibility graphs and interval graphs, *Canadian Journal of Mathematics*, **16**, (1964), 539-548.

M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, (1980).

E. Harley and A.J. Bonner, A flexible approach to genome map assembly, *Proceedings of the International Symposium on Intelligent Systems for Molecular Biology*, (1994), 161-169.

E. Harley, A.J. Bonner and N. Goodman, Good Maps are Straight, *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, (1996), 88-97.

E. Harley and L.D. Stein, Personal communication, (1996).

N. Korte and R.H. Möhring, An incremental linear-time algorithm for recognizing interval graphs, *SIAM Journal on Computing*, **18**, (1989), 68-81.

C.G. Lekkerkerker and J.C. Boland, Representation of a finite graph by a set of intervals on the real line, *Fundamenta Mathmaticae*, **51**, (1962), 45-64.

S. Olariu, An optimal greedy heuristic to color interval graphs, *Information Processing Letters*, **37**, (1991), 65-80.

D.J. Rose, R.E. Tarjan and G.S. Lueker, Algorithmic aspects of vertex elimination of graphs, *SIAM Journal on Computing*, **5**, (1976), 266-283.

D.R. Shier, Some aspects of perfect elimination orderings in chordal graphs, *Discrete Applied Mathematics*, **7**, (1984), 325-331.

R. Tamarin and R.W. Leavitt, *Principles of Genetics*, Wm. C. Brown Publishers, (1991).

J.D. Watson, M. Gilman, J. Witkowski and M. Zoller, *Recombinant DNA*, Scientific American Books, (1992).