Graph Algorithms for Assembling Integrated Genome Maps

by

Eric Richard Harley

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

Abstract

Graph Algorithms for Assembling Integrated Genome Maps

Eric Richard Harley

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2003

This dissertation is a computational study in the field of bioinformatics, a relatively new discipline at the intersection of computer science and biological science. As such, the dissertation contributes to each of these disciplines. The problems we address within biological science are in the area of physical mapping, i.e., mapping experimentally-produced fragments of a genome to their original locations in the genome. The thesis makes two primary contributions in this area: (*i*) a method to improve contig identification and verification, and (*ii*) a method which improves contig formation by uniformly integrating probe and overlap data. The problems addressed within computer science arise in the above application and are in the area of heuristic graph algorithms. The thesis makes three primary contributions in this area: (*i*) a modification of a well-known algorithm for listing maximal cliques, which markedly improves its efficiency on large, sparse graphs; (*ii*) a new clustering algorithm which is appropriate for "noisy" interval graphs; and (*iii*) a method which reveals piecewise linearity in a noisy interval graph.

The methods and algorithms are tested on human genome mapping data produced by the Human Genome Project and also on simulated data. The results from applying our structure graph algorithm to Sequenced Tag Site (STS) mapping data reveal a network-like pattern where linear segments often correlate with data from continuous pieces (contigs) of the genome. The underlying interval graph nature of the probe mapping data is thus revealed, thereby facilitating identification and extraction of contigs.

Results from the application of our "virtual probe" technique for integrating probe and overlap data indicate that this method can produce longer double-linkage contigs than STS probes alone, and that it provides a new way of assessing overlap data. Results of experiments on enumeration of cliques in large, sparse graphs representing genome mapping data show that our modification of the Bron-Kerbosch algorithm requires time which is approximately linear in the number of vertices, whereas other tested algorithms had worse than linear time complexity. Results from tests of the clustering algorithm on simulated noisy overlap data indicate that it effectively counters the increase in clique numbers caused by false negatives.

Dedication

For my father Ralph Harley, my wife Ewa, our children Zenon, Daniel, Konrad and Adam, and my siblings Peter, Sandi, Susan, Calvin, Bonnie and Robin.

Acknowledgements

I gratefully acknowledge the help of the following people. My supervisor Dr. Anthony Bonner has spent countless hours with me over the years, discussing results and suggesting new experiments. His enthusiasm and attention to detail have been remarkable. The members of my departmental committee, Dr. Ken Sevcik, Dr. Derek Corneil, Dr. Alberto Mendelzon, and Dr. Ricky Chan provided me many helpful suggestions and criticisms, which certainly improved the quality of this thesis. Dr. Corneil's expert counsel regarding interval graph theory and graph theory in general played a very important role in shaping this thesis. Dr. Nathan Goodman has co-authored several papers with Dr. Bonner and me, and he has been an invaluable source of ideas, knowledge and cheerful support. Dr. Lincoln Stein helped me with the genome simulator and answered many technical questions regarding the Whitehead genomic data. Dr. Mendelzon's Hy+ group, especially Mariano Consens and Dimitra Vista, were a ready source of advice and consultation regarding their excellent Hy+ data visualization system. The LaTeX2e style and skeleton files prepared by Francois Pitt were used to format this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Bioinformatics

The central thesis of this dissertation is that analysis of physical mapping data for genomes can be improved by methods based in computational graph theory. Applying graph theory to this biological problem presents challenges which lead to improved and new algorithms in computer science, while simultaneously contributing to genomics. Thus, this dissertation in computer science is also a study in the field of bioinformatics, a relatively new discipline at the intersection of computer science and biological science.

At first glance, the intersection of these two disciplines appears empty. Biological science is the study of life, which might be defined extensionally as a set which precisely contains you and all of your relatives (since it appears that all forms of life are descendants of a single cell which assembled on earth over 3 billion years ago). Computer science is the study of computers and computable problems, where computers are machines which can be programmed to perform calculations and manipulate data (through input, storage, retrieval, transformation and output). Clearly, computers are needed to store, manage and analyze the huge amounts of data generated in the life sciences, but basic applications of databases and programming are not in general studies of computable problems. If this were the extent of the overlap, then bioinformatics would still be computational biology, but it would not involve basic computer science. Computer science is drawn into the picture, however, because biological applications typically lead to problems which require innovation, theoretical insight and new algorithms.

For example, an application which queries biological data stored in many heterogeneous databases around the world motivates research into better ways to integrate, represent and query data across multiple databases [24]. An application which compares and aligns sequences stored in protein and DNA sequence databases stimulates research into the problem of multiple string alignment [49]. Applications which analyze noisy, combinatoric data from experimental protocols in genomics stimulate fundamental research into graph and combinatoric algorithms which aims for efficient, approximate solutions with error bounds [51]. These are just a few of many examples where the algorithmic and organizational challenges of computational applications in biology lead to advances in both biology and computer science, creating an exciting nonempty intersection, bioinformatics.

Research in bioinformatics begins after primary data have been collected, which means that computational biologists often rely on collaboration and free access to experimental information in a variety of public databases [78]. This dissertation is no exception — it is based on data and advice from scientists working on the Human Genome Project (HGP) at the Centre d'Etudes du Polymorphisme Humain and Généthon (CEPH/Généthon) and Whitehead Institute for Biomedical Research/MIT Center for Genome Research (WI/MIT). This collaboration was possible because of the laudable data release policy of HGP, whereby primary data are made available to the public shortly after collection. Primary data examined in this thesis were obtained from the public ftp sites of the above genome centers during the mapping phase of the Human Genome Project.

1.2 Graph Terminology

This section lists relevant definitions from graph theory (e.g., see [41]) and describes the nomenclature used throughout this thesis. Let an arbitrary graph G be denoted by $G = (V, E)$, where V is the set of vertices of the graph G, and $E \subseteq V \times V$ is the set of edges. Thus, the presence of an undirected edge between vertex i and vertex j is indicated by $(i, j) \in E$.

I will use the following conventional symbols to quantify aspects of a graph $G = (V, E)$:

- $n = |V|$ denotes the number of vertices;
- $m = |E|$ denotes the number of edges;
- c denotes the number of maximal cliques (defined below);
- μ denotes the number of maximal independent sets (defined below);
- $\rho = 100 \times 2m/(n(n-1))$, denotes the edge density, i.e., the percent of edges in a graph relative to the maximum number;
- $Adj(x) = \{v \in V | (x, v) \in E\}$, i.e, the vertices adjacent to x, also referred to as the neighbours of x in G.

A *clique* of G is defined to be a complete subgraph, i.e., a subgraph in which each pair of nodes is joined by an edge. Thus, $C = (S, F)$ is a clique of G if $S \subseteq V$, $F \subseteq E$ and $\forall i, j \in S, (i, j) \in F$ if $i \neq j$. A *maximal clique* is a clique which is not a proper subgraph of another clique. The terms clique and maximal clique will be used interchangeably when it is clear that I mean maximal clique. A complete graph on n nodes is denoted K_n. Figure 1.1(a) shows an example of a small undirected graph, and panel (c) shows its maximal cliques. The maximal cliques are shown here as subgraphs and sets. For simplicity, I usually use the set representation. Finding cliques in a graph G is a problem which arises frequently in both applications and theory of graphs. In applications, the

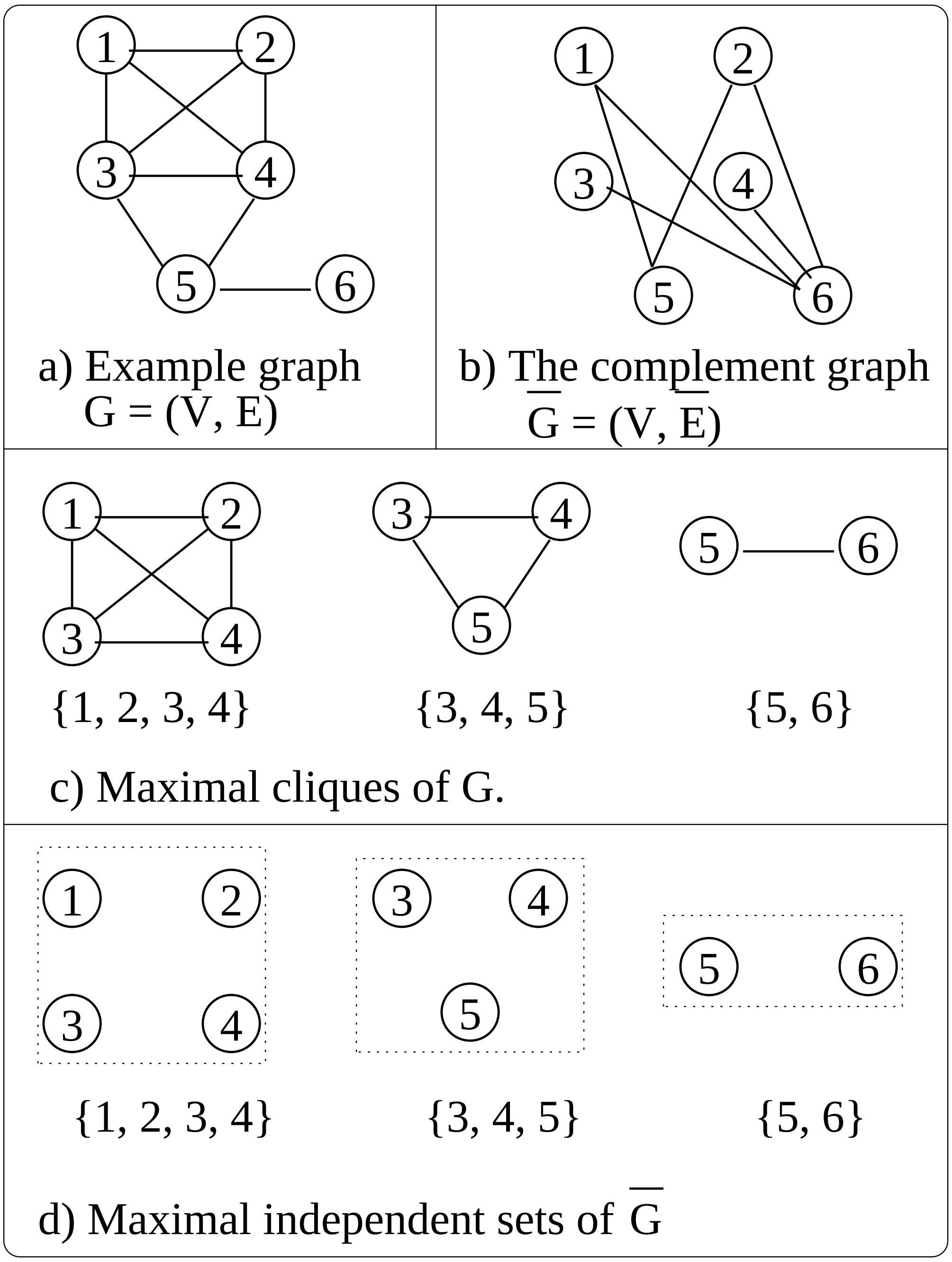


Figure 1.1: (a) A small undirected graph G. (b) The complement graph $\overline{G}$. (c) The maximal cliques of G. (d) The maximal independent sets of $\overline{G}$.

edges in a graph typically represent associations between the entities that are represented by the nodes. For example, if the nodes represent people, then edges might represent friendships, or if the nodes represent words, then edges might represent common meaning. In general, cliques represent groups of mutually related entities.

The complement of a graph G is a graph $\overline{G} = (V, \overline{E})$, which has the same vertex set as G, but the complementary set of edges, i.e., $(i, j) \in \overline{E}$ if and only if $(i, j) \notin E$, for all $i, j \in V$. An *independent set* is a subgraph in which no pair of nodes is connected by an edge. Thus, $I = (S, F)$ is an independent set of G if $S \subseteq V$, $F \subseteq E$ and $\forall i, j \in S, (i, j) \notin F$. A maximal independent set (MIS) is an independent set which is not a proper subgraph of another maximal independent set. Clearly, the vertex set of a maximal clique in G is equivalent to the vertex set of a maximal independent set in the complement graph, $\overline{G}$. This is illustrated in Figure 1.1, where panel (b) shows the complement of the graph in panel (a). The maximal independent sets of the complement graph (panel (d)) are the same as the maximal cliques in panel (c), when represented as sets (rather than subgraphs). An algorithm which can list the cliques of a graph G can also be made to list the independent sets of G simply by supplying the algorithm with the complement of G, $\overline{G}$. Similarly, an algorithm for listing independent sets can also list maximal cliques.

A graph can be represented in a computer using an adjacency matrix or an adjacency list, as shown in Figure 1.2. An adjacency matrix is an $n \times n$ binary array, where n is the number of nodes. The value 1 at position $[i, j]$ indicates that there is an edge between nodes i and j, and a value 0 indicates the contrary. This representation allows fast verification as to whether i is connected to j, and comparatively slow listing of all the neighbours of i. The main disadvantage of the matrix representation is that it can require much more space to represent a large sparse graph than the corresponding adjacency list representation.

The adjacency list in Figure 1.2(c) is shown as a linked data structure, consisting of

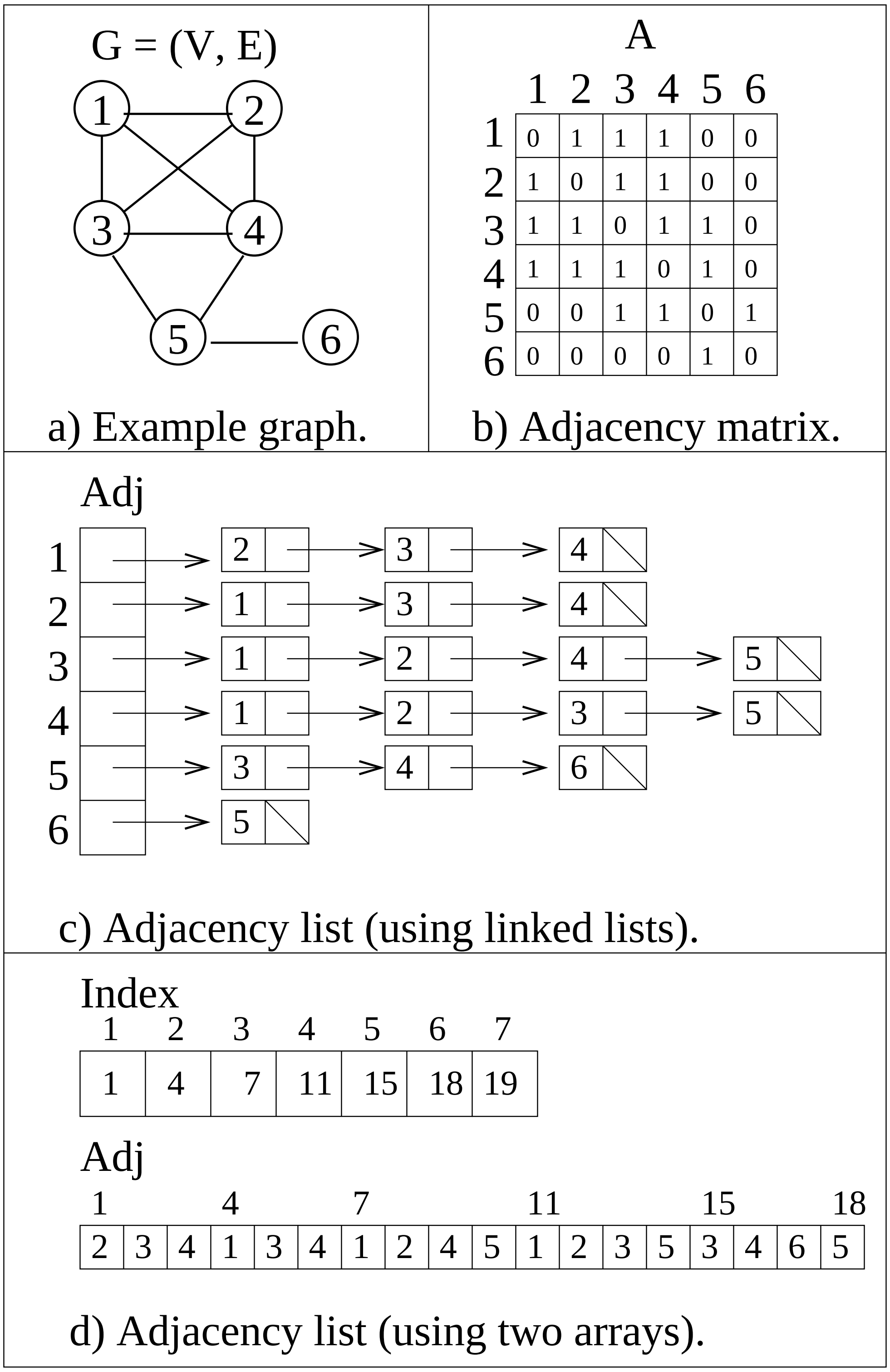


Figure 1.2: (a) Example of a small undirected graph G. (b) The adjacency matrix representation of G. (c) An adjacency list representation of G, using linked lists. (d) An adjacency list representation using two arrays.

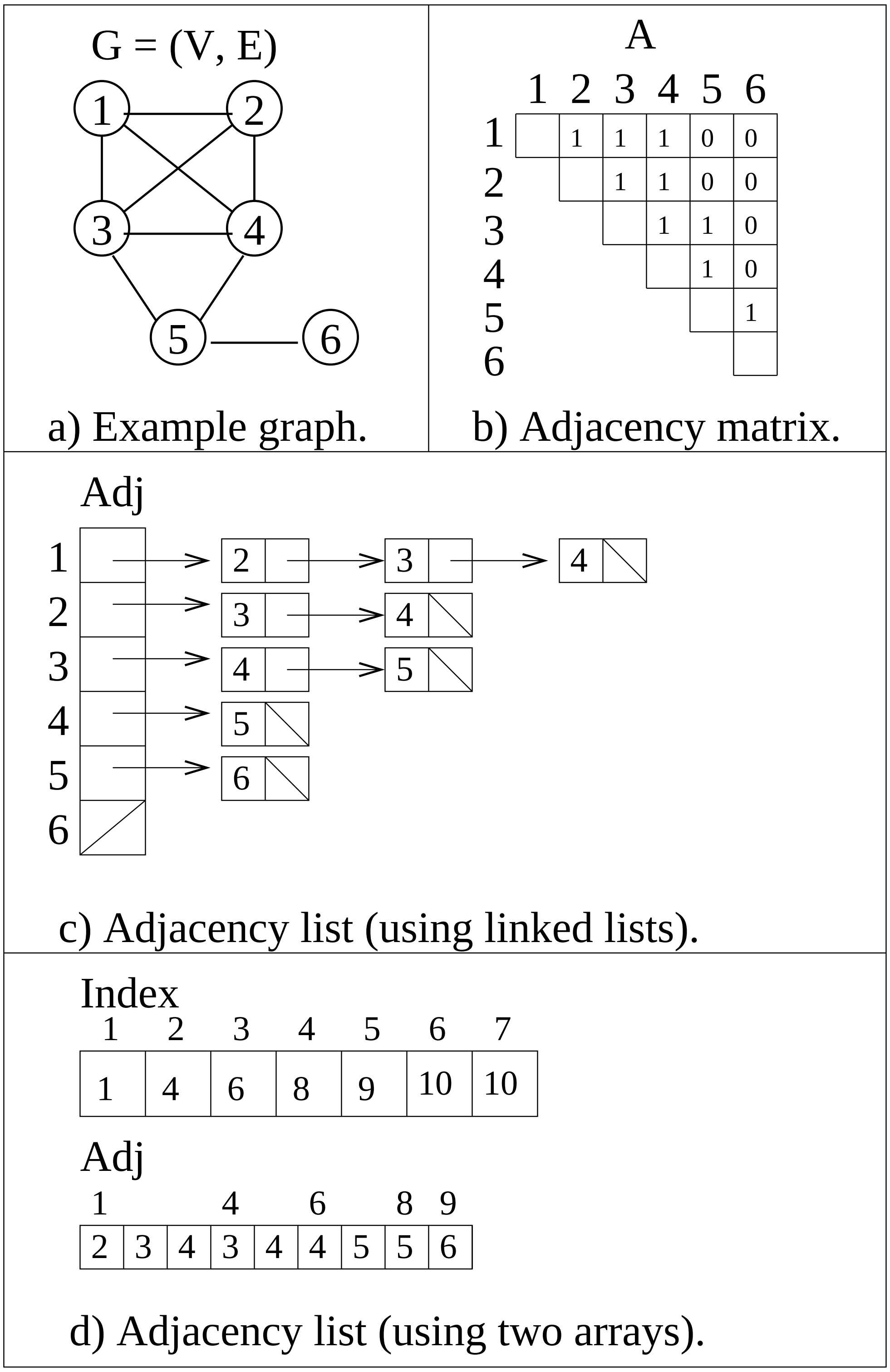


Figure 1.3: Adjacency list with abbreviated adjacency information. Adjacency of two vertices is checked as in the previous figure, after the preliminary step of swapping the vertices if the first vertex is larger than the second.

a one-dimensional array $Adj[n]$ of pointers to *Nodes*, where a Node is a data structure with 2 fields: one field for a node label and one field for a pointer to another Node. The list of nodes to which $A[i]$ points is a sorted linked list which represents the nodes that are adjacent to node i in the graph. If the application is such that the graph is not changing, then the data structure illustrated in Figure 1.2(d) can be used [53]. In this structure, two integer arrays take the place of the array of pointers and the linked lists of (c). The *Index* array indicates the starting point in the *Adj* array for each adjacency list. For example, the neighbours of vertex i can be found in $Adj[j]$, for $j = Index[i]$ to $j = Index[i+1]-1$. The array representation of the adjacency list provides slightly faster access to the adjacency lists than is obtained with the linked list representation. Both versions of the adjacency list are slow compared to the adjacency matrix for answering the question, "Is vertex i adjacent to vertex j?". The question must be answered by traversing the links starting at $Adj[i]$ and stopping when either the node j is found or a node with a label larger than j is found, or when the end of the list is encountered. On the other hand, the adjacency list allows relatively fast access for answering the question "What are neighbours of node i?". A critical advantage of the adjacency list is that it takes less space than the matrix when representing a large sparse graph.

The amount of space used to store the adjacency information can be cut in half as shown in Figure 1.3. This is possible, since we assume here that the graphs are undirected. Therefore, we only need to store edges (i, j), where $i < j$. The matrix representation can be stored in a two-dimensional array if the computer language allows rows of unequal length. Otherwise, the matrix can be stored in a one-dimensional array, and the look-up function which checks if vertex i is adjacent to vertex j can use a formula to calculate which element to access. The list representation is in the same style as before, but now the list for vertex i only shows vertices $j > i$ which are adjacent to i. Answering the question "Is vertex i adjacent to vertex j?" requires an extra step to compare i and j and possibly another step to swap the parameters so that we look for the larger vertex in the list of the smaller. These data structures provide more efficient storage of the adjacency information, but their drawback is that it becomes much more time consuming to answer the question, "What are the neighbours of vertex i?". Answering this question requires scanning not only the row or list that represents neighbours of vertex i, but also all of the rows or lists representing neighbours of vertices j, where $j < i$.

1.3 Genomics Terminology

In this section, I introduce just enough basic terminology and information from genomics so that a reader who has ignored the field will understand the motivation for this research. Since the motivation involves physical maps of YAC and STS data produced in the Human Genome Project, I attempt here to define these and related concepts from the bottom up. An excellent source for a more in depth discussion of these topics is provided by Lewin [59].

The genome of an organism is its store of hereditary data – the set of molecules directly responsible for holding and transmitting to new generations the information required to make the organism. In 1944 Avery showed that in bacteria these informational molecules are deoxyribonucleic acid (DNA) [98]. Others later showed that DNA is not only the genetic material of prokaryotes (bacteria) but also eukaryotes (plants and animals) and most viruses. The genetic material of some viruses is ribonucleic acid (RNA), a molecule closely related to DNA. In 1953, Watson and Crick showed that the structure of DNA is that of a nonbranching, double helix. Each strand of the double helix has a 'backbone' consisting of a chain of sugars (deoxyribose molecules connected by phosphate bonds). Each sugar is covalently attached to a nitrogenous base, either adenine (A), thymine (T), cytosine (C) or guanine (G). Each base from one strand of a double helix is noncovalently linked to a complementary base in the other strand: base A pairs with T, and C pairs with G. Thus, common units of length for DNA are a base pair (bp), a thousand base pairs (kb), a million (Mb), and a billion base pairs (Gb). Complementarity is the foundation of replication, where each strand serves as a template for regeneration of the sequence of bases in the partner strand. Duplication of the genetic material by replication allows cell division without loss of information.

Information in the genetic material is largely code for assembling proteins, the other major type of macromolecule in cells besides nucleic acids. Proteins function as enzymes (catalysts), transport molecules, structural components, protective wrappers, gene regulators, to name a few of their activities. Just as a nucleic acid is a non-branching chain of nucleotides, a protein is one or more polypeptides, where a polypeptide is a non-branching chain of subunits called amino acids. The sequence of amino acids in a protein determines its three-dimensional shape and its biochemical reactivity. The correspondence between nucleotide sequence in DNA and amino acid sequence in protein is called the genetic code. Triplets of nucleotides (called *codons*) specify particular amino acids. For example, the triplet AGC corresponds to amino acid serine. The sequence of codons in DNA specifies the sequence of amino acids in the protein. Some codons serve as start and stop signals for the process where the information in DNA is used to construct proteins. The first major step in this process is transcription, where a complex enzyme called RNA polymerase traverses the DNA from a start signal to a stop signal synthesizing a complementary strand of RNA. The RNA so formed (or sometimes modified by splicing out unnecessary regions) serves as a messenger (mRNA) which carries the information to cellular components called ribosomes. Ribosomes attach to the mRNA and translate the sequence of codons into a sequence of amino acids which are covalently joined to form a polypeptide. Thus, Avery, Watson, Crick and other biochemists showed that the conformation and function of proteins, which account for most observed genetic traits, are directly determined by the sequence of bases in DNA. This sequence comprises the heritable information of an organism.

The information in the genome has a number of different natural units. The largest unit is the genome itself, which is the set of DNA molecules in a cell, excluding DNA belonging to organelles or invading organisms. Each DNA molecule is a unit of the genome called a chromosome. A bacterium generally has just one chromosome. A human somatic (body) cell has 46 chromosomes, 23 from each of the person's parents. The 23 chromosomes from one parent include one of the two sex-determining chromosomes, X or Y, and 22 autosomes (nonsex chromosomes). Autosomes are named by number roughly in order of their length: chromosome 1 is the longest (about 279 Mb [92]) and chromosomes 21 and 22 are the shortest (45 Mb and 48 Mb, respectively [92]). A pair of autosomal chromosomes (for example, two versions of chromosome 7) are nearly identical in sequence (termed 'homologous'), and together they determine a particular set of traits. The pair of sex chromosomes determines the sex of the individual: XX = female and XY = male. Thus, in human somatic cells another informational unit is the chromosome pair.

A fundamental unit of information within a chromosome is the gene, the basic unit of inheritance discovered by Mendel. A gene typically codes for a polypeptide. The number of human genes has been estimated at 100,000 for many years [59], then in the neighbourhood of 25,000 to 35,000 [90, 96] based on rough draft sequence data, and still more recently at 65,000 to 75,000 [100]. The number of distinct human proteins is estimated to be about 90,000 [30]. The average size of a human (or mammalian) gene is about 20 kb, while the average size of a bacterial gene is about 1 kb. The mammalian gene is larger not because it codes for longer polypeptides, but rather because it often contains extra 'junk' DNA not used for coding. Thus, the gene has subunits called exons (used for coding) and introns (noncoding inserts between exons). Exons have subunits called codons, described above. The subunits of a codon are its three nucleotides. In addition to the above hierarchy of informational units (genome, chromosome, gene, exon/intron, codon, base), there are also units of information concerned with regulating when and where certain genes or groups of genes are expressed as proteins.

The structure of genetic information in all known genomes is essentially the same.

Genomes are found not only in the cells of free-living organisms like bacteria and humans, but also in organelles (mitochondria and chloroplasts) and infectious agents (plasmids and viruses). Mitochondria and chloroplasts are thought to have originated as free-living organisms which became symbionts inside other cells. Through the course of evolution the engulfed organisms lost part of their genome as their existence became interdependent with that of their host. Plasmids and viruses also depend on host cells for existence, though the reverse is not usually the case. Plasmids and viruses are basically naked DNA, although a virus is packaged in a protein coat for part of its life cycle. The genomes of plasmids and viruses contain only some of the genes required for their replication, hence their dependence on a host cell. A plasmid is a small, usually circular, self-replicating DNA molecule found in bacteria or yeast. It replicates when the host cell replicates and is usually present in a constant, low 'copy number', for example, one plasmid per cell. A plasmid is typically extrachromosomal, i.e., separate from the host genome, but it may at times integrate with it. A virus has a more complicated life cycle than that of a plasmid. Like a plasmid, a virus can spend time quietly integrated in the host genome, but when it separates, it begins explosive growth leading to lysis of the host cell. Also, unlike a plasmid, a virus can package its DNA within protein and survive outside a cell on its way to infect other cells. A virus which infects bacteria is called a bacteriophage (or phage).

Biochemists have developed genetic engineering techniques for isolating plasmids and phages and modifying them so that they become suitable carriers, i.e., *vectors*, of exogenous DNA. One other commonly used type of vector is the yeast artificial chromosome (YAC). A YAC is a linear strand of DNA in a yeast cell, where the strand is a self-replicating element by virtue of having three special DNA sequences (a telomere, a centromere and an origin of replication). These special sequences allow the YAC to propagate with the yeast cell as an artificial chromosome. Techniques for inserting foreign DNA into vectors rely heavily on bacterial *restriction enzymes*, which can cleave duplex DNA at specific sites. A restriction enzyme is used to cut the vector to prepare it for insertion, and also to extract a fragment of DNA from (for example) the human genome. An enzyme called a *ligase* is added to a mixture of the severed vectors and the foreign fragments, and the ligase covalently links some of the fragments into the vectors. The vectors are then placed in contact with cells that they can infect, typically bacterial cells or yeast cells.

Vectors from phages or plasmids or combinations of phage and plasmid DNA (cosmids) have been engineered to allow selection of the host-vector system and detailed analysis of the clones. The amount of foreign DNA that these vectors can hold is limited to about 40 to 100 kb. This means that a library based on these vectors must be extremely large to cover a eukaryotic genome. YAC vectors were engineered which could hold 1 Mb of foreign DNA, and the YAC system seemed like a promising way to give long range continuity to a library of foreign DNA [11]. However, it turned out that YACs were not suitably stable, and they could end up holding more than one foreign insert — a situation termed *chimerism*. Today the vector of choice is called a BAC, or *bacterial artificial chromosome*, which is based on a plasmid in E. coli called an F-factor replicon. A BAC can hold upwards of 300 kb of foreign DNA with stability and little chimerism [83]. Most of the DNA sequencing of the human genome is based on BACs. The term *clone* refers to both the population of identical cells used to grow a particular DNA fragment and to the set of identical fragments so produced. The clone serves as a source for macroscopic amounts of the foreign DNA insert, since it can be expanded as desired.

Thousands of such clones carrying fragments from throughout the genome or from one region of interest comprise a library for the genome or region. Libraries are clearly needed if one wants to study parts of the genome, e.g., genes, in isolation. They are also needed in large sequencing projects. The alternative, sequencing the genome as a whole, is not possible with current technology, since sequencing machines typically 'read' lengths of DNA in the range 500 bp to 750 bp [96]. Therefore, for this technical reason and for reasons of reproducibility, a sequencing project uses a library of very short fragments (1 to 2 kb). The short-fragment clones may be obtained by shearing clones from another library that has longer inserts. For example, the Human Genome Project uses a library of BAC clones for this purpose, where the average insert size is 160 kb [90]. Since the human genome is so large (3.2 Gb) and complex, it helps to assemble the sequence in stages. For a number of years scientists in the Human Genome Project used a library of YAC clones with very large inserts (1 Mb), but these clones turned out to be less stable than required.

When it is necessary to extract the insert from a library clone, cells from the clone are homogenized to release the DNA and other cellular material. The DNA is removed from the homogenate, and then the artificial chromosome is separated from the host DNA on the basis of size. The insert of foreign DNA is cut from the plasmid vector using the appropriate restriction enzyme and separated by size or hybridization. The extracted fragment (which might contain one or more genes of interest) can then be studied in terms of sequence or expression.

1.4 Physical Mapping

In this section, I give a brief introduction to physical mapping, since that is the application for which I develop algorithms in this dissertation. A physical map is a directory for a library of clones.

Physical mapping is the process which assigns the cloned fragments in a library to their relative or absolute sites of origin in the genome. Figure 1.4 illustrates portions of two kinds of physical maps. In this diagram, clones are labeled C1 through C8. In the upper panel, the little tick marks on the clones and chromosome denote restriction enzyme sites. In the lower panel, the vertical bars represent probes which are used to identify sites on the clones. These maps are described in more detail below, but the general idea is simple. Relative assignment is a matter of lining up the clones to show

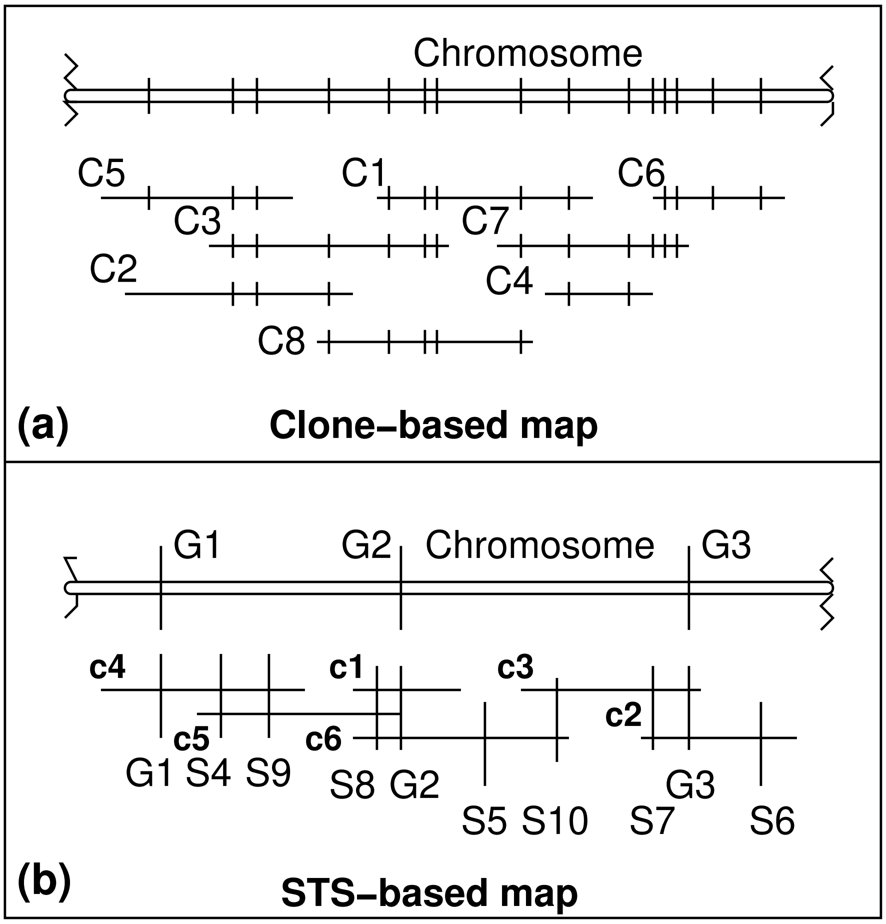


Figure 1.4: (a) Schematic of clone-based mapping. (b) Schematic of STS-based mapping.

how they overlap and how they can be laid out to form continuous genomic regions and ultimately complete chromosomes. Absolute assignment is achieved when the sequence of the genome is known and the sequence of the clone can be identified at a particular location in the genome sequence. Absolute assignment is clearly ideal, but it usually depends on an initial relative assignment. Relative assignment of clone position is what is normally meant by the term physical mapping, and it will be the meaning I employ below.

Physical mapping of the clones in a library is an important preliminary step to sequencing a large genome for a number of reasons. These reasons are described by scientists working on the Human Genome Project, whose approach is succinctly described as 'map first — sequence later' [71, 89]. The approach taken in the Human Genome Project is roughly the following: (*i*) prepare a large-insert library ; (*ii*) physically map the large-insert library ; (*iii*) select clones for sequencing ; (*iv*) prepare a short-insert library for each selected clone by shearing ; (*v*) sequence the short inserts ; (*vi*) assemble the sequence of a large insert from the sequences of its corresponding short inserts ; and then (*vii*) assemble the sequence of the genome from the sequence of the large inserts using the physical map as a guide.

One reason for the physical mapping step (*ii*) is that it facilitates selection of clones in step (*iii*), where it is desirable to choose clones which cover the genome without much redundancy. Secondly, the physical map facilitated international collaboration since clones used in various laboratories could be related by easily identifiable genomic markers in the physical map. The third and most important reason for creating a physical map concerns the assembly steps (*v* - *vii*). Assembly of genome-wide, unmapped short sequences would be very difficult if not impossible, because the human genome contains many regions where sequence is repeated [71]. The intermediate step of assembling short sequences into the sequence of a mapped clone greatly reduces the challenge presented by the repeat regions.

Venter *et al.* argues that these challenges can be met without physical mapping [96]. Celera Genomics sequenced the fly genome and produced a draft sequence of the human genome without taking the time and trouble to do extensive physical mapping. Rather than physical mapping, the Celera approach relies on extensive computational analysis and a different experimental protocol. However, scientists in the Human Genome Project point out that some physical mapping data were used in assembling the sequence of the fly genome [89], and that the especially hard work of finishing the sequence of the human genome is where the physical map will be essential [71, 90]. Genome projects in the future may rely more on comparison with sequenced genomes and less on new physical maps [71].

There are a number of methods which can be used for physical mapping, and typically a map is created by integration of results from as many methods as possible. One method is restriction fragment fingerprinting. In this method a clone is subdivided by cutting it with a restriction enzyme. The lengths of the resulting fragments comprise a fingerprint of the clone. The fingerprint can be made more informative by using more than one restriction enzyme, or a combination of enzymes at one time. The fingerprints of clones are compared to determine which clones overlap, i.e., share part of the same genomic region. The overlapping pattern of fingerprinted clones is the physical map. Figure 1.4(a) illustrates a number of clones lined up in this fashion. The clones are shown with little ticks indicating the deduced positions of the restriction enzyme sites. After determining a path of overlapping clones and the position of at least one clone on the chromosome, then the positions of the restriction enzyme sites can determined for the chromosome, as shown. The collection of clones and chromosomes aligned in this way is called a clone-based map since the focus is on the clones and their fingerprints.

Another variation of fingerprinting uses *partial digestion*. In this version the fingerprint is made more informative by labeling one end of the clone radioactively and stopping the digestion reaction when each restriction site has about a 50% chance of being cut. For any given site A, at least some clones are cut at A but not cut between A and the labeled end. Thus, when the sizes of the labeled fragments are determined, there is a length for each distance from the labeled end to a restriction site. The pattern of positions of restriction sites on clones is used to determine which clones overlap and to align them accordingly to form a physical restriction map.

Overlap of clones can also be assayed by direct hybridization, however the number of reactions that would have to be tested is generally too high. A compromise to this method is to derive probes from a subset of the clones by using the polymerase chain reaction (PCR) between known repeat sequences called Alu sites. The Alu-PCR probe is then hybridized with other clones to detect overlap between the source clone and the target clone. If the probe from source clone A hybridizes with target clones B, C and D, one infers that A overlaps with B, C and D, but one makes no inference regarding whether B, C and D mutually overlap. This is in contrast to the case of much smaller probes discussed below, where one assumes mutual overlap of all clones that the probe hits.

Another common method of physical mapping employs probes called sequenced tagged sites (STS). An STS is a short region (about 300 bp) of the genome which can be detected using the PCR. Careful selection procedures are used to screen out STSs which identify regions which are repeated in the genome. Attempts are made to choose STSs which are well distributed throughout the entire genome. The mapping assay determines which clones contain which STSs. If several clones contain the same STS, then it is assumed that they overlap. The resulting overlapping pattern of clones pinned together by STSs forms the physical map. An STS-based map is illustrated in Figure 1.4(b). The vertical bars symbolize STSs.

The physical map can be tied to a genetic map by polymorphic STSs. Some of the STSs in Figure 1.4(b) are labeled with a G to denote STSs that have been genetically mapped. Such STSs anchor the map to the chromosome. Genetic maps determine the relative position and separation of heritable markers called polymorphisms. In general, a polymorphism is any site along the DNA where sequence varies within the population. To be useful in genetic mapping, the polymorphism must be observable in some way, sometimes as a simple phenotypic trait distinguishable to the eye, or sometimes as a trait observable through biochemical tests or sequencing. These markers frequently move from one chromosome to another during meiosis, by a mechanism called recombination, or crossing-over. The closer two markers are together, the more likely they are to stay together, and conversely, the farther they are apart the more likely a recombination event will separate them. The principle is captured in a simple formula which expresses the distance between polymorphisms in centiMorgans (cM):

$$Map\ distance = \frac{Number\ of\ recombinants \times 100}{Total\ number\ of\ progeny}$$

[59]. The formula is applicable only over short distances, but this is not a problem since the distances are additive. In other words, if A and C are markers that are too far apart to be measured by this method, the distance can be measured in shorter steps (from A to B and from B to C, if B lies between A and C) and summed up. Thus, the separation in cM of polymorphic STSs can be determined, as well as their position relative to genes and other markers. They thus serve as a great aid in anchoring the physical map to the chromosomes, as is done in the human maps of Hudson *et al.* [47] and Schuler *et al.* [84].

STSs can also be ordered and positioned along chromosomes by the method of radiation hybrid mapping [21, 70]. This method uses a panel of hybrid cells formed from irradiated human cells and nonirradiated hamster cells. The hybrid cells contain randomly broken fragments of human chromosomes. STSs are screened against the panel of hybrid cells, and the results are analyzed in a way similar to linkage analysis in genetic mapping. The probability that two STSs which are normally on the same chromosome are separated by a random break is proportional to the distance between them. The result of the analysis is an ordering of the STSs as well as a coordinate in centiRays (cR) for each STS. The correspondence between centiRays and physical distance in kilobases is approximately constant throughout the genome and depends on the intensity of radiation used to form the panel. The radiation hybrid map therefore links the physical map to real coordinates on the chromosomes.

The physical map can also be aligned with the cytological map. Cytological maps characterize the shape of chromosomes and the positions of microscopically visible light and dark bands. The banding is a constant feature, and particular bands can be associated with particular genes by a process called in situ hybridization. The process can involve isolating mRNA which is expressing a gene, making a DNA copy (cDNA) from the mRNA, radioactively labeling the cDNA and hybridizing it to the chromosome.

The radioactivity then marks the band corresponding to the gene in question. A similar process called fluorescence in situ hybridization (FISH) involves attaching fluorescent markers to a clone and allowing the clone to hybridize with the DNA at a time in the cell cycle (metaphase) when the chromosomes are visible under the light microscope [13, 87]. The labeled clone binds to its site of origin through complementarity. The position of the clone on the chromosome thereby links the physical map to approximate real coordinates.

Thus, there are many approaches to physical mapping, each with its own detailed analysis (only sketched above). Each method naturally has limitations and susceptibility to certain forms of noise in its data. Noise may cause *false positives* where overlaps are recorded which do not exist, or *false negatives* where true overlaps go unnoticed. The term noise is a catch-all for the variables which are not totally under control in an experiment and which may obscure the signal one is trying to measure. Some noise is caused by experimental error, such as cross-contamination of samples. Other noise is from biological variation, such as unstable clones, where the foreign DNA changes over time. The quality of data can vary with the number of cloned fragments for each region in the genome (i.e., the *coverage*). Some regions may even be unclonable [8]. Regions from distant parts of the genome sometimes share the same sequence. These repeat regions can make it difficult to interpret the overlap data [14]. The many challenges of creating a physical map of a large and complex genome induce mappers to combine results of several methods where possible.

In this dissertation I refer to probe data (meaning STS data), overlap data (meaning either fingerprint or Alu-PCR data), genetic mapping data and radiation hybrid mapping data. I use each type of data in a processed form (as provided by the source genome center). Probe data are in the form: STS x hits clone a. Overlap data are in the form: clone a overlaps with clone b. Genetic mapping data are in the form: STS x is at position g cM on Chromosome c. Radiation hybrid data are in the form STS x is at position r cR on Chromosome c.

1.5 The Human Genome Project

In this section I give a brief overview of the Human Genome Project, since it is the source of much of the data analyzed in this dissertation.

The idea of making a concerted, publicly funded effort to sequence the entire human genome was discussed at scientific meetings from 1985 to 1987 (see reviews by Roberts [77] and the International Human Genome Sequencing Consortium [90]). Advocates of the idea maintained that the sequence would revolutionize the approach to medicine and biology by providing a 'dictionary' and 'periodic table' of life. Critics of the idea raised a number of concerns, including: (*i*) that sequencing technology was not yet equal to the task, (*ii*) that the project would deflect money away from more useful research, (*iii*) that the project was 'big science', meaning that it was more like a business undertaking than real science, (*iv*) that much of the sequence would be useless, so why not locate and sequence particular genes as needed, (*v*) that it would be better to start with much smaller genomes of model organisms, and (*vi*) that the task would be too boring for creative minds. Advocates and critics were gathered together on a panel representing the U.S. National Research Council (NRC), and they came to an agreement in 1988 approving the Human Genome Project. The panel recommended a phased approach to the project including (*i*) the production of physical and genetic maps of the human chromosomes, (*ii*) analysis of genomes of model organisms (bacteria, yeast, nematode, fruit fly and mouse), (*iii*) development of mapping and sequencing technology, (*iv*) a pilot project sequencing a million-base genome, (*v*) sequencing the human genome, and (*vi*) research into ethics and laws concerning the project. The time frame for the project was 15 years, with a budget of $200 million per year (i.e., a total of 3 billion dollars). The U.S. National Institute of Health (NIH) became the main funding and controlling agency of the project, although international funding and collaboration was encouraged. Thus, the idea of sequencing the human genome had become a functioning, multi-billion dollar project by late 1990 [90].

Quite surprisingly, the Human Genome Project has kept on schedule. Physical mapping of yeast and worm genomes had started earlier [20, 72], and these maps were nearly complete by 1990 [77]. Physical maps and genetic maps of the human genome appeared in the early 1990's, culminating in an STS-based map in 1995 [47] and a gene map in 1996 [84]. Similar progress was made in mapping the mouse genome [26, 69]. These mapping projects produced the immediate benefit of reducing the time to find typical human disease genes. They also laid the groundwork for the sequencing projects.

Prior to the initiation of the Human Genome Project in 1990, few genomes had been completely sequenced — only those of a few viruses and organelles (mitochondria and chloroplasts) [90]. By the time the draft sequence of the human genome was published in 2001, the sequences of genomes of at least 42 species had been published, including 31 bacteria, seven archaea, one fungus, two animals and one plant [90]. I list below some of the major milestones in terms of genomes sequenced. The sequencing projects receiving public funding under the Human Genome Project include those of yeast, nematode and human. Lengths of the genomes are designated in units of kb (kilobase = one thousand nucleotide pairs), Mb (megabase = one million nucleotide pairs), and Gb (gigabase = one billion nucleotide pairs). Where two lengths are given, the first is the amount sequenced, and the second is an estimate of the total length of the genome. Typically, part of eukaryotic genomes, called the heterochromatin, is left unsequenced for technical reasons related to the region being highly repetitive and difficult to clone [8].

- 1977: ΦX174 (bacteriophage, 5.4 kb): first complete (genomic) sequence of any organism or virus [80];

- 1995: *Haemophilus influenzae* (a bacterium, 1.8 Mb): first complete sequence of a free-living organism [27];

- 1996: *Saccharomyces cerevisiae* (yeast, 12.1 Mb/ 12.9 Mb): first complete sequence of a eukaryote [33];

- 1997: *Escherichia coli* (bacterium, 4.6 Mb): complete sequence of the most studied laboratory organism [6];

- 1998: *Caenorhabditis elegans* (nematode, 97 Mb/ 98 Mb): first complete sequence of a multicellular organism [88];

- 2000: *Drosophila melanogaster* (fruit fly, 120 Mb/ 180 Mb): first complete sequence of an insect [1];

- 2000: *Arabidopsis thaliana* (ragweed pollen, 115 Mb/125 Mb): first complete sequence of a plant [86];

- 2001: *Homo sapiens* (human, 2.7 Gb/3.2 Gb): first draft sequences of a vertebrate [90, 96];

- 2003: *Homo sapiens* expected finished sequence [77].

Parallel to the development of maps and sequences, there have been studies and meetings focusing on the legal, ethical and social issues arising from this new source of information [67, 91]. Thus, the Human Genome Project will probably achieve all of its goals within the allotted 15-year time frame (1990-2004).

The scientific success of the HGP can be attributed to international collaboration and to improvements in sequencing and mapping technology. The temporal success of the project can be attributed in part to unexpected competition from a privately funded company called Celera Genomics. Celera began sequencing the human genome using an approach which was potentially faster than that used by the International Human Genome Sequencing Consortium (representing the HGP). As a consequence, the Consortium modified their plans in order to race with Celera (fearing problems with future data access if Celera published first) [90]. The race turned out to be a tie, as both groups published a draft sequence of comparable quality at the same time [90, 96]. A brief review of the politics, personalities and scientists involved in this historical accomplishment is provided by Roberts [77].

1.6 Motivation

This research arose from discussions between one of the mentors of this thesis, Dr. Anthony Bonner at the University of Toronto, and scientists at Whitehead Institute for Biomedical Research/MIT Center for Genome Research (WI/MIT) (notably Dr. Nathan Goodman, then Director of Informatics at WI/MIT). At that time (1994), participants of the Human Genome Project (including WI/MIT) were working on the task of generating a YAC-based physical map of the human genome.

A physical map typically starts with a set (the whole library) of unassigned and unordered clones. Each clone can be considered an island, since initially there is no indication to connect one clone to another. As overlaps between clones are determined, the islands merge into multiclone islands. An island of clones is called a *contig* when the overlapping clones would form a contiguous sequence if placed at their sites of origin in the genome. Often islands are not contigs, because noise, experimental error and repeat regions introduce apparent overlaps between noncontiguous clones.

One object of physical mapping is to generate overlap data until the resulting contigs merge into one contig per chromosome. Confirmation that islands are contigs is clearly an essential component of this task. Islands which are not contigs must be separated into contigs by identifying and removing the false overlaps. Contig identification is difficult, and early results from genome centers often showed islands labeled as contigs which later proved to be several cross-linked contigs. A major goal of this thesis is the development of a graph algorithm which facilitates verification and identification of contigs.

The second theme of this thesis concerns integration of overlap data with probe data. Genome centers collected a huge amount of each of these types of mapping data using YACs and STSs. However, integration of the two data types to form a physical map is generally an *ad hoc* process. No integration method with a sound theoretical foundation exists for this purpose. A common approach is to use one form of data as a secondary source of information to disambiguate problems encountered in the analysis of the primary form of data. For example, in the STS-based physical map of Hudson *et al.*[47], overlap data are used to corroborate ambiguous STS data and to help make connections between probe-based contigs. In the YAC-based map of Chumakov *et al.* [13], paths of overlapping clones are constructed between genetically mapped probes, and the credibility of the path depends (in an unclear way) on the length of the path and the number of clones. I present an algorithm which integrates probe and overlap data in a uniform manner, and I describe the theory supporting this approach. The method is based on forming clusters of mutually overlapping clones, and then treating these new entities as though they were probes.

1.7 Contributions

Ideally, a thesis in bioinformatics should produce results which advance both biological science and computer science. The biological problems addressed in this thesis are in the area of physical mapping, i.e., mapping experimentally-produced fragments of a genome to their original locations in the genome. The thesis makes two primary contributions in this area: (i) a method to improve contig identification and verification, and (ii) a method which improves contig formation by uniformly integrating probe and overlap data. The problems addressed within computer science arise in the above application and are in the area of heuristic graph algorithms. The thesis makes three primary contributions in this area: (i) a modification of a well-known algorithm for listing maximal cliques, which markedly improves its efficiency on large, sparse graphs; (ii) a new clustering algorithm which is appropriate for 'noisy' interval graphs; and (iii) a method which reveals piecewise linearity in a noisy interval graph.

The methods and algorithms are tested on human genome mapping data obtained from Centre d'Etudes du Polymorphisme Humain and Généthon (CEPH/Généthon) and Whitehead Institute for Biomedical Research/MIT Center for Genome Research (WI/MIT). These tests show that the structure-graph method readily reveals the underlying interval graph nature of the probe mapping data and thereby helps to identify and extract contigs. The tests also show that the virtual-probe technique which integrates probe data and overlap data can produce longer double-linkage contigs than probe-based analysis alone, and provides a new way of assessing overlap data.

I develop a simulator of genome data and use it to test the algorithms in a controlled fashion. I show that for large graphs (25,000 nodes) representing genome overlap data, my modification of the Bron-Kerbosch algorithm is at least 100 times faster than than several published algorithms for enumerating the maximal cliques. A new clustering algorithm is tested on simulated noisy overlap data where it is demonstrated to effectively counter the proliferation of cliques caused by false negatives.

1.8 Overview

The rest of this dissertation is structured as follows.

In Chapter 2, I review literature related to physical mapping and the theory of interval graphs.

In Chapter 3, I quantitatively describe the characteristics of the STS and YAC data which form the basis of analyzes in subsequent chapters. I show that there is a great deal of cross linkage in this type of data, causing observation of overlaps which do not reflect contiguity in the genome. Cross linkage obfuscates contig identification.

In Chapter 4, I describe an algorithm which is designed to reveal hidden interval-graph structure in probe mapping data. Interval-graph structure is the kind of structure expected of contigs, and its primary characteristic is linearity. The linearity is made obvious by the structure-graph algorithm when used in conjunction with a graph visualization tool. I show visualizations of genomic data and demonstrate the extraction of contigs from nonlinear islands.

In Chapter 5, I present a modification of the Bron-Kerbosch algorithm for listing the maximal cliques of a graph. I compare the modified algorithm with the unmodified algorithm and several other published algorithms. This algorithm is used in Chapter 6.

In Chapter 6, I present an algorithm which integrates overlap and probe data. Advantages of the integration are demonstrated on genomic mapping data.

In Chapter 7, I use simulated data to show that false negatives are a potential problem for the virtual probe technique, since they cause a single set of mutually overlapping clones to be represented as many cliques. I introduce a clustering algorithm to counter this effect.

In Chapter 8, I summarize the contributions that the dissertation makes to genomics and computer science. I also suggest future work in this area.

There are three appendices. Appendix A describes the simulator for genome data. Appendix B gives the details of an algorithm that removes small cycles from a structure graph. Appendix C provides a listing of the C-code for the modified Bron-Kerbosch algorithm.

Chapter 2

Brief Survey of Physical Mapping

A physical map of a genome is essential to a molecular geneticist just as a catalogue or index is necessary for the users of any library. The map is an aid in locating disease genes, sequencing and studying the function of sections of the DNA. A major goal of the Human Genome Project is to construct detailed physical maps of the human genome and the genomes of other organisms. A physical map gives the position on a genome of landmarks, genes and fragments of DNA maintained as clones in microorganisms. Most genomic experiments do not determine positions of markers on the DNA directly, but rather by their location relative to other markers. Inferring relative or actual positions from the experimental data is called *map assembly*. Assembling complete and detailed maps of large genomes requires the integration of many forms of experimental data. Integrated maps typically contain information from cytological, genetic and physical maps. Each of these types of a map was introduced in Chapter 1.

2.1 Theory

Imagine you have many identical copies of a long strip of paper that is colored at one random point red, at another point orange, ...and so on, so that it has n different uniquely colored marks on it, placed at random along its length. This is represented in Fig-

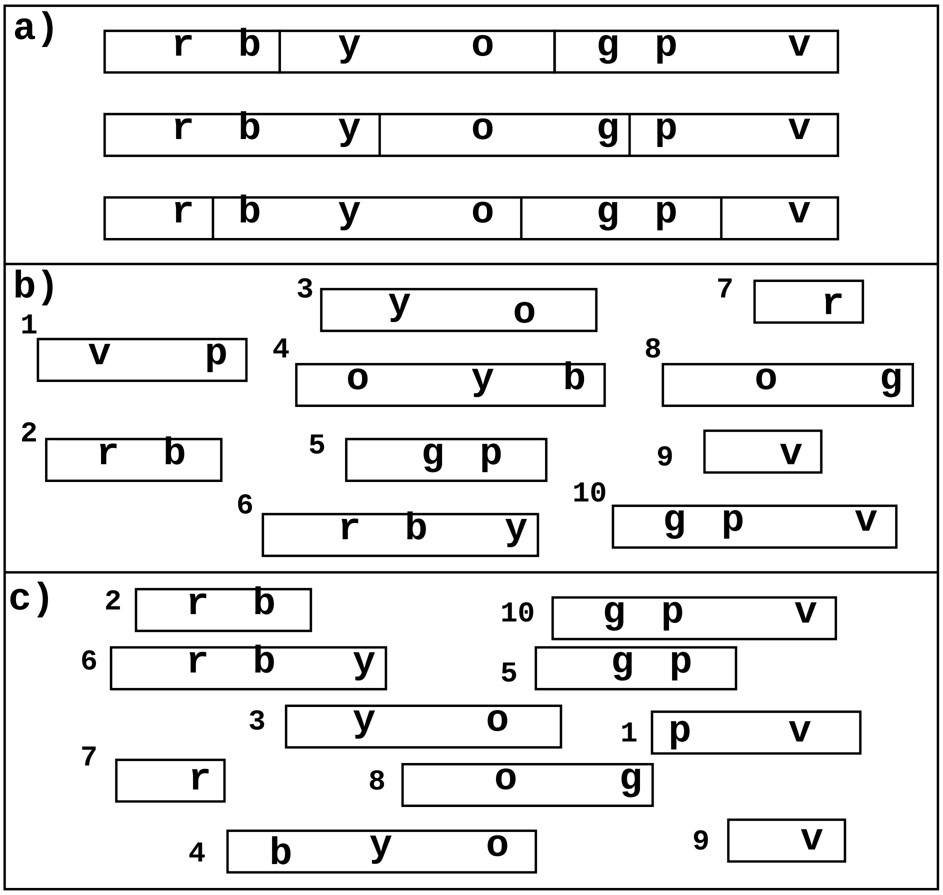


Figure 2.1: Identical strips of paper with markers

a) Each strip of paper corresponds to a copy of the chromosome. The letters represent uniquely colored points on the paper, and they correspond to random STS locations on a chromosome. b) The strips are cut at random places and shuffled. c) The strips are realigned according to common markers.

ure 2.1(a). Next, take each strip individually and cut it at random into many short strips ('fragments') Figure 2.1(b). Finally, shuffle m of the fragments as in Figure 2.1(b). The problem then is to determine the pattern of colors on the original strip of paper. The end product should be something like that of Figure 2.1(c). This is analogous to the physical mapping problem, where the initial strips of paper represent a single chromosome. The typical case, however, would be to start with many copies of all of the chromosomes. Thus, we should really have several different strips of paper in part (a). Part (b) of Figure 2.1 corresponds to cutting up the chromosomes into small fragments for cloning. Part (c) corresponds to the result of physical mapping, which provides order and orientation for the clones in the library.

Physical Approach

One obvious way to proceed in solving this problem is to look for fragments that have a red mark (denoted **r** in Figure 2.1) and place them together, overlapping at the red mark. Hopefully some of these fragments will include another color, say yellow. We would gather all of the yellow-marked fragments and place them on the red-yellow fragments in a way that is consistent with the growing pattern. Continuing in this fashion we could extend the set of overlapping pieces until finally the pattern for the entire length of paper is determined. The assembly requires a scan for each of n colors through $m - i$ fragments, where i fragments have been ordered so far. Each of the searches and subsequent alignments takes worst-case time proportional to n, since there could be n colors on a fragment. Thus, the total assembly time is $O(mn^2)$.

Matrix Approach

Suppose the fragments are numbered $1...m$, and we are given an $m \times n$ binary matrix where a 1 in position (i, j) means that fragment i contains color j, and a 0 means it does not. The matrix for the example data in Figure 2.1(b) is shown in Table 2.1.

This is a little less information than we have in the previous form of this problem,

	r	o	y	g	b	v	p
1	0	0	0	0	0	1	1
2	1	0	0	0	1	0	0
3	0	1	1	0	0	0	0
4	0	1	1	0	1	0	0
5	0	0	0	1	0	0	1
6	1	0	1	0	1	0	0
7	1	0	0	0	0	0	0
8	0	1	0	1	0	0	0
9	0	0	0	0	0	1	0
10	0	0	0	1	0	1	1

	r	b	y	o	g	p	v
1	0	0	0	0	0	1	1
2	1	1	0	0	0	0	0
3	0	0	1	1	0	0	0
4	0	1	1	1	0	0	0
5	0	0	0	0	1	1	0
6	1	1	1	0	0	0	0
7	1	0	0	0	0	0	0
8	0	0	0	1	1	0	0
9	0	0	0	0	0	0	1
10	0	0	0	0	1	1	1

Table 2.1: Initial matrix (left) and consecutive-ones matrix (right)

	r	b	y	o	g	p	v
7	1	0	0	0	0	0	0
2	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0
4	0	1	1	1	0	0	0
3	0	0	1	1	0	0	0
8	0	0	0	1	1	0	0
5	0	0	0	0	1	1	0
10	0	0	0	0	1	1	1
1	0	0	0	0	0	1	1
9	0	0	0	0	0	0	1

Table 2.2: Final matrix with consecutive ones along the diagonal.

because now we might know that a fragment has red, yellow and blue, but the matrix does not indicate the order of these colors on the fragment. Nevertheless, we can find a solution in a similar way. Consider any fragment, say fragment i. The colors that it contains must occur consecutively in the overall color pattern. Therefore we permute the columns of the matrix to bring all the 1's of row i together. We could carry on much as the we did in the previous case by considering another row which has some 1's in common with row i. The result after all the required column permutations is a matrix in which the 1's in each row are consecutive. The matrix on the left in Table 2.1 can be converted to the one on the right in this figure by the following moves: $\mathbf{b} \leftrightarrow \mathbf{o}$, $\mathbf{o} \leftrightarrow \mathbf{g}$, $\mathbf{p} \leftrightarrow \mathbf{v}$, where the $i \leftrightarrow j$ means swap columns i and j. A less efficient series of swaps might also be obtained as one works row by row without much forethought. A matrix which can be successfully permuted in this way is said to have the *consecutive ones property for rows.* Once the columns/colors are permuted to satisfy each fragment, it is a simple matter to order the fragments (rows) appropriately so that the ones of the matrix fall along and adjacent to the main diagonal, as shown in Table 2.2. The order of the rows (using the original row numbers) gives a possible reconstruction of the order of the fragments and colors in the original long strip of paper. Note the correspondence between the order of rows in Table 2.2 and the order of fragments in Figure 2.1(c).

In 1965, Fulkerson and Gross [29] published an $O(m^2n)$ algorithm to test an $m \times n$ matrix for the consecutive ones property for rows. (Actually, their algorithm was $O(mn^2)$ for testing a matrix for the consecutive ones property for columns, which is equivalent.) Surprisingly, however, the problem can be solved in linear time [7]. Booth and Lueker [7] developed a data structure called a PQ-tree which very nicely reflects the information and constraints characteristic of this problem and a number of other related problems. Each such problem involves a set U of n elements, colors in our example. We wish to find permissible permutations of the elements of U, given that certain subsets $S \subseteq U$ must occur consecutively. In our example, each fragment is a subset S of colors which must occur together in the final permutation. The leaves of the PQ-tree are the elements of U, and the internal nodes are of two types which imply permissible permutations of the children. Children of a P node can be ordered in any way at all, as long as they are together. Children of a Q node have a specified order, but the order can be reversed. Booth and Lueker showed that by using the PQ-tree they could solve the consecutive ones problem in time $O(n + m + f)$, where f is the number of nonzeroes in the matrix. Moreover, the final PQ-tree is a representation of not just one solution, but all permissible solutions.

Graph Approach

So far we have discussed a physical representation of this problem and a matrix representation. Now we consider a graph representation. Suppose we denote each fragment of paper by a node in a graph, and we draw an edge between nodes/fragments which have a color in common. For example, all fragments that have a red mark are mutually connected. The graph which results is called an *interval graph*. More formally:

> A graph $G = (V, E)$, with vertex set V and edge set E, is said to be an *interval graph* if each vertex $v \in V$ can be paired with an interval I_v on the real line such that there is an edge $(u, v) \in E$ *iff* interval I_u overlaps interval I_v.

Just as we lost a little information going from the physical representation to the matrix representation, we lose a little more information going to the graph representation. For example, suppose three fragments have a yellow mark, and they are a subset of five fragments that have a red mark. In the matrix representation we know that the three yellow fragments must be together within the red group. In the interval graph representation we just know that the five nodes representing the red group must be together, since the yellow subset is not distinguished in any way. All nodes in the red group are mutually connected, which is the definition of a *clique* in graph theory. They form a *maximal clique*, assuming they are not in turn a subset of another color group.

Following this example, it may not be surprising that interval graphs can be characterized in terms of their maximal cliques, as was achieved by Fulkerson and Gross [29]:

> A graph $G = (V, E)$ is an interval graph if and only if there is an ordering of its maximal cliques such that for any vertex, v, the maximal cliques containing v occur consecutively. In other words, the matrix representing vertices as rows and maximal cliques as columns has the consecutive ones property for rows.

An efficient algorithm for recognizing that a graph is interval and simultaneously producing the latter so called *consecutive arrangement* of maximal cliques, was designed by Booth and Lueker [7] (in the same paper where they address the consecutive ones problem). The time complexity of their algorithm is $O(n + m)$, where n is the number of vertices and m is the number of edges (as defined in Chapter 1). In the first phase of the Booth and Lueker algorithm, the maximal cliques are identified using *lexicographic breadth first search* (LBFS). (The LBFS algorithm was designed by Rose, Tarjan and Lueker [79] for finding perfect elimination orderings in chordal graphs. A *chordal graph* is a graph with no induced cycles of size greater than three, i.e., the only cycle without a chord is a triangle. A *chord* is an edge between two nonconsecutive vertices in the cycle.) In the second phase, a consecutive arrangement of the maximal cliques is obtained using the PQ-tree.

The Booth and Lueker PQ-tree is a relatively simple construction, but manipulating and adjusting the tree during updates is a detailed process involving several template and replacement patterns. Korte and Möhring [54] simplify the update process by complicating the structure of the PQ-tree. The complication involves adding a set of vertices (from the interval graph) to each P-node and several sets to each Q-node (one for each child of the Q-node). In the case of the P node, the set consists of vertices common to all the maximal cliques in the subtree rooted at P. For a Q node with children, $Q_1, \ldots, Q_m$, the set associated with child Q_i consists of vertices shared by all maximal cliques in the subtree rooted at Q_i and an adjacent subtree (rooted at Q_j). The resulting tree is referred to as a *modified* PQ-tree, (MPQ-tree for short).

Below we summarize some of the properties of MPQ-trees and the algorithm for recognition of interval graphs, as presented by Korte and Möhring. First we introduce some notation. Let T be a PQ-tree of graph G = (V,E) and let T' be the associated MPQ-tree. Let $Adj(v)$ denote the set of vertices adjacent to vertex v. As in the Booth and Lueker algorithm, the first step of the Korte and Möhring algorithm is a lexicographic breadth first search, which produces an ordering $\lambda = v_1, \ldots, v_n$ of the vertices. Given this ordering, let $u < v$ denote that u is to the left of v in λ, and $V' < v$ denote that every vertex in the set $V' \subseteq V$ is to the left of v.

1. T' can be obtained from T in $O(n + m)$ time and represents G in $O(n)$ space.

2. Each maximal clique of G corresponds to a path in T' from the root to a leaf, assuming each vertex is placed as near as possible to the root.

3. An LBFS-ordering $\lambda = v_1, \ldots, v_n$ of G can be obtained in $O(n+m)$ time (as shown by Rose *et al.* [79]).

4. Given the latter ordering λ, let G_i denote the subgraph of G induced by $V_i = \{v_1. \ldots, v_i\}$. Then G is an interval graph if and only if, for each $i = 1, \ldots, n-1$, G_i is an interval graph and $Adj(v_{i+1}) \cap V_i$ is contained in an *outer clique* of G_i, i.e., a clique which is first or last in some consecutive arrangement of the maximal cliques. In this case, v_{i+1} is a *simplicial vertex* of G_{i+1}, i.e., a vertex belonging to only one maximal clique.

Property 4 is the basis for the Korte-Möhring algorithm for constructing the MPQ-tree for a graph G in an iterative fashion. The graph is traversed by LBFS. Assume at step i we have successfully constructed MPQ-tree T_i corresponding to the subgraph G_i (which therefore is interval). On encountering the next vertex, v_{i+1}, we must check that the neighbors of v_{i+1} all belong to some outer clique of G_i, otherwise G_{i+1} is not an interval graph. Korte and Möhring show that this check can be done in time proportional to $Adj(v_{i+1})$. If G_{i+1} is an interval graph, then T_i must be updated to include v_{i+1}. Like Booth and Lueker, they use templates during the update process, but apparently in a simpler way. They show that the update requires $O(|Adj(v_{i+1})|)$ amortized time. Thus, testing whether the graph G is interval and constructing its MPQ tree can be done in $O(n+m)$ time.

In summary, Korte and Möhring present an algorithm which is very similar to the Booth and Lueker algorithm but with an interesting modification to the data structure which facilitates the implementation and time complexity analysis.

Since the Korte-Möhring paper there have been a number of attempts to further simplify the recognition of interval graphs by abandoning altogether the difficulties of PQ-trees. The simplest algorithm to date is that of Corneil, Olariu and Stewart [17]. Their starting point is the following characterization of interval graphs due to Lekkerkerker and Boland [57]:

> A graph is an interval graph if and only if it is chordal and asteroidal triple-free.

An *asteroidal triple* is an independent set of three vertices such that there is a path between any two of the vertices which avoids the neighborhood of the third. Intuitively, the presence of an asteroidal triple gives a graph two dimensions, whereas interval graphs are basically one-dimensional. That is, all nodes on an interval graph (or other asteroidal triple-free graph) are within one edge of any path that stretches from one extreme of the graph to the other. (Corneil *et al.* define two nodes with this property as a *dominating pair* [18].) In addition to the above theorem, the authors use a third characterization of interval graphs due to Olariu (and others independently):

> A graph is interval if and only if there is an order $<$ on the vertices such that for any three vertices u, v, w if $u < v$ and $v < w$ then the presence of edge uw implies the existence of edge uv.

In our strips of paper analogy, if fragment u starts before fragment v which starts before fragment w in a left to right ordering, and fragment u overlaps fragment w, then naturally u overlaps all or part of v. An ordering of the fragments where this fails to be true for u, v, w creates what they call an *umbrella* from u to w, as though the edge from u to w curved upwards to avoid v. The Corneil-Olariu-Stewart algorithm involves five LBFS sweeps of the graph. The first sweep is the original Rose-Tarjan-Lueker LBFS algorithm, in which there is an arbitrary choice when several nodes are tied for having the next highest label. The next three sweeps are called LBFS+ sweeps, where ties are decided by the highest index in the previous sweep. The last sweep, called LBFS*, decides among tied vertices by choosing two candidates based on the previous two sweeps. The choice between the two candidates depends on their and their neighbors' edge relationships with nodes that come after the set of tied vertices. The ordering produced by the final sweep is umbrella free if and only if the original graph is an interval graph.

The Corneil-Olariu-Stewart algorithm is linear as is the Booth-Leuker algorithm, but it has the advantage that it can be coded in a few days whereas the latter algorithm can take months of effort. It is not based on finding maximal cliques. However, a linear order of the maximal cliques (as specified by Fulkerson and Gross above) can easily be obtained from the final node order. One disadvantage of the Corneil-Olariu-Stewart algorithm compared to the Booth-Leuker algorithm is that the former does not provide all of the node orderings consistent with the implied constraints, i.e., all of the Olariu-orderings. However, this feature could easily be added to their algorithm (Derek Corneil, personal communication).

The Interval Graph Sandwich Problem

Golumbic and Shamir [36] propose the following problem. Suppose that, when constructing the graph representing the overlap data for the colored fragments of paper, there are some edges of which we are certain, E_1, and some other ambiguous edges E_2 which may or may not belong. A graph $G = (V, E)$ such that the edge set E includes all of E_1 and a portion (none, some or all) of E_2 is called a *sandwich graph* for (E_1, E_2). The *interval graph sandwich problem* is to answer the question: is there a sandwich graph for (E_1, E_2) which is an interval graph [35, 36].

Golumbic and Shamir [36] investigate the complexity of this problem since real versions of the paper strips analogy typically involve ambiguities. Clearly, if the ambiguous set E_2 is empty then the sandwich graph is simply $G = (V, E_1)$, and whether or not it is interval can be answered in linear time as described above. Similarly, if E_1 and E_2 together form the complete set on V, then the answer is 'yes', since the complete graph is an interval graph. The general case, however, is shown by Golumbic and Shamir to be NP-complete. Their proof uses the asteroidal triple-free characterization of interval graphs and a reduction from the Not-All-Equal 3-Satisfiability problem.

In practical terms this result means that the interval graph sandwich problem, unlike the ideal problem, is probably not even polynomial, let alone linear. When the data are ambiguous and noisy, we must resort to heuristics which aim to provide approximate solutions or sets of possible solutions.

2.2 Practice

To convert the paper strips analogy to the physical mapping problem, we replace the initial identical long strips of paper with many copies of the human genome. The colored marks on the paper correspond to unique and identifiable sites (short subsequences) in the genome. Cutting each strip of paper into many fragments corresponds to subdividing the genome with a *restriction enzyme*, an enzyme which recognizes and cuts at a specific four to eight letter sequence. Selecting and numbering a subset of the fragments to analyze corresponds to creating a library of some of the shorter pieces of DNA. The library in this case is a set of yeast clones with human DNA fragment inserts. When a yeast cell in such a clone divides, it reproduces a replica of its own DNA and also a copy of the inserted fragment of human DNA.

A first generation physical map (Cohen *et al.* [14])

As an introduction to the problem of physically mapping the human genome, we review a paper [14] by Cohen, Chumakov and Weissenbach, who head a team of researchers at the Centre d'Etude Polymorphism Humaine (CEPH) in Paris, France. The center has a library of 35000 yeast artificial chromosomes (YACs), cloned fragments of the human genome. This paper provides background information regarding experimental methods and lays out the strategy for the construction of a preliminary physical map. The paper is brief but of considerable interest since it is written by leaders in the field of genetic and physical mapping.

Two reasons are cited for the importance of the YAC library and physical map:

1. medicine — the library and map are urgently needed for the identification of human disease genes, and

2. research — they provide an opportunity to study structure and function of the genome.

Four sources of data are employed by Cohen *et al.* in constructing their maps:

fingerprints Restriction enzymes were used to obtain patterns of fragments for each of 33000 YACs. Since both human and yeast fragments are present in the digest, human medium repeat sequences THE and L1 were used to isolate the relevant fragments. The set of sizes of subfragments of one YAC is compared with that of another YAC to detect overlap.

small single copy landmarks The presence of a Sequenced Tagged Site (STS) on two or more YACs is evidence that these YACs overlap. Cohen improves on this method by using polymorphic STSs which can be genetically mapped to sites in the genome. Two thousand polymorphic STS probes were used to locate the positions of 6600 clones.

large single copy landmarks Hybridization target and probe fragments were generated from 25000 and 5000 YACs, respectively, using the polymerase chain reaction (PCR) applied to sequences between Alu repeat regions. Similar targets were made from monochromosomal somatic cell hybrids. Hybridization experiments then revealed overlaps among YACs and between YACs and chromosomes.

hybridization to metaphase chromosomes Fluorescent In Situ Hybridization (FISH) was used to position 500 YACs on metaphase chromosomes. The YACs selected for this experiment contained polymorphic STSs which were evenly distributed throughout the genome, about one every 7.4 centiMorgan (cM). Results of this experiment allowed correlation among genetic maps (showing positions of polymorphic STSs) and physical maps (showing overlapping YACs) and cytogenetic maps (showing positions of bands on chromosomes).

Since the library provides about tenfold coverage of the genome, and many of the overlap relationships have been determined by the above methods, one might expect that the next part of the paper would present a map showing the sequence of the YACs and markers for each chromosome. The authors point out that a number of biological and experimental artifacts prevent them at this point from attaining this ideal. The following problematic features of this type of data are discussed:

chimerism A large proportion (40-50%) of the clones are composed of two or more fragments from unrelated parts of the genome. Thus, a sequence of overlapping YACs may artificially connect distant portions of the genome.

homologous regions Throughout the genome there are repeated sequences. The number of repeats for a given region varies from millions (for the Alu region which is so frequent as to be considered ubiquitous) down to single copy repeats. STSs are screened in order to avoid using as a probe a region which is repeated, but the authors point out there is no absolute way to prove the success of these efforts.

Like chimers, homologous regions can lead to overlaps between YACs which are not contiguous in the genome.

unclonable regions There will be gaps in the library if certain regions of the genome cannot be cloned in yeast.

unstable regions It is generally assumed that a clone is a constant, unvarying source of a fragment of the genome. However, portions of the insert in a YAC clone may be deleted or rearranged over time.

experimental errors Mistakes in experimental protocol are unavoidable and can lead to false negatives and false positives in the overlap data.

Due to the above difficulties, the authors do not present a definitive map, but rather as the title suggests, a first approximation. Their strategy is to use the four types of data mentioned above in the following way. First, position the genetically mapped STSs on the chromosomes using the FISH data. Second, position other STSs relative to these anchors using overlap data. The number of YACs allowed to stretch across the gap between two genetically mapped STS anchors is termed the level of the map. The higher the level of the map, the less confidence one has in its correctness. At higher levels chimeric YACs or YACs with repeat regions may cause a YAC path to falsely connect two anchors as the path extends from one chromosome to another and back again. On the other hand, the lower the level, the less coverage of the genome. At level 1, only 11% of the total genome is covered with contiguous YACs. At levels 3, 5 and 7 the proportions are 30%, 70% and 87%, respectively.

This paper contains interesting discussion of experimental methods and problems, and invites international collaboration in generating the physical map of the human genome. A subsequent paper by this group [13] presents a more extensive map and more details of the steps taken in map assembly: (1) construct a framework of genetically mapped STSs; (2) use various forms of YAC overlap data to assemble reasonable minimal YAC paths between the mapped markers, where 'reasonable' means not using STSs or YACs that belong to other regions of the map; (3) manually inspect paths for consistency and absence of cycles; (4) extend or improve paths by doing specific experiments; and (5) map some YACs that contain genetic markers to the cytogenetic map using FISH.

An STS-Based Map of the Human Genome (Hudson *et al.* [47])

The above papers present a clone-based approach to physical mapping. The map is anchored to genetically mapped STSs, but the prime objective is to deduce the order of overlapping clones which stretch from one anchor to the next. The paper by Hudson *et al.* [47] takes an STS-based approach. The work is led by the team at the Whitehead Institute/MIT Center for Genome Research (WI/MIT) in collaboration with CEPH. The authors point out that because YAC clones are so highly chimeric and susceptible to rearrangements, they are not suitable for sequencing. Their goal has changed from mapping the clones (with the help of STSs) to mapping 30,000 STSs (with the help of the clones). Given a dense map of STSs, they say that any region of the genome (specified by the STS it contains) can be quickly isolated by screening that STS against a library of clones appropriate for sequencing, such as bacterial artificial chromosome clones or cosmids.

This paper describes the assembly of an STS-based map of the human genome consisting of 15,000 STSs screened against the CEPH YAC library. The map is estimated to cover 95% of the genome. The rate of false positives (data indicating STS hits which are incorrect) is estimated to be less than 5% and the rate of false negatives (absence of STS hits which should be in the data) at about 20%. The steps WI/MIT uses for map assembly are as follows: (1) link together STSs that share at least two YACs, thus forming *doubly-linked contigs*; (2) place the contigs in the map by using genetic and radiation-hybrid data; (3) fill the gaps between contigs with the help of Alu-PCR and fingerprint data and single-linkage STS data; and (4) order the STSs within contigs using a simulated annealing algorithm.

Rule Based Approaches

The leading human genome centers use a variety of types of data and and a mixture of *ad hoc*, manual and formal procedures to piece the data together into a map. Here we review a paper titled *CPROP: A Rule-Based Program for Constructing Genetic Maps* by Stanley Letovsky and Mary B. Berlyn [58], describing a formal rule-based approach to genetic mapping. The authors have also used their method for physical mapping and have placed a physical map of the human genome on the World Wide Web.

Letovsky and Berlyn describe a software utility which they have designed for the construction of genetic maps. The software package involves constraint propagation, and hence is called CPROP. Constraint propagation is a method borrowed from artificial intelligence, where it is applied to the problem of task scheduling. Construction of genetic maps is similar to task scheduling in that both involve creating a linear sequence of items (genes or tasks) given certain pairwise constraints, such as *A is before B*. There are also differences between these two problems. For example, genetic maps often contain sections whose local order has been deduced but whose global orientation is unknown, (something not found in task scheduling), while task scheduling may include deadlines (a concept absent in genetic mapping). The CPROP system was originally developed to aid in the construction of genetic maps for bacterial data.

A distinctive feature of CPROP is that the map which it creates represents not only what is known about the order and separation of the genes, but also what is not known. There are two ways that knowledge and uncertainty are conveyed in CPROP. Knowledge of ordering is represented by pairwise relations like 'gene A is before gene B', and knowledge of separation is represented by a measure of distance between genes A and B. Uncertainty in ordering is captured by *local ordering windows* (LOWs), which represent a frame of reference within which the terms *left, right, before,* and *after* have meaning, but a meaning that could be reversed in the global (chromosomal) frame of reference. Uncertainty in the separations of genes is represented by giving lower and upper bounds on separations.

The lower and upper bounds may be thought of as 100% confidence intervals, although the confidence must of course be less than 100%. The authors admit to imprecision regarding the statistical meaning of the confidence intervals, but they cite two justifications for their approach. First, most of the data on separations in bacterial genetics are not published with statistical variation, partly because biological variation is even greater than variation arising from measurement errors. Second, even when statistical variation is known, the decision to use a 95% or 98% or 99.5% confidence interval (or something else) is *ad hoc*. Whatever the lower and upper bounds on a separation mean exactly, the idea is that the true separation should lie somewhere in that range. If through constraint propagation, CPROP detects an inconsistency, the user is notified and advised to change certain bounds.

There are many experimental techniques for obtaining genetic data. Nevertheless, the various forms of data are reduced to tuples of the form

(locusA, locusB, i, LowerBound, UpperBound)

for input to CPROP. The above tuple means that locus A is before locus B in local ordering window i (an integer), and their separation is between LowerBound and UpperBound. For each LOW i there is another LOW $-i$ which contains the opposite local ordering. Given these inputs, CPROP uses the mechanical inference engine of constraint propagation to draw *derived constraints*. Deductions are based on simple inference rules such as *Transitivity*: if A is before B and B is before C in LOW i, then A is before C in LOW i. An example of a rule of inference for separations is the *Triangle Equality*: if A is before B and B is before C, and the pair of bounds on the separation of A and B is $[l_1, u_1]$, and the pair of bounds on the separation of B and C is $[l_2, u_2]$, then the separation of A and C is given by $[l_1 + l_2, u_1 + u_2]$.

The rules are applied iteratively to achieve *closure*, i.e., until no further deductions can be made. However, an important point is that the rule set is not *complete*. Not all orders and separations are derived from the data that could possibly be derived. This apparent shortcoming is actually a bonus. If the rule set were complete, then the algorithm would be exponential since a related problem called *satisfaction of betweeness constraints* is NP-hard. The authors have made the rules such that CPROP does its calculations in polynomial time. Their experience suggests that CPROP's ability to reason about genetic constraints is equal to that of a human. (This is not to say that CPROP's ability in map construction equals that of a human since CPROP relies on the user to resolve conflicts in constraints.)

When deriving constraints, CPROP applies rules to closure. For example, it obtains the *transitive closure* of the *before* relation. Obtaining the closure is important since the derivation of a fact may trigger the application of a rule. However, when presenting the *result map*, some simplification is required. If A is before B and B is before C, then the user typically only wants to see those two facts and not the third related fact that A is before C. Thus, at the end of the deduction period, CPROP does the opposite of transitive closure, *transitive reduction*, to simplify the set of conclusions. In addition to a table of conclusions, CPROP draws a map representing the positions of the loci and their error bars. The map is not as detailed as the table and does not contain as much information, but it is more readily understood.

In summary, Letovsky and Berlyn describe an integrated system of programs called CPROP for construction of genetic maps. CPROP takes over a job which was previously performed painstakingly by humans. The formation of databases containing genetic mapping information and the desire for reproducibility motivated the creation of CPROP. Other analytical tools for making genetic maps use a *maximum likelihood* approach, where a map is presented as a *total order* with an associated probability that the given total order is the true order. The philosophy behind CPROP is that it is better to represent the map as a *partial order*, where uncertainty in order is made explicit in the representation. For example, if the relative order of loci A and B is not known, then they are placed side by side in the map. The CPROP system also does consistency checking, and in the face of inconsistencies it awaits resolution by the human user.

At least two other groups have also investigated the application of formal logic systems to genome mapping [56, 82]. These rule-based approaches are founded on an interval algebra which relates pairs of intervals by binary relations such as *not-after, not-contained, disjoint, overlaps, contains*, to name a few. As in Letovsky's system, each relation is said to hold in a local orientation frame or window. For each such window there is a complementary window in which the order of fragments is reversed. A table of reduction rules is given so that within a local orientation frame pairs of relations may be reduced to a simpler relation. The satisfiability problem on the complete network of relations is NP-complete, but Schmeltzer has shown that restrictions of the problem may be solved in $O(n^3)$ time [82]. I implemented a simplified version of the system described by Lee *et al.* [56] using the Hy+ visualization system [15]. Queries ran very slowly in my implementation (results not shown), even for small data sets, perhaps because the system required formation of relations on all pairs of fragments. Schmeltzer [82] implements his system on an existing temporal reasoning software platform and uses the Floyd-Warshall algorithm to reduce the relation network. He mentions that he has tested his system only on small data sets and is now including the system in a larger knowledge representation system for genomic data.

Mapping the Genome: Some Combinatorial Problems (Karp [51])

This paper [51] by R.M. Karp is an overview of some of the combinatorial problems associated with the molecular biology of DNA. The computer scientist is encouraged to attack these problems with the arsenal of algorithms developed for other purposes, sometimes purely theoretical. Karp adds a note of caution warning that the computer scientist should avoid the temptation of looking solely at the more tractable noiseless versions of these problems. Experimental noise, error and variation are inescapable in biological science.

Karp briefly reviews biological terms and methods, and he makes the interesting analogy that a probe provides a kind of random access into the genome. He discusses the Lander-Waterman model, which is fundamental to physical mapping experiments, in which one attempts to determine the order of probes and clones on a chromosome. Of primary interest is the definition of *coverage* as $C = nL/N$, where n is the number of clones, L is the length of each clone (assumed to be constant), and N is the length of the DNA. Lander and Waterman [55] show that the number of *gaps*, regions of the DNA not found in clones, is approximately $ne^{-C} + O(1)$, and the fraction of DNA not covered by any clone is e^{-C}. We can apply this formula to the CEPH library mentioned above, which has 35,000 YACs, each of about a million base pairs in length. Since the human genome has 3 billion base pairs, the coverage is about $(35000 \times 10^6 / 3 \times 10^9) = 11$. Using the Lander-Waterman formulas, one would expect only 0.00002 of the genome (i.e., about 60,000 bp) to have no corresponding clone, and just .6 + O(1) gaps, i.e., close to as few gaps as possible. However, as we shall see in the next chapter, the data cannot be trusted when there is only a single clone covering a particular region, so the physical map would have more gaps than implied by this calculation.

Subsequent sections of the paper review some of the problems associated with physical mapping and the approaches taken to date, with emphasis on work from Karp's group. These problems are outlined below.

The double digest problem is the problem of reconstructing the sequence of clones or sections of DNA given three sets of fragment lengths for each clone, sets A, B, C. Set A is composed of lengths obtained when the clone is completely digested with restriction enzyme A. Similarly, set B comes from digestion with enzyme B. Set C consists of lengths arising from simultaneous digestion with enzymes A and B. Even in the noiseless case, this problem is NP-hard [51].

The partial digest problem is similar to the double digest problem, except that a single enzyme is used and the digestion is only partial. This time the set of lengths is assumed to contain the length of separations between each pair of restriction sites on the clone. The complexity of this problem is the same as that of polynomial factorization, and thus unlikely to be NP-complete, according to Karp. In the presence of noise, probabilistic approaches are taken, treating it as a maximum likelihood problem [51].

Physical mapping using unique probes presents a problem which, in the noiseless case, is much simpler than the above two approaches to physical mapping. In fact, the problem reduces to the problem of determining if a matrix has the *consecutive ones property for columns*. As we have discussed earlier, there are linear time algorithms for this problem. In the presence of noise the problem becomes one of finding the minimum number of zeroes that must be changed to ones in the matrix to convert it into a matrix having the consecutive ones property. This problem is NP-hard. Karp outlines a fast method which his group has invented to deal with this problem. The method assumes that there are no false positives and that no clone properly contains another. A bipartite graph is formed with probes in one partition and clones in the other and edges between probes and clones which are associated (corresponding to the ones in the matrix). A divide and conquer approach is then applied by deleting from the graph a probe and all of the clones to which it is adjacent. The problem is thereby split into three parts: a left component, a right component and a group of isolated vertices. The strategy is applied recursively until it will not subdivide, at which point brute force is invoked. In the case of false positives they use simulated annealing. Karp mentions that they are studying methods to deal with chimeras.

Physical mapping using nonunique probes is a somewhat harder problem than its unique probe counterpart. Karp approaches this problem using a Poisson model for the placement of probes along the DNA. The problem is cast as a maximum likelihood problem, and as such seems quite intractable. Karp simplifies the problem by approximating the likelihood function so that solutions can be obtained in practice, though it is still NP-hard [51].

Mapping using interval graphs involves first estimating the probability that two clones overlap, and second finding an interleaving of the clones (i.e., physical map) which best agrees with these probabilities. The problem becomes one of finding an interval graph which maximizes a certain formula relating the agreement of the interleaving to the probabilities of overlaps. This problem also is NP-hard [35, 36], but Karp's group has some heuristics borrowed from the Lin-Kernighan algorithm for the traveling-salesperson problem which give solutions in quadratic time.

Sequencing problems are analogous to the physical map problem, but deal with short sequences of DNA. Fragments of about 200 to 500 base pairs in length can be sequenced, and the problem is to reconstruct the sequence of the whole DNA molecule. This problem is called the *shortest superstring problem*: given several short strings, find the shortest string that contains all the short strings as substrings. In the presence of noise and error, the problem becomes one of determining the most likely sequence for the DNA molecule which is the source of the imperfectly sequenced fragments.

In conclusion, Karp shows that there are a number of interesting and difficult problems in the field of molecular biology, in particular in DNA research, which computer scientists should consider within their domain. The paper ends with the admonishment not to ignore noise and error when addressing these problems. Combinatorial optimization, the most common approach to the above problems, offers a solution which in some sense is 'most likely'. Karp makes the interesting point that such solutions are only of value if they are very close to the true answer. He suggests that further experimentation should follow up on such a solution to attempt corroboration. It seems clear that either there should be some representation of the uncertainty associated with a suggested solution or that groups of possible solutions should be represented.

Chapter 3

Characteristics of STS/YAC Data

In this chapter I provide background and basic characterization of the human genome mapping data which I analyze in subsequent chapters. Some of the results in this chapter appear in papers by the author, A. Bonner and N. Goodman [44, 45, 46].

3.1 Background

Detailed study and sequencing of the human genome requires indexed libraries of clones of overlapping sections of human DNA. Such libraries serve as reservoirs of genomic fragments for subsequent study of sequence, function and malfunction of genes. The index takes the form of a *physical map* which assigns each clone to its location in the genome. The map together with the library provides direct access to specific locations in the genome, thus facilitating the isolation of disease genes, annotation of genes in terms of expressed proteins, and the assembly of sequence data.

Typically a genome center will have several libraries, each characterized by a different average length of cloned fragment. A library with large inserts is valuable for 'walking' the genome in search of a particular gene and also for providing long range continuity in sequencing studies. In 1987 Olson and coworkers [11] described a method for cloning large exogenous fragments of DNA in yeast artificial chromosomes (YACs). By 1990

Olson's group [48] improved the technique by selecting fragments of particular lengths to clone and by increasing the average insert size to 380 kb. Further improvements to the technique allowed CEPH/Généthon [13] to construct a library of megabase-sized human-genomic fragments. CEPH/Généthon made this library available to the genome community for use in the Human Genome Project. Large scale physical mapping of the library was undertaken both at CEPH/Généthon and WI/MIT genome centers. CEPH produced clone-based maps in 1993 [14] and 1995 [13]. WI/MIT in collaboration with CEPH/Généthon assembled an STS-based map of the human genome consisting of 15,000 STSs screened against the CEPH/Généthon YAC library [47].

Cohen *et al.* [14] point out that assembly of these physical maps is complicated by a number of biological phenomena and experimental artifacts, such as chimerism, homologous regions, unclonable regions, unstable clones, distribution biases of genetic markers, non-uniform efficiency of probe screening, and experimental errors. These problems necessitate careful analysis of several forms of data, such as hybridization of probes and clones, restriction enzyme fingerprinting of clones, radiation-hybrid mapping of probes, genetic mapping of probes, and fluorescent in-situ hybridization (FISH) of clones to metaphase chromosomes. Hybridization data indicate complementarity between probes and clones. The probes may be derived from polymorphic sequenced tagged sites (STSs) or inter-Alu sections of clones, and detection of hybridization is achieved through the polymerase chain reaction (PCR). Hybridization and fingerprint data indicate which pairs of clones overlap, while the mapping experiments localize clones or probes to particular regions of the chromosome.

3.2 Source of data

The primary source of STS data in my research is the Whitehead Institute/MIT Center for Genome Research (WI/MIT), in particular, Release 10 (May, 1996), available by anonymous ftp (genome.wi.mit.edu, directory /pub/human_STS_releases/may96). The source for fingerprint and Alu-PCR data along with additional STS data is the Centre d'Etude du Polymorphisme Humain (CEPH) and Généthon, in particular, the March 1995 Release available by anonymous ftp (ceph-genethon-map.cephb.fr, /pub/ceph-genethon-map/STS/29MAR95.DAT). These data were used by the genome centers to form the maps of the human genome published by Chumakov *et al.* [13], Hudson *et al.* [47] and Schuler *et al.* [84].

3.3 Probe data

An STS, or sequence-tagged site, is a known sequence of about 300 base pairs (bp), whose presence on a clone can be assayed by the polymerase chain reaction (PCR). STS probes serve two purposes: they mark a unique site in the genome, and they indicate which YACs overlap. Mapping methods often rely heavily on STS screening of the clones in the library.

Table 3.1 characterizes the STS data from WI/MIT Release 10 (May, 1996) and the STS data from CEPH/Généthon (March, 1995), and the union of the two data sets. I define a *hit* as a definite or disambiguated address in the WI/MIT data and an address without the verified negative annotation (N) in the CEPH/Généthon data. The YAC count includes only those hit by at least one STS, and the STS count includes only those that hit at least one YAC. A total of 20,203 STSs were screened against 25,344 YACs in the WI/MIT data. It may seem strange that the standard deviation of the hits per YAC is less in the merged data (3.2) than in the two separate parts (4.3 and 3.5). However, most of the YACs are the same in the two data sets, so some convergence in the number of hits per YAC is to be expected. When I refer to STS data subsequently, I will be referring to the merged CEPH/Généthon and WI/MIT data of column 3 in Table 3.1.

Table 3.1 shows that the average number of YACs hit per STS is about 6, with

	WI/MIT	CEPH/Généthon	Merge
STSs	12,527	7,026	15,136
hits/STS	6.40 ± 3.9	5.3 ± 3.9	6.2 ± 4.0
YACs	21,051	18,298	24,855
hits/YAC	3.8 ± 4.3	2.0 ± 3.5	3.8 ± 3.2

Table 3.1: Statistical characterization of STS data.

a standard deviation of 4. This statistic reflects the *observed* YAC coverage, i.e., the average number of YACs containing a particular region of the genome. The *expected* YAC coverage can be estimated by the ratio of the total length of the YACs to the total length of the genome. The average length of the YACs in Table 3.1 is 1,005 kb (standard deviation 480 kb; kb = 1000 base pairs). YAC sizes were obtained by anonymous ftp from `ceph-genethon-map.cephb.fr` in `/pub/ceph-genethon-map/SIZING`. Since there are about 25,000 YACs, the total length of YACs is about 25×10^9 base pairs. The genome is estimated to be 3 billion base pairs, so the expected YAC coverage is eight (or ten if we include 4,000 YACs which were screened but not hit by an STS). Comparison of these two estimates of coverage suggests a 25-40% rate of *false negatives* in the merged STS data, i.e., instances where an STS is on a YAC but the corresponding probe does not hit it. This is in line with an estimate by WI/MIT [47] of 23% false negatives in their STS data.

The large standard deviation in hits per STS indicates nonuniformity in YAC coverage and/or in the rate of false negatives. The distribution of hits per STS is shown as a bar graph in Figure 3.1(a). From this figure we see that the number of YACs most commonly hit by an STS is one. Over the range of two to seven hits per STS the number of STSs is fairly constant. Thereafter, the frequency of STSs gradually tapers off reaching 0 beyond 30 hits per STS. The bar at 15-30 hits per STS represents the sum over this range. Table 3.1 also shows that on average 3.8 STS probes hit each YAC; (YACs that

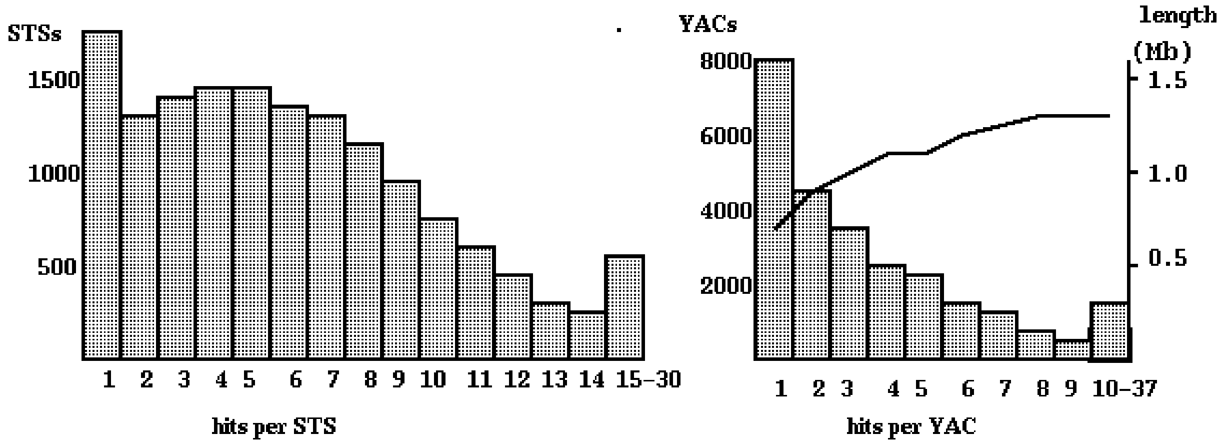


Figure 3.1: (left) Hits per STS probe; (right) hits per YAC

have no hits are not included in this average). Again, the large standard deviation (3.2) reflects nonuniformity in the data. Figure 3.1(b) shows the corresponding distribution. The most frequent number of hits per YAC is 1, and thereafter the frequency decreases monotonically as the number of hits per YAC increases to 37. The bar at 10-37 hits per YAC represents the sum over this range.

The YACs in the merged STS data of Table 3.1 vary in size from 10 kb to 2,500 kb (average 1,005 kb $\pm$ 480 kb). The lengths are plotted as a function of hits per YAC in Figure 3.1(b). It appears that longer YACs tend to include more STSs, but certainly not proportionately, since while the number of hits goes through five doublings from 1 to 32, the average length barely doubles once. A caveat to this observation is that CEPH/Généthon states that "for most cases, the resolution was limited to the size of 1700-1800 kb." In cases where several sizes are recorded for the same clone, we have used the average. The curve is only slightly raised if in place of the average we use the total of the sizes reported for a single YAC.

The bar graphs of Figure 3.1 reveal some of the complexity hidden by the simple average statistics listed in Table 3.1. Although the YAC coverage is nominally 8 to 10 throughout the genome, an STS is twice as likely to hit just a single YAC as it is to hit ten. Although there are enough STSs that there could be one every 200 kb, i.e., five on a typical megabase-size YAC, the most common number per YAC is 1, and some YACs have over 30. These appear to be examples of two problems mentioned by Cohen [14], — distribution biases and nonuniformity of probe screening.

3.4 Overlap Data

Two clones A and B are inferred to overlap and form an unordered *overlap pair* based on fingerprint data when the CEPH/Généthon file `RELATIONS` lists clone B among the clones overlapping with clone A (and vice versa). The fingerprint data give rise to 49,383 overlap pairs among 31,392 YACS (yeast artificial chromosomes). Overlap of clones A and B is also inferred if one of the two clones, acting as an Alu-PCR probe, hybridizes with the other at a unique address, i.e., the hybridization is scored 'Unique' (U) by CEPH/Généthon. The CEPH/Généthon data show the results for experiments with 8,785 YACs used as Alu-PCR probes and 24,576 YACs used as targets. These Alu-PCR data produce 55,891 distinct overlap pairs.

A third method of inferring overlap of clones A and B is by observing that they both hybridize with the same STS probe. Using STS probe data from WI/MIT (12,527 STSs and 21,051 YACs) and CEPH/Généthon (7,026 STSs and 18,298 YACs) I derived 364,536 overlap inferences, comprising 215,397 distinct overlap pairs. Overlaps inferred from the STS data corroborate 54% of the overlaps from the fingerprint data and 37% of the overlaps from the Alu-PCR data. The number of distinct overlap pairs in all three forms of data (STS, fingerprint, and Alu-PCR) combined is 271,257.

The table in Figure 3.2(a) compares the amount of overlap information in the WI/MIT and CEPH/Généthon merged dataset as provided by each of these types of data. The STS portion of this data set produces about four times as many overlap pairs as does the

Data Type (no. of probes)	Number of YACs	overlap inferences	distinct overlap pairs
STS (15,136)	24,855	364,536	215,397
Alu-PCR (8,785)	24,576	59,437	55,891
Fingerprint	31,392	49,383	49,383
Combined		473,356	271,257

a)

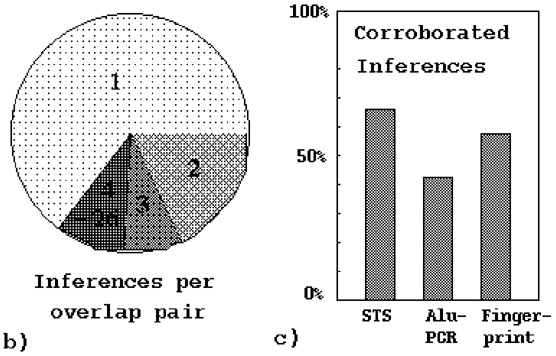


Figure 3.2: Overlap statistics for merged data

Alu-PCR or fingerprint data. An overlap between clones A and B can be inferred independently once for each STS probe which hybridizes with both A and B, once if A is an Alu-PCR probe which hybridizes with B, once if B is an Alu-PCR probe which hybridizes with A, and once if A and B have similar fingerprints. The weight of an overlap may be defined as the number of inferences supporting it. The pie chart in Figure 3.2(b) shows the percentage of overlap pairs with given weights when all three sources of overlap data are combined. Almost two thirds of the overlap pairs are supported by a single inference. The bar graph in Figure 3.2(c) shows that the percentage of overlap inferences of each data type which are corroborated by another inference of any data type. According to this chart, it would appear that the Alu-PCR data set is slightly less reliable than STS or fingerprint data sets.

3.5 Noise

Each STS is designed to correspond to a single site in the genome and thereby of course to a single chromosome. Unique chromosome assignments have been determined by WI/MIT and CEPH/Généthon for 95% of the STSs of the merged group of Table 3.1. YACs, in turn, can be associated with particular chromosomes indirectly by virtue of the chromosome assignments of the STSs that hit them. Table 3.2 shows the distribution of

Chromosome	Number of STSs	Number of YACs
Chr1	1,135	2,629
Chr7	660	1,782
Chr14	414	1,104
Chr21	324	1,078
ChrX	649	1,234
ChrY	78	168
unassigned	761	496
multiply assigned	0	7,546
total	15,136	24,855

Table 3.2: STS assignments and YAC associations to a few representative chromosomes.

STSs and YACs for a few representative chromosomes.

Table 3.2 shows that there are typically two or three times as many YACs associated with a chromosome as there are STSs assigned to it, even though the total number of YACs is not quite twice the total number of STSs. This anomaly is a consequence of multiple associations for some YACs. About 30% of the YACs are associated with more than one chromosome, whereas there are no multiply assigned STSs. Some explanations for the ambiguity in YAC identity are described below.

Chimeras Some YAC addresses are chimeric, in that they appear to represent more than one genomic fragment. Three possible causes of this phenomenon are: (1) in some cases there may be more than one YAC in a well ('pseudochimeras'); (2) genomic fragments may be ligated together during the process of adding the vector arms to the insert ('co-ligation chimeras'); and (3) two YAC molecules may be transformed into one host cell, and later be recombined to form a hybrid molecule by the DNA repair machinery ('post-transformation chimeras') ([39], [40] and Thomas Hudson, personal communication). CEPH/Généthon estimates that 40-50% of the YACs in their library are *chimeric* [13]. Therefore, many YACs which are hit by STSs from more than one chromosome may in fact contain pieces of more than one chromosome.

Repeat regions It is well known [98] that sequences ranging in size from 100 bp to 100,000 bp may be duplicated and appear at several locations in the genome. Some, like the Alu sequence, occur millions of times. Although STS probes are screened to avoid these repeat regions, some nonunique probes may slip through the screening process. Thus, a YAC containing a repeat region common to two or more chromosomes may be hit by STS probes assigned to these chromosomes and thereby become associated with two or more chromosomes. Evidence of nonunique STSs will be presented when we discuss graphical analysis of these data, but their frequency can be assumed low compared to that of chimeras.

False Positives A third cause of multiple chromosome association is random false positives. In addition to being hit by STSs from its own chromosome, a YAC may, through experimental error, appear to be hit by an unrelated STS. WI/MIT estimates that 6% of the STS hits are false positives.

Of the three above causes of ambiguity in YAC identity, chimerism appears to dominate. In fact, it may seem odd that only 30% of the YACs are associated with more than one chromosome when 40-50% of the YACs are chimeric. One factor contributing to this discrepancy is that chimeras consisting of fragments from a single chromosome are not detected by differences in chromosome assignment of STSs. A more important factor is the density of STSs. If a YAC is hit by just one STS (the mode in the right panel of Figure 3.1), then even if the YAC is chimeric, it can only be associated with one chromosome. That is, the probability of discovering that a chimeric YAC is chimeric becomes nonzero for two STS hits and increases with STS density, but is never unity.

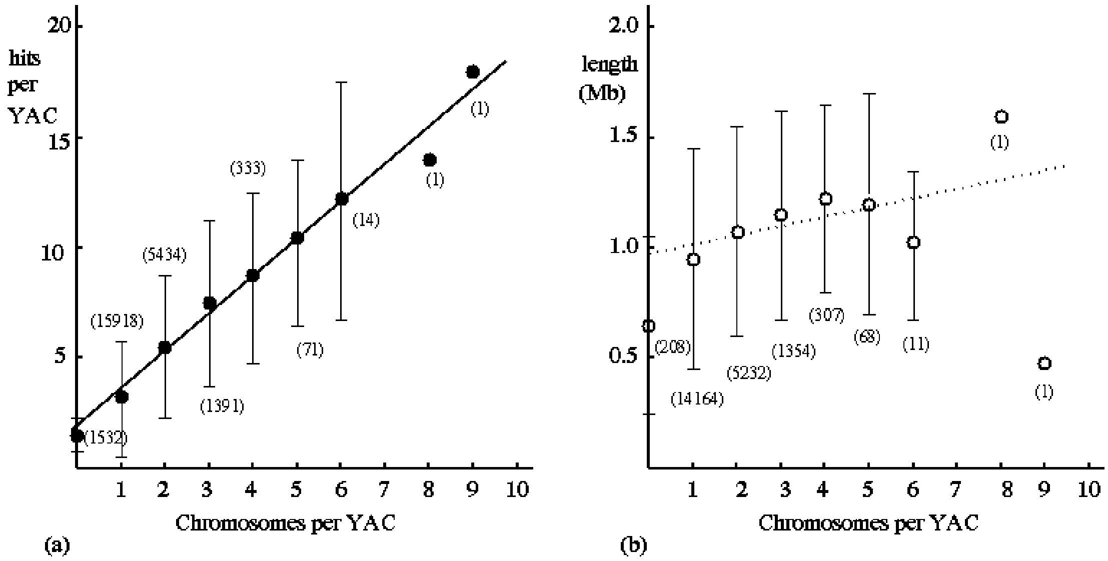


Figure 3.3: (a) Hits and (b) length per YAC vs. number of chromosome associations

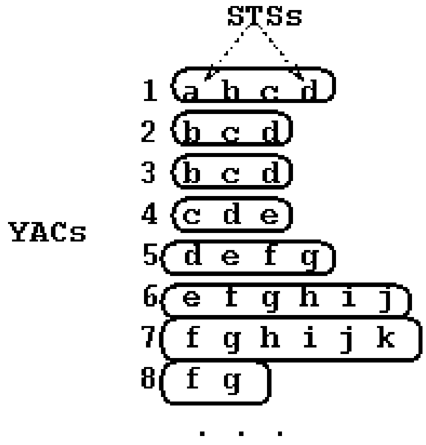
CEPH/Généthon has analyzed the percent of YACs which are chimeric as a function of size of YACs, using Alu-PCR hybridization data [13]. They list a number of difficulties with this method of analysis, but conclude that in their library of medium sized YACs (the Mark II library) the larger YACs contain a higher percentage of chimeric YACs than the smaller ones, whereas in the Mega-YAC library the fraction of chimeric YACs was independent of size (being about 30% for sizes ranging from about 50 kb to 1700 kb). In Figure 3.3 I complement this result by analyzing the *degree* of chimerism per YAC, i.e., the number of chromosomes from which the YAC is apparently made. Thus, I divide the YACs into groups depending on the number of chromosomal associations per YAC. The number of YACs in each group is shown in parentheses near each data point in Figure 3.3. In the vertical direction of the left panel (a), we plot the average number of STSs that hit the YACs with the indicated number of chromosome associations. Standard deviation bars are shown except where there is only one YAC in the sample. The total number of YACs with more than one chromosome association is 7245, or 29.3% of the total (24,695). The plot in Figure 3.3(a) shows that hits-per-YAC is linearly correlated with the number of associations per YAC. This is natural considering that to be associated with n chromosomes, a YAC must be hit by at least n STS probes, each belonging to a different chromosome. The slope of the line, however, suggests that to reveal n chromosomal associations requires about $2n$ hits on average. Thus, it may be expected that more YACs will be found to be chimeric and higher multiplicity of chromosome-associations will be revealed as STS density is increased.

Figure 3.3(b) shows the average length (CEPH measurement) of the YACs for each class defined by the number of chromosome associations per YAC. Numbers in parentheses represent the number of YACs in the class. This plot shows that there is not a strong correlation between the length of YACs and the number of chromosome associations, contrary to what one might suppose. Presumably, recombination of parts of the chimera within the host cell and a selective advantage for growth of cells with smaller inserts contribute to this result (Thomas Hudson, personal communication).

Chimeric addresses, nonunique STSs and false positives imply incorrect proximities in a genomic map, while false negatives and gaps in coverage weaken the ability to infer proximity. False links between chromosomes (as determined by the assignment of the STSs involved) are easily detected and removed. False links within a single chromosome are more troublesome. In the next chapter, I show that graphs are useful for the detection and analysis of these experimental artifacts.

3.6 STS Intersection Graphs

STS data can be represented as a *matrix*, where the rows correspond to YACs and the columns correspond to STSs. An entry at position (i, j) in the matrix can be {1, 0 or ?}, meaning that STS j {hits, does not hit, or ambiguously hits} YAC i. WI/MIT uses a probabilistic method to convert some question marks to zeros or ones, as described in their data release. I have used the graphical logic programming and data visualization system Hy+ [15] to implement a rule used by WI/MIT to resolve ambiguous hits [43],



	a	b	c	d	e	f	g	h	i	j	k	...
1	1	1	1	1	0	0	0	0	0	0	0	...
2	0	1	1	1	0	0	0	0	0	0	0	...
3	0	1	1	1	0	0	0	0	0	0	0	...
4	0	0	1	1	1	0	0	0	0	0	0	...
5	0	0	0	1	1	1	1	0	0	0	0	...
6	0	0	0	0	1	1	1	1	1	1	0	...
7	0	0	0	0	0	1	1	1	1	1	1	...
8	0	0	0	0	0	1	1	0	0	0	0	...
..					...							

Figure 3.4: STS-YAC data and matrix representation.

On the left, YACs labeled $1 \ldots 8$ *are sketched with STS sites labeled* $a \ldots k$. *On the right is part of the matrix corresponding to this picture, assuming that data were gathered by a perfect experiment. A '1' at position* (i, j) *in the matrix indicates that YAC* i *is hit by STS probe* j. *A '0' indicates a miss.*

but here we simply use the disambiguations provided by WI/MIT. We treat WI/MIT disambiguated addresses as hits (ones) and other ambiguous addresses as misses (zeroes). Figure 3.4 contains a sketch of STS positions on YACs and the corresponding binary matrix for data gathered by an imaginary perfect experiment.

One goal of physical mapping is to translate the matrix of experimental data into a *model* that shows the relative positions of STSs and YACs in the genome. An *interval* model representing STS-YAC data is a set of intervals (the YACs) and a set of points (the STSs) on the real line. For example, a model consistent with the data of Figure 3.4 is shown in Figure 3.5. In this schematic, YACs are represented by horizontal line segments and STSs by vertical bars. Intersection of a bar with a line segment indicates that the STS probe hits the YAC. Although the model in Figure 3.5 successfully explains the data of Figure 3.4, it is not unique in this respect. The relative order of some probes in the model is arbitrary. For example, positions of probes f and g could be interchanged

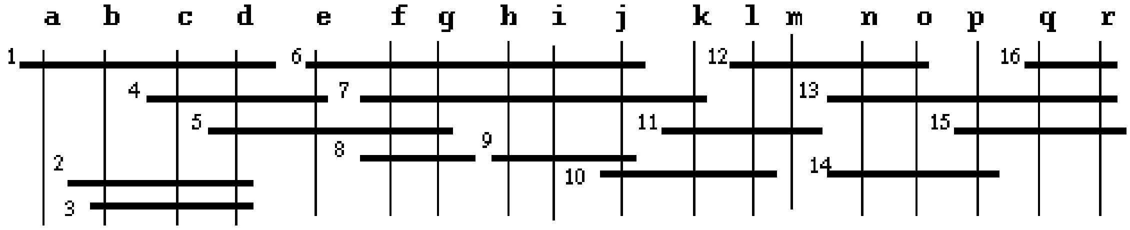


Figure 3.5: Interval model of STS-YAC data

YACs are represented as horizontal line segments; STSs as vertical bars.

without affecting the degree of consistency with the data, since these probes hit exactly the same subset of YACs.

Typically no interval model exactly matches real data. Experimental noise and artifacts create inconsistencies and invalidate the assumptions inherent in the interval model. For example, it might be assumed that if two STS probes hit at least n YACs in common (for some appropriate integer n), then the corresponding STSs are no farther apart in the genome than the length of a YAC. Figure 3.7 (discussed below) shows that there is probably no value of n which will guarantee the validity of this assumption, but for $n = 2$ it may be correct about 94% of the time. Therefore, before building a detailed model, it is helpful to look at the data in a general way to see that they meet certain preconditions. Genome mappers typically do this in various *ad hoc* ways, such as checking that chromosome assignments and genetic positions of connected STSs are reasonable. However, positional data are not always available. In the next chapter, I present an independent method for checking the coherence of genomic data. Our graphical verification method can reveal problems even in the absence of chromosome and genetic assignments.

We start by representing the STS-YAC data as a weighted intersection graph. In this type of graph, nodes represent sets of items, and there is an edge between each pair of nodes that have a nonempty intersection. Each edge is assigned a numerical weight equivalent to the size of the intersection. Weighted intersection graphs are described in

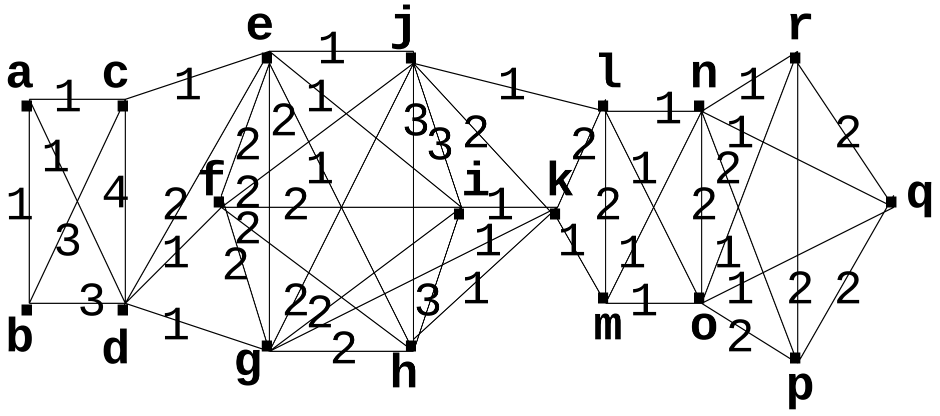


Figure 3.6: Single-linkage STS-intersection graph for example data.

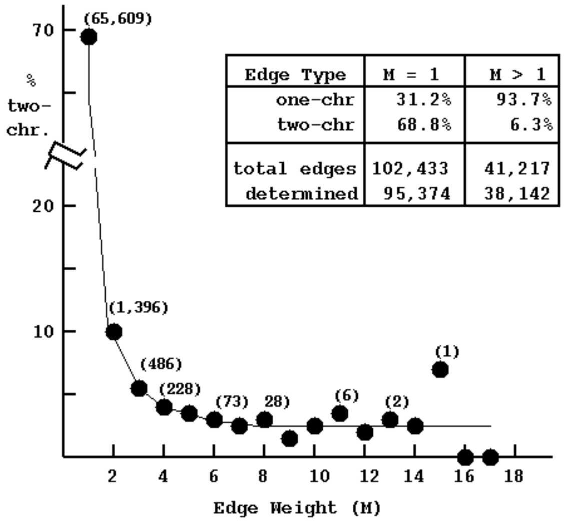
The nodes correspond to STSs in the example data of Figures 3.4 and 3.5. An edge with weight M between two nodes indicates that the corresponding two STSs hit M YACs in common.

texts on graph theory, for example Harary [41].

In the STS-intersection graph, each node corresponds to an STS and indirectly to the set of YACs hit by the STS, i.e., a cluster of mutually overlapping YACs. In general, a cluster could be constructed from many sources of overlap data, as described in Chapter 6. There is an edge of weight $M > 0$ between two nodes if the corresponding STSs hit M YACs in common. Figure 3.6 shows the intersection graph for the example data of Figures 3.4 and 3.5. Edge (a, b) has weight $M = 1$ since probes a and b hit one YAC in common, whereas edge (b, c) has weight $M = 3$, since b and c hit 3 YACs in common. This graph captures the relationships detailed in the matrix in a visually meaningful form.

The intersection graph for the merged WI/MIT and CEPH/Généthon STS data consists of 665 connected components, one of which contains 94% of all the STS nodes. This anomalous aggregation of most of the nodes into a single component must be caused by edges which link STSs belonging to different chromosomes. The table inset in Figure 3.7

characterizes edges according to their weight and whether they are *one-chromosome*, i.e., having endpoints assigned to the same chromosome, or *two-chromosome*, i.e., having endpoints assigned to two different chromosomes. If one or both endpoint STS probes are unassigned then the edge is classified as *undetermined*. It can be seen from the inset table that there are many edges which ‘cross’ chromosomes: 68.8% of the unit weight edges and 6.3% of the nonunit weight edges (not counting undetermined edges) are of the two-chromosome type. These values appear in the second row of the inset table.



Edge Type	M = 1	M > 1
one-chr	31.2%	93.7%
two-chr	68.8%	6.3%
total edges	102,433	41,217
determined	95,374	38,142

Figure 3.7: Chromosome content and edge weight.

The general infidelity of unit weight edges ($M = 1$) suggests that most unit weight edges are caused by chimeric YACs and random false positives. Chimeras and random false positives are expected to give rise to edges of unit weight since it is unlikely that two noncontiguous regions of the genome will be linked at random (physically as a chimera or experimentally as noise) more than once. On the other hand, a YAC that connects contiguous regions of the genome, has a good chance of contributing to a strong ($M > 1$) edge in the intersection graph, since there will often be other YACs connecting the same regions (assuming good coverage in that region). Most good links will therefore have nonunit weight. To help make the intersection graphs reflect proximity in the genome, we consider the subgraphs for which $M \geq 2$. This is the *double-linkage* strategy [4] used by WI/MIT.

Double linkage is a good method for eliminating spurious weak connections caused by chimeras or random false positives, but it provides no defense against the many spurious strong links ($M > 1$). It seems unlikely that chimerism accounts for many of these edges because of their number and weight. Assuming a random chimeric process, the probability of a chimeric two-chromosome link with a weight M decreases very rapidly with increasing M. One expects few such links with weight 2, and next to none with weight 3 or more. However, Figure 3.7 shows that the weights of two-chromosome edges run as high as $M = 15$, and there are close to a thousand two-chromosome edges with $M > 2$. It seems likely that most of the nonunit weight edges that link STSs on different chromosomes result from nonunique STSs, i.e., STSs that hit repeat regions in the genome.

The presence of strong spurious edges ($M > 1$) creates links and loops among what would otherwise be separate components of the intersection graph. This effect is represented by a schematic in Figure 3.8. Imagine that an intersection graph of STSs in a contig, like the graph in Figure 3.6, can be compressed along its long axis into a straight line segment. (In Chapter 4, we will see how to do this.) Panel (a) of this Figure 3.8 shows four separate contigs represented in this way as straight line segments. Panel (b) shows the kind of branching structure that can result when the contigs are tied together by spurious cross links (points **a, b** and **c**). The spurious link **d** within contig **C4** causes a large scale loop. As a result, the intersection graph has fewer components than it should have, and it displays a complex web-like pattern which is difficult to disentangle and map.

For example, consider the intersection graph for the merged STS data of Table 1. Even after filtering out edges of unit weight, over one-third of the STSs (about 5700) fall in one anomalously huge connected component. There are about 4000 small connected

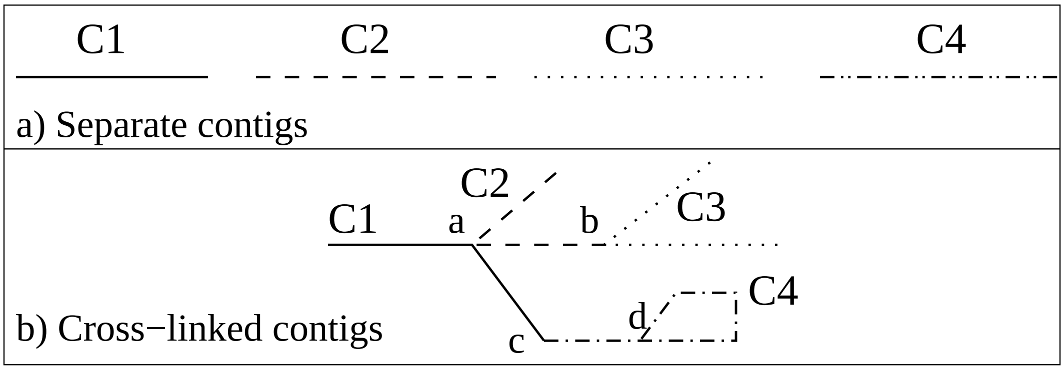


Figure 3.8: Schematic of four contigs joined by spurious double links at a, b, c and d.

components, of which about 3200 are isolated vertices, and the rest contain less than 100 STSs each. The left panel of Figure 3.9 is a special representation of the double-linkage intersection graph of the large component, designed to show the high interconnectivity among chromosomes. In this representation, all STSs that are assigned to a chromosome n are superimposed at a node labeled n, for $n = 1 \ldots 22, X, Y$. Clearly most chromosomes are connected to many other chromosomes even in double-linkage graphs. The right panel of Figure 3.9 shows the double-linkage intersection graph for STSs that are either assigned to chromosome 7 or unassigned. Thus, this panel may be considered to be the isolation and expansion of node 7 of the graph in the left panel. The graph for chromosome 7 has many components, suggesting that most spurious links can be removed by filtering the data for one chromosome at a time.

Although edges with weight $M \geq 2$ are generally reliable, many of these strong edges are misleading. The unwanted edges connect widely separated regions of the genome and thereby artificially cross link contigs. Restriction of the analysis to STSs belonging to a particular region or chromosome helps to reduce false links. The next chapter describes a method that makes the presence of these 'cross-links' easily visible in many cases, whether they link different chromosomes or distant sites on the same chromosome.

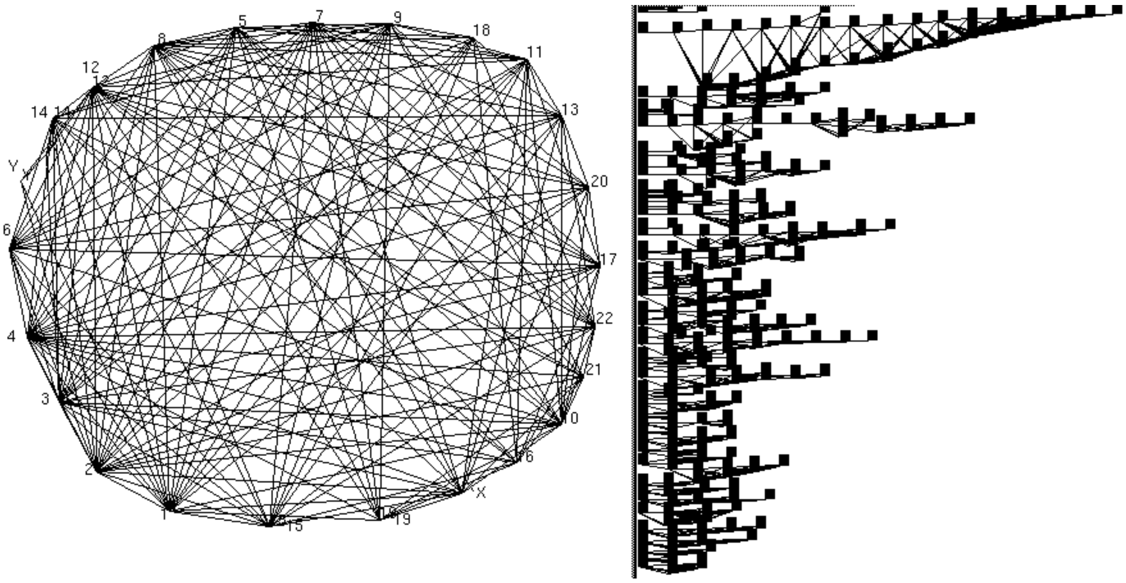


Figure 3.9: Interconnection of chromosomes in a double-linkage intersection graph (left) and isolation of data for a single chromosome (right). See text for details.

Chapter 4

Revealing hidden interval graph structure

This chapter addresses the important problem of contig identification. Most of the work in this chapter is published in a paper titled *Revealing hidden interval graph structure in STS-content data* by this author, A. Bonner and N. Goodman [45]. The ideas and some of the results are also discussed in another paper by the same authors [44]. I present several graph algorithms which create a simplified view of probe data. The view, called a *structure graph*, provides a quality check — consistent data produce a straight line, while anomalous data produce branches and loops. Verification and/or extraction of simple paths from a structure graph should be a preliminary step in data analysis. Following this step, data corresponding to contigs can be analyzed in standard ways to construct a physical map.

The following sections present a theoretical basis for the structure graph method, along with examples of its application to STS data from human genome centers. The technique has general applicability to any data that have an underlying (albeit noisy) interval model. Thus, in Chapter 6 I apply the method to simple overlap data.

4.1 Introduction

In Chapter 3, I introduced *weighted intersection graphs* to represent STS-YAC data. Each node of the graph simultaneously represents an STS and the set of presumably mutually overlapping clones which the STS hits. An edge between two nodes indicates that the two STSs hit one or more clones in common, or equivalently, that the two sets of clones have a nonempty intersection. Each edge is assigned a weight M, where M is the number of clones the two sets have in common. Intuitively, an edge is evidence that the two sets of clones are close together in the genome, and the weight of the edge is the strength of that evidence. The strength of unit weight edges is insufficient in the data examined here, since as shown in the previous chapter, most of these edges do not correlate with proximity. Therefore, as proposed by Arratia *et al.* [4], I typically remove edges of unit weight from our graphs, since that filters out many false links between widely separated regions of the genome. The resulting graph is called a *double-linkage* intersection graph.

As the amount of mapping data increases, the intersection graphs rapidly become large, unwieldy and extremely complicated. Much of the complexity arises from small-scale details. In contrast, the large-scale structure of the graph should be simple, reflecting the linearity of chromosomes. When the large-scale structure is not linear, it reveals large-scale problems with the developing genome map. To elucidate this structure, I have developed several algorithms for graph abstraction. The idea is to abstract and compress the detailed structure of a large and unwieldy graph to give a much simpler graph which reveals the overall structure of the larger graph. The abstracted graph can be automatically laid out and displayed on a screen, from which one can see large-scale anomalies in the mapping data. Most graphs in this dissertation were laid out automatically and displayed using the Hy+ data visualization system developed by Alberto Mendelzon's group at the University of Toronto [15]. Some of the smaller graphs were laid out by hand in Hy+ or drawn using Xfig [81].

4.2 Structure Graphs and Perfect Data

First, let us review some definitions from graph theory. A graph $G = (V, E)$ is given by a set of vertices V and a set of edges E. An edge is a pair $(u, v) \in E$, where $u, v \in V$. If $(u, v) \in E$ then u and v are said to be *adjacent* or *neighbouring* vertices. A *path* $P(v_1, v_l)$ from vertex v_1 to vertex v_l is a sequence of vertices $v_1, v_2, \ldots, v_l$ such that for $1 \leq j < l$, (v_j, v_{j+1}) is an edge of G. The *length* of a path is the number of edges, $l - 1$. A path P is *simple* if no vertex on the path is repeated. It is *chordless* if no edge in G connects nonconsecutive vertices of P. Finally, breadth first search (BFS) is a classical graph traversal algorithm which partitions the nodes of a graph G into *layers*. The nodes in layer L_i are a distance of i edges from a given starting node or, in the case of several starting nodes, from the nearest node in the set of starting nodes. I will write $BFS(G, u)$ to denote application of the breadth first search algorithm to graph G starting at node u, and $BFS(G, S)$ when the algorithm starts from a set of nodes S. Pseudocode for the BFS algorithm can be found in many books on graph theory, for example the one by Golumbic [34].

I define a simplification of a graph G, called a *structure graph* G'. The simplification is achieved by coalescing nodes of G into sets called *blobs*. Intuitively, a blob is a set of nodes from a small, localized region of the graph. I have experimented with different methods of defining these localized regions, and the method I find to be most insensitive to noise in the data while preserving underlying structure is based on BFS. A *blob* is defined to be a connected component within a BFS layer; *i.e.*, a set of nodes that belong to the same layer and that are connected by a path using only nodes in this layer. A structure graph is formed from an STS intersection graph G by the algorithm shown in Figure 4.1.

```
Procedure Structure−Graph(G) ;
Input:  graph G = (V,E) ;
Output: Structure graph G' = (V',E') ;
{
  Step 0:  b = 0 ;
  Step 1:  BFS(G,u), for some u in V ;
           let x denote some vertex in the last layer
  Step 2:  S = Make−Blob(x) ; V' = { blob[x] } ;
  Step 3:  BFS(G,S) ;
  Step 4:  foreach  v in V
           { if v is not in a blob { Make−Blob(v) ; V' = V' ∪ {blob[v]} } }
  Step 5:  foreach (u,v) in E where blob[u] ≠ blob[v]
           { E' = E' ∪ { (blob[u], blob[v]) } }
  Step 6: return ( G' = (V',E') ) ;

  Function Make−Blob(v)
  Input:  Vertex v in V
  Output: a subset S_b of V
  {
    b = b + 1 ;  S_b = { v} ;  blob[v] = b ;  L = layer[v] ;  T = { v } ;
    while  T is not empty  do {
         Remove a node x from T ;
         foreach neighbour y of x {
              if y is not yet in a blob and layer[y] is L
               { T = T ∪ { y } ;  S_b = S_b ∪ { y } ; blob[y] = b ; }
         }
    }
    print ( "blob " + b + " is " + S_b ) ;
  }
}
```

Figure 4.1: Structure-Graph algorithm

The algorithm consists of a main procedure, *Structure-Graph*, and a function, *Make-Blob*. The main procedure invokes the breadth-first search algorithm BFS twice — the first BFS determines an extremity of the intersection graph, and the second BFS defines new BFS layers relative to this extremity. For simplicity, the BFS function is omitted, but let us assume that the number of the layer to which a node $y \in V$ belongs according to the most recent BFS is stored globally as $layer[y]$. The *Structure-Graph* procedure calls the function *Make-Blob* once for each blob formed, and constructs the edges of the structure graph. The function Make-Blob(v) determines which vertices $w \in V$ belong to the same blob as v, based on the most recent BFS. It increments the blob counter b and outputs the blob as a set, S. It updates a global array which defines the blobs, by setting $blob[w] = b$ for each vertex w in the set S.

More explicitly, Step 0 initializes the blob counter b to zero. In Step 1, there is a BFS traversal of G starting at an arbitrary node $u \in V$. This serves to define BFS layers $L_i, i = 1 \ldots n$ relative to the starting node, and an array *layer* such that $layer[v]$ is the layer number of vertex v. The last node visited in the BFS traversal is saved as x. In Step 2 function **Make-Blob** is called to create the blob S_x containing x. The result is that S_x contains x, along with the neighbours of x that are in the last layer L_n, and the neighbours of those neighbours that are in L_n, and so on. In Step 3, there is a second BFS traversal of G starting from the set S_x. This defines a new BFS layering of G to be used when constructing the rest of the blobs in Step 4. In Step 5, $blob[v]$ is the number of the blob containing v. An edge is created between distinct blobs if an element of one blob is adjacent to an element of the other in the original graph. (The weight of each edge is also determined as the number of pairs of adjacent elements connecting the two blobs. The weights can be used in experiments aimed at removing cross links in the structure graph, however, I have not had success in this regard.)

Function Make-Blob(v) increments the blob counter b and creates a new blob S_b which is initialized to contain node v. The new blob must also contain any other node in v's

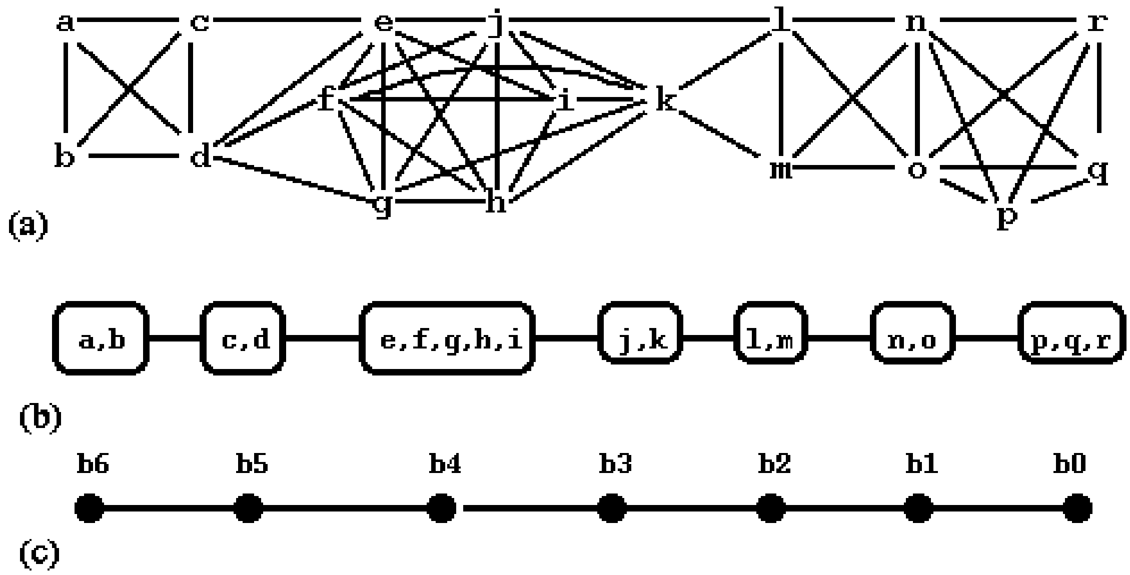


Figure 4.2: Structure graph

(a) Intersection graph for example data in Figure 3.4. (b and c) The corresponding structure graph showing (b) and hiding (c) the contents of the blobs. See text for details of construction.

BFS layer L which can be reached via a path in L. This is accomplished by first placing v on a queue. Nodes are then removed from the queue one by one. For each node x taken from the queue, one must consider its neighbours, i.e., the nodes connected to x by an edge in the graph G. If x's neighbour y is in the same layer L and y is not yet in the blob, then y is put in S_b and is placed on the queue. In this way, one finds all the nodes that can be reached from v within its layer. The blob number and the set of nodes in the blob are printed to a file for subsequent analysis.

As an example, I construct the structure graph for the hypothetical data of Figure 3.4 whose intersection graph is reproduced in Figure 4.2(a). For the purposes of this example, I ignore the weights on the edges, whereas in real experiments I construct the structure graph from a double-linkage intersection graph. Let the arbitrary starting node in Step

1 be h. The BFS layers are then $L_0 = \{h\}$, $L_1 = \{e, f, g, i, j, k\}$, $L_2 = \{c, d, l, m\}$, $L_3 = \{a, b, n, o\}$, and $L_4 = \{p, q, r\}$. The last node visited in the BFS could be any of p, q, or r; let it be $x = r$. Step 2 forms the first blob as $S_x = blob_0 = \{p, q, r\}$. The second BFS layering of Step 3, starting at the set $\{p, q, r\}$ produces $L_0 = \{p, q, r\}$, $L_1 = \{n, o\}$, $L_2 = \{l, m\}$, $L_3 = \{j, k\}$, $L_4 = \{e, f, g, h, i\}$, $L_5 = \{c, d\}$, and $L_6 = \{a, b\}$. Each of these layers is internally connected, so the blobs formed in Step 4 are the same as the layers of Step 3, i.e., $blob_i = L_i, i = 0, 1, \ldots, 6$. Step 5 forms the edges of the structure graph by connecting blobs which contain adjacent nodes. For example, $(blob_0, blob_1)$ is an edge in the structure graph since (r, o) is an edge in G, and $r \in blob_0$ and $o \in blob_1$. Figures 4.2(b) and (c) show two representations of the resulting structure graph. The upper representation (b) displays the blobs as boxes which explicitly list the STS contents. The lower representation (c) displays blobs as simple nodes, while hiding the contents. The latter, simpler representation is the one used in other figures of this dissertation.

Clearly the structure graph in this example is simpler than the intersection graph since it has fewer nodes and edges. More importantly, it captures the underlying linear structure of intersection graphs derived from STS data by compressing local complexity. The theorem below generalizes this result. The theorem refers to "perfect STS-YAC data" which is here defined to mean that the data fit a model where the YACs correspond to horizontal line segments along a horizontal axis (the real line), and the STSs correspond to vertical line segments which intersect one or more of the horizontal line segments. The intersection graph is created as described above, assuming no false positives and no false negatives, i.e., there is an edge between two nodes if and only if the corresponding vertical line segments (STSs) intersect the same horizontal line segment (YAC).

Theorem 1 *The structure graph derived from a connected component of an STS intersection graph for perfect STS-YAC data from a linear genome by the Structure-Graph Algorithm is a simple chordless path.*

Proof 1: I first sketch a proof of this theorem based on results of Corneil *et al.* [16] regarding the theory of unit interval graphs, and then we give a more detailed proof with reference to STSs and YACs. A graph is a *unit interval graph* if each vertex v_i can be identified with a unit interval (from v_{iL} to v_{iR}) along the real line such that there is an edge between two vertices if and only if the corresponding intervals overlap. Figure 4.3 gives a simple illustration of a unit interval model and the corresponding graph. Corneil

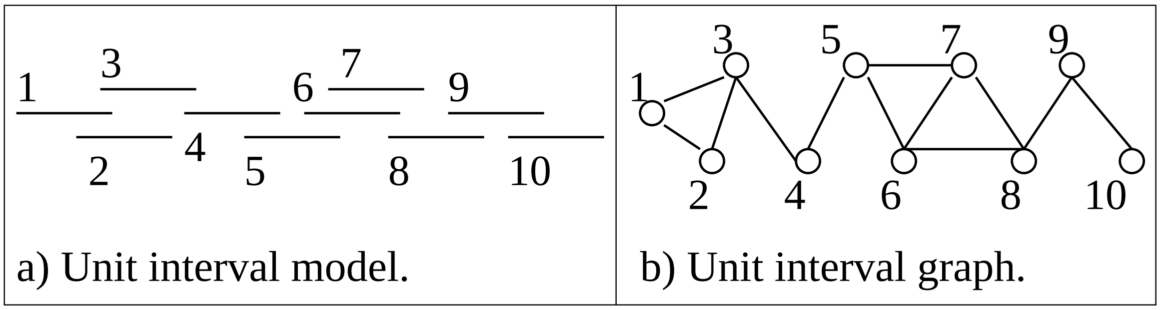


Figure 4.3: Example of a unit interval model and graph with a natural labeling.

et al. [16] define a *natural labeling* of the vertices to be one where $v_{iL} \le v_{(i+1)L}$, (i.e., the lower the vertex number, the more to the left is the corresponding interval). Corneil *et al.* also define a *left anchor* as a vertex which can serve as v_1 in some natural ordering.

A theorem of Roberts [76] cited by Corneil *et al.* [16] states that:

> A graph G is a unit interval graph iff there is an order $<$ on the vertices such that for each vertex v, the set comprising v and its neighbours in G is consecutive with respect to this order.

This theorem provides an easy way to prove that an STS intersection graph for perfect STS-YAC data is a unit interval graph. The proof of this fact will let us relate our results to those of Corneil *et al.* [16]. Note that there are two graph models related to STS intersection graphs. The interval model maps the YACs to line segments of various lengths positioned along the real line and STSs to vertical line segments intersecting the YACs. This model is shown again in the left panel of Figure 4.4 below. From this

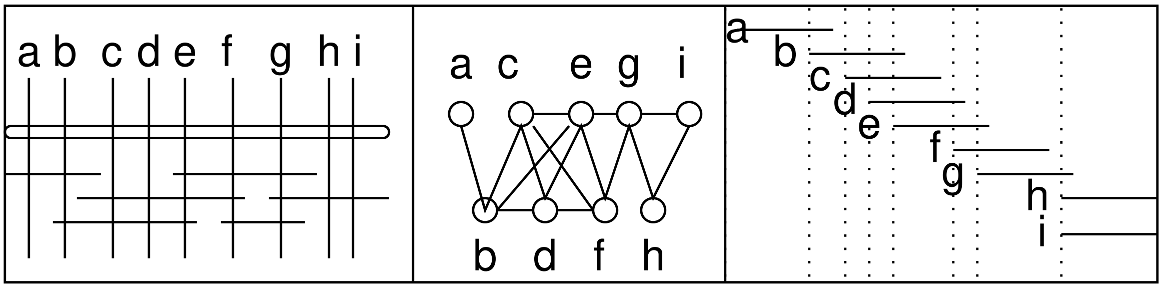


Figure 4.4: (Left): The interval model. (Center): The STS-intersection graph. (Right): The unit interval model.

model one constructs the STS intersection graph (center panel of the figure). I show below that an STS-intersection graph has a unit-interval model, as sketched in the right panel of Figure 4.4. The horizontal intervals of unit length correspond to vertices in the STS-intersection graph. The vertical dotted lines are not part of the model, but rather are drawn as a visual aid in determining which intervals overlap. Each overlap maps to an edge in the STS-intersection graph.

Proof that an STS intersection graph is a unit interval graph:

Consider an STS intersection graph G for perfect STS-YAC data from a linear genome and let $<$ be the genomic order of the STSs (see Figure 4.5 below). I claim that for each

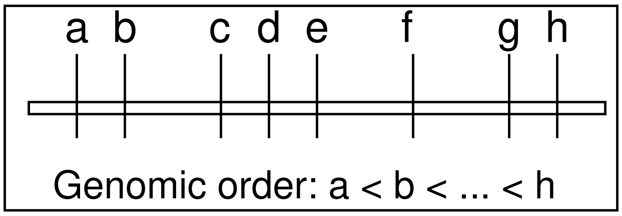


Figure 4.5: Illustration of genomic order of STSs defined by position on linear genome.

vertex v in G, the set comprising v and its neighbours in G is consecutive with respect to this order. Suppose the contrary, i.e., that the claim does not hold for a certain vertex x. Let $S = \{v_0, v_1, v_2, \ldots, v_k\}$ be the set comprising x (defined as v_0) and its neighbours in G. Since S is not consecutive in the order $<$, there must be at least one vertex $w \notin S$, for which $v_i < w < v_j$, where $v_i, v_j \in S$. Since w cannot be x, it follows that either

$x < w < v_j$ or $v_i < w < x$. In the first case, there must be a YAC connecting x and

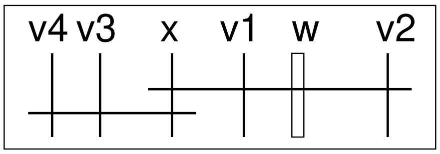


Figure 4.6: Example of the contrary hypothesis: x has neighbours v_1, v_2, v_3, v_4 but $\{x, v_1, v_2, v_3, v_4\}$ is not consecutive in the genomic order $<$.

v_j (since they are neighbours), and this YAC must also connect w to x, since the YAC is a continuous piece of the genome and w is between x and v_j. This is a contradiction, however, since that would make x and w neighbours in G, whereas we assumed they are not. The second case gives rise to the same contradiction. Therefore, the claim is true, and according to Robert's theorem the STS intersection graph for perfect data is a unit interval graph.

I use two results concerning unit interval graphs from Corneil *et al.* [16]:

> *Proposition 2.1(2):* Let I be a unit interval model, and $v_1, v_2, \ldots, v_n$ be the natural labeling for I. Let L_k be the set of vertices of level k of the BFS starting from vertex v_1. Then *(omitting here results (1) and (3))* (2) For each k, L_k is a clique.

and

> *Theorem 2.3:* If G is a unit interval graph, any vertex z returned by the following procedure *Get_left_anchor(G)* is a left anchor:

1. Choose an arbitrary $u \in V$.
2. Perform a BFS on G, starting from u.
3. Let L_t be the last level of the BFS, and z be a vertex in L_t such that $deg(z) = min deg(w) : w \in L_t$.

4. Return(z).

The correspondence between these (earlier) results of Corneil *et al.* [16] and our Structure-Graph algorithm and theorem is very close, despite independent development. The input graph G in the algorithm Structure-Graph(G) is an STS-intersection graph, which is a unit interval graph. Step 1 of the Structure-Graph Algorithm is almost the same as the procedure Get_left_anchor(G), except that an arbitrary vertex x is chosen in the last layer rather than a vertex of minimum degree. In Step 2 of the structure algorithm, one forms a set of vertices (called a blob) which includes x and other vertices connected to x and in the last BFS layer. The last BFS layer is either a subgraph which is a single connected component at one 'end' of G, or it is two connected components, one at either 'end' of G. In the former case the blob contains all of the vertices of the last layer, including the vertex of minimum degree. In the latter case, the blob contains all of the vertices that are in the last layer and at one 'end' of G, which may or may not include the vertex of minimum degree. In either case, the blob that is formed in Step 2 has the properties of a left anchor, so that the BFS performed in Step 3 creates layers which are cliques (cf. Corneil's Proposition 2.1(2)). This leads to Theorem 1, since each BFS layer forms a single blob (and moreover a clique), and the BFS layers by definition are joined in a linear sequence, creating a simple path as the structure graph. Δ

The purpose of presenting *Proof 1* above is to show how closely this work ties in with previous work on unit interval graphs. Proof 1, however, involves a bit of 'hand waving' when our Structure-Graph algorithm does not explicitly choose a vertex of minimum degree in Step 1 from which to do the BFS in a subsequent step. Instead, in Step 2 I form something more diffuse, a 'blob', from which to do the BFS in Step 3. There is an advantage to working with a rather diffuse blob as compared to a special vertex when designing an algorithm to be applied to genomic data. Genomic data tend to be noisy, and the blobs in our algorithm help to smooth and hide local noise which does not disturb the overall structure. Therefore, I now present a more detailed proof of Theorem 1, which incorporates blobs and which does not rely on results in other papers.

Proof 2: For clarity and relevance to the problem at hand, I present the arguments with reference to idealized STSs and YACs, i.e., perfect data as defined above. To translate to the interval model replace the word 'YAC' with 'horizontal line segment on the real line' and the word 'STS' with 'vertical line segment intersecting the real line'. Also, for simplicity of expression, the proof assumes that there is an edge between two vertices in the intersection graph if and only if the two corresponding STSs hit at least one YAC in common. This is actually 'single linkage', but the arguments are easily translated to 'double linkage' by replacing the word 'YAC' in the proofs below by the phrase 'two or more YACs'. Similarly, the proof above that the STS-YAC intersection graph for perfect data is a unit interval graph requires only insignificant changes to convert it to a proof that the double-linkage STS-YAC intersection graph for perfect data is a unit interval graph.

Let $G = (V, E)$ be an STS-intersection graph which is the input to the Structure-Graph algorithm. Assume that G is a single connected component whose nodes and edges correspond to noise-free STS-YAC data. (For a disconnected graph, the following arguments apply to each component separately.) For each $v_i \in V$ let v_i denote both the node and the corresponding STS; it will be clear from the context whether I mean node or STS. The assumption that G is connected is equivalent to saying that all the STSs lie in a single *contig*, i.e., the STSs are located in a group of overlapping YACs which together comprise an uninterrupted section of the genome. Let the true genomic order of STSs in the contig be $\alpha = v_1, \ldots, v_n$. Define *left* and *right* so that STS v_i is to the left of STS v_j if $i < j$, and denote this by $v_i < v_j$. We first prove the following propositions.

Proposition 1 *Consider STS/nodes $u, v, w \in V$ such that u is to the left of v, and v is to the left of w in the genomic order α. Assume breadth first search is executed starting at node u. Then the BFS distance from u to v is less than or equal to that from u to w.*

More briefly,

If $u < v < w$ and $BFS(G, u)$ then $layer[v] \leq layer[w]$.

Proof: Suppose the contrary, i.e., that $layer[v] > layer[w]$. Then there is a path in G from u to w which is strictly shorter than any path from u to v. Let $v_0, v_1, \ldots, v_k$ be the vertices on this path, where $v_0 = u$ and $v_k = w$. This path corresponds in the model to a sequence of overlapping YACs, joining v_0 to v_1, v_1 to v_2, $\ldots$, and v_{k-1} to v_k. Figure 4.7 illustrates this concept for $k = 6$. The YACs in this sequence correspond to

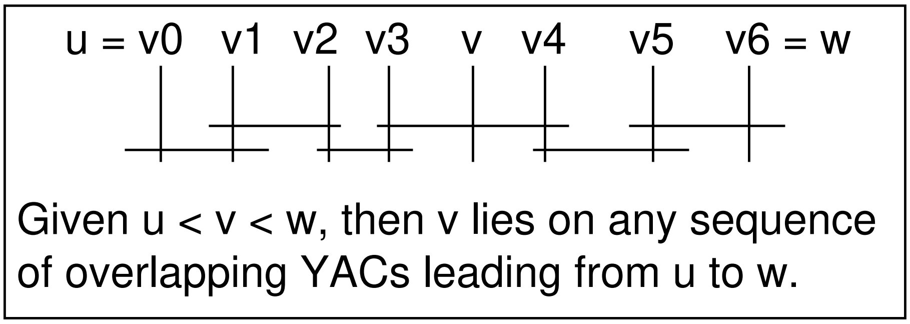


Figure 4.7: Illustration of a concept in the proof of Proposition 1.

a continuous piece of genome connecting u and w...and since the genome is assumed to be linear, this piece of genome must include STS v. Therefore, v must lie on one of the YACs of this sequence and between one of these pairs of STSs, say between v_i and v_{i+1}, where $0 \leq i < k$. But this creates a path in G, $v_0, v_1, \ldots, v_i, v$ which is shorter than the path from u to w. This is a contradiction. Δ

Proposition 2 *Assume STSs v and w ($v \neq w$) both lie to the right of u or both lie to the left of u and BFS starts from u.*

(i) If $layer[v] < layer[w]$ then v lies between u and w in the genome.

(ii) If $layer[v] = layer[w]$ then there is an edge between v and w in G. That is nodes in the same layer and on the same side of u form a clique.

Proof: (i) Suppose the contrary, i.e., that $layer[v] < layer[w]$ and STSs v and w lie to the right of u, but v does not lie between u and w. Then v lies to the right of w, i.e., $u < w < v$, as shown in the figure below. A sequence of overlapping YACs corresponding

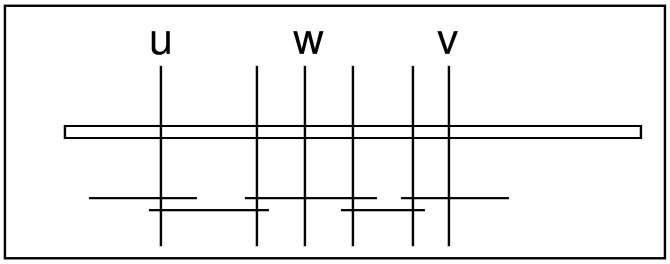


Figure 4.8: Illustration of the contradictory supposition in the proof of Proposition $2(i)$.

to a shortest path from u to v forms a continuous piece of genome stretching from u to v. Therefore, the sequence of YACs must cross the position of w, and since there are no false negatives, STS w must hit one them. Each of these YACs corresponds to an edge, so there is a path from u to w which has no more than $layer[v]$ edges. Thus, $layer[w] <= layer[v]$, contradicting the initial supposition. Δ

Proof: (ii) Suppose $layer[v] = layer[w]$ and $v < w$ and both lie to the right of u. Let r be a node in the layer $layer[v] - 1$ which shares an edge with w. Then by (i), $r < v$,

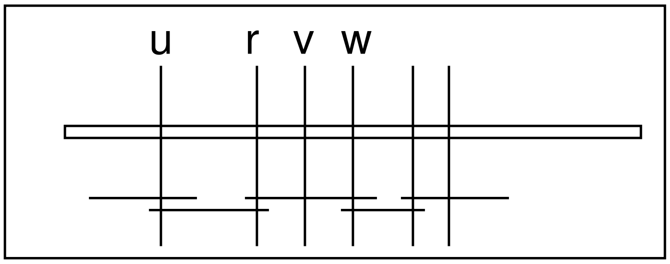


Figure 4.9: Illustration of the concept in the proof of Proposition $2(ii)$.

and since $v < w$, the YAC which extends from r to w must cross the position of v. Since there are no false positives, that YAC must be hit by STS v as well as STS w. Thus, v and w share an edge. Δ

Proposition 3 *Each layer of $BFS(G, S_x)$ is connected, i.e., any two nodes in a given layer are connected via a path that lies within that layer.*

Proof: Consider the last BFS layer, L_{last}, after performing BFS(G,u) in Step 1 of the Structure-Graph algorithm in Figure 4.1. I have defined left and right with respect to the genomic order $<$, above. Define $S_{left} \subseteq L_{last}$ as $S_{left} = \{v \in L_{last} | v < u\}$, i.e., the subset of the last layer consisting of nodes to the left of u. Define S_{right} similarly to be the nodes in the last layer that are to the right of u. For example, in Figure 4.10 below, $L_{last} = \{a, b, c, l, m, n\}$ and $S_{left} = \{a, b, c\}$, and $S_{right} = \{l, m, n\}$. As a second example,

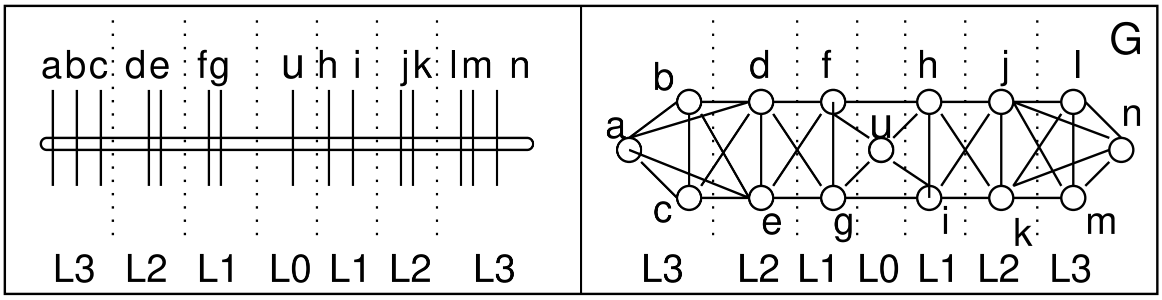


Figure 4.10: Schematic illustrating a case where the last BFS(G,u) layer contains both 'leftmost' and 'rightmost' nodes.

in Figure 4.11 below, $L_{last} = \{a, b, c\}$ and $S_{left} = \{a, b, c\}$, and $S_{right} = \{\}$.

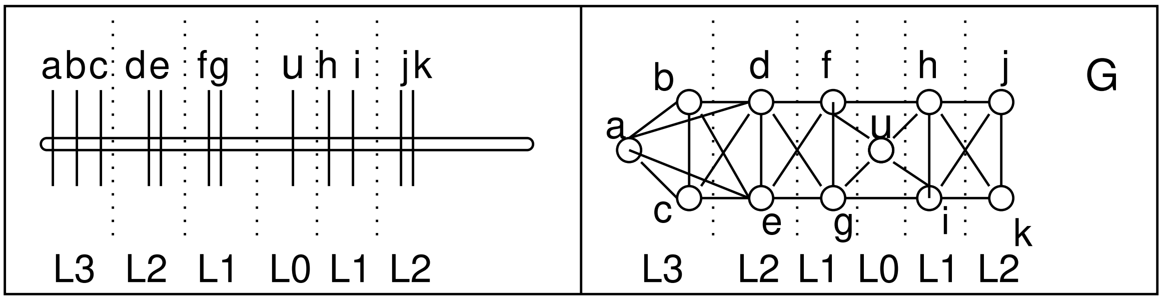


Figure 4.11: Schematic illustrating a case where the last BFS(G,u) layer contains 'leftmost' nodes.

This proof has four steps.

1. The first step is to show that S_{left} is 'leftmost', in the sense that for any $v \in S_{left}$ and $w \notin S_{left}$, $v < w$. This follows from Proposition 2, i.e., since v is in the last BFS layer and w is not, then the distance in the genome from u to v is greater that the distance from u to w, and since v is to the left of u, then $v < w$. Similarly, S_{right} is 'rightmost'.

2. The second step is to show that the blob S_x formed in Step 2 of the Structure-Graph algorithm, is such that $S_x = S_{left}$, or $S_x = S_{right}$, or $S_x = S_{left} \cup S_{right}$.

   Node x is selected as an arbitrary node of the last layer. Blob S_x is formed from nodes in S_{last} which are connected to x via paths within S_{last}. From line (ii) of Proposition 2, it follows that S_{left} is a clique and S_{right} is a clique. Furthermore, if a node in S_{left} shares an edge with a node in S_{right} then S_{left} and S_{right} are connected via nodes in the last layer, and $S_x = S_{left} \cup S_{right}$. Otherwise they are not connected, and either $S_x = S_{left}$ (when $x \in S_{left}$), or $S_x = S_{right}$ (when $x \in S_{right}$).

3. The third step is to show that Proposition 3 holds if $S_x = S_{left} \cup S_{right}$. Assume $S_x = S_{left} \cup S_{right}$. As mentioned above, in this case there is an edge from a node in S_{left} to a node in S_{right}, which means a YAC Y extends from a node in S_{left}, through u, to a node in S_{right}, as shown in Figure 4.12 below. Since all nodes of

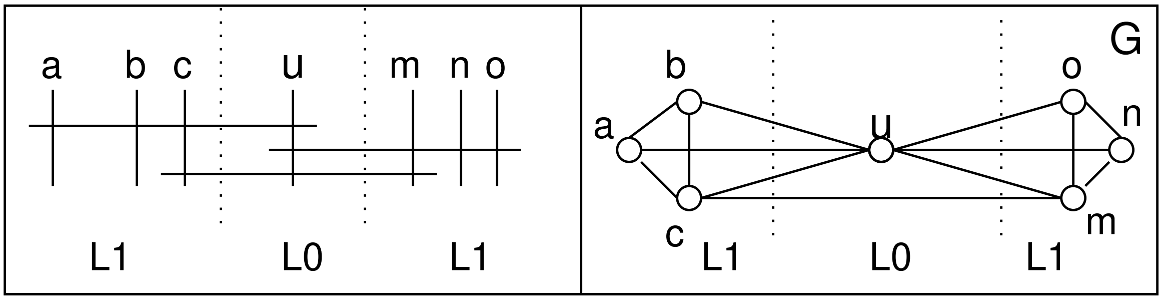


Figure 4.12: Schematic illustrating the case where blob $S_x = S_{left} \cup S_{right}$.

S_x lie in the last layer, and by virtue of YAC Y some of these nodes are clearly just one edge away from u, it follows that there are only two BFS layers: $L_0 = \{u\}$ and $L_1 = \{v \in V | v \neq u\}$. Therefore, for this special case, L_0 is a single node and L_1 is S_x. In Step 3 of the Structure-Graph algorithm, $BFS(G, S_x)$ produces just two layers: the zeroth layer is S_x, leaving only one node, u, for the next layer. Therefore each layer is connected. This completes the proof of the Proposition 3 in the special case.

4. The fourth step is to show that Proposition 3 holds if $S_x = S_{left}$. Assume $S_x = S_{left}$. I will show by induction on the number of layers in $BFS(G, S_x)$ of Step 3 that each layer is connected. Notice first that since S_x is the set of starting nodes for the breadth first search, these nodes are initially considered *visited.* Since nodes of S_x are leftmost, any unvisited neighbour is necessarily on the right, and moreover, is a neighbour of the rightmost STS of S_x, say r. Thus, $BFS(G, S_x)$ produces the same layering as $BFS(G, r)$. The following inductive proof of the proposition uses this fact.

   Basis: Layer L_0 is S_x, which by definition is connected.

   Induction: Suppose that the first k layers are each connected. Let r' be the rightmost node of layer L_k. By Proposition 1, any node v between r and r' is in a layer less than or equal to L_k, and therefore is already visited. Thus, any unvisited neighbour of a node in L_k is to the right of r' and (because there are no false negatives) is also a neighbour of r'. Therefore, STS/nodes of layer L_{k+1} lie on a YAC with r' and are not only connected by a path in this layer but form a clique in G. This completes the induction. Δ

Proof of Theorem 1: Notice that BFS defines layers such that nodes of layer L_i have neighbours only in layers L_{i-1}, L_i, and layer L_{i+1}. There is an edge between two blobs A and B if and only if there is an edge between a node in A and a node in B. Now, since a blob is a connected component within a layer, a blob cannot have a neighbour in its own

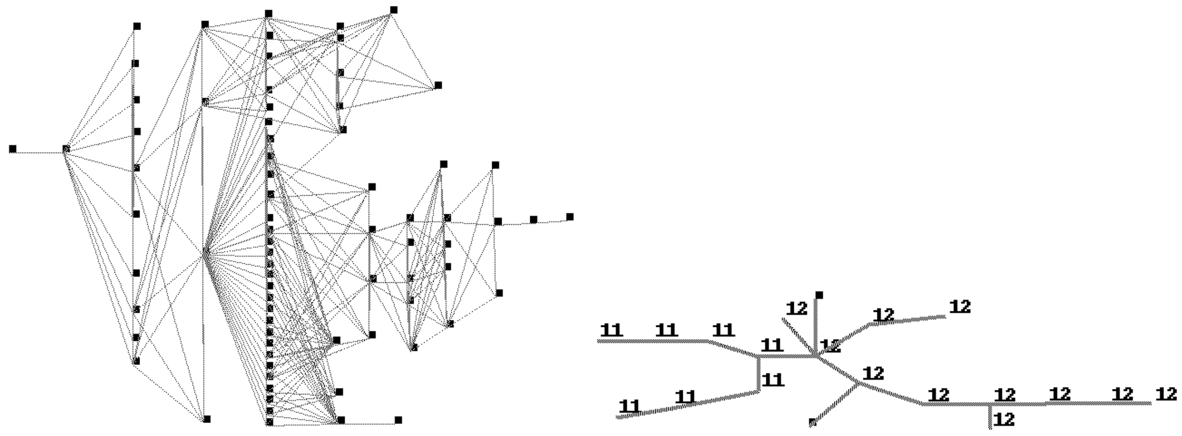


Figure 4.13: STS intersection graph and structure graph.

A typical small component from the STS intersection graph for the whole genome. On the left is the intersection graph containing 75 STS nodes and 534 undirected edges, and on the right is the corresponding structure graph containing 21 nodes (blobs) and 20 edges. The blobs are labeled according to chromosome as described in the text. This component appears to involve two contigs.

layer. Therefore, a blob in layer L_i can have neighbours only in layers L_{i-1} and L_{i+1}. By Proposition 3, each layer is connected and therefore comprises a single blob. Therefore each blob has at most *one* neighbour in the forward layer and *one* in the previous layer. Thus, S_x, which is layer L_0, connects to one blob, layer L_1, which in turn will connect to only one other blob, layer L_2, etc. By induction, the resulting structure graph is a simple chordless path. This concludes the proof of the theorem. Δ

Note that Theorem 1 would also hold for a slightly simpler Structure-Graph algorithm in which one does a single breadth first search rather than two. (The single BFS would start from a layer composed of an arbitrary node u and all of its neighbours.) Our algorithm uses two BFS applications so that blob formation will be defined relative to one of the 'ends' of the graph. This reduces the arbitrariness in blob definition, so that there is a chance that different starting points u in the first BFS may lead to exactly the same blobs.

4.3 Structure Graphs and Noisy Data

Let us now switch from considering perfect data to applying the Structure-Graph algorithm to real, noisy data. According to Theorem 1, deviations in a structure graph from a simple path similar to that of Figure 4.2 reflect deviations from ideal data. The simple path structure is robust to the extent that it is undisturbed by false negatives which do not interrupt connectivity within a BFS layer. More disruptive false negatives create small branches, typically of length 1 or 2. Non-unique STSs and chimeric YACs can produce long branches. Thus, the main advantage of representing data as a structure graph is to verify whether or not the data are reasonably 'clean' and free from distortions.

For example, Figure 4.13 shows the STS intersection graph and its corresponding structure graph for one of the small connected components from the whole genome STS intersection graph. This example is typical in that the structure graph has on the order of ten-fold fewer edges than the original graph and three or four-fold fewer nodes. This condensing of nodes and edges simplifies representation of the graph, but more importantly, the shape of the structure graph says something about the coherence of the data. When the shape is branching, as in Figure 4.13, this suggests that the connected component is not a single contig but rather two or more that are artificially joined. To confirm this inference I have labeled the blobs according to the chromosome assignment of the majority of STSs in each blob. When no chromosome has a majority of STS assignments, the node is left unlabeled. This connected component appears to be composed of a contig from chromosome 11 cross linked with a contig from chromosome 12.

The structure graph for the largest component of the intersection graph for merged

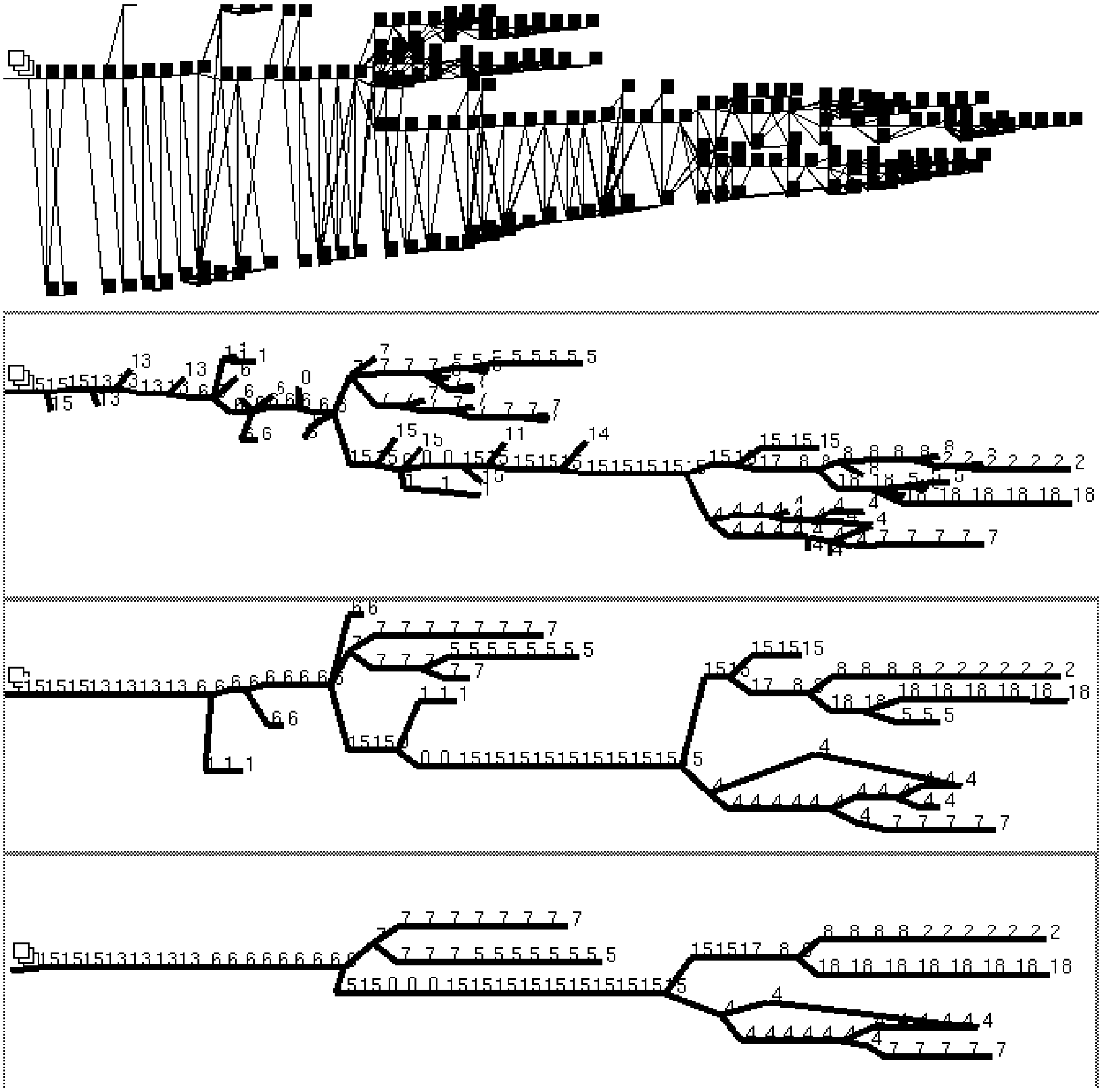


Figure 4.14: STS intersection graph and trimmed structure graphs.

(a) (Top) One component (440 nodes, 1,540 edges) of the STS double-linkage intersection graph for the whole genome (from merged STS data of Whitehead Release 8 and CEPH/genethon March, 1995). (b) The corresponding structure graph (144 nodes, 144 edges). Icons in the structure graph represent the chromosomes to which the majority of STSs within a blob are assigned. (c) The effect of trimming branches of length one (115 nodes, 115 edges), and (d) trimming branches of length up to three (96 nodes, 96 edges).

STS data is too large to be usefully displayed on one page, so I have selected for display in Figure 4.14 a fifteen-fold smaller component obtained from the previous release of STS data. The STS intersection graph is at the top, and directly under it is the corresponding structure graph. Each blob is labeled according to the chromosome assignment of the majority of its STSs or '0' if there is no majority. There are many branches and one cycle, but as in Figure 4.13, most linear segments appear to be homogeneous in chromosomal content. Clearly this connected component is composed of a number of artifactually linked contigs. A sequence of blobs for a given chromosome is usually linear, but not necessarily, as the cycle for Chromosome 4 demonstrates. Thus, even in the absence of obvious warning signs like changes in chromosome assignment, a branching or looping structure graph indicates problems with the data. In a later section, I discuss identification of contigs in structure graphs, but first I describe a method for simplifying the structure graph even further.

4.4 Trimmed Structure Graphs

The structure graph can be cleaned up by removal of very short branches and small cycles. The rationale for this beautification is that these deviations from linearity can be caused by false negatives, which are not relevant to structure. Moreover, some layout algorithms exaggerate the length of edges in short branches or cycles, making them visually distracting. An example of a short branch caused by a false negative is shown in Figure 4.15, where a single false negative disrupts the double-linkage connectivity within one of the BFS layers, causing a branch of length one. Similarly, the example at the bottom of Figure 4.15 shows how a small cycle can be caused by two false negatives.

Judging from this example, one might decrease the number of short branches by defining a blob in terms of single-linkage connections within a BFS layer, while using double-linkage to establish the BFS layers. This requires only a small modification to

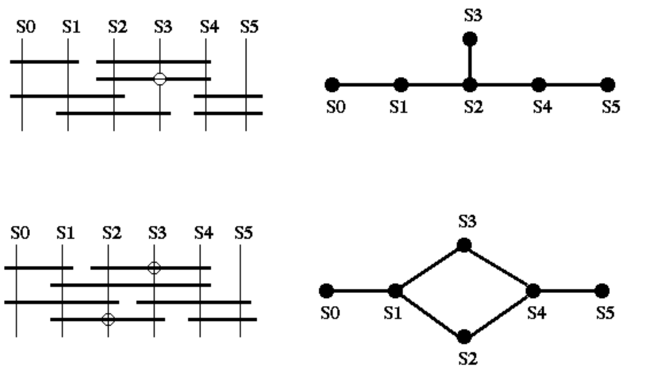


Figure 4.15: Effect of false negatives on a structure graph.

On the top left is a model of STS-YAC data which includes a false negative (the open circle) for STS S3. This gives rise to a branch of length one in the structure graph (top right). On the bottom left is a model which includes false negatives for STS probes S2 and S3. The two false negatives cause a small cycle in the structure graph (bottom right). (In these two examples the structure graphs are isomorphic with the corresponding double-linkage intersection graphs.)

the Make-Blob function so that it uses unit weight edges as well. This modification to the structure algorithm reduces the number of unit-length branches from 28 to 16 in the data used for Figure 4.14. The use of single linkage within blobs and double linkage for global structure might be a better approach than using double linkage for everything, however, for the sake of simplicity I chose the latter (strict double linkage) for the experiments shown in this thesis.

Next I present two algorithms that attempt to hide the effects of false negatives, but first two remarks: (1) not all false negatives disrupt linear structure, and (2) not all short branches and cycles are caused by false negatives. With regard to the first point, note that the proof of Theorem 1 shows that for perfect data, not only is each layer of the second BFS a connected component, but a clique as well. A blob is defined as a set of nodes that are connected within a BFS layer (rather than as a clique) so that structure will be somewhat insensitive to false negatives. False negatives which do not interrupt connectivity within a BFS layer have no effect on the resulting structure graph. With regard to the second point, the length of a branch or cycle representing part of a contig is determined by the length of the contig and the position on the contig at which it spuriously joins with another contig. Removal of short branches or cycles sometimes removes structural clues. The algorithms in this section produce simplified views of the structure graph which are intended to be viewed alongside the original structure graph, without replacing it as the primary source of structural information.

Removal of short branches

Let $degree(v)$ denote the number of neighbours of vertex v in the subgraph of G which is currently under consideration (i.e., G minus branches which have already been removed). A *branch* is defined as a simple chordless path $v_1, \ldots, v_n$ in the current graph, where $degree(v_i) = 2$, for $v_i \in \{v_2, \ldots, v_{n-1}\}$, and $degree(v_n) = 1$, and either $degree(v_1) = 1$, (in which case the branch is an isolated component), or $degree(v_1) = 2$ and v_1 is connected to the rest of the graph by edge (v_0, v_1), where $degree(v_0) > 2$. The connecting node v_0 is called the *root* of the branch. The degree of the root decreases by one if the branch is removed. Figure 4.16 gives gives an example of an isolated branch and two examples of a rooted branch. When the following algorithm removes branches of length 5, it would remove nodes v_1 to v_6 in the isolated branch of panel (a), and nodes v_1 to v_5 in the rooted branch of panel (b). When unit-length branches are removed, v_1 would be removed in panel (c).

The *trim* algorithm below takes as input a structure graph $G = (V, E)$ and a parameter L, and returns G', a subgraph of G obtained by removing all branches of lengths 1 to L. A branch is defined with respect to the subgraph of G remaining after zero or more

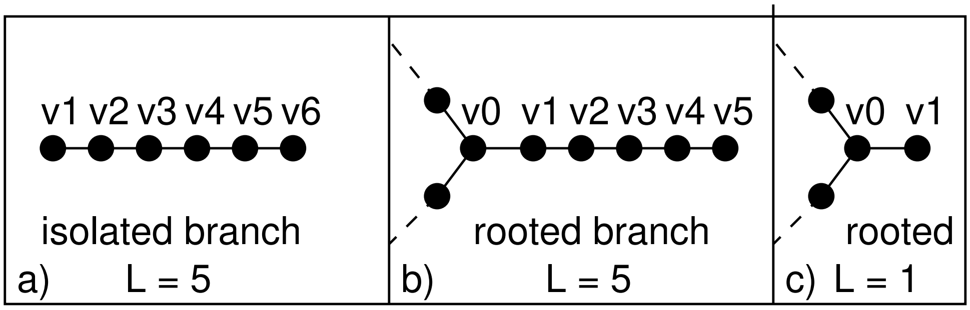


Figure 4.16: Examples of isolated and rooted branches.

branches have been removed. As branches are removed, paths which were not branches before may become branches in the remaining subgraph, and different final graphs can result depending on the order in which branches are removed. The trim algorithm therefore specifies the order of branch removal according to length of branches. It first removes branches of length 1 until a subgraph is obtained that has no branches of this length. Then branches that are one edge longer are exhaustively removed. The length to be removed is gradually increased up to the specified maximum length L. It is not hard to show that if branches of lengths 1 to k are removed in this fashion, then when branches of length $k+1$ are removed, the resulting subgraph will not have any branches of length less than $k+2$. For this reason, the degree of n_2 does not have to be checked in the line which reads "n_2 := the unvisited neighbour of n_1", since $degree(n_2) = 1$, necessarily.

Figure 4.14 shows the effect of the trim algorithm on the structure graph of one component of the STS intersection graph for human genome data. Two different values of the trim parameter are used: $L = 1$ and $L = 3$. The overall structure is more apparent in the trimmed versions. There is some loss of structure for the case $L = 3$, since ends of segments belonging to Chromosomes 6 and 7 become clipped, and segments of Chromosome 1 and 5 are removed. Trimming off branches of length one, however, simplifies the appearance of the graph without sacrificing gross structure. Henceforth, the phrase 'trimmed' structure graph refers to a structure graph minus branches of length one.

Procedure Trim(G,L)

Input: $G = (V, E)$, L
i.e., structure graph G and maximum length $L \geq 1$ of branches to be removed.

Output: Trimmed structure graph $G_L = (V_L, E_L)$, where G_L is obtained from G by first removing branches of length 1 until none exist in the resulting graph, then similarly remove branches of length 2, and so on up to length L.

begin {Trim}

$G_L = G$;
Foreach $v \in V$
1. set $degree(v)$ equal to the number of neighbours of v in G.
2. **if** $degree(v) = 1$ **then** $S_1 = S_1 \cup \{v\}$

end {foreach}

For $depth := 1$ to L
  Foreach $v \in S_1$, {i.e., each node of degree 1}
    $n_1 := v$
    $branch := \{\}$
    For $step := 1$ to $depth$ **do**
      $branch := branch \cup \{n_1\}$
      mark n_1 as visited
      $n_2 :=$ the unvisited neighbour of n_1
      $n_0 := n_1$
      $n_1 := n_2$ {advance along the branch}
    end {for step}
    if $degree(n_2) = 1$ **then**
      begin {short, isolated branch}
        $branch := branch \cup \{n_2\}$
        delete $branch$ from G_L
      end
    else if $degree(n_2) > 2$ **then**
      begin {short, attached branch}
        delete $branch$ from G_L
        remove n_0 from adjacency list of n_2
        $degree(n_2) := degree(n_2) - 1$
      end
  end {foreach v}
end {for depth}
return(G_L)

end {Trim}

Figure 4.17: An algorithm to remove branches of length L from the structure graph.

Removal of small cycles

When a linear segment of a structure graph splits in two and immediately rejoins, creating a small cycle, the most likely cause is false negatives (*cf.* Figure 4.15). A remedy to this type of noise is to aggregate blobs which are in the same BFS layer and are connected by a path through the next (higher) layer as well as by a path through the previous (lower) layer. This is easy to implement by introducing a parameter to the structure graph algorithm which indicates whether the definition of blob is to be extended in this way. This modification to the structure algorithm typically unites two blobs of a cycle of length 4. For example, nodes S2 and S3 in the cycle of Figure 4.15 are placed in a single blob, and the modified structure graph becomes linear. The algorithm for removing small cycles is provided in Appendix B.

Figure 4.18 is an example of the application of these trim algorithms to real data. From left to right the graphs in this figure are: (a) the largest connected component in the double-linkage intersection graph for STSs assigned to Chromosome 21 along with unassigned STSs; (b) the structure graph; (c) the result of trimming branches of length one; (d) the result of removing small cycles. This sequence of graphs illustrates a progressive reduction of detail leading to a graph which clearly reveals whether the structure is branching, complex, or straight. In this case the final trimmed structure graph is a simple path, consistent with the component being a single contig.

In general, small cycles have not been a problem, especially since they do not affect the Hy+ layout to the extent that small branches do. Therefore, unless otherwise specified, I do not remove cycles from structure graphs (trimmed or otherwise) shown in this thesis.

4.5 Identifying Contigs

Theorem 1 states that an STS intersection graph of a contig should transform into a linear structure graph. The above examples of branching structure graphs (Figures 4.13

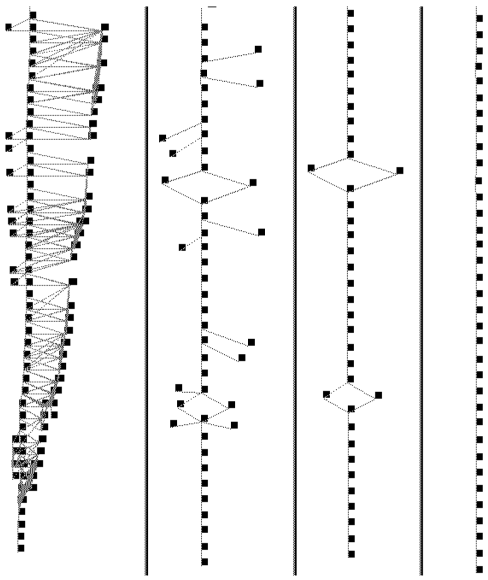


Figure 4.18: STS intersection and structure graphs for a component of Chromosome 21.

(a) STS-intersection graph for the largest component of Chromosome 21-related STS data (263 nodes, 1593 edges). (b) The corresponding structure graph (49 nodes, 50 edges). (c) Structure graph from (b) after trimming branches of length one (38 nodes, 39 edges). (d) Structure graph from (c) after removal of small cycles (36 nodes, 35 edges).

and 4.14) are clearly not single contigs, since each branch has STSs from a different chromosome. The example of a linear structure graph, Figure 4.18, could be a single contig since its STSs are from one chromosome. However, it is obvious that linearity of a structure graph only suggests but does not imply conclusively that the data represent a single contig, since two distinct contigs could be artificially joined end to end. To confirm that a linear structure graph or a linear portion of a nonlinear structure graph corresponds to a single contig, we use positional data as well as chromosome assignment data.

Genetic mapping results for 5,264 STSs are available from CEPH/Généthon ([25], internet `ftp://ftp.genethon.fr/pub/Gmap/Nature-1995/data/`). Radiation hybrid

(RH) mapping results for about 12,500 STSs are available from WI/MIT ([47], ftp `genome.wi.mit.edu`, internet `http://www-genome.wi.mit.edu`). Of the 15,136 STSs in the merged dataset of Table 3.1, 14,375 have chromosome assignments, 4,058 have genetic position assignments, and 4,527 have RH position assignments. (Subsequent to these experiments more extensive genetic and radiation hybrid maps have been published [70].) In Figure 4.19, I plot RH position versus genetic position for STSs assigned to chromosome 5 and 11. From this figure, one can see that the correspondence between radiation hybrid and genetic position is approximately linear with a slope that depends on the chromosome. For chromosome 5 the slope is approximately 3 centiRays per centiMorgan, and for chromosome 21 the slope is approximately 5 centiRays per centiMorgan. Details of the least squares linear regression are shown in the figure.

Blobs in a structure graph correspond roughly to BFS layers, and therefore they have a separation of one edge, which corresponds to the length of one clone. The average length of a clone in this case is about 1 Mb, which in genetic maps is about 1 cM. Thus, on average, an edge in the structure graph should cover about 1 cM or 3 to 5 cR, for chromosomes 5 and 11. As an arbitrary rule of thumb: if an edge appears to cover 10 times this distance, then I will deduce that it artificially joins two contigs.

The next rule of thumb is based on common sense. If one observes two contigs, A and B, where the STSs of contig A cover a range (A_{low} to A_{high}) of genetic or radiation hybrid positions which spans the range covered by contig B, (i.e., $A_{low} < B_{low}$ and $B_{high} < A_{high}$), then one can deduce that A is not a contig.

To summarize, a connected component (or portion thereof) from a double-linkage intersection graph should meet the following criteria to be put forth as a potential contig:

C1: the STS assignments are to a single chromosome;

C2: the structure is a simple path;

C3: along the simple structure path there is a relatively smooth and monotonic change

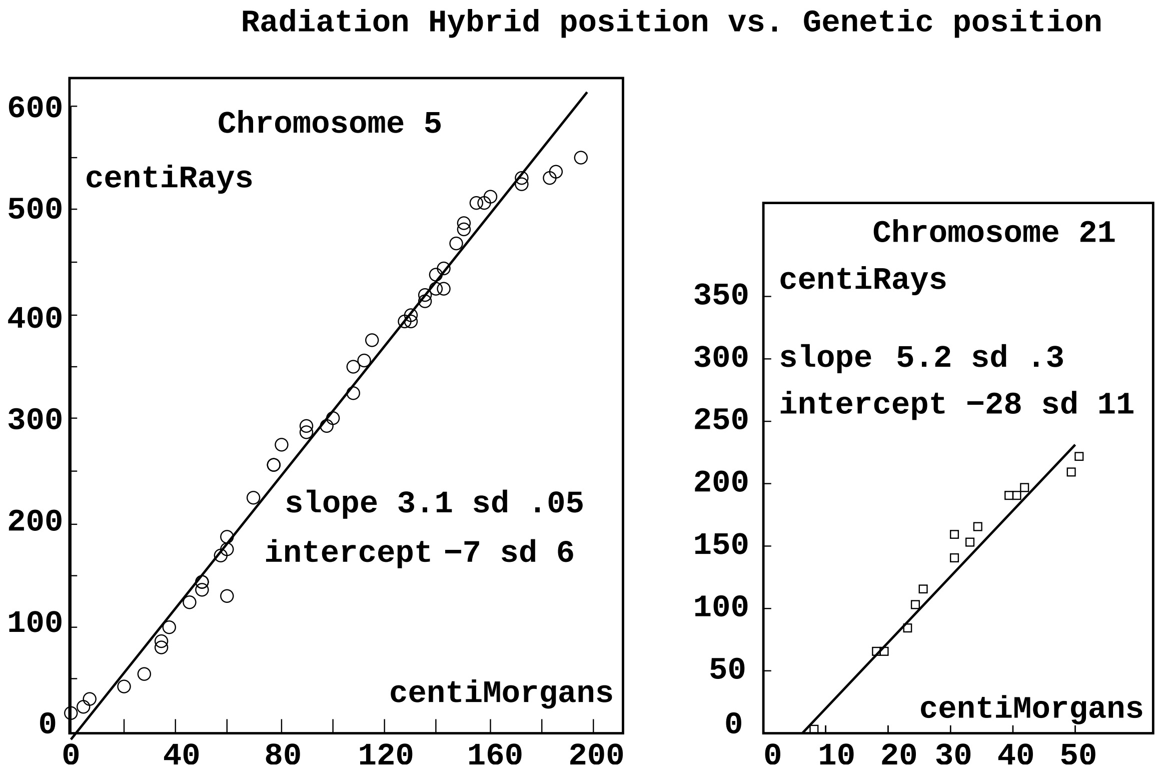


Figure 4.19: Linear correlation of genetic position vs. radiation hybrid position.

CEPH genetic position (centiMorgans) is plotted against WI/MIT radiation hybrid position (centiRays) for STSs in our dataset which have been mapped both ways: (a) Chromosome 5; (b) Chromosome 21.

of positions with a rate of approximately 4 cR/edge or 1 cM/edge and where no edge spans a change of 40 cR or 10 cM.

C4: there is no other connected component with positions lying inside the range of the positions covered by this component.

Criteria C1 and C4 are part of the mapping strategies of WI/MIT and CEPH/Généthon. Our additions to the list are C2 and C3. These four criteria do not ensure that a connected component corresponds to a contig, of course, but they make it a very good candidate. They also suggest a procedure for contig extraction:

Step 1. Isolate data for a single chromosome (according to C1);

Step 2. Form the intersection graph and the structure graph;

Step 3. Find simple paths (according to C2) in the structure graph and check that they meet C3;

Step 4. Examine the candidates from Step 3 to see which paths meet C4, and appropriately subdivide the ones which appear to require insertions of spanned contigs.

Isolating data for a single chromosome, say *Chr c*, in Step 1 can be done in several ways: method (1): use only data for STSs assigned to *Chr c*; method (2): use STSs assigned to *Chr c* but include also unassigned STSs; or method (3): start from the STS intersection graph for the whole genome and isolate a subgraph defined on edges, such that at least one endpoint of an edge is unassigned or assigned to *Chr c*. Method (1) is the most restrictive, since it only considers STSs known to belong to *Chr c*. Method (2) includes unassigned STSs in case some belong to the chromosome of interest. Method (3) includes any STS which is doubly linked to an STS belonging to *Chr c* or to an unassigned STS. This method is the least likely to make a mistake of omission, since it tolerates incorrect assignments of STSs which are adjacent to a correctly assigned STS. Method (3) also includes the first nodes of branches to other chromosomes. This may be considered an advantage or disadvantage, depending on whether one wants to include in the map of a contig STSs which hit repeat regions in the contig. Results shown in Figure 4.18 and subsequent Figures are based on Method (2) (with a modification mentioned below). I also tried Method (3) and obtained similar graphs.

In this paragraph, I explain a modification to Method (2) which reduces the number of unassigned STSs which are included with the STSs assigned to the chromosome of interest. I exclude unassigned STSs which appear to belong to a chromosome other than the one of interest, judging from chromosome associations of YACs which are hit by the unassigned STS. Recall that a YAC is ‘associated’ with a chromosome if it is hit by an

STS assigned to that chromosome. I use the following rule for guessing chromosome assignment of STSs which hit at least two YACs:

> Assign an otherwise unassigned STS (which hits two or more YACs) to Chromosome A if the number of its YACs associated with Chromosome A is at least two and greater than the number associated with any other chromosome.

The rule is based on the supposition that the YACs hit by an STS belonging to Chr z each contain a portion of Chr z, whereas the other chromosomes represented in these YACs (if chimeric) are random and vary independently. I tested this rule by applying it to STSs for which the chromosome assignment is known. Given an assigned STS x which hits a set of YACs Y, I determine the chromosome associations for each YAC in Y ignoring the contribution from STS x. I then apply the rule, and if a rule-based assignment can be made, I compare the guessed chromosome with the given chromosome assignment. It turns out that 97% of the guesses are correct using this rule, and a guess can be made for 90% of the 13,342 assigned STSs which hit at least 2 YACs. Thus, in Step 1, I used Method (2) with the modification that prior to the selection of STSs, a little over half of the unassigned STSs were assigned a chromosome using the above rule.

I applied these criteria for contig identification to data from two chromosomes. The results for chromosome 21 are shown in Figure 4.20. The largest component of this data appears in Figure 4.18. The trimmed structure graphs are shown in Figure 4.20(a). (Two small components composed entirely of unassigned STSs are not shown, since there is no evidence suggesting that they belong to this chromosome. Also, I ignore components composed of a single STS.) Each component is linear, meeting criterion C2. To check criterion C3, I have written beside each blob the average position of STSs in the blob, or the range when the range is at least 2% of the highest value. Some blobs do not contain a mapped STS, in which case no position coordinate is shown. A section of the largest component which appears to have relatively smoothly changing genetic and/or RH positions is encircled by a box. (The encircled region includes an implausible jump

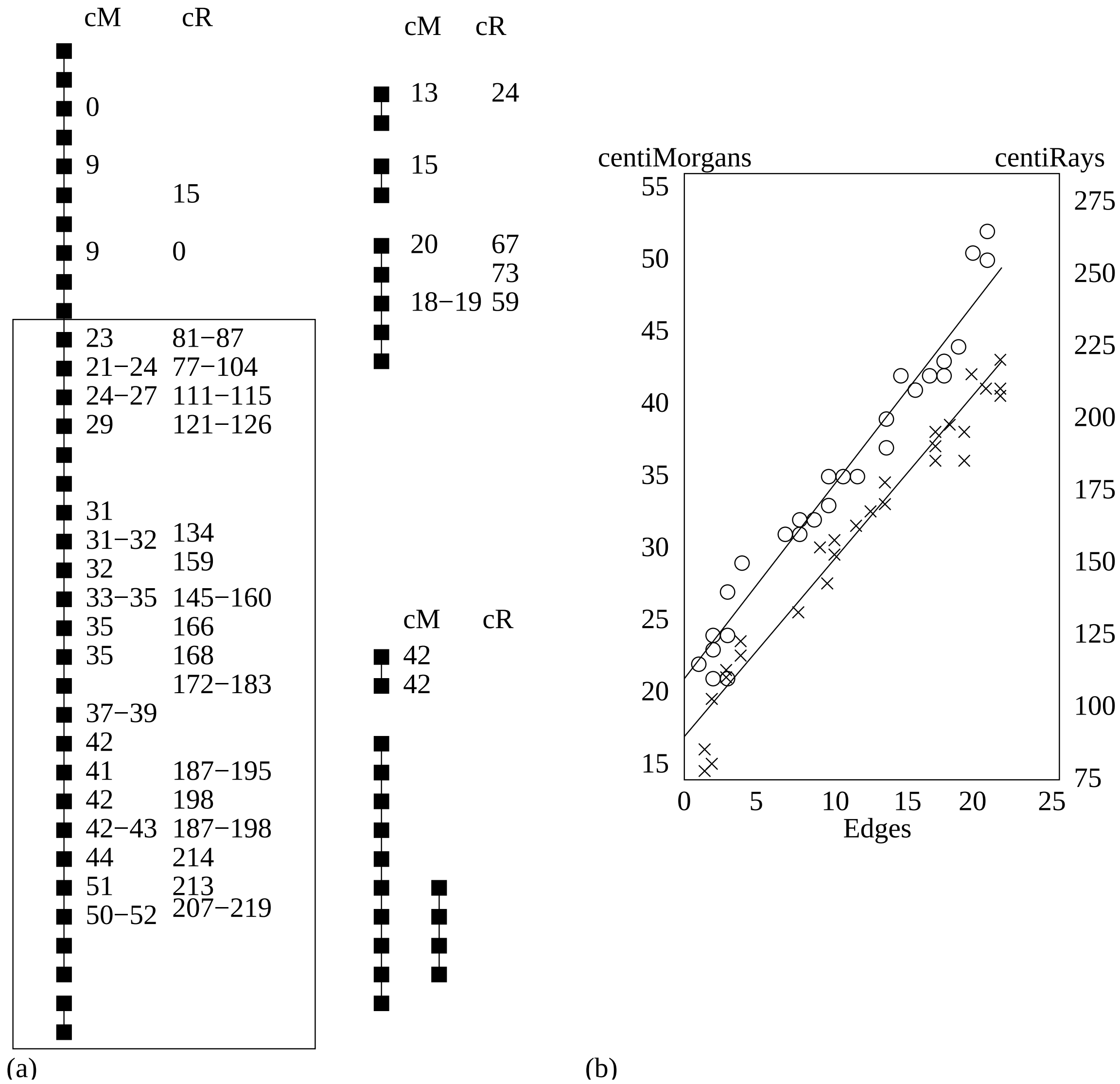


Figure 4.20: Positions along double-linkage components of Chromosome 21

(a) The trimmed structure graph of all the double-linkage connected components containing more than one STS for Chromosome 21 and unassigned data. The largest component in this figure is the same as the one in Figure 4.18. The genetic positions in centiMorgans (cM) and the radiation hybrid positions in centiRays (cR) are shown beside the blobs. The value shown is either the average for STSs within the blob, or the range, in the case that the spread of the range is at least 2% of the highest value in the blob. (b) Positions of blobs in the boxed region of the large component are plotted against edge number (cM (○), cR (×)). The genetic positions fit a straight line with slope 1.30 cM/edge (sd .05), and the radiation hybrid positions fit a straight line with slope 6.2 cR/edge (sd .2).

in genetic positions from 44 cM to 51 cM, but on the other hand the RH positions of these two blobs differ by just 1 cR.) In Figure 4.20(b) the positions are plotted against edge number. Genetic positions and RH positions are each linearly correlated with edge number in the encircled region. The least squares fit line for genetic positions has slope 1.30 cM/edge (SD 0.05), and the line for RH positions has slope 6.2 cR/edge (SD 0.2). Thus, the outlined region of the large component satisfies C3.

Beyond the outlined region of the large component, i.e., below 22 cM or 81 cR, there are substantial jumps and nonmonotonic changes in the genetic and RH positions. Moreover, with regard to C4, there are three small components with positions in the 13 cM to 20 cM range which must prefix the encircled region and a small component at 42 cM which must go in the middle. These additions to the encircled region make it approximate the WI/MIT single-linkage contig WC21.1, which WI/MIT constructs by breaking up the doubly-linked components to conform to genetic and RH positions. Two of the components do not contain any mapped STSs and therefore cannot be checked with regard to C3 or C4.

Analysis of Chromosome 5 is more complicated for two reasons: there are 58 connected components in the double-linkage graph (as opposed to seven for Chromosome 21), and some of the connected components for Chromosome 5 have complex structures. As an example, consider the largest of these components, the intersection graph of which is shown in Figure 4.21(a). The trimmed structure graph for this component is shown in Figure 4.21(b), where each blob is represented as a letter or a letter and a digit, for easy reference. The average genetic position or range of genetic positions is shown beside the label. RH positions corroborate this picture but are omitted for clarity. In contrast to Figure 4.18, this structure graph is branched and looped, strongly suggesting the presence of several contigs.

If one matches branches which have similar positions, three paths stand out as potential contigs. These paths are illustrated in Figure 4.21(c): P1 = A-B-C-J2-J1-D-E-F-

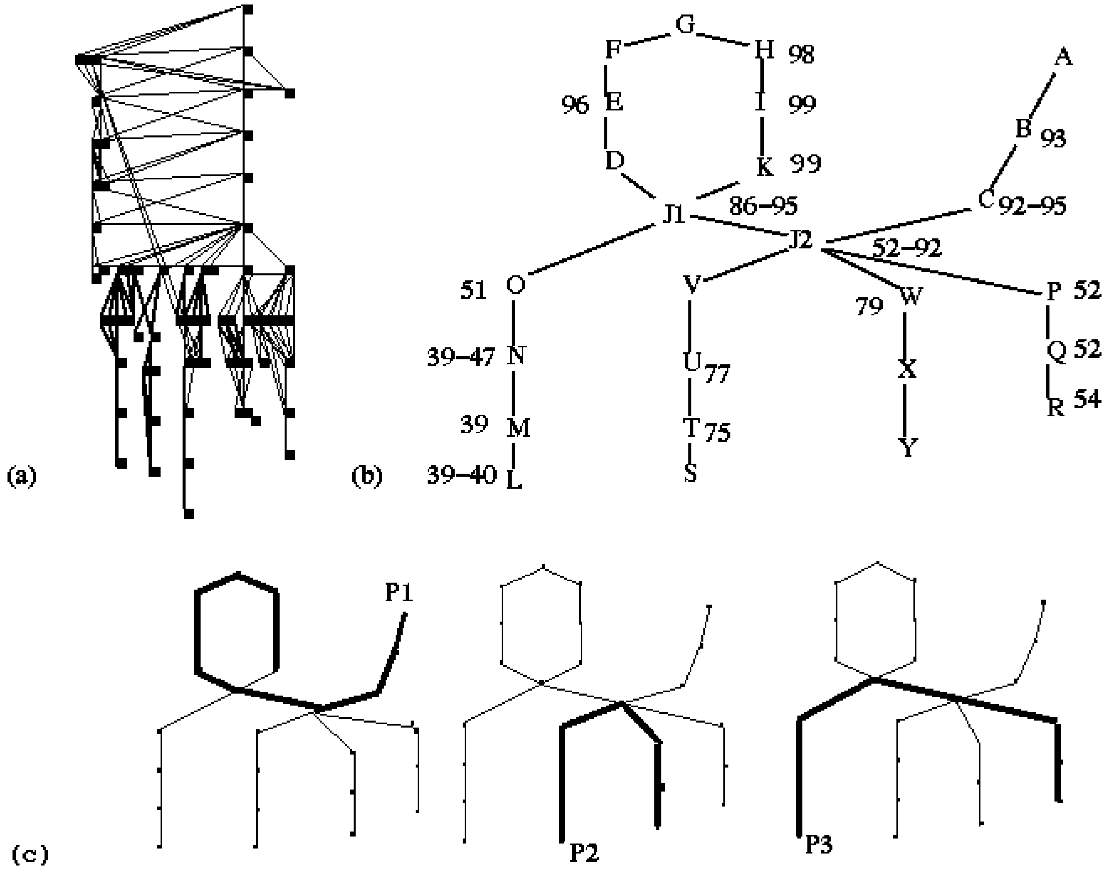


Figure 4.21: STS intersection graph and structure graphs of Human Chromosome 5.

(a) The largest component of the STS-intersection graph for STSs assigned to Chromosome 5 or unassigned (134 nodes, 1,046 edges). (b) The corresponding structure graph (26 nodes, 26 edges). Genetic positions in centiMorgans are shown. (c) Paths suggesting possible contigs.

G-H-I-K; P2 = S-T-U-V-J2-W-X-Y; and P3 = L-M-N-O-J1-J2-P-Q-R. It is of interest to examine in detail the area where the paths cross. Blobs J1 and J2 are at the juncture of several paths. The broad range of positions in these blobs (J2: 52-92 cM and J1: 86-95 cM) may be attributed to the fact that their STSs belong to more than one contig. Figure 4.22 is a double-linkage STS intersection graph of the junction area. Each STS is represented by the letter denoting its parent blob in Figure 4.21. Genetic positions in centiMorgans are shown. It can be seen that many of the blobs are almost cliques, as is expected of perfect data. The exceptions are blobs J1, J2, and K.

Now consider the proposed paths with regard to criterion C3. Since junction blobs J1 and J2 have wide ranges of positions, we look for routes (in the STS-intersection graph) through these blobs using only STSs which are consistent in position with P1, P2 or P3 as a whole. It can be seen in Figure 4.22 that the portion of path P1 from C (at 92 cM) to D (which leads to 96 cM) requires only the STS at 92 cM in J2 and the one at 95 cM in J1. Using these positions for the junction blobs and plotting STS position against edge number in the path, one obtains a linear correlation with least-squares-fit slope of 0.63 cM/edge (SD 0.07) and 1.5 cR/edge (SD 0.2) (graph not shown). Thus, P1 roughly satisfies C3, although the rates of change are about half of what is expected. Similarly, structure path P2 from W (at 79 cM) to V (which leads to 77 cM) requires only the two STSs at 77 cM in J2. Thus, P2 has a route through the junction visiting only nodes with positions congruent to the rest of the path. The genetic positions (and RH positions, not shown) along P2 are few but appear to satisfy C3. It is interesting to note that for both P1 and P2, the corresponding *shortest paths* in the STS-intersection graph follow STSs with a reasonable sequence of positions.

Path P3, however, from O (at 51 cM) to P (at 51-52 cM) has a cut point at an incompatible STS in J1, *viz.* D5S672 at 86 cM. Thus, P3 does not satisfy C3, since there can be no monotonic reasonable rate of change of position in this region. On the other hand, the problematic STS, D5S672, probably lies in a repeat region, since it is doubly

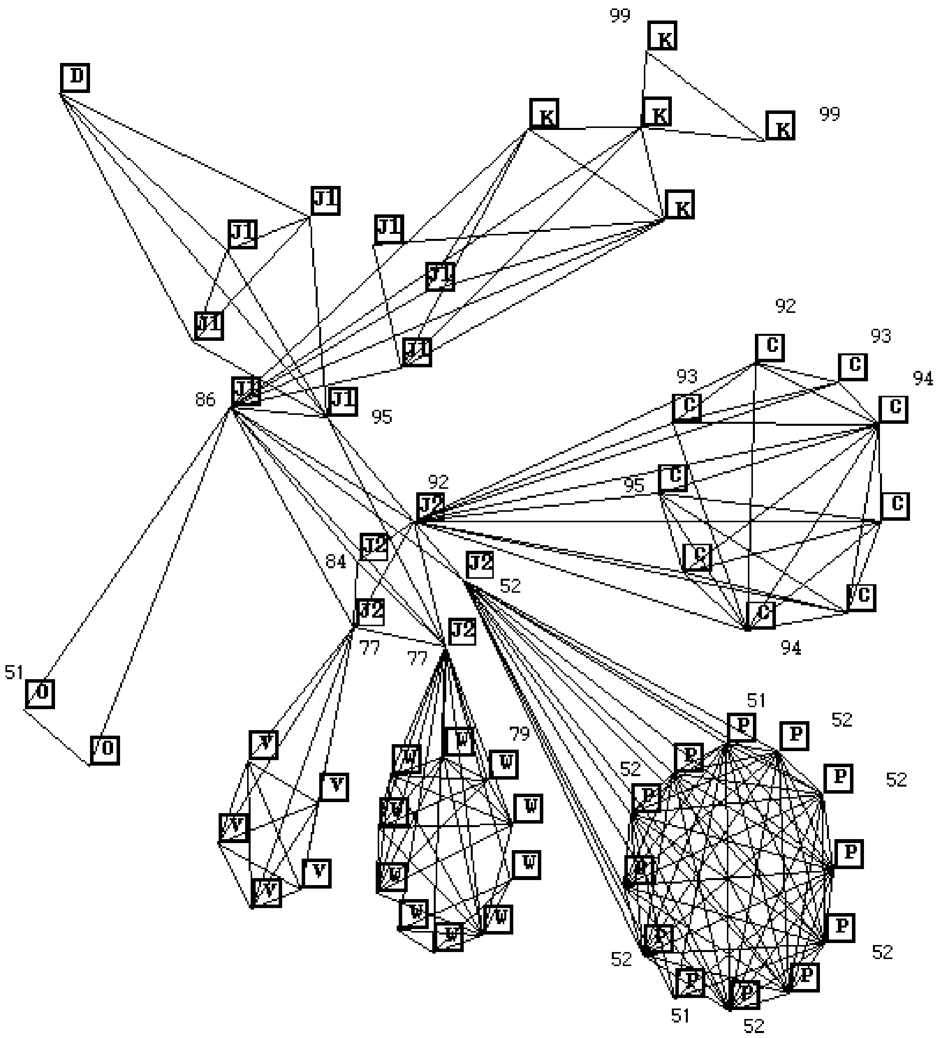


Figure 4.22: STS connections at the junction.

Double-linkage intersection graph for STSs in and around the two junction blobs J1 and J2 of Figure 4.21 (66 nodes, 400 edges). Genetic positions in cM are shown. The names and positions of STSs in junction J1 are as follows, clockwise from the top: D5S1459 (287 cR), MC, D5S1703 (304 cR), D5S1452, D5S428 (95 cM), D5S672 (86 cM), D5S1418, and WI-10009. The names and positions of STSs in J2 are as follows (clockwise from the top): D5S641 (92 cM, 291 cR), D5S631 (52 cM), D5S637 (77 cM, 254 cR), D5S629 (77 cM), and D5S646 (84 cM).

linked to STSs at 51, 52, 77, 92 and 95 cM, in which case it has been incorrectly assigned a single position of 86 cM. It seems reasonable to ignore position assignments of STSs which are not unique, since in fact they have no single position. Lack of uniqueness need not exclude an STS from use in contig construction if one defines the YACs of the contig to be those participating in the shortest paths from one end of the contig to the other. However, even ignoring the genetic position of D5S672, path P3 does not meet criterion C3, since the region M(39 cM) - N(39-47 cM) - O(51 cM) covers 12 cM in just two edges. Thus, path P3 is probably not a contig.

Finally, let us check criterion C4 by examining the ranges of positions covered by the set of simple paths satisfying C3. Path P1 covers the range (93-99 cM, 285-304 cR). The components nearest this range of positions are each composed of two STSs. One of the components has markers at (100 cM, 307 cR), and thus lies on the high side of P1. The other contains a marker at 289 cR, and thus may lie in the middle of P1. RH positions along this path (shown in Figure 4.23) are somewhat nonmonotonic, so in the absence of further evidence, I will assume that these two STSs lie below P1. (The WI/MIT group on the other hand chooses to place these two STSs (contig WC-1481) between what I here call blobs C and J1 of path P1). Path P2 has markers in the range (75-79 cM, 249-257 cR). Its nearest neighbours are a component of six STSs at (74-75 cM) and a component of two STSs at (253 cR). Thus, path P2 may also fail to satisfy C4. Path P3, which failed criterion C3, also clearly fails C4, since its range (39-54 cM, 101-143 cR) spans that of two other components, one of seven STSs at (41 cM, 111-120 cR) and one of five STSs at (44-47 cM, 128 cR).

Path P1 is shown in Figure 4.23 as a double-linkage STS graph and also as a structure graph. I have omitted the not unique STS D5S672 from blob J1, since doing so makes a natural break in the cycle, removing the edge linking K and J1. Three of the J1 STSs are then attached only to blob K, and the remaining four J1 STSs are linked to blob D. This division of blob J1 is corroborated by the RH positions (cf. Figure 4.23)

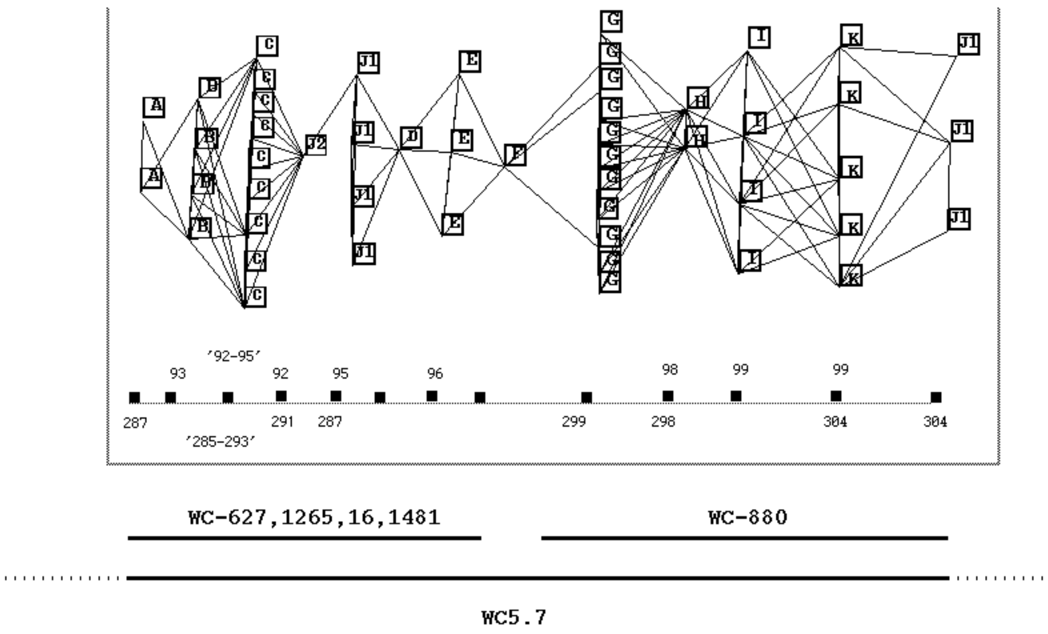


Figure 4.23: Path P1 isolated as a contig.

(Top): Contig/path P1 of Figure 4.21 is shown as a double-linkage STS-intersection graph. Icons for the STSs are those of the parent blobs. The layout is done by hand to emphasize blob groupings. Some STSs have been removed from the junction blobs J1 and J2 as described in the text. (Middle): The structure graph for this path. Average or range of genetic and radiation hybrid positions for mapped STSs are shown above (cM) and below (cR) the blobs. (Bottom): Sketch of approximate correspondence of this contig with WI/MIT double-linkage contigs WC-627, WC-1265, WC-16, WC-1481, WC-880 and with WI/MIT single-linkage contig WC5.7. See text for discussion.

since the J1 STS D5S1703 (304 cR) attaches to K, whereas the J1 STS D5S1459 (287 cR) attaches to D, making a reasonable sequence of positions along P1. Only one STS, D5S641, from blob J2 is included in the contig, since the other STSs are not necessary for connectivity and have incongruent positions. D5S641 is mapped to 92 cM, but it too appears to be nonunique, since it is doubly linked to STSs at 52, 77, 77, 84, 86 and 95 cM. For a rough comparison with contigs identified by WI/MIT, (cf. `http://www-genome.wi.mit.edu/cgi-bin/contig/phys_map`), I have included in Figure 4.23 a sketch of the extent of their double- and single-linkage contigs in this region. Path P1 involves STSs from the following WI/MIT double-linkage contigs: WC-1265, WC-627, WC-16, and WC-880. As mentioned above, WI/MIT interrupts this region with the two STSs of contig WC-1481. Further, they combine portions of their 'double-linkage contigs' and a few more on either side through single-linkage to form WC5.7. The end result is approximately the same in this case, but I think the use of structure graphs would have helped WI/MIT arrive at their solution more quickly and with fewer errors along the way. For example, they would not have called many of their components contigs, had they seen their branching structure. Also, they might have addressed more quickly the problem of breaking up the branching components, which they eventually did with the help of radiation hybrid positions and genetic positions.

These examples illustrate that structure graphs: (*i*) provide a visual aid in identifying errors and anomalies in physical mapping data, (*ii*) provide a means to distinguish between *connected component* and *contig* even in the absence of positional data, and (*iii*) help in extracting contigs from the much larger and more complex STS-intersection graphs. For example, at the above web site, WI/MIT provides the sequence of the STSs in WC5.7 and the names of the contigs to which the STSs belong. The STS sequence corresponds to the following sequence of double-linkage contigs: WC-1625, WC-196, WC-1265, WC-627, WC-16, WC-627, WC-16, WC-627, WC-16, WC-627, WC-16, WC-627, WC-1481, WC-627, WC-880, WC-1581, WC-1060, WC-589, WC-372, WC-1332. This sequence shows that several of the early WI/MIT 'contigs' had to be later broken into pieces and rearranged to fit the genetic and radiation hybrid positioning data. Thus, many of the 'contigs' were not true contigs. Structure graphs can be used to help avoid such misnomers and misidentifications. Thus, structure graphs simplify the construction of physical maps, and can improve their completeness and accuracy. As contigs are identified, the corresponding data can be passed to an algorithm for ordering the STSs and clones in the map.

4.6 Conclusion

This chapter presents a new way to look at physical mapping data. The view is immediately informative — a difference in shape as obvious as the difference between a straight line and a branching line tells the observer whether or not the data are close to ideal. The view is easy to generate — it consists of two breadth-first search (BFS) traversals of an intersection graph, along with grouping of nodes which are connected within a BFS layer. I prove that ideal data from a linear chromosome are transformed in this way into a 'structure' graph which is linear. I show how to remove the effects of false negatives, so that near-ideal data produce trimmed structure graphs which are linear.

I examined STS-YAC data from WI/MIT and CEPH/Généthon. Although these data are far from ideal, their complex structure graph can sometimes be teased apart, first by chromosome and then by region to produce short linear paths potentially corresponding to contigs. To guide the extraction process, I propose four criteria characterizing a contig. The criteria translate to a simple procedure for identifying and isolating contigs in STS-YAC data. The procedure is used to isolate a double-linkage contig from Chromosome 21 and one from Chromosome 5.

The structure graph is a simplified representation of physical mapping data designed to facilitate the discovery of problematic areas. The effectiveness of the method for contig extraction is greatly enhanced by positional data from genetic mapping and radiation hybrid mapping experiments. The method is applicable not only to STS-YAC data, but also to any form of physical mapping data which can be translated into intersection graphs. In Chapter 6, I discuss such a translation for ALU-PCR and fingerprint data.

Chapter 5

Listing Cliques

This chapter describes an improved algorithm for listing the maximal cliques (i.e., maximal complete subgraphs) of large sparse graphs. My interest in this problem arises from a suggestion by Dr. Nathan Goodman that genomic overlap data could be analyzed in terms of maximal cliques. The nodes of the *overlap graph* in question would represent clones and the edges would indicate which pairs of clones overlap. In the next chapter, this suggestion is developed into a uniform method for analyzing probe-based and overlap-based data using the concept of virtual probe. A virtual probe is defined there to be a region of the genome shared by a maximal set of mutually overlapping clones. Finding virtual probes is equivalent to finding the maximal cliques in the overlap graph.

A brute-force algorithm for listing maximal cliques is impractical for all but very small graphs, since in the simplest case it involves exploring all 2^n subsets of an n-node graph. The simple heuristic of sorting the vertices in order of increasing degree and then finding the cliques for vertices of smaller degree first provides a big improvement in the brute-force algorithm for genomic overlap graphs (results not shown). This approach applied to the overlap graph obtained from the merged data of the whole genome can find almost all of the maximal cliques within a day, but still takes seemingly forever on the vertices of large degree (the largest degree was 120). Since I wanted an exact and complete solution in reasonable time, I implemented the well-known branch-and-bound algorithm of Bron and Kerbosch [10]. This algorithm (with a modified data structure) was able to list the cliques for the merged data in a few hours. Although this time frame was practical, it became apparent that the time could be decreased by more than tenfold with a small modification which I will discuss in a subsequent section. This chapter presents experiments with the modified Bron-Kerbosch algorithm, comparing it with the original and several other published clique-listing algorithms.

5.1 Clique- and MIS-listing algorithms

The analysis of algorithms dealing with cliques is classic in graph theory. For example, the problem of finding the largest clique in a graph is one of the most familiar NP-hard problems (see for example, Garey and Johnson [31]). Listing all of the maximal cliques should be at least as hard as finding the largest clique. In fact, Moon and Moser [64] describe a graph which contains an exponential number of maximal cliques. Their construction is the complement of a graph consisting of k separate components which are 3-cliques, K_3. An example of graph with four such components ($k = 4$) is shown in Figure 5.1. Note that the number of nodes is $n = 3k$, i.e., $n = 12$ in this example. Also note a maximal independent set consists of k nodes, one taken from each component. Thus, there are $3^k = 3^{n/3}$ maximal independent sets in a graph of this type. The complement of this graph has that number of cliques. Moon and Moser show that no other graph has more cliques, as a function of n. Listing the cliques of a graph must take at least as many steps as there are cliques. Thus, in general, enumeration of all the maximal cliques in a graph may be exponential in the size of the graph.

The genomic overlap graph studied in this application is essentially a noisy interval graph. As discussed in Chapter 2, the maximal cliques of an interval graph can be found in linear time. Such an algorithm applied to an overlap graph which is not a perfect

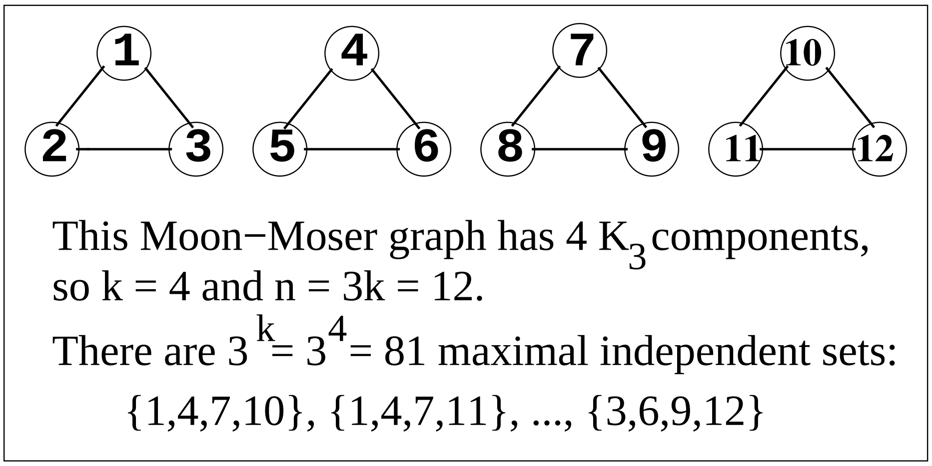


Figure 5.1: Example of the type of graph which Moon and Moser showed to have an exponential number of maximal independent sets. The complement of this graph has an exponential number of cliques.

interval graph might stall and produce nothing useful, but it might be possible to adjust it so that it gives an approximate listing of the cliques. However, I did not explore this possibility of a fast approximate solution, since it seemed more appropriate to find an exact solution if at all practical.

Pardalos *et al.* [74, 75] review the development of algorithms for enumerating the maximal cliques of a graph. According to their review, probably the first such algorithm published was that of Harary and Ross [42] in 1957. Several other clique-listing algorithms appeared in subsequent years, and in 1970 Augustin and Minker [5] analyzed and compared some of them. They found the algorithm of Bierstone (previously unpublished) to be the most efficient among those tested. Mulligan and Corneil [66] corrected two errors in the Augustin-Minker listing of the Bierstone algorithm.

In 1973, Bron and Kerbosch [10], proposed two *backtracking* algorithms for enumerating the maximal cliques of a graph. The characteristic of a backtracking algorithm is that it explores a search tree of possible solutions, and after each branch of the tree is explored, the algorithm 'backtracks' to an earlier-visited node in the tree and explores a new branch. Systematic exploration of the search tree prevents the regeneration of already-found cliques and avoids exploration of some branches which cannot lead to new solutions.

To understand the Bron-Kerbosch (BK) algorithm and the concept of backtracking, first consider a brute force algorithm, EXTEND(NEIGHBOURS), which forms the shell of the BK algorithms. This algorithm enumerates maximal cliques of a graph G = (V, E), where we assume that the following information is global to the EXTEND routine: (1) adjacency information for the graph G, and (2) an array, CLIQUE, which holds a developing clique. Initially CLIQUE is empty, and $NEIGHBOURS = V$. In general, $NEIGHBOURS \subseteq V$ contains vertices not in CLIQUE but which are adjacent (i.e., neighbours) to all of the vertices in CLIQUE. The algorithm has the following steps:

1. If NEIGHBOURS is empty then
   (a) print CLIQUE, since it holds a maximal clique
   (b) return
2. Otherwise, for each vertex $x \in NEIGHBOURS$
   (a) add x to CLIQUE,
   (b) $EXTEND(NEIGHBOURS_x)$, where $NEIGHBOURS_x = NEIGHBOURS \cap Adj(x)$
   (c) remove x from CLIQUE.

This algorithm can be viewed as tracing paths through search trees, as illustrated in Figure 5.2, for the example graph of Figure 1.1. (The style of this drawing includes ideas from Tomita *et al.* [93].) Each oval represents a call to EXTEND(NEIGHBOURS), where the vertices within the oval are the vertices in NEIGHBOURS. Thus, the oval at the top of the figure represents the first call, EXTEND(V). Each vertex in the top oval forms the root of a search tree which is traversed as the vertex x is added to CLIQUE

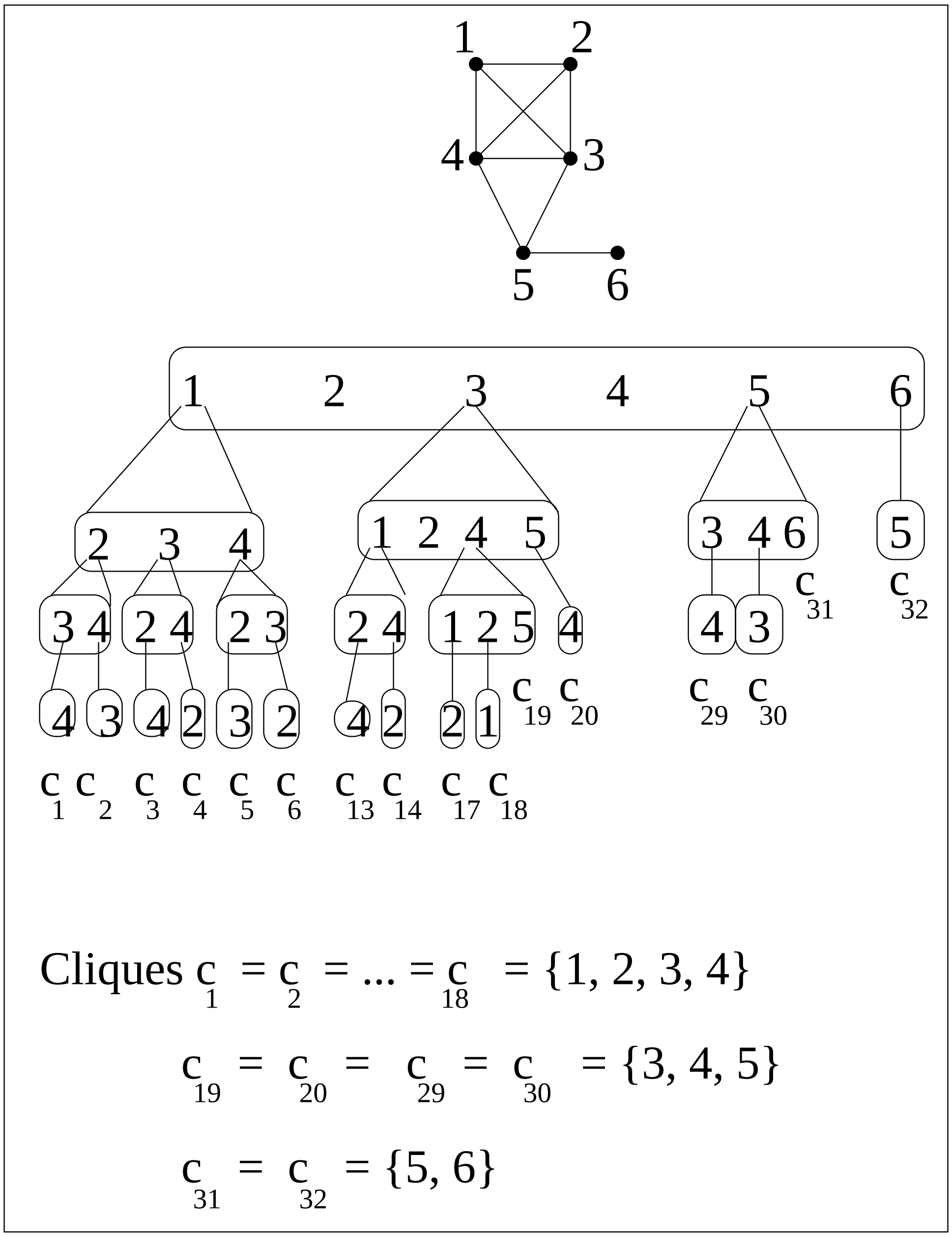


Figure 5.2: Illustration of a search tree for finding cliques using the brute force algorithm. Search trees rooted at 2 and 4 are omitted for clarity. See text for explanation.

in Step 1. The children of a node are the vertices in $NEIGHBOURS_x$. A path from root to leaf represents a maximal clique. In this example, 32 cliques are found, although there are only 3 unique cliques in the graph. Many duplicates of cliques are found since the search trees are traversed in full.

The problem of duplicates can be corrected by insisting that as each clique is built, the values added to CLIQUE must be added in order of increasing numeric value. To implement this, we add a simple heuristic to prune any branch of the tree where a child node is numerically smaller than the parent node (assuming the nodes are numbered 1 to n). Cliques are then found in in lexicographic order, if the selection of x from $NEIGHBOURS$ is in order of increasing size. This is typical when the adjacency lists are sorted, as shown in Figure 1.2. The algorithm EXTEND(NEIGHBOURS) then becomes:

1. If NEIGHBOURS is empty then
   (a) print CLIQUE, since it holds a maximal clique
   (b) return
2. Otherwise, for each vertex $x \in NEIGHBOURS$
   (a) if CLIQUE is not empty and x is less than the last value added to CLIQUE then continue at the top of this loop (Step 2).
   (b) add x to CLIQUE,
   (c) $EXTEND(NEIGHBOURS_x)$
   (d) remove x from CLIQUE.

Figure 5.3 shows the pruned search trees which are traced by this algorithm on the example graph. Clearly, the pruned search tree leads to a much more efficient solution.

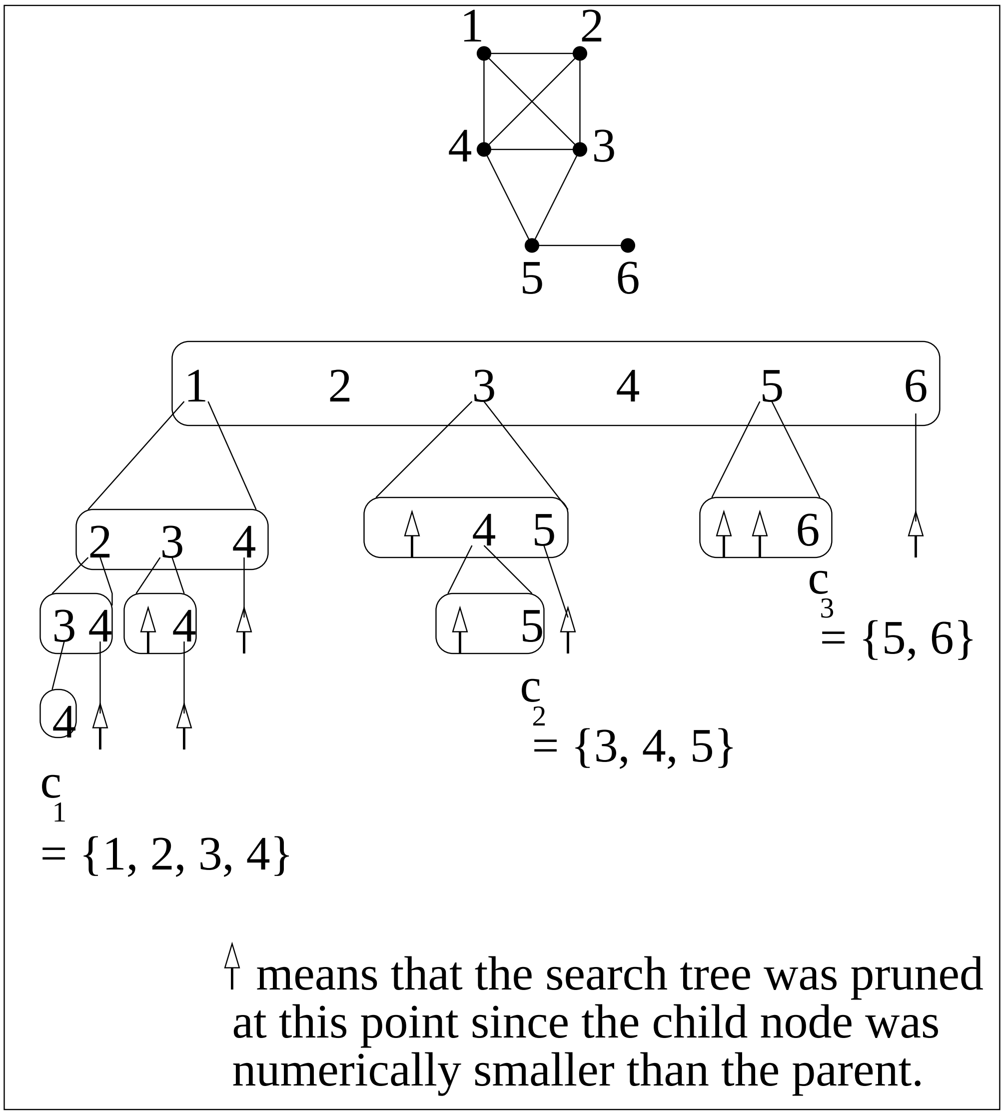


Figure 5.3: Illustration of a search tree for finding cliques lexicographically. The heuristic prunes branches where the vertex sequence is not increasing. This eliminate duplicates. Search trees rooted at 2 and 4 are omitted for clarity. See text for explanation.

As mentioned, the above backtracking algorithm forms the shell of the Bron and Kerbosch algorithms [10]. Bron and Kerbosch express the shell a little differently, however, since they partition NEIGHBOURS into two parts: NOT and CANDIDATES, i.e., $NEIGHBOURS = NOT \cup CANDIDATES$ and $NOT \cap CANDIDATES = \emptyset$. The

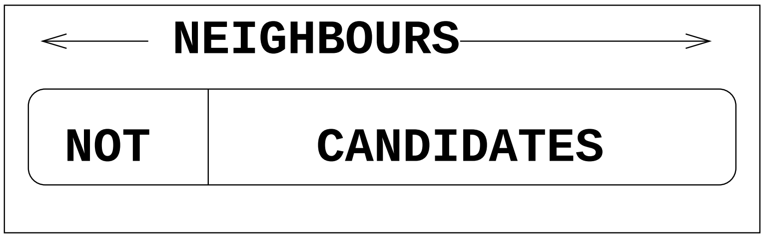


Figure 5.4: Illustration of the partitioning of NEIGHBOURS into NOT and CANDIDATES for the Bron-Kerbosch algorithm.

set NOT contains vertices which have already been used to extend the current vertices of CLIQUE. In the lexicographic algorithms above, NOT would contain the vertices of NEIGHBOURS that are numerically smaller than the current node in the search tree. These are the values which we do not want to add to CLIQUE, since the resulting cliques have already been found. The rest of the vertices in NEIGHBOURS belong to CANDIDATES, and they are worth adding to CLIQUE, since they may generate new cliques. Recall that we have a maximal clique when NEIGHBOURS is empty, which means when both NOT and CANDIDATES are empty. Thus, the shell of the Bron-Kerbosch algorithm is EXTEND(NOT, CANDIDATES):

1. If CANDIDATES and NOT are both empty then
   (a) Print CLIQUE, since it holds a maximal clique.
   (b) Return.
2. If CANDIDATES is empty (and NOT is not empty) then return, since this branch does not generate a new maximal clique.

3. Otherwise, for each vertex $x \in CANDIDATES$

   (a) Add x to CLIQUE.

   (b) Form $NOT_x = NOT \cap Adj(x)$ and $CANDIDATES_x = CANDIDATES \cap Adj(x)$

   (c) $EXTEND(NOT_x, CANDIDATES_x)$

   (d) Remove x from CLIQUE and move x from CANDIDATES to NOT.

To this shell, Bron and Kerbosch add two heuristics. The first heuristic is an early-detection scheme for a situation where NOT will never get empty. When this situation arises then the search tree can be pruned early. It occurs when a vertex in NOT is found to be adjacent to all vertices in CANDIDATES. After all, NOT and CANDIDATES become smaller during step 2(b) above, when we intersect NEIGHBOURS with the neighbourhood of vertex x, which is selected from CANDIDATES. If there is a vertex $q \in NOT$ which is adjacent to each $x \in CANDIDATES$, then q will never be removed from NOT. The final clique obtained when CANDIDATES is empty will not be maximal, since q would be adjacent to all of its elements. Version 1 of the Bron and Kerbosch algorithm implements this heuristic.

Figure 5.5 shows the search tree for the example graph when using the first heuristic of Bron and Kerbosch. In this figure the set NEIGHBOURS is drawn with two compartments, as in Figure 5.4. The left compartment is for NOT, and the right side is for CANDIDATES. The first heuristic, labeled H_1, is applied in three places. In the leftmost application, vertex 2 in NOT is adjacent to the only vertex, 4, in CANDIDATES. In the next application, vertex 1 in NOT is adjacent to both vertices 3 and 4 of CANDIDATES. In the third application vertex 3 of NOT is adjacent to the only vertex, 5, of CANDIDATES.

The second heuristic of Bron and Kerbosch is a scheme for making the above situation (where a vertex $q \in NOT$ is adjacent to each vertex $x \in CANDIDATES$) happen as

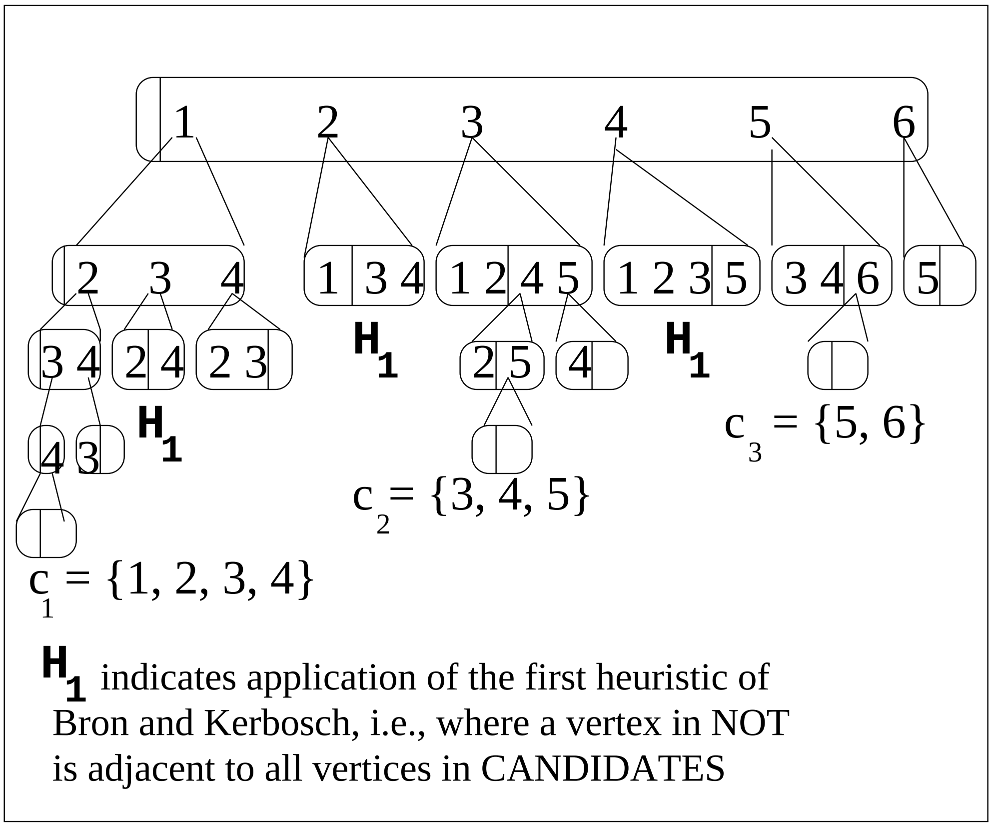


Figure 5.5: Illustration of the search tree for finding cliques using the first heuristic of Bron and Kerbosch. See text for explanation.

early as possible. Imagine a counter associated with each vertex q in NOT, where the counter counts the number of vertices in CANDIDATES to which q is not connected. Bron and Kerbosch call this value the number of *disconnections*. We can apply the first heuristic when there is a vertex q in NOT with zero disconnections, since that means q is adjacent to all of the vertices in CANDIDATES. To arrive at this state quickly, Bron and Kerbosch add two more features to the EXTEND routine: (1) just before Step 3 (where each vertex x is chosen from CANDIDATES), choose (from NEIGHBOURS) a fixed point fp which has the smallest disconnection counter, and (2) during Step 3, select vertices x which are not adjacent to fp. In this case, as each vertex x is removed from CLIQUE and placed in NOT in Step 3(d), the counter for fp decreases by one. No counter can decrease by more than one at this step, and since fp starts with a minimal counter, its counter reaches zero as quickly as any counter could possibly reach zero. When the counter for fp reaches zero, then the first heuristic kicks in, and the search tree is pruned.

One detail omitted to simplify the above discussion is that when fp is chosen from NEIGHBOURS, it might be chosen from NOT or from CANDIDATES. If it is chosen from CANDIDATES, then at Step 3 it is the first x to be chosen, so that it will be in NOT for subsequent selections of x. See the Bron and Kerbosch paper [10] for the details of how they neatly implement these two heuristics, using a single array to hold NOT and CANDIDATES.

Figure 5.6 shows the search trees for the example graph used in earlier figures. This figure is drawn in the style of Tomita *et al.*, [93]. In the first call to EXTEND, vertex 1 is chosen as the fixed point since it is one of the vertices in NEIGHBOURS adjacent to the most vertices in CANDIDATES (i.e., with the least disconnections). This selection divides CANDIDATES into neighbours of 1, which will be ignored at this level, and nonneighbours of 1, which will have search trees. This division is the result of the combination of the two Bron-Kerbosch heuristics, and the efficacy of these heuristics is

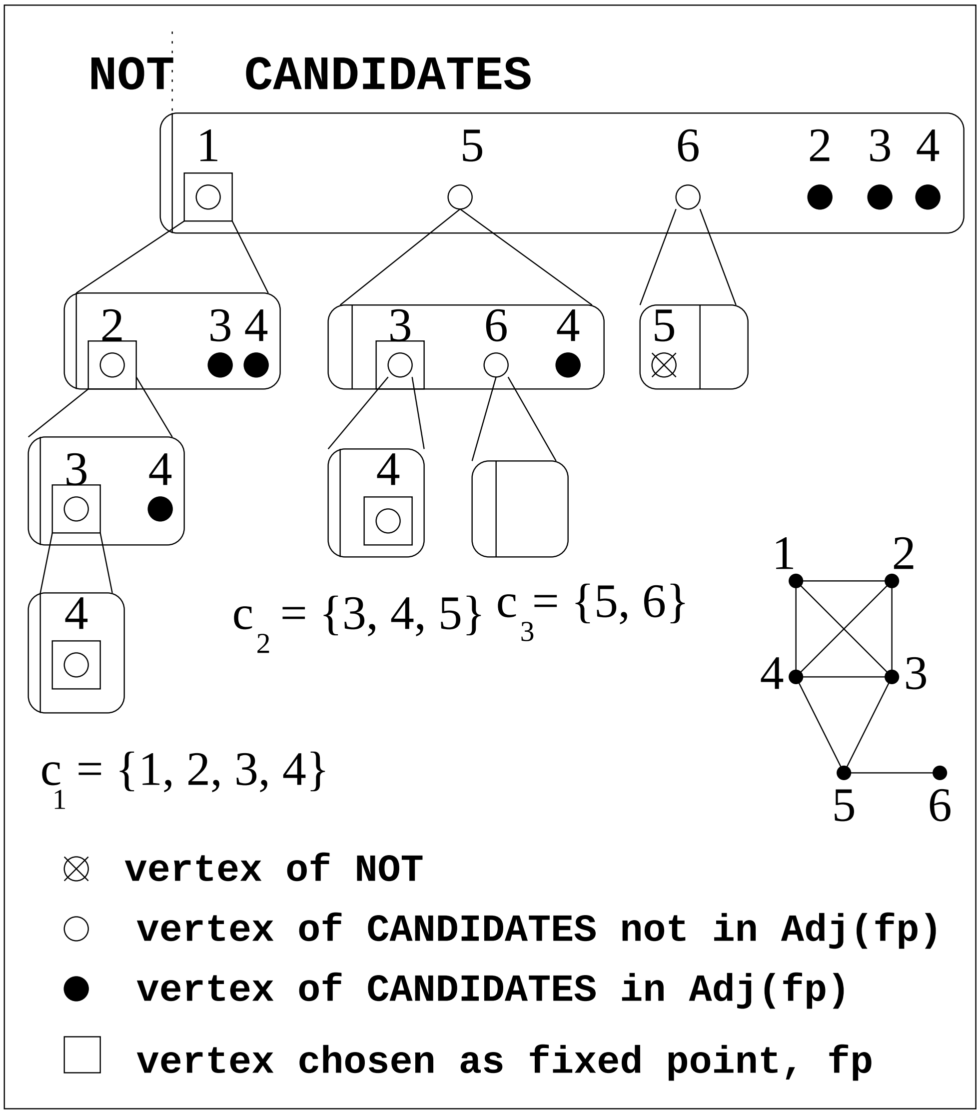


Figure 5.6: Illustration of the search tree for finding cliques using both heuristics of Bron and Kerbosch. See text for discussion.

evident by the number of nodes (colored black) which do not need further exploration. Note that the tree rooted at 6 has a child node 5, but at this point 5 is in NOT and CANDIDATES is empty, so a return is made according to Step 2 of the BK algorithm, as listed above. Also note the empty rounded rectangle which is the leaf of the search tree resulting in clique c_3. This represents Step 1 in our BK listing above, and similar empty rectangles should be drawn for cliques c_1 and c_2, but are omitted for lack of space. In the implementation of the BK algorithm [10] the detection and printing of a maximal clique is actually done after Step 3(b) in order to avoid an extra call to EXTEND.

Bron and Kerbosch tested their algorithm along with that of Bierstone on random graphs of 10 to 50 nodes and edge densities $\rho = 10\% - 95\%$. They found that the version of their algorithm which used both heuristics was markedly faster than the Bierstone algorithm. Over the range of graphs tested, their algorithm found cliques in approximately constant time per clique.

In 1976, Johnston [50] presented a systematic development of a group of clique-listing algorithms which turned out to be variations on the Bron-Kerbosch algorithm. The discussion above of the brute force algorithms leading into the Bron-Kerbosch algorithm is influenced by Johnston's exposition. Johnston tested variations of the BK algorithm on small graphs of up to 48 vertices with edge-densities of 0% to 100%, large graphs of up to 1000 vertices with edge-density of 5%, and graphs of 100 vertices and densities ranging from 5% to 50%. The variation that was most efficient depended on the size and density of the graph. For large graphs with density less than 25% the most efficient variation was one Johnston called the *simplified Bron-Kerbosch* algorithm (SBK). In general, this version was several times faster than the original Bron-Kerbosch algorithm (BK). Johnston compares in theory (without testing) several other enumerative algorithms [2, 3, 23, 73] and concludes that they are not potentially better than the Bron-Kerbosch variations. Finally, Johnston presents a theoretical argument to suggest that in general the BK algorithms will not enumerate cliques in time which is constant per clique.

In 1977, Tsukiyama *et al.* [94] introduced an algorithm for enumerating all of the maximal independent sets (MISs) of a graph, based on Bierstone's vertex sequence method [5] and a new backtracking method. I will denote their algorithm by the authors' initials, TIAS, as do Loukakis and Tsouros [61]. Tsukiyam *et al.* showed that their algorithm had a theoretical time bound of $O(nm\mu)$, where μ is the number of MISs. Tsukiyama *et al.* tested their algorithm on Moon-Moser graphs with k = 4 to 13, and on random graphs with n = 10 to 50 and densities of 25%, 50% and 75%. Figure 5.7 replicates their experiment on random graphs of 50% density, using a C implementation of their algorithm. The logarithm of processing time per MIS (in microseconds) is plotted against the logarithm of the number of edges. The graphs have n = 50, 75, 100, 125, and 150 nodes. The points are the averages of triplicate runs, and the line through the points is a least squares line: slope 0.881 (standard deviation (sd) 0.051) and intercept -0.855 (sd 0.171). This experiment was performed on an Intel Celeron processor running at 1 Ghz, with 128 MB of RAM. The experiments by Tsukiyama *et al.* [94] show that the CPU time per MIS is almost linear in the number of edges. Our experiment approximately corroborates their result over graphs that are a few times larger than what they tested. As an aside, I note that the processor in 1977 experiment took about 55 ms per MIS, compared to about 0.044 ms/MIS in this experiment (both values measured on random graphs of 50% density and n = 50).

In 1981, Loukakis and Tsouros [61] presented a new backtracking algorithm (LTMIS) for listing MISs. Their experiments on graphs of 20 to 220 vertices with edge densities from 20% to 90% indicated that the LTMIS algorithm is two to fifteen times faster than the BK algorithm and at least three times faster than the TIAS algorithm. Loukakis [60] subsequently improved their algorithm.

Figure 5.8 below compares TIAS to LTMIS in terms of average time taken to find the MISs of three random graphs with 50% density. LTMIS is 20 times faster than TIAS when n = 150, and the ratio steadily increases with n over the range tested. Loukakis

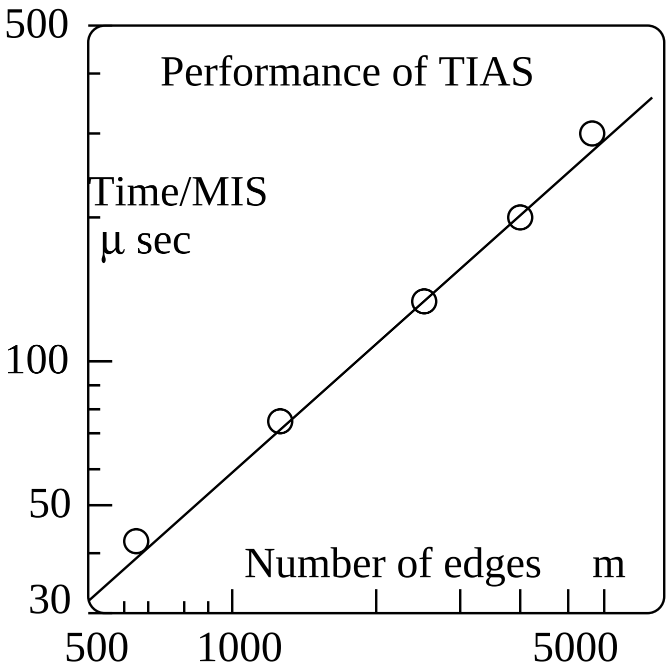


Figure 5.7: Performance of TIAS.

Processing time per MIS is plotted against the number of edges (log-log scale). The input graphs are random with 50% density and n = 50, 75, 100, 125, and 150 nodes. Averages of triplicates are shown.

and Tsouros had similar results at lower densities, but were not able to test TIAS at 50% density random graphs.

Figure 5.8 also compares LTMIS to BK on the same random graphs with 50% density. I denote the BK algorithm by "BK(matrix)" to emphasize that this version of the algorithm uses a matrix to implement the adjacency information. Later I will discuss a linked-list implementation. In this experiment, the BK(matrix) algorithm ran on the complements of the graphs that were the input to the LTMIS algorithm. The complementary graphs were prepared in two ways with the same results. The first method was to use a separate program to complement the adjacency information, which was then fed to the BK(matrix) program. The second method was to add a few lines to BK(matrix) to complement the matrix after reading the adjacency information. In the first method,

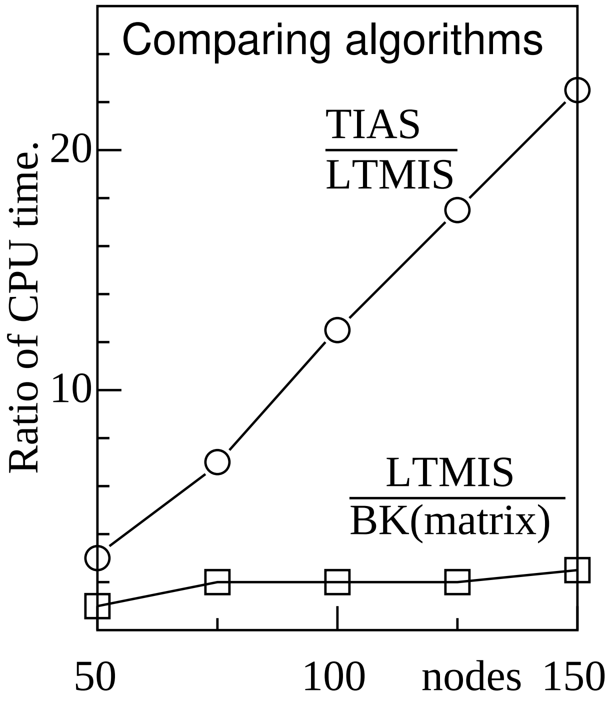


Figure 5.8: Relative performance of algorithms listing MISs of random graphs with n = 50, 75, 100, 125 and 150 nodes, and 50% density. The upper curve is TIAS time over LTMIS time, and the bottom curve is LTMIS time over BK(matrix) time.

the time to complement the input graph is not included in the MIS-computation time, whereas it is in the second method. Complementing the matrix takes negligible time compared to the time spent in finding the cliques (or MISs), so these two methods produced the same results. The figure shows that BK(matrix) is about twice as fast as LTMIS on these input data. The ratio of time taken by LTMIS to that taken by BK(matrix) on 50% density random graphs was: 1.3, 1.9, 2.0, 2.1, and 2.4 for n = 50, 75, 100, 125, and 150, respectively. This observation seems contrary to the results of Loukakis and Tsouros [61]. Their implementation of BK is 7 to 2 times slower than LTMIS for random graphs of 50% density in the range n = 20 to 90, respectively. The reason for this difference is not easily determined, since Loukakis and Tsouros do not discuss their implementation of BK. The C-coding of LTMIS used here is a close translation of the Fortran-coding which Loukakis and Tsouros provide [61].

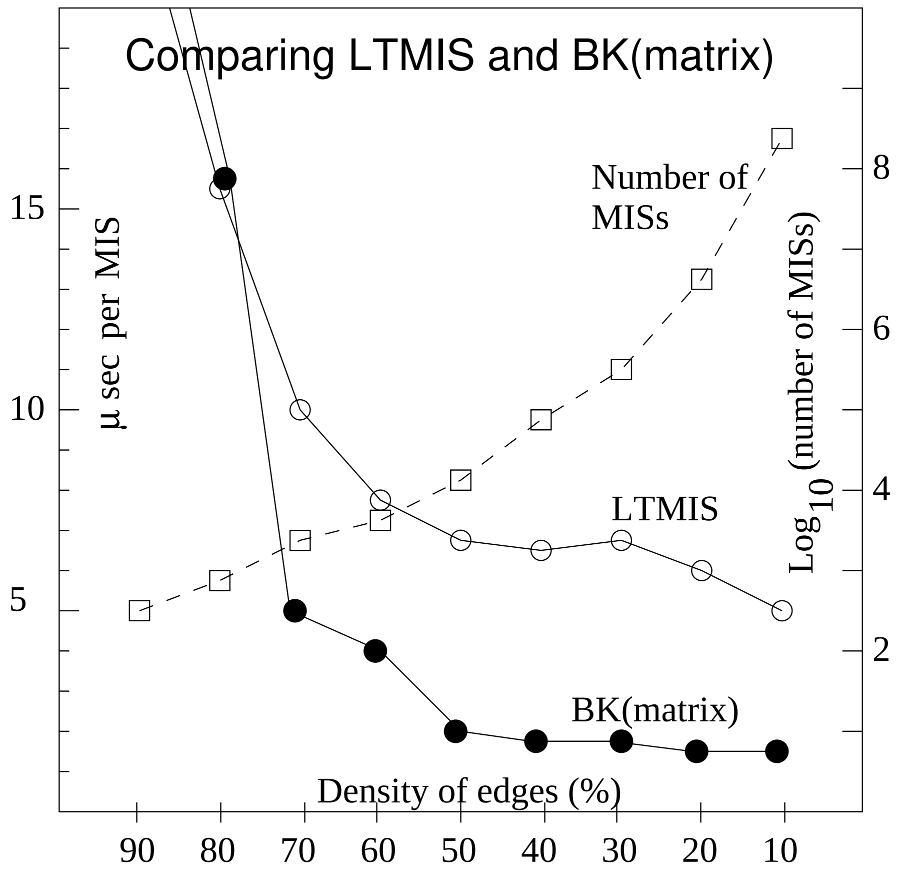


Figure 5.9: CPU time in microseconds per MIS for LTMIS and BK(matrix) on random graphs of 100 nodes and varying edge densities (averages of triplicates). The processing time in these experiments does not include output. The dashed line plots the logarithm of the average number of MISs per graph.

Figure 5.9 tests the performance of LTMIS and BK(matrix) on random graphs of 100 nodes and densities ranging from 10% to 90%. The points shown are the averages of three experiments. As in the previous figure, a few lines of code are added to BK(matrix) so that the adjacency matrix is complemented before the cliques are found. Each clique found by BK(matrix) is then an MIS of the input graph. The processing time per MIS is relatively high at 90% and 80% densities, but the total processing time there is smallest, in the range 10 to 20 ms. At these high densities, most of the processing time is spent setting up the data structures, since there are very few MISs (333 at 10% and 846 at

20% density). As the density decreases, the number of MISs increases to 278 million at 90% density. Storing millions of MISs takes too much disk space, therefore the output was turned off in the experiments represented in this figure. As in the previous figure, BK(matrix) is generally a fewfold faster than LTMIS. At first glance it might seem that these two figures are inconsistent at their point of intersection, n = 100 and 50% density. At this data point in Figure 5.8 the ratio of LTMIS time to BK(matrix) time is about 2:1 (i.e., 170 ms : 87 ms), while in Figure 5.9 the ratio is about 3:1 (i.e., 110 ms : 33 ms). However, the reason for the difference is that the MISs are printed in Figure 5.8 and not printed in Figure 5.9. Subtracting the time for printing makes a difference in the ratio.

In 1985, Chiba and Nishizeki [12] developed an algorithm CN for listing all the maximal cliques of a graph in $O(a(G)m)$ time per clique, where $a(G)$ is the *arboricity* of graph G. The arboricity of a graph is defined in Harary's text [41] as the minimum number of edge-disjoint spanning forests into which G can be decomposed. They showed that for any graph G, $a(G) \leq ceiling(n/2)$. Chiba and Nishizeki did not implement their algorithm (N. Chiba, personal communication), but on theoretical grounds it was expected to be an improvement: "Our algorithms are as fast as the previous ones, if any, and a factor n is often reduced to $a(G)$ in the running time." [12]. In particular, they claimed that for sparse graphs the $O(a(G)m)$ time complexity of the CN algorithm is a big improvement on the $O(nm)$ time complexity of TIAS. Part of the reasoning is that TIAS lists MISs and therefore would work on a graph complementary to the input of CN. Thus, not only is $a(G) < n$, but also in sparse graphs the factor of m would be small for CN and large for TIAS.

Figure 5.10 compares C-code implementations of CN and TIAS with regard to the time to list the cliques of random graphs with 10% density. The times shown are averages of triplicates, and they do not include a preliminary step for each of the algorithms. The preliminary step for CN is to sort the vertices in order of increasing degree, and the preliminary step for TIAS is to complement the input graph. The experiment was run

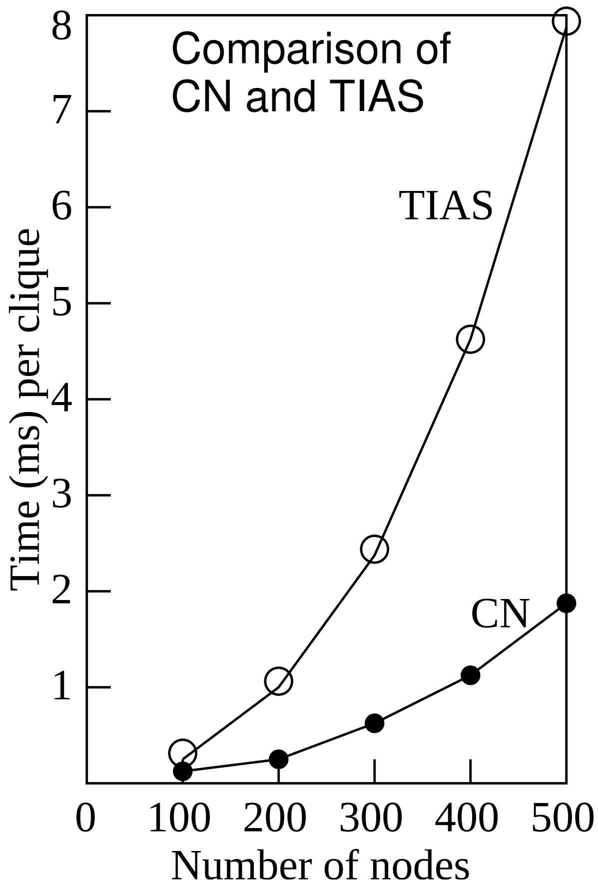


Figure 5.10: CPU time in milliseconds per clique for CN and TIAS on random graphs of 10% density (averages of triplicates). The average number of cliques ranged from 342 for n = 100 to 15162 for n = 500. The input to TIAS is the complement of these graphs.

on an Intel Celeron processor running at 1 Ghz, with 128 MB of RAM. The graph shows that CN is faster than TIAS by a factor of about 4 for random graphs of 10% density over the range n = 100 to n = 500.

The most recent improvement on the time complexity of listing cliques is by Tomita *et al.* [93]. The algorithm which Tomita *et al.* discuss is basically the Bron-Kerbosch algorithm, with an interesting change in the way that the cliques are reported. As vertices are added to CLIQUE, they are printed. When CLIQUE holds a maximal clique, then the string "clique," is printed. When a vertex is removed from CLIQUE, the string "back," is printed. The reason for this change is that it takes O(n) time to print each clique in the usual way, which makes for less than optimal performance in terms of time complexity. Tomita *et al.* then build the recurrence relations for this modification of the Bron-Kerbosch algorithm and report the solution to be $O(3^{n/3})$. This, of course, is the optimal time complexity, considering the existence of the Moon and Moser graph [64] discussed above. Tomita *et al.* [93] point out that without the modification to the printing, the Bron-Kerbosch algorithm has complexity $O(n3^{n/3})$.

5.2 Modifications of the BK Algorithm

This section discusses a couple of simple modifications to the original Bron-Kerbosch algorithm which dramatically improve its performance on the overlap graphs studied here.

The first modification is not really to the algorithm, but rather to the data structure holding the adjacency information of the graph. For large overlap graphs, this must be changed from a matrix to an adjacency list (see Figure 1.2). The largest graph tested in the Bron and Kerbosch paper [10] has $n = 50$ nodes, for which the matrix representation is the natural choice. The overlap graph for merged STS, fingerprint, and Alu-PCR data contains 30,744 nodes (YACs), and 271,273 undirected edges. Assuming one byte per memory location, the adjacency matrix would require about 900 MB of RAM. Workstations used in this study have only 64 to 128 MB RAM, so the matrix representation is not feasible. I will denote the BK algorithm which is based on the matrix data structure as BK(matrix), and the BK algorithm using the adjacency list as BK(list). BK(list) finds the cliques (about 40,000) in the merged overlap data in three hours on a Sun Sparc Station 10 workstation.

The second modification is a small but significant change in the algorithm to reduce its running time. As described above, on entry to the EXTEND function, the arrays CANDIDATES and NOT are scanned to locate the node with the smallest number of edges to other nodes in CANDIDATES. This search requires time quadratic in the size of the set CANDIDATES. Although the scan sets up efficient choices of nodes with which to extend CLIQUE, the scan becomes very time consuming as the number of nodes in the graph increases. The modification reduces the range of this scan by applying the steps of the basic algorithm locally to small subgraphs of the original overlap graph. Each subgraph is composed of a node and its neighbors. When all maximal cliques for a given node are found, the node is marked as *done* using a global array, and this information is merged with the NOT array before moving on to the next subgraph. In essence, the modification applies the BK algorithm to search each subgraph S_i, $(1 \leq i \leq n)$, induced by node i and its neighbors $Adj(i)$, for maximal cliques constrained to involve only vertices $j >= i$, if the subgraphs S_i are explored in order of increasing i. The algorithm does not rely on this assumption, however. I will denote this algorithm as MBK(list), i.e., modified Bron-Kerbosch algorithm. It uses the abbreviated adjacency list data structure of Figure 1.3(c), since for some graphs the full adjacency list was too large. For the purpose of testing performance, I have also implemented MBK using a matrix, in which case the algorithm will be denoted MBK(matrix). Whereas BK(list) took 3 hours to find the cliques in the merged overlap graph, MBK(list) took 3 minutes (both algorithms running on a Sun Sparc Station 10). A listing of the C code for MBK(list) is provided in Appendix C, along with a more detailed comments.

5.3 Experimental comparisons

This section presents results comparing the modified Bron-Kerbosch algorithm with the original Bron-Kerbosch algorithm and with several other maximal-clique listing algorithms.

Published experimental comparisons among clique-enumeration algorithms are generally quite limited. First, they typically involve only a small subset of the published algorithms. For example, Bron and Kerbosch [10] compare their algorithm with only one other (that of Bierstone). Tsukiyama *et al.* [94] show experimental results for their algorithm only. Chiba and Nishizeki [12] and Tomita *et al.* [93] show no experimental results at all. Loukakis and Tsouros [61] compare their algorithm with just two others. One of the contributions of this thesis is the coding in C and systematic comparison of five maximal clique algorithms in addition to the modified Bron-Kerbosch algorithm presented here. The C code for these algorithms will be freely and easily accessible so that researchers can readily compare their algorithms with several published ones.

The second aspect in which published comparisons of maximal-clique algorithms are limited is the type of graph used for input. Typically the experiments are run on small random graphs, usually with a few hundred nodes or less. The Moon-Moser graph [64] is also frequently used since it has the most cliques per node. These two types of graph provide a reasonable testbed for the algorithms, but it must be understood that the algorithms may have quite different relative performance on other types of graphs.

Of particular interest here is how various maximal-clique listing algorithms perform on overlap graphs ranging in size from tens to thousands of nodes. The overlap graph is basically a noisy interval graph, where the interval model corresponds to overlapping genomic fragments. The depth of coverage of the fragments (i.e., the number of layers of fragments at a given point) is typically less than 10 in the experiments we discuss. Therefore, as the number of nodes increases, the overall density of edges decreases. Thus, a large overlap graph of about 30,000 nodes is very sparse, with an edge density of about $\rho = 0.1\%$.

The preceding section mentions that the modified Bron-Kerbosch algorithm provides a dramatic improvement in the run time on the merged overlap graph. This section makes comparisons of several algorithms over a wide range of sizes of overlap graphs and random graphs. The algorithms are all implemented in C and compiled with the gcc compiler on a Linux or Unix platform. The implementations of the algorithms and their data structures are briefly discussed below.

BK(matrix) The Bron-Kerbosch algorithm [10] was implemented in C by translating the Algol 60 code published by Bron and Kerbosch [10]. An adjacency matrix (in the style of Figure 1.2(b)) is used to represent the edges, as is the case in the original paper.

BK(list) This implementation of the Bron-Kerbosch algorithm uses the abbreviated linked list (in the style of Figure 1.3(c) to hold the adjacency information. A linked list was used in place of the matrix, since some of the computers used in this study could not hold in RAM the adjacency matrix for graphs larger than 5000 nodes.

LTMIS This is the MIS-listing algorithm of Loukakis and Tsouros [61]. Code in C for LTMIS was produced by translating their FORTRAN code. Following the FORTRAN code, the C implementation of LTMIS uses the array implementation of adjacency list shown in Figure 1.2(d).

It should be noted that LTMIS finds maximal independent sets. Therefore, when the input to the other algorithms is a graph G, then the input for LTMIS is the complement of G, $\overline{G}$. A separate program produces the complement graph, and the time for this preprocessing step is not included in the computation time of LTMIS.

TIAS This is the MIS-listing algorithm of Tsukiyama, Ide, Ariyoshi and Shirakawa [94]. Code in C for TIAS was produced by implementing the published pseudocode. [94]. Adjacency information is stored as in LTMIS, using the array implementation of the adjacency list (Figure 1.2(d)). As with LTMIS, when TIAS is used to find cliques, the input is preprocessed to represent the complementary graph.

CN This is the maximal clique listing algorithm of Chiba and Nishizeki [12]. Code in C for CN was produced by implementing the published pseudocode [12]. Adjacency information is stored in a minimized linked list (in the style of Figure 1.3(c)).

The CN algorithm specifies that the nodes of the graph be sorted in ascending order according to degree. A program separate from CN does the sorting, producing an isograph G_{sorted} which becomes the input to the CN algorithm. The preprocessing time of sorting is not counted as part of the computation time of the algorithm (although it might be reasonable to do so).

SBK This is Johnston's simplified Bron-Kerbosch algorithm [50], which he shows to be faster than BK under certain conditions. Code in C for SBK was produced by implementing the published pseudocode [50]. Adjacency information is stored in a minimized linked list (in the style of Figure 1.3(c)).

MBK(list) This is the modification of the Bron-Kerbosch algorithm discussed in the preceding section. Adjacency information is stored in a minimized linked list (in the style of Figure 1.3(c)) since some graphs of genomic data were too large in the full linked list. Adjacency information for each vertex is read from disk a second time when calculating the maximal cliques involving that vertex. The reason for this is that the minimized adjacency list does not allow easy access to all of the neighbours of a given vertex.

MBK(matrix) This is the same algorithm as MBK(list), but using a matrix to store the adjacency information.

In many experiments with large and small graphs of various types, it was confirmed that all of the algorithms (BK(matrix), BK(list), SBK, MBK(list), MBK(matrix), LT-MIS, TIAS, CN) produce exactly the same set of cliques. Comparisons of the listed cliques require sorting the output files, since each algorithm produces cliques in a different order and with different ordering of the elements of each clique. Once the output files were sorted, the comparison was done using the unix program cmp. Moreover, in each timing experiment it was verified that the algorithms produced the same total number of cliques. The reported times include the time required to write the maximal cliques to a file, unless otherwise stated.

I next compare the efficiency of these algorithms for enumerating cliques in genomic overlap data. The overlap graphs were generated using simulated genome overlap data rather than real genome data in order to allow direct control over the size and number of input graphs. A description of the simulator and its use here can be found in the Appendix A. The results in Tables 5.1, 5.2 and 5.3 and Figure 5.11 show that the MBK algorithm is much faster than any of the other algorithms tested on these simulated overlap data. Table 5.3 shows the same data in terms of ratio of the time taken by the given algorithms and MBK(list). Clearly, the performance of the TIAS, CN, LTMIS, SBK, and BK relative to MBK steadily gets worse as n increases. MBK is the only algorithm tested for which the processing time grows linearly with the number of vertices in the overlap graph. This linear dependency is shown in Figure 5.12. A few more detailed comments for each algorithm are given below.

BK(matrix) In this set of experiments, on graphs with $n \leq 200$, BK(matrix) is as fast as MBK(matrix). As the number of vertices increases further, however, the performance of BK(matrix) progressively falls further behind that of MBK(matrix). At about 6000 vertices, BK(matrix) takes almost six times as long as MBK(matrix) to enumerate the cliques. The matrix implementation for these two algorithms is limited by the memory size and stack limit of the computer. In this experiment, the computer had 128 MB of RAM and no stack limit. The matrix implementation in this case was not suitable for graphs larger than about 7000 vertices.

n	m	ρ (%)	c	TIAS(matrix)	BK(matrix)	MBK(matrix)
101	799	16	539	1,410	10	10
205	1,567	7.5	993	12,750	20	20
406	3,122	3.8	2,024	106,233	50	40
801	6,189	1.9	3,893	***	136	83
1,602	12,607	0.98	8,041	***	420	173
3,202	24,736	0.48	15,979	***	1376	386
6,402	49,631	0.24	32,114	***	4793	836

Table 5.1: Time (ms) for matrix-based TIAS, BK and MBK algorithms to enumerate maximal cliques in simulated genomic overlap data.

Notes: (i) Headings are: n = number of vertices; m = number of edges; ρ = edge density; c = number of cliques; TIAS(matrix) = the algorithm of Tsukiyama et al. *[94] (with an adjacency matrix for edges and an initial step to complement the matrix so that cliques are found rather than MISs); BK(matrix) = the original Bron Kerbosch algorithm (with an adjacency matrix for edges) [10]; MBK(matrix) = the matrix implementation of the modified BK algorithm (iii) The adjacency matrix was* 7000×7000 *(of type short) and memory limitations did not allow further increase in size required for another line in the table, with n = 12,806. (iv) The graphs are formed from simulated mapping data where the coverage is 10 and the false negative rate is 30% and the false positive rate is 10%. (vi) Values are averages of three experiments. The standard deviation for the elapsed time of BK and MBK is in each case less than 10 ms. Timing was based on the clock() function in C, which seems to have a resolution of 10 ms on my system. Thus, the relative error is large for times that are less than 50 ms. (vii) *** indicates the experiment was not done, since TIAS was taking too long. (viii) These experiments were performed on a 500 Mhz Pentium III processor with 128 MB of RAM.*

n	$\rho(\%)$	c	CN	LTMIS	BK(list)	SBK	MBK(list)
101	16	539	446	37	37	30	30
205	7.5	993	1,660	154	70	50	43
406	3.8	2,024	6,866	637	183	120	73
801	1.9	3,893	26,023	2,437	603	350	133
1,602	0.98	8,041	109,800	10,243	2,270	1,206	260
3,202	0.48	15,979	***	***	9,270	4,730	526
6,402	0.24	32,114	***	***	45,520	23,177	1,506
12,806	0.12	65,125	***	***	208,347	103,893	2,180
25,616	0.061	130,889	***	***	858,393	428,453	4,527

Table 5.2: Time (ms) for list-based algorithms to find and print maximal cliques in simulated genomic overlap data.

*Notes: (0) The experimental conditions here are the same as in Table 5.1. The number of edges is shown in the previous table, except for the two new rows, where $m = 99722$ for $n = 12806$, and $m = 199418$ for $n = 25616$. (i) The headings are: n = number of vertices; ρ = edge density; c = number of cliques; CN = the algorithm of Chiba and Nishizeki [12]; LTMIS = the algorithm of Loukakis and Tsouros [61]; BK(list) = the Bron Kerbosch algorithm using an adjacency list for edges [10]; SBK = Johnston's simplified Bron Kerbosch algorithm [50]; MBK(list) = the modified Bron Kerbosch algorithm with a linked list implementation ; (ii) *** represents* not done *for reasons of time or space. For example, when n = 1600, the complement graph for LTMIS requires about 10 MB of storage. The CN experiments were stopped for reasons of time.*

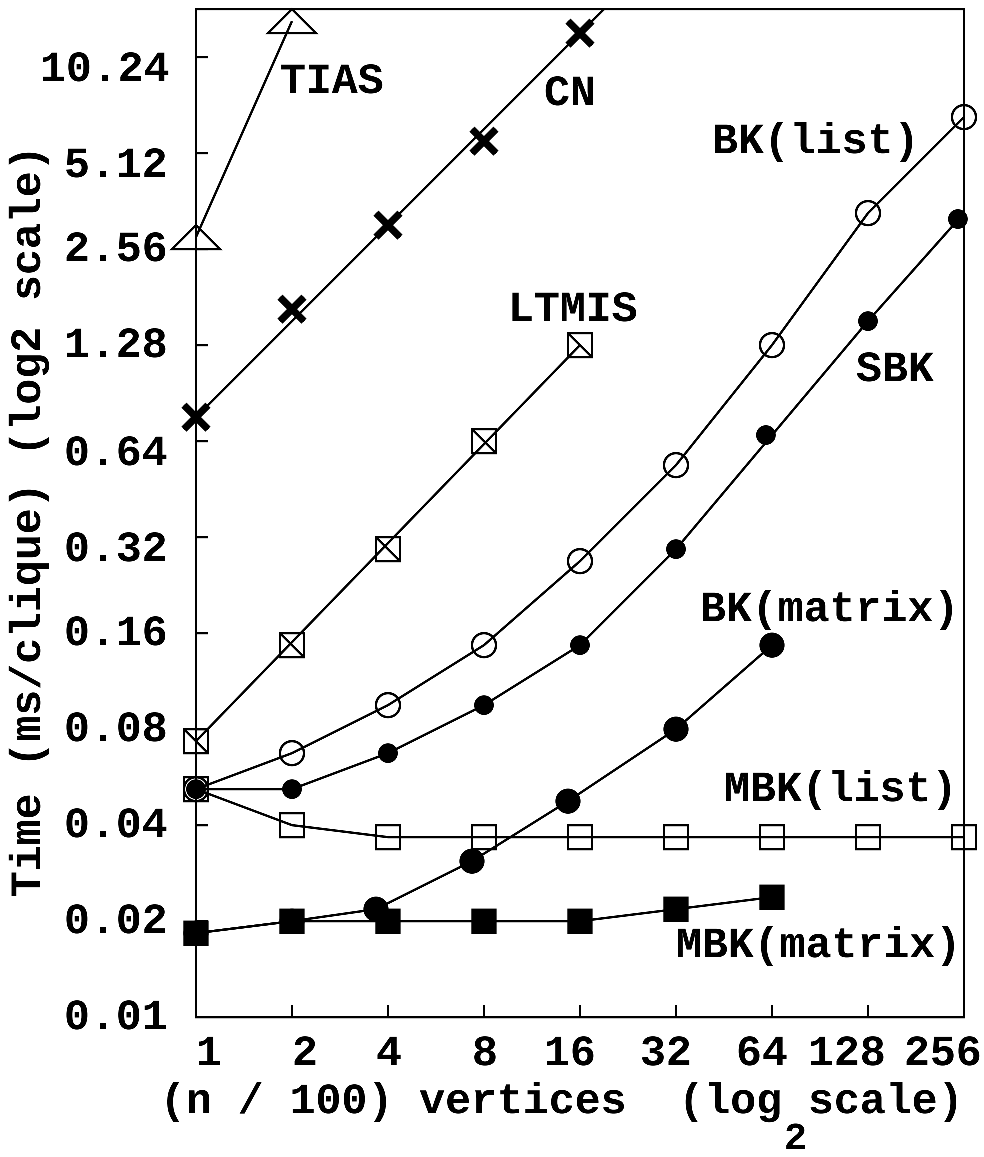


Figure 5.11: Performance of clique-listing algorithms on simulated genomic data.

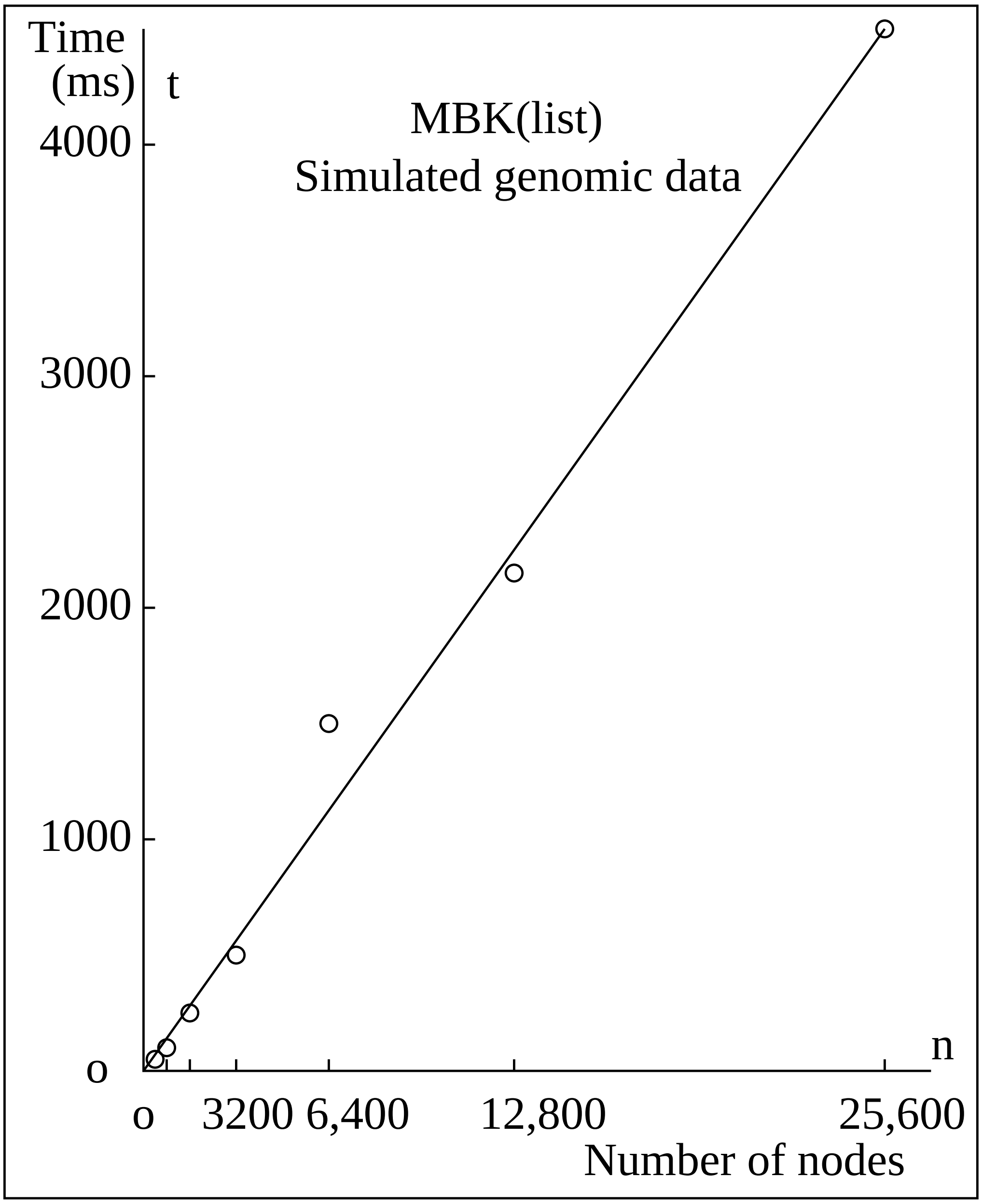


Figure 5.12: Performance of MBK(list) on simulated genomic data.

This is a linear-linear plot of the MBK(list) results from Table 5.2. A least squares fit is shown, with slope = 0.176 ms/node (s.d. 0.006) and y-intercept = 28.6 ms (s.d. 58).

n	$\frac{TIAS(matrix)}{MBK(matrix)}$	$\frac{BK(matrix)}{MBK(matrix)}$	$\frac{BK(list)}{MBK(list)}$	$\frac{SBK}{MBK(list)}$	$\frac{CN}{MBK(list)}$	$\frac{LTMIS}{MBK(list)}$
101	140	1.0	1.2	1.0	14.9	1.2
205	640	1.0	1.6	1.2	38.6	3.6
406	2660	1.4	2.5	1.6	94.0	8.7
801	***	1.6	4.5	2.6	196	18.3
1,602	***	2.4	7.7	4.6	422	39.4
3,202	***	3.6	17.6	9.0	***	***
6,402	***	5.7	30.2	15.4	***	***
12,806	***	***	95.6	47.7	***	***
25,616	***	***	190	94.6	***	***

Table 5.3: Ratio of the time for the given algorithms relative to MBK to list maximal cliques in simulated genomic overlap data. This is another representation of the data from the previous two tables.

BK(list) Comparing BK(list) and BK(matrix) shows the difference in time attributable to the data structure. BK(list) is 2 to 10 times slower than BK(matrix) in the range $n = 100$ to $n = 6400$. BK(list), however, has the critical advantage that it is applicable to much larger values of n than BK(matrix) can manage. In this experiment BK(matrix) was limited to graphs of about 7000 nodes or less, while BK(list) handled the full range up to $n = 25,600$.

The comparison between BK(list) and MBK(list) shows the effect of the feature in MBK which focuses the BK heuristics to small subgraphs. On small genomic overlap graphs of about 100 nodes BK(list) and MBK(list) are very similar in speed (BK(list) to MBK(list) ratio 1.2). However, the ratio of their speeds steadily increases with n, and at $n = 25,600$ BK(list) is 190 times slower than MBK(list).

SBK Johnston's algorithm SBK uses the same data structure as BK(list). SBK tends to be about two times faster on this data than BK(list). Like BK(list), it is similar in speed to MBK(list) on smaller graphs of 200 nodes or less, but progressively worsens in relative performance as n increases. At 25,600 nodes it is 95 times slower than MBK(list).

CN CN is 15 times slower than MBK(list) on a graph of 100 nodes, and its performance quickly worsens relative to MBK(list) as n increases. When $n = 1600$, CN is already about 400 times slower than MBK(list). There seemed to be no point in testing CN for larger values of n.

LTMIS The performance of LTMIS is similar to MBK(list) for the small graph of 100 vertices, but its relative performance drops off as n increases. It is 39 times slower than MBK(list) on a graph of 1600 vertices. The complement graphs were taking exorbitant amounts of space, so LTMIS was not tested on larger graphs.

MBK(list) Figure 5.12 shows that MBK(list) enumerates all maximal cliques in an amount of time which is approximately linear with the size of the overlap graph. The performance of MBK(matrix) is almost linear in n, but not quite. MBK(matrix) is about two times as fast as MBK(list) over the range tested. MBK(list) is at least 90 times better than any other tested algorithm for the graph with over 25,000 vertices. Similar improvement in computation time is also obtained using real genome overlap graphs. For example, MBK(list) was about 110 times faster than BK(list) on the 30,700-vertex overlap graph for merged data. On a Pentium III running at 500 Mhz, MBK(list) finds the 40,070 cliques in 9.1 seconds, whereas BK(list) requires 17 minutes.

Thus, the time taken by TIAS, CN and LTMIS grows much too quickly with the size of the overlap graph for these algorithms to be useful on large graphs of this type. The Bron-Kerbosch algorithm based on a list data structure, BK(list), or Johnston's simplified version also based on a list data structure, SBK, are practical for overlap graphs up to

n	m	c	BK(matrix)	MBK(matrix)
100	522	354	0	3
200	2,070	1525	20	17
400	8,290	9110	127	117
800	33,140	59,130	1,020	957
1600	132,510	431,024	9,540	9,250

Table 5.4: Time (ms) for algorithms to list maximal cliques in random graphs with 10% density.

Notes: (i) Headings are as in previous tables. (ii) Random graphs were constructed using the method described by Loukakis and Tsouros [61]. (iii) Values are averages of 3 experiments on a Pentium III processor running at 500 Mhz with 128 MB of RAM.

25,000 nodes, but these too would soon be impractical for larger graphs. The algorithm of choice for enumerating cliques of large overlap graphs is clearly MBK(list), which is nearly 100 times better than its nearest rival for overlap graphs as large as 25,000 vertices. The factor of improvement would no doubt continue to grow with n, since MBK(list) behaves linearly with n while the other versions appear to behave quadratically.

It is of interest to ask whether MBK is also faster than the other algorithms on other types of graphs. In Table 5.4 I compare the MBK(matrix) and BK(matrix) on random graphs with 100 to 1600 vertices, and with constant edge density of 10%. There is very little difference in the performance of the two algorithms, suggesting that on random graphs with 10% density the additional heuristic incorporated in MBK has little effect. The heuristic is designed to help when the density of edges is low, so in Table 5.5 I compare a few of the algorithms on random graphs of increasing size and decreasing density. The result here is similar to that in Table 5.2, since MBK behaves linearly with

n	m	ρ (%)	c	BK(list)	SBK	MBK(list)
1600	12792	1.0	10951	1900	1060	330
3200	25592	0.5	23338	8070	4270	690
6400	51192	0.25	48569	38760	19840	1430
12800	81913	0.10	81188	146820	74600	1580
25600	163199	0.050	163833	621610	305640	3300

Table 5.5: Time (ms) for algorithms to list maximal cliques in random graphs with decreasing density and increasing size.

Notes: (i) Headings are as described in previous tables. (ii) These random graphs were formed using the method of Loukakis and Tsouros [61], modified to use a linked list rather than a matrix. (iii) Values are from single experiments on a Pentium III running at 500 Mhz with 128 MB of RAM.

n while BK and SBK appear to be quadratic. The result is that MBK(list) is about 90 times faster than SBK and about 190 times faster than BK(list) on the largest graph tested (25,600 vertices). The matrix versions of BK and MBK were tested on the smaller graphs of this experiment. The times for BK(matrix) were 470, 1540 and 5580 ms, and the times for MBK(matrix) were 220, 490, and 1280 ms, respectively, for $n = 1600$, 3200 and 6400.

The random graphs in Table 5.5 are not entirely satisfactory, because they become so sparse that most of the cliques are just two or three nodes. In the experiment of Table 5.6 I try to make graphs which are, in a sense, long and narrow like overlap graphs, yet with random connections. To achieve this, I give each vertex i (up to $n - 20$) five random edges to vertices from $i + 1$ to $i + 20$. Since vertex i may also receive edges from lower numbered vertices, the degree of each vertex turns out to be approximately nine. This experiment shows the same trend as the others. As n increases and the density decreases,

n	m	ρ (%)	c	BK(list)	MBK(list)
1600	7,226	0.56	4,515	1,530	90
3200	14,388	0.28	9,103	5,990	190
6400	28,878	0.14	18,295	27,660	370
12800	57,817	0.071	36,525	131,480	770
25600	115,762	0.035	73,423	577,000	1,550

Table 5.6: Time (ms) for BK(list) and MBK(list) algorithms to list maximal cliques in 'narrow' random graphs with decreasing density and increasing size.

Notes: (i) Headings are as described in previous tables. (ii) These random graphs are shaped like overlap graphs, i.e., long and narrow (in a certain sense). Each vertex i is connected at random to about 9 other vertices in the range $i-20$ to $i+20$. (iii) Values are from single experiments on a Pentium III running at 500 Mhz with 128 MB of RAM.

the performance of MBK is linear while that of BK is approximately quadratic. In this case MBK is about 370 times faster than BK for the largest graph tested ($n = 25600$).

5.4 Summary

On large, sparse graphs obtained from genomic data, simulated genomic data, or random data, the modified version of the Bron-Kerbosch algorithm (MBK) is manyfold faster than the original version (BK). It is also manyfold faster than any of four tested clique-enumeration algorithms published since the Bron-Kerbosch algorithm (i.e., TIAS, CN, LTMIS, SBK). The factor of improvement tends to get larger as n increases, provided the density of edges is decreasing. Clearly, MBK(list) is ideal for this application, since empirically the time that it takes increases approximately linearly with the size of the overlap graph, assuming constant coverage.

Other observations of interest are as follows.

- The results for the LTMIS algorithm are surprising, since Loukakis and Tsouros [61] show that LTMIS is two to fifteen times faster than the BK algorithm for listing MISs of random graphs of up to 220 nodes. I have tested these two algorithms for listing MISs and cliques on genomic overlap graphs or random graphs, and BK is consistently faster than LTMIS. For example, see Figure 5.9, where for a variety of densities of random graphs with 100 vertices, BK is several times faster than LTMIS. I do not have an explanation for this difference.

- The results in this thesis seem to be the first practical test of the CN algorithm of Chiba and Nishizeki [12]. My experiments show CN to be several times faster than TIAS, which is in accordance with its improved time complexity. However, CN performs very poorly compared to LTMIS or variations of BK. This might be surprising to the authors of CN and the readers of their paper.

- Johnston's SBK algorithm [50] is about twice as fast as BK(list) in my experiments on overlap and random graphs. As mentioned above, however, SBK is almost 100 times slower than MBK on genomic graphs of about 25,000 nodes.

Chapter 6

Uniform Integration of Genome Mapping Data

The methods for analyzing overlap data are distinct from those for analyzing probe data, making integration of the two forms awkward. Conversion of overlap data to probe-like data elements would facilitate comparison and uniform integration of overlap data and probe data using software developed for analysis of STS data. In this chapter I show that overlap data can be effectively converted to probe-like data elements by extracting maximal sets of mutually overlapping clones. Most of the work in this chapter is published in a paper titled *Uniform integration of genome mapping data using intersection graphs*, by this author, A. Bonner and N. Goodman [46]. The ideas are also discussed in another paper by the same authors [44].

6.1 Introduction

Overlap data are fundamental to the analysis of genomes, because it indicates connectivity and similarity among cloned fragments. Inferences of connectivity are critical in the construction of *physical maps* reflecting the position of these cloned fragments in the genome. Some maps are based almost entirely on overlap data, while others are based almost entirely on probe experiments, where the relative position and connectivity of clones is determined by hybridization of probes to clones. Methods of analysis of overlap data and probe data are entirely different, making integration of these two forms of data difficult. I present here a method where overlap data can be analyzed as if it were probe data. This method makes it possible to examine overlap data in new ways, and to integrate probe and overlap data in a uniform manner.

Overlap data commonly arise from fingerprint and Alu-PCR experiments on cloned DNA fragments of genomes, as well as from comparison of sequences of clones. Chapter 3 lists the details of the overlap data analyzed here. There are a number of software systems designed for detailed analysis of fingerprint data (see, for example, work by Bouffard *et al.* [9], Fonstein and Haselkorn [28], Gillett *et al.* [32], and Whittaker *et al.* [99],). In general, the goal of these systems is to determine not only which clones overlap, but also their order and the position of sites for restriction enzymes within each clone.

In papers where overlap and probe information are combined, they are typically examined separately, as in the map-assembly methods of CEPH/Généthon [13] and WI/MIT [47]. The WI/MIT approach is to use fingerprint and Alu-PCR data to disambiguate STS data and to connect STS double-linkage contigs, while the CEPH/Généthon approach is to construct paths of overlapping clones between genetically mapped STS markers. The disadvantage of these *ad hoc* approaches is that arbitrary rules are developed regarding when and how to use the overlap data. For example, WI/MIT apparently does not use fingerprint overlap data to help determine the order of clones or probes within a contig, thus missing a potential contribution of the fingerprint data. CEPH/Généthon describe overlap paths according to the number of overlaps required to connect markers — the longer the path, the less credible; thus, special rules are required for the interpretation of the data.

In contrast to previous approaches, the method described here converts overlap data to probe-like data elements by extracting maximal sets of mutually overlapping clones.

These sets are called *virtual probes* since each one determines a site in the genome corresponding to the region which is common among the clones of the set. Overlap data, represented as virtual probes, can then be analyzed with the same rules and techniques as have been developed for probe data. There are many graphical methods and systems developed for analysis of probe data [22, 38, 45, 62, 65, 68, 85, 97]. These probe-analysis systems can be applied directly to virtual probes, either as the only data type or in combination with conventional probes. Separate analysis of these two different data types by the same system presents a unique opportunity for comparing probe and overlap results based on a single standard. Integrated analysis of the two data types by the same system increases the amount of data available to the system from which to make inferences, resulting in larger contigs (as I will show) and possibly more accurate ordering of probes, cliques and clones.

Converting overlap data to virtual probes is equivalent to finding the maximal cliques of a graph. For this task I use the modified version (discussed in Chapter 5) of the Bron-Kerbosch maximal-clique algorithm [10]. I convert fingerprint and Alu-PCR overlap data to virtual probes. The virtual probes are then analyzed using double-linkage intersection graphs and structure graphs. These methods of analysis were designed for analysis of STS data, but they become applicable to overlap data when it is converted to virtual probes. I also integrate STS data with fingerprint and Alu-PCR overlap data and analyze the integrated data in terms of virtual probes. The integrated data produce longer double-linkage contigs than STS data alone. Thus, the virtual probe technique provides (*i*) a new way to examine overlap data, (*ii*) a basis on which to compare overlap data and probe data using the same systems and standards, and (*iii*) a unique and useful way to uniformly integrate overlap data and probe data.

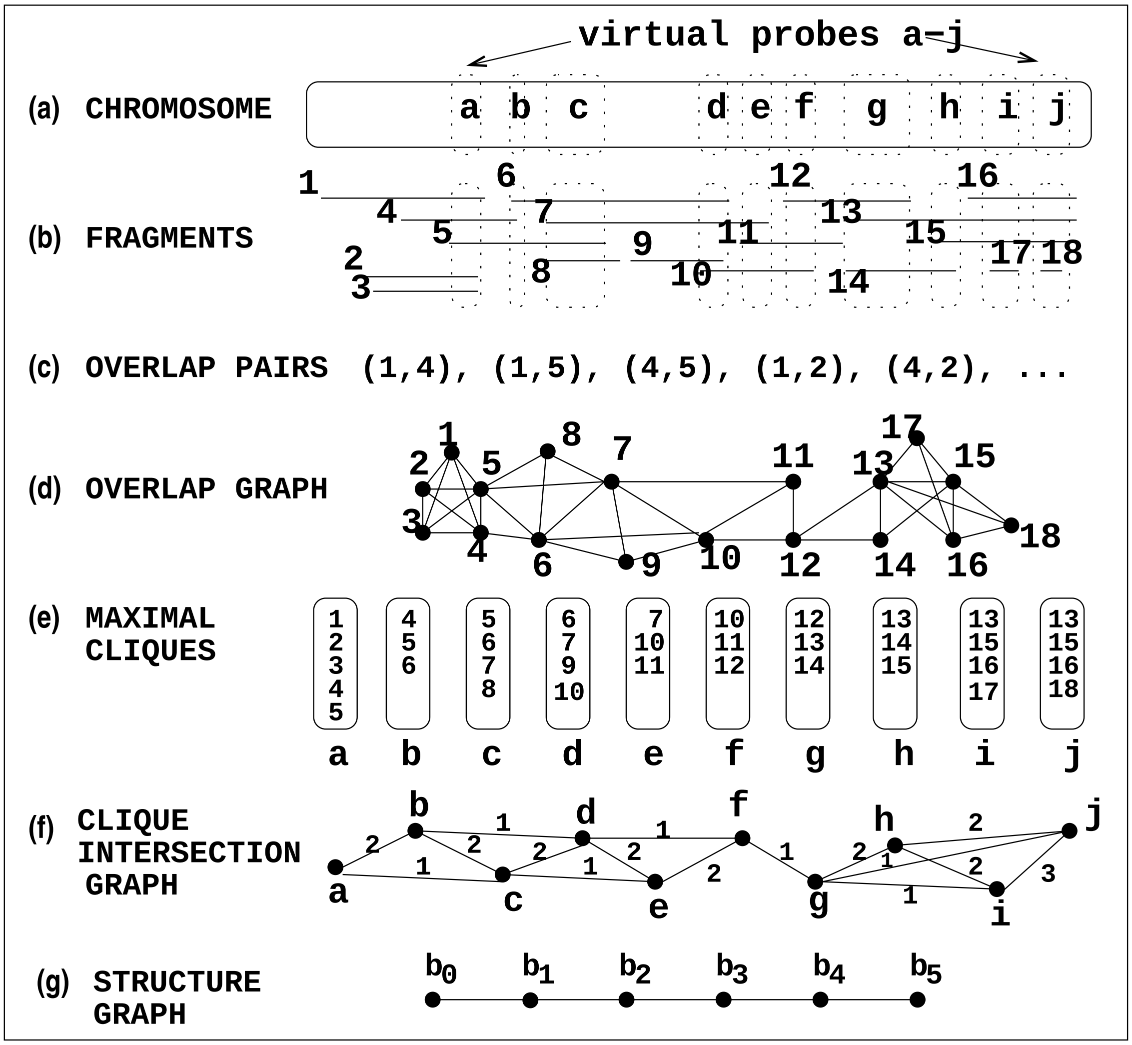


Figure 6.1: Clique model for analysis of overlap data

6.2 Virtual Probe Model

Figure 6.1 illustrates the development and use of virtual probes. Part (a) of this figure shows a chromosome which is the source of a library of overlapping cloned fragments of DNA. Part (b) shows numbered cloned fragments directly under their region of origin on the chromosome. Assume that an experiment determines which fragments overlap. Some of the resulting overlap data are listed in part (c), which shows pairs of identifiers representing overlapping clones. For the purpose of this sketch, we assume that the experiment is perfect, i.e., all of the overlaps in the sketch result in observed overlap pairs, and there are no false-positive pairs. In this case, the sketch in (b) implies a unique set of overlaps in (c), although the converse is not true. Part (d) represents the overlap data as an overlap graph. This is a straightforward representation of overlap data, where nodes correspond to clones, and an edge means that the corresponding clones overlap. Each edge (a,b) in this graph is in one-to-one correspondence with an overlap pair (a,b) in part (c). Figure 6.1(e) shows groups of nodes which form maximal cliques in the example overlap graph.

Notice that each maximal clique corresponds to a region of mutual overlap among its constituent clones, as illustrated by the dotted boxes superimposed on the clones sketched in Figure 6.1(b). In turn, each of these regions represents a site in the genome, as indicated by the placement of the clique labels on the chromosome in Figure 6.1(a). I call these regions *virtual probes* since (like STS probes) they mark sites in the genome, but (unlike STS probes) they are identified indirectly by groups of overlapping clones rather than by sequence. Each virtual probe has breadth equal to the width of the region of mutual overlap. In theory, the genomic region of one virtual probe cannot overlap with that of another, since this would imply that one maximal clique was a subset of another — a contradiction of terms.

Clique intersection graphs are directly analogous to STS intersection graphs. Each node represents a maximal clique/virtual probe in the overlap graph. Figure 6.1(f) shows the weighted intersection graph using labels a,b, ..., j for the cliques found in Figure 6.1(e). For example, the edge from node a to node b is shown with a weight of 2, since cliques a and b have two elements in common: fragments 4 and 5.

In all subsequent graphs of this chapter, edges of unit weight are excluded to produce a *double-linkage* intersection graph. As shown in Chapters 3 and 4, the double-linkage filter greatly reduces the number of edges attributable to false positives and the effects of chimerism. Another filter for false positives is the size of cliques. A clique of size n in the overlap graph represents $\frac{n(n-1)}{2}$ overlapping pairs of clones, and each overlap corroborates

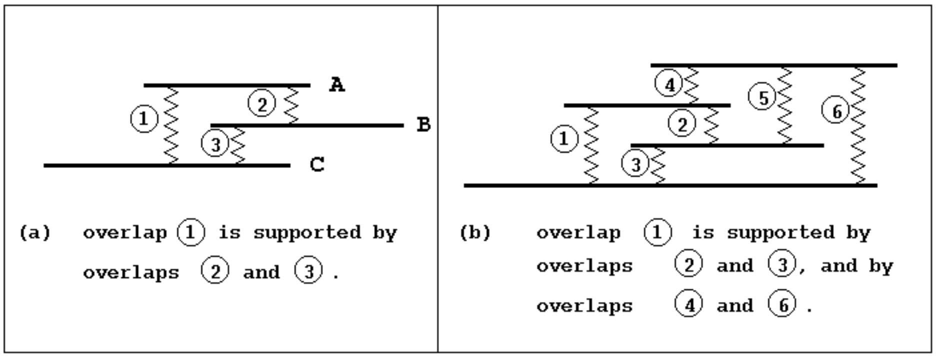


Figure 6.2: Mutual corroboration of overlaps in cliques.

the other overlaps. Figure 6.2(a) illustrates this concept in schematic form for a clique of three mutually overlapping clones. The fact that clone A overlaps clone B, and clone B overlaps clone C implies that clones A and C are near each other in the genome, and this fact provides some support for an observation of overlap between clones A and C. The larger the clique the more certain one may be that the component clones do indeed overlap at a common point, as illustrated in Figure 6.2(b). Cliques of just two clones of course lack this internal corroboration, and therefore we ignore them in order to filter out potential false overlaps. These cliques cannot contribute to a double-linkage connected component anyway — an edge of weight two to a clique of size two would imply that the clique is a subset of its neighbour, contradicting the assumed maximality of the cliques.

The structure graph resulting from this example is shown in Figure 6.1(g). It is a simple path, as will be the case whenever the data are perfect (i.e., when the data can be modeled by a set of overlapping line segments), according to Theorem 1 of Chapter 4. Note that in general, we form structure graphs from double linkage intersection graph, but for illustration purposes I did not remove the edges of unit weight in this example.

Assignment of cliques to chromosomes:

In Chapter 4 I discussed four steps for analysis of probe data, the first step being to isolate data for a single chromosome. To perform this step on virtual probes, we need to first assign them to chromosomes. We do this in the same way that we guessed assignments for unassigned STSs in Chapter 4.

> Assign a virtual probe to Chromosome A if the number of its clones associated with Chromosome A is at least two and greater than the number associated with any other chromosome.

This rule gives very accurate assignments (97% correct) when tested on STS data, as described in Chapter 4. The correctness of its assignments to virtual probes in general cannot be quantitatively assessed, but the fact that it leads to reasonable results on Alu-PCR and fingerprint data suggest that it is reliable here as well.

6.3 Results

6.3.1 Alu-PCR Data

I will briefly examine CEPH/Généthon Alu-PCR data using the virtual-probe method to show that this method provides a useful and informative view of Alu-PCR data in isolation. I applied the modified BK algorithm to extract all of the maximal cliques from an overlap graph of Alu-PCR data. The total number of maximal cliques was 34,643, ranging in size from 2 to 14 YACs. Cliques of size two (20,285) were subsequently ignored as unreliable and not useful for double-linkage graphs. The size distribution of the cliques is shown in Figure 6.3. Cliques were assigned to chromosomes according to the rule described above. This resulted in unambiguous assignment of 87.7% of the cliques, while 3.6% of the cliques could not be assigned because they were equally associated with more than one chromosome, and 8.7% were not assigned because they did not have at least two YACs associated with a single chromosome.

I formed an intersection graph from the 14,358 cliques of size three or more. In this case, I did not first isolate virtual probes for a single chromosome (Step 1 mentioned in Chapter 4), since I wanted to compare cross-linkage in this type of data with that in

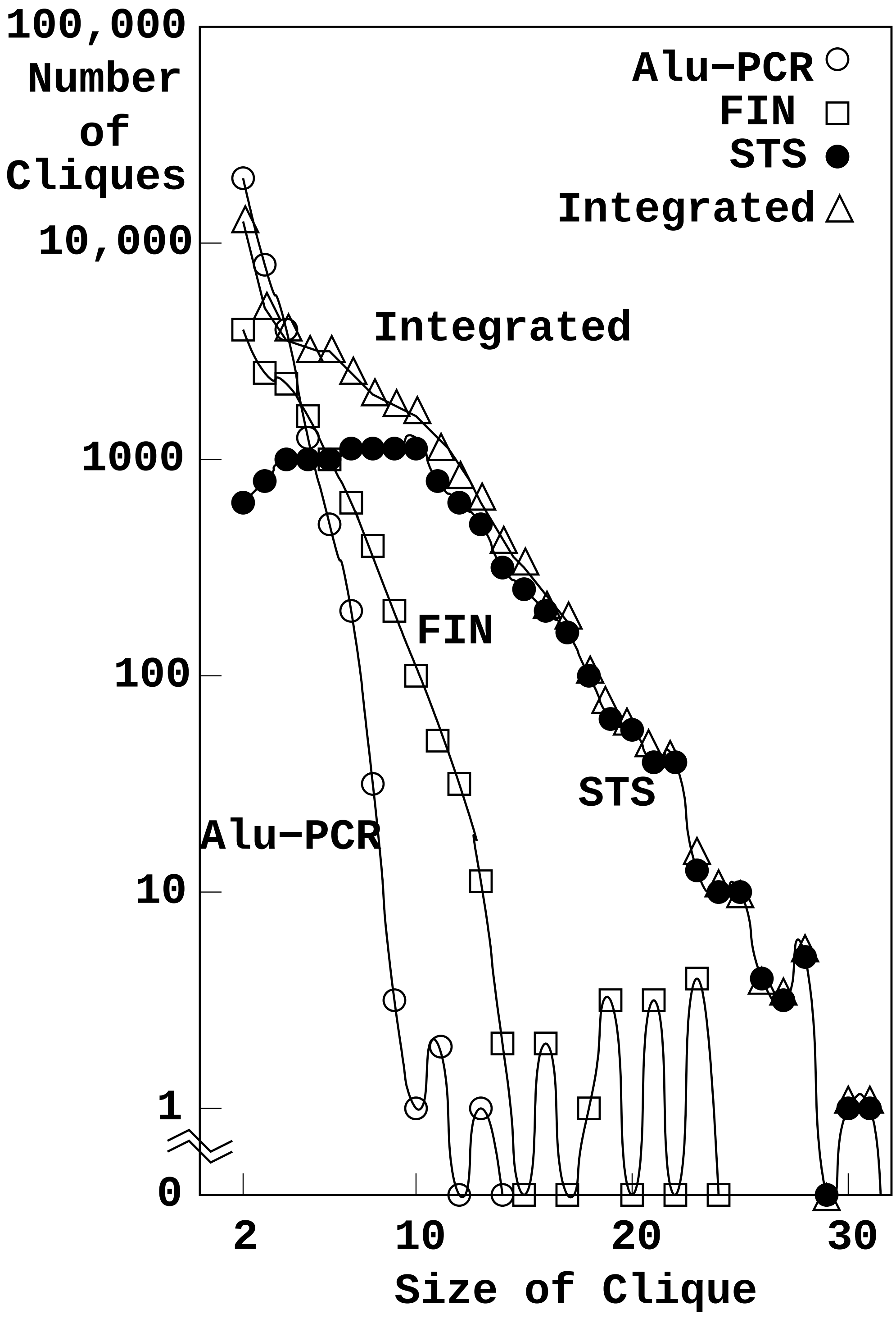


Figure 6.3: Size distribution of cliques.

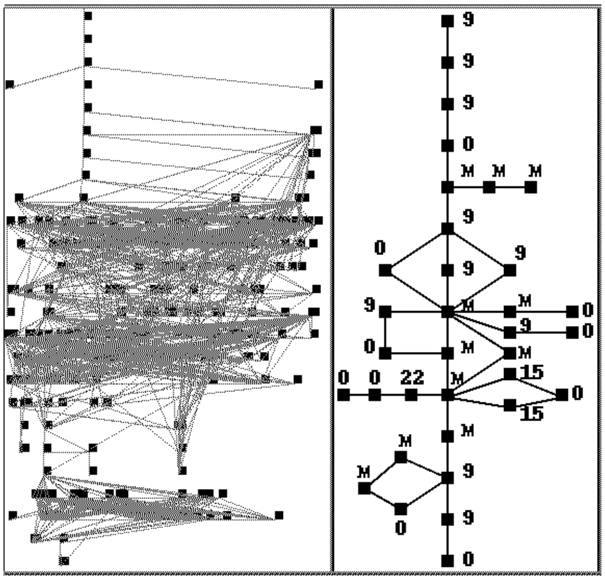


Figure 6.4: Alu-PCR virtual probes: *(left)* Double-linkage intersection graph (238 nodes, 1,805 edges); *(right)* structure graph (35 nodes, 40 edges)

STS data. One huge connected component comprised 96% of the nodes, an anomaly common in STS data at this level of analysis (i.e., single-linkage). This clearly indicates cross-linkage among virtual probes belonging to different chromosomes. Simple counting reveals that 19% of the unit-weight edges connect virtual probes assigned to different chromosomes (considering only edges between virtual probes that were successfully assigned to chromosomes). This rate of two-chromosome edges in single-linkage data is much less than the 69% shown for STS probe data in Figure 3.7. Only about 1.2% of the edges with weight $M > 1$ are cross-links of this type, which leads us to examine only the double-linkage intersection graph.

The largest component of the double-linkage clique-intersection graph is shown in Figure 6.4, along with the corresponding trimmed structure graph. The nodes in the structure graph are labeled using the chromosome-assignment rule applied to blobs. Nodes which are not assigned a chromosome because they are equally associated with multiple chromosomes are labeled ‘M’, while blobs that do not have at least two clones associated with a single chromosome are labeled ‘0’. The labeled structure graph suggests that this component is a contig from Chromosome 9, along with some cross links to parts of chromosomes 15 and 22. I do not analyze this component further, since the point here is only to show that the virtual probe technique makes it possible to examine overlap data in new ways. The right panel of Figure 6.4 reveals in a clear and intuitive way structural information about the possible contigs in this component. Attempting to glean similar information from the left panel by conventional means is difficult, which is why such components were often mistaken for contigs.

6.3.2 Fingerprint Data

In this section, I use virtual probes to form a new view of the fingerprint data from CEPH/Généthon. I first extracted all of the maximal cliques from the overlap graph with the modified BK algorithm. The total number of maximal cliques was 12,710, including 3,718 cliques of size two which were discarded as potential noise. Using the chromosome-assignment rule, 93% of the remaining cliques were unambiguously assigned to chromosomes. An intersection graph based on cliques from fingerprint data was formed, without first isolating data for a particular chromosome. The largest component comprised 66% of the nodes. This indicates some cross-linkage, and in fact 14% of the unit-weight edges link virtual probes that belong to different chromosomes. This rate is similar to that found in Alu-PCR data (19%), and again much less than the rate of 69% found in STS-probe data. Only 0.8% of the edges with weight $M > 1$ are cross-links, leading us as usual to use the double-linkage filter. Note that without virtual probes, this filter is not available when analyzing overlap data. Figure 6.5 shows the largest component of the double-linkage intersection graph and the corresponding structure graph. Nodes in the structure graph are labeled using the chromosome-assignment rule applied to blobs. This component appears to be a fairly clean and simple contig from Chromosome 4.

Thus, the virtual probe technique makes possible simple noise filtration and analysis of fingerprint and Alu-PCR overlap data by methods developed for probe data. Noise filtration results from the corroboration involved in forming the probes and from the ability to use double-linkage analysis on the resulting probes.

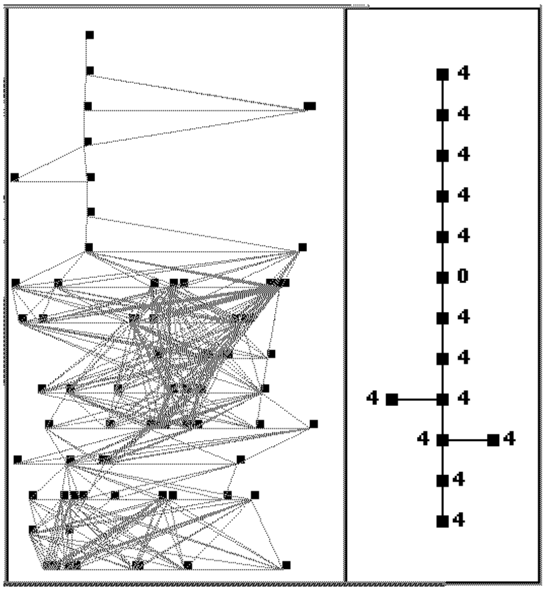


Figure 6.5: Fingerprint virtual probes: *(left)* Double-linkage intersection graph (82 nodes, 592 edges); *(right)* structure graph (15 nodes, 14 edges)

6.3.3 Merged Data

In this section I use virtual probes to integrate STS, Alu-PCR and fingerprint data. The STS data were first converted to overlap data by assuming that the clones hit by a particular STS mutually overlap, and then I combined the STS overlap data with the 'pure' overlap data. The total number of maximal cliques generated using our modified BK algorithm on the integrated overlap data was 40,060, including 12,378 of size two that were discarded as potential noise. Figure 6.3 shows that the number of cliques from the merged data decreases rapidly with clique size, though not as quickly as in the case of cliques from Alu-PCR or fingerprint data. Cliques of size twelve or more are primarily based on STS data. Application of the chromosome-assignment rule to cliques of size three or more resulted in 95% of the cliques being unambiguously assigned. The clique

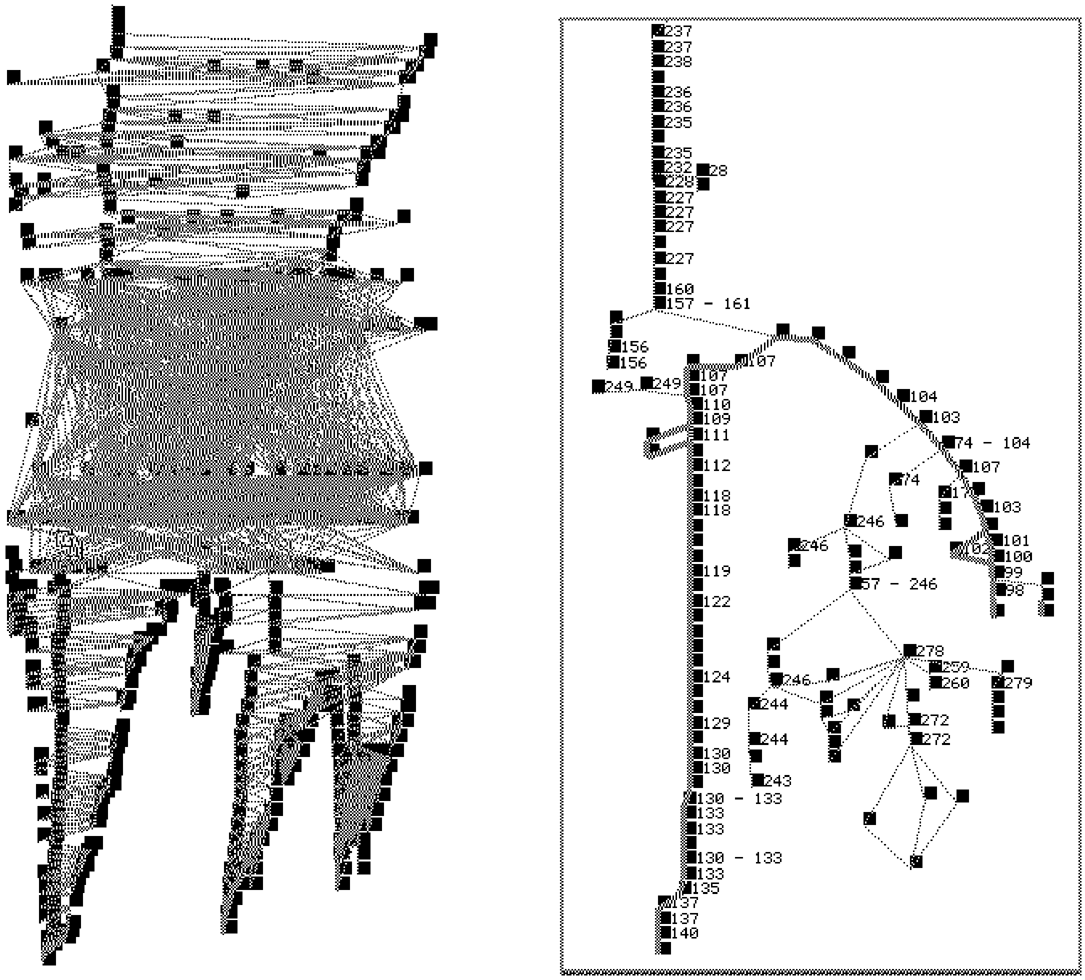


Figure 6.6: Virtual probes for integrated data from Chromosome 1: *(left)* Double-linkage intersection graph (1,196 nodes, 5,839 edges); *(right)* structure graph labeled with genetic positions in cM (134 nodes, 165 edges).

intersection graph is largely one component containing 99.5% of the nodes, indicating abundant cross-linkage. In fact, 58% of the unit-weight edges cross-link virtual probes from different chromosomes, while 2.6% of the edges with weight $M > 1$ are cross-links.

Removing the unit-weight edges, I obtained a double-linkage clique intersection graph for the genome-wide data, but even this graph is almost entirely one component, comprising 24,304 of the 27,692 nodes. This reflects considerable cross linkage for edges with weight $M \geq 2$, a phenomenon we also see in STS data alone (Chapters 3 and 4, [45]).

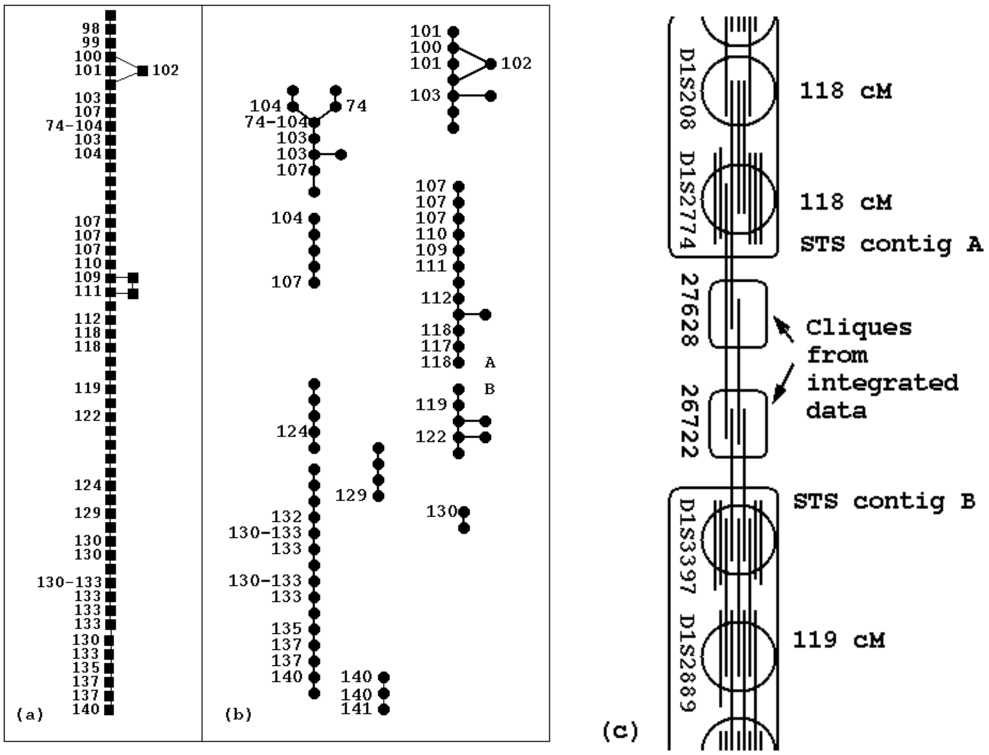


Figure 6.7: *(a)* Proposed contig extracted from integrated data. *(b)* Corresponding STS contigs. *(c)* Sketch illustrating how cliques from integrated data link STS contigs A and B of panel (b).

This connected component can be broken up according to chromosome by assigning virtual probes to chromosomes and then filtering out edges which link probes belonging to different chromosomes. This results in a number of connected components for each chromosome. Figure 6.6 shows the largest component formed from virtual probes assigned to Chromosome 1 or unassigned. The corresponding structure graph (*right*) is branched and looped, indicating that this component cannot be a single contig.

I have labeled nodes in the structure graph with genetic positions in order to help visualize where the contigs lie in this component. Labeling with radiation hybrid positions produces similar results (not shown). A blob is given a range of positions as determined by the STSs that it contains. A blob is considered to contain an STS if the set of clones in the blob is a superset of the set of clones hit by the STS. The longest potential contig stretches from about 98 cM to 140 cM. This contig is shown in Figure 6.7(a). Figure 6.7(b) shows that many double-linkage contigs in STS data alone are required to cover this range of genetic positions. Similarly, the WI/MIT STS-based map of the human genome shows about a dozen double-linkage contigs in the range 98-140 cM (http://carbon.wi.mit.edu:8000/cgi-bin/contig/phys_map). In the region 80-140 cM of Chromosome 1, the *double-linkage* coverage in the integrated data is similar in extent to that of the *single-linkage* STS contigs in the WI/MIT map. Thus, data integration using virtual probes leads to longer double-linkage contigs.

As a direct illustration of how cliques from integrated data can link separate STS double-linkage contigs, I show in Figure 6.7(c) the details of the linkage between the STS contigs at 118 cM and 119 cM, labeled A and B, respectively in Figure 6.7(b). The diagram shows that two cliques of integrated data (in this case mostly fingerprint data and some STS data) serve to link the STSs in the two contigs, creating a longer double-linkage contig. This illustrates that contigs are automatically extended as a consequence of our general approach, without introducing separate algorithms or rules for this purpose.

In Figure 6.8, pairs of parallel line segments are used to represent schematically double-linkage contigs covering a range of genetic positions given by the scale on the side of the box. This illustrates the correspondence between two of the double-linkage contigs from integrated data compared to the numerous STS double-linkage contigs WI/MIT have for the same region. Figures 6.7 and 6.8 illustrate that the uniform integration of STS, fingerprint and Alu-PCR data afforded by the virtual probe technique results in much more extensive double-linkage coverage. The unpaired line segments in Figure 6.8(c) represent single-linkage contigs in the WI/MIT map. The figure shows that in the region 80-140 cM of Chromosome 1, the *double-linkage* coverage in the integrated data is similar in extent to that of the *single-linkage* STS contigs in the WI/MIT map. In this type of noisy data, double-linkage coverage is much more reliable than single-linkage coverage. In fact, single-linkage components by themselves are useless. Therefore, there

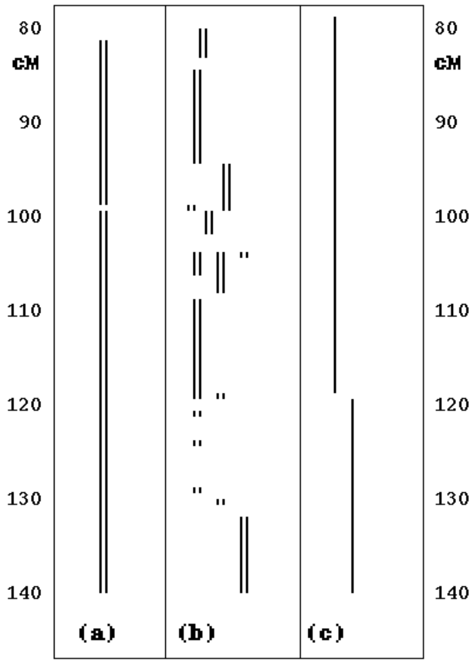


Figure 6.8: *(a)* Sketch of coverage in cM of two contigs from integrated data. *(b)* Corresponding double-linkage STS contigs in WI/MIT map. *(c)* Corresponding single-linkage STS contigs in WI/MIT map.

is a clear benefit in integrating overlap data with probe data using virtual probes.

6.4 Discussion

I have presented a new method to analyze and integrate overlap data with probe data. The method is based on the conversion of overlap data to maximal cliques. The cliques correspond to sites in the genome, and are therefore called virtual probes. Analysis of the overlap graph in terms of virtual probes offers the following advantages — (1) it provides a simple way to filter out many false positives: simply ignore cliques of size two and construct double-linkage intersection graphs; (2) virtual probes make possible a seamless integration of numerous forms of overlap and probe data; and (3) systems designed for analysis of probe data can be applied without modification to overlap data transformed to virtual probes. The MBK algorithm of Chapter 5 is effective and efficient for this type of data.

I applied the method to Alu-PCR and fingerprint overlap data and used the probe-analysis technique of structure graphs to produce some sample results. This illustrated two points: (*i*) that virtual probes from typical overlap data produce double-linkage components not unlike those produced in STS data, and (*ii*) that methods developed for analysis of probe data can be applied to overlap data when it is represented as virtual probes. The method was applied to integrated data (Alu-PCR, fingerprint and STS data together), and I showed that the resulting double-linkage contigs were much more extensive than those obtained using STS data alone. Extraction of these contigs from complex graphical components is possible with additional positional evidence such as genetic or radiation hybrid markers. Once contigs are extracted, an ordering algorithm could be applied to the virtual probes, although this is not done here. The resulting order should be more accurate when using virtual probes of integrated data than when using STS data alone, since more information is associated with each probe. The method of virtual probes may also be useful in sequence assembly where the problem of forming contigs based on overlap data recurs.

Chapter 7

Clusters

7.1 Introduction

The problem of grouping data points into clusters is one that is encountered in many disciplines, and there are many algorithms for this purpose. Mirkin [63] and Corneil and Woodward [19] provide reviews of approaches to and applications of clustering. Intuitively a cluster is a group of data items that are similar or which belong together. In practice, the definition of a cluster is embodied in the algorithm which is used to create the cluster. Clustering algorithms are generally divided into two major classes: hierarchical and nonhierarchical.

Hierarchical clustering methods are similar to the grouping of ideas in a document. The basic data elements form one-element clusters at the lowest level of the hierarchy. In the case of the document, the ideas would be the basic data elements. At the next level, related data elements are grouped into larger clusters, just as ideas are grouped into clusters called subsections. More inclusive groupings occur as one progresses up the hierarchy, until a single cluster contains all of the data elements. In the example, the clusters at higher levels would be sections, then chapters, and finally a single cluster equal to the document itself. Hierarchical clustering methods can be divided into two classes: divisive and agglomerative. Divisive hierarchical methods have a top-down strategy for forming clusters. They start with an all-inclusive cluster which is then divided into smaller clusters at the next level of the hierarchy. Then these clusters are subdivided further, creating a tree of clusters, where the leaves are the individual data items. Agglomerative hierarchical clustering methods form clusters with a bottom-up strategy. These are more common than the divisive stategies. They start with clusters that each contain one data point, and then use rules to form larger clusters by uniting clusters at one level into clusters at the next level.

Nonhierarchical clustering methods create clusters which all lie on a single level, i.e., there is no nested structuring of the clusters. For example, finding maximal cliques in a graph is a nonhierarchical method of clustering the nodes. Nonhierarchical clustering algorithms can be further divided into methods which produce potentially overlapping clusters and those which produce nonoverlapping clusters. Maximal cliques are potentially overlapping clusters. Similarly, a method which clusters words according to their meaning could produce overlapping clusters, since some words have multiple meanings.

In this chapter, I introduce a nonhierarchical, nonoverlapping, clustering algorithm which may be useful in some applications of the virtual probe technique discussed in Chapter 6. The clustering algorithm is designed to counter the proliferation of maximal cliques in the presence of false negatives. Figure 7.1 illustrates this problem. In the following discussion, it convenient to use the standard notation for a complete graph on n vertices: K_n.

This sketch shows that ten mutually overlapping clones produce an overlap graph which contains a single clique K_{10} if all of the overlaps are observed. However, if some overlaps are not observed, then there are multiple cliques, i.e., the overlap graph will contain more than one clique. In particular, if on average 20 percent of the overlaps are not observed, then the overlap graph contains about 9 cliques on average. For clarity, let us refer to the cliques that result from false negatives as *daughter* cliques, since they

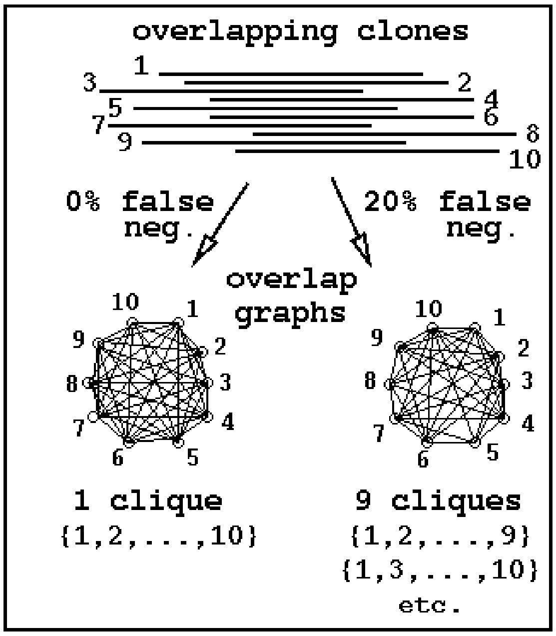


Figure 7.1: Proliferation of cliques caused by false negatives.

are in effect multiple daughters of one single parent clique. I specify an average number of daughter cliques (as opposed to the the exact number), since the actual number of daughters depends on which edges are omitted from the original parent K_n graph. Edges are omitted based on calls to a pseudorandom number generator which generates random numbers in the range $[0, 1)$, i.e., from 0 to 1, including 0 but not 1. Each edge is considered in turn, and an edge is omitted if the next random number is less than f, the false negative rate (for example, $f = .20$ for a 20% false negative rate).

Figure 7.2 shows the average number of daughter cliques resulting for other rates of false negatives applied to an initial complete graph of 10 nodes (parent K_{10}). The error bars indicate standard deviations calculated after 10 trials. In this figure, I count only daughter cliques with at least three nodes, since according to the method described in Chapter 6, virtual probes must correspond to at least three clones. The figure shows that for K_{10} the greatest proliferation of daughters occurs in the range 20 to 40 percent false negatives.

As one might expect when dealing with cliques, the problem of daughter cliques gets

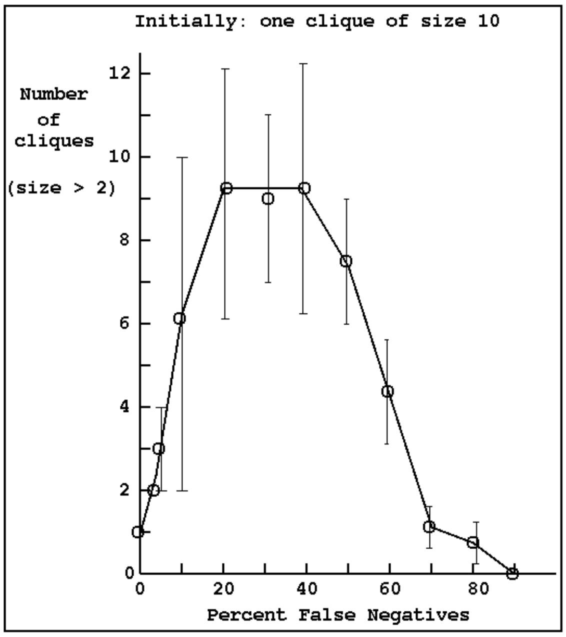


Figure 7.2: Proliferation of cliques vs. rate of false negatives.

exponentially worse as the size of the parent clique increases. For example, given a rate of 20 percent false negatives, K_{20} has 78 daughters on average, while K_{40} has 3000 daughters on average. Figure 7.3 shows this result graphically for several values of K_n. The figure shows the average number of daughters as a function of n, given a 20% false negative rate.

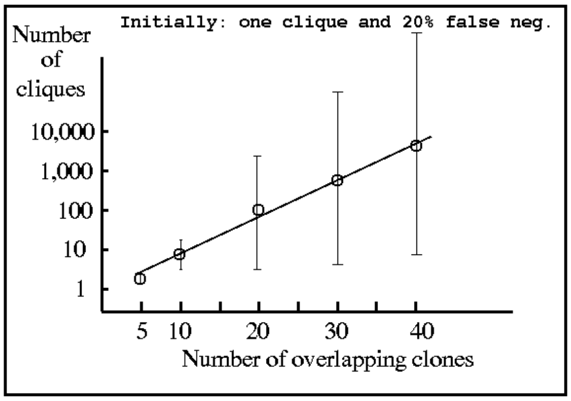


Figure 7.3: Exponential growth in cliques caused by false negatives.

7.2 Algorithm

It is possible that at certain edge densities, the addition of false positives might tend to increase the number of cliques. I have not observed this effect nor studied the possibility, but it is easy to see cases where false negatives give rise to many daughter cliques.

WI/MIT reports that the STS/YAC physical mapping data can have up to 40% false negatives [47]. However, it is worth noting that the daughter cliques correspond to valid sites in the genome, just as virtual probes do. The problem is that they are redundant and there are more of them than needed or wanted. If there are a great many daughter cliques, then their number could reduce the efficiency of subsequent steps in the analysis of the overlap graph. Therefore, in this section I develop a clustering algorithm which helps to counter this problem.

Algorithm Form Clusters

Input: $G = (V, E)$

Output: Set of clusters of G.

Step 1: Let S be the set of all pairs (u, v) such that $u, v \in V$ and $(u, v) \notin E$ but $\exists x$ such that $(u, x), (x, v) \in E$.

Step 2: Order S according to the number of paths of length two between u and v for each pair (u, v), highest to lowest.

Step 3: For each $(u, v) \in S$, taken in order, determine if adding edge (u, v) to the graph decreases the number of maximal cliques. If so, add the edge, i.e., $E = E \cup \{(u, v)\}$. Stop when S is exhausted or when 500 elements in a row fail to decrease the number of cliques.

Step 4: Iterate Step 3 three times.

Step 5: Return the maximal cliques of the resulting modified graph.

Figure 7.4: Detailed cluster algorithm

The algorithm in effect tries to reconstruct the parent clique. It forms a group of nodes called a cluster, which in the ideal case would contain the same nodes as the parent clique. The group is called a cluster rather than a clique, since its graph is not necessarily complete. Missing edges are presumed to be false negatives. To form these clusters, the algorithm considers pairs of nodes which are not connected by a single edge, but which are no more than two edges apart. The presence of one or more paths of length two between the nodes suggests that they are at least close to each other. I assume that the more paths of length two between the nodes, the more likely the two nodes should be connected by an edge. Considering the most likely pairs first, I add edges where the addition of the edge decreases the number of cliques. When this step is done, I define the clusters to be the maximal cliques in the modified graph. The details of the cluster algorithm are shown in Figure 7.4.

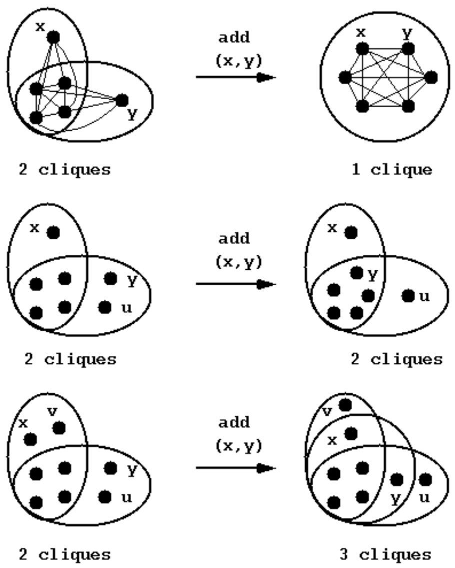


Figure 7.5: The number of cliques may increase, decrease or stay the same on adding edge (x,y). Ellipses enclose maximal cliques.

The key step of the cluster algorithm is to compare the number of maximal cliques in a graph $G = (V, E)$ with the number in the graph $G' = (V, E \cup (u, v))$. The algorithm would be impractical if this comparison required counting all the cliques in G and all the cliques in G'. Luckily, we only need to count the number of cliques in a small subgraph of G (and G') defined by the set of nodes $u, v \cup N(u) \cup N(v)$, where $N(u)$ is the set of nodes adjacent to u. The reason for this is that addition of the edge (u, v) does not affect cliques which do not include either u or v.

The addition of an edge can have three possible effects: the number of cliques may increase, decrease or stay the same. Figure 7.5 illustrates this statement by example, in schematic form. Consider nodes x and y, which initially are not directly connected, but which are connected by a path of length two. Assume that the set of nodes shown comprises all of the nodes that share an edge with either x or y. An ellipse is drawn around a group of nodes which form a maximal clique. In the top panel, addition of the edge (x, y) causes two maximal cliques to become one maximal clique, so this is an instance of decreasing the number of maximal cliques. In the middle and bottom panels I omit the edges from the graph for clarity, since the ellipses already denote maximal complete subgraphs. In the middle panel, the addition of edge (x, y) causes y to belong to both of the existing maximal cliques, without changing the number of maximal cliques. The lower panel shows an example of a case where adding the edge (x, y) causes the formation of a new maximal clique, while leaving the initial two maximal cliques intact. Thus, this is an example of increasing the number of maximal cliques by adding an edge.

7.3 Testing

To test the efficacy of the clustering algorithm, I used simulated overlap data. Description of the simulator can be found in Chapter 5. In these experiments, I use simulated chromosomes of length approximately 50 Mb. Then clones of average length 1 Mb were generated. The number of clones generated is determined by the parameter called coverage. The coverage is defined as the total length of clones generated divided by the length of the chromosome. Thus, if n clones of length 1 Mb are generated for a chromosome of length 50 Mb, then the coverage is $n/50$. Intuitively, the coverage is the average tiling depth of the clones, if they are laid out on the chromosome at their sites of origin. In effect, the coverage is very close in value to the average size of maximal cliques in the resulting overlap graph. This is clear from the definition of a virtual probe as a maximal set of overlapping clones — the number of overlapping clones will in general be fairly close to the coverage. Since the problem of daughter cliques is very sensitive to the size of the parent cliques, I vary the coverage while testing the clustering algorithm.

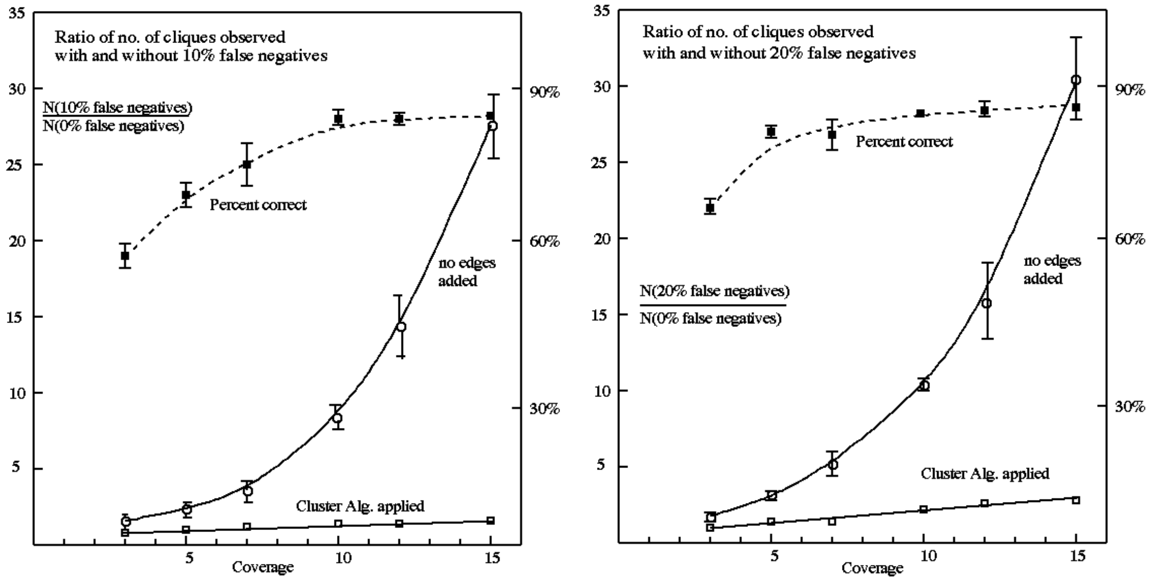


Figure 7.6: Application of the cluster algorithm to simulated data.

The results of these experiments are shown in Figure 7.6. I vary coverage from 3 to 15. The panel on the left represents results of experiments when the false negative rate is 10%, and the panel on the right has false negative rate equal to 20%. Each data point shows the average and standard deviation of 4 experiments. The solid curve with circles as data points shows the average number of maximal cliques in the presence of false negatives (either 10% or 20%) divided by the average number of maximal cliques when there are no false negatives. For ease of reference, let this ratio be called the daughter ratio. The curve through the data points rises from about 2 to about 30 as the coverage is increased from 3 to 15, because of the increase in the number of daughter cliques as the size of the parent cliques is increased. The lower curve with squares representing data points shows the daughter ratio after applying the clustering algorithm. The clustering algorithm clearly controls daughter ratio keeping it near 1 or 2.

In effect, the clustering algorithm redefines a virtual probe as a cluster of apparently overlapping clones. The definition of *cluster* is embodied in the algorithm of Figure 7.4. The algorithm considers pairs of nodes which are not linked by an edge, but which apparently correspond to clones which are close, since they are linked by a path of length two. The algorithm then adds the edge to the graph if doing so will reduce the number of cliques. When it completes the job of considering edges for addition, then it calls the maximal clique algorithm to list all of the cliques. Each clique is called a cluster, since it may include an edge or two which was not observed in the genomic experiments. Clearly it is important to know if the edges that are added are the correct ones, since it would be of little value to control the daughter ratio by adding edges that do not belong. One advantage of using simulated data is that we know which edges were removed during the application of the false negative rate. Thus, the filled circles and the dotted curves show the percent of correct guesses made by the clustering algorithm. For low coverage values, the algorithm is correct in its edge additions only about 60 or 70% of the time. This is to be expected, since the sparseness of the edges leaves little information to guide the guesses. However, for the higher coverage experiments, the algorithm is correct close to 90% of the time.

7.4 Summary

A potential problem with the virtual probe method as described in Chapter 6 is the production of daughter cliques caused by false negatives. Despite a relatively high false negative rate in the genomic data, I had no trouble applying the virtual probe method to analyze the data in Chapter 6. However, it is possible and probable that in experiments with higher coverage, daughter cliques could result in many more virtual probes than should be observed.

I have presented in this chapter a clustering algorithm which tends to reverse the generation of daughter cliques and reform the parent clique. The idea is simple — if removing certain edges causes a proliferation of daughter cliques, then these edges should be detectable by the fact that replacing them reduces the number of cliques. The algorithm is heuristic. It involves first finding all of the pairs of nodes which are not joined by an edge but which are joined by a path of length two. These node-pairs are ordered by the number of two-step paths joining them. I consider the most heavily joined pairs first, and add edges to the graph if so doing decreases the number of cliques. I stop examining node-pairs when some number of them fail to provide edges to add to the graph. In this experiment that number is arbitrarily set to 500. If this parameter were smaller or larger by a factor of 10, it would probably make little difference, but I did not experiment much with it. The whole process is iterated three times, since after each addition of edges there may be new node-pairs which could decrease the number of cliques.

I tested the method on simulated data, since only with simulated data can one determine if the algorithm is adding the correct edges. The results indicate that the algorithm does very well, since at coverages above 10 it is correct almost 90% of the time, and it effectively suppresses the generation of daughter cliques. Comparison of the method designed here with published methods of clustering should be done in future studies.

Chapter 8

Conclusion

This dissertation is submitted in the Department of Computer Science, but its focus lies in the intersection of computer science and genomics. Thus, as mentioned in the Introduction, it is really a dissertation in bioinformatics. Such works may become increasingly frequent in the future, since the use of computers permeates modern biology. There are databases for proteins, genes, genome sequences, expressed sequence tags (ESTs), single nucleotide polymorphisms (SNPs), etc. There are so many databases that there is a database of databases. There is software for sequence alignment, protein structure analysis, drug discovery, physical mapping, gene discovery, combinatoric analysis of various experiments, etc. Roos [78] points out that it may not always be clear to biologists what topics or applications in their field would be considered as research in computer science. On the other hand, Karp [51] suggests that there are a great many such topics. The area of physical mapping, addressed in this dissertation, is one of them. Confirmation of this statement lies in the fact that this research makes contributions to both genomics and computer science.

8.1 Contributions

The primary contributions to genomics are the structure algorithm and method for integrating probe data and overlap data. The structure algorithm transforms mapping data to a graph which is easily visualized and whose apparent properties indicate whether the data corresponds to a contig. I proved that perfect data from a contig produce a simple path structure graph, and I showed that minor imperfections such false negatives or uncorroborated false positives have little effect on the resulting shape. Thus, the key feature of the structure graph is whether or not it forms a simple path. Simple paths may correspond to contigs, while branching or looping paths almost certainly do not. I have shown that the paths in a complex structure graph can be used in conjunction with other mapping information to extract data corresponding to a contig. I also provided subsidiary algorithms for removing short branches and cycles from the structure graph, but these algorithms are not necessary for effective use of the structure graph.

The structure algorithm is a simple test that reveals important information about the mapping data which is not readily obtainable by other means. The information indicates which part of the data may correspond to clones lying in a contiguous region of the genome, and which parts of the data are problematic. The theoretical formalization makes it easier to express the steps taken in mapping — for example, 'reasonable paths' become 'straight paths', where straight has a precise definition. Second, the choice of a correct path in a branching structure graph cannot always be based on genetic or radiation hybrid data, since this positional information is absent from some paths and occasionally ambiguous. It may be that the choice of path at a junction of branches can sometimes be based on local attributes of the nodes or edges near the junction. The formalism presented here facilitates the study of this possibility. Finally, the value of a simple visual test of goodness or badness in mapping data is obvious — it is analogous to the value of a circuit tester in that bad or questionable components can be quickly and simply identified.

The virtual probe method allows seamless integration of overlap and probe data as well as comparison of the two data types on common grounds. Integration based on virtual probes leads to more extensive double-linkage graphs. One can see this effect in that some of the double-linkage contigs created here contain two to four double-linkage contigs produced by standard methods. Whatever method is used to order nodes of a contig, the more overlap information associated with the nodes the better the final order will be. Thus, the use of virtual probes representing integrated data rather than one type of data in isolation should lead to a more accurate map. Virtual probes also provide additional means to filter false positives from the data: ignore cliques of size two and construct double-linkage intersection graphs. Software systems designed for analysis of probe data cannot be applied without modification to overlap data by representing the overlap data as virtual probes corresponding to nodes in an intersection graph. The experimental results provide new views of Alu-PCR data and fingerprint data, and show the results of integrating several types of data from major genome centers. The substitution of virtual probes of integrated overlap data for STSs leads to more extensive double-linkage graphs. This is apparent in that some of the double-linkage contigs produced by this method contain two to four Whitehead double-linkage contigs. Virtual probe and structure analysis together allow better use of data, which may decrease the amount of data required, thereby decreasing cost.

One of the contributions to computer science presented in this dissertation is an improvement on a well-known clique enumeration algorithm. My modification of the Bron-Kerbosch algorithm [10] greatly improves its efficiency on large sparse graphs. For example, given a genomic overlap graph of 25,000 nodes and coverage of five, the modified version of the BK algorithm (MBK(list)) was over 100 times faster than another published modification (SBK), and over 200 times faster than the original BK algorithm modified to use an adjacency list, BK(list). Three other algorithms published more recently than BK, i.e., TIAS, CN, and LTMIS, were not at all practical on this size of graph. Even larger factors of improvement could be elicited by increasing the size of the graph, since the time taken by the MBK algorithm seems to increase linearly with n, while other tested algorithms appear to be at least quadratic in the size n of the input graph. MBK was also much more efficient that other algorithms on large, sparse random graphs.

A second contribution to computer science is a new clustering algorithm. This clustering algorithm is designed to address the problem of daughter cliques caused by false negatives. The new clustering algorithm appears to deal very effectively with the problem for which it is designed. It suppresses the generation of daughter cliques caused by the presence of false negatives in the data, and it forms clusters which approximate well the original virtual probes. Before designing a new clustering algorithm, I tried two published algorithms [37, 95] with unsatisfactory results. However, the testing of published clustering algorithms was not extensive, and therefore these experiments are not shown here.

A third contribution to computer science is an algorithm which reveals interval graph structure in noisy interval graphs. Its use in physical mapping is demonstrated here, but it may have other applications.

Thus, the dissertation offers new results to both fields, computer science and physical mapping.

8.2 Future Work

There are a number of areas in which the work in this dissertation could be extended.

- *EST clustering:* Expressed sequence tags (ESTs) are short sequences obtained from mRNA. Huge sets of these sequences (e.g., 80,000 or more ESTs) are analyzed for sequence overlap, in an attempt to group ESTs into clusters of mutually overlapping ESTs representing genes. The clique, cluster and structure graph algorithms may find application in this area.

- *Ordering algorithms:* When I began this research, we proposed a three phase approach to map assembly: overlap, linkage and ordering [44]. The overlap phase corresponded to the virtual probe analysis; the linkage phase corresponded to the structure graph analysis. The ordering phase is where probes and clones are ordered and a physical map is generated for each contig identified in the linkage phase. This entails finding one or more orders for the probes or clones which best fit the data. In fact there will be many orders which cannot be distinguished because there will be sets of probes which hit the very same YACs. Moreover, given the noise in the data, there will be probes whose position will be indistinguishable even though they hit slightly different sets of YACs.

  I developed an algorithm for ordering probes based on a probabilistic approach which addressed the following problem. Given the STS-YAC interaction data for two probes, A and B, how does one decide whether or not to reject the following *null hypothesis*:

  H_0 : STS A and STS B each lie on the same N YACs.

  If the null hypothesis cannot be rejected at some accepted level of confidence, then A and B could be grouped together as opposed to linearly ordered, to show that we are not sure that STS A and STS B are actually on different YACs. In the extreme, probes A and B may hit exactly the same YACs, in which case H_0 cannot be rejected at any level of confidence.

  This algorithm should be tested extensively on simulated data, and it should be compared with several algorithms which are currently in use. The map I propose to generate will be a sequence of bins. Each bin will contain probes whose order cannot be decided given the data and the specified level of confidence. A bin is like a Booth-Leuker P-node [7], except that a probe may appear in more than one bin. In this way, I hope to represent the uncertainty in the map, and the multiplicity of equivalent solutions, thus satisfying Karp's and Letovsky's concerns mentioned in Chapter 2.

It would also be interesting to develop an ordering algorithm based on the algorithm of Corneil *et al.* [16] for producing a natural labeling of a unit interval graph. The ideal STS-intersection graph is a unit interval graph, but real STS-intersection graphs are quite noisy. It is unknown what modifications of the natural labeling algorithm would be necessary to produce reasonable orderings of STSs in real data. Another question to ask in this context would be how can one generate all of the orderings which are consistent with the data to some specified degree. There might be too many consistent orderings to be useful, but there might also be a concise representation.

- *Clustering application:* The clustering algorithm presented here is quite unusual with regard to its solution to the false negative problem. It would be of interest to systematically test other clustering algorithms on this application and to test our clustering algorithm on other applications, in order to compare efficacy.

- *Comparison of algorithms:* As mentioned in Chapter 5, there are not many published empirical comparisons of enumeration algorithms for cliques or MISs. Chapter 5 is a step in this direction, but there are a few more algorithms I would like to add to the collection. I would also like to broaden the range of random graphs tested, and also include some special graph types. It would then be of interest to compare the empirical results with the theoretical complexities of the algorithms.

- *Graph Queries and Visualization:* Our initial interaction with WI/MIT resulted in the development of a number of Hy+ queries for analyzing genome data [43]. WI/MIT was sufficiently interested in this line of research that they obtained the Hy+ system from Dr. Mendelzon, and invited Dr. Bonner and this author to install it and demonstrate the queries. I became distracted from this line of research when

I ran into problems with contig identification (one of the main topics of this thesis). It would be of interest to pursue this avenue of research again and to see what new applications in genomics would be suitable for the power and flexibility of Hy+ queries and visualization.

Appendix A

Simulator

In this appendix we briefly describe the simulator package which was used in this work to simulate genomic data. Dr. Lincoln Stein (then at WI/MIT), kindly designed and initiated the simulator and gave it to this author to complete and extend. The simulator is written in Perl as a number of components which are intended to run as a pipeline, the output of one component being the input of another. The components are briefly described below.

boulderio.pl implements Dr. Stein's 'boulder' I/O format which is used for passing data along the pipe. To read about Stein's recent work on the 'Boulder Data Interchange Format' see *http://stein.cshl.org/software/boulder/*.

gasdev.pl implements a normally distributed random number generator.

chromosome.pl generates a set of chromosomes of different lengths.

repeats.pl adds repeated regions to the chromosomes.

unstable.pl adds unstable regions to the chromosomes.

unclonable.pl adds unclonable regions to the chromosomes.

fingerprints.pl adds fingerprint probes to the chromosomes.

genetic_sts.pl adds genetic markers to the chromosomes.

sts.pl adds STS probes to the chromosomes.

yacs.pl consolidates chromosome information and generates perfect YACs.

chimera.pl combines YACs to form chimeras and randomizes order.

Each of the above routines takes parameters which default to values thought to be reasonable for the human genome. Usage information can be seen by giving the parameter -h. The following command uses the default parameters to generate STS- and fingerprint-annotated YACs for a genome the size of the human genome.

```
chromosome.pl | repeats.pl | unstable.pl | unclonable.pl | fingerprints.pl | sts.pl
| yacs.pl | chimera.pl > temp.yacs
```

The defaults in the above command are equivalent to the following:

chromosome.pl -n24 -m200 -s70 i.e., 24 chromosomes, mean size 200 Mb ± 70 (standard deviation). The output looks like:

```
CHROMOSOME=C01
CLENGTH=174.9
=
CHROMOSOME=C02
etc.
```

repeats.pl -f.0001 -m3000 -s500 -k1 i.e., repetitive DNA/total DNA: .0001, mean size of repeats: 3000 bp ± 500 bp; kinds: 1. (The defaults here are not meant to represent the human genome which may have 50% repetitive sequences. I set a low frequency of repeats to observe the effect on STS data. An STS which binds to a high frequency repeat would be discarded.) The output looks like:

```
CHROMOSOME=C01
CLENGTH=174.9
REPETITIVE=8.1196-8.1222-1
REPETITIVE=12.7541-12.7574-1
etc.
```

which means that a type-1 repeat region occurs at the indicated coordinates.

unstable.pl -f.001 -m.02 -s.008 i.e., 0.1% of the genome is unstable; mean size of unstable regions is 20 kb ± 8. This adds annotations to the chromosome of the form

UNSTABLE=48.766-48.781
UNSTABLE=67.935-67.944
UNSTABLE=77.209-77.234
etc.

which indicate the start and stop positions of the unstable regions.

unclonable.pl -f.001 -m.02 -s.008 i.e., 0.1% of the genome is unclonable; mean size of unclonable regions is 20 kb ± 8. This component adds lines like

UNCLONABLE=3.135-3.161
UNCLONABLE=12.030-12.055
UNCLONABLE=45.214-45.222
etc.

which indicate the start and stop positions of the unclonable regions.

fingerprints.pl -m10 -s5 -f10 -g2 i.e., mean size of fingerprints is 10 kb ± 5, and number of fingerprints per Mb is 10 ± 2. This adds lines of the following type which indicate the location of a probe (in Mb) and the length of the probe (in kb):

PROBE=0.051:8.8
PROBE=0.140:12.3
PROBE=0.165:15.8
etc.

genetic_sts.pl -d10 -e1 i.e., there is a genetic marker every 10 Mb on average. The standard deviation of the difference between the position in centimorgans and real position is 1. Note: the real position is in Mb from one end of chromosome, and therefore, it is numerically similar to position in centimorgans. This component adds lines of the following type which indicate a genetically mapped STS is at <real position>:<chromosome><genetic position>

STS=6.931:C01cM7
STS=16.125:C01cM17
STS=20.514:C01cM19
etc.

sts.pl -f2 -g.3 i.e., 2 STSs ± .3 per Mb. This component produces lines like:

STS=1.685:STS00006
STS=1.729:STS00007
STS=2.216:STS00008
etc.

which are of the form STS=<position>:<identification number>.

yacs.pl -f6 -m1.0 -s.2 -d.2 i.e., 6 YACs per Mb with mean size 1.0 Mb ± 0.2 and deletion probability of 0.2. (If a YAC has an unstable area, the probability is 0.2 that it will be deleted.) Note that when the mean size is 1 Mb, then the value in the -f parameter corresponds to the desired coverage.

This component does not add to the input, but rather produces its own output of the form:

YAC_NAME=YAC00001
YAC_POSITION=C01:0.032-0.890
YAC_LENGTH=0.858
YAC_FINGERPRINT=8.8 12.3 15.8 6.2 5.3
YAC_STS=STS00001
=
YAC_NAME=YAC00002
YAC_POSITION=C01:0.061-0.950
YAC_LENGTH=0.889
YAC_FINGERPRINT=12.3 15.8 6.2 5.3
YAC_STS=STS00001 STS00002
=
etc.

Each group of entries, terminated by a line with '=' as the only character, specifies a YAC identifier, the YAC chromosome, the start and stop positions on the chromosome, the length, the fingerprint probes, and STSs that it would exhibit.

chimera.pl -f.3 30% of the YACS generated are chimeric. This component produces output similar to that of yacs.pl, except that some YACs are chimeric, like the following one which is composed of two YACs:

```
YAC_NAME=YAC01071
YAC_POSITION=C01:54.684-55.692 C01:43.252-44.125
YAC_LENGTH=1.881
YAC_FINGERPRINTS=3.8 0.3 12.8 12.9 8.1 11.0 11.1 2.4 8.2 4.0 3.3 7.8
11.4 10.7
YAC_STS=STS00126 STS00127 STS00167 STS00128 STS00168 STS00169
STS00170 STS00171
YAC_CONSTITUENTS=YAC00540 YAC00439
=
```

Facts of the form 'hit("sts(447)","yac(1315)").' for chromosome 1 can be extracted from temp.yacs with the command:

hit.pl -f.1 -g.05 -c1 < temp.yacs > temp.hits

which gives 10 percent false negatives (default 0), 5 percent false positives (default 0), and sets Chromosome 1 is the only chromosome of interest. A similar command, **overlapSim.pl**, extracts overlap data with a specified rate of false negatives and positives. The output of overlaps are lines of the form

8 1 3 4 5 6 9 11 12 14 15 16 17 18 136 342 395

meaning that clone 8 overlaps with clones 1, 3, 4, ..., 342, 395.

All of these components need not be used at once, but rather may be combined as needed. The experiments in Chapter 5 were modeled in the following way.

1. chromosome.pl -n1 -m100 -s2 | yacs.pl -f10 -m1 -s2 > temp.yacs

2. overlapSim.pl temp.yacs 0.3 0.1

The above commands generate one chromosome with a size of about 100 Mb, from which 1 Mb-size YACs are generated with a coverage of about 10. Then overlaps are determined for these YACs, allowing for 30% false negatives and 10% false positives. The false negative rate is the probability that a true overlap will be ignored, and the false positive rate is the probability that a random overlap will be generated. The false positive rate corresponds to the number of random overlaps divided by the total of true and random overlaps. To generate the graphs used in Table 5.1, I used chromosomes with lengths from 10 Mb to 2,560 Mb and clones of average length 1 Mb with coverage equal to 10. The false negative rate was set to 0.3, and false positive rate was 0.1. The algorithms gave similar performance when the false positive and false negative rates were set to zero.

Appendix B

Removal of Small Cycles

When a linear segment of a structure graph splits in two and immediately rejoins, creating a small cycle, the most likely cause is false negatives (*cf.* Figure 4.15). The following algorithm compresses small cycles that span just three BFS layers by uniting blobs in BFS layer L_i if they are connected by a path through blobs in layer L_{i+1} and also by a path through blobs in layer L_{i-1}.

Procedure Remove Small Cycles(G);

Input: $G = (V, E)$

i.e., structure graph G, where nodes are blobs: $V = b_1, \ldots, b_n$.

Output: modified structure graph $G' = (V', E')$, where G' is obtained from G by uniting blobs which are in the same BFS layer, provided they are connected by a path through blobs in the next layer and by a path through blobs in the previous layer.

begin {Remove Small Cycles}

1. **BFS**(G, b_1) ;

   i.e., do a BFS traversal of G starting at blob labeled b_1. This serves to define BFS layers $L_i, i = 1 \ldots n$ relative to the starting node, and a function *layer* such that $layer(b)$ returns the layer number of vertex b.

2. **for** $i = 1$ to n

   if b_i is not yet in a new blob **then** $S_i :=$ **New-Blob**(b_i) ;

   end {for}

   i.e., subprocedure **New-Blob** is called to create a new blob containing b_i and any other blob in the same layer as b_i if connected to b_i both through the next layer and the previous layer.

3. **foreach** $v \in V$

   foreach $u \in V$ such that $(u, v) \in E$

   if $blob(u) \neq blob(v)$ **then put** $(blob(u), blob(v)) \in E'$;

   end {foreach}

   end {foreach}

   i.e., create a structure graph edge between distinct blobs if there is an edge in the original graph G connecting a member of one blob to a member of the other.

Subprocedure New-Blob(v)

Input: graph $G = (V, E)$, vertex $v \in V$, and BFS layer, $layer(u)$, for each $u \in V$;

Output: a blob number $b = blob(v)$, incremented on each call, and a set $S_b \subseteq V$ such that $v \in S_b$ and for $u \neq v, u \in S_b$ iff u is connected to v by two paths: one that goes through the next BFS layer after the layer for v, and one that goes through the BFS layer prior to that of v. That is, there exist paths $P_1(u, v)$ and $P_2(u, v)$ such that foreach $z \in P_1(u, v)$, $layer(v) <= layer(z) <= layer(v) + 1$, and foreach $z \in P_2(u, v)$, $layer(v) - 1 <= layer(z) <= layer(v)$.

begin

$b = b + 1$ { b is the blob counter, a global variable }

$S_b = \{v\}$ and $blob(v) = b$

$hi\_layer = layer(v) + 1$;

$lo\_layer = layer(v) - 1$;

Place v on the queue

Mark v ;

while the queue is not empty do {FORWARD PATHS}

    take node x from the queue

    foreach neighbour y of x

        if y is not marked **and** $layer(v) <= layer(y) <= hi\_layer$ **then**

            put y on the queue and mark y

            if $layer(y) = layer(v)$ **then** $forward(y) = true$

        end{ if }

    end{ foreach}

end { while }

Empty queue and unmark all nodes.

Place v on the queue

Mark v ;

while the queue is not empty do {BACKWARD PATHS}

    take node x from the queue

    foreach neighbour y of x

        if y is not marked **and** $lo\_layer <= layer(y) <= layer(v)$ **then**

            put y on the queue and mark y

            if $layer(y) = layer(v)$ **and** $forward(y)$ **then**

                $S_b = S_b \cup \{y\}$;

                $blob(y) = b$;

            end{if}

        end{ if }

end{ foreach}

end { while }

The Remove Small Cycles algorithm typically unites two blobs in a cycle of length 4, called a C4. For example, nodes S2 and S3 in the C4 of Figure 4.15 are placed in a single blob, and the modified structure graph becomes linear. Figure 4.18 provides an example of the application of these algorithms to a structure graph.

Appendix C

The Modified Bron-Kerbosch Algorithm

In this appendix, I include a complete listing of the C code for MBK(list), i.e., the modified Bron-Kerbosch algorithm with a list-based implementation.

To see exactly what the modification is, read the comments labeled Steps 0 through 6. You will note that these steps are all in the *main* function, whereas the main part of the Bron-Kerbosch algorithm [10] is in the *extend* function. See Chapter 5 for a discussion of the latter function. Thus, the modification is really only in how the graph is supplied to the *extend* function. Bron and Kerbosch supply the graph all at once to extend, as follows:

```
for (i = 1 ; i <= ymax ; i++)
{
   all[i] = i ;
}
cc = 0 ;  // the index into the clique stack
extend(0, ymax, all) ;
```

In the modified version below, I feed the graph to the extend function line by line, where one line consists of a vertex y0 and its neighbours. Details of how this is accomplished are discussed in the comments below.

One last note of explanation might be helpful. It seems strange that the program reads the input file twice. This is wasteful of time, considering that I/O is typically slow and all of the information required can be obtained with one reading. The reason I do this is to cut down on the space required to hold the adjacency information. Today's desktop workstations would not have trouble holding the adjacency list for the overlap graph of the merged data, but a number of years ago it was a problem. Therefore, I used the abbreviated adjacency list illustrated in Figure 1.3, which requires only half as much memory as the full adjacency list. The abbreviated list works fine for checking if (i, j) is an edge, but is no good for finding all of the neighbours of vertex i. Therefore, to get the neighbours, I read the input file a second time, holding on to each line just long enough to find the cliques involving y0. (The first value on the line is y0, and the subsequent values are neighbours of y0.)

```
/**
 * FILE: mbk.c
 * PURPOSE: to read a file of adjacency information and
 *   then enumerate the maximal cliques.  The algorithm used
 *   is that of Bron and Kerbosch, with a small modification
 *   in that the graph is fed to the EXTEND function one line
 *   at a time, where one line holds the adjacency information
 *   for one vertex, y0.  An array called done keeps track of
 *   which vertices are done in the sense that we have found
 *   all the cliques which involve them.  The steps then are
 *   as follows:
 *   Step 0: read the whole input file to set up the adjacency
 *           list.  Close the file and reopen it to read from
 *           the beginning.
 *   REPEAT STEPS 1-5 UNTIL END OF FILE
 *     Step 1: read y0 and its NEIGHBOURS (one line of input)
 *     Step 2: use DONE and NEIGHBOURS to set up array ALL
 *     Step 3: put y0 in the clique stack
 *     Step 4: call the unmodified Bron-Kerbosch EXTEND with the
 *             parameter ALL so that it can build cliques based
 *             on y0.
 *     Step 5: update the done array to indicate that we are done
```

```
 *            with y0.
 */

#include <stdio.h>
#include <time.h>

#define LINE_LENGTH 2000  //maximum characters per line
#define YMAX 31000        //the maximum number of vertices
#define NMAX 400          //the maximum degree of a vertex

int ymax = -1 ;           //the largest numbered vertex
int cc ;                  //the index into compsub
int compsub[400] ;        //the clique stack
int count_cliques ;       //the number of cliques
FILE *f3, *fp, *fopen() ; //file pointers


/*            DEFINITIONS FOR ADJACENCY LIST     */


struct adj { /* basic entry of an adjacency list */
  int vertex ;
  struct adj *next ;  /* next entry in the adj. chain */
} ;

/* The array of adjacency lists is...*/
struct adj *connected[YMAX] ;
struct adj *np ;
struct adj *install() ;
struct adj *lookup() ;
struct adj *allocate() ;
struct adj qq ;
int count_installed ;


/*                    MAIN                    */


main(argc, argv)
int argc ;
char *argv[] ;

{ /* MAIN BODY */

  int ii, jj ;
```

```
long int start_time, MN_ElapsedTime() ;
int done[YMAX] ;
int all[NMAX] ;
int neighbours[NMAX] ;
int y0, ce, cei, ne ;

if ((f3 = fopen("cliques","w")) == NULL) {
   printf("cannot open cliques\n") ;
   exit(1) ;
}

//CHECK USAGE

if (argc != 2) {
  printf("USAGE: mbk input_file\n") ;
  exit(1) ;
}

//OPEN THE INPUT FILE

if ((fp = fopen(*++argv, "r") ) == NULL) {
  printf( "cannot open %s\n", *argv) ;
  exit(1) ;
}

//START THE TIMER

start_time = MN_ElapsedTime() ;
printf("Initial time is %d\n", start_time) ;

//STEP 0: READ THE FILE OF ADJACENCY INFORMATION

graphinput(fp, done) ;

fclose(fp) ;

//REOPEN THE INPUT FILE

if ((fp = fopen(*argv, "r") ) == NULL) {
  printf( "cannot open %s\n", *argv) ;
  exit(1) ;
}
else printf("Second opening of %s\n", *argv) ;

//STEP 1: READ A VERTEX y0 AND ITS NEIGHBOURS
```

```
  while ( (y0 = oneline(fp, neighbours, &ce)) ) {
    //EACH LINE

    //STEP 2A: TRANSFER DONE NEIGHBOURS TO ALL

    ne = 0 ;
    for (ii = 0 ; ii <= ce ; ii++ ) {
       if (done[neighbours[ii]]) { /* done list */
          ne++ ;
          all[ne] = neighbours[ii] ;
       }
    }

    //STEP 2B: TRANSFER NOT DONE NEIGHBOURS TO ALL

    cei = ne ;
    for (ii = 0 ; ii <= ce ; ii++ ) {
       if (!done[neighbours[ii]]) { /* done list */
          cei++ ;
          all[cei] = neighbours[ii] ;
       }
    }
    ce = cei ;

    //STEP 3: START THE CLIQUE WITH y0

    compsub[1] = y0 ;
    cc = 1 ;

    //STEP 4: CALL THE BRON-KERBOSCH EXTEND FUNCTION

    extend(ne, ce, all) ;

    //STEP 5: UPDATE THE DONE ARRAY, SINCE Y0 IS NOW DONE

    done[y0]++ ;

  } //EACH LINE

  printf("\n%d cliques printed\n", count_cliques) ;
  printf("End time is %d milliseconds\n", MN_ElapsedTime()) ;

} /* MAIN BODY */
```

```
/*                    ONELINE                         */

/* the value returned is y0 */
/* and neighbours is an array of neighbours of y0 */
/* *count_num is the index of the last neighbour */



oneline(fp, neighbours, count_num) /* read one line of OVER */
FILE *fp ;
int neighbours[] ;
int *count_num ;
{
  char line[LINE_LENGTH] ;    //maximum line length
  int y0, count ;
  int ii ;

  if ( fgets(line, LINE_LENGTH, fp) == NULL ) return(0) ;
  count = 0 ; /* the moving index in the line array */
  y0 = get_num(line,&count) ; /* the first value */
  *count_num = -1 ;

  while ( (ii = get_num(line,&count)) != 0 ) {
     neighbours[++(*count_num)] = ii ;
  }
  return(y0) ;
} /* ONELINE  */

/*
 *              GET NUM
 * based on p. 58
 * get next integer value at s[index],
 * see p. 58, Kernighan and Ritchie
 * the index is advanced to after the position of value obtained
 * returns 0 if index puts you at the end of string s
 */

get_num(s,index)
char s[] ;
int  *index ;
{
  int i, n, len ;
```

```
  len = strlen(s) ;

  for ( i = *index ; i <= len  && (s[i] == ' ' || s[i] == '\n'
              || s[i] == '\t') ; i++  ) ;

  for ( n = 0 ; i <= len && s[i] >= '0' && s[i] <= '9' ; i++ )
      n = 10 * n + s[i] - '0' ;

  *index = i ;
  return(n) ;
} /* GET NUM */


/*               EXTEND  (of Bron and Kerbosch)       */

extend(ne, ce, old)
int ne, ce, *old ;
{
  int *new ;
  int nod, fixp, ne1 ;
  int newne, newce, i, j, count, pos, p, s, sel, minnod ;
  int ii ;

  new = (int *)malloc((ce+1)*sizeof(int)) ;

  minnod = ce ;
  nod = 0 ;

  //DETERMINE EACH COUNTER VALUE AND LOOK FOR MINIMUM

  for (i = 1 ; i <= ce ; i++) {

    if ( minnod == 0 ) break ;
    p = old[i] ;
    count = 0 ;

    //COUNT DISCONNECTIONS:

    for (j = ne + 1 ; j <= ce ; j++ ) {

      if ( count >= minnod ) break ;

      if ( lookup(p,old[j]) != NULL ) { /* if not connected */
        count++ ;
```

```
//SAVE POSITION OF POTENTIAL CANDIDATE:

        pos = j ;
      }
    } /* END J */

    //TEST THE NEW MINIMUM

    if (count < minnod) {
       fixp = p ;
       minnod = count ;
       if (i <= ne) {
         s = pos ;
       }
       else {
         s = i ;
         nod = 1 ;
       }
    } /* END NEW MINIMUM */

  } /* END I  */
  /*
   * COMMENT if fixed point initially chosen from candidates then
   *         number of disconnections will be preincreased by one;
   */

  //BACKTRACKCYCLE:

  //SEARCH TREE LOOP
  for (nod = minnod + nod; nod > 0 ; nod--) {
    ne1 = ne + 1 ;

    //INTERCHANGE:

    p = old[s] ;
    old[s] = old[ne1] ;
    sel = old[ne1] = p ;

    //FILL NEW SET not

    newne = 0 ;

    //NEW NOT
    if ( ne ) {
      for (i = 1; i <= ne ; i++) {
```

```
if ( lookup(sel,old[i]) == NULL ) {
  //i.e., if connected
  newne++ ;
  new[newne] = old[i] ;
}
      }
    } //NEW NOT

    //FILL NEW SET CAND

    newce = newne ;

    for (i = ne + 2 ; i <= ce ; i++ ) {
      if ( lookup(sel,old[i]) == NULL ) {
//i.e., if connected
        newce++ ;
        new[newce] = old[i] ;
      }
    }

    //ADD sel TO COMPSUB:

    compsub[++cc] = sel ;

    if ( newce == 0 ) {
      //NO EXTENSIONS SO OUTPUT THE CLIQUE
      count_cliques++ ;
      for ( ii = 1 ; ii <= cc ; ii++ )
fprintf( f3, "%d ", compsub[ii] ) ;
      fprintf( f3, "\n" ) ;
    } //END NO EXTENSIONS

    else {
      //YES EXTENSIONS
      if ( newne < newce ) {
//EXTEND
extend(newne,newce,new) ;
      }//DONE EXTEND
    }

    //REMOVE sel FROM COMPSUB

    cc-- ;

    //ADD TO not:
```

```
    ne++ ;

    if (nod > 1) {

      //SELECTION OF NEXT CANDIDATE DISCONNECTED TO FIXED POINT:

      for ( s = ne + 1 ; s <= ce; s++ ) {
         if ( lookup(fixp, old[s]) != NULL ) break ;
 //i.e., if not connected
      }
    } //END SELECTION
  }//END BACKTRACKCYCLE (SEARCH TREE LOOP)

  free(new) ;
}// END EXTEND


/*
 *                     MN_ELAPSEDTime
 *
 * see man 3c clock for description of clock()
 * it returns time in microseconds
 * and wraps around in 2147 seconds (36 min)
 */

/*long int MN_ElapsedTime( void ) */
long int MN_ElapsedTime()
{
   static long int time = 0;
   long int et;

   et = time;
   time = clock();
   et = time - et;
   return( (long)et / 1000 );
}
```

```
/*                  SPARSE GRAPHINPUT                        */



graphinput(fp, all) /* read the input graph as lines of OVER */
FILE *fp ;
int all[] ;
{
  char line[LINE_LENGTH] ;
  int y0, count, n, m, s ;
  int ii, count_lines = 0 ;

  // INITIAL ALLOCATION

  for ( ii = 0 ; ii < YMAX ; ii++ ) {
    np = allocate() ;
    connected[ii] = np ;
    connected[ii]->next = NULL ;
    connected[ii] -> vertex = 0 ;
  }



  while ( fgets(line, LINE_LENGTH, fp) != NULL ) {
      count_lines++ ;

      if ( strlen(line) >= LINE_LENGTH -1 ) {
        printf("Line %d has length >= %d \n", count_lines,
       LINE_LENGTH - 1) ;
        printf( "Increase LINE_LENGTH and recompile\n" ) ;
        exit(4) ;
      }

      count = 0 ; /* the moving index in the line array */
      y0 = get_num(line,&count) ; /* the first value */
      if ( y0 > ymax ) ymax = y0 ;

      all[y0] = 0 ; /* indicates that this one is not done yet */

      while ( (n = get_num(line,&count)) != 0 ) {
        if ( n > YMAX ) {
  printf("Line %d has a value (%d) ", count_lines, n) ;
  printf("which is greater than YMAX (%d).\n", YMAX) ;
  printf("Increase YMAX and recompile.\n") ;
  exit(10) ;
```

```
}
        if ( n > y0 && install(y0,n) == NULL ) exit(1) ;
        if (n > ymax) ymax = n ;
      }
  }

  printf( "\nGRAPHINPUT: %d lines read; max y was %d\n",
  count_lines, ymax ) ;

} /* GRAPHINPUT  */




/*                      LOOKUP
 *
 * Return NULL if you find it
 * and a pointer to its would be parent if you don't find it
 * It is assumed that connected(y,y), according to BK
 */

struct adj *lookup(v,x) /* look for x in connected[v] */
int v, x ;
{ /* LOOKUP */
  struct adj *np, *parent ; /* np is ptr to adjacency element */
  int tt ;

  if ( v == x ) {
    return( NULL ) ; /* by definition */
  }

  if ( x < v ) { /* SWAP */
     tt = x ;
     x = v ;
     v = tt ;
   }

  parent = connected[v] ;
  for ( np = connected[v] ; np != NULL ; np = np -> next ) {
     // SCAN THE LIST
     if ( np -> vertex  == x ) {
        return( NULL ) ; /* FOUND */
     }

     if ( np -> vertex > x ) {
```

```
        return( parent ) ;
       }

      else {
        //MOVE ON IN LIST
        parent = np ;
      } /* MOVE ON */

    } //SCAN THE LIST

   return(parent) ;
  } //LOOKUP

  /*
   *                   INSTALL
   *
   *
   * Put x into the adjacency list of y
   */

  struct adj *install(v,x)
  int x, v ;
  {
    //INSTALL
    struct adj *np, *parent ; /* np is ptr to an adjacency element */
    int ii ;

    if ( ( parent = lookup(v,x) ) == NULL ) { /* FOUND */
       printf("Odd: v = %d, x = %d is already installed\n", v, x) ;
       exit(3) ;
    }

    else if ( parent != connected[v] ) {
       // INSERT AFTER NODE
       np = allocate() ;
       np->next = parent->next ;
       parent->next = np ;
       np -> vertex = x ;
    } //INSERT AFTER NODE

    else if ( parent -> vertex ) {
       // PARENT HAS A VERTEX
       if ( x > parent -> vertex ) {
          //INSERT AFTER
          np = allocate() ;
```

```
         np->next = parent->next ;
         parent->next = np ;
         np -> vertex = x ;
      } //INSERT AFTER
      else {
         //INSERT BEFORE
         np = allocate() ;
         np->next = parent ; /* new node goes before connected */
         connected[v] = np ; /* connected now points to new node */
         np -> vertex = x ;
      }//INSERT BEFORE
   }//PARENT HAS A VALUE

   else {
     //NO ALLOCATION
     parent -> vertex = x ;
   }// NO ALLOCATION

   count_installed++ ;
   return(np) ;

}//INSTALL

/*
 *                      ALLOCATE
 *
 * allocate an adjacency cell
 * exit if you can't
 */

struct adj *allocate()
{/* ALLOCATE */
  struct adj *np ;

  np = (struct adj *) malloc(sizeof(*np)) ;

  if ( np == NULL ) {
    // NO ROOM
    printf("INSTALL oddity: malloc returned NULL, no space\n") ;
    exit(3) ;
  }//NO ROOM

  return(np) ;

}//ALLOCATE
```

Bibliography

[1] M. D. Adams et al. The genome sequence of *drosophila melanogaster*. *Science*, 287:2185–2195, 2000.

[2] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2:1–6, 1973.

[3] A. P. Ambler, H. G. Barrow, C. M. Brown, R. M. Burstall, and R. J. Popplestone. A versatile computer controlled assembly system. In *Third International Joint Conference on Artificial Intelligence: Advance Papers*, pages 298–303, 1973.

[4] R. Arratia, E. S. Lander, S. Tavare, and M. S. Waterman. Genomic mapping by anchoring random clones: A mathematical analysis. *Genomics*, 11:806–827, 1991.

[5] J. G. Augustin and J. Minker. An analysis of some graph theoretical cluster techniques. *J. Assoc. Comput. Mach.*, 17:571–588, 1970.

[6] F. R. Blattner, G. Plunkett III, C. A. Bloch, N. T. Perna, V. Burland, M. Riley, J. Collado-Vides, J. D. Glasner, C. K. Rode, G. F. Mayhew, J. Gregor, N. W. Davis, H. A. Kirkpatrick, M. A. Goeden, D. J. Rose, B. Mau, and Y. Shao. The complete genome sequence of *escherichia coli* K-12. *Science*, 277:1453–1462, 1997.

[7] 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*, 13:333–379, 1976.

[8] P. Bork and R. Copley. Filling in the gaps. *Nature*, 409:818–820, 2001.

[9] G. Bouffard, J. Ostell, and K. E. Rudd. GeneScape: a relational database of *escherichia coli* genomic map data for Macintosh computers. *CABIOS*, 8:563–567, 1992.

[10] C. Bron and J. Kerbosch. Algorithm 457. finding all cliques of an undirected graph. *Communications of the ACM*, 16:575–577, 1973.

[11] D. T. Burke, G. F. Carle, and M. V. Olson. Cloning of large segments of exogenous DNA into yeast by means of artificial chromosome vectors. *Science*, 236:806–812, 1987.

[12] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14:210–223, 1985.

[13] I. M. Chumakov et al. A YAC contig map of the human genome. *Nature*, 377 Suppl.:175–298, 1995.

[14] D. Cohen, I. Chumakov, and J. Weissenbach. A first generation physical map of the human genome. *Nature*, 336:698–701, 1993.

[15] M. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*. PhD thesis, Department of Computer Science, University of Toronto, 1994.

[16] D. G. Corneil, H. Kim, S. Natarajan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Information Processing Letters*, 55:99–104, 1995.

[17] D. G. Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm? In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 175–180, 1997.

[18] D. G. Corneil, S. Olariu, and L. Stewart. Linear time algorithms for dominating pairs in asteroidal triple-free graphs. *SIAM Journal on Computing*, 28:1284–1297, 1999.

[19] D. G. Corneil and M. E. Woodward. A comparison and evaluation of graph theoretical clustering techniques. *INFOR, Canadian Journal of Operational Research and Information Processing*, 16:74–89, 1978.

[20] A. Coulson, J. Sulston, S. Brenner, and J. Karn. Toward a physical map of the genome of the nematode *caenorhabditis elegans*. *Proc. Natl Acad. Sci. USA*, 83:7821–7825, 1986.

[21] D. R. Cox, M. Burmeister, E. R. Price, S. Kim, and R. M. Myers. Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes. *Science*, 250:245–250, 1990.

[22] A. J. Cuticchia, J. Arnold, and W. E. Timberlake. ODS: ordering DNA sequences — a physical mapping algorithm based on simulated annealing. *CABIOS*, 2:215–219, 1993.

[23] S. R. Das. On a new approach for finding all the modified cut-sets in an incompatibility graph. *IEEE Trans. Comput.*, C-22:187–193, 1973.

[24] S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Jr. Stoeckert. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40:512–531, 2001.

[25] C. Dib et al. A comprehensive genetic map of the human genome based on 5,264 microsatellites. *Nature*, 380:152–154, 1996.

[26] W. F. Dietrich et al. A comprehensive genetic map of the mouse genome. *Nature*, 380:149–152, 1996.

[27] R. D. Fleischmann et al. Whole-genome random sequencing and assembly of *haemophilus influenzae*. *Science*, 269:496–512, 1995.

[28] M. Fonstein and R. Haselkorn. Physical mapping of bacterial genomes. *Journal of Bacteriology*, 177:3361–3369, 1995.

[29] D. R. Fulkerson and O. A. Gross. Incidence matrices and interval graphs. *Pacif J. Math.*, 15:835–855, 1965.

[30] D. J. Galas. Making sense of the sequence. *Science*, 291:1257–1260, 2001.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman and Company, 1979.

[32] W. Gillett, L. Hanks, G. K. Wong, J. Yu, R. Lim, and M. V. Olson. Assembly of high-resolution restriction maps based on multiple complete digests of a redundant set of overlapping clones. *Genomics*, 33:389–408, 1996.

[33] A. Goffeau et al. Life with 6000 genes. *Science*, 274:562–567, 1996.

[34] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, 1980.

[35] M. C. Golumbic, H. Kaplan, and R. Shamir. Graph sandwich problems. *J. of Algorithms*, 19:449–473, 1995.

[36] M. C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *J. Assoc. Comput. Mach.*, 40:1108–1133, 1993.

[37] C. C. Gotlieb and S. Kumar. Semantic clustering of index terms. *Journal of ACM*, 15:493–513, 1968.

[38] E. D. Green and P. Green. Sequence-tagged site (STS) content mapping of human chromosomes: theoretical considerations and early experiences. *PCR Methods Appl*, 2:77–90, 1991.

[39] E. D. Green, H. C. Riethman, J. E. Dutchik, and M. V. Olson. Detection and characterization of chimeric yeast artificial-chromosome clones. *Genomics*, 11:658–669, 1991.

[40] M. Haldi, V. Perrot, M. Saumier, T. Desai, D. Cohen, D. Cherif, D. Ward, and E. S. Lander. Large human YACs constructed in a rad52 strain show a reduced rate of chimerism. *Genomics*, 24:478–484, 1994.

[41] F. Harary. *Graph Theory*. Addison-Wesley Publishing Company, 1969.

[42] F. Harary and I. C. Ross. A procedure for clique detection using the group matrix. *Sociometry*, 20:205–215, 1957.

[43] E. Harley and A. J. Bonner. A flexible approach to genome map assembly. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 161–169. AAAI Press, 1994.

[44] E. Harley, A. J. Bonner, and N. Goodman. Good maps are straight. In *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, pages 88–97. AAAI Press, 1996.

[45] E. Harley, A. J. Bonner, and N. Goodman. Revealing hidden interval graph structure in STS-content data. *Bioinformatics*, 15:278–285, 1998.

[46] E. Harley, A. J. Bonner, and N. Goodman. Uniform integration of genome mapping data using intersection graphs. *Bioinformatics*, 17:487–494, 2001.

[47] T. J. Hudson et al. An STS-based map of the human genome. *Science*, 270:1945–1954, 1995.

[48] T. Imai and M. V. Olson. Second-generation approach to the construction of yeast artificial-chromosome libraries. *Genomics*, 8:297–303, 1990.

[49] T. Jiang and M. Li. Optimization problems in molecular biology. In D. Z. Du and J. Sun, editors, *Advances in Optimization and Approximation*, pages 195–216. Kluwer Academic Publishers, 1994.

[50] H. C. Johnston. Cliques of a graph — variations on the Bron-Kerbosch algorithm. *International Journal of Computer and Information Sciences*, 5:209–238, 1976.

[51] R. M. Karp. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 278–285. ACM Press, 1993.

[52] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, INC, 1978.

[53] D. E. Knuth. *The art of computer programming, Vol. 1: Fundamental algorithms*. Addison-Wesley Publishing Company, 1968.

[54] N. Korte and Möhring, R. H. An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal of Computing*, 18:68–81, 1989.

[55] E. S. Lander and M. S. Waterman. Genomic mapping by fingerprinting random clones: A mathematical analysis. *Genomics*, 2:231–239, 1988.

[56] A. Lee, E. Rundensteiner, S. Thomas, and S. Lafortune. An information model for genome map representation and assembly. Technical Report SDE-TR-163-93, University of Michigan, Dept of Electrical Engineering and Computer Science, Ann Arbor, MI 48109-2122, 1993.

[57] C. G. Lekkerkerker and J. C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.

[58] S. Letovsky and M. B. Berlyn. CPROP: A rule-based program for constructing genetic maps. *Genomics*, 12:435–446, 1992.

[59] B. Lewin. *Genes IV*. Oxford University Press, New York, 1990.

[60] E. Loukakis. A new backtracking algorithm for generating the family of maximal independent sets of a graph. *Comp. and Maths. with Appls.*, 9:583–589, 1983.

[61] E. Loukakis and C. Tsouros. A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically. *Computing*, 27:249–266, 1981.

[62] C. Magness and P. Green. *SEGMAP User's Manual, Version 3.48, Copyright 1994-1996*, 1996.

[63] B. Mirkin. *Mathematical Classification and Clustering*. Kluwer Academic Publishers, 1998.

[64] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math*, 3:23–28, 1965.

[65] R. Mott, A. Grigoriev, E. Maier, J. Hoheisel, and H. Lehrach. Algorithms and software tools for ordering clone libraries: application to the mapping of the genome of Schizosaccharomyces pombe. *Nucleic Acids Research*, 8:1965–1974, 1993.

[66] G. D. Mulligan and D. G. Corneil. Corrections to Bierstone's algorithm for generating cliques. *J. Assoc. Comput. Mach.*, 19:244–247, 1972.

[67] T. F. Murphy and M. A. Lappé, editors. *Justice and the Human Genome Project*. University of California Press, Berkeley, 1994.

[68] P. M. Nadkarni, A. Banks, K. Montgomery, J. LeBlanc-Stracewski, P. Miller, and K. Krauter. CONTIG EXPLORER: Interactive marker-content map assembly. *Genomics*, 31:301–310, 1996.

[69] C. Nusbaum, D. K. Slonim, K. L. Harris, B. W. Birren, R. G. Steen, L. D. Stein, J. Miller, W. F. Dietrich, R. Nahf, V. Wang, O. Merport, A. B. Castle, Z. Husain, G. Farino, D. Gray, M. O. Anderson, R. Devine, L. T. Horton Jr., W. Ye, X. Wu, V. Kouyoumjian, I. S. Zemsteva, Y. Wu, A. J. Collymore, D. F. Courtney, J. Tam, M. Cadman, A. R. Haynes, C. Heuston, T. Marsland, A. Southwell, P. Trickett, M. A. Strivens, M. T. Ross, W. Makalowski, Y. Xu, M. S. Boguski, N. P. Carter, P. Denny, S. D.M. Brown, T. J. Hudson, and E. S. Lander. A YAC-based physical map of the mouse genome. *Nature Genet.*, 22:388–393, 1999.

[70] M. Olivier et al. A high-resolution radiation hybrid map of the human genome draft sequence. *Science*, 291:1298–1302, 2001.

[71] M. V. Olson. Clone by clone by clone. *Nature*, 409:816–818, 2001.

[72] M. V. Olson, J. E. Dutchik, M. Y. Graham, G. M. Brodeur, C. Helms, M. Frank, M. MacCollin, R. Scheinman, and T. Frank. Random-clone strategy for genomic restriction mapping in yeast. *Proc. Natl Acad. Sci. USA*, 83:7826–7830, 1986.

[73] R. E. Osteen and J. T. Tou. A clique-detection algorithm based on neighborhoods in graphs. *Int. J. Comput. Inf. Sci.*, 2:257–268, 1973.

[74] P. M. Pardalos, I. M. Bomze, M. Budinich, and M. Pelillo. The maximum clique problem. In D. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization, Supplement, Vol. A*, pages 1–74. Kluwer Academic Publishers, 1999.

[75] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4:301–328, 1994.

[76] F. S. Roberts. *Representations of indifference relations.* PhD thesis, Stanford University, 1968.

[77] L. Roberts. Controversial from the start. *Science*, 291:1182–1188, 2001.

[78] D. S. Roos. Bioinformatics — trying to swim in a sea of data. *Science*, 291:1260–1261, 2001.

[79] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination of graphs. *SIAM Journal on Computing*, 5:266–283, 1976.

[80] F. Sanger, G. M. Air, B. G. Barrell, H. L. Brown, A. R. Coulson, J. C. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage ϕ X174 DNA. *Nature*, 265:687–695, 1977.

[81] T. Sato and B.V. Smith. *Xfig User Manual Version 3.3.3d, Copyright 1997-2000*, 2001.

[82] O. Schmeltzer. Using temporal reasoning for genome map assembly. In *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, pages 332–340. AAAI Press, 1995.

[83] H. Schmitt, U. Kim, T. Slepak, N. Blin, M. Simon, and H. Shizuya. Framework for a physical map of the human 22q13 region using bacterial artificial chromosomes (BACs). *Genomics*, 33:9–20, 1996.

[84] D. Schuler et al. Gene map of the human genome. *Science*, 274:540–546, 1996.

[85] C. Soderlund and I. Dunham. SAM: A system for iteratively building marker maps. *CABIOS*, 11:645–655, 1995.

[86] The Arabidopsis Genome Initiative. Analysis of the genome sequence of the flowering plant *arabidopsis thaliana*. *Nature*, 408:796–815, 2000.

[87] The BAC Resource Consortium. Integration of cytogenetic landmarks into the draft sequence of the human genome. *Nature*, 409:953–958, 2001.

[88] The *C. elegans* Sequencing Consortium. Genome sequence of the nematode *c. elegans:* a platform for investigating biology. *Science*, 282:2012–2018, 1998.

[89] The International Human Genome Mapping Consortium. A physical map of the human genome. *Nature*, 409:934–940, 2001.

[90] The International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–919, 2001.

[91] The Organization for Economic Co-operation and Development. *The Global Human Genome Project*. OECD, Paris, France, 1995.

[92] The University of California in Southern California. Human genome project working draft. http://genome.cse.ucsc.edu/, 2001.

[93] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for finding all the cliques. Technical Report TR-C5, UEC, 1988.

[94] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximum independent sets. *SIAM Journal on Computing*, 6:505–517, 1977.

[95] P. K. T. Vaswani. A technique for cluster emphasis and its application to automatic indexing. *Information Processing*, 68:1300–1303, 1968.

[96] J. C. Venter et al. The sequence of the human genome. *Science*, 291:1304–1351, 2001.

[97] Y. Wang, R. A. Prade, J. Griffith, W. E. Timberlake, and J. Arnold. ODS_BOOTSTRAP: assessing the statistical reliability of physical maps by bootstrap resampling. *CABIOS*, 6:625–634, 1994.

[98] J. D. Watson, J. Tooze, and D. T. Kurtz. *Recombinant DNA*. Scientific American Books, W.H. Freeman and Company, New York, 1983.

[99] C. C. Whittaker, M. O. Mundt, D. J. Faber, V. Balding, R. L. Dougherty, R. L. Stallings, S. W. White, and D. C. Torney. Computations for mapping genomes with clones. *International Journal of Genome Research*, 1:195–226, 1993.

[100] F. A. Wright, W. J. Lemon, W. D. Zhao, R. Sears, D. Zhuo, J.-P. Wang, H.-Y. Yang, T. Baer, D. Stredney, J. Spitzner, A. Stutz, R. Krahe, and B. Yuan. A draft annotation and overview of the human genome. *Genome Biology*, 2:1–18, 2001.