Dissertation

Dynamic Representation of Consecutive-Ones Matrices and Interval Graphs

Submitted by

William M. Springer II

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2015

Abstract

DYNAMIC REPRESENTATION OF CONSECUTIVE ONES MATRICES AND INTERVAL GRAPHS

We give an algorithm for updating a consecutive-ones ordering of a consecutive-ones matrix when a row or column is added or deleted. When the addition of the row or column would result in a matrix that does not have the consecutive-ones property, we return a well-known minimal forbidden submatrix for the consecutive-ones property, known as a Tucker submatrix, which serves as a certificate of correctness of the output in this case, in $O(n \log n)$ time. The ability to return such a certificate within this time bound is one of the new contributions of this work. Using this result, we obtain an $\mathrm{O}(n)$ algorithm for updating an interval model of an interval graph when an edge or vertex is added or deleted. This matches the bounds obtained by a previous dynamic interval-graph recognition algorithm due to Crespelle. We improve on Crespelle's result by producing an easy-to-check certificate, known as a Lekkerkerker-Boland subgraph, when a proposed change to the graph results in a graph that is not an interval graph. Our algorithm takes $O(n \log n)$ time to produce this certificate. The ability to return such a certificate within this time bound is the second main contribution of this work.

Table of Contents

List of Figures

CHAPTER 1

Introduction

A graph is an *interval graph* if it is the intersection graph of a set of intervals on the line. In other words, a graph is an interval graph if it has a representation with one interval for each vertex such that two vertices are adjacent if and only if the corresponding intervals intersect. Such a set of intervals is an *interval model* of the interval graph (see figure 1.1).

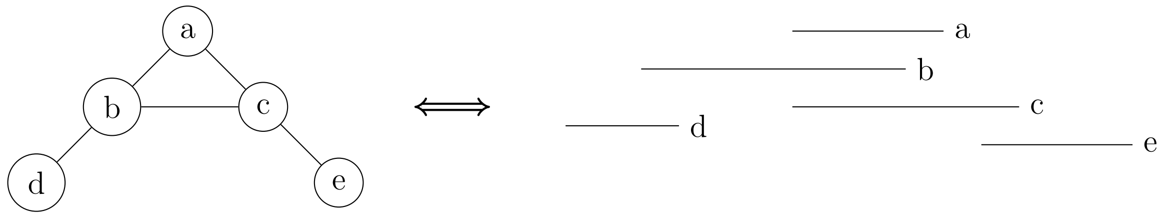


Figure 1.1. An interval model of an interval graph

One of the first applications of interval graphs arose in the late 1950s in establishing that genes reside on a single molecule of DNA of enormous length [Ben59]. The linear topology of a DNA molecule had been known since 1953, when it was described by Watson and Crick, and it was also known that the collection of genes had a linear arrangement along the chromosome. However, it was not known whether the genetic information in a chromosome is written on a single DNA molecule or organized around multiple independent DNA molecules or one with small branches.

Benzer isolated 145 mutant strains of a bacteria-infecting virus, theorizing that each mutation occupied a contiguous region of the genome. Bacteria infected with two of the strains would give rise to viable viruses only if the regions occupied by the two mutations did not intersect. While one of the strains had to be discarded due to having non-contiguous mutations, he was able to find an interval graph on the other 144 strains, providing strong evidence that the fine structure of the genome was linear.

This gave rise to research on characterizations of this graph class, as well as on efficient algorithms to recognize whether a graph is an interval graph, and, if so, to find a set of intervals that represents the graph. In 1974, Booth and Lueker gave the first linear-time algorithms for recognizing interval graphs and finding an interval model [BL76].

A *subgraph* of a graph is any graph obtainable by deleting an arbitrary subset of its vertices and edges, while an *induced subgraph* is one obtainable by deleting a subset of its vertices. In 1962, Lekkerkerker and Boland characterized the minimal non-interval graphs. This gives a characterization of the interval graphs as those graphs that do not contain one of Lekkerkerker and Boland's subgraphs as an induced subgraph. The first linear-time algorithm for finding such a graph in a non-interval graph was given recently by Lindzey and McConnell [LM14]. (See Figure 1.2.)

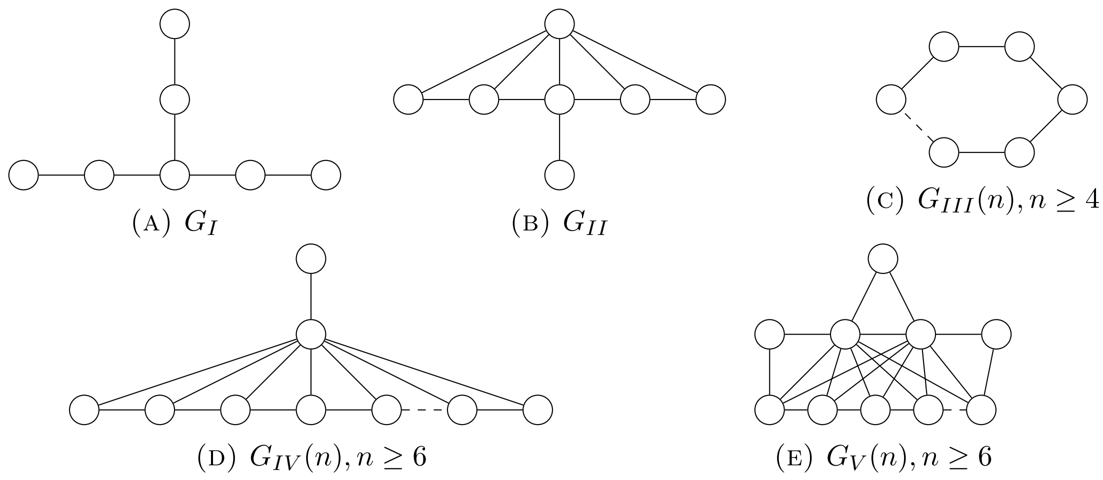


FIGURE 1.2. The Lekkerkerker-Boland graphs.
The dashed edges indicate infinite classes by indicating where an arbitrary number of vertices fitting the pattern can be inserted. For example, the picture of the G_{III} indicates that any cycle of length greater than or equal to four is a Lekkerkerker-Boland subgraph. G_I and G_{II} each depict a single graph, rather than a set of graphs.

A *clique* of a graph is a set of vertices that are pairwise adjacent. A clique is *maximal* if it is a subset of no larger clique of the graph, and is a *maximum clique* if it is as large as every clique in the graph. The size of a maximum clique of a graph is called the *clique* *number* of the graph. The problem of finding the clique number, or a maximum clique, in a graph is a well-known NP-hard problem.

A *(proper) coloring* of a graph is an assignment of colors to the vertices such that no edge has two endpoints of the same color. The minimum number of colors required to color a graph is called the *chromatic number* of the graph. Finding the chromatic number of a graph is also a well-known NP-hard problem. However, as a result of Booth and Lueker's algorithm, we can find a maximum clique and minimum coloring in the special case where the graph is an interval graph, by a greedy algorithm commonly given in introductory textbooks [CLRS09, KT05]. These texts also give a linear-time greedy algorithm for finding the minimum number of cliques required to cover the vertex set V, known as the *clique cover number*, and the size of a maximum stable set, known as the *stability number*; in these texts, the problem is often known as the *activity scheduling* problem.

Another characterization of interval graphs is obtained as follows. A binary matrix has the *consecutive-ones property* if there exists an ordering of its columns, called a *consecutive-ones ordering*, such that the 1s in each row are consecutive (see figure 1.3). A *clique matrix* of a graph has one row for each vertex and one column for each maximal clique, and a 1 in row i, column j if vertex i is a member of clique j. A graph is an interval graph if and only if its clique matrices have the consecutive-ones property by a well-known result of Gilmore and Hoffman [GH64]. This can be seen as follows. Two vertices are adjacent if and only if they are members of a common maximal clique. Therefore, the blocks of 1s in the rows of a consecutive-ones ordering of a clique matrix, when interpreted as intervals on the column sequence, form an interval model of the graph. Conversely, given an interval model, it is easy to identify maximal cliques with points on the line, where the intervals that contain

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

FIGURE 1.3. A consecutive-ones matrix

each point form a maximal clique. Arranging the columns of a clique matrix according to this order gives a consecutive-ones ordering of it.

This observation is the basis of Booth and Lueker's algorithm for recognizing interval graphs. At the heart of this algorithm is an algorithm to recognize whether a binary matrix has the consecutive-ones property, and if it does, to return a consecutive-ones ordering of it.

In 1972, Tucker characterized the minimal matrices that are not consecutive-ones matrices. They are minimal in the sense that deletion of any row or column gives rise to a matrix that has the consecutive-ones property. An *induced submatrix* of a matrix is a matrix obtained by deleting a subset of its rows and columns. Tucker's result gives a characterization of the consecutive-ones matrices as those that do not contain one of his matrices as an induced submatrix. (See Figure 1.4.) An algorithm for finding a Tucker submatrix in a binary matrix if it does not have the consecutive-ones property was first given by Dom, Guo, and Niedermeier [DGN10], and a linear-time algorithm was given recently by Lindzey and McConnell [LM14].

Algorithms that return such forbidden structures are of great interest, since they give an easy certificate that the output of an implementation has not been compromised by an implementation bug [McC04, KMMS03, HK]. Such an algorithm is called a *certifying algorithm.* An interval model gives an easy-to-check certificate that a graph is an interval graph, and a Lekkerkerker-Boland subgraph gives an easy-to-check certificate that it is not. Similarly, a consecutive-ones ordering gives an easy-to-check certificate that a binary matrix

$$\begin{array}{c|cccccc|}
0 & 1 & 1 & & & & \\
1 & & 1 & 1 & & & \\
2 & & & 1 & 1 & & \\
\vdots & & & & \ddots & \ddots & \\
k-2 & & & & & 1 & 1 \\
k-1 & 1 & 0 & \dots & 0 & 0 & 1
\end{array}$$

(A) $M_I(k), k \geq 3$

$$\begin{array}{c|cccccc|}
0 & 0 & 1 & 1 & \dots & 1 & 1 \\
1 & 1 & 1 & & & & \\
2 & & 1 & 1 & & & \\
\vdots & & & \ddots & \ddots & & \\
k-2 & & & & 1 & 1 & 0 \\
k-1 & 1 & \dots & 1 & 1 & 0 & 1
\end{array}$$

(B) $M_{II}(k), k \geq 4$

$$\begin{array}{c|ccccccc|}
0 & 1 & 1 & & & & & \\
1 & & 1 & 1 & & & & \\
2 & & & 1 & 1 & & & \\
\vdots & & & & \ddots & \ddots & & \\
k-2 & & & & & 1 & 1 & 0 \\
k-1 & 0 & 1 & \dots & 1 & 1 & 0 & 1
\end{array}$$

(C) $M_{III}(k), k \geq 3$

$$\begin{array}{c|cccccc|}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}$$

(D) M_{IV}

$$\begin{array}{c|ccccc|}
0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 1 & 1 & 1 & 0 \\
3 & 1 & 0 & 0 & 1 & 1
\end{array}$$

(E) M_V

Figure 1.4. The Tucker matrices

has the consecutive-ones property and a Tucker submatrix gives one that it does not. Thus, the linear-time algorithms that return these give certifying algorithms for interval-graph recognition and recognition of consecutive-ones matrices. However, certifying algorithms were given earlier for these two problems that returned other kinds of certificates [KMMS06, McC04].

A *dynamic recognition algorithm* for interval graphs is one that either allows deletion or insertion of an edge or a vertex in an interval graph or else rejects the change as resulting in a graph that is not an interval graph. This gives an *on-line* algorithm, where the entire input is not required in advance of running the algorithm. It is also useful for finding maximal interval subgraphs in a graph that is not an interval graph. Such an approach is useful for genomic applications that produce a graph that, in theory, should be an interval graph, but that in practice is usually not, due to errors in the data. A dynamic recognition algorithm for interval graph recognition was previously given by Crespelle, with a bound of $\mathrm{O}(n)$ per operation [Cre10]. Our result differs from that of Crespelle in that it gives a certifying algorithm for the problem, by providing a certificate in $O(n \log n)$ time in the case that the algorithm rejects the change as producing a graph that is not an interval graph. Our certificate is a Lekkerkerker-Boland subgraph. In addition, the algorithm is considerably simpler than Crespelle's.

To accomplish this, we develop a dynamic recognition algorithm for consecutive-ones matrices, which determines whether, when a row or column is added to a consecutive-ones matrix, the result is a consecutive-ones matrix, in $\mathrm{O}(n)$ time. This result was also given by Crespelle. Our new contribution is the production of a certificate that the new matrix is not a consecutive-ones matrix, in $O(n \log n)$ time. Our certificate is a Tucker submatrix.

CHAPTER 2

Preliminaries

When not otherwise indicated, we use conventional notation consistent with that of [CLRS09].

Let $G = (V, E)$ be a graph and v be a vertex. Given a subset $\emptyset \subset X \subseteq V$, $G[X]$ is the subgraph of G induced by X. $G - x$ is $G[V \setminus \{x\}]$, and $G - X$ is $G[V \setminus X]$. If $G = (V, E)$ is a graph, $G[A]$ is an induced subgraph, and $x \in V \setminus A$, $G[A] + x$ denotes $G[A \cup \{x\}]$. If $S \subset V \setminus A$, $G[A] + S$ denotes $G[A \cup S]$.

The *complement* of a graph G, denoted $\overline{G}$, is a graph on the same set of vertices, where two vertices are adjacent in $\overline{G}$ only if they were not adjacent in G. A cycle in the graph is a set of vertices $v_1, v_2, v_3, ..., v_k, v_1$. A *chord* on the cycle is an edge between two vertices in the cycle that is not itself part of the cycle; for example, between v_1 and v_3 in the cycle above. A *chordless cycle* is a cycle on at least four vertices that has no chords.

The *(open) neighborhood* of a vertex, v, denoted $N(v)$, is the set of neighbors of v. The *closed neighborhood* of v, denoted $N[v]$, is equal to $N(v) \cup \{v\}$.

Given a binary matrix, M, let $size(M)$ denote the number of rows, columns and 1s in M. A sparse representation of M requires $O(size(M))$ space. When dealing with consecutive-ones ordering of matrices, we can represent them in *compact form*, where each row is given by the two columns containing the first and last 1 in the row. This gives a representation in $O(n)$ space.

CHAPTER 3

Literature Review

3.1. Relationship of Interval Graphs to Other Graph Classes

In this section, we review the literature on a set of highly structured graph classes related to interval graphs. The *chordal graphs* are the class of undirected graphs that do not have a chordless cycle of size four or more as an induced subgraph. Interval graphs are a subclass of the chordal graphs. Chordal graphs are characterized by a natural elimination order. The *comparability graphs* are those undirected graphs whose edges can be assigned orientations so that the result is a *transitive DAG*, that is, a DAG such that whenever (u, v) and (v, w) are both directed edges, so is (u, w). Such DAGs model transitive relations that arise in many combinatorial problems. A graph is an interval graph if and only if it is chordal and its complement is a comparability graph.

All of these graphs have the property that the size of a maximum clique is equal to the chromatic number. The complements of these graphs also have this property. Many otherwise NP-hard problems are polynomial on these graph classes. In addition, these classes share the property that none contains an odd chordless cycle or its complement as an induced subgraph.

This gave rise to the conjecture that all of these properties are universal to all graphs where the size of the maximum clique is equal to the chromatic number and that are hereditary with respect to this property. This class became known as the class of *perfect graphs*. This conjecture has been proven after decades of intensive research.

3.1.1. Hereditary Graph Classes.

A class of graphs is *hereditary* if, for every graph G in the class, every induced subgraph of G is also in the class. Well-known examples of hereditary graphs are bipartite graphs, which are the graphs whose chromatic number is less than or equal to two, *planar graphs*, which are graphs that can be drawn in the plane without crossing any edges, *directed acyclic graphs* (DAGs), which are directed graphs that have no directed cycles, and *forests*, which are undirected graphs that have no cycles.

When it can be shown that every graph in a hereditary class has a vertex with a certain property, this gives rise to an *elimination order*, which is obtained by repeatedly finding and eliminating a vertex that has this property. The hereditary property ensures that when one vertex with the property is deleted, the resulting induced subgraph also has a vertex with the property.

For example, every DAG has a *source*, which is a vertex that has no incoming directed edges. Iteratively deleting sources gives rise to an elimination order called a *topological sort*, which is an ordering of the vertices where all edges are directed from earlier to later vertices. Every planar graph has a vertex of degree at most five. Repeatedly finding vertices of degree at most five gives an elimination order that shows that a planar graph has $O(n)$ edges. Moreover, if each vertex is labeled with its at-most five neighbors at the time it is deleted, this gives rise to a structure that takes $O(n)$ space and can answer in $O(1)$ time whether two vertices are adjacent: this can be determined by finding whether one of them is in the list of at-most five nighbors of the other. For arbitrary graphs, structures that support an $O(1)$ time bound for this operation take $\Theta(n^2)$ space. A forest has a *leaf*, which is a vertex of degree at most one. (In fact, it is either a one-vertex graph or has at least two leaves). Iteratively removing a leaf and labeling it with its neighbor, if it has one, gives rise to an elimination ordering where every vertex has at most one neighbor among its successors, namely, its parent in a rooting of the forest.

Sometimes the existence of a certain elimination order *characterizes* the graph class. For example, DAGs are those graphs that have a topological sort. By the above observations, every DAG has a topological sort. For the converse, suppose a non-DAG, G, has a topological sort. Then G contains a directed cycle, C. Some vertex v on C must be earliest in the topological sort, but then the edge to v from its predecessor on C is directed from a later to an earlier vertex, a contradiction.

Another property of hereditary graphs is that they can always be characterized by a set of *forbidden subgraphs*. For a hereditary class C, let a *minimal non-member* of C be a graph that is not in C but where deletion of any *vertex* of the graph results in a member of C. The set F of such graphs is a well-defined, but possibly infinite, set. Then C can be characterized as exactly those graphs that contain no member of F as an induced subgraph, and F is the set of *forbidden subgraphs* for C.

Such a characterization is useful if it is easy to characterize the members of F. By definition, the class of DAGs consists of those graphs that have no directed cycle on three or more vertices. This implies that DAGs are those graphs that have no induced directed cycle on three or more vertices, which is seen as follows. If a graph has an induced directed cycle, then it is clearly not a DAG. Conversely, if G is not a DAG, then it contains a directed cycle, C. If the vertices of C induce a directed cycle, then C satisfies the claim. Otherwise, there is a directed edge e not on C but whose endpoints are on C. Then the union of $\{e\}$ and the part of C that is a directed path from the head of e to its tail gives a smaller directed cycle. A 3-cycle is an induced 3-cycle, and an application of the operation yields a smaller cycle of size at least three. Iterating this operation on each cycle thus obtained, we must eventually find an induced cycle.

It is well-known that the bipartite graphs are those graphs that have no odd cycle. By an argument similar to the one above for DAGs, they are those graphs that have no *induced* odd cycles. Similarly, the forests are those graphs that have no *induced* cycle.

A graph G is *homeomorphic* to G' if contraction of vertices of G that have degree two yields a graph that is isomorphic to G'. By a celebrated theorem from the 1920s, Kuratowski characterized planar graphs as those graphs that do not contain a subgraph homeomorphic to the complete graph, K_5, on five vertices or the complete bipartite graph, $K_{3,3}$, that has three vertices in each bipartition class [Kur30]. His use of homeomorphism is a simple way to characterize the set of minimal forbidden induced subgraphs, which would be difficult to enumerate explicitly.

Many problems that have industrial applications can be represented with graphs, allowing them to be solved using standard graph algorithms. On arbitrary graphs, however, many of those problems are NP-hard. When we can add the restriction that a graph belongs to some specified hereditary graph class, such intractable problems often yield efficient solutions. For example, the otherwise NP-hard problem of finding a longest path in a graph can be solved in linear time on DAGs, using the fact that they have a topological sort. Similarly, finding a maximum stable set is an NP-hard problem, but takes polynomial time on bipartite graphs.

Clearly, the interval graphs are hereditary, since there exists an interval model for every interval graph, and an interval model of any induced subgraph can be obtained by deleting some of the intervals from the model.

3.1.2. Chordal Graphs. Chordal graphs have a number of useful properties. The number of maximal cliques in an arbitrary graph is exponential in the worst case. Chordal graphs have at most n maximal cliques, and the sum of cardinalities of their cliques is at most $n + m$. They have a simple recognition algorithm.

In addition to having linear-time algorithms for finding the chromatic number and a maximum clique, a maximum stable set, and a minimum clique cover, they have linear-time algorithms for finding the set of maximal cliques. Chordal graphs have applications in combinatorics and linear algebra, such as in solving sparse systems of linear equations [Ros70].

Given a graph G and a vertex x, x is *simplicial* if $N[x]$ induces a complete subgraph. To obtain a useful elimination ordering on chordal graphs, we need to show that every chordal graph contains a simplicial vertex. Since we will prove this by induction, it is easier to strengthen the induction hypothesis by showing that every chordal graph is either is a clique (in which case every vertex is simplicial) or contains at least two nonadjacent simplicial vertices [Dir61, LB62], as follows.

Given two nonadjacent vertices, a, b, a *minimal a, b vertex separator* is a minimal set S of vertices whose removal cuts all paths from a to b.

We first show that every minimal a, b vertex separator S in a chordal graph induces a complete subgraph. Suppose for some chordal graph G, there exist nonadjacent vertices a and b and a minimal a, b vertex separator, S, that does not induce a complete subgraph. Then there exist $x, y \in S$ such that x and y are not neighbors. Let G_a and G_b be the connected components of $G - S$ that contain a and b, respectively. Since S is a minimal a, b separator, x and y each have neighbors in G that belong to G_a and they each have neighbors in G that belong to G_b. Therefore, there exists a path from x to y in G whose internal vertices are members of G_a; let P_a be such a path of minimum length. Let P_b be such a path of minimum length whose internal vertices are in G_b. Then $P_a \cup P_b$ is a cycle of length at least four. Since x and y are nonadjacent, xy is not a chord. The minimality of P_a implies that there can be no chord whose endpoints are both on P_a. Similarly, there can be no chord whose endpoints are both on P_b. The only remaining possibility for a chord in C is one between an internal vertex of P_a and an internal vertex of P_b. However, since S is an a, b separator, there can be no such chord. The assumption that S does not induce a complete subgraph contradicts the chordality of G.

Using this, we can now prove that every chordal graph is either complete or has two nonadjacent simplicial vertices. Suppose that some chordal graph is neither a clique nor contains two nonadjacent simplicial vertices. Let G be the smallest chordal graph that is not a clique and does not contain two nonadjacent simplicial vertices. Since G is not a clique it has two nonadjacent vertices a and b. There exists a minimal vertex separator S for a and b. Let G_a and G_b denote the components of $G - S$ containing a and b, respectively.

As a result, $S + G_a$ is a smaller chordal graph, and is either a complete subgraph or contains two nonadjacent simplicial vertices. Similarly, $S + G_b$ is a clique or contains two nonadjacent simplicial vertices. Since S induces a complete subgraph, G_a and G_b must each contain a simplicial vertex. As the two sets have no edges between them, these vertices remain simplicial in G, proving the claim.

The existence of a simplicial vertex leads to a simple algorithm for recognizing chordal graphs [FG65]. Locate a simplicial vertex and remove it from the graph; as chordal graphs are hereditary, if the original graph was a chordal graph, the new graph will always have another simplicial vertex. Eventually, either the entire graph has been removed (and G was chordal) or no simplicial vertex remains (and G was not a chordal graph). An ordering of the vertices is a perfect elimination ordering if, as each vertex in sequence is removed from the graph, the next vertex becomes (or remains) a simplicial vertex.

Moreover, the existence of a perfect elimination ordering is a characterization of chordal graphs, just as the existence of a topological sort is a characterization of the class of DAGs,

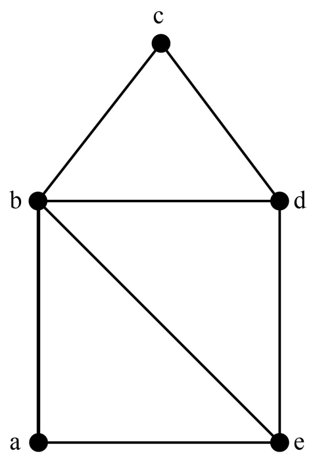


FIGURE 3.1. A chordal graph
One perfect elimination ordering is {c,d,e,b,a}

which is seen as follows. We have shown that every chordal graph has a perfect elimination ordering. Conversely, suppose some non-chordal graph has a perfect elimination ordering. Then it has a chordless cycle, C. Some vertex, v, must be earliest in the perfect elimination ordering. Its two neighbors on C are two nonadjacent vertices that appear later in the ordering, contradicting the definition of a perfect elimination ordering.

A naive algorithm for finding a perfect elimination ordering would be to check, for each vertex, whether its neighborhood is a complete subgraph; for each vertex we check this takes time proportional to the sum of the degrees of its neighbors. This gives a polynomial algorithm for recognizing chordal graphs, since this procedure will succeed on a graph if and only if it has a perfect elimination ordering.

However, we can improve this to $O(n+m)$ for the entire algorithm by using *lexicographical breadth-first search* (lex-BFS) [RTL76]. In standard BFS, a vertex is removed from the front of a queue, and its undiscovered neighbors are inserted to the back of the queue. For two vertices u and v, u gets priority for extraction from the queue if its first neighbor that is extracted from the queue is extracted earlier than the first neighbor of v that is extracted from the queue. Otherwise, they are *tied*; they are inserted at the same time to the back of the queue, in arbitrary order.

In Lex-BFS, when two vertices are tied, the tie is broken in favor of the vertex whose second neighbor that is extracted from the queue is extracted earlier, and when these are also tied, the ties are broken by the third neighbors, etc. This is similar to the lexicographic ordering of strings: the relative order of two strings is determined by the first letter, or by the second letter if the first letters are tied, or by the third letter if the first two are tied, etc.

```
Data: A graph with the label $\emptyset$ assigned to each vertex
for $i = n$ downto 1 do
    select unnumbered vertex $v$ whose label is lexicographically largest;
    assign $v$ the number $i$;
    for $w \mid w \in Adj(v)$ and $w$ unnumbered do
        Append $i$ to label($w$);
    end
end
```

Algorithm 1: Lexicographic BFS

In order to maintain the time bound, we need to be able to select the vertex with the largest label in $O(n + m)$ time over all vertices. This can be implemented by maintaining a queue of equivalence classes of vertices instead of a queue of vertices. Each equivalence class is a set of vertices that are currently tied. When a vertex is removed from the queue, one is selected arbitrarily from the equivalence class at the front of the queue, and this class is removed if it is now empty. Let v be the removed vertex. Each equivalence class is split into two consecutive equivalence classes, one containing neighbors of v and one containing non-neighbors of v, with the neighbors of v getting higher priority. (Some classes are not split if they contain only neighbors or only non-neighbors of v.) This is easily accomplished in time

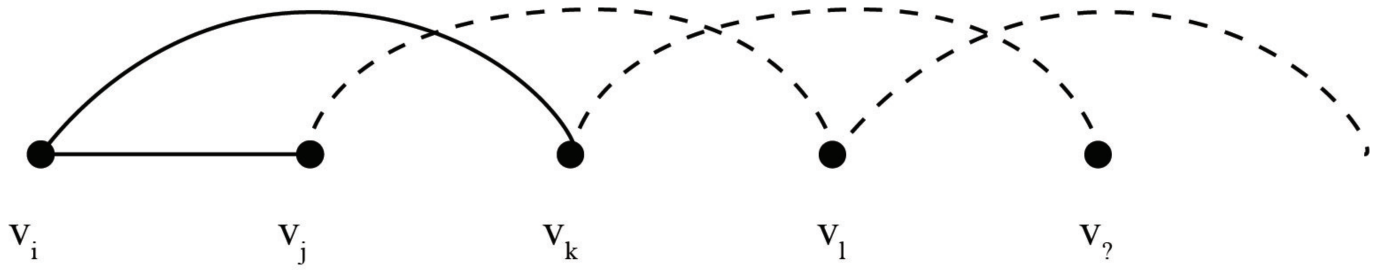


Figure 3.2. The first vertex must be simplicial

proportional to the degree of v if the equivalence classes are implemented with doubly-linked lists, for a total of $O(n + m)$ time.

We claim that the outcome is a perfect elimination ordering. Suppose this is not the case. Let σ be a minimal ordering that can be produced by this algorithm such that the first vertex in the ordering is not simplicial. Call this v_i and suppose it has neighbors v_j and v_k that do not have an edge between them; let v_j be the neighbor that comes first in the ordering. Then of the three vertices, v_k was added first and k was added to the label for v_i but not v_j. But v_j must have a larger label than v_i, since it was processed first, which means it must be adjacent to another vertex v_l with a larger number that v_i is not. But then v_l cannot be adjacent to v_k (or $\{v_l, v_k, v_i, v_j, v_l\}$ would be a chordless cycle of length 4) so why was v_k processed before v_j? It must be adjacent to yet another vertex, and so on *ad infinitum.* Since we always need to add another vertex, but the graph is finite, our original assumption was incorrect.

A similar method is called maximum cardinality search [TY84]; in this case, rather than labeling vertices and choosing the unprocessed vertex with the largest label we choose the one that is adjacent to the greatest number of already-numbered vertices. When we process a vertex we update the count on each unprocessed neighbor and mark the vertex as processed. A similar *reductio ad absurdum* proof as above shows that the result will be a perfect elimination ordering.

Interestingly, these two methods find overlapping subsets of the perfect elimination orderings [Gol80]. Rose, Tarjan, and Lueker's algorithm creates the perfect elimination ordering backwards, first choosing a vertex to leave for last, then a vertex adjacent to it to leave for next to last, and so on. Shier later provided additional methods that generate all perfect elimination orderings iteratively [Shi84]. Chandran et. al. improved this to linear time using Gray codes and characterized perfect elimination orderings as being exactly those orderings of the vertices such that for every set of vertices $\{x,y,z\}$ where x and z are nonadjacent and y belongs to a minimal x-z separator, at least one of x and z precedes y in the ordering [CIRS03].

We are now ready to prove that the number of cliques in a chordal graph is at most n, and that their sum of cardinalities is at most $n+m$, a result of Fulkerson and Gross [FG65]. Suppose G is chordal, and thus has a perfect elimination ordering. If K is a maximal clique, let v be be its earliest vertex. At the time when v is eliminated, $K = N[v]$. This relationship assigns at most one maximal clique K to each vertex v, and gives the bound of $|K| = N[v]$, yielding a sum of cardinalities of $n+m$ for the maximal cliques in a chordal graph. Moreover, this shows that G can have at most n maximal cliques.

Given a perfect elimination ordering, we can color the vertices in a greedy manner. Process the vertices from right to left; for each vertex encountered, assign it the first color not already taken by a neighbor to its right. A vertex will then receive color k only if it and its neighbors that follow it in the ordering form a clique of size k. The number of colors used for the entire graph is then equal to the size of the largest clique (and is thus optimal), giving us both the chromatic number of the graph and the size of the maximum clique, and a vertex of color k and its neighbors to the right are a maximum clique.

Consider the problem of finding a maximum stable set in a graph. Take the first vertex in the perfect elimination ordering and add it to the set S, then remove all of its neighbors from the ordering and recurse. No vertex added to S is a neighbor of any other vertex in S so this is a stable set. Each of the neighborhoods removed from the ordering is a complete subgraph, so at most one vertex from that neighborhood can be a member of any stable set. S contains one vertex from each of these neighborhoods, so S is a maximum stable set. Since the size of a stable set is a lower bound on the clique cover number and the $|S|$ removed complete subgraphs cover V, they are a minimum clique cover.

3.1.3. Comparability Graphs. A *partial order*, or *poset relation*, is a set V along with a binary relation R that has the following properties:

- Reflexivity: $(a, a) \in R$ for all $a \in V$;
- Antisymmetry: $(a, b) \in R, a \neq b \Longrightarrow (b, a) \notin R$;
- Transitivity: $(a, b), (b, c) \in R \Longrightarrow (a, c) \in R$.

An example is the $\subseteq$ relation on the power set of a set X, where the elements of V are the subsets of X.

A poset relation can be represented with a directed graph that has one vertex for each element of V and a directed edge (a, b) if $a \neq b$ and $(a, b) \in R$. The directed graph is a transitive graph that must be a DAG, since it is transitive and antisymmetric.

For $a, b \in V$, a and b are *comparable* if either $(a, b) \in R$ or $(b, a) \in R$; otherwise, they are incomparable. For example, if X and Y are members of the power set, X and Y are comparable in the $\subseteq$ relation if one of them is a subset of the other. A graph is a *comparability graph* if it models the comparability relation of some poset on V with an edge between a, b if and only if $a \neq b$ and a and b are comparable. The *transitive orientation* problem consists of finding such a poset relation, given a comparability graph. One must assign orientations to the edges so that the resulting digraph is transitive. A linear time bound was first given by McConnell and Spinrad [MS99]. (See Figure 3.3.) Because of McConnell and Spinrad's algorithm, this gives a linear time bound for finding a minimum coloring and maximum clique of a comparability graph.

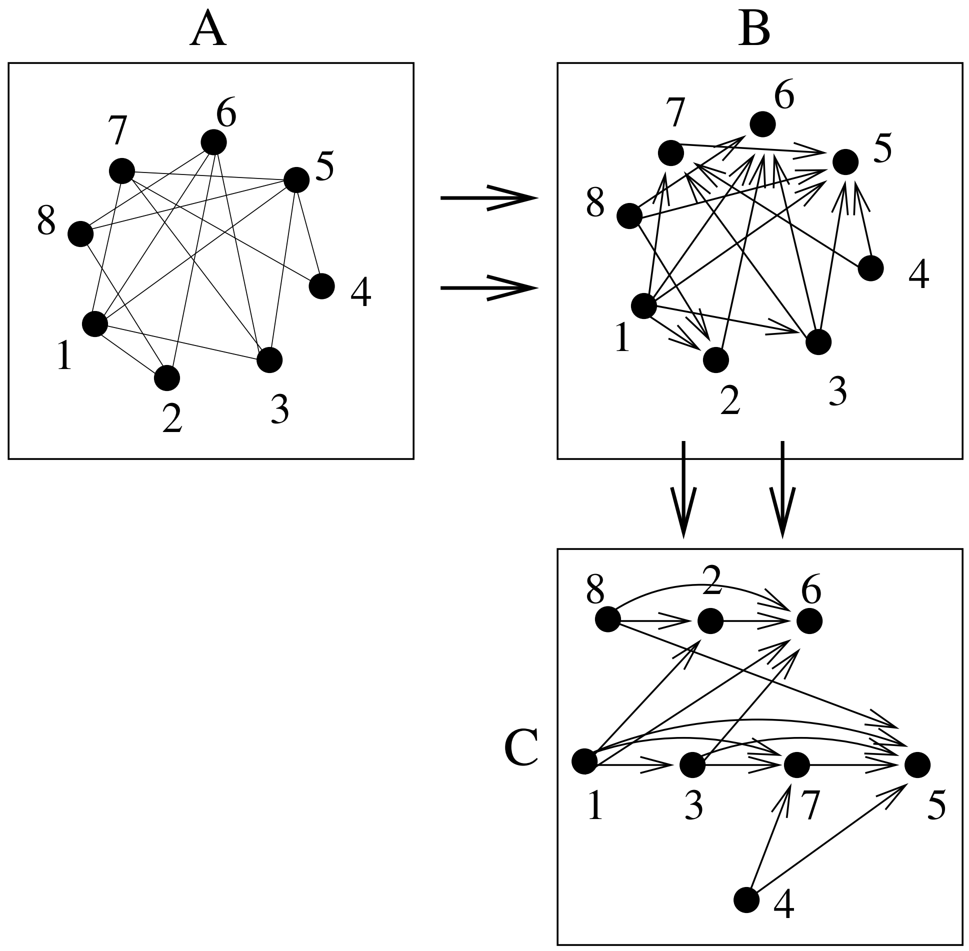


FIGURE 3.3. Finding a transitive orientation of a comparability graph

Since the orientation is acyclic, the resulting orientation of any complete subgraph $G[K]$ contains a directed path of length $|K|$. Conversely, given a transitive orientation of a comparability graph, all of the vertices on a directed path form a complete subgraph, due to the transitive edges. The transitive orientation is a DAG, and it takes $O(n+m)$ time to label each vertex with the length of the longest path originating at the vertex in a DAG, so this gives an algorithm for finding a maximum clique. Moreover, when two vertices are adjacent, one of them receives a higher number than the other in the resulting numbering. Therefore, the numbering is a coloring of the graph. It must be a minimum coloring, because it uses a number of colors equal to the size of a maximum clique, and all vertices of this clique receive different colors in every valid coloring. (See Figure 3.4.)

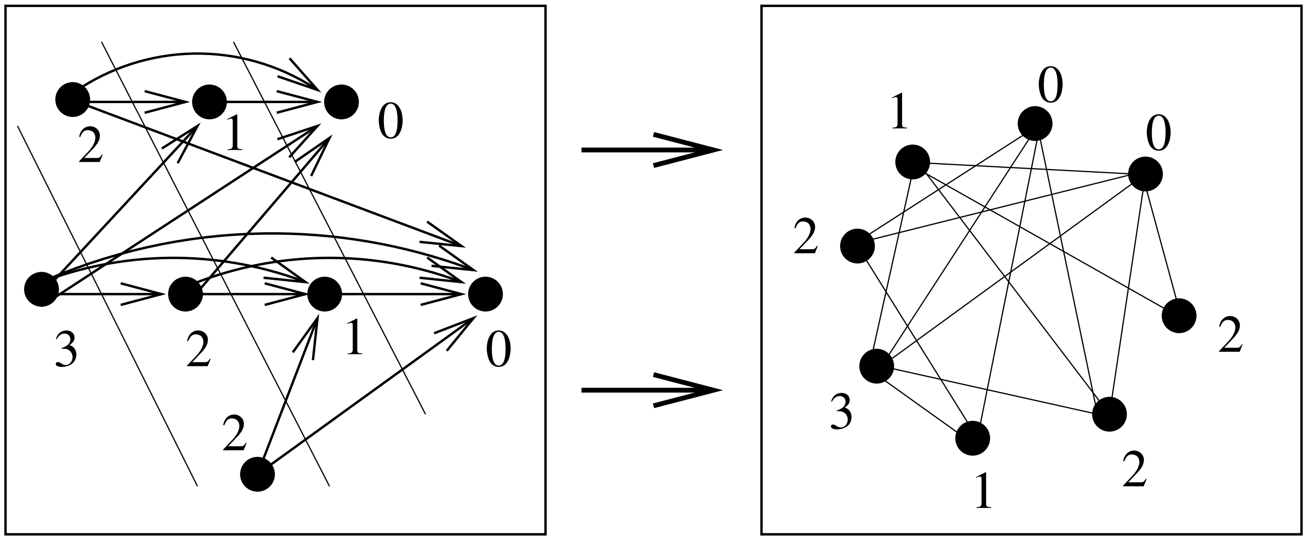


Figure 3.4. Finding a minimal coloring of a comparability graph

3.1.4. Interval Graphs. Interval graphs are a subclass of chordal graphs, which is seen as follows. Suppose an interval graph contains a chordless cycle C. In an interval model, let x be the vertex on the cycle with the leftmost right endpoint and y be the vertex on the cycle with the rightmost left endpoint. Since some pairs of vertices on C are nonadjacent, the right endpoint of x precedes the left endpoint of y. Let P_1 be one of the paths from x to y on C, and let P_2 be the other. In the interval model, the intervals corresponding to vertices of P_1 must cover the gap between the right endpoint of x and the left endpoint of y. This is also true for P_2. This implies a chord between a vertex of P_1 and a vertex of P_2, contradicting the chordlessness of C.

That interval graphs are chordal graphs is also implied by the inclusion of G_{III} in the Lekkerkerker-Boland graphs. That interval graphs are a proper subclass of chordal graphs can be seen from the other Lekkerkerker-Boland graphs, all of which are chordal but are not interval graphs.

The complements of interval graphs are a subclass of the comparability graphs, which is seen as follows. In an interval model of an interval graph $G = (V, E)$, let us define the relation $\{(x, y) | x, y \in V$ and the right endpoint of x precedes the left endpoint of $y\}$. This relation is clearly a transitive orientation of the complement of G. That the complements of interval graphs are a proper subclass of the comparability graphs is seen from the fact that a six-cycle is a comparability graph, but its complement is not chordal, hence not an interval graph.

It is therefore surprising that a graph is an interval graph if and only if it is chordal and its complement is a comparability graph [GH64].

Another characterization stems from the fact that interval graphs are hereditary, hence they can be characterized in terms of their forbidden subgraphs. This set of graphs is given by Lekkerkerker and Boland [LB62], and is depicted in Figure 1.2. The Lekkerkerker-Boland graphs are minimally non-interval graphs of two types: the chordless cycles and the *asteroidal triples*. An asteroidal triple is a set of three pairwise nonadjacent vertices such that there exists a path between any two of the three that does not intersect the neighborhood of the third.

Because interval graphs are chordal, they have a perfect elimination ordering. Given an interval model of an interval graph G, one perfect elimination ordering is to arrange the vertices by right endpoint, since it is easy to see that the vertex with the first right endpoint is a simplicial vertex. The elementary greedy algorithm for finding an stable set and a minimum coloring of an interval model that is given in elementary texts such as [CLRS09] and [KT06] are the special case of the algorithms described above for these problems on chordal graphs, applied to this particular elimination ordering. Therefore, they also find a maximum clique and a minimum clique cover. That the size of the minimum clique cover is equal to the size of the stable set gives an alternative proof of correctness to the maximum stable set algorithm given in those texts.

3.1.5. Perfect Graphs.

An induced subgraph that is a cycle of length $2n+1$ for $n \geq 2$ is known as an *odd hole* and an induced subgraph that is its complement is known as an *odd anti-hole*.

The clique number of a graph is a lower bound on the chromatic number, since the vertices of a maximum clique must have different colors in any coloring of a graph. Similarly, the stability number is a lower bound on the clique cover number. These lower bounds are not always tight, however. For example, an odd hole has a clique number of two and a clique cover number of three. It also has a stability number of n and a clique cover number of $n+1$. In each graph class we've examined so far, however, the chromatic number of a graph is equal to its clique number, and the stability number is equal to its clique cover number. Since the graph classes we have examined are hereditary, it follows that none of them can have an odd hole or an odd anti-hole. Finding a minimum coloring, maximum clique, minimum clique color, or maximum stable set is NP-hard on an arbitrary graph, yet for each of the classes discussed so far, there is a polynomial time algorithm for each of these problems.

These observations motivated the definition of the class of *perfect graphs* in the early 1960s, to study the relationships between these properties. A graph is *perfect* if its clique number is equal to its chromatic number and it is hereditary with respect to this property. Note that the perfect graphs also include bipartite graphs, as well as many other classes of hereditary structured graphs, such as the *line graphs* of bipartite graphs, *distance-hereditary graphs* [HM90], *cographs* [CLS81], *permutation graphs* [Gol80], *probe interval graphs* [MM99, ZSF$^+$94, JS01, MN09], and *threshold tolerance graphs* [MRT88], as well as the complements of graphs in these classes. In the early 1960's, Berge conjectured the following [Ber61]:

(1) **The Perfect Graph Conjecture**: A graph is perfect if and only if its complement is perfect, that is, its stability number is equal to its clique cover number and it is hereditary with respect to this property.

(2) **The Strong Perfect Graph Conjecture:** A graph is perfect if and only if it does not have an odd hole or odd anti-hole.

(3) Finding a maximum clique, minimum coloring, maximum stable set and minimum clique cover takes polynomial time on every perfect graph.

Among others, Fulkerson worked intensively on the perfect graph conjecture for a number of years [Ful69, Ful71, Ful72]. One day, he learned that Laslo Lovasz, an undergraduate who had not heard of Fulkerson's work, had proven the theorem [Lov72]. Spurred on by this knowledge, he succeeded in proving it himself within a day, too late to receive credit. The third conjecture was proved by Groetschel, Lovasz and Schrijver [GLS88] some years later. The strong perfect graph conjecture implies the perfect graph conjecture, and was considered one of the most important open problems in graph theory until it was proven in the early 2000s [CRST06]. These conjectures are now known as the Perfect Graph Theorem and the Strong Perfect Graph Theorem.

Until the proof of the perfect graph theorem, the class of graphs with no odd holes or odd anti-holes was known as the *Berge graphs*, but it was not known whether this class was equal to the class of perfect graphs. Its proof rendered trivial the proofs of perfection of graphs in many graph classes, as well as their complements, since it reduced the proof to just establishing that no graph in the class can have an odd hole or odd anti-hole. For example, König's celebrated theorem from the 1930s, which is that the stability number and the clique cover number are equal in bipartite graphs [Ko31] now had a trivial proof, which was the observation that bipartite graphs can have no odd hole or odd anti-hole. A much simpler alternative to the proof given above that chordal graphs are perfect is the observation that a hole is a chordless cycle and that an odd anti-hole contains a chordless cycle. A much simpler alternative to the proof given above that a comparability graph is perfect is the observation that an odd hole or odd anti-hole is not transitively orientable.

3.2. Relevant Algorithms

3.2.1. Recognition Algorithms for Interval Graphs. Booth and Lueker's algorithm recognizes whether a graph G is an interval graph by first testing whether it is chordal using the algorithm of [RTL76]. If it is not, then it cannot be an interval graph. Otherwise, as we have seen, the algorithm of [RTL76] gives a perfect elimination ordering, and this can be used to find the maximal cliques in $O(n+m)$ time. The question now reduces to whether the clique matrix has a consecutive-ones ordering.

To resolve this question, they developed an $O(size(M))$-time algorithm that finds a consecutive-ones ordering of a given binary matrix M, or else determines that this is not possible. By applying this algorithm to the clique matrix of G, they either obtain a consecutive-ones ordering, which gives an interval model of G, or else determines that the clique matrix does not have a consecutive-ones ordering, in which case, G is chordal but not an interval graph.

Their algorithm proceeds row-by-row through the matrix, updating a structure, called a *PQ tree*, that gives an implicit representation of all consecutive-ones orderings of the part of the matrix examined so far, or else determines that inclusion of the next row results in a matrix that has no consecutive-ones ordering. The PQ tree represents the consecutive-ones orderings as follows. The leaves of the tree are one-element sets; each leaf contains one of the columns of the matrix, and each column is contained in one leaf. Each internal node is the set of columns given by the union of its leaf descendants; equivalently, it is the union of its children. The root of the tree is the set of all columns (see figure 3.5).

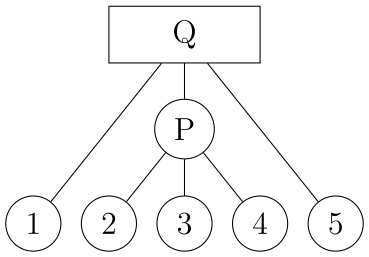


FIGURE 3.5. A PQ tree
Columns 2, 3, and 4 are children of a P node and can have any relative order. Column 1 will always be at one end of the matrix, and column 5 will always be at the other end. This enforces the constraints that the sets {1,2,3,4},{2,3,4}, and {2,3,4,5} are consecutive.

Each internal node is either a *P node* or a *Q node*. In the tree, the children of a Q node can be ordered in one of two ways, where one is the reverse of the other. This order or its reverse is given as part of the representation. The children of a P node can be ordered arbitrarily. By choosing an assignment of orderings of children for each internal node of the tree, one obtains a left-to-right ordering of the leaves, which gives one of the consecutive-ones orderings of the columns of the matrix. Conversely, the set of consecutive-ones orderings of the columns of the matrix are precisely those obtainable in this way. For any consecutive-ones matrix, the PQ tree is unique.

Booth and Lueker's algorithm takes $O(n)$ time to update the PQ tree when the next row of the matrix is considered, or else to reject the consecutive-ones property, where n is the number of columns. However, the amortized bound is linear in the size of the submatrix considered so far, giving an $O(size(M))$ algorithm for recognizing the consecutive-ones property. For a chordal graph, the size of the clique matrix is $O(n+m)$, so this gives a linear-time algorithm for recognizing interval graphs.

3.2.2. Incremental and Dynamic Algorithms.

An incremental algorithm for determining whether a graph lies in a graph class adds one vertex at a time and determines whether the new graph lies in the class. An advantage occurs when the incremental step can be carried out much more efficiently than it can by applying a non-incremental algorithm to the new graph, using knowledge about the graph obtained in previous incremental steps. Some incremental algorithms are fully dynamic: given any graph in the class, they can determine whether the deletion or addition of an edge or vertex (rather than only the addition of a vertex) results in another graph that also resides in the class.

Such algorithms are useful for finding a maximal induced subgraph of a graph that lies in the class. For interval graphs, this is useful for genomic applications, such as Benzer's, that theoretically produce an interval graph, but rarely do so in practice because of errors in the data. Large subgraphs that are interval graphs can be found using a dynamic recognition algorithm for interval graphs.

Incremental algorithms (some of which are fully dynamic) exist for many classes of graphs, including chordal graphs [Iba99, BSS05, kY06, BHV06], cographs [SS04], directed cographs [CP06], distance-hereditary graphs [CT07, GP07], P_4-sparse graphs [NPP06], and proper interval graphs [Sha01].

3.2.3. Algorithms for Finding Tucker Submatrices and Lekkerkerker-Boland Subgraphs.

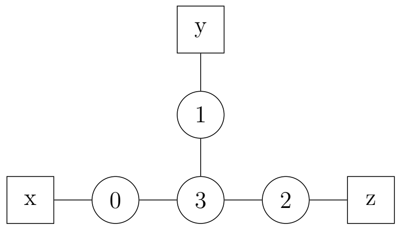
Booth and Lueker's algorithm recognizes interval graphs by recognizing whether adding a row to the clique matrix causes it to lose the consecutive-ones property. The Tucker matrices are a set of five minimal forbidden submatrices that cannot be consecutive-ones ordered [Tuc72].

An O($\Delta^3 m^2 n(m+n^2)$) algorithm for finding a Tucker submatrix in a matrix that does not have the consecutive-ones property was given by [DGN10], where Δ is the maximum number of 1s in any row. An O(n*size(M)) algorithm was obtained by [CST12]. A O(size(M)) (linear) time bound was recently obtained by Lindzey and McConnell [LM14].

Given a non-interval graph G, Lindzey and McConnell's algorithm first determines whether the graph is chordal, using the algorithm of [RTL76]. If it is not, the algorithm of [TY85] returns a chordless cycle in linear time, which is a G_{III}. Otherwise, they reduce the problem to that of finding a Tucker submatrix of the clique matrix, which must exist, since otherwise G would be an interval graph. Most of the details of their algorithm consist of the solution to this problem, which is of interest in its own right. Once they find a Tucker submatrix, they exploit the relationship between the Tucker matrices and the Lekkerkerker-Boland graphs that have the Tucker matrices in their clique matrices, depicted in Figure 3.6. They prove that whenever a Tucker matrix occurs in the clique matrix of a chordal graph, it can always be extended to the clique matrix of an induced subgraph in the way described by the picture. This is not immediate; it is not even true for clique matrices of non-chordal graphs.

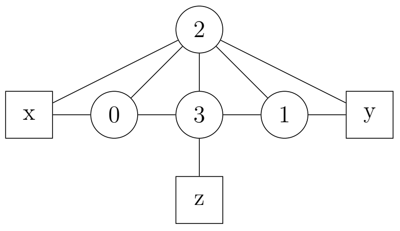
An important point is that these algorithms are not incremental; incremental algorithms for these problems are the main new contributions of the present work.

$$\begin{array}{c|cccccc|}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
x & 1 & 0 & 0 & 0 & 0 & 0 \\
y & 0 & 0 & 1 & 0 & 0 & 0 \\
z & 0 & 0 & 0 & 0 & 1 & 0
\end{array}$$



(A) M_{IV}/G_I

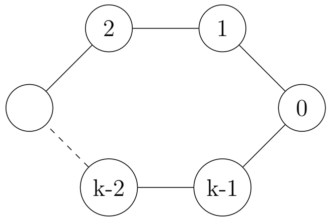
$$\begin{array}{c|ccccc|}
0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 \\
2 & 1 & 1 & 1 & 1 & 0 \\
3 & 1 & 0 & 0 & 1 & 1 \\
\hline
x & 0 & 1 & 0 & 0 & 0 \\
y & 0 & 0 & 1 & 0 & 0 \\
z & 0 & 0 & 0 & 0 & 1
\end{array}$$



(B) M_V/G_{II}

$$\begin{array}{c|ccccccc|}
0 & 1 & 1 & & & & & \\
1 & & 1 & 1 & & & & \\
2 & & & 1 & 1 & & & \\
\vdots & & & & & \ddots & \ddots & \\
k-2 & & & & & & 1 & 1 \\
k-1 & 1 & 0 & \dots & & 0 & 0 & 1
\end{array}$$

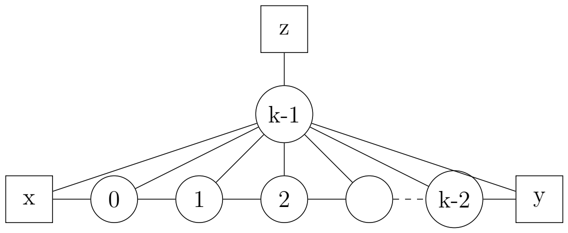
(C) $M_I(k)$



(D) $G_{III}(n), n \geq 4$

$$\begin{array}{c|ccccccc|}
0 & 1 & 1 & & & & & \\
1 & & 1 & 1 & & & & \\
2 & & & 1 & 1 & & & \\
\vdots & & & & \ddots & \ddots & & \\
k-2 & & & & & 1 & 1 & 0 \\
k-1 & 0 & 1 & \dots & 1 & 1 & 0 & 1
\end{array}$$

(E) $M_{III}(k), k \geq 3$



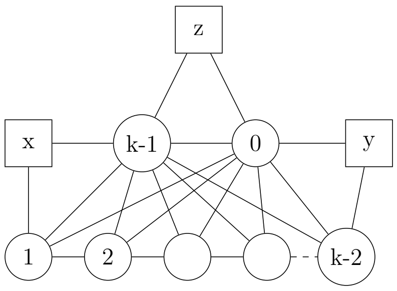
(F) $G_{IV}(n), n \geq 6$

$$\begin{array}{c|ccc|}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 \\
\hline
x & 0 & 1 & 0 \\
y & 0 & 0 & 1 \\
z & 1 & 0 & 0
\end{array}$$

(G) $M_I(3)$

$$\begin{array}{c|cccccc|}
0 & 0 & 1 & 1 & \dots & 1 & 1 \\
1 & 1 & 1 & & & & \\
2 & & 1 & 1 & & & \\
\vdots & & & \ddots & \ddots & & \\
k-2 & & & & 1 & 1 & 0 \\
k-1 & 1 & \dots & 1 & 1 & 0 & 1 \\
\hline
x & 1 & & & & & \\
y & & & & & 1 & \\
z & & & & & & 1
\end{array}$$

(H) $M_{II}(k), k \geq 4$



(I) $G_V(n), n \geq 6$

Figure 3.6. The Tucker matrices as they relate to the Lekkerkerker-Boland graphs

CHAPTER 4

A Fully Dynamic Certifying Algorithm for Recognition of Consecutive-Ones Matrices

In this chapter, we present a fully dynamic certifying algorithm to recognize consecutive-ones matrices. Given an arbitrary consecutive-ones matrix and a row or column to add or remove, we return either a consecutive-ones ordering of the new matrix or a proof, in the form of a Tucker submatrix, that this is impossible.

4.1. Basic Operations

As discussed in chapter 3.2.1, a PQ tree is used to represent all possible consecutive-ones orderings of a matrix in O(n) space. A variant of the PQ tree with additional labels, which is related to the MPQ tree of [KM89], is described in [McC04]. Each internal node of the PQ tree can be considered to be a set of columns, namely, the columns corresponding to leaf descendants. A row's *label* is a label on the least common ancestor of the cliques that contain it. It is shown that for every child of this node, the vertex is either contained in every clique descendant of the child or is contained in none of them. If the least common ancestor is a P node, it is contained in every clique descendant of every child. If it is a Q node, it is contained in the clique descendants of a consecutive set of children. In either case, the label indicates the first and last child whose leaf descendants contain the vertex.

Each internal node carries a list of such labels, and collectively they form the interval model of a submatrix. If the node is a Q node, this submatrix has a unique consecutive-ones ordering, up to reversal. Let us call this the *labeled PQ tree*, and the labels *interval labels.*

The following are the elementary operations we can use:

(1) Given a PQ tree and a set $\mathcal{C}$ of columns, modify the tree to get a new PQ tree that enforces an additional constraint that $\mathcal{C}$ is consecutive, or report that this is impossible. This is the incremental step of Booth and Lueker's PQ-tree construction algorithm, and it takes $O(n)$ time [BL76].

(2) Given a consecutive-ones matrix M in compact form, find the PQ tree for M. This takes $O(n)$ time using an algorithm from [MdM05].

(3) Given the labeled tree, find the matrix M that it represents, in compact form, in $O(n)$ time. This can be done by labeling each node with its first and last leaf descendants in postorder, and reconstructing a row $\mathcal{R}$, given its label, to begin at the first leaf descendant of the leftmost child contained in $\mathcal{R}$ and end at the last leaf descendant of the rightmost child contained in $\mathcal{R}$.

(4) Given a PQ tree, turn it into a labeled PQ tree in $O(n)$ time. Use Harel and Tarjan's least-common ancestor algorithm [HT84] to find the least common ancestor of the endpoints of each row, and the child that contains the right endpoint of each row. Reversing the model and repeating this gives the child containing the left endpoint. The first and last column labels of each node are obtained by induction in postorder.

4.2. Dynamic Updates on Consecutive-Ones Matrices

4.2.1. Deletion of rows and columns and insertion of rows.

Given a consecutive-ones ordering of a matrix, removing a row or column (and adjusting column labels as required) gives a consecutive-ones ordering of the new matrix; we can then use operation 2 to get the new PQ tree and operation 4 to get the new labeled tree. No further work is required.

To insert a row, take the PQ tree of the matrix and apply operation 1; reject the row if this would result in the matrix not having the consecutive-ones property. Otherwise, get the new PQ tree and apply operation 4 to turn it into the labeled PQ tree.

4.2.2. Inserting a column. Let c be the column to be added and let $\mathcal{R}$ be the set of rows that have a 1 in c. Suppose this can be done without undermining the consecutive-ones property. In a consecutive-ones ordering of $M+c$, there may be a set $\mathcal{R}_1$ of rows that have their right endpoints in the new column and a set $\mathcal{R}_2$ of rows that have their left endpoints in the new column. If these sets are nonempty then they must be ordered by containment, and prior to the addition of $\mathcal{R}$ they are disjoint.

Given a consecutive-ones ordering of M before c is added, let R_1 be the row of $\mathcal{R}$ with the leftmost right endpoint and let R_2 be the row of $\mathcal{R}$ with the rightmost left endpoint. $\mathcal{R}_1$ and $\mathcal{R}_2$ are both nonempty if and only if R_1 and R_2 are disjoint, in which case we can assume without loss of generality that $R_1 \in \mathcal{R}_1$ and $R_2 \in \mathcal{R}_2$. Then the elements of $\mathcal{R}_2$ are those rows whose left endpoints lie to the right of R_1, while the elements of $\mathcal{R}_1$ are those rows whose right endpoints lie to the left of R_2.

Let R_1' and R_2' be minimal elements of $\mathcal{R}_1$ and $\mathcal{R}_2$ respectively and add a temporary row $R_1' \cup R_2'$ to the matrix. The new matrix has the consecutive-ones property, because $M + c$ has the consecutive-ones property and adding c requires that R_1' and R_2' meet at c. Find the consecutive-ones ordering and then discard the temporary row. Since R_1' and R_2' are minimal members of $\mathcal{R}_1$ and $\mathcal{R}_2$, every row of $\mathcal{R}_1$ is now consecutive with every row of $\mathcal{R}_2$ and c must be inserted between them.

Suppose that inserting c disrupts the consecutive-ones ordering of the matrix. If it disrupts the consecutiveness of 1s in a row R not contained in c, then it undermines the consecutive-ones property because R intersects both $\mathcal{R}_1$ and $\mathcal{R}_2$, which must abut each

$$\begin{array}{c|ccccc|} & & & c & & \\ R_1 & 1 & 1 & 1 & 0 & 0 \\ R_1 & 0 & 1 & 1 & 0 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 1 \\ & 0 & 1 & 1 & 1 & 0 \end{array}$$

(A) The new column is successfully inserted

$$\begin{array}{c|ccccc|} & & & c & & \\ R_1 & 1 & 1 & 1 & 0 & 0 \\ R_1 & 0 & 1 & 1 & 0 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 1 \\ R & 0 & 1 & 0 & 1 & 0 \end{array}$$

(B) Disruption of 1s in a row not containing c

$$\begin{array}{c|ccccc|} & & & c & & \\ R_1 & 1 & 1 & 1 & 0 & 0 \\ R_1 & 0 & 1 & 1 & 0 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 0 \\ R_2 & 0 & 0 & 1 & 1 & 1 \\ R' & 1 & 0 & 1 & 0 & 0 \end{array}$$

(C) Disruption of 1s in a row that contains c

FIGURE 4.1. Inserting the new column

other; this contradicts $M + c$ being a consecutive-ones matrix. On the other hand, if c disrupts consecutiveness of 1s in a row R' that contains c, then R' must belong to $\mathcal{R}_1$ or $\mathcal{R}_2$, contradicting that $\mathcal{R}_1 \cup \mathcal{R}_2$ is consecutive-ones ordered. So c can be inserted between $\mathcal{R}_1$ and $\mathcal{R}_2$ and we can then compute the new labeled PQ tree in $O(n)$ time.

Suppose one of $\mathcal{R}_1$ or $\mathcal{R}_2$ is empty. Let $\mathcal{I}$ be the intersection of rows of $\mathcal{R}$ in the consecutive-ones ordering of M before c is inserted.

Suppose the set $\mathcal{S}$ of rows that contain $\mathcal{I}$ and are not members of c is nonempty. Then c can only be inserted at one extreme end of $\mathcal{I}$, missing the members of $\mathcal{S}$, which must all have endpoints at that extreme. If two members of $\mathcal{S}$ properly overlap, then $M + c$ cannot have the consecutive-ones property. Otherwise, the rows in $\mathcal{S}$ are ordered by containment. Let S be a maximal member of $\mathcal{S}$. Insert a temporary row $S \backslash \mathcal{I}$ and verify that the new matrix has the consecutive-ones property. If it does not, then $M + c$ cannot have the consecutive-ones property because S occupies columns on either side of $\mathcal{I}$ in every consecutive-ones arrangement and c must be inserted inside of or next to $\mathcal{I}$. If it does, obtain a consecutive-ones ordering and discard the temporary row. All members of $\mathcal{S}$ now end at one endpoint of $\mathcal{I}$. If a row S' that is not a member of c spans this endpoint and the adjacent column outside of $\mathcal{I}$, then $M + c$ does not have the consecutive-ones property because $\mathcal{S}$ and S' properly overlap and jointly cover $\mathcal{I}$ in every consecutive-ones ordering. Otherwise, c can be inserted next to $\mathcal{I}$.

Suppose $\mathcal{S}$ is empty. Let $\mathcal{A}$ be nonmembers of c that intersect $\mathcal{I}$. Let c_1 and c_2 be the columns immediately to the left and right of $\mathcal{I}$ in the current ordering. If every consecutive pair of columns in $\mathcal{I} \cup \{c_1, c_2\}$ share a member of $\mathcal{A}$, then this is true in every consecutive-ones ordering, so c cannot be inserted between any two of them and $M + c$ is not a consecutive-ones matrix. If not, then let A_1 be a member of $\mathcal{A}$ that contains c_1 and let A_2 be a member of $\mathcal{A}$ that contains c_2, if they exist. Since no nonmember of c contains $\mathcal{I}$, $A_1 \neq A_2$ and each of these properly overlaps $\mathcal{I}$, forcing c_1 and c_2 to opposite sides of $\mathcal{I}$ in every consecutive-ones ordering of M and preventing c from being inserted adjacent to $\mathcal{I}$, so c must be inserted between a pair of columns of $\mathcal{I}$ do not share a member of $\mathcal{A}$. Otherwise we insert c between two elements of $\mathcal{I} \cup \{c_1, c_2\}$ that do not share a member of $\mathcal{A}$, giving us a consecutive-ones ordering of $M + c$, which we can then use to update the PQ tree in $\mathrm{O}(n)$ time.

4.3. Returning a Certificate

4.3.1. Summary of [LM14]. As previously mentioned, Lindzey and McConnell gave a (non-incremental) linear-time ($\mathrm{O}(\text{size}(M))$) algorithm for finding the Tucker submatrix in a matrix that does not have the consecutive-ones property. We briefly describe their algorithm, then change it to be an incremental algorithm that can be run in $\mathrm{O}(n \log n)$ time.

$$\begin{array}{c|ccccccc|}
 & 1 & 2 & & 3 & 4 & 5 & \\
c & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
a & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
d & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
b & 0 & 0 & 1 & 1 & 1 & 0 & 1
\end{array}$$

FIGURE 4.2. A rotation of M_V, with extra columns

Given a matrix M that does not have the consecutive-ones property, we find a submatrix M' of M that is an ordering of a subset of the rows of M and also does not have the consecutive-ones property. The way we find M' is to find the smallest prefix of M that does not have the consecutive-ones property; every Tucker matrix in the prefix must contain the final row. We move that row to the beginning of the matrix and recurse on the new prefix to find another row that must also belong to every Tucker matrix. We do this five times, or until there are no remaining rows to be processed. If only four rows remain, then those rows make up an M_{IV} or M_V, and it's trivial to choose the appropriate columns. Otherwise, the remaining matrix contains a Tucker matrix that is not M_{IV} or M_V; additionally, every such Tucker matrix contains the five selected rows.

If at least five rows remain, then the submatrix contains an M_I, M_{II}, or M_{III}. The overlap graphs of all three of these are simple cycles, which means that if we remove any one of the first five rows (call it Z), the rest of its connected component in the overlap graph forms a chordless path and has the consecutive-ones property. Additionally, the path contains nonadjacent rows A and B such that A and B each contain a 1 of Z and a 0 of Z lies between them in any consecutive-ones ordering of the path. We choose the row that has the leftmost right endpoint in the path and contains a 1 of Z to be A and choose the row with the rightmost left endpoint that contains a 1 of Z to be B; let P be the chordless path from A to B. If no such A and B are found, we choose a different row (from the five we know to belong to the Tucker matrix) to be Z; we know that at least one of the five will offer valid choices for A and B. The addition of Z would make the matrix lose the consecutive-ones property, so a 0 of Z lies between the two 1s. This is a chordless path, so the ordering is unique up to reversal and A and B will always have a 0 of Z between them when the path is consecutively ordered. Adding Z may create chords in the overlap graph other than on A and B, so $P \cup Z$ may not be the rows of the Tucker matrix. However, taking the minimal prefix P_1 of P such that $P_1 \cup Z$ does not have the consecutive-ones property, and then taking the minimal suffix P_2 of P_1 such that $P_2 \cup Z$ does not have the consecutive-ones property gives a minimal set of rows $P_2 \cup Z$ that does not have the consecutive-ones property. Finding the correct subset of columns of those rows gives a Tucker matrix.

4.3.2. Finding the submatrix. Given the matrix M that has the consecutive-ones property and the new row Z to be added, we need to find the minimal prefix of M such that $M \cup Z$ does not have the consecutive-ones property. Given a PQ tree for the first k rows, we attempt to add Z to the tree; if we can, then we do not yet have the entire prefix and instead of actually adding Z we need to try a longer prefix. Making this check takes O(n) time; in fact, making this check after each row is added is Booth and Lueker's algorithm. Instead, we do a binary search of M using this as the probe operation; this allows us to find the minimal prefix within O(log n) attempts, for a total runtime of O(n log n).

Let M_k be the first row to which we could not add Z. Every Tucker matrix in the submatrix M′ = M_1,M_2,...,M_k,Z contains both M_k and Z. Rename Z to Z_1 and M_k to Z_2 and repeat the process on the submatrix M_1,M_2,...,M_{k-1}, this time finding the longest prefix to which we cannot add both Z_1 and Z_2. We stop after finding Z_5 or being unable to do so; each of the selected rows is now part of every Tucker matrix over the remaining submatrix. Each iteration takes O(n log n) time, for a total of O(n log n) over O(1) iterations. Again, if fewer than five rows were found, these are all the rows of an M_{IV} or M_V and choosing the columns is trivial. If not, we proceed to finding the BFS of the overlap graph.

4.3.3. Finding the BFS of the overlap graph. In [LM14], running BFS on the overlap graph takes O(size(M)) time because for each row we have to check every column contained in that row to see if any other rows start or end in that column. This means that we do O(1) work for each column that does not result in adding a row to the BFS queue. Over all rows, this takes time proportional to the number of rows (for adding new rows to the queue) plus time proportional to the number of 1s in the matrix (for failing to add 1s to the queue), for a total of O(size(M)). By avoiding the extra checks, we can improve this to O(n log n).

We maintain a queue of rows that have been seen (that is, properly overlap an already-processed row) but have not yet been processed; whenever a new row is seen, it is added to the queue. In order to maintain our time bound, we need to spend no more than O(log n) amortized time for each vertex we add to or remove from the queue, regardless of how many other rows it overlaps.

As in [LM14], label each column c_i with a set of rows that have their left endpoint at that column and radix sort them in descending order of right endpoint; call this list L_i. Similarly, create a list of those rows whose right endpoints are at c_i, sorted by left endpoint in ascending order; call this R_i.

Given these lists, let the rightward column key of c_i be the maximum right endpoint in the list L_i; this is the right endpoint at the front of the list. We store all rightward keys in a Sleator-Tarjan path [ST83], which permits finding the maximum element, cutting, concatenating, and adding a constant to all elements of the list in O(log n) amortized time. This permits us two new operations, each of which requires O(log n) amortized time:

```
Data: A consecutive-ones ordered matrix $M$ with columns $c_1, c_2, ...$ and a starting
      row $S$
Result: A BFS tree on the overlap graph of rows of $M$, rooted at $S$
Let $Q$ be an empty queue of rows;
Enqueue $S$ to $Q$;
Let $\mathcal{R}$ be the rows of $M$ other than $S$;
for $i$ is a column subscript do
    Let $R_i$ be a doubly-linked list of rows in $\mathcal{R}$ whose right endpoint is in $c_i$, sorted in
    ascending order of left endpoint;
    Let $L_i$ be a doubly-linked list of rows in $\mathcal{R}$ whose left endpoint is in $c_i$, sorted in
    descending order of right endpoint;
end
while $Q$ is not empty do
    Dequeue a row $R$ from $Q$;
    Let $[j,k]$ be the compact representation of $R$;
    while the first element $R'$ in some $R_q \in \{R_j, R_{j+1}, ..., R_{k-1}\}$ has a left endpoint to
    the left of $c_j$ do
        Remove $R'$ from $R_q$;
        Let $h$ be the left endpoint of $R'$;
        Remove $R'$ from $L_h$;
        Assign $R$ as the parent of $R'$;
        Enqueue $R'$ to $Q$;
    end
    while the first element $R'$ in some $L_q \in \{L_{j+1}, L_{j+2}, ..., L_k\}$ has a right endpoint to
    the right of $c_k$ do
        Remove $R'$ from $L_q$;
        Let $h$ be the right endpoint of $R'$;
        Remove $R'$ from $R_h$;
        Assign $R$ as the parent of $R'$;
        Enqueue $R'$ to $Q$;
    end
end
```

Algorithm 2: BFS on the overlap graph in O(n log n) time

(1) Find the column with the maximum rightward or leftward column key in a range (c_j, ..., c_k): cut the list before c_j and after c_k, request the maximum value, then undo the cuts with concatenations.

(2) Change the key of any c_h: cut the list before and after c_h, change its key using the "add a constant" operation, and undo the cuts with concatenations.

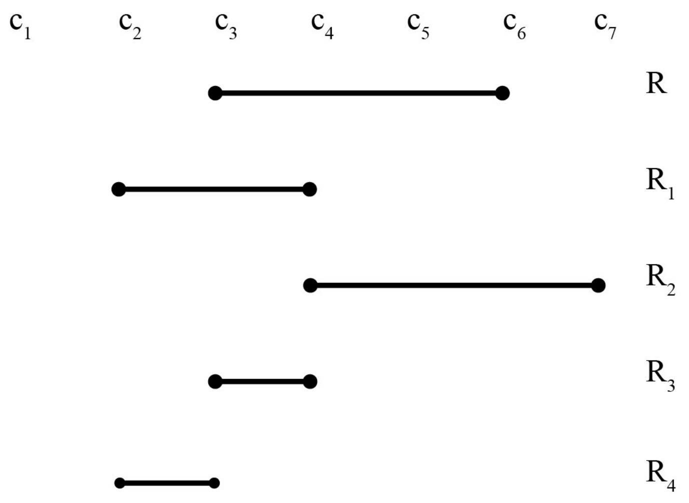


FIGURE 4.3. Processing the columns of a row R
R is being processed. R_1, R_2, and R_4 properly overlap R and will be added to the queue; R_3 will not.

Any time a row is removed from the BFS queue, we need to find all other rows that properly overlap it. Suppose row i contains columns c_j, ..., c_k. We use the first operation above to get the column with the maximum key in [c_{j+1}, ..., c_k]. If that key is greater than the right endpoint of row i (that is, c_k), we add the top row from that column's L list to the queue (removing the column from its L and R lists so it cannot be found again) and decrement the column, all in O(log n) time. We do this one time for each row that is added to the BFS queue, plus one additional time when we fail to add a new row to the queue. We then execute the symmetric loop to find rows that overlap on the left. Overall, we spend O(log n) time for each row we insert into the queue, plus an additional O(log n) time when it comes off the queue, for a total of O(n log n) time over all rows.

4.3.4. FINDING THE TUCKER MATRIX. Given the overlap graph, we choose A and B as described in section 4.3.1 and use the probe operation from section 4.3.2 to find the path P_1. We then reverse P_1 and run the operation again to find P_2, all in O(n log n) time. The rows in P_2 are the rows of the Tucker matrix. It remains to find the columns. We have a consecutive-ones ordering of all but the last row, and the rows are represented by their endpoint columns. All the rows except the last one form a path in a connected component in the overlap graph, so the ordering of the endpoints is uniquely constrained up to reversal. As a result, by ordering the rows in ascending order of left endpoint, we can easily associate them with the corresponding rows of a rotation of a Tucker matrix and identify intervals on the columns where we need to select exactly one column (which one will be selected is determined from where the 1s occur in the last row, which is the only row that is not consecutive-ones ordered). Each of these operations can be done in O(n log n) time, which results in a total of O(n log n) for the entire process of finding the Tucker matrix.

4.4. Presentation of the Certificate

So far we have described how to determine whether a matrix with the consecutive-ones property continues to have that property after adding or removing a row or column, and how to return a certificate proving that the answer is correct. It remains to describe how the user can easily verify the certificate. We assume that the user has checked the certificate of a successful operation after each step and so trusts that the current ordering of the matrix is a consecutive-ones ordering. If a row or column is deleted, no certificate is required that it remains one. If a row is successfully inserted, we provide a new ordering of the columns that will allow the insertion of that row; the user must then verify that this is a legal ordering. If a column is successfully inserted, we provide an ordering of the other columns that allows the new column to be inserted; again, the user must verify that this is a legal ordering.

4.4.1. When the new matrix has the consecutive-ones property. Rather than providing the user with a PQ tree and requiring him to verify that it is correct, we provide instructions on how to modify the matrix. If we are simply removing a row or column, the user removes that row or column from his matrix (and renumbers the remaining columns, if a column was removed) and no further proof is required; otherwise, we need to demonstrate that the new ordering is correct. If the user has a PQ tree, we can provide a list of the rotations we made to rearrange the columns; making the same rotations will allow the user to generate the same ordering. If the user only has a compact matrix, we can still convince him that our reordering of the columns is a valid consecutive-ones ordering, as follows.

Let T be the PQ tree for the current consecutive-ones matrix and let the leaves of T be numbered $1...n$ according to the current ordering. The first part of the certificate will be a permutation π, and the user must verify that π is a permitted permutation in the PQ tree.

In O(n) time, the user first verifies that π is, in fact, a permutation; this can be done using an array of size n to verify that all elements from $1...n$ appear exactly once in π.

The next step is to verify that π is a permutation that is permitted by the PQ tree. The user labels each leaf u of the tree with its current number $p(u)$ and its number in π, $\pi(u)$. He then labels each internal node with the minimum element $p_l(u)$ and maximum element $p_r(u)$ of $\{p(\mathrm{c})$: c is a leaf descendant of $u\}$, as well as the minimum element $\pi_l(u)$ and maximum element $\pi_r(u)$ of $\{\pi(c)$: c is a leaf descendant of $u\}$. This takes O(n) time working by induction from the leaves to the root: $p_l(u)$ is the minimum of $\{p_l(w)$: w is a child of $u\}$ and $p_r(u)$ is the maximum of $\{p_r(w)$: w is a child of $u\}$. For each node, if $p_r(u)$ - $p_l(u) \neq \pi_r(u)$ - $\pi_l(u)$, the user rejects the certificate. Otherwise, π is a valid permutation if all nodes of the tree are P nodes.

If T contains any Q nodes, the user also must verify that all children of each Q node are in the same order or its reverse. Let (w_1, w_2, ..., w_k) be the ordering of the children of a Q node. The constraints are followed if one of the following conditions is true:

- For each i from 1 to k-1, $\pi_l(w_{i+1}) = \pi_r(w_i) + 1$

- For each i from 1 to k-1, $\pi_l(w_i) + 1 = \pi_r(w_{i+1})$

These two checks convince the user that π is an allowed permutation, which means that because T is a PQ tree on the columns of the matrix, the ordering given by π preserves the consecutive-ones property for the matrix represented by the PQ tree. Because T is the correct PQ tree, the endpoints of the interval corresponding to each row are either $\{p_l(u), p_r(u)\}$ for some node u or are $\{p_l(w_i), p_r(w_j)\}$ for some Q node, and pointing out u or w_i and w_j will convince the user that π is a consecutive-ones ordering. Finally, the user builds a data structure to provide, for each interval, the minimum and maximum number in the interval; checking that these are the endpoints of the interval before the permutation convinces the user that all adjacencies have been maintained. This can be done in O(n) time over all rows using a Cartesian tree [Vui80], where the heap value of a node is the node's position in p and its binary search key is its position in π.

Now that we have shown that π is a valid consecutive-ones ordering of the current matrix, we can make the requested change. If we are adding a row, the user verifies that all 1s in the new row are consecutive in the new column ordering. If we are adding a column, the user verifies that for each row, the new column does not contain a 1 where the two adjacent columns both contain 0s or a 0 where the two adjacent columns both contain 1s. This takes O(n) time by checking endpoints.

4.4.2. When the new matrix does not have the consecutive-ones property.

When a new row or column is added to the matrix and causes it to no longer have the consecutive-ones property, we find a Tucker submatrix as described in section 4.3. In doing so, we identify rows and columns of the matrix with the rows and columns of a Tucker matrix. Verifying the existence of the Tucker matrix proves to the user that the new matrix does not have the consecutive-ones property. Because all rows except one of the Tucker submatrix are consecutive-ones ordered (and thus can be given in compact form) this takes O(n) time.

CHAPTER 5

Dynamic Updates on Interval Graphs

In the previous chapter, we described how to determine whether a consecutive-ones matrix with a row or column added or deleted retains the consecutive-ones property, and how to return a forbidden submatrix if it does not. In this chapter, we take the matrix to be the clique matrix of an interval graph, and rather than adding and removing rows and columns we add and remove vertices and edges, which may result in the creation or destruction of cliques. Most operations remain the same; however, adding or removing a vertex (unlike adding or removing a row) can also alter the number of cliques. We use the same basic operations as before (where operation 2 now takes an interval graph I and operation 3 gives us an interval representation of the graph), with one additional operation:

(5) Given an interval model $\mathcal{I}$ of G and a new node x, find the set $\mathcal{C}$ of cliques of G that are subsets of $N(x)$ in $\mathrm{O}(n)$ time. Mark the intervals corresponding to members of $N(x)$. Traverse I, keeping two counters: c_1, which keeps track of the current number of intervals, and c_2, which keeps track of how many of them are members of $N(x)$. Each time you encounter a left endpoint, increment c_1, and if it is the left endpoint of a member of $N(x)$, increment c_2. Each time you encounter a right endpoint, decrement c_1, and if it is the right endpoint of a member of $N(x)$, decrement c_2. Each time you come to a clique point, include the clique in $\mathcal{C}$ if c_2 currently equals c_1.

5.1. Deletion of Vertices and Deletion and Insertion of Edges

Deletion or insertion of an edge reduces to deletion and reinsertion of one of its endpoints. It remains to describe how to delete and insert a vertex in $O(n)$ time. After each operation, we update the labeled PQ tree for the current graph's clique matrix.

To delete a vertex, construct an interval model from the labeled PQ tree using operation 3. Delete the vertex's interval from $\mathcal{I}$. Use operation 2 to build the PQ tree from the new model. Use operation 4 to turn it into the new labeled PQ tree.

5.2. Deleting x from an Interval Model of $G + x$

Suppose G is our current graph, x is the new vertex, and $G + x$ is an interval graph. Consider where x's interval I_x lies in a model $\mathcal{I}'$ of $G + x$. All but the first and last clique points spanned by x remain clique points when x is removed. Suppose the first clique point fails to be a clique point of G. Call this clique B_1. (See Figure 5.1).

B_1 contains x and a set U of vertices whose rightmost clique is B_1. It can't contain any left endpoint except that of x or it would continue to be a clique point of G. Similarly, if the last clique of x fails to be a clique of G, call this clique B_2, and x meets a set W of vertices that have B_2 as their leftmost clique.

U induces a complete subgraph of G, which is also contained in the clique A_1 to the left of B_1 (otherwise B_1 would continue to be a clique in G). Similarly, W induces a complete subgraph of G, and W is a subset of a clique A_2 to the right of B_2. $U \cup W$ induces two disjoint complete subgraphs.

After x is removed, A_1 becomes the rightmost clique of each member of U. The remaining clique points spanned by x (from C_1 to C_2 in figure 5.1) remain clique points in $\mathcal{I}' \setminus \{x\}$, so they correspond to cliques of G that are subsets of $N(x)$. Call these cliques $\mathcal{C}$.

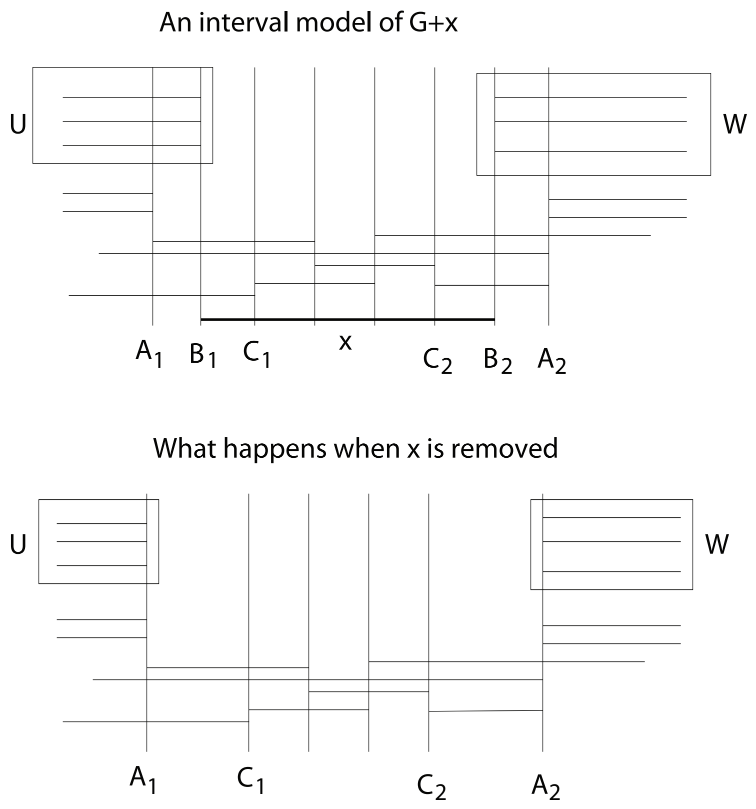


FIGURE 5.1. Removing x from an interval model of $G + x$.

5.3. INSERTING A VERTEX

5.3.1. THE CASE WHERE $\mathcal{C}$ IS NONEMPTY. The foregoing observations tell us how to get an interval model $\mathcal{I}'$ for $G + x$ from an interval model $\mathcal{I}$ of G. Generate $\mathcal{I}$ from the labeled PQ tree of G using operation 3. Using operation 5, find the set $\mathcal{C}$ of cliques of G that are subsets of $N(x)$. Since $\mathcal{C}$ must be consecutive in $\mathcal{I}'$, apply the incremental step of Booth and Lueker to get a PQ tree that enforces this constraint, get the labeled PQ tree using operation 4, and use operation 3 to get a new interval model $\mathcal{I}_2$ where $\mathcal{C}$ is consecutive.

Mark the intervals corresponding to $N(x)$ and identify the set $U \cup W$ of neighbors of x that do not intersect $\mathcal{C}$'s interval in $\mathcal{I}_2$. Since $U \cup W$ induces two disjoint complete subgraphs in G, it is trivial to partition $U \cup W$ into $\{U, W\}$ in $\mathrm{O}(n)$ time.

Let a *minimal* element I_U of U be a shortest interval in U. A minimal interval I_W of W is defined similarly. Let $\mathcal{C}_U$ and $\mathcal{C}_W$ be the cliques of G contained in I_U and I_W. With two applications of the incremental step of Booth and Lueker, add the constraints that $\mathcal{C}_U \cup \mathcal{C}$ and $\mathcal{C} \cup \mathcal{C}_W$ must each be consecutive to get a new PQ tree that enforces this. Get an interval model $\mathcal{I}_3$ from this PQ tree, via the labeled PQ tree, as before.

Since I_U is minimal, all members of U are contained in every clique of G that I_U is, so they are contained in the rightmost clique A_1 of I_U. Since $\mathcal{C} \cup \mathcal{C}_W$ is consecutive, A_1 is consecutive with the leftmost clique C_1 of $\mathcal{C}$. We can extend the left endpoint of x and the right endpoints of U to meet, creating the new clique B_1. This step can be omitted when U is empty. A symmetric operation applies to W and x.

This yields an interval model for $G + x$. Create the PQ tree for $G + x$ from this model using operation 2 and the labeled PQ tree for $G + x$ from this PQ tree using operation 4.

5.3.2. The case where $\mathcal{C}$ is empty. Suppose $G + x$ is an interval graph and $\mathcal{C}$ is empty. If $N(x)$ is complete, adding x reduces to adding a new column using the algorithm of section 4.2.2. It remains to show what happens when $N(x)$ is not a complete subgraph. Operation 5 detects this by finding no cliques and verifying that $c_2 < |N(x)|$ throughout the scan of the interval model.

Let $\mathcal{I}'$ be an interval model of $G + x$. Since $N(x)$ is not a complete subgraph, U, W, A_1, A_2, B_1, and B_2 all exist. Since $U \cup W$ induces two disjoint complete subgraphs, in any interval model $\mathcal{I}$ of G every member of one of them precedes every member of the other. Operation 5 finds that c_2 is monotonically increasing, then monotonically decreasing as it

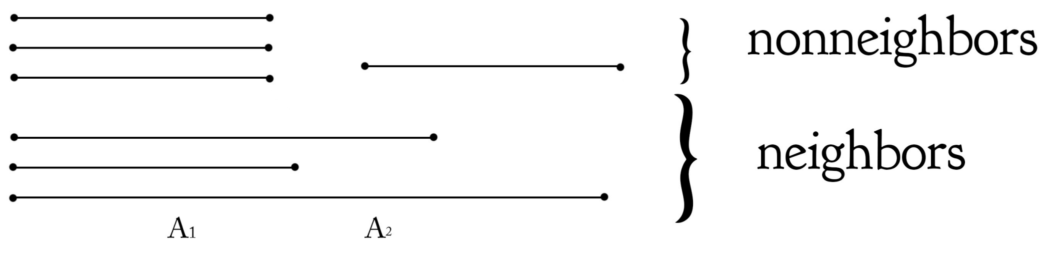


Figure 5.2. Adding x to the interval model of G
Every neighbor of x belongs to A_1, and some end there. If x can be inserted, its new clique will be to the right of A_1.

crosses endpoints of U, then increases when the first endpoint of W is encountered. The right endpoints of neighbors of x encountered up to that point are U, and the left endpoints encountered after that point are W.

We again find minimal elements of U and W, then use the incremental step of Booth and Lueker to modify the PQ tree so that it enforces the additional constraint that $\mathcal{C}_U \cup \mathcal{C}_W$ is consecutive. This forces the right endpoints of U and the left endpoints of W to lie in consecutive cliques. We add x between those clique points, and stretch the endpoints of U and W to meet the endpoints of x, creating and placing B_1 and B_2.

5.4. Returning a Certificate

The algorithm in section 4.3 returns a Tucker submatrix when modifications to a consecutive-ones matrix cause it to lose the consecutive-ones property. When the consecutive-ones matrix is the clique matrix of an interval graph, we can modify the algorithm to also return a Lekkerkerker-Boland subgraph in the modified graph, in O(n log n) time.

5.4.1. Preparation of the Clique Matrix. If the zero-one matrix represents an interval graph and adding x causes it to lose the consecutive-ones property, then adding x may result in many new cliques being added to the graph. Rather than deal with all of them, we will find a minimally non-consecutive-ones submatrix to feed to the algorithm below.

If G is an interval graph and $G + x$ is not, then x is a vertex of every Lekkerkerker-Boland subgraph. If two nonadjacent members of $N(x)$ are adjacent to the same connected component of $G - N(x)$, then the graph is not chordal and we can return a chordless cycle in $\mathrm{O}(n)$ time given a model of G.

Otherwise, assume that $G + x$ is chordal. In section 5.3, we give an $\mathrm{O}(n)$ algorithm for determining whether $G+x$ is an interval graph. We can use the same algorithm to determine whether $G[A] + x$ is an interval graph, where A is the vertices of a prefix of the rows of a clique matrix. Using this as the probe operation, do a binary search to find the maximal prefix A of the clique matrix of G such that $G[A] + x$ is an interval graph, in $\mathrm{O}(n \log n)$ time. Let y be the next row after A. $G[A] + y$ and $G[A] + x$ are both interval graphs, but $G[A] + x + y$ is not. Let M' be a consecutive-ones ordered clique matrix of $G[A]$. Let M_x be a consecutive-ones ordered clique matrix of $G[A] + x$ and let M_y be a consecutive-ones ordered clique matrix of $G[A] + y$.

Suppose x and y are nonadjacent. Then the cliques of $G[A]+x+y$ are cliques of $G[A]+x$ or $G[A] + y$. The cliques of $G[A] + x$ that contain x are cliques of $G[A] + x + y$. The cliques of $G[A] + y$ that contain y are cliques of $G[A] + x + y$. The columns of M' to which x was not added in M_x and y was not added in M_y are the remaining cliques.

Suppose x and y are adjacent. Let $X = \{c|c$ is a column of M_x and $c[A]$ is not a column of $M'\}$. X contains at most two columns, because adding x to $G[A]$ introduces at most two cliques (since $G[A] + x$ is an interval graph). Similarly, let $Y = \{c|c$ is a column of M_y and $c[A]$ is not a column of $M'\}$, and $|Y| \leq 2$. Add y to each column in X that is a subset of its neighborhood, and add x to each column in Y that is a subset of its neighborhood. Eliminate any element of X that now duplicates an element of Y. For each column of M'

to which x was added to get M_x, add x to the corresponding column of M_y. If x and y do not yet share a column, add a column c_{xy} equal to $(N(x) \cap N(y) \cap A) \cup \{x, y\}$.

In either case, we get a clique matrix M_{xy} of $G[A] + x + y$ in O(n) time. If x and y are nonadjacent, M_y gives a consecutive-ones ordering of $M_{xy}[A + y]$ except for the columns in X. If x and y are adjacent, M_y gives a consecutive-ones ordering of $M_{xy}[A + y]$ except for columns in X and possibly a column c_{xy}. In each case, there is a set $\mathcal{Z}$ of at most three columns of $M_{xy}[A + y]$ that are not in the consecutive-ones ordering given by M_y.

For each $c \in \mathcal{Z}$, add $c[A+y]$ to M, one by one, to yield a larger consecutive-ones ordering or determine that this is not possible, again in O(n) time.

If this is successful, then M_{xy} is consecutive except in its last row, and the algorithm of section 4.3.2 can be used to obtain a Tucker submatrix in O(n log n) time.

Otherwise, let M be the consecutive-ones ordered columns of $M_{xy}[A]$ when insertion of $c \in \mathcal{Z}$ has failed. Since $|\mathcal{Z}| \leq 3$, one or two elements of $\mathcal{Z}$ may have already been added successfully. M is a consecutive-ones matrix, but $M + c$ is not. We can then find the minimal prefix B of rows such that $M[B] + c[B]$ is not a consecutive-ones matrix, as described above.

5.5. Presentation of the Certificate

If the altered graph is no longer an interval graph, then we returned a Tucker matrix, which can be converted to a Lekkerkerker-Boland subgraph and annotated with an asteroidal triple (or chordless cycle) in O(n) time. Given the paths between the vertices of the asteroidal triple, it is easy for the user to verify the forbidden subgraph, also in O(n) time.

Otherwise we instruct the user on how to reorder the cliques, as described in section 4.4, and the user verifies that this gives a valid interval representation of the graph before adding or removing a vertex.

If we are removing a vertex x, then up to two cliques may disappear, and the certifying algorithm returns the columns to be deleted. The user verifies that the only left endpoint in one of the columns and the only right endpoint in the other column belong to x, and thus those columns cease to be maximal cliques and can be removed. After removal, the remaining columns are renumbered but do not need to be reordered and it is trivial to relabel the endpoints of the remaining vertices, in $\mathrm{O}(n)$ time.

When we are adding a vertex x, then up to two cliques may be added. After verifying the new permutation, the user inserts the (up to) two additional columns. Of the two new columns, x has the only left endpoint in one column and the only right endpoint in the other. Each 1 in the two columns (and no 0) occurs between the endpoints of the corresponding row, showing that no adjacencies other than x's have changed. Using x's left and right endpoints, the user then verifies that x's adjacencies are correctly represented. All of this takes $\mathrm{O}(n)$ time.

Bibliography

[Ben59] S. Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. U.S.A.*, 45:1607–1620, 1959.

[Ber61] C. Berge. Fārbung von Graphen deren sämtliche bzw. deren ungerade Kreise starr sind. In *Zusammenfassung), Wissenschaftliche Zeitschrift, Martin Luther Universitt Halle-Wittenberg, MathematischNaturwissenschaftliche Reihe*, pages 114–115, 1961.

[BHV06] A. Berry, P. Heggernes, and Y. Villanger. A vertex incremental approach for maintaining chordality. *Discrete Math.*, 306(3):318–336, 2006.

[BL76] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13:335–379, 1976.

[BSS05] A. Berry, A. Sigayret, and J. Spinrad. Faster dynamic algorithms for chordal graphs, and an application to phylogeny. In *Graph-Theoretic Concepts in Computer Science*, 2005.

[CIRS03] L. S. Chandran, L. Ibarra, F. Ruskey, and J. Sawada. Generating and characterizing the perfect elimination orderings of a chordal graph. *Theo*, 307:303–317, 2003.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[CLS81] D. G. Corneil, H. Lerchs, and L. K. Stewart. Complement reducible graphs. *Discrete Applied Mathematics*, 3:163–174, 1981.

[CP06] C. Crespelle and C. Paul. Fully dynamic recognition algorithm and certificate for directed cograph. *Discrete Appl. Math.*, 154(12):1722–1741, 2006.

[Cre10] Christophe Crespelle. Fully dynamic representations of interval graphs. In *Graph-Theoretic Concepts in Computer Science*, volume 5911 of *Lecture Notes in Computer Science*, pages 77–87. 2010.

[CRST06] M. Chudnovsky, N. Robertson, P. Seymour, and R. Thomas. The strong perfect graph theorem. *Annals of Mathematics*, 164:51–229, 2006.

[CST12] C. Chauve, T. Stephen, and M. Tamayo. Efficient algorithms for finding Tucker patterns. *CoRR*, abs/1206.1837, 2012.

[CT07] D. Corneil and M. Tedder. An optimal, edges-only fully dynamic algorithm for distance hereditary graphs. In *In 24th International Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science*, 2007.

[DGN10] M. Dom, J. Guo, and R. Niedermeier. Approximation and fixed-parameter algorithms for consecutive ones submatrix problems. *Journal of Computer and System Sciences*, 76(34):204 – 221, 2010.

[Dir61] P. Dirac. On rigid circuit graphs. *Abh. Math. Sem. Univ. Hamburg*, 25:71–76, 1961.

[FG65] D. R. Fulkerson and O. Gross. Incidence matrices and interval graphs. *Pacific J. Math.*, 15:835–855, 1965.

[Ful69] D. R. Fulkerson. The perfect graph conjecture and pluperfect graph theorem. In *2nd Chapel Hill Conf on Combin. Math. and its Appl.*, 1969.

[Ful71] D. R. Fulkerson. Blocking and anti-blocking pairs of polyhedra. *Mathematical Programming*, 1(1):168–194, 1971.

[Ful72] D. R Fulkerson. Anti-blocking polyhedra. *Journal of Combinatorial Theory, Series B*, 12(1):50 – 71, 1972.

[GH64] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, 16:539–548, 1964.

[GLS88] M. Grötschel, L. Lovsz, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1988.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[GP07] E. Gioan and C. Paul. Dynamic distance hereditary graphs using split decomposition. In *Algorithms and Computation*, 2007.

[HK] Pinar Heggernes and Dieter Kratsch. Linear-time certifying algorithms for recognizing split graphs and related graph classes.

[HM90] P. L. Hammer and F. Maffray. Completely separable graphs. *Discrete Applied Mathematics*, 27:85–99, 1990.

[HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.

[Iba99] L. Ibarra. Fully dynamic algorithms for chordal graphs. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 923–924, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[JS01] J.L. Johnson and J.P. Spinrad. A polynomial time recognition algorithm for probe interval graphs. *Proceedings of the Twelfth Annual ACM-SIAM Symposium on DiscreteAlgorithms*, 12:477–486, 2001.

[KM89] N. Korte and R.H. Moehring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, pages 68–81, 1989.

[KMMS03] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *Proceedings of the Fourteenth Annual ACM-SIAM Symposium Discrete Algorithms (SODA 03)*, pages 866–875, 2003.

[KMMS06] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. *SIAM J. Comput*, (36):2006, 2006.

[Ko31] Dénes Kõnig. Gráfok és mátrixok. *Matematikai és Fizikai Lapok*, 38:116–119, 1931.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, Boston, 2006.

[Kur30] C. Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamata Mathematicae*, pages 271–283, 1930.

[kY06] T. kyzy Yrysgul. A fully dynamic algorithm for recognizing and representing chordal graphs. In *Ershov Memorial Conference*, pages 481–486, 2006.

[LB62] C. Lekkerker and D. Boland. Representation of finite graphs by a set of intervals on the real line. *Fund. Math.*, 51:45–64, 1962.

[LM14] N. Lindzey and R. M. McConnell. Linear-time algorithms for finding Tucker submatrices and Lekkerkerker-Boland subgraphs. *CoRR*, abs/1401.0224, 2014.

[Lov72] L. Lovász. Normal hypergraphs and the perfect graph conjecture. *Discrete Mathematics*, 2(3):253 – 267, 1972.

[McC04] R. M. McConnell. A certifying algorithm for the consecutive-ones property. *Proceedings of the 15th Annual ACM-SIAM Symposium onDiscrete Algorithms (SODA04)*, 15:761–770, 2004.

[MdM05] R. M. McConnell and F. de Montgolfier. Algebraic operations on pq trees and modular decomposition trees. *LNCS*, 3787:421–432, 2005.

[MM99] T.A. McKee and F.R. McMorris. *Topics in Intersection Graph Theory.* Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[MN09] R. M. McConnell and Y. Nussbaum. Linear-time recognition of probe interval graphs. In *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 349–360, 2009.

[MRT88] C. L. Monma, B. Reed, and W. T. Trotter. Threshold tolerance graphs. *Journal of Graph Theory*, 12:343–362, 1988.

[MS99] R. M. Mcconnell and J. P. Spinrad. Modular decomposition and transitive orientation. *Discrete Math*, 201:189–241, 1999.

[NPP06] S. D. Nikolopoulos, L. Palios, and C. Papadopoulos. A fully dynamic algorithm for the recognition of P4-sparse graphs, 2006.

[Ros70] D. J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597 – 609, 1970.

[RTL76] D. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[Sha01] P. Hell.and R. Shamir.and R. Sharan. A fully dynamic algorithm for recognizing and representing proper interval graphs. *SIAM J. Comput.*, 31(1):289–305, 2001.

[Shi84] D. R. Shier. Some aspects of perfect elimination orderings in chordal graphs. *Discrete Applied Mathematics*, 7(3):325 – 331, 1984.

[SS04] R. Shamir and R. Sharan. A fully dynamic algorithm for modular decomposition and recognition of cographs. *Discrete Appl. Math.*, 136(2-3):329–340, 2004. The 1st Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW 2001).

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362 – 391, 1983.

[Tuc72] A. Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory, Series B*, 12:153–162, 1972.

[TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984.

[TY85] E. E. Tarjan and M. Yannakakis. Addendum: Simple linear-time algorithms to test chordality of graphs,test acyclicity of hypergraphs, and selectively reduce acyclichypergraphs. *SIAM Journal on Computing*, 14:254–255, 1985.

[Vui80] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, April 1980.

[ZSF$^+$94] P. Zhang, E. A. Schon, S. G. Fischer, E. Cayanis, J. Weiss, S. Kistler, and P. E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of dna. *Comput. Appl. Biosci.*, 10(3):309–317, June 1994.