Computer Science Department

TECHNICAL REPORT

ALGORITHMIC ASPECTS OF PERFECT GRAPHS

by

Martin Charles Golumbic

July 1980

Report No. 019

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012

ALGORITHMIC ASPECTS OF PERFECT GRAPHS

by

Martin Charles Golumbic

July 1980

Report No. 019

Courant Institute of Mathematical Sciences
251 Mercer Street
New York, N. Y. 10012

This work was supported in part by the
National Science Foundation under
Grant No. MCS 78-03820

Algorithmic Aspects of Perfect Graphs

by

Martin Charles Golumbic

Consider a collection $C = \{c_i\}$ of courses being offered by a major university. Let T_i be the time interval during which course c_i is to take place. We would like to assign courses to classrooms so that no two courses meet in the same room at the same time.

This problem can be solved by properly coloring the vertices of the graph $G = (C,E)$ where $c_i c_j \in E \Leftrightarrow T_i \cap T_j \neq \emptyset$. We may interpret each color as corresponding to a different classroom. The graph G is an interval graph, since it is represented by intersecting time intervals.

This example is especially interesting because efficient, linear-time algorithms are known for coloring interval graphs with a minimum number of colors. (The minimum coloring problem is NP-complete for general graphs.)

In this paper we will survey a number of topics in algorithmic graph theory which involve classes of perfect graphs. We will also discuss some recent applications of perfect graphs to computer science. The intention of this article is to provide an understanding of the main research directions which have been investigated and to suggest possible new areas of research. The sections of this paper are numbered to correspond with the chapters of the author's book "Algorithmic Graph Theory and Perfect Graphs". The interested reader is referred to this book for further study.

2. The Design of Efficient Algorithms

Algorithmic complexity analysis deals with the quantitative aspects of problem solving. It addresses the issue of what can be computed within a *practical* or *reasonable* amount of time and space by measuring the resource requirements exactly or by obtaining upper and lower bounds for them. Complexity is actually determined at three levels: the problem, the algorithm, and the implementation. Naturally, we want the best algorithm which solves our problem, and we want to choose the best implementation of that algorithm.

Consider the problem of determining whether an undirected graph G is connected. A mathematically elegant solution is the following: G is connected if and only if $I + M + M^2 + M^3 + \cdots + M^{n-1}$ has no zero entries where M is the adjacency matrix of G, I is the identity matrix, and n is the number of vertices of G. However, using this theorem as an algorithm would require much more work (matrix multiplication and addition) than is actually needed to test connectivity. A better way would be to traverse the edges of the graph. The following algorithm will test connectivity and find a spanning tree efficiently.

<u>Standard Spanning Tree Algorithm</u> (SST)

<u>Step I</u>: Start with a tree T consisting of one arbitrary vertex and no edges.

<u>Step II</u>: If T contains all the vertices of G, then STOP [T is a spanning tree]. Otherwise, do step III.

<u>Step III</u>: Add to T an edge (x,y) which joins a vertex y not yet in T to a vertex x already in T. If no such edge exists, then STOP [there is no spanning tree; G is not connected]. Otherwise, go to step II.

In Step III of our algorithm there may be several edges (x,y) eligible to be added to T. We call such an edge a

candidate edge. Various priorities can be established to guide the choice of candidates, and each priority will yield a slightly different algorithm. If candidates are stored in a queue, then SST gives a breadth-first search (BFS) of G. Storing candidates in a stack SST does a depth-first search (DFS). If the edges have costs associated with them, and if the candidate with minimum cost is always chosen, then SST produces a minimum cost spanning tree (MST). Similarly, shortest path algorithms and critical path algorithms can also be designed by adapting SST with a suitable priority for choosing candidates.

The complexity of the spanning tree algorithm depends on how the graph is stored and whether anything special is done to the candidate edges. The table below summarizes these complexities.

Complexity of the Standard Spanning Tree Algorithm

Candidates in a	Adjacency Matrix Stored as an Array	Adjacency Sets Stored as Lists or Sequentially
Stack (DFS)	$O(n^2)$	$O(n + e)$
Queue (BFS)	$O(n^2)$	$O(n + e)$
Reverse Heap (MST)	$O(n^2 + e \log e)$	$O(n + e \log e)$

A graph problem is said to be *linear* in the size of the graph if it has an algorithm which can be implemented to run in $O(n + e)$ steps on a graph with n vertices and e edges. Thus testing connectivity is a linear graph problem. This is usually the best that one could expect for any nontrivial graph problem since every vertex and every edge would probably

have to be examined at least once. A problem is called *polynomial* if it has an algorithm which can run in $O(p(n))$ steps where p is a polynomial function.

The algorithmic graph problems that we will examine in this survey paper include recognizing various classes of perfect graphs and finding minimum colorings, minimum clique covers, maximum cliques, and maximum stable sets. We will be particularly interested in special purpose polynomial algorithms designed to solve these problems for particular classes of perfect graphs. The reason such algorithms are important is that for arbitrary graphs these last four problems are NP-complete. That is, they are in a large class of problems which all currently require an exponential amount of running time and which are all related in such a way that if any one of them could be solved in polynomial time, then so could all problems in this class.

3. Perfect Graphs

An undirected graph $G = (V,E)$ is *perfect* if it satisfies any of the following equivalent conditions:

(P_1) $\quad \omega(G_A) = \chi(G_A)$ $\quad$ (for all $A \subseteq V$)

(P_2) $\quad \alpha(G_A) = \theta(G_A)$ $\quad$ (for all $A \subseteq V$)

(P_3) $\quad \omega(G_A)\ \alpha(G_A) \geq |A|$ $\quad$ (for all $A \subseteq V$)

The equivalence of (P_1) - (P_3) is known as the Perfect Graph Theorem.

An open question whose solution has eluded researchers for two decades is to prove or disprove the following conjecture of Claude Berge.

Strong Perfect Graph Conjecture (SPGC). An undirected graph G is perfect if and only if in G and in $\bar{G}$ every odd cycle of length ≥ 5 has a chord.

Although proving the SPGC seems to be a mathematical rather than an algorithmic problem it does raise an interesting algorithmic question.

Is there a polynomial algorithm which recognizes whether or not an undirected graph G has an odd chordless cycle of length ≥ 5?

We have no answer to this question. However, if there is such an algorithm and if the SPGC is true, then it would answer another open question,

Is there a polynomial algorithm which recognizes whether or not an undirected graph G is perfect?

In a very recent paper Grötschel, Lovász, and Schrijver [1980] have shown that the ellipsoid method of solving linear programming problems can be applied to obtain a polynomial algorithm to find maximum stable sets and minimum colorings for perfect graphs. Also, since G is perfect if and only if its complement $\bar{G}$ is perfect, this same approach can be used to find maximum cliques and minimum clique covers. The major importance of this result is that it generalizes what had been known for certain classes of perfect graphs. Although the complexity of the algorithm is polynomial, it may not be practical to implement. As the authors point out, it is not intended to compete with the special purpose algorithms designed to solve these problems for interval graphs, comparability graphs, triangulated graphs, and other classes of perfect graphs which so often arise in applications.

4. Triangulated Graphs

An undirected graph G is called *triangulated* if every cycle of length strictly greater than 3 possesses a chord, that is, an edge joining two nonconsecutive vertices of the cycle. In the literature, triangulated graphs have also been called *chordal, rigid-circuit, monotone transitive* and *perfect elimination* graphs.

A vertex x of G is called *simplicial* if its adjacency set Adj(x) induces a complete subgraph of G, i.e., Adj(x) is a clique (not necessarily maximal). Dirac [1961], and later Lekkerkerker and Boland [1962], proved that a triangulated graph always has a simplicial vertex (in fact at least two of them), and using this fact Fulkerson and Gross [1965] suggested an iterative procedure to recognize triangulated graphs based on this and the hereditary property. Namely, *repeatedly locate a simplicial vertex and eliminate it from the graph, until either no vertices remain and the graph is triangulated or at some stage no simplicial vertex exists and the graph is not triangulated.* The correctness of this procedure is given in Theorem 4.1. Let us state things more algebraically.

Let $G = (V,E)$ be an undirected graph and let $\sigma = [v_1, v_2, \ldots, v_n]$ be an ordering of the vertices. We say that σ is a *perfect vertex elimination scheme* (or *perfect scheme*) if each v_i is a simplicial vertex of the induced subgraph $G_{\{v_i, \ldots, v_n\}}$. In other words, each set

$$A_i = \{v_j \in \mathrm{Adj}(v_i) \mid j > i\}$$

is complete. For example, the graph G_1 in Figure 4.1 has a perfect vertex elimination scheme $\sigma = [a,g,b,f,c,e,d]$. It is not unique; in fact G_1 has 96 different perfect elimination schemes. In contrast to this, the graph G_2 has no simplicial vertex, so we cannot even start constructing a perfect scheme — it has none.

Algorithm 4.1. Maximum cardinality search.

Input: The adjacency sets of an undirected graph $G = (V,E)$.
Output: An ordering σ of the vertices.
Method: The vertices are numbered from n to 1 in the order that they are selected in line 3. This numbering fixes the positions of an elimination scheme σ. For each unnumbered vertex x, the *label* of x will consist of the number of numbered vertices adjacent to x. The vertices can then be ordered according to their labels. Ties are broken arbitrarily. The algorithm is as follows:

```
1.          assign the label 0 to each vertex;
2.          for  i ← n  to  1  by  - 1  do
3. select:     pick an unnumbered vertex v with largest label;
4.             σ(i) ← v; [this assigns to v the number i]
5. update:     for each unnumbered vertex w ∈ Adj(v) do
6.                 add 1 to label (w); end
            end
```

The fact that maximum cardinality search can be used to recognize triangulated graphs is demonstrated by the next theorem.

Theorem 4.3. An undirected graph $G = (V,E)$ is triangulated if and only if the ordering σ produced by Algorithm 4.1 is a perfect vertex elimination scheme.

Proof. If G has only 1 vertex, then the proof is trivial. Assume that the theorem is true for all graphs with fewer than n vertices and let σ be the ordering produced by Algorithm 4.1 when applied to a triangulated graph G. By induction, it is sufficient to show that $v = \sigma(1)$ is a simplicial vertex of G.

CLAIM: G may not contain a chordless path $\mu = [u,v_1,v_2,\cdots,v_k,w]$ with $k \geq 1$ satisfying the property

$$\sigma^{-1}(v_i) < \sigma^{-1}(u) < \sigma^{-1}(w) \quad \text{for all } i\ . \qquad (1)$$

Suppose G contains such a path μ, and choose μ such that $\sigma^{-1}(u)$ is largest possible. Since u was numbered before v_k and since v_k , but not u, is adjacent to w, there must be some vertex x such that $\sigma^{-1}(u) < \sigma^{-1}(x)$ which is adjacent to u but not to v_k. Let j be the largest index such that x is adjacent to v_j where we let $v_0 = u$. Then the path $\mu' = [x,v_j,\ldots,v_k,w]$ must be chordless, since its only possible chord xw would give a chordless cycle of length ≥ 4. If $\sigma^{-1}(x) < \sigma^{-1}(w)$, then μ' would satisfy (1) and contradict the maximality of $\sigma^{-1}(u)$ since $\sigma^{-1}(v_k) < \sigma^{-1}(u) < \sigma^{-1}(x)$. So it must be that $\sigma^{-1}(w) < \sigma^{-1}(x)$. But this implies that $\mu'' = [w,v_k\ldots,v_j,x]$ satisfies (1) and also contradicts the maximality. It follows that no such path μ can exist in G, which proves the claim.

Now let $v = \sigma^{-1}(1)$ and suppose that v is not simplicial. Choose $u,w \in \mathrm{Adj}(v)$ with $uw \notin E$ so that $\sigma^{-1}(u) < \sigma^{-1}(w)$. Then the path $[u,v,w]$ satisfies (1), which contradicts the claim. Therefore, v is simplicial and, by induction, σ is a perfect elimination scheme. The converse follows from Theorem 4.1. □

The complexity of Algorithm 4.1 is linear in the size of G. One such efficient implementation is the following. Let S_i be the set of unnumbered vertices whose label is i, and let S_i be represented by a doubly linked list. For each vertex we store its label i and a pointer to its position in the set S_i. When a vertex v is numbered it is removed from its set, and we move each adjacent vertex w up by one set; this can be executed in $O(1 + \text{degree}(v))$ steps. Thus, the entire algorithm will be $O(|V| + |E|)$.

In order to use MCS to recognize triangulated graphs, we need an efficient method to test whether or not a given ordering σ of the vertices is a perfect elimination scheme. Such an algorithm is given in Rose, Tarjan, and Lueker [1976] and has complexity $O(|V| + |E|)$. (See also Golumbic [1980, pp. 88-91].)

Fast Algorithms for the Coloring, Clique, Stable Set and Clique Cover Problems on Triangulated Graphs

Let $G = (V,E)$ be a triangulated graph, and let σ be a perfect elimination for G . It was first pointed out by Fulkerson and Gross [1965] that every maximal clique was of the form $\{v\} \cup A_v$ where

$$A_v = \{x \in \mathrm{Adj}(v) \mid \sigma^{-1}(v) < \sigma^{-1}(x)\} .$$

However, some of these sets $\{v\} \cup A_v$ will not be maximal, and we would like to filter them out. This can be accomplished in order to find the chromatic number and maximal cliques of a triangulated graph in $O(|V| + |E|)$ time.

The problem of finding the stability number $\alpha(G)$ of a triangulated graph and a clique cover of size $\alpha(G)$ is solved by Gavril [1972]. A linear implementation of his algo-algorithm can be obtained by using techniques of Rose, Tarjan and Lueker [1976].

Let σ be a perfect elimination scheme for $G = (V,E)$. We define inductively a sequence of vertices $y_1, y_2, \ldots, y_t$ in the following manner: $y_1 = \sigma(1)$; y_i is the first vertex in σ which follows y_{i-1} and which is not in $A_{y_1} \cup A_{y_2} \cup \ldots \cup A_{y_{i-1}}$; all vertices following y_t are in $A_{y_1} \cup \ldots \cup A_{y_t}$. Hence, $V = \{y_1, y_2, \ldots, y_t\} \cup A_{y_1} \cup \ldots \cup A_{y_t}$. The following theorem applies.

Theorem 4.4 (Gavril [1972]). The set $\{y_1, y_2, \ldots, y_t\}$ is a maximum stable set of G, and the collection of sets $Y_i = \{y_i\} \cup A_{y_i}$ $(i = 1,2,\ldots,t)$ comprises a minimum clique cover of G.

Proof. The set $\{y_1, y_2, \ldots, y_t\}$ is stable since if $y_j y_i \in E$ for $j < i$, then $y_i \in A_{y_j}$ which cannot be. Thus, $\alpha(G) \geq t$. On the other hand, each of the sets $Y_i = \{y_i\} \cup A_{y_i}$ is a clique, and so $\{Y_1, \ldots, Y_t\}$ is a clique cover of G. Thus, $\alpha(G) = \theta(G) = t$, and we have produced the desired maximum stable set and minimum clique cover. □

5. Comparability Graphs

An undirected graph $G = (V,E)$ is a *comparability graph* if there exists an orientation (V,F) of G satisfying

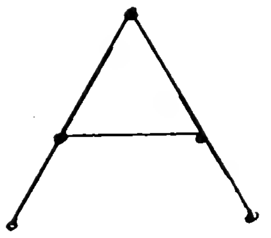
$$F \cap F^{-1} = \emptyset \;, \qquad F + F^{-1} = E \;, \qquad F^2 \subseteq F \;,$$

where $F^2 = \{ac \mid ab, bc \in F \text{ for some vertex } b\}$ and F^{-1} is the reversal of F. The relation F is a strict partial ordering of V whose comparability relation is exactly E, and F is called a *transitive orientation* of G (or of E). Comparability graphs are also known as *transitively orientable* graphs and *partially orderable* graphs. Examples of some comparability graphs can be found in Figure 5.1.

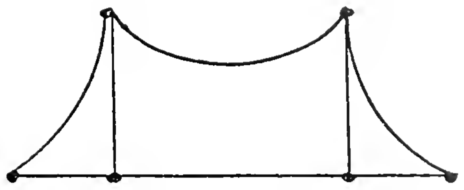
Let us see what happens when we try to assign a transitive orientation to the 4-cycle (Figure 5.2a). Arbitrarily choosing $ab \in F$ *forces* us to orient the bottom edge toward b and the top edge toward d (for otherwise transitivity would be violated). These in turn force the remaining edge to be oriented toward d. Applying the same idea to the graph in in Figure 5.2b we find that a contradiction arises, namely, choosing $ab \in F$ forces successively the orientations cb,cd,cf,ef,bf,ba. This graph is not a comparability graph. We now make the notion of forcing more precise.

Define the binary relation Γ on the edges of an undirected graph $G = (V,E)$ as follows:

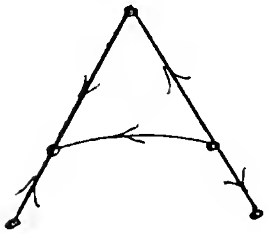
$$ab \;\Gamma\; a'b' \quad \text{iff} \quad \begin{cases} \text{either } a = a' \text{ and } bb' \notin E \\ \text{or} \quad\; b = b' \text{ and } aa' \notin E \end{cases}$$



The A Graph



The Suspension Bridge Graph



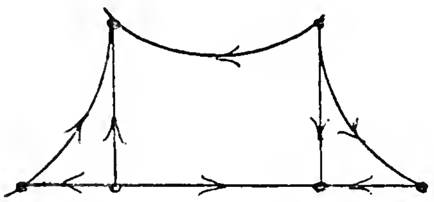
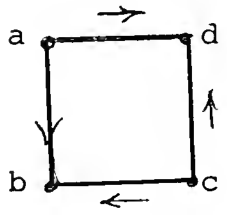
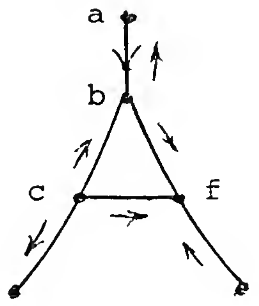


Figure 5.1 Transitive Orientations of Two Comparability Graphs



(a)



(b)

Figure 5.2 Examples of Forcing. The arbitrary choice of ab ∈ F forces the other indicated orientations.

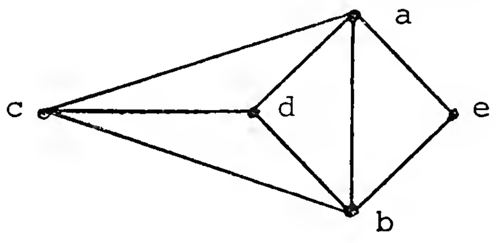


Figure 5.3

We say that ab *directly forces* a'b' whenever ab Γ a'b'. Since E is irreflexive, ab Γ ab; however, ab $\not\Gamma$ ba. The reader should not continue until he is convinced of this fact.

The reflexive, transitive closure Γ^* of Γ is easily shown to be an equivalence relation on E and hence partitions E into what we shall call the *implication classes* of G. Thus edges ab and cd are in the same implication class if and only if there exists a sequence of edges

$$ab = a_0b_0 \ \Gamma \ a_1b_1 \ \Gamma \ \ldots \ \Gamma \ a_kb_k = cd \ , \quad \text{with } k \geq 0 \ .$$

Such a sequence is called a Γ-*chain* from ab to cd, and we say that ab (eventually) *forces* cd whenever ab Γ^* cd.

Examples. The graph G of Figure 5.3 has 8 implication classes:

$A_1 = \{ab\}$, $A_2 = \{cd\}$, $A_3 = \{ac,ad,ae\}$, $A_4 = \{bc,bd,be\}$,
$A_1^{-1} = \{ba\}$, $A_2^{-1} = \{dc\}$, $A_3^{-1} = \{ca,da,ea\}$, $A_4^{-1} = \{cb,db,eb\}$.

On the other hand, the graph in Figure 5.2 b has only one implication class:

$$A = \{ab,cb,cd,cf,ef,bf,ba,bc,dc,fc,fe,fb\}.$$

Let A be an implication class of an undirected graph, G, and let $\hat{A} = A \cup A^{-1}$ denote the symmetric closure of A. It can be shown that if G has a transitive orientation F, then either $F \cap \hat{A} = A$ (F completely agrees with A) or $F \cap \hat{A} = A^{-1}$ (F completely disagrees with A) and, in either case, $A \cap A^{-1} = \emptyset$. The converse of this is also valid, namely, if $A \cap A^{-1} = \emptyset$ for *every* implication class A, then G has a transitive orientation.

<u>Remark</u>. Many readers may wonder whether an arbitrary union of implication classes $F = \bigcup_i A_i$ satisfying $F \cap F^{-1} = \emptyset$ and $F + F^{-1} = E$ is necessarily a transitive orientation of G. The answer is no. As a counterexample, consider a triangle which has $8 = 2^3$ such orientations two of which fail.to be transitive.

Methods for determining the exact number of transitive orientations t(G) of a given undirected graph G have been developed by Shevrin and Filippov [1970] and Golumbic [1977a], and a characterization of uniquely partially orderable graphs (i.e., t(G) = 2) is given in Shevrin and Filippov [1970] and Trotter, Moore and Sumner [1976]. These results and others are discussed in detail in Golumbic [1980].

We shall now describe an algorithm for calculating transitive orientations and for determining whether or not a graph is a comparability graph. This technique is a modification of one first presented by Pnueli, Lempel and Even [1971]. A discussion of its computational complexity will follow.

Let G = (V,E) be an undirected graph. A partition of the edge set $E = \hat{B}_1 + \hat{B}_2 + \ldots + \hat{B}_k$ is called a G-*decomposition* of E if B_i is an implication class of $\hat{B}_i + \ldots + \hat{B}_k$ for all $i = 1,2,\ldots,k$. A sequence of edges $[x_1y_1, x_2y_2, \ldots, x_ky_k]$ is called a *decomposition scheme* for G if there exists a G-decomposition $E = \hat{B}_1 + \hat{B}_2 + \ldots + \hat{B}_k$ satisfying $x_iy_i \in B_i$ for all $i = 1,2,\ldots,k$. In this section the term *scheme* will always mean a decomposition scheme.

For a given G-decomposition there will be many corresponding schemes (any set of representatives from the B_i). However, for a given scheme there exists exactly one corresponding G-decomposition. A scheme and G-decomposition can be constructed by the following procedure:

Algorithm 5.1. Decpmposition Algorithm

Let $G = (V,E)$ be an undireced graph. Initially let $i = 1$ and $E_1 = E$.

Step I: Arbitrarily pick an edge $e_i = x_iy_i \in E_i$.

Step II: Enumerate the implication class B_i of E_i containing x_iy_i.

Step III: Define $E_{i+1} = E_i - \hat{B}_i$.

Step IV: If $E_{i+1} = \emptyset$, then let $k = i$ and STOP; otherwise, increase i by 1 and go back to step I.

Clearly, the Decomposition Algorithm yields a scheme $[x_1y_1,\ldots,x_ky_k]$ and corresponding G-decomposition $\hat{B}_1 + \ldots + \hat{B}_k$ for any undirected graph G. Moreover, if y_ix_i had been chosen instead of x_iy_i for some i, then B_i^{-1} would replace B_i in the G-decomposition. Applying the algorithm to the graph in Figure 5.3, the scheme [ac,bc,dc] gives the G-decomposition for which $B_1 = A_3$, $B_2 = A_4 + A_1^{-1}$ and $B_3 = A_2^{-1}$. In this example notice that although ba and bc were not Γ-related in the original graph, once $\hat{B}_1$ is removed they become Γ-related in the remaining subgraph and their implication classes merge. In general, it can be shown that each implication class of E_{i+1} will be the union of either 1 or 2 implication classes of E_i.

The next theorem legitimizes the use of G-decompositions as a constructive tool for deciding whether an undirected graph is a comparability graph, and if so, producing a transitive orientation. Proofs of this theorem can be found in Golumbic [1977a] or Golumbic [1980].

Theorem 5.1 (The TRO Theorem) Let $G = (V,E)$ be an undirected graph with G-decomposition $E = \hat{B}_1 + \ldots + \hat{B}_k$. The following statements are equivalent:

(i) $G = (V,E)$ is a comparability graph;

(ii) $A \cap A^{-1} = \emptyset$ for all implication classes A of E;

(iii) $B_i \cap B_i^{-1} = \emptyset$ for $i = 1,\ldots,k$.

Furthermore, when these conditions hold, $B_1 + \ldots + B_k$ is a transitive orientation of E.

By combining the TRO Theorem with the Decomposition Algorithm, we obtain an algorithm for recognizing comparability graphs and assigning a transitive orientation.

Algorithm 5.2. TRO Algorithm

Input: An undirected graph $G = (V,E)$.

Output: A transitive orientation F of edges of G if FLAG has final value 0, or a message that G is not a comparability graph if FLAG has final value 1.

Method: The entire algorithm is as follows:

```
initialize:     $i \leftarrow 1$;  $E_i \leftarrow E$;  $F \leftarrow \emptyset$;   FLAG $\leftarrow 0$;
I:   arbitrarily pick an edge $x_iy_i \in E_i$;
II:  enumerate the implication class $B_i$ of $E_i$ containing $x_iy_i$;
             if $B_i \cap B_i^{-1} = \emptyset$ then
                    add $B_i$ to F;
             else
                  FLAG $\leftarrow 1$;    [G is not a comparability
                                            graph];
III: define $E_{i+1} \leftarrow E_i - \hat{B}_i$;
IV:  if $E_{i+1} = \emptyset$ then
             $k \leftarrow i$; STOP      [F is a transitive
                                          orientation of G];
     else
             $i \leftarrow i + 1$;  go to I;
```

The sequence of arbitrary choices made in line I of the algorithm determines which of the many transitive orientations of G is produced by the algorithm. A different scheme may give a different transitive orientation. But, when you try out a few different schemes you will notice a remarkable

phenomenon: No matter how the arbitrary choices for G are made, the number of iterations k will always be the same. This phenomenon is actually true for any graph G. A characterization of the underlying mathematical structure which causes it is given in Golumbic [1977a, 1980].

A more detailed version of Algorithms 5.1 and 5.2 will suggest how we may construct a G-decomposition and test transitive orientability of an undirected graph $G = (V,E)$ in $O(\delta|E|)$ time and $O(|V| + |E|)$ space where δ is the maximum degree of a vertex. Let $G = (V,E)$ be an undirected graph with vertices $v_1, v_2, \ldots, v_n$. In the algorithm below we use the function

$$\mathrm{CLASS}(i,j) = \begin{cases} 0 & \text{if } v_i v_j \notin E \\ k & \text{if } v_i v_j \text{ has been assigned to } B_k \\ -k & \text{if } v_i v_j \text{ has been assigned to } B_k^{-1} \\ \text{undefined} & \text{if } v_i v_j \in E \text{ has not yet been assigned} \end{cases}$$

and $|\mathrm{CLASS}(i,j)|$ denotes the absolute value of $\mathrm{CLASS}(i,j)$.

Algorithm 5.3. Decomposition Algorithm (detailed version)

Input: The adjacency sets of an undirected graph $G = (V,E)$ with vertices $v_1, v_2, \ldots, v_n$.

Output: A G-decomposition of the graph given by the final value of CLASS, and a variable FLAG which is 0 if the graph is a comparability graph and 1 otherwise. If the algorithm terminates with FLAG equal to 0, then a transitive orientation of G is obtained by combining all edges having positive CLASS.

Method: The algorithm proceeds until all edges have been explored. In the kth iteration an unexplored edge is placed in B_k (its CLASS is changed to k). Whenever an edge is

placed into B_k it is explored using the recursive procedure of Figure 5.4 by adding to B_k those edges Γ-related to it in the graph E_k. (Notice that $v_i v_j \in E_k$ if and only if either $|\text{CLASS}(i,j)|$ equals k or is undefined throughout the kth iteration.) The variable FLAG is changed from 0 to 1 the first time a B_k is found such that $B_k \cap B_k^{-1} \neq \emptyset$. At that point it is known that G is not a comparability graph (by Thoerem 5.1). The algorithm is as follows.

```
initialize:   k ← 0;  FLAG ← 0;
     for each edge $v_i v_j$ in E do
        if CLASS(i,j) is undefined then do
            k ← k + 1;
            CLASS(i,j) ← k;  CLASS(j,i) ← -k;
            EXPLORE(i,j);
        end;
     end;
```

```
        procedure EXPLORE(i,j):
loop 1:    for each m ∈ Adj(i) such that [m ∉ Adj(j)
                            or |CLASS(j,m)| < k] do
              if CLASS(i,m) is undefined then do
                 CLASS(i,m) ← k;  CLASS(m,i) ← -k;
                 EXPLORE(i,m);  end
              else if CLASS(i,m) = -k  then do
                 CLASS(i,m) ← k;  FLAG ← 1;
                 EXPLORE(i,m);  end
           end loop 1
loop 2:    for each m ∈ Adj(j) such that
                 [m ∉ Adj(i) or |CLASS(i,m)| < k] do
              if CLASS(m,j) is undefined  then do
                 CLASS(m,j) ← k;  CLASS(j,m) ← -k;
                 EXPLORE(m,j);  end
              else if CLASS(m,j) = -k then do
                 CLASS(m,j) ← k;  FLAG ← 1;
                 EXPLORE(m,j);  end
           end loop 2
        return
        end
```

Figure 5.4

Complexity analysis: We begin by specifying an appropriate data structure. The adjacency sets are stored as linked lists sorted into increasing order. The element of the list Adj(i) which represents edge $v_i v_j$ will contain j, CLASS(i,j), a pointer to CLASS(j,i), and a pointer to the next element on Adj(i). The storage requirement for this data structure is $O(|V| + |E|)$, and the entire initialization of the data structure can be accomplished in linear time.

The crucial factor in the analysis of our algorithm is the time required to access or assign the CLASS function. Consider the first loop of EXPLORE(i,j). Two temporary pointers simultaneously scan Adj(i) and Adj(j) looking for values of m which satisfy the condition in the *for* statement. This loop can be executed in $O(d_i + d_j)$ steps. The second loop is done similarly, hence the time complexity of EXPLORE(i,j) is $O(d_i + d_j)$.

In the main program, a pointer scans each adjacency list successively in the *for* loop implying a time complexity of $O(|E|)$. Finally, the algorithm calls EXPLORE once for each edge or its reversal (both if their implication classes are not disjoint). Therefore, since

$$\sum_{v_i v_j \in E} (d_i + d_j) = O(\delta |E|)$$

it follows that the time complexity for the entire algorithm (including preprocessing the input) is at most $O(\delta |E|)$.

Coloring and Other Problems on Comparability Graphs

Suppose that G is a comparability graph, and let F be a transitive orientation of G. A *height* function h can be placed on V as follows: $h(v) = 0$ if v is a sink; otherwise, $h(v) = 1 + \max \{h(w) \mid vw \in F\}$. The height function can be assigned in linear time using a recursive depth-first search,

and it is a proper vertex coloring of G. The number of colors used will be equal to the number of vertices in the longest path of F, and since, by transitivity, every path in F corresponds to a clique of G, the height function will yield a coloring which uses exactly $\omega(G)$ colors which is the best possible. Therefore, from the transitive orientation F we can assign a minimum coloring to G using the height function in $O(|V| + |E|)$ steps, and, at the same time, calculate a maximum clique of G. We will illustrate this by solving the more general problem of finding a maximum weighted clique of a comparability graph.

(If all vertices have the same weight, then the problem is reduced to the usual problem of finding a clique of maximum cardinality.) In general the maximum weighted clique problem is NP-complete, but when restricted to comparability graphs it becomes tractable.

Algorithm 5.4 Minimum Coloring and Maximum Weighted Clique of a Comparability Graph.

Input: The adjacency sets of a transitive orientation F of a comparability graph $G = (V,E)$ and a weight function w defined on V.

Output: A minimum coloring of G and a clique K of G whose weight is maximum.

Method: We use a modification of the height calculation technique employing the recursive depth-first search procedure SEARCH in Figure 5.5. To each vertex v we associate its COLOR and its cumulative weight $W(v)$ which equals the weight of the heaviest path from v to some sink. A pointer is assigned to v designating its successor on that heaviest path. Once the cumulative weights are assigned the clique K is calculated beginning the line labeled *retrace*. The algorithm is given in the form of a procedure.

```
          procedure MAXWEIGHT CLIQUE (V,F):
             for all $v \in V$ do
                if v is unsearched then
                   SEARCH(V);
             end
retrace:     select $y \in V$ such that $W(y) = \max\{W(v) \mid v \in V\}$;
             $K \leftarrow \{y\}$;   $y \leftarrow$ POINTER(y);
             while $y \neq \Lambda$ do
                $K \leftarrow K \cup \{y\}$;   $y \leftarrow$ POINTER(y);
             end
             return K;
          end
```

We conclude with an interesting polynomial-time method for finding $\alpha(G)$, the size of the largest stable set of a comparability graph G. We transform a transitive orientation (V,F) of G into a transportation network by adding two new vertices s and t and edges sx and yt for each source x and sink y of F. Assigning a lower capacity of 1 to each vertex, we initialize a compatible integer-valued flow and then call a minimum-flow algorithm. The value of the minimum flow will equal the size of the smallest covering of the vertices by cliques which in turn will equal the size of the largest independent set since every comparability graph is perfect. Such a minimum flow algorithm can run in polynomial time.

```
procedure SEARCH(v):
   if Adj(v) = ∅ then do
      W(v) = w(v);  POINTER(v) ← Λ;  COLOR(v) ← 0;  end
   else do
      for all x ∈ Adj(v) do
         if x is unsearched then
            SEARCH(x);  end
      select y ∈ Adj(v) such that W(y) = max{W(x)|x∈Adj(v)};
      W(v) ← w(v) + W(y);  POINTER(v) ← y;
      select z ∈ Adj(v) such that COLOR(z)=max{COLOR(z) |
                                                 z ∈ Adj(v)};
      COLOR(v) ← 1 + COLOR(z);
      end
   return
end
```

Figure 5.5

6. Split Graphs

An undirected graph $G = (V,E)$ is a *split graph* if there is a partition $V = S + K$ of its vertex set into a stable set S and a complete set K. Since a stable set of G is a complete set of the complement $\bar{G}$ and vice versa, G is a split graph if and only if its complement $\bar{G}$ is a split graph. Földes and Hammer [1977] have given the following characterization of split graphs.

Theorem 6.1. Let G be an undirected graph. The following conditions are equivalent:

(i) G is a split graph

(ii) G and $\bar{G}$ are triangulated graphs

(iii) G contains no induced subgraph isomorphic to $2K_2$, C_4 or C_5.

An alternate characterization of split graphs in terms of degree sequences is the following result of Hammer and Simeone [1977].

Theorem 6.2. Let $G = (V,E)$ be an undirected graph with degree sequence $d_1 \geq d_2 \geq \ldots \geq d_n$, and let $m = \max\{i \mid d_i > i-1\}$. Then, G is a split graph if and only if

$$\sum_{i=1}^{m} d_i = m(m-1) + \sum_{i=m+1}^{n} d_i .$$

Furthermore, if this is the case, the m vertices of largest degree will be a maximum complete set of G.

A simple recognition algorithm for split graphs can be designed by applying Theorem 6.2. If this is done, it can easily be seen that the complexity of recognizing split graphs is $O(n \log n)$. The same complexity applies for the clique problem and the stable set problem on split graphs. However, the Hamiltonian circuit problem on split graphs is NP-complete.

7. Permutation Graphs

Let $\pi = [\pi_1, \pi_2, \cdots, \pi_n]$ be a permutation of the numbers $1,2,\ldots,n$. We define the undirected graph $G[\pi] = (V,E)$ as follows:

$$V = \{v_1, v_2, \ldots, v_n\}$$

and

$$(v_i, v_j) \in E \quad \text{iff} \quad (i-j)(\pi_i^{-1} - \pi_j^{-1}) < 0 .$$

Two vertices are joined by an edge if they occur out of their proper order reading the sequence π left to right (see Figure 7.1).

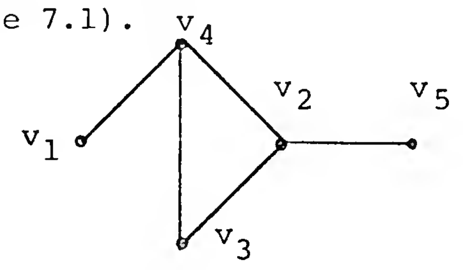


Figure 7.1. The Graph $G[4,1,3,5,2]$.

If we *reverse* the sequence π, each pair of numbers which occur in the correct order in π will now be in the wrong order, and vice versa. Thus, the permutation graph we obtain will be the complement of $G[\pi]$. This shows that the complement of a permutation graph is also a permutation graph.

Another property of the graph $G[\pi]$ is that it is transitively orientable. If we orient each edge toward its larger endpoint, then we will obtain a transitive orientation F. For, suppose $(v_i, v_j) \in F$ and $(v_j, v_k) \in F$, then $i < j < k$ and $\pi_i^{-1} > \pi_j^{-1} > \pi_k^{-1}$, which implies that $(v_i, v_k) \in F$. This is only half of the story; we actually have the following result of Pneuli, Lempel, and Even [1971].

<u>Theorem 7.1</u>. An undirected graph G is a permutation graph if and only if G and $\bar{G}$ are comparability graphs.

Theorem 7.1 suggests an algorithm for recognizing permutation graphs, namely, applying the transitive orientation algorithm to the graph and to its complement. If we succeed in finding transitive orientations, then the graph is a permutation graph. To find a suitable permutation we can follow the construction procedure in the proof of the theorem, which can be found in Golumbic [1980]. The entire method requires $O(n^3)$ time and $O(n^2)$ space.

Permutation graphs are useful in a number of applications (Even, Pnueli, and Lempel [1972], Tarjan [1972], Golumbic [1980]). Of particular interest in this context is the following very efficient coloring algorithm for G[π].

<u>Algorithm 7.1. Coloring a Permtuation Graph</u>

<u>Input</u>: A permutation $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ of the numbers $\{1, 2, \ldots, n\}$.

<u>Output</u>: A coloring of the vertices G[π] and the chromatic number χ of G[π].

<u>Method</u>: The vertices of G[π] are assigned colors in the order $\pi_1, \pi_2, \ldots, \pi_n$, although the graph itself is never actually calculated. A counter k will keep track of the total number of colors used so far, and an array LAST(c) will contain the number of the vertex which was the last

to receive color c. During the jth time through the loop we color π_j with the smallest color q satisfying $\pi_j \geq \text{LAST}(q)$. The entire algorithm is as follows:

```
              procedure
1. initialize:  k ← 0;  for i←1 to n do LAST(i) ← 0;  end
2. loop:        for j ← 1 to n do
3.                 m ← min {q | $\pi_j \geq$ LAST(q)};
4.                 COLOR($\pi_j$) ← m;
5.                 LAST(m) ← $\pi_j$;
6.                 k ← max{k,m};
                end loop
7.              $\chi$ ← k;
              end
```

Example. Let us illustrate Algorithm 7.1 on the permutation $\pi = [4,1,3,5,2]$. After the initializations in line 1 the following assignments will be made in the loop:

j ← 1	j ← 2	j ← 3	j ← 4	j ← 5
m ← 1	m ← 2	m ← 2	m ← 2	m ← 3
COLOR(4) ← 1	COLOR(1) ← 2	COLOR(3) ← 2	COLOR(5) ← 2	COLOR(2) ← 3
LAST(1) ← 4	LAST(2) ← 1	LAST(2) ← 3	LAST(2) ← 5	LAST(3) ← 2
k ← 1	k ← 2	k ← 2	k ← 2	k ← 3

Thus the chromatic number of $G[\pi]$ is 3 and a 3-coloring has been assigned.

The complexity of Algorithm 7.1 is $O(n \log \chi)$ if line 3 is implemented using binary search. A proof of the correctness of this algorithm can be found in Golumbic [1981]. Algorithm 7.1 can be used to color any permutation graph G in $O(n \log n)$ time provided we are given the permutation π and the isomorphism $G \rightarrow G[\pi]$. If we do not have π, then we would use Algorithm 5.4.

8. Interval Graphs

An undirected graph G is called an *interval graph* if its vertices can be put into one-to-one correspondence with a set of intervals I of a linearly ordered set (like the real line) such that two vertices are connected by an edge of G if and only if their corresponding intervals have nonempty intersection. We call I an *interval representation* for G. (It is unimportant whether we use open intervals or closed intervals; the resulting class of graphs will be the same.)

The following characterization of interval graphs is due to Gilmore and Hoffman [1964].

Theorem 8.1. An undirected graph G is an interval graph if and only if G is a triangulated graph and its complement $\bar{G}$ is a comparability graph.

The coloring, clique, stable set, and clique cover problems can be solved in polynomial time for interval graphs by using the algorithms of Sections 4 or 5, and a recognition algorithm could be obtained by combining the algorithms for triangulated graphs and comparability graphs. However, the recognition algorithm presented in Booth and Lueker [1976] is asymptotically more efficient. They have shown that a data structure called a PQ-tree can be used to obtain a linear algorithm.

Interval graphs have become particularly useful mathematical structures for modeling real world problems. The line, on which the intervals rest, may represent anything that is normally regarded as one-dimensional. The linearity may be due to *physical restriction* such as blemishes on a microorganism, speed traps on a highway, or files in sequential storage in a computer. It may arise from *time dependencies* as in the case of the life span of persons or cars, or jobs

on a fixed time schedule. A *cost function* may be the reason as with the approximate worth of some fine wines or the potential for growth of a portfolio of securities.

The task to be performed on an interval graph will vary from problem to problem. If what is required is to find a coloring or a maximum weighted stable set or a large clique, then fast algorithms are available. If a Hamiltonian circuit must be found, then there are no known efficient algorithms (unless the graph has more structure than just being an interval graph). Also, the speed with which such a problem can be solved will depend partially on whether we are given simply the interval graph G, or, in addition, an interval representation of G.

We have already seen one application of interval graphs in the opening paragraph of this article. The interested reader is referred to Roberts [1976, 1978] and Golumbic [1980] for numerous other applications. We will discuss here a recent application of interval graphs to optimal macro substitutions suggested by Golumbic, Goss, and Dewar [1980].

The compiler or interpreter for a microcomputer system may be regarded as a byte sequence which resides in main memory. Due to restrictions on the size of main memory, it is desirable to compact this byte sequence. One technique is to define a set of macro substitutions which allow occurrences of specified byte subsequences to be replaced by single bytes. The subsequences are restored dynamically at run time by use of an associated table.

Figure 8.1 shows a sequence of hexidecimal digits of length 36. Since the digits E and F do not appear, they may be used to indicate macros. Choosing E = 6A2 and F = 43B96 the original sequence may be reduced to length 20. Notice that when two macros overlap, only one can be replaced. This overlapping phenomenon, therefore, restricts how the macro table may be applied.

Original Sequence : $\underline{6A2}C\overline{43B96}0D60\underline{6A2}1C78\underline{6A2}\overline{43B9}\underline{\overline{6}A2}3C\underline{6A2}5$

Macro Table: E=6A2 F=43B96 OVERLAP

Abbreviated Sequence: ECF0D60E1C78EFA23CE5

Figure 8.1. Macro Substitution.

The problem to be solved is to choose an optimal set of macro substitutions and an order for performing the substitutions which minimizes the total length of the byte sequence and associated table. Formally we require the following.

Input: A byte sequence B of length n.

Output: A set of m *macros* each of length $\leq$ k and an *order for performing the substitutions* such that the total length of the abbreviated sequence and macro table is minimized.

The reason for specifying a bound on the length of the macros is that in practice we may want them to be very short compared to the length of the original sequence.

Notice that there are actually two aspects to the problem:

(1) choosing a macro set, and

(2) using the macro set optimally.

Let $B = \langle b_1, b_2, \ldots, b_n \rangle$ be a sequence of bytes and let k be a fixed constant. The length of B is denoted by $|B| = n$. A subsequence $\langle b_i, \ldots, b_j \rangle$ of B is denoted by B[i,j]. Clearly, $|B[i,j]| = j-i+1$. The *weighted interval graph* G = (V,E,w) that we will associate with B is defined as follows: The vertex set V consists of all intervals [i,j] satisfying $1 \leq j-i \leq k-1$; two vertices v = [i,j] and u = [i',j'] are connected by an edge iff they intersect, i.e., either $i' \leq j \leq j'$ or $i \leq j' \leq j$; the weight w(v) of a vertex v = [i,j] is equal to j-i which represents the number of bytes that would be saved by replacing B[i,j] by a single byte.

It is easy to see that the number of vertices of G is slightly less than kn and the number of edges is less than but on the order of k^3n. Furthermore, the graph does not actually have to be calculated and stored since any query about adjacency of vertices can be answered by a simple comparison of the indices of their corresponding subsequences.

Let M be a subset of V and let

$$B[M] = \{B[i,j] \mid [i,j] \in M\}.$$

We may think of B[M] as the macro table generated by M. To perform the macro substitutions we would find all occurrences of these macros and then choose a subset of the occurrences, no two of which intersect, to be abbreviated. Such a subset corresponds precisely to a stable set of the interval graph G. (Notice that this model does not permit embedding one macro in another macro.) Moreover, to make the abbreviated sequence as short as possible, we would like a stable set whose weight is maximum. (The weight of a subset of vertices is the sum of the weights of its members.) This method is summarized in Figure 8.2.

```
procedure SUBSTITUTION(M):
   $C(M) \leftarrow \{[i,j] \in V \mid B[i,j]=B[i',j']$ for some $[i',j'] \in M\}$;
   $X(M) \leftarrow$ MAXIMUM WEIGHTED STABLE SET OF THE
            INDUCED SUBGRAPH $G_{C(M)}$;
   $SAVINGS(M) \leftarrow \sum_{u \in X(M)} w(u) - \sum_{v \in M} w(v)$;
end
```

Figure 8.2. Finding an Optimal Macro Substitution for a Given Set of Macros

The set C(M) consists of all intervals representing "candidate" subsequences which may be replaced using the macro table B[M]. Of these candidates only the subsequences

represented by $X(M)$ will be replaced. The SAVINGS is calculated by summing the savings obtained for each macro substitution and subtracting the cost of storing the macro table.

Using SUBSTITUTION we obtain the following algorithm which gives an optimal solution to the general problem.

Algorithm 8.1

```
loop:    for all $M \subseteq V$ such that $|M| = m$ do
            call SUBSTITUTION(M);
         end loop
         return the M and X(M) whose SAVINGS(M) is maximum;
```

The number of passes through the loop in Algorithm 8.1 is on the order of $\binom{kn}{m}$ since G has $O(kn)$ vertices. (In practice, some of the subsets M may be ruled out due to other criteria, for example, by requiring that macros begin with certain designated bytes. This would lower the number of passes.) The complexity of SUBSTITUTION depends on how efficiently we are able to find $C(M)$ and $X(M)$ for a given M. Using a modification of the deterministic pattern matching algorithm of Morris and Pratt [1970], $C(M)$ can be calculated in $O(m(k+n))$ time. See also Aho, Hopcroft and Ullman [1976, Chapter 9]. Since a maximum stable set of an interval graph $G = (V,E)$ may be found in time $O(|V| + |E|)$, $X(M)$ can be calculated in $O(k^3 n)$ time. Hence, we conclude that the worst case complexity of SUBSTITUTION is $O(m(k+n)+k^3 n)$ and the worst case complexity of Algorithm 8.1 is

$$O(c_{m,k}\, n^{m+1}) \quad \text{where} \quad c_{m,k} \approx \left[\frac{ek}{m}\right]^m \frac{k^3}{\sqrt{2\pi m}}$$

which is, in terms of the length of the input sequence, a polynomial whose degree depends on the constant m.

Notice that our model has not allowed the embedding of macros in other macros. A reason for this could be that it is impractical to implement the stack necessary to allow embedding. In some applications one may choose to allow embedding. If this is the case, a similar model can be designed which uses overlap graphs rather than interval graphs. An *overlap graph* is the same as an interval graph in which there are no edges between pairs of vertices whose corresponding intervals have one properly contained in the other.

Our Algorithm 8.1 and SUBSTITUTION will also be optimal using the overlap graph model. Their respective complexities, in this case, will each be raised by one power of kn. This follows from the fact that a maximum weighted stable set of an overlap graph $G = (V,E)$ can be calculated in $O(|V|\cdot|E|)$ time, (see Gavril [1973], and Golumbic [1980, Chapter 11].

The problem of macro substitution was recently applied to MICRO SPITBOL for an Incoterm SPD20/40 supporting 64K of main memory. The byte sequence for MICRO SPITBOL required 23,110 bytes of storage. There were 176 unused opcodes which were designated to represent macros. That is, $n = 23110$ and $m = 176$ and we set $k = 20$.

Since the time complexity of Algorithm 8.1 would be high for this application, an effective technique for finding a ne-- optimal solution was needed. A combination of heuristics and SUBSTITUTE reduced the size of the sequence to 17,920 bytes and produced a macro table of 962 bytes. This represents a saving of 4,228 bytes of main storage, a saving of 20%. It should be pointed out that an increased cost of obtaining a very good macro substitution may be justified by the fact that this is done only once per compiler and machine and the result presumably will be used many, many times.

References

A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* Addison-Wesley, 1976.

C. Berge, *Graphs and Hypergraphs,* North-Holland, Amsterdam, 1973.

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

G. A. Dirac, On rigid circuit graphs, *Abh. Math. Sem. Univ. Hamburg 25* (1961), 71-76.

S. Even, A. Pnueli, and A. Lempel, Permutation graphs and transitive graphs, *J. Assoc. Comput. Mach. 19* (1972), 400-410.

S. Földes and P. L. Hammer, Split graphs, *Proc. 8th Southeastern Conf. on Combinatorics, Graph Theory and Computing* (F. Hoffman *et al.*, eds.), *Congressus Numerantium XIX*, Utilitas Math., Winnipeg, 1977, pp. 311-315.

D. R. Fulkerson and O. A. Gross, Incidence matrices and interval graphs, *Pacific J. Math. 15* (1965), 835-855.

F. Gavril, Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph, *SIAM J. Computing 1* (1972), pp. 180-187.

F. Gavril, Algorithms for a maximum clique and a minimum independent set of a circle graph, *Networks 3* (1973), pp. 261-273. [The class of circle graphs is equivalent to the class of overlap graphs.]

P. C. Gilmore and A. J. Hoffman, A characterization of comparability graphs and of interval graphs, *Canadian J. of Math. 16* (1964), pp. 539-548.

M. C. Golumbic, Comparability graphs and a new matroid, *J. Combin. Theory B 22* (1977a), 68-90.

M. C. Golumbic, The complexity of comparability graph recognition and coloring, *Computing 18* (1977b), 199-208.

M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, N. Y., 1980.

M. C. Golumbic, *An Introduction to Algorithmic Analysis and Combinatorial Algorithms*, to appear, 1981.

M. C. Golumbic, C. F. Goss, and R.B.K. Dewar, Macro Substitutions in MICRO SPITBOL — A combinatorial analysis, *Proc. 11th Southeastern Conf. on Combinatorics, Graph Theory, and Computing* (F. Hoffman *et al.* eds.), *Congressus Numerantium*, Utilitas Math., Winnipeg, 1980.

M. Grötschel, L. Lovász, and A. Schrijver, The ellipsoid method and its consequences in combinatorial optimization, Inst. für Ökonometrie und Operations Research, Univ. Bonn, Report No. 80151-OR, Jan. 1980.

P. L. Hammer and B. Simeone, The splittance of a graph, Univ. of Waterloo, Research Report CORR-77-39, 1977.

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

J. H. Morris and V. R. Pratt, A linear pattern matching algorithm, Tech. Report No. 40, Computing Center, University of California, Berkeley, Calif., 1970.

A. Pnueli, A. Lempel, and S. Even, Transitive orientation of graphs and identification of permutation graphs, *Canad. J. Math. 23* (1971), 160-175.

F. S. Roberts, *Discrete Mathematical Models, with Applications to Social, Biological and Environmental Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

F. S. Roberts, *Graph Theory and Its Application to Problems of Society*, NSF-CBMS Monograph No. 29, SIAM Publ., Philadelphia, Pa., 1978.

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

L. N. Shevrin and N. D. Filippov, Partially ordered sets and their comparability graphs, *Siberian Math. J. 11* (1970), 497-509.

R. E. Tarjan, Sorting using networks of queues and stacks, *J. Assoc. Comput. Mach. 19* (1972), 341-346.

R. E. Tarjan, Maximum cardinality search and chordal graphs, Stanford Univ. Lecture Notes CS 259, unpublished, 1976.

DATE DUE	BORROWER'S NAME
[illegible]	J. Hsieh