ALGORITHMIC ASPECTS OF VERTEX ELIMINATION ON DIRECTED GRAPHS

by

Donald J. Rose
Robert E. Tarjan

STAN-CS-75-531
NOVEMBER 1975

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

ALGORITHMIC ASPECTS OF VERTEX ELIMINATION ON DIRECTED GRAPHS

Donald J. Rose [*]
Applied Mathematics, Aiken Computation Laboratory
Harvard University, Harvard, Massachusetts 02138

Robert Endre Tarjan [**]
Computer Science Department
Stanford University, Stanford, California 94305

Abstract

We consider a graph-theoretic elimination process which is related to performing Gaussian elimination on sparse systems of linear equations. We give efficient algorithms to:

(1) calculate the fill-in produced by any elimination ordering;

(2) find a perfect elimination ordering if one exists; and

(3) find a minimal elimination ordering.

We also show that problems (1) and (2) are at least as time-consuming as testing whether a directed graph is transitive, and that the problem of finding a minimum ordering is NP-complete.

Keywords: directed graph, elimination ordering, fill-in, Gaussian elimination, NP-complete problem, perfect elimination graph, sparse linear system.

[*] Research partially supported by the Office of Naval Research under contract N00014-67-A-0298-0034 at Harvard University.

[**] Research partially supported by a Miller Research Fellowship at University of California, Berkeley; by National Science Foundation, grant DCR72-03752 A02 and by the Office of Naval Research contract NR 044-402. Reproduction in whole or in part is permitted for any purpose of the United States Government.

1. Introduction and Notation.

A directed graph (digraph) is a pair $G = (V,E)$ where V is a finite set of $n = |V|$ elements called vertices and $E \subset \{(v,w) \mid v,w \in V, v \neq w\}$ is a set of $e = |E|$ ordered vertex pairs called edges. Given $v \in V$, the set $A(v) = \{w \in V \mid (v,w) \in E\}$ is the set of vertices adjacent out from v; the set $B(v) = \{u \in V \mid (u,v) \in E\}$ is the set of vertices adjacent into v. $A(v)$ and $B(v)$ are called the adjacency sets for vertex v. $|A(v)| = d_O(v)$ is the out-degree of v; $|B(v)| = d_I(v)$ is the in-degree of v. The notation $v \rightarrow w$ means $w \in A(v)$; $v \not\rightarrow w$ means $w \notin A(v)$. If $W \subseteq V$, the induced subgraph $G(W)$ of G is the subgraph $G(W) = (W,E(W))$ where

$$E(W) = \{(w,y) \in E \mid x,y \in W\} .$$

For distinct vertices $v,w \in V$, a v,w path of length k is a sequence of distinct vertices $\mu = [v = v_1, v_2, \ldots, v_{k+1} = w]$ such that $v_i \rightarrow v_{i+1}$ for $i = 1,2,\ldots,k$. A graph is strongly connected if there is a path from every vertex to every other. A cycle of length k is a sequence of distinct vertices $\mu = [v_1, v_2, \ldots, v_k]$ such that $v_i \rightarrow v_{i+1}$ for $1,2,\ldots,k$ and $v_k \rightarrow v_1$. A graph is acyclic if it has no cycles. The transitive closure of graph $G = (V,E)$ is the graph $G^+ = (V,E^+)$, where $(v,w) \in E^+$ if and only if $v \neq w$ and there is a path from v to w in G. A graph G is transitive if it is equal to its transitive closure. Equivalently, G is transitive if and only if for all u,v,w, $u \rightarrow v$ and $v \rightarrow w$ together imply $u \rightarrow w$ or $u = w$. A clique is a graph $G = (V,E)$ with $E = \{(v,w) \mid v,w \in V, v \neq w\}$.

For a graph $G = (V,E)$ with $|V| = n$, an ordering of V is a bijection $\alpha: \{1,2,\ldots,n\} \leftrightarrow V$. $G_\alpha = (V,E,\alpha)$ is an ordered graph.

For a vertex v, the <u>deficiency</u> $D(v)$ is the set of edges defined by

$$D(v) = \{(x,y) \mid x \to v, v \to y, x \not\to y, x \neq y\} .$$

The graph $G_v = (V-\{v\}, E(V-\{v\}) \cup D(v))$ is called the <u>v-elimination graph</u> of G. For an ordered graph $G_\alpha = (V,E,\alpha)$, the <u>elimination process</u>

$$P(G_\alpha) = [G = G_\alpha, G_1, G_2, \cdots, G_{n-1}]$$

is the sequence of elimination graphs defined recursively by $G_0 = G$, $G_i = (G_{i-1})_{\alpha(i)}$ for $i = 1,2,\ldots,n-1$. If $G_i = (V_i,E_i)$ for $i = 0,1,\ldots,n-1$, the <u>fill-in</u> $F(G_\alpha)$ is defined by

$$F(G_\alpha) = \bigcup_{i-1}^{n-1} D_{i-1}(\alpha(i))$$

where $D_{i-1}(\alpha(i))$ is the deficiency of $\alpha(i)$ in G_{i-1}, and the <u>elimination graph</u> G_α^* is defined by

$$G_\alpha^* = (V, E \cup F(G_\alpha)) .$$

The notion of vertex elimination arises in the solution of sparse systems of linear equations by Gaussian elimination. Given any $n \times n$ matrix $M = (m_{ij})$ which represents the coefficients of a set of linear equations, we can construct an ordered graph $G_\alpha = (V,E,\alpha)$, where vertex v_i corresponds to row i, and $(v_i,v_j) \in E$ if and only if $m_{ij} \neq 0$ and $i \neq j$. The unordered graph $G = (V,E)$ corresponds to the equivalence class of matrices PMP^T, where P is any permutation matrix.

If we solve the system with coefficients M using Gaussian elimination, eliminating variables in the order $1,2,\ldots,n$, then the edges $D_{i-1}(v_i)$ correspond exactly to the new non-zero elements created when row i is eliminated (assuming no lucky cancellation of non-zero elements). For further discussion of this correspondence, see [10,11]. To make the elimination process efficient, we might, for example, like to create no more non-zero elements than necessary, that is, to find an elimination ordering which minimizes the fill-in.

Given a graph $G = (V,E)$, an ordering α of V is a perfect elimination ordering of G if $F(G_\alpha) = \emptyset$. Thus α is a perfect ordering if $u \to v$, $v \to w$, $\alpha^{-1}(u) < \min(\alpha^{-1}(v), \alpha^{-1}(w))$ together imply $u = w$ or $u \to w$. A graph which has a perfect elimination ordering is called a perfect elimination graph. An ordering α is a minimal elimination ordering of G if no other ordering β satisfies $F(G_\beta) \subset F(G_\alpha)$ where the containment is proper. An ordering α is a minimum elimination ordering of G if no other ordering β satisfies $|F(G_\beta)| < |F(G_\alpha)|$. Any elimination graph G_α^* is a perfect elimination graph, since α is a perfect ordering of this graph. Any perfect ordering of a graph is minimum, and any minimum ordering is minimal. If a graph is a perfect elimination graph, any minimal ordering is perfect.

A problem important in practice is that of finding a minimum elimination ordering for any graph G. We shall show that this problem is NP-complete.*/ To balance this negative result, we give polynomial-time algorithms for some simpler problems. We present

*/ The so-called NP-complete problems are the hardest problems solvable by non-deterministic Turing machines in polynomial time. Either all the NP-complete problems have polynomial-time algorithms, or none of them do. Such famous hard problems as the traveling salesman problem, the satisfiability problem of propositional calculus, and the maximum clique problem are NP-complete. See [2,6].

methods for

(1) computing the fill-in produced by any ordering α, in $O(ne)$ time;

(2) generating a perfect ordering, if one exists, in $O(ne)$ time;

(3) generating a minimal ordering, in $O(n^4)$ time.

(For all time bounds we assume $n < e+1$, which holds for all graphs that are at least weakly connected.)

We shall also show that any method for either (1) or (2) can be used to test whether an arbitrary directed graph is transitive. Thus, achieving a time bound of better than $O(\min(ne, n^{2.81}))$ for either of these problems would improve on the best bound known for Boolean matrix multiplication [3]. The restrictions to undirected graphs of problems (1), (2), (3) have been considered in [12], which presents an $O(n+e')$ algorithm for (1) (where e' is the number of edges in G^*_α), an $O(n+e)$ algorithm for (2), and an $O(ne)$ algorithm for (3) (Ohtsuki [9] has also devised an $O(ne)$ algorithm for (3)).

This paper is divided into several sections. Section 2 contains some properties of fill-in and elimination orderings which provide a basis for the algorithms. Section 2 also informally describes the algorithms. Section 3 contains implementations of the algorithms and analyses of their running times. Section 4 describes relationships among problems (1), (2), and the transitive closure problem, and between the minimum ordering problem and the NP-complete problems.

2. Properties of Fill-in and Elimination Orderings.

Our first result characterizes the fill-in produced by any elimination ordering.

Lemma 1. Let $G = (V,E,\alpha)$ be an ordered digraph. Then (v,w) is an edge of $G_\alpha^* = (V, E \cup F(G_\alpha))$ if and only if there exists a path $\mu = [v = v_1, v_2, \ldots, v_{k+1} = w]$ in G_α such that

(1) $\quad \alpha^{-1}(v_i) < \min(\alpha^{-1}(v), \alpha^{-1}(w)) \quad$ for $2 \leq i \leq k$.

Proof. We show by induction on $\ell = \min(\alpha^{-1}(v),\alpha^{-1}(w))$ that, given any edge (v,w) in G_α^* , there exists a path from v to w with the required property (1). If $\ell = 1$, (v,w) is in G and (1) holds vacuously. Suppose the result holds for all $1 \leq \ell \leq \ell_0$ and let $\ell = \ell_0+1$. If $v \to w$ in G then (1) again holds vacuously. Otherwise $(v,w) \in F(G_\alpha)$ and we have by the definition of $F(G_\alpha)$ an $x \in V$ with $\alpha^{-1}(x) < \min(\alpha^{-1}(v),\alpha^{-1}(w))$ and $v \to x$, $x \to w$ in G^*_α . The induction hypothesis implies the existence of v, x and x, w paths in G_α satisfying (1) and combining these paths gives the required v, w path.

The converse is established by induction on k , the length of μ . If $k = 1$, $v \to w$ in G_α^* trivially. Suppose the converse holds for all $k \leq k_0$ and let $k = k_0+1$. Let $\mu = [v = v_1, v_2, \ldots, v_{k+1} = w]$ be a path in G_α satisfying (1) and choose $x = v_i$ such that $\alpha^{-1}(v_i) = \max\{\alpha^{-1}(v_j) \mid 2 \leq j \leq k\}$. The induction hypothesis implies that $v \to x$ and $x \to w$ in G_α^* , hence $v \to w$ in G^*_α .

We can use this lemma as the basis of an $O(ne)$ time algorithm for computing the fill-in produced by any ordering α. It is sometimes more efficient, however, to compute the fill-in directly. If v and w are any vertices, $(v,w) \in E \cup F$ if and only if either $(v,w) \in E$ or there is a u such that $\alpha^{-1}(u) < \min\{\alpha^{-1}(v), \alpha^{-1}(w)\}$, $(v,u) \in E \cup F$, and $(u,w) \in E \cup F$. We can compute the fill-in edges (v,w), in increasing order on the value of $\alpha^{-1}(v)$, by using this observation. This method of computation is called row elimination or Doolittle elimination [4] when it is used to carry out numeric, rather than symbolic, Gaussian elimination. Section 3 discusses two algorithms for computing fill-in, one based on this direct method and one based on Lemma 1.

Lemma 2. Let $G = (V,E)$ be a perfect elimination graph, with perfect ordering α. Let $x \in V$ and let $G' = (V, E \cup D(x))$. Then α is a perfect elimination ordering of G'.

Proof. We must show that given $w \rightarrow y$, $y \rightarrow z$ in G' with $w \neq z$ and $\alpha^{-1}(y) < \min(\alpha^{-1}(w), \alpha^{-1}(z))$, it follows that $w \rightarrow z$ in G'. We must consider three cases. If $(w,y),(y,z) \in E$, $(w,z) \in E$ since α is perfect. If $(w,y),(y,z) \in D(x)$, then $w \rightarrow x$, $x \rightarrow z$ in G and $(w,z) \in E \cup D(x)$. The last case is $(w,y) \in E$, $(y,z) \in D(x)$ (or equivalently $(w,y) \in D(x)$, $(y,z) \in E$). In this case $y \rightarrow x$, $x \rightarrow z$ in G and $y \not\rightarrow z$ in G. If $w = x$, then $(w,z) \in E$. Otherwise (i.e., if $w \neq x$), $\alpha^{-1}(x) > \alpha^{-1}(y)$ since α is perfect, and $w \rightarrow x$ in G, also since α is perfect. But $w \rightarrow x$, $x \rightarrow z$ in G imply $(w,z) \in E \cup D(x)$. ☐

Corollary 1. If $G = (V,E)$ is a perfect elimination graph and x is any vertex, the x-elimination graph $G_x = \{V-\{x\}, E(v-\{x\}) \cup D(x)\}$ is also a perfect elimination graph.

Corollary 2. If $G = (V,E)$ is a perfect elimination graph and x is any vertex with $D(x) = \emptyset$, there is a perfect elimination ordering α with $\alpha(1) = x$.

Corollary 2 implies the correctness of the following algorithm for finding a perfect ordering if one exists.

```
algorithm PERFECT(G): begin
   i := 0;
   while G has some vertex v with $D(v) = \emptyset$ do begin
      i := i+1;
      $\alpha(i)$ := v;
      delete v and all incident edges from G;
end end PERFECT;
```

If this algorithm succeeds in ordering all the vertices, G is perfect; if not, G is not perfect. Section 3 gives an $O(ne)$ time implementation of this algorithm.

Our next results give properties of minimal orderings. Let $G = (V,E)$ be a graph. A set F of edges is a fill-in for G if $E \cap F = \emptyset$ and $G' = (V, E \cup F)$ is a perfect elimination graph. F is a minimal fill-in if no set $F_0 \subset F$ is such that $G_0 = (V, E \cup F_0)$ is a perfect elimination graph.

<u>Lemma 3</u>. Let $G = (V,E)$ be a perfect elimination graph. Suppose $F \neq \emptyset$ is a fill-in for G. Let $G' = (V, E \cup F)$. Then $\exists f \in F$ such that $G'-f = (V, E \cup F - \{f\})$ is perfect elimination (i.e., $F-\{f\}$ is a fill-in).

<u>Proof</u>. We prove the lemma by induction on $n = |V|$. If $n \leq 2$, the result is obvious since any graph with one or two vertices is perfect elimination. Suppose the result is true for all $n \leq n_0$ and let $n = n_0+1$. Let $R = \{x \mid D(x) = \emptyset\}$ where $D(x)$ is the deficiency in G and let $S = \{x \mid D'(x) = \emptyset\}$ where $D'(x)$ is the definiciency in G'. We know $R \neq \emptyset$ and $S \neq \emptyset$. We must consider two cases.

(i) For some $x \in S$ there exists an edge $f \in F$ of the form $f = (u,x)$ or $f = (x,u)$. By Corollary 2 there is a perfect elimination order β for G' with $\beta(1) = x$. Then β is also a perfect order for $G'-f$.

(ii) Case (i) does not hold. We prove $\exists x \in S$ with $F \not\subseteq D(x)$. Pick any $z \in S$. If $F \not\subseteq D(z)$, let $x = z$. Otherwise, since $D(z) \subseteq F$, $F = D(z)$. In this case, let x be any vertex in R. By Corollary 2, there is a perfect ordering α of G such that $\alpha(1) = x$, and by Lemma 2, α is a perfect ordering of G'. Thus $x \in S$. Since $D(x) = \emptyset$, $F \not\subseteq D(x)$.

Now $G_x = (V-\{x\}, E(V-\{x\}) \cup D(x))$ and $G'_x = (V-\{x\}, E(V-\{x\}) \cup F \cup D(x))$ are perfect elimination by Corollary 1. By the induction hypothesis $\exists f \in F-D(x)$ such that G'_x-f is perfect elimination. But then $G'-f$ is perfect elimination since $f \notin D(x)$. □

Lemma 3 gives the following theorem.

Theorem 1. Let $G = (V,E,\alpha)$ be an ordered graph. Then α is a minimal elimination ordering if and only if for each $f \in F(G_\alpha)$, $G_\alpha^* - f = (V, E \cup F(G_\alpha) - \{f\})$ is not perfect elimination.

Suppose $G = (V,E)$ is a graph and F is a fill-in for G. Lemmas 2 and 3 lead to the following recursive procedure for finding a minimal fill-in $F_0 \subseteq F$.

```
procedure MINFILL $(V,E,F,F_0)$; begin
   declare $F_1, F_2$ set variables local to procedure MINFILL;
   if $G = (V,E)$ has no vertex $x$ with $D(x) \subset F$ then $F_0 := F$
   else begin
        let $x$ be a vertex with $D(x) \subset F$;
call 1: MINFILL $(V-\{x\}, E(V-\{x\}) \cup D(x), F(V-\{x\}) - D(x), F_1)$;
        if $D(x) = \emptyset$ then $F_0 := F_1$
        else begin
   call 2: MINFILL $(V, E \cup F_1, D(x), F_2)$;
           $F_0 := F_1 \cup F_2$;
end end end MINFILL;
```

It is not hard to see that this procedure works correctly: If every vertex x has $D(x) = F$ then F is clearly minimal. If some vertex x has $D(x) \subset F$, then the first recursive call on MINFILL produces a fill-in $F_1 \subseteq F$ minimal for G_x. If $D(x) = \emptyset$, F_1 is minimal for G by Corollary 2. If $D(x) \neq \emptyset$, then for all proper subsets $F_1' \subset F_1$ and all subsets $F_2 \subseteq D(x)$, $G'' = (V, E \cup F_1' \cup F_2)$ is not a perfect elimination graph by Lemma 2 and Corollary 2. Thus, if $F_2 \subseteq D(x)$ is a minimal fill-in for $G'' = (V, E \cup F_1)$, then $F_1 \cup F_2$ is a minimal fill-in for G.

Combining MIN FILL with the algorithms for computing fill-in and finding a perfect ordering, it is easy to build an algorithm to find a minimal ordering. Section 3 contains such an algorithm, implemented so that it runs in $O(n^2e')$ time, where $e' = |E \cup F|$.

We conclude this section with a lemma proved in [5] giving a necessary condition for a graph to be perfect elimination. We shall use the lemma in Section 4.

Lemma 4 (Haskins and Rose [5]). Let G be a perfect elimination graph. Then for every set X of $k > 2$ vertices there is a subset Y of k-1 vertices such that any cycle on X has a subsequence which is a cycle on Y .

3. Implementation and Complexity of the Algorithms.

Fill-in.

To calculate the fill-in produced by an ordering α using Lemma 1 we must find the vertices w reachable from each vertex v by a path whose intermediate vertices satisfy (1). To find paths which start at a fixed vertex v and have this property, we conduct a search starting from v. First we allow the search to pass through only the vertex of lowest elimination number. Then we extend the search through vertices of second lowest number, third lowest number, and so on. In this way we can find appropriate paths efficiently. A program to implement this method appears below in Algol-like notation. Given an ordered digraph $G_\alpha = (V,E,\alpha)$ with adjacency list $A(v)$ for each $v \in V$, it calculates the edges in G_α^*.

```
Algorithm FILL1 ($G_\alpha$): begin
    for i := 1 until n do begin
        v = $\alpha(i)$ ;
        for j := 1 until n do begin
            reach(j) := $\emptyset$;
            mark(j) := false;
        end;
        mark(i) := true;
        for w $\in$ A(v) do begin
            mark($\alpha^{-1}$(w)) := true;
            add w to reach ($\alpha^{-1}$(w));
            mark (v,w) as an edge of $G_\alpha^*$;
        end;
search: for j := 1 until i-1 do
            while reach(j) $\neq \emptyset$ do begin
                delete a vertex w from reach(j);
                for z $\in$ A(w) do if $\neg$ mark($\alpha^{-1}$(z)) then begin
                    mark ($\alpha^{-1}$(z)) := true;
                    if $\alpha^{-1}$(z) > j then begin
                        add z to reach′($\alpha^{-1}$(z));
                        mark (v,z) as an edge of $G_\alpha^*$;
                    end else add z to reach(j);
    end end end FILL1;
```

It is easy to show, using Lemma 1, that this algorithm correctly calculates the fill-in produced by α . The time required per execution of statement search is O(e) since each vertex v can only have mark(v) set true once and thus each edge can only be examined once. The total time for algorithm FILL1 is thus O(ne) . FILL1 requires O(e) storage space, plus space for the output.

For graphs with a small number of edges but a large fill-in, FILL1 is an efficient way to compute the fill-in. For graphs with smaller fill-in, it is more efficient to use a direct method based on the observation following Lemma 1. The only tricky part of such an algorithm is avoiding adding edges to the fill-in twice. To handle this difficulty, we use a bit vector fill(j) which records, for some current value of i , whether $(\alpha(i),\alpha(j))$ has been added as a fill-in edge.

```
algorithm FILL2 ($G_\alpha$): begin
    for j := 1 until n do fill(j) := false;
    for i := 2 until n do begin
        list := $\emptyset$;
        for $(\alpha(i),w) \in E$ do begin
            fill($\alpha^{-1}(w)$) := true;
            if $\alpha^{-1}(w) < i$ then add w to list;
        end;
        while list $\neq \emptyset$ do begin
            delete some w from list;
            for $(w,y) \in E \cup F$ with $\alpha^{-1}(w) < \alpha^{-1}(y)$ do
                if $\neg$fill($\alpha^{-1}(y)$) then begin
                    add $(\alpha(i),y)$ to F;
                    if $\alpha^{-1}(y) < i$ then add y to list;
        end end
    for $(\alpha(i),w) \in E \cup F$ do begin fill($\alpha^{-1}(w)$) := false;
end end FILL2;
```

It is immediate that this algorithm correctly computes the fill-in F produced by an ordering α . FILL requires (e') storage space, where $e' = |E \cup F|$. To estimate the time requirements of FILL2, let $d_1(v) = \{(v,w) \in E \cup F \mid \alpha^{-1}(v) > \alpha^{-1}(w)\}$ and let $d_2(v) = \{(v,w) \in E \cup F \mid \alpha^{-1}(v) < \alpha^{-1}(w)\}$. Then FILL2 requires $0\left(e' + \sum_{v \in V} d_1(v) \cdot d_2(v)\right)$ time.

Algorithm FILL2 has the advantage that its computation time is proportional to the number of arithmetic operations necessary to do numeric Gaussian elimination. Thus FILL2 can be used to precompute the fill-in for a numeric equation solver at a cost of only a constant factor in the running time. (See [16].) This is not necessarily true of FILL1. However, for sparse graphs or graphs with large fill-in algorithm FILL1 is more efficient.

Perfect Orderings.

To implement the perfect ordering algorithm so that it is efficient, we need lists to keep track of the deficiencies of each vertex. We use the following lists. For each $v \in V$, $D(v)$ is a list of triples (x,v,y) such that (x,y) is in the deficiency of v . For each $x \in V$, $L(x)$ contains one pointer to each occurrence of a triple of the form (x,v,y) in some $D(v)$, and one pointer to each occurrence of a triple of the form (y,v,x) in some $D(v)$. When a vertex x is deleted from the graph, we use $L(x)$ to update the deficiency lists of the vertices. We need two other variables: $a(v)$ is a Boolean array used to help initialize the D and L lists, and N is a list of the vertices v with $D(v) = \emptyset$.

Until $N = \emptyset$, the algorithm must carry out the following steps: find a vertex in N ; delete it and its incident edges from the graph; and update the D lists appropriately. An implementation is presented below.

```
     algorithm PERFECT(G); begin
         for v∈V do begin D(v) := L(v) := ∅; a(v) := false end;
         N := ∅;
         comment compute initial deficiencies;
   init: for u∈V do begin
             for w ∈ A(u) do a(w) := true;
             for v ∈ A(u) do
                 for w ∈ A(v) do if ¬ a(w) and (w ≠ u) then begin
                     add triple (u,v,w) to D(v);
                     add to lists L(u) and L(w) pointers to this triple (u,v,w);
                 end;
             for w ∈ A(u) do a(w) := false
         end;
         comment initialize list of deletable vertices;
         for v∈V do if D(v) := ∅ then add v to N;
         comment delete as many vertices as possible;
         i := 0;
 delete: while N ≠ ∅ do begin
             delete some vertex u from N;
             α⁻¹(u) := i := i+1;
             α(i) := u;
  update: for p ∈ L(u) do begin
                 delete from D(v) the triple (x,v,y) at which p points,
                     if this triple has not been deleted already;
                 if D(v) = ∅ then add v to N;
         end end;
         comment if i := n then G is a perfect elimination graph;
             otherwise G is not a perfect elimination graph;
     end PERFECT;
```

This program clearly implements algorithm PERFECT correctly. We analyze the running time of the program. For each edge (u,v), the program spends $O(1+ d_0(v))$ time in the initialization loop init .

The total time spent in <u>init</u> is thus $O\left(n+e+\sum_{v\in V} d_I(v)\, d_O(v)\right)$.

Since all the entries in the D and L lists are created in <u>init</u>, and each vertex is added to N at most once, the total storage requirements of PERFECT are $O\left(n+e+\sum_{v\in V} d_I(v)\, d_O(v)\right)$. The time spent executing statement <u>delete</u> is $O\left(n+e+\sum_{v\in V} d_I(v)\, d_O(v)\right)$ since the amount of time spent in <u>update</u> is proportional to the number of entries in the D and L lists, and the amount of time spent in <u>delete</u> outside of <u>update</u> is $O(n)$. Thus PERFECT requires $O\left(n+e+\sum_{v\in V} d_I(v)\, d_O(v)\right)$ time total. Since $d_O(v) \leq n$ for all v and $\sum_{v\in V} d_I(v) = e$, the running time is $O(ne)$. If $d_I(v)+d_O(v) \leq d$ for all vertices, the bound is $O(nd^2)$. If storage space is at a premium, PERFECT can be implemented to run in $O(n+e)$ space and $O(n^2e)$ time.

<u>Minimal Orderings</u>.

We can use procedure MINFILL in combination with FILL and PERFECT to compute a minimal ordering for any graph. Given a graph $G = (V,E)$ we choose any ordering α and calculate its fill-in $F = F(G_\alpha)$ using FILL. Next, we compute certain sets which MINFILL needs for its calculations. These include the deficiency $D(x)$ in G_α^* for each vertex x, the set $DF(x) = \{(u,v) \mid u \rightarrow x, x \rightarrow v \text{ in } G_\alpha^*, u \neq v, \text{ and } (u, v) \text{ is a fill-in edge}\}$ for each x, and certain lists necessary for updating the graph and the sets $D(x)$ and $DF(x)$. Then we apply MINFILL(F). MINFILL(F) is coded as a recursive procedure which, given a graph $G = (V,E)$ and a fill-in F , finds a minimal fill-in $F_0 \subset F$ and updates G to include the edges in F_0 . Once a minimal fill-in F_0 is found, we apply PERFECT to find a perfect ordering β of $G' = (V, E \cup F_0)$. This ordering is a minimal ordering of the original graph G . An outline of the algorithm appears below.

```
algorithm MINIMAL(V,E); begin
    procedure MINFILL(F); begin
delete:  while some undeleted vertex x has $D(x) = \emptyset$ and $\exists$ an edge
              (u,x) or (x,u) in F do
                 delete all edges (u,x) or (x,u) from F, updating
                    lists representing graph accordingly;
split:   if some undeleted vertex x has $(D(x) = \emptyset)$ and $(DF(x) \subset F)$
            then begin
                 delete edges in DF(x) from fill-in and add to graph
                    temporarily;
                 delete x and incident edges from graph;
                 F := F-DF(x);
    call 1:      MINFILL(F);
                 add x and incident edges to graph;
                 if $DF(x) \neq \emptyset$ then begin
                    delete edges in DF(x) from graph and add to fill-in;
                    F := DF(x);
         call 2:    MINFILL(F);
            end end else add all edges in F to $F_0$ and to graph, and
                 set F := $\emptyset$;
    end MINFILL;
    find any ordering $\alpha$ of vertices V;
    compute fill-in F = $F(G_\alpha)$ using FILL($G_\alpha$);
    compute initial deficiencies;
    $F_0$ := $\emptyset$;
    MINFILL(F);
    comment as MINFILL executes, it adds to $F_0$ and to the graph
        edges which are found to be in a minimal fill-in;
    find a perfect ordering $\beta$ of graph $G_0 = (V, E \cup F_0)$ using
        PERFECT($G_0$);
    comment $\beta$ is a minimal ordering of G;
end MINIMAL;
```

We still need to fill in the details of this algorithm and to estimate its time and space requirements. The tricky part of the implementation is representing the deficiencies so that they are easy to update. We use various lists similar to those used in PERFECT; we need extra lists here since we must keep track of the fill-in edges. For each $v \in V$, $A(v)$ and $B(v)$ are adjacency lists for v in G, and $A'(v)$ and $B'(v)$ are adjacency lists for v in $G' = (V, E \cup F)$. $M(v,w)$ is an $n \times n$ matrix such that $M(v,w) = 0$ if $(v,w) \notin E \cup F$, $M(v,w) = 1$ if $(v,w) \in F$, and $M(v,w) = 2$ if $(v,w) \in E$. For each vertex v, $D(v)$ is a list of edges (u,w) such that $u \to v$, $v \to w$ in $E \cup F$, $u \neq w$, and $u \not\to w$ in $E \cup F$. $DF(v)$ is a list of edges (u,w) such that $u \to v$, $v \to w$ in $E \cup F$, $u \neq w$, and $(u,w) \in F$. $P(u,v,w)$ is an array of pointers such that $P(u,v,w) = 0$ if $(u,w) \notin D(v) \cup DF(v)$ otherwise. For each $v \in V$, $g(v)$ = *true* if v has not been deleted from the graph; $g(v)$ = *false* if v has been deleted.

Below is an implementation of MINIMAL which uses these data structures.

```
      algorithm MINIMAL(G): begin
         procedure MINFILL(F); begin
delete:     while some vertex x has g(x) and (D(x) = ∅) and
               ((A'(x)-A(x))∪ (B'(x)-B(x)) ≠ ∅) do begin
                  comment delete edges in fill-in which are incident
                     to x from graph;
                  for (u,v) ∈ (A'(x)-A(x)) ∪ (B' (x)-B(x)) do begin
                     F := F-{(u,v)};
                     M(u,v) = 0;
                     for w∈V do if g(w) then begin
                        if P(u,v,w) ≠ 0 then begin
                           delete corresponding entry in D(v) ∪DF(v);
                           P(u,v,w) := 0;
                        end;
                        if P(w,u,v) ≠ 0 then begin
                           delete corresponding entry in D(u) ∪DF(u);
                           P(w,u,v) := 0;
                  end end end;
                  for (u,v) ∈ (A'(x)-A(x)) ∪ (B'(x)-B(x)) do
                     for w∈V do if (M(u,w) > 0) and (M(w,v) > 0)
                        then begin
                           delete entry (u,v) from DF(w) using pointer
                              P(u, w, v);
                           add entry (u,v) to D(w) and put a pointer to
                              this entry in P(u,w,v);
                           delete (u,v) from A'(u) and B'(v);
                        end;
            end delete;
split:      if some vertex x has g(x) and (D(x) = ∅) and (|DF(x)| < |F|)
               then begin
                  comment delete edges in DF(x) from fill-in and add to
                     graph temporarily;
                  comment also delete x and incident edges;
                  for (u,v) ∈A(x) ∪B(x) do M(u,v) := 0;
                  for (u,v) ∈DF(x) do M(u,v) := 2;
```

```
                for distinct $u,v,w \in V$ such that $(g(u) = g(v) = g(w) =$
                   true) and $x \in \{u,w\}$ do
                   if $P(u,v,w) \neq 0$ then begin
                      delete corresponding entry from $D(v) \cup DF(v)$;
                      $P(u,v,w) := 0$;
                   end;
                g(x) := false;
                for $(u,v) \in DF(x)$ do
                   for $w \in V$ do
                      if $(P(u,w,v) \neq 0)$ and $(w \neq x)$ and g(w) then begin
                         delete corresponding entry from DF(w);
                         $P(u,v,w) := 0$;
                      end;
                F := F-DF(x);
       call 1:  MINFILL(F);
                comment restore x and incident edges to graph;
                comment delete edges in DF(x) from graph and add
                   to fill-in;
                g(x) := true;
                for $(u,v) \in A(x) \cup B(x)$ do $M(u,v) := 2$;
                for $(u,v) \in DF(x)$ do $M(u,v) := 1$;
                for distinct $u,v,w \in V$ such that $x \in \{u,w\}$ or $(u,w) \in DF(x)$
                   and $(x \neq v))$ do
                   if $M(u,v) > 0$ and $M(v,w) > 0$ then begin
                      if $M(u,w) = 0$ then add (u,w) to D(v) and
                         put a pointer to this entry in P(u,v,w);
                      if $M(u,w) = 1$ then add (u,w) to DF(v) and put
                         a pointer to this entry in P(u,v,w);
                   end;
       call 2:  if $DF(x) \neq \emptyset$ then begin F := DF(x); MINFILL(F) end;
            end else for $(u,v) \in F$ do begin
                comment F is a minimal fill-in;
                comment add edges in F to $F_0$ and to graph;
                comment delete all edges from fill-in;
                M(u,v) := 2;
                for w∈V do if P(u,w,v) ≠ 0 then begin
                    delete corresponding entry from DF(v);
                    P(u,w,v) := 0;
                end;
                add (u,v) to F_0 and to E;
                add v to A(u);
                add u to B(v);
                F := ∅;
            end split;
        end MINFILL;
        find any ordering α of vertices V;
        compute fill-in F = F(G_α) using FILL(G_α);
        comment initialization;
        compute matrix M(v,w);
        for v∈V do D(v) := DF(v) := ∅;
        for u,v,w ∈ V do
            if (M(u,v) > 0) and (M(v,w) > 0) and (u ≠ w) then begin
                if M(u,w) = 0 then add (u,w) to D(v) and put a pointer
                    to this entry in P(u,v,w)
                else if M(u,w) = 1 then add (u,w) to DF(v) and put a
                    pointer to this entry in P(u,v,w)
                else P(u,v,w) := 0;
            end else P(u,v,w) := 0;
        F_0 := ∅;
        for v∈V do g(v) := true;
        MINFILL(F);
        find a perfect ordering β of G_0 = (V, E ∪ F_0) using PERFECT(G_0);
        comment β is a minimal ordering of G;
    end MINIMAL;
```

A few observations help in seeing that this program correctly implements MINFILL. Matrix M and Boolean array g always encode the current graph, with deleted vertices excluded. Every deleted vertex has all its incident edges in E (not in F) when it is deleted. When a vertex is deleted, the value of DF(x) is left intact, as are all pointers of the form $P(v,x,w)$. This gives us a place to save DF(x) , and makes updating the graph after call 1 easier. The graph updating throughout the program is straightforward.

It is an interesting exercise to figure out the resource requirements of the algorithm. Let e' be the number of edges in the graph G_α^* where α is the arbitrary ordering selected initially. We shall show that the total number of calls on MINFILL is $O(e')$, the maximum depth of nested calls on MINFILL is $O(n)$, and MINIMAL uses $O(n^3)$ space and $O(n^2e')$ time. We make several observations which lead to these bounds. First, the time spent in MINIMAL outside of MINFILL is clearly $O(n^3)$. Also, the storage required, not counting storage for the procedure parameter F in MINFILL, is clearly $O(n^3)$.

Now consider the nested recursive calls on procedure MINFILL. Either a procedure call MINFILL(F) is a bottom-level call on MINFIL or it leads to two nested calls MINFILL(F') and MINFILL(F"), where $F' = F\text{-}DF(x) \neq \emptyset$ and $F'' = DF(x) \neq \emptyset$. Thus the nested calls on MINFILL may be represented as a binary tree. The topmost vertex of the tree corresponds to the outer call MINFILL($F(G_\alpha)$) . Each leaf of the tree corresponds to an innermost call on MINFILL. If $F_1, F_2, \ldots, F_k$ are the values of the parameters in these innermost calls, then $F_i \cap F_j = \emptyset$ and $F_i \subseteq F(G_\alpha)$ for all i, j . Since $|F(G_\alpha)| \leq e'$, $k \leq e'$, and the total number of calls on MINFILL is $O(e')$.

Consider the depth of nested calls on MINFILL. Suppose the call MINFILL(F) leads to a call MINFILL(F') with $F' = F\text{-}DF(x)$ by statement call 1 with $F' = F\text{-}DF(x)$ and to a call MINFILL(F″) with $F'' = DF(x)$ by statement call 2 . Suppose we name x the splitting vertex for the call MINFILL(F). Vertex x is absent from all graphs considered during the execution of MINFILL(F'). The fill-in is always contained within DF(x) for all graphs G' considered during the execution of MINFILL(F″). Thus x cannot be a splitting vertex for the calls MINFILL(F'), MINFILL(F″), or any calls nested within them. It follows that each nested call on MINFILL has a different splitting vertex (unless it is an innermost call with no splitting vertex) and the maximum depth of nested calls on MINFILL is $O(n)$.

Since parameter storage space for one call on MINFILL is $O(n^2)$, the total parameter storage requirements for nested calls on MINFILL are $O(n^3)$, and the total storage required by MINIMAL is $O(n^3)$.

Consider the time used during one call on MINFILL, not counting time spent in nested calls. Time spent testing the condition in while loop delete is $O(n)$ if we keep track of the sizes of all $A(x)$, $A'(x)$, $B(x)$, $B'(x)$ as the graph changes. Time spent executing while loop delete is $O(n)$ per edge deleted from the fill-in. Once an edge is deleted in step delete , it never reappears. Thus the total time spent in delete over all calls on MINFILL is $O(ne')$ to test the condition plus $O(ne')$ to delete edges from F and update the graph.

Time spent testing the if condition in statement split is $O(n)$ if we keep track of the size of each DF(x) and the size of F as the graph changes. Time spent executing the then branch of split is $O(n^2)$ to update the graph by deleting and later adding x , $O(n)$ time per edge in DF(x) to update the graph, and $O(e')$ time generating each nested call on MINFILL (since the sets F-DF(x) and DF(x) together have at most $O(e')$ elements). Thus the total time spent in the *then* branch of split over all calls on MINFILL is $O(n^2e')$ plus $O(n)$ time per edge in DF(x) .

The time spent in the *else* branch of split is $O(n)$ per edge added to F . An edge added to F_0 is added to the graph and never deleted. Thus the total time spent in the *else* branch of split over all calls on MINFILL is $O(ne')$.

In summary the total time required by MINFILL is $O(n^2e')$ plus $O(n)$ time for each edge in each set DF(x) where x is a splitting vertex. If x is the splitting vertex for the call MINFILL(F), each edge in DF(x) must be in F . The two nested calls MINFILL(F') with F' = F-DF(x) and MINFILL(F") with F" = DF(x) produced by MINFILL(F) have parameters which are disjoint sets. Thus each edge can only occur in $O(n)$ parameters, since the maximum depth of nested calls on MINFILL is $O(n)$. Thus the $O(n)$ time per edge in each set DF(x) , when summed over all splitting vertices, is $O(n^2e')$. The total time required by MINFILL is thus $O(n^2e')$, and MINIMAL requires $O(n^2e')$ time, $O(n^3)$ space, $O(e')$ calls on MINFILL, and an $O(n)$ maximum depth of nested calls on MINFILL. If storage space is costly, we can implement MINIMAL to run in the same time using only $O(ne')$ storage space, or to run in $O((ne')^2)$ time using only $O(e')$ storage space.

4. Computational Relationships with Other Problems.

In this section we show that algorithms FILL and PERFECT cannot be improved too much without finding a new and better transitivity-testing algorithm, and that the minimum fill-in problem is very hard. In particular, we show that (1) any algorithm which computes an ordering's fill-in can be used to compute the transitive closure of a graph; (2) any algorithm which tests whether a graph has a perfect elimination order can be used to test a graph for transitivity; and (3) any algorithm which determines whether a graph has a fill-in of some size e' or less can be used to test a propositional formula for satisfiability.

Fill-in, Perfect Orderings, and Transitivity.

Given any acyclic graph $G = (V,E)$, consider the graph $G_2 = (V_2,E_2)$, where $V_2 = \{v(i) \mid v \in V,\ i \in \{1,2\}\}$ and $E_2 = \{(v(2),w(2)) \mid (v,w) \in E\} \cup \{(v(1),v(2)) \mid v \in V\}$. Let α be an ordering on V such that $(v,w) \in E$ implies $\alpha^{-1}(v) < \alpha^{-1}(w)$. (Such an ordering is called a topological sorting of G [7].) Let α_2 be the ordering on V_2 defined by $\alpha_2^{-1}(v(i)) = n(2-i) + \alpha^{-1}(v)$.

Applying Lemma 1, it is clear that the fill-in $F((G_2)_{\alpha_2})$ is defined by

$$F((G_2)_{\alpha_2}) = \{(v(1),w(2)) \mid \exists \text{ a path from } v \text{ to } w \text{ in } G\}.$$

Given G, it is easy to construct G_2 in $O(n+e)$ time. Thus we have

Theorem 2. Given an acyclic graph G, we can construct in $O(n+e)$ time a graph G_2 with $2n$ vertices and $n+e$ edges, and an ordering α_2, such that the edges in $F((G_2)_{\alpha_2})$ correspond one-to-one with the edges in the transitive closure of G.

Thus any algorithm for computing fill-in can be converted into an algorithm (with the same time and space requirements, to within a constant factor) for computing the transitive closure of an acyclic graph. (The requirement that the graph be acyclic is not a significant restriction; see[3,8].) Thus the fill-in problem is at least as hard as the transitive closure problem.

Given any acyclic graph $G = (V,E)$, consider the graph $G_3 = (V_3,E_3)$, where $V_3 = \{v(i) \mid v \in V,\ i \in \{1,2,3\}\} \cup \{s\}$, and $E_3 = \{(u(i),v(j)) \mid (u,v) \in E,\ i < j\} \cup \{(s,v(1)) \mid v \in V\}$ $\cup \{(v(3),s) \mid v \in V\} \cup \{(s,v(3)) \mid v \in V\}$. Given G, it is easy to construct G_3 in $O(n+e)$ time.

Lemma 5. G is transitive if and only if G_3 is perfect elimination.

Proof. Suppose G is transitive. Then for all distinct u, v, w, $u \to v$ and $v \to w$ in G imply $u \to w$. Let α be any ordering of the vertices of G_3 such that

$$\alpha^{-1}(v(2)) \in \{1,2,\ldots,n\} \text{ for } v \in V,$$
$$\alpha^{-1}(v(1)) \in \{n+1,\ldots,2n\} \text{ for } v \in V,$$
$$\alpha^{-1}(v(3)) \in \{2n+1,\ldots,3n\} \text{ for } v \in V, \text{ and}$$
$$\alpha^{-1}(s) = 3n+1 .$$

If G is transitive, elimination of the vertices $\{v(2)\}$ causes no fill-in, since $(u(1),v(2))$, $(v(2),w(3)) \in E_3$ imply $(u(1),w(3)) \in E_3$.

Then elimination of the vertices $\{v(1)\}$ causes no fill-in, since $(s,v(1))$, $(v(1),w(3)) \in E_3$ imply $(s,w(3)) \in E_3$. Then elimination of the vertices $\{v(3)\}$ causes no fill-in, since s is the only remaining vertex adjacent to any $v(3)$. Thus, if G is transitive, G_3 is perfect elimination.

For the converse, suppose $u \to v$ and $v \to w$ in G . Consider the cycle $\mu = [u(1),v(2),w(3),s]$ in G_3 . It follows from Lemma 4 that if G_3 is perfect elimination, there must be an edge in G_3 joining $u(1)$ and $w(3)$ or joining $v(2)$ and s . The only such edge possible is $(u(1),w(3))$. Thus $u \to w$ in G , and G is transitive. $\square$

Summarizing we have

<u>Theorem 3</u>. Given an acyclic digraph G , we can construct in $O(n+e)$ time a graph G_3 with $3n+1$ vertices and $3e+3n$ edges such that G is transitive if and only if G_3 is perfect elimination.

Thus any algorithm for testing whether a graph is perfect elimination can be used to test a graph for transitivity, at a cost of only a constant factor in the running time. Munro [8] has shown that the transitive closure of a graph can be computed in $O(n^{2.81})$ time using Strassen's fast matrix multiplication method [13]. Various problems, including transitive reduction [1] and Boolean matrix multiplication [3] are known to be computationally equivalent to transitive closure. There may be a way to solve the fill-in and perfect ordering problems in $O(n^{2.81})$ time, by reducing them to transitive closure problems, but any improvement beyond $O(n^{2.81})$ would improve the best bound known for Boolean matrix multiplication.

Minimum Orderings and the Satisfiability Problem.

Now we show that the problem of finding a minimum elimination ordering for a graph is NP-complete. For this purpose, we formulate the problem in the following way: given any graph $G = (V,E)$ and a size e', does G have an ordering which produces a fill-in of e' edges or less? To show that this problem is NP-complete, we must demonstrate that (1) there is a non-deterministic polynomial-time algorithm for solving the problem; and (2) given any instance P of a known NP-complete problem $\mathcal{P}$, there is a polynomial-time transformation which converts P into a graph G and a size e', such that the answer to P is "yes" if and only if G has an ordering with a fill-in of e' edges or less. (For those not familiar with the notion of an NP-complete problem, references [2,6] provide an extensive discussion.)

Part (1) is easy: to discover whether G has an ordering which produces a fill-in of size e' or less, we guess an ordering and calculate its fill-in using FILL. Guessing an ordering and checking its fill-in clearly require polynomial time. This algorithm is non-deterministic; it can guess all possible orderings. If one of them produces fill-in e' or less, the algorithm answers "yes".

Part (2) is quite a bit harder. For $\mathcal{P}$ we choose the satisfiability problem of propositional calculus, which is known to be NP-complete [2]. Let P be any propositional formula with m variables. We may assume that P is in conjunctive normal form with three literals per clause [2]. Let P have k clauses. We shall construct a digraph $G(P)$ and a size $e'(P)$ such that $G(P)$ has an ordering with fill-in of size $e'(P)$ or less if and only if there is a truth assignment to the variables which makes P true.

We use letters x, y, z to denote variables and p, q, r to denote literals (variables or their negations). We use $\bar{x}$ to denote the negation of x; we regard $\bar{\bar{x}}$ as another notation for x. $G(P)$ will contain some individual vertices and some <u>cliques</u> of various sizes. If X and Y are cliques, we use a single "edge" (X,Y) as shorthand to denote all possible edges from vertices in X to vertices in Y. Similarly, the "edge" (v, X) will denote all possible edges from vertex v to vertices in clique X.

The basic building block in the construction is the "ground" configuration illustrated in Figure 1, consisting of a vertex v and three cliques X_1, X_2 and X_3. Observe that vertex v must be eliminated first and vertices in X_1 second in any perfect ordering of this grahh. (Any other ordering produces fill-in at least $\min\{|X_2|, |X_1|\cdot|X_1|\cdot|X_3|, |X_1|\cdot|X_2|\}$.) This construction also works if all the edges are reversed.

Without loss of generality we may assume that for each variable x, x and $\bar{x}$ occur the same number of times in the clauses of P. (Otherwise, we can add a suitable number of dummy clauses of the form $x\bar{x}\bar{x}$ or $xx\bar{x}$.) We may also assume that no occurrence of any variable x follows an occurrence of $\bar{x}$ in a clause.

For each variable x in P, $G(P)$ contains two vertices, one corresponding to x and one to $\bar{x}$. We shall use x and $\bar{x}$ to denote these vertices. For each clause $(p \vee p \vee r)$ in P, $G(P)$ contains three vertices and three cliques, denoted by $pqr(i)$ for $i = 1,2,3$, and $X_{31}(pqr(i))$ for $i = 1,2,3$. Thus each literal occurring in a clause has one vertex and one clique corresponding to it (i.e., pqr(2) , X_{31}(pqr(2)) correspond to the literal q in ($p \vee q \vee r$)). G(P) contains nineteen other cliques, denoted by $X_{20}, X_{50}, X_{32}, X_{33}$, and X_{ij} for $i = 1,2,4,5,6$ and $j = 1,2,3$. The cliques are arranged into six grounds.

Table 1 gives all the adjacencies in G(P) and the sizes of all the cliques appear in Table 2. The sizes of the cliques are chosen to make the calculations simple, not to be as small as possible. Figure 2 illustrates G(P) for $P = (\bar{x} \vee \bar{y} \vee z) \wedge (x \vee y \vee \bar{z})$.

It is clear that the size of G(P) is polynomial in the length of P , and that G(P) can be constructed in polynomial time given P . G(P) is designed so that producing a small fill-in requires that vertices corresponding to the false literals for some truth assignment of P must be eliminated before any vertices in cliques X_{11}, X_{20}, X_{61} , X_{50} , or X_{41} . If some truth assignment satisfies P , there is a corresponding elimination order which produces a small fill-in. If no truth assignment satisfies P , there is no elimination order with small fill-in. The next result formalizes this idea, and finishes the proof that the minimum fill-in problem is NP-complete.

Theorem 4. G(P) has an ordering with fill-in $\left(m + \frac{3k}{2} + 1\right)b-1$ or less if and only if P is satisfiable.

Proof. First we show that if F is not satisfiable, every ordering of G(F) produces fill-in $\left(m + \frac{3k}{2} + 1\right)b$ or greater. Suppose F is not satisfiable. Let α be any elimination ordering. We must consider several cases.

(i) Some vertex pqr(i) representing a literal is eliminated after some vertex in $X_{31}(pqr(i))$. Then depending on whether a vertex in $X_{31}(pqr(i))$, X_{32} , or X_{33} is eliminated first, the fill-in is at least c^2 , c^4 , or c^4 . Examining Table 2, we see that $b \leq (2m+7k+1)c$ and $\left(m + \frac{3k}{2} + 1\right)b \leq \left(m + \frac{3k}{2} + 1\right)(2m+7k+1)c \leq c^2$.

(ii) Case (i) does not hold and some vertex pqr(i) representing a literal is eliminated before the corresponding variable vertex (p if $i = 1$, q if $i = 2$, r if $i = 3$). Then the fill-in is at least $c^2 \geq \left(m + \frac{3k}{2} + 1\right)b$.

(iii) Cases (i) and (ii) do not hold and some vertex v not a variable vertex, not a literal vertex, not in X_{32} or X_{33} , and not in any $X_{31}(pqr(i))$ is eliminated before any vertex in X_{11} is eliminated. The first such vertex v eliminated causes a fill-in of at least $c^2 \geq \left(m + \frac{3k}{2} + 1\right)b$.

(iv) Cases (i), (ii), and (iii) do not hold and at most $m + \frac{3k}{2} - 1$ vertices among the x , $\bar{x}$, and pqr(i) are eliminated before any vertex in X_{11} . Then the first elimination of a vertex in X_{11} causes a fill-in of at least $\left(m + \frac{3k}{2} + 1\right)b$.

(v) Cases (i), (ii), (iii), and (iv) do not hold. Then before any vertex in $X_{61} \cup X_{20} \cup X_{50} \cup X_{41}$ is eliminated, either two vertices x and $\bar{x}$ or three vertices pqr(1) , pqr(2) , and pqr(3) must have been eliminated (since F is not satisfiable). Either case produces a fill-in of at least $c^2 \geq \left(m + \frac{3k}{2} + 1\right)b$.

Now suppose that P is satisfiable. Choose a truth assignment for the variables which satisfies P. Consider the elimination order given in Table 3. A careful examination of the adjacencies given in Table 1 reveals that the fill-in listed in Table 3 is correct. The size of the fill-in is bounded by $(m+4k)c+\left(m+\frac{3k}{2}\right)b+m+3k \leq \left(m+\frac{3k}{2}+1\right)b-1$. □

The graph G(P) used in this construction is not strongly connected, but it can be made strongly connected by adding a new vertex s and edges (s,x) and (x,s) for all vertices x in the original graph. Such an addition does not affect the minimum fill-in.

It seems likely that there is a similar construction which shows that the minimum fill-in problem for undirected graphs is NP-complete, but such a construction is still undiscovered. (See [12] for a discussion of elimination orderings on undirected graphs.)

5. Remarks.

This paper has given an $O(ne)$ algorithm for computing the fill-in of any elimination ordering on a graph, an $O(ne)$ algorithm for finding a perfect ordering, and an $O(n^2e')$ algorithm for finding a minimal ordering. There may be a way to solve the fill-in and perfect ordering problems in $O(n^{2.81})$ time, but any improvement beyond this would improve upon the best algorithm known for transitive closure. The minimal ordering algorithm may be improvable to $O(n^2e)$ or even to $O(n^3)$.

The construction in Section 4 shows that the problem of finding a minimum ordering is NP-complete. Since this probably implies that exponential time is required to find a minimum ordering, any practical method for getting a small fill-in must be based on a heuristic. Two heuristics, the minimum degree heuristic and the minimum fill-in heuristic [11], seem to work well in practice, but there are no theoretical results to support this assertion. The theoretical study of such heuristics seems to be a good area for future research. See [12,13,16] for further comments regarding related issues.

References

[1] A. Aho, M. Garey, and J. Ullman, "The transitive reduction of a directed graph," SIAM J. Comput., Vol. 1 (1972), 131-137.

[2] S. Cook, "The complexity of theorem-proving procedures," Proceedings Third Annual ACM Symposium on Theory of Computing (1971), 151-158.

[3] M. Fischer and A. Meyer, "Boolean matrix multiplication and transitive closure," Twelfth Annual Symposium on Switching and Automata Theory (1971), 129-131.

[4] G. E. Forsythe and C. B. Moler, Computer Solution of Linear Algebraic Equations, Prentice-Hall, Englewood Cliffs, N. J.(1967).

[5] L. Haskins and D. Rose, "Toward characterization of perfect elimination digraphs," SIAM J. Comput., Vol. 2 (1973), 217-224.

[6] R. Karp, "Reducibility among combinatorial problems, " Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, N. Y. (1972), 85-104.

[7] D. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., (1968), 258-265.

[8] I. Munro, "Efficient determination of the transitive closure of a directed graph," Info. Proc. Letters, Vol. 1 (1971), 56-58.

[9] T. Ohtsuki, "A fast algorithm for finding an optimal ordering in the vertex elimination on a graph," SIAM J. Comput., to appear.

[10] D. J. Rose, "Triangulated graphs and the elimination process," Journal of Mathematical Analysis and Applications, Vol. 32 (1970), 597-609.

[11] D. J. Rose, "A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations," Graph Theory and Computing, R. Read, ed., Academic Press, N. Y., (1973), 183-217.

[12] D. Rose, R. Tarjan, and G. Lueker, "Algorithmic aspects of vertex elimination on graphs," SIAM J. Comput., to appear.

[13] D. Rose and R. Tarjan, "Algorithmic aspects of vertex elimination," Proceedings Seventh Annual ACM Symposium on Theory of Computing (1975), 245-254.

[14] V. Strassen, "Gaussian elimination is not optimal," Numer. Math., Vol. 13, (1969), 354-356.

[15] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput., Vol. 1, (1972), 146-160.

[16] R. Tarjan, "Graph theory and Gaussian elimination," Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, to appear.

Vertex or Clique	Size	Adjacencies In	Adjacencies Out
x	1	X_{20}, X_{22}	$\bar{x}, X_{11}, X_{41}$, all pqr(i) containing x or $\bar{x}$, all X_{31}(pqr(i)) with pqr containing x or $\bar{x}$ exceptr(1)), X_{31}(pxr(2)), X_{31}(pqx(3)).
x	1	x, X_{22}	X_{11}, X_{61}, X_{41}, $\bar{x}$qr (i) for i = 1,2,3, p$\bar{x}$r(i) for i = 2,3, pq$\bar{x}$(3), X_{31}(pqr(i)) with pqr containing x or x <u>except</u> X_{31}(xqr(1)), X_{31}(xqr(1)), X_{31}(pxr(2)), X_{31}(p$\bar{x}$r(2)), X_{31}(pqx(3)), X_{31}(pq$\bar{x}$(3)).
pqr(1)	1	$X_{50}, X_{52}, X_{20}, X_{22}$, $p, \bar{p}, q, \bar{q}, r, \bar{r}$ corresponding to variables, $p, \bar{p}$ corresponding to negation of variables	pqr(2), X_{31}(pqr(1)) ,X_{11}
pqr(2)	1	pqr(1) ,X_{52}, X_{20}, X_{22}, $p, \bar{p}, q, \bar{q}, r, \bar{r}$ corresponding to variables, $p, \bar{p}, q, \bar{q}$ corresponding to negation of variables	pqr(3) ,X_{31}(pqr(2)) ,X_{11}

<u>Table 1</u>: Clique sizes and adjacencies in G(P)
Values of constants appear in Table 2.

Vertex or Clique	Size	Adjacencies In	Adjacencies Out
pqr(3)	1	pqr(2),X_{52},X_{20},X_{22}, p,p, q,$\bar{q}$,r,$\bar{r}$	X_{41}, X_{31}(pqr(3)), X_{11}
X_{41}	c	pqr(3), X_{52},X_{20},X_{22},X_{21}, X_{43},x,$\bar{x}$	X_{42}
X_{42}	c^2	X_{41},X_{43}	X_{43}
X_{43}	c^2	X_{42}	X_{41},X_{42}
X_{50}	c	X_{51}	pqr(1), X_{11},X_{12},X_{31}(pqr(i))
X_{51}	1	X_{52}	X_{53},X_{50},X_{31}(pqr(i))
X_{52}	c^2	X_{53}	X_{51},X_{53},pqr(i),X_{41},X_{11}, X_{12}, X_{31}(pqr(i))
X_{53}	c^2	X_{51},X_{52}	X_{52}
X_{31}(pqr(1))	c^2	pqr(1),q,$\bar{q}$,r,$\bar{r}$,X_{50},X_{51}, X_{52},X_{20},X_{21},X_{22},X_{33},p if p is the negation of a variable	X_{32}
X_{31}(pqr(2))	c^2	pqr(2),p,$\bar{p}$,r,$\bar{r}$,X_{50},X_{51}, X_{52},X_{20},X_{21},X_{22},X_{33},q if q is the negation of a variable	X_{32}

<u>Table 1</u> (cont.)

Vertex or Clique	Size	Adjacencies In	Adjacencies Out
$X_{31}(pqr(3))$	c^2	$pqr(3), p, \bar{p}, q, \bar{q}, X_{50}, X_{51}, X_{52}, X_{20}, X_{21}, X_{22}, X_{33}, \bar{r}$ if r is the negation of a variable	X_{32}
X_{32}	c^2	$X_{31}(pqr(i)), X_{33}$	X_{33}
X_{33}	1	X_{32}	$X_{31}(pqr(i)), X_{32}$
X_{11}	c^2	$x, x, X_{20}, X_{22}, X_{50}, X_{52}, pqr(i), X_{13}$	X_{12}
X_{12}	b	$X_{20}, X_{22}, X_{50}, X_{52}, X_{11}, X_{13}$	X_{13}
X_{13}	1	X_{12}	X_{11}, X_{12}
X_{61}	c	$\bar{x}, X_{63}, X_{22}$	X_{62}
X_{62}	c^2	X_{61}, X_{63}	X_{63}
X_{63}	c^2	X_{62}	X_{61}, X_{62}
X_{20}	c	X_{21}	$x, X_{11}, X_{12}, X_{41}, X_{31}(pqr(i))$, pqr (i)
X_{21}	1	X_{22}	$X_{23}, X_{20}, X_{41}, X_{31}(pqr(i))$
X_{22}	c^2	X_{23}	$X_{21}, X_{23}, X_{11}, X_{12}, X_{61}, x, \bar{x}, X_{41}, X_{31}(pqr(i)), pqr(i)$
X_{23}	c^2	X_{21}, X_{22}	X_{22}

Table 1 (cont.)

$$c = (m + \frac{3k}{2} + 1)(2m + 7k + 1) + 1$$

$$b = (m + 4k)c + m + 3k + 1$$

$$|X_{13}| = |X_{21}| = |X_{33}| = |X_{51}| = 1$$

$$|X_{12}| = b$$

$$|X_{20}| = |X_{50}| = |X_{41}| = |X_{61}| = c$$

All other cliques have size c^2.

Table 2: Constants for clique sizes in G(P).

Elimination Order	Fill-in	Size
Vertices corresponding to false variables and negations of true variables	$X_{20} \to x$, $x \to X_{61}$	mc
Vertices pqr(i) corresponding to false literals, in order pqr(1) ,pqr(2) ,pqr(3)	$X_{50} \to$ pqr(2), $X_{50} \to$ pqr(3) pqr (1) $\to X_{41}$, pqr(2) $\to X_{41}$	$\leq 4kc$
X_{11}	$x \to X_{12}$, $x \to X_{12}$, pqr(i) $\to X_{12}$	$(m + \frac{3k}{2})b$
X_{13}		0
X_{12}	-	0
X_{20}	$X_{21} \to x, X_{21} \to x, X_{21} \to$ pqr(i)	$m + \frac{3k}{2}$
X_{21}		0
X_{23}	-	0
X_{22}	-	0
Rest of vertices corresponding to variables	-	0
X_{50}	$X_{51} \to$ pqr(i)	$\frac{3k}{2}$
X_{51}	-	0

Table 3: Elimination order for G(P) if P is satisfiable.

Elimination Order	Fill-in	Size
X_{53}		0
X_{52}		0
rest of vertices pqr(i) in order pqr(1) , pqr(2) , pqr(3)		0
X_{31}(pqr(i))		0
X_{33}		0
X_{32}		0
X_{41}		0
X_{43}	-	0
X_{42}	-	0
X_{61}	-	0
X_{63}	-	0
X_{62}		0

<u>Table 3</u> (cont.)

Total fill-in is $\leq (m + 4k)c + (m + \frac{3k}{2})b + m + 3k$

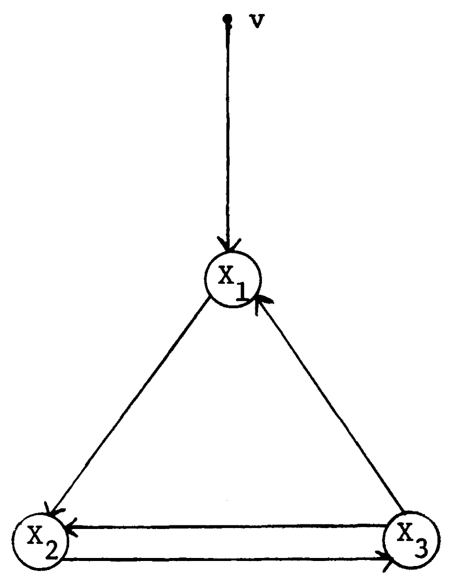

Figure 1: A "ground" configuration used as building block in NP-completeness construction. Point v is a single vertex; circles denote cliques.

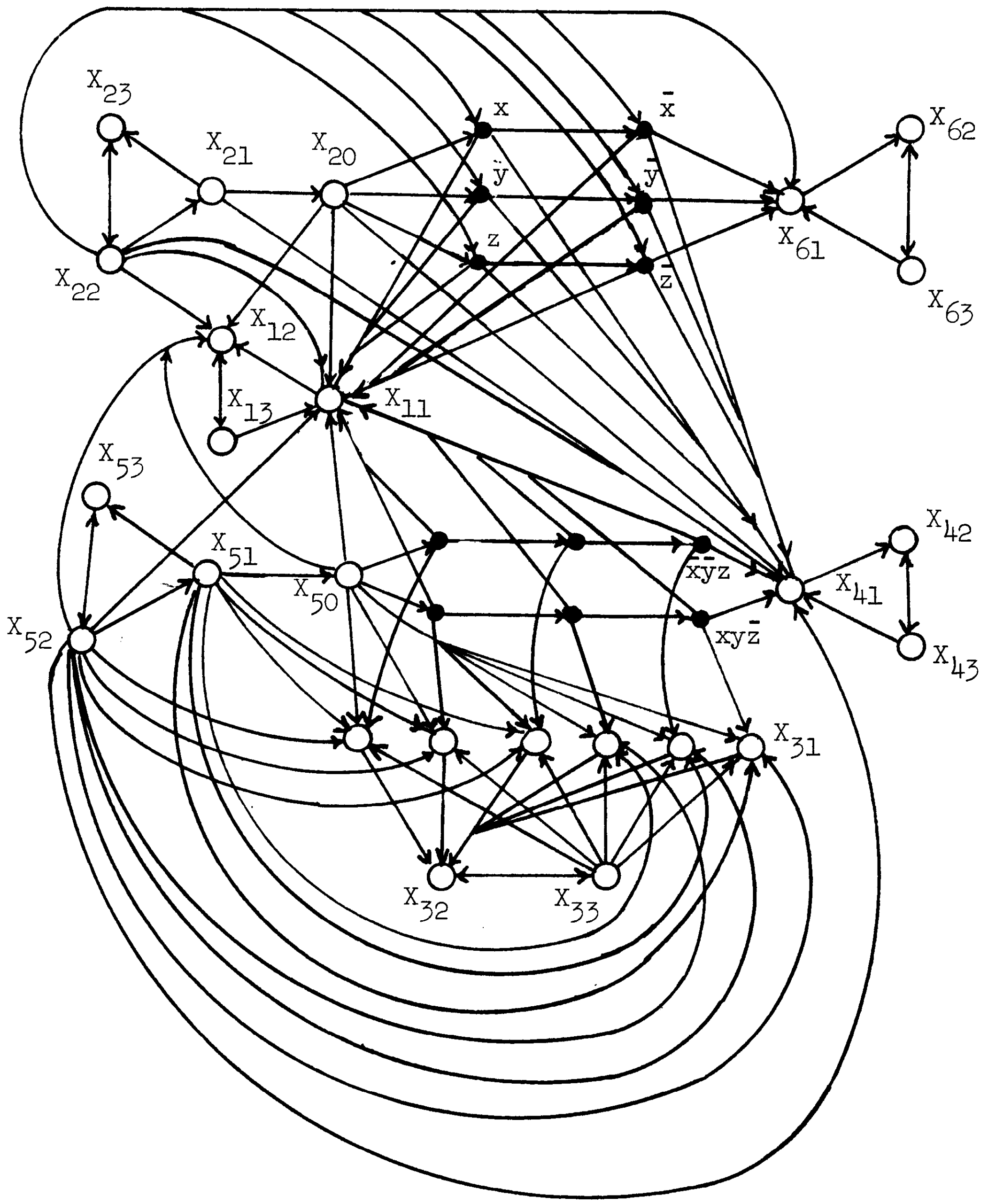

Figure 2(a). Edges in G(P) for $P = (\bar{x} \vee \bar{y} \vee z) \wedge (x \vee y \vee \bar{z})$.

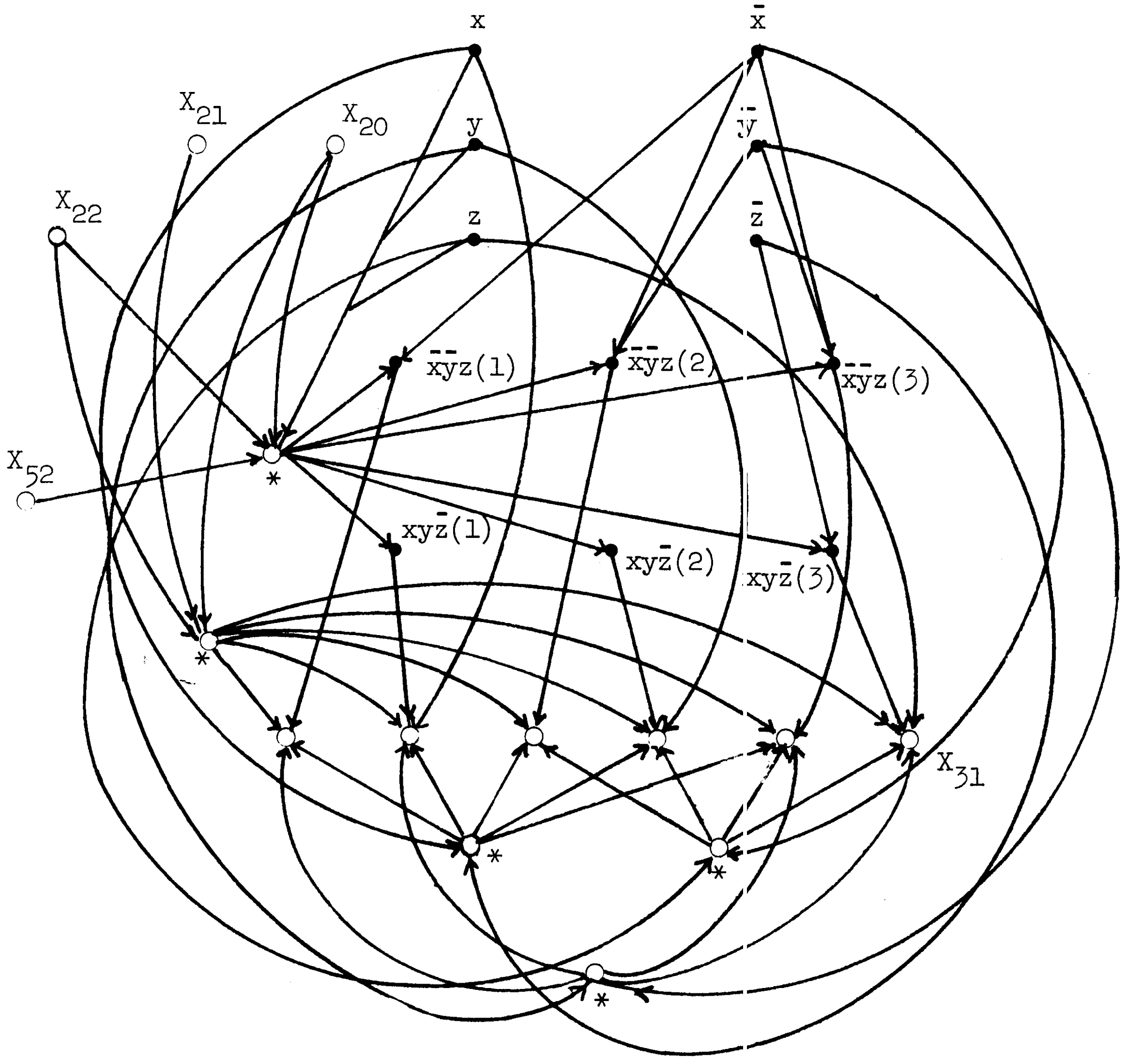

Figure 2(b). Elimination of starred vertices gives the rest of the edges in G(P) .