RECOGNITION AND ORDERING ALGORITHMS CONCERNING GLOBAL INHERITANCE IN LU FACTORIZATIONS

by

TERENCE ARTHUR SLATER
B.Sc., University of Victoria, 1972
B.Sc., University of Victoria, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the Department
of
Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE June 13, 1988

We accept this thesis as conforming
to the required standard

[redacted]
Dr. D. Dale Olesky

[redacted]
Dr. John A. Ellis

[redacted]
Dr. Pauline van den Driessche

[redacted]
Dr. P. Hell

Supervisor Dr D Dale Olesky

Abstract

The concept of forward lower restricted orderings of digraphs arises naturally from the study of inherited entries in LU factorizations of matrices If a matrix A has a unique left unit LU factorization $A = LU$ then an entry $u_{i,j}$ of U with $j \geq i$ is said to be inherited from A if $u_{i,j} = a_{i,j}$ for reasons related to the combinatorial structure of A and not because of the specific values of the nonzero matrix elements It can be proved that the entire strict upper triangular part of A is inherited if and only if the digraph D of the matrix has the property that for all i and j with $i < j$ there is no path of length greater than one from i to j in D such that all intermediate nodes of the path are in $\{1, 2, \ldots, i-1\}$ Any digraph on the node set $\{1, 2, \ldots, n\}$ having this property is said to be forward lower restricted (FLR) ordered

We state and prove theorems which characterize the FLR ordered trees and the maximal FLR ordered digraphs Polynomial time algorithms for deciding if a given ordered digraph is FLR ordered and for finding an FLR ordering of an arbitrary digraph are presented The latter algorithm also detects that a digraph is not FLR orderable In addition, a more efficient algorithm for finding FLR orderings of digraphs containing a spanning tree is given Some results concerning an attempt to characterize FLR orderable digraphs in terms of their cycle structure are included

All of the results of this thesis also extend to inheritance of entries in the matrix L of the LU factorization and to inheritance in UL factorizations

Examiners

[redacted]

Dr D Dale Olesky

[redacted]

Dr John A Ellis

[redacted]

Dr Pauline van den Driessche

[redacted]

Dr P Hell

Table of Contents

CHAPTER 1

Introduction

A problem frequently encountered in numerical computing is to find a solution to a system of n linear equations in n variables. This problem has been, and still is, the subject of considerable research interest.

There is a well known algorithm called Gaussian elimination which is frequently employed in the solution of systems of linear equations. It is used to compute a triangular factorization of the coefficient matrix of the system, if possible, from which a solution to the system is easily obtained. In its simplest form, Gaussian elimination may terminate prematurely to avoid dividing by zero, even though a solution to the system does exist. This difficulty is usually overcome by using a pivoting strategy to permute the rows and/or the columns. Pivoting is also used to obtain an algorithm which is numerically stable.

Another problem arises when very large systems are to be solved. Gaussian elimination requires $O(n^3)$ time to factor an $n \times n$ matrix so as n becomes large, the solution time rapidly becomes prohibitively large.

The goal of much recent research in this area has been to develop efficient techniques to solve very large sparse systems (i.e. having few nonzero elements in the coefficient matrix). To benefit from sparsity, the coefficient matrix is subjected to a preliminary analysis before the matrix is factored. This must be done with care to ensure that the time spent analyzing the matrix does not normally exceed the time saved in factoring it.

During the analysis phase a good pivoting order is selected and data structures are set up for the input matrix and other required storage. Historically the pivoting order selected has been influenced by considerations of stability, storage and computational efficiency. Stability is concerned with ensuring that small pivots do not occur during factorization. Storage and computational efficiency are frequently related and usually concerned with ensuring that relatively few zero elements are filled in with nonzero elements when the matrix is overwritten with its factorization.

In [15] the goal of preserving zero entries is generalized. Consideration is given there to the problem of determining situations in which an entry in the coefficient matrix remains unchanged by the factorization. Conditions on the digraph of the matrix are found which

characterize the circumstances under which individual entries (local inheritance) or the entire [strict] upper triangular part (global inheritance) are preserved. Global inheritance for the strict upper triangular part is characterized by the digraph of the matrix being forward lower restricted ordered.

This thesis is primarily concerned with developing algorithms to solve two problems concerning forward lower restricted ordered digraphs. The first problem is to recognize when a digraph is forward lower restricted ordered. The second problem is to find a renumbering, if it exists, for the nodes of a digraph such that the resulting ordered digraph is forward lower restricted ordered.

A secondary concern of this thesis is to obtain characterizations of digraphs having special properties that are of theoretical interest in the study of inheritance. The first characterization obtained is for invariantly ordered trees. The second characterization obtained is for forward lower restricted ordered digraphs which are maximal with respect to their edge sets. The third characterization studied is for strongly connected forward lower restricted orderable digraphs in terms of their cycle structure.

Chapter 2 provides the required background material concerning digraphs and matrices. Although most of this material is well known, it is included for completeness and precision since the terminology sometimes varies in the literature.

Chapter 3 introduces the concept of inheritance in triangular factorizations. The terminology is established and the principal theorems concerning the subject are stated. The position of inheritance with respect to some well known sparse matrix concepts is then discussed.

Chapter 4 contains the principal results of this thesis. Three algorithms are presented dealing with the recognition problem and the ordering problem. For each algorithm a proof of correctness and estimate of complexity are given, followed by an illustrative example where useful. Characterizations of invariantly ordered trees and maximal forward lower restricted ordered digraphs are also obtained in this chapter. Some work on the problem of characterizing strongly connected forward lower restricted orderable digraphs in terms of their cycle structure is also presented.

In chapter 5 the work in chapter 4 is extended to a larger class of problems. Extensions of the principal results are made to orderings related to forward lower restricted orderings. Also considered are less restrictive conditions on the input digraphs for some of the

algorithms.

Chapter 6 contains a brief summary of the work in this thesis. The problems that have been solved are discussed with attention being paid to restrictions on the applicability of the solutions. Problems that have been considered but not solved are briefly discussed. Finally some problems not considered for inclusion in this thesis are mentioned as suggestions for further work.

CHAPTER 2

Background material

This chapter contains the background material required by subsequent chapters. The terminology used is standard whenever possible. There are some terms, however, which are defined differently in various references. The most convenient definition is then used. There are also some terms for concepts which do not seem to be used in the literature. Appropriate names are given to these concepts.

The background material for digraphs is presented in section 2.1. Besides definitions and terminology, some basic algorithms and useful lemmas are also stated. In section 2.2, the background material for matrices, Gaussian elimination, and LU factorization is presented. The relationship between matrices and digraphs is also discussed. The format that is used to specify algorithms in this thesis is described in section 2.3.

2.1 Digraphs, graphs and orderings

The background material for digraphs, graphs and orderings is presented in this section. The fundamental definitions for digraphs, graphs and orderings are stated in section 2.1.1. Section 2.1.2 is concerned with data structures for representing digraphs in an algorithm. In section 2.1.3 some basic algorithms are described and in section 2.1.4 some lemmas that are useful in other chapters are stated and proved.

2.1.1 Fundamental definitions

2.1.1.1 Digraphs

A *digraph* (*directed graph*) is an ordered pair $G = (V, E)$, where V is a finite set and E is a set of ordered pairs of elements of V. The elements in V are referred to as *nodes* and the elements of E as *edges*. The notation $u \rightarrow v$ will usually be used for the edge (u, v) and the notation $u \rightarrow v \in E$ indicates that the edge $u \rightarrow v$ is in E, not that v is in E. Some references use the term *arcs* instead of edges when referring to digraphs, but this

thesis uses the same term for both graphs and digraphs since it causes no ambiguity to do so. Conventionally n and e are used to denote the cardinalities of V and E respectively unless otherwise indicated.

If either $u \rightarrow v \in E$ or $v \rightarrow u \in E$ then u and v are said to be *adjacent*. If $u \rightarrow v \in E$ then u is called the *tail* of the edge and v the *head*. The *indegree* of $u \in V$ ($indeg\ (u)$) is defined as the number of nodes $v \in V$ such that $v \rightarrow u \in E$. The *outdegree* of $u \in V$ ($outdeg\ (u)$) is defined as the number of nodes $v \in V$ such that $u \rightarrow v \in E$.

A *path* P of *length* $t-1$ *from* v_1 *to* v_t in G is a finite non-empty sequence $P = (v_1 \rightarrow v_2, v_2 \rightarrow v_3, \ldots, v_{t-2} \rightarrow v_{t-1}, v_{t-1} \rightarrow v_t)$ of edges in E such that either $t = 2$ (in which case the path consists of a single edge) or $t > 2$ and the head of the i^{th} edge in the sequence is equal to the tail of the $(i+1)^{st}$ edge for $i = 1, 2, \ldots, t-2$. P will usually be denoted by $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_t$. P is a *cycle* if $v_t = v_1$ and a *loop* if additionally the cycle is of length 1 (i.e. $P = v_1 \rightarrow v_1$). G is said to be *acyclic* if it contains no cycles.

A *subpath* S of a path P is a subsequence of P such that S is a path and furthermore, is said to be a *proper subpath* if $S \neq P$. A path P is a *simple path* if P does not have a cycle as a subpath. A cycle C is a *simple cycle* if C does not have a proper subpath which is a cycle. The *diameter* of G is the length of the longest simple path in G.

It should be noted that there is some disagreement in the literature on the definition of paths and cycles. Some references use the above definitions (see [18] for example), while others require paths and cycles to be simple, and use the terms *walk* and *closed walk* if the path is not simple (see [2] for example).

A node $v \in V$ is said to be *reachable* from $u \in V$ if there is a path from u to v in G. Given $W \subset V$ and $u, v \in V - W$, we say that v is *reachable from u through W* if there is a path $P = u \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_{t-1} \rightarrow q_t \rightarrow v$ from u to v in G such that $t \geq 1$ and $\{q_1, q_2, \ldots, q_t\} \subseteq W$. In particular P must have length at least 2.

A *subdigraph* of G is a digraph $G_1 = (V_1, E_1)$, where $V_1 \subseteq V$ and $E_1 \subseteq E$. G_1 is said to be *spanning* if $V_1 = V$. G_1 is *induced by* V_1 if $E_1 = E \cap (V_1 \times V_1)$. If $v \in V$ then the subdigraph induced by $V - \{v\}$ is denoted by $G - v$. Suppose G is acyclic and $E_{red} = \{i \rightarrow j \in E$ *there is no path of length* ≥ 2 *in* G *from* i *to* $j\}$. Then the subdigraph $G_{red} = (V, E_{red})$ is the *reduced* subdigraph of G (see [18], p. 9).

G is said to be *strongly connected* if for every ordered pair (u, v) of distinct nodes from V, there is a simple path in G from u to v. A strongly connected *component* of G is

an induced subdigraph $G_1 = (V_1, E_1)$ of G such that if $G_2 = (V_2, E_2)$ is any strongly connected subdigraph of G for which $V_1 \subseteq V_2$ then $V_1 = V_2$.

2.1.1.2 Graphs

A *graph* is a digraph $G = (V, E)$ such that $u \rightarrow v \in E$ if and only if $v \rightarrow u \in E$. The term *undirected graph* is used synonymously with graph when we wish to emphasize that it is not an arbitrary digraph. Sometimes a graph is defined by requiring that E be a set of unordered pairs of not necessarily distinct elements of V (see e.g. [2], p. 1). Since this definition is less convenient in an algorithmic setting, it will not be used. Since a graph is a special type of digraph, all of the definitions from section 2.1.1.1 are applicable to graphs, except as modified in this section.

For graphs, the indegree is the same as the outdegree for each node, and this common value is referred to as the *degree* of the node, denoted by $deg(u)$. It is also standard to use the term *connected* for a strongly connected graph. A graph is *acyclic* if the only cycles it contains have length 2.

If G is an arbitrary digraph which is not necessarily a graph, then a *subgraph* of G is a subdigraph which is also a graph. If G is a digraph, then the *maximum subgraph* of G is the spanning subgraph $G_1 = (V, E_1)$, where $i \rightarrow j$ is in E_1 if and only if both $i \rightarrow j$ and $j \rightarrow i$ are in E.

A *forest* is an acyclic graph, and a connected forest is called a *tree*. A *rooted tree* is a triple $T_v = (V, E, v)$, where $T = (V, E)$ is a tree and $v \in V$ is a distinguished node called the *root*. A *spanning tree* of an arbitrary digraph is a spanning digraph which is a tree. A *subtree* of a tree T is a subgraph of T which is also a tree. A node of degree 1 in a tree is said to be *pendant*.

A pendant node in a rooted tree, excluding the root itself, is called a *leaf*. If u and v are distinct nodes in a rooted tree with root r, then u is said to be an *ancestor* of v (and v is a *descendent* of u) if there is a simple path from r to v that passes through u. Furthermore, u is the *parent* of v and v is a *child* of u if u is an ancestor of v and $u \rightarrow v$ is an edge in the tree. If v and w have the same parent node, then they are called *siblings*.

2.1.1.3 Orderings

An *ordered digraph* is a triple $G_\alpha = (V, E, \alpha)$ where $G = (V, E)$ is a digraph and $\alpha: \{1, 2, \ldots, n\} \to V$ is a bijection. The bijection α is called an *ordering* on G. The terms *labelled* or *numbered* are also used in some references instead of *ordered*. The terms *ordered graph*, *ordered tree* and *ordered rooted tree* are used for the corresponding digraph specializations. Since an ordering is defined in terms of the node set V only, it is possible for a given ordering to apply to any number of digraphs with a common node set. This is done in several places in this thesis. For the rest of this section $G = (V, E)$ is an arbitrary digraph, α is an ordering on G and G_α is the corresponding ordered digraph.

A *forward* [*backward*] edge in G_α is an edge $u \to v$ such that $\alpha^{-1}(u) < \alpha^{-1}(v)$ [$\alpha^{-1}(u) > \alpha^{-1}(v)$]. A path in an ordered digraph is said to be a *forward path* [*backward path*] if all edges in the path are forward [backward].

α is said to be a *topological sort* of G if $\alpha^{-1}(u) < \alpha^{-1}(v)$ for every edge $u \to v \in E$. It is easily shown that a topological sort exists for G if and only if G is acyclic.

If $V = \{1, 2, \ldots, n\}$ then the identity function $\iota: V \to V$ defines the *natural ordering* of G.

If G is a forest, then α is said to be an *invariant ordering* (and G_α is said to be *invariantly ordered*) if for each $v \in V$ there is at most one $u \in V$ with $\alpha^{-1}(u) > \alpha^{-1}(v)$ such that u is adjacent to v.

If G is a graph then α is said to be a *minimum degree ordering* (and G_α is said to be *minimum degree ordered*) if α is any ordering that could be produced by the minimum degree algorithm (see 2.1.3.4).

In most applications, at most one ordering at a time is considered for a digraph. In these cases the ordering subscript is dropped, for instance G is used in place of G_α.

2.1.2 Representations

This section is concerned with how digraphs are represented in an algorithm. The node set of a digraph G may be an arbitrary finite set, but for a representation it is convenient to assume that $V = \{1, 2, \ldots, n\}$, where the association of nodes in V with integers in $\{1, 2, \ldots, n\}$ is made in some convenient manner. For instance, if G is an ordered digraph with ordering α, then $v \in V$ is identified with $\alpha^{-1}(v)$. Consequently, α

can be assumed to be the identity

There are two ways that the edge set is usually represented. The simplest representation is the *adjacency matrix* which is an $n \times n$ Boolean matrix M where $m_{i,j} = 1$ if and only if $i \rightarrow j \in E$. G is a graph if and only if the adjacency matrix is symmetric. The adjacency matrix representation has the advantage of constant time access to a particular edge entry but requires $O(n^2)$ time to traverse the digraph and requires $O(n^2)$ space for storage. This is the representation to use if the digraph is dense in the sense that $e >> n$.

The other common representation is the *adjacency list* which is an n-vector ADJ where $ADJ[i]$ is the list of nodes j for which $i \rightarrow j \in E$. With this representation the access time for an edge entry is $O(\max outdeg(i))$ if the lists are unsorted and $O(\log(\max outdeg(i)))$ if the lists are sorted and realized as static arrays. A digraph traversal takes time $O(n+e)$. The storage required is also $O(n+e)$. This representation is most useful when the digraph is sparse in the sense that $e << n^2$.

When representing an ordered digraph by an adjacency list structure it is sometimes useful to use a pair of adjacency lists. The *forward adjacency list* $FADJ$ is defined so that $FADJ[i]$ is the list consisting of $\{ j \mid i \rightarrow j \text{ is a forward edge} \}$. The *backward adjacency list* $BADJ$ is defined similarly for backward edges. Normally both $FADJ$ and $BADJ$ are sorted lists since sorting can be done with no extra work when $FADJ$ and $BADJ$ are computed from ADJ (see section 2.1.3.3).

2.1.3 Basic algorithms

There are a number of simple digraph algorithms which will be useful in simplifying the statement of some of the algorithms in Chapter 4. This section gives informal descriptions of these algorithms together with their execution times.

2.1.3.1 Sorting the adjacency lists of an ordered digraph

It is frequently more efficient when working with ordered digraphs to assume that the adjacency lists are sorted. One way to do this involves reversing the digraph twice. The reverse of a digraph $G = (V, E)$ is the digraph $G_r = (V, E_r)$ where $i \rightarrow j \in E_r$ if and only if $j \rightarrow i \in E$.

Assume $V = \{1, 2, \ldots, n\}$, the ordering on G is the identity and E is represented by adjacency list ADJ. The algorithm first computes the adjacency list $RADJ$ for E_r. It loops from n down to 1, where for each j in $ADJ[i]$, i is added to the head of the list $RADJ[j]$. The result is that each list in $RADJ$ is sorted. By applying the algorithm once more to $RADJ$, this time a sorted adjacency list for G is obtained. The time required to sort the adjacency list of G is $O(n+e)$.

2.1.3.2 Computing FADJ and BADJ for an ordered digraph

A simple way to do this is to compute $RADJ$ using the algorithm in 2.1.3.1 and then apply a modified version of the same algorithm to $RADJ$, where for $j \in RADJ[i]$, i is added to the head of $FADJ[j]$ if $i > j$ and otherwise i is added to the head of $BADJ[j]$. This computes $FADJ$ and $BADJ$ in time $O(n+e)$ and both adjacency lists are sorted.

2.1.3.3 TOPSORT: Topological sort of an acyclic digraph

A simple algorithm to compute a topological sort of an acyclic digraph $G = (V, E)$ (see [18], pp. 4-7 for instance) can be described recursively as follows. If $n = 1$ and $V = \{v\}$ then $\alpha(1) = v$ defines a topological sort. If $n > 1$, then choose $v \in V$ such that $indeg(v) = 0$ and define $\alpha(1) = v$. Such a v exists since G is acyclic. The digraph $G - v$ has $n-1$ nodes and by induction may be assumed to be topologically sorted via $\alpha: \{2, \ldots, n\} \to V - \{v\}$. The ordering $\alpha: \{1, 2, \ldots, n\} \to V$ is then a topological sort of G. By careful selection of data structures, this algorithm can be executed in time $O(n+e)$.

2.1.3.4 MINDEG: Minimum degree algorithm for graphs

This is an algorithm for computing an important class of orderings on graphs, the minimum degree orderings (see section 2.1.1.3). For our purposes it is most convenient to use a recursive version of the algorithm. For input digraph $G = (V, E)$ and integer i, MINDEG(G, i) computes a bijection α in the following steps.

(1) $\alpha(i) \leftarrow v$ where v has minimum degree in G

(2) **if** $|V| > 1$ **then** MINDEG $(G - v, i+1)$

Any bijection resulting from MINDEG(G, 1) is an ordering on G. The ordering produced by MINDEG depends on which minimum degree nodes are chosen in each recursive execution of step 1, so that in general a minimum degree ordering for a graph is not unique. We will only be using MINDEG when the input digraph is a forest. In this case MINDEG executes in time $O(n)$.

2.1.3.5 MAXSUBGRAPH: Maximum subgraph of a digraph

The maximum subgraph is the largest (undirected) graph contained in the digraph. Some algorithms can be made to run faster if the maximum subgraph of the input digraph has certain properties.

The input to MAXSUBGRAPH is a digraph $G = (V, E)$ where $V = \{1, 2, \ldots, n\}$ and E is represented by a sorted adjacency list ADJ. The output is a sorted adjacency list $MADJ$ for the maximum subgraph of G.

Initially $MADJ[i] = \emptyset$ for all $i \in V$. MAXSUBGRAPH then loops through i from 1 to n, where for each $j \in ADJ[i]$ such that $j > i$ and $i \in ADJ[j]$, j is added to $MADJ[i]$ and i is added to $MADJ[j]$. The time required to execute MAXSUBGRAPH is $O(n + e)$.

2.1.4 Lemmas

This section contains two simple lemmas concerning strongly connected digraphs which are needed in later work.

Lemma 2.1 : Let $G = (V, E)$ be a strongly connected digraph and let $u \rightarrow v \in E$. Then there is a simple cycle C in G such that $u \rightarrow v$ is an edge in C.

Proof By the definition of strong connectivity, there is a simple path P from v to u in G. If C is the path formed by appending the edge $u \rightarrow v$ to P, then C is a simple cycle. ■

Lemma 2.2 : Let $G = (V, E)$ be a strongly connected digraph, let C be a simple cycle in G and let $v \in V$ be such that v is not on C. Then there is some node c on C such that there is a path from c to v that does not contain any node on C other than c.

Proof Let d be an arbitrary node on C. By strong connectivity there is a simple path $P : (d = v_1) \rightarrow v_2 \rightarrow \quad \rightarrow (v_k = v)$. Let i be the largest index for which v_i is a node on C. Then $c = v_i$ is the required node. ■

2.2 Matrices, factorizations and Gaussian elimination

The background material concerning matrices is presented in this section, with particular emphasis given to triangular factorizations and Gaussian elimination. The fundamental definitions and terminology applying to matrices, vectors, LU and UL factorizations and Gaussian elimination are presented in section 2.2.1. Methods for solving systems of linear equations are discussed in section 2.2.2. In section 2.2.3 representations of matrices by digraphs are described.

2.2.1 Fundamental definitions and terminology

2.2.1.1 Matrices and vectors

The fundamentals of matrix algebra and determinant theory can be found in many references (see e.g. [14], pp. 1-32 for a summary of the essential concepts). This section introduces the terminology to be used in this thesis and the definitions of special types of matrices.

Matrices are denoted by upper case letters and matrix elements by the corresponding lower case letter with subscripts. For instance, an $n \times m$ matrix A will also be denoted by $(a_{i,j})$ where $1 \leqslant i \leqslant n$ is the row index and $1 \leqslant j \leqslant m$ is the column index. An $n \times n$ matrix is also referred to as an *order* n matrix or as a square matrix if the size is not important. A *vector* is an $n \times 1$ matrix and a $1 \times n$ matrix is called a *row vector*. The components of vectors and row vectors will usually be written with single subscripts since the missing subscript is known to be 1.

Let $A = (a_{i,j})$ be an order n matrix. A is *upper [lower] triangular* if $a_{i,j} = 0$ whenever $i > j$ [$j > i$]. A is *strictly upper [lower] triangular* if $a_{i,j} = 0$ whenever $i \geq j$ [$j \geq i$]. An upper [lower] triangular matrix is a *unit* upper [lower] triangular matrix if $a_{i,j} = 1$ whenever $i = j$. If $a_{i,j} = 0$ whenever $i \neq j$ then A is a *diagonal* matrix. A is *upper [lower] Hessenberg* if $a_{i,j} = 0$ whenever $i > j+1$ [$j > i+1$]. A matrix which is both upper and lower Hessenberg is zero everywhere off the main diagonal and the sub-diagonals immediately above and below. Such a matrix is said to be *tridiagonal* for obvious reasons.

A square matrix A is said to be *combinatorially symmetric* provided $a_{i,j} = 0$ if and only if $a_{j,i} = 0$.

A *permutation matrix* is a square matrix whose elements are all zero except for precisely one element equal to 1 in each row and column. If A is an $n \times m$ matrix, B is an $m \times n$ matrix and P is an order n permutation matrix, then multiplying A on the left by P permutes the rows of A and multiplying B on the right by P permutes the columns of B.

Let A be an order n matrix. Then A is *reducible* if either $n = 1$ and $A = 0$ or $n > 1$ and there exists a permutation matrix P such that

$$PAP^T = \begin{bmatrix} B & C \\ 0 & D \end{bmatrix}$$

where for some k, $1 \leq k < n$, B is a $k \times k$ submatrix, C is a $k \times n-k$ submatrix, D is an $n-k \times n-k$ submatrix, and the zero submatrix has size $n-k \times k$. If A is not reducible then it is said to be *irreducible*.

There is a normal form associated with reducibility. Let A be an arbitrary square matrix. Then there is a permutation matrix P such that

$$PAP^T = \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,r-1} & B_{1,r} \\ 0 & B_{2,2} & \cdots & B_{2,r-1} & B_{2,r} \\ & & \cdots & & \\ & & \cdots & & \\ 0 & 0 & \cdots & 0 & B_{r,r} \end{bmatrix}$$

where each $B_{i,i}$ is a square submatrix which is either irreducible or an order 1 zero matrix. This is known as the *reduced normal form* of A. There is also a block lower triangular analog which is sometimes more convenient to use. We will use the term *reduced normal form*

for either the block lower or block upper triangular form

2.2.1.2 Factorizations

Suppose the matrix equation $U\mathbf{x} = \mathbf{b}$ is to be solved for $\mathbf{x}$ where U is an order n nonsingular upper triangular matrix and $\mathbf{x}$ and $\mathbf{b}$ are n-vectors. There is a simple algorithm usually called *backward substitution* which in the absence of rounding errors computes the solution $\mathbf{x}$ (see e.g. [24] algorithm 1.3, p. 108). Similarly if L is a nonsingular lower triangular matrix of order n then $L\mathbf{x} = \mathbf{b}$ can be solved by a related algorithm called *forward substitution*. If L and U are respectively lower and upper triangular nonsingular matrices and $A = LU$ then the matrix equation $A\mathbf{x} = \mathbf{b}$ can be solved for $\mathbf{x}$ by first solving $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ and then solving $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

A decomposition $A = LU$, where L is lower triangular and U is upper triangular is referred to as an *LU factorization*. There is no guarantee that an LU factorization exists even if A is nonsingular. For example

$$PAP^T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

has no LU factorization. However if A is nonsingular then there exist permutation matrices P and Q such that PAQ has an LU factorization.

An LU factorization of A is not unique. If D is a nonsingular diagonal matrix of order n then $LU = (LD^{-1})(DU)$ where LD^{-1} is still lower triangular and DU is upper triangular.

The effect of left multiplication of a matrix M by a diagonal matrix D is to multiply each row i of M by $d_{i,i}$. Therefore if U is nonsingular then D can be chosen so that DU has any desired sequence of nonzero elements along the diagonal. Alternatively, if L is nonsingular then D can be chosen so that LD^{-1} has a specified nonzero diagonal. The usual choices are to normalize either L or U so that one of them has 1 at each diagonal position. If L is normalized in this way then the resulting factorization is called a *left unit LU factorization* (also called a *Doolittle decomposition*). If U is normalized then it is called a *right unit LU factorization* (also called a *Crout decomposition*).

If A is positive definite then it can be shown that A has an LU factorization of the form LL^T (see e.g. [24] theorem 3.8, pp. 140-141). The factorization $A = LL^T$ is known as the *Cholesky factorization* of A.

The order of the triangular factors in the above factorizations with the lower preceding the upper is completely arbitrary It is just as valid to consider factorizations where the order is reversed Suppose A is factored as $A = UL$ where U is upper triangular and L is lower triangular If U is normalized to be unit triangular then the factorization is called a *left unit UL factorization* If L is so normalized then the factorization is called a *right unit UL factorization* There exist analogues of the LU results for the UL case

2.2.1.3 Gaussian elimination

For the purposes of this thesis Gaussian elimination (hereafter referred to as GE) is an algorithm for computing a left unit LU factorization of a nonsingular matrix Storage requirements are minimized by overwriting the upper triangular part of A with U and the strict lower triangular part of A with L (omitting the unit diagonal) The algorithm appears as follows

(1) **for** k **from** 1 **to** $n-1$ **do begin**

(2) **for** i **from** $k+1$ **to** n **do begin**

(3) $a_{i\,k} \leftarrow \dfrac{a_{i\,k}}{a_{k\,k}}$

(4) **for** j **from** $k+1$ **to** n **do begin**

(5) $a_{i\,j} \leftarrow a_{i\,j} - a_{i\,k}a_{k\,j}$ **end end end**

In step 5 the value of $a_{i\,k}$ is the new value computed in the previous step 3 For given k the new entry $a_{i\,j}$ computed in step 5 is often denoted by $a_{i}^{(k)}_{j}$ and the $n-k \times n-k$ submatrix of entries computed by the loop 2 to 5 by $\hat{A}_k$ The time to execute GE is $O(n^3)$

As a practical algorithm GE has some deficiencies If one of the *pivot elements* $a_{k\,k} = 0$ for some k $1 \leqslant k < n$ then GE can not be executed to completion Even if the first $n-1$ pivots are nonzero the algorithm may behave poorly if it encounters pivots which are small compared to other matrix elements as rounding error may result in an unacceptable loss of precision in the subtraction in step 4 This is a manifestation of *instability* in GE (see 2 2 1 4)

These problems with GE are usually remedied by employing a pivoting strategy to reorder the rows and/or the columns of the matrix so that unnecessarily small pivots do not occur With respect to the solution of systems of linear equations (see section 2 2 2),

row interchanges correspond to reordering the equations and column interchanges to renaming the variables. The two most popular pivoting strategies are *partial pivoting* and *complete pivoting*.

The partial pivoting strategy interchanges two rows of A in the k^{th} iteration of the outer loop of GE such that the new element in position k,k satisfies $|a_{k,k}| = \max\{|a_{i,k}| \; k \leq i \leq n\}$. In almost all practical cases partial pivoting is adequate to ensure numerical stability. There are, however, exceptional cases where small pivots do occur when partial pivoting is used. GE with partial pivoting is denoted by GEPP.

The complete pivoting strategy does lead to a stable algorithm but is more expensive to implement. In the k^{th} iteration of the outer loop of GE, two rows and/or two columns may be interchanged so that the new element in position k,k satisfies $|a_{k,k}| = \max\{|a_{i,j}| \; k \leq i, j \leq n\}$. GE with complete pivoting is denote by GECP.

2.2.1.4 Stability and conditioning

The concepts of *conditioning* and *stability* are important to any discussion of numerical algorithms. Suppose $f: D \subseteq \mathbf{R}^m \rightarrow \mathbf{R}^n$ is a function. If $x \in D$, then x can be regarded as the data that defines a *problem* and $f(x)$ as the answer to the problem. The problem is said to be *ill-conditioned* (equivalently f is *ill-conditioned at* x) if for some $\hat{x}$ close to x, $f(\hat{x})$ is not close to $f(x)$. The solution of an ill-conditioned problem may be very inaccurate if, for example, the data specifying the problem is inexact or if the computed solution is subject to round-off errors.

Suppose an algorithm $\hat{f}$ is defined to produce approximate solutions to problems in the domain of f. $\hat{f}$ is said to be a *stable* algorithm if for any $x \in D$, there is some $\hat{x}$ close to x such that $f(\hat{x})$ is close to $\hat{f}(x)$. In other words, for given input data $\hat{f}$ produces a solution which is close to the exact solution of a nearby problem. The significance of a stable algorithm is that it does not introduce inaccuracies of its own which are significantly larger than the inaccuracies due to using approximate data. See e.g. [24], pp. 68-80 for a more detailed discussion of stability.

2.2.2 Direct solution of systems of linear equations

A direct method of solution for a system of linear equations will, in the absence of rounding errors, produce an exact solution to the system if it exists. In contrast an iterative method of solution attempts to find a sequence of approximate solutions that converges to an exact solution. Iterative methods will not be considered further in this thesis. Consider a system of n equations in the n unknowns $x_1, x_2, \ldots, x_n$.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned} \tag{2.1}$$

System (2.1) is equivalent to the matrix equation $A\mathbf{x} = \mathbf{b}$ if the usual element naming conventions are observed. If A is nonsingular, then this matrix equation is solved for $\mathbf{x}$ by first using GE to compute a left unit LU factorization of A (or using GEPP or GECP to compute a left unit LU factorization of PA or PAQ, respectively, where P and Q are permutation matrices). The equation $L\mathbf{y} = \mathbf{b}$ is solved for $\mathbf{y}$ and then $U\mathbf{x} = \mathbf{y}$ is solved for $\mathbf{x}$ (if GEPP is used solve $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{x} = \mathbf{y}$ and if GECP is used solve $L\mathbf{y} = P\mathbf{b}$ and $U\mathbf{z} = \mathbf{y}$, then compute $\mathbf{x} = Q\mathbf{z}$).

Solving the triangular systems by forward and backward substitution requires time $O(n^2)$ while the time required to compute the LU factorization is $O(n^3)$. For large systems most of the work is done computing the LU factorization. If there are several systems to be solved using the same matrix A and different vectors $\mathbf{b}$, then the LU factorization need only be computed once, saving considerable time. If only one system is to be solved, then some time can be saved by not computing the complete LU factorization. L need not be stored since the solution to $L\mathbf{y} = \mathbf{b}$ can be computed at the same time that U is being computed.

2.2.3 Representing a matrix by a digraph

Let A be a matrix of order n. Define $D(A)$, the *digraph of* A, by $D(A) = (V, E)$ where $V = \{1, 2, \ldots, n\}$ and $i \to j$ is in E if and only if $a_{i,j} \neq 0$. Equivalently let $\hat{A}$ be the matrix that has 1 in every position in which A is nonzero and 0 everywhere else. Then $D(A)$ is the digraph with adjacency matrix $\hat{A}$. $D(A)$ is a graph (undirected) if and only if A is combinatorially symmetric.

A matrix A is irreducible (see 2.2.1.1) if and only if $D(A)$ is strongly connected (see e.g. [14], theorem 6.2.24, p. 362). This result is indicative of the type of problem for which the concept of the digraph of a matrix is useful, that is problems for which the positions of zero and nonzero elements are important but not the value of the nonzeros. Such problems are called qualitative (as opposed to quantitative) problems.

Problems for which the digraph of a matrix can be used profitably generally involve several phases. The digraph representation is useful in what is sometimes called the symbolic phase; other representations may be used in other phases. For instance, if GE is applied to a sparse matrix being efficiently represented with respect to storage, then the digraph of the matrix is useful in determining which zero entries will suffer fill-in (see section 3.1) and therefore require the reservation of additional storage. See [9], [21] and section 3.5 in this thesis for further details.

2.3 Algorithm format

An attempt has been made to use a standard format for presenting the algorithms in chapter 4. The statement of each algorithm consists of a number of sections.

Input

This section describes the initial data required by the algorithm.

Output

This section describes the precise value(s) returned by the algorithm.

Data structures

Any non-primitive data structures such as queues and stacks which are used by the algorithm, and are not input data, are listed and described here. If any of the data structures have access functions, then they are also defined here. Not listed are simple structures like Boolean and integer variables.

Sub-algorithms

This section contains a description (or a reference to section 2.1.3) for each sub-algorithm used.

Algorithm

A descriptive version of the algorithm is provided. It is designed to be sufficiently

rigorous to allow correctness to be proved and complexity to be estimated, and yet be simple enough to be easily understood

CHAPTER 3

Inheritance in LU and UL factorizations

This chapter contains a brief survey of the known results concerning inheritance in LU and UL factorizations. The fundamental definitions and theorems concerning inheritance are stated and some useful lemmas are stated and proved. As far as is known, the only paper concerned specifically with this subject is [15]. The proofs of the theorems may be found there.

In section 3.1 the concept of inherited matrix entries in LU factorizations is introduced and some fundamental definitions and terminology are discussed. The local inheritance problem for LU factorizations (in which a single matrix entry is inherited by L or U) is considered in section 3.2. In section 3.3 the global inheritance problem for LU factorizations (in which the entire [strict] upper or lower triangular part of a matrix is inherited) is considered. In section 3.4 the results of sections 3.2 and 3.3 are extended to UL factorizations. In section 3.5 some topics related to inheritance (fill-in for instance) are surveyed and the importance of inheritance to sparse matrix analysis is discussed.

3.1 Fundamental definitions and terminology

In many sparse matrix applications, a matrix is represented in a form in which as few nonzero elements as possible are stored. It is important for the prediction of storage requirements to know which zero entries will become nonzero during a matrix operation. Of particular interest is the *fill-in* that occurs when a matrix is overwritten by its unit LU factorization. We say that fill-in occurs at position $i\ j$ if and only if $a_{i,j} = 0$ initially but $a_{i,j} \neq 0$ after some iteration of step 5 of GE (see 2.2.1.3).

If no fill-in occurs in some zero entry following a matrix operation, then the value of that entry has been inherited from the original matrix. It is then reasonable to look at generalizations of this in which arbitrary matrix entries are inherited.

As an example, suppose A is a lower Hessenberg matrix and there exists a left unit LU factorization $A = LU$. Then a simple computation shows that $u_{i,j} = a_{i,j}$ for all

$i = 1, 2, \ldots, n-1$ and $j > i$. The equality is independent of the actual numbers assigned to any entries $a_{i,j}$. This example illustrates a simple case of global inheritance (see section 3.3).

If A is an arbitrary matrix, then it is possible that $u_{i,j} = a_{i,j}$ for some i and j because of a fortuitous choice of numbers for the entries of A. Situations such as this are of no interest in the study of inheritance. Instead, the customary assumption in sparse matrix analysis, that accidental cancellations do not occur when studying zero patterns, is extended to more general situations. The following definition is given in [15]. "Given a matrix A with digraph $D(A)$, we say that two values f and g computable from the entries of A are *equal generically* (written $f = g$ (generically)) if $f(\hat{A}) = g(\hat{A})$ for all $\hat{A}$ such that $D(\hat{A}) = D(A)$."

Definition 3.1: Let A be a matrix which has a unique left [right] unit LU factorization $A = LU$. An entry $u_{i,j}$ [$l_{i,j}$] in U [L] is said to be *inherited* from A if $u_{i,j} = a_{i,j}$ (generically) [$l_{i,j} = a_{i,j}$ (generically)].

Let A be a matrix of order n. Define $d_k(A)$ to be the determinant of the leading principal submatrix of order k. Let D be a digraph on n nodes. Then $\mathbf{A}_D$ denotes the set of all $n \times n$ matrices A such that $D(A)$ is a subdigraph of D and $d_k(A) \neq 0$ for $k = 1, 2, \ldots, n-1$. The reason for the restriction on the principal minors of the matrix is to ensure that a left [right] unit LU factorization of A exists and is unique (see [15], theorem 1.1).

3.2 Local inheritance

This section is concerned with the *local inheritance problem*, in which conditions are sought under which $a_{i,j} = u_{i,j}$ [$a_{i,j} = l_{i,j}$] in the left [right] unit LU factorization $A = LU$, for some specific pair $i \leq j$ [$j \leq i$].

Definition 3.2: (see [15] section 2) Let A be a matrix of order n and let $1 \leqslant i \; j \leqslant n$. Then A is said to be $(i \; j)$ *lower restricted* if j is not reachable from i through $\{1 \; 2 \quad \min\{i \; j\}-1\}$ in $D(A)$. In such a case the natural ordering on $D(A)$ is said to be an $(i \; j)$ *lower restricted ordering*.

If $G = (V \; E \; \alpha)$ is an arbitrary ordered digraph then G is also said to be $(i \; j)$ lower restricted provided $\alpha(j)$ is not reachable from $\alpha(i)$ through $\{\alpha(1) \; \alpha(2) \quad \alpha(\min\{i \; j\}-1)\}$.

Theorem 3.1: (see [15] theorem 2.2) Let D be a digraph on n nodes and let $1 \leqslant i \leqslant j \leqslant n$ $[1 \leqslant j \leqslant i \leqslant n]$. Then for all $A \in \mathbf{A}_D$ $u_{i\,j} = a_{i\,j}$ $[l_{i\,j} = a_{i\,j}]$ in the left [right] unit LU factorization of A if and only if either

(i) A is $(i \; j)$ lower restricted or

(ii) if j is reachable from i through nodes $p_1 \quad p_t \in \{1 \quad i-1\,[j-1]\}$ then $\det A\,[\{1 \; 2 \quad ,i-1\,[j-1]\}-\{p_1 \quad p_t\}] = 0$ for all $A \in \mathbf{A}_D$.

The determinant in condition (ii) of theorem 3.1 is a complementary principal minor of A and condition (ii) is referred to as the *complementary minor condition*.

3.3 Global inheritance

This section is concerned with the *global inheritance problem* in which conditions are sought under which $a_{i\,j} = u_{i\,j}$ $[a_{i\,j} = l_{i\,j}]$ in the left [right] unit LU factorization $A = LU$ for all pairs $i < j$ (or $i \leqslant j$) $[j < i$ (or $j \leqslant i)]$.

Definition 3.3: Let A be a matrix of order n. Then A is said to be *forward* [*backward*] *lower restricted* if it is $(i \; j)$ lower restricted for all $i \; j$ such that $1 \leqslant i < j \leqslant n$ $[1 \leqslant j < i \leqslant n]$. In such a case the natural ordering on $D(A)$ is said to be a *forward* [*backward*] *lower restricted ordering*. In either usage the term *forward* [*backward*] *lower restricted* is hereafter abbreviated to FLR [BLR].

An arbitrary ordered digraph is also said to be FLR [BLR] ordered if it is (i, j) lower restricted for all i, j such that $1 \leqslant i < j \leqslant n$ $[1 \leqslant j < i \leqslant n]$. (See the note following definition 3.2.)

Global inheritance for all pairs $i < j$ $[j < i]$ is characterized in the following theorem. Unlike the corresponding characterization for local inheritance, the complementary minor condition is no longer required, resulting in a simpler statement.

Theorem 3.2: (see [15], theorem 3.1, corollary 3.2 and section 4) Let D be a naturally ordered digraph on the node set $V = \{1, 2, \ldots, n\}$. Then, for all $A \in \mathbf{A}_D$ and for all pairs i, j such that $1 \leqslant i < j \leqslant n$ $[1 \leqslant j < i \leqslant n]$, $u_{i,j} = a_{i,j}$ $[l_{i,j} = a_{i,j}]$ in the left [right] unit LU factorization of A if and only if D is FLR [BLR] ordered.

Global inheritance for all pairs $i \leqslant j$ $[j \leqslant i]$ has a similar characterization which is the subject of the following corollary to theorem 3.2.

Corollary 3.1: (see [15], corollary 3.4) Let D be a naturally ordered digraph on the node set $V = \{1, 2, \ldots, n\}$. Then, for all $A \in \mathbf{A}_D$ and for all pairs i, j such that $1 \leqslant i \leqslant j \leqslant n$ $[1 \leqslant j \leqslant i \leqslant n]$, $u_{i,j} = a_{i,j}$ $[l_{i,j} = a_{i,j}]$ in the left [right] unit LU factorization of A if and only if D is (i, j) lower restricted for all pairs $i \leqslant j$ $[j \leqslant i]$.

A matrix A which satisfies the conditions of corollary 3.1 is a member of a severely restricted class of matrices. Since A is (i, i) lower restricted for each i, $D(A)$ cannot contain a k-cycle for any $k \geqslant 2$. In other words, the only possible cycles in $D(A)$ are loops. Consequently $D(A)$ and any subdigraphs with more than one node are not strongly connected. This has two important consequences.

Firstly, $\det A = \prod_{i=1}^{n} a_{i,i}$ from the alternating sum characterization of the determinant. (See e.g. [14], p. 8 for the definition of the alternating sum characterization.)

Secondly, all principal submatrices of A are reducible. This means that the reduced normal form of A is upper triangular. In the terminology of [14], p. 26, A is *essentially triangular*.

Matrices for which $u_{i,j} = a_{i,j}$ for all pairs $i \leqslant j$ will not be considered beyond this chapter the principal focus of this thesis being FLR ordered matrices Theorem 3 2 justifies studying FLR ordered digraphs instead of matrices Although any practical results obtained will ultimately apply to matrices it is convenient to work entirely within the domain of digraphs

We note that an FLR ordered digraph remains FLR ordered if an arbitrary set of loops is added to or removed from the digraph Similarly a digraph which is not FLR ordered does not become FLR ordered if any set of loops is added or removed From these observations it is clear that loops are irrelevant to the study of FLR ordered digraphs Therefore it will be assumed that all digraphs are loop-free for the remainder of this thesis

The following two lemmas define tests for the FLR [and BLR] property in ordered digraphs which in practice are more convenient than definition 3 3

Lemma 3.1: Let $D = (V\ E\ \alpha)$ be an ordered digraph Then D is not FLR [BLR] ordered if and only if there are three distinct nodes i j and $k \in V$ such that $\alpha^{-1}(j) < \alpha^{-1}(i) < \alpha^{-1}(k)$ $i \rightarrow j$ [$j \rightarrow i$] is in E and there is a forward [backward] path not containing i from j to k [k to j]

Proof Only the FLR case is proved the BLR case is similar Assume without loss of generality that $V = \{1\ 2 \quad n\}$ and α is the identity function Suppose there exist three nodes i j and k with the given properties If the forward path from j to k is $P = j \rightarrow j_1 \rightarrow \quad \rightarrow j_r \rightarrow k_1 \rightarrow k_2 \rightarrow \quad \rightarrow k$ where k_1 is the first node on P such that $k_1 > i$ then k_1 is reachable from i through $\{j\ j_1, \quad, j_r\} \subseteq \{1\ 2 \quad i-1\}$ Therefore since D is not $(i\ k_1)$ lower restricted it is not FLR ordered

Conversely suppose D is not FLR ordered Then for some nodes i and k $i < k$ there is a simple path $P = (i = j_0) \rightarrow j_1 \rightarrow \quad \rightarrow j_{r-1} \rightarrow (j_r = k)$ with $j_1\ j_2 \quad j_{r-1} \in \{1\ 2 \quad i-1\}$ The first edge in P is a backward edge This means that there is a largest index p such that $j_{p-1} \rightarrow j_p$ is a backward edge Since the last edge in P is a forward edge $p < r$ Then the three nodes j_{p-1} j_p and j_r and the forward path $j_p \rightarrow j_{p+1} \rightarrow \quad \rightarrow j_r$ satisfy the requirements of the lemma ■

A forward path $j \rightarrow j_1 \rightarrow \quad \rightarrow j_s \rightarrow k$ is said to *jump* i if $j < i < k$ and $i \notin \{j_1\ j_2, \quad, j_s\}$

Lemma 3.2: Let $D = (V\ E\ \alpha)$ be an FLR [BLR] ordered digraph and let C be an arbitrary simple cycle in D. Then C contains precisely one backward [forward] edge.

Proof. Only the FLR case is proved; the BLR case is similar. Assume without loss of generality that $V = \{1\ 2\ \ldots\ n\}$ and α is the identity function. C contains at least one backward edge since the edge in C with tail equal to the largest node in the node set of C must be a backward edge.

Suppose C contains at least two backward edges. If C is expressed as $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow (c_r = m) \rightarrow c_1$ where $m = \max\{c\ \ c \text{ is a node in } C\}$ then $c_r \rightarrow c_1$ is a backward edge and $c_{r-1} \rightarrow c_r$ is a forward edge. Since there is at least one other backward edge in C, there is a maximum index p, $1 < p < r$, such that $c_{p-1} \rightarrow c_p$ is a backward edge. By lemma 3.1, the existence of the three nodes c_{p-1}, c_p and c_r and the forward path $c_p \rightarrow c_{p+1} \rightarrow \ldots \rightarrow c_{r-1} \rightarrow c_r$ implies that D is not FLR ordered. Since this is a contradiction, C has only one backward edge. ■

3.4 Local and global inheritance in UL factorizations

Let A be a matrix of order n. Then by an analog of theorem 1.1 of [15], A has a unique UL factorization if and only if

$$\det A\,[\{n-k+1\ n-k+2\ \ldots\ n\}] \neq 0 \text{ for } k = 1\ 2\ \ldots\ n-1 \qquad (3.1)$$

Let D be a digraph on n nodes. Then $\mathbf{A}^D$ denotes the set of all $n \times n$ matrices A such that (3.1) is true and $D(A)$ is a subdigraph of D.

Local inheritance for UL factorizations is considered first. Let A be a matrix of order n and let $1 \leq i\ j \leq n$. Then A is said to be $(i\ j)$ *upper restricted* if j is not reachable from i through $\{\max\{i, j\}+1\ \max\{i\ j\}+2\ \ldots\ n\}$ in $D(A)$. In such a case the natural ordering on $D(A)$ is said to be an $(i\ j)$ *upper restricted ordering*. The following analog of theorem 3.1 is then true.

Theorem 3.3: (see [15], section 5) Let D be a digraph on n nodes and let $1 \leqslant i \leqslant j \leqslant n$ $[1 \leqslant j \leqslant i \leqslant n]$. Then for all $A \in \mathbf{A}^D$, $u_{i,j} = a_{i,j}$ $[l_{i,j} = a_{i,j}]$ in the right [left] unit UL factorization of A if and only if either

(i) A is (i, j) upper restricted, or

(ii) if j is reachable from i through nodes $p_1, p_2, \ldots, p_t \in \{j+1, [i+1], \ldots, n\}$, then $\det A[\{j+1, [i+1], \ldots, n\} - \{p_1, \ldots, p_t\}] = 0$ for all $A \in \mathbf{A}^D$.

The global inheritance problem for UL factorizations is also completely analogous to the LU case. Let A be a matrix of order n. Then A is said to be *forward* [*backward*] *upper restricted* if it is (i, j) upper restricted for all i, j such that $1 \leqslant i < j \leqslant n$ $[1 \leqslant j < i \leqslant n]$. In such a case the natural ordering on D (A) is said to be a *forward* [*backward*] *upper restricted ordering*. In either usage the term *forward* [*backward*] *upper restricted* is hereafter abbreviated to FUR [BUR]. The result analogous to theorem 3.2 is the following theorem.

Theorem 3.4: (see [15], section 5) Let D be a naturally ordered digraph on the node set $V = \{1, 2, \ldots, n\}$. Then, for all $A \in \mathbf{A}^D$ and for all pairs i, j such that $1 \leqslant i < j \leqslant n$ $[1 \leqslant j < i \leqslant n]$, $u_{i,j} = a_{i,j}$ $[l_{i,j} = a_{i,j}]$ in the right [left] unit UL factorization of A if and only if D is FUR [BUR] ordered.

Corollary 3.2: Let D be a naturally ordered digraph on the node set $V = \{1, 2, \ldots, n\}$. Then, for all $A \in \mathbf{A}^D$ and for all pairs i, j such that $1 \leqslant i \leqslant j \leqslant n$ $[1 \leqslant j \leqslant i \leqslant n]$, $u_{i,j} = a_{i,j}$ $[l_{i,j} = a_{i,j}]$ in the right [left] unit UL factorization of A if and only if D is (i, j) upper restricted for all pairs $i \leqslant j$ $[j \leqslant i]$.

The comments following corollary 3.1 also apply in this case. That is to say if A satisfies the conditions of corollary 3.2, then $\det A = \prod_{i=1}^{n} a_{i,i}$ and A is essentially triangular. Finally the lemmas of section 3.3 also have the following obvious analogs.

Lemma 3.3: Let $D = (V\ E\ \alpha)$ be an ordered digraph. Then D is not FUR [BUR] ordered if and only if there are three distinct nodes i, j and $k \in V$ such that $\alpha^{-1}(j) < \alpha^{-1}(i) < \alpha^{-1}(k)$, $k \rightarrow i$ $[i \rightarrow k]$ is in E and there is a forward [backward] path not containing i from j to k $[k$ to $j]$.

Lemma 3.4: Let $D = (V\ E\ \alpha)$ be an FUR [BUR] ordered digraph and let C be an arbitrary simple cycle in D. Then C contains precisely one backward [forward] edge.

3.5 Inheritance and sparse matrix analysis

For the purposes of this thesis we follow [6] among others and define a matrix to be *sparse* if there is an advantage to be gained by exploiting its zero elements. The term *sparsity* or *sparsity pattern* is used to describe the pattern of zero and nonzero elements in a matrix. A nonzero element or a zero element that is explicitly stored is referred to as an *entry*.

Sparse matrix analysis is concerned with finding ways to reduce the computational effort and storage requirements associated with performing matrix operations on sparse matrices. Of particular interest to this thesis is the solution of sparse systems of equations. It is often the case that the solution of very large systems is prohibitively expensive without making good use of sparsity.

In the remainder of this section a brief survey of some of the topics central to sparse matrix analysis is given and inheritance issues are mentioned where applicable.

3.5.1 Sparse matrix representations

There are two goals of any useful scheme for representing a sparse matrix. The first goal is to represent the matrix using minimum space so that any matrix element can be accessed. The second goal is to represent the matrix so that some required set of matrix operations can be performed efficiently. These goals may conflict.

There are a large number of representations in use which satisfy one or both of these goals to varying degrees. Often no single representation is the best at all stages in a

computation so the maximum benefit can be achieved by using several different representations. Brief descriptions of some of the more common representations are given below.

Coordinate The entries are stored as triples ($a_{i,j}$ i j). This has the advantages of simplicity, efficiency of storage and easy extensibility, provided sufficient unused space is included in the data structures. The disadvantage is the difficulty of accessing an arbitrary matrix element, which is particularly evident if access by rows or columns is required. This representation is well suited for representing user supplied data to a program.

Row-wise The entries are stored by rows as pairs ($a_{i,j}$ j). Also included are two integer arrays, indexed by the row number, which store the index of the start of each row's entries and the number of entries in the row. The advantages of this representation are efficient storage and easy access by rows. The disadvantages are difficulty of accessing entries by columns and difficulty of adding additional entries. A *Column-wise* representation is defined similarly. If efficient access by columns as well as rows is required, then Gustavson [11] suggests storing both the matrix and its transpose in row-wise format. The row-wise representation is best suited to static situations where it is known that no new entries will have to be added (possibly because some zeros at points of fill-in have been stored explicitly).

Row-linked The entries are stored as triples ($a_{i,j}$ j *ILINK*) where *ILINK* is either a pointer to the next entry in the row, or nil. Also stored is an array indexed by row number and containing pointers to the start of the rows' entries. The advantages of this representation are easy access by rows and easy insertion of new entries. The disadvantages are extra storage for the *ILINK* pointers and difficulty in accessing entries by columns. A *column-linked* representation is defined similarly. If efficient access by columns as well as rows is required then either of two methods can be used depending on whether storage or computational efficiency is most critical. For computational efficiency, 5-tuples ($a_{i,j}$, i j *ILINK* *JLINK*) are stored together with arrays of pointers to the start of each row's and each column's entries. If storage is most critical, then Curtis and Reid [4] suggest storing triples ($a_{i,j}$, *ILINK* *JLINK*) together with arrays of pointers to the start of each row's and each column's entries. The negative of each row number (column number) is used to indicate the null pointer, so finding the row (column) index involves following pointers until the end

of the row (column) is found. Linked representations are well-suited to dynamic situations where it is unknown how many new entries may have to be inserted.

Band. Let β be the smallest integer such that $a_{i,j} = 0$ whenever $|i - j| > \beta$. Then β is called the *semibandwidth* and $2\beta + 1$ is called the *bandwidth* of A. In the band representation the main diagonal and the nearest β subdiagonals on either side are stored as vectors of length n. The advantages are simple computations, no storage for pointers and no fill-in outside of the band when GE is executed with pivots chosen from the diagonal is executed. The disadvantages are that the storage may be inefficient if many zeros lie within the band and GE using off-diagonal pivots can double the width of the band in the factored form. This representation is best suited to matrices for which the entries are clustered close to the main diagonal.

Envelope. The main diagonal is stored as a vector of length n, the strict lower triangular part is stored in row-wise format, where all elements from the first nonzero in a row up to (not including) the diagonal are stored as entries, and the strict upper triangular part is similarly stored column-wise (unless the matrix is symmetric, in which case only one triangular part is stored). For a combinatorially symmetric matrix, the *profile* is the number of elements from the strict lower triangular part that are stored. The advantages, disadvantages and applicability of this representation are the same as in the band method. The difference is that the envelope method usually requires less storage for a given matrix and therefore has wider applicability.

Block forms. For very large matrices it may be impractical or impossible to store the complete matrix in memory, regardless of the representation used. For these matrices a partition of the matrix is found and the blocks are stored separately in auxilliary storage together with information on how they fit together. The blocks are read in and out as required. For extremely large systems this process can be iterated, so that the blocks themselves are partitioned, to any required depth. Additionally, smaller matrices may be permuted into a form for which a block representation has computational advantages. The advantages of a block form are that matrices can be accommodated which might otherwise be too large, different representations can be used for the blocks according to their structure and position in the matrix, and computation may be more efficient. The disadvantage is the overhead required to manage the partition.

3.5.2 Analyze - factor - solve decomposition

A decomposition that has proved useful for producing efficient modular software for solving sparse systems of equations recognizes three phases in the solution process (see [6] p. 95 for instance).

Analyze The matrix is analyzed to determine a good sequence of pivots and to prepare data structures for use in the factorization. Depending on the sophistication of the analysis procedures, one of several choices for representations and/or factorization methods may be chosen. The input matrix is then transferred to the internal data structure expected by the factorization method.

Factor An LU factorization is computed based on the results of the analysis phase. Other functions of this phase may include estimating the condition number of the matrix (see e.g. [24] p. 190) and monitoring stability.

Solve The LU factorization is used to produce a solution, if one exists.

There may be a computational advantage in adhering to this structure. If several systems are to be solved using matrices with identical sparsity patterns, then a single analysis phase may be sufficient for all of them. An example of this occurs when a system of non-linear equations is solved by Newton's method and A is the Jacobian of the system.

If several systems with the same matrix but different right hand side vectors are to be solved, then a single factorization suffices. An example of this occurs when iterative refinement is used to improve an approximate solution.

Although the factorization and solution phases have their own interesting subtleties, the remainder of this section will be concerned, for the most part, with the analysis phase.

3.5.3 Permutation to block triangular form

Consider a system $A\,\mathbf{x} = \mathbf{b}$ where A is block lower triangular with nonsingular square diagonal blocks.

$$A = \begin{bmatrix} A_{11} & 0 & & 0 \\ A_{21} & A_{22} & & 0 \\ & & & \\ A_{r1} & A_{r2} & & A_{rr} \end{bmatrix}$$

If **b** and **x** are partitioned conformally with A then the solution to the system can be found by a formal sequence of forward substitutions $A_{11}x_1 = b_1$ is solved for x_1 and then for $k = 2, 3, \ldots, r$ the systems $A_{kk}x_k = b_k - \sum_{j=1}^{k-1} A_{kj}x_j$ are solved in succession for x_k.

Only the diagonal blocks need to be factored since the off-diagonal blocks are used only as multipliers. This has two main effects. The large factorization has been reduced to a sequence of smaller problems which are, in general, easier to solve. Also, static storage can be used for the off-diagonal blocks so some storage efficiency may result.

If A is an arbitrary nonsingular matrix, then the block lower triangular form can be computed in time $O(n\tau)$ where τ is the number of *entries* (nonzeros or explicitly stored zeros) in A. The algorithm operates in three steps. In the first step an asymmetric permutation is found which results in a matrix with no zeros on the diagonal. This is the problem of finding a transversal and can be done in $O(n\tau)$ time (see [12] for instance). There is a more complicated algorithm due to Hopcroft and Karp [13] which runs in $O(n^{1/2}\tau)$ but often runs more slowly in practice than the $O(n\tau)$ algorithm so is not usually used. In the second step, the strongly connected components of the digraph of the permuted matrix are found in time $O(n + \tau)$ using an algorithm of Tarjan [25]. In the third step a symmetric permutation is determined such that all nodes in a strong component are consecutively numbered and such that the resulting matrix has the required block lower triangular form. This takes time $O(n + \tau)$ so the entire algorithm takes time $O(n\tau)$.

The diagonal blocks are irreducible. This means that, whenever it is convenient, it can be assumed, without loss of generality, that a coefficient matrix is irreducible or equivalently that its digraph is strongly connected. (See algorithm 4.3 and section 5.2.1 for an example of this kind of situation.)

3.5.4 Ordering to minimize fill-in

The decision problem associated with finding an ordering such that the permuted matrix suffers minimum fill-in during GE has been shown to be NP-complete (see [23] in the asymmetric case [26] in the symmetric case) It is therefore likely that the only efficient algorithms for producing orderings with relatively little fill-in will be heuristic ones

The most widely applicable heuristic algorithms are based on a simple ordering scheme known as the Markowitz criterion [17] Suppose $k \geqslant 0$ steps of GE have been executed using the Markowitz criterion to choose pivots For each *entry* $a_{ij}^{(k)}$ in $\hat{A}_k$, the active trailing $n-k \times n-k$ principal submatrix (see 2 2 1 3) define two integers $r(a_{ij}^{(k)})$ and $c(a_{ij}^{(k)})$ to be the number of *entries* other than $a_{ij}^{(k)}$ in the same respective row and column in $\hat{A}_k$ Then choose as the next pivot an *entry* in $\hat{A}_k$ that minimizes the product $r(a_{ij}^{(k)})\, c(a_{ij}^{(k)})$

In the symmetric case the Markowitz criterion is the same as the minimum degree criterion (see e g [9])

The algorithm described above does not take stability concerns into account See section 3 5 6 for a discussion of how stability can be incorporated into the Markowitz scheme

3.5.5 Ordering to minimize bandwidth or profile

Bandwidth and profile are concepts normally applied to combinatorially symmetric matrices so that restriction is assumed The decision problem associated with finding a permutation of a symmetric matrix such that the permuted matrix has minimum bandwidth has been shown to be NP-complete (see [20]) As with fill-in heuristic algorithms are used in practice

A simple and effective algorithm for reducing the bandwidth is the Cuthill-McKee algorithm (see [5]) which is applied to the graph of the matrix The algorithm operates as follows A node is selected and numbered 1 One choice which works fairly well is a node of minimum degree but a better choice is a pseudo-peripheral node (see [10], [7] or [21] p 99) Then for i from 1 to n number the unnumbered nodes adjacent to the node numbered i in increasing order of degree Assuming the existence of a starting node the complexity is

$O(de)$ where d is the maximum degree of any node (see [9] p. 60). A pseudo-peripheral node can be found in time $O(e\sqrt{n})$ (see [19]). The method for finding pseudo-peripheral nodes in [8] is more expensive in the worst case, but tends to be faster in practice so is more widely used.

Although the Cuthill-McKee method is also effective for reducing the profile of the envelope of a matrix, reversing the ordering causes no increase in the profile and frequently reduces it further (see [16]). The resulting algorithm is known as the Reverse Cuthill-McKee algorithm.

3.5.6 Stability considerations

The algorithms that have been described for reducing fill-in, bandwidth and profile are too simple to be practical since pivoting for stability has not been considered. Since it is unlikely that a pivot chosen for sparsity is the same pivot that would be chosen for stability, there is a conflict. The conflict is often resolved by the technique of *threshold pivoting*.

Suppose that following step k of GE, a pivot is to be selected from some set S based on a criterion other than stability. Then threshold pivoting restricts selection to the subset

$$\hat{S} = \{ a_{ij}^{(k)} \in S \mid |a_{ij}^{(k)}| \geq t \max_{S} |a_{rs}^{(k)}| \}$$

where $0 < t \leq 1$ is a constant called the *threshold parameter*. A good empirical value for the threshold parameter is $t = 0.1$ (see [6], p.139).

3.5.7 Inheritance in the analysis phase

Suppose a matrix A is FLR ordered. When A is factored as $A = LU$, the off-diagonal elements of U are inherited from A and need not be computed. Therefore close to half of the computations normally required in a factorization need not be done. The study of inheritance has the potential of providing another tool to sparse matrix analysis.

It seems unlikely that ordering for *global* inheritance can be a practical tool in sparse matrix analysis since the property is almost certainly rare in real problems. For instance, it can easily be shown that any combinatorially symmetric matrix whose graph contains a

cycle of length greater than 2 is not FLR orderable. Nevertheless, global inheritance concepts are still of theoretical interest and account for most of the research done for this thesis.

Of more practical interest would be ordering so that inheritance occurred in some leading principal submatrix. Apart from inheritance in a principal submatrix, it would be useful, for example, if all entries were inherited for $a_{i,j}$ with $j > i+p$ for some fixed p.

There are a number of points which might tend to limit the usefulness of results concerning sub-global inheritance. One is the need to maintain stability. Another is conflict with orderings for other purposes such as reducing fill-in, bandwidth or profile.

CHAPTER 4

FLR recognition, ordering and characterization

Chapter 4 contains the principal results of this thesis. Three general problems are considered. The first problem is to find an efficient algorithm to decide if a given ordering on an arbitrary digraph is an FLR ordering. The second problem is to find an efficient algorithm to compute an FLR ordering on an arbitrary digraph, when one exists, or to determine that such an ordering does not exist. The third problem is to characterize the set of FLR orderable digraphs. The first problem will be referred to as the *recognition* problem, the second as the *ordering* problem, and the third as the *characterization* problem.

The recognition problem for arbitrary digraphs is solved in section 4.1.

The ordering problem for trees is solved in section 4.2, by giving a characterization result.

In section 4.3 the ordering problem for strongly connected digraphs is solved (and this is extended to arbitrary digraphs in chapter 5). An algorithm is presented which returns an FLR ordering if one exists, and detects if no FLR ordering exists.

In section 4.4 the class of strongly connected digraphs is further specialized to the class of digraphs containing a spanning tree as a subgraph. The edges that may be added to an invariantly ordered tree so as to preserve the FLR property are classified and the class of maximal FLR ordered digraphs is characterized. An algorithm is presented which solves the ordering problem for digraphs containing a spanning tree.

In section 4.5 some additional work on the characterization problem is presented.

Throughout this chapter, it is assumed that all digraphs are loop-free. As noted in section 3.3, loops are irrelevant to the study of FLR ordered digraphs.

4.1 Recognition algorithm for FLR ordered digraphs

In this section the recognition problem for arbitrary ordered digraphs is solved. Algorithm 4.1 provides a simple polynomial time solution to this problem.

Algorithm 4.1 is based on lemma 3.1 which implies the existence of a particular kind of path when an ordered digraph is not FLR ordered. Specifically, there must be a backward edge $i \rightarrow j$ and a forward path, from j, that *jumps* i. This suggests decomposing the algorithm into two procedures.

The first procedure computes H, the set of all pairs (i, j), such that i precedes j in the ordering and there is a forward path from i that jumps j.

The second procedure intersects H with $\{(i, j) \mid j \rightarrow i$ *is a backward edge* $\}$. The ordering is an FLR ordering if and only if this intersection is empty.

Algorithm 4.1: FLR recognition algorithm

Input. An ordered digraph $G = (V, E, \alpha)$. It is assumed that $V = \{1, 2, \ldots, n\}$ and α is the identity function.

Output. The algorithm returns *TRUE* if G is FLR ordered and returns *FALSE* otherwise.

Data Structures. $H[i, j]$, defined for all i and j such that $1 \leqslant i < j \leqslant n-1$, is a Boolean-valued structure representing the set H defined above. It is assigned the value 1 if a forward path is found from i that jumps j. Otherwise it retains its initial value of 0. *FADJ* and *BADJ* are, respectively, the forward and backward adjacency lists of G (see 2.1.3.2). They are sorted so that *FADJ* $[i]$ and *BADJ* $[i]$ are in ascending order for all i.

Algorithm.

(1) **for** i from 1 to n **do begin**

(2) compute *FADJ* $[i]$ and *BADJ* $[i]$ **end**

(3) **for** i from 1 to $n-2$ **do begin**

(4) **for** k from $i+1$ to $n-1$ **do begin**

(5) $H[i\ k] \leftarrow 0$ **end end**

(6) **for** i from $n-2$ down to 1 **do begin**

(7) **for** each $j \in FADJ[i]$ **do begin**

(8) **for** k from $i+1$ to $n-1$ **do begin**

(9) **if** $i+1 \leqslant k < j$ **then** $H[i\ k] \leftarrow 1$

(10) **else if** $j < k \leqslant n-1$ **then** $H[i\ k] \leftarrow H[i\ k] \vee H[j\ k]$

end end end

(11) **for** j from 2 to $n-1$ **do begin**

(12) **for** each $i \in BADJ[j]$ **do begin**

(13) **if** $H[i\ j] = 1$ **then return** *FALSE* and stop **end end**

(14) **return** *TRUE* and stop

Theorem 4.1: Algorithm 4.1 correctly decides if an ordered digraph is FLR ordered.

Proof It is first shown that the algorithm correctly computes the Boolean values of $H[i\ j]$ for $i < j$. The proof is a variant of induction in which the base case is $i = n-2$ and the inductive step has the form i implies $i-1$.

For the base case $H[n-2\ n-1] \leftarrow 0$ in line 5 and is set to 1 in line 9 if and only if $(n-2) \rightarrow n$ is an edge in E. This is the correct value. Since the algorithm never resets an H value to 0, the base case is correctly computed.

For the inductive hypothesis, assume that for some i, $1 < i \leqslant n-2$, and for all r, $i \leqslant r \leqslant n-2$, $H[r\ j]$ is correctly computed for all j, $r < j \leqslant n-1$.

For the inductive step, consider the case for $i-1$. Let $i-1 < k \leqslant n-1$ and suppose $FADJ[i-1] = \{j_1, j_2, \ldots, j_t\}$ where $j_s < j_{s+1}$ for all s such that $1 \leqslant s < t$ if $t > 1$. Then to prove $H[i-1\ k]$ is correctly computed, there are three cases to consider based on the size of k with respect to the nodes in $FADJ[i-1]$.

Case 1 $i - 1 < k < j_1$ In this case $H[i-1\ k] = 1$ and is correctly set to this value in line 9 when $j = j_1$

Case 2 $t > 1$ and $j_s \leqslant k < j_{s+1}$ In this case $H[i-1\ k] = 1$ and is correctly set to this value in line 9 when $j = j_{s+1}$

Case 3 $j_t \leqslant k \leqslant n-1$ In this case since any forward path from $i-1$ must pass through some j_s it follows that $H[i-1\ k] = 1$ if and only if there is a forward path from some j_s that jumps k But this is the case if and only if $j_s < k$ and $H[j_s\ k] = 1$ and by the induction assumption this value is correctly computed It follows that $H[i-1\ k]$ is correctly computed in some execution of line 10

This proves that the values of $H[i\ j]$ are correctly computed By lemma 3 1 a digraph is not FLR ordered if and only if there exists a backward edge $j \rightarrow i$ and a forward path from i that jumps j But this is true if and only if $i \in BADJ[j]$ and $H[i\ j] = 1$ in which case the algorithm returns *FALSE* in line 13 Since it returns *TRUE* if no such nodes i and j exist the algorithm is correct ■

Theorem 4.2 : The running time for Algorithm 4 1 is $O(ne + n^2)$ where n is the number of nodes and e the number of edges in the input digraph The running time for a digraph with no isolated nodes is $O(ne)$

Proof Lines 1 and 2 require $O(n + e)$ time (see 2 1 3 2) lines 2 to 5 require $O(n^2)$ time and lines 11 to 13 require $O(e)$ time This leaves lines 6 to 10 to consider Consider a single edge $i \rightarrow j$ where $j \in FADJ[i]$ $O(n)$ elements of H are adjusted for this edge requiring $O(1)$ time for each adjustment The time to process each edge is therefore $O(n)$ Since there are $O(e)$ such edges the total time spent in lines 6 to 10 is $O(ne)$ The running time for the algorithm is therefore $O(ne + n^2)$

Since $e \geqslant n/2$ if there are no isolated nodes in the digraph the running time simplifies to $O(ne)$ in this case ■

4.2 Characterization of the FLR orderings of a tree

This section is concerned with solving the ordering problem for trees by proving a characterization result. The equivalence of FLR orderings, invariant orderings, and minimum degree orderings for trees is proved in theorem 4.3. This obviates the need to construct a new algorithm since the minimum degree algorithm can be used to produce FLR orderings on trees. Nevertheless, another algorithm is described which is more useful than the minimum degree algorithm under certain circumstances.

Theorem 4.3 : Let $T = (V, E, \alpha)$ be an ordered tree. Then the following conditions on α are equivalent.

(1) α is an FLR ordering on T.

(2) α is an invariant ordering on T.

(3) α is a minimum degree ordering on T.

Proof The equivalence of (1) and (2) is proved in [15, thm. 3.3].

(2) implies (3). Suppose α is an invariant ordering on T. It will be proved by induction on n that α is a minimum degree ordering on T. The base case for $n = 1$ is obviously true. Assume that $n > 1$ and for all trees on $n-1$ or fewer nodes, an invariant ordering is also a minimum degree ordering. Since α is an invariant ordering there is precisely one node in V that is adjacent to $\alpha(1)$. Thus $\beta : \{1, 2, \ldots, n-1\} \rightarrow V - \alpha(1)$ defined by $\beta(i) = \alpha(i+1)$ is an invariant ordering on $T_1 = T - \alpha(1)$, and by the induction hypothesis, β is also a minimum degree ordering on T_1. It is now clear that α is a possible result of applying the algorithm MINDEG (see 2.1.3.4) to T, by selecting $\alpha(1)$ as the first node to be numbered in step 1 of this algorithm.

(3) implies (2). Suppose α is a minimum degree ordering on T. When a node u is numbered by MINDEG it is pendant in the subdigraph of the current recursive call. This means that at most one node adjacent to u is numbered after u. Therefore α is an invariant ordering on T. ■

If T is a forest but not a tree, then not all of theorem 4.3 remains true. The equivalence of conditions (1) and (2) is still true as is the implication (3) implies (2). The implication (2) implies (3) is not true, however. For example, consider $T = (V, E, \alpha)$ where $V = \{1, 2, 3, 4\}$, $E = \{1 \rightarrow 4, 4 \rightarrow 1, 2 \rightarrow 3, 3 \rightarrow 2\}$ and α is the identity function. α

is clearly an invariant ordering but is not a minimum degree ordering. After numbering node 1 and removing it from T, node 4 has least degree and should be numbered next.

The minimum degree algorithm numbers the nodes in a rooted tree in a manner which is usually, but not always, bottom-up. There is also a top-down algorithm for producing invariant orderings which is more convenient in some applications, most notably algorithm 4.4.

This top-down algorithm operates as follows. Suppose a tree T and a root r in T are input. Then r is numbered with n and the remaining nodes in T are numbered in decreasing order from $n-1$ to 1 as they are encountered in a depth-first traversal of T from r.

An implementation would most likely use a stack to keep track of the numbering. Initially the stack contains the node r only. A loop is then executed in which the top entry on the stack is popped and numbered, after which its children, if any, are pushed onto the stack. The loop terminates if the stack is empty at the beginning of some iteration. Since a node receives a larger number than any of its children, it is adjacent to at most one larger numbered node. Consequently the ordering produced is an invariant ordering.

The algorithm works regardless of the node chosen to be the root of the tree and regardless of the order in which the children of a node are pushed onto the stack. See section 4.4 for further properties of FLR orderings on trees.

4.3 FLR orderings on strongly connected digraphs

The ordering problem for strongly connected digraphs is considered in this section. The principal result is algorithm 4.3 which solves the problem in polynomial time. The requirement of strong connectivity is vital to the proof that the algorithm terminates. The restriction to strongly connected digraphs can be circumvented in a natural way, however, the extension to arbitrary digraphs being made in chapter 5.

One of the key concepts behind algorithm 4.3 is the feasible set corresponding to an edge.

Definition 4.1 : Let $G = (V, E)$ be a digraph. For each edge $i \rightarrow j$ in E define the *feasible set* $FEAS[i, j]$ to be the set of all nodes in $V - \{i, j\}$ that are not reachable from j in the induced subdigraph $G - i$.

If $i \rightarrow j \in E$ then $FEAS[i, j]$ can be interpreted as the set of all nodes $k \in V$ for which it is *feasible* that $\alpha^{-1}(k) > \alpha^{-1}(i)$ when α is an FLR ordering on G and $\alpha^{-1}(i) > \alpha^{-1}(j)$. Feasible sets are used in algorithm 4.3 to construct acyclic digraphs from which FLR orderings can be derived. See example 4.1 later in this section.

Algorithm 4.2 computes the feasible sets of an arbitrary digraph. It uses a simple method which is adapted from a digraph exploration algorithm in [18], pp. 18-19.

Algorithm 4.2 : An algorithm to compute the feasible sets of a digraph

Input. A digraph $G = (V, E)$ with adjacency list structure ADJ, where $V = \{1, 2, \ldots, n\}$.

Output. A set $FEASIBLE$ of triples $(i, j, F[i, j])$, where $F[i, j]$ is the feasible set associated with edge $i \rightarrow j$.

Data Structures. $ACTIVE$ is a stack used to store a set of nodes from which outgoing edges are to be explored. $REACH$ is a Boolean array used to represent the set of nodes which are reachable from j without passing through i.

Algorithm

(1) Initialize $FEASIBLE$ to be empty.

(2) **for** each edge $i \rightarrow j$ in E **do begin**

(3) Initialize $ACTIVE \leftarrow \{j\}$, $REACH \leftarrow \{i, j\}$

(4) **while** $ACTIVE \neq \emptyset$ **do begin**

(5) pop node v from $ACTIVE$;

(6) **for** each $w \in ADJ[v]$ **do begin**

(7) **if** $w \notin REACH$ **then begin**

(8) add w to *REACH*

(9) push w onto *ACTIVE*

end end end

(10) $F[i, j] \leftarrow V - REACH$

(11) add $(i, j, F[i, j])$ to *FEASIBLE* **end**

(12) **return** *FEASIBLE*

Theorem 4.4 : Algorithm 4.2 correctly computes the feasible sets of the input digraph.

Proof To prove correctness, it must be shown that for each edge $i \rightarrow j$, the corresponding feasible set $FEAS[i, j]$ is identical to the set $F[i, j]$ computed in line 10.

Suppose $v \notin FEAS[i, j]$, and let P_v be a shortest path not passing through i from j to v. A simple induction on the length of P_v shows that v is added to *REACH* in some execution of line 8. Therefore $v \notin F[i, j]$. This proves that $F[i, j] \subseteq FEAS[i, j]$.

Conversely, if $v \in FEAS[i, j]$, then any path from j to v passes through i. Since $i \in REACH$, the test in line 7 fails at some point on every path from j to v. Therefore v is never added to *REACH*, so $v \in F[i, j]$. ■

Theorem 4.5 : Algorithm 4.2 runs in time $O(ne + e^2)$ where e is the number of edges and n is the number of nodes in the input digraph. If the digraph has no isolated nodes, then the running time is $O(e^2)$.

Proof (cf. [18], pp. 18-19) Consider the time required to find $F[i, j]$ for an arbitrary edge $i \rightarrow j$. The initialization in line 3 takes $O(n)$ time. Popping and pushing are $O(1)$ operations on *ACTIVE*, as are testing for membership and changing an entry in *REACH*. Consequently any of lines 7, 8, and 9 is $O(1)$ each time it is executed. Since each of these lines is executed at most once for each edge in the digraph, the total time expended in it is $O(e)$. $O(n)$ time is required by line 5 for popping nodes from *ACTIVE*. Finally $O(n)$ time is required in lines 10 and 11 to compute $F[i, j]$. The time to compute the feasible set for a single edge is therefore $O(n + e)$. Since one feasible set is constructed for each edge in the digraph, the total time for the entire algorithm is $O(ne + e^2)$. If G has no isolated nodes, then $e \geq n/2$, so the time bound simplifies to $O(e^2)$. ■

Algorithm 4.2 is used in a pre-processing step for algorithm 4.3. Although the implementation envisioned does not require that the sets $FEAS[i, j]$ be sorted lists within $FEASIBLE$, the method of computing $FEAS[i, j]$ from $REACH$ allows such sorting at no extra cost. Consequently the feasible sets computed by algorithm 4.2 are assumed to be sorted.

As another preliminary to algorithm 4.3, a connection between FLR orderings on a digraph and topological sorts of related acyclic digraphs is established. It is shown how the feasible sets of a digraph G (for which an FLR ordering exists) can be used to construct an acyclic digraph for which any topological sort is an FLR ordering of G; this is the basis for the ordering algorithm.

Definition 4.2 : Let $G = (V, E)$ be a digraph, let $P = (V, D)$ be an acyclic digraph and let $v \in V$. Denote by $FLRO_G(v)$ the set of FLR orderings α on G such that $\alpha(n) = v$, and by $TS_P(v)$ the set of topological sorts β on P such that $\beta(n) = v$. If $FLRO_G(v) \neq \emptyset$ and $FLRO_G(v) = TS_P(v)$, then P is said to be an *FLR orderer* of G *(with respect to v)*.

Lemma 4.1 Let $G = (V, E)$ be a strongly connected digraph and let $v \in V$ be such that $FLRO_G(v) \neq \emptyset$. Then, for each edge $i \rightarrow j \in E$, either $\alpha^{-1}(i) < \alpha^{-1}(j)$ for all $\alpha \in FLRO_G(v)$ or $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$.

Proof If $i \rightarrow j \in E$, then by lemma 2.1 there is a simple cycle C in G containing the edge $i \rightarrow j$. There are two cases to consider.

If v is on C, then the edge in C with tail v is a backward edge with respect to all $\alpha \in FLRO_G(v)$. By lemma 3.2, C has precisely one backward edge with respect to any FLR ordering on G. Consequently $i \rightarrow j$ is either a forward edge with respect to all $\alpha \in FLRO_G(v)$ or a backward edge with respect to all $\alpha \in FLRO_G(v)$.

If v is not on C, then by lemma 2.2 there is some node c on C such that there is a path from c to v that does not contain any node of C other than c. For each edge $r \rightarrow s$ in C, $r \neq c$, there is a path from s to v that does not pass through r. Then by lemma 3.1, $\alpha^{-1}(r) < \alpha^{-1}(s)$ for all $\alpha \in FLRO_G(v)$. Consequently node c is uniquely determined and the edge with tail c is the unique backward edge in C with respect to all $\alpha \in FLRO_G(v)$.

■

Theorem 4.6 : Let $G = (V, E)$ be a strongly connected digraph, let $v \in V$ be such that $FLRO_G(v) \neq \emptyset$ and let $P = (V, D)$ be an acyclic digraph (with the same node set as G). Then the following two statements are equivalent.

(1) $FLRO_G(v) = TS_P(v)$.

(2) For each pair of nodes j, i in V, i is reachable from j in P if and only if $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$.

Proof Suppose statement (1) is true. If i is reachable from j in P, then $\beta^{-1}(j) < \beta^{-1}(i)$ for any topological sort β on P, and consequently $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$. For the converse, suppose $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$, but i is not reachable from j in P. It is easily shown that if node i is not reachable from j in P, then there is some $\beta \in TS_P(v)$ such that $\beta^{-1}(i) < \beta^{-1}(j)$. This is a contradiction. Therefore statement (1) implies statement (2).

Suppose statement (2) is true. If $\alpha \in FLRO_G(v)$ and $j \rightarrow i$ is an edge in P, then by assumption, $\alpha^{-1}(j) < \alpha^{-1}(i)$. Consequently, α is a topological sort of P, so that $FLRO_G(v) \subseteq TS_P(v)$. To show the reverse inclusion, suppose $\beta \in TS_P(v)$, $i \rightarrow j$ is a backward edge (with respect to β) in G, and K is a forward path (with respect to β) in G from j to k. Then by lemma 4.1, either $\alpha^{-1}(i) < \alpha^{-1}(j)$ for all $\alpha \in FLRO_G(v)$ or $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$. The first case is not possible since there would then be a path in P from i to j and consequently $\beta^{-1}(i) < \beta^{-1}(j)$, a contradiction. By similar reasoning, K is a forward path with respect to all $\alpha \in FLRO_G(v)$. Since each such α is an FLR ordering, it follows that $\alpha^{-1}(k) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$. Hence there is a path from k to i in P, which implies that $\beta^{-1}(k) < \beta^{-1}(i)$. Consequently, β is an FLR ordering on G. Therefore statement (2) implies statement (1). ■

The above theorem has several implications. Firstly, an FLR orderer exists for any strongly connected FLR orderable digraph. Secondly, a digraph is an FLR orderer (wrt v) if and only if both its transitive closure and reduced subdigraph are also FLR orderers (wrt v). Thirdly, statement (2) implies the equivalence of the problem of computing an FLR orderer (wrt v) and the problem of identifying all pairs of nodes i, j such that $\alpha^{-1}(j) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$. We use this equivalence in the following discussion pertaining to the construction of FLR orderers.

Our construction of an FLR orderer (with respect to a given v), if it exists, uses the feasible sets to add edges to a *base* acyclic digraph known to be a subdigraph of the

transitive closure of every FLR orderer (wrt v). Given that $FLRO_G(v) \neq \emptyset$ the base acyclic digraph is $P_0 = (V, D_0)$ where $D_0 = \{ u \rightarrow v \mid u \in V - \{v\} \}$. The following simple example illustrates one situation in which the contents of a feasible set allow an edge to be added to the base.

Consider a feasible set $FEAS[r, s]$ of G such that $v \notin FEAS[r, s]$ and $v \neq r, s$. Then there is a path in G from s to v that does not pass through r. Therefore $\alpha^{-1}(r) < \alpha^{-1}(s)$ for every $\alpha \in FLRO_G(v)$. Consequently (see theorem 4.6) there is a path from r to s in any FLR orderer of G (wrt v) and the edge $r \rightarrow s$ can be added to P_0. In fact, this example represents the only situation in which a single feasible set in isolation determines an edge to be added to P_0. All other situations require the consideration of several feasible sets.

Lemma 4.2 will show how an FLR orderer of G can be constructed incrementally from P_0 by considering the feasible sets of G one at a time and adding the appropriate edges. The relevant information concerning previously considered feasible sets is contained in the structure of the current acyclic digraph derived from P_0. To access this information it is convenient to define three functions A_H, B_H, and U_H on the node set V of an arbitrary digraph H. Define $A_H(i) = \{ j \in V \mid$ there is a path from i to j in $H \}$, $B_H(i) = \{ j \in V \mid$ there is a path from j to i in $H \}$, and $U_H(i) = \{ j \in V \mid j \notin A_H(i)$ and $j \notin B_H(i) \}$. H is acyclic if and only if $A_H(i) \cap B_H(i) = \emptyset$ for all $i \in V$. $A_H(i)$, $B_H(i)$, and $U_H(i)$ are respectively the *Above*, *Below*, and *Unknown* sets of H with respect to i. The information accessed by A_H, B_H, and U_H is highly redundant. If either A_H or B_H is known, then the other two functions can be computed. It is, however, simpler to state lemma 4.2 and algorithm 4.3 in terms of all three functions.

Lemma 4.2: Let $G = (V, E)$ be a digraph and let $H_0 = (V, D)$ be an acyclic digraph on the same node set. Define access functions A_H, B_H and U_H for an arbitrary digraph H as above. Suppose an edge $i \rightarrow j$ in E satisfies the hypothesis of one of cases (a) to (d) below. If the action in the applicable case is taken, producing a new edge set D_1, then $H_1 = (V, D_1)$ is acyclic. Furthermore, if $H_t = (V, D_t)$ is obtained recursively from H_0 by any t applications of the cases (a) to (d) corresponding to distinct edges $i \rightarrow j \in E$, and if any one of these edges is subsequently reconsidered, then no new edges will be added to D_t. If in addition $FLRO_G(v) \neq \emptyset$ and every $\alpha \in FLRO_G(v)$ is a topological sort of H_0, then every such α is also a topological sort of H_t.

(a) If $j \in B_{H_0}(i)$, then add edge $k \rightarrow i$ to D for each $k \in U_{H_0}(i)$ such that $k \notin FEAS[i, j]$.

(b) If $j \in A_{H_0}(i)$, then make no changes to D.

(c) If $j \in U_{H_0}(i)$ and $A_{H_0}(i) \not\subseteq FEAS[i, j]$, then add edge $i \rightarrow j$ to D.

(d) If $j \in U_{H_0}(i)$, $A_{H_0}(i) \subseteq FEAS[i, j]$ and $U_{H_0}(i) - \{j\} \subseteq FEAS[i, j]$, then make no changes to D.

Proof It is first proved that H_1 is acyclic. If H_1 contains a cycle, then for every two nodes i_1 and i_2 on the cycle, $i_2 \in A_{H_1}(i_1)$ and $i_2 \in B_{H_1}(i_1)$. If the addition of an edge $k \rightarrow i$ in case (a) creates a cycle, then this edge must be on the cycle and consequently $k \in A_{H_0}(i)$. But this is a contradiction since $k \in U_{H_0}(i)$. Similarly, in case (c), if a cycle is created, then the edge $i \rightarrow j$ must be on it and $j \in B_{H_0}(i)$. But this is not possible since $j \in U_{H_0}(i)$. Therefore H_1 is acyclic.

Suppose $H_t = (V, D_t)$ is obtained from H_0 by any t applications of the cases (a) to (d) corresponding to distinct edges in E, and suppose one of these edges $i \rightarrow j$ is subsequently reconsidered.

If either of cases (a) or (b) applied originally to $i \rightarrow j$, then it still applies, since edges are never removed when constructing H_t from H_0, only added. Consequently, no new edges will be added to H_t in these cases.

If case (c) applied originally, then the edge $i \rightarrow j$ would have been added to D so that $j \in A_{H_t}(i)$. Thus case (b) applies to $i \rightarrow j$ in H_t and no new edges are added.

Suppose case (d) applied originally to $i \rightarrow j$. Then $j \in U_{H_r}(i)$, $A_{H_r}(i) \subseteq FEAS[i, j]$ and $U_{H_r}(i) - \{j\} \subseteq FEAS[i, j]$, where H_r is the acyclic

digraph constructed up to the point where the edge was first considered. If cases (b) or (d) apply to $i \rightarrow j$ (with respect to H_t) then no new edges are added. If case (a) applies then since $j \notin U_{H_t}(i)$ and $U_{H_t}(i) \subseteq U_{H_r}(i)$, $U_{H_t}(i) \subseteq FEAS[i, j]$, so no new edges are added to H_t. If case (c) applies then $j \in U_{H_t}(i)$ and $A_{H_t}(i) \not\subseteq FEAS[i, j]$. If $k \in A_{H_t}(i)$ but $k \notin FEAS[i, j]$ then $k \notin A_{H_r}(i)$ implying that either $k \in B_{H_r}(i)$ or $k \in U_{H_r}(i)$. In the former case $k \in B_{H_t}(i)$ which is a contradiction. In the latter case since $k \neq j$, $k \in FEAS[i, j]$ which is also a contradiction. Therefore case (c) does not apply to $i \rightarrow j$ in H_t. Consequently no edges are added to H_t if case (d) applied originally.

We now assume that $FLRO_G(v) \neq \emptyset$ and every $\alpha \in FLRO_G(v)$ is a topological sort of H_0, and prove that the same property holds for H_t. It is sufficient to prove the result for H_1 since a simple induction argument then proves the general case. In cases (b) and (d) no changes are made to D so there is nothing to prove.

In case (a), $j \in B_{H_0}(i)$ which implies that $\alpha^{-1}(j) < \alpha^{-1}(i)$ for any topological sort of H_0 and in particular for all $\alpha \in FLRO_G(v)$. If $k \notin FEAS[i, j]$ then $\alpha^{-1}(k) < \alpha^{-1}(i)$ for all $\alpha \in FLRO_G(v)$ (see the explanation following definition 4.1). Therefore if D_1 is produced from D by adding the edge $k \rightarrow i$ for each $k \in U_{H_0}(i)$ such that $k \notin FEAS[i, j]$ then every $\alpha \in FLRO_G(v)$ is still a topological sort of $H_1 = (V, D_1)$.

If case (c) is applicable then by similar reasoning $\alpha^{-1}(i) < \alpha^{-1}(j)$ for all $\alpha \in FLRO_G(v)$ and each such α is a topological sort of H_1. ■

The four cases (a) to (d) of the above lemma do not necessarily partition the edge set. We make the following definition.

Definition 4.3 : Let $G = (V, E)$ be a digraph, let $H = (V, D)$ be an acyclic digraph and let $i \rightarrow j \in E$. Then $i \rightarrow j$ is said to be *immediate*, with respect to H, if any of cases (a) to (d) of lemma 4.2 are applicable when the edge is considered. Otherwise $i \rightarrow j$ is said to be *deferred*, with respect to H.

If $i \rightarrow j$ is deferred with respect to H, then $j \in U_H(i)$, $A_H(i) \subseteq FEAS[i, j]$ and $U_H(i) - \{j\} \not\subseteq FEAS[i, j]$. Such edges are called *deferred* since it is impossible to make a decision regarding the relationship of i and j in the ordering until more

information is available from a consideration of other edges. Hence action is deferred. The information in a feasible set corresponding to an *immediate* edge may be used immediately however.

We wish to use lemma 4.2 to recursively construct an acyclic digraph from the base acyclic digraph P_0 by considering each edge in E. Depending on the selection order we obtain an acyclic digraph P_e and a set S of deferred edges after each edge in E has been considered once. Lemma 4.3 proves that if G is strongly connected then the edges in S are immediate if reconsidered with respect to P_e; specifically case (a) of lemma 4.2 is applicable.

Lemma 4.3 : Let $G = (V, E)$ be a strongly connected digraph, let $v \in V$, let P_0 be the base acyclic digraph (with respect to v), let $P_e = (V, D_e)$ be the acyclic digraph constructed from P_0 by taking the action in the applicable case of lemma 4.2 for each edge in E (selected in arbitrary order), and let S be the set of deferred edges (with respect to this order of selection). If an arbitrary edge $i \rightarrow j$ in S is reconsidered (with respect to P_e), then $j \in B_{P_e}(i)$.

Proof Let $i \rightarrow j$ be in S. By lemma 2.1, there is some simple cycle C in G containing this edge. If v is on C, then every edge in C is immediate when first considered (since either $A_{P_0}(i) \not\subseteq FEAS[i, j]$ or one of i and j equals v), which is a contradiction. Hence, by lemma 2.2, there is some node c on C such that there is a path from c to v not containing any other node of C. Every edge $i_1 \rightarrow i_2$ in C, except the edge with tail c, is immediate when first considered, since $A_{P_0}(i_1) \not\subseteq FEAS[i_1, i_2]$. When the action in the applicable case of lemma 4.2 is taken for any such edge (either case (b) or case (c) applies), the result is a path from i_1 to i_2 in P_e. Consequently, $c = i$, and when $i \rightarrow j$ is selected from S, $j \in B_{P_e}(i)$. ■

Lemmas 4.2 and 4.3 prove that if G is strongly connected and $FLRO_G(v) \neq \emptyset$ then a particular set of edges can be added to the base acyclic digraph P_0, producing a particular acyclic digraph P such that $FLRO_G(v) \subseteq TS_P(v)$. Theorem 4.7 proves that P is in fact an FLR orderer of G.

Theorem 4.7 : Let $G = (V\ E)$ be a strongly connected digraph, let $v \in V$ be such that $FLRO_G(v) \neq \emptyset$ and let P_0 be the base acyclic digraph (with respect to v). Suppose the acyclic digraph P_e is constructed from P_0 by the application of the applicable case of lemma 4.2 to each edge in E (selected in some arbitrary order) and S contains the set of deferred edges with respect to this order of selection. If the acyclic digraph P is constructed from P_e by reconsidering the edges in S and taking the actions specified in lemma 4.2 for each, then P is an FLR orderer of G (with respect to v).

Proof It is only necessary to show that $TS_P(v) \subseteq FLRO_G(v)$. Suppose $\beta \in TS_P(v)$ but $\beta \notin FLRO_G(v)$. Then there exist distinct nodes i, j and k in V such that $i \rightarrow j \in E$, $\beta^{-1}(j) < \beta^{-1}(i) < \beta^{-1}(k)$ and $k \notin FEAS[i\ j]$. This means that $j \notin A_P(i)$ and $k \notin B_P(i)$. By lemma 4.3, $i \rightarrow j$ is immediate either the first time it is seen or when selected from S. Denote by P_r the acyclic digraph constructed up to the time that $i \rightarrow j$ is immediate. It follows that one of four cases must be true:

(1) $j \in B_{P_r}(i)$ and $k \in U_{P_r}(i)$

(2) $j \in B_{P_r}(i)$ and $k \in A_{P_r}(i)$

(3) $j \in U_{P_r}(i)$ and $k \in U_{P_r}(i)$

(4) $j \in U_{P_r}(i)$ and $k \in A_{P_r}(i)$

If case (1) is true, then case (a) of lemma 4.2 is applicable and $k \rightarrow i$ is added to the acyclic digraph. This results in $k \in B_P(i)$, which is a contradiction.

If case (2) is true, then $\alpha^{-1}(j) < \alpha^{-1}(i) < \alpha^{-1}(k)$ for all $\alpha \in FLRO_G(v)$, since each such α is a topological sort of P. This is a contradiction since $i \rightarrow j$ is in E and $k \notin FEAS[i\ j]$.

If case (3) is true, then either case (c) or case (d) of lemma 4.2 may be applicable. Case (c) gives the contradiction that $j \in A_P(i)$, while if case (d) is applicable then k must be in $FEAS[i\ j]$, which is also a contradiction.

If case (4) is true then only case (c) of lemma 4.2 may be applicable, giving again the contradiction that $j \in A_P(i)$.

Since the assumption that $\beta \notin FLRO_G(v)$ leads to a contradiction, P is an FLR orderer of G ■

If $FLRO_G(v) = \emptyset$, then this condition is detectable as the following corollary implies.

Corollary 4.1 : Let $G = (V\ E)$ be a strongly connected digraph, let $v \in V$ and let P_0 be the base acyclic digraph (with respect to v). If P is constructed from P_0 in the same manner as in theorem 4.7, then $FLRO_G(v) = \emptyset$ if and only if there is some edge $i \rightarrow j$ such that $j \in B_{P_r}(i)$ and $A_{P_r}(i) \not\subseteq FEAS[i\ j]$, where P_r is the acyclic digraph constructed up to the time that $i \rightarrow j$ is immediate.

Proof Suppose $FLRO_G(v) = \emptyset$. If $\beta \in TS_P(v)$ then, since β is not an FLR ordering, there exist distinct nodes i, j and k in V such that $i \rightarrow j \in E$, $\beta^{-1}(j) < \beta^{-1}(i) < \beta^{-1}(k)$ and $k \notin FEAS[i\ j]$. By lemma 4.3, $i \rightarrow j$ is immediate either the first or second time it is considered. One of the four cases in the proof of theorem 4.7 must be true when $i \rightarrow j$ is immediate. Cases (1), (3), and (4) all yield contradictions regardless of whether $FLRO_G(v) \neq \emptyset$. Consequently case (2) must be true which implies that $j \in B_{P_r}(i)$ and $k \in A_{P_r}(i)$.

Conversely, suppose there is some edge $i \rightarrow j$ such that $j \in B_{P_r}(i)$ and there is some $k \in A_{P_r}(i)$ such that $k \notin FEAS[i, j]$. If $FLRO_G(v) \neq \emptyset$ then by theorem 4.7 P is an FLR orderer of G. But this leads to a contradiction since $\alpha^{-1}(j) < \alpha^{-1}(i) < \alpha^{-1}(k)$ for any $\alpha \in TS_P(v)$ and no ordering with this property can be an FLR ordering of G. ■

The ordering algorithm for strongly connected digraphs can now be presented.

Algorithm 4.3 : An FLR ordering algorithm for strongly connected digraphs

Input A strongly connected digraph $G = (V\ E)$ where $V = \{1\ 2\ \ \ \ ,n\}$.

Output An FLR ordering of G, if one exists, otherwise a message of failure.

Data Structures *FEASIBLE* is the array of triples returned by algorithm 4.2. The queue *FQUEUE* is used to store a subset of triples from *FEASIBLE*. *HIGH* stores the set of nodes which are possible high nodes for the ordering. *ADJ* is an adjacency list for an acyclic digraph P which will be an FLR orderer of G if the algorithm terminates successfully. If it is discovered that a node j must be numbered higher than i for every $\alpha \in FLRO_G(v)$, then this information is indicated by a path from i to j in P. A, B, and U are respectively the Above, Below and Unknown functions on P, the subscripts being dropped.

Sub-algorithms Algorithm 4.2 is used to compute the set *FEASIBLE*. The algorithm *TOPSORT* (see section 2.1.3.3) is used to return a topological sort of an acyclic digraph.

Algorithm

(1) Compute the set $FEASIBLE$ using algorithm 4.2.

(2) Initialize $HIGH$ to be a stack containing all of the nodes in V.

(3) **while** $HIGH \neq \emptyset$ **do begin**

(4) pop the top node v from $HIGH$. /* v is considered as the highest numbered node in the ordering */

(5) initialize $FQUEUE$ to be a queue containing all members of $FEASIBLE$.

(6) initialize an acyclic digraph P with node set V and adjacency list ADJ such that $ADJ[i] = \{v\}$ for all $i \in V - \{v\}$ and $ADJ[v] = \emptyset$.

(7) initialize $NEXT \leftarrow FALSE$.

(8) **while** $NEXT = FALSE$ and $FQUEUE \neq \emptyset$ **do begin**

(9) remove $(i, j, FEAS[i, j])$ from $FQUEUE$.

(10) **if** $j \in B(i)$ **then**

(11) **if** $A(i) \not\subseteq FEAS[i, j]$ **then**

(12) $NEXT \leftarrow TRUE$. /* No FLR ordering with v high (cor. 4.1) */

(13) **else for** each $k \in U(i)$ such that $k \notin FEAS[i, j]$ **do** add i to $ADJ[k]$. /* This is case (a) of lemma 4.2 */

endif

(14) **else if** $j \in U(i)$ **then**

(15) **if** $A(i) \not\subseteq FEAS[i, j]$ **then**

(16) add j to $ADJ[i]$. /* This is case (c) of lemma 4.2 */

(17) **else if** $U(i) - \{j\} \not\subseteq FEAS[i, j]$ **then**

(18) put $(i, j, FEAS[i, j])$ on the end of $FQUEUE$. /* $i \rightarrow j$ is a deferred edge */

endif endif endif endif

end /* while */

(19) **if** *FQUEUE* = ∅ and *NEXT* = *FALSE* **then**

(20) **return** *TOPSORT* (*P*) as the FLR ordering α and stop **endif**

end /* while */

(21) **return** "No FLR ordering exists" and stop

Theorem 4.8 : Algorithm 4.3 is correct.

Proof Since the edges added to P are precisely those specified in lemma 4.2, lemma 4.3 proves that each iteration of the loop initiated in line 8 terminates. Since at most n iterations of this inner loop are executed, the algorithm terminates.

Suppose there exists an FLR ordering on G and let v be the first node selected from *HIGH* for which $FLRO_G(v) \neq \emptyset$. By theorem 4.7, the acyclic digraph P constructed in the iteration of the inner loop corresponding to v is an FLR orderer of G. The algorithm then executes line 20 which returns a topological sort of P which is an FLR ordering of G.

If there does not exist an FLR ordering on G, then corollary 4.1 proves that line 12 is executed at some time for each choice of v from *HIGH*. Consequently *HIGH* becomes empty without line 20 being executed and the algorithm correctly executes line 21. ■

Theorem 4.9 : Algorithm 4.3 runs in time $O(n^3e)$ where n is the number of nodes and e the number of edges in the input digraph.

Proof Consider the time to process a single feasible set in lines 9 to 18. The computation of $A(i)$, $B(i)$ and $U(i)$, represented as Boolean arrays, require $O(n^2)$ time if a method similar to that used in algorithm 4.2 is used, since P may have $O(n^2)$ edges. The determination of the set to which j belongs then takes $O(1)$ time. Computing the Boolean array representing *FEAS* $[i, j]$ takes $O(n)$ time. Then determining if $A(i) \subseteq FEAS[i, j]$ or computing the set of nodes in $U(i)$ not in *FEAS* $[i, j]$ each take $O(n)$ time. At most $O(n)$ edges are added to *ADJ* for a single feasible set and, since *ADJ* does not have to be sorted, adding an edge takes $O(1)$ time. The time to process a single feasible set is therefore $O(n^2)$.

Since lemma 4.3 shows that $O(e)$ feasible sets are processed for a given choice for the high node and there are at most n high nodes the time spent in lines 9 to 18 for the entire algorithm is $O(n^3e)$.

Line 1 is executed once only and requires $O(ne + e^2)$ time. Line 2 is also executed only once and requires $O(n)$ time. Lines 4 to 7 are executed $O(n)$ times and each execution requires $O(n + e)$ time to initialize *FQUEUE* and *ADJ* for a total time of $O(n^2 + ne)$. Line 20 is executed only once and requires at most $O(n^2)$ time.

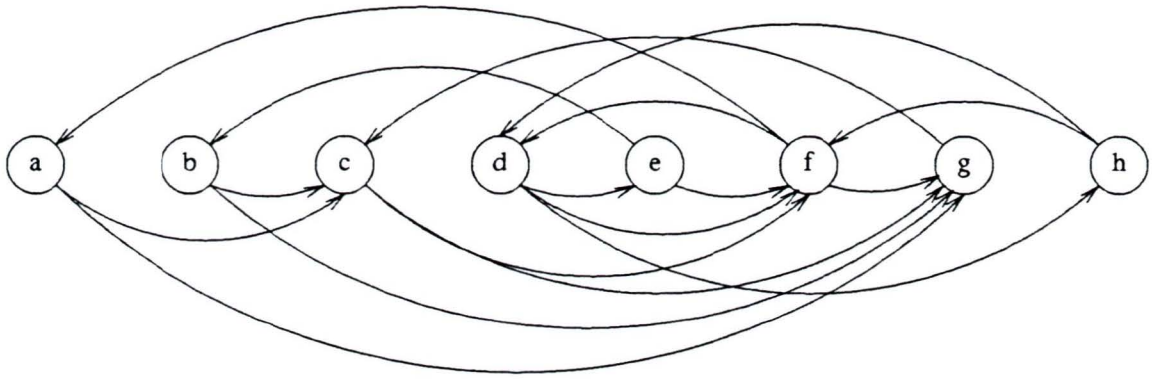
The total time for the algorithm is therefore $O(n^3e)$. ■

It should be noted that it is possible to improve the running time for algorithm 4.3 for certain input digraphs. This is most easily done by computing the maximum (undirected) spanning subgraph M of the input digraph G (see 2.1.3.5) and examining some of its properties. If M is not a forest then no FLR ordering exists for G so the message in line 21 can be output immediately.

A more subtle saving in time results if some nodes can be excluded from *HIGH* when it is initialized in line 2. The simplest situation occurs when $i \rightarrow j$ is an edge in M (so that $j \rightarrow i$ is also an edge) and $FEAS[i, j] = FEAS[j, i] = \emptyset$. Then any FLR ordering of G must have either i or j as the highest numbered node in the ordering. *HIGH* can therefore be initialized to $\{i, j\}$. Furthermore, if there are two disjoint pairs with this property then no FLR ordering can exist.

Example 4.1 : This is an example illustrating algorithm 4.3. A strongly connected 8 node digraph G is input. To avoid confusion with the numbers assigned by the algorithm, the nodes are identified using the letters a to h. Both a linear layout and the equivalent adjacency list representation of G are given below.

Linear layout of G



Adjacency lists of G

a	*c g*
b	*c g*
c	*f g*
d	*e f h*
e	*b f*
f	*a d g*
g	*c*
h	*d f*

The feasible sets corresponding to edges in G are computed using algorithm 4.2 and are listed below. The format for representing the feasible set corresponding to the edge $i \rightarrow j$ is $(i \rightarrow j \mid FEAS[i\ j])$.

Feasible sets for the edges of G

$(a \to c \mid \emptyset)$
$(a \to g \mid \emptyset)$
$(b \to c \mid \emptyset)$
$(b \to g \mid \emptyset)$
$(c \to f \mid \emptyset)$
$(c \to g \mid \{a\ b\ d\ e\ f\ h\})$
$(d \to e \mid \{h\})$
$(d \to f \mid \{b\ e\ h\})$
$(d \to h \mid \{b\ e\})$
$(e \to b \mid \emptyset)$
$(e \to f \mid \{b\})$
$(f \to a \mid \{b\ d\ e\ h\})$
$(f \to d \mid \{a\})$
$(f \to g \mid \{a\ b\ d\ e\ h\})$
$(g \to c \mid \emptyset)$
$(h \to d \mid \emptyset)$
$(h \to f \mid \emptyset)$

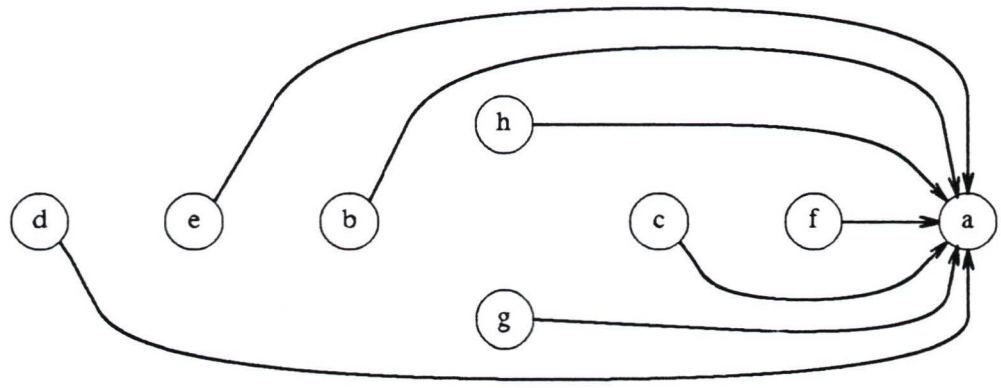
The algorithm executes a loop in which a node v is selected to be the highest numbered node in the ordering and an attempt is made to construct an FLR orderer of G (with respect to v). In doing this an acyclic digraph P with adjacency list ADJ is computed. The algorithm tends to include a lot of edges in P that become redundant at some later stage. In the interest of clarity, only the reduced subdigraph of P is retained since the algorithm works just as well if this is done.

In line 5 *FQUEUE* is initialized to contain all of the feasible sets in the order in which they are listed above.

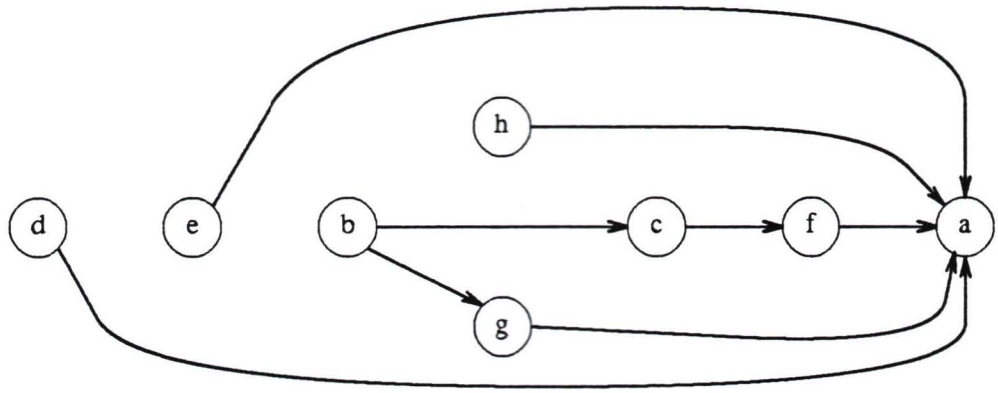
This example was chosen so that no FLR ordering exists in which node a is the high node. There is, however, an FLR ordering with b high. These two cases will be examined in detail. For each of the two cases, the initial state of the acyclic digraph P is illustrated. The feasible sets are then examined, one at a time, and the changes to P are listed with explanations. The *case* of a feasible set refers to the applicable case in lemma 4.2. At appropriate points the diagram of P is updated by adding the new edges. As mentioned above, the redundant edges are deleted so that P is always reduced.

Case when a is the high node

initial state of P

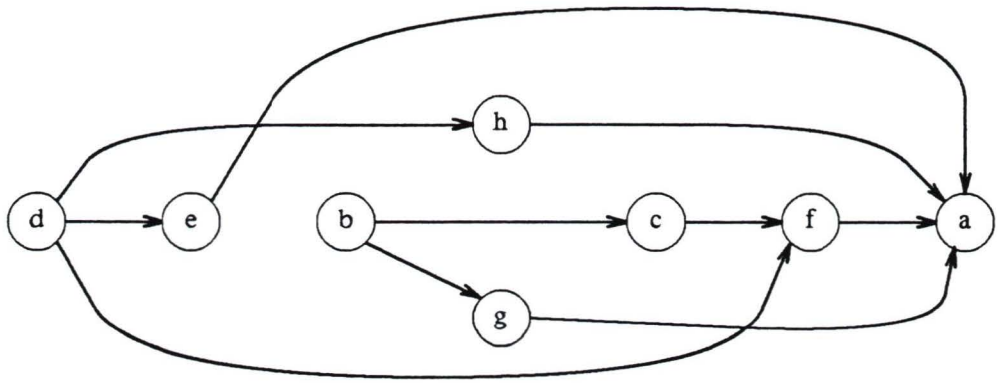


$(a \rightarrow c \mid \emptyset)$	case (a) no edges added $U(a) = \emptyset$
$(a \rightarrow g \mid \emptyset)$	case (a) no edges added $U(a) = \emptyset$
$(b \rightarrow c \mid \emptyset)$	case (c) add edge $b \rightarrow c$ to P
$(b \rightarrow g \mid \emptyset)$	case (c) add edge $b \rightarrow g$ to P
$(c \rightarrow f \mid \emptyset)$	case (c) add edge $c \rightarrow f$ to P

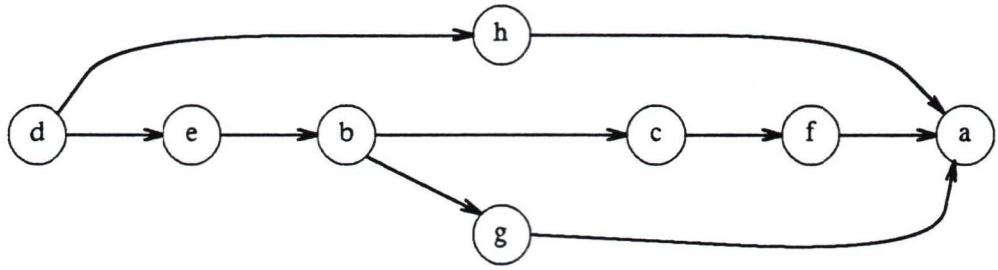


$(c \rightarrow g \mid \{a, b\ d\ e, f\ h\})$	case (d) no edges added to P
$(d \rightarrow e \mid \{h\})$	case (c) add edge $d \rightarrow e$ to P
$(d \rightarrow f \mid \{b\ e\ h\})$	case (c) add edge $d \rightarrow f$ to P

$(d \rightarrow h \mid \{b\ e\})$ case (c) add edge $d \rightarrow h$ to P



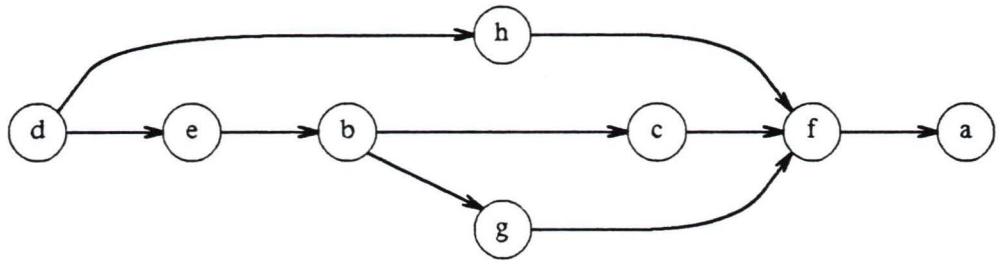
$(e \rightarrow b \mid \varnothing)$ case (c) add edge $e \rightarrow b$ to P



$(e \rightarrow f \mid \{b\})$ case (b) no edges added to P

$(f \rightarrow a \mid \{b\ d, e, h\})$ case (b) no edges added to P

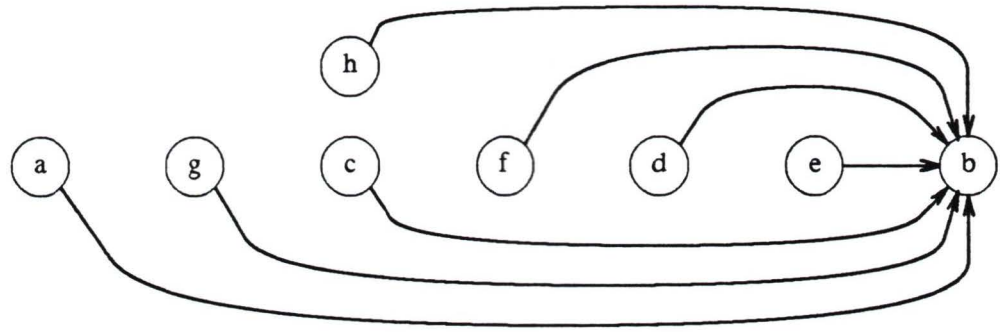
$(f \rightarrow d \mid \{a\})$ case (a) $U(f) - FEAS[f\ d] = \{g\ h\}$ add edges $g \rightarrow f$ and $h \rightarrow f$ to P



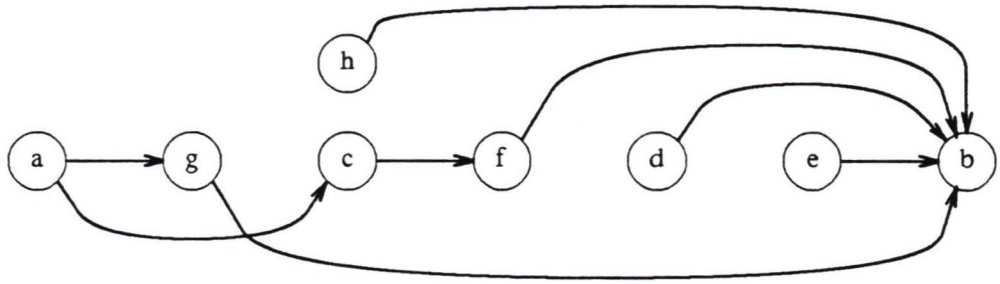
$(f \rightarrow g \mid \{a\ b\ d\ e\ h\})$	case (a) $U(f) = \emptyset$ no edges added to P
$(g \rightarrow c \mid \emptyset)$	case (c) add edge $g \rightarrow c$ to P
$(h \rightarrow d \mid \emptyset)$	$A(h) \not\subseteq FEAS[h\ d]$ so line 12 terminates this iteration and returns control to line 3 where a new high node is selected

Case when b is the high node

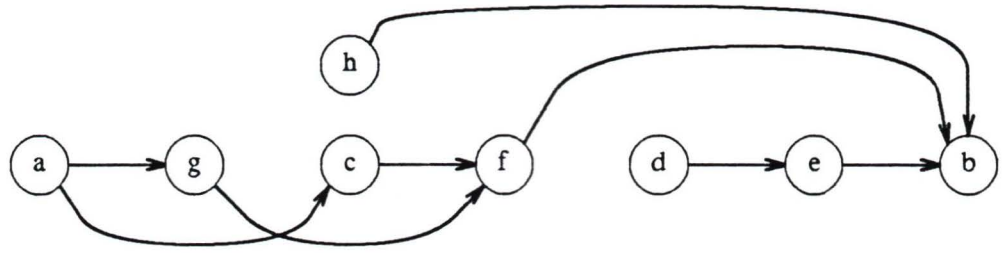
initial state of P



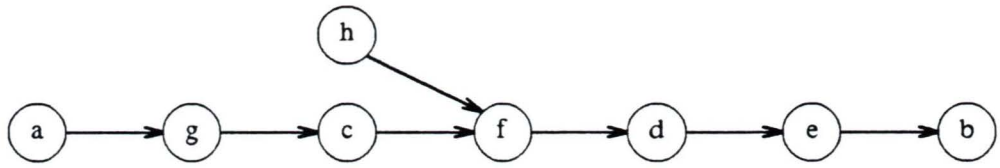
$(a \rightarrow c \mid \emptyset)$	case (c) add edge $a \rightarrow c$ to P
$(a \rightarrow g \mid \emptyset)$	case (c) add edge $a \rightarrow g$ to P
$(b \rightarrow c \mid \emptyset)$	case (a) $U(b) = \emptyset$ no edges added to P
$(b \rightarrow g \mid \emptyset)$	case (a) $U(b) = \emptyset$ no edges added to P
$(c \rightarrow f \mid \emptyset)$	case (c) add edge $c \rightarrow f$ to P



$(c \rightarrow g \mid \{a\ b\ d\ e\ f\ h\})$	case (d) no edges added to P
$(d \rightarrow e \mid \{h\})$	case (c) add edge $d \rightarrow e$ to P
$(d \rightarrow f \mid \{b\ e\ h\})$	deferred line 18
$(d \rightarrow h \mid \{b\ e\})$	deferred
$(e \rightarrow b \mid \emptyset)$	case (b) no edges added to P
$(e \rightarrow f \mid \{b\})$	deferred
$(f \rightarrow a \mid \{b\ d, e\ h\})$	case (a) $U(f) - FEAS[f\ a] = \{g\}$, add edge $g \rightarrow f$ to P



$(f \rightarrow d \mid \{a\})$	case (c) add edge $f \rightarrow d$ to P
$(f \rightarrow g \mid \{a\ b, d\ e, h\})$	case (a) $U(f) - FEAS[f\ g] = \emptyset$ no edges added to P
$(g \rightarrow c \mid \emptyset)$	case (c) add edge $g \rightarrow c$ to P
$(h \rightarrow d \mid \emptyset)$	case (c) add edge $h \rightarrow d$ to P
$(h \rightarrow f \mid \emptyset)$	case (c) add edge $h \rightarrow f$ to P

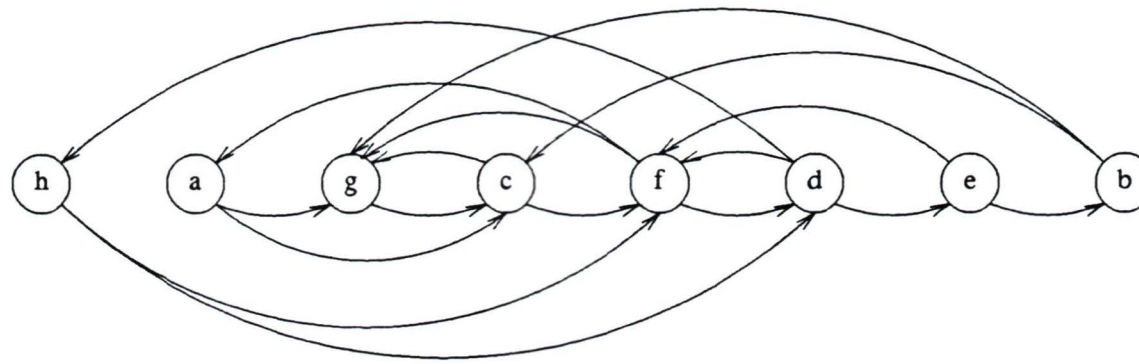


Consider the deferred feasible sets again.

$(d \rightarrow f \mid \{b\ e\ h\})$	case (a): $U(d) = \emptyset$, no edges added to P
$(d \rightarrow h \mid \{b\ e\})$	case (a): $U(d) = \emptyset$, no edges added to P
$(e \rightarrow f \mid \{b\})$	case (a): $U(e) = \emptyset$, no edges added to P

Since *FQUEUE* is now empty, a topological sort of P produces an FLR ordering of G. There are four possible FLR orderings of G, depending on the numbering of node h relative to a, g, and c. Assuming that h is numbered 1, G has the FLR ordering below.

FLR ordered linear layout of G. The FLR ordering is represented pictorially by ordering the nodes from left to right according to the numbers assigned by the algorithm.



End of example. □

4.4 Digraphs containing a spanning tree as a subgraph

The special case of a digraph having a spanning tree as a subgraph is considered in this section. Theorem 4.3 in section 4.2 established the equivalence of FLR invariant and minimum degree orderings on trees. It was also demonstrated how invariant orderings can be obtained such that an arbitrary node receives the largest number. This method for obtaining invariant orderings is exploited in algorithm 4.4 to solve the ordering problem for digraphs which contain a spanning tree.

Two other problems are also considered in this section. The first problem is to characterize the digraphs which are maximal (with respect to the edge set) FLR ordered digraphs. The second problem is to characterize the edges that may be added to an invariantly ordered tree such that the resulting digraph remains FLR ordered. It is shown that these problems are closely related. The solution to the second problem is useful in developing algorithm 4.4.

Before proceeding with the main results of this section it is convenient to state and prove two simple lemmas.

Lemma 4.4 : Let $T = (V, E, \alpha)$ be an invariantly ordered tree and suppose the node $r = \alpha(n)$ is designated as the root of T. If P is any simple path in T starting from r, then α^{-1} is a monotonically decreasing function along P. In particular, if S is a subtree of T, then the maximum of $\alpha^{-1}(s)$, for $s \in S$, is achieved when s is the root of S.

Proof. Suppose, on the contrary, that there is some path P from r to t in T such that α^{-1} is not monotonically decreasing along P. Then, without loss of generality, the node m for which $\alpha^{-1}(m)$ is minimized on P is not r or t. But then m is adjacent to two nodes on P which are numbered greater than m under α, contradicting the assumption that α is an invariant ordering.

The second statement follows immediately, since for any subtree S with root s and for any node t in S, there is a simple path from r to t that passes through s. ■

In the following lemma the concept of *maximum subgraph* as defined in section 2.1.1.2 is used. It should be noted that the maximum subgraph of a digraph is by definition a graph (undirected).

Lemma 4.5 : Let $G = (V\ E\ \alpha)$ be an FLR ordered digraph. Then the maximum subgraph H of G is a forest.

Proof. Suppose H is not a forest. Then there is a simple cycle $C = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_t \rightarrow v_1$ in H for some $t \geq 3$. Assume without loss of generality that $\alpha^{-1}(v_1)$ is the minimum of α^{-1} on the nodes in C. Since H is a graph the two paths $v_t \rightarrow v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1 \rightarrow v_t$ both exist in G. But then, since both $v_2 \rightarrow v_1$ and $v_t \rightarrow v_1$ are backward edges, one of the two paths must violate the FLR condition on G regardless of the relative numbering of v_2 and v_t. ■

Definition 4.4 : Let F be the set of all FLR ordered digraphs. Define a partial ordering on F by $(V_1\ E_1\ \alpha_1) \leqslant (V_2\ E_2\ \alpha_2)$ if $V_1 = V_2$, $\alpha_1 = \alpha_2$, and $E_1 \subseteq E_2$. A maximal element of F with respect to this partial ordering is called a *maximal forward lower restricted* (*MFLR*) ordered digraph.

Theorem 4.10 : Let $G = (V\ E\ \alpha)$ be a strongly connected MFLR ordered digraph. Then G contains a spanning tree.

Proof. Let $H = (V\ E_1)$ be the maximum spanning subgraph of G. Then, by lemma 4.5, H is a forest. It will be shown that H is a tree. This will be done by assuming that H has more than one component tree and deriving a contradiction.

Without loss of generality, assume that $V = \{1\ 2\ \cdots\ n\}$ and α is the identity function. Suppose H has more than one component tree. For each component define the root to be the node i which maximizes $\alpha^{-1}(j)$ in the component and identify this component by T_i. By assumption there exist at least two components T_n and T_m, where m is the largest node not in T_n. The contradiction will be achieved by showing that there must be some node j in T_n for which both $m \rightarrow j$ and $j \rightarrow m$ are edges in E, thereby connecting T_n and T_m into a single tree.

Claim 1. Let $i < k$ and suppose that s is a node in T_i, t is a node in T_k, and $s \rightarrow t$ is an edge in G. Then $s = i$.

Proof of claim 1. If $s < i$, then there is a simple path from i to s to t to k violating the FLR condition assumed for G. □

Since G is strongly connected, there is a simple path P from m to n in G. P contains nodes of the components T_m and T_n only, since otherwise the FLR condition on G is

violated. By claim 1, any edge from a node in T_m to a node in T_n must have tail m. Let $K = \{k_1, \ldots, k_t\}$ be the set of all nodes in T_n such that $m \rightarrow k_i$ is an edge in G for $i = 1, \ldots, t$. Then K is non-empty. Also, the FLR condition implies that $k_i > m$ for $i = 1, \ldots, t$.

The maximality condition on G requires that the edge $n \rightarrow i$ is in G for $i = 1, 2, \ldots, n-1$. This means that there exists a smallest node j in T_n for which there is some r in T_m such that $r < j$ and $j \rightarrow r$ is an edge in G. For each $s = 1, 2, \ldots, t$ there is a path $j \rightarrow r \rightarrow \cdots \rightarrow m \rightarrow k_s \rightarrow \cdots \rightarrow n$. If the FLR condition on G is not to be violated, then either $j = n$ or the subpath $k_s \rightarrow \cdots \rightarrow n$ passes through j. In either case, j is the root of a subtree in T_n containing all of the nodes in K. In particular, by lemma 4.4, $j \geq k_i > m$ for $i = 1, 2, \ldots, t$.

Claim 2. Edge $j \rightarrow m$ is in G.

Proof of claim 2. Suppose $j \rightarrow m$ is not in G. Then, because of the maximality of G, the digraph obtained by adding this edge to the edge set E is not an FLR ordered digraph. Clearly, the only way that the addition of this edge could violate the FLR condition on G is if there is a forward path $P = m \rightarrow k_i \rightarrow \cdots \rightarrow n$ in G that jumps j. Note that the only forward edges out of m are $m \rightarrow k_i$ for some node k_i in K. Let P_i be the subpath created from P by deleting the edge $m \rightarrow k_i$. Let Q_i be the simple path from j to k_i in T_n. Then, by lemma 4.4, Q_i consists of backward edges only. The concatenation of Q_i and P_i is a path in G which violates the FLR condition. This is a contradiction. Therefore $j \rightarrow m$ is in G. □

Claim 3. Edge $m \rightarrow j$ is in G.

Proof of claim 3. Suppose edge $m \rightarrow j$ is not in G. Then, because of the maximality of G, the digraph G_1 obtained by adding this edge to the edge set E is not an FLR ordered digraph. Since $m \rightarrow j$ is a forward edge, it will be included in a simple path in G_1 of the form $P = i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_p \rightarrow m \rightarrow j \rightarrow j_1 \rightarrow \cdots \rightarrow j_q$ where $i_1 \rightarrow i_2$ is a backward edge, all of the other edges are forward edges and $j_q > i_1$ (it is possible that $p = 1$, in which case $i_2 = m$, and for $q = 0$, in which case j is the terminal node of P and $j > i_1$). In particular, $j \neq i_1$.

The first thing to be noticed is that i_1 is the root of a subtree in T_n that contains the set K, since otherwise there is a forward path in T_n from some k_s to n that does not pass through i_1. Then the concatenation of the paths from i_1 to m and from m to k_s to n

violates the FLR condition in G. The same argument used in claim 2 then shows that $i_1 \rightarrow m$ is also in G.

Consider again the path P. By the minimality of j, $j < i_1$ and since both are the roots of subtrees containing the set K, j is in the subtree rooted at i_1. Consequently there is a backward path Q in T_n from i_1 to j. The concatenation of Q and the subpath $j \rightarrow j_1 \rightarrow \cdots \rightarrow j_q$ of P would violate the FLR condition in G unless $i_1 = j_r$ for some r, $1 \leq r \leq q$. Consequently P is not simple which is a contradiction. Therefore $m \rightarrow j$ is in G. □

Claim 2 and claim 3 provide the contradiction that proves the theorem. ■

Theorem 4.10 partially solves the problem of characterizing MFLR ordered digraphs. Since any MFLR ordered digraph contains a spanning tree and any FLR ordering on a tree is an invariant ordering, this characterization can be completed by solving the second problem, characterizing the FLR-preserving edges that can be added to an invariantly ordered tree.

The following definition provides a useful classification of the edges in a digraph which contains an invariantly ordered spanning tree.

Definition 4.5 : Let $T = (V, E, \alpha)$ be an invariantly ordered rooted tree for which $\alpha(n)$ is the root, and let $K = (V, V \times V)$ be the complete graph on V. The edges in E are called *tree edges* and the edges in $V \times V - E$ are classified as follows. $\alpha(i) \rightarrow \alpha(j)$ is a *class 1* edge if $\alpha(j)$ is in the subtree of T rooted at $\alpha(i)$. $\alpha(i) \rightarrow \alpha(j)$ is a *class 2* edge if $j > i$ and $\alpha(j)$ is in a subtree of T rooted at a sibling of $\alpha(i)$. Otherwise $\alpha(i) \rightarrow \alpha(j)$ is a *class 3* edge.

See example 4.2 for examples of class 1, 2 and 3 edges.

Lemma 4.6 : Let T and K be defined as in definition 4.5 where $V = \{ 1, 2, \ldots, n \}$ and α is the identity. If i is in the subtree T_k of T rooted at k and $i \neq k$, then

(a) any path in K starting at i and consisting entirely of class 1 and class 2 edges contains only nodes in $T_k - k$.

(b) any path P in K from i to some j, $j > k$, and consisting entirely of class 1, class 2 and tree edges must pass through k.

Proof Part (a) is proved by a simple induction argument. For the base case, note that the head of a class 1 edge from i is a descendent of i, and the head of a class 2 edge is in the subtree rooted at a sibling of i, and so both are in $T_k - k$. Assume that any path from i consisting of $s \geq 1$ class 1 or class 2 edges terminates in $T_k - k$, then any path of $s + 1$ class 1 or class 2 edges also terminates in $T_k - k$ by an application of the base case.

Part (b) is proved next. By lemma 4.4, j is not in T_k. From part (a), if P did not pass through k, there would be a tree edge from some node q in T_k with $q \neq k$ to some node p not in T_k. There is no such tree edge, since its presence would imply a cycle in T. Therefore P must pass through k. ■

Theorem 4.11 : Let $T = (V, E, \alpha)$ be an invariantly ordered rooted tree for which $\max \{ \alpha^{-1}(v) \mid v \in V \}$ is attained at the root. Suppose an arbitrary set of class 1 and class 2 edges is added to E. Then the resulting digraph remains FLR ordered. Furthermore, if any class 3 edge is added to E, then the resulting digraph is not FLR ordered.

Proof Without loss of generality it is assumed that $V = \{ 1, 2, \ldots, n \}$ and α is the identity.

Suppose a set of class 1 and class 2 edges is added to E. Suppose there is a path $i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_q$ in the resulting digraph such that $i_1 \rightarrow i_2$ is a backward edge, all other edges are forward edges and $i_q > i_1$. Since $i_1 \rightarrow i_2$ is a backward edge, it is either a class 1 edge or a tree edge. In either case i_2 is in the subtree rooted at i_1. Lemma 4.6(b) and lemma 4.4 then imply that the subpath $i_2 \rightarrow \cdots \rightarrow i_q$ passes through i_1, so does not violate the FLR condition. The digraph therefore remains FLR ordered when this set of class 1 and class 2 edges is added.

Suppose a class 3 edge is added to E. Then there are two cases to consider.

Case 1 The edge is $i \rightarrow j$ $j < i$ but j is not in the subtree rooted at i Let k be the nearest common ancestor of both i and j Then i $j < k$ and there is a forward path in T from j to k that does not pass through i The digraph resulting from adding the backward edge $i \rightarrow j$ to E is therefore not FLR ordered

Case 2 The edge is $i \rightarrow j$ $j > i$ but j is not in any subtree rooted at a sibling of i By lemma 4 4 j is not in the subtree rooted at i either If p is the parent of i then j is not in the subtree rooted at p and the nearest common ancestor k of p and j has the property that $k > p$ This means that there is a simple path of forward tree edges $j \rightarrow j_1 \rightarrow j_2 \rightarrow \quad \rightarrow j_q \rightarrow k$ not containing p Since the backward edge $p \rightarrow i$ is also in T adding the edge $i \rightarrow j$ to E results in a digraph which is not FLR ordered

Since these cases cover all possibilities for class 3 edges adding any class 3 edge to E results in a digraph which is not FLR ordered ■

Corollary 4.2 : Let T be an invariantly ordered tree with ordering α Then there is a unique MFLR ordered digraph with ordering α having T as a spanning tree

Proof By theorem 4 11 the result of adding all class 1 and class 2 edges to T is an FLR ordered digraph and any MFLR ordered digraph having T as a spanning tree may have only class 1 class 2 and tree edges ■

The next lemma is useful for detecting a condition in which no FLR ordering on G can exist A test in algorithm 4 4 uses this result to justify reporting that no FLR ordering exists

Lemma 4.7 : Let $G = (V \; E)$ be a digraph which contains a spanning tree T Suppose that for some choice of a root r for T there is some node p and a subset $S = \{ j_0 \; j_1 \quad j_{k-1} \}$ $k > 1$ of the children of p such that there is an edge from j_i into the subtree rooted at $j_{(i+1) \bmod k}$ for $i = 0 \; 1 \quad k-1$ Then no FLR ordering exists for G

Proof By theorem 4 11 it is sufficient to prove that for any choice of a root t for T and any invariant ordering α of T such that $\alpha(n) = t$, there exists some class 3 edge in G

For p as the choice of root the hypothesis of the lemma implies that there is no way to number the nodes in S so that each of the k edges defining S is a class 2 edge Therefore

at least one of these edges is a class 3 edge

If a node that is not in one of the subtrees rooted at some j_i for $i = 0\ 1 \ldots k-1$ is chosen as the root then S is still a subset of the children of p in this new rooted tree A class 3 edge exists for the same reason as above

Suppose that a node q from the subtree rooted at $j_{(i+1) \bmod k}$ is chosen as the root of T Let s be the node in the original subtree rooted at $j_{(i+1) \bmod k}$ such that the edge $j_{i \bmod k} \rightarrow s$ was used in defining S Then the subtree rooted at p in this new rooted tree contains $j_{i \bmod k}$ but does not contain s The edge $j_{i \bmod k} \rightarrow s$ is therefore a class 3 edge for any invariant ordering of T for which $\alpha(n) = q$ ■

When the hypothesis of lemma 4 7 is not satisfied but a class 3 edge is found to exist it may be possible to select another node from the tree to be the root and define another invariant ordering so that the edge is no longer class 3 Some nodes can be excluded from consideration a priori This is the content of the next lemma which is used in algorithm 4 4 to justify restricting the number of possible roots to $O(\log n)$

Lemma 4 8 : Let G be a digraph whose maximum subgraph is a tree T Suppose for some choice of a root r for T there is some node $i \neq r$ and an edge $i \rightarrow j$ where j is not in the subtree rooted at the parent node p of i Then the only nodes that can possibly be the highest numbered nodes in FLR orderings of G are in the subtree rooted at p

Proof Suppose that node t is the highest numbered node in some FLR ordering on G When t is designated as the root of T theorem 4 11 implies that the edge $i \rightarrow j$ is either a class 1 or a class 2 edge If $i \rightarrow j$ is a class 1 edge then j is in the subtree rooted at i which implies either that $t = i$ or that t is a descendent of i (in the original tree rooted at r) If $i \rightarrow j$ is a class 2 edge then either $t = p$ or t is a descendent of p (in the original tree rooted at r) excluding nodes in the subtree rooted at i These two cases together imply the result ■

Theorem 4 11 together with lemmas 4 7 and 4 8 provide the outline of an algorithm for finding FLR orderings for digraphs containing a spanning tree As mentioned in section 4 2 an FLR ordering for a tree can be obtained by arbitrarily choosing a root node which receives the largest number and then numbering the rest of the tree in depth-first fashion

There is complete freedom regarding the order in which the children of a node are stacked during this procedure. Algorithm 4.4 attempts to find an order for the children of a node such that numbering them in that order results in none of them having outgoing class 3 edges. Lemmas 4.7 and 4.8 are used respectively to abort execution and choose a new root for the tree. The format of algorithm 4.4 differs somewhat from the Pascal-like format of previous algorithms. This was done for the sake of readibility since the algorithm requires deep nesting and makes premature exits from various depths in the nesting.

Algorithm 4.4: FLR ordering algorithm for digraphs containing a spanning tree

Input A digraph $G = (V, E)$ containing a spanning tree $T = (V, E_1)$. Assume that $V = \{1, 2, \ldots, n\}$. The edge set E_1 is represented by an adjacency list structure *TREEADJ*. The edges in $E - E_1$ are represented by the adjacency list structure *ADJ* and are referred to as non-tree edges.

Output An FLR ordering α onto V if one exists, otherwise a message of failure.

Data Structures *ROOTTREE* is a tree used to store the set of nodes which can possibly be the root of T in some FLR ordering of G. *STK* is a stack used to store nodes for future numbering, where the node on top is to receive the largest available number. *CHILD* is an acyclic digraph on the node set V, where an edge $i \rightarrow j$ means that j is a child of i. *PARENT* is an acyclic digraph on V, where an edge $i \rightarrow j$ means that j is the parent of i. The adjacency lists of these two digraphs are referenced as *CHILD* $[i]$ and *PARENT* $[i]$ respectively. Since *PARENT* $[i]$ contains either a single entry or is empty, *PARENT* can be represented by an n-place array. Note that the union of the edge sets of *CHILD* and *PARENT* is E_1, the edge set of T. In an efficient implementation, only the *PARENT* array would be computed, but it is more convenient to express the algorithm in terms of both. The asymptotic time complexity of algorithm 4.4 is not affected if both *CHILD* and *PARENT* are computed. The adjacency list A is used to construct an acyclic digraph to keep track of numbering precedences for the children of a node. This is not the same as the $A(i)$ function of algorithm 4.3.

Sub-algorithms *TOPSORT* (algorithm 2.1.3.3) returns a topological sort of an acyclic digraph.

SPLIT. The input is a tree $T = (V, D)$ (represented by an adjacency list) and a designated root node t. It is assumed that $V = \{1, 2, \ldots, n\}$. The output is a pair of acyclic

digraphs *CHILD* and *PARENT*. *CHILD* [i] denotes the adjacency list of node i in *CHILD* and consists of the list of children of i (with respect to T and t). *PARENT* is represented by a vector where *PARENT* [i] denotes the parent of node i in T with respect to the root node t. *SPLIT* uses a stack *STK* with stack items consisting of pairs (i p) where p is the parent of i (except if $i = t$ in which case p is null). Initially *STK* contains the single pair (t $\emptyset$). *CHILD* [i] = $\emptyset$ for all i and *PARENT* [t] = 0. *SPLIT* then enters a loop in which (i p) is popped from *STK* and for each $j \in ADJ$ [i] such that $j \neq p$, *PARENT* [j] $\leftarrow i$, j is added to *CHILD* [i] and (j, i) is pushed onto *STK*. The loop exits when *STK* = $\emptyset$. The time to execute *SPLIT* is $O(n)$.

BALANCE. This algorithm has a tree as input and outputs a node r in the tree such that when r is the root of the tree, no subtree rooted at a child of r contains more than half of the nodes in the tree. Initially, a root r is chosen arbitrarily and for each node i the number of nodes in the subtree rooted at i is computed as $Z[i]$. This can be done recursively in $O(n)$ time by defining $Z[i] = 1$ if i is a leaf and setting $Z[i] \leftarrow 1 + \sum_{j \in CHILD[i]} Z[j]$ in the recursive step. *BALANCE* now operates by comparing the value of $Z[i]$ to $\frac{n}{2}$ for each $i \in CHILD[r]$. If a child i is found for which $Z[i] > \frac{n}{2}$ then repeat with the subtree rooted at i and the original value of n. When a node i is found for which $Z[j] \leqslant \frac{n}{2}$ for all $j \in CHILD[i]$, return i. *BALANCE* executes in time $O(n)$.

RESTRICT. This algorithm has a rooted tree T and a node x as input. If x is a node in T, then the subtree rooted at x is returned. Otherwise the empty set is returned. The algorithm operates by searching T in a depth-first manner until x is found or all nodes in T have been exhausted. If x is found, then the subtree rooted at x can be computed in $O(n)$ time. The time to execute *RESTRICT* is $O(n)$.

Algorithm

(1) Initialize *ROOTTREE* $\leftarrow T$.

(2) If *ROOTTREE* is empty, then do step 7 (no FLR ordering exists). Otherwise, choose a root $t \leftarrow$ *BALANCE* (*ROOTTREE*); compute *CHILD* and *PARENT* from *SPLIT* (*TREEADJ*, t); initialize a stack *STK* to contain the root t as its only element; $N \leftarrow n$.

(3) If STK is empty then do step 6 (an FLR ordering of G has been found). Otherwise pop the top node ι from STK. $\alpha(N) \leftarrow \iota$. $N \leftarrow N-1$. If $CHILD[\iota]$ is empty then repeat step 3. Otherwise initialize an acyclic digraph with node set equal to $CHILD[\iota]$ and empty adjacency list A.

(4) Consider each non-tree edge $j_r \rightarrow k$ originating at some $j_r \in CHILD[\iota]$.

(4.1) If ι is not reachable from k in $PARENT$ then $j_r \rightarrow k$ is a class 3 edge (no FLR ordering can exist with the current choice of root t). $ROOTTREE \leftarrow RESTRICT(ROOTTREE, \iota)$. do step 2.

(4.2) Otherwise determine the unique $j_s \in CHILD[\iota]$ such that k is in the subtree rooted at j_s.

(4.2.1) If k is in the subtree rooted at j_r then take no action ($j_r \rightarrow k$ is a class 1 edge).

(4.2.2) If k is in the subtree rooted at j_s where $j_s \neq j_r$ then

(4.2.2.1) If there is a path from j_s to j_r in A then do step 7 (by lemma 4.7 no FLR ordering exists for G).

(4.2.2.2) If there is no path from j_s to j_r in A and there is also no path from j_r to j_s in A then add $j_r \rightarrow j_s$ to A.

(5) Apply $TOPSORT$ to A. push the nodes in $CHILD[\iota]$ onto STK in ascending order of the numbers assigned to the nodes by $TOPSORT$. do step 3.

(6) Return the numbering assigned to the nodes of V (it is an FLR ordering). Stop.

(7) Return "No FLR ordering exists". Stop.

Theorem 4.12 : Algorithm 4.4 correctly returns an FLR ordering if one exists and correctly returns a message of failure if no FLR ordering exists.

Proof Suppose algorithm 4.4 executes step 6 returning an ordering α. Let $q = \alpha(n)$. Then q is the current root of the tree when α is found. Since step 6 is only executed when STK is empty and this only happens after every node has been pushed onto STK and subsequently popped and numbered, all nodes are assigned numbers by α. Since a node is always numbered before any of its children are pushed onto STK, α is an invariant ordering on T (see section 4.2).

It remains to be shown that α is an FLR ordering on G. Consider a node i where $CHILD[i] = \{ j_1, j_2, \ldots, j_m \}$ relative to the q-rooted tree and suppose $j_r \rightarrow k$ is a non-tree edge in E for some r, $1 \leq r \leq m$. By theorem 4.11 it must be shown that $j_r \rightarrow k$ is a class 1 or class 2 edge. Consider the instance of step 4 in the algorithm in which this edge is processed.

If substep 4.1 is applicable, so that k is not in the subtree rooted at i, then step 2 is executed and a new root is chosen for the tree. Since this does not happen, substep 4.2 is applicable.

If substep 4.2.1 is applicable, then k is in the subtree rooted at j_r, and the edge is class 1.

If substep 4.2.2 is applicable, then k is in the subtree rooted at some j_s where $j_s \neq j_r$. If substep 4.2.2.1 is applicable, then step 7 is executed. Since this does not happen, substep 4.2.2.2 is applicable. The result is that there is a path from j_r to j_s in the acyclic digraph on $CHILD[i]$ following the execution of substep 4.2.2.2. When step 5 is executed for this acyclic digraph (the test in case 4.2.2.1 ensures that the digraph is acyclic), node j_r precedes j_s in the ordering, so that j_s is pushed onto STK after j_r. j_s is then numbered before j_r, receiving a larger number. In addition, all descendents of j_s, in particular k, are pushed onto STK above j_r, and receive larger numbers. It follows that $j_r \rightarrow k$ is a class 2 edge. Therefore α is an FLR ordering on G.

To complete the proof it must be shown that whenever step 7 is executed, no FLR ordering exists for the digraph. If step 7 is executed, then either substep 4.2.2.1 is executed or *ROOTTREE* is found to be empty in an execution of step 2.

If substep 4.2.2.1 is executed, then k is in the subtree rooted at some $j_s \in CHILD[i]$, $j_s \neq j_r$, and there is a path from j_s to j_r in A. Edges are only added to A in substep 4.2.2.2 so there is a sequence $(j_s = j_{s_0}), j_{s_1}, \ldots, (j_{s_p} = j_r)$ in $CHILD[i]$ for some $p \geq 1$, such that there is an edge from j_{s_q} into the subtree rooted at $j_{s_{q+1}}$ for $q = 0, 1, \ldots, p-1$. Since the edge from j_r into the subtree rooted at j_s completes the satisfaction of the hypothesis of lemma 4.7, no FLR ordering exists and the algorithm was correct in executing step 7.

Suppose *ROOTTREE* is found to be empty in some execution of step 2. Each time *ROOTTREE* is modified in step 4.1 it is because a situation satisfying the hypothesis of lemma 4.8 is encountered. The algorithm *RESTRICT* modifies *ROOTTREE* in accordance

with the conclusion of lemma 4.8. Therefore, if *ROOTTREE* is found to be empty, it is because any choice of root results in a class 3 edge. By theorem 4.11 no FLR ordering exists for G. ■

Theorem 4.13 : The time required to execute Algorithm 4.4 is $O(n \log n + m \log n(d + c + \min(m, c^2)))$ where n is the number of nodes in G, m is the number of non-tree edges in G, d is the diameter of T, and c is the maximum degree in T of any node in V.

Proof. Step 2 initiates an outer loop which is executed at most once for each node that is not excluded as a possible root for the tree by an application of *RESTRICT*. Since *RESTRICT* and *BALANCE* ensure that the number of possible roots under consideration is reduced by at least half from one iteration to the next, this outer loop is executed at most $O(\log n)$ times.

Consider now the time required to execute a single iteration of this outer loop. Executing *BALANCE* and *SPLIT* require $O(n)$ time each, so the time spent in step 2 is $O(n)$.

The total time spent in step 3 for popping and numbering nodes and for initializing the acyclic digraphs is also $O(n)$.

Step 4 is executed $O(m)$ times, once for each non-tree edge. Consider the time required to process one edge $j_r \rightarrow k$. A single test requiring $O(d)$ time is made for the existence of a path in *PARENT* (the length of the longest path in *PARENT* is no greater than the diameter d). The child of i through which this path passes, if it exists, is determined at the same time at no extra cost. We may also have to make tests for paths from j_s to j_r and from j_r to j_s in A. There are at most $O(\min(m, c^2))$ edges and $O(c)$ nodes in A, so the time for these tests is $O(c + \min(m, c^2))$. The actions that may be taken in step 4 include a call to *RESTRICT*, which takes $O(n)$ time but is executed only $O(\log n)$ times for the entire algorithm, the addition of an edge to A, which takes $O(1)$ time since the edges do not have to be sorted, and some other simple $O(1)$ actions. The total time spent in step 4 over all non-tree edges, excluding the call to *RESTRICT*, is $O(md + mc + m \min(m, c^2))$.

Consider the total time spent in step 5. It is the same as the time required if a topological sort is done on the union of all of the acyclic digraphs that are presented to step 5.

Since each non-tree edge contributes at most one edge to A, the total time in step 5 is $O(n+m)$.

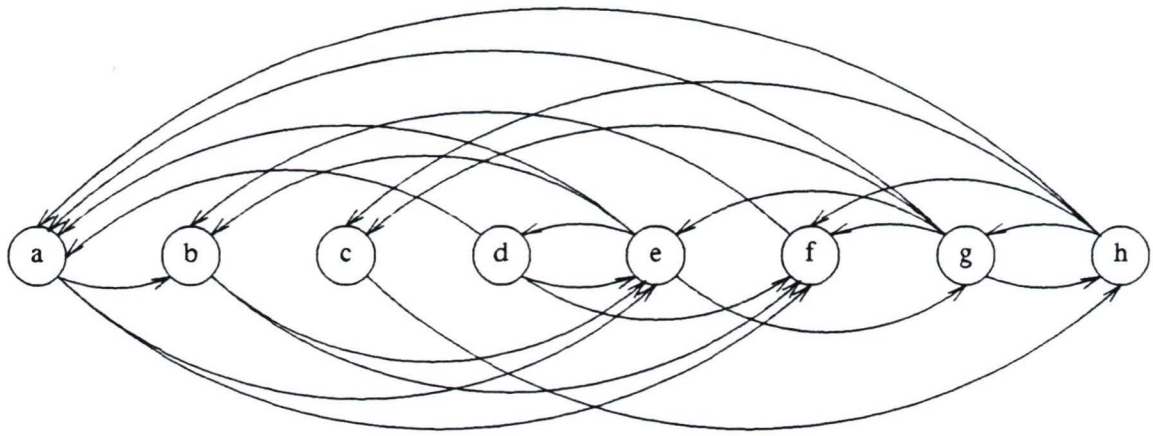
The total time spent in steps 2, 3, 4, and 5 for a single iteration of the outer loop is therefore $O(n + md + mc + m \min(m, c^2))$. The outer loop is executed $O(\log n)$ times. Additionally steps 1 and 6 require only $O(n)$ time, and all calls to *RESTRICT* require $O(n \log n)$ time in total. Therefore the time required to execute the entire algorithm is $O(n \log n + m \log n(d + c + \min(m, c^2)))$. ■

Under certain assumptions the time complexity of algorithm 4.4 is considerably better than the bound established for algorithm 4.3. In particular consider the class $\mathbf{D_p}$ of digraphs, where a digraph G is in $\mathbf{D_p}$ if and only if G contains some spanning tree and any spanning tree T of G has no node with degree greater than p. If algorithm 4.4 is applied to the class $\mathbf{D_p}$, then since $m < e$ and $d < n$, the time complexity is bounded by $O(n \log n\, e)$. This is much better than the complexity of $O(n^3 e)$ computed for algorithm 4.3.

It should be noted that in the worst case, when c is comparable to n, the time for algorithm 4.4 is bounded by $O(n^2 \log n\, e)$. This is still better than the time bound computed for algorithm 4.3.

Example 4.2 : This is an example illustrating the operation of algorithm 4.4 on an 8 node digraph G containing a spanning tree T. To avoid confusion with the numbers assigned by the algorithm, the letters a to h are used to identify nodes. Both a linear layout and the equivalent adjacency list representation of G are given below. The adjacency list is split into two parts, one for the tree edges and one for the non-tree edges.

Linear layout of G



Adjacency lists of tree edges

a	e
b	e f
c	h
d	e
e	a b d g
f	b
g	e h
h	c g

Adjacency lists of non-tree edges

a	b f
b	$\emptyset$
c	$\emptyset$
d	a f
e	$\emptyset$
f	$\emptyset$
g	a c f
h	a f

Algorithm 4.4 is run on this example and the actions taken are described in a series of numbered statements. Each statement is accompanied by a correspondingly numbered diagram which illustrates the state of knowledge following the actions in the statement. Each diagram takes the form of a subdigraph of the input digraph together with certain other information.

It must be stressed that these diagrams do not serve the same purpose as those in example 4.1, being primarily illustrative rather than an integral part of the algorithm. In particular, algorithm 4.4 does not construct a digraph on n nodes, only small digraphs on the children of a node. These small acyclic digraphs are not illustrated.

Each diagram contains the following information:

(1) The tree structure, indicated by solid bold lines. The root is at the top.

(2) The class 1 and 2 non-tree edges that have been considered up to that point, indicated by thin solid arrows.

(3) The numbers (α^{-1} values) currently assigned to any of the nodes. These are indicated in the circles along with the node names.

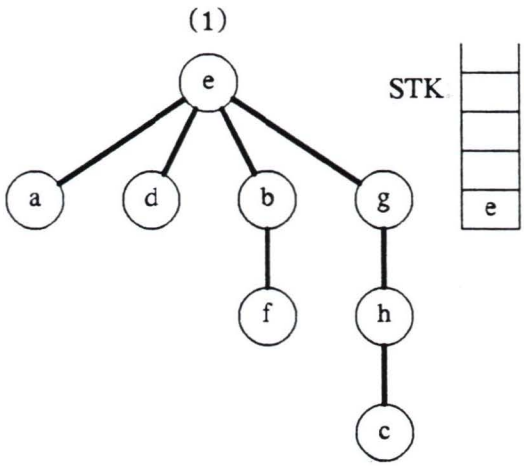
(4) The contents of *STK*, growing upward.

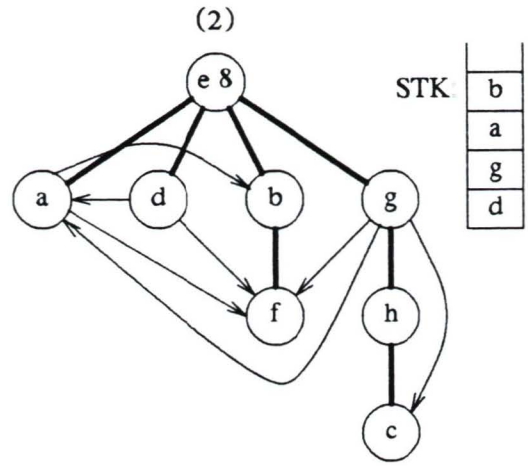
(5) Any class 3 edges found, indicated by dotted arrows.

Execution of the algorithm

(1) *ROOTTREE* is initially set to be a copy of the entire tree. When *BALANCE* is applied, node e is selected as the root of the tree, regardless of the starting node. *PARENT* and *CHILD* for this e-rooted tree are then computed. *STK* is initialized to contain e.

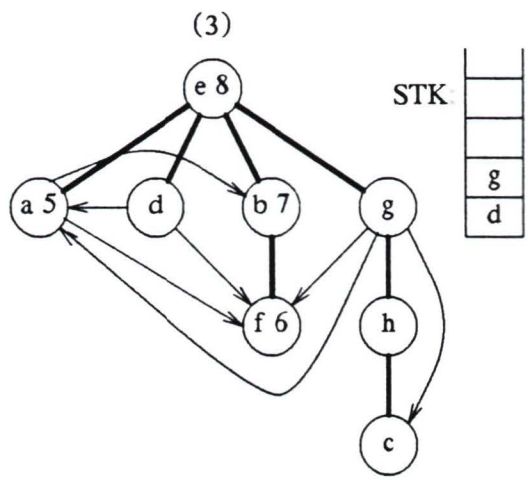
(2) Node e is popped from *STK* and numbered 8. The four children a, d, b, and g of e form the node set of an acyclic digraph P, with an initially empty adjacency list A. The non-tree edges out of a, d, b, and g are used to fill in the edges in A. The non-tree edge $a \rightarrow b$ in G, from a into the subtree rooted at b, adds the edge $a \rightarrow b$ to A; edge $a \rightarrow f$ adds nothing more; edge $d \rightarrow a$ adds edge $d \rightarrow a$ to A; edge $d \rightarrow f$ adds edge $d \rightarrow b$ to A; edge $g \rightarrow a$ adds edge $g \rightarrow a$ to A; edge $g \rightarrow c$ is a class 1 edge so adds nothing; edge $g \rightarrow f$ adds edge $g \rightarrow b$ to A. When P is topologically sorted, two possible orderings result: $g\ d\ a\ b$ and $d\ g\ a\ b$. Assuming the latter, d, g, a, and b are pushed onto *STK*, in that order.

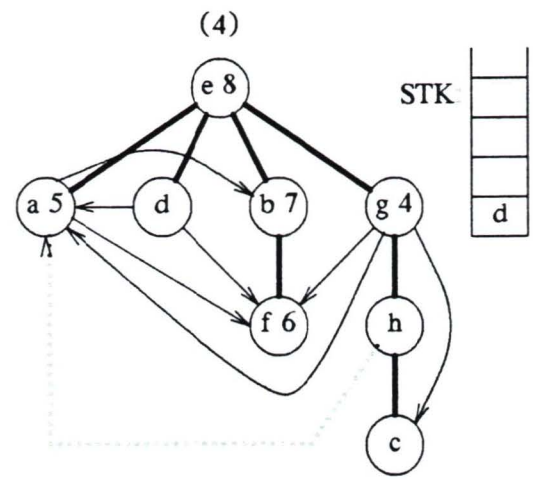




(3) Node b is popped from *STK* and numbered 7. Its only child f has no outgoing non-tree edges, so is automatically pushed onto *STK*. Node f is then popped from *STK* and numbered 6. Node a is popped and numbered 5.

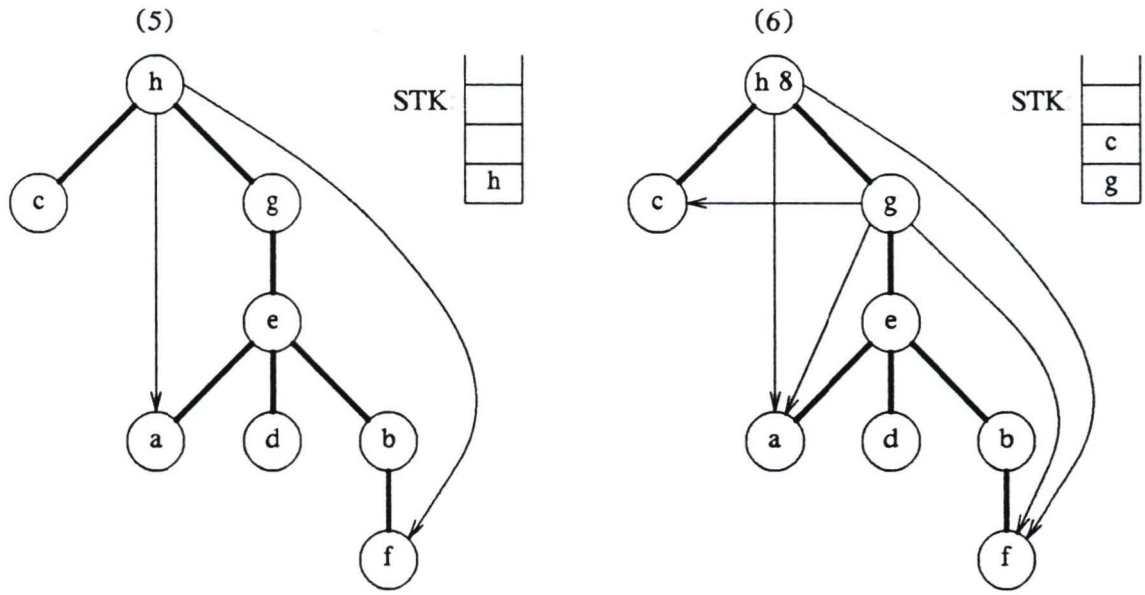
(4) Node g is popped from *STK* and numbered 4. The non-tree edges out of the only child h are examined. Since the head of edge $h \rightarrow a$ is not in the subtree rooted at g, $h \rightarrow a$ is a class 3 edge. This causes the termination of the current iteration. *RESTRICT* then sets *ROOTTREE* to be the subtree rooted at g (containing nodes g, h, c).





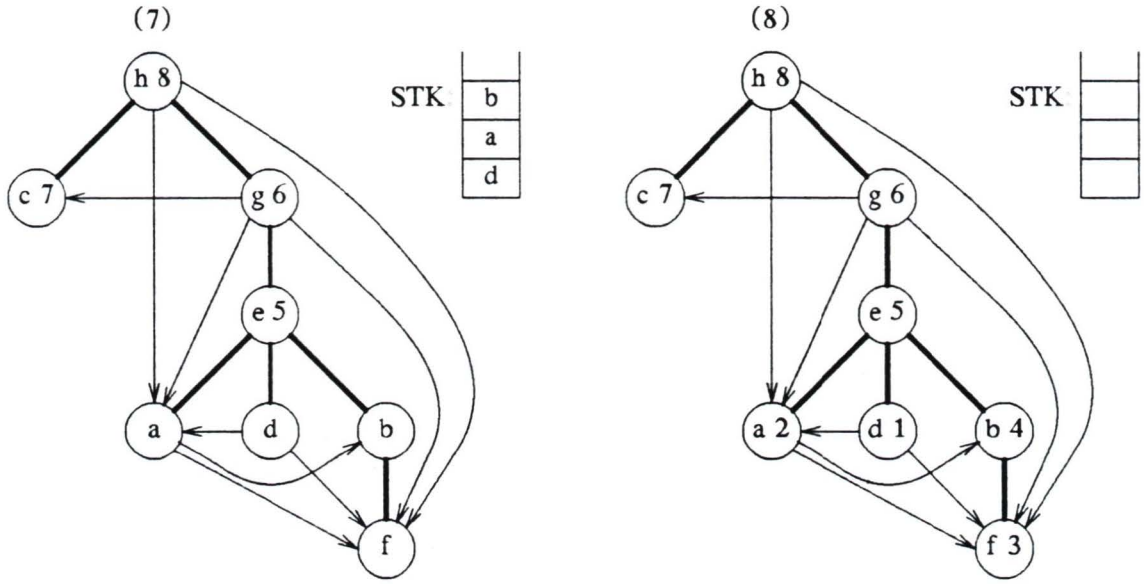
(5) *BALANCE* is applied to the new *ROOTTREE* and node h is selected as the new root. *PARENT* and *CHILD* for the h-rooted tree T are computed. *STK* is initialized to contain h. All non-tree edges out of h are automatically class 1 edges.

(6) Node h is popped and numbered 8. The two children c and g form the node set of an acyclic digraph P with an initially empty adjacency list A. The edge $g \rightarrow a$ is a class 1 edge so adds nothing to A. edge $g \rightarrow c$ adds edge $g \rightarrow c$ to A. edge $g \rightarrow f$ is a class 1 edge. After P is topologically sorted, g is pushed onto *STK*, followed by c.

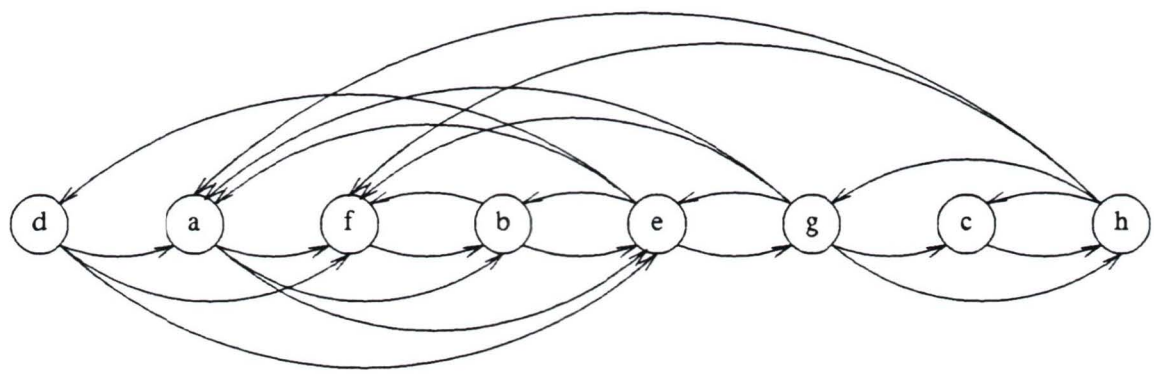


(7) Node c is popped and numbered 7. Node g is popped and numbered 6. Its only child e has no outgoing non-tree edges so is pushed onto *STK*. Node e is popped and numbered 5. The three children a, d, and b form the node set of the acyclic digraph P with initially empty adjacency list A. The edge $a \rightarrow b$ adds edge $a \rightarrow b$ to A. edge $a \rightarrow f$ adds nothing more to A. edge $d \rightarrow a$ adds edge $d \rightarrow a$ to A. edge $d \rightarrow f$ adds edge $d \rightarrow b$ to A. After P is topologically sorted, d, a, and b are pushed onto *STK* in that order.

(8) Node b is popped and numbered 4. Its only child f has no outgoing non-tree edges so is pushed onto *STK*. Node f is popped and numbered 3. Node a is popped and numbered 2. Node d is popped and numbered 1. This numbering is an FLR ordering of G.



FLR ordered linear layout of G The FLR ordering is represented pictorially by ordering the nodes from left to right according to the numbers assigned by the algorithm



End of example □

4.5 FLR orderings and the cycle structure of digraphs

The problem of characterizing the set of FLR orderable digraphs in terms of their cycle structure is considered in this section. The goal is to find a mathematical, as opposed to an algorithmic, test for FLR orderability.

The cycle structure seems to be the most obvious property to use for characterization because of lemma 3.2. Let $G = (V, E)$ be a digraph. Lemma 3.2 shows that if C is a simple cycle in G, then any FLR ordering is severely constrained with respect to the numbering of the nodes on C. Specifically, if α is an FLR ordering on G, then α numbers the nodes in C so that precisely one edge in C is a backward edge. This restriction then applies to each cycle in any set $\{C_1, C_2, \ldots, C_k\}$ of simple cycles in G. It is reasonable to expect that a knowledge of the relationships among the simple cycles in G would be useful in determining if an FLR ordering exists for G.

We first consider what can be learned by considering a single cycle. The following theorem strengthens lemma 2.2 for the case of an FLR ordered digraph.

Theorem 4.14: Let $G = (V, E, \alpha)$ be an FLR ordered digraph and let $C = (V_1, E_1)$ be a simple cycle in G. Suppose $v \in V$ is such that $\alpha^{-1}(v) > \alpha^{-1}(u)$ for all $u \in V_1$ and suppose there is a path P from some $c \in V_1$ to v such that c is the only node in V_1 that lies on P. Then $\alpha^{-1}(c) = \max\{\alpha^{-1}(u) \mid u \in V_1\}$.

Proof Suppose on the contrary that $\alpha^{-1}(c_r) = \max\{\alpha^{-1}(u) \mid u \in V_1\}$ and $\alpha^{-1}(c) < \alpha^{-1}(c_r)$. Then the concatenation of the path from c_r to c (in C) with P is a path that violates the FLR condition. This contradiction establishes the theorem. ■

As a corollary to theorem 4.14 we obtain a test which excludes certain nodes from being the highest numbered node in an FLR ordering.

Corollary 4.3 : Let $G = (V\ E)$ be a digraph, let $C = (\hat{V}\ \hat{E})$ be a simple cycle in G and let $v \in V - \hat{V}$. Suppose for $i = 1\ 2$ there exist distinct nodes $c_i \in \hat{V}$ and paths P_i from c_i to v such that c_i is the only node in $\hat{V}$ that lies on P_i. Then v is not the highest numbered node in any FLR ordering of G.

Corollary 4.3 has the potential of providing a means to make algorithm 4.3 execute faster by restricting the size of the set *HIGH*. It is not clear whether this can be done in an efficient manner or if the asymptotic time complexity can be reduced. It is also unclear whether any useful characterization of FLR orderable digraphs can result from further investigations in this area.

A more promising approach is to consider the relationship between pairs of simple cycles. We establish a necessary condition for the existence of an FLR ordering on G in terms of the set of simple cycles of G. First it is helpful to establish some terminology.

Definition 4.6 : Let $G = (V\ E)$ be a digraph, let $C = (\hat{V}\ \hat{E})$ be a simple cycle in G and let $v \in \hat{V}$. An ordering α on C is said to be *v-natural* if $C = \alpha(1) \rightarrow \alpha(2) \rightarrow \quad \rightarrow \alpha(k) \rightarrow \alpha(1)$ where $k = |\hat{V}|$ and $v = \alpha(1)$.

It is clear that a v-natural ordering on C is unique and is computed by numbering the nodes of C in the order they are encountered in a traversal of C starting from v.

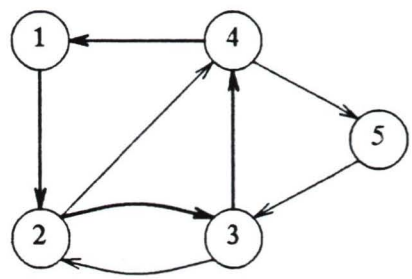
Definition 4.7 : Let $G = (V\ E\ \alpha)$ be an ordered digraph and let $W \subseteq V$ be non-empty. The numbering β on W *inherited from* α is defined inductively as follows. $\beta(1) = w$ where $\alpha^{-1}(w) = \min\{\alpha^{-1}(v)\ \ v \in W\}$. Having defined $\beta(i)$ for *some* $i < |W|$ define $\beta(i+1) = w$ where $\alpha^{-1}(w) = \min\{\alpha^{-1}(v)\ \ v \in W \text{ and } \alpha^{-1}(v) > \alpha^{-1}(\beta(i))\}$.

Definition 4.8 : Let $G = (V\ E)$ be a digraph, let $C_1 = (V_1\ E_1)$ and $C_2 = (V_2\ E_2)$ be simple cycles in G, let $W = V_1 \cap V_2$ and let $k = |W|$. Then C_1 and C_2 are said to be *consistent* if either $k \leqslant 2$ or for any $w \in W$ the numberings on W inherited from the w-natural orderings on C_1 and C_2 are identical. Two simple cycles which are not consistent are said to be *inconsistent*.

Note that if C_1 and C_2 are inconsistent then for *every* $w \in W$ the two numberings of W inherited from the w-natural orderings of C_1 and C_2 are different

There are two simple informal tests for consistency in simple cycles One test involves traversing each cycle starting from a common node If the set of nodes in common is encountered in the same order on both cycles then the cycles are consistent Another test is to discover if there is a one to one mapping of the union of the node sets of the two simple cycles into a circle in the plane such that each cycle can be traversed in a single clockwise trip around the circle The cycles are consistent if and only if this is possible

Example 4.3 : The following digraph on 5 nodes illustrates two inconsistent cycles $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$ with edges indicated in bold and $2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 2$ with light edges



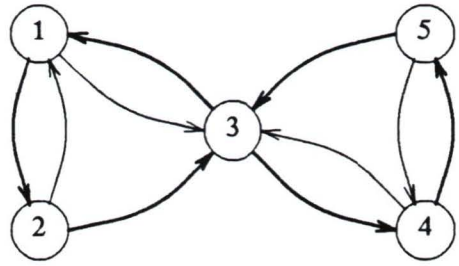
Theorem 4.15 : Let $G = (V\ E)$ be a digraph and let α be an FLR ordering on G Then any two simple cycles in G are consistent

Proof Let $C_1 = (V_1, E_1)$ and $C_2 = (V_2, E_2)$ be simple cycles in G let $W = V_1 \cap V_2$ and let $k = |W|$ Assume without loss of generality that $k \geq 3$

Suppose C_1 and C_2 are inconsistent Let $w \in W$ be such that $\alpha^{-1}(w) = \min\{\alpha^{-1}(v)\ v \in W\}$ Since the w-natural orderings β_1 of C_1 and β_2 of C_2 produce different inherited numberings of W at least one of them say the numbering inherited from β_1 produces a numbering different from the numbering inherited from α This means that C_1 has the form $C_1 = w \rightarrow \quad \rightarrow a \rightarrow \quad \rightarrow b \rightarrow \quad \rightarrow c \rightarrow w$ for some a and b in W such that $\alpha^{-1}(a) > \alpha^{-1}(b)$ In view of the minimality of $w \in W$, C_1 must have at least two backward edges with respect to α By lemma 3 2 this is a contradiction Therefore C_1 and C_2 are consistent ■

Theorem 4.15 establishes a necessary condition for the existence of an FLR ordering on a digraph. It is not a sufficient condition, however, as the following example shows.

Example 4.4 : This is an example of a 5 node, strongly connected digraph for which all simple cycles are pair-wise consistent but which is not FLR orderable. The two simple 3-cycles are indicated by bold edges.



This digraph is not FLR orderable since $FEAS[1,2] = \varnothing = FEAS[2,1]$ and $FEAS[4,5] = \varnothing = FEAS[5,4]$ (see the discussion immediately preceding example 4.1). □

Example 4.4 has other interesting properties. It contains a spanning tree and provides a counterexample to the converse of lemma 4.7. It also contains a cut vertex at node 3, but it is not clear if this is significant.

Any attempt to study properties of the set of all simple cycles has to contend with the large number of cycles that may exist. The number of simple cycles in G can be as large as $\sum_{i=2}^{n} \left| \begin{matrix} n \\ i \end{matrix} \right| (i-1)!$ in the case of a complete graph. The number of cycles that must be considered when studying consistency can often be greatly reduced by using maximal cycles.

Definition 4.9 : Let $G = (V, E)$ be a digraph and let S be the set of all simple cycles in G. Define a partial ordering on S by $C_1 \leqslant C_2$ if there is some subset K of the edges in C_1 such that C_2 is the result of replacing each edge $i \rightarrow j$ in K by a path starting from i and terminating at j. Then a maximal element of S with respect to this partial ordering is said to be a *maximal cycle* in G.

If C is an arbitrary simple cycle in G, then a maximal cycle M can be constructed inductively from C by repeatedly replacing one edge by a longer path until any such replacement results in a cycle which is not simple.

In general, many different maximal cycles can be derived from a simple cycle C, depending both on the order in which edges are replaced and on the paths used to replace them. It should be noted that no matter how the maximal cycle is derived, all nodes in C remain in M and retain their order in a traversal of M. The following lemma is therefore true.

Lemma 4.9 : Let $G = (V, E)$ be a digraph, let $C = (V_1, E_1)$ be a simple cycle in G and let $M = (V_2, E_2)$ be a maximal cycle derived from C. Then M and C are consistent.

Lemma 4.10 : Let $G = (V, E)$ be a digraph, let C_1 and C_2 be simple cycles in G and let M_1 and M_2 be maximal cycles such that $C_i \leqslant M_i$ in the partial ordering defined in definition 4.9, for $i = 1, 2$. If M_1 and M_2 are consistent, then C_1 and C_2 are consistent.

Proof Let U be the set of nodes common to C_1 and C_2 and let W be the set of nodes common to M_1 and M_2. Then $U \subseteq W$. Assume, without loss of generality, that $|U| \geqslant 3$ and let $u \in U$. Then, by definition 4.8, the u-natural orderings of M_1 and M_2 induce identical inherited numberings of W, hence of U. By lemma 4.9 the u-natural orderings of M_1 and C_1 induce identical inherited numberings of C_1 and hence of U. Similarly the u-natural orderings of M_2 and C_2 induce identical inherited numberings of U. The consistency of C_1 and C_2 follows. ■

The importance of lemma 4.10 is that questions concerning consistency of cycles can often be expressed in terms of the maximal cycles only. This has two benefits. There are usually only a small number of maximal cycles relative to the total number of cycles in a

digraph and the maximality condition can sometimes be exploited (see lemma 4.11 for instance). It should be noted that the number of maximal cycles in a digraph can still be very large. For instance the complete graph on n nodes has $(n-1)!$ maximal cycles.

Maximal cycles were used in preliminary work on the ordering problem and gave useful insight, although the algorithms eventually produced make no use of them. They may be of more use in defining special classes of digraphs with good properties with respect to FLR orderings. For instance, the case of digraphs which are the union of their maximal cycles is currently under study.

It should be noted that the digraph in example 4.4 is not the union of its maximal cycles. At this time it is suspected, but not proven, that if a digraph G is the union of its maximal cycles and every pair of maximal cycles in G is consistent, then G is FLR orderable. The converse is clearly false. A counterexample is the digraph in example 4.4 with the edge $2 \rightarrow 1$ removed, which has an FLR ordering but for which the edge $5 \rightarrow 4$ is not in any maximal cycle.

CHAPTER 5

Extensions of the algorithms and principal theorems

The results of chapter 4 are extended in this chapter. Extensions are made to orderings related to FLR orderings and consideration is given to extending the applicability of the principal algorithms.

In section 5.1 the algorithms and principal theorems of chapter 4 are modified to apply to BLR, FUR, and BUR orderings. In section 5.2, an extension of algorithm 4.3 to arbitrary digraphs is made, and the possibility of extending algorithm 4.4 is discussed.

5.1 Extensions to BLR, FUR, and BUR orderings

The four ordering types FLR, BLR, FUR, and BUR are closely related and simple transformations can be made between them with the help of lemmas 5.1 and 5.2 below. Consequently, all results of chapter 4 have immediate analogs for BLR, FUR, and BUR orderings.

Lemma 5.1 : An ordered digraph $G = (V, E, \alpha)$ is FLR [FUR] ordered if and only if the ordered reverse digraph $G_r = (V, E_r, \alpha)$ (see 2.1.3.1) is BLR [BUR] ordered.

Proof. Only the FLR-BLR case is considered; the other case is similar. Without loss of generality, assume that $V = \{1, 2, \ldots, n\}$ and α is the identity. By lemma 3.1, G is not FLR ordered if and only if there are three nodes $j < i < k$ such that $i \rightarrow j$ is in E and there is a forward path from j to k that jumps i. But then $j \rightarrow i$ is in E_r, and there is a backward path in G_r from k to j that jumps i. By lemma 3.1, G_r is not BLR ordered. The converse is proved similarly. ■

Definition 5.1 : Let $\alpha: \{1, 2, \ldots, n\} \rightarrow V$ be a bijection. Then α_r, the *reverse* of α, is the bijection defined by $\alpha_r(i) = \alpha(n+1-i)$ for $i = 1, 2, \ldots, n$.

Lemma 5.2 : Let $G = (V, E)$ be a digraph. Then α is an FLR [BLR] ordering on G if and only if α_r is a BUR [FUR] ordering on G.

Proof Immediate from lemmas 3.1 and 3.3. ■

The relationships between the orderings established in lemmas 5.1 and 5.2 are summarized in Fig. 5.1.

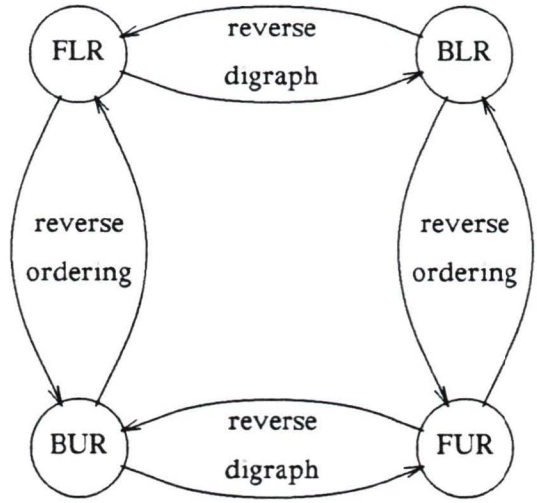


Fig. 5.1

5.1.1 Extensions of the recognition algorithm

The recognition algorithm (algorithm 4.1) can easily be modified to recognize FUR orderings. Fig. 5.1 indicates that an ordered digraph $G = (V, E, \alpha)$ is FUR ordered if and only if the reverse digraph with the reverse ordering $G_r = (V, E_r, \alpha_r)$ is FLR ordered. Algorithm 4.1 can therefore be used to recognize FUR orderings if, in a pre-processing step, the input ordered digraph is reversed, and its ordering is also reversed.

The adjacency lists $RADJ[i]$ for the reverse digraph can be computed in time $O(n+e)$ (see 2.1.3.1). As written, algorithm 4.1 requires that $V = \{1, 2, \ldots, n\}$ and the ordering be the identity. If G_r is to be input in this form, then the nodes in V must be renumbered so that node i is renumbered as $n+1-i$ for $i = 1, 2, \ldots, n$. Given the adjacency lists $RADJ[i]$, the renumbered adjacency lists can be computed in time

$O(n+e)$. Since the running time of algorithm 4.1 dominates the time to compute G_r, the time to execute the FUR version of the algorithm is still $O(ne+n^2)$.

BLR and BUR versions of the recognition algorithm are obtained even more easily by respectively reversing only the input digraph and reversing only the input ordering.

5.1.2 Extensions of theorem 4.3

Theorem 4.3 can be modified to apply to BLR, FUR, and BUR orderings. Since the reverse of a graph (undirected) is identical to the graph itself, lemma 5.1 implies that an ordering on a graph is an FLR [FUR] ordering if and only if it is a BLR [BUR] ordering. This means that theorem 4.3 is automatically true for BLR orderings and a single extension is required to cover FUR and BUR orderings.

An ordered forest $G = (V, E, \alpha)$ will be said to be *reverse invariantly ordered* if for each $v \in V$, there is at most one $u \in V$ such that u is adjacent to v and $\alpha^{-1}(u) < \alpha^{-1}(v)$. Define a *reverse minimum degree ordering* to be the reverse of a minimum degree ordering.

With these definitions theorem 4.3 can be modified to characterize FUR (and therefore BUR) orderings of trees.

Theorem 5.1: Let $T = (V, E, \alpha)$ be an ordered tree. Then the following conditions on α are equivalent.

(1) α is an FUR ordering on T.

(2) α is a reverse invariant ordering on T.

(3) α is a reverse minimum degree ordering on T.

5.1.3 Extensions of algorithms 4.3 and 4.4

A method similar to the one used to modify the recognition algorithm can be used with algorithm 4.3 to find BLR, BUR, or FUR orderings on strongly connected digraphs. A pre-processing and/or a post-processing step may be necessary as summarized below for the various types of orderings to be returned.

BLR The input digraph is reversed and the output ordering (if any) is returned unchanged

BUR The input digraph is unchanged while the output ordering is reversed

FUR The input digraph is reversed and the output ordering is also reversed

Algorithm 4.4 can be similarly modified to find BLR, BUR, or FUR orderings on digraphs containing a spanning tree as a subgraph. The tree adjacency list *TREEADJ* is unchanged, regardless of the ordering type, while the above three cases apply to the input subdigraph represented by *ADJ*, and to the output ordering.

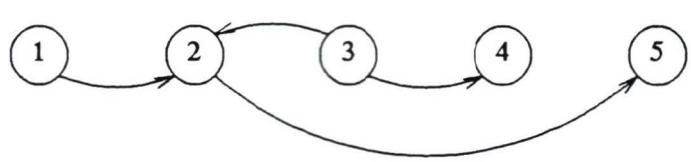
5.2 Relaxing restrictions on the applicability of algorithms

Algorithm 4.3 and 4.4 both have severe restrictions placed on their applicability. This section examines these restrictions and the consequences of relaxing them.

5.2.1 Strong connectivity in algorithm 4.3

Strong connectivity is essential to the proof that algorithm 4.3 terminates (see lemma 4.3). It is easy to construct simple examples of digraphs that are not strongly connected for which the algorithm does not terminate.

Example 5.1 : Algorithm 4.3 fails to terminate when the following five node digraph is input.



The feasible sets for this digraph are

$FEAS[1,2] = \{3, 4\}$

$FEAS[2,5] = \{1,3,4\}$

$FEAS[3,2] = \{1,4\}$

$FEAS[3,4] = \{1,2,5\}$

Suppose node 1 is selected in step 3 to be the highest numbered node in the ordering. Then the edges $1 \rightarrow 2$, $2 \rightarrow 5$ and $3 \rightarrow 4$ are immediate and add no edges to the acyclic digraph P being constructed by the algorithm. The edge $3 \rightarrow 2$ is deferred when it is seen for the first time, and since the immediate feasible sets add no edges to P, it remains deferred forever. Therefore step 5 of the algorithm will cycle forever, always executing case 5.2.2. □

Example 5.1 shows that algorithm 4.3 does not necessarily terminate if applied directly to a digraph which is not strongly connected. There is a natural way to circumvent this problem, however.

If $G = (V, E)$ is an arbitrary digraph, then the set $S = \{s_1, s_2, \ldots, s_t\}$ of strongly connected components of G can be computed in time $O(n + e)$ (see [25]). Define a digraph $H = (S, D)$ where $s_p \rightarrow s_q \in D$ if and only if $s_p \neq s_q$ and there is an edge $i \rightarrow j \in E$ such that $i \in s_p$ and $j \in s_q$. H is acyclic, since a cycle $s_{p_1} \rightarrow s_{p_2} \rightarrow \cdots \rightarrow s_{p_t} \rightarrow s_{p_1}$ in H would connect $s_{p_1}, s_{p_2}, \ldots, s_{p_t}$ into a single strongly connected component.

If any element of S is not an FLR orderable digraph, then G itself is not FLR orderable. Assume, therefore, that each strongly connected component of G has an FLR ordering.

Since H is acyclic, there exists a topological sort $\beta: \{1, 2, \ldots, t\} \rightarrow S$. Denote by s_i the component $\beta(i)$ in S and let k_i be the number of nodes in s_i and α_i be an FLR ordering on s_i. The following algorithm produces an FLR ordering α on G, given the orderings α_i.

(1) $N \leftarrow n$

(2) **for** i from 1 to t **do begin**

(3) **for** j from k_i down to 1 **do begin**

(4) $\alpha(N) \leftarrow \alpha_i(j)$

(5) $N \leftarrow N - 1$.

end end

(6) **return** α

This algorithm numbers the nodes in component s_i in the same relative order as α_i and ensures that all such nodes are numbered higher than any node in any s_j for $j > i$. In other words, all edges joining nodes in different components in S are backward edges.

It is easily shown that α is an FLR ordering on G. Suppose that u, v and w are distinct nodes in G such that $u \rightarrow v$ is a backward edge (with respect to α) and there exists a forward path in G from v to w. As noted above, any forward path in G must be contained in a single component s_i in S. If u is also in s_i, then $\alpha^{-1}(w) < \alpha^{-1}(u)$ since α and α_i are identical on s_i, except for a constant additive factor, and α_i is an FLR ordering on s_i. If u is in some other component s_j, then all nodes in s_j are numbered higher than any node in s_i. Consequently $\alpha^{-1}(w) < \alpha^{-1}(u)$. Since in either case there do not exist three nodes which satisfy the hypothesis of lemma 3.1, α is an FLR ordering on G. We have therefore proved the following theorem.

Theorem 5.2 : Let G be an arbitrary digraph. Then G is FLR orderable if and only if all of the strongly connected components of G are FLR orderable.

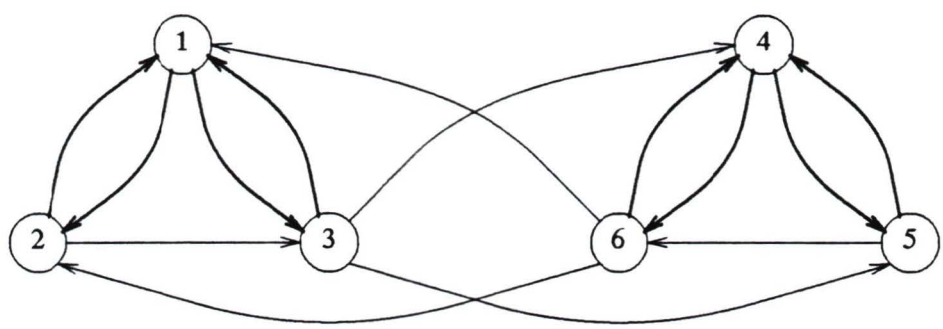
The time required to find an FLR ordering for an arbitrary digraph, when such an ordering exists, can be estimated. Denote by t the number of strongly connected components of G, by n and e the respective number of nodes and edges in G, and by $\hat{n}$ and $\hat{e}$ the respective maximum number of nodes and edges in any strongly connected component of G. The time required to compute the set S of strongly connected components of G is $O(n + e)$. The edge set of H can be computed by a method similar to algorithm 4.2. An array *REACH* of size t must be initialized for each of the t components for a total time of $O(t^2)$. Additionally, constant work must be done for each edge in G. The time to compute the edge set of H is therefore $O(e + t^2)$. The time required to compute all of the FLR orderings α_i is $O(t\hat{n}^3\hat{e})$. The time to compute α given the α_i orderings is $O(n)$. The total time required to obtain α when it exists, is $O(n + e + t^2 + t\hat{n}^3\hat{e})$. The time required when an FLR ordering does not exist is $O(n + e + t\hat{n}^3\hat{e})$ since the edge set of H need not be computed.

5.2.2 The spanning tree condition in algorithm 4.4

It does not appear that the spanning tree condition on the input digraph can be weakened in any significant way. The effect of choosing a root for the spanning tree is to decompose the ordering problem into a set of smaller sub-problems. Each sub-problem involves ordering the children of a single node only, and is therefore easier to solve than the original problem. Except for extreme examples, the total time required to solve all of the sub-problems is less than the time required to solve the original problem by the more general procedure in algorithm 4.3.

When the maximum subgraph contains more than one component tree, this useful decomposition is lost. It is tempting to use a method similar to the one used in 5.2.1 to try to extend algorithm 4.4 to digraphs whose maximum subgraph is an arbitrary forest. However, if we analogously apply algorithm 4.4 to each subdigraph induced by the component trees of the maximum subgraph, obtaining an FLR ordering for each, then there does not appear to be any natural way to use these orderings to produce an FLR ordering of the original digraph, as example 5.2 shows.

Example 5.2: The following 6-node strongly connected digraph G has two tree components in its maximum subgraph such that the subdigraphs induced by each such tree are FLR orderable. Furthermore, the 2-node digraph H, with node set consisting of these two subdigraphs and edge set defined as in 5.2.1, is also FLR orderable.



However, G is clearly not FLR orderable since $FEAS[1,2] = FEAS[2,1] = \emptyset$ and $FEAS[4,5] = FEAS[5,4] = \emptyset$ (see the discussion following theorem 4.9). □

The reason that algorithm 4 4 cannot be extended to arbitrary digraphs in this manner seems to be that the digraph H does not necessarily have any desirable properties such as acyclicity Therefore the existence of FLR orderings on the subdigraphs induced by the components does not automatically imply the existence of an FLR ordering on the digraph itself

CHAPTER 6

Conclusions

A number of topics concerning forward lower restricted ordered digraphs have been considered. This chapter contains a summary of the results of this thesis, a discussion of the practical importance of this work to sparse matrix analysis, and suggestions for further work.

6.1 Summary of results

The recognition, ordering and characterization problems were all considered in chapter 4, and results of varying degrees of completeness were attained for each.

The recognition problem was solved for arbitrary ordered digraphs with the presentation of algorithm 4.1. It correctly determines if an arbitrary ordered digraph is FLR ordered and has running time $O(ne + n^2)$.

Algorithm 4.3 was presented to solve the ordering problem for arbitrary strongly connected digraphs. It correctly determines if a strongly connected digraph can be FLR ordered and returns an FLR ordering in the affirmative case. The execution time for this algorithm is no worse than $O(n^3e)$. It was noted that a preliminary analysis of the feasible sets could sometimes dramatically decrease the execution time.

Algorithm 4.4 was also presented to solve the ordering problem, but only for a restricted class of strongly connected digraphs, those containing spanning trees as subgraphs. A bound of $O(n^2 \log n\ e)$ on the running time was found which is less than the bound established for algorithm 4.3.

Several characterization problems were considered. The problem of characterizing the FLR ordered trees was solved by theorem 4.3 which expanded on theorem 3.3 in [15] (establishing the equivalence of FLR and invariant orderings on trees) to prove the equivalence of FLR and minimum degree orderings as well.

The problem of characterizing the strongly connected maximal FLR ordered digraphs was accomplished in theorems 4.10, 4.11, and corollary 4.2. It was shown that a strongly

connected MFLR ordered digraph is completely characterized by the invariantly ordered spanning tree it contains. No useful characterization was obtained when the strongly connected condition was removed.

The problem of characterizing the FLR orderable digraphs in terms of their cycle structure was investigated. The requirement that any two maximal cycles be consistent was proved to be a necessary condition for the existence of an FLR ordering for an arbitrary digraph. Example 4.4 showed that this requirement is not a sufficient condition, however, even if the digraph is strongly connected or contains a spanning tree.

Extensions to the work in chapter 4 were considered in chapter 5. All of the principal results for FLR ordered digraphs can be modified in a natural manner to apply to BLR, FUR, and BUR ordered digraphs also.

It was shown how algorithm 4.3 could be used in an ordering algorithm for arbitrary digraphs. Algorithm 4.3 is applied to each strongly connected component of the input digraph. The digraph is FLR orderable if and only if each component is, and an FLR ordering is easily computable from any set of FLR orderings of the components.

The dependence of algorithm 4.4 on the input digraph containing a spanning tree was also considered. No useful results were obtained when this condition was relaxed. An example was presented which suggests that decomposition methods for extending the applicability of algorithm 4.4 will not work.

6.2 Practical importance of this work

The practical importance of algorithms 4.3 and 4.4 in sparse matrix analysis is slight at this time. The orderings produced by algorithms 4.3 and 4.4 are generated by purely combinatorial means, no use being made of the numerical values of the nonzeros. In order to be of practical value in the analysis phase of an algorithm to solve systems of linear equations, these ordering algorithms would have to incorporate stability considerations in the computation of the orderings. Otherwise, subsequent pivoting for stability would almost certainly destroy the inheritance property. It is unlikely that threshold pivoting can be incorporated into either algorithm without greatly reducing the number of digraphs for which FLR orderings can be found.

The time-space tradeoffs for various implementations of the algorithms were not considered. This topic is of special concern in algorithms 4.1 and 4.3 which assume that large structures (H in 4.1 and *FEASIBLE* in 4.3) are computed in a pre-processing step.

The subject of fast heuristic algorithms, especially as applies to the problem of permuting a matrix such that sub-global inheritance occurs (see section 3.5.7 for instance), was not investigated. Such algorithms may have greater practical value than the global ordering algorithms presented in this thesis.

Given an ordered digraph, no algorithm was presented in this thesis to efficiently determine the pairs of nodes (i, j) for which the digraph is (i, j) lower restricted. Such an algorithm is likely to be easy to develop.

An interesting problem is to decide, for a given matrix A and integer k, if there exists a symmetric permutation PAP^T for which at least k elements in the upper triangular part of A are inherited. By analogy to a similar problem with respect to fill-in, it is suspected that this problem may be NP-complete (see [23] pp. 188-194).

Perhaps the most promising direction for research into the practical applications of inheritance is to obtain as many characterizations as possible of FLR orderable digraphs, along the lines of what was attempted in section 4.5. It might then be possible to identify classes of problems that could be modelled so that the resulting systems admit FLR orderings. Additionally, such characterizations could lead to more efficient ordering algorithms.

Appendix

This appendix contains a FORTRAN implementation of algorithms 4.2 and 4.3. Five subroutines FLRORD, FEASET, ABOVE, BELOW, and TOPSORT are included.

Subroutine FLRORD implements algorithm 4.3. It is meant to be called by an application program and in turn calls the other four subroutines. FLRORD expects a strongly connected digraph to be input. If a digraph which is not strongly connected is input, then FLRORD will only return an error code if an infinite loop develops in the computation (see example 5.1). Otherwise, providing no other errors are detected, it will correctly determine if an FLR ordering exists. The output from FLRORD consists of an FLR ordering for the input digraph, when such an ordering exists, along with a code describing how the computation terminated.

Subroutine FEASET implements algorithm 4.2. It returns a queue structure containing the feasible sets corresponding to all of the edges in the input digraph. Also returned is a code which indicates how the computation terminated.

Subroutine ABOVE implements a depth first search of the input digraph starting from a designated node i. The output is the *above* set with respect to i.

Subroutine BELOW returns the *below* set with respect to a designated node i in the input digraph. The *below* set could be computed by applying ABOVE to the reverse of the input digraph. However, we choose to reduce the storage overhead required for a computation of the reverse digraph by computing the *below* set directly. Searches are conducted from each node, attempting to find a path to i, in such a way that each edge is considered at most twice. Every node k which is found to lie on some path to i is inserted in the *below* set.

Subroutine TOPSRT returns a topological sort of an input acyclic digraph and also detects if the digraph is not acyclic. See section 2.1.3.3 for an outline of the algorithm.

In the sections which follow, the parameters that are required by the subroutines are described and program listings are given.

A.1 Subroutine FLRORD

Purpose

To return an FLR ordering of a digraph when one exists, and to detect if no such ordering exists.

Input parameters

N, E - the number of nodes and number of edges, respectively, in the input digraph.

ADJNCY, ADJHED - a pair of arrays representing the adjacency lists of the digraph. ADJNCY is an integer array of dimension E+1 containing the adjacency lists. ADJHED is an integer array of dimension N+1 containing the indices in ADJNCY of the start of each adjacency list. The adjacency list of node I is empty if ADJHED(I) = ADJHED(I+1) and otherwise lies between the indices ADJHED(I) and ADJHED(I+1) - 1 in ADJNCY.

HIGH, HITOP - an integer array of dimension N and an integer variable, respectively, representing the stack of all nodes that are to be considered for the highest numbered node in the ordering. HITOP is the index of the top of the stack.

Output parameters

PERM - an integer array of dimension N used to return an FLR ordering when one exists. PERM(I) is the value of the ordering function at integer I.

RTCODE - returns an integer which indicates whether an FLR ordering was found or whether certain error conditions were detected. The meanings of the values returned are summarized in the following table.

RTCODE	MEANING
0	an FLR ordering is returned in PERM
1	no FLR ordering exists for the input digraph
2	ERROR - a larger ORDRER array must be used
3	ERROR - a larger FEASBL array must be used
4	ERROR - the digraph passed to TOPSRT is not acyclic
5	ERROR - the input digraph is not strongly connected

Working parameters

FEASBL, FEASIZ - FEASBL is an integer array of dimension FEASIZ to be used for storing the queue of feasible sets. A queue structure is imposed on FEASBL by

maintaining pointers HEAD and TAIL to the head and tail of the queue respectively. Each feasible set is represented by 5 fields (NXT, SIZ, I, J, FEAS(I,J)) arranged in contiguous blocks in FEASBL. NXT is the index in FEASBL of the start of the representation of the next feasible set, and SIZ is the number of nodes in FEAS(I,J). The NXT pointer of the last feasible set in FEASBL points to the start of the feasible set representation at the head of the queue. FEASIZ must have a value of at least F + 4*E, where F is the sum of the sizes of all of the feasible sets. The calling program is required to provide a sufficiently large value FEASBL array.

ORDRER, ORDLNK, ORDHED, ORDSIZ, ENDMRK - a linked list structure used to represent the FLR orderer being constructed. ORDRER is an integer array of dimension ORDSIZ used to store the adjacency lists. ORDLNK is an integer array of dimension ORDSIZ where ORDLNK(I) is the index in ORDRER of the node following ORDRER(I) on an adjacency list. The null pointer is indicated by 0. ORDHED is an integer array such that ORDHED(I) is the index in ORDRER of the start of the adjacency list of I. An empty adjacency list for node I is indicated by ORDHED(I) = 0. ENDMRK is an integer array of dimension N used to mark the index in ORDRER at which an adjacency list ends. The calling program must provide sufficiently large ORDRER and ORDLNK arrays.

NEXT - an integer array of dimension E used to store the link structure of the FEASBL queue so that it can be restored.

ABOV, BELO, UNKNWN, FEAS - logical arrays of dimension N used to represent the *above*, *below*, *unknown* (see pp. 44-45 for the definition of these terms) and *feasible* sets of nodes, respectively. A node I is indicated as being in one of these sets if and only if the value at index I is TRUE.

ACTIVE - an integer array of dimension N required by the subroutines ABOVE, BELOW and FEASET.

REACH - a logical array of dimension N required by BELOW and FEASET.

COMPLT - a logical array of dimension N required by BELOW.

INDEG, ZINDEG - integer arrays of dimension N required by TOPSRT.

Subroutines called

FEASET, ABOVE, BELOW, TOPSRT

Notes

(1) - References to steps or cases in the comments of the program refer to Algorithm 4.3.

Program

```
      SUBROUTINE FLRORD(N, E, ADJNCY, ADJHED, HIGH, HITOP, FEASIZ, FEASBL,
     *                  ORDSIZ, ORDRER, ORDLNK, ORDHED, ENDMRK, NEXT,
     *                  ABOV, BELO, UNKNWN, FEAS, ACTIVE, REACH, COMPLT,
     *                  INDEG, ZINDEG, PERM, RTCODE)
      INTEGER N, E, ADJNCY(E+1), ADJHED(N+1), HIGH(N), HITOP, FEASIZ,
     *        FEASBL(FEASIZ), ORDSIZ, ORDRER(ORDSIZ), ORDLNK(ORDSIZ),
     *        ORDHED(N), ENDMRK(N), NEXT(E), ACTIVE(N), INDEG(N), ZINDEG(N),
     *        PERM(N), RTCODE
      LOGICAL ABOV(N), BELO(N), UNKNWN(N), FEAS(N), REACH(N), COMPLT(N)
      INTEGER I, J, K, V, NXT, MARK, TAIL, FRTCOD, TRTCOD, QSIZE,
     *        ITERS, HEAD, SIZ
      LOGICAL CNTAIN
C
C     ------------------------------------
C     Step (1)  Compute the feasible sets.
C     ------------------------------------
      CALL FEASET(N, E, ADJNCY, ADJHED, ACTIVE, REACH, FEASIZ, FEASBL,
     *            FRTCOD)
      IF (FRTCOD .EQ. 0) THEN
C       ------------------------------------
C       Save the link structure of FEASBL.
C       ------------------------------------
        NXT = 1
        DO 100 I=1, E
          NXT = FEASBL(NXT)
          NEXT(I) = NXT
  100   CONTINUE
C       ------------------------------------------------------
C       Step (3)  Start of main loop -- select next high node
C       ------------------------------------------------------
  200   IF (HITOP .GT. 0) THEN
          V = HIGH(HITOP)
```

```
            HITOP = HITOP-1
C           ---------------------------------------
C           Restore the link structure of FEASBL
C           ---------------------------------------
            NXT = 1
            DO 300 I=1 E
              FEASBL(NXT) = NEXT(I)
              NXT = NEXT(I)
 300        CONTINUE
            TAIL = NEXT(E-1)
            QSIZE = E
            ITERS = 2*E
C           ------------------------------------------------------
C           Initialize the base acyclic digraph of the FLR orderer
C           ------------------------------------------------------
            MARK = 1
            DO 400 I=1,N
              IF (I .NE. V) THEN
                IF (MARK .GT. ORDSIZ) THEN
C                 ------------------------------------------------------
C                 Indicate that a larger ORDRER array must be supplied
C                 ------------------------------------------------------
                  RTCODE = 2
                  RETURN
                ENDIF
                ORDHED(I) = MARK
                ORDRER(MARK) = V
                ORDLNK(MARK) = 0
                ENDMRK(I) = MARK
                MARK = MARK+1
              ELSE
                ORDHED(I) = 0
              ENDIF
 400        CONTINUE
            HEAD = 1
```

```
C     ------------------------------------------------------------
C     Step (4)  If QSIZE = 0  then the FEASBL queue is empty
C     ------------------------------------------------------------
 500  IF (QSIZE  GT  0) THEN
        ITERS = ITERS-1
        IF (ITERS  LT  0) THEN
          RTCODE = 5
          RETURN
        ENDIF
C       --------
C       Step (5)
C       --------
        I = FEASBL(HEAD+2)
        J = FEASBL(HEAD+3)
        ----------------
        Loops are ignored
        ----------------
        IF (I  NE  J) THEN
          CALL ABOVE(N  I  ORDSIZ  ORDRER  ORDLNK  ORDHED  ACTIVE ABOV)
        ENDIF
        IF ((I  NE  J)  AND  ( NOT  ABOV(J))) THEN
C         ------------------------
C         Case (5 1) or (5 2) is true
C         ------------------------------------------------------------
C         Transfer FEAS(I J) into an array with same format as ABOV
C         ------------------------------------------------------------
          DO 600 K=1 N
            FEAS(K) =  FALSE
 600      CONTINUE
          SIZ = FEASBL(HEAD+1)
          DO 700 K=4 SIZ+3
            FEAS(FEASBL(HEAD+K)) =  TRUE
 700      CONTINUE
```

```
C     ------------------------------------------
C     Determine if ABOV is contained in FEAS
C     ------------------------------------------
      K = 1
      CNTAIN = TRUE
800   IF (( NOT ABOV(K)) OR FEAS(K)) THEN
        K = K+1
        IF (K LE N) THEN
          GO TO 800
        ENDIF
      ELSE
        CNTAIN = FALSE
      ENDIF
C     ------------------------------------------
C     Discriminate between cases (5.1) and (5.2)
C     ------------------------------------------
      CALL BELOW(N I ORDSIZ ORDRER ORDLNK ORDHED ABOV
     *           ACTIVE REACH COMPLT BELO)
C     ------------------------------------------
C     Compute the Unknown set with respect to I
C     ------------------------------------------
      DO 850 K=1 N
        IF (K NE I) THEN
          IF (( NOT ABOV(K)) AND ( NOT BELO(K))) THEN
            UNKNWN(K) = TRUE
          ELSE
            UNKNWN(K) = FALSE
          ENDIF
        ELSE
          UNKNWN(K) = FALSE
        ENDIF
850   CONTINUE
      IF (BELO(J)) THEN
        IF ( NOT CNTAIN) THEN
```

```
C                 ------------
C                 Case (5 1 1)
C                 ------------
                  GO TO 200
                ELSE
C                 ------------
C                 Case (5 1 2)
C                 ------------
                  DO 900 K=1 N
                    IF (UNKNWN(K) AND ( NOT FEAS(K))) THEN
                      IF (MARK GT ORDSIZ) THEN
                        RTCODE = 2
                        RETURN
                      ENDIF
                      ORDRER(MARK) = I
                      ORDLNK(ENDMRK(K)) = MARK
                      ORDLNK(MARK) = 0
                      ENDMRK(K) = MARK
                      MARK = MARK+1
                    ENDIF
900               CONTINUE
                ENDIF
              ELSE
C                 ----------
C                 Case (5 2)
C                 ----------
                  IF ( NOT CNTAIN) THEN
C                   ------------
C                   Case (5 2 1)
C                   ------------
                    IF (MARK GT ORDSIZ) THEN
                      RTCODE = 2
                      RETURN
                    ENDIF
                    ORDRER(MARK) = J
```

```
                    ORDLNK(ENDMRK(I)) = MARK
                    ORDLNK(MARK) = 0
                    ENDMRK(I) = MARK
                    MARK = MARK+1
                  ELSE
                    K = 1
 1000               IF ((K NE J) AND UNKNWN(K) AND ( NOT FEAS(K))) THEN
C                     ------------------------------
C                     Case (5 2 2) Deferred edge
C                     ------------------------------
                      TAIL = HEAD
                      HEAD = FEASBL(HEAD)
                      GO TO 500
                    ENDIF
                    K = K+1
                    IF (K LE N) THEN
                      GO TO 1000
                    ENDIF
                  ENDIF
                ENDIF
              ENDIF
C             ------------------------------
C             Remove from FEASBL queue
C             ------------------------------
              HEAD = FEASBL(HEAD)
              FEASBL(TAIL) = HEAD
              QSIZE = QSIZE-1
              GO TO 500
            ELSE
C             ---------
C             Step (6)
C             ---------
              CALL TOPSRT(N ORDSIZ ORDRER ORDLNK ORDHED INDEG
     *                    ZINDEG PERM TRTCOD)
              IF (TRTCOD EQ 0) THEN
```

```
C                 ---------------------------------------------
C                 Indicate valid FLR ordering returned in PERM
C                 ---------------------------------------------
                  RTCODE = 0
                  RETURN
                ELSE
C                 ------------------------------------------------------
C                 Indicate that a non-acyclic digraph was passed to TOPSRT
C                 ------------------------------------------------------
                  RTCODE = 4
                  RETURN
                ENDIF
              ENDIF
            ELSE
C             ---------------------------------
C             Step (7)  No FLR ordering exists
C             ---------------------------------
              RTCODE = 1
              RETURN
            ENDIF
          ELSE
C           ---------------------------------------------------
C           Indicate that a larger FEASBL array must be supplied
C           ---------------------------------------------------
            RTCODE = 3
            RETURN
          ENDIF
          END
```

A.2 Subroutine FEASET

Purpose To compute the feasible sets of a digraph

Input parameters

N E - the number of nodes and number of edges respectively in the input digraph

ADJNCY ADJHED - a pair of arrays representing the adjacency lists of the digraph. See subroutine FLRORD for more details.

FEASIZ - the dimension of FEASBL (see output parameters).

Output parameters

FEASBL - an integer array of dimension FEASIZ used to return the queue containing the feasible sets of the input digraph. See subroutine FLRORD for an explanation of the organization of the FEASBL queue.

RTCODE - returns an integer which indicates whether the feasible sets were successfully computed or an error occurred. The meanings of the values returned are summarized in the following table.

RTCODE	MEANING
0	FEASBL is correctly computed
1	ERROR - a larger FEASBL array must be used

Working parameters

ACTIVE - an integer array of dimension N used in a depth first search of the digraph.

REACH - a logical array of dimension N used in conjunction with ACTIVE to indicate the nodes encountered in a depth first search.

Notes

(1) - References to steps in the program comments refer to Algorithm 4.2.

Program

```
      SUBROUTINE FEASET(N, E, ADJNCY, ADJHED, ACTIVE, REACH, FEASIZ,
     *                  FEASBL, RTCODE)
      INTEGER N, E, ADJNCY(E+1), ADJHED(N+1), ACTIVE(N), FEASIZ,
     *        FEASBL(FEASIZ), RTCODE
      LOGICAL REACH(N)
      INTEGER I, J, K, K1, K2, INDJ, INDK, MARK, ADJST, ADJEND, KADJST,
     *        KADJEN, NXT, SIZ, COUNT, ACTTOP
C
      MARK = 1
      DO 600 I=1,N
        IF (MARK+3 .GT. FEASIZ) THEN
```

```
C     ------------------------------------------------------------------
C     A larger FEASBL array and dimension FEASIZ must be provided
C     ------------------------------------------------------------------
      RTCODE = 1
      RETURN
    ENDIF
    ADJST = ADJHED(I)
    ADJEND = ADJHED(I+1)-1
    IF (ADJEND .GE. ADJST) THEN
C     ------------------------------------------------
C     Compute feasible sets for all edges with tail I
C     ------------------------------------------------
      DO 500 INDJ=ADJST, ADJEND
        NXT = MARK
        SIZ = MARK+1
        FEASBL(MARK+2) = I
        J = ADJNCY(INDJ)
        FEASBL(MARK+3) = J
        MARK = MARK+4
C       ------------------------------------------------------------
C       Step (3)  Initialize REACH and ACTIVE for the edge I -> J
C       ------------------------------------------------------------
        DO 100 K=1,N
          REACH(K) = .FALSE.
100     CONTINUE
        REACH(I) = .TRUE.
        REACH(J) = .TRUE.
        ACTTOP = 1
        ACTIVE(ACTTOP) = J
200     IF (ACTTOP .GT. 0) THEN
C         ------------------------------------------------
C         Step (4)  Pop the top node from the ACTIVE stack
C         ------------------------------------------------
          K1 = ACTIVE(ACTTOP)
          ACTTOP = ACTTOP-1
```

```
                KADJST = ADJHED(K1)
                KADJEN = ADJHED(K1+1)-1
                IF (KADJEN  GE  KADJST) THEN
                  DO 300 INDK=KADJST  KADJEN
                    K2 = ADJNCY(INDK)
C                   ------------------------------------------------------------
C                   Step (5)  If K2 is not in REACH then add to REACH and ACTIVE
C                   ------------------------------------------------------------
                    IF ( NOT  REACH(K2)) THEN
                      REACH(K2) =  TRUE
                      ACTTOP = ACTTOP+1
                      ACTIVE(ACTTOP) = K2
                    ENDIF
 300              CONTINUE
                ENDIF
                GO TO 200
              ENDIF
              COUNT = 0
C             ------------------------------------------------------------
C             Step (6)  Compute FEAS(I J) and insert it into FEASBL
C             ------------------------------------------------------------
              DO 400 K=1 N
                IF ( NOT  REACH(K)) THEN
                  IF (MARK  GT  FEASIZ) THEN
                    RTCODE = 1
                    RETURN
                  ENDIF
                  FEASBL(MARK) = K
                  MARK = MARK+1
                  COUNT = COUNT+1
                ENDIF
 400          CONTINUE
```

```
C              ------------------------------------------------------------
C              Insert the size of the feasible set and a pointer to the next one
C              ------------------------------------------------------------
               FEASBL(SIZ) = COUNT
               FEASBL(NXT) = MARK
 500         CONTINUE
           ENDIF
 600     CONTINUE
C        ---------------------------------------------
C        Make the last feasible set point to the first one
C        ---------------------------------------------
         FEASBL(NXT) = 1
C        -------------------------------------------------
C        Indicate FEASBL has been successfully computed
C        -------------------------------------------------
         RTCODE = 0
         RETURN
         END
```

A.3 Subroutine ABOVE

Purpose

To compute the *above* set with respect to a given node in a digraph.

Input parameters

N - the number of nodes in the input digraph.

I - the node with respect to which the *above* set is to be computed.

ORDSIZ, ORDRER, ORDHED, ORDLNK - specifies the adjacency lists of the input digraph. See subroutine FLRORD for further details.

Output parameter

ABOV - a logical array specifying the *above* set with respect to I.

Working parameter

ACTIVE - an integer array of dimension N used as a stack in a depth first search of the digraph.

Program

```
      SUBROUTINE ABOVE(N, I, ORDSIZ, ORDRER, ORDLNK, ORDHED, ACTIVE,
     *                 ABOV)
      INTEGER N, I, ORDSIZ, ORDRER(ORDSIZ), ORDHED(N), ORDLNK(ORDSIZ),
     *        ACTIVE(N)
      LOGICAL ABOV(N)
      INTEGER K, MARK, ACTTOP
C
C     ---------------------------------------
C     Initialize ABOV and the ACTIVE stack
C     ---------------------------------------
      DO 100 K=1,N
        ABOV(K) = .FALSE.
  100 CONTINUE
      ACTTOP = 1
      ACTIVE(ACTTOP) = I
C     ------------------------------
C     Start depth first search from I
C     ------------------------------
  200 IF (ACTTOP .GT. 0) THEN
        MARK = ORDHED(ACTIVE(ACTTOP))
        ACTTOP = ACTTOP - 1
  300   IF (MARK .NE. 0) THEN
          K = ORDRER(MARK)
C         ------------------------------------------------------------
C         If K has already been seen, then ignore it, otherwise add it
C         to the Above set of I and put it on the ACTIVE stack
C         ------------------------------------------------------------
          IF ( .NOT. ABOV(K)) THEN
            ABOV(K) = .TRUE.
            ACTTOP = ACTTOP+1
            ACTIVE(ACTTOP) = K
```

```
        ENDIF
        MARK = ORDLNK(MARK)
        IF (MARK .NE. 0) THEN
          GO TO 300
        ENDIF
      ENDIF
      GO TO 200
    ENDIF
    RETURN
    END
```

A.4 Subroutine BELOW

Purpose

To compute the *below* set with respect to a given node in a digraph.

Input parameters

N - the number of nodes in the input digraph.

I - the node with respect to which the *below* set is to be computed.

ORDSIZ, ORDRER, ORDHED, ORDLNK - specifies the adjacency lists of the input digraph. See subroutine FLRORD for further details.

Output parameter

BELO - a logical array specifying the *below* set with respect to I.

Working parameters

ACTIVE - an integer array of dimension N used as a stack in a depth first search of the digraph.

REACH - a logical array of dimension N used to indicate the set of nodes that have been reached in depth first searches of the digraph. It is used to prevent duplicate searches of paths.

COMPLT - a logical array of dimension N used to indicate the set of nodes for which all outgoing edges have been considered. A node K on the ACTIVE stack is examined twice. The first time, when COMPLT(K) is FALSE, all outgoing edges are examined

and any adjacent nodes not yet in REACH are placed on the stack. The second time when COMPLT(K) is TRUE, BELO(K) is determined and K is popped from the stack.

Program

```
      SUBROUTINE BELOW(N, I, ORDSIZ, ORDRER, ORDLNK, ORDHED, ABOV,
     *                 ACTIVE, REACH, COMPLT, BELO)
      INTEGER N, I, ORDSIZ, ORDRER(ORDSIZ), ORDHED(N), ORDLNK(ORDSIZ),
     *        ACTIVE(N)
      LOGICAL ABOV(N), REACH(N), COMPLT(N), BELO(N)
      INTEGER J, K, L, ACTTOP, INDXL
C
C     ------------------------------------------------------------------------
C     Initialize BELO, COMPLT, and REACH.  The node I and any nodes in the
C     Above set are automatically removed from consideration.
C     ------------------------------------------------------------------------
      DO 100 K=1,N
        BELO(K) = .FALSE.
        COMPLT(K) = .FALSE.
        IF ((K .EQ. I) .OR. (ABOV(K))) THEN
          REACH(K) = .TRUE.
        ELSE
          REACH(K) = .FALSE.
        ENDIF
  100 CONTINUE
C     ----------------------------
C     Start searching for paths to I.
C     ----------------------------
      DO 500 J=1, N
        IF ( .NOT. REACH(J)) THEN
C         ----------------------------------------------------------------
C         J has not yet been seen.  Initialize a stack with J as the only entry.
C         ----------------------------------------------------------------
          ACTTOP = 1
          ACTIVE(ACTTOP) = J
  200     IF (ACTTOP .GT. 0) THEN
```

```
            K = ACTIVE(ACTTOP)
            IF ( NOT  COMPLT(K)) THEN
C             ------------------------------------------------------------
C             The outgoing edges from K have not yet been considered
C             ------------------------------------------------------------
              INDXL = ORDHED(K)
 300          IF (INDXL  NE  0) THEN
C               -------------------------
C               Outdegree(K) is not zero
C               -------------------------
                L = ORDRER(INDXL)
                IF ((BELO(L))  OR  (L  EQ  I)) THEN
C                 ---------------------------------
C                 There exists a path from K to I
C                 ---------------------------------
                  BELO(K) =  TRUE
                ELSEIF ( NOT  REACH(L)) THEN
                  ACTTOP = ACTTOP+1
                  ACTIVE(ACTTOP) = L
                  REACH(L) =  TRUE
                ENDIF
                INDXL = ORDLNK(INDXL)
                GO TO 300
              ENDIF
C             ------------------------------------------
C             All outgoing edges from K have been explored
C             ------------------------------------------
              COMPLT(K) =  TRUE
            ELSE
C             ------------------------------------------------------------------
C             All outgoing paths from K have been completely explored and it is
C             known which nodes adjacent from K are in the Below set of I
C             ------------------------------------------------------------------
              ACTTOP = ACTTOP-1
              IF ( NOT  BELO(K)) THEN
```

```
                INDXL = ORDHED(K)
 400            IF ((INDXL .NE. 0) .AND. ( .NOT. BELO(K))) THEN
C                 ------------------------------------------------------
C                 If any neighbor of K is in the Below set of I then add K
C                 to the Below set
C                 ------------------------------------------------------
                  IF (BELO(ORDRER(INDXL))) THEN
                    BELO(K) = .TRUE.
                  ENDIF
                  INDXL = ORDLNK(INDXL)
                  GO TO 400
                ENDIF
              ENDIF
            ENDIF
            GO TO 200
          ENDIF
        ENDIF
 500  CONTINUE
      RETURN
      END
```

A.5 Subroutine TOPSRT

Purpose

To compute a topological sort of an acyclic digraph.

Input parameters

N - the number of nodes in the input digraph.

ORDSIZ, ORDRER, ORDHED, ORDLNK - specifies the adjacency lists of the input digraph. See subroutine FLRORD for further details.

Output parameters

PERM - an integer array of dimension N used to return the topological sort.

RTCODE - returns an integer which indicates the status of the computation. The meanings of the values returned are summarized in the following table.

RTCODE	MEANING
0	a topological sort is returned in PERM
1	ERROR - the input digraph is not acyclic

Working parameters

INDEG - an integer array of dimension N used to record the indegrees of the nodes in the input digraph.

ZINDEG - an integer array of dimension N used as a stack to store the nodes with zero indegree before they are numbered.

Program

```
      SUBROUTINE TOPSRT(N, ORDSIZ, ORDRER, ORDLNK, ORDHED, INDEG,
     *                  ZINDEG, PERM, RTCODE)
      INTEGER N, ORDSIZ, ORDRER(ORDSIZ), ORDLNK(ORDSIZ), ORDHED(N),
     *        INDEG(N), ZINDEG(N), PERM(N), RTCODE
      INTEGER I, J, COUNT, INDXJ, ZINTOP
C
C     ----------------------------------
C     Compute the indegrees for all nodes
C     ----------------------------------
      DO 100 I=1,N
        INDEG(I) = 0
  100 CONTINUE
      DO 300 I=1,N
        INDXJ = ORDHED(I)
  200   IF (INDXJ .NE. 0) THEN
          J = ORDRER(INDXJ)
          INDEG(J) = INDEG(J)+1
          INDXJ = ORDLNK(INDXJ)
          GO TO 200
        ENDIF
  300 CONTINUE
```

```
C     ------------------------------------------------------
C     Construct the stack of all nodes with zero indegree
C     ------------------------------------------------------
      ZINTOP = 0
      DO 400 I=1 N
        IF (INDEG(I) EQ 0) THEN
          ZINTOP = ZINTOP+1
          ZINDEG(ZINTOP) = I
        ENDIF
  400 CONTINUE
      COUNT = 0
  500 IF (ZINTOP GT 0) THEN
        I = ZINDEG(ZINTOP)
        ZINTOP = ZINTOP-1
        COUNT = COUNT+1
C       ------------------------------------------------------
C       I is the next node numbered in the topological sort
C       ------------------------------------------------------
        PERM(COUNT) = I
        INDXJ = ORDHED(I)
  600   IF (INDXJ NE 0) THEN
C         ------------------------------------------------------------------
C         Reduce by 1 the indegrees of all nodes on the adjacency list of I
C         ------------------------------------------------------------------
          J = ORDRER(INDXJ)
          INDEG(J) = INDEG(J)-1
          IF (INDEG(J) EQ 0) THEN
C           --------------------
C           Add J to ZINDEG
C           --------------------
            ZINTOP = ZINTOP+1
            ZINDEG(ZINTOP) = J
          ENDIF
          INDXJ = ORDLNK(INDXJ)
          GO TO 600
```

```
            ENDIF
            GO TO 500
         ENDIF
         IF (COUNT .LT. N) THEN
C           ------------------------------
C           The digraph contains a cycle
C           ------------------------------
            RTCODE = 1
         ELSE
            RTCODE = 0
         ENDIF
         RETURN
         END
```

Bibliography

[1] BARKER V A (ed.) *Sparse Matrix Techniques* Lecture Notes in Mathematics no 572 Springer-Verlag Berlin 1977

[2] BONDY J A and U S R MURTY *Graph Theory with Applications* North-Holland 1979

[3] BUNCH J R and D J ROSE (eds.) *Sparse Matrix Computations* Academic Press 1976

[4] CURTIS A R and J K REID *The solution of large sparse unsymmetric systems of linear equations* J Inst Maths Applics **8** (1971) pp 344-353

[5] CUTHILL E *Several strategies for reducing the bandwidth of matrices* in Rose and Willoughby (1972) pp 157-166

[6] DUFF I S A M ERISMAN and J K REID *Direct Methods for Sparse Matrices* Oxford University Press Oxford 1986

[7] GEORGE A *Solution of linear systems of equations direct methods for finite-element problems* in Barker (1977) pp 52-101

[8] GEORGE A and J W H LIU *An implementation of a pseudo-peripheral node finder* ACM Trans Math Softw **5** (1979) pp 284-295

[9] GEORGE A and J W H LIU *Computer Solutions of Large Sparse Positive Definite Systems* Prentice Hall 1981

[10] GIBBS N E W G POOLE Jr and P K STOCKMEYER *An algorithm for reducing the bandwidth and profile of a sparse matrix* SIAM J Numer Anal **13** (1976) pp 236-250

[11] GUSTAVSON F G *Some basic techniques for solving sparse systems of linear equations* In Rose and Willoughby (1972) pp 41-52

[12] GUSTAVSON F G *Finding the block lower-triangular form of a sparse matrix* in Bunch and Rose (1976) pp 275-289

[13] HOPCROFT J E and R M KARP *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs* SIAM J Computing **2** (1973) pp 225-231

[14] HORN R and C R JOHNSON *Matrix Analysis* Cambridge University Press 1985

[15] JOHNSON C R D D OLESKY and P van den DRIESSCHE *Inherited matrix entries LU factorizations* submitted

[16] LIU J W H and A H SHERMAN *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices* SIAM J Numer Anal **13** (1976) pp 198-213

[17] MARKOWITZ H M *The elimination form of the inverse and its application to linear programming* Management Sci **3** (1957) pp 255-269

[18] MEHLHORN K *Data Structures and Algorithms 2 Graph Algorithms and NP-Completeness* Springer-Verlag 1984

[19] PACHL J K *Finding pseudoperipheral nodes in graphs* J of Computer and System Sciences **29** (1984) pp 48-53

[20] PAPADIMITRIOU Ch H , *The NP-completeness of the bandwidth minimization problem* Computing **16** (1976), pp 263-270

[21] PISSANETSKY S *Sparse Matrix Technology* Academic Press 1984

[22] ROSE D J and R A WILLOUGHBY (eds) *Sparse Matrices and there Applications* Plenum Press New York 1972

[23] ROSE D J and R E TARJAN *Algorithmic aspects of vertex elimination on directed graphs* SIAM J Appl Math **34** (1978) pp 176-197

[24] STEWART G W *Introduction to Matrix Computations* Series on Computer Science and Applied Mathematics Academic Press 1973

[25] TARJAN R E *Depth-first search and linear graph algorithms* SIAM J Computing **1** (1972) pp 146-160

[26] YANNAKAKIS M *Computing the minimum fill-in is NP-complete* SIAM J Alg Disc Meth **2** (1981) pp 77-79

VITA

Surname	SLATER	Given Names	TERENCE ARTHUR
Place of Birth	Selkirk, Man	Date of Birth	May 7, 1950

Educational Institutions Attended, with Dates of Entering and Leaving

University of Victoria	1968	to	1972
Yale University	1972	to	1973
University of Victoria	1984	to	1986
University of Victoria	1986	to	1988

Degrees, Diplomas, Etc., Awarded, with Dates and Names of Institutions

B.Sc. (Honors)	1972	University of Victoria
B.Sc.	1986	University of Victoria

Honors and Awards

The Governor General's Medal, 1972

Yale University Fellowship, 1972/73

NSERC Postgraduate Scholarships, 1986/87 and 1987/88

Partial Copyright License

I hereby grant the right to lend my thesis (the title of which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis

RECOGNITION AND ORDERING ALGORITHMS CONCERNING GLOBAL INHERITANCE IN LU FACTORIZATIONS

Author

TERENCE ARTHUR SLATER

April 14, 1988

Date