PROGRAM FLOW GRAPH DECOMPOSITION

By

SOLAYMAN MAHMOUD REFAE

Bachelor of Science

Beirut University College

Beirut, Lebanon

1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1996

PROGRAM FLOW GRAPH DECOMPOSITION

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

K. M. Hopkins

Thomas C. Collins

Dean of the Graduate College

PREFACE

The purpose of this thesis involved the implementation, validation, complexity analysis, and comparison of two graph decomposition approaches. The two approaches are: Forman's algorithm for prime decomposition of a program flow graph, and Cunningham's approach for decomposing a program digraph into graph-oriented components. To validate the two implementations, each was tested with six inputs. Comparison of these two approaches was based on these dimensions: time and space complexities, composability, repeated decomposition, and uniqueness.

Forman's algorithm appears to have four advantages over Cunningham's algorithm: 1. the algorithm overhead (i.e., the time and space complexities) was lower in Forman's algorithm; 2. Forman's algorithm yields a unique set of decomposed units, whereas Cunningham's does not; 3. in Forman's algorithm, reconstructing the original graph from the decomposed prime graphs results in the original graph that was decomposed, whereas in Cunningham's algorithm, the attempt at the reconstruction of the original graph from the decomposed parts does not always yield the graph that was decomposed; 4. Forman's approach can be used to decompose a graph until it is irreducible (all its part are primes), whereas in Cunningham's algorithm, the algorithm decomposes the graph only once even if it is still decomposable. Thus, Forman's approach could be recommended as a program flow graph decomposition algorithm.

Implementation of the decomposition techniques could help in better software comprehension and can be used in the development of some software reusability tools.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my major advisor, Dr. Mansur Samadzadeh, for his intelligent supervision, advice, guidance, assistance, and continuous encouragement that helped me to successfully complete my degree. His constructive criticism and his moral support is greatly appreciated. My appreciation extends to my other committee members Drs. Blayne Mayfield and Khaled Gasem for serving on my committee.

I would like to give my special appreciation and gratitude to my parents, who always believed in me and my abilities, for their moral and financial support and encouragement. Moreover, I wish to thank my brothers, sisters, aunts, and cousins for their moral support.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

Software development has evolved a great deal in the past decade as demand for computer technology has increased. To remain competitive in the software market, improved software development cycle resulting in relatively bug-free products are the key issues to successful marketing. Software development is impeded by the complexity of software and the lack of scalability of programming techniques and tools.

It is a well known fact that software technology lags behind hardware advancements. Software development houses do not stop after the production of their software products, rather they continue supporting and maintaining their products. Maintenance of software products include errors corrections, enhancements, and adjustments to the software [Regson 93].

An attempt to understand a piece of software has to be made before trying to maintain it. So, it is generally easier to modify, enhance, or correct a piece of software if it is more understandable [Regson 93]. Such considerations have formed the need for a quest for better software technology, especially in the area of software reusability, and better software comprehension.

Representation by directed graphs is one of the conventional approaches used for understanding the structure of a complex system [Harary et al. 65] [Lendaris 80]. The

wide applicability of directed graphs can be attributed to the fact that in many types of complex systems the directions of interactions among the elements are of importance for understanding their structures [Burns 77].

Decomposition can be effective technique for better software comprehension. This thesis work involved the implementation of two different decomposition techniques and comparing them. The remainder of this thesis report is organized as follows. Chapter II introduces the definitions of some abstract representations of programs. This chapter also presents a number of program flow graph decomposition techniques and their uses. Chapter III describes the two graph decomposition algorithms that were implemented, namely Cunningham's algorithm [Cunningham 82] and Forman's algorithm [Forman 79]. This chapter also contains the comparison and analysis of the two algorithms based on a number of inputs. Chapter IV summarizes the research and outlines the related future work.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Literature Review

The main objective of this research is to build on the established foundation related to directed graph decomposition. Several decomposition techniques of graphs have been discussed in the literature for various reasons. Hopcroft and Tarjan discuss the decomposition of graph into triconnected components [Hopcroft and Tarjan 73]. In the seventies Maddux introduced the concept of a prime program [Maddux 75]. McCabe discussed the decomposition of the control flow graph of a program into basis paths or circuits in order to calculate the software complexity of the program using the cyclomatic number [McCabe 76]. Maurer offered two algorithms, one to decompose a directed graph and the other is to decompose an undirected graph [Maurer 76].

Chinn and Thoelecke discuss the decomposition of a graph into primal graphs that could be used in the formation of another graph [Chinn and Thoelecke 83]. Muller and Spinrad discussed the modular decomposition of a graph which leads to solving problems in graph recognition and isomorphism [Muller and Spinrad 89]. Baranov and Bregman presented a method for the decomposition and synthesis of automata [Baranov and Bregman 93]. Habib et al. described the decomposition of inheritance graphs into

independent subgraphs, or modules, which are inheritance graphs themselves [Habib et al. 95]. Su offers an algorithm for decomposing a graph into cliques [Su 95].

Forman used program decomposition into primes to solve the abstract data flow analysis problem [Forman 79]; the implementation and discussion of this algorithm is part of this thesis work. Cunningham offered an algorithm [Cunningham 82] to decompose a digraph into two digraphs; the implementation and discussion of this algorithm is part of this thesis work.

2.2 Graph Theory

The definitions included in this section are conventional and they are based on the three main references on graph theory [Deo 74] [Gibbons 85] [Hopcroft and Tarjan 73] that were used for this thesis.

Geometrically, a graph is defined to be a set of points (vertices) which are interconnected by a set of lines (edges). For a graph G, we denote its vertex set by V and the edge set by E, and write G = (V, E). Figure 1 shows a directed graph G = ($\{x_1, y_1, x_2, y_2\}$, E).

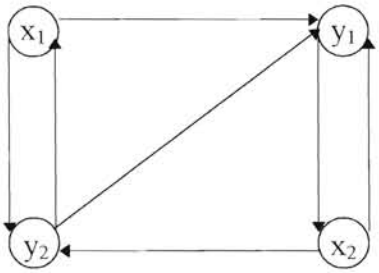


Figure 1. A sample directed graph G = ($\{x_1, y_1, x_2, y_2\}$, E)

Each edge can be specified by the two vertices (called the end points, or the tail and the head respectively) that it connects. An edge having the same vertex as both its end vertices (tail and head) is called a self loop. If the edges are ordered pairs, the graph is directed. If the edges are unordered pairs of vertices, the graph is undirected.

If E is a multiset, that is, if an edge may occur several times between the same pair of nodes, then G is a multigraph (see Appendix A for a definition). Every digraph (see Appendix A for a definition) yields an undirected graph by deleting its edge directions. A graph G is said to be connected if every distinct pair of nodes is connected by a chain. Likewise, a digraph G is said to be strongly connected (diconnected) if every node has an entry path and an exist path.

2.3 Graph Models

A graph is a general and abstract term. There are various graph models or abstract representations of a program that have been defined and used in the literature for different purposes. For example, control flow graphs (CFGs) and data flow graphs (DFGs) are used in compilers for optimization. Data dependency graphs (DDGs) can be used to measure data dependency complexity.

2.3.1 Control Flow Graph

A control flow graph is a directed graph with the nodes representing the basic blocks (a sequence of instructions with no branches) of a program. A CFG is also defined as a two-dimensional representation of a program that displays the flow of control of a program [Aho and Ullman 73]. Formally, the control flow graph of a program is a 4-

tuple, F = (N, E, *a*, *z*), where N is a finite set of nodes, E is a finite set of directed edges ($E \subseteq N \times N$), *a* is the entry node whose indegree is zero, and *z* is the exit node whose outdegree is zero [Regson 93]. Figure 2 shows a program segment and its control flow graph, where each node in the CFG represents a single executable statement in the program.

```
Program example(input, output);

1. Var
2. x: integer;
3. readln(x);
4. if (x > 0)
5.    x := x + 1;
6. end.
```

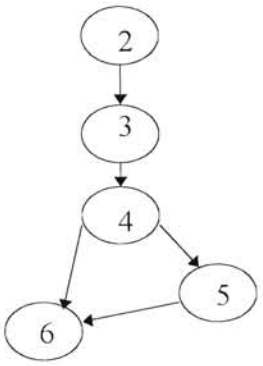


Figure 2. A program and its control flow graph

2.3.2 Data Dependency Graph

A data dependency graph (DDG) represents data dependencies among the statements in a program. A DDG is a directed graph in which the nodes represent variable definitions and the edges represent possible data dependencies [Regson 93]. The edges of a DDG represent possible dependencies between definitions. A statement that may alter the value of a variable is called a variable definition.

There are two types of data dependencies flow-order and def-order. There is a flow-order dependence edge from node X to node Y, if there exists at least one variable defined in X and used in Y and if a path exists in the corresponding CFG from X to Y. In

order for a def-order dependence edge from node X to node Y to exist a set of condition have to hold:

1- Both X and Y must define the same variable.

2- Both X and Y should be on the same path in the corresponding CFG.

3- Another node Z exists such that there exists a flow-order dependence between X and Z, and between Z and Y.

4- X occurs to the left of Y in the abstract syntax-tree of the program.

Figure 3 shows the data dependency graph corresponding to the given code segment.

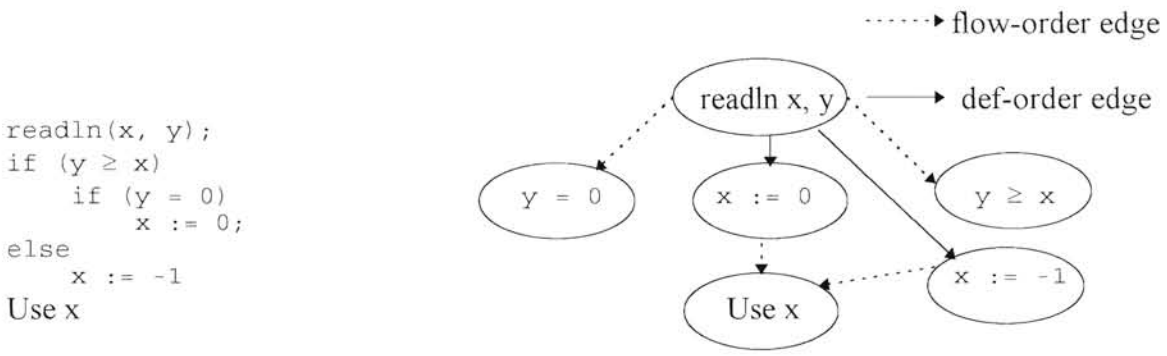


Figure 3. An if-then-else condition and its data dependency graph (Source: [Premkumar 94])

2.3.3 Program Dependency Graph

A program dependency graph (PDG) is a graph of a program in which the nodes represent the statements and the predicate expressions, and the edges incident to the nodes represent both the data and control dependencies in the program [Regson 93]. An example of a PDG is shown in Figure 4 for the sample program given in Figure 3. The flow-order dependence edges are represented by bold face arrows, the def-order

dependence edges are represented by thin face arrows, and the dashed arrows indicate the flow control in a program.

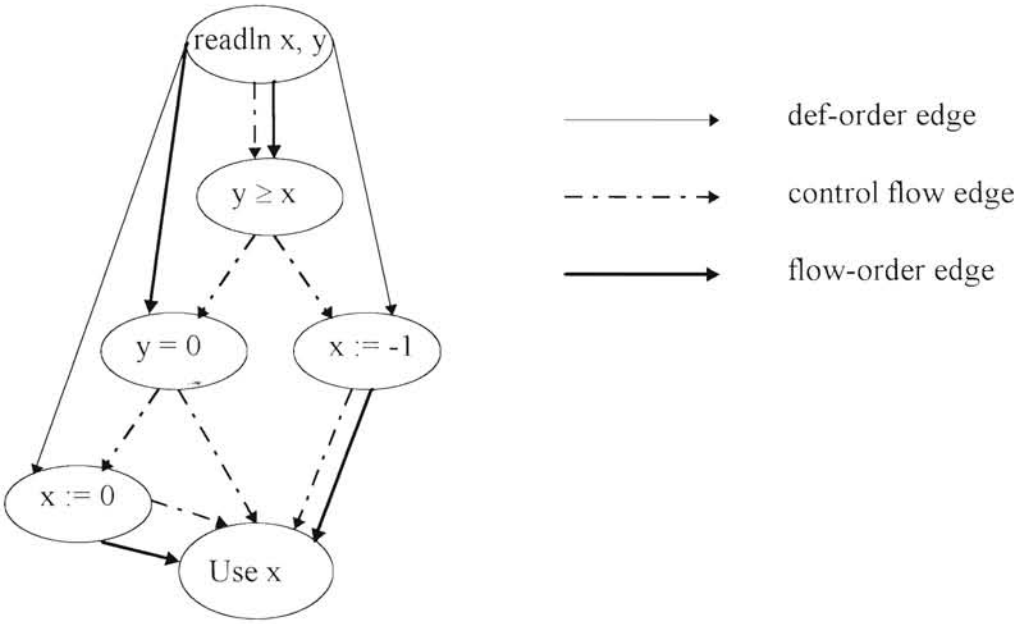


Figure 4. Program dependency graph of the program in Figure 3

2.4 Some Decomposition Techniques

Several different decomposition techniques of a graph have been discussed in the literature for various reasons. Some of those techniques are discussed in detail in the following subsections.

2.4.1 Decomposition of a Graph into Paths and Circuits

McCabe described a graph-theoretic software complexity measure called the cyclomatic number, V(G), of a graph and illustrated how it can be used to manage and control the complexity of programs [McCabe 76]. He developed a technique that

provides a quantitative basis for program modularization based on program control flow graph decomposition.

The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is V(G) = e - n + 2p. Figure 5 depicts a control flow graph G. The maximum number of linearly independent circuits in G or V(G) is 9-6+2, with p = 1.

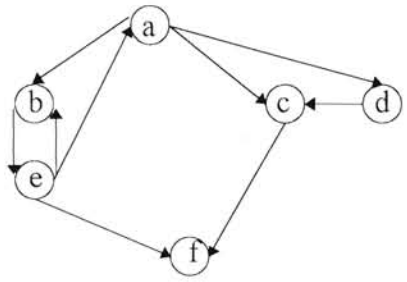


Figure 5. A program control flow graph (Source: [McCabe 76])

In a diconnected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits. Any circuit (or path) in G can be expressed as a linear combination of a basis set of independent circuits (or paths). Figure 6 shows the five paths that constitute one set of basis paths for the graph of Figure 5.

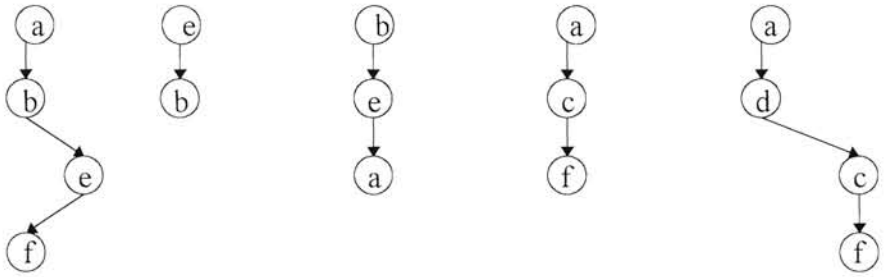


Figure 6. Five basis paths of the control flow graph of Figure 5

Thus the cyclomatic number of a graph is the number of basis circuits (or paths) that can be combined to make up any possible circuit (or path) in the graph [McCabe 76]. For instance, the path abeabebebef is expressible as (abea) + 2(beb) + (abef).

2.4.2 Decomposition of a Graph into Prime Subgraphs

A prime program is a one-in-one-out subgraph (a subgraph that has only one entry node and one exit node) that does not properly contain any one-in-one-out subgraph [Forman 79]. Prime program decomposition consists of building a hierarchy of one-in-one-out control structure elements.

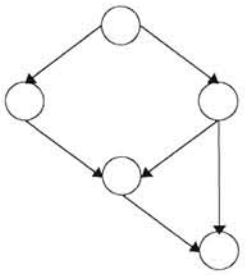


Figure 7. Flow graph of a prime program

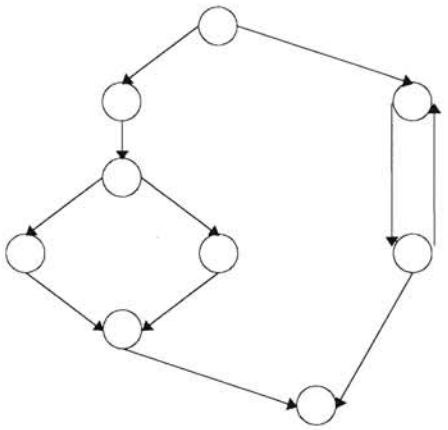


Figure 8. Flow graph of a program that is not prime

It is important to mention that all common control structures are considered prime programs [Forman 82]. Figure 7 and 8 show prime and non-prime flow graphs. Two methods of decomposing a flow graph into prime graphs are briefly discussed in the following two subsections.

2.4.2.1 Fenton-Whitty's Approach

The Fenton-Whitty scheme is a technique that is used to decompose a graph into prime subgraphs.

F =

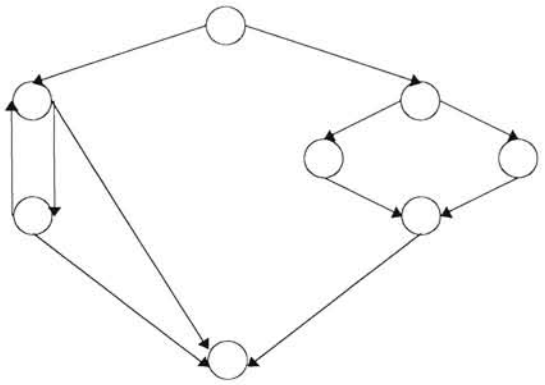


Figure 9. A program flow graph (Source: [Fenton and Whitty 86])

A flow graph allows nodes of arbitrary outdegree. In general a node of outdegree n is called an n-ary predicate node ($n \geq 2$), while nodes of outdegree 1 are called procedures nodes. The decomposition of F_2 in F_1 (where F_2 is a subflow graph of F_1) is the flow graph obtained by collapsing F_2 to a single arc (x, z'), where z' is the stop node of F_2 and x is a new procedure "replacing" F_2 [Fenton and Whitty 86]. The resulting flow graph is denoted F_1 (x for F_2). An example of a flow graph of a program and its prime subgraphs is given in Figures 9 and 10. The decomposition algorithm decomposes the

flow graph into its underlying atom (i.e., a prime) as well as its nested subflow graphs [Elliot et al. 88].

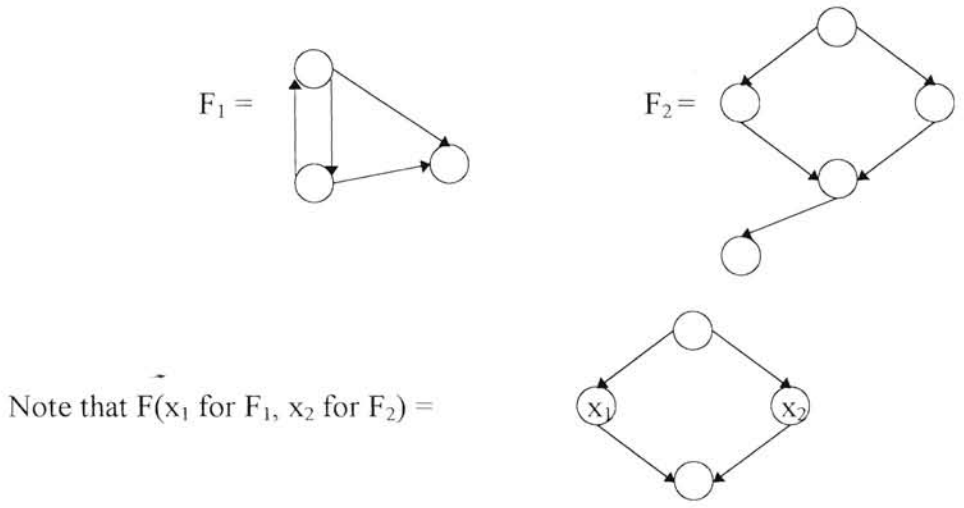


Figure 10. Decomposition of the flow graph F of Figure 9 into prime subgraphs (Source: [Fenton and Whitty 86])

2.4.2.2 Forman's Scheme

Forman describes the decomposition of m-graphs (see Appendix A for definition) into prime m-graphs [Forman 79]. M-graphs are uninterpreted flowchart schemas and are used because of their close relation to control structures in programming languages [Forman 82]. Intuitively, the prime program decomposition of an m-graph is equivalent to a set of prime programs together with a relation that forms a tree.

Let M be an m-graph. An ordered pair of arcs (x, y) is called a subprogram cutset if (x, y) is a cutset of M and all paths from x to the entry/exit contain y. A subprogram cutset separates an m-graph into two blocks. The exterior block contains the entry/exit node, while the interior block does not. M is called a prime program if it contains at least

three nodes and the only subprogram cutsets of M are either (entry(M), exit(M)) or subprogram cutsets (x, y) such that head(x) = tail(y). Figure 12 shows the prime program decomposition of the m-graph given in Figure 11.

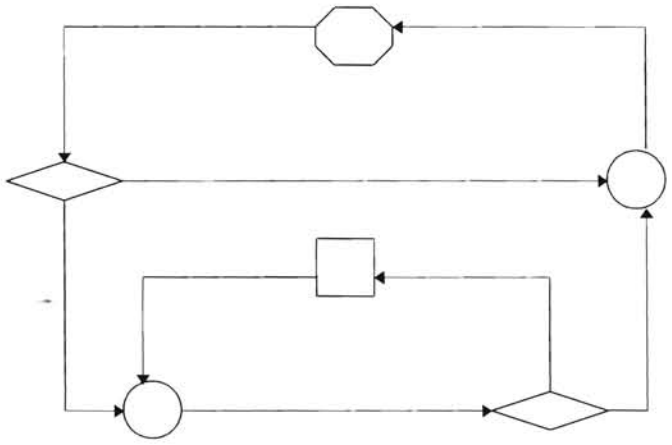


Figure 11. An example of an m-graph

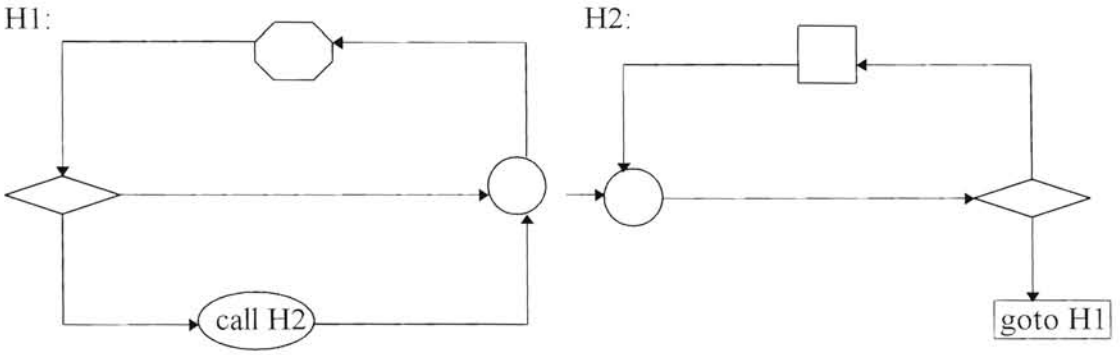


Figure 12. The two m-graphs resulting from decomposing the m-graph of Figure 11

Forman proposed to decompose m-graphs into a hierarchy of primes [Forman 79]. Once the hierarchy is formed, analysis can be performed on the hierarchy rather than on the original m-graph. The hierarchy is formed by finding subprogram cutsets and replacing the interior with a special kind of assignment node, which is termed a call node,

and making the interior an m-graph to which the call node points. When this operation can no longer be performed upon the hierarchy, the result is called prime program decomposition, because all the m-graphs in the hierarchy are primes [Forman 82].

The concept of prime programs is applied as a decomposition technique to the global data flow analysis problem [Forman 82].

CHAPTER III

IMPLEMENTATION AND COMPARISON

In the literature, different approaches for decomposing a digraph have been discussed. Some of those techniques were outlined in Chapter II. This chapter discusses the properties and the implementation platform of the two graph decomposition algorithms that were implemented as part of this thesis. A comparison of the two algorithms concludes this chapter.

3.1 Forman's Algorithm

3.1.1 Overview

In the decomposition algorithm presented by Forman [Forman 79], the problem of decomposing an m-graph (see Appendix A for a definition) into subgraphs is transformed from the set of m-graphs to a set of tree structures, which are called "spanning charts" (see Appendix A for a definition). The algorithm consists of three steps:

1- Build the spanning chart.

2- Build the tier-i paths.

3- Test the tier-i paths for subprogram cutsets.

3.1.2 Description of the Algorithm

The algorithm conforms with the three steps described in the last subsection. Step 1 transforms the m-graph into a spanning chart. Step 2 marks each node with the number of the tier-i path (see its definition in Appendix A) to which the node belongs. Step 3 (process_tier) searches those paths that belong to the same set of tier-i paths for subprogram cutsets.

The basic algorithm, which is equal to Step 3, works just the way the definitions of prime program and prime program decomposition imply it should. There are two sets of m-graphs, PRIMES and LEAVES, that form a hierarchical m-graph that is equivalent to the m-graph being decomposed. LEAVES contains the m-graphs that may not be prime. Each member of LEAVES is processed by finding subprogram cutsets, removing the interior, and placing the interior in LEAVES. Primes contains the prime m-graphs. The first level of a stepwise refinement looks as follows.

```
PRIMES := ∅
LEAVES := {M}
While LEAVES ≠ ∅ do
      Q := member(LEAVES) "selects random member"
      LEAVES := LEAVES -{Q}
      (*) "Find all subprogram cutsets of Q.
           For each subprogram cutset found, place
           the interior in LEAVES and remove the interior
           from Q."
      od
```

Subprogram cutsets are found by testing each member of PAIRS, the set of candidate ordered pairs for subprogram cutsets. This leads to a second level of refinement for the step marked with an (*) above.

```
PAIRS := {(x, y) | x ∈ arcs(Q) and y ∈ arcs(Q)
            and x ≠ y and head(x) ≠ tail(y)
            and (x, y) ≠ (entry(Q), exit(Q))}
While PAIRS ≠ ∅ do
            (x, y) := member(PAIRS)
            PAIRS := PAIRS - {(x, y)}
            "If (x, y) is a subprogram cutset of Q, then
             place the interior of (x, y) in LEAVES and
             remove the interior from Q."
      od
```

The whole algorithm may be stated with the aid of the predicate IS_CUTSET. IS_CUTSET((x, y), M) is true if and only if the ordered pair (x, y) is a subprogram cutset of the m-graph M.

The algorithm uses m-graphs (see Appendix A for a definition) to represent a program flow graph. An m-graph uses four types of nodes to represent a program statement. The four different node types that comprise an m-graph are:

1- The entry/exit node.

2- The predicate node.

3- The join node.

4- The assignment node.

An m-graph representing a subprogram flow graph uses two more nodes in addition to the four mentioned above. These two node types are:

1- The call node.

2- The goto node.

As to the representations of the control structure (see Appendix A for a definition), some of them are different in Forman's algorithm from what they are in Cunningham's algorithm. Some of the most common ones are shown in Figure 13.

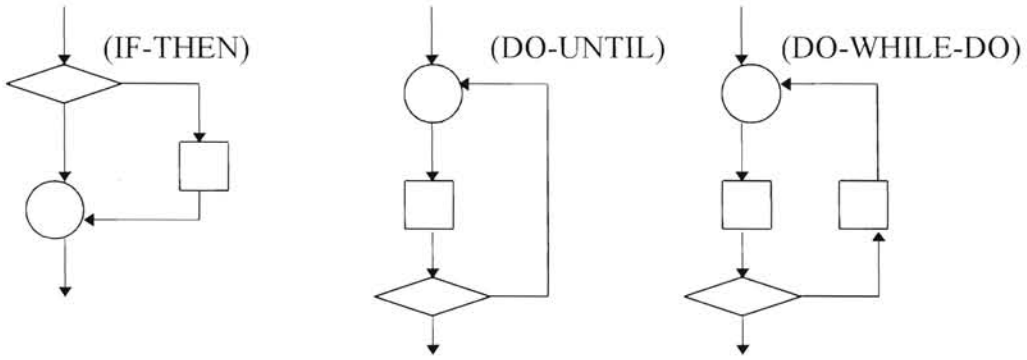


Figure 13. Some control structure representations as used by Forman (Source: [Forman 79])

3.2 Cunningham's Algorithm

3.2.1 Overview

A decomposition of diconnected digraphs has been described by Cunningham [Cunningham 82]. In his paper, Cunningham deals with finite and simple digraphs (see Appendix A for a definition), where E(G) or the edge set is a set that does not contain self loops, and the vertex set is a finite set called V(G). An improvement of Cunningham's algorithm in terms of its complexity was introduced by Bouchet [Bouchet 87]. In his paper, Bouchet discussed a different way for finding a split of a digraph based on Cunningham's algorithm.

3.2.2 Description of the Algorithm

The decomposition problem as it is described by Cunningham [Cunningham 82] can be stated using an example as follows:

Given edges (x_1, y_1), (x_2, y_2) of G and a set $S \subseteq V(G)$ satisfying $x_1, y_2 \in S$, $x_2, y_1 \notin S$, and $|S| \geq 2$, find, if there is a split $\{v_1, v_2\}$ of G such that $x_2, y_1 \notin v_1$ and $v_1 \subseteq S$.

Cunningham's algorithm [Cunningham 82] to solve the above-mentioned decomposition problem is presented below.

```
begin
    S := {x1, y2};              /* initialize the set S to contain nodes
                                   x1 and y2, S will contain one of
                                   the split sets at the end of the
                                   algorithm */
    T := S;                     /* initialize the set T to be equal to the
                                   set S, T is used as a set variable
                                   throughout the algorithm */
    while T ≠ ∅ do              /* while the algorithm has not finished
                                   splitting the graph */
      Select p ∈ T;             /* choose p to be one of T's elements */
      T := T\{p};               /* delete p from the set T */
      for q ∈ V(G) do           /* choose q to be one of V(G)'s elements,
                                   q shouldn't be in V(G) and the
                                   predicate P should be true in order for
                                   q to be one of the nodes that could
                                   be split from the original graph */
        if q ∉ S and [P(x1, y1, p, q) or
                                  P(x2, y2, q, p)] then
          S := S ∪ {q};         /* if q could be split from the original
                                   graph, then q will be added to the sets
                                   S and T */
          T := T ∪ {q};
        endif
      endfor
    endwhile
end
```

The following definitions [Cunningham 82] are necessary for the algorithm. If $(x, y) \in E(G)$ and $p, q \in V(G)$, we say that $P(x, y, p, q)$ is true if the following condition fails: $(p, q) \in E(G)$ if and only if $(p, y), (x, q) \in E(G)$. Likewise, $\delta(A)$: refers to the set $\{(x, y): (x, y) \in E(G), x \in A, y \notin A\}$. Cunningham's algorithm of splitting a digraph G into G_1 and G_2, is based on the following proposition:

Let G be a diconnected digraph, let $S \subseteq V(G)$ such that $|S| \geq 2$ and $|V(G)\backslash S| \geq 2$, where $V(G)\backslash S$ is the set containing all vertices in V(G) but not in S, and let $(x_1, y_1) \in \delta(S)$, $(x_2, y_2) \in \delta(V(G)\backslash S)$. Then $\{S, V(G)\backslash S\}$ is a split of G if and only if there does not exist $p \in S$, $q \in V(G)\backslash S$ such that $P(x_1, y_1, p, q)$ or $P(x_2, y_2, q, p)$ is true.

Upon termination of the algorithm, if we had $x_2 \in S$, $y_1 \in S$, or $|S| = n - 1$, then it can be said that there is no split, otherwise the result of the split will be $\{S, V(G)\backslash S\}$.

In order to improve the complexity of that algorithm, Bouchet [Bouchet 87] decomposed it into three parts:

1- The original program that initializes the set S calls subprogram FILLSTACK and subprogram SEPAR.

2- Subprogram FILLSTACK, which is called when the program is initiated to place some vertices in S', and that will contain one of the split sets at the end of the algorithm.

3- Subprogram SEPAR, which is called to check if a split has occurred.

Subsequently, Bouchet [Bouchet 87] shows that SEPAR and FILLSTACK, which are $O(n^3)$, improve his algorithm over Cunningham's [Cunningham 82], which is $O(n^4)$.

This thesis was concerned with implementing the algorithm introduced by Cunningham [Cunningham 82]. As to the representation of the nodes, Cunningham's

algorithm represents all program statements with the same node. So, it only has one node type.

In Cunningham's algorithm, the representation of some the control structures (See Appendix A for a definition) is different than Forman's. Figure 14 presents some of these control structures as they are presented and processed in Cunningham's algorithm.

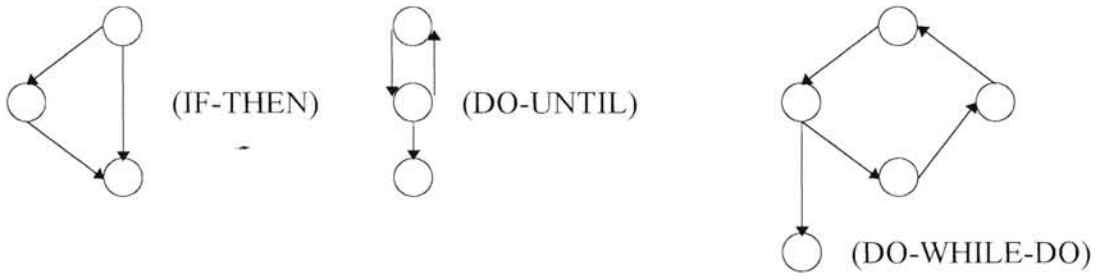


Figure 14. Some control structure representations as used by Cunningham (Source: [Cunningham 82])

3.3 Implementation Platform and Environment

The two algorithms were implemented on a Sequent Symmetry S/81 under the DYNIX/ptx operating system [SEQ 90], and C was used as the programming language.

The Symmetry S/81 is a mainframe-class multiprocessor system developed by Sequent Computer System, Inc. Sequent S/81 is a shared memory, tightly-coupled multiprocessor. It also has hardware supporting mutual exclusion. The load is balanced and the tasks are distributed in a multi-user environment to increase throughput and improve response time. UNIX compatible software can run on the Symmetry S/81 without modification or with slight modification.

3.4 Time and Space Complexities

This section discusses the complexity of the two graph decomposition techniques that were discussed in Chapter III.

The complexity of Forman's algorithm is taken from Forman's thesis [Forman 79]. Steps 1 and 2 of the Algorithm (see Subsection 3.1.2) are linear or $O(\lambda)$, where λ is the number of arcs in the input m-graph. Step 3 is no worse than $O(\lambda^3)$. Because the m-graphs are represented by their adjacency matrix, the space complexity is $O(n^2)$, where n is the number of nodes in the input m-graph.

On the other hand, the time complexity of Cunningham's algorithm (see Subsection 3.2.2) for decomposing a digraph G is $O(n^4)$, where n is the number of nodes in G [Cunningham 82]. In his paper, Bouchet [Bouchet 87] introduced a new way called local complementation to improve the time complexity of Cunningham's algorithm for finding a split of a digraph. Bouchet succeeded in decreasing the time complexity from $O(n^4)$ to $O(n^3)$. The space complexity of the algorithm is $O(n^2)$, where n is the number of the nodes in the graph. This is because a digraph is represented by its adjacency matrix.

3.5 Comparison

This section compares the two graph decomposition approaches based on the following criteria [Regson 93]:

(a) Composability - i.e., whether or not any of the resulting units of decomposition is reusable;

(b) Repeated decomposition - i.e., whether the decomposition process can be applied repeatedly;

(c) Uniqueness - i.e., whether the decomposition technique yields a unique set of decomposed units;

Prime subgraphs, resulting from the decomposition of the flow graph of a program, are generally good candidates for reusability. Primes can be replaced by single nodes in the original flow graph. In the prime decomposition of a flow graph, when using Forman's approach, reconstructing the decomposed graph from the decomposed parts (or reusable parts) can be done by replacing the single nodes, introduced in the decomposition process, with the subgraphs to which they correspond.

The decomposition of a diconnected digraph of a program, using Cunningham's scheme, is a graph-theoretic concept that is not closely related to the semantic structure of the program. Hence, the potential for the reuse of the decomposed parts is not high. When a digraph is decomposed into two digraphs in this method, the original digraph can be obtained by the operation of union. This can be done by deleting the marker, which is introduced during the decomposition process (see Subsection 3.2.2 for details), from both of component digraphs and combining them together as follows: if $(x, v) \in G_1$ and $(v, y) \in G_2$, then $(x, y) \in G$, where $x \in G_1$, $y \in G_2$ and v is the marker; or, if $(v, x) \in G_1$, $(y, v) \in G_2$, then $(y, x) \in G$.

The prime program flow graph decomposition technique utilizing Forman's algorithm involves repeated decomposition in the process of building the prime decomposition (as explained in Chapter III). In Cunningham's algorithm, this repeated decomposition can be applied to one or both of the decomposed digraphs to find out whether they are further reducible.

Forman's approach uses triconnected components for building the prime tree [Tarjan and Valdes 80]. Hopcroft and Tarjan show that the triconnected components of a graph are unique [Hopcroft and Tarjan 73]. Therefore, decomposing an m-graph into primes will result in a unique prime subgraphs.

CHAPTER IV

SUMMARY AND FUTURE WORK

The main purpose of this thesis was to implement the decomposition techniques proposed by Cunningham [Cunningham 82] and Forman [Forman 79]. Two programs were developed, the first one decomposes a digraph into two digraphs, whereas the second one decomposes a graph into prime graphs. The two decomposition approaches were compared and analyzed based on a number of specified dimensions.

Possible future work to extend and utilize the work done in this thesis includes the following. A technique can be created to construct composite graphs from the previously decomposed subgraphs. By the same token, the decomposed subgraphs of program flow graphs can be used as reusable units in the construction of other programs. Thus, a repository of decomposed units (or subprograms) can serve as a software parts catalog stored in the form of flow graphs (or program codes). Another area of future work would be in extending the programs, presented in this thesis to run in the X-windows environment. Such programs can present the output graphs pictorially and accept a graphical input instead of an adjacency matrix.

REFERENCES

[Aho and Ullman 73] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Baranov and Bregman 93] S. Baranov and L. Bregman, "Automata Decomposition and Synthesis with PLAM," *Microprocessing and Microprogramming*, Vol. 38, pp. 759-766, September 1993.

[Bouchet 87] A. Bouchet, "Digraph Decompositions and Eulerian Systems," *SIAM Journal of Algebraic and Discrete Methods,* Vol. 8, No. 3, pp. 323-337, July 1987.

[Burns 77] J. R. Burns, "Converting Signed Digraphs to Forrester Schematics and Converting Forrester Schematics to Differential Equations," *IEEE Transactions on Systems, Man, and Cybernetics,* Vol. SMC-7, No. 10, pp. 695-707, October 1977.

[Chinn and Thoelecke 83] P. Z. Chinn and P. A. Thoelecke, *Factoring Graphs into Primal Graphs*, Department of Mathematics, Humboldt State University, Technical Report, Arcata, CA, 1983-84.

[Cunningham 82] W. H. Cunningham, "Decomposition of Directed Graphs," *SIAM Journal of Algebraic and Discrete Methods*, Vol. 3, No. 2, pp. 214-228, June 1982.

[Deo 74] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[Elliot et al. 88] J. J. Elliot, N. E. Fenton, S. Linkman, G. Markham, and R. Whitty (Editors), "Structured-Based Software Measurement," Alvey Project SE/069, 1988, Department of Electrical Engineering, South Bank, Polytechnic, Borough Road, London, UK.

[Fenton and Whitty 86] N. E. Fenton and R. Whitty, "Axiomatic Approach to Software Metrication Through Program Decomposition," *The Computer Journal*, Vol. 29, No. 4, pp. 330-339, 1986.

[Forman 79] I. R. Forman, "On the Decomposition of Programs into Primes," Ph.D. Thesis, Computer Science Department, University of Maryland, College Park, MD, 1979.

[Forman 82] I. R. Forman, "Global Data Flow Analysis by Decomposition into Primes," *Proceedings of the Sixth International Conference on Software Engineering*, pp. 386-392, Tokyo, Japan, September 1982.

[Gibbons 85] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, New York, NY, 1985.

[Habib et al. 95] M. Habib, M. Huchard, and J. Spinrad, "A Linear Algorithm to Decompose Inheritance Graphs into Modules," *Algorithmica*, Vol. 13, No. 6, pp. 573-591, June 1995.

[Harary et al. 65] F. Harary, R. Z. Norman, and D. Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs*, Wiley, New York, NY, 1965.

[Hopcroft and Tarjan 73] J. E. Hopcroft and R. E. Tarjan, "Dividing a Graph into Triconnected Components," *SIAM Journal of Computing*, Vol. 2, No. 3, pp. 135-158, September 1973.

[Lendaris 80] G. G. Lendaris, "Structural Modeling: A Tutorial Guide," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-10, No. 12, pp. 807-840, June 1980.

[Maddux 75] R. A. Maddux, "A Study of Computer Program Structure," Ph.D. Thesis, Computer Science Department, University of Waterloo, Waterloo, Canada, July 1975.

[Maurer 76] M. C. Maurer, "Unite de la Decomposition d'un Graphe en Joint Suivant un Graphe Joint-Irreductible, d'une Famille de ses Sous-Graphes," *C. R. Acad. Sci. Paris*, Vol. 283, pp. 289-292, September 1976.

[McCabe 76] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engneering*, Vol. SE-2, No. 4, pp. 308-320, December 1976.

[Muller and Spinrad 89] J. H. Muller and J. Spinrad, "Incremental Modular Decomposition," *Journal of the Association for Computing Machinery (JACM)*, Vol. 36, No. 1, pp. 1-19, January 1989.

[Premkumar 94] J. Premkumar, "Translation of Simple C Programs to Program Dependence Webs," Masters Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, July 1994.

[Regson 93] C. P. Regson, "Program Flow Graph Decomposition as a Model of Software Comprehension," Masters Thesis, Computer Science Department, Oklahoma State University, Stillwater, OK, July 1993.

[Sedgewick 88] R. Sedgewick, *Algorithms*, Addison-Wesley Publishing Company, Reading, MA, 1988.

[SEQ 90] Symmetry Multiprocessor Architecture Overview. Sequent Computer Systems, Inc., 1990.

[Su 95] Xy Su, "An Algorithm for the Decomposition of Graphs into Cliques," *Journal of Graph Theory*, Vol. 20, No.2, pp. 195-202, 1995.

[Tarjan and Valdes 80] R. E. Tarjan and J. Valdes, "Prime Subprogram Parsing of a Program," *Proceedings of the Seventh Annual Symposium on the Principles of Programming Languages*, pp. 95-105, Las Vegas, NV, January 1980.

APPENDICES

APPENDIX A

GLOSSARY

CFG: Control flow graph.

Control Structures: The common control structures are: do-while-do loop, do-until loop, if-then, if-then-else, and while-do loop.

DDG: Data Dependency graph.

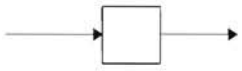
Diconnected: Strongly connected.

Digraph: Directed graph.

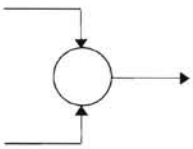
Flow chart: A pictorial representation of the algorithm of a program.

M-graph: A diconnected graph that contains a unique one-in-one-out node called entry/exit node and is constructed from the following four types of nodes:

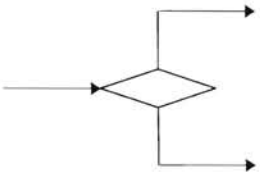
Entry/exit node



Assignment node



Join node



Predicate node

Multigraph: A graph in which an edge may occur more than one time between the same pair of nodes. A graph with parallel edges.

Multiset: A set where elements can occur several times.

PDG: Program dependency graph.

Primal Graph: A component graph; a graph can be written as a sum of distinct primal graphs.

RFG: Reducible flow graph.

Simple Digraph: A digraph that has no self loops or parallel edges.

Software Complexity: The level of difficulty to understand, change, and maintain software.

Spanning Chart: A spanning tree of a graph G = (V, E) is a graph G′ = (V, E′), where G′ is a tree that includes every node in G, and E′ is a subset of E.

Tier-i: A path from a goto node whose corresponding join node is on a tier-(i-1) path to the first possible predicate node on a tier-j path where $j \leq i-1$. The unique path from the entry of the m-graph to its exit is called the tier-0 path.

APPENDIX B

TRADEMARK INFORMATION

DYNIX/ptx: A registered trademark of the Sequent Computer System, Inc.

Symmetry S/81: A registered trademark of the Sequent Computer System, Inc.

UNIX: A registered trademark of AT&T.

APPENDIX C

PROGRAM LISTING

The following are the two files that were used in the implementation of the two algorithms.

cunn.c - This file contain the implementation of Cunningham's algorithm [Cunningham 82].

forman.c - This file contain the implementation of Forman's algorithm [Forman 79].

The following is the cunn.c file.

```
/*************************************************************************

This program implements Cunningham's algorithm for digraph decomposition
[Cunningham 82].  The program takes as input a digraph in the form of an
adjacency matrix.   Then it tries to decompose that graph into two
digraphs.  After that, if a split has happened, it will print the two
sets resulting from splitting.

*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define Maxnd  50                       /* maximum number of nodes in an
                                           input graph */

int     checkP();
int     check_digraph();
void    build_graph();
void    algorithm();
void    split_check();
void    build_sub();
void    relabel();
void    print_subgraphs();

FILE  *in;

int graph[Maxnd][Maxnd],                /* representation of a graph as an
                                           input adjacency matrix */
```

```
      S[Maxnd],                       /* contains the vertices of the set
                                         S */
      S2[Maxnd],                      /* contains the vertices in the set
                                         V(G)\S */
      T[Maxnd],                       /* contains the vertices in the set
                                         T */
      Vertice[Maxnd],                 /* contains all the vertices in the
                                         graph */
      Sub1[Maxnd][Maxnd],             /* adjacency matrix of the
                                         decomposed graph, S */
      Sub2[Maxnd][Maxnd],             /* adjacency matrix of the
                                         decomposed gaph, V(G)\S */
      error,                          /* a flag that is turned on when
                                         something goes wrong or if there
                                         is a wrong input */
        nodenbr,                      /* number of nodes in the graph */
      Snode,                          /* number of nodes in S */
      S2node,                         /* number of nodes in S2 */
      Tnode;                          /* number of nodes in T */

main()
 {
   int     relabel_time = 0,          /* contains the number of
                                         relabelings done */
           no_split = 0;              /* a flag to check whether a split
                                         has occurred */

                                      /* open input file */
     if((in = fopen("/v/refae/thesis/inp1","r")) == NULL)
      { printf("Sorry, the input file cannot be opened\n");
        exit(1);
      }
     error = 0;                       /* initialize error flag */
       build_graph(&error);           /* call procedure to start reading
                                         the input and building the graph
                                         */
     if(error == 0)                   /* if no error has occurred, call
                                         the procedure to do the
                                         split and then call the split-
                                         check procedure to see whether a
                                         split has occurred */

       {
       algorithm();
       if(split_check() == 0)         /* check whether a split has
                                         occurred */

           no_split = 1;
           while((relabel_time < (nodenbr - 1)) && (no_split == 1))
                                      /* while the number of relabelings
                                         done is less than the number of
                                         the input graph nodes and
                                         no split has occurred yet perform
                                         the following */

             {
                 relabel();           /* call the relabeling procedure */
                 relabel_time++;
                                      /* check whether a split has
                                         occurred */
                 if(split_check() == 1)
                       no_split = 0;
             }
           if(no_split == 1 )
                 printf("There is no split.\n");
```

```
            else                        /* if a split has occurred, print
                                           the two graphs resulting from the
                                           decomposition */
                    print_subgraphs();
        }
      fclose(in);                       /* close input file */
   }

/*************************************************************************

This procedure reads the input graph to be decomposed from the input
file  into an adjacency matrix.

*************************************************************************/

void build_graph(int *error)
  {
   int
      node1,node2,                      /* they represent the first and
                                           second node of some edge of the
                                           input graph, respectively */
      i,j;                              /* used as looping variables */
      fscanf(in,"%d\n",&nodenbr);       /* read the number of nodes */

                                        /* initialize the array of vertices
                                           */
       for (i= 0;i<nodenbr;i++)
            Vertice[i] = i+1;
        *error = 0;
      while((fscanf(in,"%d %d\n",&node1,&node2) != EOF) &&
                  (*error == 0))
      {
                                        /* build the graph */
          if((node1>nodenbr)||(node1<1)||(node2>nodenbr)||(node2<1))
            {                           /* improper input */
             printf("Invalid node number, program terminated.\n ");
                    *error = 1;
            }
           else                         /* checking for self loop */
            if(node1 == node2)
             {                          /* input graph has a self loop */
              printf(" Self loop not accepted, program terminated.\n");
              *error = 1;
             }
            else
            {
                                        /* if there is an edge going from
                                           node1 to node2 */
                  graph[node1][node2] = 1;
            }
      }
   if(check_digraph())                  /* check whether the input graph is
                                           a digraph */
   {
    printf("The input graph is not a digraph, program terminated.\n");
    *error = 1;
   }
  }
/*************************************************************************

This procedure checks whether the input graph is a digraph.
```

```
**************************************************************************/

int check_digraph()
  {
   int      notdigraph=0,          /* to check whether the input graph
                                      is a digraph */
            i,j,
            Sum;                   /* used to check the sum of the
                                      columns and rows of the input
                                      graph whether every one of them
                                      is bigger than 1, which means
                                      that every node has at least one
                                      entry and one exit */

      for(i=1;(i<=nodenbr)&&(!notdigraph);i++)
      {
       Sum = 0;                    /* check whether every node in the
                                      input graph has at least one exit
                                      */

       for(j=1;j<=nodenbr;j++)
        Sum += graph[i][j];
        if(Sum>=1)                 /* if all nodes in the input graphs
                                      have at least one exit node, then
                                      check whether they also have at
                                      least one entry arc or vertix.
                                      Otherwise, they won't be
                                      reachable and the input graph
                                      won't be a digraph */

         for(j=1;j<=nodenbr;j++)
          Sum += graph[j][i];
                                   /* if one of the nodes in the input
                                      graph doesn't have an entry or
                                      exit, then the input graph is not
                                      a digraph */
         if(Sum == 0) notdigraph = 1;
      }
return notdigraph;
 }

/**************************************************************************

The following procedure applies Cunningham's decomposition algorithm
[Cunningham 82].

**************************************************************************/

   void algorithm()
   {
    int
      p,                           /* the vertex chosen from T */
      q,                           /* the vertex chosen from V(G) */
      node_in_S,                   /* flag to check whether a vertex is
                                      in S */
      qinS,                        /* to check whether q is in S */
      i,j,k;                       /* used as looping variables */

                                   /* initializing the sets S and T
                                      to contain x1 y2 */

        S[0]=T[0]=1;
        S[1]=T[1]=nodenbr;

        Snode = Tnode = 2;
```

```
while(Tnode > 0)                /* while the algorithm has not
                                   finished splitting the graph */
{
      p = T[Tnode - 1];         /* choose p to be an element from T,
                                   because we will try every
                                   element in T to check whether it
                                   can be split from the original
                                   graph and added to the set S, so
                                   at the end of the algorithm the
                                   set S would be one of the split
                                   sets */
      T[Tnode - 1] = 0;         /* deleting element p from T
                                   because it was split from the
                                   graph */
      Tnode--;                  /* decrement the number of nodes in
                                   T by 1 */
      vertnb = nodenbr;
        for(i= 0;i<nodenbr;i++)
         {
            q = Vertice[i];     /* choose q to be one of the
                                   vertices */
                                /* check if q is in S */
            qinS = 0;
            for(j= 0;j<Snode;j++)
             {
                  if( q == S[j]) qinS = 1;
             }
                                /* if q is not in S and one of the
                                   predicates P(x1, y1, p, q) or
                                   P(x2, y2, q, p) is true, then
                                   node q would be a good candidate
                                   to be added to the split set S
                                   (refer to page 4 to see how to
                                   check whether P is true ) */

            if((qinS==0)&&(checkP(1,2,p,q)||(checkP(3,4,q,p))))
             {
                  S[Snode] = q;
                  T[Tnode]=q;
                  Snode++;      /* increment the number of nodes in
                                   S by one after adding node q to
                                   it */
                  Tnode++;      /* increment the number of nodes in
                                   T by one after adding node q to
                                   it */
             }
          }/* end of for */
}/* end of while */

                                /* fill the V(G)/S nodes in the set
                                   S2 */
k = 0;
   for(i=0;i<nodenbr;i++)
     {
```

```
                                        /* check whether the node in V(G) is
                                           in S */
          node_in_S = 0;
          for(j=0;(j<Snode)&&(node_in_S==0);j++)
            if(Vertice[i] == S[j])
                  node_in_S = 1;
          if(node_in_S != 1)            /* if the node is not in S, add it
                                           to S2, which conatins the nodes
                                           of V(G)\S */
              S2[k++] = Vertice[i];
        }
      S2node = nodenbr - Snode;         /* number of nodes in S2 is equal to
                                           the number of the input graph
                                           nodes minus the number of nodes
                                           in S1 */
   }
*************************************************************************

This procedure checks whether there was a split.  Then, if a split has
occurred the procedure calls build_sub() procedure, to build the graphs
of the two split sets, S and V(G)\S.

*************************************************************************/

 void split_check()
  {
    int     No_Split,               /* flag to check whether a split has
                                       occurred */
            node_in_S,              /* flag to check if a node is in the
                                       set S */
            i,j;                    /* used as looping variables */

                                    /* check if x2, y1 are in S, or if S
                                       has n-1 nodes, i.e., no split has
                                       occurred */
       No_Split = 0;                /* initialize the No_split variable
                                       */
      if(Snode == (nodenbr -1))     /* if number of nodes in S after
                                       the split is equal to (n-1),
                                       where the number of nodes in the
                                       graph, then no split has occurred
                                       */
            No_Split = 1;           /* turn the split flag on, meaning
                                       that no split has occurred */
      else

          for(i=0;i<Snode;i++)      /* if the set S contain y1 or x2
                                       after running the program, then
                                       no split has occurred */
        {
         if((S[i] == 2)||(S[i]==3))
            No_Split = 1;
         }
         if(No_Split == 1)          /* if no split has occurred */
            printf("There is no Split\n");
         else
            {                       /* if a split has occurred call
                                       build_sub procedure to start
                                       building the graphs of the two
                                       split sets, S and V(G)\S */
```

```
            build_sub();
          }
    }

/*************************************************************************

This procedure builds the graphs of two split sets, S and V(G)\S.

*************************************************************************/
void build_sub()
 {
  int i,j;                              /* used as looping variables */

  for(j=0;j<Snode;j++)
   {
    for(i=0;i<Snode;i++)                /* using the original input graph
                                           connections among nodes, the two
                                           split graphs are built as two
                                           adjacency matrices.  Sub1 will
                                           represent the adjacency matrix
                                           of the first subgraph and Sub2
                                           will represent the second one */
     Sub1[S[j]][S[i]] = graph[S[j]][S[i]];
   }
  for(j=0;j<S2node;j++)
   {
    for(i=0;i<S2node;i++)
     Sub2[S2[j]][S2[i]] = graph[S2[j]][S2[i]];
   }
                                        /* add marker to the two Subraphs */
     nodenbr++;
                                        /* add the marker node to nodes in
                                           Sub1 and Sub2 */
     S[Snode++] = S2[S2node++] = nodenbr;
                                        /* connect the nodes in Sub1 to the
                                           marker */
     for(i=0;i<(Snode-1);i++)
      {
        for(j=0;j<(S2node-1);j++)
         {
          if(graph[S[i]][S2[j]] == 1)
           Sub1[S[i]][S[Snode-1]] = 1;
          if(graph[S2[j]][S[i]] == 1)
           Sub1[S[Snode-1]][S[i]] = 1;
         }
      }
                                        /* connect the nodes in Sub2 to the
                                           marker */
     for(i=0;i<(S2node-1);i++)
      {
        for(j=0;j<(Snode-1);j++)
         {
          if(graph[S2[i]][S[j]] == 1)
           Sub2[S2[i]][S2[S2node-1]] = 1;
          if(graph[S[j]][S2[i]] == 1)
           Sub2[S2[S2node-1]][S2[i]] = 1;
         }
      }
   }
 }
/*************************************************************************

The following procedure print the two decomposed subgraphs.  It will
print them as adjacency matrices
```

```
**************************************************************************/

void print_subgraphs()
{
 int i,j;                              /* used as looping variables */

   printf(" Subgraph 1 :\n");          /* print subgraph 1 as an adjacency
                                          matrix showing the edges among
                                          nodes */
   for(i=0;i<=nodenbr;i++)
    {
     for(j=0;j<=nodenbr;j++)
      {
      if(Sub1[S[i]][j] == 1)
        printf("%d -> %d\n",S[i],j);
      }
    }
   printf(" Subgraph 2 :\n");          /* print subgraph 2 as an adjacency
                                          matrix showing the edges among
                                          nodes */
   for(i=0;i<=nodenbr;i++)
    {
     for(j=0;j<=nodenbr;j++)
      {
      if(Sub2[S2[i]][j] == 1)
        printf("%d -> %d\n",S2[i],j);
      }
    }
}

/**************************************************************************

This procedure's job is to do the relabeling, i.e., to flip the
vertices' labels among each other without affecting the consistency of
the input graph.

**************************************************************************/
 void relabel()
 {
   int Temp[Maxnd],                    /* a temporary array used to hold
                                          the flipped node */
      i,j;                             /* used as looping variables */

                                       /* saving the graph's first node
                                          connections, to other graph
                                          nodes, into the array Temp */
      for(i=0;i<nodenbr;i++)
            Temp[i] = graph[0][i];

                                       /* moving up all the input graph's
                                          nodes but the first one */
      for(i=0;i<(nodenbr-1);i++)
       {
         for(j=0;j<nodenbr;j++)
            graph[i][j] = graph[i+1][j];
       }
                                       /* moving the first input graph
                                          node, which is saved in Temp, to
                                          be the last node in the graph */
            for(i=0;i<nodenbr;i++)
             graph[nodenbr][i] = Temp[i];
```

```
                                        /* call the procedure, algorithm,
                                           trying to split the graph
                                           vertices */
            algorithm();

 }

/*************************************************************************

This procedure checks whether the predicate P(x, y, p, q) is true.  We
can say that the predicate is true if one of these two conditions is
true:
-(p, q) is an edge in the graph G, whereas either (p, y) or (x, q) is
not an edge in the graph G.
-(p, q) is not an edge in the graph G, whereas (p, y) and (x, q) are
edges in the graph G.

*************************************************************************/

 int checkP(x,y,p,q)
  int x,y,p,q;
  {
    if (((graph[p][q])&&((!graph[p][y])||(!graph[x][q]))) ||
         ((!graph[p][q])&&(graph[p][y] && graph[x][q])))
      return 1;                         /* if P(x, y, p, q) is true */
    else
      return 0;                         /* if P(x, y, p, q) is not true */
  }
```

The following is the forman.c file.

```
/*************************************************************************

This program implements Forman's algorithm for decomposing an m-graph
into prime subgraphs.  It takes as input m-graph in the form of
adjacency matrix. Then it tries to build the m-graph spanning chart, by
that the problem of finding the prime program decomposition of an m-
graph is transformed into a problem of decomposing spanning charts.
After that, the tier-i paths will be developed and finally, it tests the
tier-i paths for subprogram cutsets.  At the end, the result is printed
in the form of decomposed spanning trees.

*************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
                                    /* these are the node types used in
                                       m-graph and its decomposed parts
                                       */
#define entry           1
#define predicate       2
#define join            3
#define assignment      4
#define goto            5
#define call            6

struct nn                           /* this structure represent the data
                                       structure of nodes in the m-graph
                                       and its decomposed parts */
      {
            int id;                 /* id number of node */
            int label;              /* label associated with node */
            struct nn *succ[2];     /* arcs to successor nodes */
            int ntype;              /* node type */
            struct nn *pred,        /* previous node in spanning tree */
                      *corr;        /* if a join node in the spanning
                                       tree this field points to the
                                       corresponding goto node */
            int tier;               /* tier number */
            int tier_count;         /* if more than one tier-i path
                                       meets at this node the value of
                                       tier_count is 2 */
            int gotol[10];          /* if a goto node in the spanning
                                       tree this field contains set of
                                       all nodes that this node goes to
                                       */
            int joinl[10];          /* if a join node in the spanning
                                       tree this field contains set of
                                       all nodes that joins this node
                                       */

            int nb_goto;            /* number of nodes in gotol */
            int nb_join;            /* number of nodes in joinl */
      };

struct mgraph                       /* data structure of the m-graph */
      {
            struct nn  entryl,      /* entry node */
                       vertices[100], /* nodes in the m-graph */
                       exit;        /* exit node */
```

```
};

struct qq                               /* queue containing pointers to the
                                           goto nodes in the m-graph */
{
        struct nn *first;
        struct qq *next;
};

struct s                                /* stack containing pointers to the
                                           goto nodes in the m-graph */
{
        struct nn *first;
        struct s *next;
};

typedef struct mgraph graph;
typedef struct qq goto_queue;
typedef struct nn node;
typedef struct s goto_st;

void initialize();
void build_graph();
void step1();
void step2();
void process_tier();
void copynode();
void Remove();
void Print_result();


graph *M;                               /* represents the m-graph to be
                                           decomposed */
goto_queue  *fr_qu,                     /* front pointer of the goto queue
                                           */
            *rear_qu;                   /* rear pointer of the goto queue */

goto_st     *fr_st;                     /* front pointer of the goto stack
                                           */

int   last_label,                       /* holds the last label number used
                                           */
      nb_goto_st = 0;                   /* number of elements in the goto
                                           stack */

FILE  *in;                              /* pointer to the input file */


main()
{
  if((in = fopen("/v/refae/thesis/forman/f_inp","r")) == NULL)
   {
   printf("We can't open the input file\n");
   exit(1);
   }
  initialize();                         /* call initialize() to initialize
                                           the variables of the program */
  build_graph();                        /* call build_graph() to read the
                                           input file and start building the
                                           m-graph */

  step1();
  step2();
```

```
  Print_result(&M->entryl,0);          /* print the result of the
                                          decomposition */
}
/*************************************************************************

 The following procedure initializes all the global variables used in
this program.

*************************************************************************/

 void initialize()
{
int i,j;

  M = malloc(sizeof(graph));           /* create space for the graph in the
                                          memory and initialize it */
  M->entryl.label = 0;                 /* initialize entry and exit nodes
                                          */

  M->entryl.pred = NULL;
  M->entryl.tier = -1;
  M->entryl.tier_count = 0;
  M->exit.label = 0;
  M->exit.pred = NULL;
  M->exit.tier = -1;
  M->exit.tier_count = 0;
 for(i=0;i<10;i++)
 {
  M->entryl.gotol[i] = 0;
  M->entryl.joinl[i] = 0;
  M->exit.gotol[i] = 0;
  M->exit.joinl[i] = 0;
 }
  M->entryl.nb_goto = 0;
  M->entryl.nb_join = 0;
  M->exit.nb_goto = 0;
  M->exit.nb_join = 0;

    for(i=0;i<100;i++)                 /* initialize all graph nodes but
                                          entry and exit ones */
     {
      M->vertices[i].label = 0;
      M->vertices[i].pred = NULL;
      M->vertices[i].tier = -1;
      M->vertices[i].tier_count = 0;
      for(j=0;j<10;j++)
       {
        M->vertices[i].gotol[j] = 0;
      M->vertices[i].joinl[j] = 0;
       }
      M->vertices[i].nb_goto = 0;
      M->vertices[i].nb_join = 0;
     }

}


/*************************************************************************

This procedure builds a graph as an adjacency matrix.  It reads the
input from a file and constructs the graph.

*************************************************************************/
```

```
 void build_graph()
{
int    nb_nodes,                        /* number of nodes */
       label,                           /* label number of the node */
       type,                            /* type of the node */
       succ1,                           /* successor node */
       succ2,                           /* if a predicate node in the graph
                                           this field points to its second
                                           successor node */
       id,                              /* id unmber of the node */
       i;                               /* used as looping variable */

  fscanf(in,"%d\n",&nb_nodes);          /* reads number of nodes in the
                                           graph */
  fscanf(in,"%d %d %d\n",&id,&type,&succ1);

  M->entry1.id = id;                    /* fill the entry node with the data
                                           read */
  M->entry1.succ[0] = &M->vertices[succ1 -2];
  M->entry1.ntype = type;

   for(i=0;i<(nb_nodes - 1);i++)
    {
     fscanf(in,"%d %d",&id,&type);      /* reads id number and node type of
                                           the graph nodes */
      if(type != 2)                     /* if the node is not a predicate
                                           i.e. it has only one successor */
        {                               /* reads the node's successor number
                                           */
       fscanf(in," %d\n",&succ1);
                                        /* make an arc from the node to its
                                           successor node */
       M->vertices[i].succ[0] = &M->vertices[succ1-2];
      }
      else
      {                                 /* if the node is a predicate
                                           i.e. it has two successors */
                                        /* reads the node's successors
                                           numbers */
       fscanf(in," %d %d\n",&succ1,&succ2);
                                        /* make an arc from the node to its
                                           successors nodes */
       M->vertices[i].succ[0] = &M->vertices[succ1-2];
       M->vertices[i].succ[1] = &M->vertices[succ2-2];
      }
                                        /* fill the node with its type and
                                           id number */
       M->vertices[i].id   = id;
       M->vertices[i].ntype = type;
    }
}


/*************************************************************************

This procedure transforms M into a spanning chart by performing a depth
first search for the join nodes which are then split.  In addition,
step1 places pointers in nodes to their predecessors in the spanning
chart.  This enables the spanning chart to be searched backwards.

*************************************************************************/
```

```
 void step1()
{
struct ss                              /* stack used in the creation of the
                                          graph spanning tree */
      {                                /* it contains the node being
                                          processed with its predecessor,
                                          repectively */
            struct nn *first,
                      *second;
            int j;                     /* if the predecessor node of the
                                          node being processed is a
                                          predicate, this field specifies
                                          which successor of that predicate
                                          is the node being procesed */

            struct ss *next;
      };
typedef struct ss stack;
stack       *top,                      /* pointer to the node in the top of
                                          the stack */
            *tmp;
int         nb_stack = 0,              /* number of elements in the stack
                                          */
            i,j;                       /* used as looping variables */
node        *v,                        /* pointer to the node being
                                          processed */
            *u,                        /* the prdecessor node of the one
                                          being processed */
            *q;                        /* pointer to the goto node being
                                          created in the process of
                                          creating the input graph spanning
                                          tree */


i=0;
top = malloc(sizeof(stack));
top->first = &M->entryl;               /* start the process by initiating
                                          the entry node to be at the top
                                          of the stack */


top->second = NULL;
top->j = -1;
top->next = NULL;
nb_stack = 1;

   while(nb_stack >0)                  /* while there are still nodes in
                                          the stack, continue the process
                                          of placing pointers in nodes to
                                          their predecessors in the
                                          spanning trees */


    {
      v = top->first;                  /* the node being processed */
      u = top->second;                 /* the predecessor of the node being
                                          processed */

      j = top->j;
      tmp = top;
      top = top->next;
      free(tmp);                       /* delete this entry, containing the
                                          node being processed and its
                                          predecessor from the stack */


      nb_stack--;                      /* decrement the number of the stack
                                          elements by one */
        if(v->label == 0)              /* if it hasn't been processed
                                          before */

         {
           i++;
           v->label = i;
           v->pred  = u;
```

```
                                /* add the successor node of the
                                   node being processed to the stack
                                   as the main node to be processed
                                   later */
     tmp = malloc(sizeof(stack));
   tmp->first = v->succ[0];
   tmp->second = v;
   tmp->j = 1;
    if(nb_stack == 0)
     {
    tmp->next = NULL;
    top = tmp;
     }
    else
     {
    tmp->next = top;
    top = tmp;
     }
    nb_stack++;                 /* increments the stack by one */
                                /* if the node being processed is a
                                   predicate then two entries of it
                                   should be entered to the stack.
                                   the first will be combining the
                                   node with its first successor and
                                   the second will be combining the
                                   node with its second successor */
    if(v->ntype == predicate)
    {
    tmp = malloc(sizeof(stack));
    tmp->first = v->succ[1];
    tmp->second = v;
    tmp->j = 2;
    if(nb_stack == 0)
     {
      tmp->next = NULL;
      top = tmp;
     }
    else
     {
      tmp->next = top;
      top = tmp;
     }
    nb_stack++;
     }
 }
else
 {                              /* if the node on the top of the
                                   stack was processed before.  Then
                                   a goto node will be created in
                                   the spanning tree to be placed
                                   between the node being processed
                                   and its predecessor node */
   q = malloc(sizeof(node));
   memset(q,'\0',sizeof(node));
   q->ntype = goto;
   i++;
   q->label = I;
                                /* add the label of the node being
                                   processed to the goto array of
                                   the nodes that this goto node
                                   goes to */
   q->gotol[q->nb_goto++] = v->label;
   q->pred = u;
   q->succ[0] = NULL;
```

```
                                        /* add the label of the node being
                                           processed to the join array of
                                           the nodes that joins this node
                                           */

            v->joinl[v->nb_join++] = v->label;
            v->corr = q;
              if (v->label == 1)        /* if the node being processed is
                                           the entry node, then the goto
                                           node will be the same as the exit
                                           node */

             {
               copynode(q,&M->exit);
               free(q);                 /* because we have substitute it
                                           with M->exit so no need for
                                           it, so its pred node will be
                                           pointing to M->exit node */

               u->succ[j-1] = &M->exit;
             }
              else

               u->succ[j-1] = q;
             }
    }
last_label = i;                         /* holds the last label number used
                                           */

}

/*************************************************************************

Step2 marks each node with the number of the tier-i path to which it
belongs.  This is done by starting with the tier-0 path and following it
backwards.  By the definition of a tier-0 path, the goto nodes of the
tier-1 paths correspond to the join nodes on the tier-0 path.  Pointers
to these goto nodes are placed both in a queue (for further processing
in step2) and in a stack (for processing in step 3).  Processing
continues in a similar manner for higher numbered tier-i paths.  In
addition, predicate nodes where two tier-i paths with the same number
come together are marked in order to enable step3 to process only those
paths that belong to the same set tier-i paths.

*************************************************************************/

 void step2()
{
 int  i,                                /* used in the keeping track of what
                                           tier number the process is in */
      nb_goto_qu,                       /* number of nodes in the goto queue
                                           */
 node *v,                               /* used to hold the elements at the
                                           top of the goto queue */
      *w,                               /* used to hold v->corr if v is join
                                           node */
      *n_tmp,                           /* temporary variable */
      *end_of_tier;                     /* used to hold the node at the top
                                           of the goto stack before it is
                                           sent as a parameter to the
                                           procedure process_tier */
 goto_queue       *q_tmp;               /* temporary pointer for a goto
                                           queue node */
 goto_st    *st_tmp;                    /* temporary pointer for a goto
                                           stack node */
```

```
v = &M->exit;                          /* start the process from the graph
                                          exit node */
v->tier = 0;                           /* the first tier is tier-0 which is
                                          the unique pathfrom the entry
                                          node of the graph to the exit
                                          node of the graph */
                                       /* Put v in the goto queue */
fr_qu = malloc(sizeof(goto_queue));
memset(fr_qu,'\0',sizeof(goto_queue));
fr_qu->first = v;
fr_qu->next = NULL;
rear_qu = fr_qu;
nb_goto_qu++;
                                       /* Put v in the goto stack */
fr_st = malloc(sizeof(goto_st));
memset(fr_st,'\0',sizeof(goto_st));
fr_st->first = v;
fr_qu->next  = NULL;
nb_goto_st++;
 while(nb_goto_qu > 0)                 /* while there is still some nodes
                                          in the goto queue, the process of
                                          marking each node with the number
                                          of the tier-i path to which it
                                          belongs continue */
  {
   v = fr_qu->first;
   q_tmp = fr_qu;
   fr_qu = fr_qu->next;
   free(q_tmp);                        /* remove v from the goto queue */
   nb_goto_qu--;
   i = v->tier;

    while((v->pred != NULL)&&(v->pred->tier == -1))
     {                                 /* while the node is not the entry
                                          node and its predecessor node
                                          hasn't been marked with a tier-i
                                          path */
    v = v->pred;
    v->tier = i;
    if(v->ntype == join)               /* if the node is a join node mark
                                          its corresponding goto node with
                                          a tier-(i+1) */
     {
      w = v->corr;
      w->tier = i +1;
                                       /* store the corresponding goto node
                                          in the goto queue and the goto
                                          stack */
      q_tmp = malloc(sizeof(goto_queue));
        memset(q_tmp,'\0',sizeof(goto_queue));
      q_tmp->first = w;
      q_tmp->next = NULL;
       if(nb_goto_qu > 0)
        {
         rear_qu->next = q_tmp;
         rear_qu = q_tmp;
        }
       else
        {
         fr_qu = rear_qu = q_tmp;
        }
       nb_goto_qu++;
       st_tmp = malloc(sizeof(goto_st));
       memset(st_tmp,'\0',sizeof(goto_st));
```

```
            st_tmp->first = v->corr;
             if(nb_goto_st > 0)
              {
               st_tmp->next = fr_st;
               fr_st = st_tmp;
                }
              else
               {
               fr_st = st_tmp;
               fr_st->next = NULL;
               }
              nb_goto_st++;
             }
    }                                  /* if more than one tier-i path
                                          meets at this node the value path
                                          of tier_count is 2 */
      if((v->pred != NULL)&&(v->tier == v->pred->tier))
         v->pred->tier_count = 2;
       }
      M->entryl.tier_count = 2;
                                       /* while there are still some nodes
                                          in the goto stack, call the
                                          process_tier procedure to process
                                          all the tier-i paths from high
                                          tier number to low */
      while(nb_goto_st > 0)
       {
        end_of_tier = fr_st->first;
        st_tmp = fr_st;
        fr_st = fr_st->next;
        free(st_tmp);
        nb_goto_st--;
        process_tier(end_of_tier);
       }
 }
/*************************************************************************

This step processes all the tier-i paths from high tier number to low by
using the stack of goto node pointers produced in step2.  In the case of
a predicate node where two tier-i paths of the same number come
together, when the node encountered for the second time, a pointer to
the predicate node is placed on the stack so that it is processed like a
goto node.   This step maintains two pointers front and rear; the
candidate pair of arcs is the arc entering the node to which front
points and the arc along the tier-i path exiting the node to which rear
points.

*************************************************************************/

 void process_tier(node *end)
 {
 node *rear,                           /* the candidate pair of arcs is the
                                          arc entering the node to which
                                          front points and the arc along
                                          the tier-i path exiting the node
                                          to which rear points */
      *front,
      *subprog_exit;                   /* points to the cutset being
                                          removed */

goto_st *st_tmp;                       /* temporary variable for goto stack
                                          */
```

```
int  Rgoto[10],                    /* temporary variables used to
                                      update the rear node's goto and
                                      join sets, when rear node type is
                                      a predicate */
     Rjoin[10],
     Goto[10],                     /* temporary variables used to find
                                      the candidate pair of arcs that
                                      could be the cutset.  The two
                                      arcs has to have same Goto and
                                      Join nodes */
     Join[10],
     l_nb_goto,                    /* variable used to hold number of
                                      goto nodes in the goto array of
                                      the node rear */
     l_nb_join,                    /* variable used to hold number of
                                      join nodes in the join array of
                                      the node rear */
     g_nb,                         /* variable used to hold number of
                                      goto nodes in the goto array of
                                      the node end */
     j_nb,                         /* variable used to hold number of
                                      join nodes in the join array of
                                      the node end */
     g_j_flag,                     /* flag to check whether the Goto
                                      set is the same as Join's set */
     front_pred,                   /* variables used to ensure that
                                      the algorithm doesn't go behind
                                      entry or after exit node,
                                      otherwise it would crash */
     rear_pred,
     i;                            /* used as looping variable */

 rear = end->pred;
 for(i=0;i<10;i++)
  {
   Rgoto[i] = end->gotol[i];       /* Rgoto is initialized to the goto
                                      set of the node end */
   Rjoin[i] = end->joinl[i];       /* Rjoin is initialized to the join
                                      set of the node end */

   Goto[i] = rear->gotol[i];       /* Goto is initialized to the goto
                                      set of the node rear */

   Join[i] = rear->joinl[i];       /* Join is initialized to the join
                                      set of the node rear */

  }
 g_nb = end->nb_goto;              /* g_nb is initialized to the number
                                      of goto nodes in the goto set of
                                      node end */
 j_nb = end->nb_join;              /* j_nb is initialized to the number
                                      of join nodes in the join set of
                                      node end */

 l_nb_goto = rear->nb_goto;        /* l_nb_goto is initialized to the
                                      number of goto nodes in the goto
                                      set of node rear */

 l_nb_join = rear->nb_join;        /* l_nb_join is initialized to the
                                      number of join nodes in the join
                                      set of node rear */

  rear_pred = 1;
```

```
                                    /* while the pair of arcs, end and
                                       rear, have the same tier path,
                                       and the rear node doesn't have
                                       more than one tier meeting at it;
                                       try to find a cutset between
                                       these two arcs */
while((rear_pred == 1)&&(rear->tier == end->tier)
        &&(rear->tier_count == 0))
 {
  if((rear->ntype != assignment)&&(rear->pred != NULL))
   {                                /* if rear is not an assignment node
                                       and it is not the entry node */
    front = rear->pred;             /* assign front to be rear's
                                       predecessor */
     for(i=0;i<front->nb_goto;i++)
      {                             /* Goto = Goto U Front's goto set */
     Goto[l_nb_goto++] = front->gotol[i];
   }
     for(i=0;i<front->nb_join;i++)
    {                               /* Join = Join U Front's join set */
      Join[l_nb_join++] = front->joinl[i];
    }
     front_pred = 1;

     while((front_pred == 1)
         &&(front->tier == rear->tier)
               &&(front->tier_count == 0))
       {                            /* while Front and rear are on the
                                       same tier path */
        g_j_flag = 0;
         for(i=0;i<10;i++)          /* check whether Goto = Join */
          {
         if(Goto[i] != Join[i])
          g_j_flag = 1;
          }
        if((g_j_flag != 1)&&
         (!((rear == end->pred)&&
               (front->pred->pred == NULL))))
         {                          /* if Goto = Join, and the arcs are
                                       not the entry and exit arcs of
                                       the graph */
                                    /* when the a subprogram cutset is
                                       found, Remove is called to create
                                       the spanning tree hierarchy */
          Remove(&front,&rear,&subprog_exit);

         for(i=0;i<10;i++)
          {                         /* Goto and Join are set to empty
                                       because the goto and join nodes
                                       have been removed */
           Goto[i] = Join[i] = 0;
           l_nb_goto = l_nb_join = 0;
          }

                                    /* put subprog_exit in the stack in
                                       order to be tested later on for
                                       subprogram cutsets in the
                                       spanning tree that was removed */
       st_tmp = malloc(sizeof(goto_st));
       memset(st_tmp,'\0',sizeof(goto_st));
       st_tmp->first = subprog_exit;
        if(nb_goto_st > 0)
         {
          st_tmp->next = fr_st;
```

```
          fr_st = st_tmp;
           }
         else
          {
           fr_st = st_tmp;
           fr_st->next = NULL;
          }
         nb_goto_st++;

           }
          if(front->pred != NULL)
           {
           front = front->pred;     /* move front back to its
                                       predecessor */
                                    /* Goto = Goto U the goto set of
                                       Front node */
          for(i=0;i<front->nb_goto;i++)
           Goto[l_nb_goto++] = front->gotol[i];
                                    /* Join = Join U the join set of
                                       Front node */
          for(i=0;i<front->nb_join;i++)
           Join[l_nb_join++] = front->joinl[i];
          }
         else
           front_pred = 0;
         }
                                    /* add the set of goto nodes from
                                        the rear node to Rgoto */
         for(i=0;i<rear->nb_goto;i++)
           Rgoto[g_nb++] = rear->gotol[i];

                                    /* add the set of join nodes from
                                        the rear node to Rjoin */
         for(i=0;i<rear->nb_join;i++)
           Rjoin[j_nb++] = rear->joinl[i];

    }
  if(rear->pred != NULL)
  {
   rear = rear->pred;
   for(i=0;i<10;i++)
    {
    Goto[i] = rear->gotol[i];      /* add the set of goto nodes from
                                       the rear node to Goto */
    Join[i] = rear->joinl[i];      /* add the set of join nodes from
                                       the rear node to Join */
    }
  }
  else rear_pred = 0;
}
if(rear->tier_count != 0)
 {
  if(rear->tier_count == 1)
    {
                                    /* if the node tier is not 2 add it
                                       to the top of the stack */
       st_tmp = malloc(sizeof(goto_st));
       memset(st_tmp,'\0',sizeof(goto_st));
       st_tmp->first = rear;
        if(nb_goto_st > 0)
         {
          st_tmp->next = fr_st;
          fr_st = st_tmp;
           }
```

```
          else
           {
            fr_st = st_tmp;
            fr_st->next = NULL;
           }
         nb_goto_st++;
        }
        rear->tier_count = rear->tier_count -1;
     }
      if(rear->ntype == predicate)    /* if the rear node is a predicate
                                         */
       {
        for(i=0;i<g_nb;i++)
        {                             /* add to the nodes in the Rgoto to
                                         the rear's goto set */
        rear->gotol[rear->nb_goto++] = Rgoto[i];
         }
        for(i=0;i<j_nb;i++)
        {                             /* add to the nodes in the Rjoin to
                                         the rear's join set */
        rear->joinl[rear->nb_join++] = Rjoin[i];
        }
        }
}
/*************************************************************************

This procedure removes the prime subgraph from the original graph.
Also, it replaces the decomposed graph with a call node in the original
graph.
*************************************************************************/

void Remove(node **fr,node **re,node **exit)
{
 node       *tmp,                     /* variable used to create the call
                                         node to the subprogram cutset */
       *temp,                         /* variable used to create the entry
                                         node of the subprogram cutset */
       *temp1,                        /* variable used to create the goto
                                         node of the subprogram cutset */
       *rear,
       *front;

 front = *fr;
 rear = *re;

  tmp = malloc(sizeof(node));
  memset(tmp,'\0',sizeof(node));
  tmp->ntype = call;                 /* the call node that will call the
                                         subprogram cutset from the
                                         original graph */

  tmp->pred = front->pred;
  if(rear->succ[0]->tier < rear->succ[1]->tier)
                                     /* all the nodes outside the
                                         subprogram have there tier
                                         number less than the ones
                                         inside the subprogram */
   {                                 /* connecting tmp with its successor
                                         */
    tmp->succ[1] = rear->succ[0];
    rear->succ[0]->pred = tmp;
   }
  else
   {
   tmp->succ[1] = rear->succ[1];
```

```
  rear->succ[1]->pred = tmp;
 }
 tmp->label = ++last_label;
                                    /* check whether the prdecessor node
                                       of the call node is a predicate
                                       */
 if(front->pred->ntype == predicate)
  { if(front->pred->succ[1] == front)
     front->pred->succ[1] = tmp;
    else
     front->pred->succ[0] = tmp;
  }
 else
  front->pred->succ[0] = tmp;
temp = malloc(sizeof(node));
memset(temp,'\0',sizeof(node));
temp->ntype = entry;                /* create an entry node for the
                                       subrpogram cutset and fill its
                                       entries */

temp->tier_count = 2;
temp->succ[0] = front;
front->pred = temp;
tmp->succ[0] = temp;
temp->label = ++last_label;
temp->tier = front->tier;
tmp->tier = temp->tier;

temp1 = malloc(sizeof(node));
memset(temp1,'\0',sizeof(node));
temp1->ntype = goto;                /* create a goto node for the
                                       subrpogram cutset and fill its
                                       entries */

*exit = temp1;
temp1->pred = rear;
temp1->tier = rear->tier;
temp1->label = ++last_label;
temp1->succ[0] = tmp->succ[1];

 if(rear->succ[0]->pred == tmp)     /* link the goto node to its
                                       predecessor */
  rear->succ[0] = temp1;
 else
  rear->succ[1] = temp1;

front = tmp;
rear = tmp;
*fr = front;
*re = rear;
}

/*************************************************************************

This procedure's job is to copy all the data from one node to another.

*************************************************************************/

 void copynode(node *x,node *y)
{
 int i;
                                    /* copy all the information from the
                                       node x to the node y, and at the
                                       end return y */

 y->id = x->id;
 y->label = x->label;
```

```
 y->succ[0] = x->succ[0];
 y->succ[1] = x->succ[1];
 y->ntype = x->ntype;
 y->pred = x->pred;
 y->corr = x->corr;
 y->tier = x->tier;
 y->tier_count = x->tier_count;
  for(i=0;i<10;i++)
    {
     y->gotol[i] = x->gotol[i];
     y->joinl[i] = x->joinl[i];
    }
 y->nb_goto = x->nb_goto;
 y->nb_join = x->nb_join;
}

/*************************************************************************

This procedure prints the prime graphs resulting from the decomposition
of the original graph.  The result is printed in the form of a spanning
tree.
*************************************************************************/

void Print_result(node *t,int l)
{
 int i,j;

if(!t) return;                          /* this procedure is recursive, so
                                           when it reaches Null, the
                                           procedure starts exiting its
                                           processes */
for(i=0;i<l;++i) printf("  ");

if(t->ntype == goto)
{
printf(" (%d,%d,%d,",t->label,t->ntype,t->tier);
for(j=0;j<t->nb_goto;j++)
printf("%d,",t->gotol[j]);
printf(")\n");
}
else
printf(" (%d,%d,%d)\n",t->label,t->ntype,t->tier);
if(t->ntype == predicate)
 {
  Print_result(t->succ[0],l+1);
  Print_result(t->succ[1],l+1);
 }
else
 {
  Print_result(t->succ[0],l+1);
 }
}
```

APPENDIX D

INPUTS/OUTPUTS LISTING

This appendix contains the input test graphs along with their corresponding outputs used to test the two algorithms. Forman's algorithm was tested using six different inputs. In what follows the test programs are presented followed by their source and a brief discussion.

The first test input is taken from Sedgewick's text [Sedgewick 88]. It is a procedure that deals with sorting an array using the insertion technique, the array to be sorted is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This procedure's job is to sort an array, a, using insertion techniques.
The major variables used in the procedure, which are global, are:

a: This is the array that contains the data to be sorted.
p: It is a pointer array that is manipulated, to restrict accessing the
   original array only for comparisons.
N: This is the number of elements in a.

*************************************************************************/

Procedure insertion
var    i,                  /* used as loop variable */
       j, v: integer;      /* variables used by the arrays a and p for the
                              comparison */
begin
  i := 1;
  while(i<N)               /* the loop initializes the P array in order to
                              produce an algorithm that will sort the index
                              array */
   begin
     P[i] := i;
     i := i + 1;
   end;
  i : = 2;
  while(i<N)               /* this loop and compare the elements in a, that
                              are indexed by the array p.  Process and
                              adjustments will happen in the array p.  At
```

```
                                the end of the algorithm, the index array
                                will be sorted so that p[1] is the smallest
                                element in the array a */
   begin
     v := p[i];
     j := i;
     while(a[p[j-1]] > a[v]) do
      begin
        p[j] := p[j-1];
        j := j - 1;
      end;
     p[j] := v;
     i := i + 1;
   end;
end;
```

The m-graph presentation of test input 1 program is to be found in Figure 15.

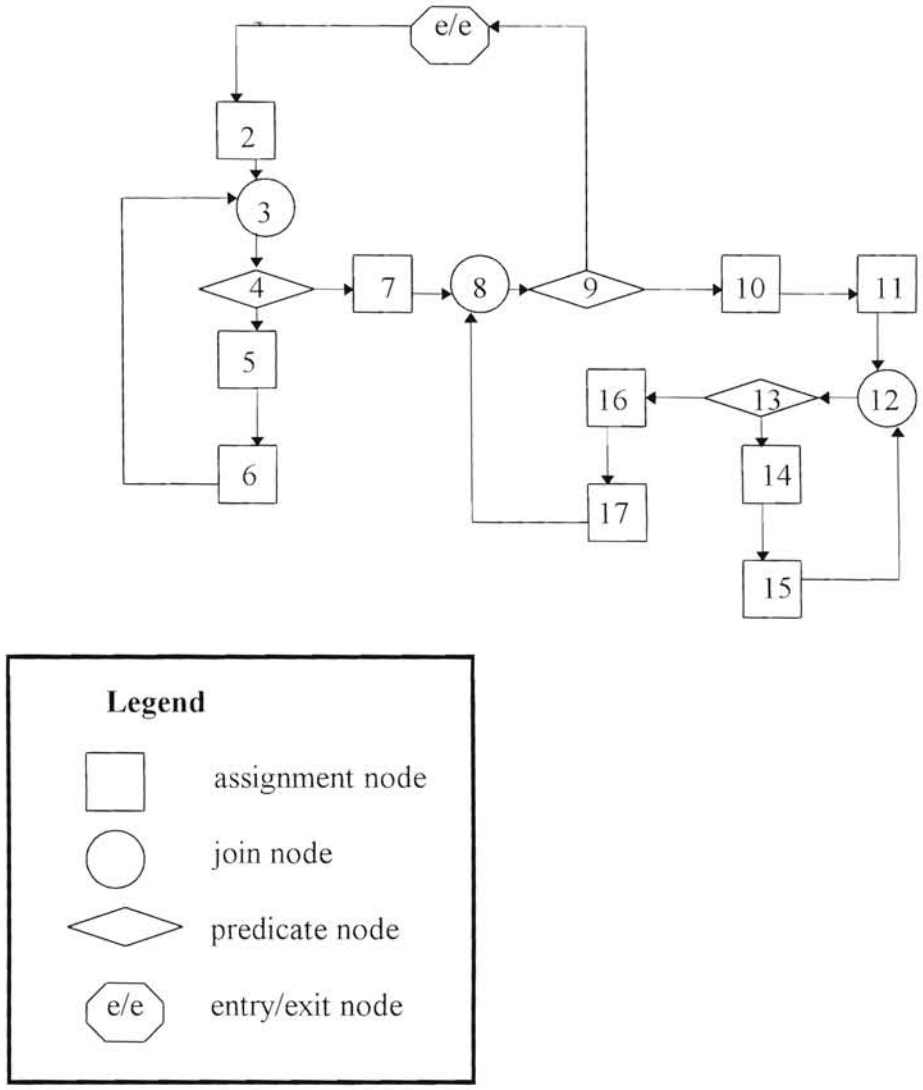


Figure 15. The m-graph of test input 1 program

The decomposed spanning tree of test input 1 m-graph is presented in Figure 16.

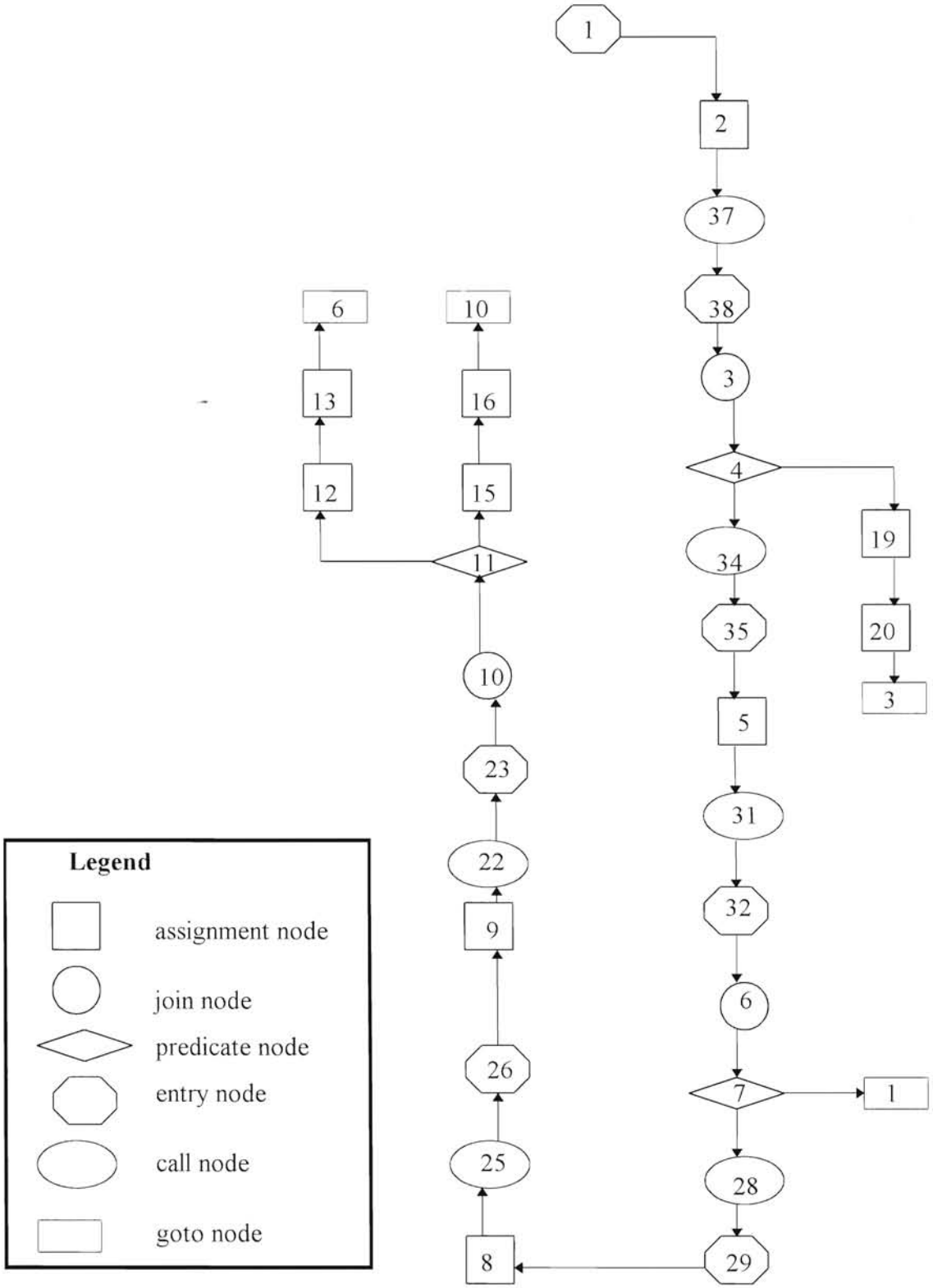


Figure 16. This is the decomposed spanning tree of the m-graph of Figure 15

The original program of test input 1 after being decomposed is represented here.

```
/*********************************************************************

This procedure's job is to sort an array, a, using insertion techniques.
The major variables used in the procedure, which are global, are:

a: This is the array that contains the data to be sorted.
p: It is a pointer array that is manipulated, to restrict accessing the
   original array only for comparisons.
N: This is the number of elements in a.

*********************************************************************/

Procedure insertion
var    i,                        /* used as loop variable */
       j, v: integer;            /* variables used by the arrays a and p
                                    for the comparison */
begin
  i := 1;
 Subprogram 1                    /* the first prime program starts here */
  while(i<N)                     /* the loop initializes the P array in
                                    order to produce an algorithm that will
                                    sort the index array */
begin
     P[i] := i;
     i := i + 1;
   end;
  Subprogram 2                   /* the second prime program starts here */
   i : = 2;
    Subprogram 3                 /* the third prime program starts here */
     while(i<N)                  /* this loop and compare the elements in
                                    a, that are indexed by the array p.
                                    Process and adjustments will happen in
                                    the array p.  At the end of the
                                    algorithm, the index array will be
                                    sorted so that p[1] is the smallest
                                    element in the array a */
      begin
       Subprogram 4              /* the fourth prime program starts here */
        v := p[i];
          Subprogram 5           /* the fifth prime program starts here */
           j := i;
            Subprogram 6         /* the sixth prime program starts here */
             while(a[p[j-1]] > a[v]) do
               begin
                p[j] := p[j-1];
                 j := j - 1;
               end;
             end Subprogram 6 /* the sixth prime program ends here */
           p[j] := v;
           i := i + 1;
          end Subprogram 5       /* the fifth prime program ends here */
       end Subprogram 4          /* the fourth prime program ends here */
    end Subprogram 3             /* the third prime program ends here */
  end Subprogram 2               /* the second prime program ends here */
end Subprogram 1                 /* the first prime program ends here */
end;
end;
```

The second test input is taken from Sedgewick's text [Sedgewick 88]. It is a function that deals with searching an array using the binary search technique, the array to be searched, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This function is to search for an element in an array a.  It uses the
binary search technique, which divide the set of records into two parts,
determines which of the two parts the key sought belongs to, then
concentrates on that part.  It keeps the set records sorted.  The major
global variables used in this function are:

a: It is an array of records, where the key is the variable in the
   record that contains the numbers to be searched.
N: It represents the number of elements to be searched.
v: It is the key to be searched for.

*************************************************************************/

Function binarysearch(v: integer): integer
Var x, l, r: integer;
begin
 l := 1;
 r := N;
 while(v <> a[x].key or l <= r)/* it compares v with the element at the
                                  middle position  of the table.  If v
                                  is smaller, then it must be in the
                                  first half of the table; if v is
                                  greater, then it must be in the second
                                  half of the table */
   begin
    x := (l + r) div 2;
    if(v < a[x].key) then
     r := x - 1;
    else
     l := x + 1;
   end;
 if(v = a[x].key) then
  binarysearch := x;
 else
  binarysearch := N + 1;
end;
```

The m-graph presentation of test input 2 program is to be found in Figure 17.

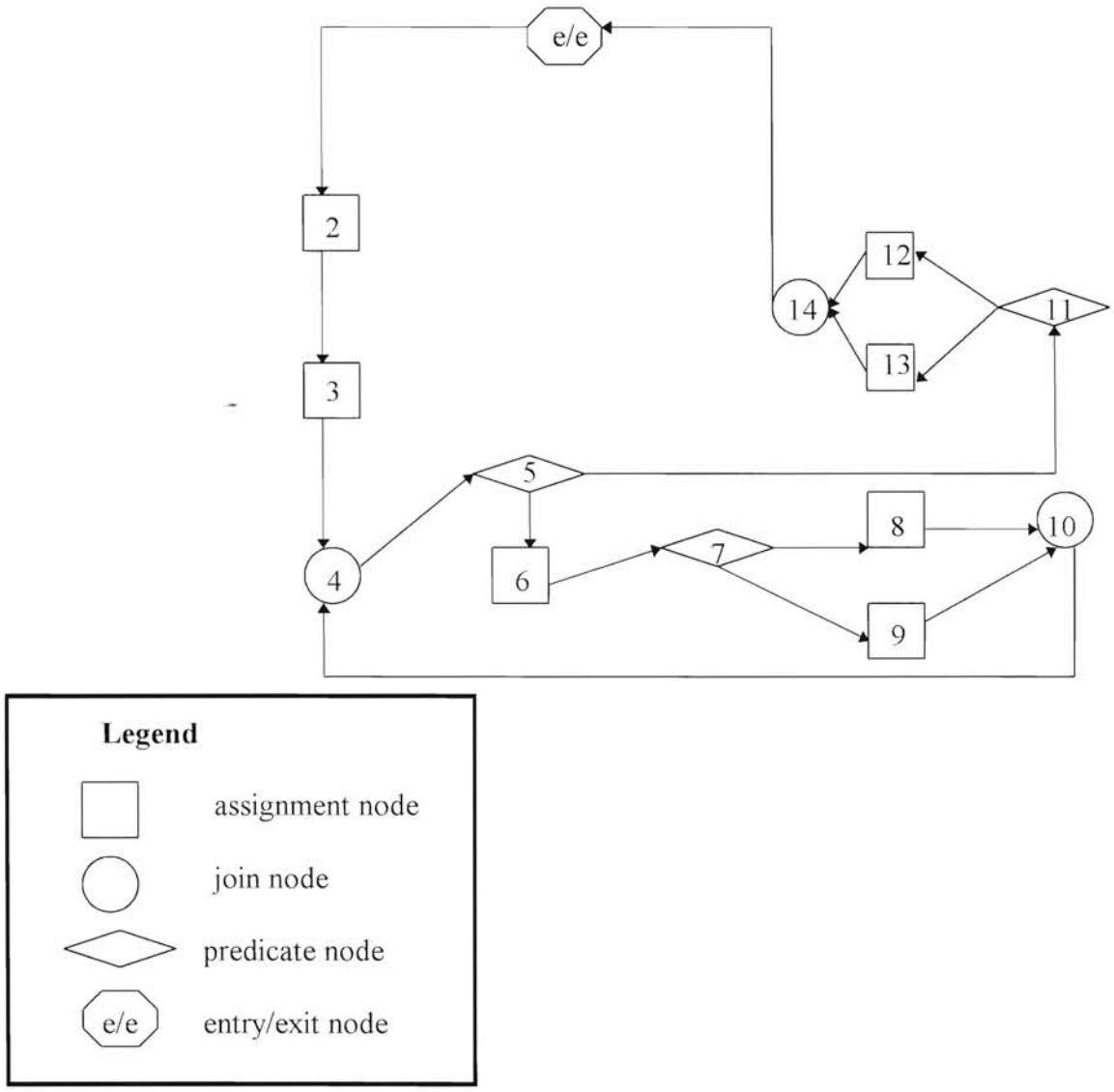


Figure 17. The m-graph of test input 2 program

The decomposed spanning tree of test input 2 m-graph is presented in Figure 18.

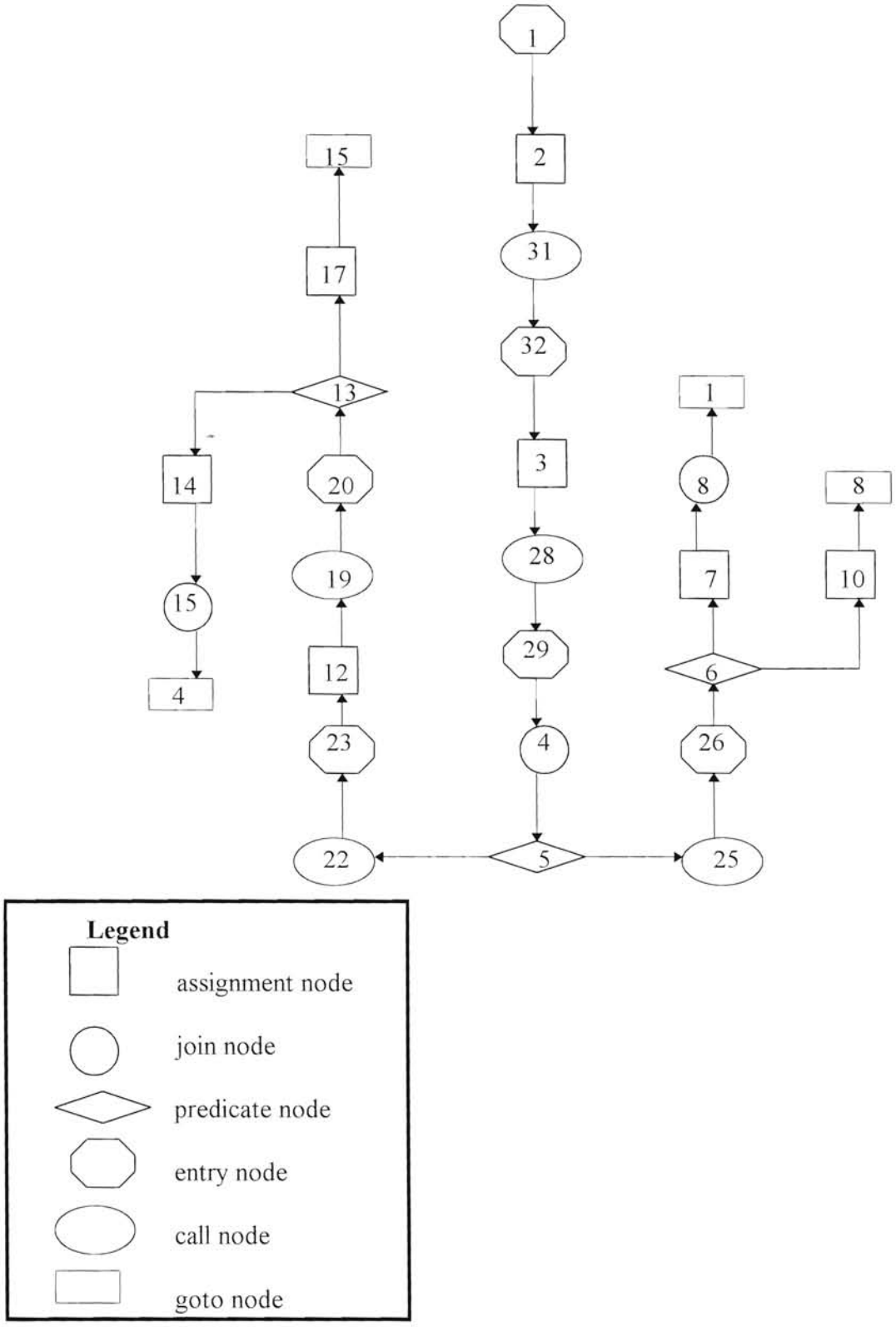


Figure 18. This is the decomposed spanning tree of the m-graph of Figure 17

The original program test input 2 after being decomposed is represented here.

```
/*************************************************************************

This function is to search for an element in an array a.  It uses the
binary search technique, which divide the set of records into two parts,
determines which of the two parts the key sought belongs to, then
concentrates on that part.  It keeps the set records sorted.  The major
global variables used in this function are:

a: It is an array of records, where the key is the variable in the
   record that contains the numbers to be searched.
N: It represents the number of elements to be searched.
v: It is the key to be searched for.

*************************************************************************/

Function binarysearch(v: integer): integer
var x, l, r: integer;
begin
 l := 1;
 Subprogram 1                  /* the first prime program starts here */
  r := N;
   Subprogram 2                /* the second prime program starts here */
    while(v <> a[x].key or l <= r)
                               /* it compares v with the element at the
                                  middle position  of the table.  If v
                                  is smaller, then it must be in the
                                  first half of the table; if v is
                                  greater, then it must be in the
                                  second half of the table */
     begin
      Subprogram 3             /* the third prime program starts here */
       x := (l + r) div 2;
        Subprogram 4           /* the fourth prime program starts here */
         if(v < a[x].key) then
           r := x - 1;
         else
           l := x + 1;
        end Subprogram 4       /* the fourth prime program ends here */
     end Subprogram 3          /* the third prime program ends here */
     end;
    Subprogram 5               /* the fifth prime program starts here */
     if(v = a[x].key) then
       binarysearch := x;
     else
       binarysearch := N + 1;
    end Subprogram 5           /* the fifth prime program ends here */
   end Subprogram 2            /* the second prime program ends here */
 end Subprogram 1              /* the first prime program ends here */
end;
```

The third test input is taken from Sedgewick's text [Sedgewick 88]. It is a procedure that deals with Gauss-Jordan elimination, the array to be processed, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

The following program represents the forward-elimination phase of
Gaussian elimination.  The major variables used in the procedure, which
are global, are:

a: This the array that contains the data to be processed.
N: This is the number of elements in the array a.

*************************************************************************/

Procedure eliminate
var i, j, k, max: integer;
      t: real;
begin
 i := 1;
 while(i<N)                        /* for each i from 1 to N, we scan down
                                      the ith column to find the largest
                                      element (in rows past the ith).  The
                                      row containing this element is
                                      exchanged with the ith, and then the
                                      ith variable is eliminated in the
                                      equations i+1 to N */
  begin
   max := i;
   j := i + 1;
   while(j<N)
    begin
     if(abs(a[j,i]) > abs(a[max,i])) then
      max := j;
      j := j + 1;
    end;
   k := i;
   while(k < N + 1)
    begin
     t := a[i,k];
     a[i,k] := a[max,k];
     a[max,k] := t;
     k := k + 1;
    end;
   j := i + 1;
   while(j<N)
    begin
     k := N + 1;
     while(k > i)                  /* eliminate the ith variable in the jth
      begin                           equation */
        a[j,k] := a[j,k] - a[i,k]*a[j,i]/a[i,j];
        k := k - 1;
      end;
     j := j + 1;
    end;
  i := i + 1;
  end;
end;
```

The m-graph representation of test input 3 program is presented in Figure 19.

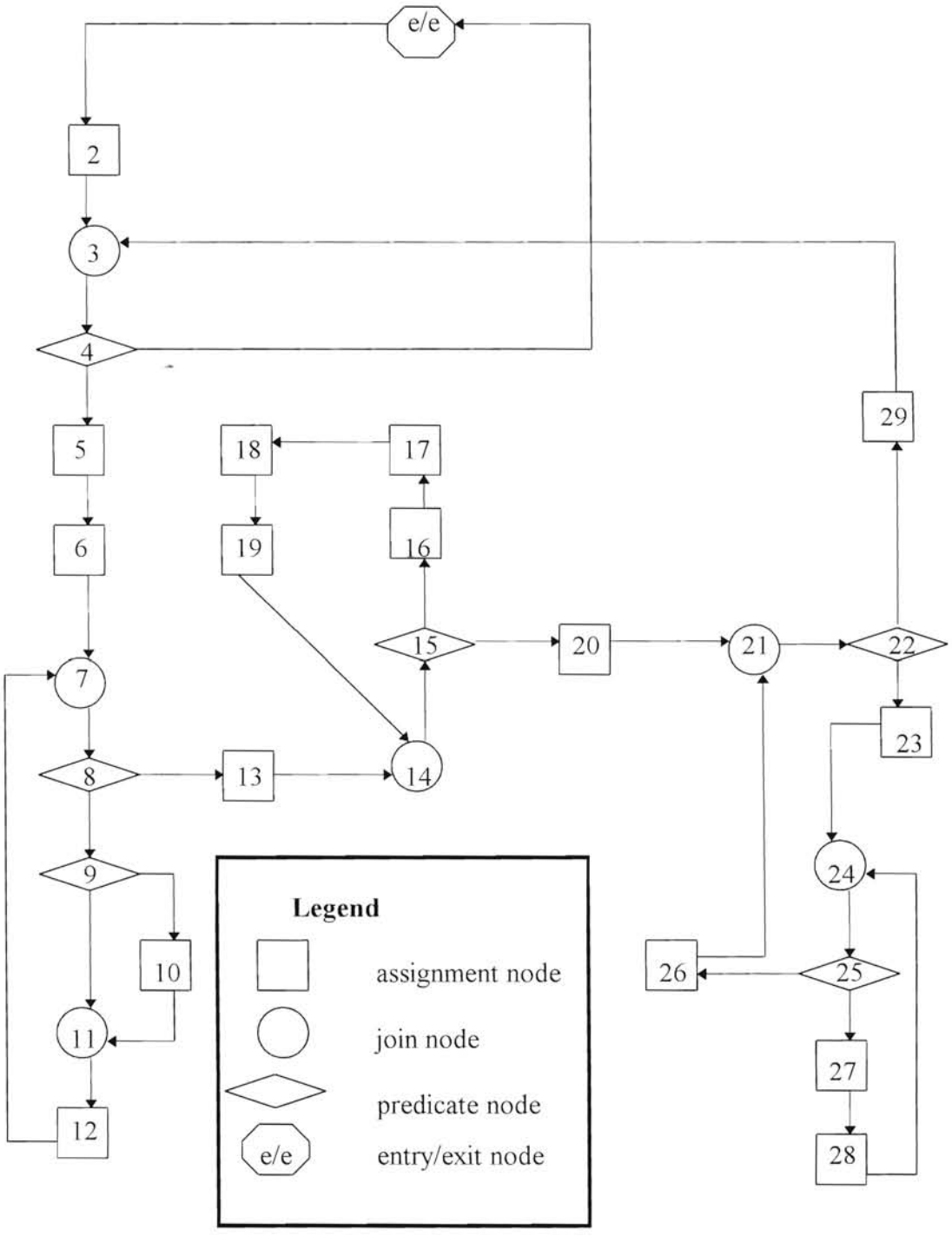


Figure 19. The m-graph of test input 3 program

The decomposed spanning tree of test input 3 m-graph is presented in Figure 20.

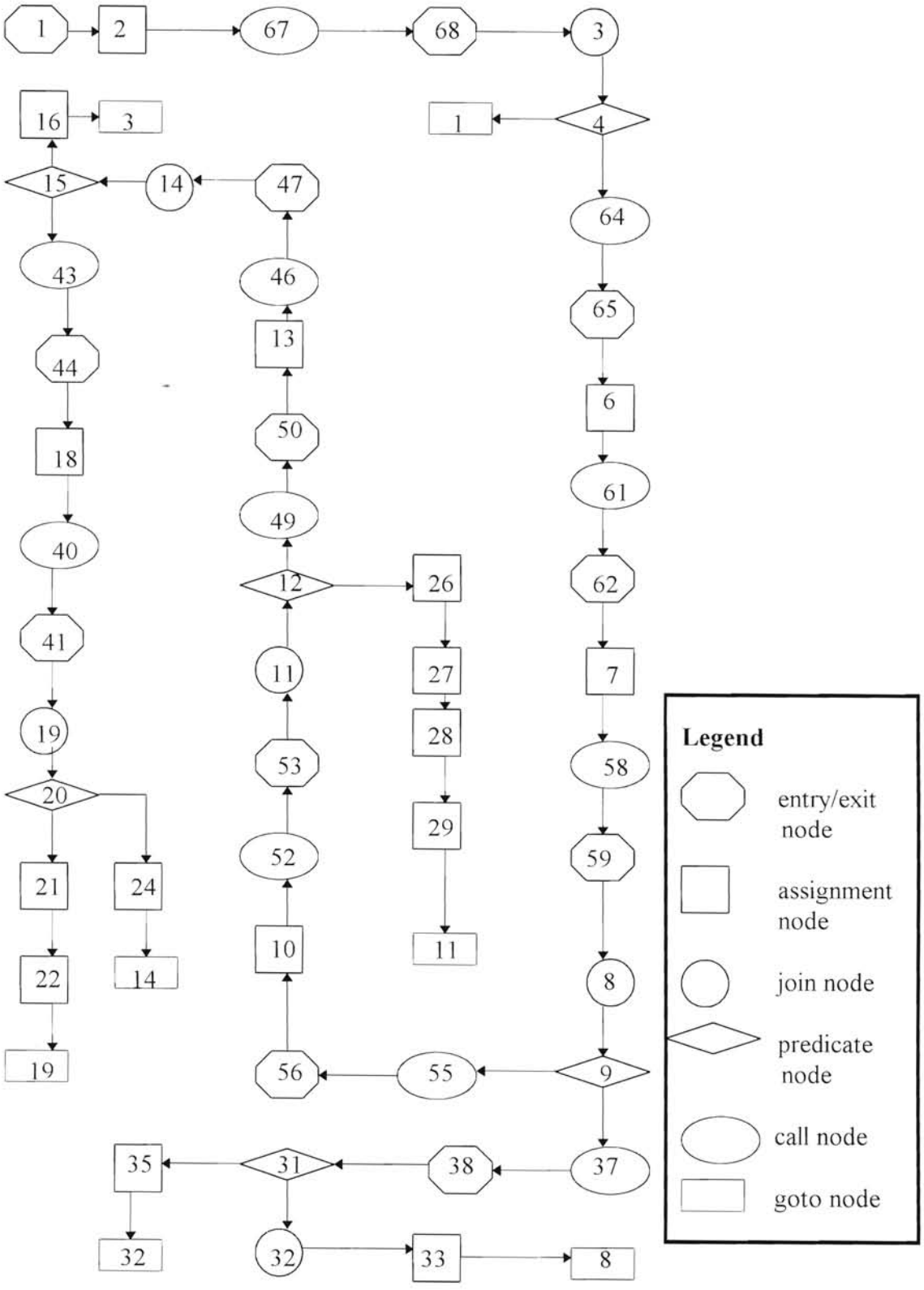


Figure 20. This is the decomposed spanning tree of the m-graph of Figure 19

The original program of test input 3 after being decomposed is presented here.

```
/*************************************************************************

The following program represents the forward-elimination phase of
Gaussian elimination.  The major variables used in the procedure, which
are global, are:

a: This the array that contains the data to be processed.
N: This is the number of elements in the array a.

*************************************************************************/

Procedure eliminate
var i, j, k, max: integer;
      t: real;
begin
 i := 1;
 Subprogram 1                    /* the first prime program starts here */
  while(i<N)   -                 /* for each i from 1 to N, we scan down
                                    the ith column to find the largest
                                    element (in rows past the ith).  The
                                    row containing this element is
                                    exchanged with the ith, and then the
                                    ith variable is eliminated in the
                                    equations i+1 to N */

   begin
    max := i;
     Subprogram 2                 /* the second prime program starts here */
      j := i + 1;
      while(j<N)
       begin
        Subprogram 3              /* the third prime program starts here */
         if(abs(a[j,i]) > abs(a[max,i])) then
         max := j;
         j := j + 1;
        end Subprogram 3          /* the third prime program ends here */
       end;
     Subprogram 4                 /* the fourth prime program starts here */
      k := i;
     Subprogram 5                 /* the fifth prime program starts here */
      while(k < N + 1)
       begin
        t := a[i,k];
        a[i,k] := a[max,k];
        a[max,k] := t;
        k := k + 1;
       end;
      Subprogram 6                /* the sixth prime program starts here */
        j := i + 1;
       Subprogram 7               /* the seventh prime program starts here
                                     */
        while(j<N)
        begin
         Subprogram 8             /* the eighth prime program starts here */
          k := N + 1;
          Subprogram 9            /* the nineth prime program starts here */
           while(k > i)           /* eliminate the ith variable in the jth
                                     equation */
            begin
             a[j,k] := a[j,k] - a[i,k]*a[j,i]/a[i,j];
```

```
            k := k - 1;
          end;
        j := j + 1;
       end Subprogram 9     /* the nineth prime program ends here */
      end Subprogram 8      /* the eighth prime program ends here */
     end;
    i := i + 1;
   end Subprogram 7         /* the seventh prime program ends here */
  end Subprogram 6          /* the sixth prime program ends here */
 end Subprogram 5           /* the fifth prime program ends here */
end Subprogram 4            /* the fourth prime program ends here */
end Subprogram 2            /* the second prime program ends here */
end Subprogram 1            /* the first prime program ends here */
 end;
end;
```

The fourth test input is taken from Premkumar's thesis [Premkumar 94]. The algorithm whose flow graph is to be decomposed follows.

```
/*****************************************************************************

The following program adds 5 to the input variable x five times, 6
twice, and 2 thrice.  Each time if the value of x is less than 1000, the
program adds 1 to x the difference of k and j times.

*****************************************************************************/

#include <stdio.h>
main()
{
 int i, j, k, x, m, h1, h2, h3, h4, h5;
 scanf("%d %d %d %d", &i, &j, &k, &x);
 i = 0;
 while(i < 10)"                  /* do 10 iterations, which are: adding 5
                                    to x 5 times, adding 6 to x 2 times,
                                    and adding 2 to x 3 times */
  {
    if(i < 7)                    /* if it didn't reach the 7th iteration,
                                    this means it is still either adding 5
                                    to x or adding 6 to x.  Otherwise, it
                                    should start adding 2 to x */
     {
       if(i < 5)                 /* if it didn't reach the 5th iteration,
                                    then keep adding 5 to x.  Otherwise,
                                    start adding 6 to x */
         x = x + 5;
       else
         x = x + 6;
     }
    else
      x = x + 2;
    m = k;
    while(k < j)                 /* this loop serves for adding 1 to x k-j
                                    times if x is less than 1000 */
     {
       if(x < 1000)
          x = x + 1;
       k = k + 1;
     }
    k = m;
    i = i + 1;
  }
 printf("%d",x);
}
```

The m-graph representation of test input 4 program is presented in Figure 21.

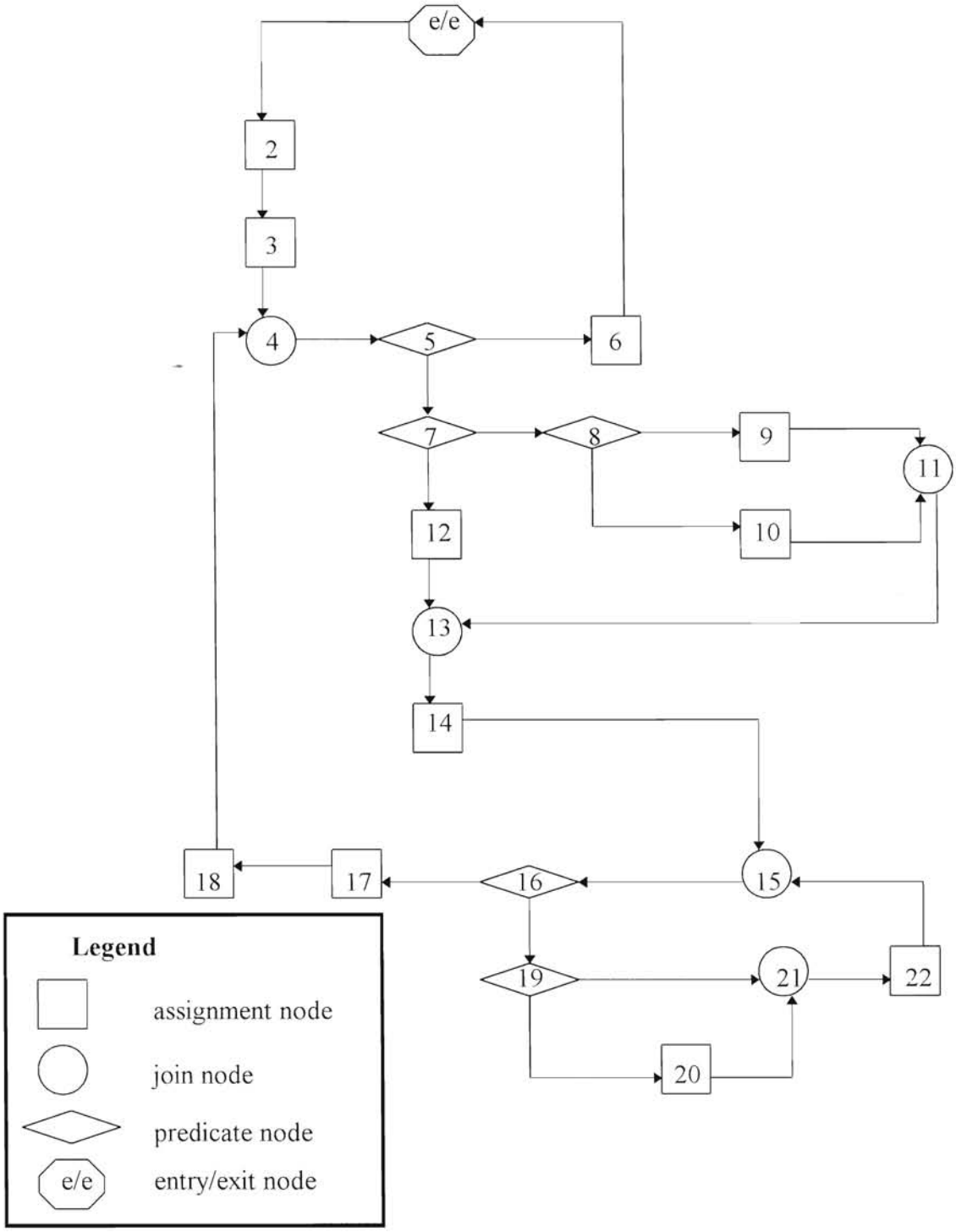


Figure 21. The m-graph of test input 4 program

The decomposed spanning tree of test input 4 m-graph is presented in Figure 22.

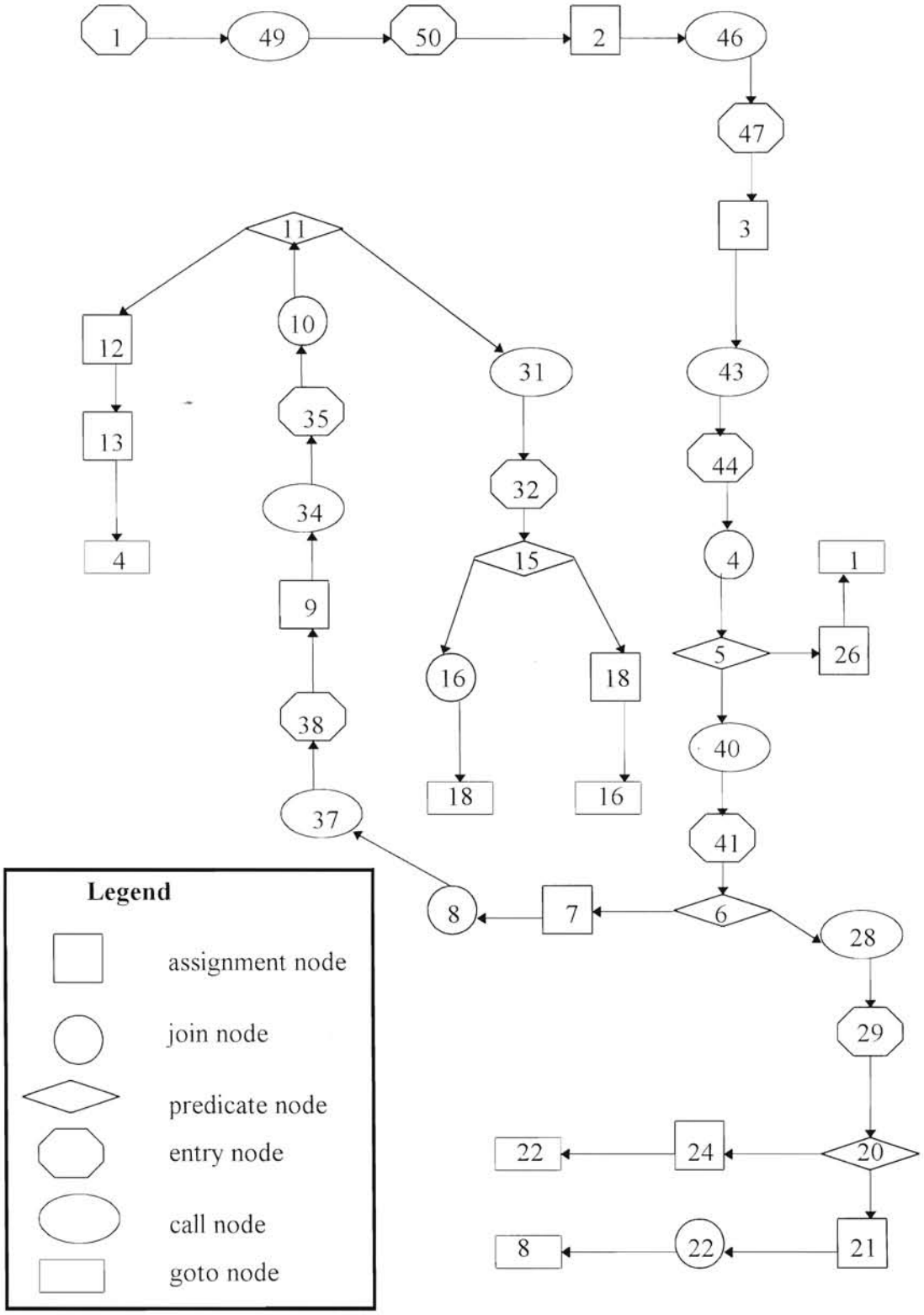


Figure 22. This is the decomposed spanning tree of the m-graph of Figure 21

The test input 4 after being decomposed is represented here.

```
/*************************************************************************

The following program adds 5 to the input variable x five times, 6
twice, and 2 thrice.  Each time if the value of x is less than 1000, the
program adds 1 to x the difference of k and j times.

*************************************************************************/

#include <stdio.h>
main()
{
 int i, j, k, x, m, h1, h2, h3, h4, h5;
Subprogram 1                    /* the first prime program starts here */
 scanf("%d %d %d %d", &i, &j, &k, &x);
  Subprogram 2                  /* the second prime program starts here */
   i = 0;
    Subprogram_3                /* the third prime program starts here */
     while(i < 10)              /* do 10 iterations, which are: adding 5
                                   to x 5 times, adding 6 to x 2 times,
                                   and adding 2 to x 3 times */

     {
      Subprogram 4              /* the fourth prime program starts here */
       if(i < 7)                /* if it didn't reach the 7th iteration,
                                   this means it is still either adding 5
                                   to x or adding 6 to x.  Otherwise, it
                                   should start adding 2 to x */

        {
          Subprogram 5          /* the fifth prime program starts here */
           if(i < 5)            /* if it didn't reach the 5th iteration,
                                   then keep adding 5 to x.  Otherwise,
                                   start adding 6 to x */

            x = x + 5;
           else
            x = x + 6;
           end Subprogram 5     /* the fifth prime program ends here */
        }
     else
      x = x + 2;
      Subprogram 6              /* the sixth prime program starts here */
       m = k;
       Subprogram 7             /* the seventh prime program starts here
                                   */
       while(k < j)             /* this loop serves for adding 1 to x k-j
                                   times if x is less than 1000 */

       {
         Subprogram 8           /* the eighth prime program starts here */
          if(x < 1000)
           x = x + 1;
          k = k + 1;
         end Subprogram 8       /* the eighth prime program ends here */
       }
       k = m;
       i = i + 1;
      end Subprogram 7          /* the seventh prime program ends here */
    end Subprogram 6            /* the sixth prime program ends here */
   end Subprogram 4             /* the fourth prime program ends here */
}
 printf("%d",x);
end Subprogram 3                /* the third prime program ends here */
```

```
 end Subprogram 2                /* the second prime program ends here */
end Subprogram 1                 /* the first prime program ends here */
}
```

The fifth test input is taken from Sedgewick's text [Sedgewick 88]. It is a function that deals with searching a string (string processing). The array to be processed, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This function does string checking.   It checks for each possible
position in the text at which the pattern could match, whether it does
in fact match.  The following program searches in this way for the first
occurrence of a pattern p[1..M] in a text string a[1..N].   The major
variables used in this function, which are global, are:

M: It represents the number of characters in the string, to be searched
   for.
N: It represents the dimension of the array to be searched for the
   string.
a: It is the array to be searched for the string.
p: It is an array containing the string to be searched.

*************************************************************************/

Function brutesearch: integer
Var   k,                          /* it is a pointer into the text */
       j: integer;                /* it is a pointer into the pattern */
begin
k := 1;
j := 1;
repeat
  if(a[k] = p[j])                 /* if the two pointers are pointing to a
                                     matching character, both of them are
                                     incremented */
   then
     begin
       k := k + 1;
       j := j + 1;
     end;
  else
   begin                          /* if j and k point to mismatching
                                     characters, then j is reset to point
                                     to the beginning of the pattern and i
                                     is reset to correspond to moving
                                     the pattern to the right one position
                                     for matching against text */
     k := k - j + 2;
     j := 1;
   end;
until(j > M or k > N)
if(j > M) then                    /* if the end of the pattern is reached
                                     (j > M), then a match has been found
                                     */
  brutesearch := k - M;
else
  brutesearch := k;
end.
```

The m-graph representation of test input 5 program is presented in Figure 23.

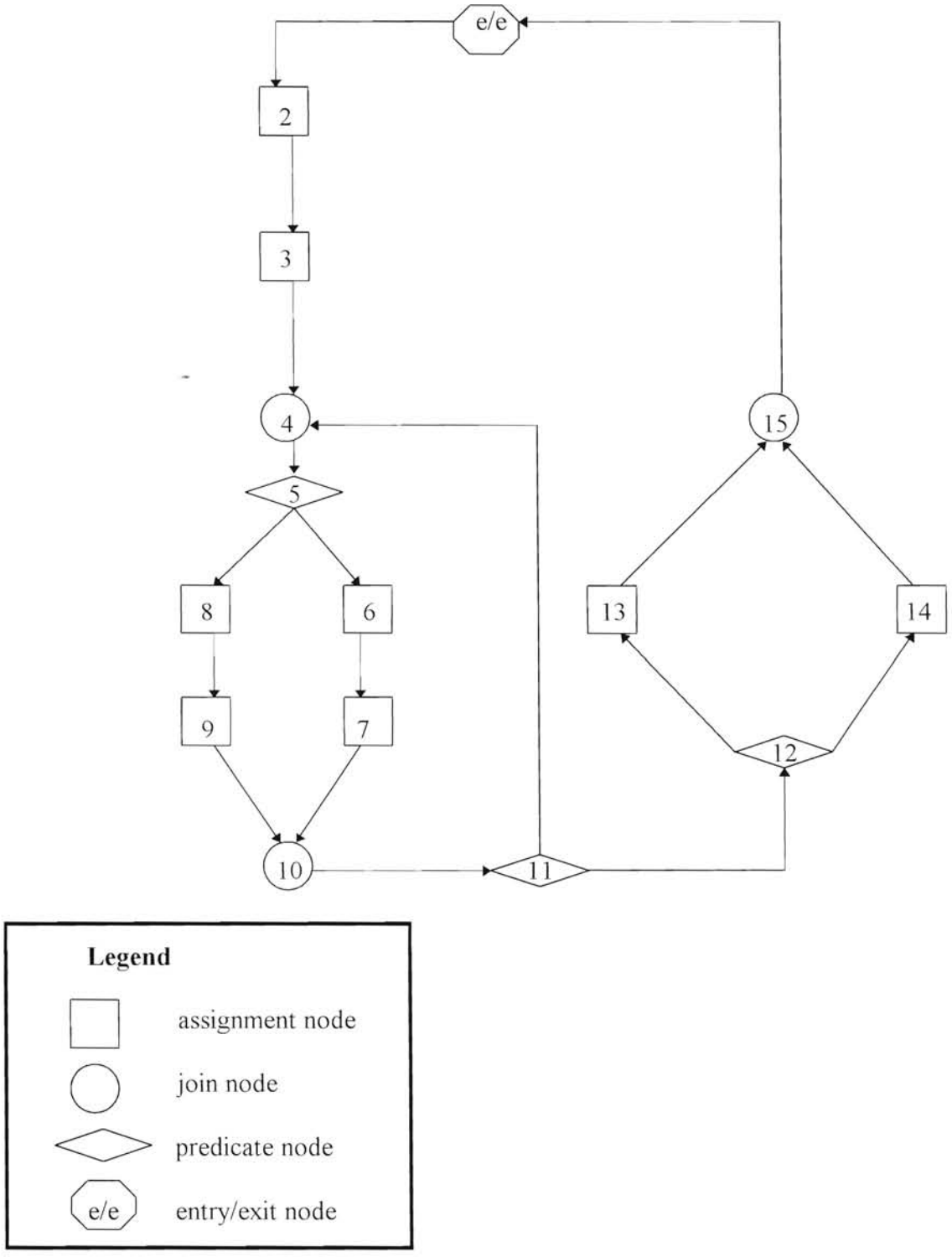


Figure 23. The m-graph of test input 5 program

The decomposed spanning tree of test input 5 m-graph is presented in Figure 24.

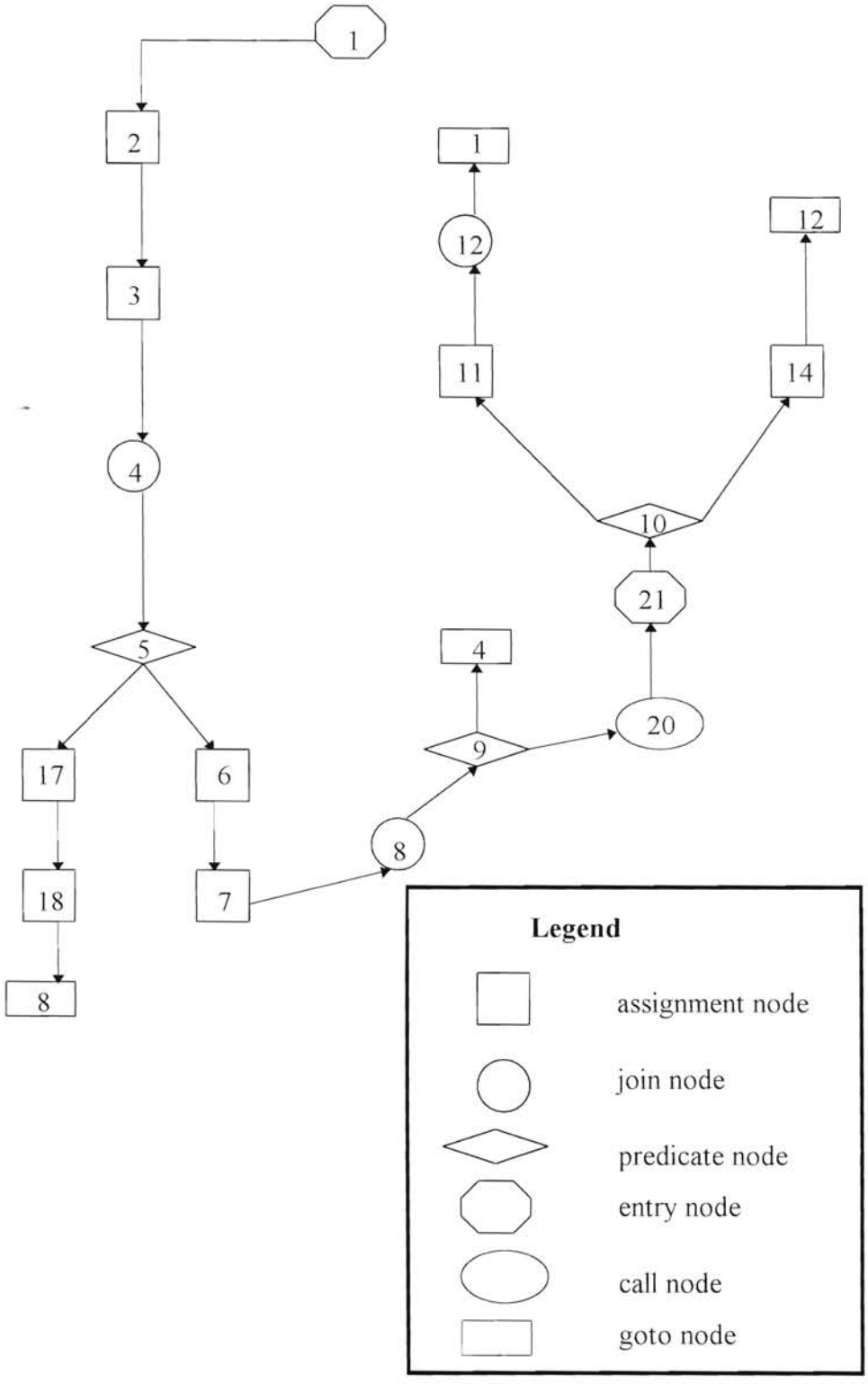


Figure 24. This is the decomposed spanning tree of the m-graph of Figure 23

The original program of test input 5 after being decomposed into prime graphs is presented here.

```
/*********************************************************************

This function does string checking.   It checks for each possible
position in the text at which the pattern could match, whether it does
in fact match.  The following program searches in this way for the first
occurrence of a pattern p[1..M] in a text string a[1..N].   The major
variable used in this function, which are global, are:

M: It represents the number of characters in the string, to be searched
   for.
N: It represents the dimension of the array to be searched for the
   string.
a: It is the array to be searched for the string.
p: It is an array containing the string to be searched.

*********************************************************************/

Function brutesearch: integer
Var   k,                          /* it is a pointer into the text */
      j: integer;                 /* it is a pointer into the pattern */
begin
k := 1;
j := 1;
repeat
  if(a[k] = p[j])                 /* if the two pointers are pointing to a
                                     matching character, both of them are
                                     incremented */
   then
     begin
       k := k + 1;
       j := j + 1;
     end;
  else
   begin                          /* if j and k point to mismatching
                                     characters, then j is reset to point
                                     to the beginning of the pattern and i
                                     is reset to correspond to moving
                                     the pattern to the right one position
                                     for matching against text */
     k := k - j + 2;
     j := 1;
   end;
until(j > M or k > N)
 Subprogram 1                     /* the first prime program starts here */
  if(j > M) then                  /* if the end of the pattern is reached
                                     (j > M), then a match has been found
                                     */
    brutesearch := k - M;
  else
    brutesearch := k;
 end Subprogram 1                 /* the first prime program ends here */
end.
```

The six test input representing an m-graph taken from Forman's thesis [Forman 79] and presented in Figure 25.

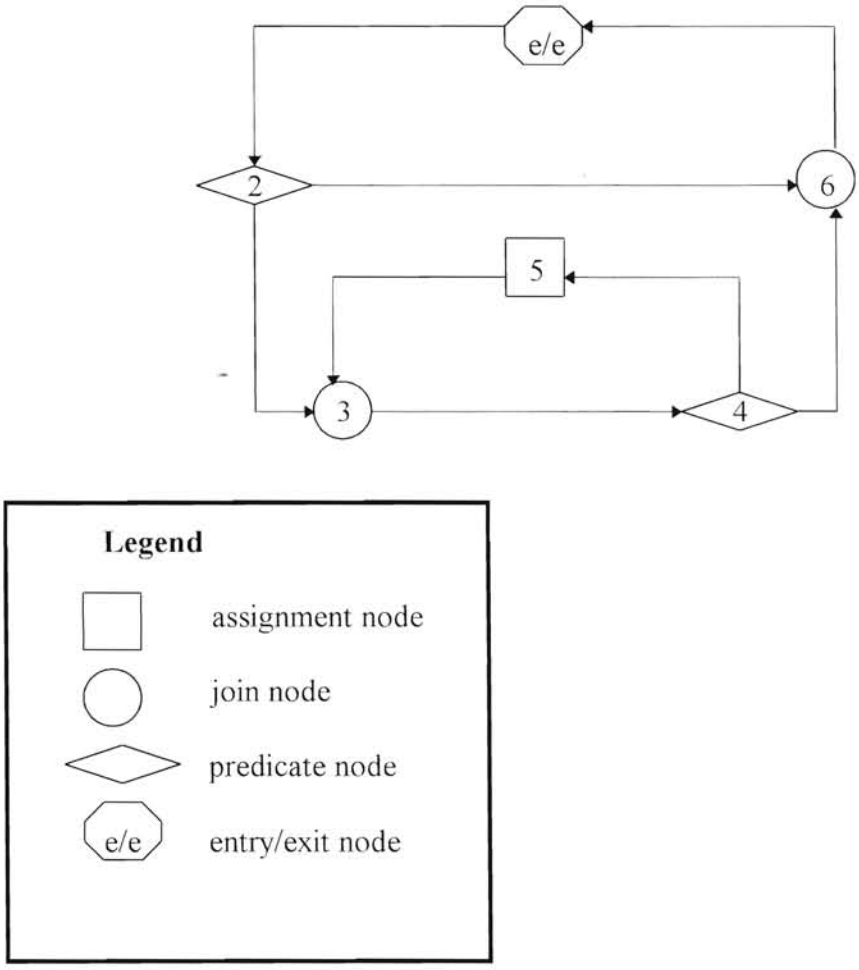


Figure 25. The m-graph of test input 6

The decomposed spanning tree of test input 6 m-graph is presented in Figure 26.

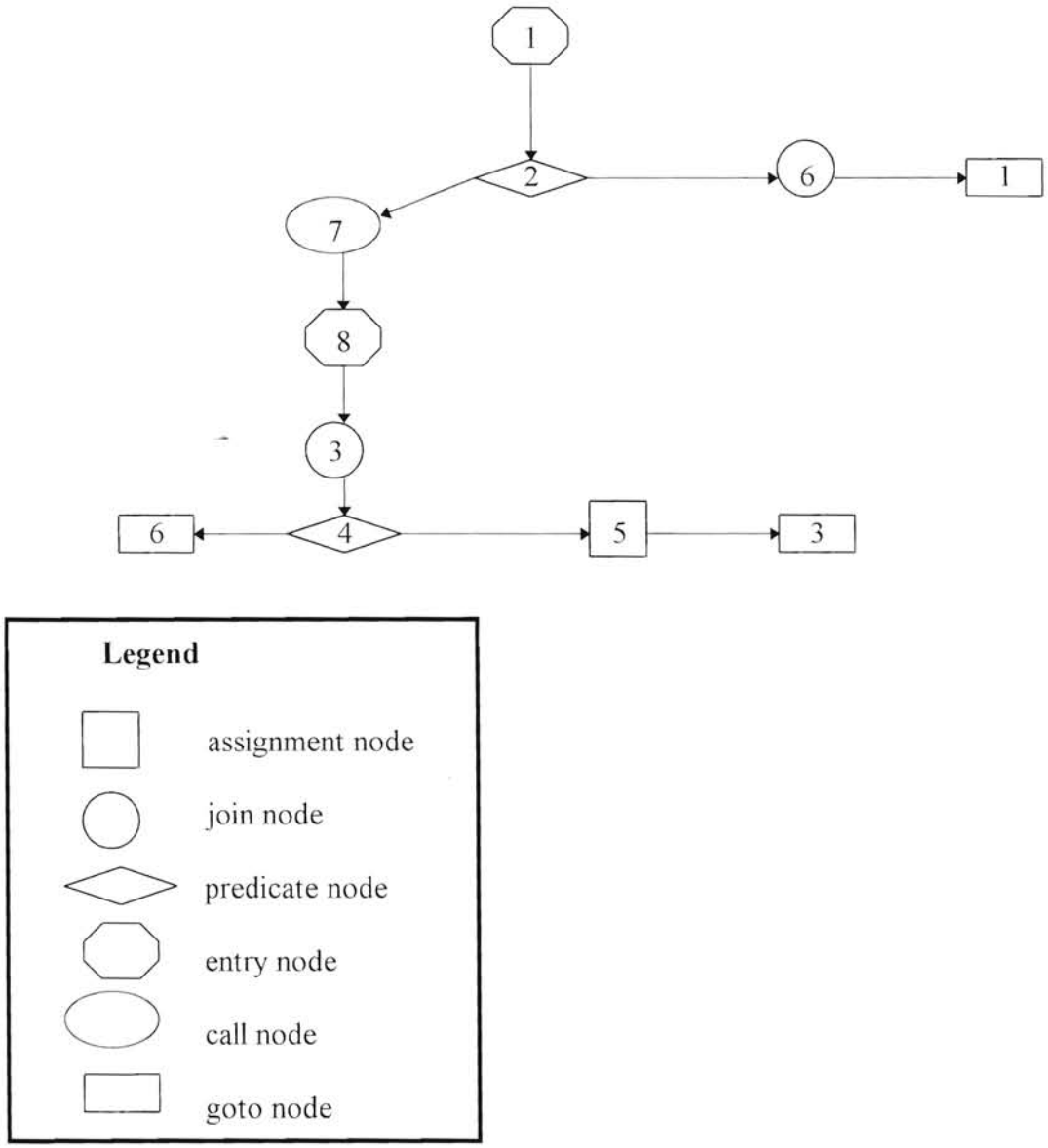


Figure 26. This is the decomposed spanning tree of the m-graph of Figure 25

Cunningham's algorithm was tested with six different inputs. The first test input is taken from Sedgewick's text [Sedgewick 88]. It is a procedure that deals with sorting an array using the insertion technique, the array to be sorted is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This procedure job is to sort an array, a, using insertion techniques.
The major variables used in the procedure, which are global, are:

a: This is the array that contains the data to be sorted.
p: It is a pointer array that is manipulated, to restrict accessing the
   original array only for comparisons.
N: This is the number of elements in a.

*************************************************************************/

Procedure insertion
var    i,                  /* used as loop variable */
       j, v: integer;      /* variables used by the arrays a and p for the
                              comparison */
begin
  i := 1;
  while(i<N)               /* the loop initializes the P array in order to
                              produce an algorithm that will sort the index
                              array */
     begin
       P[i] := i;
       i := i + 1;
     end;
  i : = 2;
  while(i<N)               /* this loop and compare the elements in a, that
                              are indexed by the array p.  Process and
                              adjustments will happen in the array p.  At
                              the end of the algorithm, the index array
                              will be sorted so that p[1] is the smallest
                              element in the array a */
     begin
       v := p[i];
       j := i;
       while(a[p[j-1]] > a[v]) do
        begin
          p[j] := p[j-1];
          j := j - 1;
        end;
       p[j] := v;
       i := i + 1;
     end;
end;
```

Figure 27 presents the digraph of the algorithm in test input 1.

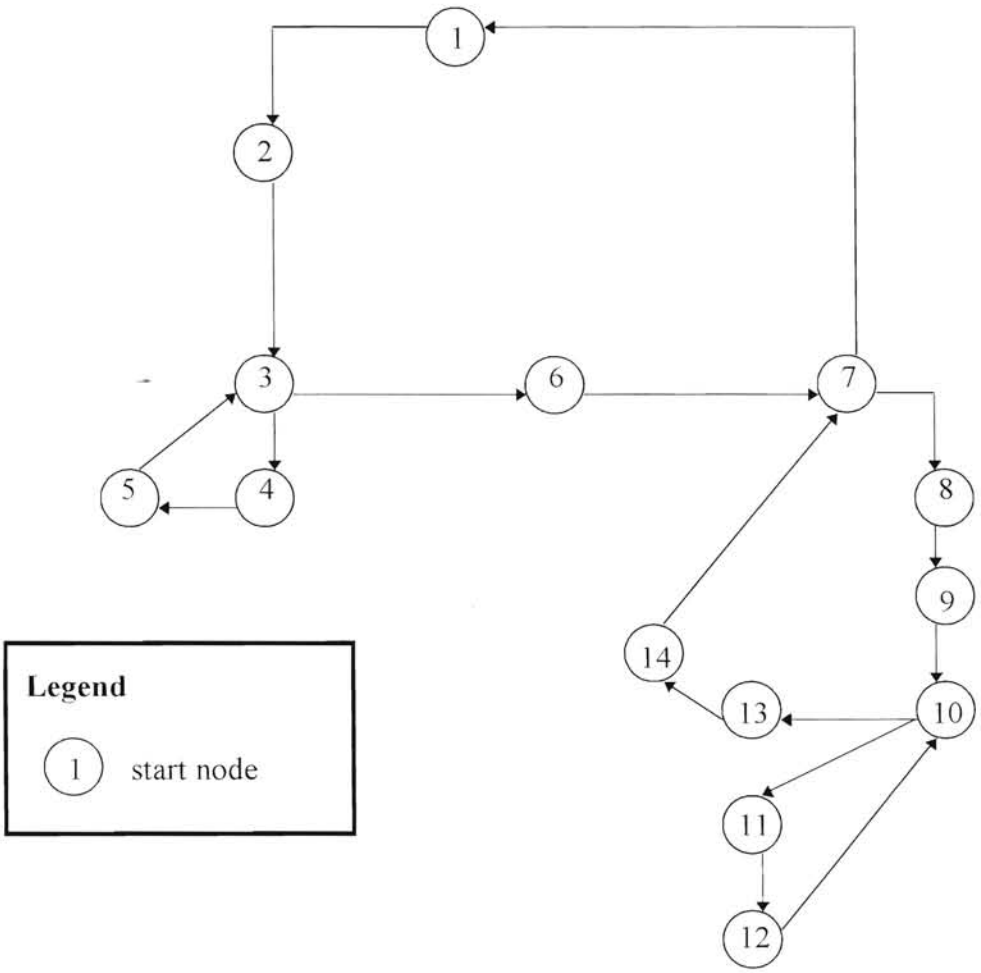
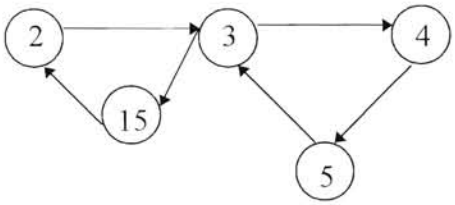


Figure 27. The digraph of test input 1 algorithm

The two digraphs that resulted from the decomposition of test input 1 digraph are presented in Figure 28, where node 15 is the marker.

Digraph 1:



Digraph 2:

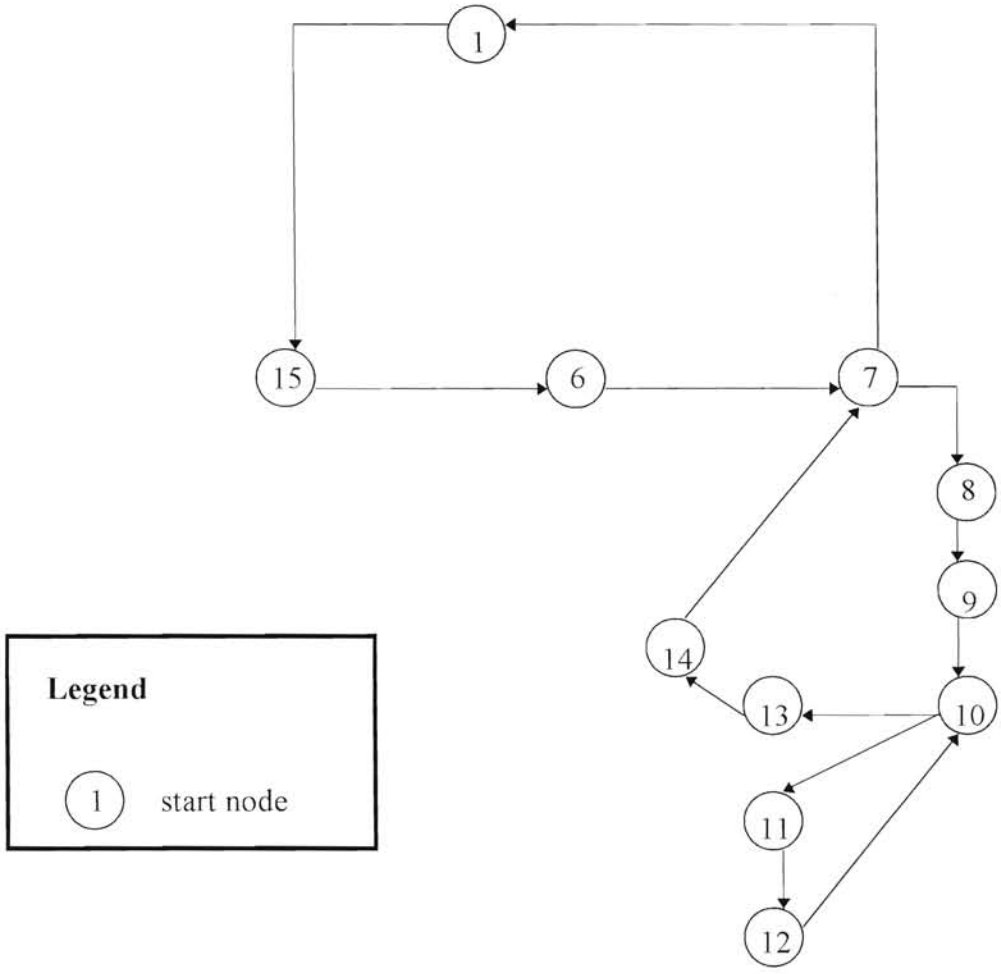


Figure 28. The two digraphs resulting from the decomposition of the digraph of Figure 27

The algorithm of test input 1 after being decomposed is presented here.

```
/*************************************************************************

This procedure job is to sort an array, a, using insertion techniques.
The major variables used in the procedure, which are global, are:

a: This is the array that contains the data to be sorted.
p: It is a pointer array that is manipulated, to restrict accessing the
   original array only for comparisons.
N: This is the number of elements in a.

*************************************************************************/

Procedure insertion
var    i,                          /* used as loop variable */
       j, v: integer;              /* variables used by the arrays a and p
                                      for the comparison */
begin

   Subprogram 1                    /* the first decomposed digraph starts
                                      here */
     i := 1;
     while(i<N)                    /* the loop initializes the P array in
                                      order to produce an algorithm that will
                                      sort the index array */

       begin
         P[i] := i;
         i := i + 1;
       end;
   end Subprogram 1                /* the first decomposed digraph ends here
                                      */
  i : = 2;
  while(i<N)                       /* this loop and compare the elements in
                                      a, that are indexed by the array p.
                                      Process and adjustments will happen in
                                      the array p.  At the end of the
                                      algorithm, the index array will be
                                      sorted so that p[1] is the smallest
                                      element in the array a */

     begin
       v := p[i];
       j := i;
       while(a[p[j-1]] > a[v]) do
        begin
          p[j] := p[j-1];
          j := j - 1;
        end;
       p[j] := v;
       i := i + 1;
     end;
end;
```

The second test input is taken from Sedgewick's text [Sedgewick 88]. It is a function that deals with searching an array using the binary search technique, the array to be searched, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This function is to search for an element in an array a.  It uses the
binary search technique, which divides the set of records into two
parts, determines which of the two parts the key sought belongs to, then
concentrates on that part.  It keeps the set records sorted.  The major
global variables used in this function are:

a: It is an array of records, where key is the variable in the record
   that contains the numbers to be searched.
N: It represents the of elements to be searched.
v: It is the key to be searched for.

*************************************************************************/

Function binarysearch(v: integer): integer
Var x, l, r: integer;
begin
 l := 1;
 r := N;
 while(v <> a[x].key or l <= r)/* it compares v with the element at the
                                  middle position  of the table.  If v
                                  is smaller, then it must be in the
                                  first half of the table; if v is
                                  greater, then it must be in the second
                                  half of the table */
   begin
      x := (l + r) div 2;
       if(v < a[x].key) then
         r := x - 1;
       else
         l := x + 1;
   end;
 if(v = a[x].key) then
  binarysearch := x;
 else
  binarysearch := N + 1;
end;
```

Figure 29 presents the digraph of the algorithm in test input 2.

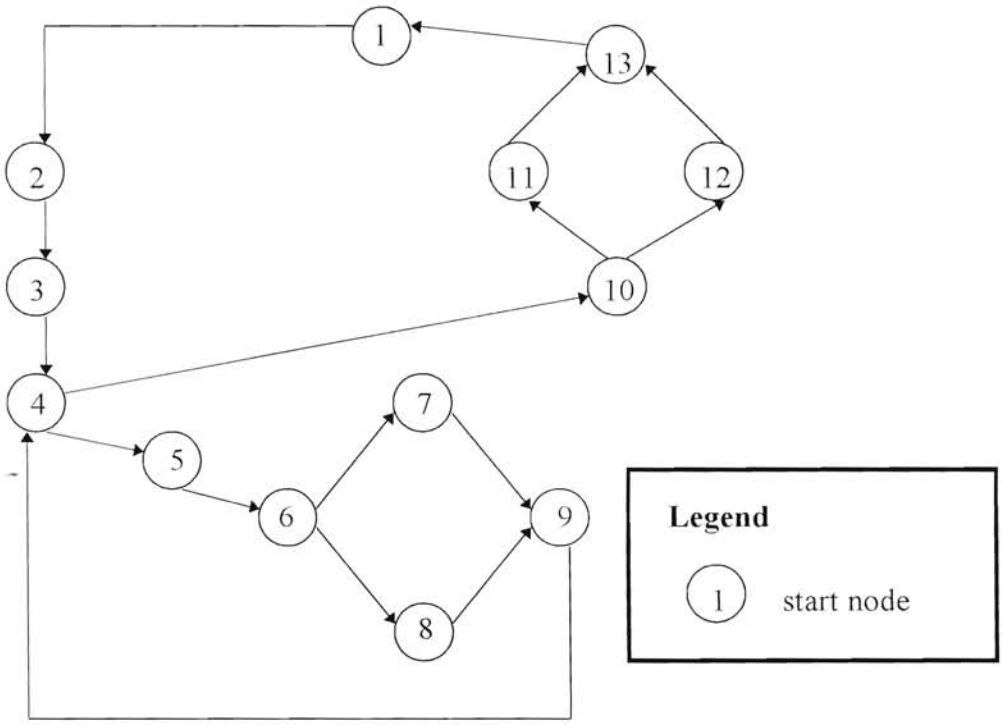
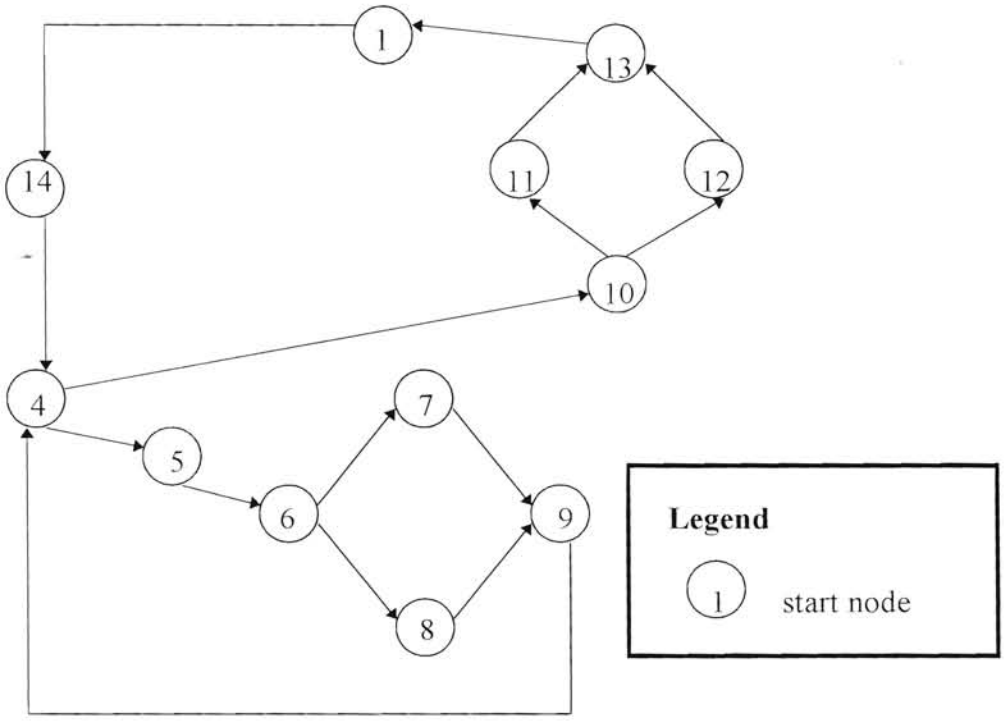


Figure 29. The digraph of test input 2 algorithm

The two digraphs that resulted from the decomposition of test input 2 digraph are presented in Figure 30, where node 14 is the marker.

Digraph 1:



Digraph 2:

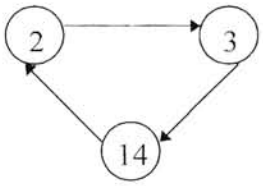


Figure 30. The two digraphs resulting from the decomposition of the digraph of Figure 29

The algorithm of test input 2 after being decomposed is presented here.

```
/*************************************************************************

This function is to search for an element in an array a.  It uses the
binary search technique, which divides the set of records into two
parts, determines which of the two parts the key sought belongs to, then
concentrates on that part.  It keeps the set records sorted.  The major
global variables used in this function are:

a: It is an array of records, where key is the variable in the record
   that contains the numbers to be searched.
N: It represents the of elements to be searched.
v: It is the key to be searched for.

*************************************************************************/

Function binarysearch(v: integer): integer
Var x, l, r: integer;
begin
 l := 1;
 r := N;
 Subprogram 1                    /* the first decomposed digraph starts
                                    here */
 while(v <> a[x].key or l <= r)/* it compares v with the element at the
                                    middle position  of the table.  If v
                                    is smaller, then it must be in the
                                    first half of the table; if v is
                                    greater, then it must be in the second
                                    half of the table */
   begin
     x := (l + r) div 2;
       if(v < a[x].key) then
         r := x - 1;
       else
         l := x + 1;
   end;
if(v = a[x].key) then
   binarysearch := x;
 else
   binarysearch := N + 1;
 end Subprogram 1                /* the first decomposed digraph ends here
                                    */
end;
```

The third test input is taken from Sedgewick's text [Sedgewick 88]. It is a procedure that deals with Gauss-Jordan elimination, the array to be processed, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

The following program represents the forward-elimination phase of
Gaussian elimination.  The major variables used in the procedure, which
are global, are:

a: This the array that contains the data to be processed.
N: This is the number of elements in the array a.

*************************************************************************/

Procedure eliminate
var i, j, k, max: integer;
      t: real;
begin
 i := 1;
 while(i<N)                        /* for each i from 1 to N, we scan down
                                      the ith column to find the largest
                                      element (in rows past the ith).  The
                                      row containing this element is
                                      exchanged with the ith, and then the
                                      ith variable is eliminated in the
                                      equations i+1 to N */
  begin
   max := i;
   j := i + 1;
   while(j<N)
    begin
     if(abs(a[j,i]) > abs(a[max,i])) then
      max := j;
      j := j + 1;
    end;
   k := i;
   while(k < N + 1)
    begin
     t := a[i,k];
     a[i,k] := a[max,k];
     a[max,k] := t;
     k := k + 1;
    end;
   j := i + 1;
   while(j<N)
    begin
     k := N + 1;
     while(k > i)                  /* eliminate the ith variable in the jth
      begin                           equation */
        a[j,k] := a[j,k] - a[i,k]*a[j,i]/a[i,j];
        k := k - 1;
      end;
     j := j + 1;
    end;
  i := i + 1;
  end;
end;
```

The digraph of the algorithm in test input 3 is presented in Figure 31.

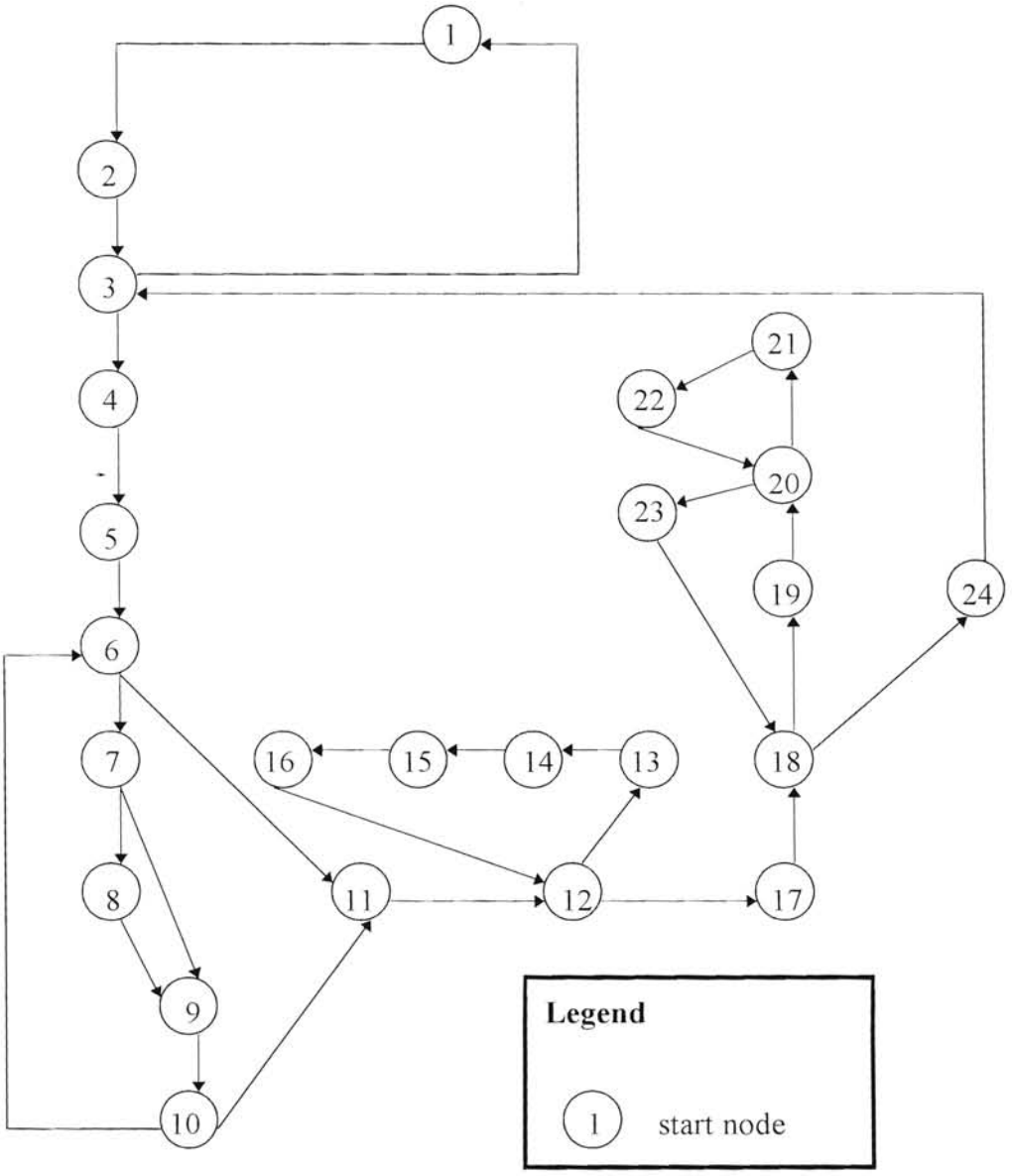


Figure 31. The digraph of test input 3 algorithm

The digraph of test input 3 could not be decomposed by Cunningham's algorithm.

The fourth test input is taken from Premkumar's thesis [Premkumar 94]. The algorithm whose flow graph is to be decomposed follows.

```
/*****************************************************************************

The following program adds 5 to the input variable x five times, 6 twice
and 2 thrice.  Each time if the value of x is less than 1000, the
program adds 1 to x the difference of k and j times.

*****************************************************************************/

#include <stdio.h>
main()
{
 int i, j, k, x, m, h1, h2, h3, h4, h5;
 scanf("%d %d %d %d", &i, &j, &k, &x);
 i = 0;
 while(i < 10)                  /* do 10 iterations, which are: adding 5
                                   to x 5 times, adding 6 to x 2 times,
                                   and adding 2 to x 3 times */
  {
    if(i < 7)                   /* if it didn't reach the 7th iteration,
                                   this means it is still either adding 5
                                   to x or adding 6 to x.  Otherwise, it
                                   should start adding 2 to x */
     {
       if(i < 5)                /* if it didn't reach the 5th iteration,
                                   then keep adding 5 to x.  Otherwise,
                                   start adding 6 to x */
         x = x + 5;
       else
         x = x + 6;
     }
    else
      x = x + 2;
    m = k;
    while(k < j)                /* this loop serves for adding 1 to x k-j
                                   times if x is less than 1000 */
     {
       if(x < 1000)
          x = x + 1;
       k = k + 1;
     }
    k = m;
    i = i + 1;
  }
 printf("%d",x);
}
```

Figure 32 presents the digraph of the algorithm in test input 4.

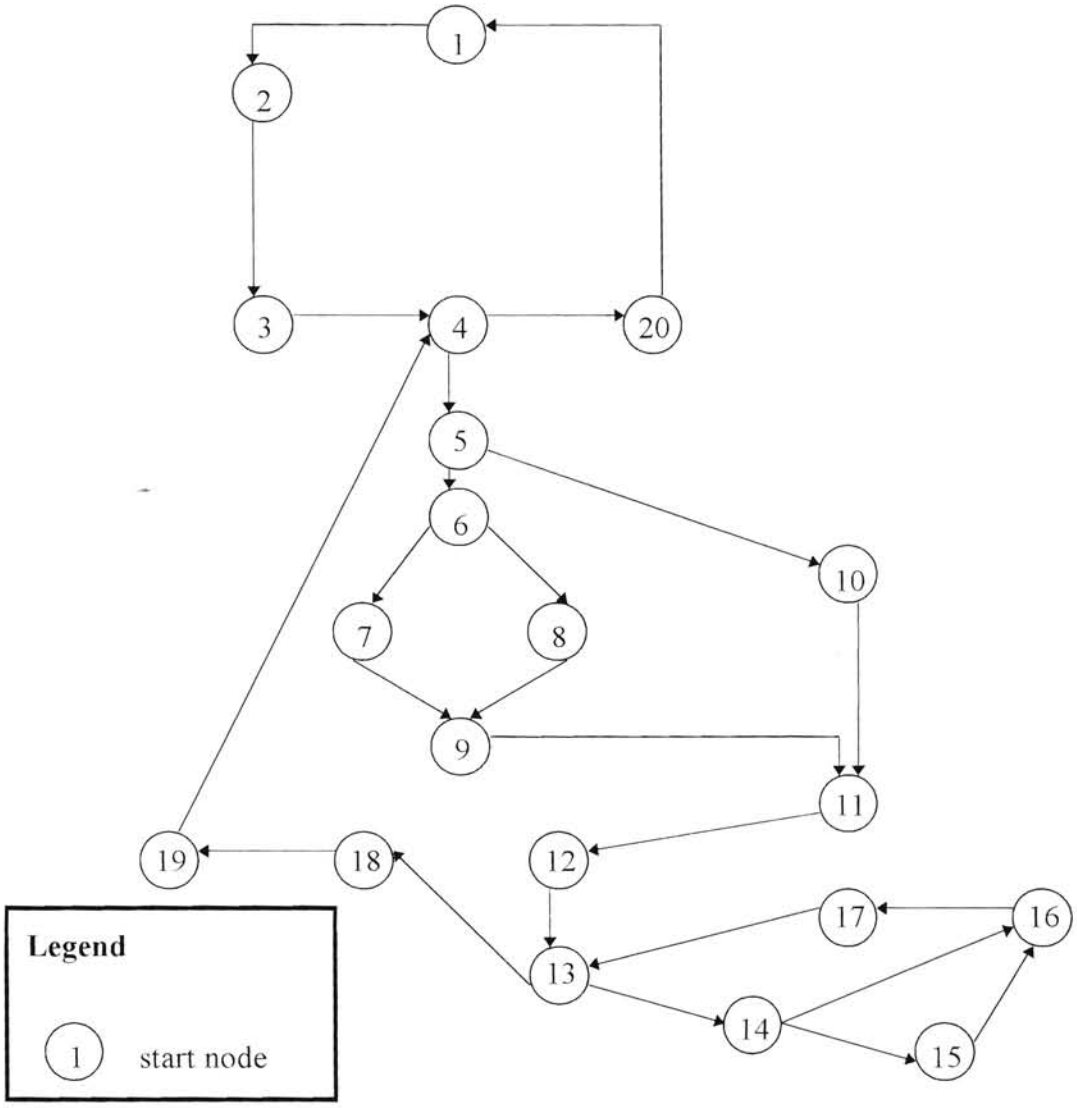
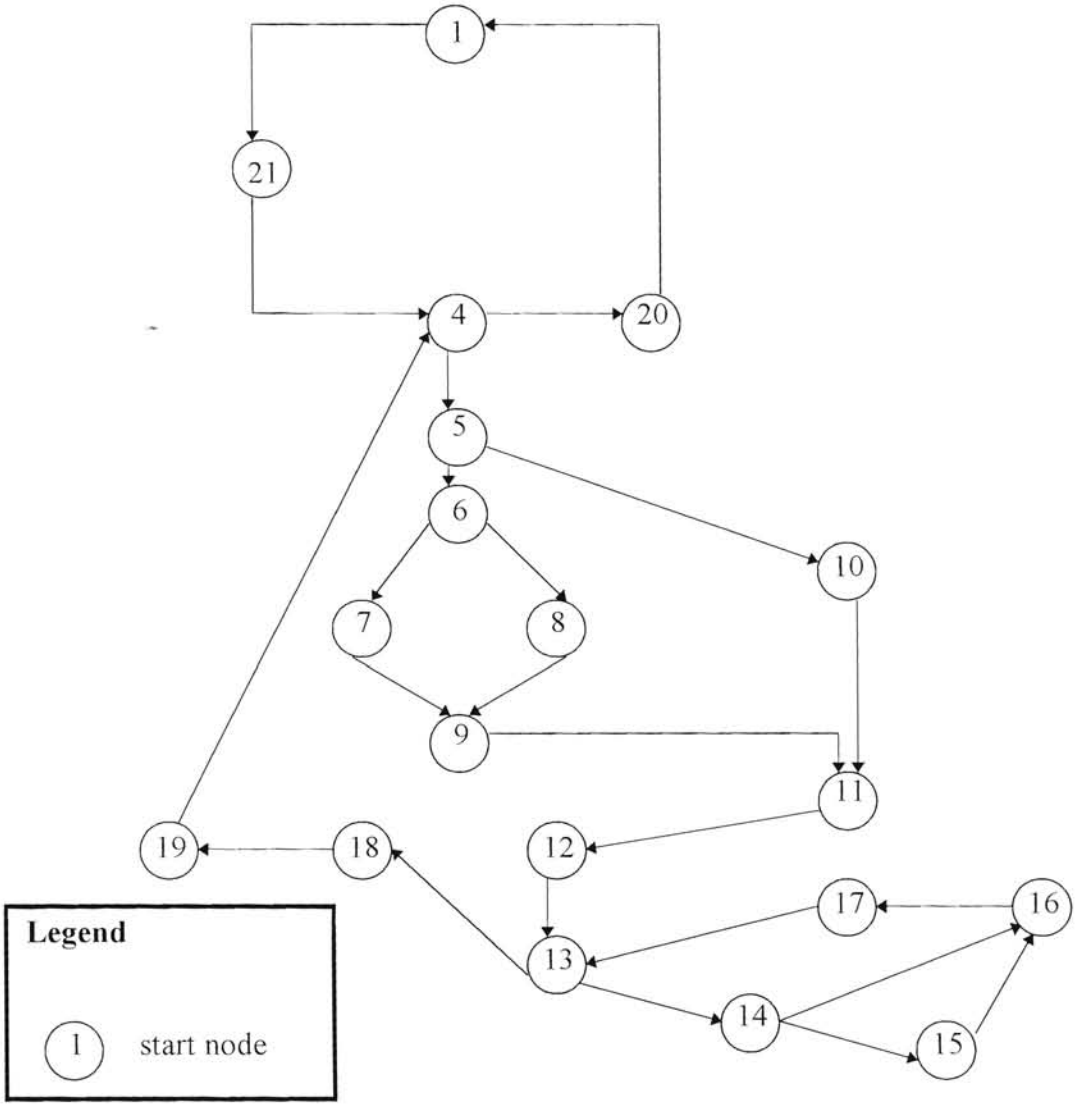


Figure 32. The digraph of test input 4 algorithm

The representation of the two digraphs that resulted from the decomposition of test input 4 digraph is shown in Figure 33, where node 21 is the marker.

Digraph 1:



Digraph 2:

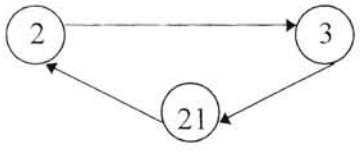


Figure 33. The two digraphs resulting from the decomposition of the digraph of Figure 32

The test input 4 algorithm after being decomposed is presented here.

```
/*************************************************************************

The following program adds 5 to the input variable x five times, 6 twice
and 2 thrice.  Each time if the value of x is less than 1000, the
program adds 1 to x the difference of k and j times.

*************************************************************************/

#include <stdio.h>
main()
{
 int i, j, k, x, m, h1, h2, h3, h4, h5;
 scanf("%d %d %d %d", &i, &j, &k, &x);
 i = 0;
  Subprogram 1                    /* the first decomposed digraph starts
                                     here */
    while(i < 10)                 /* do 10 iterations, which are: adding 5
                                     to x 5 times, adding 6 to x 2 times,
                                     and adding 2 to x 3 times */
     {
       if(i < 7)                  /* if it didn't reach the 7th iteration,
                                     this means it is still either adding 5
                                     to x or adding 6 to x.  Otherwise, it
                                     should start adding 2 to x */
        {
          if(i < 5)               /* if it didn't reach the 5th iteration,
                                     then keep adding 5 to x.  Otherwise,
                                     start adding 6 to x */
            x = x + 5;
          else
            x = x + 6;
        }
       else
        x = x + 2;
       m = k;
        while(k < j)              /* this loop serves for adding 1 to x k-j
                                     times if x is less than 1000 */
          {
            if(x < 1000)
               x = x + 1;
            k = k + 1;
          }
        k = m;
        i = i + 1;
     }
    printf("%d",x);
  end Subprogram 1                /* the first decomposed digraph ends here
                                     */

}
```

The fifth test input is taken from Sedgewick's text [Sedgewick 88]. It is a function that deals with searching a string (string processing). The array to be processed, a, is a global variable. The algorithm whose flow graph is to be decomposed follows.

```
/*************************************************************************

This function does string checking.  It checks for each possible
position in the text at which the pattern could match, whether it does
in fact match.  The following program searches in this way for the first
occurrence of a pattern p[1..M] in a text string a[1..N].  The major
variables used in this function, which are global, are:

M: It represents the number of characters in the string, to be searched
   for.
N: It represents the dimension of the array to be searched for the
   string.
a: It is the array to be searched for the string.
p: It is an array containing the string to be searched.

*************************************************************************/

Function brutesearch: integer
Var   k,                          /* it is a pointer into the text */
       j: integer;                /* it is a pointer into the pattern */
begin
k := 1;
j := 1;
repeat
  if(a[k] = p[j])                 /* if the two pointers are pointing to a
                                     matching character, both of them are
                                     incremented */
   then
     begin
       k := k + 1;
       j := j + 1;
     end;
  else
   begin                          /* if j and k point to mismatching
                                     characters, then j is reset to point
                                     to the beginning of the pattern and i
                                     is reset to correspond to moving
                                     the pattern to the right one position
                                     for matching against text */
     k := k - j + 2;
     j := 1;
   end;
until(j > M or k > N)
if(j > M) then                    /* if the end of the pattern is reached
                                     (j > M), then a match has been found
                                     */
  brutesearch := k - M;
else
  brutesearch := k;
end.
```

The digraph presentation of test input 5 algorithm is shown in Figure 34.

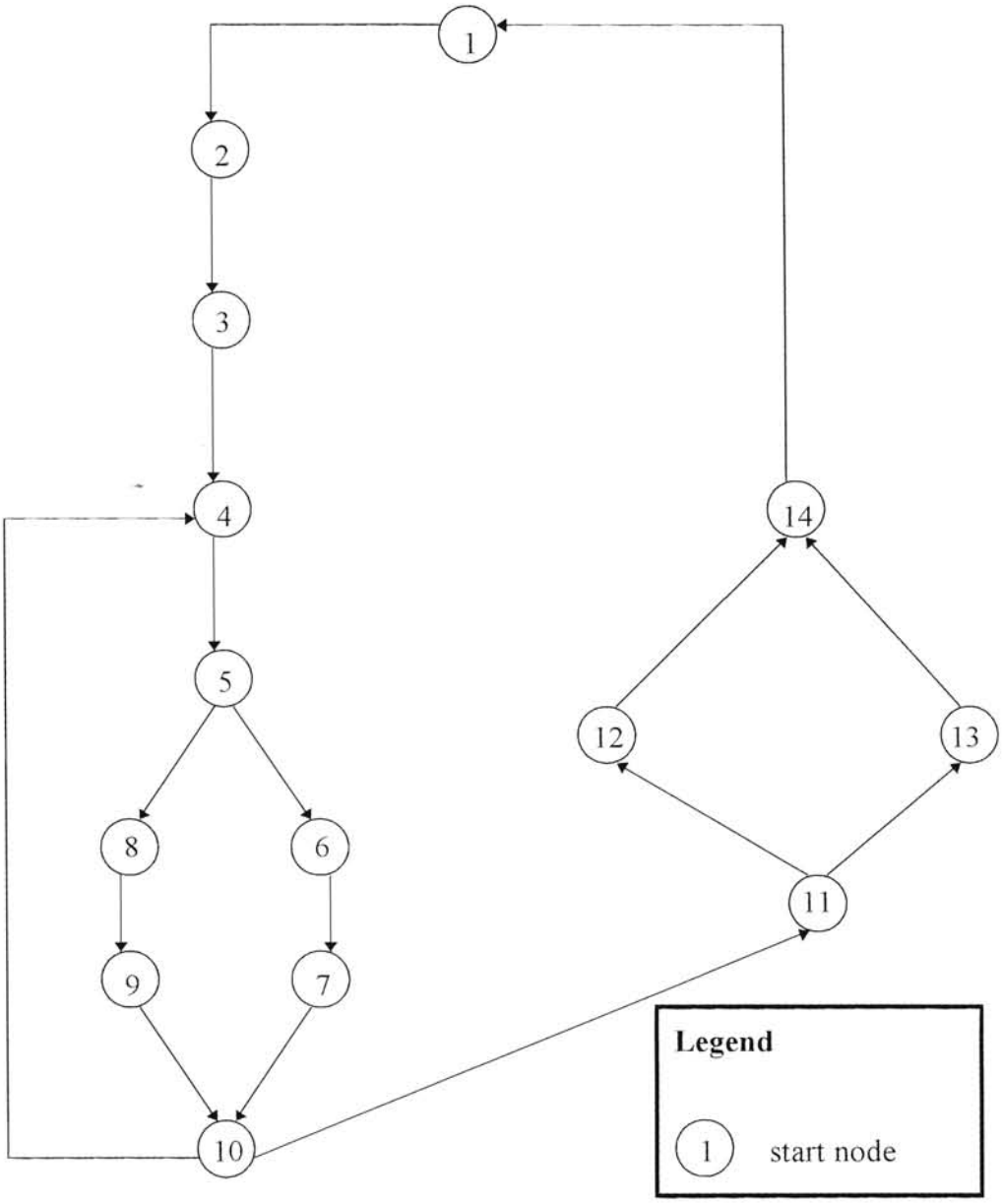
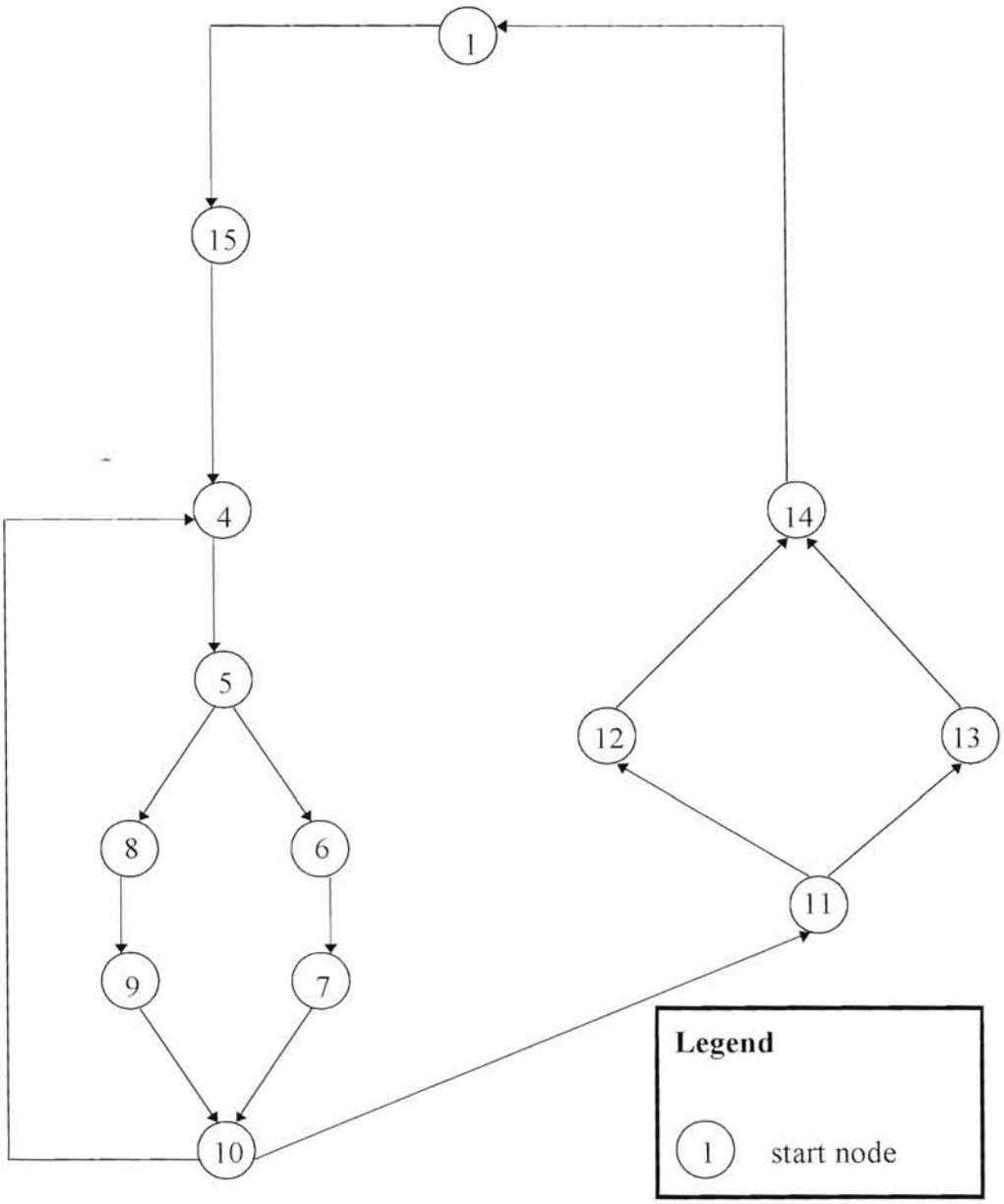


Figure 34. The digraph of test input 5 algorithm

The two digraphs that resulted from the decomposition of test input 5 digraph are presented in Figure 35, where node 15 is the marker.

Digraph 1:



Digraph 2:

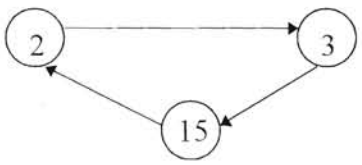


Figure 35. The two digraphs resulting from the decomposition of the digraph of Figure 34

The algorithm of test input 5 after being decomposed is presented here

```
/*************************************************************************

This function does string checking.   It checks for each possible
position in the text at which the pattern could match, whether it does
in fact match.  The following program searches in this way for the first
occurrence of a pattern p[1..M] in a text string a[1..N].   The major
variables used in this function, which are global, are:

M: It represents the number of characters in the string, to be searched
   for.
N: It represents the dimension of the array to be searched for the
   string.
a: It is the array to be searched for the string.
p: It is an array containing the string to be searched.

*************************************************************************/

Function brutesearch: integer
Var   k,                          /* it is a pointer into the text */
       j: integer;                /* it is a pointer into the pattern */
begin
k := 1;
j := 1;
 Subprogram 1                     /* the first decomposed digraph starts
                                     here */
  repeat
    if(a[k] = p[j])               /* if the two pointers are pointing to a
                                     matching character, both of them are
                                     incremented */
     then
       begin
         k := k + 1;
         j := j + 1;
       end;
    else
     begin                        /* if j and k point to mismatching
                                     characters, then j is reset to point
                                     to the beginning of the pattern and i
                                     is reset to correspond to moving
                                     the pattern to the right one position
                                     for matching against text */
       k := k - j + 2;
       j := 1;
     end;
  until(j > M or k > N)
  if(j > M) then                  /* if the end of the pattern is reached
                                     (j > M), then a match has been found
                                     */
    brutesearch := k - M;
  else
    brutesearch := j;
 end Subprogram 1                 /* the first decomposed digraph ends here
                                     */
end.
```

The sixth test input has been taken from Cunningham's paper [Cunningham 82]. The digraph in test input 6 is presented in Figure 36.

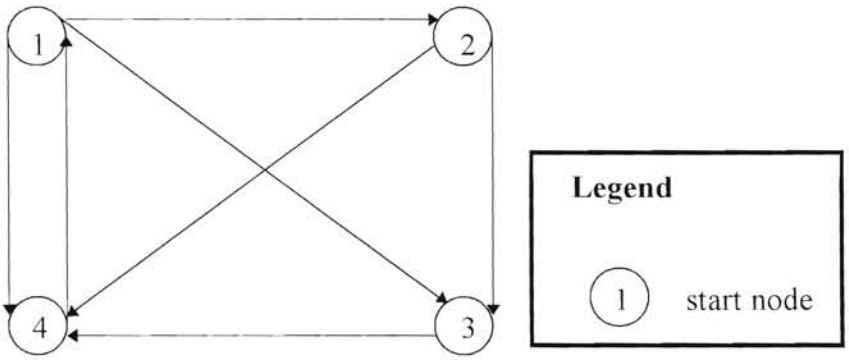
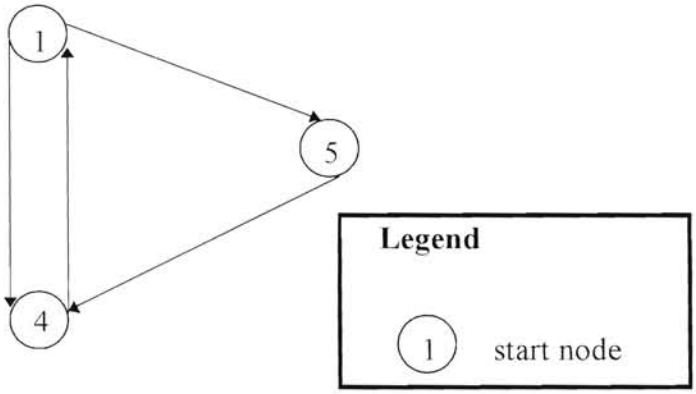


Figure 36. The digraph of test input 6
(Source: [Cunningham 82])

Digraph 1:



Digraph 2:

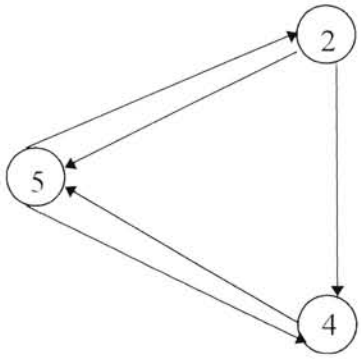


Figure 37. The two digraphs resulting from the decomposition of the digraph of Figure 36

The two digraphs that resulted from the decomposition of test input 6 digraph are presented in Figure 37, where node 5 is the marker.

VITA

SOLAYMAN MAHMOUD REFAE

Candidate for the Degree of

Master of Science

Thesis: PROGRAM FLOW GRAPH DECOMPOSITION

Major Field: Computer science

Biographical:

Personal Data: Born in Sour, Lebanon, January 1, 1971, son of Mahmoud and Fatina Refae.

Education: Graduated from Cadmous High School, Sour, Lebanon, in June 1988; received Bachelor of Science degree in Computer Science from Beirut University College, Beirut, Lebanon. Completed requirements for the Master of Science degree at The Computer Science Department at Oklahoma State University in July 1996.

Professional Experience: Computer Programmer, Ezsoft Company, Beirut, Lebanon, November 1992-December 1993.