SENSITIVITY OF GRAPH-THEORETIC METRICS TO EDGE DIRECTIONS FOR STRUCTURED AND UNSTRUCTURED PROGRAMS

By

CHINSYH HU

Bachelor of Engineering

Tamkang University

Taiwan, Republic of China

1984

Sumitted 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
December,1990

Thesis
1990
H8135a
cop. 2

SENSITIVITY OF GRAPH-THEORETIC METRICS TO EDGE DIRECTIONS FOR STRUCTURED AND UNSTRUCTURED PROGRAMS

Thesis Approved:

M. Samadzadeh-H.

Thesis Adviser

Blayne E. Mayfield

J P Chandler

Dean of the Graduate College

PREFACE

Program complexity can be measured based on graph-theoretic metrics such as cyclomatic number, track number, normal number, etc. This thesis explores the question of changing the directions of the edges on a flow graph and its impact on the graph-theoretic complexity metrics. Both structured and unstructured flow graphs are considered in this study.

I wish to express my deepest gratitude to Dr. Mansur H. Samadzadeh, my advisor, for his encouragement and confidence in this study. His continuous guidance based on his rich experience has made a major contribution to my career. His support and understanding are forever appreciated.

I also thank Drs. B. E. Mayfield, J. P. Chandler, and K. M. George for serving on my advisory committee, for their valuable comments, encouragement, and advisements.

Finally, I want to send my gratefulness and thanks to my parents for their support, encouragement, and patience.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A program should have several properties including being easy to read, correct, maintain, and test. Complexity of a program is strongly related to these properties and is a significant determinant of a software system's success or failure [McCabe89]. There are a relatively large number of metrics for measuring software complexity. By using these metrics, software engineering managers can estimate the cost and schedule of projects and the error-proneness of software systems, among other things.

There are many program complexity metrics that have been proposed and studied for conventional, procedural programming languages. The most basic and still widely used complexity metric is "program length". The notion of program length is not standardized. For example, Halstead's interpretation of the length of a program depends on the number of operators and operands used in the program [Halstead72 and 77].

Obviously, static or surface measures cannot directly and accurately reflect the running time. For example, a program may contain decisions and/or loops. The number of times that the cycles in a program flow graph (or the loops in a program) will be executed cannot in general be determined from the syntax of a program. The running time depends on the values of the variables and/or the predicates in the decisions and loops. A fifty-line program, containing twenty five consecutive "IF-THEN" structures, could have 33,554,432 (= 2^{25}) distinct control paths [McCabe76]. It is the input data that determines which paths will be executed and how many times.

The required running time and storage space, which are functions of the input data, serve as dynamic measures whereas software complexity measures are considered static measures.

Several graph-theoretic complexity measures have shown that software complexity to a large extent is independent of program size and dependent on the decision structure (of the flow graph) of a program [McCabe76 and Elshoff78]. However, it is worth noting that adding or subtracting functional statements (assignment statements), while affecting the bulk, does not change the decision structure complexity of a program.

For a flow graph, there are a number of metrics (including cyclomatic number, track number, normal number, number of intervals, and number of spanning trees) that can be considered complexity measurements of the flow graph. A major objective of this thesis is to calculate and compare the above-mentioned metrics for a number of different flow graphs, some structured and others unstructured, and to explore their interrelationships and interdependences.

There are a number of other metrics that are based on concepts from graph theory. Schneidewind and Hoffmann proposed "path count" and "reachability" [Schneidewind79] as metrics in a flow graph. The "path count" [Tai83] of a node is the number of distinct paths to reach that node in a flow graph (with the restriction that no loop iterates more than once). The "reachability" of a flow graph is the sum of the path counts over all the nodes in the graph. The average reachability of a flow graph is its reachability divided by the number of nodes in the graph [Tai83].

Oviedo studied the problem of program complexity in the context of high level languages [Oviedo80]. He developed some techniques to measure program attributes and formulated a model of program complexity. Oviedo attempted to formalize the notion of program complexity by defining it in terms of control flow and data flow characteristics of programs. The model which is based on the control flow and data flow is presumably more accurate and reliable than models of program complexity which are based only on the program control flow or on the number of program operators and operands [Oviedo80].

The data flow information in a program has been used for measuring program complexity from other perspectives and approaches also. One is to use the data flow within a module [Iyengar82 and Oviedo80]. Another approach is to use the data flow among modules to define complexity metrics [Henry81a and 81b]. Woodward proposed the "knot count" metric, which is based on the number of intersections of control flow paths in a program's text [Woodward79]. Harrison et al. carried out an analysis and comparison of data flow oriented and other program complexity metrics [Harrison82].

Data flow based complexity metrics can provide guidance for data flow based testing. A number of testing criteria based on data flow information in programs have been proposed [Laski82, Laski83, Rapps82, and Tai80].

This thesis is organized as follows. The next chapter is a review of some concepts from graph theory that are used in the subsequent chapters of this thesis. Each complexity metric has its own characteristics and represents the complexity of the control or data flow graph of a program from a different viewpoint. Some of the more well known metrics are discussed in detail in Chapter III. Chapter IV explores the question of sensitivity of graph-theoretic metrics to edge directions and finally Chapter V consists of the results, summary, and some areas of possible future work.

CHAPTER II

LITERATURE REVIEW

This chapter consists of a brief review of the graph-theoretic background information that is used in the rest of this thesis. Definitions for different types of graphs and the related concepts are presented followed by a definition of structuredness and a discussion of dominator trees.

2.1 Graphs

A graph G consists of two sets V and E. V is a set of vertices (or nodes) and E is a set of edges which are pairs of vertices. (The graph theory notation and definitions used in this thesis are basically from [Horowitz82].) V(G) and E(G) represent the sets of vertices and edges of a graph G, respectively. We also can write G = (V, E) to represent a graph. If the edges are unordered pairs of nodes in a graph G, the graph is called an undirected graph (or simply a graph). Thus, the pairs (v_1,v_2) and (v_2,v_1) represent the same edge $\{v_1,v_2\}$. In a directed graph, an edge is represented by an ordered pair (v_1,v_2), where v_1 is the tail and v_2 is the head of the edge. Therefore (v_1,v_2) and (v_2,v_1) represent two different edges. Figure 1 illustrates a directed graph G_1 = ({1, 2, 3, 4, 5, 6}, {(1, 2), (2, 3), (3, 4), (3, 6), (4, 4), (4, 5), (5, 2)}).

In a directed graph G, the in-degree of a vertex v is defined to be the number of edges for which v is the head. The out-degree of a vertex v is defined to be the number of edges for which v is the tail. Vertex 3 of G_1 in Figure 1 has in-degree 1 and out-degree 2. If a vertex has in-degree 0, it is called a source (or a root) of the graph. Conversely, if a vertex has out-degree 0, it is called a sink (or a leaf) of the graph. Vertex 1 is the source and vertex 6 is the sink of G_1 in Figure 1. A node whose out-degree is greater than or equal to 2, is called a decision node. Similarly, a node whose in-degree is greater than or equal to 2 is called a collecting node. In Figure 1, vertex 3 is a decision node and vertex 2 is a collecting node.

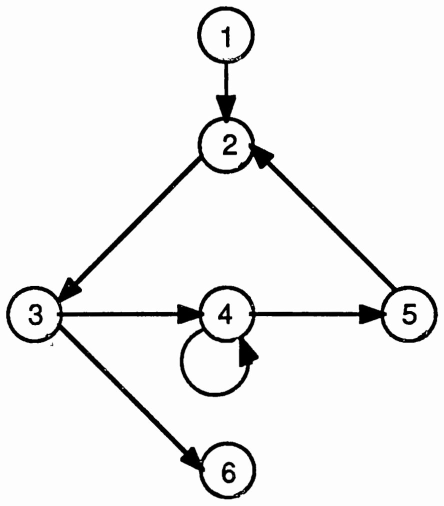


Figure 1. A directed graph G_1.

If there is an edge with the same vertex as its head and tail, it is called a sling. For example, (v_4,v_4) is a sling in G_1. A path from vertex v_p to vertex v_q in a graph G is a sequence of vertices v_p, v_1, v_2, ..., v_n, v_q such that (v_p,v_1), (v_1,v_2),...,(v_n,v_q) are edges in E(G). The length of a path is the number of edges on it. A simple path is a path in which all vertices except possibly the first and last are distinct.

A cycle is a simple path in which the first and last vertices are the same. The path (2, 3, 4, 5, 2) is a cycle of G_1 in Figure 1. In this thesis, we will be dealing with the flow graphs which are directed and have a unique source and a unique sink.

2.2 Adjacency Matrices

There are several representations for graphs, such as adjacency matrices, adjacency lists, and adjacency multilists [Horowitz82]. The choice of a representation will depend upon the application and the function that one expects to perform on the graph. In this thesis, the adjacency matrix is used to represent a graph.

Let G = (V, E) be a graph with n vertices. The adjacency matrix of G is a 2-dimensional n by n array, say A, with the property that A(i, j) = 1 iff the edge (v_i,v_j) exists in E(G) and 0 otherwise. The adjacency matrix for a directed graph need not be symmetric. Using an adjacency matrix, space needed to represent a graph with n vertices is n^2 bits. When a graph is sparse, most of the elements of the matrix are zero's. For a directed graph, a row sum is the out-degree of the vertex corresponding to that row. Conversely, the column sum is the in-degree of the vertex corresponding to that column.

The adjacency matrix of the graph G_1 of Figure 1 is depicted in Figure 2. From the adjacency matrix, we can answer the following questions, among other things. Which vertex is the source (column elements are 0's) or sink (row elements are 0's)? How many edges are there in G? Is there any sling or cycle in G?

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

Figure 2. The adjacency matrix of graph G_1 of Figure 1.

2.3 Structuredness

Directed graphs can be divided into structured and unstructured graphs based on the programs that they represent. Directed graphs are used to represent programs in the form of control flow or data flow graphs. If a directed graph contains the following four anomalous control structures (Figure 3), it is called an unstructured graph [McCabe76].

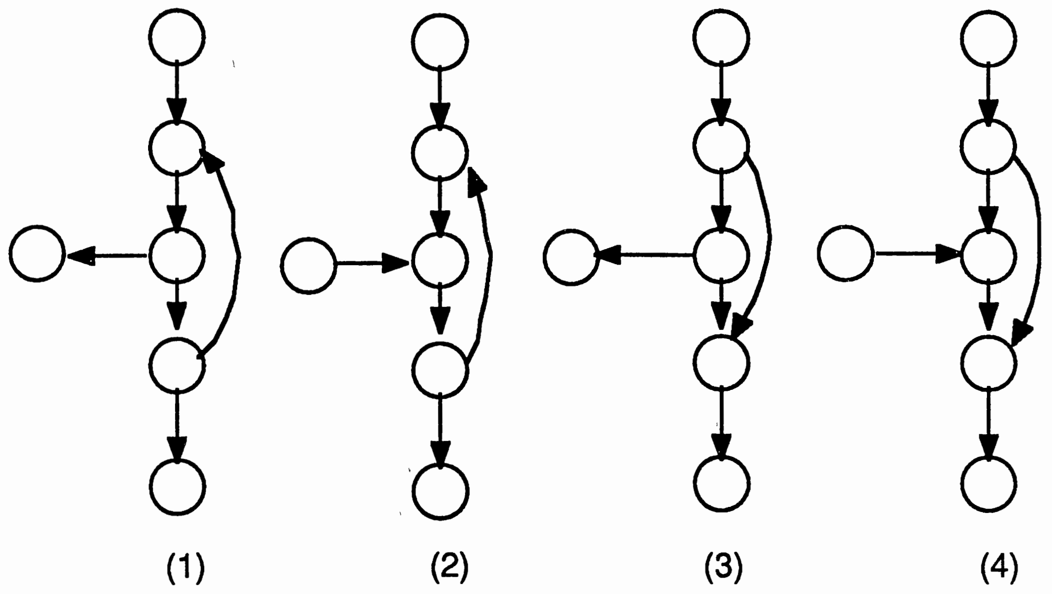
(1) branching out of a loop

(2) branching into a loop

(3) branching out of a decision

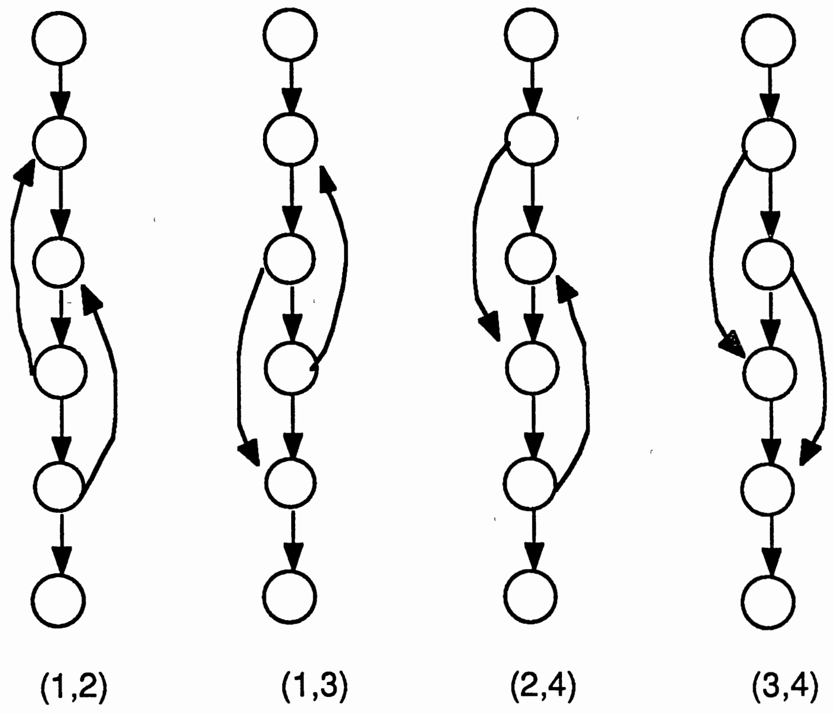
(4) branching into a decision

Actually, an unstructured directed graph cannot just exist with only one form of unstructuredness out of the four listed above. It exists with at least one pair of the four situations [McCabe76]. Some unstructured flow graphs are shown in Figure 4. Conversely, a directed graph without branching out of/into a loop/decision, is called a structured graph.



(1) (2) (3) (4)

Figure 3. Four basic forms of unstructuredness.



(1,2) (1,3) (2,4) (3,4)

Fegure 4. Some unstructured flow graphs.

For example, let's consider two short programs written in C [Lin89]. The first program is to calculate the value of x^y. The flow graph of the program is structured (see Figure 5). The second program is to search a value F from an ordered array A(N) with N elements, by binary search. The flow graph of this second program is unstructured (see Figure 6).

```
    main()
    {
1   int  x,y,z,power;
    scanf("%d,%d",&x,&y);
    if(y<0)
2   power = -y;
    else
3   power = y;
4   z=1;
5   while(power != 0)
     {
6    z = z*x;
     power = power - 1;
     }
7   if(y<0)
8   z = 1/z;
9   printf("the answer is %d",z);
    }
```

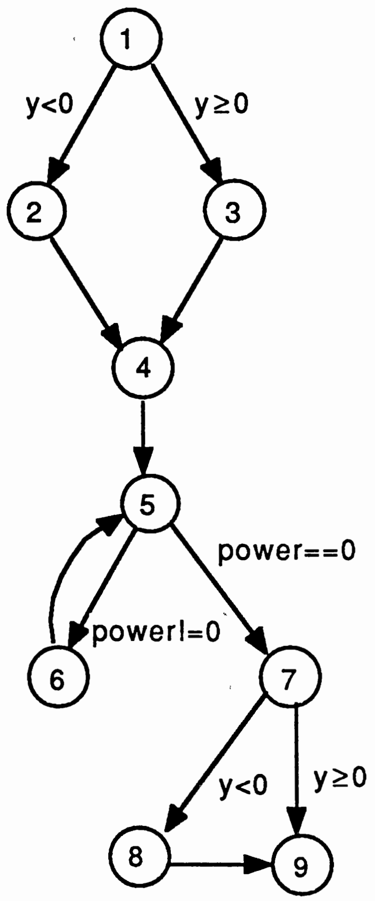


Figure 5. A structured program and its flow graph.

```
                  binary_search(A[N],F)
                  {
1                 int  low,high,mid,mpos;
                  mpos=0;
                  low=high=N;
2     binary:     mid=(low+high)/2;
3                 if(high<low) goto end;
4                 if(F==A[mid]) goto found;
5                 if(F > A[mid]) goto upper;
6                 high=mid-1;
                  goto binary;
7     upper:      low=mid+1;
                  goto binary;
8     found:      mpos=mid;
9     end:        return(mpos);
                  }
```

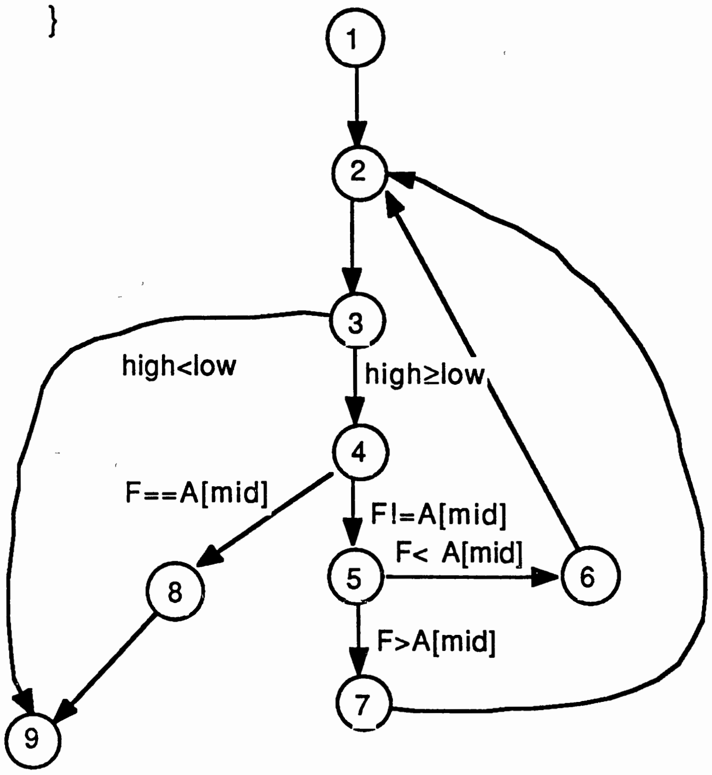


Figure 6. An unstructured program and its flow graph.

2.4 Dominator Trees

In a directed graph with a unique root node, if every path from the root to a node n has to go through a node d, node d is said to dominate node n, written d dom n [Aho86]. Under this definition, every node dominates itself and the entry node of a loop dominates all the nodes in the loop. A useful way of presenting the dominating relation is to construct a Dominator Tree. Consider the graph G_2 in Figure 7 with root node 1.

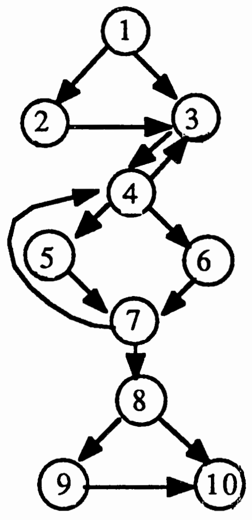


Figure 7. A directed graph G_2.

The root node dominates all nodes in G_2. Node 2 only dominates itself since paths can begin at 1 and go through 3 to the other nodes hence bypassing 2. Node 3 dominates all nodes except 1 and 2. Node 4 dominates all but 1, 2, and 3. Nodes 5 and 6 only dominate themselves, since paths can go through either one. Node 7 dominates 7, 8, 9, and 10. Node 8 dominates 8, 9, and 10. Node 9 and

10 dominate themselves only. So the Dominator Tree for graph G_2 can be depicted as follows [Aho86].

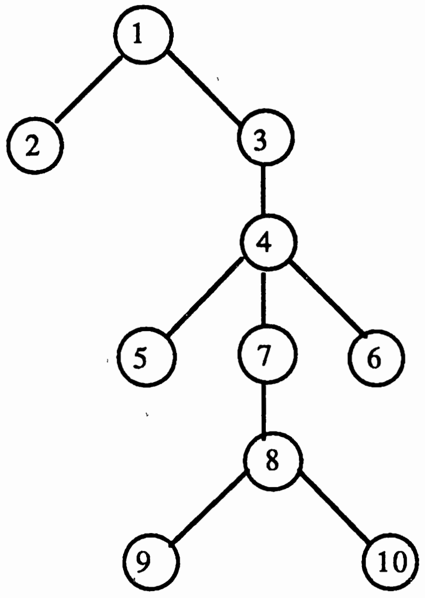


Figure 8. The dominator tree of graph G_2.

One important application of the dominator information is to help determine the loops of a flow graph. There are two essential properties of loops.

1. A loop has at least one entry node, called the header of the loop, which dominates all nodes in the loop.
2. There must be at least one path from the header back to the header again.

A good way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails; such edges are called back edges. Given a back edge (n,d), Aho et al. defined the natural loop of the edge to be d plus the set of nodes that can reach n without going through d [Aho86]. Node d is called the header of the loop. For example, in graph G_2 in Figure 7, 4 dom 7 and there is an edge from 7 to 4; thus there is at least one loop between 4 and 7. Similarly, 3 dom 4 and there is an edge from 4 to 3; hence there is at least one loop between 3 and 4.

CHAPTER III

GRAPH-THEORETIC METRICS

This chapter contains a brief explanation of several graph-theoretic metrics. The metrics discussed in this chapter are studied in Chapter IV regarding their sensitivity to edge directions.

3.1 Cyclomatic Number

The cyclomatic number is an important characteristic of a flow graph [McCabe76]. It can be used as a basic complexity measurement. 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 9 illustrates the cyclomatic number of some well known control structrues [McCabe76].

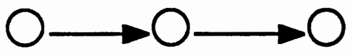
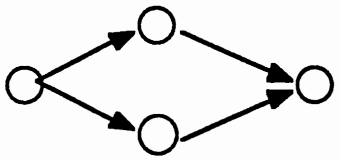
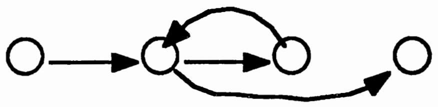
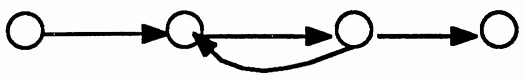
	Control Structure	Cyclomatic Number
SEQUENCE		V=2-3+2=1
IF-ELSE-THEN		V=4-4+2=2
WHILE		V=4-4+2=2
UNTIL		V=4-4+2=2

Figure 9. Some control structures and their cyclomatic numbers.

For example, the cyclomatic number of the flow graph G_3 in Figure 10 is V(G)= 9-7+2=4.

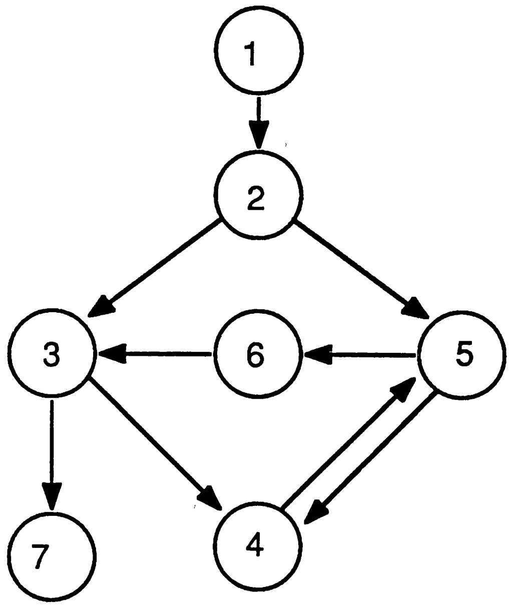


Figure 10. A directed graph G_3.

3.2 Track Number

A directed graph G, without slings or multiple edges, can be divided into a number of paths. A path $p_a = (v_0, v_1, \ldots, v_n)$ is called a track [Culik81] if it is any one of the three different types of paths (simple open, simple closed, or snare), where $v_i \neq v_j$ for all i, j = 0, 1, ..., n-1 and n ≥1 (Figure 11). A track decomposition is to divide a graph G into tracks which are not mutually crossing one another (defined below). Of course, there are many ways to divide a graph G and get different tracks on it. The way to get the minimal number of tracks is called the maximal track decomposition. Culik has shown that the number of tracks under maximal track decomposition is equal to $|E_G| - |V_G| + |ISC_G| + |Outp_G|$, where $|E_G|$ and $|V_G|$ are the numbers of edges and vertices of G respectively, ISC_G is the set of all input strong components of G (an input strong component is a strongly connected component such that no edge starts from 'outside' and terminates 'inside' it), and $Outp_G$ is the set of all sinks of G. In a special case where a connected graph G has a unique source and a unique sink, the track number is the same as the cyclomatic number $(|E_G| - |V_G| + 2)$ [Culik81 and McCabe76].

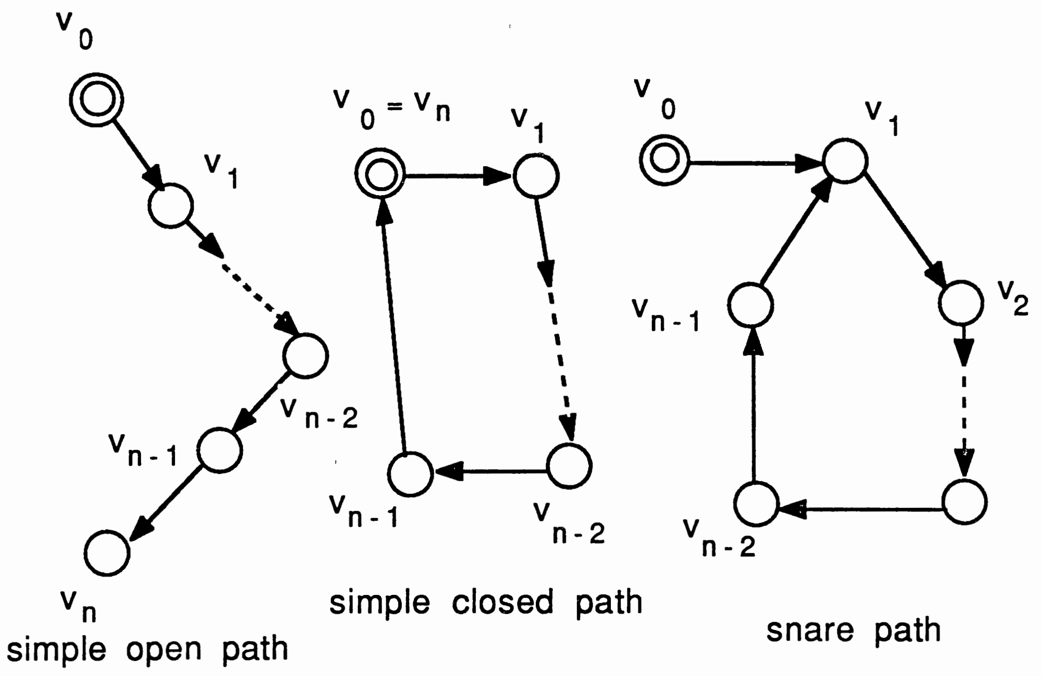


Figure 11. Three types of paths.

A set of tracks $p_1, p_2, \ldots, p_k$, $k \geq 2$, is called mutually crossing if either any two of the tracks are crossing or the starting vertex of one track p_2 belongs to the other track p_1 and the ending vertex of track p_1 belongs to the other track p_2 (Figure 12).

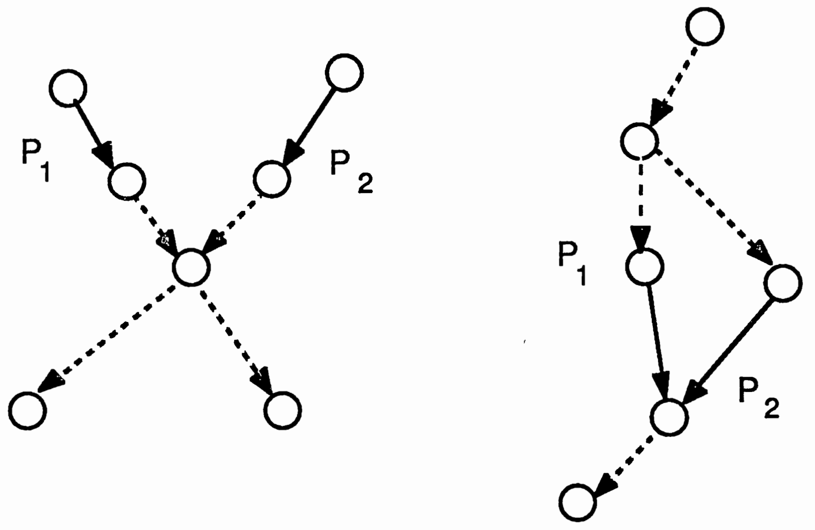


Figure 12. Two types of mutually crossing tracks.

For example, the following four sets of tracks of the directed graph G_3 in Figure 10 are different maximal track decompositions. Each set has the same number of tracks, which is called the track number of G_3.

a. (1, 2, 3, 7), (2, 5, 6, 3), (3, 4, 5), (5, 4)

b. (1, 2, 5, 4, 5), (5, 6, 3, 4), (2, 3), (3, 7)

c. (1, 2, 5, 6, 3, 7), (2, 3), (3, 4, 5), (5, 4)

d. (1, 2, 5, 4, 5), (5, 6, 3, 7), (2, 3), (3, 4)

3.3 Normal Number

A normal form of a directed graph G=(V,E) is a directed graph $G^*=(V^*,E^*)$ that is functionally equivalent with G [Culik80a]. A normal form has the form of a tree with some leaves which are bent back to some of the earlier vertices belonging to the paths on which they sit [Culik81]. (The concept of being bent back or having a back edge will be made clear in the rest of this section.) If there are no rooted parallel simple paths in G, we call G is an almost-tree [Culik77 and 80b]. The normal form of G doesn't have to be a tree, it is an almost-tree. There exists exactly one simple path from root to any vertex and there does not exist a parallel simple path starting from root on the normal form. A normal number is defined by the number of simple paths on the normal form G^* [Culik81]. If the normal form G^* is not unique, we take the minimal number of simple paths as the normal number of G [Culik81].

A directed graph G is a rooted tree if there exists one source vertex r and there is exactly one path from r to each other vertex $v \neq r$, called the rooted path of v. For each vertex v of a rooted graph G, the rooted subtree T_V is defined as follow: T_V contains exactly those vertices and edges of G which belong to paths in G starting from vertex v and terminating at the leaves which are the last vertices of the tracks on a maximal track decomposition. In a directed tree, if the leaves can be bent back to earlier vertices on the paths from the root, they are called good leaves. Conversely, the leaves that can not be bent back to earlier vertices on the paths from the root are called bad leaves [Culik77].

In order to construct a normal form G^* from a directed graph G, there are three steps. The first step is to do the maximal track decomposition. For example, we can divide the directed graph G_3 of Figure 10 into four tracks (1, 2, 3, 7), (2, 5, 6, 3), (3, 4, 5), and (5, 4). The last vertices of the tracks will be the leaves of G^*. The set of these vertices is L={7, 3, 5, 4}. The second step is to construct the subtrees T_L which start from a root in L and end at the other vertices in L. There is no subtree for a sink node that happens to be in L. The subtrees of T_L for graph G_3 of Figure 10, are as follow

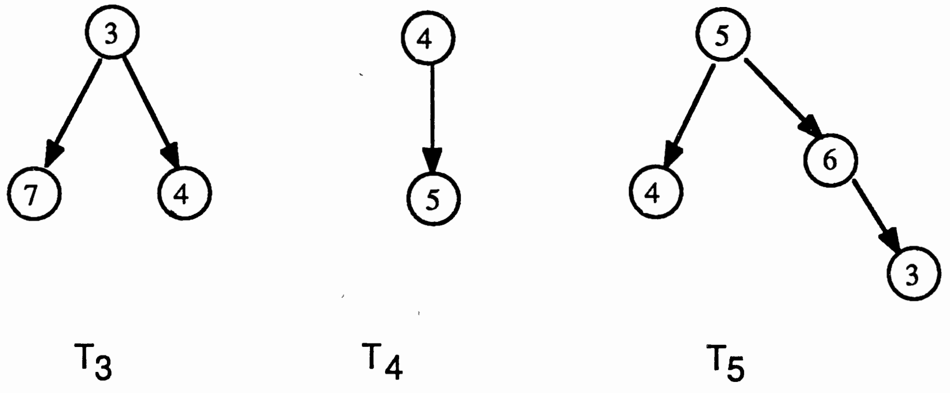


Figure 13. The subtrees of G_3 with roots in L={7, 3, 5, 4}.

At this point, all leaves of the subtrees are bad leaves. If we furl T_3 by substituting T_4 and then T_5, T_3' can be obtained. The leaves 3 and 4 in T_3' are good leaves but 7 is still a bad leaf.

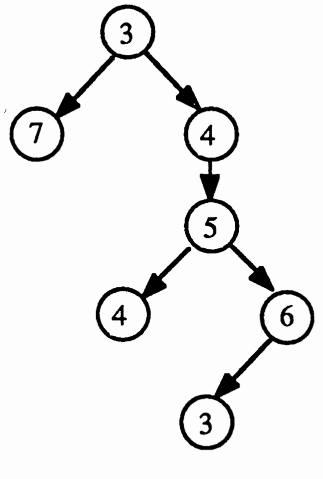


Figure 14. The subtree T_3'.

The third step is to furl or expand the graph at the root, and use the subtrees to construct the normal form G^*. Eventually, all the leaves of a normal form G^* should be good leaves or the sink of

G. If there are bad leaves, we have to furl the bad leaves by the subtrees T_L. The furling process of G_3 is depicted in Figure15. The number of good leaves is the normal number [Culik77]. The path from the root to a good leaf is a simple closed loop [Culik81].

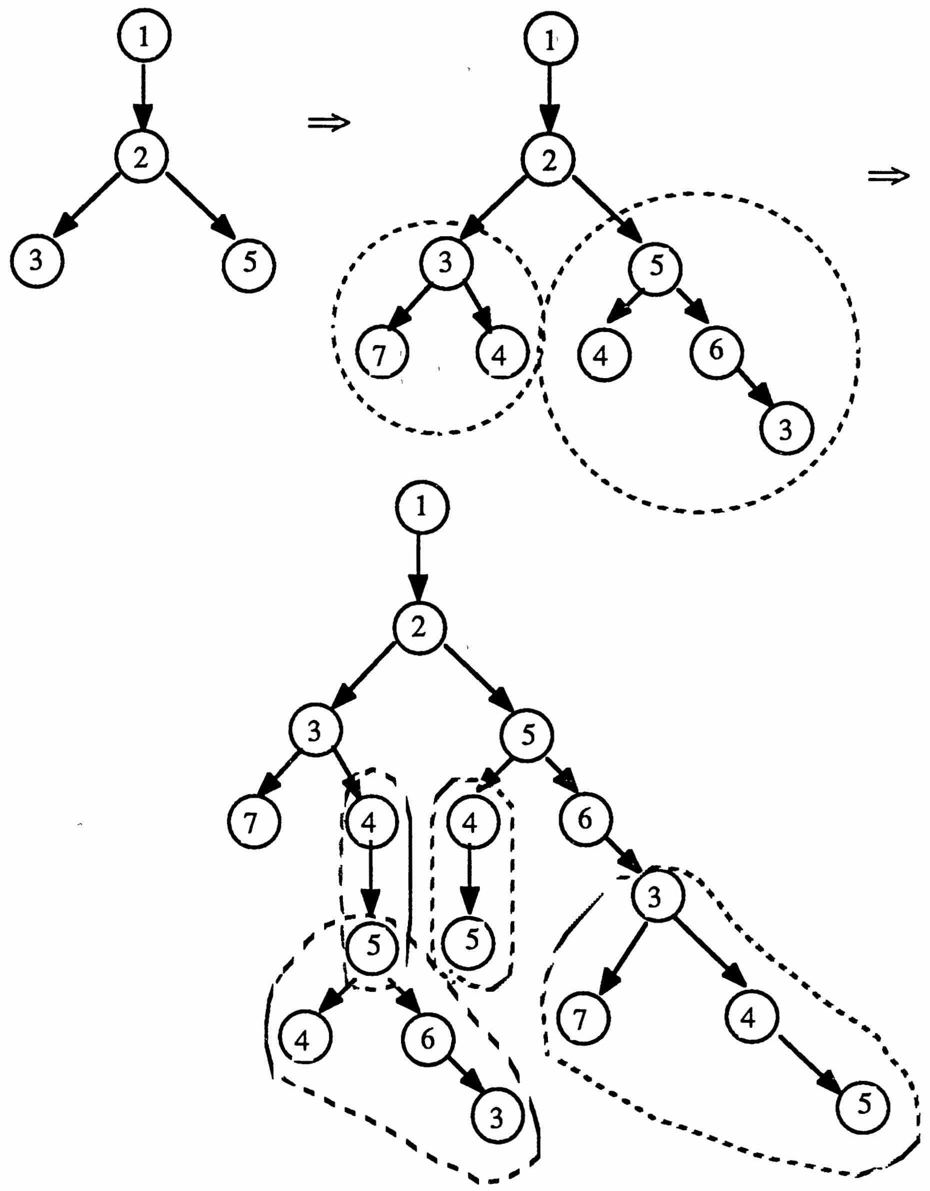


Figure 15. The furling process of the normal form G_3^* of the graph G_3 in Figure 10.

3.3.1 Example

Consider the directed graph G_4 in Figure 16. What is the normal form of it?

The first step is to do the maximal track decomposition. We can get the tracks (1, 2, 3, 4, 6, 7, 8, 4), (3, 5, 7), (6, 9, 2), (8, 10) and L={4, 7, 2, 10} and the sink is vertex 10.

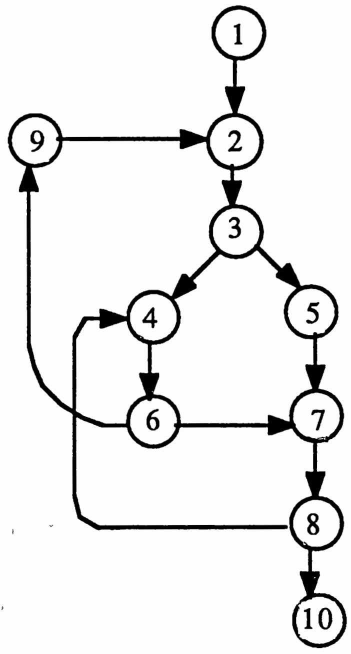


Figure 16. A directed graph G_4.

The second step is to construct the subtrees of T_L.

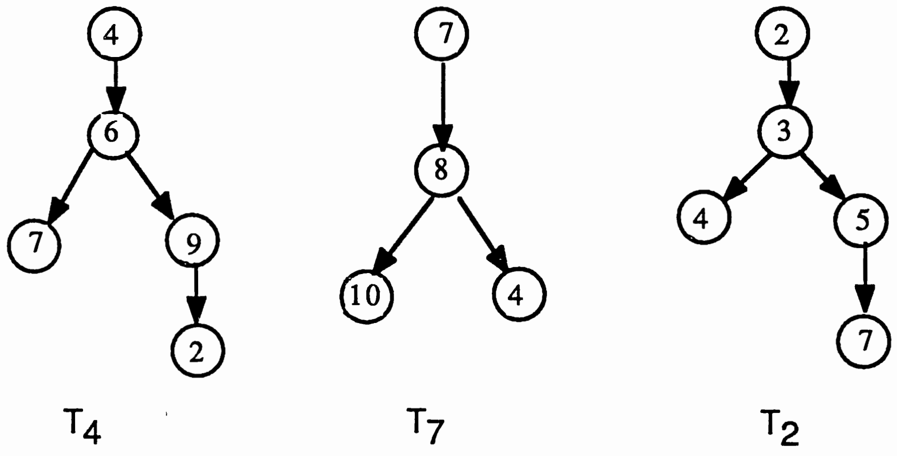


Figure 17. The subtrees of G_4 with roots in L={4, 7, 2, 10}.

The third step is to construct a normal form of G_4^* and furl the graph starting from the root. Eventually, we can get four good leaves, and four simple closed loops, (1, 2, 3, 4, 6, 7, 8, 4), (1, 2, 3, 4, 6, 9, 2), (1, 2, 3, 5, 7, 8, 4, 6, 7), and (1, 2, 3, 5, 7, 8, 4, 6, 9, 2), of the normal form G_4^*. So the normal number of the directed graph G_4 is four.

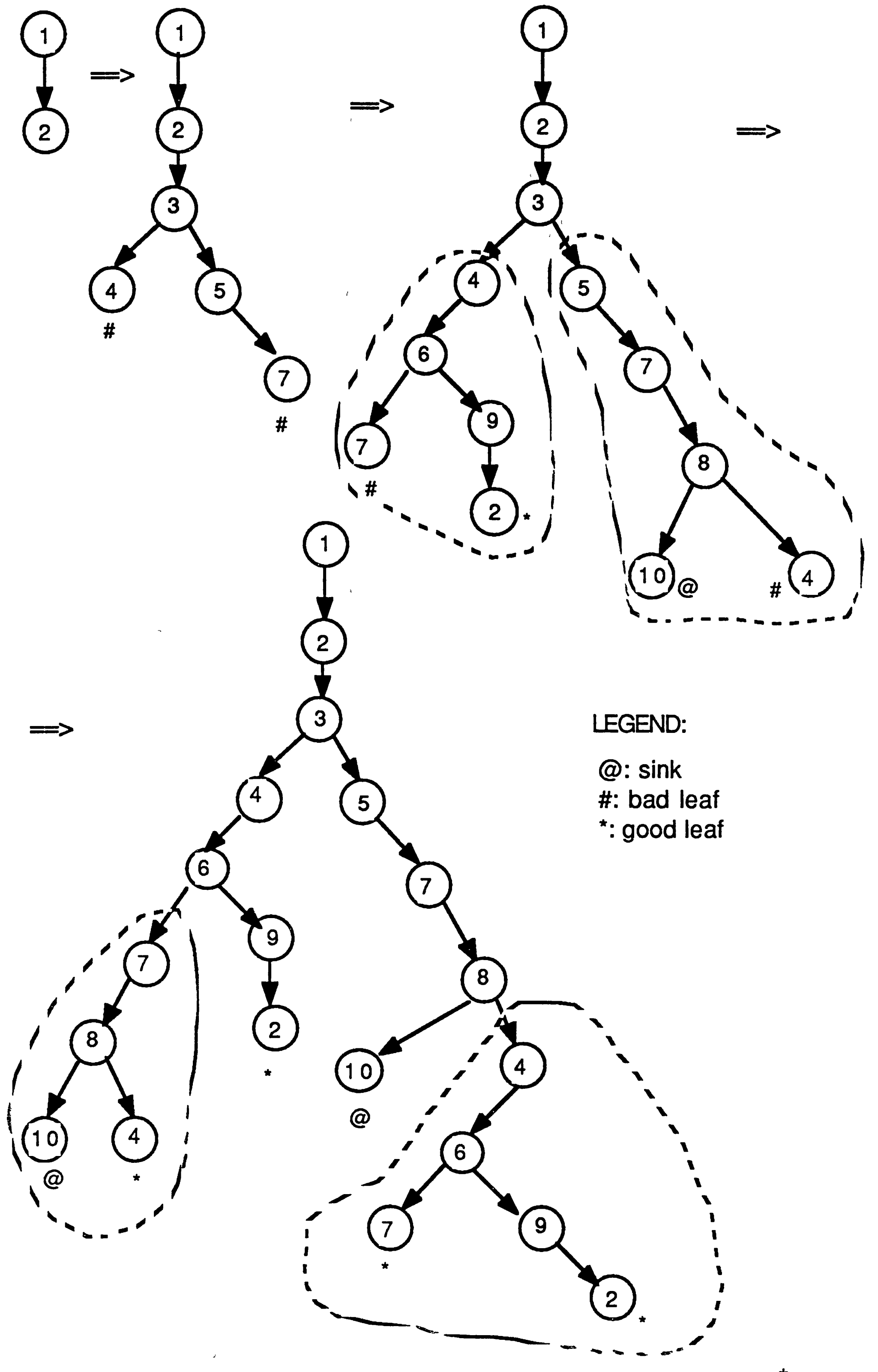


Figure 18. The furling process of the normal form G_4^*.

3.4 Number of Intervals

There are a variety of flow-graph concepts, such as "interval analysis" that are basically related to structured flow graphs. Aho defined a natural loop as follow [Aho86]. An "interval" is defined as a natural loop plus an acyclic structure that dangles from the nodes of the loop in a flow graph [Aho86]. An important property of intervals is that every interval has a header node that dominates all the nodes in the interval. Formally, for a given flow graph G with source node n_0 and a node n of G, the interval with header n, denoted I(n), is defined recursively as follows [Aho86].

1. n is in I(n).
2. If m is not the source node and all the predecessors of m are in I(n), then m is also in I(n).
3. Nothing else is in I(n).

For example, consider the flow graph G_2 in Figure 7. Let us find the interval partition of the graph G_2. We start from the source node 1 and put it into I(1) as the header. Add node 2 to I(1) because 2's predecessor is node 1 only. We cannot add node 3 to I(1) because its predecessors are not just node 1 and 2 but also node 4. Thus, I(1) = {1,2}. We may now compute I(3). Add node 3 as the header of I(3). But I(3) only contains node 3, since node 4's predecessor is not just node 3 but also node 7. Thus, I(3) = {3}. Now let node 4 be the header of I(4), then we can add 5, 6, 7, 8, 9 and 10 to I(4), because all the necessary predecessors are in I(4) = {4, 5, 6, 7, 8, 9, 10}. The graph G_2 in Figure 7 is depicted by intervals as follows.

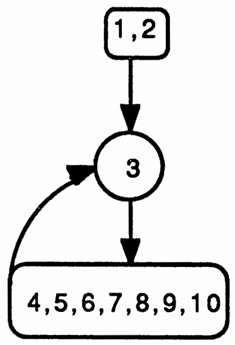


Figure 19. The "intervals" of graph G_2.

We can take the intervals as new nodes and do an interval partition again on the interval graph of G_2. By doing that successively for graph G_2, we will get the following set of graphs.

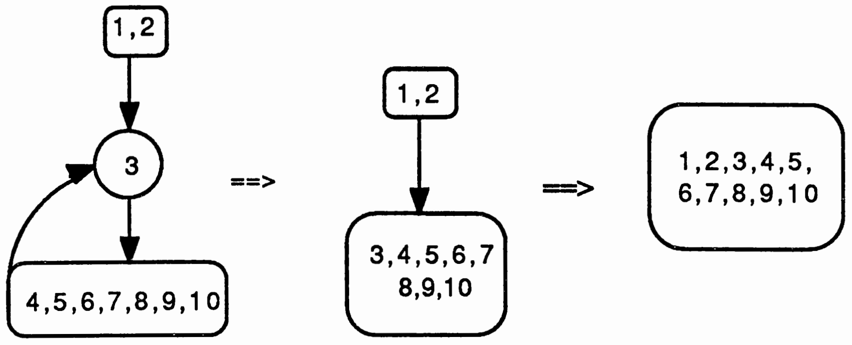


Figure20. Successive intervals of graph G_2.

3.5 Number of Spanning Trees

A subgraph H of a rooted directed graph G is called a directed spanning tree of G if H is a directed tree which includes all the vertices of G. If r is the root of G, then it is also the root of H. To construct a directed spanning tree, simply start from the root, edge by edge, and add each time an edge of G from a vertex already reachable from r in H to one which is not reachable yet [Even79].

Even proposed a method to compute the number of directed spanning trees of a given directed graph with a specified root [Even79].

The in-degree matrix D of a directed graph G=(V,E), where V={1, 2, 3,..., n}, is defined as follows [Even79].

$$D(i,j)=\begin{cases} d_{in}(i) & \text{if } i=j, \text{ where } d_{in}(i) \text{ is the in-degree of } v_i, \\ -k & \text{if } i \neq j, \text{ where } k \text{ is the number of edges in G from } i \text{ to } j. \end{cases}$$

The number of directed spanning trees with root r of a directed graph with no self-loops is given by the minor of its in-degree matrix which results from the erasure of the r-th row and column [Even79].

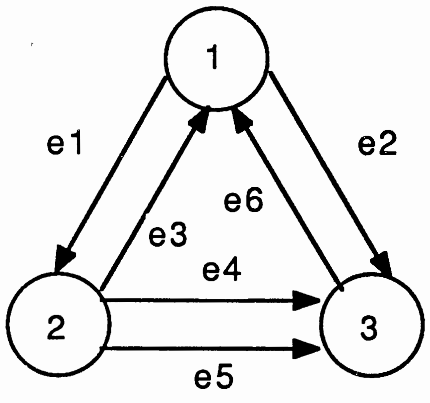


Figure 21. A directed graph G_5.

For example, consider the graph G_5 in Figure 21, the in-degree matrix D of G_5 is as follows.

$$D = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 1 & -2 \\ -1 & 0 & 3 \end{bmatrix}$$

Assume that we want to compute the number of directed spanning trees with root 2. We erase the second row and column. The resulting determinant is

$$\begin{vmatrix} 2 & -1 \\ -1 & 3 \end{vmatrix} = 5$$

Similarly, assume that we want to compute the number of directed spanning trees with root 1 or 3. The resulting determinants are as follows.

$$\begin{vmatrix} 1 & -2 \\ 0 & 3 \end{vmatrix} = 3 \qquad \begin{vmatrix} 2 & -1 \\ -1 & 1 \end{vmatrix} = 1$$

The five directed spanning trees with root 2 of the directed graph G_5 are as follows.

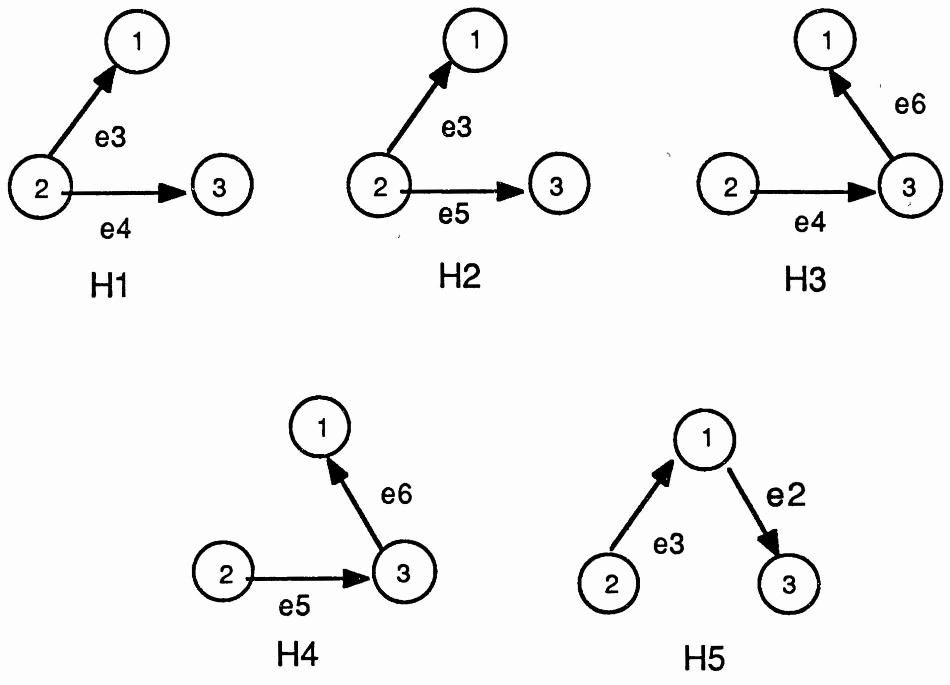


Figure 22. The five directed spanning trees of G_5 with root 2.

As a second example, let's calculate the number of spanning trees of G_3 with root 1 in Figure 10?

The in-degree matrix of G_3 is

$$\begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & -1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The determinant after erasing the first row and column of the in-degree matrix is 5. So the number of spanning trees with root 1 of G_3 is 5. The five spanning trees are depicted in Figure 23.

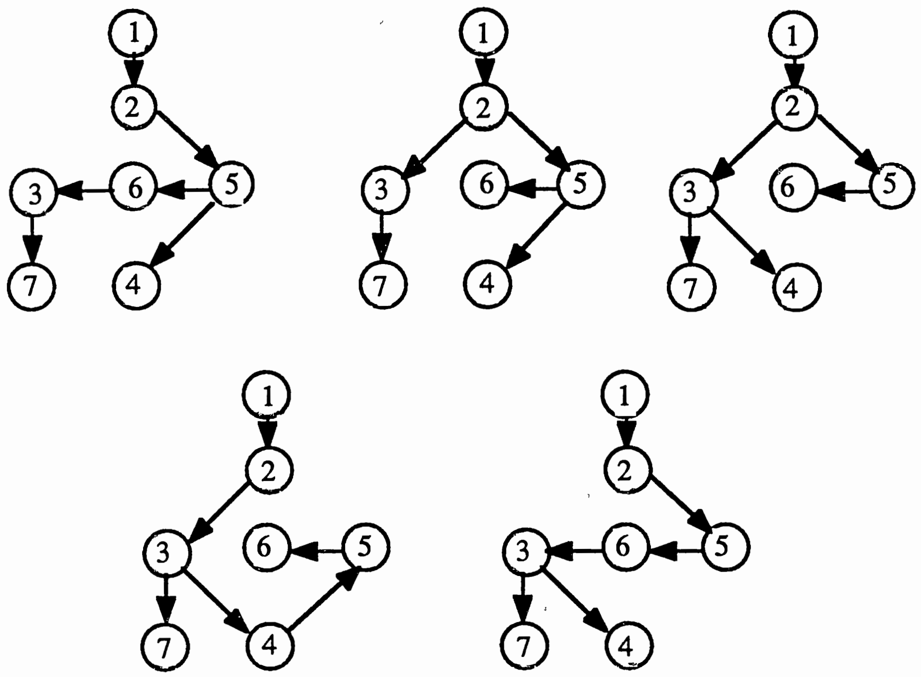


Figure 23. The five directed spanning trees of G_3 with root 1.

3.6 Summary

There are a large number of metrics that attempt to measure the (conceptual) complexity of programs. A number of these metrics are directly or indirectly based on graph theory. A widely used measure of program complexity is the cyclomatic number or the number of linearly independent paths of a graph representing the control flow of a program. If a directed graph has a unique source and a unique sink, the cyclomatic number is the same as the track number which is the number of tracks on any maximal track decomposition of a graph. The normal number of a flow graph is another graph-theoretic measure which is the (minimal) number of loops among all possible normal forms in the graph. An interval of a directed graph is defined as a natural loop plus an acyclic structure that dangles from the nodes of the loop in the graph. The number of intervals that is obtained by "interval partition algorithm" is called the number of intervals of the flow graph. For a directed graph G, that is without slings (self loops) but has a root, we can construct a directed spanning tree H of G with the same root. The number of all possible H's is called the number of spanning trees of G.

CHAPTER IV

SENSITIVITY OF METRICS TO EDGE DIRECTIONS

This chapter explores the sensitivity of the graph-theoretic metrics discussed in Chapter III to edge directions for structured and unstructured flow graphs. All the flow graphs that are discussed in this thesis have a unique source and a unique sink (see Section 2.1 for the definition of source and sink). For a directed graph G, if we change the directions of all of the edges, the value of some of the graph-theoretic metrics will change; but some of the other metrics will not change regardless of whether the graph is a structured or an unstructured flow graph. For example, the cyclomatic number is defined based on the number of edges and vertices. So it will not change as a result of changing the directions of the edges.

4.1 Structured Flow Graphs

A directed graph, without branching out of/into a loop/decision, is called a structured flow graph (see Section 2.3). A structured flow graph (without slings and multiple edges) may be composed from the six basic flow graphs (see Figure 25) which have their own basic dominator structures. With those structures, it is easy to obtain the dominator tree of a structured flow graph.

If we change all the edge directions of a structured flow graph

G, we can get a new graph G'. Some of the metrics of G' don't need to be recomputed. They can be derived from the original metrics. For example, the cyclomatic and track numbers are the same for G and G'. The normal number of G can be obtained from the dominator tree of G with back edges. Calculation of the normal number of G using this method is discussed later in this section.

If the dominator tree of a structured flow graph has back edge(s), there should be some loop(s) in it. For example, the dominator tree of graph G_2 in Figure 7 with the back edges appears in Figure 24.

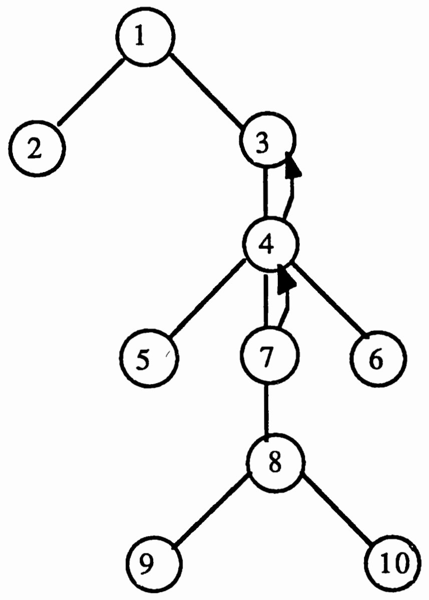


Figure 24. The dominator tree of G_2 with back edges.

For a structured flow graph, we can define a "decision factor" for a node r of the dominator tree as follows: 1 for SEQUENCE and SINGLE-LOOP, 2 for DECISION and DECISION-LOOP, and n for CASE and CASE-LOOP (see Figure 25). The decision factor of node r is determined based on the nodes which are one lower level than node r.

For example in Figure 24, the decision factor is 2 for nodes 1(DECISION), 4(CASE-LOOP), and 8(DECISION); and the decision factor is 1 for nodes 3(SINGLE-LOOP) and 7(SEQUENCE).

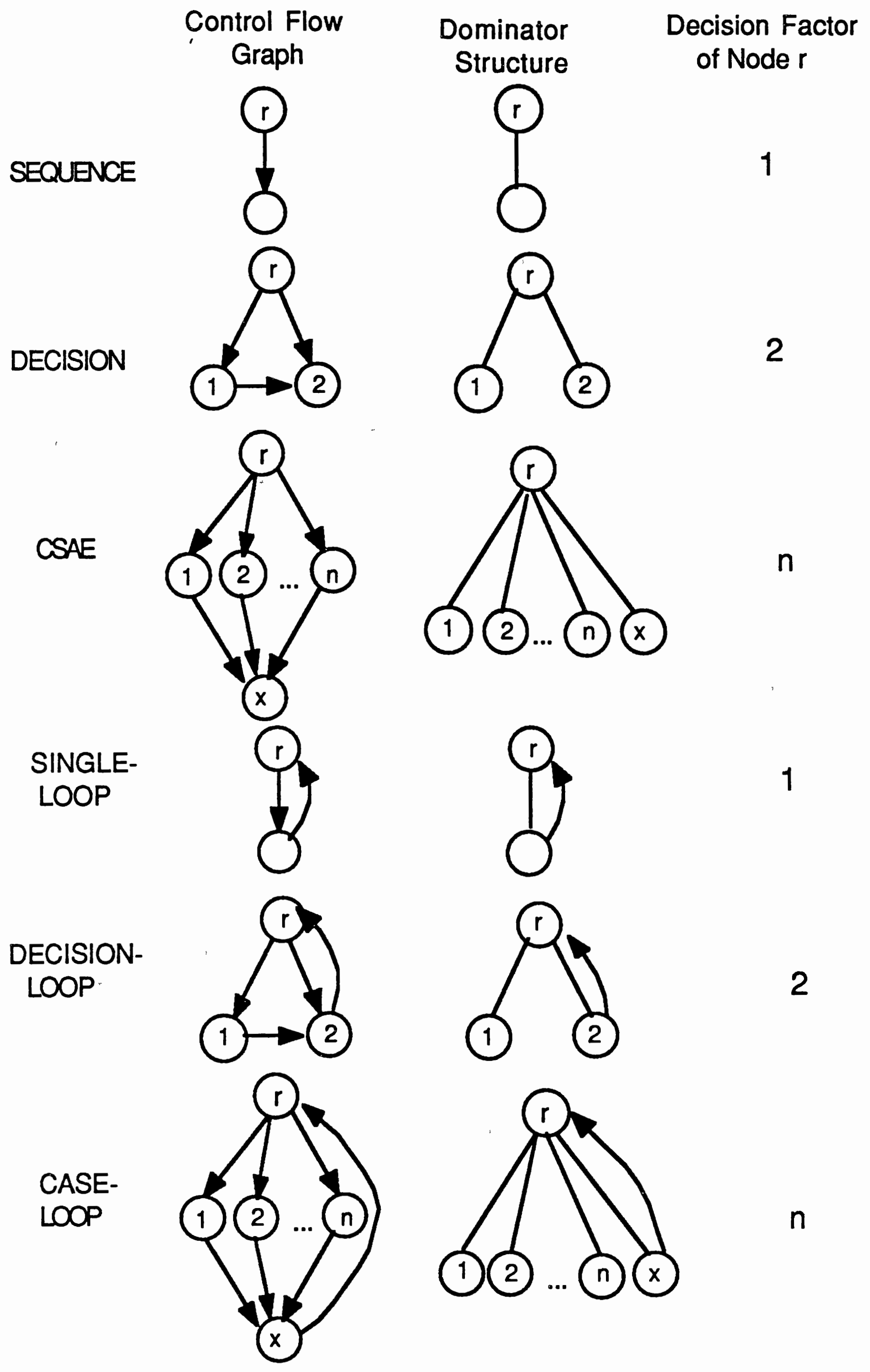


Figure 25. The six basic flow graphs, their dominator structures, and the decision factors of some of the vertices.

The number of loops that exist in a flow graph is called the normal number. One way to obtain this number was mentioned in Section 3.3. For a structured flow graph, we also can get this number by another method which uses dominator trees with back edges and decision factors. With this method, we don't need to recompute the normal number if we change all the edge directions of G

ALGORITHM 4.1

To get the normal number of a flow graph by using its dominator tree with back edges and decision factors.

INPUT: The dominator tree with back edges and decision factors of a flow graph.

OUTPUT: The normal number of the flow graph.

METHOD:

Step 1 Find a path $(V_1,V_2,...,V_n)$ from the source node V_1 to the head node, V_{n-1}, of the back edge (V_n,V_{n-1}) from the dominator tree with back edges of a structured flow graph.

Step 2 The product of the decision factors of all of the nodes on the path $(V_1,V_2,..,V_{n-1})$ is the total number of loops from the source to the head node, V_{n-1}, of the back edge.

Step 3 If there exists another back edge, go to Step 1.

Step 4 The sum of all the numbers calculated by Step 2 is the total number of the simple closed loops of the structured flow graph (i.e., the normal number).

For example, in the dominator tree of G_2 with back edges in Figure 24, there are two back edges (4,3) and (7,4). By Step 1, the path for (4,3) is (1,3,4). By Step 2, the product of the decision factors of the nodes on the path is 2*1=2. Similarly, the path for (7,4) is (1,3,4,7) and the product of the decision factors of the nodes on the path is 2*1*2=4. By Step 4, the sum is 2+4=6. So the normal number is 6.

Since the number of back edges of a dominator tree is limited ($O(n^2)$ in the worst case), this method will always halt. The number computed by this method is the number of simple closed loops starting from the root of the flow graph based on one of the definitions of the normal number. The computational time and space complexity of Algorithm 4.1 depends on the representation of the dominator tree, the number of nodes, and the number of back edges.

If we change all the edge directions of a structured flow graph, a new graph G' results. The normal number of G' can be derived either by the method outlined in Section 3.3 or the method discussed in this section which is based on constructing the dominator tree with back edges. We can also obtain the normal number from the original dominator tree with back edges of G. The method is outlined below and its properties are similar to Algorithm 4.1.

ALGORITHM 4.2

To get the normal number of a flow graph with reversed edge directions by using its original dominator tree with back edges and decision factors.

INPUT: The original dominator tree with back edges and decision factors of a flow graph.

OUTPUT: The normal number of a flow graph with reversed edge directions.

METHOD:

Step 1 Find a path $(V_1,V_2,...,V_n)$ from a sink V_1 bottom-up to the head node V_n of the forward edge (V_{n-1},V_n).

Step 2 The product of the decision factors of all of the nodes on the path $(V_1,V_2,...,V_n)$ yields the total number of loops.

Step 3 If there exists another forward edge, go to Step 1.

Step 4 The sum of all the numbers calculated by Step 2 is the normal number of G'.

For example, if we change all the edge directions of G_2, the normal number of G_2' can be calculated by following the steps of the algorithm as shown below.

Step 1 path 1=(10,8,7,4)

Step 2 The product of the decision factors of the nodes on the path is 1*2*1*2=4.

Step 1 path 2=(10,8,7,4,3)

Step 2 The product of the decision factors of the nodes on the path is 1*2*1*2*1=4.

Step 4 The sum is 4+4=8 which is the normal number of G_2'.

For this particular example, the normal numbers of G_2 and G_2' are different. Let's take another example and discuss it in more detail. Suppose a structured flow graph G_6 is given (Figure 26) and we need to answer the following questions.

1. What is the normal number of this graph?
2. What is the normal number after we change the edge directions?

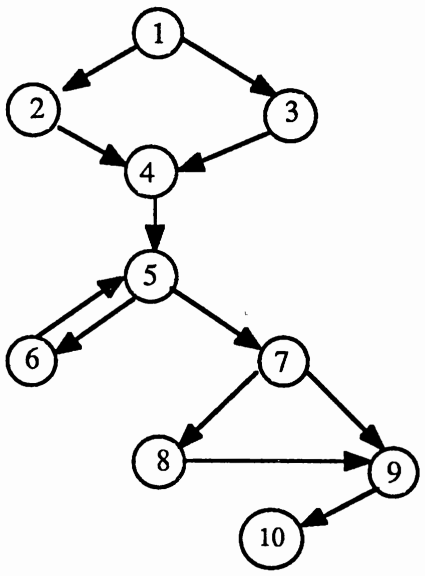
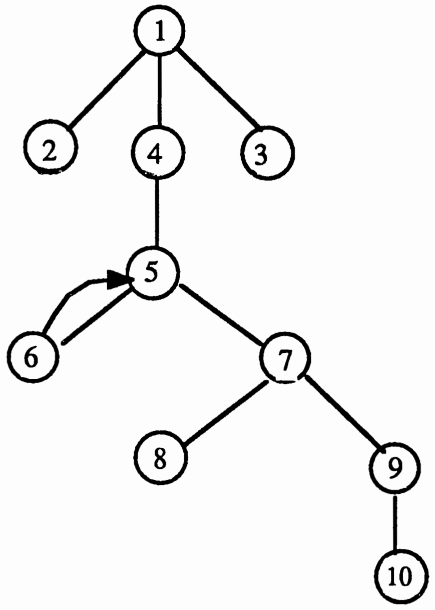


Figure 26. A structured flow graph G_6.

The answers to the questions can be obtained by following the steps of the algorithms described earlier in this section.

1. The dominator tree of G_6 with back edge (6,5) is as follows.



. Figure 27. The dominator tree of G_6 with a back edge.

By Step 1, the path is (1, 4, 5, 6). By Step 2, the number of loops in the path is 2*1*1=2. Since it only has one back edge, the normal number is 2.

2. Change the sink to source. By Step 1, the path is (10, 9, 7, 5, 6, 5). By Step 2, the number of loops in the path is 1*1*2*1*1*1=2. So the normal number of G_6' is 2 also.

Based on the aboved-mentioned discussion, we can make the following observations. For a structured flow graph, the cyclomatic number, which is dependent on the number of edges and vertices, will be the same however we change the edge directions. If a structured flow graph has a unique source and a unique sink, the track number of the graph should be the same as its cyclomatic number. So the track numbers of G and G' will be the same for a structured flow graph. The number of spanning trees will also be the same if we change the edge directions. Since a structured flow graph has no branching out of/into loops/decisions, corresponding to the out-degree of any decision vertex there should be a collecting vertex with the same in-degree. The number of spanning trees is the product of the in-degrees of all collecting vertices. After changing the edge directions, the new flow graph with a new root which is the sink of the original graph must have the same number of spanning trees.

4.2 Unstructured Flow Graphs

A directed graph, with branching out of/into loops/decisions is called an unstructured flow graph (see Section 2.3). Intuitively, an unstructured program has a higher complexity than a structured one with the same number of edges and vertices on their flow graphs. If we change all the edge directions of an unstructured flow graph G, an edge branching into/out of a decision/loop of G will become an edge branching out of/into a decision/loop of G'. So the new graph G' will also be unstructured.

Let's consider the unstructured flow graph G_3 in Figure 10. The cyclomatic number and track number of G_3 are four (see Sections 3.1 and 3.2). The simple closed loops of the normal form G_3^* are (1, 2, 3, 4, 5, 4), (1, 2, 3, 4, 5, 6, 3), (1, 2, 5, 4, 5), and (1, 2, 5, 6, 3, 4, 5) (see Section 3.3) hence the normal number of G_3 is four. The number of spanning trees of G_3 with root 1, which are depicted in Figure 23, is five (see Section 3.5). The intervals of G_3 are depicted in Figure 28 and the number of intervals of G_3 is four.

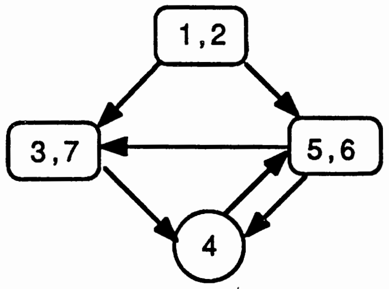


Figure 28. The intervals of graph G_3.

If we change all edge directions, we get graph G_3' in Figure 29.

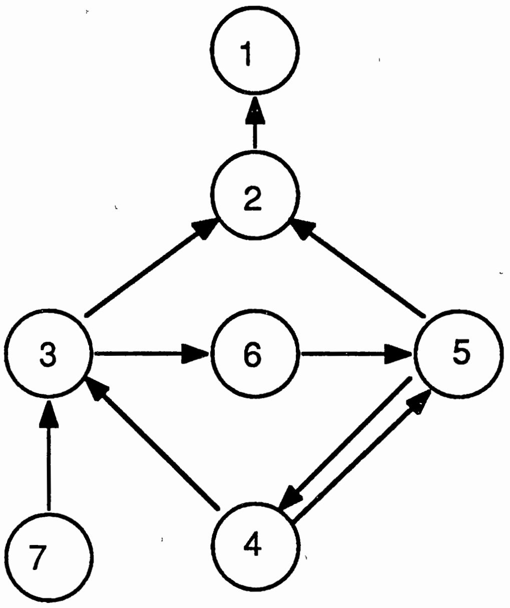


Figure 29. An unstructured graph G_3' which is graph G_3 with reversed edges.

The source and sink of G_3' are nodes 7 and 1, respectively. The edges (3, 2) and (5, 2) which are branching out of a loop (3, 6, 5, 4, 3) of G_3' correspond to the edges (2, 3) and (2, 5) which are branching into a loop (3, 4, 5, 6, 3) of G_3. The graph G_3' is an unstructured flow graph also. Now let's calculate the metrics which were discussed in Chapter III for the graph G_3'.

1. The cyclomatic number is 9-7+2=4.
2. The sets of tracks of G_3' under maximal track decomposition may be any of the following situations:
   a. (7, 3, 2, 1), (3, 6, 5, 2), (5, 4, 3), and (4, 5);
   b. (7, 3, 6, 5, 4, 5), (4, 3), (3, 2, 1), and (5, 2);
   c. (7, 3, 6, 5, 4, 5), (4, 3), (5, 2, 1), and (3, 2);
   d. (7, 3, 6, 5, 4, 3), (4, 5), (3, 2, 1), and (5, 2);
   e. (7, 3, 6, 5, 4, 3), (4, 5), (5, 2, 1), and (3, 2);
   f. (7, 3, 6, 5, 2, 1), (3, 2), (5, 4, 3), and (4, 5).

   The track number of G_3' is four.
3. The simple closed loops of G_3' are (7, 3, 6, 5, 4, 3) and (7, 3, 6, 5, 4, 5). The normal number of G_3' is two.
4. The spanning trees of G_3' rooted at 7 are as follows.

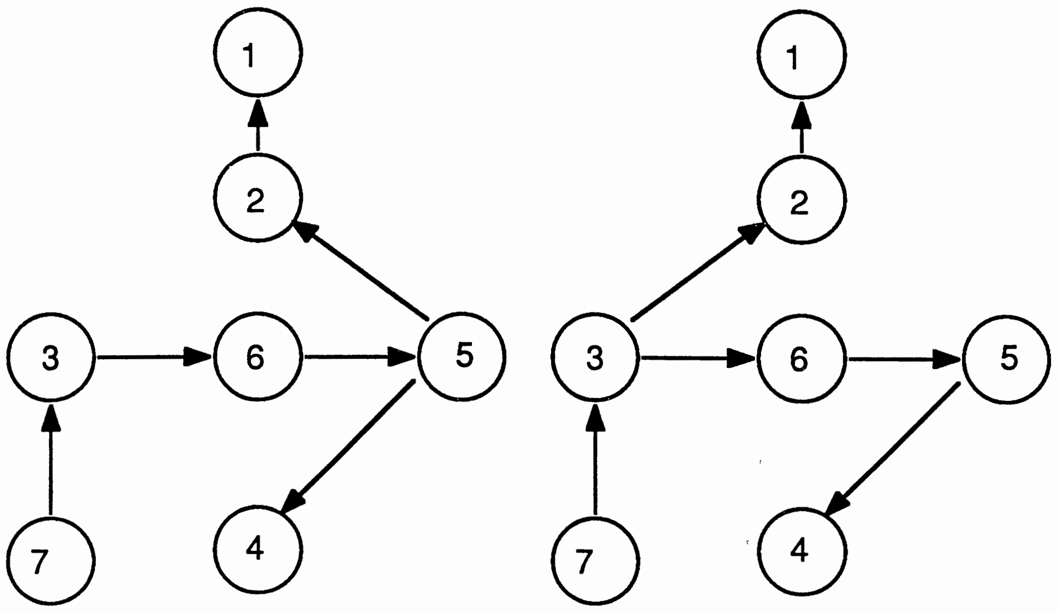


Figure 30. The two spanning trees of G_3' with root 7.

The number of spanning trees of G_3' with root 7 is two.

5. The intervals of G_3' are as follows.

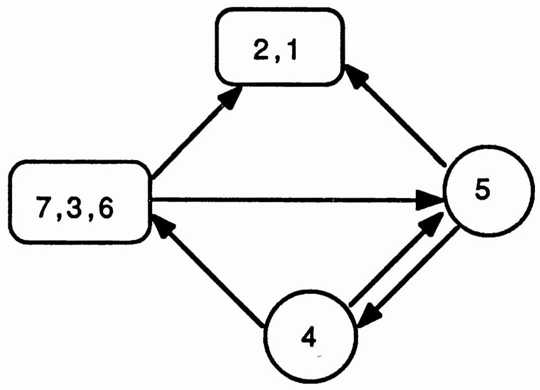


Figure 31. The intervals of graph G_3'.

The number of intervals of G_3' is four.

Based on the aboved-mentioned discussion, we can make the following observations. For an unstructured flow graph G (with a unique source and a unique sink), the cyclomatic number and track number are identical and they don't change for G' (the same graph as G with reversed edge directions). Since these two metrics are dependent on the number of edges and vertices, they will not change as a result of changing the edge directions.

The normal number of an unstructured flow graph G is dependent on the loops and the structure of vertices and edges that lead into the loop from the source. After we change the edge directions of G, the loops have the same vertices but the structure of vertices and edges (i.e., the structure of the vertices and edges leaving the loops to lead to the sink of G) are different. There seems to be no relationship between the structures of getting into a loop and leaving out of a loop. So the normal numbers of G and G' don't seem to be related and will have to be calculated separately.

The number of spanning trees of an unstructured flow graph is dependent on the given root and the in-degree matrix (see Section 3.5). The in-degree of a collecting node and the out-degree of a decision node are not related in an unstructured flow graph, so there are no relationships between the collecting nodes and decision nodes. The in-degree matrices of G and G' are different. Therefore the number of spanning trees of G and G' are in general different.

The intervals of a flow graph are dependent on the nodes' precedence from the source and the acyclic structure. There are no relationships between the nodes' precedence of an unstructured flow graph from the source and sink and a structured flow graph. So, in general, the intervals of G and G' are different.

4.3 Validation

A number of flow graphs, which have different number of vertices and edges, were examined in order to explore the sensitivity of the metrics to edge directions for structured and unstructured flow graphs (see APPENDIX A).

These twenty four flow graphs are reproduced from [Culik79, Culik81, McCabe76, McCabe89, and Aho86], or designed by the other. Some are structured (TABLE I), the others are unstructured (TABLE II).

TABLE I

GRAPH-THEORETIC NUMBERS OF A STRUCTURED FLOW GRAPH (G)
AND THE SAME GRAPH WITH REVERSED EDGES (G')

Graph	No. of Nodes	No. of Edges	Cyclomatic or Track No.	Normal Number G	Normal Number G'	No. of Spanning Trees G	No. of Spanning Trees G'	No. of Collecting Nodes	No. of Decision Nodes
G1	7	8	3	1	2	2	2	2	2
G2	7	8	3	2	2	1	1	2	2
G5	7	9	4	2	2	4	4	2	2
G13	10	12	4	2	2	4	4	3	3
G15	10	13	5	3	2	6	6	2	2
G16	10	14	6	6	8	8	8	4	4
G21	12	15	5	4	1	4	4	2	2
G22	12	15	5	3	6	4	4	4	4
G23	12	16	6	0	0	32	32	5	5

TABLE II

GRAPH-THEORETIC NUMBERS OF AN UNSTRUCTURED FLOW GRAPH (G) AND THE SAME GRAPH WITH REVERSED EDGES (G')

Graph	No. of Nodes	No. of Edges	Cyclomatic or Track No.	Normal Number G	Normal Number G'	No. of Spanning Trees G	No. of Spanning Trees G'	No. of Collecting Nodes	No. of Decision Nodes
G3	7	8	3	2	2	1	1	2	2
G4	7	9	4	4	2	5	2	3	3
G6	7	10	5	3	5	10	6	3	3
G7	7	10	5	0	0	12	9	3	2
G8	7	10	5	0	0	12	12	3	3
G9	7	10	5	2	6	6	12	3	3
G10	7	10	5	6	4	4	2	4	3
G11	7	10	5	4	6	4	3	3	3
G12	10	12	4	4	4	3	3	3	3
G14	10	13	5	3	5	4	6	4	4
G17	10	15	7	7	8	52	33	6	5
G18	12	14	4	2	4	2	6	2	3
G19	12	15	5	2	6	4	4	4	4
G20	12	15	5	4	4	6	8	3	4
G24	19	25	8	11	16	12	24	5	6

4.4 Summary

In this chapter, we have discussed the sensitivity of graph-theoretic metrics to edge directions for structured and unstructured flow graphs. Cyclomatic number and track number, which are calculated based on the number of vertices and edges, do not change as a result of the change of edge directions. The number of spanning trees and the number of intervals are also the same for structured ones because there are no branching out of/into loops/decisions. If we change all the edge directions of an unstructured flow graphs G, then the new graph G', except for having the same number of vertices and edges, is independent of G. The normal number, number of spanning trees, and number of intervals of G and G' are in general independent for an unstructured flow graph. They have to be recalculated by the methods discussed in Chapter III.

CHAPTER V

RESULTS, SUMMARY, AND FUTURE WORK

5.1 Results

For a structured flow graph, the number of binary collecting nodes and binary decision nodes should be the same, because there are no edges branching out of/into loops/decisions. If we change the edge directions, the decision nodes will be changed to collecting nodes, and vice versa. TABLE I contains some graph-theoretic numbers of structured flow graphs. We can have the following observations.

1. The number of spanning trees of G and G' are the same.
2. The number of collecting nodes and decision nodes are the same.
3. Cyclomatic and track numbers are the same.
4. The normal numbers of G and G' are not necessarily the same.

TABLE II contains some graph-theoretic numbers of unstructured flow graphs. Basically, G and G' are two independent graphs, except for having the same number of vertices and edges. We can have the following observations.

1. The number of spanning trees of G and G' are not necessarily the same.
2. The number of collecting nodes and decision nodes are not necessarily the same.

3. Cyclomatic and track numbers are the same.
4. The normal numbers of G and G' are not necessarily the same.

5.2 Summary

In this thesis, adjacency matrices were used to represent the flow graphs. A directed graph is used to represent a program in the form of a control flow or data flow graph. If a directed graph contains branching out of/into decisions/loops, it is called an unstructured flow graph. Otherwise it is a structured flow graph. Actually, an unstructured directed graph cannot exist with only one form of unstructuredness. It exists with at least one pair of the four possible situations.

There are a large number of metrics which attempt to measure the conceptual complexity of programs. The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is V(G)=e-n+2p. Some of the properties of the cyclomatic number are as follows.

1. V(G) >= 1.
2. V(G) is the maximal number of linearly independent paths in G.
3. Inserting/deleting functionsl elements to/from G does not change V(G).
4. Inserting a new edge increases V(G) by one.
5. V(G) only depends on the number of decisions and not the decision structure of G.

Track number is the number of tracks of a flow graph under the maximal track decomposition which breaks a graph into a set of tracks such that no tracks of the set are mutually crossing. We can construct many different maximal track decompositions. The number of tracks is equal to the cyclomatic number if the graph has a unique source and a unique sink.

The normal number is the minimal number of loops among all possible normal forms of a graph. A program is in a normal form if its flow graph is a cycle-free almost-tree (some leaves are turned back to their ancestors).

An interval of a directed graph is defined as a natural loop plus an acyclic structure that dangles from the nodes of the loop in the graph. The number which is obtained by the "interval partition algorithm" is called the number of intervals of the flow graph. An important property of intervals is that every interval has a header node that dominates all the nodes in the interval.

For a directed graph G, which is without slings (self-loops) but has a root, we can construct a directed spanning tree H of G with the same root. The number of all possible H's is called the number of spanning trees of G.

The cyclomatic number and track number, which are dependent on the number of edges and vertices, do not change as a result of the change of edge directions. The number of spanning trees and the number of intervals of G and G' will not change for a structured graphs as a result of changing the edge directions. But these two metrics have no relationships in an unstructured flow graph. The normal numbers of G and G' are not necessarily the same no matter whether they are structured or not.

5.3 Future Work

In this thesis, five graph-theoretic metrics were studied to gauge their sensitivity to edge direction changes. Other metrics can be used to represent the complexity of a program from other viewpoints. The input of the program in APPENDIX C is the adjacency matrix of a flow graph. The flow graphs were drawn manually. Programs to transform a given program to a flow graph do exist for some languages (e.g., FORTRAN). A general-purpose program can be written to transform a program written in a number of different languages to a flow graph.

REFERENCES

[Aho86]
Aho, A. V., Sethi, R., and Ullman, J. D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, Mass,1986.

[Bollobas78]
Bollobas, B., Extremal Graph Theory, Academic Press Inc. (LONDON) LTD., 1978.

[Cooper71]
Cooper, D. C., "Programs for mechanical program verification," in Machine Intelligence, B. Maltzer, D. Michie (eds.),1971, pp. 43-59.

[Culik77]
Culik, K., "Extensions of rooted trees and their applications," Discrete Mathematics, Vol. 18, 1977, pp. 131-148.

[Culik79]
Culik, K., "The cyclomatic number and the normal number of programs," ACM SIGPLAN Notices, Vol. 14, No. 4, Apr. 1979, pp. 12-17.

[Culik80a]
Culik, K., "Entry strong components and their application (in computer science)," Congressus Numerantium, Vol. 28, 1980, pp. 335-350.

[Culik80b]
Culik, K., "What is a flowchart loop and about structured programming," ACM SIGPLAN Notices, Vol. 15, No. 1, Jan. 1980, pp. 45-57.

[Culik81]
Culik, K. and Lang, S. D., "Two fundamental numbers of directed graph," Congressus Numerantium, Vol. 32, 1981, pp. 231-251.

[Elshoff78]
Elshoff, J. L. and Marcotty, M., "On the use of cyclomatic number to measure program complexity," ACM SIGPLAN Notices, Vol. 13, No. 12, Dec. 1978, pp. 29-40.

[Even79]
Even, S., Graph Algorithms, Computer Science Press, Rockville, Maryland,1979.

[Halstead72]
Halstead, M. H., "Natural laws controlling algorithm structure?" ACM SIGPLAN Notices, Vol. 7 No. 2, Feb. 1972, pp. 19-26.

[Halstead77]
Halstead, M. H., Elements of Software Science, New York, Elsevier North-Holland, 1977.

[Harrison82]
Harrison, W., Magel, K., Kluczny, R., and DeKock, A., "Applying software complexity metrics to program maintenance," IEEE Computer, Vol. 15, No. 9, Sept. 1982, pp. 65-79.

[Henry81a]
Henry, S., Kafura, D., and Harris, K., "On the relationships among three software metrics," Performance Evaluation Review, Vol. 10, No. 1, 1981, pp. 81-88.

[Henry81b]
Henry, S. and Kafura, D., "Software structure metrics based on information flow," IEEE Trans. on Software Engineering, Vol. SE-7, No. 5, Sept. 1981, pp. 510-518.

[Horowitz82]
Horowitz, E. and Sahni, S., Fundamentals of Data Structures, Rev. Ed., Prentice- Hall, 1982.

[Iyengar82]
Iyengar, S. S., Parameswaran, N., and Fuller, J., "A measure of logic complexity of programs," Computer Languages, Vol. 7, 1982, pp. 147-160.

[Knuth74]
Knuth, D. E., "Structured programming with go to statements," ACM Computing Surveys, Vol. 6, Dec. 1974, pp. 261-301.

[Laski82]
Laski, J. W., "On data flow guided program testing," ACM SIGPLAN Notices, Vol. 17, No. 9, Sept. 1982, pp. 62-71.

[Laski83]
Laski, J. W. and Korel, B., "A data flow oriented program testing strategy," IEEE Trans. on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 347-354.

[Lin89]
Lin, J. C. and Chang, C. G., "Zero-one integer programming model in path selection problem of structural testing," Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, COMPSAC89, 1989, pp. 618-627.

[Linger79]
Linger, R. C., Mills, H. D., and Witt, B. I., Structured Programming Theory and Practice, Addison-Wesley Publishing Company, 1979.

[Manna73]
Manna, Z., "Program schemas," in Currents in the Theory of Computing, Aho, A. V.(ed.), Prentice-Hall, 1973, pp. 90-142.

[Mayeda72]
Mayeda, W., Graph Theory, John Wiley & Sons Inc., Canada, 1972.

[McCabe76]
McCabe, T. J., "A complexity measure," IEEE Trans. on Software Engineering, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

[McCabe89]
McCabe, T. J., "Design complexity measurement and testing," Communications of the ACM, Vol. 32, No. 12, Dec. 1989, pp. 1415-1425.

[Oulsnam82]
Oulsnam, G., "Unravelling unstructured programs," The Computer Journal, Vol. 25, No. 3, 1982, pp. 379-387.

[Oviedo80]
Oviedo, E. I., "Control flow, data flow, and program complexity," Proceedings of the Fourth Annual International Computer Software & Applications Conference, COMPSAC80, 1980, pp. 146-152.

[Rapps82]
Rapps, S. and Weyuker, E. J., "Data flow analysis techniques for test data selection," Proceedings of the Sixth International Conference on Software Engineering, 1982, pp. 272-278.

[Schneidewind79]
Schneidewind, N. F. and Hoffmann, H. M., "An experiment in software error data collection and analysis," IEEE Trans. on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 276-286.

[Tai80]
Tai, K. C., "Program testing complexity and test criteria," IEEE Trans. on Software Engineering, Vol. SE-6, No. 6, Nov. 1980, pp. 531-538.

[Tai83]
Tai, K. C., "A program complexity metric based on data flow information in control graphs," Proceedings of the Seventh International Conference on Software Engineering, 1983, pp. 239-248.

[Temperley81]
Temperley, H. N. V., Graph Theory And Applications, Ellis Horwood Limited, New York, 1981.

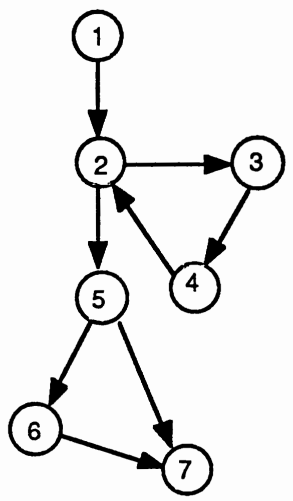
[Tutte84]
Tutte, W. T., Graph Theory, Addison-Wesley Publishing Company, Menlo Park, California, 1984.

[Woodward79]
Woodward, M. R., Hennell, M. A., and Hedley, D., "A measure of control flow complexity in program text," IEEE Trans. on Software Engineering, Vol. SE-5, No. 1, Jan. 1979, pp. 45-50.

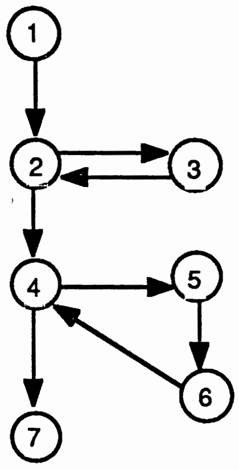
APPENDIX A

TWENTY FOUR TESTBED FLOW GRAPHS

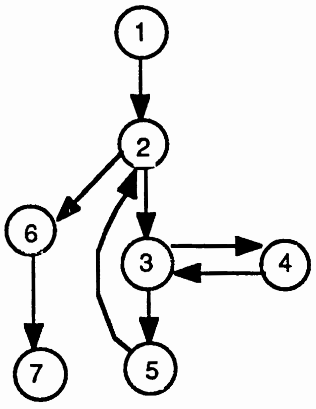
TWENTY FOUR TESTBED FLOW GRAPHS



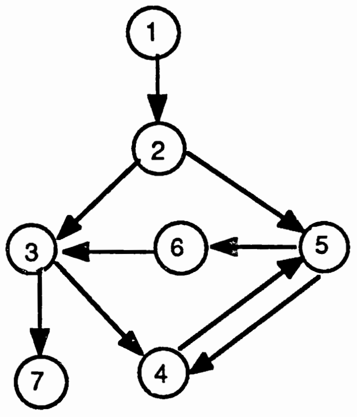
G1



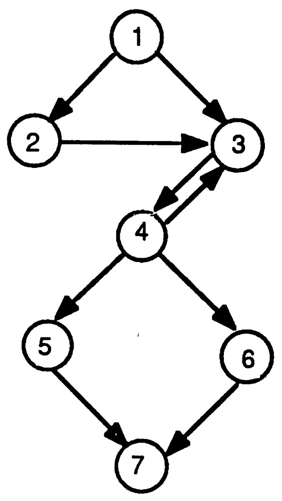
G2



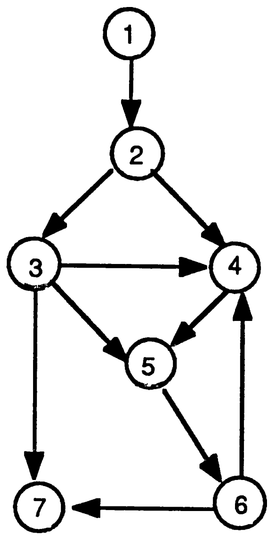
G3



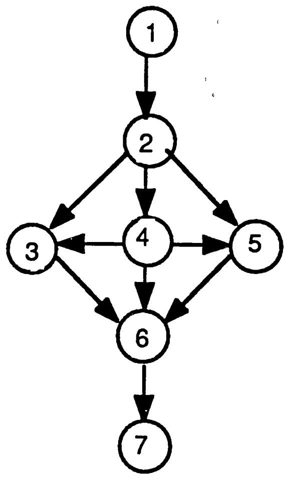
G4



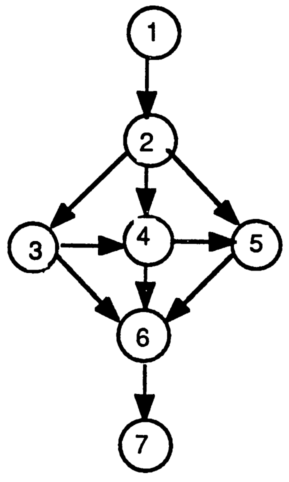
G5



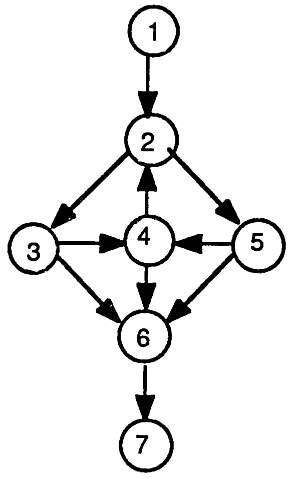
G6



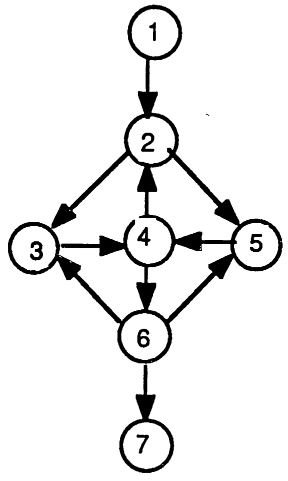
G7



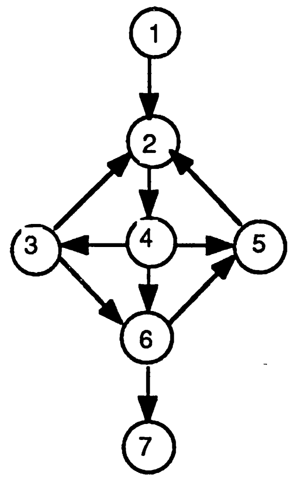
G8



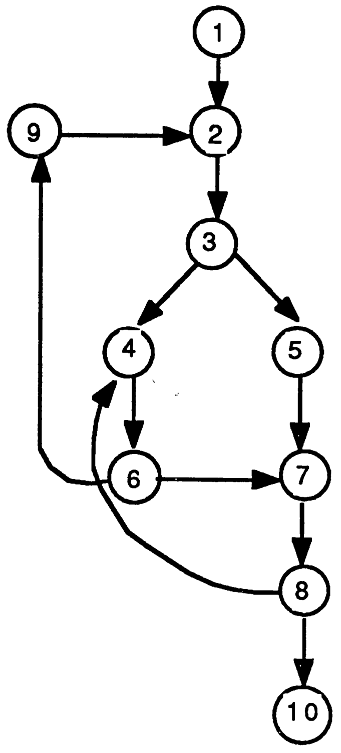
G9



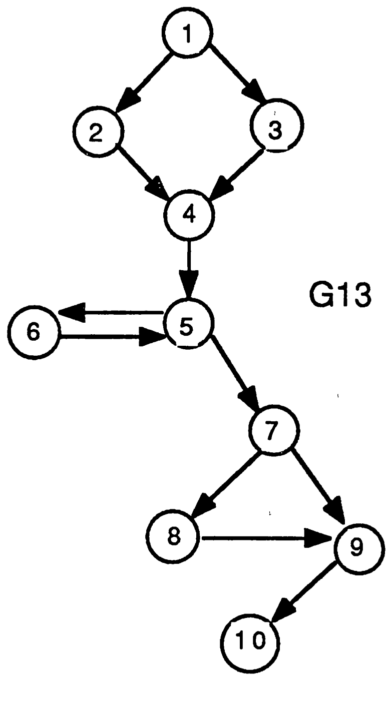
G10



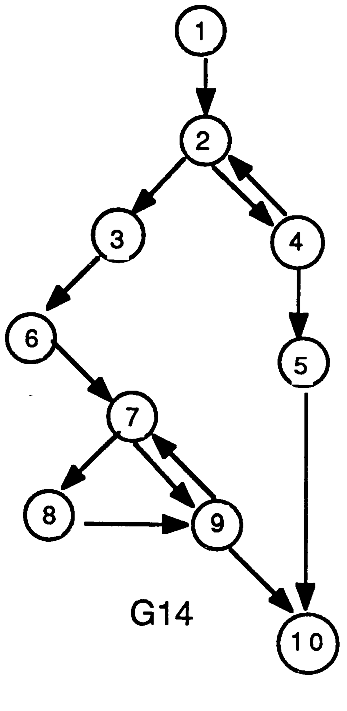
G11



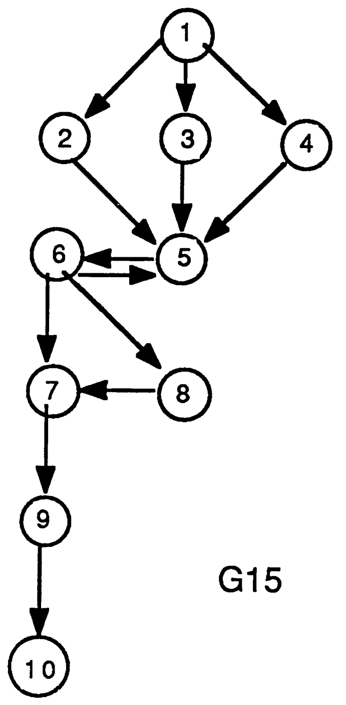
G12



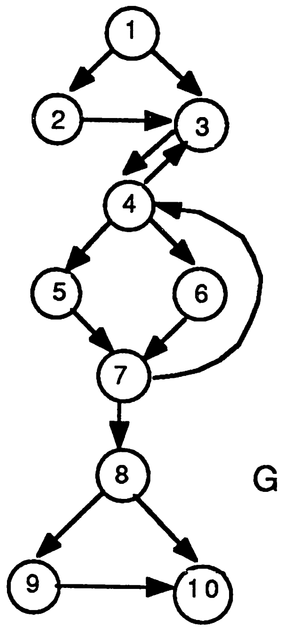
G13



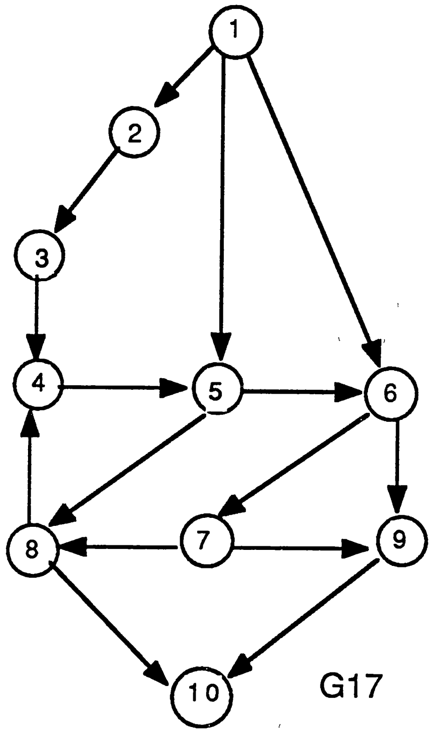
G14



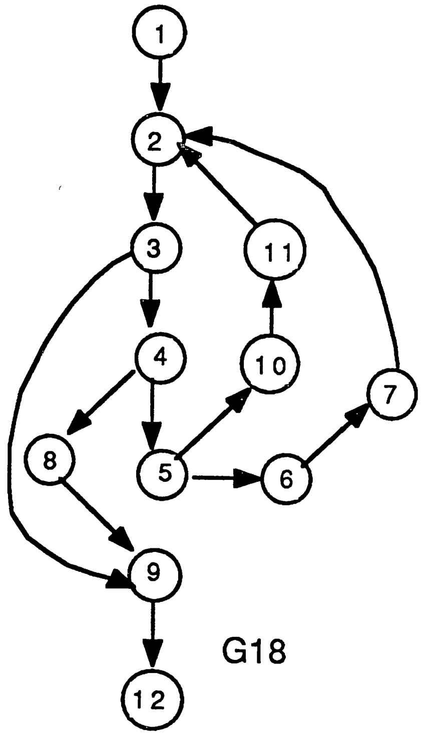
G15



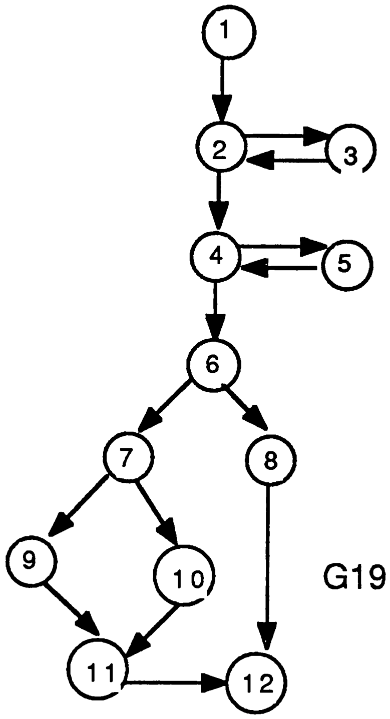
G16

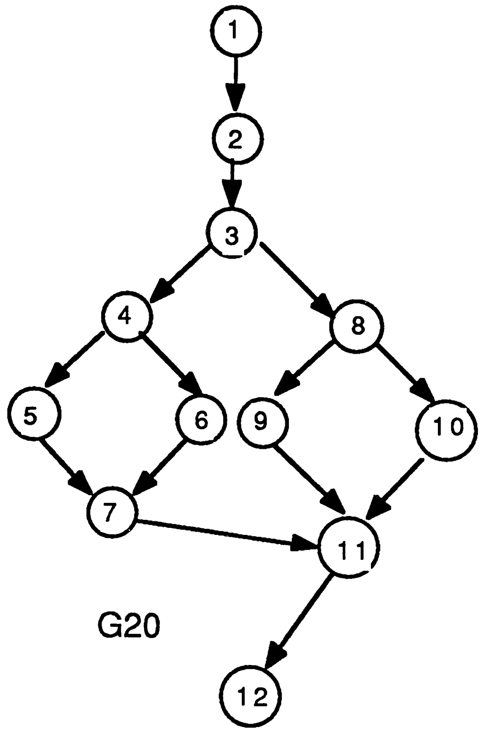


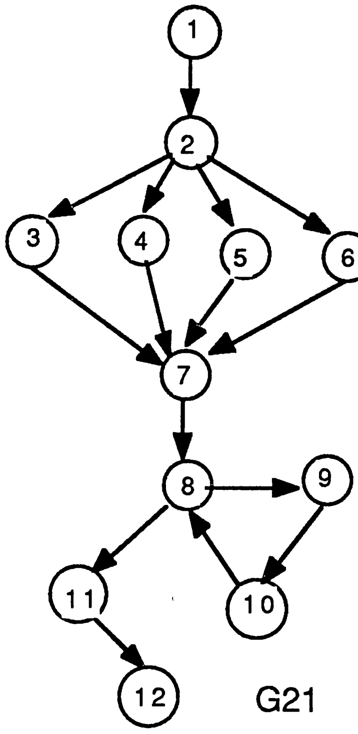
G17

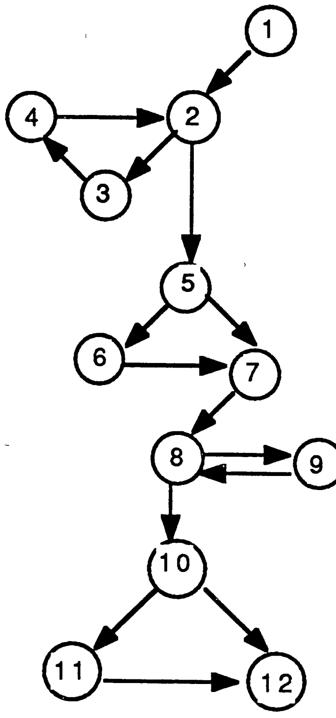


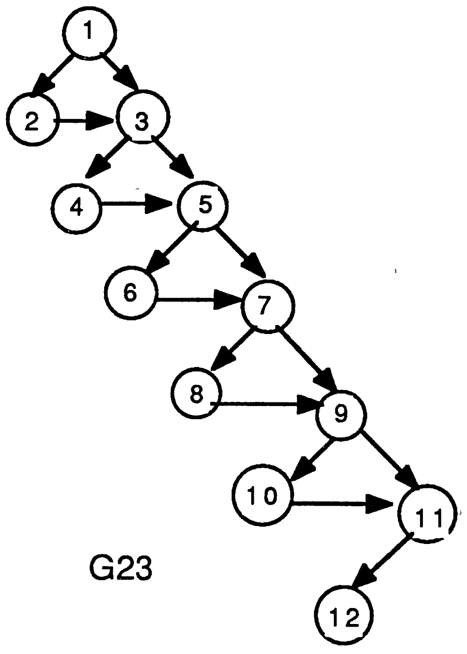
G18

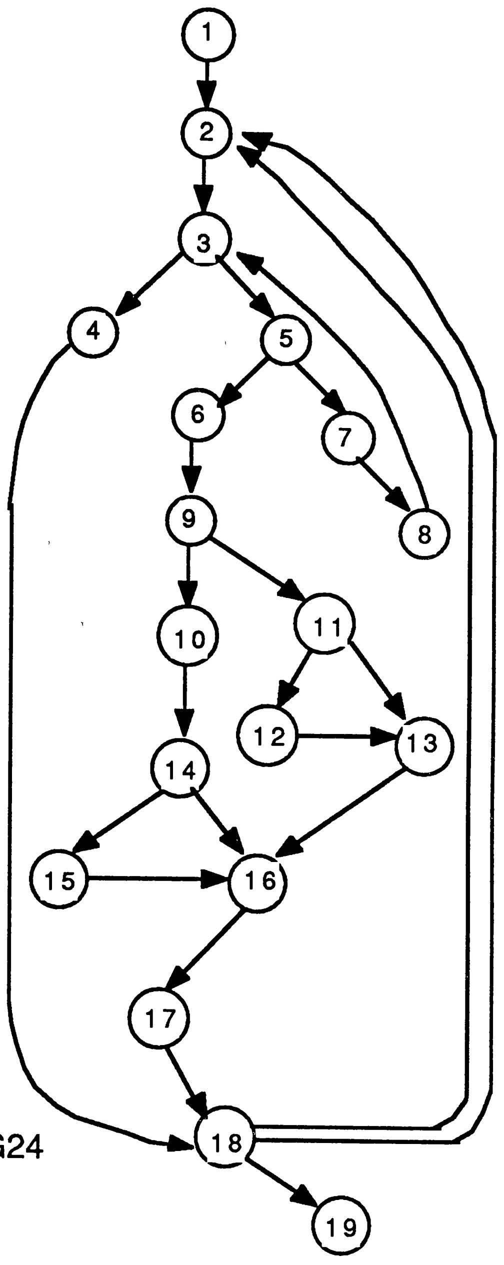

G19


G20


G21


G22

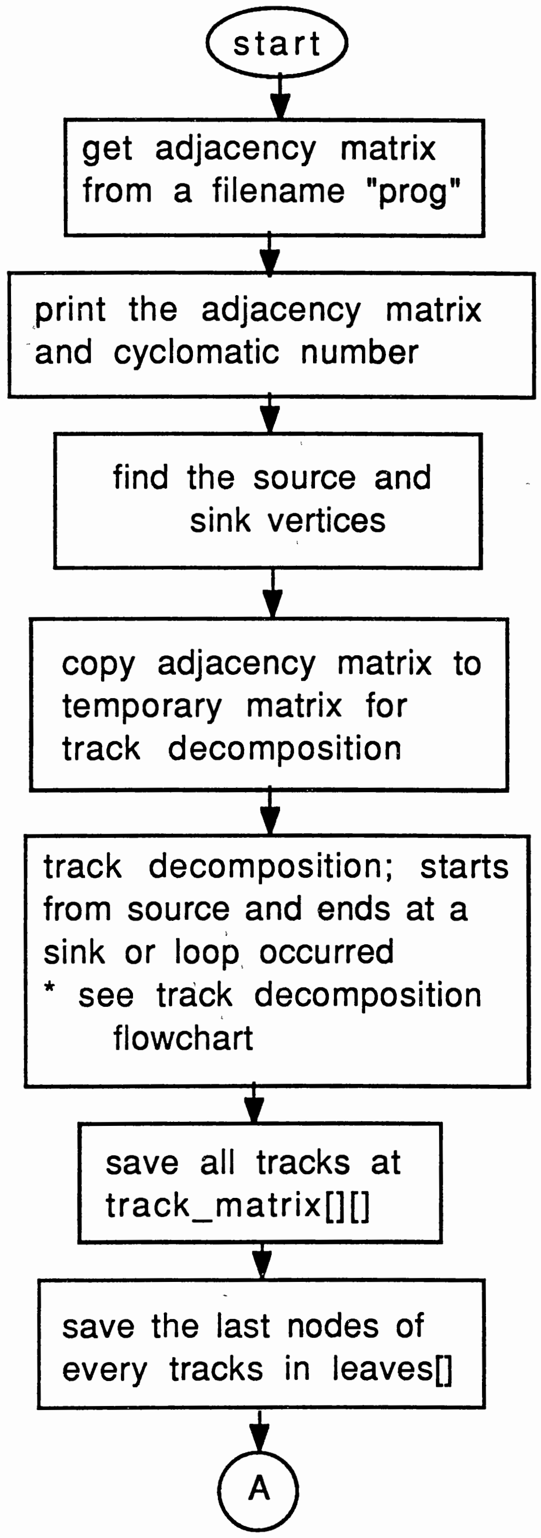

G23

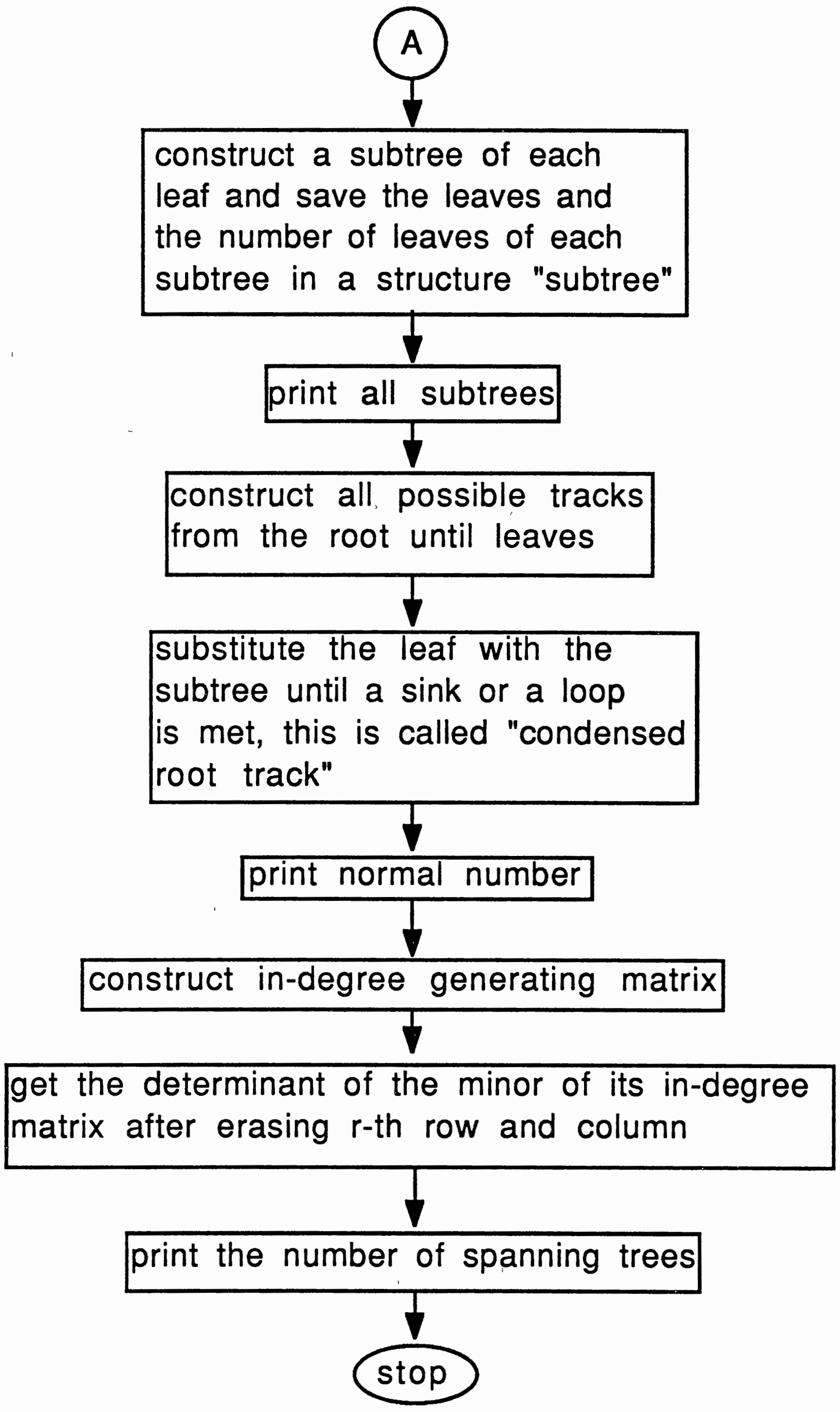

G24

APPENDIX B

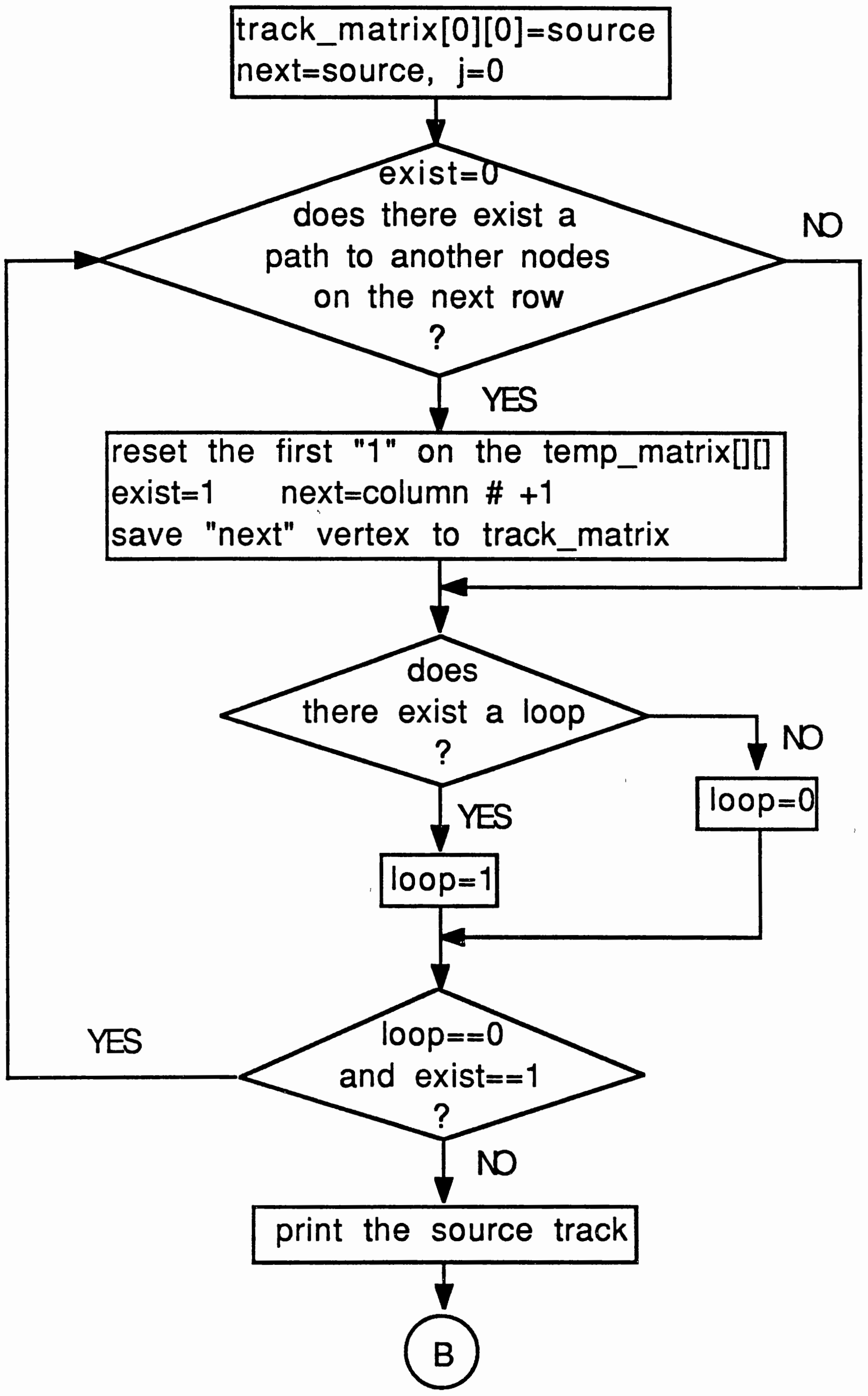
FLOWCHART OF THE PROGRAM IN APPENDIX C

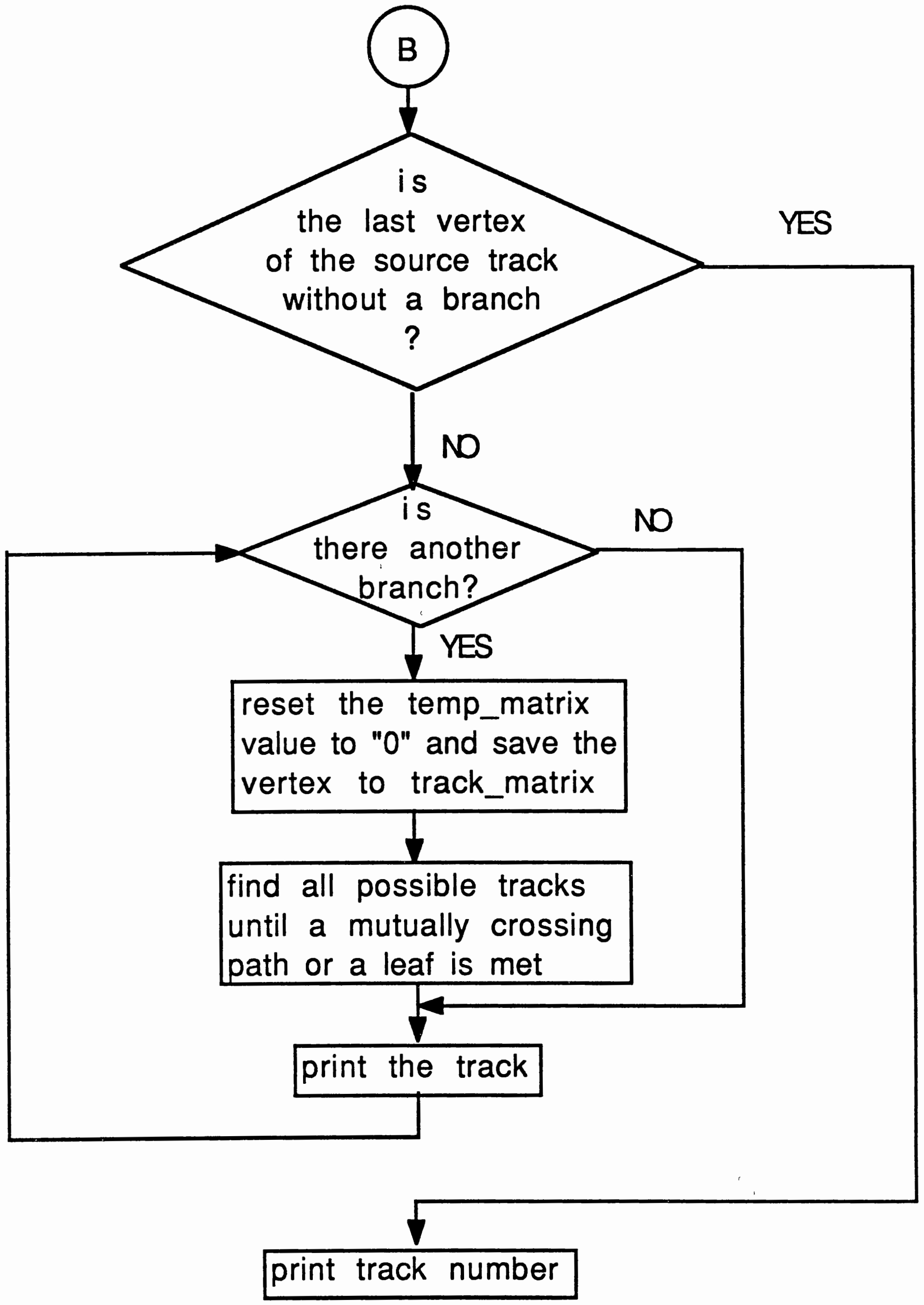
FLOWCHART OF THE PROGRAM IN APPENDIX C


start
get adjacency matrix from a filename "prog"
print the adjacency matrix and cyclomatic number
find the source and sink vertices
copy adjacency matrix to temporary matrix for track decomposition
track decomposition; starts from source and ends at a sink or loop occurred
* see track decomposition flowchart
save all tracks at track_matrix[][]
save the last nodes of every tracks in leaves[]
A


A
construct a subtree of each leaf and save the leaves and the number of leaves of each subtree in a structure "subtree"
print all subtrees
construct all possible tracks from the root until leaves
substitute the leaf with the subtree until a sink or a loop is met, this is called "condensed root track"
print normal number
construct in-degree generating matrix
get the determinant of the minor of its in-degree matrix after erasing r-th row and column
print the number of spanning trees
stop

track decomposition flowchart


track_matrix[0][0]=source
next=source, j=0
exist=0
does there exist a path to another nodes on the next row ?
NO
YES
reset the first "1" on the temp_matrix[][]
exist=1 next=column # +1
save "next" vertex to track_matrix
does there exist a loop ?
NO
loop=0
YES
loop=1
YES
loop==0 and exist==1 ?
NO
print the source track
B


B
is the last vertex of the source track without a branch ?
YES
NO
is there another branch?
NO
YES
reset the temp_matrix value to "0" and save the vertex to track_matrix
find all possible tracks until a mutually crossing path or a leaf is met
print the track
print track number

APPENDIX C

PROGRAM WRITTEN TO CALCULATE THE METRICS

(FOR MEASURING CYCLOMATIC NUMBER, TRACK NUMBER,
TRACK DECOMPOSITION, NORMAL NUMBER, AND
THE NUMBER OF SPANNING TREES)

PROGRAM WRITTEN TO CALCULATE THE METRICS

```
/*  Description

    This program determines program complexity by using cyclomatic
complexity, track number, normal number, and the number of
spanning trees.  The first step is to construct an adjacency matrix
from the flow graph.  The cyclomatic number C(G) of a graph G with n
vertices, e edges and p connected components is

C(G) = e-n+2p

    The second step is to find track number by track decomposition.
There are lots of paths to decomposite the flow graph, but the
number of the tracks of a set under the maximal track decomposition
must be the same i.e. track number.  Every sinks of the tracks are
called leaves.  Every leaves except the sink can build a tree from the
original flow graph.

    The third step is to construct a normal form of a directed graph.
The minimal number loops of the directed graph is called the normal
number.

    The fourth step is to construct the number of spanning trees.  The
number of directed spanning trees with root r of a directed graph
with no self-loops is given by the minor of its in-degree matrix
which results from the erasure of the r-th row and column.          */


#define vertex_num 7
#include <stdio.h>
int adj_matrix[vertex_num][vertex_num];  /* adjacency matrix   * /
int temp_matrix[vertex_num][vertex_num]; /* temporary matrix    */
int track_matrix[vertex_num[vertex_num];
                                   /* track decomposition matrix    */
int loop_comp[vertex_num][vertex_num];  /*  cyclic loop matrix */
int track_comp[vertex_num];    /* current track components          */
int  source,  sink,  leaves[vertex_num];
int  good_leaves[vertex_num],bad_leaves[vertex_num];
int  normal_num,edge_num,track_num,complexity;
int  leaves_num,good_leaves_num,bad_leaves_num,loop_num;
struct  sub_tree                    /* to construct a subtree of a leaf* /
{
   int  root;                                  /* the root of the subtree* /
   int  leaf_num;             /* the number of leaves of the subtree  * /
   int leaf[vertex_num];/* the components of leaves of    subtrees */
};
struct  sub_tree  tree[vertex_num];


/**************************************************/
/ *                                                                              * /
/*     Main program                                                           * /
/ *                                                                              * /
/**************************************************/

main()
{
int  i,j,k,m,exist,next,loop;
get_matrix();                                 /* get adjacency matrix  * /
prt_matrix();                              /* print adjacency matrix   * /
get_source_sink();          /* to find the source and sink vertices  * /
for(i=0;i<vertex_num;i++)   /* copy adjacency matrix to temporary
                                                                matrix              */
for(j=0;j<vertex_num;j++)
temp_matrix[i][j]=adj_matrix[i][j];
j=0;
track_matrix[0][0]=source;  /* track decomposition from source  * /
next=source;
do {
   exist=0;
   for(i=0;i<vertex_num  &&  temp_matrix[next-1][i]==0;i++)
   ;         /* to search the next component of the track and reset
                 the matrix if the vertex is not a sink, there exist a
                 next compent                                              * /
   temp_matrix[next-1][i]=0;
   if(i  !=  vertex_num)
      {
      exist=1;
      next=i+1;
      j++;
      track_matrix[0][j]=next;          /* to get the root track   * /
      for(k=0;k<j && track_matrix[0][k] != next;k++);
      if(k<j)
      loop=1;
      for(i=0;track_matrix[0][i]>0;i++)
      track_comp[i]=track_matrix[0][i];    /* copy the first to the
                                        track component            * /
      } /* if */
   }while(exist==1 && loop==0);
leaves[leaves_num++]=next;                  /* get the first leaf   * /
track_num++;
printf("\ntrack %d = ( ",track_num);
for(i=0;i<=j;i++)                    /* print the first track        * /
printf("%d    ",track_matrix[track_num-1][i]);
printf(")\n");
i=0;
while(track_comp[i]>0)  /* from the root track to get the other
                                                   tracks          * /
{
   next=track_comp[i];
   for(j=0;j<vertex_num  &&  temp_matrix[next-1][j]==0;j++)
   ;
   if(j==vertex_num)
      i++;
   else
      {
      temp_matrix[next-1][j]=0;
      track_matrix[track_num][0]=track_comp[i];
      m=1;
      do
         {
         exist=0;                            /* reset the exist  * /
         next=j+1;
         track_matrix[track_num][m++]=next;
         for(j=0;j<vertex_num  &&  temp_matrix[next-1][j]==0;j++);
         if(j<vertex_num  &&  cross(next)==0)
            {
            temp_matrix[next-1][j]=0;      /* exist the next one  * /
            exist=1;
            }
         }
      while(exist==1);
      track_num++;
      printf("\ntrack %d = ( ",track_num); /* print new track     * /
      for(i=0;track_matrix[track_num-1][i]);
      printf("%d    ",track_matrix[track_num-1][i]);
      printf(")\n");
      i=0;
      }
   }
printf("\nTrack Number = %d\n",track_num);
search_leaves();             /* to get all leaves                   * /
track_num=1;                     /* to construct subtree            * /
for(i=0;i<leaves_num;i++)
   {
   tree[i].root=leaves[i];
   get_subtree(i,leaves[i]);
   }
track_num=1;
get_roottree(source);                /* to construct a root tree   * /
loop_num=track_num-1;     /* right now, it has loop_num loops  * /
track_num=1;
for(i=0;i<loop_num;i++)
   {
   load_track(i); /* load track components from temp_matrix     * /
   for(j=0;track_comp[j]>0;j++);
      m=track_comp[j-1];
      while(m !=sink && checkloop(m) ==0)
         {
         for(k=0;leaves[k] != m ;k++);
         m=tree[k].leaf[0];   /* only get the leaves of a subtree   * /
         track_comp[j++]=m;
         for(loop=1;loop<tree[k].leaf_num;loop++)
            save_temp(tree[k].leaf[loop]);
         }
   if(m  != sink)
   normal_num++;
   prtroot();
   }
printf("\n\nNormal Number = %d\n",normal_num);
}
/*******************************************************/
/* Get the adjacency matrix from the file "prog"                     */
/*******************************************************/
```

```
get_matrix()
{
int  i,j;
FILE *ip;
ip=fopen("prog","r");
for(i=0;i<vertex_num;i++)
for(j=0;j<vertex_num;j++)
fscanf(ip,"%d",&adj_matrix[i][j]);
fclose(ip);
}


/*****************************************************/
/* Print the adjacency matrix                       * /
/*****************************************************/
prt_matrix()
{
int  i,j,k;
printf("\n\nAdjacency  Matrix\n\n        ");
for(i=1;i<vertex_num;i++)
printf("%3d",i);
printf("\n      ");
for(i=0;i<vertex_num;i++)
printf("---");
for(j=1;j<vertex_num;j++)
   {
   printf("\n%3d|",j);
   for(i=0;i<vertex_num;i++)
   printf("%3d",adj_matrix[j-1][i]);
   }
k=0;
for(i=0;i<vertex_num;i++)
for(j=0;j<vertex_num;j++)
if(adj_matrix[i][j]==1)
k++;
printf("\n\n\nCyclomatic  Complexity=%3d\n",k-vertex_num+2);
}
```

```
/***********************************************************/
/* Find the source and sink vertices                       */
/***********************************************************/
get_source_sink()
{
int  i,j;
for(i=0,source=0;i<vertex_num  &&  source==0;i++)
   {
   for(j=0;j<vertex_num  &&  adj_matrix[j][i]==0;j++)
   ;
   if(j==vertex_num)
   source=i+1;
   }
for(i=0,sink=0;i<vertex_num  &&  sink==0;i++)
   {
   for(j=0;j<vertex_num  &&  adj_matrix[i][j]==0;j++)
   ;
   if(j==vertex_num)
   sink=i+1;
   }
printf("\nsource = %d , sink = %d \n",source,sink);

/***********************************************************/
/* Whether the track decomposition in mutually crossing or not * /
/***********************************************************/
cross(m)
int  m;
{
int  j,k;
k=0;
for(j=0;track_comp[j]>0;j++)
if(m==track_comp[j])
k=1;
;
if(k==0)
track_comp[j]=m;
return(k);
}

/***********************************************************/
/* Save the last vertex of each track in the leaves array  * /
/***********************************************************/
```

```
search_leaves()
{
int  i,j,k,m;
for(i=0;i<track_num;i++)
   {
   for(j=0;track_matrix[i][j]>0;j++);
   k=track_matrix[i][j-1];
   for(m=0;leaves[m] != k && m<leaves_num;m++);
   if(m==leaves_num)
      {
      leaves[m]=k;
      leaves_num++;
      }
   }
printf("\n\nLeaves Number = %d\n\nleaves = ( ",leaves_num);
for(i=0;i<vertex_num;i++)
printf("%d    ",leaves[i]);
printf(")\n");
}

/*****************************************************/
/* Clear temporary matrix                            */
/*****************************************************/
cleartemp()
{
int  i,j;
for(i=0;i<vertex_num;i++)
for(j=0;j<vertex_num;j++)
temp_matrix[i][j]=0;
}

/*****************************************************/
/* Save track and current vertex m in temporary matrix */
/*****************************************************/
save_temp(m)
int  m;
{
int  i;
for(i=0;track_comp[i]>0;i++)
temp_matrix[loop_num][i]=track_comp[i];
temp_matrix[loop_num++][i-1]=m;
}
```

```
/**********************************************************/
/* Load track components from m-th row of the temporary matrix* /
/**********************************************************/
load_track(m)
int m;
{
int i;
for(i=0;i<vertex_num;i++)
track_comp[i]=0;
for(i=0;temp_matrix[m][i]>0;i++)
track_comp[i]=temp_matrix[m][i];
}

/**********************************************************/
/* Construct a subtree of each leaf and print it             * /
/**********************************************************/
get_subtree(i,m)
int i,m;
{
int j,k,n,r;
cleartemp();
loop_num=1;
temp_matrix[0][0]=m;
for(k=0;k<loop_num;k++)
   {
   load_track(k);
   for(j=0;track_comp[j]>0;j++)
   m=track_comp[j-1];
   do
   {
   r=0;        /* a flag to check tracks number              * /
   for(n=0;n<vertex_num;n++)
      {
      if(adj_matrix[m-1][n]>0)            /* if not a sink       * /
         {
         if(checkleaf(n+1)>0)             /* if it is a leaf     * /
            {
            if(r>0)         /* there exists at least two tracks  * /
            prttrack1(i,n+1);
            else
            prttrack(i,n+1);        /* there exist only one track  * /
            }
```

```
        else  if(checkleaf(n+1)==0)          /*if  not  a  leaf          * /
           {
           r++;
           if(r==1)
           track_comp[j++]=n+1;
           if(r>1)              /*  if  there  exists  another  track   * /
           save_temp(n+1);
           }
        }  /*   if * /
     }  /*  for   * /
     m=track_comp[j-1];
     }
     while(r>0);
  }  /*for* /
}

/*****************************************************************/
/* Print the tracks after track decomposition                   * /
/*****************************************************************/
prttrack(s,k)
int  s,k;
{
int  i;
printf("\nsubtree  %d  (  ",track_num++);
for(i=0;track_comp[i]>0;i++)
printf("%d    ",track_comp[i]);
printf("%d  )\n",k);
i=tree[s].leaf_num;
tree[s].leaf[i]=k;
tree[s].leaf_num++;
}

prttrack1(s,k)
int  s,k;
{
int  i,j;
for(i=0;track_comp[i]>0;i++)
;
j=i-1;
printf("\nsubtree  %d  (  ",track_num++);
for(i=0;i<j;i++)
printf("%d    ",track_comp[i]);
printf("%d  )\n",k);
```

```
i=tree[s].leaf_num;
tree[s].leaf[i]=k;
tree[s].leaf_num++;
}

/*********************************************************/
/* Construct a root tree which ends at bad leaves        */
/*********************************************************/
getroottree(m)
int  m;
{
int  j,k,n,r;
cleartemp();
loop_num=1;
temp_matrix[0][0]=m;
for(k=0;k<loop_num;k++)
   {
   load_track(k);
   for(j=0;track_comp[j]>0;j++)
   ;
   m=track_comp[j-1];
   do
   {
   r=0;        /* a flag to check tracks number               */
   for(n=0;n<vertex_num;n++)
      {
      if(adj_matrix[m-1][n]>0)            /* if not a sink       */
         {
         if(checkleaf(n+1)>0)             /* if it is a leaf     */
            {
            if(r>0)          /* there exists at least two tracks */
            prtcycle1(i,n+1);
            else
            prtcycle(i,n+1);         /* there exist only one track */
            }
         else if(checkleaf(n+1)==0)       /*if not a leaf        */
            {
            r++;
            if(r==1)
            track_comp[j++]=n+1;
            if(r>1)            /* if there exists another track  */
            save_temp(n+1);
            }
```

```
          }  /*  if * /
       }  /*  for   * /
       m=track_comp[j-1];
       }
       while(r>0);
    }  /*for* /
}

/*************************************************************/
/* Print the condensed root tree which is substituted by leaves   * /
/*************************************************************/
prtcycle(k)
int k;
{
int i;
printf("\nroot-tree  %d  (  ",track_num++);
for(i=0;track_comp[i]>0;i++)
    {
    temp_matrix[track_num-2][i]=track_comp[i];
    printf("%d    ",track_comp[i]);
    }
printf("%d  )  \n",k);
temp_matrix[track_num-2][i]=k;
}

prtcycle1(k)
int k;
{
int  i,j;
for(i=0;track_comp[i]>0;i++)
;
j=i-1;
printf("\nroot-tree  %d  (  ",track_num++);
for(i=0;i<j;i++)
    {
    temp_matrix[track_num-2][i]=track_comp[i];
    printf("%d    ",track_comp[i]);
    }
printf("%d  )\n",k);
temp_matrix[track_num-2][i]=k;
}
```

```
/**********************************************************/
/* Construct and print root tree which ends at leaves array      * /
/**********************************************************/
prtroot()
{
int  i;
printf("\ncondensed root-track %d = ( %d",track_num++,
track_comp[0]);
for(i=1;track_comp[i]>0;i++)
   printf("==>%d",track_comp[i]);
printf("    )\n");
}

/**********************************************************/
/* Check whether the vertex m is a leaf or not                   * /
/**********************************************************/
checkleaf(m)
int  m;
{
int  i,j;
i=0;
for(j=0;j<leaves_num;j++)
if(leaves[j]==m)
i=1;
return(i);
}

/**********************************************************/
/* Check whether vertex m is a loop component or not            * /
/**********************************************************/
checkloop(m)
int  m;
{
int  i,j;
for(i=0,j=0;track_comp[i]>0;i++)
if(track_comp[i]==m)
j++;
return(j-1);
}

/**********************************************************/
/* Get the number of spanning trees from in-degree matrix       * /
/**********************************************************/
```

```
spanning_tree()
{
int  i,j,k,n;
double  s,c,x[vertex_num][vertex_num],y[vertex_num][vertex_num];
      /* add the nondiagonal elements in each column to diagonal
          element and change the sign of every nondiagonal elements* /
for(i=0;i<vertex_num;i++)
   {
   k=0;
   for(j=0;j<vertex_num;j++)
      {
      temp_matrix[j][i]  =  -adj_matrix[j][i]; /* change sign         * /
      k  +=  adj_matrix[j][i];      /* add nondiagonal elements       * /
      }
   temp_matrix[i][i]=k;       /* put the result to diagonal           * /
   }
printf("\n\nIndegree Generating Matrix\n\n      ");
for(i=1;i<vertex_num+1;i++)
printf("%3d",i);
printf("\n      ");
for(i=0;i<vertex_num;i++)
printf("---");
for(j=1;j<vertex_num+1;j++)
   {
   printf("\n%3d|",j);
   for(i=0;i<vertex_num;i++)
      printf("%3d",temp_matrix[j-1][i]);
   }
      /* copy the indegree generating matrix to x and y matrix and
            change the value from integer to double precision        * /

for(i=0;i<vertex_num-1;i++)
for(j=0;j<vertex_num-1;j++)
   {
   x[i][j]=temp_matrix[i+1][j+1];
   y[i][j]=x[i][j];
   }
s=1.0;                      /* set the initial value is 1 for matrix  * /
n=vertex_num-1;
for(i=0;i<n-1;i++)
{
if(y[i][i]  !=  0,0)
```

```
   s *= y[i][i];
else                        /* if the diagonal is 0, then change with
                                      the nonzero column              * /
   {
   for(k=i+1;k<n;k++)
      {
      if(y[i][k] != 0.0)
         {
         for(j=i;j<n;j++)
            {
            c=y[j][k];
            y[j][k]=y[j][i];
            y[j][i]=c;
            }               /* change column                          * /
         goto a1;
         } /* if */
      } /* for */
   s *= y[i][i];
   goto a2;                 /* if the row is 0, then end              * /
a1:
    s *= -y[i][i];
   } /* else */
for(j=i+1;j<n;j++)   /* reset the first element of each column to 0
                                 of the first row in matrix           * /
   {
   if(y[i][j] != 0.0)
      {
      c=y[i][j]/y[i][i];              /* get the ratio                * /
      for(k=i;k<n;k++)
      y[k][j] -= y[k][i]*c;           /* and substitute it            * /
      } /* if */
   } /* for */
} /* for */
s *= y[n-1][n-1];
k=s;
a2:
printf("\n\nThe Number of Spanning trees of root =");
printf(" %d is %d\n",source,k);
}
```

VITA

Chinsyh Hu

Candidate for the Degree of

Master of Science

Thesis: SENSITIVITY OF GRAPH-THEORETIC METRICS TO EDGE DIRECTIONS FOR STRUCTURED AND UNSTRUCTURED PROGRAMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Taoyuan, Taiwan, Republic of China, Aug. 12, 1962, the son of Mr. Fu-an Hu and Mrs. Su-in Liu.

Education: Graduated from the Chung-kung Senior High School, Taipei, Taiwan, in July 1980; received the Bachelor of Engineering degree with a major in Electronic Engineering from Tamkang University, in June, 1984; completed requirements for the Master of Science degree at Oklahoma State University in December, 1990.

Professional Experience: Engineer in Development Engineering Department, Power Semiconductor Division, General Instruments of Taiwan Ltd., Taipei, Taiwan, Republic of China, 1986-1988.