SOME ALGORITHMS FOR THE ANALYSIS OF COMPUTER PROGRAMS

by

Lloyd D. Fosdick
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-126-78 April, 1978

INTERIM TECHNICAL REPORT

U.S. ARMY RESEARCH OFFICE

CONTRACT NO. DAAG29-78-G-0046

Approved for public release;
Distribution Unlimited

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

Field	Entry
1. REPORT NUMBER	
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (*and Subtitle*)	Some Algorithms For The Analysis Of Computer Programs
5. TYPE OF REPORT & PERIOD COVERED	Interim Technical Report
6. PERFORMING ORG. REPORT NUMBER	CU-CS-126-78
7. AUTHOR(s)	Lloyd D. Fosdick
8. CONTRACT OR GRANT NUMBER(s)	Contract number DAAG29-78-G-0046
9. PERFORMING ORGANIZATION NAME AND ADDRESS	University of Colorado at Boulder Boulder, Colorado 80309
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS	U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709
12. REPORT DATE	April, 1978
13. NUMBER OF PAGES	25
14. MONITORING AGENCY NAME & ADDRESS (*if different from Controlling Office*)	
15. SECURITY CLASS. (*of this report*)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	NA

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

NA

18. SUPPLEMENTARY NOTES

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Program Validation

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

The analysis of computer programs is an important part of program translation, error detection, optimization, and documentation. It consists of two distinct activities: the construction of an abstract model of a program, given the program itself in some language such as FORTRAN, and the extraction of information from the program by examination of the model. A labeled, directed graph is a model that is often used. In recent years workers in theoretical computer science have constructed and analyzed algorithms for solving problems on labeled, directed graphs which are directly related to important problems (over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

CONTENTS

Page

I. Abstract 1

II. 1. Introduction 1

2. Program Abstractions 2

3. Algorithms 12

4. Conclusions 23

III. Bibliography 24

FIGURES

	Page
Fig. 2.1	3
Fig. 2.2	4
Fig. 2.3	5
Fig. 2.4	7
Fig. 2.5	8
Fig. 2.6	9
Fig. 2.7	11
Fig. 3.1	14
Fig. 3.2	15
Fig. 3.3	18
Fig. 3.4	19
Fig. 3.5	20
Fig. 3.6	22

ABSTRACT. The analysis of computer programs is an important part of program translation, error detection, optimization, and documentation. It consists of two distinct activities: the construction of an abstract model of a program, given the program itself in some language such as FORTRAN, and the extraction of information from the program by examination of the model. A labeled, directed graph is a model that is often used. In recent years workers in theoretical computer science have constructed and analyzed algorithms for solving problems on labeled, directed graphs which are directly related to important problems arising in the analysis of computer programs. Some of these algorithms and their applications are described. The discussion does not assume a knowledge of graph theory.

1. INTRODUCTION. The analysis of computer programs is important in error detection, optimization, documentation, translation and many other activities associated with the design, construction, and maintenance of software. It is difficult because of the size and complexity of programs. Generally speaking, program analysis has been an unorganized subject, consisting of a variety of ad hoc techniques with little unifying structure, but this situation is changing. One reason for this change is the development of good algorithms for recognizing the implicit relationships in directed graphs. I will describe some of these algorithms and how they can be used on problems in program analysis. In doing so I hope to provide an indication of how we can begin to organize the subject of program analysis.

I want to make a careful distinction between the analysis of algorithms [1] and the analysis of programs. The analysis of an algorithm starts with the algorithm and focuses on the problem of obtaining time and memory space bounds or expectation values in terms of a few parameters of the underlying problem. The analysis of a program starts with a program, say in FORTRAN or COBOL, and proceeds to some abstract model of it on which the analysis is performed. The "program" is expressed exactly as it will be, or has been, read into the store of the computer. The construction of the abstract model of the program is a critical step in the analysis of programs, critical because decisions must be made about which information is to be discarded and which is to be retained, and the analysis will suffer if too much or too little is discarded. Program analysis is concerned with time and memory space requirements but it is also concerned with far more detailed information than is algorithm analysis; for example, program analysis is concerned with which variables are assigned values in certain areas of the program, which subroutines are called by which other subroutines, which variables depend on which other variables, and so forth. Finally, because of the large amount of information that arises in program analysis, most of the work, if not all, is done by computers whereas the analysis of algorithms is done by humans.

2. PROGRAM ABSTRACTIONS. The basic structure for a program abstraction is a directed graph [2]. The subject of graphs dates back to Euler and the famous Königsberg bridge problem [3]. Graphs appeared in automatic computing at an early date [4] and since then they have been used extensively to represent machines, programs, data structures and problems attacked by computers such as network flow problems. A directed graph is shown in Fig. 2.1. It is denoted by G(N,E) where N is a set of points called nodes, joined by a set, E, of directed lines, called edges. Letters or numbers are used to identify nodes as in

$$N = \{a, b, c, d, e, f\} \text{ or } N = \{1, 2, 3, 4, 5, 6\}$$

and ordered pairs of nodes are used to represent edges, as in

$$E = \{(a,b), (b,c), (b,d), (d,e), (e,d), (d,f), (c,f)\}$$

to represent the set of edges of the graph in Fig. 2.1. The edge (a,b) is said to leave a and enter b. The notation $|S|$ is used to represent the number of elements in the set S; $|N| = 6$ and $|E| = 7$ for the graph shown in Fig. 2.1. A path in a graph is a sequence of nodes joined by edges; for example $a \to b \to d \to e$ and $b \to c \to f$ are paths in Fig. 2.1. Sometimes intermediate nodes along the path are suppressed and the notation $a \overset{*}{\to} e$ is used to denote a path from node a to node e. The length of a path is the number of edges on the path: the length of $a \to b \to c \to f$ is three. A path in which the first and last nodes are the same is a cycle; for example, $d \to e \to d$ and $e \to d \to e \to d \to e$ are cycles in Fig. 2.1. A path such as $a \to b \to d \to e \to d$ in Fig. 2.1 is said to contain a cycle. If no path in the graph is a cycle the graph is called acyclic. If every node, save one called the root, has exactly one edge entering it and the root has no edges entering it, then the graph is a tree. A tree is an acyclic graph but the converse is not true. A picture of a tree and an acyclic graph which is not a tree are shown in Fig. 2.2. If (i,j) is an edge in a tree then node i is called the parent of node j.

It is common practice to represent control flow with a graph called a flow graph. In such a representation the nodes identify statements and the edges identify the order of execution of the statements. When one considers creating such an abstraction mechanically it becomes immediately evident that a simple one-to-one mapping of statements to nodes is not always adequate. An example of the kind of problem that can be encountered is illustrated with the DO statement in FORTRAN which actually consists of three, more elementary, statements separated by a group of statements following the DO. This is illustrated in Fig. 2.3. Even though a node in a flow graph may not represent a statement exactly because of such complications it is convenient, and should cause no confusion here, to speak of nodes in flow graphs as representing statements. A node in a flow graph with no edges entering it is called an entry node, and a node with no edges leaving it is called an exit node. A subroutine in ANS FORTRAN would be represented by a flow graph with one entry node, corresponding to the first executable statement and one or more exit nodes corresponding to RETURN statements. It is customary to restrict flow graphs to have a single entry node and to represent them with the notation $G(N,E,n_0)$ where n_0, an element of the set N, is the single entry node.

The nodes of a flow graph may represent larger structures than individual statements: they may represent statement blocks, cycles, subroutines and so

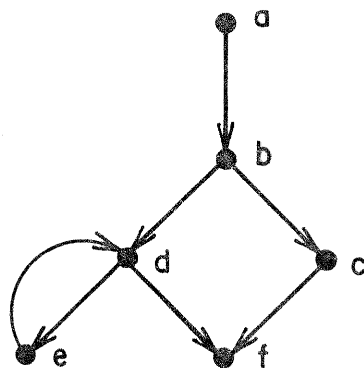


Fig. 2.1

A directed graph. The dots represent nodes, the directed lines represent edges. The node set is N = {a, b, c, d, e, f}, the edge set is E = {(a,b), (b,c), (b,d), (d,e), (e,d), (d,f), (c,f)}.

Fig. 2.2

Both of these graphs are acyclic but only the graph on the left is a tree. The graph on the right is not a tree because it has a node with two edges entering it.

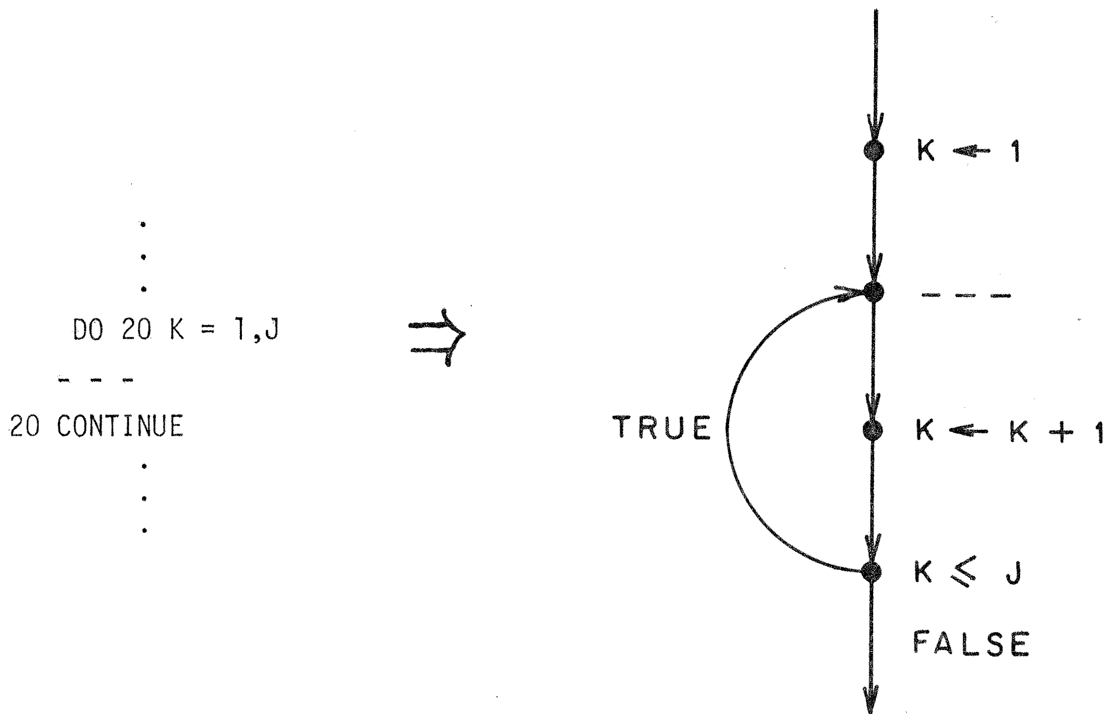


Fig. 2.3

The mapping of statements onto the nodes of a flow graph is not always 1-1. The mapping of the DO statement in FORTRAN is an example of such an exception, as shown here.

forth. Each of these is a different program abstraction retaining some aspects of control flow and suppressing others. A statement block is a sequence of statements such that control always flows from one immediately to the next; thus a branch statement can only appear at the end of a block and only the first statement can be branched to by some other statement. The transformation of a flow graph in which nodes represent statements to one in which nodes represent statement blocks is illustrated in Fig. 2.4. The resulting graph is more compact but still shows all branches in control flow and all cycles. The transformation of such a graph into one in which cycles are suppressed by collecting them into single nodes is illustrated in Fig. 2.5. Here cycle information is lost but connectivity relationships between cycles are retained. In graph terminology the nodes of the resulting flow graph represent the strongly connected components of the original flow graph. When the nodes of a flow graph represent subroutines or procedures and the edges represent one or more procedure calls, the graph is named a call graph. This is illustrated in Fig. 2.6. Since FORTRAN prohibits recursion a call graph for a FORTRAN program must be acyclic. The use of a call graph together with flow graphs for individual subroutines naturally partitions the abstract representation of a program into more manageable form for analysis.

Given a flow graph the following questions may be asked about its structure:

1) Is it acyclic?
2) What nodes lie on some path from a given node?
3) Is it possible to construct a path which includes a given set of nodes (edges)?
4) Can you find a path from the entry node to an exit node which does not include both members of certain pairs of edges?
5) Which sets of nodes have the property that there is a path from any node to any other node of the set?

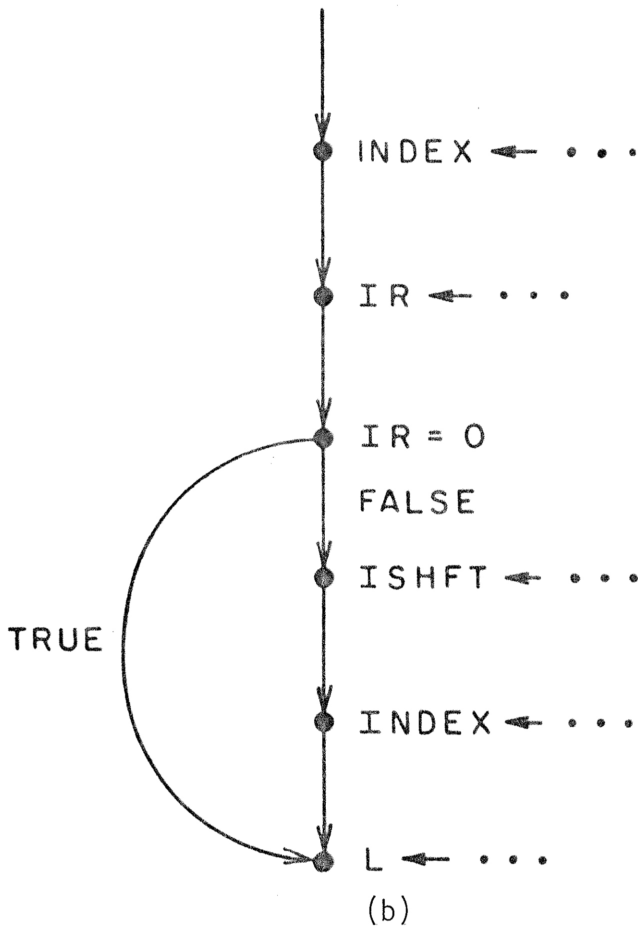
These questions and others are related to important questions that may be asked about a program:

1') Is it possible for this program to loop indefinitely during execution?
2') Can statement s_i be executed before statement s_j?
3') What is the smallest set of test data needed to execute every statement (traverse every edge) once?
4') Can you find an executable sequence of statements?
5') Can statement s_i and s_j be in a cycle?

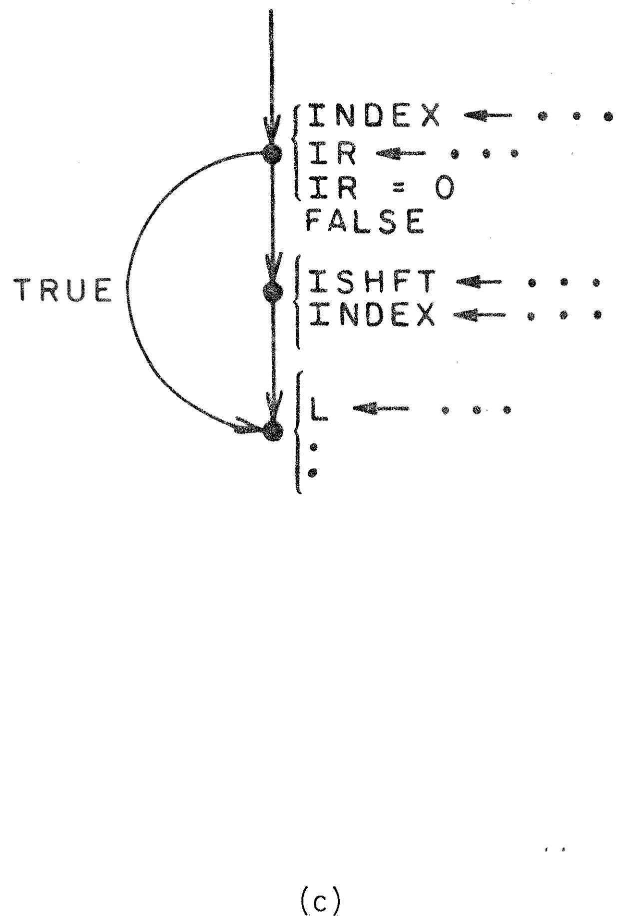
The questions about graph structure clearly can be answered by enumeration since the graphs are finite, but graphs derived from programs may be large and enumeration impractical. Therefore it is important to find algorithms which are significantly faster than enumeration and it is important to know about the complexity of these problems. It is known, for example, that question 4 is a problem that is called NP-complete [5]: this means that the worst case execution time of any algorithm for solving this problem on

```
      IF ( - - ) GO TO 90
          .
          .
          .
90    INDEX = J/K
      IR = J - INDEX * K
      IF (IR.EQ.0) GO TO 95
      ISHFT = K - IR
      INDEX = INDEX + 1
95    L = M (INDEX)
          .
          .
          .
```

(a)



(b)



(c)

Fig. 2.4

(a) Segment of a FORTRAN PROGRAM: (b) Segment of a flow graph derived from the program segment, with nodes representing individual statements; (c) Segment of a flow graph, derived from the graph in (b), in which nodes represent statement blocks.

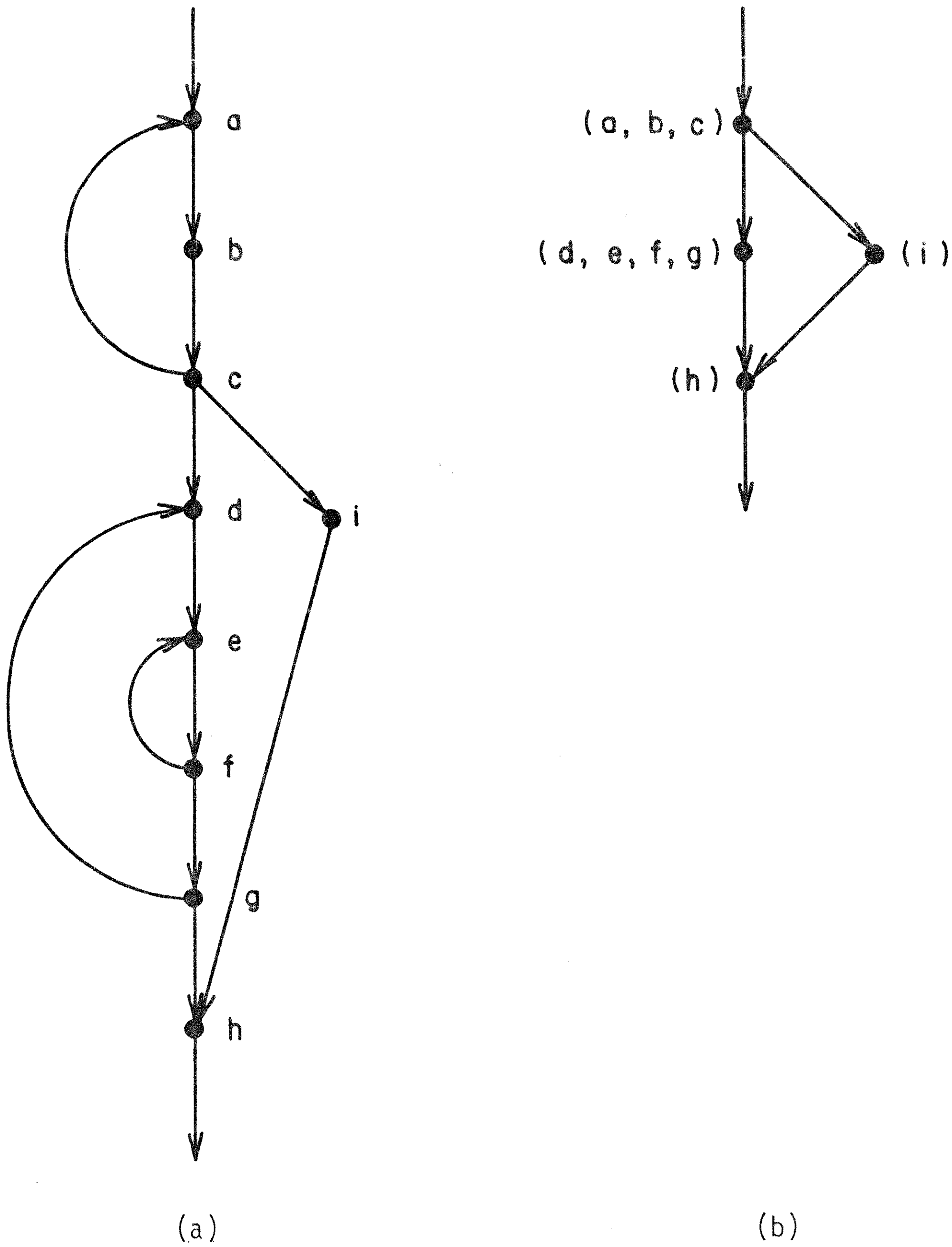
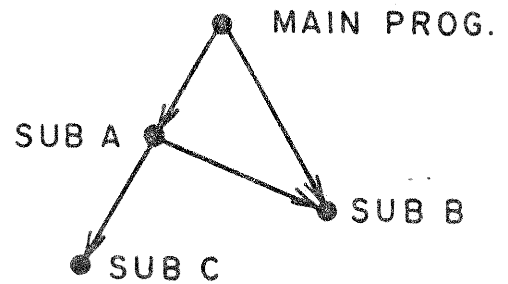


Fig. 2.5

(a) Segment of a flow graph with cycles; (b) Segment of a flow graph, derived from (a), in which nodes represent strongly connected components.

```
(MAIN PROGRAM)
   .
   .
   .
 CALL SUB A (- - -)
   .
   .
   .
 CALL SUB B (- - -)
   .
   .
   .
 CALL SUB A (- - -)
   .
   .
   .
 END
 SUBROUTINE SUB A (- - -)
   .
   .
   .
 CALL SUB B (- - -)
   .
   .
   .
 CALL SUB C (- - -)
   .
   .
   .
 END
 SUBROUTINE SUB B (- - -)
   .
   .
   .
 END
 SUBROUTINE  SUB C (- - -)
   .
   .
   .
 END
```

(a)



(b)

Fig. 2.6

(a) FORTRAN program with subroutine calls; (b) Call graph for program in (a)

arbitrary graphs is almost certainly going to be exponential in the size of the graph as measured by $|N|$ and $|E|$. Other questions in this group can be answered by very fast algorithms; for instance question 5 can be answered by an algorithm that has time complexity $O(|N|+|E|)$.*

Association of information describing data actions with the nodes of a flow graph makes it possible to analyze sequences of data actions, called data flow. For example, consider a particular variable, say X, and label each node with the symbol r, d, or e according as X is referenced (its value is fetched from memory as in $Y \leftarrow X+1$), defined (a value is assigned to X as in $X \leftarrow Y+1$), or X is not referenced or defined. If more than one action on the variable takes place as in $X \leftarrow X+1$ then an appropriate sequence of action symbols, rd, is attached to the node. An example is shown in Fig. 2.7: every path in this labeled graph corresponds to a sequence of data actions as illustrated below for the variables X and Y.

Path	Data Actions	
1→2→3→5→7	e d r e	(X)
	e e r e e	(Y)
1→2→3→4→6→4→7	e d r r r d r e	(Y)
	e e r r e r e	(Y)

Looking at the data flow described on the right it is evident that Y must be assigned a value before entry to the program since there are paths on which the first action is r. It also appears that X need not be assigned a value before entering the path since it is defined before any reference; further consideration shows that X does not need to be assigned a value before entering any path starting at n_o.

Analysis of data flow can provide answers to the following questions about a program:

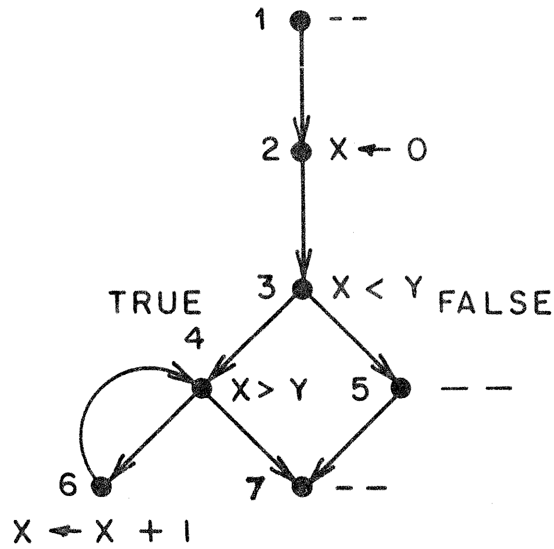
1) Are undefined variables referenced?
2) Are there unnecessary definitions of values?
3) Which parameters in a procedure call need to be assigned values before entry?
4) Which parameters in a procedure call may have altered values upon return?

---

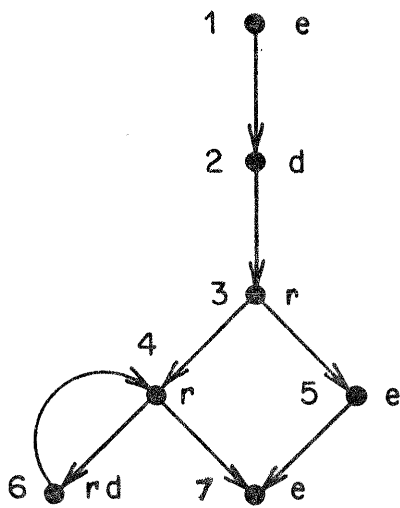
*Let f $(|N|,|E|)$ represent the time to execute the algorithm as a function of $|N|$ and $|E|$, then time complexity $O(|N|+|E|)$ means

$$\lim_{|N|+|E|\to\infty} \left(\frac{f(|N|,|E|)}{|N|+|E|} \right) = k, \quad k \text{ a constant unequal to zero.}$$ Time

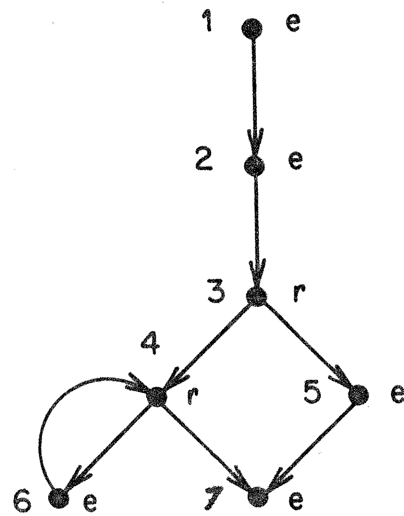
complexity $O(|N|+|E|)$ implies that the approximation $f(|N|,|E|) \cong k \times (|N|+|E|)$ may be used for large $|N|+|E|$.



(a)



(b)



(c)

Fig. 2.7

(a) Flow graph with nodes numbered for identification and relevant statements corresponding to nodes indicated; (b) Flow graph, derived from (a), with data actions on X indicated; (c) Flow graph derived from (a) with data actions on Y indicated.

Analysis of data flow can detect programming errors. If the first data action on a path is r then it is likely that a data initialization has been omitted, or the reference action is incorrect because the variable was misspelled, or the referencing statement is in the wrong place, and so forth. Similarly if a d is followed by a d without an intervening r then a variable may have been misspelled causing the redundant definition or causing the omission of the intervening reference. Analysis of data flow can also assist in the global optimization of programs, in providing documentation aids, and in program modification.

It should be evident from this that algorithms for manipulating graphs and for extracting implied relationships in graphs have many applications in program analysis. However, not all useful abstractions require graphs. In some applications, for example, sets of variables are sufficient: the set of variables declared as formal parameters to a procedure should be a subset of the set of variables used in a procedure; sets of variables which are equivalent in the sense that all variables in the set represent the same memory location are important for a consideration of aliasing. Thus algorithms for set operations are also important for program analysis. Here, however, our attention is directed at graph algorithms.

3. ALGORITHMS. A flow graph contains explicit and implicit information. Explicit information is information associated with a node or edge which is independent of information at other nodes or edges. Thus it is local information about the program, determined at a cost that is independent of the program size. Examples of explicit information are: the statement type; the list of variables appearing in a statement; the branch condition on an edge, for example the fact that $A \geq 0$ is true if the edge is traversed during execution. Implicit information may be associated with a node or edge also, or it may be associated with a larger structure including the entire flow graph: it is dependent on information at more than one node or edge, perhaps all of them. Thus implicit information is global information that may, and usually does, depend on the program's size. Examples of implicit information are: the set of statements which can be reached on all paths from a given statement; the set of variables on which the first data action will be definition on all paths from a given statement; the set of statements which are on all paths to a particular statement. Explicit information is collected at the time the abstract model of the program is created. It is, in fact, part of the model itself. Implicit information is derived from the model. It is the derivation of this implicit information that is the focus of attention in the subsequent discussion.

Two mechanisms are frequently used for deriving information from a flow graph: one consists of performing graph transformations, [6,7] the other consists of performing a search on a graph [8,9]. When graph transformations are used a flow graph $G(N,E,n_0)$ is transformed into another flow graph $G(N',E',n_0')$: the transformation is not only one of structure but also one of information attached to nodes and edges. This approach generally consists of a sequence of transformations, each being an elementary transformation in some sense. Through an appropriately chosen sequence of transformations global information about the program can be collected and distributed to appropriate nodes. A search consists of moving over the graph in a systematic manner dictated by the relationships between nodes implied by the edges. During the search global information about relationships can be collected. Transformations may be used to assist the search, and transformations may be used in place of a search.

Two kinds of search on a flow graph are common: depth-first and breadth-first. A depth-first search is defined by the following algorithm:

Algorithm (DFS)

1. Push the entry node on a stack and mark it (this is the first node visited, nodes are marked to prevent visiting them more than once).
2. While the stack is not empty do
   2.1 If there is an edge from the node at the top of the stack to an unmarked node then push the unmarked node on the stack and mark it else pop the stack.
3. Stop.

A breadth-first search is defined by the following algorithm:

Algorithm (BFS)

1. Put the entry node on a queue and mark it.
2. While the queue is not empty do
   2.1 If there is an edge from the node at the head of the queue to an unmarked node then add the unmarked node to the end of the queue and mark it else remove the node at the head of the queue
3. Stop.

A search defines a numbering of the nodes determined by the order in which they are visited. Two numberings of importance associated with DFS are preorder numbering and postorder numbering: preorder numbering corresponds to the order in which the nodes are first visited in a depth-first search and postorder numbering corresponds to the order in which they are last visited in a depth-first search.* These numberings are illustrated in Fig. 3.1. When a graph is a tree preorder numbering assures that every node has a higher number than its parent and postorder numbering assures that every node has a lower number than its parent. This is illustrated in Fig. 3.2. When an arbitrary graph has a preorder numbering the presence of an edge (v_i,v_j) such that $v_i > v_j$ implies the graph is not a tree but the converse is not true. When an arbitrary graph has a postorder numbering the presence of an edge (v_i,v_j) such that $v_i < v_j$ implies the presence of a cycle; removal of all edges with this property transforms the graph into an acyclic directed graph, but not necessarily a tree. It is important to recognize that a preorder or postorder numbering can be done quickly. The time required for a depth-first search increases linearly with the number of edges in a connected graph: each edge is traversed once in a forward direction and once in the backward direction, thus the time complexity for the DFS algorithm is $O(|E|)$.

---

*There has been some confusion in the literature with this terminology. Our definition corresponds to recent usage [1] but differs from Knuth [10].

(page 14 is unavailable for this report)

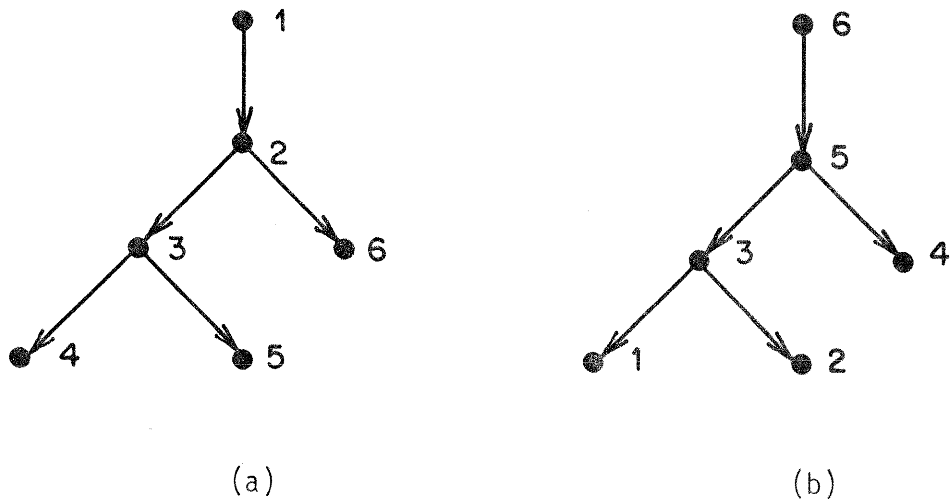


Fig. 3.2

(a) Tree with nodes numbered in preorder, each node has a higher number than its parent; (b) Tree with nodes numbered in postorder, each node has a lower number than its parent.

It is useful for a number of applications to know whether a pair of nodes can be on a path. This gives rise to the following problems: given a node, u, determine all nodes v such that there is a path from u to v (symbolically this is expressed as the set $S(u) = \{v | u \overset{*}{\rightarrow} v\}$); and given a node, u, determine all nodes v such that there is a path from v to u (symbolically this is expressed as $P(u) = \{v | v \overset{*}{\rightarrow} u\}$). Any algorithm for solving the first problem can be used to solve the second problem simply by reversing the directions of the edges. The reflexive and transitive closure of a graph is defined as the set of all ordered pairs of nodes (u,v) such that there is a path (including a path of zero length) from u to v (symbolically this is $\{(u,v) | u \overset{*}{\rightarrow} v\}$). If paths of zero length are excluded then the set is called the transitive closure (symbolically this is $\{(u,v) | u \overset{+}{\rightarrow} v\}$).

An algorithm by Warshall [11] for computing transitive closure is based on matrix multiplication and has time complexity $O(|N|^3)$. It is not difficult to see that the transitive closure could be computed by performing $|N|$ depth-first searches, one search from each node. This approach would require a time proportional to $|N||E|$. Of course $|E| \leq |N|^2$ so in the worst case the time required would also be proportional to $|N|^3$. But for programs it is more common that $|E| \leq k|N|$ where k is a constant and so the time would be proportional to $|N|^2$.

A clever idea described by Schnorr [12] may allow further improvement of the computation time for transitive closure. Consider a node v of $G(N,E,n_0)$ and suppose $v \overset{*}{\rightarrow} u_1,\ v \overset{*}{\rightarrow} u_2, \ldots, v \overset{*}{\rightarrow} u_k$ where $k = \lceil |N|/2 + 1 \rceil$. Now consider a graph $G'(N,E')$ derived from G by simply reversing the direction of all of the edges. Let w be a node and suppose in G: $w \overset{*}{\rightarrow} r_1,\ w \overset{*}{\rightarrow} r_2, \ldots, w \overset{*}{\rightarrow} r_k$ where k is defined as before. It is easy to see that the sets $\{u_1, u_2, \ldots, u_k\}$ and $\{r_1, r_2, \ldots, r_k\}$ must have at least one node in common since $2 \times \lceil |N|/2 + 1 \rceil$ equals $|N| + 2$ or $|N| + 3$ according as $|N|$ is even or odd. This idea is the basis of an algorithm which uses breadth-first search to compute transitive closure but terminates the search from a node after $\lceil |N|/2 + 1 \rceil$ nodes have been reached, unless it terminates earlier because no more nodes can be reached. Then the idea just described is applied to complete the computation of the transitive closure. The interesting property of this algorithm is that its <u>expected</u> time for execution is $O(|N| + |E|^*)$ where $|E|^*$ is the expected number of edges. The model used for computing the expected values consists of an ensemble of random graphs with $|N|$ nodes and for any pair of nodes u and v there is constant probability p for an edge (u,v). Unfortunately, this ensemble probably does not match the ensemble for real programs very well.

A determination of strongly connected components is useful in some aspects of program analysis. A strongly connected component, SCC, of a flow graph $G(N,E,n_0)$ is a subset of N defined as follows: for every pair of nodes u, v in the SCC there is a path from u to v and v to u in G, and no more nodes may be added to SCC preserving this property. In general a graph may have more than one SCC. Once the SCC's of G have been determined it is possible to make a transformation $G(N,E,n_0) \rightarrow G'(N',E',n_0)$ where the elements of N' are the SCC's of G and the elements of E' represent paths between components implied

by the elements of E: the idea is illustrated in Fig. 2.5. The flow graph G' is acyclic and so a partial ordering of the nodes is possible. In particular, if the nodes of G' are numbered in postorder then on every path, $n_0 \overset{*}{\rightarrow} n$, $n \in N'$, the numbering of the nodes will be in decreasing order. This partial ordering is useful in data flow analysis. A fast algorithm for computing SCC's has been described by Tarjan [13]. In this algorithm a pair of numbers, preorder(n) and lowlink(n), is associated with each node n; preorder(n) is the preorder number defined earlier, lowlink(n) is defined by

$$\text{lowlink}(n) = \min_{S(n)}[\text{preorder}(v)],\ S(n) = \{v \mid n \overset{*}{\rightarrow} v\}.$$

This numbering is illustrated in Fig. 3.3. Tarjan shows that a pair of nodes are in the same SCC if and only if they have the same lowlink number and that the lowlink numbers can be computed using a depth-first search. The time complexity for Tarjan's algorithm applied to a flow graph is $O(|N|+|E|)$.

In testing a program the notion of coverage or completeness of a set of tests is important [14,15]. One measure of test coverage is the percentage of nodes executed or edges traversed in the flow graph. Randomly selected input data for a set of tests will give poor coverage, therefore some care in choosing test data is necessary. In attempting to choose the test data carefully the question of whether it is possible to execute a given set of nodes in a test arises. This question is closely related to the following one about a flow graph of the program: given the flow graph $G(N,E,n_0)$ and a set N', $N' \subseteq N$, is there a path $n_0 \overset{*}{\rightarrow} n$, $n \in N$, which includes every node in N'. Note that if the answer to the graph question is no, then the answer to the question about the program is certainly no; however, if the answer to the graph question is yes one cannot infer that the answer to the question about the program is yes because it may not be possible to satisfy all of the branch conditions as illustrated in Fig. 3.4. Gabow, Maheshwari, and Osterweil [5] have described an algorithm for answering the flow graph question. The idea is based on a consideration of an acyclic directed graph which would result from an arbitrary flow graph if one were to replace all SCC's by single nodes. Note that if any node v in N', the set of nodes to be included in the path, is in a SCC then a path to any member of that SCC can be extended to include v. Thus the original graph question only needs to be asked about an acyclic directed graph. In presenting the algorithm we use the word "frontier" for the set of entry nodes (initially the frontier is $\{n_0\}$) removing a node from the graph implies all edges leaving the node are also removed. Here is the algorithm:

```
until the frontier is empty or the frontier contains
      more than one node in N' do
   if the frontier is a singleton then remove this node
   else remove a frontier node that is not in N';
stop.
```

If the graph is empty when this algorithm terminates then there is a path $n_0 \overset{*}{\rightarrow} n$ which includes all nodes of N', and if the graph is not empty then there is no such path. This algorithm is illustrated in Fig. 3.5. The time complexity for this algorithm is $O(|E|)$ and since the time complexity for getting

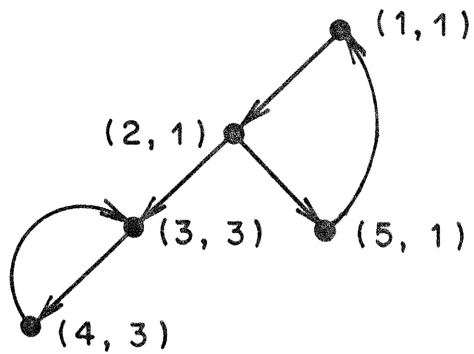


Fig. 3.3

Flow graph with nodes labeled (i,j) where i is the preorder number and j is the lowlink number. All nodes with the same lowlink number are in the same strongly connected component and nodes with different lowlink numbers are in different strongly connected components.

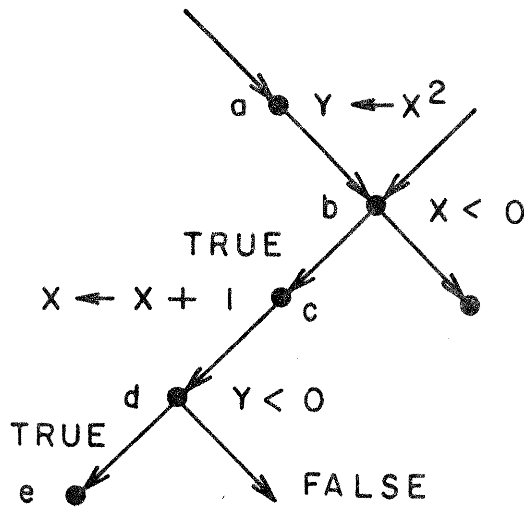


Fig. 3.4

Flow graph with nodes labeled, and associated program statements indicated. The path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ is unexecutable since the computation at a implies $y \geq 0$ and traversing the edge (d,e) implies $y < 0$.

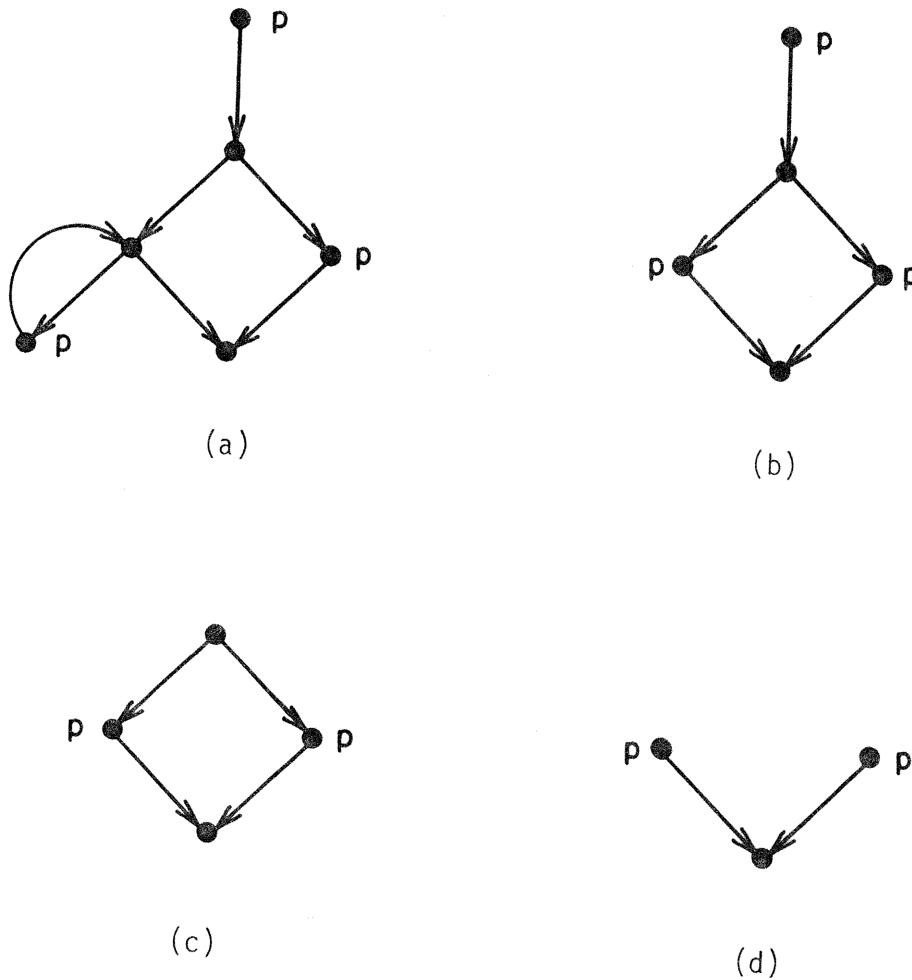


Fig. 3.5

Illustration of steps in the Maheshwari, Gabow, Osterweil algorithm to determine existence of a path through a set of nodes. Nodes to be on the path are marked by p in (a). Nodes in the same SCC are collapsed into a single node resulting in (b). The frontier node is removed resulting in (c). The frontier node in (c) is removed resulting in (d). Now there are two nodes on the frontier both of which are to be on the path and the algorithm stops. Since the graph is not empty at this point the conclusion is there is no path in the graph (a) which includes all nodes marked p.

the SCC's by Tarjan's algorithm is also $O(|E|)$ it follows that the overall time complexity for answering the original graph question is $O(|E|)$.

Data flow analysis has applications in global program optimization [16,17], error detection [9], discovery of unexecutable paths [18,19], and detection of deadlock [20]. A basic problem in data flow analysis is the so-called live variable problem. This problem can be stated as follows: given a flow graph $G(N,E,n_0)$ with the nodes labeled to show the reference (r) actions and definition (d) actions on a variable x, and given n, $n \in N$, does there exist a path from n such that the first action on x is r. If the answer to this question is yes, then x is said to be live at n, otherwise it is dead at n. If only one variable and one node were involved the easiest way to answer this question would be to conduct a depth-first search from n. However, the normal situation is that this question is to be answered for many variables at all nodes of the flow graph. A number of algorithms for treating this problem have been described in the literature [6,7,8,16,21]. A particularly simple and effective algorithm is the one due to Hecht and Ullman [8]. The main ideas are as follows. We associate three sets with every node: at node n these sets are ref(n), def(n), live(n). The sets are initialized as follows for all n, $n \in N$:

1. $\text{ref}(n) = \{V \mid V \text{ a variable referenced at } n\}$
2. $\text{def}(n) = \{V \mid V \text{ a variable defined at } n\}$

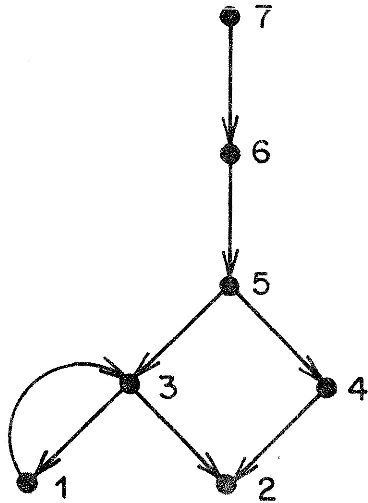
   (for simplicity we assume here that a variable is not referenced and defined at the same node.)
3. $\text{live}(n) = \emptyset$ (the empty set)

After this initialization ref(n) and def(n) are not changed, but live(n) is modified by applying the following formula iteratively to the nodes of $G(N,E,n_0)$:

$$\text{live}(n) = \bigcup_{k \in S(n)} \left(\left(\text{live}(k) \cap \neg\, \text{def}(k) \right) \cup \text{ref}(k) \right)$$

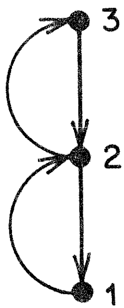
where $S(n) = \{j \mid (n,j) \in E\}$, the set of successors of n. Two examples of the application of this formula are shown in Fig. 3.6. Hecht and Ullman have shown [8] that if this formula is applied iteratively to the nodes of $G(N,E,n_0)$ in postorder, the convergence is quite rapid. In practice the number of iterations can be expected to be less than four or five. The application of this algorithm to the detection of uninitialized variables in entire programs has been thoroughly discussed by Fosdick and Osterweil [9].

There is an interesting connection between an important problem in program analysis and non-linear optimization. I mentioned earlier that a path in a program flow graph may not represent a sequence of statements that could actually be executed: Fig. 3.4 illustrates this situation. Suppose that we are given a path $n_0 \overset{*}{\rightarrow} n$ in a flow graph and we wish to determine whether the path can be executed. To do this imagine moving along the path writing down the predicates that must be satisfied at every branch. In doing this we must take into account changes in values of variables caused by assignments so that a particular variable name represents the same value in every predicate. A necessary and sufficient condition for the path to be executable is that the



n	ref(n)	def(n)	live(n) (initial)	live(n) (final)
1	x	∅	∅	x,y
2	y	∅	∅	∅
3	x,y	∅	∅	x,y
4	z	∅	∅	y
5	x	∅	∅	x,y,z
6	∅	y	∅	x,y,z
7	∅	∅	∅	x,z

(a)



n	ref(n)	def(n)	live(n) (initial)	live(n) (after 1 iter.)	live(n) (final)
1	x	∅	∅	z	x,y,z
2	z	∅	∅	x,y,z	x,y,z
3	y	∅	∅	x,y,z	x,y,z

(b)

Fig. 3.6

(a) Illustration of live sets, for given sets ref(n) and def(n) on a graph. Nodes are numbered in postorder and just one application of the live formula to the nodes in postorder produces the final live sets.
(b) Another illustration of live sets. In this case two applications of the live formula to the nodes in postorder is required to obtain the final live sets.

system of predicates written down is consistent; that is, there must exist an assignment of values to variables appearing therein such that every predicate is satisfied. Now this issue of consistency is exactly the same one that arises in trying to determine whether there is a feasible region defined by the constraints in a non-linear (or linear) optimization problem. This problem is difficult and there are no really good algorithms for dealing with it. Clarke has discussed the problem and some experience in attempting to solve it in a recent paper [22].

4. CONCLUSION. Recent work has provided a number of algorithms which have important applications in the analysis of computer programs. While much work remains to be done in the development of these algorithms, we are now in a position to build some important program analysis tools based on these algorithms. Such tools could be used for program optimization, error detection, testing, and documentation.

These tools should be organized into a library that is easy to use. This will take careful planning. Consistent patterns of use must be developed, and portability, and adaptability to various language dialects must be taken into account. It is a large and difficult challenge, but one which we should accept.

Bibliography

1. Aho, Alfred V.: Hopcroft, John E.; and Ullman, Jeffrey D. The Design and Analysis of Computer Algorithms. Addison-Wesley (1974).

2. Berge, Claude. Graphs and Hypergraphs, North-Holland (1973).

3. Biggs, Norman 1.; Lloyd, E. Keith; and Wilson, Robin J. Graph Theory 1736-1936. Clarendon Press, Oxford (1976).

4. Goldstine, Herman H. and von Neumann, John. Planning and coding problems for an electronic computing instrument. In John von Neumann Collected Works, Volume 5, Abraham H. Taub, Ed., Pergamon Press (1961).

5. Gabow, Harold N.; Maheshwari, Sachindra N.; and Osterweil, Leon J. On two problems in the generation of program test paths. Proc. IEEE Trans. on Soft. Eng. SE-2,3 (Sept. 1976), 227-231.

6. Graham, Susan L.; and Wegman, Mark. A fast and usually linear algorithm for global flow analysis. JACM 23,1 (Jan. 1976), 172-202.

7. Allen, F.E.; and Cocke, J. A program data flow analysis procedure. CACM 19,3 (March 1976),137-147.

8. Hecht, Matthew S.; and Ullman, Jeffrey D. A simple algorithm for global data flow analysis. SIAM J. Computing 4 (Dec. 1975), 519-532.

9. Fosdick, Lloyd D. and Osterweil, Leon J. Data flow analysis in software reliability. ACM Comp. Surveys 8,3 (Sept. 1976), 305-330.

10. Knuth, Donald E. The Art of Computer Programming, Vol. 1. Addison-Wesley (1968).

11. Warshall, S. A theorem on boolean matrices. JACM 9,1 (Jan. 1962), 11-12.

12. Schnorr, C.P. An algorithm for transitive closure with linear expected time. In Lecture Notes in Computer Science, v. 48, G. Goos and J. Hartmanis, Eds., Springer-Verlag (1977).

13. Tarjan, Robert E. Depth-first search and linear graph algorithms. SIAM J. Computing 1,2 (June 1972), 146-160.

14. Brown, John R. Getting better software cheaper and quicker, in Practical Strategies for Developing Large Software Systems, Ellis Horowitz, Editor. Addison-Wesley (1975).

15. Huang, J.C. An approach to program testing. ACM Comp. Surveys 7,3 (Sept. 1975), 113-128.

16. Schaefer, J.C. A Mathematical Theory of Global Program Optimization. Prentice-Hall (1973).

Bibliography cont'd

17. Ullman, Jeffrey D. and Aho, Alfred V. The Theory of Parsing, Translation, and Computing: Volume II. Prentice-Hall (1973).

18. Osterweil, Leon J. The detection of unexecutable program paths through static data flow analysis. Tech. Rept. 110, Dept. of Computer Science, University of Colorado, Boulder, CO. 80309 (May 1977).

19. Bollacker, Lee A. An algorithm for detecting unexecutable paths through program flowgraphs. Tech. Rept. 112, Dept. of Computer Science, University of Colorado, Boulder,CO. 80309 (Jan. 1978).

20. Saxena, Ashok. Static detection of deadlocks. Tech. Rept. 122, Dept. of Computer Science, University of Colorado, Boulder,CO. 80309 (Dec. 1977).

21. Kennedy, Kenneth. Node listings applied to data flow anlaysis. Proc. 2nd ACM Symposium on Principal of Programming Languages. Palo Alto, CA. (Jan. 1975).

22. Clarke, Lori A. A system to generate test data and symbolically execute programs. IEEE Trans. on Software Engineering SE-2 (Sept. 1976), 215-222.