IBM Research

# GRAPH-THEORETIC CONSTRUCTS FOR PROGRAM CONTROL FLOW ANALYSIS

F. E. Allen/J. Cocke

July 11, 1972

RC 3923

# GRAPH-THEORETIC CONSTRUCTS FOR PROGRAM CONTROL FLOW ANALYSIS

by

F. E. Allen
J. Cocke

IBM*
Thomas J. Watson Research Center
Yorktown Heights, New York

ABSTRACT: An approach to the static global analysis and codification of program control flow based on the "interval" partitioning of control flow graphs is given. Topics covered include dominance relations, intervals and node splitting. Also included are several new algorithms and some results of analyzing the control flow of programs by the methods described.

Key Words: program flow graph, directed graph, program analysis, object code optimization

*This work was partially supported by the National Science Foundation under grant NSF-GJ-95 at the Courant Institute of Mathematical Sciences, New York University.

RC 3923 (#17789)
July 11, 1972
Computer Sciences

## INTRODUCTION

Program control flow analysis exists in one form or another in any optimizing compiler and has been described in several papers. An early paper by Prosser [16] described the use of Boolean matrices (or, more particularly, connectivity matrices) in flow analysis. The use of "dominance" relationships in flow analysis was first introduced by Prosser and much expanded by Lowry and Medlock [12]. References [12 and 13] describe compilers which use various forms of control flow analysis for optimization. Some more recent developments in the area are reported in [7] [9] and [10].

The underlying motivation for all the different types of control flow analysis is the need [3] to codify the flow relationships in the program. The codificatior may be in connectivity matrices, in predecessor-successor tables, in dominance lists, etc. Whatever the form, the purpose is to facilitate the determination of the flow relationships; in other words to facilitate answering such questions as: Is this an inner loop?, if an expression is removed from the loop, where can it be correctly and profitably placed?, which variable definitions can affect this use?

In this paper the basic control flow relationships are expressed in a directed graph. Various graph constructs are then found and shown to codify interesting global relationships. The extraction of branch-target paths from a program is not considered. Language dependent questions are avoided and little is said about some of the practical aspects of implementing the ideas presented here. We have tried to present the material at a level which is understandable to both the practitioner and the theoretician.

The first part of the paper is an extensive revision of a previously published paper [2] but includes a previously unpublished algorithm for finding back-dominators. The part of the paper starting with "Node Splitting" gives details of the material presented in references [6 and 8].

The first section of the paper, "Basic Concepts," is primarily a catalog of relevant information about directed graphs and their use in expressing control flow relationships. In the second section of the paper "Dominance Relationships" are defined in terms of the basic concepts introduced in the first section. Many of the concepts presented in these sections have appeared in the literature before.

The third section, "Intervals," discusses a graph construct previously described in [2, 3, 5 and 7]. In this section intervals are defined, an algorithm is given for their construction, and their properties are given. Also discussed are procedures for finding other graph constructs in terms of the interval constructs. The fourth section, "Partitioning Graphs by Intervals," describes a hierarchical sequence of graph partitions by means of intervals.

The fifth section, "Node Splitting," describes a technique for modifying those graphs not fully amenable to interval analysis in order to generate equivalent graphs which can be analyzed. The approach described here is based on work done in cooperation with Dr. Raymond Miller [6] of IBM and Mr. Richard Sites of Stanford.

The last section before the summary reports on some results of applying the described control flow analysis to some FORTRAN programs.

## BASIC CONCEPTS

In order to precisely define the basis for the control flow analysis which will be described later, some basic concepts are given in this section. Some of these are taken directly from graph theory [4], some have appeared in the literature before and others are introduced with this paper.

A directed graph, G, can be denoted by G = (B,E) where B is the set of nodes (blocks) $\{b_1, b_2, \ldots, b_n\}$ in the graph and E is the set of directed edges $\{(b_i, b_j), (b_k, b_\ell), \ldots\}$. Each directed edge is represented by an ordered pair $(b_i, b_j)$ of nodes (not necessarily distinct) which indicate that a directed edge goes from node $b_i$ to node $b_j$. Another common way of viewing a directed graph G is as a set of nodes B and a successor function $\Gamma_G^1$ which maps G into G such that $\Gamma_G^1(b_i) = \{b_j \mid (b_i, b_j) \in E\}$. We call this set the set of immediate successors of a node. The inverse of the successor function $\Gamma_G^{-1}$ gives the immediate predecessors of a node: $\Gamma_G^{-1}(b_j) = \{b_i \mid (b_i, b_j) \in E\}$. A directed graph is depicted in Fig. 1. The nodes here and throughout the paper are identified by arbitrarily assigned numbers.

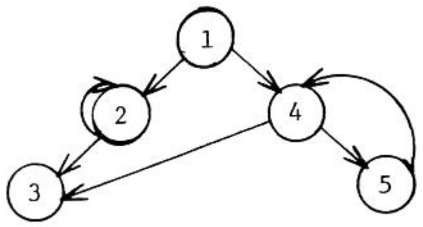


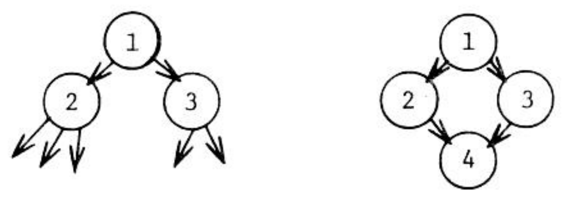
Figure 1

The graph in figure 1 has B = {1,2,3,4,5} , E = {(1,2), (2,2), (2,3), (1,4), (4,3), (4,5), (5,4) , $\Gamma_G^1(2) = \{2,3\}$ , $\Gamma_G^{-1}(3) = \{2,4\}$, etc.

A directed graph is <u>connected</u> if any node in the graph can be obtained (reached) from any other node by successive applications of $\Gamma_G^1$ and/or $\Gamma_G^{-1}$. We will assume throughout this paper that the graphs being discussed are both directed and connected. Before introducing more graph concepts, the relevance of graphs to program control flow is introduced.

A <u>basic block</u> is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). It may of course have many predecessors and many successors and may even be its own predecessor and successor. Program entry blocks have predecessors that are not in the program; program terminating blocks never have successors in the program.

An <u>extended basic block</u> (EBB) is a sequence of program instructions each of which, with the exception of the first instruction, has one and only one immediate predecessor and that predecessor precedes it (though not necessarily immediately) in the extended basic block. An extended basic block is therefore a set of basic blocks $EBB = \{(b_1, b_2, \ldots b_n\}$ such that for any $b_j$, $j \neq 1$, $\Gamma^{-1}(b_j) = \{b_i\}$ for one and only one $i$, $1 \leq i < j$. Thus, the nodes in subgraph (a) in Figure 2 can be grouped to form either $EBB_1 = \{1,2,3\}$ or $EBB_2 = \{1,3,2\}$. Node 4 in subgraph (b) of Fig. 2 must be the first block of an extended basic block because it has more than one predecessor.



(a) Figure 2 (b)

Extended basic blocks are of interest for certain types of analyses because such a block can be subjected to an essentially linear analysis of its internal dependencies. Since the control flow relationships between the basic blocks in a EBB exhibit a tree structure, the blocks in an EBB can be ordered so that a stacking mechanism can be used when determining internal data dependency relationships. Although nearly all of the concepts to be discussed in this paper can be applied to nodes which represent extended basic blocks rather than blocks, it is easier to visualize many of these concepts in terms of basic blocks. For this reason, extended basic blocks will not be further discussed in this paper.

A <u>control flow graph</u> is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths. Everything that is said about directed graphs in this paper holds for control flow graphs.

A <u>subgraph</u> of a directed graph, $G = (B,E)$, is a directed graph $G' = (B',E')$ in which $B' \subset B$, $E' \subset E$, $G \cap G' = G'$ and $G \cup G' = G$. It follows that the successor function $\Gamma^{1}_{G'}$, defined for $G'$ must "stay within" $G'$; that is for $b'_i \in B'$, $\Gamma_{G}'(b'_i) = \{b'_j \mid (b'_i,b'_j) \in E'$ . Consider the directed graph, $G$, in Figure 3.

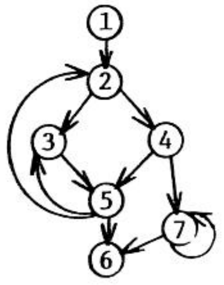
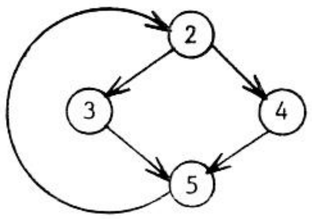


Figure 3

One of the many subgraphs in G is $G' = (B', E')$ in which $B' = \{2,3,4,5\}$ and

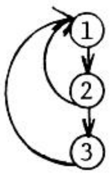
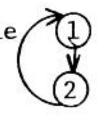
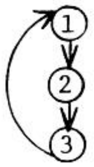
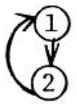
E' = {(2,3), (2,4), (3,5), (4,5), (5,2)}. G' can be depicted by:



A path in a directed graph is a directed subgraph, P, of ordered nodes and edges obtained by successive applications of the successor function. It is expressed as a sequence of nodes $(b_1, b_2, \ldots b_n)$ where $b_{i+1} \in \Gamma_P^1(b_i)$. The edges are implied: $(b_i, b_{i+1}) \in E$. The nodes and the implied edges are not necessarily unique. A path in the graph in Figure 3 is (2,3,5,3,5,2,4). A node, q, is said to be a successor of a node, p, if there exists some path $P = (b_1, \ldots b_n)$ for which $b_1 = p$ and $b_n = q$. In the same situation p is said to be a predecessor of q. It should be noted that a node can be both a predecessor and a successor of another node: $P_1 = (p, \ldots, q)$ and $P_2 = (q, \ldots, p)$.

A closed path or cycle (here we deviate from standard terminology [4]) is a path in which $b_n = b_1$. The cycle is a simple cycle if, with the exception of $b_n$, the nodes in the cycle are distinct; otherwise it is a composite cycle. Consider the graph in Fig. 3: it has the following simple cycles (3,5,3), (5,3,5), (2,3,5,2), (3,5,2,3), (5,2,3,5), (2,4,5,2), (4,5,2,4), (5,2,4,5), (7,7). One of the composite cycles is (2,3,5,3,5,2). Since it will usually be uninteresting to consider cycles containing the same nodes and edges but in a different order, we will generally select a first node and describe the cycle relative to that node. A 2-cycle is a cycle with two edges such as (3,5,3); an n-cycle is a cycle with n edges.

A prime cycle is a simple cycle no subset of whose nodes in the full graph is also a simple cycle. Thus


has simple cycles

and

and prime cycle


The length of a path is the number of edges in the sequence. More formally, a distance function is defined such that for any path $P = (b_1, b_2, \ldots, b_n)$, $\delta(P) = n-1$. Since the shortest path $\delta_{min}$ between two points p and q is often of interest it will now be defined: $\delta_{min}(p,q) = \text{MIN}(\delta(P_1), \delta(P_2) \ldots)$ for all $P_i = (p, \ldots, q)$. The shortest path then is the $P_i$ for which $\delta(P_i) = \delta_{min}(p,q)$.

A strongly connected region of a directed graph is a directed subgraph in which there is a path from any node in the subgraph to any other node. It immediately follows from this definition that every node lies on at least one closed path and is, therefore, its own predecessor and its own successor. Closed paths (cycles) are, therefore, a special kind of strongly connected region -- one which has a strict ordering. A strongly connected region R of a directed graph is maximal if it is not contained in another strongly connected region and does not intersect another strongly connected region. A properly nested set of strongly connected regions is a partially ordered set $d = \{R_1, R_2 \ldots R_n\}$ such that for $i < j$ either $R_i \cap R_j = \phi$ or $R_i \cap R_j = R_i$ i.e., either $R_i$ and $R_j$ are disjoint or $R_j$ covers $R_i$.

The use of a nested set of strongly connected regions in control flow analysis for optimization was first suggested in [1]. In that approach to control flow analysis, a set, D, of disjoint sets of nested strongly connected regions is found:

$$D = \left\{\{R_1, R_2, \ldots, R_n\} \quad , \quad \{R'_1, R'_2, \ldots, R'_n\}, \ldots\right\}$$

or, for the sake of brevity, $D = \{d, d', \ldots\}$. Each $R_n$ is a maximal, strongly connected region which thereby assures that sets of nested strongly connected regions are disjoint. We will now consider some of the properties of the above construct in a directed graph, G:

1. D does not necessarily cover G. If there are nodes in G which are not in any strongly connected region, then they will not be in D.

2. Each $d \in D$ is partially ordered.

3. D is unordered.

4. If a node, p, is an element of a strongly connected region, it is in one and only one d. For $p \in d$ where $d = \{R_1, R_2, \ldots, R_n\}$ then $p \in R_n$ and may be an element of several nested $R_i$.

Since much of the control flow analysis involves exposing relationships between nodes in the control flow graph the construct, D, codifies several useful relationships. It has however several limitations: It does not establish an ordering on the total graph and by the very nature of a general strongly connected region, there is no ordering relationship on the nodes within the region other than that given by the immediate successor-predecessor relationships.

## DOMINANCE RELATIONSHIPS

Several interesting and useful constructs can be established from "back dominance" and "forward dominance" relationships. Before defining these relationships, two special kinds of nodes must be defined. A node in a directed graph, G, which has no successors in G is called a terminal or exit node. Thus, letting x denote an exit node, $\Gamma_G^1(x) = \phi$. An entry node, e, is a node in the program control flow graph, C, which contains a program entry point. Several of the constructs about to be described depend upon having only one such node in the control flow graph. An arbitrary initial entry node $e_0$ is introduced into the control flow graph as an immediate predecessor of all entry nodes:

$$\Gamma_C^1(e_0) = \{e_i \mid e_i \text{ is an entry node}\} \quad \text{and} \quad \Gamma^{-1}{}_C\,(e_0) = \phi$$

Since $e_0$ essentially represents the set of all external program predecessors of the entry points, the control flow graph has not been invalidated. Having modified the control flow graph to contain $e_0$, it is possible to view the control flow graph as a directed graph with one initial node where an initial node is a node with no predecessors.

Any reference to a graph in the remainder of this paper will be to a connected directed graph with a single entry node, $e_0$, and a set of exit nodes $X = \{x_1, x_2, \ldots\}$. Having established entry and exit nodes, we can now define the dominance relationships which exist in a directed graph and are of interest in control flow analysis. (For information of their role in optimization, the reader is referred to [12]).

A node, $b_i$, is said to <u>back dominate</u> or <u>predominate</u> a node, $b_k$, if $b_i$ is on every path from $e_0$ to $b_k$. Let $\mathcal{P} = \{P \mid P = (e_0, \ldots, b_k)\}$. Then the <u>set of back dominators</u>, $BD(b_k)$, of $b_k$ consists of all of the nodes, other than $b_k$ itself, which are on all paths from $e_0$ to $b_k$. In other words

$$BD(b_k) = \{b_i \mid b_i \neq b_k \text{ and } b_i \in \bigcap \mathcal{P}\}.$$

The <u>immediate back dominator</u> $b_i$ of node $b_k$ is the back dominator which is "closest" to $b_k$; that is for all $b_i$ and $b_j$ in $BD(b_k)$, $b_i$ is the node for which

$$\delta_{min}(b_i, b_k) = \text{Minimum } (\delta_{min}(b_j, b_k),\ \delta_{min}(b_j', b_k), \ldots) \ .$$

It can be shown that there is one and only one immediate back dominator of a node $b_k \neq e_0$. Suppose that there were two such nodes: $b_i$ and $b_i'$. Then $\delta_{min}(b_i, b_k) = \delta_{min}(b_i', b_k)$. But this can only occur if $b_i$ and $b_i'$ are on separate paths or if $b_i = b_i'$. Since a back dominator must be on every path, $b_i$ must equal $b_i'$. Furthermore, there must be at least one back dominator, $e_0$, since $e_0$ is on every $P \in \mathcal{P}$.

Another interesting observation which can be made is that the set of back dominators $BD(b_k)$ of node $b_k$ can be strictly ordered by the immediate back dominance relationship. This follows from the previous paragraph since, if $b_i$ is the immediate back dominator of $b_k$ and if $b_i \neq e_0$, $b_i$ must itself have one and only one immediate back dominator. From this it follows that the relationship of all the backdominating nodes in a control flow graph can be depicted by a tree.

The set of back dominators of node $b_k$ can be represented by $BD(b_k) = \{b_1, b_2, b_3, \ldots\}$. where $b_1 = e_0$, $b_i$ is the immediate back dominator of $b_{i+1}$ and $b_i$ is the back dominator of all $b_j$, $i < j \leq k$. Consider the control flow graph in Figure 3. $BD(1) = \phi$, $BD(2) = \{1\}$ , $BD(3) = \{1,2\}$ , $BD(4) = \{1,2\}$ , $BD(5) = \{1,2\}$ , $BD(6) = \{1,2\}$ , $BD(7) = \{1,2,4\}$ . An algorithm for finding the back dominator of every node in a control flow graph is now given.

Algorithm A: The set of back dominators $BD(b_i)$ of each node, $b_i$, in a control flow graph containing nodes $(e_0, b_2, b_3, \ldots b_n)$ is found by this algorithm.

1. Assume each node $b_i$ is back dominated by every other node in the graph. This is expressed by initializing $BD(b_i)$: $BD(b_i) = \{e_0, \ldots b_j \ldots\}$ for all $b_j \neq b_i$

2. The back dominator list of the initial entry node $e_0$ is initialized. $BD(e_0) = \phi$

3. For each $b_i$, $i > 1$, form $BD(b_i) = \bigcap_j (BD(b_j) \cup b_j)$ for all immediate predecessors $b_j$ of $b_i$. Before replacing the existing $BD(b_i)$ with the newly formed $BD(b_i)$, note if they differ.

4. If it was noted that any $BD(b_i)$ formed in step 3 was different, then repeat step 3.

The number of steps to find all of the back dominators of all of the nodes by algorithm A is bounded by $n^2$ where n is the number of nodes in the graph.

In practice, however, it appears to frequently take only 3 or 4 repetitions.

A modification can be made to the stated algorithm which improves it. The nodes can be ordered as follows: number $e_o$ as 1. Number all immediate successors of $e_o$ with 2,3,...i. Now number all unnumbered immediate successors of 2 with i+1, i+2...j, all unnumbered immediate successors of 3 with j+1, j+2,...k, etc. All the back dominators of a given node $b_i$ therefore have a smaller number than the given node. By processing the nodes according to their assigned numbers, the process should terminate in fewer steps.

Another byproduct of this modification is that if the back dominator list of a node is ordered by node number, then this is precisely the linear ordering of the back dominators of the node - the node with largest number being the immediate back dominator.

As an example consider the graph in Figure 3 as redrawn in Figure 4 in order to renumber the nodes as described.

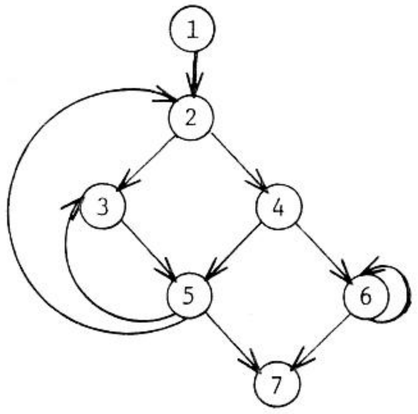


Figure 4

The application of algorithm A to the graph in figure 4 follows.

Back Dominators

| node | initial | Iteration 1st | Iteration 2nd | formula |
|---|---|---|---|---|
| 1 | $\phi$ | | | |
| 2 | 1,3,4,5,6,7 | 1 | 1 | $(BD(1) \cup 1) \cap (BD(5) \cup 5)$ |
| 3 | 1,2,4,5,6,7 | 1,2 | 1,2 | $(BD(2) \cup 2) \cap (BD(5) \cup 5)$ |
| 4 | 1,2,3,5,6,7 | 1,2 | 1,2 | $(BD(2) \cup 2)$ |
| 5 | 1,2,3,4,6,7 | 1,2 | 1,2 | $(BD(3) \cup 3) \cap (BD(4) \cup 4)$ |
| 6 | 1,2,3,4,5,7 | 1,2,4 | 1,2,4 | $(BD(4) \cup 4) \cap (BD(6) \cup 6)$ |
| 7 | 1,2,3,4,5,6 | 1,2 | 1,2 | $(BD(5) \cup 5) \cap (BD(6) \cup 6)$ |

Example 1

In this example only 1 iteration was needed to get the back dominator list for each node but, as usual, another iteration was needed to determine that we were finished. If the nodes had not been renumbered, that is if the graph in Figure 3 were used, 3 iterations would have been needed.

It is not necessary, obviously, to use lists in implementing this procedure. Bit vectors can be established with bit positions representing the nodes. The set operations of intersection and union can then be replaced with the much faster boolean operations of and and or.

A node $b_i$ is said to forward dominate or post dominate a node $b_k$ if $b_i$ is on every path from $b_k$ to all exit nodes. By introducing a node $x_0$ into the graph such that $\Gamma_G^{-1}(x_0) = X$, the set of exit nodes defined earlier, forward dominance relationships analogous to the back dominance relationships can be developed. Since the development so closely parallels that for back dominance it will not be given. Suffice it to say that the set of forward dominators, $FD(b_k)$, of node $b_k$ can be expressed by

$FD(b_k) = \{b_1, b_2, \ldots, b_j\}$ where $b_j = x_0$, $b_1$ is the immediate forward dominator of $b_k$ and for all i, $1 < i \leq j$, $b_i$ is the immediate forward dominator of $b_{i-1}$.

An <u>articulation node</u> in a graph is a node which lies on every entry-exit path. Thus for any graph with a single entry point, $e_0$, the forward dominators of $e_0$ are, together with $e_0$, the articulation nodes of the graph. Assuming node 6 in the graph in Figure 3 is the only exit node, nodes 1,2, and 6 are articulation nodes for the graph. An algorithm for finding articulation nodes is given in the next section.

All of these constructs are of interest during program optimization analysis: a strongly connected region is a generalization of a program loop and identifies a situation in which use-definition relationships cannot always be depicted by a tree or cycle free graph; a back dominating node forms a focal point into which code can be moved or against which common sub-expressions can be eliminated [Ref. 12] because of the guarantee that the back dominator will be executed at least once before any node which it back dominates. None of these constructs however gives a good processing order for exposing complex relationships such as the data flow in a program. A construct is now defined which codifies the dominance relationships and the partial orderings implied by the predecessor-successor relationships which exist in a control flow graph.

## INTERVALS

Given a node h, an <u>interval</u> I(h) is the maximal, single entry subgraph for which h is the entry node and in which all closed paths contain h. The unique interval node h is called the <u>interval head</u> or simply the <u>header node</u>. An interval can be expressed in terms of the nodes in it: $I(h) = (b_1, b_2, \ldots b_n)$; any edge $(b_i, b_j)$ for $b_i$ and $b_j \in I(h)$ is implicitly in I(h). $b_1 = h$.

By selecting the proper set of header nodes, a graph may be partitioned into a unique set of intervals. (A <u>partition</u> of a graph G is a set of subgraphs $g_1, g_2, \ldots, g_n$ such that $g_i \subset G$, $\bigcup_i g_i = G$ and for all $i \neq j$, $g_i \cap g_j = \phi$. Thus a graph partition covers the original graph with a set of disjoint subgraphs.) An algorithm for partitioning a graph, G, into a unique set of intervals is now given:

<u>Algorithm B</u>:

Given a control flow graph this algorithm finds the set $\mathcal{I}$ of intervals in the graph.

1. Establish a set H for header nodes and initialize it with $e_0$.
2. For $h \in H$ find I(h) as follows:
   - 2.1 Put h in I(h) as the first element of I(h)
   - 2.2 Add to I(h) any node all of whose immediate predecessors are already in I(h).
   - 2.3 Repeat 2.2 until no more nodes can be added to I(h).
3. Add to H all nodes in G which are not already in H and which are not in I(h) but which have immediate predecessors in I(h). Therefore a node is added to H the first time any (but not all)

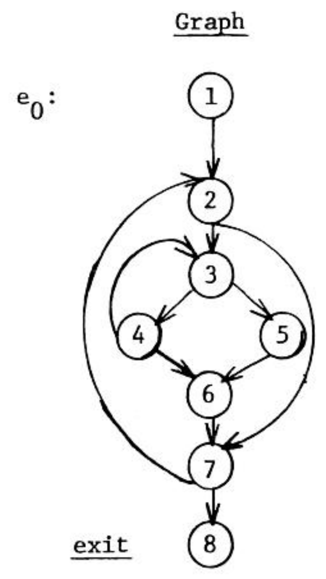
of its immediate predecessors become members of an interval.

4. Add I(h) to the set $\mathcal{J}$ of intervals being developed.

5. Select the next unprocessed node in H and repeat steps 2,3,4,5. When there are no more unprocessed nodes in H, the procedure terminates.

Before giving an example and before discussing the properties of the graph partition constructed by the above algorithm, a few comments on the algorithm itself may be of interest. In a program written by the authors to implement this algorithm, indicators are left on each node as to whether or not it is in H and, if not in H, a count is kept of the number of times it has been looked at during the development of the current interval. This latter count is kept because, once a block is added to the current interval, only its immediate successors are candidates for addition to the interval. Thus a quick comparison of the number of actual predecessors against the number of times the node is visited as a successor of interval nodes determines whether or not it can become a member of the current interval. Using such techniques, an edge in the graph is never traversed more than once. Thus the execution time for the algorithm is directly proportional to the number of edges in the graph.

Figure 5 illustrates the partitioning of a graph into itervals:



Intervals

I(1) = 1

I(2) = 2

I(3) = 3,4,5,6

I(7) = 7,8

(the naming of the nodes is, as usual, arbitrary)

Figure 5

It can be shown [2,7] that the interval finding algorithm, algorithm B, does indeed produce a set of intervals each of which satisfies the definition of an interval and further that the set provides a unique partition of the graph.

Before considering some interesting properties of intervals, the meaning of "interval exit node" needs to be more carefully defined: an interval exit node is any node in an interval, I(h), which either has no successors (i.e. is a terminal node for the entire graph) or has at least one immediate successor which is not in I(h).

Some properties of intervals are:

1. The header node of an interval back dominates every node in the interval.
2. Any strongly connected region in an interval must contain the interval head. Therefore, if an interval contains a strongly

connected region then there exists a path from every node in the region to every node in the interval.

3. The order of nodes in an interval list (called the process order) is such that if the nodes on an interval list are processed in the order given, then all interval predecessors of a node will have been processed before the given node. (The interval predecessors of a node b are all those nodes in the interval which are on loop free paths from the header to b.)
4. The interval header is an articulation node for the interval.
5. All forward dominators of the interval header which are also interval members are, along with the header, articulation nodes for the interval.

Consider the interval in figure 6 with header node 1 and exit node 6 and assume the order in which nodes became interval members results in I(1) = (1,2,3,5,4,6).

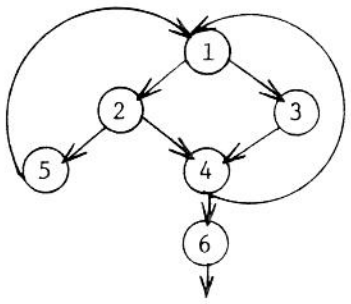


Figure 6

Clearly node 1 back dominates every node in I(1). The strongly connected region of I(1) consists of the nodes 1,2,3,4 and 5. The articulation nodes of the interval are 1, 4 and 6.

In certain applications a special graph construct called a "two-terminal subgraph" may be of interest. Defined in terms of intervals, a <u>two-terminal subgraph</u> is an interval with one exit node. Since an interval can have only one entry node, the motivation for the term should be apparent. The interval in Figure 6 is an example of the two-terminal subgraph.

Algorithms will now be given for finding the strongly connected region in an interval, the articulation nodes of the interval, and for each node in the interval the list of interval nodes which back dominate it. These algorithms can be embedded in the interval finding algorithm thereby generating not only the intervals but their internal relationships in "one pass" through the edges in the graph.

Up to this point in the paper it has been completely satisfactory to represent members of a set in terms of a list of the elements in it. For example, $I(h) = (b_1,b_2,\ldots,b_n)$ where $b_1$ represents the name of the node in position i in the interval. Although we will continue to use the list form of representation in the algorithms described, another form could be introduced which more directly suggests the relationships involved as well as a possible implementation approach. A bit vector notation could be used in which, for a given interval, $I(h) = (b_1, b_2,\ldots,b_n)$, bit position i represents node $b_i$. By remembering the correspondence between bit positions and node names, no information is lost. Boolean operations rather than the set operations shown could then be used. Also the relative order of the nodes in the interval is automatically kept by the bit vector positions. Since it would complicate the exposition, the bit vector form will not be used in describing the algorithms.

The next algorithm generates a back dominator list, $BD(b_i)$, for each node $b_i$

in the interval. Each back dominator list as generated is unordered. Since, however, the relative ordering of nodes in the interval can be used to order the nodes in the back dominator list, the correct ordering can be determined. By using the bit vector representation, the ordering is kept automatically. If, in that representation, a bit is one in the back dominator vector if and only if the block represented by that position is a back dominator, then the right most one bit in the vector represents the immediate back dominator.

Algorithm C

This algorithm finds the interval back dominators of each node in an interval. It can be embedded in Algorithm B.

1. Assign the interval head a back dominator list of zero.
2. For the next node, $b_j$, in the interval list (or for the one just added if this algorithm is embedded in Algorithm B) form $BD(b_j) = \bigcap_i (BD(b_i) \cup b_i)$ for all nodes, $b_i$, which are immediate predecessors of $b_j$.
3. Repeat 2 until all nodes in the interval have been processed.

As an example consider the interval of Figure 6 for which I(1) = (1,2,3,5,4,6). The algorithm generates the following back dominator list for each node by the operations shown.

| Nodes (in order) | Immediate Predecessors | Operation | BD List for Each Node |
|---|---|---|---|
| 1 | - | (Assignment) | 0 |
| 2 | 1 | $0 \cup 1$ | 1 |
| 3 | 1 | $0 \cup 1$ | 1 |
| 5 | 2 | $1 \cup 2$ | 1,2 |
| 4 | 2,3 | $(1 \cup 2) \cap (1 \cup 3)$ | 1 |
| 6 | 4 | $1 \cup 4$ | 1,4 |

Example 2

In the next algorithm, D, the interval articulation nodes are found by using the back dominators of interval exits. The result of algorithm D is a list, A, of articulation nodes for the interval.

Algorithm D

The interval articulation nodes are found by this one step algorithm.

1. $A = \bigcap_i (b_i \cup BD(b_i))$ for all $b_i$ which are interval exits.

Consider the interval of Figure 6 and the back dominator lists given in Example 2. Since node 6 is the only interval exit, $A = 6 \cup (1,4) = 1,4,6$. If node 5 were also an exit, then $A = [5 \cup (1,2)] \cap [6 \cup (1,4)]$ and the only articulation node would be 1.

The next algorithm - algorithm E, finds all of the interval predecessors of a node.

Algorithm E

The interval predecessors, $IP(b_i)$, for each node, $b_i$, in an interval are found by:

1. Assign the interval head an interval predecessor list of zero: $IP(b_i) = 0$.
2. For the next node, $b_j$, in the interval list $IP(b_j) = \bigcup_i (b_i \cup IP(b_i))$ for all nodes $b_i$ which are immediate predecessors of $b_j$.
3. Repeat step 2 until all nodes in the interval have been processed.

Considering again the graph in Figure 6 the following IP lists are generated by Algorithm E:

| Nodes | Imm. Pred. | Operation | IP Lists |
|---|---|---|---|
| 1 | - | (Assignment) | 0 |
| 2 | 1 | $1 \cup 0$ | 1 |
| 3 | 1 | $1 \cup 0$ | 1 |
| 5 | 2 | $2 \cup 1$ | 1,2 |
| 4 | 2,3 | $(2 \cup 1) \cup (3 \cup 1)$ | 1,2,3 |
| 6 | 4 | $4 \cup (1,2,3)$ | 1,2,3,4 |

Example 3

The next algorithm, F, uses the results of algorithm E for the interval "latching" nodes to find the strongly connected region in the interval. A latching node is any node in the interval which has the header node as an immediate successor. An equivalent definition for a latching node is that it is any node in the interval which is an immediate predecessor of the interval head. In Figure 6 nodes 4 and 5 are latching nodes. It should be noted that the interval head itself can be a latching node. From previous observations it follows that if the interval does not contain any latching nodes, then the interval does not contain a strongly connected region. The following algorithm would be invoked only if the interval had at least one latching node.

Algorithm F

The strongly connected region, SCR, of an interval can be found by this one step algorithm.

1. $SCR = \bigcup_i (b_i \cup IP(b_i))$ for all $b_i$ which are interval latching nodes.

Using the results of Example 3, we get $SCR = [4 \cup (1,2,3)] \cup [5 \cup (1,2)]$. Therefore the strongly connected region of the interval in Figure 6 has the nodes (1,2,3,4,5).

Another algorithm, F', for finding the strongly connected region in an interval is to start from the latching nodes and iteratively mark all immediate predecessors until the header node is reached and marked. Whenever a marked predecessor is found in this algorithm, it is not necessary to continue the marking of its immediate predecessors since they will already have been marked. This algorithm has the advantage of not requiring that the IP lists be set up and is probably preferable if the only use of IP lists is to find the strongly connected region.

A formal description of Algorithm F' is not given; the informal description should adequately suggest such a description.

## PARTITIONING GRAPHS BY INTERVALS

Having considered the properties of any given interval, the properties of the set of intervals $\mathcal{I} = \{I(h_1), I(h_2), I(h_3)\ldots\}$ generated by the interval finding algorithm, Algorithm B, will be considered. It can be shown [2] that a set of intervals does indeed form a unique partition of a graph G. (Recall that our definition of G assumes a single entry node, $e_o$.) Thus for a given G:

1. $\mathcal{I}$ covers G.
2. The elements of $\mathcal{I}$ are disjoint. Therefore, for any I(h) and I(h') in $\mathcal{I}$, $I(h) \cap I(h') = \phi$.
3. $\mathcal{I}$ is unique.

Having described the relationship of the total set of intervals to the total graph and, prior to that, having shown some of the inter-relationships of nodes in a given interval, we now want to enlarge the scope of an interval so that the inter-relationships of larger sets of nodes can be codified.

The intervals described thus far have been formed from the elemental nodes of the graph (the basic blocks of the control flow graph). For reasons which will be apparent shortly, we designate these intervals as the <u>basic</u> or <u>first order intervals</u> and the graph from which they were derived as the basic or first order graph. Since we will be deriving higher order graphs and intervals we will use superscripts to designate the order, e.g., $I^1(h) \epsilon \mathcal{I}^1$.

A <u>second order graph</u> is derived from the first order graph and its intervals by making each first order interval into a node. The immediate predecessors of such a node in the second order graph are all the immediate predecessors of the original header node which were not members of the interval; the immediate successors of such a node are all of the immediate, non-interval successors of the original interval exit nodes.

Second order intervals are the intervals in the second order graph. With respect to the second order graph, they have all of the properties derived for the first order intervals. Since the nodes of the second order intervals are first order intervals, we have by our procedure derived some inter-interval relationships.

Successively higher order graphs can be derived until the n-th order graph either consists of a single node or is "irreducible". This latter case will be described after we give an example of a graph which "reduces" to a single node. In the example in Figure 7 only multi-node intervals are renamed in the derived graph.

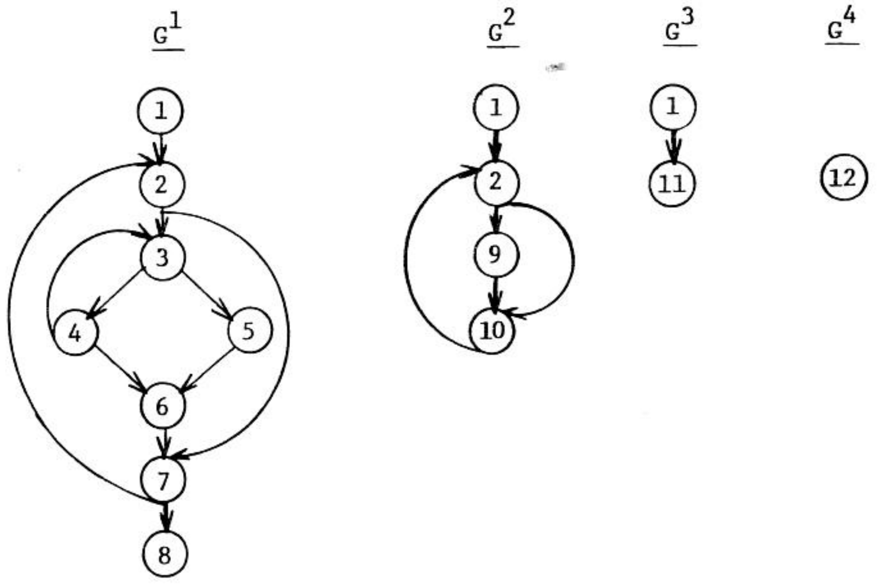


**Figure 7**

The interval sets of the graphs in Figure 7 are

| $\underline{\mathcal{I}^{1}}$ | $\underline{\mathcal{I}^{2}}$ | $\underline{\mathcal{I}^{3}}$ | $\underline{\mathcal{I}^{4}}$ |
|---|---|---|---|
| $I^{1}(1) = 1$ | $I^{2}(1) = 1$ | $I^{3}(1) = 1, 11$ | $I^{4}(12) = 12$ |
| $I^{1}(2) = 2$ | $I^{2}(2) = 2,9,10$ | | |
| $I^{1}(3) = 3,4,5,6$ | | | |
| $I^{1}(7) = 7,8$ | | | |

Another diagram, borrowed from a currently unpublished paper by Patricia Goldberg of IBM, more graphically depicts the relationships involved. It is given in Figure 8.

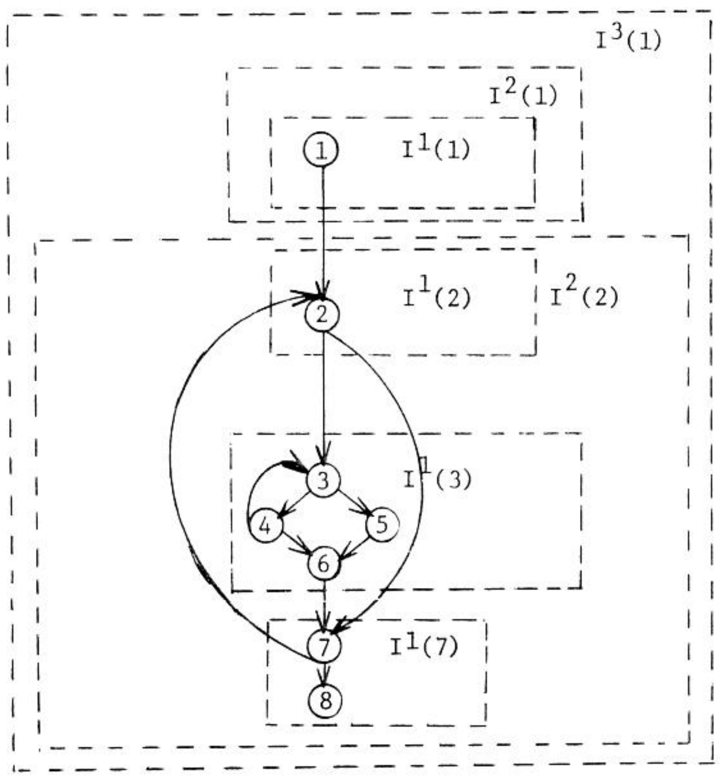


Figure 8

A reducible graph is a graph which contains at least one non-trivial interval. (A trivial interval is a subgraph consisting of a single node and no edges. Thus is a non-trivial interval.) An irreducible graph is a graph containing only trivial intervals. Figure 9 has three examples of irreducible graphs.

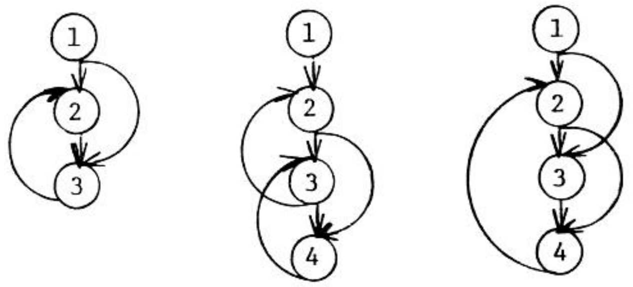


Figure 9

A fully reducible graph is a reducible graph all of whose derived graphs which contain more than one node are also reducible. $G^1$ in Figure 7 is an example of a fully reducible graph.

Given a fully reducible graph $G^1$ with successive derivations $G^2$, $G^3 \ldots G^n$, then several observations can be made:

1. $G^n$ is a single node.
2. Every node in $G^1$ is in one and only one interval $I^1(h) \in \mathcal{I}^1$ which is in turn a node in $G^2$ and hence in one and only one interval $I^2(h)$ in $\mathcal{I}^2$, etc. Thus a basic block can be viewed as the inner element of a nested sequence of increasingly more global regions which are depicted as nodes. Therefore

   $$b_i \in I^1 \in I^2 \ldots \in I^n.$$

3. By virtue of their memberships in interval lists, an order for

processing all the nodes in a program is established. Although the order is clearly not unique, it establishes a means of selecting the "next node for processing" in many situations.

The next section discusses in detail the problem of "splitting nodes" in an irreducible graph to form an equivalent graph which can be further reduced. Although we go into this problem in great detail, it should be put into perspective in terms of its practical importance. As shown in the section of this paper which summarizes the analysis of 72 FORTRAN programs, only 5 of the 72 program control flow graphs were not fully reducible. Knuth [11] found that all of 50 randomly selected FORTRAN programs were fully reducible. The detailed discussion of node splitting given here reflects the interest in the problem rather than its importance for the analysis of program control flow graphs; its relevance to other problems may be more significant.

## NODE SPLITTING

When an irreducible graph containing more than 1 node has been found during interval analysis, an equivalent graph can be constructed which can then be further reduced. Consider the example in Figure 10.

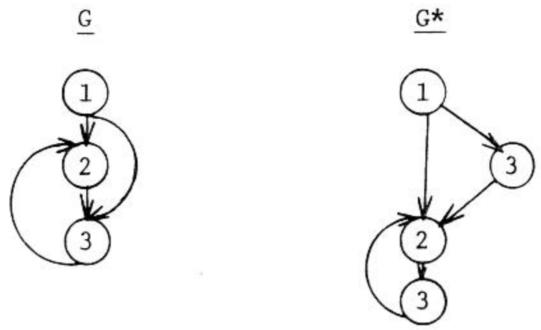


Figure 10

The graph G is irreducible but the graph G* is "equivalent" to G and is reducible. G* has been constructed from the irreducible graph by making two copies of node 3 - one for each input edge. In this example, node 2 is as good a choice for duplication as node 3. In many cases, however, it is not at all obvious which node or nodes should be copied. Figure 11, which is the most complicated irreducible graph found during the analysis of 72 FORTRAN programs, should convince the reader of the potential difficulties.

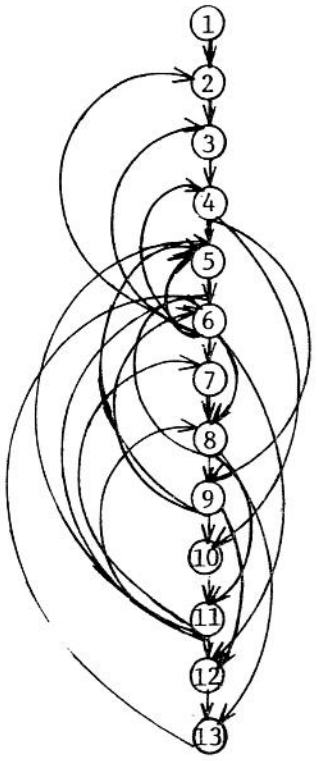


Figure 11

Node splitting is the process of selecting and replicating certain nodes of a given graph, G, and generating an equivalent graph, G*. Two graphs, G and G*, are said to be equivalent graphs if the set of all entry-exit paths in one is identical to the set of all entry-exit paths in the other. If G is a program control flow graph, then G* is an equivalent representation of the original flow. We may visualize this as being accomplished by replicating the instructions in the nodes but in actual practice instruction replication is not done. All of the control flow analysis is performed to yield a codification of the control flow; replicated nodes are therefore only notationally replicated in the codification and are not actually replicated.

Since the purpose of node splitting in program control flow analysis is to transform an irreducible graph into an equivalent graph which can be further reduced, we need to consider in more detail what characterizes an irreducible graph and how to transform it into one which contains non-trivial intervals.

Several assertions can be made about an irreducible graph G:

A1. With the exception of the entry node, every node in G has more than one immediate predecessor. Since G is irreducible, it contains only trivial intervals (i.e. intervals consisting of a single node and no edges). However a node with only one immediate predecessor would form an interval with its predecessor. Therefore, there cannot be such a node.

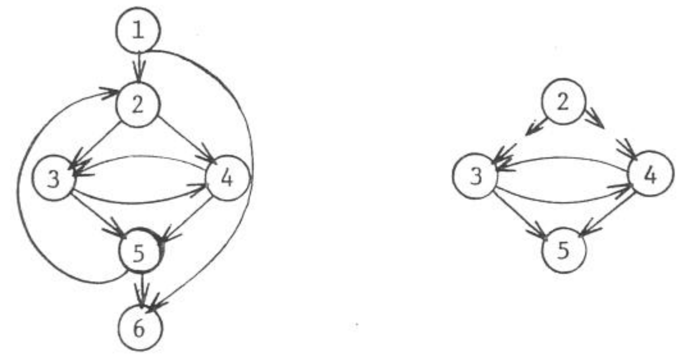
A2. At least one immediate successor of G's entry node, $e_0$, must be in a strongly connected region. Consider the set S of immediate successors of $e_o$: $S = \{s_1, s_2 \ldots s_n\}$ . By A1 we know that every $s_i$ must have more than one immediate predecessor. Therefore in addition to the path $(e_o, s_i)$ there must be a path $(e_o, s_j, \ldots s_i)$. But $s_j$ also has more than one predecessor so there must be a path $(e_o, s_k \ldots, s_j, \ldots s_i)$. Since there are only a finite number of nodes in S and each one has a predecessor in S, then there must be a closed path containing at least one of the nodes in S.

This gives us the asserted strongly connected region.

This result is interesting in that it shows that every irreducible graph must contain at least one strongly connected region. We now extend this to a class of subgraphs.

A3. At least one immediate successor of any back dominating node in G must be in a strongly connected region. Pick any back dominating node, d, in G and consider the subgraph, G', which d back dominates. Since d back dominates every node in G', all paths into G' must go through d, i.e. d with G' must be a single entry subgraph. We can therefore prove the assertion by exactly the arguments used in A2 with d substituted for $e_o$.

In Figure 12(a) nodes 1 and 2 are back dominators. In Figure 12(b) node 2 and the subgraph which it back dominates have been extracted and shown separately.



(a) Figure 12 (b)

Not all of the nodes back dominated by a given back dominator are necessarily in the same strongly connected region or indeed in any strongly connected region. Figure 13 shows this. Node 1, the only back dominator in the graph, back dominates nodes in two disjoint strongly connected regions (2,3) and (4,5) and a node, 6, which is not in any strongly connected region.

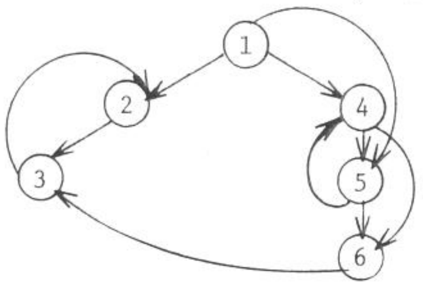


Figure 13

The back dominating nodes may or may not be in strongly connected regions themselves. In the graph in Figure 11, which has back dominators 1,2,3 and 4, all but node 1 are in a strongly connected region.

A4. At least one of the maximal strongly connected regions in a subgraph back dominated by a back dominator d will have d as its only immediate predecessor. Consider the set of maximally strongly connected regions $R = \{R_1, R_2 \ldots\}$. Suppose every $R_i \in R$ has a predecessor other than d. Certainly if every $R_i \in R$ has a predecessor in another $R_j \in R$ $(i \neq j)$ then, since the number of maximally strongly connected regions is finite, some of the elements of R must together form a strongly connected region. Since this violates the condition that the elements of R are maximal, it is not possible for every $R_i$ to have a predecessor in another $R_j \in R$ $(i \neq j)$. Therefore if our hypothesis is true, some $R_i \in R$ must have an immediate predecessor which is not in a strongly connected region and which is not a successor of a strongly connected region. But there cannot be such a node in the subgraph. (Suppose there were. Then, since such a node must have two immediate predecessors, neither of which are in strongly connected regions or are successors of strongly connected regions, the fact that there are only a finite number of nodes leads to the contradiction). Since there must be an $R_i$ having only immediate predecessors which are not in a strongly connected region or are not successors of strongly connected regions and such immediate predecessors cannot exist in the subgraph, there must be an $R_i$ having only d, the back dominator of the subgraph, as an immediate predecessor.

We now consider some characteristics of strongly connected regions in an irreducible graph.

A5. All simple cycles in an irreducible graph must be multiple entry. From the definition of intervals, any simple cycle $C = (b_1, b_2, \ldots b_1)$ in which $b_1$ is the only entry node, must form all or part of an interval. Therefore, since an irreducible graph does not contain any intervals, one of the other nodes must also be an entry node.

A6. All prime cycles in an irreducible graph must be multiple entry. Since a prime cycle is just a special case of a simple cycle, this assertion follows immediately from A5.

Recalling that all cycles in an interval pass through the interval head and that the interval head back dominates all the nodes in the interval, we now consider how to transform an irreducible graph into a graph containing intervals. Essentially the approach is to select a node or set of nodes which are good candidates for potential interval heads and then to copy (split) successors of these nodes so that intervals are in fact exposed. It is important of course that the resultant graph be equivalent to the original.

Figure 14 is an example of an irreducible graph in which nodes 1 and 2 were selected as candidates for interval heads. The graph is transformed and then reduced.

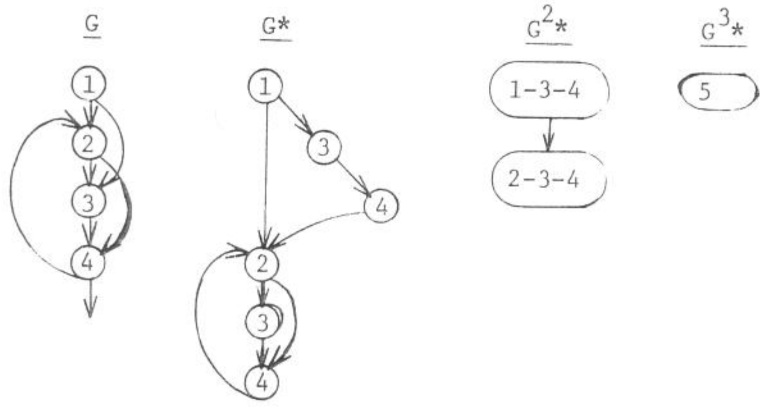


Figure 14

Before describing how nodes are selected for use as potential interval heads, their use in constructing the transformed graph G* from the irreducible graph G is given.

A pivot node is a node in G which is selected to head an interval in the transformed graph, G*. The transformation is done so that the pivot node p back dominates a subgraph in G*. This subgraph is constructed by not allowing any successor of p to have any immediate predecessor which is not also a successor of p. The nodes which do have such predecessors are split. One copy of the split node will have only p or successors of p as predecessors; the other copy will have the remaining predecessors. Both copies have all the original successors. The successors of both 1 and 2 in G* in Figure 14 exemplify these characteristics. Later we will see that several copies of a node may occur when several p's are used.

In general there will be a set of pivot nodes $P^+$ used in the transformation of an irreducible graph. This set is formed by: $P^+ = e_o \cup P$ where P is the set of pivot nodes selected from the interior of the graph. P must contain at least one node since it would not be possible to transform the graph by the construction to be given if $e_o$ were the only pivot node.

C1: <u>Construction of G*</u>: Given an irreducible graph G and a set of pivot nodes $P^+$, the equivalent graph G* is constructed as follows:

1. A set of subgraphs is constructed - one for each element of $P^+$. For $p \in P^+$, Gp is constructed by:
   (a) putting p in $G_p$
   (b) copying into $G_p$ all nodes which are not other pivot nodes and which can be reached from p without going through other pivot nodes and all edges which can be traversed without going through a pivot node.

2. Form G* from the $G_p$ subgraphs by connecting each $G_p$ to all of its immediate predecessors in all the other subgraphs. A pivot node appears only once in G*.

In Figure 15, G* is obtained from G by using the pivot nodes 1 and 4. The dotted lines in the figure indicate the connections made in step 2 of the construction.

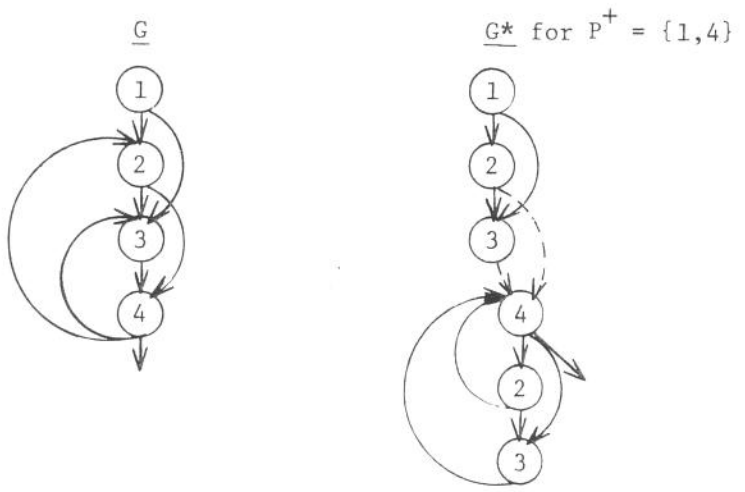


Figure 15

The construction is now shown to produce a graph equivalent to the irreducible one.

A7. G* is equivalent to G. If G and G* are not equivalent, then there is an entry-exit path in one which is not in the other. For this to happen, there is an edge in one which is not in the other. Given any edge $(b_i, b_j)$ in G, we will show that it must appear at least once in G*. If $b_j$ is a pivot node, then step 2 assures us that $(b_i, b_j)$ are connected. If $b_j$ is not a pivot node, step 1 forces us to have included it at least once in G* if $b_i$ can be reached from a pivot node. But since all nodes in G are reachable from $e_o$, $b_i$ must be reachable directly from $e_o$ or through some pivot node. Therefore, $(b_i, b_j)$ must exist in at least one subgraph and hence in G*. Now given an edge $(b_i, b_j)$ in G*, we assert that it must have existed in G since the construction does not introduce any edges in G* which did not exist in G.

Figure 16 gives another example of a node splitting tranformation.

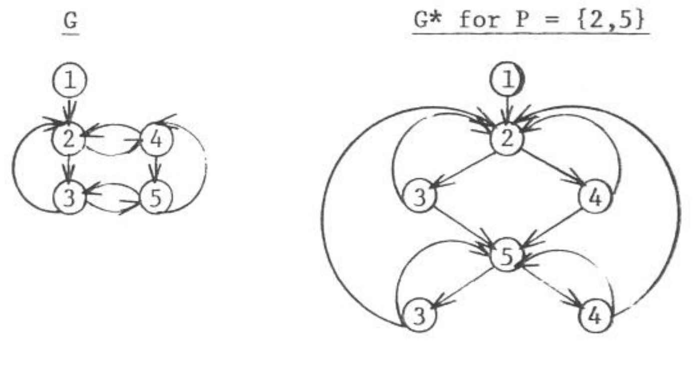


Figure 16

The G*'s in Figures 15 and 16 are both fully reducible; G* in Figure 15 will reduce to a single node on the second iteration through the interval analyzer while G* in Figure 16 will reduce to a single node on the third iteration.

Having shown how a set of pivot nodes is used to construct a graph equivalent to a given graph, we now consider how to select pivot nodes from the given graph. Since each pivot node, p, is to head an interval, p must be made to back dominate a subgraph in which all cycles pass through p. The construction of G* from G assures that p back dominates the subgraph $G_p$ formed for it. It does not however assure that all or even any of the cycles in $G_p$ Pass through p. In order for this to happen, p must have been a node in a cycle. The elements of P, the set of pivot nodes, is selected therefore so that they become the entry nodes of single entry cycles in the transformed graph. This is done by selecting a set of nodes which "break" the multiple entry cycles in G. (A set of nodes is said to break a graph if the graph becomes acyclic when the nodes in the set are removed from the graph). Clearly any node in a prime cycle can be used to break the cycle.

A8. A set of nodes, P, which breaks all the prime cycles in a graph breaks the graph, i.e., renders the graph acyclic when removed. Suppose this were not the case, that is suppose that on removing the set of nodes from a graph, the graph still had a cycle, C, in it. We can assume that C is a simple cycle without any loss in generality since a composite cycle contains simple cycles. Since C is not broken, it is not a prime cycle so therefore it must contain a prime cycle. But any prime cycle has at least one element of P. Therefore C must contain an element of P. But this is impossible since all elements of P have been deleted from the graph.

Assuming that the set of pivot nodes, P, contains a node from every prime cycle in the graph G, the following three assertions can be made.

A9. Each subgraph, $G_p$, constructed for a node $p \in P^+$ forms an interval with the pivot node as its head. By construction (C1-step 1), only those edges and nodes which can be reached from p without going through or including other pivot nodes are in the subgraph. The subgraph has by this construction the single entry node p. We now need to show that all cycles go through p. Since the only pivot node in $G_p$ is p and since by A8 the original graph G would become acyclic if all pivot nodes were removed, all cycles in $G_p$ must go through p.

A10. Each $G_p$ will either form an interval by itself in G* or will be included in a larger interval. By step 2 of C1, in which the $G_p$ subgraphs are connected, only pivot nodes have immediate predecessors in other subgraphs. Thus each $G_p$ retains its interval properties.

A11. The number of nodes in $G^{*2}$, the graph derived from G*, is less than or equal to the number of elements in $P^+$. This follows directly from A10 since each interval in G* is a node $G^{*2}$.

It is harder to place a useful bound on the number of edges in $G^{*2}$. Certainly there are no more edges than were needed in connecting the $G_p$ subgraphs during the construction. An edge in $G^{*2}$ reflects the existance of at least one path from one pivot node in G to another pivot node.

The graph resulting from node splitting is not necessarily fully reducible.

Figure 17 is an example of such a graph.

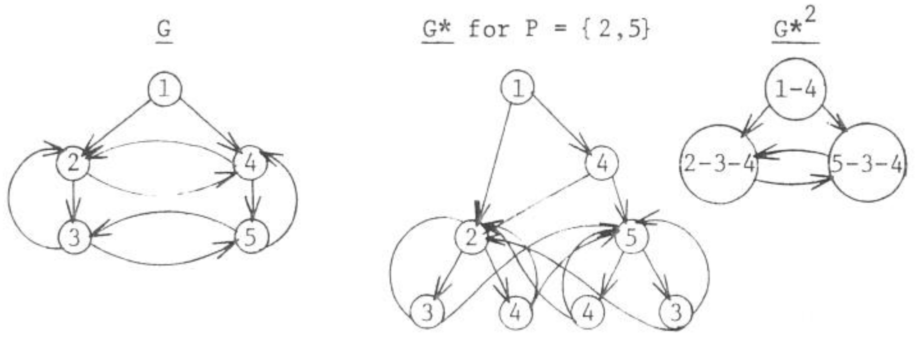


Figure 17

Even though we do not have easy methods or even criteria for determining which of several possible sets of pivot nodes for a given graph is "best" we do know that there must be a pivot node for every prime cycle and that minimizing the number of pivot nodes is probably desirable. (Examples exist in which a "more reducible" graph is constructed by choosing a set of pivot nodes larger than the minimal set. However, a minimal set seems in most cases to be preferable to a non-minimal set.) The selection of pivot nodes can therefore be viewed as a process which

a) finds all the prime cycles in the graph

and b) finds the smallest set of nodes which breaks all the prime cycles. (There may of course be more than one such set but one is arbitrarily selected.)

We will first consider the problem of finding the smallest set of nodes which break all the prime cycles. This problem can be expressed as a boolean function F, in conjunctive normal form (product of sums). For the example in Figure 11 which has prime cycles (5,6), (6,8), (8,11), (6,12,13), (4,9,6) and (4,10,11,6), the function F would be

$$F = (5+6)(6+8)(8+11)(6+12+13)(4+6+9)(4+6+10+11)$$

This expresses that the possible pivot nodes for that example are 5 or 6 and 6 or 8 and 8 or 11 and 6 or 12 or 13, etc. Expressed in this form the problem then becomes the problem of finding a minimum set of irredundant prime implicants and we can appeal directly to existing solutions [14,15] to the problem.

A "prime implicant table," T, is established in which each row represents a prime cycle and each column a node in the graph. The boolean entries have the following meanings:

$T_{ij} = 1$ if the prime cycle arbitrarily numbered i contains node j.

$T_{ij} = 0$ otherwise

The prime implicant table for the prime cycles of the graph in Figure 11 is

| cycles \ nodes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | 1 | 1 | | | | | | | |
| 2 | | | | | | 1 | | 1 | | | | | |
| 3 | | | | | | | | 1 | | | 1 | | |
| 4 | | | | | | 1 | | | | | | 1 | 1 |
| 5 | | | | 1 | | 1 | | | 1 | | | | |
| 6 | | | | 1 | | 1 | | | | 1 | 1 | | |

Before giving the part of Petrick's method [15] which we will use for selecting a set of nodes which "covers" the prime cycles, we need some definitions:

1) A set of nodes, P, is said to <u>cover</u> a set of cycles C if at least one node from every cycle is in P.

2) A minimum cover is a covering set of minimum cardinality.

3) Cycle $C_i$ covers cycle $C_j$ if every node in $C_j$ is also in $C_i$.

4) Node $N_i$ is covered by node $N_j$ if every cycle containing $N_i$ also contains $N_j$.

In finding a minimum cover, these last two relationships are interesting because

1) if $C_i$ covers $C_j$, then $C_i$ can be discarded since any node selected to break $C_j$ will also break $C_i$. This cannot happen with the original prime cycles since $C_i$ would not be prime if it covered $C_j$ but it will arise during the execution of the algorithm for finding the set P.

2) if node $N_i$ is covered by node $N_j$, then $N_i$ can be discarded since $N_j$ is in the same cycles as $N_i$ and probably in other cycles so is preferable to $N_i$.

Consider the prime implicant table given for Figure 11. We immediately see that we can discard node 4 from consideration since every cycle which contains node 4 (cycles 5 and 6) also contains node 6. Hence 4 could not be in our minimum cover. The same is true of node 5, node 9, node 10, node 12 and node 13. After eliminating those nodes as candidates for the minimum cover, the prime implicant table becomes:

| nodes / cycles | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | 1 | | | | | | | |
| 2 | | | | | | 1 | | 1 | | | | | |
| 3 | | | | | | | | 1 | | | 1 | | |
| 4 | | | | | | 1 | | | | | | | |
| 5 | | | | | | 1 | | | | | | | |
| 6 | | | | | | 1 | | | | | 1 | | |

We can now eliminate from consideration the remnants of cycle 2 since it covers cycle 1 (and cycles 5 and 6). At this point also it is clear that node 6 must be a member of the minimum cover since it is the only node left for consideration in at least one cycle. Having picked node 6, we delete it and all cycles containing it from the table and examine what is left.

| Nodes / Cycles | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | | | | | | | | 1 | | | 1 | | |

The choice at this point is completely arbitrary - either node 8 or node 11 is chosen. Thus the set of pivot nodes for figure 11 is either P = {6,8} or P = {6,11}. Either of these sets would render that graph acyclic if removed.

Determining the minimum cover is not always as straightforward as the preceeding example would indicate. A <u>sure pivot</u> is a node which must be selected to obtain a minimum cover. Node 6 in the preceding example is a sure pivot and was identifiable as such when it became the only node remaining in a cycle after eliminating the covered nodes and covering cycles from the prime implicant table. It sometimes happens that no more covered nodes or covering cycles can be eliminated and no more sure pivots exist. Consider the prime implicant table for the graph, G, in Figure 17.

| Cycles \ Nodes | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | 1 | 1 | | |
| 2 | | 1 | | 1 | |
| 3 | | | 1 | | 1 |
| 4 | | | | 1 | 1 |

There are no covered nodes, covering cycles or sure pivots. Petrick [15] gives a "branching" method for obtaining a minimal cover in this situation. For the purposes of node splitting for program control flow graphs, it is probably acceptable to apply a heuristic whenever a minimum cover is not apparent. The resulting cover is not necessarily minimal but the algorithm is faster. A good heuristic seems to be:

1. Pick the node which is in the most cycles as a pivot node or,

2. if there are several nodes which satisfy 1, arbitrarily select one of them.

In either case, the selected node is deleted from the prime implicant table and the process of selecting pivot nodes continued. We now give the pivot node selection algorithm.

Algorithm G

Given a set of cycles, this algorithm finds a covering set of nodes, P. P will be a minimum cover if the heuristic step, step 5, is not invoked.

1. Delete all covering cycles.

2. Delete all covered nodes.

3. If any covered nodes were deleted, then return to step 1; otherwise go to step 4.

4. Are there any cycles containing only one node? If so, these are pivot nodes: add them to P, discard all cycles containing these nodes and go to step 6. If not, go to step 5.

5. (heuristic step). If there were no cycles containing only 1 node, then select pivot node by picking the node in the most cycles or, if there are several such nodes, arbitrarily selecting one of them. Add the selected node to P and discard all cycles containing the node.

6. Any nodes left? If so, return to step 1 otherwise stop.

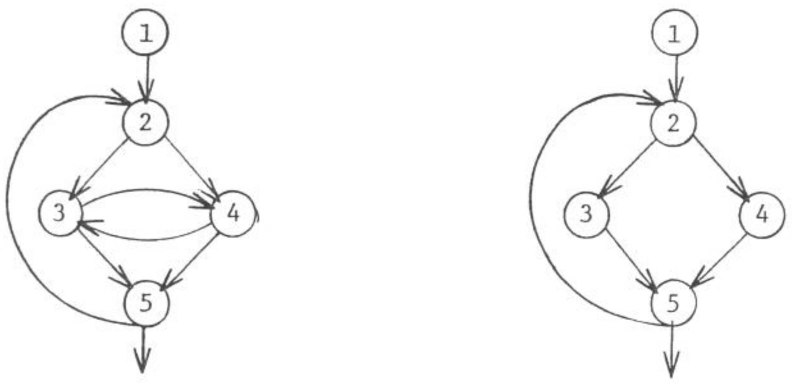
We now turn to the problem of finding the prime cycles in the irreducible graph. Since the graph is irreducible, there are no 1-cycles, (cycles of the form ). The shortest cycles are 2-cycles and, since there are no 1-cycles, all the 2-cycles in the graph must be prime. The first problem then is to find all 2-cycles in the graph. There are several alternatives, none of which we give in any detail.

1. A graph search can be performed by considering each node in a strongly connected regions of the graph and checking to see if any successors of the successors of the node are the node itself. By arbitrarily numbering the nodes and looking at them in order it is not necessary to consider any successors $b_j$ of a node $b_i$ for which $j < i$.

2. A boolean connectivity matrix, C, [1,16] can be constructed for the graph and then squared to form $C^2$. The diagonal elements in $C^2$ indicate nodes in 2-cycles. The actual 2-cycles can be distinguished either by looking at the graph or by using an auxiliary distance matrix [1].

Our preference for irreducible program control flow graphs is method 1.

Having found and recorded all of the 2-cycles the edges in the cycle are deleted from the graph. Before doing this, however, it is worthwhile checking to see if any of the cycles will contain a sure pivot. This will occur if a node in a 2-cycle has only one successor - the other node in the cycle. When this occurs, the other node, the node with more than one successor, is a sure pivot. When such a sure pivot is found, we will record the cycle as containing only that node and then both nodes and all of their edges are deleted from the graph. The graph, G, in Figure 15 has two 2-cycles: (2-4) and (3-4). In the (3-4) cycle, 3 has only one successor so 4 will be selected as a sure pivot. By deleting node 4 and all of its edges from the graph, the graph becomes acyclic and we do not have to search further for prime cycles. The graph in Figure 18(a) has the 2-cycle (3-4) but does not have a sure pivot.



(a) Figure 18 (b)

In Figure 18(b) the edges involved in the (3-4) cycle have been removed.

After the 2-cycles have been found and the graph modified by deleting the edges of the 2-cycles and by deleting the sure pivots, the search for prime cycles continues. For this we can use a "node coalescing" method developed in cooperation with Dr. Raymond Miller of IBM. This method reduces the problem of finding successively longer cycles to the problem of finding 2-cycles. The essence of the method is to merge a node into its predecessors and then look to see if any 2-cycles have been created by this merge.

## Algorithm H

Given a method for finding the 2-cycles in a graph, this algorithm finds n-cycles by a node-coalescing method. We start by assuming that there are no 1-cycles or 2-cycles.

1. Select a node $b_i$ in a strongly connected region for merging with its immediate predecessors. The node $b_i$ is selected by picking the node with the minimum number of predecessors which are also in the strongly-connected region. If there are several such nodes one is arbitrarily chosen.

2. For each predecessor $b_p$ of the selected node $b_i$ a new node $b_k$ is formed. $b_i$ is deleted as a successor of each $b_p$. The new node $b_k$ then gets all of the predecessors of $b_p$ and all of the successors of $b_i$. This is the node-coalescing step.

3. Record with each $b_k$ the nodes that went into its formation. This may involve updating previously recorded information.

4. Look for any 2 cycles which may have been formed by step 2. If there aren't any, return to step 1, otherwise go to step 5.

5. For each 2-cycle found by step 4, determine the n-cycle which it represents. This is done by using the information recorded with each node by step 3 regarding the node's constituents.

Consider the example given in Figure 19.

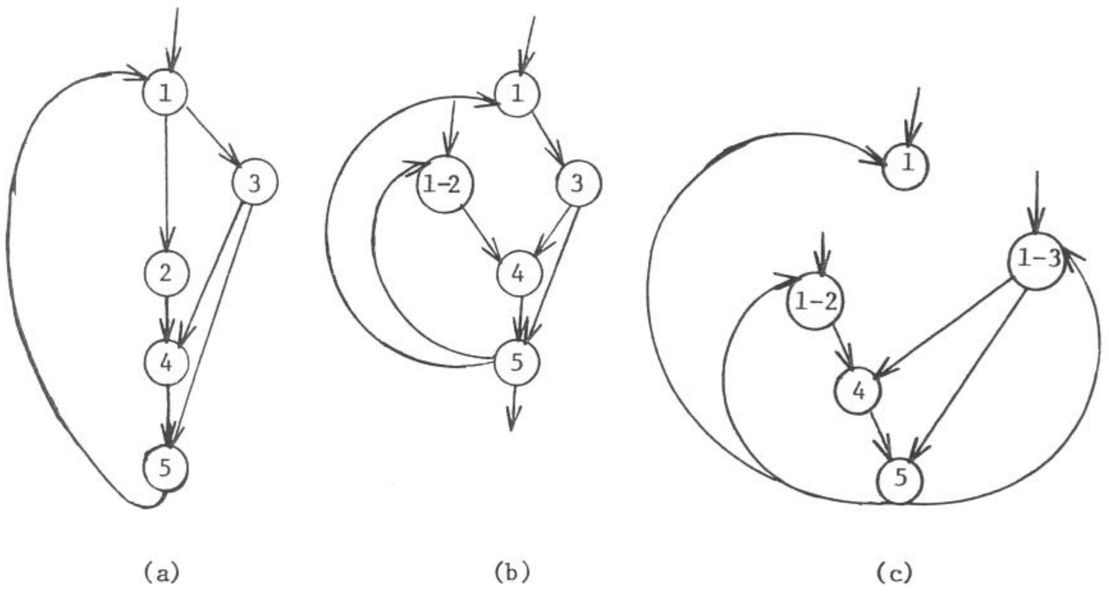
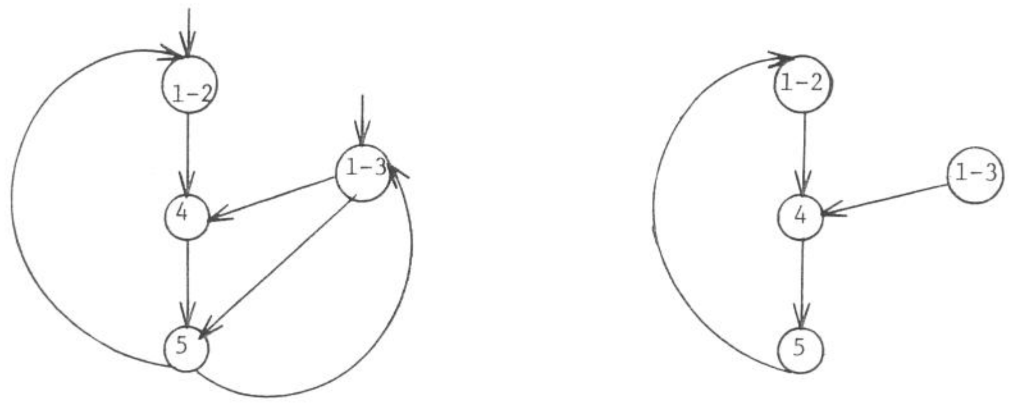


Figure 19

Nodes 2 and 3 each have one predecessor. Arbitrarily choose one of them, say 2, for coalescing with its predecessor. Figure 19(b) shows the result of coalescing nodes 1 and 2. There still aren't any 2-cycles, so nodes must again be coalesced. Picking node 3 and coalescing it with 1 we get Figure 19(c). In this graph, there is a 2-cycle ((1-3)-5). Thus we have found a 3-cycle (1-3-5). Note that node 1 is no longer in the strongly connected region and therefore need not be considered in the analysis.

Algorithm H can be used to find all of the simple cycles in the graph. However, since we are interested only in prime cycles. It is not necessary to find all simple cycles. We can avoid doing this by deleting edges from a graph of coalesced nodes whenever a 2-cycle is found. A 2-cycle in the graph of coalesced nodes is either prime or contains a prime cycle in the original graph. There is therefore no need to continue coalescing nodes involved in a 2-cycle. The edges of the 2-cycle are therefore deleted from the graph of coalesced nodes. This is the same deletion as happens when true 2-cycles are found in the original graph. It should be noted however that it is not possible to immediately identify pivots in anything but true 2-cycles. Figure 20(a) is the graph of Figure 19(c) (with extraneous node 1 omitted. Figure 20(b) shows the edges of the 2-cycle ((1-3)-5) deleted. The only strongly connected part left in the graph consists of nodes ((1-2)-4-5). Algorithm H can be applied to just those nodes and the 4-cycle found.



(a) (b)

Figure 20

As cycles are found the prime cycles can be identified simply by throwing away any cycle which covers another cycle. The prime cycles are those which are left.

The complete algorithm for finding the prime cycles in a graph is now given.

Algorithm I

This algorithm finds the set C of prime cycles in a graph.

1. Find all of the 2-cycles in the graph and record each one in C.
2. Delete the edges involved in the 2-cycles. Call this graph G'.
3. Is there still a strongly connected region in G'? If not, stop; otherwise continue.
4. Use algorithm H, the node coalescing algorithm, to obtain new cycles and a new G' graph containing coalesced nodes.
5. Delete all edges involved in these cycles from G'.
6. For each $C_i$ found in step 4 check to see if it covers or is covered by any $C_j \in C$. If $C_i$ covers any cycle already in C, $C_i$ is discarded. If $C_i$ is covered by a cycle $C_j$, $C_i$ replaces $C_j$ and $C_j$ is discarded. If $C_i$ is neither covered by nor covers any cycle already in C, then $C_i$ is added to C.
7. Return to step 3.

A slight modification to algorithm I gives an algorithm which does not find all prime cycles but, by identifying the sure pivots which exist in a 2-cycle and deleting them from the original graph, gives a graph in which the remaining prime cycles can be found more quickly.

Algorithm I'

This algorithm is a modification of algorithm I. It does not find all prime cycles in the graph but finds the sure pivots identifiable in true 2-cycles.

1. Look for any 2-cycles in the graph. If there are none, go to step 3. If there is one, check to see if it has a sure pivot. (That is check to see if one of the nodes in the cycle has only one successor, namely the other node, and if so, the node with multiple successors is a sure pivot.) Record a cycle with a sure pivot as a cycle having just one node - the pivot node. Record a cycle not having a sure pivot in the usual way.

2. Modify the graph. If the cycle found in 1 had a sure pivot, delete all nodes in the cycle and all their edges from the graph. Otherwise just delete the edges involved in the cycle. Return to 1.

3-7 Same as algorithm I.

We now have algorithms for selecting pivot nodes in an irreducible graph:

1. Algorithm I', which uses H, finds some pivot nodes and gives a set of prime cycles from which the remainder are to be selected.

2. Algorithm G completes the selection of pivot nodes.

Based upon our earlier observations about back dominating nodes and strongly connected regions, we now consider a means of restricting node splitting to subgraphs which must contain at least a subset of the total set of pivot nodes and nodes to be split. By such a restriction, we will avoid excessive node copying and will be able to restrict the node splitting analysis to subgraphs having fewer nodes. Figure 21 is an example of a graph which has been analyzed in its entirety. Later (Figure 22) only a subgraph (consisting of nodes 4-5-6) will be analyzed and fewer nodes will be split.

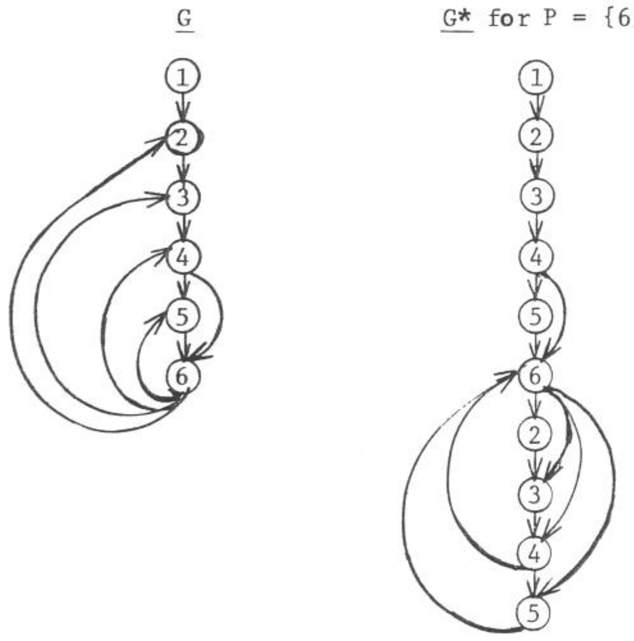


Figure 21

Consider the set T of back dominating nodes which do not back dominate any other back dominators. Since the back dominators in any control flow graph are partially ordered by the immediate back dominance relationship, they can be represented by a tree whose root node is the single entry node of the graph. Let the leaf or terminal nodes of the back dominance tree form the set T. By assertion A3 we know that in an irreducible graph each element of T must back dominate a set of nodes at least some of which are in multiple entry cycles. If $S_t$ is the subgraph back dominated by $t \in T$, then $S_t$ must contain at least a subset of the total set of pivot nodes in G. By treating

$t$ as an $e_o$ and $G_t$ as a total graph consisting of $t$ and $S_t$ but having no edges from $S_t$ to $t$ we will have isolated a graph to which all previous assertions, constructions and observations can be applied. Pivot nodes can be found in it and the graph simplified - effecting a simplification of the entire graph G.

Since the subgraphs back dominated by the elements of T, the terminal back dominating nodes in the graph, are disjoint, it is clear that each one of these subgraphs must eventually reduce to a single node if the entire graph is to reduce. Indeed a possible node splitting algorithm would be to identify and reduce to a single node each $G_t$. Having replaced each $t \in T$ and its subgraph by a single node, we could then reconsider the entire graph. The graph might then be reducible but if it were not a new set of terminal back dominating nodes would exist and the procedure could be repeated. The actual algorithm is based upon these observations but attempts to reduce the entire graph each time the $G_t^*$ subgraphs are being reduced. In Figure 22 the back dominating nodes are 1,2,3 and 4 with 4 as the only terminal backdominator. $T = \{4\}$.

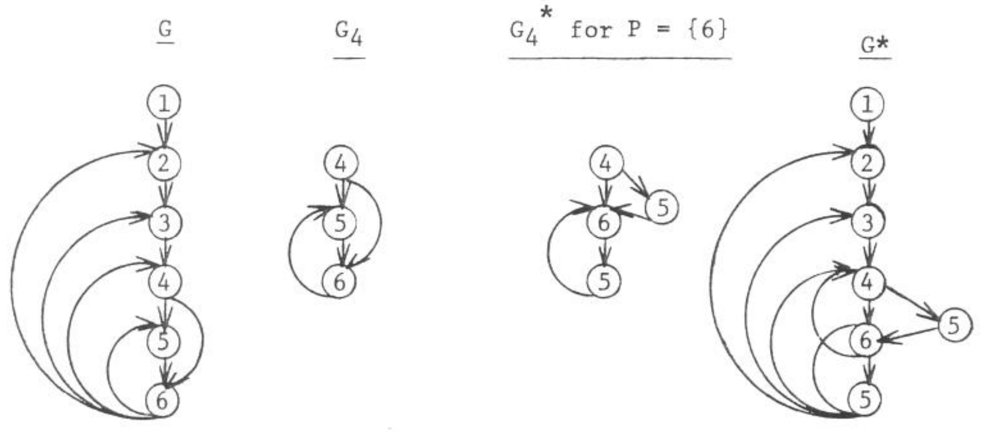


Figure 22

G* completely reduces without further node splitting.

Since the pivot nodes must be in prime cycles, it is not necessary to consider the entire subgraph back dominated by an inner back dominator $t \in T$. We can consider just the maximal strongly connected regions in $S_t$. Indeed each such region can be treated independently. By assertion A4 we know that one of the maximal strongly connected regions back dominated by $t \in T$ must have $t$ as its only immediate predecessor. Call such a region $R_t$.

Assuming an $R_t$ has been selected, we now give the construction of G* after the pivot nodes P in $R_t$ have been determined:

<u>C2</u>: Given $P^+ = P \cup t$, $R_t$ and G, the graph G* is constructed as follows:

1. For each $p \in P^+$ construct $G_p$ as in step 1 of C1 but with the restriction that only $t$ and the nodes in $R_t$ (i.e., in the strongly connected region) are in $G_p$.
2. Form $G^*_t$ from the $G_p$ subgraphs. This is done as in step 2 of C1.
3. Form G* by a process which can be viewed as replacing $t$ and $R_t$ in G by $G^*_t$.

In Figure 23 t = 1, P = {4}.

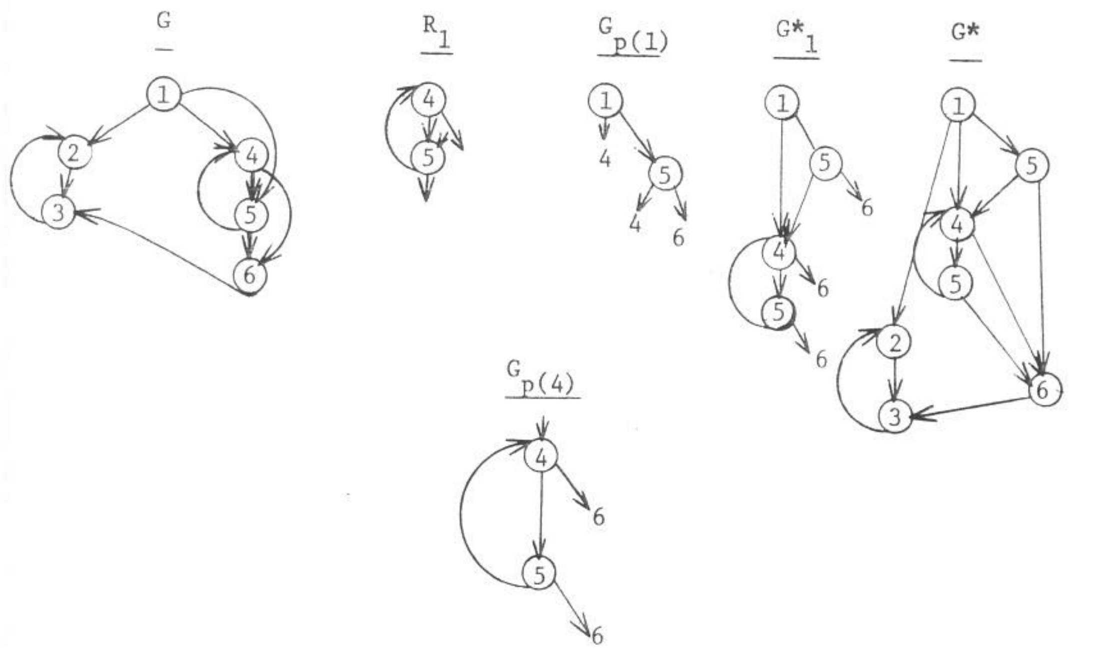


Figure 23

We now give a node splitting algorithm.

Algorithm J: Node Splitting

This algorithm transforms an irreducible graph G into an equivalent graph G* containing non-trivial intervals.

1. Find the set of innermost back dominators, T, in the graph G by finding all of the back dominators of each node (by Algorithm A) and then eliminating all that back dominate other back dominators.

2. For each inner back dominator, $t \in T$, find the set of disjoint strongly connected regions which contain the immediate successors

of t. The implemented procedure works by tracing from t and stopping on any node not back dominated by t and on any immediate successor of t which has already been found to be in a strongly connected region.

3. Having found the set of strongly connected regions for t, pick the one which does not contain any predecessors outside the strongly connected region (other than t). Call this $R_t$.

4. Find the prime cycles in the selected strongly connected region, $R_t$, by algorithm I'.

5. Find the set of pivot nodes P in $R_t$ by algorithm G.

6. After all pivots have been found for this strongly connected region, repeat the process for the next unprocessed $R_t$ then repeat for the next $t \in T$.

7. The graph G* is constructed from the set of pivot nodes by the construction given in C2.

We conclude this section of the paper with two examples.

Applying Algorithm J to Figure 11 - the step-by-step result is:

1. The back dominating nodes in Figure 11 are 1,2,3,4 with 4 as the only innermost back dominator. Therefore $T = \{4\}$.

2. There is only one strongly connected region back dominated by node 4. It is shown in Figure 24.

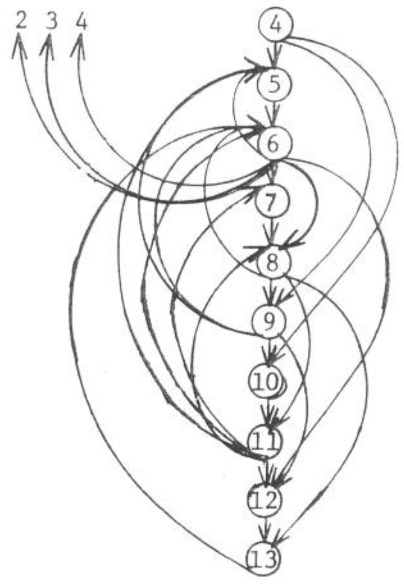


Figure 24

3. The region found in step 2 is, of course, the region which will be used.

4. The 2-cycle (5,6) is found and 6 is identified as being a sure pivot. Since 6 is a sure pivot, both 5 and 6 are deleted from the graph being analyzed. Therefore, the two cycle (6,8) is never found. The other two cycle (8,11) is found and the edges involved in the cycle are deleted from the graph. Although either 8 or 11 will eventually be a pivot, we have no way of selecting one at this time. The graph now appears as shown in Figure 25.

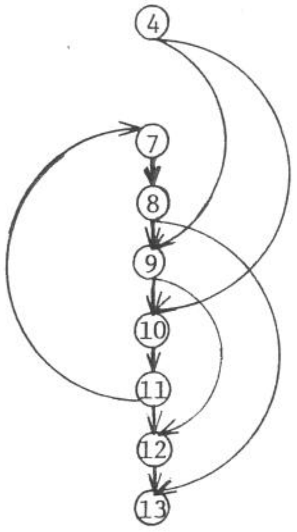


Figure 25

The cycle (7,8,9,10,11) is found by the coalescing process described in algorithm H which is used by algorithm I'. However by step 7 of algorithm I' the cycle (7,8,9,10,11) covers cycle (8-11) and is discarded.

5. The prime implicant table is

| Nodes / Cycles | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|
| $C_1$ | | x | | | | | | | |
| $C_2$ | | | | x | | | x | | |

Cycle $C_1$ which consisted of nodes 5 and 6 is represented only by 6, a sure pivot, in the table. Node 8 is covered by node 11 and 11 by 8. We arbitrarily delete one of them from the table, say 8.

6. The pivots are therefore 6 and 11 - which together break every cycle in the region.

7. Since there was only one terminal back dominating node, node 4, the analysis is complete and the equivalent graph is generated. Because of its size, it is not shown: it contains 25 nodes - nodes 1,2,3,4,6 and 11 occur once, nodes 7 and 8 occur twice and nodes 5,9,10,12 and 13 occur three times. The second derived graph is again irreducible but much simpler.

Algorithm J is now applied to the graph in Figure 26

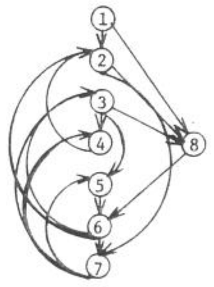


Figure 26

The only back dominating node is 1 so the whole graph is to be analyzed. There are no 2-cycles but five 3-cycles are found in the order given in the prime implicant table which is

| Nodes / Cycles | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| $C_1$ | x |  |  |  | x |  | x |
| $C_2$ |  | x |  |  | x |  | x |
| $C_3$ |  |  |  | x | x | x |  |
| $C_4$ |  | x |  | x | x |  |  |
| $C_5$ | x |  | x |  |  | x |  |

In applying algorithm G we get that:

node 3 is covered by node 6 and can be deleted.

node 4 is covered by node 2 (and 7) and can be deleted.

node 5 is covered by node 6 and can be deleted.

node 8 is covered by node 6 and can be deleted.

The table is now

| Nodes / Cycles | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| $C_1$ | x | | | | x | | |
| $C_2$ | | | | | x | | |
| $C_3$ | | | | | x | x | |
| $C_4$ | | | | | x | | |
| $C_5$ | x | | | | | x | |

Since cycles $C_2$ and $C_4$ each contain a single node, node 6, it must be a pivot node. It is put in P and all cycles containing it are deleted from the table. This leaves only one cycle, $C_5$, in the table.

| Nodes / Cycles | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| $C_5$ | x | | | | | x | |

Node 7 is covered by node 2 (and 2 by 7); we eliminate one, say 7. The pivot nodes are therefore 2 and 6. Note that 2 and 6 (or 7 and 6) break every cycle in the graph.

## SOME RESULTS

Seventy-two FORTRAN IV programs and subprograms were analyzed by a program which found basic blocks and intervals. The 72 programs were randomly selected, running programs. The program characteristics and the results of the analysis are now summarized.

1. Number of source statements in a program (excluding comments):

   Range: 11 to 83

   Average: 122

   Median: 60

2. Number of basic basic blocks

   Range: 2 to 274

   Average: 38

   Median: 22

3. Source statements per basic block (Note: statement count includes declaratives):

   Range: 1.20 to 42

   Average: 4.97

   Median: 2.84

4. Number of iterations through the interval analyzer for the 67 fully reducible programs (i.e. order of the final single node graph)

   Range: 1 to 9

   Average: 2.85

   Median: 3

5. Maximum number of basic blocks in a first level interval

| | |
|---|---|
| Range: | 2 to 81 |
| Average: | 10.66 |
| Median: | 7 |

6. The five programs which were not fully reducible had the following characteristics:

| | No. of Statements | No. of Basic Blocks | Order of Irred. graph | No. of blocks in irred. graph |
|---|---|---|---|---|
| 1 | 629 | 204 | 3 | 3(Fig. 10) |
| 2 | 438 | 109 | 2 | 5 |
| 3 | 838 | 274 | 5 | 13(Fig. 11) |
| 4 | 68 | 30 | 2 | 8 |
| 5 | 254 | 151 | 5 | 11 |

## SUMMARY

Methods for the static global analysis of program control flow graphs were given. Based on some basic concepts from graph theory, a new back dominance algorithm was given, and graph partitioning into intervals was described. Since a hierarchical partition of a graph into successively more global regions does not always result in a complete reduction of a graph into a simple region, a graph transformation may be needed to permit such a reduction. A node splitting algorithm was given which will transform an irreducible graph into an equivalent graph which can be further reduced. Finally, the results of applying the interval analysis to 72 FORTRAN programs were given.

ACKNOWLEDGEMENTS

Dr. Raymond Miller of IBM and Mr. Richard Sites of Stanford University contributed to the node splitting solution. Many aspects of intervals were first formalized by Dr. J. T. Schwartz. The authors wish to thank all of these people.

## REFERENCES

1. Allen, F. E., "Program Optimization", Annual Review in Automatic Programming, Vol. 5, Pergamon, New York, 1969.

2. Allen, F. E., "Control Flow Analysis", Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, July 1970.

3. Allen, Frances E., "A Basis for Program Optimization" Proceedings of IFIP Congress 71. To be published by North Holland, Amsterdam, Holland.

4. Berge, C., The Theory of Graphs, Methuen & Co., Ltd., London, 1964.

5. Cocke, John, "Global Common Sub-Expression Elimination," Proc. of a Symposium on Compiler Optimization, SIGPLAN Notices, July 1970.

6. Cocke, John and Miller, Raymond, "Some Analysis Techniques for Optimizing Computer Programs," Proc. Second Intl. Conf. of Systems Sciences, Hawaii, Jan. 1969.

7. Cocke, John and Schwartz, J. T., "Programming Languages and their Compilers", Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, N.Y., April 1970.

8. Cocke, J., "On Certain Graph-Theoretic Properties of Programs", IBM Research Report RC 3391, T. J. Watson Research Center, Yorktown Heights, N. Y., June 1971.

9. Earnest, C. P., Balke, K. G. and Anderson, J., "Analysis of Graphs by Ordering of Nodes", JACM, Jan. 1972, pp. 23-42.

10. Hecht, M. S. and Ullman, J.D., "Flow Graph Reducibility," Proceedings Fourth Annual ACM Symposium on Theory of Computing, May 1972.

11. Knuth, Donald E., "An Empirical Study of FORTRAN Programs", Report No. CS-186 Computer Science Dept., Stanford University, Stanford, California.

12. Lowry, Edward S. and Medlock, C. W., "Object Code Optimization", CACM, Jan. 1969, pp. 13-22.

13. Mendicino, Sam. F., et al., "The LRLTRAN Compiler," CACM, Nov. 1969, pp. 747-755.

14. Petrick, S. R., "A Direct Determination of the Irredundant Forms of a Boolean Function from the Set of Prime Implicants", USAF Cambridge Research Center, Bedford, Mass., Tech. Report AFCRL-56-110, April 1956.

15. Petrick, S. R., "On the Minimal Covering Problem", USAF Cambridge Research Laboratories, Bedford, Mass., Tech. Report AFCRL-63-148, June 1963.

16. Prosser, R. T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams," Proc. Eastern Joint Computer Conf., December 1959, Spartan Books, New York, pp. 133-138.