Control Flow Analysis

Frances E. Allen
IBM CORPORATION

INTRODUCTION

Any static, global analysis of the expression and data relationships in a program requires a knowledge of the control flow of the program. Since one of the primary reasons for doing such a global analysis in a compiler is to produce optimized programs, control flow analysis has been embedded in many compilers and has been described in several papers. An early paper by Prosser [5] 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 [6]. References [6,8,9] describe compilers which use various forms of control flow analysis for optimization. Some recent developments in the area are reported in [4] and in [7].

The underlying motivation in all the different types of control flow analysis is the need to codify the flow relationships in the program. The codification may be in connectivity matrices, in predecessor-successor tables, in dominance lists, etc. Whatever the form, the purpose is to facilitate determining what the flow relationships are; 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 first section of the paper, "Basic Concepts," is primarily a catalog of relevant information about directed graphs; similar information can be found in any introductory material on the subject. (Reference [2], for example, covers this material.) The use of directed graphs to express control flow relationships is also given in the first section.

In the second section of the paper, "Dominance Relationships" are defined in terms of the basic concepts introduced in the first section. Most of the concepts in this section have, as previously

mentioned, appeared in the literature before.[1,5,6]

The third section, "Intervals," discusses a graph construct defined by Dr. John Cocke and described in [3] and [4]. In this section intervals are defined, a procedure is given for their construction, and their properties are given in a series of assertions. Also discussed are procedures for finding other graph constructs in terms of the interval constructs.

The fourth section, "Partitioning Graphs by Intervals," describes a hierarchial sequence of graph partitions by means of intervals.

The last section before the summary gives a procedure and an example of "The Use of the Interval Construct in Global Analysis."

BASIC CONCEPTS

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. Thus, there exists a successor function Γ_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. It may be empty. The inverse of the successor function Γ_G^{-1} gives the immediate predecessors of a node: $\Gamma_G^{-1}(b_j) = \{b_i \mid (b_i,b_j) \in E\}$. It too may be empty.

A directed graph is connected if any node in the graph can be obtained (reached) from any other node by successive applications of Γ_G^1 and/or Γ_G^{-1}. We wll 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 basic block 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 predessors and many successors and may even be its own successor. Program entry blocks might not have predecessors that are in the program; program terminating blocks never have successors in the program.

A control flow graph 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 subgraph 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 $G \cup G' = G$. Furthermore, the successor function $\Gamma_{G'}^1$, defined for G' must "stay within" G'; that is for

$$b_i' \in B',\ \Gamma_{G'}^1(b_i') = \{b_j' \mid (b_i',b_j') \in E'\}.$$

Consider the following directed graph, G, in which the nodes have been arbitrarily named by numbering.

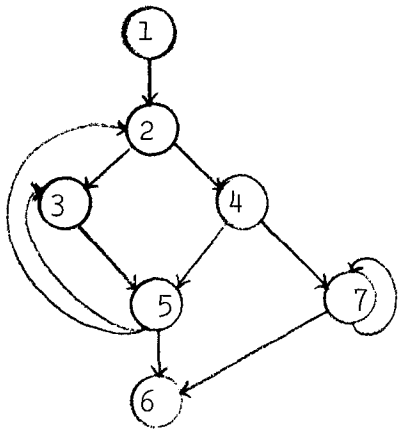
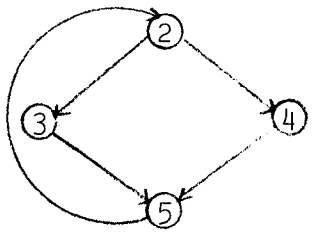


Fig. 1

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 $b_{i+1} \in \Gamma_P^i(b_i)$. The edges are implied: $(b_i,b_{i+1}) \in E$. The nodes and the implied edges are not necessarily unique. A path in G, the graph in Fig. 1, is (2,3,5,3,5,2,4). It should be observed that the examples show how some of the developed notation is to be used: the nodes of the graph are arbitrarily but uniquely named and b stands for any such name.

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 circuit is a path in which $b_n = b_1$. The circuit is a simple circuit if, with the exception of b_n, the nodes in the circuit are distinct; otherwise it is a composite circuit. Consider the graph in Fig. 1: it has the following simple circuits: (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 circuits is (2,3,5,3,5,2). Since it will generally be uninteresting to consider circuits containing the same nodes and edges but in a different order, we will generally select a first node and describe the circuit relative to that node.

The length of a path is the number of edges in the sequence. More formally, a distance function δ is defined such that for any path $P = (b_1,b_2,\ldots,b_n)$, $\delta(P) = n-1$. Since the shortest path δ_{min} between two points p and q is often of interest it will now be defined: $\delta_{min}(p,q) = MIN(\delta(P_1),\delta(P_2)\ldots)$ 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 (circuits) are, therefore, a special kind of strongly connected region -- one which has a strict ordering. A strongly connected region R of a directed

graph G is maximal if there does not exist another strongly connected region, R', in G for which $R \cap R' \neq \emptyset$. 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 $R_i \cap R_j = 0$ 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 connection regions is found:

$$D = \{\{R_1, R_2, \ldots, R_n\}, \{R_1', R_2', \ldots, R_n'\}, \ldots\}$$

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.

As an example consider the graph in Figure 1: $D = \{\{(3,5),(2,3,4,5)\},\{7\}\}$. Since much of the control flow analysis involves knowning relationships between nodes in the control flow graph, the construct, D, codifies several useful relationships. However it has several limitations, the more serious of which are that it does not establish an ordering on the total graph and that, 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, $\Gamma_G^1(x) = \emptyset$. This definition suffices for control flow graphs but, since a program entry point may also be the first node in a closed path and thereby have a predecessor, an analogous definition for entry nodes does not suffice. An entry node, e, is a node in the program control flow graph, C, if it 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:

$$\Gamma_C^1(e_0) = \{e_i \mid e_i \text{ is an entry node}\} \quad \text{and} \quad \Gamma_C^{-1}(e_0) = \emptyset.$$

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 should look at reference [6].)

A node, b_i, is said to back dominate or predominate 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 set of back dominators $BD(b_k)$ of b_k consists of all of the blocks, other than b_k itself, which are on all paths from e_0 to b_k. In other words

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

The immediate back dominator 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 now be shown that there is one and only one immediate back dominator of a node $b_k \neq e_0$. For 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_k 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 k are strictly ordered by the minimum distance function δ_{min}. 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 have one and only one immediate back dominator.

The set of back dominators of node b_k can be represented by

$$BD(b_k) = (b_1, b_2, b_3, \ldots, b_j) \text{ where } b_1 = e_0 ,$$

b_i is the immediate back dominator of b_{i+1} and b_i is the back dominator of all b_j, $i < j \leq k$.

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, the set of forward dominance relationships analogous to the back dominance relationships can be developed. Because 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 articulation node 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.

INTERVALS

Given a node h, an interval 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 interval head or simply the header node. 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).

By selecting the proper set of header nodes, a graph may be partitioned into a unique set of intervals. (A partition 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 $i \neq j$, $g_i \cap g_j = \emptyset$. Thus a graph partition covers the original graph with a set of disjoint subgraphs.) A procedure for partitioning a graph, G, into a unique set of intervals is now given:

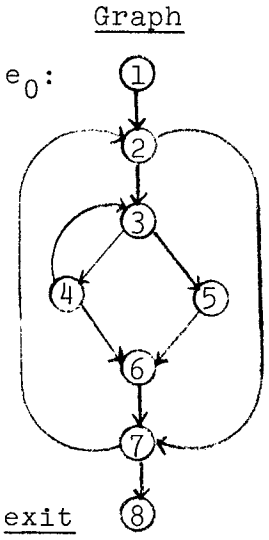
Procedure A.

1. Establish a list H for header nodes and initialize it to 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 For any $b \in G$ for which $\Gamma_G^{-1}(b) \subset I(h)$ add b to I(h).
Thus a node is added to an interval if and only if all of its immediate predecessors are already in the interval.
2.3 Repeat 2.2 until no more nodes can be added to I(h).
3.1 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 are members of an interval.
3.2 Add I(h) to the set of intervals being developed.
4. Select the next unprocessed node in H and repeat steps 2,3,4.
If 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 procedure, a few comments on the procedure itself may be of interest. In a program written by the author to implement this procedure, indicators were left on each node as to whether or not it was in H, and if not in H, a count was kept of the number of times it had been looked at during the development of the current interval. This latter count was 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 determined whether or not it could become a member of the current interval. Using such techniques an edge in the graph will never be traversed more than once. Thus the execution time for the procedure is directly proportional to the number of edges in the graph.

The following example illustrates the partitioning of a graph into intervals:



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 2

It will now be shown that the procedure given does indeed produce a set of intervals each of which satisfies the definition for an interval. It will later be shown that they collectively provide a unique partition of the graph. Thus we need to show that any I(h) is maximal, is single entry, and that all closed paths in I(h) contain h.

Assertion 1. I(h) has only one possible entry node, h. Suppose there was another node b ε I(h), b $\neq$ h, which was also an entry node. Then b must have at least one immediate predecessor which is not in I(h). But this is impossible since b became a member of the interval only when all of its immediate predecessors were already interval members. Hence there can be only one possible entry node. It should be further noted that h $\neq e_0$ will have at least one predecessor outside the interval since by step 3 in the procedure it became a header node because it had a predecessor in an interval to which it did not belong.

Assertion 2. All closed paths in I(h) contain h. Suppose there is a closed path P = $(b_1,b_2,\ldots,b_n,b_1)$ which does not contain h. By the notational definition established for paths b_{i-1} is an immediate predecessor of b_i. Hence b_i cannot become a member of I(h) until b_{i-1} is a member. Also b_1 cannot become a member until b_n does, and b_n cannot become a member until b_{n-1} does, etc. Therefore all closed paths in I(h) must contain h.

Assertion 3. I(h) is maximal. This follows from step 2.3: nodes are added to I(h) until no more can be.

Some properties of intervals which result from the construction in procedure A are now given.

Assertion 4. The header node of an interval back dominates every node in the interval. Since by Assertion 1, the only possible entry to an interval is through the header node, the header node must lie in every path from e_0 to any block in the interval.

A somewhat restricted successor function L_I^1 is now defined for the interval I(h): a local successor function, $L_I^1(b_i)$ is defined for I(h) such that for $b_i \in I(h)$ $L_I^1(b_i)$ is the set of all immediate successors of b_i which are in the interval but are not the header node. In other words

$$L_I^1(b_i) = \{b_j \mid b_j \in \Gamma_I^1(b_i) \text{ and } b_j \neq h\} .$$

The local predecessor function is the inverse of L_I^1 i.e. is L_I^{-1} in which $L_I^{-1}(h) = \emptyset$.

Using the local successor function a special type of interval path can be defined: a forward path is a path $F = (b_1, b_2, \ldots, b_n)$ where $b_{i+1} \in L_I^1(b_i)$. It can be noted that all nodes on all forward paths from h to any node in I(h) are also in I(h).

Assertion 5. The nodes in an interval are partially ordered by the local successor function. Given an interval $I(h) = (b_1(=h), b_2, b_3 \ldots, b_n)$ if $i < j$ then either b_i is a predecessor of b_j on some forward path or b_i and b_j do not co-exist on any forward path. This follows from the fact that, with the exception of h, all immediate predecessors of a node must be interval members before the node can become a member.

Assertion 6. The relative ordering of the nodes in a back dominator list and the nodes in an interval must be the same. If b_i is a back dominator of b_j and both are in an interval I(h), clearly b_i must precede b_j in the interval list because it is impossible to reach b_j without having first reached b_i.

Assertion 7. For any interval member $b_k \neq h$ with back dominator list $BD(b_k) = (b_1(=e_0), b_2, \ldots, b_j)$ then for $b_i = h$, $b_i \in BD(b_k)$ and all blocks b_ℓ following b_i on the back dominator list ($i < \ell \leq j$), b_ℓ is a member of the interval. A back dominator b_ℓ must be on all paths from e_0 to b_k. Since it follows h on the back dominator list it must be on all paths and, hence all forward paths, from h to b_k. Therefore it must be an interval member.

Assertion 8. Any strongly connected region in an interval must contain the interval head. This follows immediately from the fact that all closed paths in I(h) must contain h. An interval cannot, therefore, contain disjoint strongly connected regions.

Assertion 9. 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. Since the header node both back dominates every node in the interval and is in the strongly connected region, and since there is a path from any node in a strongly connected region to any other node, there must be a path from every node in the region to every node in the interval. Unless the entire interval is strongly connected, it will not be the case that there exists a path from every node in the interval to every node in the region.

For example in interval I(3) in Figure 2, there are paths from node 4 to every node in the interval.

As a consequence of 9, it should be noted that there can be a path from b_j to b_i when b_i precedes b_j on the interval list. If there is such a path then b_j must be in the strongly connected region. It is still true however that there does not exist a forward path in which b_j is a predecessor of b_i.

Consider the following interval with header node 1 and exit node 6.

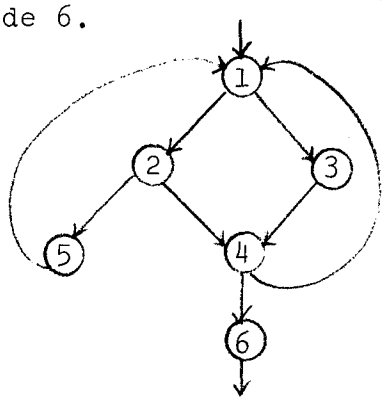


Figure 3

Assume
I(1) = (1,2,5,3,4,6)

Assume the order in which nodes have become interval members results in I(1) = (1,2,5,3,4,6). Clearly there are paths from 4 to all of its predecessors in the interval list.

Before making the next assertion, 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).

Assertion 10. The interval header is an articulation node for the interval. Since the header node is the only entry to the interval it must be on every entry-exit path. An interval header is not necessarily an articulation node for the total graph.

Assertion 11. All forward dominators of the interval header which are also interval members are, along with the header, the articulation nodes for the interval. This assertion can be shown by exactly the same reasoning that led us to assert that the forward dominators of e_0 are the articulation nodes, and together with e_0, the only articulation nodes of the total graph.

The articulation nodes of the interval in Figure 3 are 1, 4 and 6 since they are on every entry-exit path.

In certain applications a special graph construct called a "two-terminal subgraph" may be of interest. Defined in terms of intervals, a two-terminal subgraph 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 3 is an example of a two-terminal subgraph.

Procedures 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 procedures can be embedded in procedure A thereby generating not only the intervals but their interval 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_i 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 procedures to be 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 procedures.

The next procedure 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 (by Assertion 6) 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.

Procedure B.

This procedure finds the back dominators of each node in an interval.

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 procedure is embedded) form $BD(b_j) = \bigcap_i (b_i \cup BD(b_i))$ 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 3 for which $I(h) = (1,2,5,3,4,6)$. The procedure generates the following back dominator list for each node by the operations shown.

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

Example 1

In the next procedure, C, the interval articulation nodes are found by using the back dominators of interval exits. The result of

procedure C is a list, A, of articulation nodes for the interval.

Procedure C.

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

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

Consider the interval of Figure 3 and the back dominator lists given in Example 1. Since node 6 is the only interval exit, $A = 6 \cup (1,2) = 1,2,6$. If node 5 were also an exit then $A = [5 \cup (1,2)] \cap [6 \cup (1,2)]$ and the articulation nodes would be 1 and 2.

The next procedure, procedure D, finds all of the local predecessors of a node.

Procedure D.

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

1. Assign the interval head a local predecessor list of zero: $LP(b_1) = 0$.
2. For the next node, b_j, in the interval list $LP(b_j) = \bigcup_i (b_i \cup LP(b_i))$ 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 example in Figure 3 the following LP lists are generated by procedure D.

Nodes	Imm. Pred.	Operation	LP Lists
1	-	(Assignment)	0
2	1	$1 \cup 0$	1
5	2	$2 \cup 1$	1,2
3	1	$1 \cup 0$	1
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 2

The next procedure, E, uses the results of procedure D 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 3 nodes 4 and 5 are latching nodes. It should be noted that the interval head itself can be a latching node. From previous assertions it follows that if the interval does not contain any latching nodes then the interval does not contain a strongly connected region. The following procedure then would be invoked only if the interval had at least one latching node.

Procedure E.

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

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

Using the results of Example 2, we get SCR=[4∪(1,2,3)]∪[5∪(1,2)]. Therefore the strongly connected region of the interval in Figure 3 is comprised of the nodes (1,2,3,4,5).

Another procedure, E', 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 procedure it is not necessary to continue the marking of its immediate predecessors since they will already have been marked. This procedure has the advantage of not requiring that the LP lists be set up and is probably preferable if the only use of LP lists is to find the strongly connected region.

A formal description of procedure E' is not given; the above informal description should adequately suggest such a description.

PARTITIONING GRAPHS BY INTERVALS

Having considered the properties of any given interval, it will now be shown that the set of intervals $\mathcal{P} = \{I(h_1), I(h_2), I(h_3)...\}$ generated by procedure A forms, as asserted, a unique partition of the graph G. Recalling the definition of a partition, we therefore need to show that $\mathcal{P}$ covers G and that for any two intervals $I(h_i)$ and $I(h_j)$ in $\mathcal{P}$, $I(h_i) \cap I(h_j) = \emptyset$. Furthermore we want to show that $\mathcal{P}$ is unique.

Assertion 12. $\mathcal{P}$ covers G. Suppose there is a b ε G which is not in any I(h) ε $\mathcal{P}$. Since G is a connected graph, b must either be e_0 or have at least one predecessor. But if b = e_0 it is an element of $I(e_0)$, the first interval constructed.

If b ≠ e_0, then, since it must have at least one predecessor, by step 2.2 it must become a member of the interval containing the predecessor or must, by steps 3 and 4, become an interval head. (In order to establish that the predecessors must be members of intervals we can recursively apply the above reasoning until b = e_0). Hence covers G.

Assertion 13. The elements of $\mathcal{P}$ are disjoint, that is for any I(h) and I(h'), elements of $\mathcal{P}$, $I(h) \cap I(h') = \emptyset$.

Assertion 13a. The header nodes must be distinct, that is for any h and h', h ≠ h'. By step 3 in Procedure A a given node can appear at most once in H and by step 4 a node in H can be processed (used to head an interval) only once.

Assertion 13b. The header node of one interval cannot be an element of another interval. Suppose interval head, h, is an element of interval I(h'). By 13a, h ≠ h'. Therefore all immediate predecessors of h must, by step 2.2, also be in I(h'). But for h to have become an interval head some but not all of its immediate predecessors must have been members of an interval, say I(h"). We will now show that h" must also be an element of I(h'). Consider an immediate

predecessor b of h which is in both intervals I(h') and I(h"). b is back dominated by both h' and h" and since the back dominators of a block are strictly ordered either h' back dominates h" or vice versa. But since h has predecessors which are not in h", h" cannot back dominate h. We can therefore conclude that h' back dominates h". By assertions 6 and 7 h" must be an element of I(h'). Proceeding inductively h must eventually equal h' which by Assertion 13a is impossible. Hence it is not possible for a header node to be an element of an interval.

Assertion 13c. The intersection of any two intervals is null. Suppose there is a b ϵ I(h) $\cap$ I(h'). By assertions 13a and 13b we know that b is not a header node of any interval including I(h) and I(h'). But for b to be in the intersection it must be a member of each interval. Hence all of the immediate predecessors of b must be in both intervals and, as a consequence, they must also be in the intersection of the two intervals. Proceeding inductively we must eventually find an interval header in the intersection which is impossible by 13a and 13b. The elements of $\mathcal{P}$ must therefore be disjoint.

Assertion 14. $\mathcal{P}$ = {I(h),I(h') ...} is unique. Having shown that each I(h) is maximal and that elements of $\mathcal{P}$ are disjoint, it is sufficient to show that the set of header nodes H is unique. Clearly the first header node, e_0, is always in the set of header nodes. Since $I(e_0)$ is maximal the set of nodes which become members of H after the construction of $I(e_0)$ is unique. Pick any h ϵ H and construct I(h). Again because intervals are maximal, the set of nodes in the graph which are immediate successors of nodes in I(h) but are not themselves in I(h) is unique. (Some of these immediate successors may however already be in H because they have immediate predecessors in intervals which were constructed before I(h) and indeed they may already have been used to construct intervals.) Since we are able to pick any node in H and, after interval construction, find a unique set of header nodes, the order of processing H does not affect the header nodes found. It follows therefore that, after e_0, header nodes can be added to H in any order. By induction we claim that the header nodes are unique and therefore $\mathcal{P}$ is unique.

Having described the relationships 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 interrelationships in larger sets of nodes can be derived.

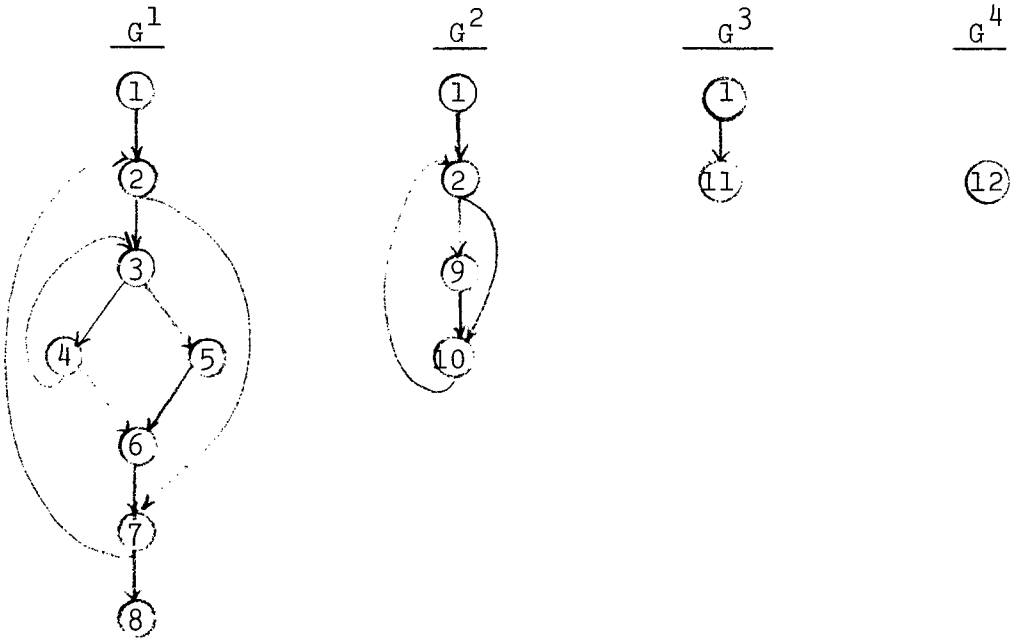
The intervals described thus far have been formed from the elemental nodes of the graph (the basic blocks of the control flow graph). For reasons which will be apparent shortly, we designate these intervals as the basic or first order intervals and the graph from which they were derived as the basic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

A second order graph is derived from the first order graph and 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 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 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 only multi-node intervals are renamed in the derived graph.



$\mathcal{I}^1$

$I^1(1) = 1$
$I^1(2) = 2$
$I^1(3)=3,4,5,6$
$I^1(7)=7,8$

$\mathcal{I}^2$

$I^2(1) = 1$
$I^2(2)=2,9,10$

$\mathcal{I}^3$

$I^3(1)=1,11$

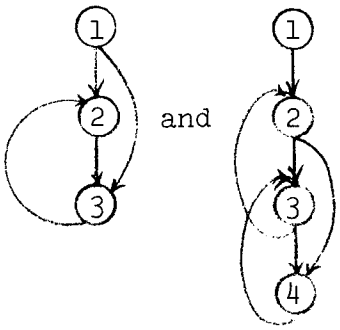
$\mathcal{I}^4$

$I^4(12) = 12$

Figure 4

A reducible graph is a graph whose n-th order derived graph is a single node. An irreducible graph is a graph for which there does not exist an n-th order derived graph consisting of a single node. This will happen whenever every interval in a graph is composed of only one node and contains no internal flow paths.

Examples of irreducible graphs are



A method for "splitting" an irreducible graph is given in reference [10]. By this method an equivalent, reducible graph is produced. It may be of interest that a program written by the author to analyze the control flow of FORTRAN programs found that over 90% of the control flow graphs were reducible. (The data consisted of 75 "real" programs.)

Assuming graph G^1 is eventually reduced to a single node several interesting observations can be made.

1. Every node in G^1 is in one and only one interval $I^1(h)$ in $\mathcal{G}^1$ which is in turn a node in G^2 and hence in one and only one interval $I^2(h)$ in $\mathcal{G}^2$, etc.

2. Therefore for a given basic block, a unique, strictly ordered set of membership in successively higher order intervals exists: $b_i \in I^1(h_1) \in I^2(h_2) \in \ldots \in I^n(h_n)$.

3. Because the nodes in an interval are partially ordered by the local successor function the nodes in the entire graph are ordered by this function. This may be depicted by:

$$I^n \left\{ \begin{array}{ll} I_1^{n-1} & \left\{ \begin{array}{ll} I_1^{n-2} & \left\{ \begin{array}{l} e_0 \\ b_1 \\ \vdots \\ b_2 \end{array} \right. \\ \vdots & \end{array} \right. \\ I_2^{n-1} & \left\{ \begin{array}{l} \\ \\ \\ h \end{array} \right. \\ \vdots & \\ I_m^{n-1} & \left\{ \begin{array}{l} \\ \\ \\ b_n \end{array} \right. \end{array} \right.$$

THE USE OF THE INTERVAL CONSTRUCT IN GLOBAL ANALYSIS

Recalling the the nodes in a graph represent the basic blocks of a program and therefore contain instructions, some uses of the interval construct in global analysis will now be sketched. The primary purpose of the skeletal procedure given to to show how some of the interval relationships may be applied to any one of many types of analyses. The analysis might typically involve looking for redundant instructions, determining variable definition and use relationships, etc.

Procedure F (Skeletal).

The use of intervals in global analysis is sketched by this procedure:

1. Process each basic block in the program, collecting information of global interest at the entry and exit. Set the order number, k, to 1.

2. For each k-order interval:

2.1 Proceed through the blocks in the interval in their interval order. For each block the information previously collected (either by step 1 or by the last iteration of step 2) is first modified to reflect the effects of interval predecessors then promulgated to interval successors.

2.2 After the completion of step 2.1, the information collected at the exits of latching nodes (if there are any) must be promulgated. This may require redoing step 2.1 after information on entry to the interval head has been modified by the information on the latching nodes.

As a result of step 2 information of global interest is left at the interval head and at the exits.

3. The information collected by processing an interval is associated with the node which represents it in the next higher order graph. The order number, k, is increased by 1 and steps 2 and 3 repeated until the n-th order graph is reached. (We are assuming it consists of a single node and no edges so does not need processing.)

4. Set k to n-2 to initialize for steps 5 and 6 which will propagate the information deposited with each node in each of the graphs back to the basic blocks.

5. Associate with the head of each k-order interval the information left at the node which corresponds to it in the k+1 order graph.

6. For each k-order interval proceed through the blocks in interval order promulgating the information from the head of the interval to each node in the interval.

7. Reduce k by one and repeat steps 5 and 6 until the first order intervals have been processed.

8. Whatever global information has been carried through steps 1 through 7 is now available at each block.

Consider the graph in Figure 4, redrawn here with some variable definitions associated with some nodes.

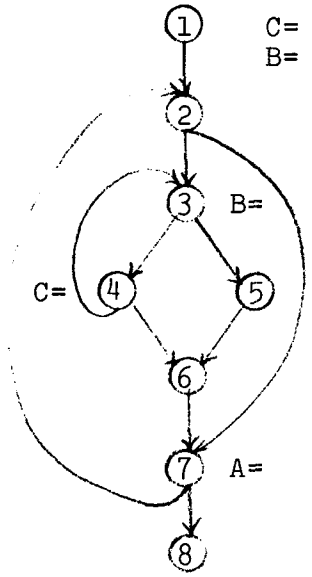


Figure 5

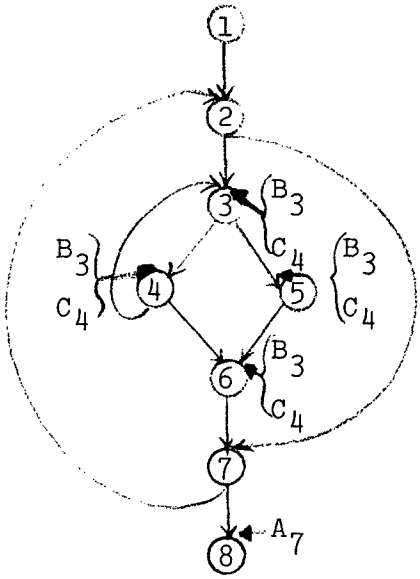


Figure 6

Suppose the analysis is to determine which basic blocks each definition can "reach". Assuming that step 1 of the procedure has codified information about the variables defined in each block, we will now very sketchily show how this information is propagated by Procedure F. The first order intervals are processed. The information associated with blocks 1 and 2 is not changed but by

processing I(3) we get, by step 2.1 followed by step 2.2, the following information.

a. the definition of B and C in blocks 3 and 4 can reach any block in the interval. This information can be left encoded with each block.

b. the definitions can also reach the interval exit and can therefore affect uses outside the interval. This is, therefore, encoded at the exit.

Interval I(7) is processed next and the fact that the definition of A in 7 can affect uses outside the interval is recorded. Figure 7 shows the information left with each node. (Subscripts have been added to the variables to identify which node it was originally in. Step 3 of the procedure now yields the graph in Figure 7.

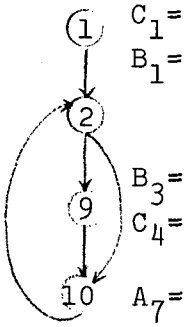


Figure 7

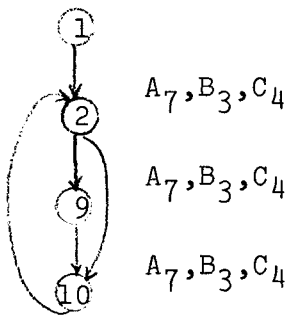


Figure 8

Repeating step 2 for the second order intervals we find that all definitions in the interval can reach every node in the interval. This information is left encoded with each node as depicted in Figure 8. Step 3 now yields Figure 9.

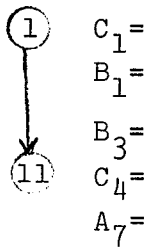


Figure 9

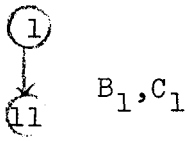


Figure 10

Processing the third order interval $I^3(1) = (1,11)$ yields the fact that B_1 and C_1 can reach node 11. Figure 10 shows this.

We now apply steps 4, 5, 6 and 7 of the procedure. Starting with the second order graph we associate the information left with nodes 1 and 11 with intervals $I^2(1)$ and $I^2(2)$. Thus the fact that B_1 and C_1 reach node 11 means that they reach the interval $I^2(2)$. This is shown in Figure 11.

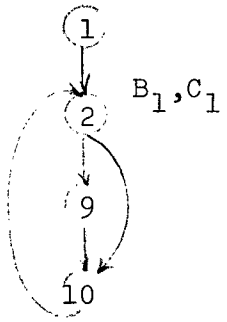


Figure 11

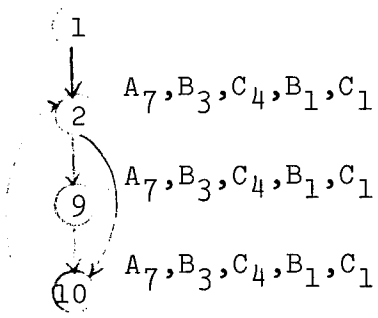


Figure 12

Step 6 propagates, through the graph of Figure 8, the information that B_1 and C_1 can reach the interval head to the other nodes in the interval. (In our attempt to merely sketch the application of Procedure F to a global analysis we have omitted some information which is obviously vital at this point: whether or not a definition reaching a node entry will be able to reach its exit(s). This information can be trivially collected during the analysis. Had it been done we would know that B_1 could not reach the exit of node 9.) The propagated information is associated with each node in the interval as shown in Figure 12. The first order graph is now processed. Step 5 associates the node information of the second order graph with their corresponding interval heads. Figure 13 shows this.

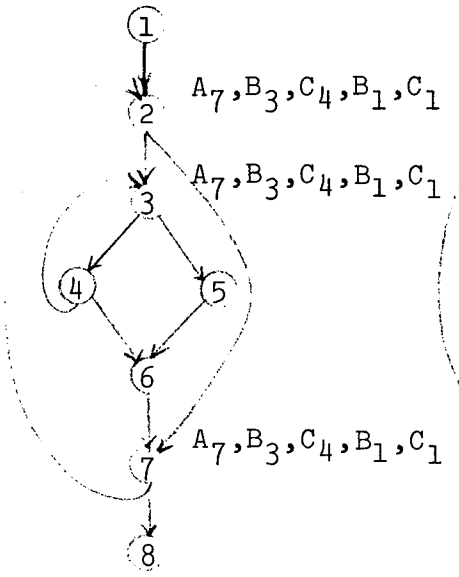


Figure 13

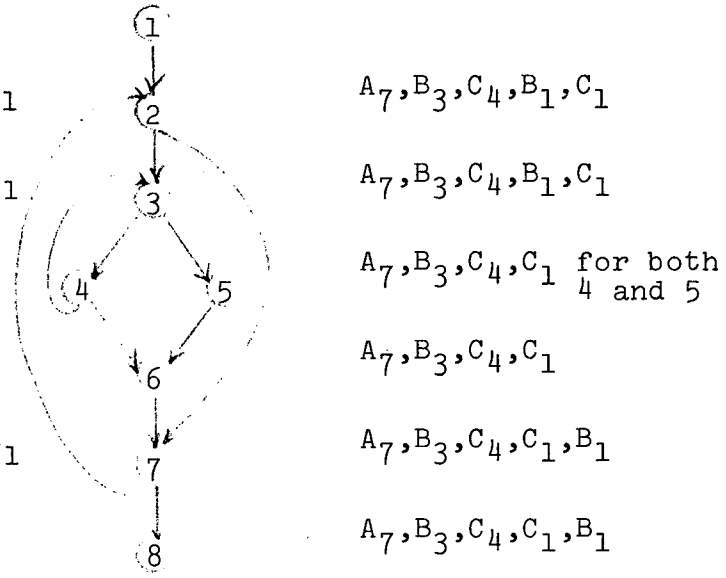


Figure 14

The information is propagated by step 6 through the graph of Fig. 6 and yields the result depicted in Fig. 14. We now know which nodes can be reached by every definition. Although the details of this example are beyond the scope of this paper it is worth commenting that bit vector techniques exist for the example in which a series of simple boolean operations on vectors codifying multiple definitions are used to carry the information through the graph.

SUMMARY

The interval construct described in this paper has many properties which facilitate global analysis and which are of particular interest in optimization. The partial ordering relationships between nodes in an interval provide a natural processing order; the ability to partition a graph into a hierarchy of intervals each of which is partially ordered lets us propagate information rapidly through the graph; the dominance relationships in a graph are easily discovered; nests of strongly connected regions can be detected. Although this paper has not shown how all of these constructs can be found most of them should be apparent. The use of intervals in optimization has only been hinted at; for a good explanation the reader is referred to reference [3].

ACKNOWLEDGEMENTS

As stated in the introduction, the interval concept is due to Dr. John Cocke who is also the major contributor of many of the other ideas in this paper. Dr. J. T. Schwartz first formalized many aspects of intervals. The author whishes to particularly thank both of these people for not only their ideas but also for the continuing encouragement.

REFERENCES

1. Allen, F. E., "Program Optimization," Annual Review in Automatic Programming, Vol. 5, Pergamon, New York, 1969.
2. Berge, C., The Theory of Graphs, Methuen & Co., Ltd., London, 1964.
3. Cocke, John, "Global Common Sub-expression Elimination," in these Proceedings.
4. Cocke, J. and Schwartz, J. T., "Programming Languages and their Compilers," Preliminary Notes, Courant Institute of Mathematical Sciences, New York Univ., N. Y., April 1970.
5. Prosser, R. T., "Applications of Boolean Matrices to the Analysis of Flow Diagrams," Proc. Eastern Joint Computer Conf. Dec. 1959, Spartan Books, N. Y., pp. 133-138.
6. Lowry, Edward S. and Medlock, C. W., "Object Code Optimization," CACM, Jan. 1969, pp. 13-22.
7. Earnest, C. P., Balke, K. G. and Anderson, J., "Analysis of Graphs by Strict Ordering of Nodes" (unpublished).
8. Busam, Vincent A. and Englund, Donald E., "Optimization of Expressions in Fortran," Comm. ACM., Dec. 1969, pp. 666-674.
9. Mendicino, Sam. F. et al., "The LPLTRAN Compiler," CACM, Nov. 1969, pp. 747-755.
10. Cocke, John and Miller, Raymond, "Some Analysis Techniques for Optimizing Computer Programs," Proc. Second Intl. Conf. of Systems Sciences, Hawaii, Jan. 1969.