FLOW GRAPH REDUCIBILITY †

Matthew S. Hecht
and
Jeffrey D. Ullman

Princeton University
Princeton, New Jersey 08540

Abstract

The structure of programs can often be described by a technique called "interval analysis" on their flow graphs. Here, we characterize the set of flow graphs that can be analyzed in this way in terms of two very simple transformation on graphs. We then give a necessary and sufficient condition for analyzability and apply it to "goto-less programs," showing that they all meet the criterion.

1. Introduction

The application of many code improvement techniques depends on globally modeling a program by a directed graph called a "flow graph." This model provides a comprehensive view of the control flow of a program. Examples of improvement possible by flow graph analysis are the detection and removal of useless and redundant statements and the moving of loop independent computation outside loops. Much of the analysis for this type of improvement hinges on the property of a flow graph called "reducibility," e.g. [1-5].

In this paper we give a definition of a flow graph and treat it as a graph theoretic construct. First, the "interval" analysis technique of Cocke and Allen [1,6] is reviewed and reducibility is defined. Next, we present a new technique for treating flow graph reducibility, namely "collapsibility," and show it equivalent to reducibility. Finally, we give a structural characterization of non-reducible flow graphs and use this characterization to obtain an interesting result about flow graphs for "goto-less programs."

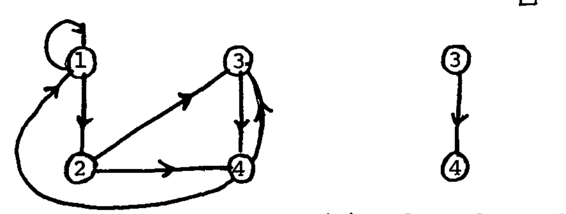
2. Necessary Concepts from Graph Theory

In this section we present the concepts from graph theory which are used in this paper.

Definition 2.1: A directed graph G is a pair (N,E), where N is a set and E is a relation on N. The elements of N are called nodes, and the ordered pairs in E are called edges.

Definition 2.2: Let $G = (N,E)$ be a graph. A graph $G' = (N',E')$ is said to be a subgraph of G if $N' \subseteq N$ and $E' \subseteq E \cap (N' \times N')$.

Example 2.1: Figure 2.1 depicts the graph $G = (\{1,2,3,4\}, \{(1,1), (1,2), (2,3), (2,4), (3,4), (4,1), (4,3)\})$ and the subgraph $S = (\{3,4\}, \{(3,4)\})$. □



(b) Subgraph S of G

(a) Directed graph G

Figure 2.1

Example of directed graph and subgraph

Definition 2.3: Let (n,m) be an edge. This edge is said to leave node n and enter node m.

Definition 2.4: The in-degree of a node is the number of edges entering n and the out-degree of a node n is the number of nodes leaving n.

Definition 2.5: A sequence of nodes $(n_0,n_1,\ldots,n_k)$, $k \geq 0$, is a path of length k from node n_0 to node n_k if there is an edge which leaves node n_{i-1} and enters node n_i for $1 \leq i \leq k$.

Definition 2.6: A cycle (or loop) is a path $(n_0,n_1,\ldots n_k)$ in which $n_0 = n_k$.

Definition 2.7: A graph is connected if, for each pair of distinct nodes (n,m),

†This work was supported by NSF grant GJ-1052.

there is a path from n to m and from m to n.

Definition 2.8: A graph is rooted if there exists at least one node r such that there is a path to all nodes from r. The node r is called a root of the graph.

Definition 2.9: Let (n,m) be an edge. Node n is called a direct ancestor of node m, and node m is called a direct descendant of node n. If there is a path from node n to node m, then n is said to be an ancestor of m,and m is a descendant of n.

It is often useful to attach certain information to either the nodes or edges of a graph. Such information is called a labeling.

Definition 2.10: Let (N,E) be a graph. A node labeling of the graph is a function f from N to a set A of node labels. An edge labeling of the graph is a function g from E to a set B of edge labels. A labeled graph refers to a graph with an associated labeling.

Example 2.2: The graph in Figure 2.1 is a rooted connected graph with node 1 as one of its roots. Node 1 is an ancestor of all other nodes in the graph. Node 2 is a direct descendant of node 1. The path (1,2,4,1) is a cycle. Node 3 has in-degree two and out-degree one. □

Definition 2.11: A tree T is a graph G = (N,E) with a specified node r in N such that:

(a) Node r has in-degree zero.

(b) Node r is a root of T.

(c) All other nodes of T have in-degree one.

Definition 2.12: An ordered tree is a tree with a linear order on the direct descendants of each node.

We follow the convention of drawing trees with the root on top and having all edges directed downward. The direct descendants of a node of an ordered tree are always lineraly ordered from left to right in a diagram.

Example 2.3: An ordered tree is represented in Figure 2.2. Node 4 is the first direct descendant of node 3 since it is the left-most direct descendant of node 3. Node 3 is the second direct descendant of the root. □

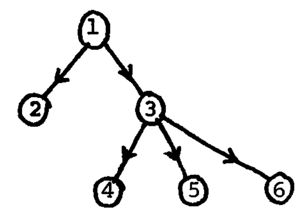


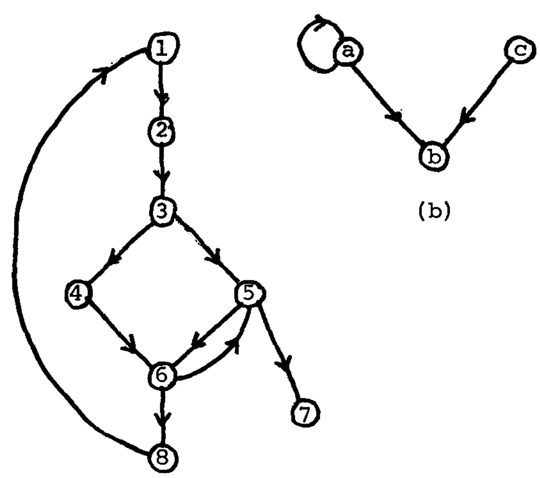
Figure 2.2

Example of a tree

Definition 2.13: A spanning tree of a graph G is a subgraph of G which is a tree and contains all nodes in the graph.

Definition 2.14: A flow graph is a 3-tuple F = (N,E,i), where (N,E) is a finite graph and i is a root of (N,E), called the initial node.

Example 2.4: Figure 2.3(a) shows a flow graph with node 1 as the initial node. Figure 2.3(b) can not be a flow graph, since it has no root. □



(a) (b)

Figure 2.3

Examples of graphs

3. Reducibility

A flow graph may be analyzed by constructs called "intervals."

Definition 3.1: Let G be a flow graph and n a node of G. The interval with header n, denoted I(n), is constructed by the following algorithm.

Algorithm A: [Cocke and Allen] Interval construction.

Input: Flow graph G and designated node n.

Output: I(n)

Method:

A1. Place n in I(n).

A2. If n' is a node not yet in I(n), n' is not the initial node, and all edges entering n' leave nodes in I(n), add n' to I(n).

A3. Repeat step A2 until no more nodes can be added to I(n). □

It should be observed that although n' in step A2 may not be well determined, I(n) does not depend on the order in which candidates for n' are chosen. A candidate at one iteration of A2 will, if it is not chosen, still be a candidate at the next iteration.

The next algorithm partitions a flow graph uniquely into disjoint intervals.

Algorithm B: [Cocke and Allen] Partion of a flow graph into intervals.

Input: A flow graph $G = (N,E,i)$.

Output: A set of disjoint intervals $I_1,\ldots,I_k$, whose union is G.

Method:

B1. Establish a list H of header nodes and a list L of intervals. Initially, H consists only of i; and L is empty.

B2. If H is empty, halt; L is the desired list of intervals.

B3. Otherwise, choose n on H, and compute I(n) by Algorithm A.

B4. Add I(n) to L. Delete n from H, but add to H any node which has a direct ancestor in I(n), but which is not already in H or in one of the intervals on L. Return to B2. □

Example 3.1: Let us consider the flow graph of Fig. 2.3(a). We begin with node 1, the initial node, on list H. Algorithm A tells us to add node 2 to I(1), then to add nodes 3 and 4. No further nodes can be added to I(1). For example, node 5 has an edge entering from 6, which is not currently in I(1), and 6 has an edge entering from 5.

We therefore place $I(1) = \{1,2,3,4\}$ on L, and add 5 and 6 to H. Then, we compute $I(5) = \{5,7\}$ and $I(6) = \{6,8\}$. Note that 1 is not added to I(6), because it is the initial node. □

Two important properties of intervals [1,3,4] are:

(1) every cycle within the interval includes the interval header, and

(2) every edge entering a node of the interval from the outside enters the header.

An interesting aspect of interval analysis is that the intervals of one flow graph can be considered as the nodes of another flow graph in which there is an edge between intervals I_1 and I_2 if and only if $I_1 \neq I_2$, and there is an edge from a node in I_1 to the header of I_2. Furthermore, this process may be repeatedly performed.

Definition 3.2: Let G be a flow graph. Then I(G), the derived graph of G, is defined as follows.

(a) The nodes of I(G) are the intervals of G.

(b) There is an edge from the node representing interval I_1 to that representing I_2 if there is any edge from a node in I_1 to the header of I_2 and $I_1 \neq I_2$.

(c) The initial node of I(G) is the interval containing the initial node of G.

Definition 3.3: Flow graph G is called irreducible if and only if $I(G) = G$.

Definition 3.4: Let G be a flow graph. The sequence $G = G_0, G_1, G_2, \ldots, G_n$ is called the derived sequence for G if $G_{i+1} = I(G_i)$, and G_n is irreducible. G_n is called the limit flow graph of G and is denoted by $\hat{I}(G)$.

Definition 3.5: Flow graph G is called reducible if and only if $\hat{I}(G)$ is a single node with no self-loop. Otherwise, it is called non-reducible.

Example 3.2: Let G_o be the graph of Fig. 2.3(a). Then $G_1 = I(G_o)$ has three nodes, corresponding to the three intervals, {1,2,3,4}, {5,7} and {6,8}, of G_o. Let these nodes be n_1, n_2 and n_3, respectively. Then G_1 is shown in Fig. 3.1. There is an edge from n_1 to n_2, for example, because of the edge in G_o from node 3 to node 5. □

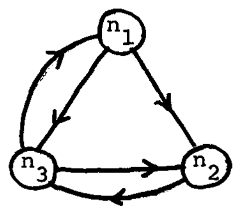


Figure 3.1

4. Collapsibility

We will define a pair of simple transformations that together have the same effect on flow graphs as the interval construction does. Moreover, it will be apparent that the data flow analysis suggested in [1,3,4,6], using interval construction, could be equally well done if construction of the derived sequence of a graph G were replaced by repeated application of our transformations.

There are various advantages to the approach taken here, compared with the interval analysis approach. For example [7] gives an $O(n \log n)$ algorithm to determine whether a flow graph is reducible. In comparison, the straightforward technique of constructing the derived sequence can take $O(n^2)$ steps if performed in the obvious way. Consider, for example, a flow graph of n nodes of Fig. 4.1. Also, [8] gives an algorithm taking $O(n \log n)$ bit vector operations to find common subexpressions in a reducible graph. In comparison, the techniques of [1,4] can require $O(n^2)$ bit vector operations. (Fig. 4.1 again suffices.)

Moreover, these transformations seem to characterize the set of reducible flow graphs in a nice way, and they lead to a further characterization of reducibility that makes it clear in many cases that the control flow structure of a given programming language will yield only reducible flow graphs. For example, the D-charts developed from "goto-less programs" [16] are all reducible. We now give the definitions of the two transformations.

Definition 4.1: Let G be a flow graph. Suppose n is a node in G with a self-loop, that is, an edge from n to itself. Transformation T_1 on node n is removal of this self-loop.

Definition 4.2: Let n_1 and n_2 be nodes in G such that n_2 has the unique direct ancestor n_1, and n_2 is not the initial node. Then transformation T_2 on node pair (n_1,n_2) is merging nodes n_1 and n_2 to one node, named n_1/n_2, and deleting the unique edge between them. Let $n \neq n_1$ and $n \neq n_2$. There is an edge from node n to n_1/n_2 if there was previously an edge from n to n_1 (there cannot be one from n to n_2), and there is an edge from n_1/n_2 to n if there was previously one to n from either n_1 or n_2 or both. n_1/n_2 has a self-loop if there was an edge from n_2 to n_1.

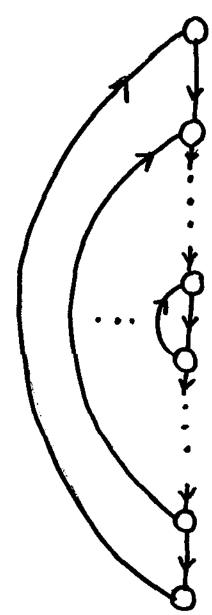


Figure 4.1

Flow Graph Requiring $O(n^2)$ Steps for Interval Analysis

Example 4.1: Figure 4.2 shows a flow graph which is transformed into a single node by one application of T_1 and two of T_2. Although T_2 is not applicable to the original graph, it becomes applicable after use of T_1. □

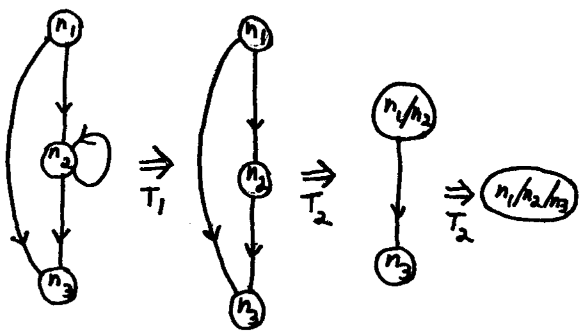


Figure 4.2

Applications of T_1 and T_2

Various authors have considered similar transformations, but from the point of view of generating graphs rather than analyzing (i.e., reducing) them. Cooper [9] considers three generating rules, one of which is the inverse of T_1 (i.e., addition of self-loops). The other two together are equivalent to the inverse of T_2. It is shown in [9] that together with a construction which is the inverse of "node splitting" [10], these generating rules are capable of building an arbitrary flow graph.

Engeler [11,12] considers "normal form flow charts," which are built by two generating rules, one the inverse of T_1 and the other equivalent to the inverse of T_2, restricted so that the two nodes involved have disjoint sets of direct descendants. Thus, the normal form flow charts are a subset of the reducible graphs. They are characterized as trees with back edges.

We now proceed to develop useful properties of the transformations T_1 and T_2.

Definition 4.3: A flow graph is called collapsible if and only if it can be transformed into a single node with no self-loop by repeated application of T_1 and T_2. Otherwise, it is called non-collapsible.

Example 4.2: The flow graph of Figure 4.3 is non-collapsible. There are no self-loops, and no node has a unique entering edge, so neither T_1 nor T_2 is applicable. On the other hand, the flow graph of Figure 4.2 is collapsible. □

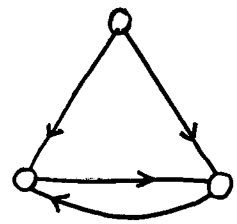


Figure 4.3

Example of a non-collapsible flow graph

T_1 and T_2 have a useful property; they form a "finite Church-Rosser" transformation [13].

Definition 4.4: Let R be a relation on a set S. Let xRy denote $(x,y)\in R$. The inverse of R, R^{-1}, is $\{(y,x) \mid (x,y)\in R\}$. R is symmetric if $R=R^{-1}$. R is reflexive if $(x,x)\in R$ for all $x\in S$. R is transitive if xRy and yRz imply xRz for all x,y,z in S.

Definition 4.5: If R_1 and R_2 are relations on S, then the composition of R_1 and R_2, denoted R_1R_2, is $\{(x,z) \mid$ for some y in S, xR_1y and $yR_2z\}$. The reflexive closure of R, denoted $R^{\#}$, is $R\cup\{(x,x) \mid x\in S\}$. The transitive closure of R, denoted R^{+}, is $R^1\cup R^2\cup R^3\cup\ldots$, where $R^1=R$ and $R^i=RR^{i-1}$ for $i\geq 2$. The reflexive transitive closure of R, denoted R^*, is $R^{\#}\cup R^{+}$. The completion of R, denoted $\hat{R}$, is $\{(x,y) \mid xR^*y$ and there is no z such that $yRz\}$.

Definition 4.6: A pair $(S,\Rightarrow)$, where S is a set and $\Rightarrow$ is a relation on S is said to be finite if for each p in S, there is a constant k_p such that if $p\overset{i}{\Rightarrow}q$,† then $i\leq k_p$. That is, there is a bound on the number of times $\Rightarrow$ can be applied in succession, beginning with any element p. We say $(S,\Rightarrow)$ is finite Church-Rosser (FCR) if it is finite, and $\overset{\wedge}{\Rightarrow}$ is a

† We place the symbols $\hat{}$,#,*,+ and i above the relation symbol $\Rightarrow$ instead of at the upper right corner, as indicated for relation R in Definition 4.5.

function, i.e., $p \overset{\wedge}{\Rightarrow} q$ and $p \overset{\wedge}{\Rightarrow} r$ implies $q = r$. If set S is understood, $\Rightarrow$ is called an <u>FCR transformation</u>.

The following theorem gives a test for the FCR property which is simpler to apply than Definition 4.6. It is proved in [13].

<u>Theorem 4.1</u>: Let $\Rightarrow$ be a relation on set S. Then $(S,\Rightarrow)$ if FCR if and only if it is finite, and for all p in S, if $p \Rightarrow p_1$ and $p \Rightarrow p_2$, then there is some q such that $p_1 \overset{*}{\Rightarrow} q$ and $p_2 \overset{*}{\Rightarrow} q$.

<u>Definition 4.7</u>: Let S be the set of flow graphs. We define the relation $\underset{i}{\Rightarrow}$, $i = 1$ or 2, by $g \underset{i}{\Rightarrow} g'$ if and only if g can be transformed into g' by an application of T_i. Let $\Rightarrow$ denote the union of $\underset{1}{\Rightarrow}$ and $\underset{2}{\Rightarrow}$. The reflexive closure, k-fold product, transitive closure, reflexive transitive closure, and the completion of $\Rightarrow$ are respectively given by $\overset{\#}{\Rightarrow}$, $\overset{k}{\Rightarrow}$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$, and $\overset{\wedge}{\Rightarrow}$.

<u>Theorem 4.2</u>: $(S,\Rightarrow)$ is FCR.

<u>Proof</u>: We use Theorem 4.1 and note that in this case, we will always be able to find q such that $p_1 \overset{\#}{\Rightarrow} q$ and $p_2 \overset{\#}{\Rightarrow} q$.

(Finiteness property). Let g be a flow graph with n nodes. Each application of T_1 or T_2 deletes at least one edge. Thus, $\Rightarrow$ is finite.

("Commutativity" property). Suppose $g \underset{i}{\Rightarrow} g_1$ and $g \underset{j}{\Rightarrow} g_2$, where $g \in S$ and $i,j \in \{1,2\}$. There are three distinct cases to consider.

<u>Case 1</u>: $(i = j = 1)$. Suppose T_1 is applied to node n_1 to yield g_1 and to node n_2 to yield g_2. If $n_1 = n_2$, then $g_1 = g_2$. If $n_1 \neq n_2$, then T_1 may be performed on n_2 in g_1 and on n_1 in g_2 to yield equal graphs. Thus, $g \Rightarrow g_1 \overset{\#}{\Rightarrow} h$ and $g \Rightarrow g_2 \overset{\#}{\Rightarrow} h$, where h is the graph resulting after applying T_1 to nodes n_1 and n_2 in g.

<u>Case 2</u>: $(i = j = 2)$. Suppose T_2 is applied to node pair (n_1,n_2) in g to yield g_1, and to node pair (n_3,n_4) in g to yield g_2. If $n_1 = n_3$ and $n_2 = n_4$, then $g_1 = g_2$. If all four nodes are distinct, then apply T_2 to (n_3,n_4) in g_1, and apply T_2 to (n_1,n_2) in g_2 to yield equal graphs. Now suppose neither of the previous subcases holds. If $n_1 = n_3$ and no other equalities hold, then Figure 4.4 shows the subgraph of interest.

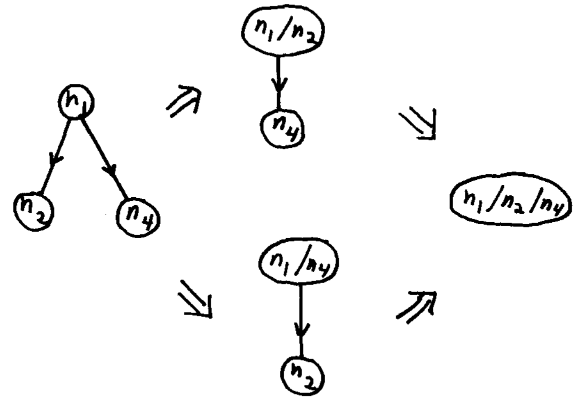


Figure 4.4

Applications of T_2

Otherwise, if $n_2 = n_3$ and no other equalities hold, then Figure 4.5 shows the subgraph of interest.

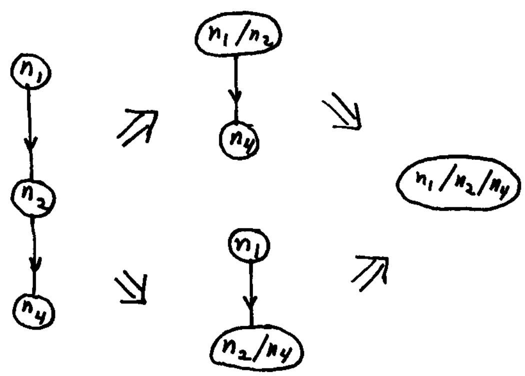


Figure 4.5

Applications of T_2

Thus, $g \Rightarrow g_1 \overset{\#}{\Rightarrow} h$ and $g \Rightarrow g_2 \overset{\#}{\Rightarrow} h$, where h is the graph resulting after applying T_2 to (n_1,n_2) and to (n_3,n_4) in g.

The case in which $n_1 = n_4$ and no other equalities hold is symmetric to the case $n_2 = n_3$ above. The case $n_1 = n_4$ and $n_2 = n_3$ is impossible, because then the flow graph has two isolated nodes, and hence must consist of only n_1 and n_2. But one of these must be the initial node, and T_2 is thus either not applicable to (n_1,n_2) or not applicable to (n_3,n_4). Since we have assumed $n_1 \neq n_2$ and $n_3 \neq n_4$, and n_2 may not be n_4 unless $n_1 = n_3$, we have considered all possibilities.

Case 3: $(i \neq j)$. Suppose T_2 is applied to node pair (n_1,n_2) in g to yield g_1, and T_1 is applied to node n_3 in g to yield g_2. Clearly, $n_2 \neq n_3$. Consequently, T_1 and T_2 do not "interfere;" T_1 may be applied to node n_3 in g_1, and T_2 may be applied to node pair (n_1,n_2) in g_2 to yield equal graphs. Thus, $g \Rightarrow g_1 \Rightarrow h$ and $g \Rightarrow g_2 \Rightarrow h$, where h is the result of applying T_2 to (n_1,n_2) and T_1 to n_3. □

5. Equivalence of Reducibility and Collapsibility

Theorems 5.1 and 5.2 establish that a flow graph is reducible if and only if it is collapsible.

Definition 5.1: Let the first n nodes added to an interval I(h) in Algorithm A be called a *partial interval*. We assume, of course, that the interval I(h) has at least n nodes, and $n \geq 1$.

Lemma 5.1: Let G be a flow graph. Then $G \overset{*}{\Rightarrow} I(G)$.

Proof: It suffices to show that a partial interval is collapsible to its header, and that connections (edges) between a partial interval and the other nodes in the flow graph are maintained. Thus, constructing the derived graph I(G) of flow graph G corresponds exactly to collapsing the intervals of G.

Inductive Hypothesis: A partial interval of n nodes is collapsible to its header, and edges between the partial interval and the other nodes of the flow graph are preserved. That is, edges leaving the partial interval to another node outside the partial interval remain. The header will have no self-loops.

Basis: The first node added to an interval is the header node. The only collapsing possible is removal of a self-loop if present. This possible application of T_1 will not destroy any edge to another node in the graph outside the partial interval.

Inductive Step: Assume that the inductive hypothesis is true for a partial interval of n nodes, and consider the addition of another node m to the partial interval. This new node only has edges entering it from nodes in the partial interval. Since the first n nodes of the partial interval are collapsible by the induction hypothesis, there will be exactly one edge from the collapsed partial interval to m. Thus, T_2 is applicable. Edges from m to nodes outside the partial interval now leave the node for the collapsed partial interval. If there is a self-loop introduced by the application of T_2, it can be removed by T_1. □

As an immediate consequence of Lemma 5.1, we have the following.

Theorem 5.1: If a flow graph is reducible, then it is collapsible.

Proof: If $\hat{I}(G) = 0^{\dagger}$, then $G \overset{*}{\Rightarrow} 0$, is by Lemma 5.1, iterated.

The converse of Theorem 5.1 is easy to prove.

Theorem 5.2: If a flow graph is collapsible, then it is reducible.

Proof: Suppose $G \hat{\Rightarrow} 0$, and let $\hat{I}(G) = G'$. By Lemma 5.1 iterated, $G \overset{*}{\Rightarrow} G'$. We must have $G' \hat{\Rightarrow} 0$. (For if $G' \hat{\Rightarrow} G''$, then $G \hat{\Rightarrow} G''$. Since $\hat{\Rightarrow}$ is a function, and $G \hat{\Rightarrow} 0$, we have $G'' = 0$.)

If $G' \neq 0$, then since $G' \hat{\Rightarrow} 0$, T_1 or T_2 is applicable to G'. We have assumed $I(G') = G'$, so every node appears on the header list when Algorithm B is applied to G'. If T_1 is applicable to node n, then I(n) does not have a self-loop in I(G'), so $I(G') \neq G'$. If T_2 is applicable to node pair (n_1,n_2), then n_2 is in $I(n_1)$, so again, $I(G') \neq I(G)$. We conclude that $G' = 0$. □

6. Characterization Theorem for Non-Reducible Flow Graphs

We will now show the existence of a certain subgraph in all and only the non-reducible flow graphs. Prior to showing this result, we present the concept of a "depth-first spanning tree" of a flow graph.

Definition 6.1: A *depth-first spanning tree* (DFST) of a flow graph G is a spanning tree that is constructed by Algorithm C.

† Let 0 represent the graph with one node and no edges.

Algorithm C: DFST of a flow graph.
Input: Flow graph G.
Output: DFST of G.
Method:

C1. The root of the DFST is the initial node of G. Let this node be the node n "under consideration."

C2. Perform step C3 until it is no longer applicable. Then perform C4 and C5.

C3. If the node n under consideration has a direct descendant x not already on the DFST, we select node x as the rightmost direct descendant of n so far found. If this step is successful, node x becomes the node n under consideration.

C4. If the node under consideration is the root, then halt.

C5. Otherwise, back up the DFST one node toward the root and consider this node by going to step C2. □

Definition 6.2: We define the spine of a DFST T to be the sequence of nodes $(n_1,n_2,\ldots,n_k)$ such that n_1 is the root of T, n_{i+1} is the rightmost direct descendant of n_i, $1 \le i \le k-1$, and n_k has no direct descendants.

We can add to the DFST T of a flow graph G the edges of G which are not edges of T. Conventionally, we will show edges of T as solid lines and edges of G not in T by dashed lines. An important property of DFST's is the following.

Lemma 6.1: [14] Let $G = (N,E,i)$ be a flow graph and $T = (N,E')$ one of its DFST's. If there is an edge (n_1,n_2) in $E - E'$, then either:

(1) n_1 is a descendant of n_2 in T,

(2) n_1 is an ancestor of n_2 in T,

(3) $n_1 = n_2$, or

(4) n_1 is to the right of n_2 in T.† □

† The notion of "to the right" has only been defined for nodes with the same direct ancestor. We can extend it naturally by saying that if n is to the right of m, then all n's descendants are to the right of all of m's descendants.

Example 6.1: Let G be the flow graph of Figure 6.1(a). If we consider nodes in the order 1,2,3,4, then back to 3, then to 5, we obtain the DFST of Figure 6.1(b). The spine is 1,2,3,5. □

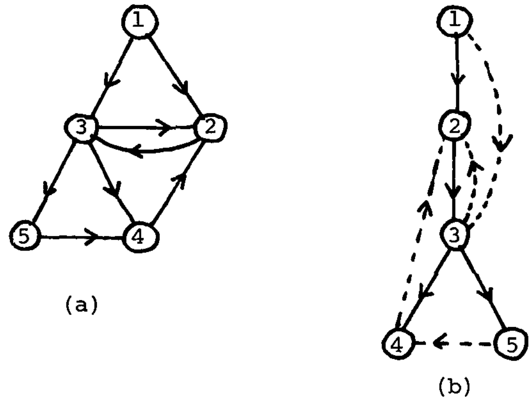


Figure 6.1

Example of Algorithm C

Definition 6.3: Let (*) denote any of the graphs represented in Figure 6.2 where the wiggly lines denote node disjoint (except for the endpoints, of course) paths; a,b,c and i are distinct, except that a and i may be the same.

(*):

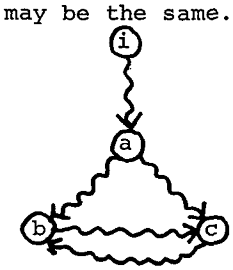


Figure 6.2

Lemma 6.2: The absence of subgraph (*) in a flow graph is preserved by T_1 and T_2.

Proof: Let G be a flow graph and let n_1 and n_2 be any two nodes in G. We observe that if a path does not exist between n_1 and n_2, then neither T_1 nor T_2 will create such a path; neither will they make two paths be node disjoint if they were not so already. □

Theorem 6.1: If a flow graph is nonreducible, then it has a subgraph of form (*).

Proof: We prove the theorem by induction on n, the number of nodes of G.

Inductive Hypothesis: Flow graph G with n nodes has a subgraph of form (*).

Basis: $(n = 3)$. This is an elementary consideration of the three cases in Figure 6.3, with the initial nodes at the top.

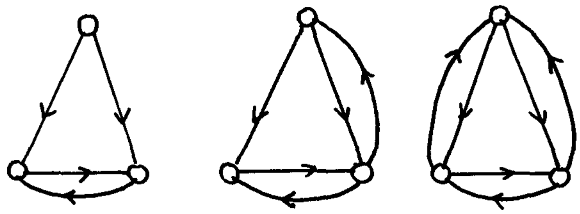


Figure 6.3

Inductive Step: $(n > 3)$. Assume that the inductive hypothesis is true, and consider a non-reducible flow graph G with n nodes. By Lemma 6.2, we may assume without loss of generality that T_1 is not applicable to G. That is, if G can become G' under repeated application of T_1, and we can show that G' has (*), then we will also have shown that G has (*). By the inductive hypothesis and Lemma 6.2, it follows that T_2 is not applicable to G. Thus, we may assume that G is irreducible. Let T be a DFST for G, and let the spine of T be $(n_1, n_2, \ldots, n_k)$.

We claim that $k \geq 3$. The initial node n_1 is on the spine. Now consider the rightmost direct descendant of the root, namely n_2. Surely n_2 exists, since $n > 1$. Node n_2 must have at least two entering edges in G, since G is irreducible (else T_2 would be applicable). By Lemma 6.1, other entering edges must come from descendants of n_2. Thus, n_2 must have at least one direct descendant, n_3.

Now find the highest number d, such that n_d has an edge (in G but not T) to some $n_i \neq n_1$ on the spine, with $i < d$. n_d always exists because, in particular, n_2 has such an edge entering. Let b be the largest number in the range $1 < b < d$, such that there is an edge from n_d to n_b in G.

Find (if possible) the first node n_a on the spine starting from the root with a forward edge (in G but not in T) entering a node n_c, such that n_c is below n_b on the spine and equal to or below n_d. Figure 6.4 depicts this situation. Notice that nodes n_a, n_b, and n_c correspond to nodes a,b, and c in (*), and n_1 corresponds to i.

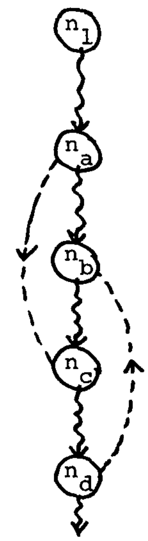


Figure 6.4

Suppose that there is no such edge (n_a, n_c) in G. Let us consider the subgraph H of G consisting of the nodes on the spine from n_b to n_d, together with their connecting edges in G. There are no edges of G entering a node in H from above other than to n_b by assumption, and there are no edges of G entering a node in H from below n_d on the spine since (n_d, n_b) is the "lowest" backward edge. Furthermore, by Lemma 6.1 no other edges enter nodes in H. Thus, any reduction by T_1 or T_2 taking place in H, with n_b treated as the initial node, will also be a valid reduction in G. Since G is irreducible, we conclude that H is likewise irreducible. Finally, since $b > 1$, the induction hypothesis applies to H. This ends the induction.

But, since H has a subgraph of form (*) with initial node n_b, it is easy to show that G has a subgraph (*) with initial node n_1 by adding the path from n_1 to n_b. □

Corollary: If G is irreducible, then it has a subgraph (*) in which the path from a to c is a single edge. □

Theorem 6.1 is stronger than a previously known result [4,15], which states that every non-reducible graph has a double entry loop. For example, Figure 6.5

shows a graph with a double entry loop which not only is reducible, but which is a "D-chart." In the next section we use Theorem 6.1 to prove that all D-charts are reducible.

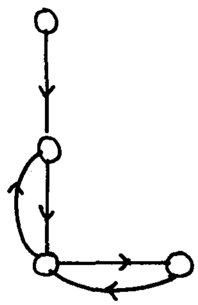


Figure 6.5

Theorem 6.2: If a flow graph G has a subgraph (*), then G is non-reducible.

Proof: We prove the result by the number of nodes, n, in G. The basis is again trivial. For the induction, suppose that G of $n > 3$ nodes is reducible, but has a subgraph (*). Let G' be the graph formed by applying T_1 to G until no longer possible. It is easy to see that G' also contains (*), and by Theorem 4.2 is reducible. Therefore T_2 is applicable to some node pair (n_1, n_2) of G'. Let n_2 be the direct descendant of n_1, and let G" be the result of applying T_2 to G'. We consider cases, depending on the relation of n_2 to (*).

Case 1: n_2 is not one of the nodes represented by (*), including the paths shown. It is straightforward in this case to show that (*) is present in G".

Case 2: n_2 is a of (*). Then n_1 must be the predecessor of a on the path from i to a. Again, (*) exists in G".

Case 3: n_2 is b or c. Since b and c each have at least two distinct predecessors, this case is impossible.

Case 4: n_2 is a node on one of the paths of (*). Then n_1 is on the same path (possibly an endpoint). (*) clearly exists in G".

Since G" has one fewer node than G, the inductive hypothesis applies to G". Therefore G" is non-reducible. But by Theorem 4.2, since G $\overset{*}{\Rightarrow}$ G", and G $\hat{\Rightarrow}$ 0, it follows that G" $\hat{\Rightarrow}$ 0, i.e., G" is reducible. We have a contradiction, and conclude that G is non-reducible. □

7. Applications of the Characterization Theorem

D-charts [16-19] or "block form programs" [20] are a restricted class of flow charts which can be implemented by a programming language having no explicit "go-to" statements. They are as powerful as general flow charts provided additional variables called "flags" are introduced to represent a history of control flow [17].

We define D-charts by informal "graph grammars." (See [21], e.g.) The graph grammars we use are similar to the grammars for formal languages, except that the production rules indicate the replacement of nodes in a labeled graph by subgraphs. For example, Figure 7.1 presents a simple definition of D-charts. The start symbol is <block>. Rule (3) in Figure 7.1 shows that a <block> may be replaced by an "iteration" structure, (while-do), and rule (2) enables possible replacement of a <block> by an "if-then-else" structure.

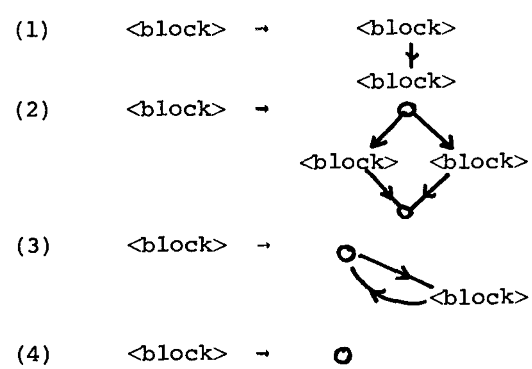


Figure 7.1

Definition 7.1: A D-chart is a flow graph which can be produced by the following rules.

(1) Begin with a single node, the initial node, labeled <block>.

(2) Replace, at will, a node n, labeled <block>, by one of the structures on the right of the → in Figure 7.1. Edges entering n now enter the highest node in each of the replacement structures. Edges leaving n now leave the lowest node in structures 1, 2 and 4 and the higher node in structure 3.

(3) If the node replaced is the initial node, the highest replacing node becomes initial.

(4) Terminate the generation process if there are no nodes labeled <block>. Otherwise return to step (2).

Example 7.1: The sequence of graphs shown in Figure 7.2 illustrate the generation of a D-chart. Figures 7.2(b), (c) and (d) are produced by rules (2), (1) and (3), respectively. Figure 7.2(e), the D-chart is produced by three applications of rule (4).

□

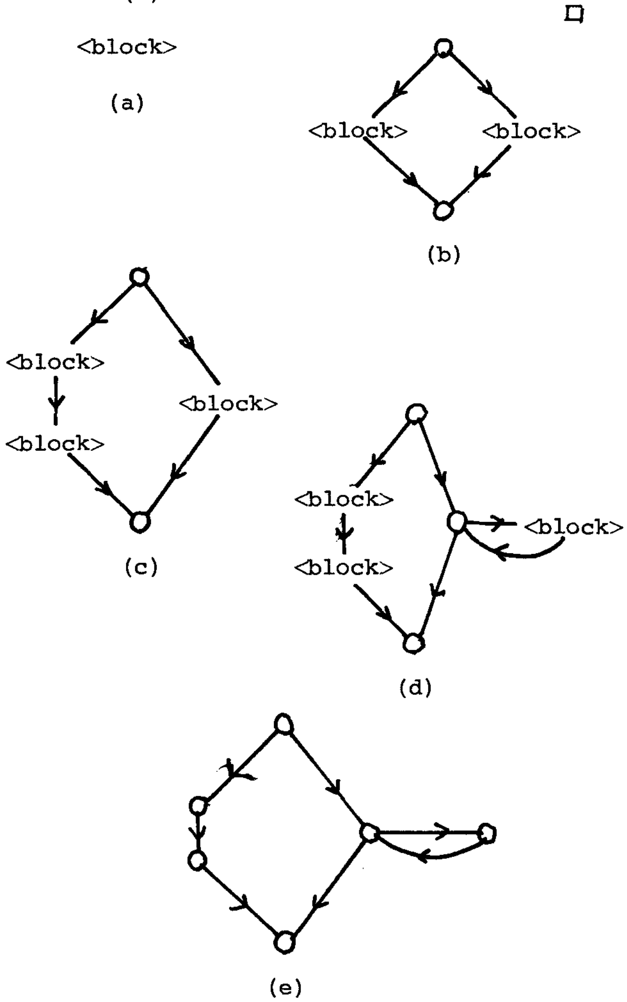


Figure 7.2

Generation of a D-chart

Theorem 7.1: Every D-chart is reducible.

Proof: We will use Theorem 6.1 and show that (*) cannot appear in a D-chart. If (*) does appear, then node a, which has at least two direct descendants must be created as the highest node in one of the replacement structures of rules (2) and (3) in Figure 7.1. These possibilities are shown in Figure 7.3 (a) and (b) respectively.

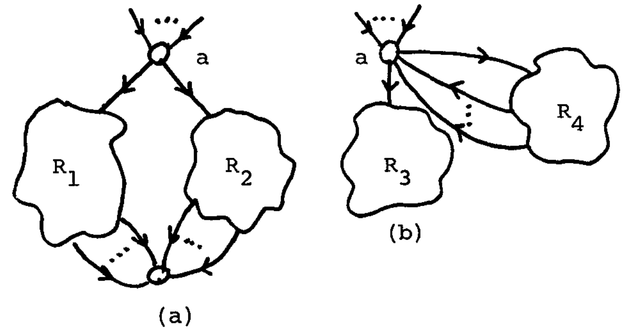


Figure 7.3

Portions of a D-chart

In Figure 7.3(a), regions R_1 and R_2 are the sets of nodes generated by the two nodes labeled <block> in Figure 7.1 (2). Since paths in (*) are node disjoint, nodes b and c must be found in R_1 and R_2, respectively. But it is elementary that there can be no paths from R_1 to R_2 that do not pass through a. Thus, no (*) exists in this case.

In Figure 7.3(b), region R_4 represents the nodes generated by the node <block> in Figure 7.1 (3), and R_3 represents the nodes accessible from a without entering R_4. We note that any node labeled <block> in the generation scheme of Definition 7.1 has out-degree at most one. Thus, b and c of (*) must appear in R_3 and R_4, respectively. Again, we observe that a path from b to c must pass through a, and we conclude the theorem.

□

Another simple example of the application of Theorem 6.1 is the following.

Theorem 7.2: The flow graphs of those FORTRAN programs whose transfers to previous statements are all caused by DO loops are reducible.

Proof: If the flow graph for such a program had subgraph (*), then the loop between nodes b and c would be part of a DO-loop, and the paths from a to b and c cannot be part of that DO loop. Since DO loops may be entered at only one point, we would conclude that b and c are the same node. Thus, (*) does not appear in such a flow graph. □

8. Conclusions

We have demonstrated that interval analysis is a special case of a more general reduction technique. This technique, the application of two transformations:

T_1 = removal of self-loops

T_2 = collapsing of a node with a single direct ancestor into that ancestor,

can be used for data flow analysis exactly as interval analysis is used.

We then showed that all and only the non-reducible flow graphs have a subgraph (*) consisting of at least three nodes, a, b and c, with node disjoint paths from the initial node to a, from a to b and c and from b to c and back. (a may be the initial node.) We used this result to prove that certain kinds of programs have reducible flow graphs.

References

[1] F.E. Allen, "Control Flow Analysis," SIGPLAN Notices, Vol. 5, pp 1-19, July 1970.

[2] F.E. Allen, "Program Optimization," Annual Review in Automatic Programming, Vol.5, Pergamon Press, New York, 1969.

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

[4] M. Schaefer, A Mathematical Theory of Global Program Analysis, unpublished notes, System Devel. Corp., Santa Monica, Calif. 1971.

[5] K. Kennedy, "A Global Flow Analysis Algorithm," International Journal of Computer Mathematics, Vol. 3, pp 5-15, December 1971.

[6] J. Cocke, "Global Common Subexpression Elimination," SIGPLAN Notices, Vol. 5, pp 20-24, July 1970.

[7] J.E. Hopcroft and J.D. Ullman, "An n log n Algorithm for Detecting Reducible graphs," to appear in Proc. 6th Annl. Princeton Conference on Information Sciences and Systems, 1972.

[8] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions," manuscript in preparation.

[9] D.E. Cooper, "Programs for Mechanical Program Verification," Machine Intelligence 6, pp 43-62, American Elsivier, New York, 1971.

[10] J. Cocke and R.E. Miller, "Some Analysis Techniques for Optimizing Computer Programs," Proc. 2nd Intl. Conf. of System Sciences, Hawaii, 1969.

[11] E. Engeler, "Structure and Meaning of Elementary Programs," Symposium on Semantics of Algorithmic Languages (Engeler Ed.) Springer-Verlag, New York, 1971.

[12] E. Engeler, "Algorithmic Approximations," JCSS, Vol. 5, pp 61-82, 1971.

[13] A.V. Aho, R. Sethi, and J.D. Ullman, "Code Optimization and Finite Church-Rosser Systems," Computer Science Laboratory, Princeton University, TR 92, April 1971.

[14] R. Tarjan, "Depth First Search and Linear Graph Algorithms," Proc. IEEE 12th Annl. Symposium on Switching and Automata Theory, Oct. 1971.

[15] F.E. Allen, private communication.

[16] E.W. Dijkstra, "Goto Statement Considered Harmful," CACM, Vol. 11, no. 3, pp 147-148, March 1968.

[17] J. Bruno and K. Steiglitz, "The Expression of Algorithms by Charts," Computer Science Laboratory, Princeton University, TR88, July 1971.

[18] D.E. Knuth and R.W. Floyd, "Notes on Avoiding 'GOTO' Statements," TR-CS 148, Computer Science Department, Stanford University, January 1970.

[19] J.R. Rice, "The GOTO Statement Reconsidered," CACM, Vol. 11, no. 8, p. 538, and reply by E.W. Dijkstra, p. 538 and p. 541, August 1968.

[20] D.C. Cooper, "Some Transformations and Standard Forms of Graphs, with Applications to Computer Programs," Machine Intelligence 2, pp 21-32, American Elsivier, New York, 1968.

[21] T. Pavlidis, "Linear and Context Free Graph Grammars," unpublished memorandum, Princeton Univ., Dept. of Elec. Eng., 1970.