ANALYSIS OF A SIMPLE ALGORITHM FOR GLOBAL DATA FLOW PROBLEMS†

by

Matthew S. Hecht
Jeffrey D. Ullman

Princeton University
Princeton, New Jersey 08540

Abstract

There is an ordering of the nodes of a flow graph G which topologically sorts the dominance relation and can be found in O(edges) steps. This ordering is the reverse of the order in which a node is last visited while growing any depth-first spanning tree of G. Moreover, if G is reducible, then this ordering topologically sorts the "dag" of G. Thus, for a reducible flow graph (rfg) there is a simple algorithm to compute the dominators of each node in O(edges) bit vector steps.

The main result of this paper relates two parameters of an rfg. If G is reducible, d is the largest number of back edges found in any cycle-free path in G, and k is the length of the interval derived sequence of G, then $k \geq d$. From this result it follows that there is a very simple bit propagation algorithm (indeed, the obvious one) which also uses the above ordering, and is at least as good as the interval algorithm for solving all known global data flow problems such as "available expressions" and "live variables."

Key Words and Phrases: Code optimization, flow graph, reducibility, interval analysis, dominance, depth-first spanning tree, global data flow analysis, available expressions, live variables.

I. Introduction

When analyzing computer programs for code improvement [A1], there is a class of problems, each of which can be solved in essentially the same manner. These problems, called "global data flow analysis problems," involve the local collection of information which is distributed throughout the program. Some examples of global flow analysis problems are "available expressions" of [C] and [U1], "live variables" [Ke], "reaching definitions" of [A2] and [A3], and "very busy variables" [S]. There are several general algorithms to solve such problems.

The "interval" approach ([A2],[A3], [AC],[C],[Ke],[S] and [AU]) collects information by partitioning the flow graph of the program into subgraphs called intervals, replacing each interval by a single node containing the local information within that interval, and continuing to define such interval partitions until the graph becomes a single node itself, at which time global information is propagated locally by reversing the reduction process.

Another approach ([V]†,[U1] and [Ki] propagates information in an obvious manner until all the required information is collected; that is, until the process converges. We shall show that this second approach (with a suitable node ordering) is no worse than the interval approach!

Prior to presenting the main result and the algorithm, we review part of the theory of reducible flow graphs.

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

† In 1961 V.A. Vyssotsky [V] implemented this kind of flow analysis (and presumably the obvious algorithm) in a Bell Laboratories 7090 FORTRAN II compiler--for strictly diagnostic purposes.

II. Background

A flow graph is a triple $G = (N,E,n_0)$, where:

(a) N is a finite set of nodes.

(b) E is a subset of NxN called the edges. The edge (x,y) enters node y and leaves node x. We say that x is a predecessor of y, and y is a successor of x.

A path from x_1 to x_k is a sequence of nodes $(x_1,\ldots,x_k)$ such that (x_i,x_{i+1}) is in E for $1 \leq i < k$. The path length of $(x_1,\ldots,x_k)$ is k-1. If $x_1=x_k$, the path is a cycle.

(c) Node n_0 in N is the initial node. There is a path from n_0 to every node.

INTERVALS

Let G be a flow graph and h a node of G. The interval with header h, denoted by I(h), is defined as follows:

(a) Place h in I(h).

(b) If m is a node not yet in I(h), m is not the initial node, and all edges entering m leave nodes in I(h), add m to I(h).

(c) Repeat step (b) until no more nodes can be added to I(h).

It should be observed that although m in (b) above may not be well-defined, I(h) does not depend on the order in which candidates for m are chosen. A candidate at one iteration of (b) will, if it is not chosen, still be a candidate at the next iteration.

It is well known that a flow graph can be uniquely partitioned into disjoint intervals, and that this process takes time proportional to the number of edges in the flow graph [A2].

If G is a flow graph, then the derived flow graph of G, denoted by I(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 J to that representing K if there is any edge from a node in J to the header of K, and $J \neq K$.

(c) The initial node of I(G) is $I(n_0)$.

The sequence $G = G_0,G_1,\ldots,G_k$ is called the derived sequence for G if $G_{i+1} = I(G_i)$, $G_{k-1} \neq G_k$, and $I(G_k) = G_k$. G_k is called the limit flow graph of G.

A flow graph G is called reducible (an rfg) if and only if its limit flow graph is a single node with no edge (henceforth called the trivial flow graph). Otherwise, it is called nonreducible.

T1 AND T2

Let $G = (N,E,n_0)$ be a flow graph and let (w,w) be an edge of G. Transformation T1 is removal of this edge.

Let y not be the initial node and have a single predecessor, x. Transformation T2 is the replacement of x, y, and (x,y) by a single node z. Predecessors of x become predecessors of z. Successors of x or y become successors of z. There is an edge (z,z) if and only if there was formerly an edge (y,x) or (x,x). (Whenever T2 is applied as described here, we say that x consumes y.)

There are two results from [HeUl] which interest us. First, if T1 and T2 are applied to a flow graph until no longer possible, then a unique flow graph results, independent of the sequence of applications of T1 and T2 actually chosen. Second, a flow graph is reducible by intervals if and only if repeated application of T1 and T2 yields the trivial flow graph.

DOMINANCE AND REGIONS

If x and y are two distinct nodes in a flow graph G, then x dominates y if every path in G from its initial node to y contains x ([P] and [LM]).

Let $G = (N,E,n_0)$ be a flow graph, let $N_1 \subseteq N$, let $E_1 \subseteq E$, and let m be in N_1. We say $R = (N_1,E_1,m)$ is a region of G with header m if in every path $x_1,\ldots,x_k$, where $X_1 = n_0$ and x_k is in N_1, there is some $i \leq k$ such that

(a) $x_i = m$; and

(b) $x_{i+1},\ldots,x_k$ are in N_1; and

(c) $(x_i,x_{i+1}),(x_{i+1},x_{i+2}),\ldots,(x_{k-1},x_k)$ are in E_1

That is, access to every node in the region is through the header only.

As we proceed to apply T1 and T2 to a flow graph, each edge of an intermediate graph represents a set of edges and each node represents a set of nodes and edges in a natural way.

We say that each node and edge in the original flow graph represents itself. If T1 is applied to node w with edge (w,w), then the resulting node represents what node w and edge (w,w) represented. If T2 is applied to x and y, with edge (x,y) eliminated, then the resulting node z represents what x, y, and (x,y) represented. In addition, if two edges (x,u) and (y,u) are replaced by a single edge (z,u), then (z,u) represents what (x,u) and (y,u) represented.

The two lemmas which follow appear in [U2].

Lemma 1: (a) Let z be a node constructed during the reduction of some flow graph G. If z represents edge (x,y) of G, then x and y are represented by z.

(b) Let w and x be (not necessarily distinct) nodes constructed during the reduction of G, and let e be the edge constructed from w to x. If e represents (y,z) of G, then y is represented by w and z by x.

(c) In any graph formed while reducing G, all nodes and edges represent disjoint sets of objects (nodes and edges).

Lemma 2: Let $G = (N,E,n_0)$ be an rfg, and let $N_1 \subseteq N$ and $E_1 \subseteq E$ be a set of nodes and edges represented by a single node at some stage of the reduction of G. Then there is a (unique) node m in N_1 such that (N_1,E_1,m) is a region of G with header m.

PARSES AND BACKWARD EDGES

Since T1 and T2 may be applied to an rfg in different sequences, it becomes necessary to discuss specific sequences of applications of T1 and T2. Informally, a "parse" of an rfg is a list of the reductions made (T1 or T2) and the regions to which they apply.

Formally, a parse π of an rfg $G = (N,E,n_0)$ is a sequence of the form (T1,u,v,S) or (T2,u,v,w,S), where u, v, and w are names of nodes and S is a set of edges. We define the parse of an rfg recursively as follows:

(a) A single node with no edge has only the empty sequence as its parse.

(b) If G' (which may not be the original flow graph in a sequence of reductions) is reduced to G" by an application of T1 to node u, and the resulting node is named v in G", then (T1,u,v,S) followed by a parse of G" is a parse of G', where S is the set of edges represented by the edge (u,u) eliminated from G'.

(c) If G' is reduced to G" by an application of T2 to nodes u and v (with u consuming v), and the resulting node is called w, then (T2,u,v,w,S) followed by a parse of G" is a parse of G', where S is the set of edges represented by the edge (u,v) in G'.

(d) In both (b) and (c) above, "representation" in G' carries over to G". That is, whatever an object represents in G' is also represented by that object in G", except for those changes in representation caused by the particular transformation (T1 or T2) currently being applied.

Let G be an rfg and let π be a parse of G. We say that an edge of G is a backward edge with respect to π if it appears in set S of an entry (T1,u,v,S) of π and a forward edge otherwise.

The next two results appear in [HeU2].

Lemma 3: The backward edges of an rfg are unique.

Lemma 4: Edge (x,y) is a backward edge of an rfg if and only if x=y or y dominates x.

DEPTH-FIRST SPANNING TREES

A depth-first spanning tree (DFST) of a flow graph G is a directed, rooted, ordered spanning tree grown by Algorithm A [T1].

Algorithm A: DFST of a flow graph.

Input: Flow graph G with n nodes.

Output: (1) DFST of G. (2) A numbering of the nodes from 1 to n (i.e., ENDORDER(m), for each node m) indicating the order in which each node was last visited.

Method: A1. The root of the DFST is the initial node of G. Let this node be the node m which is visited first in Step A2. $i \leftarrow 1$.

A2. [Visit node m.] If node m has a successor x not already on the DFST, select x as the right-most son of m found so far in the spanning tree. If this step is success-

full, node x becomes the node m to be visited next by repeating Step A2. If there is no such x, go to Step A3.

A3: Let m be the node being visited ENDORDER(m) ← i. i ← i + 1.
If m is the root, then halt. Otherwise, climb down the DFST one node toward the root and visit this node again by returning to Step A2. □

If (u,v) is an edge in a DFST, then u is the father of v and v is the son of u. The ancestor and descendant relations are the transitive closures of the father and son relations.

Let $G = (N,E,n_0)$ be a flow graph and let $T = (N,E')$ be a DFST of G. The edges in E-E' fall into three classes

(a) Edges which run from ancestors to descendants we call forward edges.†

(b) Edges which run from descendants to ancestors or from a node to itself we call back edges.

(c) Edges which run between nodes which are unrelated by the ancestor-descendant relation we call cross edges.

The notion of "to the right" in a DFST has only been defined for nodes with the same father. We extend it by saying that if x is to the right of y, then all of x's descendants are to the right of all of y's descendants. Thus, if (u,v) is a cross edge of a DFST, then u is to the right of v.

Lemma 5: [HeU2] The backward edges of an rfg G are exactly the back edges of any DFST for G.

III. Node Ordering and a Dominator Algorithm

Let T be a DFST of a flow graph G with n nodes. We consider two orderings of the nodes of G.

(a) ENDORDER--as defined in Algorithm A.

(b) rENDORDER--where rENDORDER(x) = n+1-ENDORDER(x), for each node x. (rENDORDER is the reverse of ENDORDER.)

We define the dag of an rfg G to be G minus all of its back edges [HeU2].

† Do not confuse this definition of "forward" edges in a DFST with the previous one for edges in an rfg. They are not necessarily the same, and context should distinguish which one is meant.

Lemma 6: The partial order defined by the dag of an rfg is a subset of the total order defined by rENDORDER.

Proof: Let G be an rfg, let G' be the dag of G, and let T be any DFST of G. It suffices to show that if there is a path in G' from the initial node to node y which includes node x, with $x \neq y$, then rENDORDER(x) < rENDORDER(y).

Suppose, in contradiction, that there are two distinct nodes x and y such that there is a path in G' from the initial node to y which includes x, and rENDORDER(x) > rENDORDER(y). Then, ENDORDER(x) < ENDORDER(y). That is, y is last visited after x is last visited while growing T.

Either y is an ancestor of x, or y is "to the right" of x in T. If y is an ancestor of x, then G' contains a cycle. This is impossible. Consequently, y is to the right of x. The path from x to y must go through a common ancestor of x and y [T1], so there would again be a cycle in G'. □

If i is a predecessor of j in an rfg, then either (i,j) is a back edge or a forward edge of an rfg. If it is a back edge, then either j dominates i or i=j (Lemma 4), and thus, i cannot dominate j. If (i,j) is a forward edge of an rfg, then rENDORDER(i) < rENDORDER(j). This is exactly the property of rENDORDER which Algorithm B uses.

Algorithm B: Computes a set DOM(m), the dominators of m, for each node m.

Input: Reducible flow graph $G = (N,E,n_0)$, $|N|=n$. The nodes are numbered from 1 to n by rENDORDER according to some DFST for G. Refer to each node by its number.

Output: Sets DOM(j) $1 \leq j \leq n$, where i is in DOM(j) if and only if i dominates j.

Method: B1. Initially, DOM(1) ← ∅, and DOM(j) ← N for $j \neq 1$.

B2. For each node $j = 2,3,\ldots,n$ in turn, DOM(j) is replaced by the intersection of $[\{k\} \cup DOM(k)]$ over all predecessors k of j such that k < j. □

Theorem 1: Algorithm B is correct. That is, after Algorithm B terminates, i is in DOM(j) if and only if i dominates j.

Proof: Let G be an rfg. We proceed by induction on j.

Inductive Hypothesis: After processing node j, i is in DOM(j) if and only if i dominates j.

Basis: (j = 1). Trivially true.

Induction Step: (j > 1). Assume the inductive hypothesis for all k < j, and consider the case for j.

If i dominates j, then surely i dominates every predecessor of j which is not i itself. Thus, i is in DOM(j).

Now, suppose i is in DOM(j), but i does not dominate j. Then there is a cycle-free path from the initial node to j which does not pass through i. Let k be the node on the path immediately before j. By Lemma 4, (k,j) cannot be a back edge, else j would dominate k or k = j, and the path would have a cycle. Thus, (k,j) is a forward edge, and rENDORDER(k) < rENDORDER(j). As i ≠ k, and i does not dominate k, we have by the inductive hypothesis that i is not in {k} ∪ DOM(k), and hence, not in DOM(j). □

If we implement the DOM sets by bit vectors, then Algorithm B requires 0(e) bit vector steps.† This follows because in a flow graph with e edges at most e bit vector intersections are computed in Step B2. Also, the node ordering (rENDORDER) assumed as input can be computed in 0(e) steps [T1].

In [AU], Aho and Ullman present an 0(ne) step algorithm to compute dominators. Purdom and Moore's algorithm [PM] has the same time bound.

Allen and Cocke [AC] suggest breadth-first ordering of the nodes to compute dominators of an arbitrary graph, but their algorithm (which is similar to Algorithm B) may require more than one pass through the nodes.

Earnest et al [EBA] present an algorithm which establishes an "interval ordering" (similar to rENDORDER, but takes more than 0(e) steps. Aho, Hopcroft and Ullman [AHU] have an 0(e log e) step algorithm to find "direct" dominators in an rfg. In [T2], Tarjan presents an algorithm for determining direct dominators in 0(e + n log n) steps.

† We shall always distinguish between "steps" and "bit vector steps" when discussing complexity. This distinction is important.

Before leaving this section, we prove another result about rENDORDER.

Lemma 7: If x dominates y, the rENDORDER(x) < rENDORDER(y).

Proof: Let G be a flow graph in which x dominates y, and let T be any DFST of G. Since any path from the initial node to y must include x, x is reached before y while growing T. Thus, x is on the backward path in T from y to the initial node. That is, ENDORDER(y) < ENDORDER(x) and rENDORDER(x) < rENDORDER(y). □

Note that Lemma 7 is not just a corollary of Lemma 6. Lemma 7 applies to non-reducible as well as reducible flow graphs.

IV. The Main Result

Following several lemmas, we establish the main result of this paper.

Definition: The depth of an rfg G is the largest number of back edges found in any cycle-free path in G.

Definition: Let G be a flow graph, let I(G) be the derived flow graph of G, and let G' be G minus all of its self-loops, where a self-loop is an edge from a node to itself. We define the length k of the derived sequence of G to be 0 if G' is the trivial flow graph, otherwise that k ≠ 0 such that

(a) $G_0 = G'$

(b) $G_{i+1} = I(G_i)$, $i \geq 0$,

(c) G_k is the limit flow graph of G,

and (d) $G_k \neq G_{k-1}$.

Lemma 8: Let G be an rfg and let G' be G at some intermediate stage of its reduction by T1 and T2. If there is a path from node u to node v in G', then there exist nodes w and x in G such that w and x are respectively represented by nodes u and v in G' and there is a path from w to x in G.

Proof: Let π be any parse of G which yields G' at some intermediate stage. The lemma is an easy induction on the number of steps of π taken to reach G'. □

Lemma 9: Let G be an rfg. Nodes entered by back edges in G head intervals in G.

Proof: The lemma is obvious for self-loops. So, let (m,h) be a back edge in G and suppose m ≠ h. Thus, h dominates m by Lemma 4. If h is the initial node, the

lemma follows. Now consider where h is not the initial node.

Suppose, in contradiction, that h does not head an interval in G. Since h must be in some interval, let it be in interval K with header k. First, we note that every interval is also a region by Lemma 2. If m is not in K, then K is not a region because conditions (b) and (c) of the definition of "region" are violated. Thus, m is in K.

As (m,h) is an edge, m must be added to K before h is. But then there is a path from the initial node to k and thence to m which does not pass through h. This would contradict the assumption that h dominates m. []

Lemma 10: If u dominates v in an rfg G, u heads an interval in G, J is the interval containing v, and $I(u) \neq J$, then I(u) dominates J in I(G).

Proof: Neither T1 nor T2 creates any new paths between nodes. Thus, if I(u) did not dominate J, then u would not dominate v. []

The following lemma is essential for theorem which follows.

Lemma 11: Let d be the depth of an rfg G, let d' be the depth of I(G), and suppose $G \neq I(G)$. If $d > d'$, then $d = d' + 1$.

Proof: Assume all the hypotheses and let P be any cycle-free path in G from p_1 to p_k containing d back edges. We shall think of P as an ordered sequence of edges $P = ((p_1,p_2),(p_2,p_3),\ldots,(p_{k-1},p_k))$, where the j-th edge in P is (p_j,p_{j+1}). Let (x_i,s_i) be the i-th back edge in P, $1 \leq i \leq d$. That is, (x_1,s_1) is the edge with the least m such that (p_m,p_{m+1}) is a back edge, and if the i-th back edge is (p_n,p_{n+1}), then the (i+1)-st back edge is the edge with the least $m > n$ such that (p_m,p_{m+1}) is a back edge. See Figure 1.

Let $S = \{s_i \mid (x_i,s_i)$ is a back edge in P$\}$. Since P is cycle-free, each s in S is distinct. Thus, $|S| = d$. Let $s_0 = p_1$.

First we show that s_{i+1} dominates s_i, for $0 \leq i \leq d-1$. Pick your favorite s_{i+1} from S. We know that s_{i+1} dominates x_{i+1} because (x_{i+1},s_{i+1}) is a back edge (Lemma 4), and we know that there is a path Q from s_i

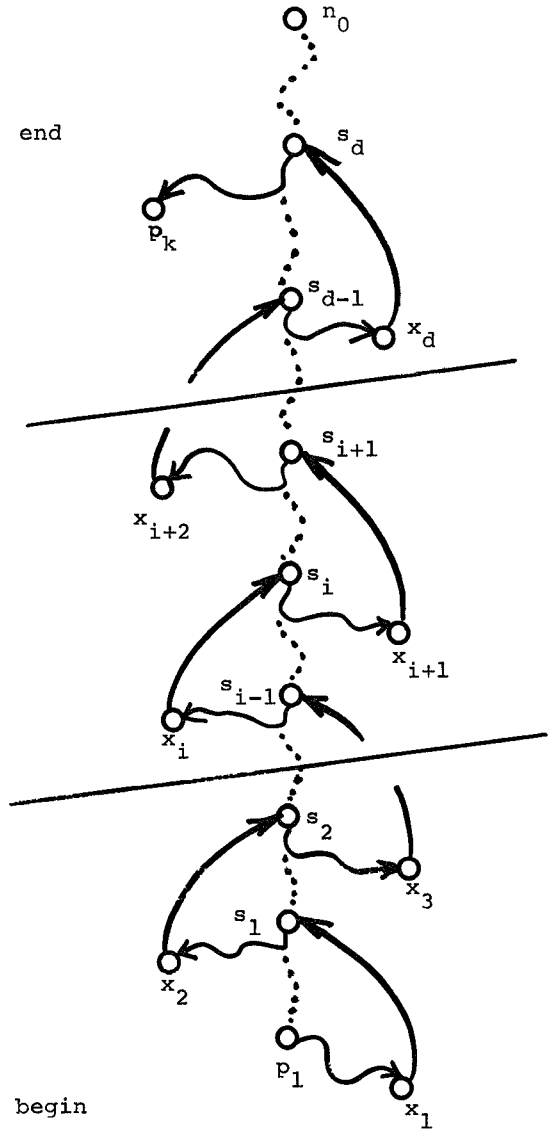

Figure 1. A cycle-free path in an rfg from p_1 to p_k containing $d > 0$ back edges.

to x_{i+1} in P which does not pass through s_{i+1}. Suppose, in contradiction, that s_{i+1} does not dominate s_i. Then, there is a path R from the initial node to s_i not containing s_{i+1}. But by concatenating paths R and Q we have a path to x_{i+1} not containing s_{i+1}. This contradicts the fact that s_{i+1} dominates x_{i+1}. Thus, s_{i+1} dominates s_i.

Now we claim that all back edges in P,

except the first one, are represented by themselves in I(G) and are back edges in I(G). That is, an edge in G represented by an edge in I(G) "still exists" as an edge in I(G), whereas an edge in G represented by a node in I(G) does not. To show this, it suffices to show that in I(G) the node representing the interval J (containing s_{i+1}) dominates, and is thus distinct from, the node representing interval K (containing x_{i+1}), where $0 \leq i \leq d-1$, and that the first back edge is represented by a node in I(G).

Since P is cycle-free, it follows that the i-th and (i+1)-st back edges are distinct and $s_i \neq s_{i+1}$. Thus by Lemma 9, J and the interval L containing s_i are distinct intervals of G. Furthermore, J dominates L in I(G), because s_{i+1} dominates s_i in G (lemma 10). See Figure 2.

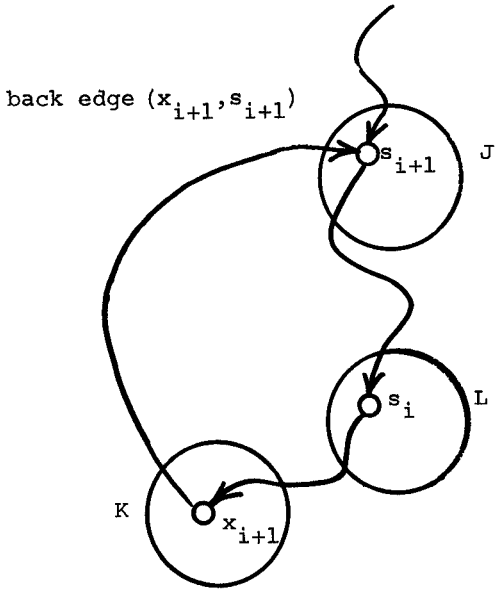

Figure 2. Intervals J, K and L of Lemma 11.

If K = L, then (x_{i+1}, s_{i+1}) represents itself in I(G) because it is an inter-interval edge. Also, it is a back edge in I(G) by Lemmas 10 and 4.

Now suppose that $K \neq L$, that is, x_{i+1} is not in L. Certainly, $K \neq J$ due to the forward path from s_i to x_{i+1} in P. Thus, J dominates K by Lemma 10. Hence, (x_{i+1}, s_{i+1}) represents itself in I(G) because it is an inter-interval edge. Also, it is a back edge in I(G) by Lemma 4.

Finally, if the first back edge represents itself in I(G), then d' = d, which contradicts the assumption d > d'. []

Theorem 2: (MAIN THEOREM) If G is an rfg with depth d and derived sequence length k, then $k \geq d$.

Proof: By induction on k.

Basis: (k = 0). G is the trivial flow graph. Thus, d = 0. Hence, $k \geq d$.

Induction Step: (k > 0). Assume the inductive hypothesis for k-1, and consider an rfg G with derived sequence of length k and depth d. Let d' be the depth of I(G).

Case 1: d > d'. Thus, d = d'+1 by Lemma 11. By the inductive hypothesis, $k-1 \geq d'$. Thus, $k-1 \geq d-1$, or $k \geq d$.

Case 2: d = d'. By the inductive hypothesis, $k-1 \geq d'$. Thus, $k \geq k-1 \geq d' = d$, or $k \geq d$.

Case 3: d < d'. This case cannot occur because T1 and T2, in their transformation of G to I(G), do not create paths between nodes in I(G) which did not already exist (Lemma 8). Hence, the back edges of I(G) are a "subset" of those of G. []

The significance of Theorem 2 is that, although the interval analysis algorithm must take about 2ke bit vector steps to solve a global flow analysis problem for an rfg with e edges [AU], there exists an obvious bit propagation algorithm to solve such problems in about de bit vector steps. (We pick up the coefficient 2 in the interval approach because, in addition to reducing the rfg to a single node, known algorithms ([A2], [A3], [AU], [C], [S] and [Ke]) reverse the interval process to propagate global information locally.)

Figure 3 shows an rfg with k = 3 and d = 1. Moreover, this rfg can be extended in an obvious way so that k is arbitrarily large, yet d remains 1. Thus, there may be a dramatic decrease in the time required to solve global data flow analysis problems using the simple bit propagation algorithm, when compared to the interval algorithm. In any event, the algorithm of [Ul] and [Ki] cannot be worse than interval analysis, and must be regarded as superior for its simplicity.

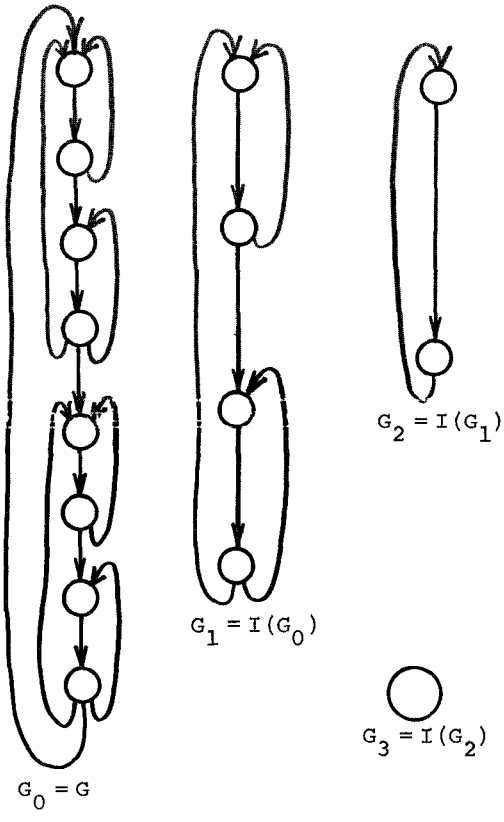

Figure 3. Flow graph G with d = 1 and k = 3.

V. Solution of Two Global Flow Analysis Problems

AVAILABLE EXPRESSIONS (From [C] and [U1].)

An expression such as A + B is available at a point p in a flow graph if every sequence of branches which the program may take to p causes A + B to have been computed after the last computation of A or B. If we can determine the set of available expressions at entrance to the nodes of a flow graph, then we know which expressions have already been computed prior to each node. Thus, we may be able to eliminate the redundant computation of some expressions within each node.

Let $\mathcal{E}$ be the set of expressions computed in a flow graph $G = (N,E,n_0)$.

Let $\mathcal{K}: N \to 2^{\mathcal{E}}$. We interpret $\mathcal{K}(x)$ as the set of expressions which are killed in node x. Informally, expression $A\theta B$ is killed if either A or B is defined within node x. (The symbol θ indicates a generic binary operator.)

Let $\mathcal{J}: N \to 2^{\mathcal{E}}$. If an expression $r = A\theta B$ is in $\mathcal{J}(x)$, then we imagine that r is generated within node x, and that neither A nor B is subsequently defined.

Let AEIN(x) and AEOUT(x), for each node x, be respectively the set of expressions available at entrance to and at exit from node x.

The fundamental relationships which enable us to compute AEIN(x) for each node x are:

AE1. $AEIN(n_0) = \phi$.

AE2. For $x \neq n_0$, AEIN(x) is the intersection of AEOUT(y) over all predecessors y of x.

AE3. $AEOUT(x) = [AEIN(x) - \mathcal{K}(x)] \cup \mathcal{J}(x)$, for each node x.

AE4. Since AE1-3 do not necessarily have a unique solution for AEIN(n), we want the largest solution.

The algorithm which follows is a bit vector algorithm and similar to those in [U1] and [Ki], except for the node ordering. We distinguish between sets and bit vectors by using AEIN for sets and AEin for bit vectors.

Algorithm C: Computes bit vectors AEin(m) for each node m.

Input: (1) Flow graph $G = (N,E,n_0)$, $|N| = n$. The nodes are numbered from 1 to n by reversing the time of last visit in a DFST of G (i.e., rENDORDER). Refer to each node by its number.

(2) Bit vectors KILL(j) and GEN(j), $1 \leq j \leq n$, where the i-th bit of KILL(j) (resp. GEN(j)) is 1 if and only if the i-th expression is in $\mathcal{K}(j)$ (resp. $\mathcal{J}(j)$). All bit vectors have length p, where p is the number of expressions.

Output: Bit vectors AEin(j), $1 \leq j \leq n$.

Method: C1. Initially, AEin(j) ← all 1's, for $2 \leq j \leq n$, and AEin(1) ← all 0's.

C2. Do Step C3 for j = 1,2,...,n in order. If any bit changes for any j, repeat Step 2. Otherwise, halt.

C3. Set AEin(j) equal to the bitwise product of $[AEin(k) \wedge \neg KILL(k)] \vee GEN(k)$,† where

† Here, the symbols $\wedge$, $\vee$ and $\neg$ stand for the AND (bitwise product), OR (bitwise sum) and NOT (bitwise complement) functions, respectively.

k ranges over all predecessors of node j. □

LIVE VARIABLES (From [Ke].)

A path in a flow graph is called definiton-clear with respect to a variable V if there is no definition of V on that path. A variable V is live at a point p in a flow graph if there is a definition-clear path for V from p to a use of V. That is, V is live if its current value might be used before V is redefined. Having determined the set of live variables at exit from each node in a flow graph, we can use this information for (among other things) register allocation--we can determine when a value should be kept in a register because of a subsequent use.

Let $\mathcal{V}$ be the set of variables occurring in a flow graph $G = (N,E,n_0)$.

Let $\mathcal{C}: N \to 2^{\mathcal{V}}$. $\mathcal{C}(x)$, the clear of x, is the set of variables which are not defined in node x.

Let $\mathcal{U}: N \to 2^{\mathcal{V}}$. $\mathcal{U}(x)$ is the set of variables which have exposed uses in node x, i.e., those variables with a definition-clear path from the entry of node x to a use within node x.

Let LVOUT(x) and LVIN(x), for each node x, be the set of variables live at exit from and on entrance to node x.

The fundamental relationships which enable us to compute LVOUT(x) for each node x are:

LV1. For each exit node w in G (i.e., w has no successors), LVOUT(w) = ϕ.

LV2. For x not an exit node, LVOUT(x) is the union of LVIN(y) over all successors y of x.

LV3. LVIN(x) = [LVOUT(x) $\cap$ $\mathcal{C}$(x)] $\cup$ $\mathcal{U}$(x), for each node x.

LV4. Since LV1-3 do not necessarily have a unique solution for LVOUT(x), we want the smallest such solution.

Let LVout be the bit vector for set LVOUT.

Algorithm D: Computes bit vectors LVout(m) for each node m.

Input: (1) Flow graph $G = (N,E,n_0)$, $|N| = n$. The nodes are numbered from 1 to n by the time of last visit in a DFST of G (i.e., ENDORDER. Refer to each node by its number.

(2) Bit vectors CLEAR(j) and XUSE(j), $1 \le j \le n$, where the i-th bit of CLEAR(j) (resp. XUSE(j)) is 1 if and only if the i-th variable is in $\mathcal{C}$(j) (resp. $\mathcal{U}$(j)). All bit vectors have length q, where q is the number of variables.

Output: Bit vectors LVout(j), $1 \le j \le n$.

Method: D1. Initially, LVout(j) $\leftarrow$ all 0's, for $1 \le j \le n$.

D2. Do Step D3 for j = 1,2,...,n in order. If any bit changes for any j, repeat Step D2. Otherwise, halt.

D3. Set LVout(j) equal to the bitwise sum of [LVout(k) $\wedge$ CLEAR(k)] $\vee$ XUSE(k), where k ranges over all successors of node j. □

VI. Analysis

The termination and correctness of Algorithms C and D follow directly from [U1] and [Ki]. We focus on the complexity.

Lemma 12: Any cycle-free path in an rfg G beginning with the initial node is monotonically increasing by rENDORDER.

Proof: Any such path contains no back edges by the proof of Lemma 11, and, thus, is a path in the dag of G. rENDORDER topologically sorts the dag of G (Lemma 6). □

Theorem 3: Step C2 of Algorithm C is executed at most d + 2 times for an rfg G.

Proof: A 0 propagates from its point of "origin"--a "kill" or the initial node--to the place where it is needed in d + 1 iterations if it must propagate along a path P of d back edges. It takes one iteration for a 0 to arrive at the tail of the first back edge of P. This follows since all edges to this point are forward or cross edges. The numbers along the path must be in increasing sequence by Lemma 12. After this point, it takes one iteration for a 0 to climb up each back edge in P to the tail of the next back edge, by the same argument. Hence, we need at most d + 1 iterations to propagate information plus one more to detect that there are no further changes. □

Theorem 4: Step D2 of Algorithm D is executed at most d + 2 times for an rfg G.

Proof: A 1 indicating a use propagates backward along a cycle-free path to a given point in d + 1 iterations if there are d back edges in the path from the point to the use. It takes one iteration for a 1

to reach the head of the d-th back edge in such a path. As in Theorem 3, we prove this by noting that forward and cross edges under ENDORDER go from higher to lower numbered nodes. An additional iteration enables us to reach the head of each succeeding back edge. []

VII. Conclusions

There is an ordering of the nodes of a flow graph G which (i) topologically sorts the dominance relation of G,

(ii) topologically sorts the dag of G if G is reducible, and

(iii) can be found in O(e) steps.

As a direct consequence, we can compute the dominators of each node in a reducible flow graph in O(e) bit vector steps.

Also, we have analyzed a simple bit propagation approach for solving global data flow analysis problems which is simple to describe, understand, and program. This approach requires at most (d+2)e bit vector steps, whereas the interval approach requires at most 2ke bit vector steps plus bookkeeping for intervals, where k and d, with $k \geq d$, are parameters of the rfg.

Moreover, although "node splitting" is necessary when using the interval approach on non-reducible flow graphs, the simple bit propagation approach works on non-reducible flow graphs directly with no such modification!

***

References

[AHU] A.V. Aho, J.E. Hopcroft and J.D. Ullman, "On Finding Lowest Common Ancestors in Trees," Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, pp. 253-265, May 1973.

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

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

[A2] F.E. Allen, "Control Flow Analysis," SIGPLAN Notices, Vol. 5, No. 7, pp. 1-19, July 1970.

[A3] F.E. Allen, "A Basis for Program Optimization," Proc. IFIP Conf. 71, North Holland Publishing Co., Amsterdam, 1971.

[AC] F.E. Allen and J. Cocke, "Graph-Theoretic Constructs for Program Control Flow Analysis," IBM Research Report RC 3923, T.J. Watson Research Center, Yorktown Heights, N.Y., July 1972.

[C] J. Cocke, "Global Common Subexpression Elimination," SIGPLAN Notices, Vol. 5, No. 7, pp. 20-24, July 1970.

[EBA] C.P. Earnest, K.G. Balke, and J. Anderson, "Analysis of Graphs by Ordering of Nodes," JACM, Vol. 19, No. 1, pp. 23-42, Jan. 1972.

[HeU1] M.S. Hecht and J.D. Ullman, "Flow Graph Reducibility," SIAM J. Computing, Vol. 1, No. 2, pp. 188-202, June 1972.

[HeU2] M.S. Hecht and J.D. Ullman, "Characterizations of Reducible Flow Graphs," TR-118, Computer Science Laboratory, Electrical Eng. Dept., Princeton Univ., Jan. 1973.

[HoU] J.E. Hopcroft and J.D. Ullman, "An n log n Algorithm for Detecting Reducible Graphs," Proc. 6th Annual Princeton Conf. on Information Sciences and Systems, pp. 119-122, March 1972.

[Ke] K. Kennedy, "A Global Flow Analysis Algorithm," International J. Computer Math., Vol. 3, pp. 5-15, Dec. 1971.

[Ki] G.A. Kildall, "Global Expression Optimization at Compile Time," in this proceedings, Oct. 1973.

[LM] E.S. Lowry and C.W. Medlock, "Object Code Optimization," CACM, Vol. 12, No. 1, pp. 13-22, Jan. 1969.

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

[PM] P.W. Purdom and E.F. Moore, "Immediate Predominators in a Directed Graph," CACM, Vol. 15, No. 8, pp. 777-778, August 1972.

[S] M. Schaefer, A Mathematical Theory of Global Flow Analysis, to appear, Prentice Hall, Englewood Cliffs, N.J., 1973.

[T1] R.E. Tarjan, "Depth-First Search and Linear Graph Algorithms," SIAM J. Computing, Vol. 1, No. 2, pp. 146-160, June 1972.

[T2] R.E. Tarjan, "Finding Dominators in Directed Graphs," to appear in Proc. 7th Annual Princeton Conf. on Information Sciences and Systems, March 1973.

[T3] R.E. Tarjan, "Testing Flow Graph Reducibility," Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, pp. 96-107, May 1973.

[U1] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions," TR-106, Computer Science Laboratory, Dept. of Electrical Eng., Princeton Univ., March 1972.

[U2] J.D. Ullman, "A Fast Algorithm for the Elimination of Common Subexpressions," Proc. 13th Symposium on Switching and Automata Theory, pp. 161-176, Oct. 1972.

[V] V.A. Vyssotsky, private communication of June 7, 1973.