SIAM J. COMPUT.
Vol. 4, No. 4, December 1975

A SIMPLE ALGORITHM FOR GLOBAL DATA FLOW ANALYSIS PROBLEMS*

MATTHEW S. HECHT† AND JEFFREY D. ULLMAN‡

Abstract. A simple, iterative bit propagation algorithm for solving global data flow analysis problems such as "available expressions" and "live variables" is presented and shown to be quite comparable in speed to the corresponding interval analysis algorithm. This comparison is facilitated by a result relating two parameters of a reducible flow graph (rfg). Namely, if G is an rfg, 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 $k \geqq d$. (Intuitively, k is the maximum nesting depth of loops in a computer program, while d is a measure of the maximum loop-interconnectedness.) The node ordering employed by the simple algorithm is the reverse of the order in which a node is last visited while growing any depth-first spanning tree of the flow graph. In addition, a dominator algorithm for an rfg is presented which takes O(edges) bit vector steps.

Key words. global code improvement, flow graph, reducibility, interval analysis, dominance, depth-first spanning tree, data flow analysis, available expressions, live variables

1. Introduction. When analyzing computer programs for code improvement [2], [4], [19], [22], 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 data flow analysis problems are "available expressions" [8], [26], "live variables" [14], "reaching definitions" [5], [6] and "very busy variables" [22]. There are several general algorithms to solve such problems.

The "interval" approach [2], [5]–[8], [14], [22] collects relevant 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 partition process.

Another approach[1] [16], [26], [27] propagates information in a simple iterative manner until all the required information is collected; that is, until the process converges. It is this approach which will be described and analyzed in this study. We shall show that this second approach (with a suitable node ordering) is time competitive with the interval approach!

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

---

* Received by the editors May 21, 1973, and in revised form August 15, 1974. This work was supported by the National Science Foundation under Grant GJ-1052.

† Department of Computer Science, University of Maryland, College Park, Maryland 20742.

‡ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.

[1] In 1961, V. A. Vyssotsky [27] implemented this kind of flow analysis (and presumably the simple iterative algorithm) in a Bell Laboratories 7090 FORTRAN II compiler—for strictly diagnostic purposes.

2. Background.

2.1 Basic definitions. A *flow graph* [2], [5], [10], e.g., is a triple $G = (N, E, s)$, where:

(a) N is a finite set of *nodes*. Let $n = |N|$.

(b) E is a subset of $N \times N$ called the *edges*. Let $e = |E|$. 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 $(x_1, \cdots, x_k)$ such that (x_i, x_{i+1}) is in E for $1 \leqq i \leqq k-1$. The *path length* of $(x_1, \cdots, x_k)$ is $k-1$. If $x_1 = x_k$ and $k > 1$, the path is a *cycle*.

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

Let G be flow graph and let h be a node of G. The *interval with header h* [2], [5], [10], [22], e.g., denoted by $I(h)$, is constructed as follows:

(a) First, set $I(h)$ to $\{h\}$.

(b) **while** m is a node not yet in $I(h)$ **and** $m \neq s$ **and** all predecessors of m are in $I(h)$ **do** add m to $I(h)$ **end**

Observe that although m in the while-loop 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 the while-loop will, if it is not chosen, still be a candidate at the next iteration.

There is a certain way to choose interval headers so that a flow graph is uniquely partitioned into disjoint intervals. This process takes $O(e)$ steps [2], [5].

If G is a flow graph, then the *derived flow graph* of G [2], [5], [10], [22], denoted by $I(G)$, is defined as follows:

(a) The nodes of $I(G)$ are the intervals of G determined from the unique interval partitioning 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 $J \neq K$. Note that there cannot be an edge from J entering a node other than the header of K.

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

The sequence $G = G_0, G_1, \cdots, G_k$ is called the *derived sequence* for G [2], [5], [10], [22] if $G_{i+1} = I(G_i)$ for $0 \leqq i \leqq k-1$, $G_{k-1} \neq G_k$, and $I(G_k) = G_k$. G_k is called the *limit flow graph* of G [2], [10].

A flow graph is called *reducible* (an rfg) [2], [5], [10], [22] if and only if its limit flow graph is the single node with no edge (henceforth called the *trivial flow graph*). Otherwise it is called *irreducible* or *nonreducible*.

Let G be a flow graph and let (x, x) be an edge of G. Transformation $T1$ [10] is removal of this edge.

Let y not be the initial node and have a single predecessor, x. Transformation $T2$ [10] 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 [10] which interest us here. First, if $T1$ and $T2$ are applied to a flow graph until no longer possible, 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.

If x and y are two distinct nodes in a flow graph G, then x *dominates* y [2], [5], [11], [19], [20] if every path in G from the initial node to y contains x.

Let $G = (N, E, s)$ be a flow graph, let $N_1 \subseteq N$, let $E_1 \subseteq E$, and let h be in N_1. We say $R = (N_1, E_1, h)$ is a *region* of G with *header* h [11], [26] if in every path $(x_1, \cdots, x_k)$, where $x_1 = s$ and x_k is in N_1, there is some $i \leqq k$ such that

(a) $x_i = h$; and

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

(c) $(x_i, x_{i+1}), (x_{i+1}, x_{i+2}), \cdots, (x_{k-1}, x_i)$ 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 [26].

We say that each node and edge in the original flow graph *represents* itself. If $T1$ is applied to node x with edge (x, x), then the resulting node *represents* what node x and edge (x, x) 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.

In [26] it was established that a node at any stage of the reduction of an rfg represents a region.

Since $T1$ and $T2$ may be applied to an rfg in different sequences, it becomes necessary to discuss specific sequences of application 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 $G = (N, E, s)$ [26] is a sequence with elements of 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) The trivial flow graph 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 of G 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 of G represented by the edge (u, v) in G'.

If G is an rfg and π is a parse of G, we call an edge of G a *backward edge wrt* (with respect to) π [26] if for some u and v this edge appears in set S of an entry $(T1, u, v, S)$ of π.

The next two results appear in [11].

LEMMA 1. *The backward edges of an* rfg *are independent of the parse.*

LEMMA 2. *Edge* (x, y) *is a backward edge of an* rfg *if and only if* $x = y$ *or* y *dominates* x.

2.2. Depth-first search. A *depth-first spanning tree* (DFST) of a flow graph G is a directed, rooted, ordered spanning tree of G grown by Algorithm A [23]. This

algorithm also defines an ordering on the nodes of G which we call[2] rPOSTORDER (reverse POSTORDER).

ALGORITHM A: Computes rPOSTORDER for the nodes of a flow graph G.

Input. $G = (N, E, s)$ is a flow graph with n nodes numbered from 1 to n in an arbitrary manner. G is represented by successor (adjacency) lists. That is, for each node x let $S(x) = \{y | (x, y) \in E\}$.

Output. A numbering of the nodes from 1 to n (rPOSTORDER) indicating the reverse of the order in which each node was last visited.

Method.

Initially all nodes are marked "new".

There is a global integer variable i with initial value n.

There is a global integer array rPOSTORDER[1 : n].

The algorithm consists of a call to DFS(s), where DFS is the procedure defined below.

```
recursive procedure DFS(x)
    mark x "old"
    while S(x) is not empty
        do
            select and delete a node y from S(x)
            if y is marked "new"
                then
                    /* add (x, y) to DFST */
                    call DFS(y)
                end
        end
    rPOSTORDER[x] ← i
    i ← i − 1
    end DFS  □
```

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

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

(a) Edges which go from ancestors to decendants we call *forward edges wrt T.*

(b) Edges which go from descendants to ancestors or from a node to itself we call *back edges wrt T.*

(c) Edges which go between nodes that are unrelated by the ancestor-descendant relation we call *cross edges wrt T.*

Since a DFST is an ordered tree, the children of each node z are ordered from left-to-right so that "younger" children of z are to the right of "older" children of z. We extend the notion of "to the right" in a DFST by saying that if x is to the

---

[2] The reason why the name "rPOSTORDER" was chosen here is worthy of mention. In the 1974 edition of [17], Knuth has renamed the binary tree traversals which were formerly (preorder, postorder, endorder) in the 1968 edition to (preorder, inorder, postorder), respectively. A DFST T of a flow graph is an ordered tree. The order in which a node was last visited while growing T is the POSTORDER (à la 1974 edition of [17]) traversal of T when T is considered as a general ordered tree.

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 wrt a DFST, then u is to the right of v [23].

LEMMA 3 [11]. *The backward edges of an* rfg *G are exactly the back edges of a* DFST for G.

In view of Lemmas 1 and 3, we can safely confuse backward edges and back edges in an rfg.

3. A dominator algorithm. Developing an idea of [7] leads, when combined with depth-first search, to a linear bit vector algorithm to find dominators.

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

(a) rPOSTORDER—as defined in Algorithm A. According to this order, x precedes y if rPOSTORDER[x]<rPOSTORDER[y].

(b) POSTORDER—where POSTORDER[x] = $n+1-$rPOSTORDER[x], for each node x. Here x precedes y if POSTORDER[x]<POSTORDER[y], i.e., if rPOSTORDER[y]<rPOSTORDER[x].

We define the dag of an rfg G [11], [26] to be G minus all of its back edges.

LEMMA 4. rPOSTORDER *topologically sorts the* dag *of an* rfg. (That is, the partial order defined by the dag of an rfg is a subset of the total order defined by rPOSTORDER.)

Proof. Let G be an rfg, let G' be the dag of G, and let T be any DFST of G. By Lemma 3, G' is G without the back edges of T. 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 rPOSTORDER[x]<rPOSTORDER[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 rPOSTORDER[x]>rPOSTORDER[y]. Then POSTORDER[x]< POSTORDER[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 [23], 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 it is not. If it is a back edge, then either j dominates i or $i = j$ (Lemma 2), and thus i cannot dominate j. If (i, j) is not a back edge of an rfg, then rPOSTORDER[i]< rPOSTORDER[j]. These properties of rPOSTORDER are exploited in Algorithm B.

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

Input. $G = (N, E, s)$ is an rfg with n nodes represented by predecessor lists. That is, for each node y, let $P(y) = \{x | (x, y) \in E\}$. We assume that the nodes are numbered from 1 to n by rPOSTORDER, and we refer to each node by this number.[3]

---

[3] This assumption can be implemented in one of two ways. One way is to replace the statement "rPOSTORDER[x]← i" by "rPOSTORDER[i]← x" in Algorithm A, and then use rPOSTORDER appropriately. Another way is to just use a "bucket sort" (e.g., see [3]) to renumber nodes.

Output. Sets DOM(j), $1 \leqq j \leqq n$, where $i \in$ DOM(j) iff i dominates j.
Method.

```
DOM(1) ← ∅
for j ← 2 to n
    do
        DOM(j) ← ∩_{k∈P(j),k<j} ({k} ∪ DOM(k))
    end □
```

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 ($j = 1$). This is 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 except i itself (if $i \in P(j)$). Thus i is placed in DOM(j) by Algorithm B.

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 2, (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 not a back edge and rPOSTORDER[k] < rPOSTORDER[j]. As $i \neq k$ and i not dominate k, we have by the induction hypothesis that i is not in $\{k\} \cup$ DOM(k) and hence not in DOM(j). □

If the DOM sets are implemented by bit vectors, then Algorithm B requires $O(e)$ bit vector steps. This follows because in a flow graph with e edges at most e bit vector intersections are computed in the for-loop of Algorithm B. Also, the node ordering (rPOSTORDER) assumed as input can be computed in $O(e)$ steps [23].

In [2], Aho and Ullman present an $O(ne)$ step algorithm to compute dominators. Purdom and Moore's algorithm [21] has the same time bound.

Allen and Cocke [7] suggest breadth-first ordering of the nodes to compute dominators of an arbitrary graph, but their algorithm (to which Algorithm B is similar) may require more than one pass through the nodes.

Earnest et al. [9] present an algorithm which establishes an "interval ordering" (similar to rPOSTORDER) but takes more than $O(e)$ steps to compute.

Aho, Hopcroft and Ullman [1] give an $O(e \log e)$ step algorithm to find immediate (closest) dominators in an rfg. In [24], Tarjan presents an algorithm for determining immediate dominators in $O(e + n \log n)$ steps for an arbitrary graph.

4. The central result. Following several lemmas, we establish the central result of this paper.

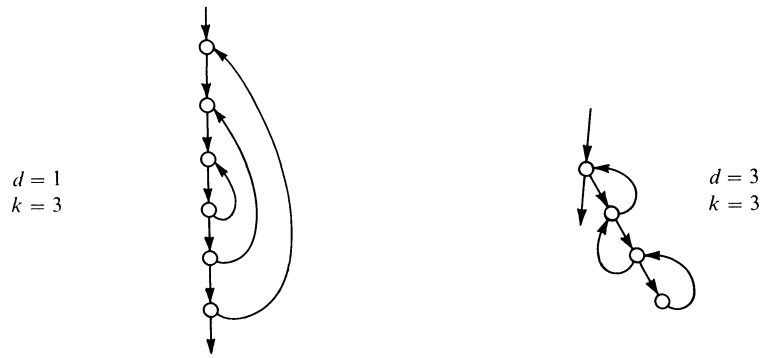
DEFINITION. The *loop-interconnectedness parameter* of an rfg G, which we shall denote by $d(G)$ or simply d, is the largest number of back edges found in any cycle-free path in G.

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 of the derived sequence* of G to be 0 if G' is the trivial flow graph, and otherwise it is that $k \neq 0$ such that

(a) $G_0 = G'$,
(b) $G_{i+1} = I(G_i)$, $i \geqq 0$,
(c) G_k is the limit flow graph of G, and
(d) $G_k \neq G_{k-1}$.

The length of the derived sequence of an rfg corresponds intuitively to the maximum "nesting depth" of the loops of a computer program. The "interconnectedness" of the loops of a computer program can be an entirely different thing. For example, Fig. 1 shows a possible configuration of three nested for-loops versus three nested while-loops with the corresponding k and d values.



(i) *Three for-loops* (ii) *Three while-loops*

FIG. 1. "*Maximum nesting depth of loops*" (k) *vs.* "*loop-interconnectedness*" (d)

LEMMA 5. *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. The lemma is an easy induction on the number of steps of π taken to reach G'. □

LEMMA 6. *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 $m \neq h$. Thus h dominates m by Lemma 2. If h is the initial node, the lemma follows. Now consider the case where h is not the initial node.

Suppose h is in interval K but does not head K. Then by the method of constructing intervals, we must conclude that m is in K, since (m, h) is an edge.

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 the header of K and thence to m which does not pass through h. This would contradict the assumption that h dominates m. □

LEMMA 7. *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 create any new paths between nodes. That is, edges of $I(G)$ are based precisely on edges of G. Thus, if $I(u)$ did not dominate J, then u would not dominate v. □

LEMMA 8. *Let d be the loop-interconnectedness parameter of an* rfg G, *let d' be the loop-interconnectedness parameter of $I(G)$, and suppose $G \neq I(G)$. Then $d' \geqq d-1$.*

Proof. Assume all the hypotheses, and let P be any circle-free path in G from p_1 to p_m containing d back edges. We write P as an ordered sequence of edges, $P=[(p_1, p_2), (p_2, p_3), \cdots, (p_{m-1}, p_m)]$, where the jth edge in P is (p_j, p_{j+1}). Let $[(x_1, r_1), (x_2, r_2), \cdots, (x_d, r_d)]$ be the subsequence of P consisting of all and only the back edges of P. (See Fig. 2.) Note that the r_i's are distinct, since P has no cycles.

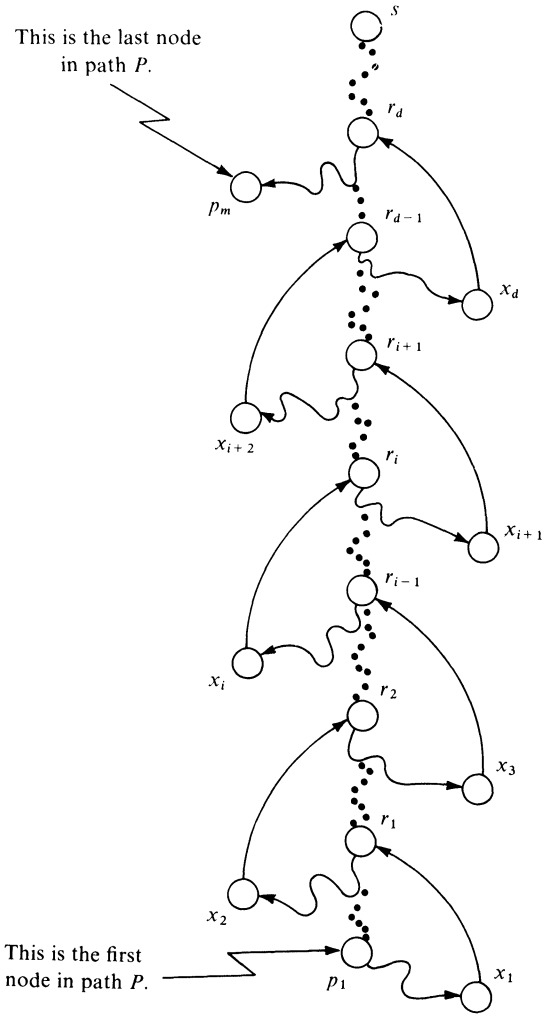


FIG. 2. *A cycle-free path P in an* rfg *from p_1 to p_m containing $d>0$ back edges*

Let J_y denote the interval containing the node y. By Lemma 6, each r_i heads an interval J_{r_i}. Also, there is a path of nonback edges from each r_i to x_{i+1}, so x_{i+1} cannot be in the interval $J_{r_{i+1}}$. Thus we have $J_{x_{i+1}} \neq J_{r_{i+1}}$, and the edge $(J_{x_{i+1}}, J_{r_{i+1}})$ is in $I(G)$. Furthermore, we know that r_{i+1} dominates x_{i+1} by Lemma 2, and so by Lemma 7, $J_{r_{i+1}}$ dominates $J_{x_{i+1}}$ in $I(G)$. Now by Lemma 2 again, we may conclude that $(J_{x_{i+1}}, J_{r_{i+1}})$ is a back edge of $I(G)$. That is to say, each back edge of G, except possibly the first, is preserved in $I(G)$. If the first is also preserved, then $d' = d$. Otherwise, $d' = d - 1$. □

THEOREM 2 (Central Theorem). *If G is an* rfg *with loop-interconnectedness parameter d and derived sequence length k, then $k \geqq d$.*

Proof. Let $d = d_0, d_1, \cdots, d_k$ be the loop-interconnectedness parameters of the members of the derived sequence. By Lemma 8, we know that $d_{i-1} \leqq d_i + 1$ for $1 \leqq i \leqq k$. Also, $d_k = 0$, since the last graph in the derived sequence is trivial. An elementary induction on i shows that $d_{k-i} \leqq i$. Thus $d_0 \leqq k$, as was to be proved. □

We shall postpone explaining the significance of Theorem 2 until after the next two sections. In these sections we present and then analyze a simple, iterative, bit propagation algorithm for global data flow analysis problems.

5. Solution of two global data flow analysis problems.

5.1. Available expressions (as in, e.g., [8], [26]). An expression such as $A + B$ is *available* at 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 $\mathscr{E}$ be the set of *expressions* computed in a flow graph $G = (N, E, s)$.

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

Let $\mathscr{G} : N \to 2^{\mathscr{E}}$. If an expression $r = A \,\theta\, B$ is in $\mathscr{G}[x]$, then we imagine that r is *generated* within node x, and that neither A nor B is subsequently assigned within x.

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[s] = $\varnothing$.

AE2. For $x \neq s$, AEIN[x] is the intersection of AEOUT[y] over all predecessors y of x.

AE3. AEOUT[x] = (AEIN[x] − $\mathscr{K}$[x]) ∪ $\mathscr{G}$[x] for each node x.

AE4. Since AE1–AE3 do not necessarily have a unique solution for AEIN[x], we want the largest solution.

The algorithm which follows is a bit vector algorithm and similar to those in [16], [26] and [27], except for the node ordering. We distinguish between sets and bits vectors by using AEIN for sets and AEin for bit vectors. Algorithm C is essentially AE3 "plugged into" AE2.

ALGORITHM C: Computes bit vectors AEin[x] for each node x.

Input. $G = (N, E, s)$ is a flow graph with n nodes represented by predecessor lists. That is, for each node y let $P(y) = \{x | (x, y) \in E\}$. The nodes are numbered from 1 to n by rPOSTORDER. We refer to each node by its rPOSTORDER number.

Bit vectors NOTKILL[j] and GEN[j], $1 \leqq j \leqq n$, are input where the ith bit of NOTKILL[j] (resp. GEN[j]) is 1 if and only if the ith expression is not in $\mathscr{K}[j]$ (resp. in $\mathscr{G}[j]$). All bit vectors have length p, where p is the number of expressions.

Output. Bit vectors AEin[j], $1 \leqq j \leqq n$.

Method.

```
AEin[1] ← all 0's
for j ← 2 to n do AEin[j] ← all 1's end
change ← true
while change
    do
        change ← false
        for j ← 2 to n    /*rPOSTORDER */
            do
                previous ← AEin[j]
                AEin[j] ← ⋀_{k∈P(j)} ((AEin[k] ∧ NOTKILL [k]) ∨ GEN[k])[4]
                if previous ≠ AEin[j] then change ← true end
            end
    end □
```

5.2. Live variables (as in, e.g., [14]). A path in a flow graph is called *definition-clear* with respect to a variable v if there is no definition of v (by assignment or input) 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 register allocation—we can determine when a value should be kept in a register because of a subsequent use.

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

Let $\mathscr{C} : N \to 2^{\mathscr{V}}$. $\mathscr{C}[x]$, the *clear* of x, is the set of variables which are not defined in node x.

Let $\mathscr{U} : N \to 2^{\mathscr{V}}$. $\mathscr{U}[x]$ 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. If x has no successors, then LVOUT[x] $= \varnothing$.

---

[4] Here, the symbols $\wedge$, $\vee$ and $\neg$ stand for the AND (bitwise product), OR (bitwise sum) and NOT (bitwise complement) functions, respectively. Note also that this expression can be evaluated for k once on each pass, after the new value of AEin[k] is computed, and then used subsequently without recomputation.

LV2. Otherwise, LVOUT[x] is the union of LVIN[y] over all successors y of x.

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

LV4. Since LV1–LV3 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 which follows is just an iterative version of LV3 "plugged into" LV2, with a suitable initialization for LVOUT, and with a suitable node ordering.

Algorithms C and D are not exactly the same. Algorithm D propagates information opposite to the direction of control flow, while C propagates it in the same direction. In Algorithm D we visit the nodes in POSTORDER rather than the reverse. Also, here we are computing OUT's rather than IN's.

ALGORITHM D: Computes bit vectors LVout[x] for each node x.

Input. $G = (N, E, s)$ is a flow graph with n nodes represented by successor lists. That is, for each node x let $S(x) = \{y | (x, y) \in E\}$. The nodes are numbered from 1 to n by rPOSTORDER. We refer to each node by its rPOSTORDER number. It is assumed that each node has at least one successor. A node without any successors can be combined with its predecessors first, with a resulting gain in efficiency.

Bit vectors CLEAR[j] and XUSE[j], $1 \leqq j \leqq n$, are input where the ith bit of CLEAR[j] (resp. XUSE[j]) is 1 if and only if the ith variable is in $\mathscr{C}$[j] (resp. $\mathscr{U}$[j]). All bit vectors have length q, where q is the number of variables.

Output. Bit vectors LVout[j], $1 \leqq j \leqq n$.

Method.

```
for j ← 1 to n do LVout[j] ← all 0's end
change ← true
while change
    do
        change ← false
        for j ← n to 1 by −1   /*POSTORDER */
            do
                previous ← LVout[j]
                LVout[j] ← ⋁_{k∈S(j)} ((LVout[k] ∧ CLEAR[k]) ∨ XUSE[k])[5]
                if previous ≠ LVout[j] then change ← true end
            end
    end  □
```

6. Analysis. The termination and correctness of Algorithms C and D follow directly from [16] and [26]. We focus on the complexity.

THEOREM 3. *The while-loop 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

[5] Analogous to Algorithm C, note that this expression can be evaluated for k once on each pass, after the new value of LVout[k] is computed, and then used subsequently without recomputation. Algorithms C and D are only presented without this feature so that each algorithm will be more readable.

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 the numbers along the path must be in increasing sequence according to rPOSTORDER. 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 iteration to detect that there are no further changes. □

THEOREM 4. *The while-loop 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 dth back edge in such a path. The proof is then analogous to that of Theorem 3. □

THEOREM 5. *If we ignore initialization, Algorithm* C *(or Algorithm* D*) takes at most* $(d+2)(e+n)$ *bit vector steps, where a bit vector step is the logical* $\wedge$ *or* $\vee$ *of bit vectors.*

Proof. Since Algorithms C and D are almost identical, we mostly refer to Algorithm D below. We assume that Algorithm D has been rewritten so that the expression $((\text{LVout}[k] \wedge \text{CLEAR}[k]) \vee \text{XUSE}[k])$ is evaluated for k once on each pass. Also, we assume that each node has at least one successor, since any node without successors can be easily combined with its predecessors first, with a resulting gain in efficiency.

For each iteration of the while-loop of Algorithm D, $2n$ bit vector steps are used to evaluate $((\text{LVout}[k] \wedge \text{CLEAR}[k]) \vee \text{XUSE}[k])$ for all k, that is, n nodes at 2 bit vector steps per node. (In Algorithm C this process uses $2(n-1)$ bit vector steps.) At most $e-n$ bit vector steps are aggregately used to perform $\vee$ over all successors of each node, because each node with m successors requires $m-1$ bit vector steps, and e successors less one for each of n nodes yields $e-n$. Since the while-loop is executed at most $d+2$ times (Theorem 3 and 4), we have at most $(d+2)(e-n+2n) = (d+2)(e+n)$ bit vector steps. □

7. Discussion and conclusions.

The given iterative bit propagation algorithm for global data flow analysis problems is conceptually simple, it is very easy to program, and the length of the resulting program is quite short. It takes $O(e)$ steps to compute the rPOSTORDER numbering of the nodes, and at most $(d+2)(e+n)$ bit vector steps to propagate the required information in a reducible graph. Moreover, the bit propagation algorithm, with no modification whatsoever, works on nonreducible graphs.

The interval analysis algorithm for global data flow analysis problems is conceptually complicated, it is quite difficult to program, and the length of the resulting program is very long. Nonreducible graphs present an additional time and programming complication.

Kennedy [15] has done some detailed comparisons of our Algorithm D and his algorithm for live variable analysis [14]. While [15] bounds the number of bit vector steps for the algorithm of [14] more carefully than for our algorithm, it is basically correct and indicates that on practical problems our algorithm will run in

time comparable to the interval analysis algorithm but may take twice as long as some flow graphs.

We can obtain a *lower* bound of $2(e+n)$ bit vector steps for any of the published interval analysis algorithms [2], [5], [6], [8], [14], [22]. In comparison, the *upper* bound for our algorithm in Theorem 5 is $(d+2)(e+n)$. Thus the ratio of our time to that of interval analysis is at most $d/2+1$, and may be less. Knuth [18] in a study of 50 FORTRAN programs found all to be reducible with derived sequence length $k \leqq 6$ and an average k of 2.75. Since $d \leqq k$ is Theorem 2, we can expect an average value of $d/2+1$ to be no more than 2.4. Thus, we may claim that on the average, propagation algorithms take no more than 2.4 times that of the corresponding interval analysis algorithms.

The above analysis has, however, been overly conservative. There are a number of other factors which tend both to argue that (a) the ratio 2.4 is higher than it should be, and (b) propagation algorithms have other virtues which make them preferable to the interval approach.

1. Propagation algorithms handle nonreducible flow graphs with no added effort, in fact, without even noting they are nonreducible.
2. The coding for a propagation algorithm is trivial in comparison with the coding necessary to implement an interval analysis algorithm.
3. The interval algorithms require additional work finding and manipulating intervals. This work requires no bit vector steps and so was not reflected in the calculation. In comparison, the cost of a depth-first search, the only significant part of our algorithm not reflected in the count of bit vector steps, is negligible.
4. In the above analysis, we assumed $k = d$. In fact, Fig. 1 indicates that $d < k$ is quite possible, especially in FORTRAN programs whose jumps are all caused by DO statements.
5. Without increasing the difficulty of coding our algorithm beyond that of the interval algorithms, we could check whether the predecessors of a node n had had their values changed since the last time we visited n. If not, no calculation at n is necessary for the current pass.

When all the above issues are taken into account, we believe a strong case can be made for using bit propagation algorithms for data flow analysis.

Acknowledgment. We appreciate the help of a referee who simplified our proofs of Lemma 8 and Theorem 2.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *On finding lowest common ancestors in trees*, Proc. 5th Ann. ACM Symp. on Theory of Computing, Austin, Tex., May 1973, pp. 253–265.

[2] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling: Vol. II—Compiling*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[3] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[4] F. E. ALLEN, *Program optimization, Annual Review Automatic Programming*, vol. 5, Pergamon Press, New York, 1969.

[5] ———, *Control flow analysis*, SIGPLAN Notices, (1970), 5, pp. 1–19.

[6] ———, *A Basis for Program Optimization*, Proc. IFIP Congr. 71, North-Holland, Amsterdam, 1971.

[7] F. E. ALLEN AND J. COCKE, *Graph theoretic constructs for program control flow analysis*, IBM Res. Rep. RC 3923, IBM T. J. Watson Res. Center, Yorktown Heights, N.Y., 1972.

[8] J. COCKE, *Global common subexpression elimination*, SIGPLAN Notices, 5 (1970), pp. 20–24.

[9] C. P. EARNEST, K. G. BALKE AND J. ANDERSON, *Analysis of graphs by ordering of nodes*, J. Assoc. Comput. Mach., 19(1972), pp. 23–42.

[10] M. S. HECHT AND J. D. ULLMAN, *Flow graph reducibility*, this Journal, 1 (1972), pp. 188–202.

[11] ———, *Characterizations of reducible flow graphs*, J. Assoc. Comput. Mach., 21 (1974), pp. 367–375.

[12] M. S. HECHT, *Topological sorting and flow graphs*, Proc. IFIP Congr. 74, August 1974, pp. 494–499.

[13] J. E. HOPCROFT AND J. D. ULLMAN, *An n* log *n algorithm for detecting reducible graphs*, Proc. 6th Ann. Princeton Conf. on Information Sciences and Systems, March 1972, pp. 119–122.

[14] K. KENNEDY, *A global flow analysis algorithm*, Internat. J. Comput. Math., 3 (1971), pp. 5–15.

[15] ———, *A comparison of global data flow analysis programs*, Rice Tech. Rep. 476-093-1, Rice Univ., Houston, Tex., 1974.

[16] G. A. KILDALL, *A unified approach to global program optimization*, Conf. Rec. of ACM Symp. on Principles of Programming Languages, Boston, Oct. 1973, pp. 194–206.

[17] D. E. KNUTH, *The Art of Computer Programming: Vol. I—Fundamental Algorithms*, 2nd ed., Addison–Wesley, Reading, Mass., 1974.

[18] ———, *An Empirical Study of FORTRAN Programs*, Software Practice and Experience, April (1971), pp. 105–134.

[19] E. S. LOWRY AND C. W. MEDLOCK, *Object code optimization*, Comm. ACM, 12 (1969), pp. 13–22.

[20] R. T. PROSSER, *Applications of Boolean matrices to the analysis of flow diagrams*, Proc. Eastern Joint Computer Conf., Spartan Books, New York, 1959, pp. 133–138.

[21] P. W. PURDOM AND E. F. MOORE, *Immediate predominators in a directed graph*, Comm. ACM, 15 (1972), pp. 777–778.

[22] M. SCHAEFER, *A Mathematical Theory of Global Program Optimization*, Prentice–Hall, Englewood Cliffs, N.J., 1973.

[23] R. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[24] ———, *Finding dominators in directed graphs*, Proc. 7th Ann. Princeton Conf. on Information Sciences and Systems, March 1973.

[25] ———, *Testing flow graph reducibility*, Proc. 5th Ann. ACM Symp. on Theory of Computing, Austin, Tex., May 1973, pp. 96–107.

[26] J. D. ULLMAN, *Fast algorithms for the elimination of common subexpressions*, Acta Informatica, 2 (1973), pp. 191–213.

[27] V. A. VYSSOTSKY, Private communication, June 7, 1973.