Global Data Flow Analysis and Iterative Algorithms

JOHN B. KAM AND JEFFREY D. ULLMAN

Princeton University, Princeton, New Jersey

ABSTRACT Kildall has developed data propagation algorithms for code optimization in a general lattice theoretic framework. In another direction, Hecht and Ullman gave a strong upper bound on the number of iterations required for propagation algorithms when the data is represented by bit vectors and depth-first ordering of the flow graph is used The present paper combines the ideas of these two papers by considering conditions under which the bound of Hecht and Ullman applies to the depth-first version of Kildall's general data propagation algorithm. It is shown that the following condition is necessary and sufficient Let f and g be any two functions which could be associated with blocks of a flow graph, let x be an arbitrary lattice element, and let 0 be the lattice zero Then $(*)$ $(\forall f,g,x)$ $[fg(0) \geq g(0) \wedge f(x) \wedge x]$ Then it is shown that several of the particular instances of the techniques Kildall found useful do not meet condition $(*)$

KEY WORDS AND PHRASES. code optimization, data flow analysis, reducible flow graph, semilattice, depth-first search, constant propagation, available expressions

CR CATEGORIES. 4.12, 5.24, 5.25

1. *Introduction*

Performing compile time optimization involves solving a class of problems each of which can be dealt with in essentially the same manner. These problems, called "global data flow analysis problems," involve determination and collection of information which is distributed throughout the program.

The interval approach [1–4, 9, 12] has been used to solve this class of problems when the flow graph of the program has a property called "reducibility." A second approach, using iteration of a data propagation step, has recently appeared in the literature [10, 13]. Its origins go back to at least 1961, when it was used by Vyssotsky in a Fortran compiler [14]. Developments regarding this approach, which we shall term the "iterative approach," have taken two directions. First, Kildall [10] expresses the class of problems which can be solved using the iterative approach in a very general lattice theoretic framework. The bit-vector representation of data used in previous work on data propagation is a special case of the techniques described in [10].

The second direction which research into iterative methods has taken is typified by [5]. There, considering only bit-vector represented data, it was shown that depth-first search provided an efficient ordering of the nodes of a flow graph, and in fact $d + 2$ iterations were sufficient for the usual kinds of data flow problems, where d, the *loop interconnectedness parameter* of a flow graph, is the maximum number of back edges (according to any depth-first spanning tree) in a cycle-free path. In practice, d is often 3 or less [11].

In the present paper we provide a necessary and sufficient condition for the bound of [5][1] to apply to Kildall's lattice theoretic formulation of flow analysis problems when

This work was supported by the National Science Foundation under Grant GJ-1052.

Authors' address: Department of Electrical Engineering, Princeton University, Princeton, NJ 08540

[1] Actually, for technical reasons, we use the bound $d + 3$ instead of $d + 2$.

depth-first ordering of the nodes is used. We then see that the applications of his algorithm suggested by Kildall do not meet the criterion. It is possible, however, that orderings other than depth-first may make iteration efficient in these cases, and we do not wish to imply that Kildall's techniques are inefficient under all possible circumstances

2. *Background*

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

(1) N is a finite set of *nodes*.

(2) E is a subset of $N \times N$ 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.

(3) n_0 in N is the *initial node*. There is a *path*[2] from n_0 to every node.

A *depth-first spanning tree*[3] (DFST) of a flow graph is a tree with order on the sons of any node (ordered tree) grown by Algorithm D [7].

```
ALGORITHM D: DFST of a flow graph $G$
Input.  Flow graph $G$ with $n$ nodes.
Output. (1) A DFST for $G$, (2) a numbering *rPostorder* of the nodes from 1 to $n$ indicating the
        reverse of the order in which each node was last visited.
Method.
[D1] The root of the DFST is the initial node of $G$. Let this node be the node $m$ which is visited in
     step D2  $i$ will be used to number nodes in rPostorder  Initially, $i \leftarrow n$.
[D2] [visit node $m$] If node $m$ has a successor $x$ not already on the DFST, make $x$ the rightmost
     son of $m$ so far placed in the spanning tree, adding edge $(m,x)$ to the tree. If such an $x$ is
     found, it becomes the node $m$ to be visited next by repeating step D2 on $x$. If there is no such
     $x$, go to step D3.
[D3] Let $m$ be the node being visited,
     rPostorder $(m) \leftarrow i$,
     $i \leftarrow i - 1$;
     **if** $m$ is the root **then halt**
     **else** execute step D2 on the father of $m$                                              □
```

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

(1) Edges which run from a node to a proper descendant are called *forward edges*.[4]

(2) Edges which run from a node to an ancestor (including itself) are called *back edges*.

(3) Edges which run between nodes unrelated by the ancestor-descendant relation are called *cross-edges*.

Observation 1. Let $G = (N,E,n_0)$ be a flow graph and let T be a DFST of G. Let a and b be nodes in G. Then (b,a) in E is a back edge if and only if rPostorder(b) $\geq$ rPostorder(a).

Observation 2. Let $G = (N,E,n_0)$ be a flow graph and T be a DFST of G. Then every cycle of G contains at least one back edge.

Definition. Let $G = (N,E,n_0)$ be a flow graph and $T = (N,E')$ be a DFST of G We define $d(G,T)$, the *loop connectedness of G with respect to T*, to be the largest number of back edges found in any cycle-free path of G. Often, when T is understood, we shall write $d(G)$ for $d(G,T)$. For the wide class of flow graphs known as "reducible" flow graphs [3], it has been shown [6] that $d(G,T)$ is in fact independent of which DFST T is chosen

[2] A *path* from n_1 to n_k is a sequence of nodes $n_1, n_2, \cdots, n_k$ such that (n_i, n_{i+1}) is in E for $1 \leq i \leq k - 1$. The path *length* is $k - 1$ If $n_1 = n_k$ and $k > 1$, the path is a *cycle*.

[3] For a quick definition, a *tree* is a flow graph such that no node has more than one predecessor The term *root* of a tree is a synonym for initial node, *son* is used for successor, and *father* for predecessor. *Ancestor* and *descendant* are terms used for the reflexive and transitive closure of the father and son relations, respectively.

[4] Some authors call edges in E' *tree* edges, reserving "forward" for edges not in E

Having introduced the terminology needed for flow graphs, we now proceed to the second area in which a series of definitions are necessary, namely lattice algebra. A *semilattice* is a set L with a binary *meet* operation $\wedge$ such that for all a, b, and c in L:

$$a \wedge a = a \text{ (idempotent)}, \quad a \wedge b = b \wedge a \text{ (commutative)},$$
$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \text{ (associative)}.$$

Given a semilattice L and elements $a, b \in L$, we say that $a \geq b$ if and only if $a \wedge b = b$, $a > b$ if and only if $a \geq b$ and $a \neq b$. We also extend the notation of the meet operation by saying $\bigwedge_{1 \leq i \leq n} x_i = x_1 \wedge x_2 \wedge \cdots \wedge x_n$.

A semilattice L is said to have a *zero element* $\mathbf{0}$ if for all $x \in L$, $\mathbf{0} \wedge x = \mathbf{0}$. L is said have a *one element* $\mathbf{1}$, if for all $x \in L$, $\mathbf{1} \wedge x = x$. We assume from here on that every semilattice has a zero element, but not necessarily a one element.

Given a semilattice L, a sequence $x_1, x_2, \cdots, x_n$ of elements of L is said to be a *chain* if for $1 \leq i < n$ we have $x_i > x_{i+1}$. L is said to be *bounded* if for each $x \in L$ there is a constant b_x such that any chain beginning with x has length at most b_x.

If L is bounded, then we can take meets over countably infinite sets if we define $\bigwedge_{x \in S} x$, where $S = \{x_1, x_2, \cdots\}$, to be $\lim_{n \to \infty} \bigwedge_{1 \leq i \leq n} x_i$. The fact that L is bounded assures us that the limit does exist.

3. *Global Data Flow Problems*

Following [10], we treat data flow analysis problems as follows. We choose a semilattice L and attach to its elements a "meaning," normally data which could reach a point in a flow graph. We associate with each node of the flow graph a function f from L to L which intuitively represents how data is transformed when control passes through the block of code represented by that node.

In what follows we find it necessary to consider the set of all functions which could be associated with some node of a flow graph. That is, having selected a semilattice L and an intended meaning for lattice elements, the admissible functions are those which reflect the action of straight-line blocks of code on elements of L. We abstract the notion of such a set of functions in the following definition.

Given a bounded semilattice L, a set of functions F on L is said to be *an admissible set of functions for L* if and only if the following conditions are satisfied:

[F1] Each $f \in F$ distributes over $\wedge$, i.e. for any x and y in L, $f(x \wedge y) = f(x) \wedge f(y)$.

[F2] There exists an identity function e in F such that for all $x \in L$, $e(x) = x$.

[F3] F is closed under composition, i.e. f and g in F implies $fg \in F$, where for all $x \in L$, $[fg](x) = f(g(x))$.

[F4] For each $x \in L$, there exists a finite subset $H \subseteq F$ such that $x = \bigwedge_{f \in H} f(\mathbf{0})$.

Conditions [F2] and [F3] reflect obvious properties of straight-line blocks of code. That is, [F3] comes from the fact that the concatenation of two blocks is a block, and [F2] comes from the reasonable assumption that a block can be empty. [F1], on the other hand, is not universally true. It is used in [10] to prove the uniqueness of the output to Kildall's algorithm. The justification for condition [F4] is the following lemma.

Observation 3. Let L be a bounded semilattice and let F be a set of functions on L such that F satisfies [F1] with respect to L. Then for any finite subset $J \subseteq L$,

$$f\Big(\bigwedge_{x \in J} x\Big) = \bigwedge_{x \in J} f(x).$$

LEMMA 1. *Given a bounded semilattice L and F a set of functions on L satisfying* [F1]–[F3], *if we let*

$$L' = \{x \mid \exists f_1, \cdots, f_n \in F \text{ and } x = \bigwedge_{1 \leq i \leq n} f_i(\mathbf{0})\},$$

then F is an admissible set of functions for L'.

PROOF. Since any subset of L, which is closed under $\wedge$, satisfies the idempotent, commutative, and associative properties with respect to $\wedge$ and hence is a semilattice, it suffices to show that F satisfies [F1]–[F3] with respect to L' if F satisfies [F1]–[F3] with respect to L.

[F1] and [F2] are trivially satisfied by F with respect to L', because L' is a subset of L. Assume F does not satisfy [F3] with respect to L', i.e. there exists $x \in L'$ and $f, g \in F$ such that $fg(x) \in L - L'$. We want to draw a contradiction. $x \in L'$ implies $x = \bigwedge_{h \in H} h(\mathbf{0})$ for some finite $H \subseteq F$. Thus

$$fg(x) = fg(\bigwedge_{h \in H} h(\mathbf{0})) = \bigwedge_{h \in H} fgh(\mathbf{0}) \quad \text{(by Observation 3).}$$

Hence $fg(x)$ should have been included in L' by definition. □

Some additional useful observations are the following.

Observation 4. Given bounded semilattice L and associated F, for all $f \in F$ and $x, y \in L$, $x \geq y$ implies $f(x) \geq f(y)$.

Observation 5. For any bounded semilattice L and any countable $J \subseteq L$, if for all $x \in J$ we have $x \geq y$, then $\bigwedge_{x \in J} x \geq y$.

We now introduce the basic formalism for our expression of data flow problems. A *data flow analysis framework* is a triple $D = (L,\wedge,F)$ where L is a bounded semilattice with meet $\wedge$, and F is an admissible set of functions for L. A *particular instance of* $D = (L,\wedge,F)$ is a pair $I = (G,M)$ where (1) $G = (N,E,n_0)$ is a flow graph and (2) $M: N \rightarrow F$ is a function which maps each node in N to a function in F.

Convention. Given a particular instance $I = (G,M)$ of $D = (L,\wedge,F)$, if the nodes of G are labeled by rPostorder with respect to a DFST of G, we associate the nodes of G with their labels. We let f_i denote $M(i)$, the function in F which is associated with node i. Let $P = i_1, i_2, \cdots, i_m, i_{m+1}$ be a path in G. Then we may use $f_P(\cdot)$ for $f_{i_m}(f_{i_{m-1}}(\cdots f_{i_1}(\cdot)\cdots))$. Note that $f_{i_{m+1}}$ is not in the composition. If $m = 0$, then $f_P = e$, the identity function.

4. *The Depth-First Version of Kildall's Algorithm*

We now give an iterative algorithm to find what is essentially the maximum solution to the equations implied by an instance of a data flow problem. It is essentially the algorithm of [10] but with the important difference that the nodes are visited in turn in the rPostorder sequence. The following definitions are essential:

$\text{PRED}(j) = \{q \mid q \text{ is a predecessor of } j\}$.
$\text{PRED}^*(j) = \{q \mid q \in \text{PRED}(j) \text{ and } q < j \text{ in rPostorder, i.e. } (q,j) \text{ is not a back edge}\}$.

ALGORITHM K

Input. A particular instance $I = (G,M)$ of data flow analysis framework $D = (L,F)$, where $G = (N,E,n_0)$ is a flow graph with k nodes. Take N to be $\{1, 2, \cdots, k\}$, and assume the nodes are numbered by rPostorder Execute the program of Figure 1

Convention. We say that n iterations of Algorithm K have been applied, where $n \geq 1$, if the **for** loop beginning at step 1 has been executed once and the **for** loop beginning at step 2 has been executed $n - 1$ times.

THEOREM 1 [10]. *Given a particular instance $I = (G,M)$ of $D = (L,\wedge,F)$ as input, Algorithm K will eventually halt. At the completion of Algorithm K,*

$$A[i] = \bigwedge_{P \in PATH(i)} f_P(\mathbf{0}), \quad 1 \leq i \leq k,$$

where $PATH(i) = \{P \mid P$ is a path in G from node 1 to node $i\}$. □

LEMMA 2. *Given instance $I = (G,M)$ of data flow analysis framework $D = (L,\wedge,F)$, and T a DFST for G, after the n-th iteration of Algorithm K,*

$$A[i] = \bigwedge_{P \in PATH^{(n)}(i)} f_P(\mathbf{0}), \quad 1 \leq i \leq \text{k},$$

```
                begin
                  temp element of $L$,
                  $A$ array [1    $k$] of elements of $L$,
                  $j$ integer, change Boolean;
                  $A[1] = \mathbf{0}$,
    Step 1.       for $j = 2$ until $k$ do $A[j] = \wedge_{q\in PRED^*(j)} f_q(A[q])$; .
                  change = true
                  while change do
                    begin
                      change = false,
    Step 2            for $j = 2$ until $k$ do
                        begin
                          temp = $\wedge_{q\in PRED(j)} f_q(A[q])$,
                          if temp $\neq A[j]$ then
                            begin
                              change = true,
                              $A[j]$ = temp
                            end
                        end
                    end
                end
```

FIG. 1. Program for Algorithm K

where $PATH^{(n)}(i) = \{P \mid P \in PATH(i)$ *and* P *contains at most* $n - 1$ *back edges according to the rPostorder induced by* $T\}$.

PROOF. The proof is by induction on n, the number of iterations of Algorithm K already completed.

Basis ($n = 1$). We proceed by induction on the rPostorder j of the nodes of G.

Basis ($j = 1$). It is obvious that the trivial path is the only element in $\text{PATH}^{(1)}(1)$ because any edge entering node 1 has to be a back edge. Thus, $A[1]$ should equal $e(\mathbf{0}) = \mathbf{0}$. As we assign $A[1] := \mathbf{0}$ in the first iteration, the basis is done.

Induction step ($j > 1$). By Observation 1, all and only the back-edge-free paths from 1 to j can be written as $\langle P,j\rangle$,[5] where P is a path from 1 to $q \in \text{PRED}^*(j)$. By the induction hypothesis, $A[q] = \wedge_{P\in \text{PATH}^{(1)}(q)} f_P(\mathbf{0})$ for each $q \in \text{PRED}^*(j)$. Step 1 of Algorithm K assures

$$A[j] = \bigwedge_{q\in \text{PRED}^*(j)} f_q(A[q]) = \bigwedge_{q\in \text{PRED}^*(j)} \bigwedge_{P\in \text{PATH}^{(1)}(q)} f_{\langle P,j\rangle}(\mathbf{0}) = \bigwedge_{Q\in \text{PATH}^{(1)}(j)} f_Q(\mathbf{0}).$$

This completes the induction on j for the case $n = 1$.

Induction step ($n > 1$). We proceed by induction on j again.

Basis ($j = 1$). It is obvious that $\mathbf{0} = \wedge_{P\in \text{PATH}^{(1)}(1)} f_P(\mathbf{0}) \geq \wedge_{P\in \text{PATH}^{(n)}(1)} f_P(\mathbf{0})$. Thus $\wedge_{P\in \text{PATH}^{(n)}(1)} f_P(\mathbf{0}) = \mathbf{0}$. After n iterations of Algorithm K, $A[1]$ is still assigned $\mathbf{0}$, so $A[1] = \wedge_{P\in \text{PATH}^{(n)}(1)} f_P(\mathbf{0})$ after the nth iteration.

Induction step ($j > 1$). Every path Q in $\text{PATH}^{(n)}(j)$ is of the form $\langle P,q\rangle$ where either:

(i) $q \in \text{PRED}^*(j)$ and $P \in \text{PATH}^{(n)}(q)$, or

(ii) $q \in \text{PRED}(j) - \text{PRED}^*(j)$ and $P \in \text{PATH}^{(n-1)}(q)$.

That is, (ii) represents the case where edge (q,j) is a back edge, i.e. $q \geq j$, and (i) represents the opposite case, where $q < j$. When in Algorithm K temp is computed for j in the nth pass, $A[q]$ will be

(a) $\wedge_{P\in \text{PATH}^{(n)}(q)} f_P(\mathbf{0})$ if $q < j$, i.e. case (i) applies to q, and

(b) $\wedge_{P\in \text{PATH}^{(n-1)}(q)} f_P(\mathbf{0})$ if $q \geq j$, i.e. case (ii) applies.

Note that part (a) follows by the inductive hypothesis for j and (b) follows by the inductive hypothesis for n.

From the above it is immediate that on the nth pass temp is set equal to $\wedge_{Q\in \text{PATH}^{(n)}(j)} f_Q(\mathbf{0})$, and thus $A[j]$ is given this value if it does not already have it. □

[5] We use $\langle P, j\rangle$ to denote a path consisting of the nodes of path P followed by j. Similar, hopefully transparent notation will be used throughout.

LEMMA 3. *Given instance $I = (G,M)$ of framework $D = (L,\wedge,F)$, and a fixed rPostorder for G, Algorithm K will halt after no more then n iterations if and only if for each node $j \in N$, and for each path $P \in PATH(j)$ there exist $P_1, \cdots, P_r$, each in $PATH^{(n-1)}(j)$, and $f_P(\mathbf{0}) \geq \bigwedge_{1 \leq j \leq r} f_{P_j}(\mathbf{0})$.*

PROOF. (*if*). In this case by Observation 5 we have

$$\bigwedge_{Q \in \mathrm{PATH}^{(n)}(j)} f_Q(\mathbf{0}) = \bigwedge_{Q \in \mathrm{PATH}^{(n-1)}(j)} f_Q(\mathbf{0}),$$

so temp $= A[j]$ will always hold on the nth pass of Algorithm K, and change will be **false** at the end of that pass.

(*only if*). Suppose that Algorithm K halts after the mth pass, $m \leq n$. Then for arbitrary node j and $P \in \mathrm{PATH}(j)$ we must have $f_P(\mathbf{0}) \geq \bigwedge_{Q \in \mathrm{PATH}^{(m-1)}(j)} f_Q(\mathbf{0})$. We must show that there is a finite subset S of $\mathrm{PATH}^{m-1}(j)$ such that $f_P(\mathbf{0}) \geq \bigwedge_{Q \in S} f_Q(\mathbf{0})$. Enumerate $\mathrm{PATH}^{(m-1)}(j)$ as $Q_1, Q_2, \cdots$ in any order, and let $x_i = \bigwedge_{1 \leq k \leq i} f_{Q_k}(\mathbf{0})$. Surely $x_i \geq x_{i+1}$ for all i. By the boundedness condition there can only be a finite number of i's for which $x_i > x_{i+1}$. Let i_0 be the last and let $S = \{Q_1, Q_2, \cdots, Q_{i_0}\}$. Then $\bigwedge_{Q \in \mathrm{PATH}^{(m-1)}(j)} f_Q(\mathbf{0}) = \bigwedge_{Q \in S} f_Q(\mathbf{0})$, so S is the desired finite subset. □

5. *The Main Result*

We are now ready to characterize those data flow analysis frameworks for which depth-first search yields "rapid" convergence of Kildall's algorithm.

THEOREM 2. *Let $D = (L,\wedge,F)$ be a data flow analysis framework. Then Algorithm K halts after at most $d(G) + 3$ iterations*[6] *for every instance $I = (G,M)$ of D and every rPostorder definable for $G = (N,E,n_0)$, if and only if D satisfies condition $(*)$:*

$$(\forall f, g \in F)(\forall x \in L)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(x) \wedge x]. \qquad (*)$$

Formally, the theorem can be stated:

$$(\forall D)\{[(\forall I = (G,M) \text{ an instance of } D)(\text{Algorithm K halts after at most } d(G) + 3 \text{ iterations})] \equiv (*)\}$$

PROOF. (*if*). By Lemma 3, it suffices to show that for each $j \in N$ and each path $P \in \mathrm{PATH}(j)$, there exist paths $P_1, \cdots, P_m$ in $\mathrm{PATH}^{(d+2)}(j)$ such that $f_P(\mathbf{0}) \geq \bigwedge_{1 \leq i \leq m} f_{P_i}(\mathbf{0})$. We want to prove the above by induction on k, the number of back edges contained in $P = i_1, i_2, \cdots, i_r$, where $i_1 = 1$ and $i_r = j$.

Basis $(0 \leq k \leq d + 1)$. This case is trivial; just let $m = 1$ and $P_1 = P$.

Induction step $(k > d + 1)$. Since P contains more than $d + 1$ back edges, it cannot be cycle free by definition of d. Let us pick the highest number a such that $i_a = i_b$ for some $b > a$. We claim that the path $P_1 = i_1, \cdots, i_a$ must contain at least one back edge, for the path $i_{a+1}, \cdots, i_r$ is cycle free and thus has at most d back edges. (i_a, i_{a+1}) may be a back edge, but P is assumed to have more than $d + 1$ back edges, so the claim follows. The path $P_2 = i_a, \cdots, i_b$ contains at least one back edge by Observation 2. We let $P_3 = i_b, \cdots, i_r$, and let P_4 be a back-edge-free path from node 1 to node i_a as shown in Figure 2. It is a property of DFST's that P_4 exists, since P_4 may follow the tree.

Now we let $x = f_{P_4}(\mathbf{0})$.

$$\begin{aligned}
f_P(\mathbf{0}) &= f_{P_3}(f_{P_2}(f_{P_1}(\mathbf{0}))) && \text{by definition}\\
&\geq f_{P_3}(f_{P_1}(\mathbf{0}) \wedge f_{P_2}(x) \wedge x) && \text{by assumption}\\
&= f_{P_3}(f_{P_1}(\mathbf{0}) \wedge f_{P_2}(f_{P_4}(\mathbf{0})) \wedge f_{P_4}(\mathbf{0}))\\
&= f_{P_3}f_{P_1}(\mathbf{0}) \wedge f_{P_3}f_{P_2}f_{P_4}(\mathbf{0}) \wedge f_{P_3}f_{P_4}(\mathbf{0})\\
&= f_{P'}(\mathbf{0}) \wedge f_{P''}(\mathbf{0}) \wedge f_{P'''}(\mathbf{0}),
\end{aligned}$$

[6] Recall that $d(G)$ stands for $d(G,T)$ where T is the DFST defining the rPostorder in question

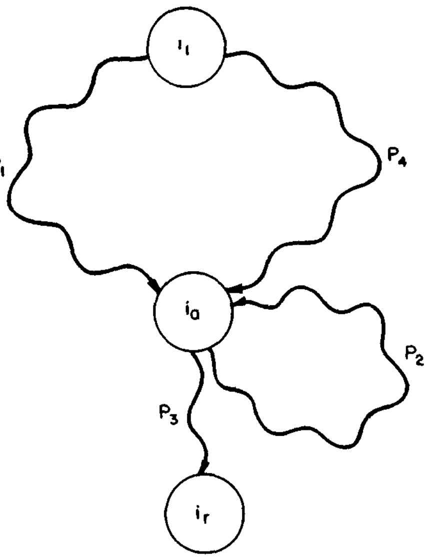
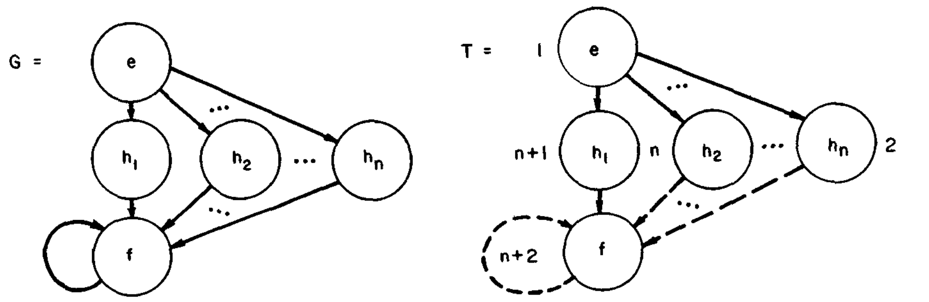


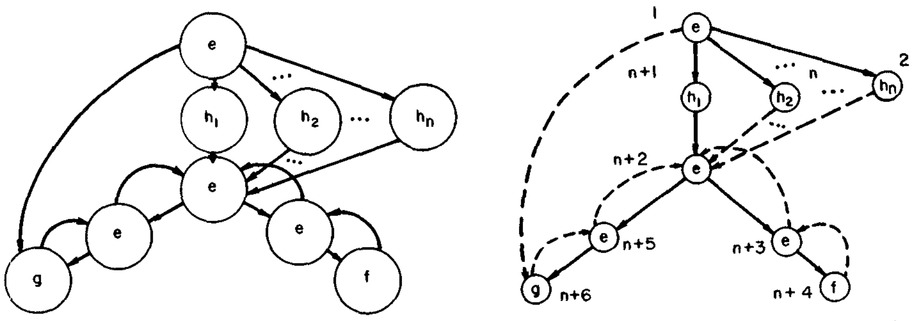
FIG 2. Decomposition of path P



a) FLOW GRAPH OF G

b) A DFST FOR G SHOWING r POSTORDER NUMBERING

FIG 3 Counterexample with $ff(x) \not\geq f(x) \wedge x$



a) FLOW GRAPH OF G

b) A DFST FOR G SHOWING r POSTORDER NUMBERING

FIG 4. Counterexample with $ff(x) \geq f(x) \wedge x$

where $P' = i_1, \cdots, i_a, i_{b+1}, \cdots, i_r$, $P'' = \langle P_4, i_{a+1}, \cdots, i_b, i_{b+1}, \cdots, i_r\rangle$, $P''' = \langle P_4, i_{b+1}, \cdots, i_r\rangle$. P', P'', and P''' are each paths in G and contain at most $k - 1$ back edges. The induction step follows immediately.

(*only if*). Suppose the condition $(*)$ is not satisfied, i.e.

$$(\exists x \in L)(\exists f, g \in F)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(x) \wedge x].$$

By condition [F4] of F being admissible for L, there exist $h_1, h_2, \cdots h_n \in F$ such that $x = \wedge_{1 \le i \le n} h_i(\mathbf{0})$. We have two cases to consider.

Case 1. $ff(x) \ngeq f(x) \wedge x$. The particular instance $I = (G,M)$ as shown in Figure 3 will do the job, where $d(G) = 0$. After the third iteration of Algorithm K we have $A[n+2] = x \wedge f(x) \wedge ff(x)$, while $A[n+2] = x \wedge f(x)$ after the second iteration. Hence Algorithm K will take at least $4 > (0+3)$ iterations.

Case 2. $ff(x) \ge f(x) \wedge x$. The particular instance $I = (G,M)$ with the DFST of G grown as shown in Figure 4 will do. We see $d(G,T) = 2$.

After the first iteration:

$A[n+2] = x$

$A[n+3] = x$

$A[n+4] = x$

$A[n+5] = x$

$A[n+6] = \mathbf{0}$

After the second iteration:

$A[n+2] = x$

$A[n+3] = x \wedge f(x)$

$A[n+4] = x \wedge f(x)$

$A[n+5] = x \wedge g(\mathbf{0})$

$A[n+6] = \mathbf{0}$

After the third iteration:

$A[n+2] = g(\mathbf{0}) \wedge f(x) \wedge x$

$A[n+3] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge ff(x)$

$= g(\mathbf{0}) \wedge f(x) \wedge x$ (by assumption)

$A[n+4] = g(\mathbf{0}) \wedge f(x) \wedge x$

$A[n+5] = g(\mathbf{0}) \wedge f(x) \wedge x$

$a[n+6] = \mathbf{0}$

After the fourth iteration:

$A[n+2] = g(\mathbf{0}) \wedge f(x) \wedge x$

$A[n+3] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$

$A[n+4] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$

$A[n+5] = g(\mathbf{0}) \wedge f(x) \wedge x$

$A[n+6] = \mathbf{0}$

After the fifth iteration:

$A[n+2] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$

$A[n+3] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0}) \wedge ffg(\mathbf{0})$

$A[n+4] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0}) \wedge ffg(\mathbf{0})$

$= g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$ (by assumption)

$A[n+5] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$

$A[n+6] = 0$

$A[n+2] = g(\mathbf{0}) \wedge f(x) \wedge x$ after the fourth iteration and $A[n+2] = g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$ after the fifth iteration. By the hypothesis that $(*)$ does not hold for f, g, and x, we have $g(\mathbf{0}) \wedge f(x) \wedge x \ne g(\mathbf{0}) \wedge f(x) \wedge x \wedge fg(\mathbf{0})$. Thus Algorithm K will take at least $6 > (d(G)+3) = 5$ iterations, and in fact, it does halt after six iterations.

In general, any $I = (G,M)$ with T' grown as shown in Figure 5 will take Algorithm K at least $2d + 2$ iterations before it halts. $\square$

We have completed the proof of Theorem 2. Let us remind the reader that Theorem 2 does not make the (false) claim that unless $(*)$ is satisfied for framework D, there is no instance $I = (G,M)$ of D for which Algorithm K takes at most $d(G) + 3$ iterations. Formally, the false statement is:

$(\forall D)(\forall$ instances $I = (G,M)$ of $D)$
[Algorithm K converges in $d(G) + 3$ iterations $\equiv (*)$].

It is also worth noting a few statements about a framework $D = (L,\wedge,F)$ which are equivalent to $(*)$ and a few that are not. We leave the proofs to the reader.

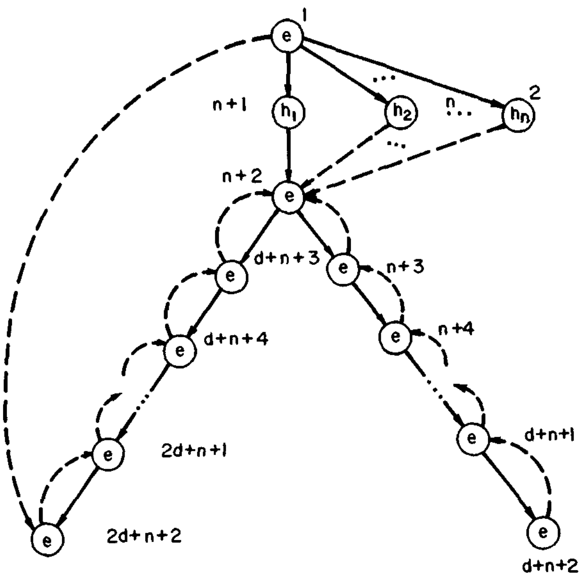


FIG. 5. DSFT T' generalizing Figure 4(b)

Observation 6. $(*)$ is equivalent to

$$(\forall f \in F)(\forall x, y \in L)[f(y) \geq y \wedge f(x) \wedge x]$$

and to

$$(\forall f, g \in F)(\forall x, y \in L)[fg(y) \geq g(y) \wedge f(x) \wedge x],$$

and if L has a one element $\mathbf{1}$, to

$$(\forall f, g \in F)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(\mathbf{1})]$$

and to

$$(\forall f \in F)(\forall x \in L)[f(x) \geq x \wedge f(\mathbf{1})].$$

Observation 7. $(*)$ is implied by

$$(\forall f \in F)(\forall x \in L)[f(x) \geq x]$$

and implies

$$(\forall f \in F)(\forall x \in L)[ff(x) \geq f(x) \wedge x],$$

but is equivalent to neither of these statements.

6. *Some Specific Data Flow Analysis Frameworks*

In his paper [10], Kildall handles constant propagation by a global data flow analysis framework CP = $(L,\wedge,F)$, where $V = \{X_1, X_2, \cdots\}$ for some infinite set of variable names; C is equal to the set of all values assumed by variables; L is the set of functions from finite subsets of V to C; $\mathbf{0} \in L$ is the function which is undefined for all X_i. The meet operation $\wedge$ on L is set intersection.[7] Intuitively, $z \in L$ stands for the information about variables which we may assume at certain points of the program flow graph. $(v,c) \in z$ implies the variable v has value c.

We define a notation for functions in F based on the sequence of assignment statements whose effect they are to model.

(1) For each assignment statement $X := \varphi(Y_1, \cdots, Y_m)$, $m \geq 1$, for X and $Y_1, \cdots,$

[7] Recall that a function $f\colon V \to C$ is a set of pairs (v,c) with $v \in V$ and $c \in C$. We shall henceforth treat members of L as subsets of $V \times C$.

Y_m in V, there is a function f in F such that for $z \in L$, $f(z) = z'$ is defined by:

(i) For $U \neq X$, $z'(U) = z(U)$ if $z(U)$ is defined, and $z'(U)$ is undefined otherwise.

(ii) $z'(X)$ is the value $\varphi(c_1, \cdots, c_m)$ if $z(Y_i) = c_i$ for $1 \leq i \leq m$, and $z'(X)$ is undefined otherwise.

We denote this function by $\langle X := \varphi(Y_1, \cdots, Y_m)\rangle$.

(2) For each assignment statement $X := c$, where $c \in C$ is a constant, the function f in F is defined as in (1), but (ii) reads:

(ii)′ $z'(X) = c$.

We denote this function by $\langle X := c\rangle$.

(3) For each assignment statement $X := Y$, where X and Y are in V, the function f in F is defined as in (1), but (ii) reads:

(ii)″ $z'(\mathrm{X}) = z(Y)$.

We denote this function by $\langle X := Y\rangle$.

(4) $e \in F$, where $e(z) = z$ for all $z \in L$.

(5) If $f, g \in F$, then $fg \in F$.

It happens that the framework CP is not distributive under the normal interpretation of "values" [8]. We shall show here that CP does not satisfy $(*)$. Then we shall give an interpretation of values for which CP is distributive and show that $(*)$ still is not satisfied.

THEOREM 3. *CP, with the usual arithmetic interpretation of "values," does not satisfy* $(*)$.

PROOF. Let x be $\{(A,3), (B,1), (C,2)\}$ and let g be the composition of the functions associated with the assignment statements: $D := 1$, $E := 2$, $A := D + E$,[8] i.e. $g = \langle A := D + E\rangle \circ \langle E := 2\rangle \circ \langle D := 1\rangle$. Let f be the function $\langle A := B + C\rangle$. Then $f(x) = x$, $g(\mathbf{0}) = \{(A,3), (D,1), (E,2)\}$, and $fg(\mathbf{0}) = \{(D,1), (E,2)\}$. Then $fg(\mathbf{0}) \neq g(\mathbf{0}) \wedge f(x) \wedge x = \{(A,3)\}$. □

In order that CP be distributive it is necessary that operator symbols be given a free interpretation (see [8]). That is, values are formulas involving integers or reals and the operator symbols. The effect of applying m-ary operator φ to formulas $F_1, F_2, \cdots, F_m$ is the formula $\varphi(F_1, F_2, \cdots, F_m)$.

THEOREM 4. *CP under a free interpretation of operators does not satisfy* $(*)$.

PROOF. The same proof as that for Theorem 3 goes through if we replace $(A,3)$ in x by $(A, 1 + 2)$. □

We should observe that only Theorem 4 is significant, since the effect on nondistributive frameworks of $(*)$ not holding has not been investigated. Also, the same remarks as we made here for the framework CP apply to the "structured partition" lattice of [10].

Now let us consider a case where $(*)$ does hold. The usual bit vector common subexpression detection strategy can be expressed as the data flow analysis framework CSE $= (L,\wedge,F)$, where L is the set of bit vectors of length n. For $x, y \in L$, $x \wedge y$ is the bitwise product (logical AND). The $\mathbf{0}$ element in L is the word of all 0's. Furthermore, L has a one element $\mathbf{1}$, namely the vector of all 1's.

F consists of functions f which may be denoted $\langle$KILL, GEN$\rangle$, where GEN and KILL are each n-bit vectors, with GEN $\wedge$ KILL $= \mathbf{0}$. For $z \in L$, we define $f(z) = (z \wedge \neg\mathrm{KILL}) \vee \mathrm{GEN}$; the symbols $\wedge$, $\vee$, and $\neg$ stand for AND (bitwise product), OR (bitwise sum), and NOT (bitwise complement), respectively.

We see that F is closed under composition, because given $g = \langle \mathrm{KILL}_1,\mathrm{GEN}_1\rangle$, $f = \langle \mathrm{KILL}_2,\mathrm{GEN}_2\rangle$, then $fg = \langle \mathrm{KILL}',\mathrm{GEN}'\rangle$, where $\mathrm{KILL}' = [\mathrm{KILL}_1 \wedge \neg\mathrm{GEN}_2] \vee \mathrm{KILL}_2$, and $\mathrm{GEN}' = [\mathrm{GEN}_1 \wedge \neg\mathrm{KILL}_2] \vee \mathrm{GEN}_2$.

$\langle \mathbf{0},\mathbf{0}\rangle$ is the identity, and we leave distributivity for the reader to check.

Finally, given $x \in L$ we have $x = \langle \mathbf{0},x\rangle(\mathbf{0})$, so condition [F4] is satisfied.

THEOREM 5. *The global optimization problem CSE satisfies condition* $(*)$.

[8] The use of infix rather than postfix notation should not confuse the reader

```
        begin
          temp: element of $L$;
          $A$: array [1 .. $k$] of elements of $L$;
          $j$: integer; change: Boolean;
          $A[1] := \mathbf{0}$;
          for $j := 2$ until $k$ do $A[j] := \mathbf{1}$;
          change := true;
          while change do
            begin
              change := false;
Step 1        for $j := 2$ until $k$ do
                begin
                  temp := $\bigwedge_{q \in PRED(j)} f_q(A[q])$;
                  if temp $\neq A[j]$ then
                    begin
                      change := true;
                      $A[j]$ := temp
                    end
                end
            end
        end
```

FIG. 6 Program for Algorithm MK

PROOF. By Observation 6, it suffices to show that if the ith bit of the word $g(\mathbf{0}) \wedge f(\mathbf{1})$ is equal to 1, then the ith bit of the word $fg(\mathbf{0})$ is also equal to 1, where $g = \langle KILL_1, GEN_1 \rangle$ and $f = \langle KILL_2, GEN_2 \rangle$.

The ith bit of $g(\mathbf{0}) \wedge f(\mathbf{1})$ is equal to 1 iff (1) the ith bit of $GEN_1 = 1$, and (2) the ith bit of $KILL_2 = 0$. Now $fg = \langle KILL', GEN' \rangle$, where $GEN' = (GEN \wedge \neg KILL_2) \vee GEN_2$. Since 1 in a bit of GEN_2 implies 0 in the corresponding bit of $KILL_2$, by (1) and (2) either $GEN_1 \wedge \neg KILL_2$ has the ith bit equal to 1 or GEN_2 has the ith bit equal to 1. Hence the word $fg(\mathbf{0})$ has the ith bit equal to 1. □

7. *A Particular Case: When L Contains a* **1** *Element*

If L contains a **1** element, we can apply a slightly modified version of Algorithm K to achieve a better time bound.

ALGORITHM MK

Input: A particular instance $J = (G,M)$ of $D = (L,\wedge,F)$, where $G = (N,E,n_0)$ is a flow graph with k nodes and L contains a **1** element. Take N to be $\{1, 2, \cdots, k\}$ and assume the nodes are numbered by rPostorder. Execute the program of Figure 6.

Convention. We say that n iterations of Algorithm MK have been applied, where $n \geq 1$, if the loop beginning at step 1 has been applied n times.

LEMMA 4. *Given instance $J = (G,M)$ of data flow analysis framework $D = (L,\wedge,F)$, where L contains a one element* **1**, *and given an rPostorder for G, after the n-th iteration of Algorithm MK,*

$$A[j] = \Big(\bigwedge_{Q \in PATH^{(n)}(j)} f_Q(\mathbf{0})\Big) \wedge \Big(\bigwedge_{Q \in XPATH^{(n)}(j)} f_Q(\mathbf{1})\Big)$$

for $1 \leq j \leq k$, where $XPATH^{(n)}(j) = \{P \mid P$ is a path in G from an arbitrary node to node j, P contains n back edges, and the first edge in path P is a back edge$\}$. If $XPATH^{(n)}(j) = \varnothing$, we let

$$\bigwedge_{Q \in XPATH^{(n)}(j)} f_Q(\mathbf{1}) = \mathbf{1}.$$

PROOF. The proof is similar to that of Lemma 2, and we omit it. □

THEOREM 6. *Given a particular instance $J = (G,M)$ of framework $D = (L,\wedge,F)$ as* *input, where L contains a one element* **1**, *Algorithm MK will eventually halt. At the completion of Algorithm MK,*

$$A[j] = \bigwedge_{Q \in PATH(j)} f_Q(\mathbf{0}), \quad 1 \leq j \leq k.$$

PROOF. A direct consequence of Lemma 4 and Theorem 1. □

LEMMA 5. *Given $J = (G,M)$ of $D = (L,\wedge,F)$, where L contains a one element* **1**, *and given an rPostorder for G, Algorithm MK will halt after no more than n iterations iff for each node q in N and for each path P from node* 1 *to node q, there exist*

(1) *paths $P_1, \cdots, P_m$, each going from node* 1 *to node q and containing no more than $n - 2$ back edges, and*

(2) *paths $Q_1, \cdots, Q_l$, each of which is a path from an arbitrary node to node q, such that each path contains $n - 1$ back edges and the first edge in the path is a back edge satisfying the condition*

$$f_P(\mathbf{0}) = \bigwedge_{1 \leq i \leq m} f_{P_i}(\mathbf{0}) \wedge \bigwedge_{1 \leq i \leq l} f_{Q_i}(\mathbf{1}).$$

PROOF. (*if*). Let

$$x = \bigwedge_{Q \in \text{PATH}^{(n-1)}(j)} f_Q(\mathbf{0}) \wedge \bigwedge_{Q \in \text{XPATH}^{(n-1)}(j)} f_Q(\mathbf{1}).$$

We must show that

$$\bigwedge_{Q \in \text{PATH}^{(n)}(j)} f_Q(\mathbf{0}) \wedge \bigwedge_{Q \in \text{XPATH}^{(n)}(j)} f_Q(\mathbf{1}) = x.$$

By hypothesis, for every $Q \in \text{PATH}^{(n)}(j)$ we have $f_Q(\mathbf{0}) \geq x$. For Q in $\text{XPATH}^{(n)}(j)$, consider Q' formed by prefixing to Q a path from the initial node along the DFST on which the given rPostorder is based. Surely $f_Q(\mathbf{1}) \geq f_{Q'}(\mathbf{0})$. But $f_{Q'}(\mathbf{0}) \geq x$ is given.

(*only if*). This part is analogous to Lemma 3 and we omit it. □

THEOREM 7. *Let $D = (L,\wedge,F)$ be a data flow analysis framework, where L contains a one element* **1**. *Then Algorithm MK halts after at most $d(G) + 2$ iterations for every instance $J = (G,M)$ of D and every rPostorder definable for $G = (N,E,n_0)$, if and only if D satisfies condition* (**):

$$(\forall f, g \in F)(\forall x \in L)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(\mathbf{1})].^{9} \qquad (**)$$

Formally the theorem can be stated:

$$(\forall D)\{[\forall J = (G,M) \text{ an instance of } D)(\text{Algorithm MK halts after } d(G) + 2 \text{ iterations})] \equiv (**)\}.$$

PROOF. (*if*). By Lemma 5, it suffices to show that for each node $j \in N$ and each path $P \in \text{PATH}(j)$, there exist paths $P_1, \cdots, P_m$ in $\text{PATH}^{(d+1)}(j)$ and paths $Q_1, \cdots, Q_l$ in $\text{XPATH}^{(d+1)}(j)$, where $d = d(G)$, such that

$$f_P(\mathbf{0}) \geq \bigwedge_{1 \leq i \leq m} f_{P_i}(\mathbf{0}) \wedge \bigwedge_{i \leq i \leq l} f_{Q_i}(\mathbf{1}).$$

We want to prove the above by induction on k, the number of back edges contained in P.

Basis $(0 \leq k \leq d)$. This case is trivial; just let $m = 1$, $l = 0$, and $P_1 = P$.

Induction step $(k > d)$. Since P contains more than d back edges, the path $P = i_1, \cdots, i_r$ must contain a cycle. Let us pick the highest number a such that $i_a = i_b$ for some $b > a$.

Case 1 (The path $i_a, i_{a+1}, \cdots, i_{b-1}, ib, \cdots, i_r$ contains at most d back edges.) The proof goes exactly as Theorem 2.

Case 2(a) (The path $i_a, i_{a+1}, \cdots, i_{b-1}, i_b, \cdots, i_r$ contains $d + 1$ back edges and $b \neq r$.) Note that (i_a, i_{a+1}) must be a back edge, because $i_{a+1}, \cdots, i_r$ is cycle free.

[9] Note that (**) is equivalent to (*) when L has a unit, by Observation 6.

We let (note that both i_{a+1} and i_{b+1} are successors of i_a): $P_1 = i_1, \cdots, i_a, i_{b+1}$; $P_2 =$ a back-edge-free path from node 1 to node i_{a+1}; $P_3 = i_{a+1}, \cdots, i_{b+1}$; $P_4 = i_{b+1}, \cdots, i_r$; $x = f_{P_2}(\mathbf{0})$.

$$\begin{aligned} f_P(\mathbf{0}) &= f_{P_4} f_{P_3} f_{P_1}(\mathbf{0})^{10} \\ &\geq f_{P_4}(f_{P_1}(\mathbf{0}) \wedge f_{P_3}(\mathbf{1})) \qquad \text{(by assumption)} \\ &\geq f_{P_4}(f_{P_1}(\mathbf{0}) \wedge f_{P_3}(x)) = f_{P_4} f_{P_1}(\mathbf{0}) \wedge f_{P_4} f_{P_3} f_{P_2}(\mathbf{0}) = f_{P'}(\mathbf{0}) \wedge f_{P''}(\mathbf{0}), \end{aligned}$$

where $P' = i_1, \cdots, i_a, i_{b+1}, \cdots, i_r$ and $P'' = P_2, i_{a+2}, \cdots, i_b, i_{b+1}, \cdots, i_r\rangle$. P' and P'' are each seen to be from 1 to $j = i_r$ with fewer than k back edges. The induction follows in this case.

Case 2(b) (The path $i_a, i_{a+1}, \cdots, i_b, \cdots, i_r$ contains $d + 1$ back edges and $b = r$, i.e. $i_a = i_b = i_r = j$.) Again we see that (i_a, i_{a+1}) must be a back edge.

Let $P_1 = i_1, \cdots, i_a$, $P_2 = i_a, \cdots, i_b$. Then $f_P = f_{P_2} f_{P_1}(\mathbf{0}) \geq f_{P_1}(\mathbf{0}) \wedge f_{P_2}(\mathbf{1})$, by assumption. Since P_1 is a path in G with fewer than k back edges and P_2 is a path in G with $d + 1$ back edges, and the first edge in P_2 is a back edge, the induction follows.

(*only if*). This direction follows from Theorem 2. □

8. *Conclusions*

We have examined Kildall's lattice-theoretic formulation of global data flow analysis problems with an eye toward when depth-first ordering (rPostorder) yields an efficient iterative algorithm. The condition

$$(\forall f, g \in F)(\forall x \in L)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(x) \wedge x] \qquad (*)$$

for a data flow analysis framework $D = (L, \wedge, F)$ was shown necessary and sufficient for the depth-first version of Kildall's algorithm to converge after $d(G) + 3$ passes on an arbitrary instance (G, M) of D.

In the case where the semilattice has a one element, condition $(*)$ is equivalent to

$$(\forall f, g \in F)[fg(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(\mathbf{1})]. \qquad (**)$$

Kildall's algorithm, using depth-first ordering and taking advantage of the presence of the one element, works in $d(G) + 2$ passes iff condition $(**)$ is satisfied.

Combining the result of [5], which shows that the loop connectedness of a reducible flow graph never exceeds its interval depth, together with the empirical results of [11], which indicates that the interval depth averages 2.75, we may expect that five or six passes will be sufficient most of the time if the data flow analysis framework meets condition $(*)$ and depth-first ordering is used in Kildall's algorithm.

It was seen that the data flow analysis frameworks used by Kildall for constant propagation and common subexpression elimination do not meet condition $(*)$, while the usual bit vector frameworks of, e g. [1, 4, 5, 9, 12, 13], have a one element and meet condition $(**)$. Thus, while Kildall's methods enable us to detect certain instances of constant propagation or common subexpressions that are not detectable by the bit vector methods, it is possible that too high a price (in terms of computation time) must be paid for the extra information gathered by this framework.

ACKNOWLEDGMENT. The authors would like to thank both referees for their careful reading of the manuscript and for a number of improvements which they suggested.

REFERENCES

1. AHO, A V., AND ULLMAN, J.D *The Theory of Parsing, Translation and Compiling, Vol. II· Compiling*. Prentice-Hall, Englewood Cliffs, N J., 1973

[10] Note that f_{P_1} is the composition of the functions associated with nodes $i_1, \cdots, i_a$ but not i_{b+1}. Similarly, f_{P_3} does not include the effect of i_{b+1} Thus f_P is the composition of f_{P_1}, f_{P_3}, and f_{P_4}, even though P is not the concatenation of paths P_1, P_3, and P_4.

2. ALLEN, F.E. Program optımizatıon. *Annual Revıew ın Automatıc Programmıng, Vol. 5*, Pergamon Press, New York, 1969, 239-307.
3. ALLEN, F.E. Control flow analysıs *SIGPLAN Notices 5*, 7 (July 1970), 1-19.
4. COCKE, J Global common subexpression elımination *SIGPLAN Notıces 5*, 7 (July 1970), 20-24.
5. HECHT, M.S., AND ULLMAN, J.D. Analysis of a sımple algorithm for global flow problems. Proc. ACM Conf. on Principles of Programming Languages, Oct. 1973, pp. 207-217.
6 HECHT, M S., AND ULLMAN, J D Characterizations of reducıble flow graphs. *J. ACM 21*, 3 (July 1974), 367-375.
7. HOPCROFT, J.E , AND TARJAN, R.E. Algorıthm 447—Efficıent algorıthms for graph manıpulation. *Comm ACM 16*, 6 (June 1973), 372-378.
8. KAM, J.B., AND ULLMAN, J D Monotone data flow analysis frameworks TR-169, Dep. of Elec. Eng., Computer Sciences Lab., Princeton U., Prınceton, N J , Jan 1975
9 KENNEDY, K A global flow analysis algorithm *Int J Computer Math. 3*, 1 (Dec 1971), 5-15.
10 KILDALL, G A Global expression optımization during compilation. TR 72-06-02, Computer Scı Group, U. of Washıngton, Seattle, Wash , June 1972. See also Proc. ACM Conf. on Princıples of Programming Languages, Oct. 1973, pp 194-206
11. KNUTH, D.E. An empırıcal study of FORTRAN programs. *Software Pract and Exper. 1*, 2 (April 1971), 105-134
12. SCHAEFER, M. *A Mathematıcal Theory of Global Flow Analysis* Prentice-Hall, Englewood Clıffs, N J., 1973.
13. ULLMAN, J D Fast algorıthms for the elimination of common subexpressions *Acta Informatıca 2*, 3 (Dec. 1973), 191-213
14. VYSSOTSKY, V.A. Prıvate communication to M.S. Hecht, June 1973

RECEIVED MARCH 1974; REVISED JANUARY 1975