A FAST AND USUALLY LINEAR ALGORITHM FOR GLOBAL FLOW ANALYSIS†

(Extended Abstract)

by

Susan L. Graham and Mark Wegman
Computer Science Division
University of California
Berkeley, California 94720

Summary

A new algorithm for global flow analysis on reducible graphs is presented. The algorithm is shown to treat a very general class of function spaces. For a graph of e edges, the algorithm has a worst case time bound of $O(e \log_2 e)$ function operations. In programming terms, the number of operations is shown to be proportional to e + the number of exit nodes from program loops. Consequently a restriction to one-entry one-exit control structures guarantees linearity. It is shown that by relaxing these time bounds, a yet wider class of function spaces can be handled.

1. Introduction

In analyzing a computer program for purposes of code improvement, program verification, or error diagnosis, it is necessary to be able to trace (at compile-time) the flow of information through a program. In connection with code improvement techniques (such as, for example, common subexpression elimination or moving invariant computation out of loops), this analysis is called "Global Flow Analysis". Until recently the principal systematic technique for global flow analysis has been the interval analysis of Cocke and Allen [2,4]. The time needed to analyze the graphical representation of a program using this method is at worst proportional to the number of edges in the graph times the number of nodes. Kennedy [12] extended interval analysis to deal with a wider class of global flow problems than had previously been handled by this method.

Hecht and Ullman [8] have presented an iterative approach to global flow analysis in which the analysis can be carried out in time proportional to the number of edges in the graph times the maximum "depth" of the graph. (In the worst case the depth is proportional to the number of nodes in the graph.) Kildall [14] has proposed and implemented several extensions to the iterative method. The method is investigated further by Kam and Ullman in [11]. Comparisons of the iterative approach with interval analysis appear both in Hecht and Ullman [8] and in Kennedy [13].

In [20], Ullman presents a somewhat complicated algorithm for common sub-expression elimination which requires, at worst, time proportional to $e \log_2 e$ for a graph with e edges. Hecht and Ullman [7,9] also provide several useful characterizations of the class of graphs, termed "reducible flow graphs", on which interval analysis can be used.

Global flow analysis is also discussed in Aho and Ullman [1] and Schaefer [17].

In this paper, we present a new algorithm for global flow analysis which combines a modification of interval analysis with a modification of the transformations introduced by Hecht and Ullman in [7] to characterize reducible flow graphs. For a very general class of information flow problems the algorithm requires time at worst proportional to $e \log e$ for a flow graph with e edges. A different analysis of the algorithm reveals that the time is proportional to the number of edges plus the sum of the number of exits from program loops. Consequently the algorithm is linear for GOTO-free programs and very nearly linear for most "well-structured" programs.

The paper is organized as follows. In the next section, the basic definitions and results about program flow graphs are presented. In section 3 we introduce the notions of information propagation problems, fast functions and acceptable assignments. In section 4 we introduce transformations T_1', T_2', and T_3'. We investigate the use of these transformations in solving information propagation problems. Section 5 contains an analysis of the number of T_1', T_2', and T_3' transformations carried out by the algorithm on a flow graph. In section 6 we outline an efficient implementation of the algorithm which requires time proportional to the number of transformations. Section 7 contains further discussion of the method and possible extensions.

The present paper is in the form of an extended abstract in which the proofs are omitted or briefly sketched. A full presentation of these results will be submitted for publication in a journal and the results will be contained in the Ph.D. dissertation of the second author.

2. Basic Notions

A directed graph $G = (N,E)$ has a set of nodes N and a set of edges E, where $E \subseteq N \times N$. That is, for all e members of E, $e = (u,v)$ for some u, v members of N. In this paper we will assume that all graphs are directed. A path $p = p_0,p_1,\ldots,p_k$, $k \geq 0$ is a sequence of nodes such that for all i between 0 and k-1,

†Research sponsored by National Science Foundation Grant GJ-43318.

(p_i,p_{i+1}) is an edge. A path $p = p_0,p_1,\ldots,p_k$ is a path from p_0 to p_k of length k. For any i, j such that $0 \leq i < j \leq k$, p_i is a predecessor of p_j and p_j is a successor of p_i, relative to p. If $j = i+1$ then p_j is an immediate successor of p_i. A path $p = p_0,p_1,\ldots,p_k$ passes through a node u if for some i, $0 \leq i \leq k$, $p_i = u$. A path $p = p_0,p_1,\ldots,p_k$ passes through an edge $e = (u,v)$ if for some i, $0 \leq i < k$, $p_i = u$, $p_{i+1} = v$. An edge (u,v) leaves u and enters v.

A cycle is a path $c = p_0,p_1,\ldots,p_k$ where $p_0 = p_k$. A trivial cycle or a loop is a cycle whose length is 1. Thus, if c is a trivial cycle, $c = p_0, p_1$ and $p_0 = p_1$. An edge (p_0,p_1) where $p_0 = p_1$ is termed a looping edge. A graph with no cycles is acyclic.

A flow graph $G = (N,E,n_0)$ is a graph with a distinguished node n_0 in N such that for all v members of N there exists a path from n_0 to v.

Let $G = (N,E,n_0)$ be a flow graph. Node x dominates node y if and only if every path from n_0 to y passes through x.

Lemma. Dominance induces a partial ordering on the nodes of a flow graph.

An interval (I,h) of a flow graph $G = (N,E,n_0)$ is a maximal set of nodes I contained in N with a header node h in I such that for all edges $e = (u,v)$ in E, if $u \notin I$ and $v \in I$ then $v = h$ and furthermore all cycles with nodes only from I pass through h.

Theorem (Cocke and Allen). Every flow graph can be partitioned into a unique set of intervals in time proportional to the number of edges.

Let $G = (N,E,n_0)$ be a flow graph. Let N' be a set of nodes, each representing an interval of G and let E' be a set of edges between nodes of N' such that (x,y) is in E' if 1) x and y represent different intervals of G and 2) there exists an edge in E from a node in the interval represented by x to the header of the node represented by y. Let n_0' be the node representing the interval containing n_0. Then $G' = (N',E',n_0')$ is the derived graph of G.

A flow graph G is reducible if there exists an integer $k \geq 0$ and a sequence of graphs $G_0,G_1,\ldots,G_k$ such that $G = G_0$, G_k is the trivial graph, and for $0 \leq i < k$ G_{i+1} is the derived graph of G_i. A flow graph which is not reducible is irreducible. Figure 1 is an example of an irreducible flow graph. The flow graphs in the other figures in this paper are reducible.

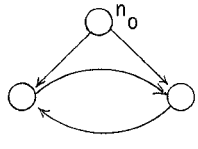


Figure 1. The paradigm irreducible flow graph

Let $G = (N,E,n_0)$ be a reducible flow graph. An edge (x,y) is a frond if y dominates x. A looping edge is a trivial frond.

The present work is based on the following two theorems of Hecht and Ullman on reducible flow graphs.

Theorem [Hecht & Ullman, 9]. A flow graph $G = (N,E,n_0)$ is reducible if and only if its edges can be partitioned into two sets S_1, S_2 such that $D = (N,S_1,n_0)$ is a directed acyclic flow graph, for any e in S_2, $D' = (N,S_1 \cup \{e\},n_0)$ is not a directed acyclic flow graph, and for every edge (x,y) of S_2, y dominates x.

Definition. Let $G = (N,E,n_0)$ be a flow graph.

If there exists a node v in N and $e = (v,v)$ is a looping edge in E then transformation $T1(G,e) \equiv (N,E-\{e\},n_0)$. Thus, $T1$ eliminates loops.

If there exists a node v in N and the only edge which enters v is $e = (u,v)$ then transformation $T2(G,e) \equiv (N-\{v\},\{(x,y) \mid x,y \in N\text{-}v$ and either $(x,y) \in E$ or $x = u$ and $(v,y) \in E\},n_0)$. Thus, transformation $T2$ eliminates v and replaces all edges leaving v by path-equivalent edges leaving u.

Figures 2 and 3 are examples of $T1$ and $T2$.

Theorem [Hecht & Ullman, 7]. Let $G = (N,E,n_0)$ be a flow graph. G is reducible if and only if there exists a sequence of flow graphs $G_0,G_1,\ldots,G_k$, $k \geq 0$ such that $G = G_0$, G_k is the trivial graph, and for $1 \leq i \leq k$, $G_{i+1} = T_j(G_i,e)$, for some $j \in \{1,2\}$ and $e \in E$ such that $T_j(G,e)$ is defined.

It can be shown that the sequence of flow graphs in the theorem is at least as long as the sequence of derived graphs for reduction by intervals.

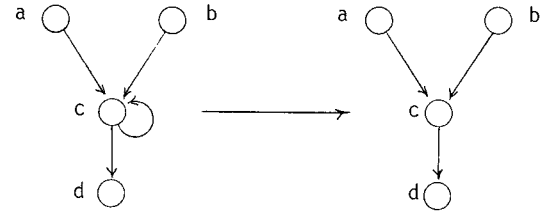


Figure 2. A T_1 transformation

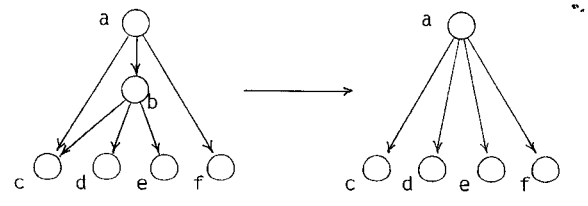


Figure 3. A T_2 transformation

Lemma. Let $G = (N,E,n_0)$ be a reducible flow graph. G contains a nontrivial cycle if and only if E contains a nontrivial frond.

3. Information Propagation Problems

In order that our global flow analysis techniques be most useful, we wish to demonstrate that they can be carried out efficiently for various program flow problems. Rather than presenting a separate algorithm for each type of code improvement, we characterize the class of problems handled by our algorithm, following, in spirit, the unifying approach of Kildall [14]. Evidence for the generality of this class is provided by Fong, Kam, and Ullman [6].

We cannot expect to obtain complete information for every kind of code improvement technique that might seem useful. For example, it is important to determine what arithmetic operations are unaffected by program flow and can therefore be determined at compile-time. Suppose there is a statement "A:=B+C;" in a program. If the value of B is 3 and the value of C is 5, every time this statement is executed then we would like to replace the statement with "A:=8;". However, since even determining whether a variable equals 1 or 0 is undecidable, we cannot detect all such instances.

Consequently, for each program analyzed, we restrict our attention to a finite set X of "facts" (for example, A = 5, C a power of 2), whose truth values at any point in the program may or may not be determinable. With each edge (u,v) in the program graph is associated a function which maps the subset of facts true at node u to the subset of facts true at node v if the control flow of the program leaves node u along edge (u,v). The task of our global flow analysis is then to associate with each node in the graph a subset of X which will always be true just before the node is executed at run-time. The association of sets of "facts" with nodes is termed an assignment.

Finding a maximal such assignment is at least as difficult as polynomial complete problems [3]. However we can find substantially faster algorithms if we restrict ourselves to assignments with the property that if X_1 is the set of facts for node u, X_2 is the set of facts for node v, and f is the function associated with edge (u,v), then $X_2 \subseteq f(X_1)$. We now make these notions more precise.†

We first define the sets of functions we consider.

Definition. Let X be a set. A function f mapping subsets of X into subsets of X is said to be monotonic if for all X_1, X_2 subsets of X such that X_1 is contained in X_2, then $f(X_1)$ is contained in $f(X_2)$.

The intersection $h = f \cap g$ of two functions f and g is defined for all subsets X_1 of X by $h(X_1) = f(X_1) \cap g(X_1)$.

The composition $h = f \circ g$ of two functions is defined for all subsets X_1 of X by $h(X_1) = f(g(X_1))$.

A set of functions F is an information propagation space if 1) F is closed under composition and intersection, 2) F is monotonic.

A set of functions F is fast if 1) F is an information propagation space, 2) for all f in F and all X_1 subsets of X, $f(X_1) \cap X_1 \subseteq f(f(X_1))$.

Lemma. The transitive closure of a set of monotonic functions under composition and intersection is an information propagation space.

As an example let $X = \{a_1,a_2,\ldots,a_n\}$. For $1 \leq i \leq n$ and any $X_1 \subseteq X$, let $f_i(X_1) = X_1 \cup \{a_i\}$, $g_i(X_1) = X_1 - \{a_i\}$. Let F be the transitive closure of the identity function and the f_i's and g_i's under composition and intersection. Since the f_i's and g_i's are monotonic, F is an information propagation space. All functions in F are of the form $f(X_1) = X_1 \cup X_2 - X_3$, where X_1, X_2, and X_3 are subsets of X. Thus, for any $X_1 \subseteq X$ $f(f(X_1)) = f(X_1)$ and F is fast. The functions in F include those using Kills and Gens from many global flow problems.

For many global flow problems of practical interest, each subset of X can be represented by a bit vector of length $|X|$. Function operations can then be implemented as Boolean word operations and are consequently very rapid.

Next we complete our specification of the class of problems we are considering. Given a flow graph $G = (N,E,n_0)$ and an information propagation space F, we will associate functions of F with edges in E by a function M from E to F. We use the notational convention that for any $e \in E$, $f_e = M(e)$. Let $p = p_0,p_1,\ldots,p_k$ be a path. We extend our notational convention to paths so that for $k > 0$ $f_p = f_{(p_{k-1},p_k)} \circ f_{(p_{k-2},p_{k-3})} \circ \cdots \circ f_{(0,1)}$. If p is the trivial path, then for any $X_1 \subseteq X$, $f_p(X_1) = X_1$.

An information propagation problem is a tuple $IP = (G,F,X,M)$ where $G = (N,E,n_0)$ is a flow graph, F is an information propagation space, X is a set (the domain of the functions in F), and M is a mapping $E \rightarrow F$.

Having stated the problem, we are now ready to define a class of solutions.

A fixed point for an information propagation problem $IP = (G,F,X,M)$ is a function $FP: N \rightarrow 2^X$ such that for all n in N, e = (m,n) in E, $FP(n) \subseteq f_e(FP(m))$ and $FP(n_0) = \phi$.†

A safe assignment to an information propagation problem $IP = (G,F,X,M)$ is a function $S: N \rightarrow 2^X$ such that for all paths $p = p_0,p_1,\ldots,p_k$ where $p_0 = n_0$ then $S(p_k) \subseteq f_p(\phi)$.

An acceptable assignment to an information propagation problem $IP = (G,F,X,M)$ is a function $AS: N \rightarrow 2^X$ such that AS is a safe assignment to IP and for all fixed points FP for IP and for all n in N, $FP(n) \subseteq AS(n)$.

---

†The reader familiar with the program verification techniques of Hoare [10] and Floyd [5] will find relationships with that work. We do not explore those issues here.

†The notation 2^X denotes the set of all subsets of X. Thus, if $S: N \rightarrow 2^X$, and $n \in N$ then S(n) is some subset of X.

Let K be a set of functions, such that if f is in K then f is a mapping $f: N \to 2^X$. We say f is maximal in K if for all f' in K either $f' = f$ or there exists n in N such that $f(n) \not\subseteq f'(n)$.

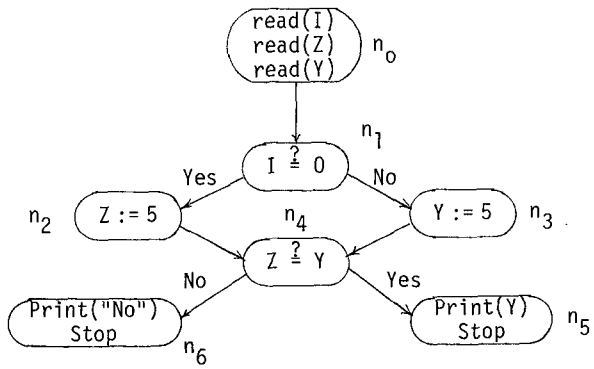
Intuitively, given a set of facts true just before a node is executed, a fixed point for that node yields a set of facts true upon entry to any successor of that node. A safe assignment associates with each node a set of facts that are provably true at that point in the program. An acceptable assignment is safe, and in addition, is at least as good as a maximal fixed point.

Example. Consider the information propagation problem of Figure 4, letting a_1 stand for "I = 0", a_2 for "Y = 5", and a_3 for "Z = 5". Let F be the transitive closure of the functions in the range of M under composition and intersection.

Let FP be the maximal fixed point for this example. Then $FP(n_0) = \phi$, $FP(n_1) = \phi$, $FP(n_2) = \{a_1\}$, $FP(n_3) = \phi$, $FP(n_4) = \phi$, $FP(n_5) = \phi$ and $FP(n_6) = \phi$.

Let S be a maximal safe assignment. Then $S(n_0) = \phi$, $S(n_1) = \phi$, $S(n_2) = \{a_1\}$, $S(n_3) = \phi$, $S(n_4) = \phi$, $S(n_5) = \{a_2, a_3\}$, $S(n_6) = \phi$.

Both FP and S are acceptable assignments. The function mapping each node to ϕ is a fixed point and a safe assignment, but it is not an acceptable assignment. Notice that the safe assignment given above is not a fixed point, since $S(n_5) \not\subseteq f_{(n_4,n_5)}(S(n_4))$. It is easily shown that every fixed point is a safe assignment.



$X = \{a_1, a_2, a_3\}$

For any $X_1 \subseteq X$, define M as

$$f_{(n_0,n_1)}(X_1) = X_1 - \{a_1\} - \{a_2\} - \{a_3\}$$
$$f_{(n_1,n_2)}(X_1) = X_1 \cup \{a_1\}$$
$$f_{(n_1,n_3)}(X_1) = X_1$$
$$f_{(n_2,n_4)}(X_1) = X_1 \cup \{a_3\}$$
$$f_{(n_3,n_4)}(X_1) = X_1 \cup \{a_2\}$$
$$f_{(n_4,n_5)}(X_1) = \begin{cases} X_1 \cup \{a_2\} & \text{if } a_3 \in X_1 \\ X_1 \cup \{a_3\} & \text{if } a_2 \in X_1 \\ X_1 & \text{otherwise} \end{cases}$$
$$f_{(n_4,n_6)}(X_1) = X_1$$

Figure 4. Flow graph and functional mapping of information propagation problem

In dealing with fast information propagation spaces, the fact that X is finite is never used and is therefore inessential in theory although it can lead to a more efficient implementation. Suppose we have an information propagation space all functions of which map finite sets to finite sets. Then even if the set X of "facts" is infinite, only a finite subset are necessary for any flow graph. Such a finite subset is easily found by the following technique. Construct an assignment B to the nodes by the following method: Set $B(n_0) = \phi$. While $B(v)$ is undefined for some v, if (u,v) is in E and $B(u)$ is defined, set $B(v) = f_{(u,v)}(B(u))$. The set $X' = \bigcup_{u \in N} B(u)$ is finite and if S is any safe assignment for $IP = (G,F,X,M)$ then for every $u \in N$, $S(u) \subseteq X'$. Thus for any u, S is a safe assignment for $IP' = (G,F,X',M)$ if and only if S is a safe assignment for IP.

4. The Transformations

In section 3 we introduced the notions of an information propagation problem and an acceptable assignment for such a problem. In this section and the next, we develop an algorithm for finding an acceptable assignment for an information propagation problem on any reducible flow graph.

In this section we describe three transformations, T_1^*, T_2^*, and T_3^*, on flow graphs. We show that given an information propagation problem on a flow graph, a graph transformed by T_1^*, T_2^*, or T_3^* has a corresponding information propagation problem. From an acceptable assignment for the transformed graph we can find an acceptable assignment for the original graph. If the information propagation space is fast then this process requires at most three functional operations (application of a function to an argument, composition of two functions, or intersection of two functions). If the space is not fast, up to $\log|X|$[†] compositions of functions can be required for a T_1^* transformation, where X is the set of "facts" of the information propagation problem.

It follows from these results that if we can reduce a flow graph to one node using these transformations, then from an acceptable assignment for the one-node graph we can find an acceptable assignment for the original graph in time proportional to the number of T_1^*, T_2^*, and T_3^* transformations needed. We show in section 5 that the number of such transformations is $O(|E|\log|E|)$ where $|E|$ is the number of edges in the original graph, and that the number of T_1^* transformations is at most $|E|$. Consequently we can find an acceptable assignment for an information propagation space with fast functions in $O(|E|\log|E|)$ functional operations and in $O(|E|\log|E| + |E|\log|X|)$ otherwise.

We now introduce the flow graph transformations.

Definition. Let $G = (N,E,n_0)$ be a flow graph. If for some v in N there exists an edge $e = (v,v)$ in E and there exists a unique u in $N - \{v\}$ such that (u,v) is in E then transformation $T_1^*(G,e) = (N,E-\{e\},n_0)$.

[†]For any set X, $|X|$ denotes the number of elements in X.

Thus T_1', when defined, has the same effect as transformation T_1 of Hecht and Ullman.

Definition. Let $G = (N,E,n_0)$ be a flow graph. If for some v in N there exists a unique u in $N-\{v\}$ such that (u,v) is in E and there exists any $e = (v,w)$ in E, where $v \neq w$, then transformation $T_2'(G,e) = (N'',E'',n_0)$ where, if v has no immediate successors other than w, then $N'' = N-\{v\}$, $E'' = E \cup \{(u,w)\} - \{(u,v),(v,w)\}$ and otherwise $N'' = N$, $E'' = E \cup \{(u,w)\} - \{(v,w)\}$.

Thus, (v,w) is removed and replaced by (u,w). If there are then no nontrivial paths from v, v is removed from N and (u,v) is removed from E. Given a node v entered by a unique edge (u,v), transformation T_2 of Hecht and Ullman connects u to all immediate successors of v and discards v and the edge entering v. In contrast, each application of transformation T_2' connects u to one of the immediate successors of v, discarding v and the edge entering v only when the last immediate successor is eliminated.

Definition. Let $G = (N,E,n_0)$ be a flow graph. G is a fan graph if every non-looping edge leaves n_0.

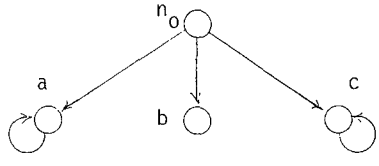


Figure 5. A fan graph

Definition. Let $G = (N,E,n_0)$ be a fan graph. If for some v in N, $e = (n_0,v)$ is the only edge entering v, then transformation $T_3'(G,e) = (N-\{v\},E-\{e\},n_0)$.

Since the node v has no immediate successors, transformation T_3', when defined, has the same effect as transformation T_2 of Hecht and Ullman.

We next relate these transformations to acceptable assignments to information propagation problems.

Lemma 4.1. Let $IP = (G,F,X,M)$ be an information propagation problem, where $G = (N,E,n_0)$ and F is fast. Let $G' = T_1'(G,e)$ be defined for some e in E. An information propagation problem $IP' = (G',F,X,M')$ can be found using only one composition of functions and one intersection of functions such that any acceptable assignment to IP' is an acceptable assignment to IP.

Sketch of Proof. For some v in N, let $e = (v,v)$ and let $e' = (u,v)$, $u \neq v$, be the only other edge entering v. Define M' such that $M'(e') = M(e') \cap M(e) \circ M(e')$ and for all a in $E-\{e,e'\}$, $M'(a) = M(a)$. Show that this satisfies the lemma. □

Notice that, unlike the previous lemma, the next two lemmas do not assume a fast information propagation space.

Lemma 4.2. Let $IP = (G,F,X,M)$ be an information propagation problem, where $G = (N,E,n_0)$. Let $G' = T_2'(G,e)$ be defined for some e in E. An information propagation problem $IP' = (G',F,X,M')$ can be found using at most one intersection of functions and one composition of functions such that by at most one functional application we can obtain an acceptable assignment to IP from an acceptable assignment to IP'.

Sketch of Proof. For some v in N, let $e' = (u,v)$, $u \neq v$, be the unique edge in E which enters v and let $e = (v,w)$, $v \neq w$ be an edge in E which leaves v. Let $e'' = (u,w)$ and let $G' = (N',E')$. Define M' such that

$$M'(e'') = \begin{cases} M(e'') \cap M(e) \circ M(e') & \text{if } e'' \in E \\ M(e) \circ M(e') & \text{otherwise} \end{cases}$$

and for all a in $E'-\{e''\}$, $M'(a) = M(a)$. If $e' \notin E'$, then for any acceptable assignment AS' to IP' let $AS(v) = f_{(u,v)}(AS'(u))$. Show that this satisfies the lemma. □

Corollary 4.3. Let IP and IP' be as in Lemma 4.2. An intersection of functions is needed to find IP' only if $|E''| < |E|$. A functional application is needed to obtain an acceptable assignment to IP from an acceptable assignment to IP' only if $|N'| < |N|$.

Proof. Follows from construction in proof of Lemma 4.2.

Lemma 4.4. Let $IP = (G,F,X,M)$ be an information propagation problem, where $G = (N,E,n_0)$ is a fan graph. Let $G' = T_3'(G,e)$ be defined for some e in E. An information propagation problem $IP' = (G',F,X,M')$ can be found using no function operations. By one functional application we can obtain an acceptable assignment to IP from an acceptable assignment to IP'.

Sketch of Proof. For some v in $N-\{n_0\}$, let (n_0,v) be the unique edge in E which enters v. For all e' in $E-\{e\}$, let $M'(e') = M(e')$. For any acceptable assignment AS' to IP' let $AS(v) = f_{(n_0,v)}(AS'(n_0))$. Show that this satisfies the lemma. □

By combining these three lemmas and the corollary, we get

Theorem 4.1. Let $IP = (G,F,X,M)$ be an information propagation problem, where $G = (N,E,n_0)$ and F is fast. Let T be the number of T_1' and T_2' transformations needed to reduce G to a graph with the single node n_0. Then we can find an acceptable assignment to IP in $|N|$ applications of functions, at most $|E|$ intersections of functions, and T compositions of functions.

Thus for any information propagation problem $IP = (G,F,X,M)$ where F is fast it remains only to analyze the number of T_1', T_2', and T_3' transformations necessary to reduce G in order to know how many function operations are needed to find an

acceptable assignment. We do this analysis in the next section.

In order to find the number of function operations needed to find an acceptable assignment if F is not fast, it suffices, by Lemma 4.2, to examine the number of operations required for each T_1' transformation and the number of T_1' transformations needed to reduce G. The latter issue is resolved in section 5. We answer the former by the following lemma.

Lemma 4.5. Let IP = (G,F,X,M) be an information propagation problem, where $G = (N,E,n_0)$ and X is a finite set. Let $G' = T_1'(G,e)$ be defined for some e in E. An information propagation problem IP' = (G',F,X,M') can be found using only $\log_2|X|+1$ compositions of functions and two intersections of functions such that any acceptable assignment to IP' is an acceptable assignment to IP.

Sketch of Proof. Let $M'(e') = M(e') \cap f_e^{\log|X|} \circ M(e')$, where for any $X_1 \subseteq X$, $f_e'(X_1) = f_e(X_1) \cap X_1$ and $f_e^{\log|X|}$ denotes the $|X|$-term composition $f_e' \circ f_e' \circ \cdots \circ f_e'$. □

5. Reduction of Flow Graphs

In this section we analyze the number of T_1', T_2', and T_3' transformations necessary to reduce a flow graph to a graph with one node, if such reduction is possible. This result combined with the results from the previous section give us the number of function operations needed for global flow analysis.

The study of the number of transformations proceeds in several stages. We first exhibit an algorithm for reducing a reducible flow graph using these transformations. We then prove the correctness of the algorithm, at the same time proving certain characteristics of its behavior. We then give two analyses of the number of transformations carried out by the algorithm. The first analysis shows that the number of transformations is at worst $O(|E|\log_2|E|)$ where E is the set of edges of the original graph. The second analysis, while being cruder than the first since it yields an $O(|N||E|)$ worst case, reveals that the algorithm is linear or nearly linear on the graphs for most programs.

The algorithm, which we refer to as Algorithm A, is written in a higher-level Algol-like language. In section 6, we show that the algorithm can be refined in such a way that the time taken to find an appropriate sequence of transformations is proportional to the number of transformations.

The heart of the algorithm is a succession of calls on a procedure Reduceset. At every call from label B of the program, Reduceset is passed a set S of nodes similar to an interval and a "header" h. The set S differs from an interval in that 1) a node in the set may have a looping edge and 2) there is a path from every node to the header (hence the graph is strongly connected). Reduceset eliminates all nontrivial cycles passing through nodes other than the header. At the final call of Reduceset at label C the entire graph, now acyclic except possibly for a looping edge through n_0 is passed to Reduceset and reduced to a fan graph. The final while loop reduces the graph to one node.

Within Reduceset, applicable transformations on the edges connecting the nodes of S can be made in arbitrary order. In fact, one need not even follow a T_1' transformation by a T_2' transformation as the algorithm indicates. (We have written the algorithm this way only to aid the exposition.)

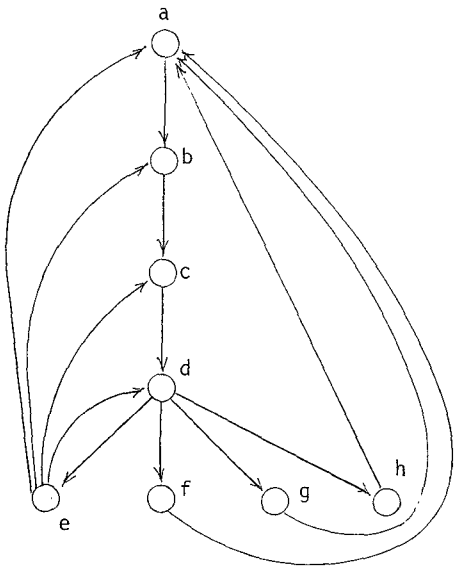
We next state the algorithm. Notice that Reduceset changes only edges between nodes in S. However procedure T_2' must inspect other edges of the graph in order to determine whether to delete a node.

Algorithm A

```
Procedure T1' (E: set of edges; v: node of looping
  edge);
     begin E:=E-{(v,v)}
     end;

Procedure T2' (N: set of nodes; E: set of edges;
  h,v,w: nodes of edges (h,v) and (v,w) in E);
     begin
          E:=E∪{(h,w)}-{(v,w)};
          if v has no immediate successor in
            G = (N,E) then
               begin N:=N-{v}; E:=E-{(h,v)}
               end;
     end;
Procedure Reduceset (S: set of nodes; h: node in S)
     begin
          while there exists an edge (v,w) in E
            with v,w ε S, v ≠ w
               such that if (u,v) ε E then u = h
                 or u = v
               do
          begin
               choose any such (v,w);
               if (v,v) ε E then T1'(E,v);
               T2'(N,E,h,v,w)
          end;
     end;

Procedure T3' (N: set of nodes; E: set of edges;
  n0,v: nodes of edge (n0,v));
     begin
          E:=E-{(n0,v)};
          N:=N-{v}
     end;


     begin comment: Main Program;
          while G contains a nontrivial frond do
            begin
               T:={u|(v,u) is a nontrivial frond};
               h:=a node in T not dominated by any
                 other node in T;
               S:={v∈N|h dominates v and there is a
                 path p from v to h such that all
                 nodes on p are dominated by h};
B:             Reduceset(S,h)
            end;

C:        Reduceset(N,n0);
          while N-{n0} is nonempty do
            begin
               choose any v in N-{n0};
               if (v,v) ε E then T1'(E,v);
               T3'(N,E,n0,v)
            end;
     end.
```

In the following example each step is a call to Reduceset.

Example.

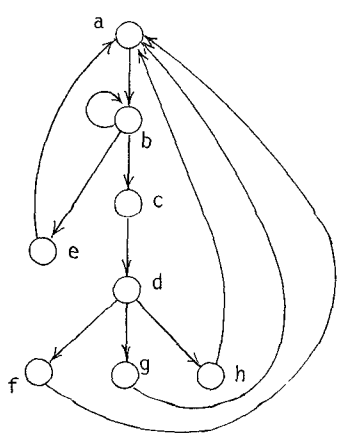


Step 1 T = {a,b,c,d}
Reduceset is called with S = {d,e}, h = d
T_2^1(N,E,d,e,d) replaces edge (e,d) by edge (d,d)

Step 2 T = {a,b,c}
Reduceset is called with S = {c,d,e}, h = c
T_1^1(E,d) deletes edge (d,d)
T_2^1(N,E,c,d,e) replaces edge (d,e) by edge (c,e)
T_2^1(N,E,c,e,c) replaces edge (e,c) by edge (c,c)

Step 3 T = {a,b}
Reduceset is called with S = {b,c,d,e}, h = b
T_1^1(E,c) deletes edge (c,c)
T_2^1(N,E,b,c,e) replaces edge (c,e) by edge (b,e)
T_2^1(N,E,b,e,b) replaces edge (e,b) by edge (b,b)

The graph now looks like



Step 4 T = {a}
Reduceset is called with S = {a,b,c,d,e,f,g,h}, h = a
T_1^1(E,b) deletes edge (b,b)
T_2^1(N,E,a,b,e) replaces edge (b,e) by edge (a,e)
T_2^1(N,E,a,e,a) replaces edge (e,a) by edge (a,a), deleting edge (b,e) and node e
T_2^1(N,E,a,b,c) replaces edge (b,c) by edge (a,c), deleting edge (a,b) and node b
T_2^1(N,E,a,c,d) replaces edge (c,d) by edge (a,d), deleting edge (a,c) and node c
T_2^1(N,E,a,d,f) replaces edge (d,f) by edge (a,f)
T_2^1(N,E,a,d,g) replaces edge (d,g) by edge (a,g)
T_2^1(N,E,a,d,h) replaces edge (d,h) by edge (a,h), deleting edge (a,d) and node d
T_2^1(N,E,a,f,a) replaces edge (f,a) by edge (a,a), deleting edge (a,f) and node f
T_2^1(N,E,a,g,a) deletes edges (g,a) and (a,g) and node g
T_2^1(N,E,a,h,a) deletes edges (h,a) and (a,h) and node h

The final graph is

The next few lemmas are used in proving Theorem 5.1 and will serve as an outline of that proof.

It follows from the second theorem of Hecht and Ullman quoted in section 2 that T_1^1 and T_3^1 transform a reducible flow graph to a reducible flow graph. In Lemma 5.1 we establish the same result for T_2^1.

Lemma 5.1. Let $G = (N,E,n_0)$ be a reducible flow graph such that, for some u, v, w in N, $u \neq v$, $v \neq w$, E contains edges $e = (v,w)$ and $e' = (u,v)$, where (u,v) is the only edge which enters v. Then $G' = T_2^1(G,e)$ is a reducible flow graph.

Proof. Omitted. □

The next lemma is used in proving termination of the algorithm.

Lemma 5.2. Let $G = (N,E)$ be a directed graph. If every node of G is entered by some edge which is not a looping edge, then G contains a non-trivial cycle.

Proof. Induction on $|N|$.

Corollary 5.3. Let $G = (N,E,n_0)$ be a flow graph with no non-trivial cycles. Then either $N = \{n_0\}$ or there exists $x \in N$, $x \neq n_0$, such that (n_0,x) and possibly (x,x) are the only edges entering x.

Next we establish the properties of the arguments of Reduceset.

Lemma 5.4. Let $G = (N,E,n_0)$ be a reducible flow graph with a nonempty set $T = \{u \in N |$ for some $v \in N$, (v,u) is a nontrivial frond of $G\}$. Let h be any node in T not dominated by another node in T. Define $S = \{v \in N |$ h dominates v and there exists a path p from v to h such that all nodes of p are dominated by $h\}$. Let $E_S = \{(x,y) \in E | x,y \in S\}$. Then

1) There are no nontrivial fronds in E which enter nodes in S other than h.

2) There is a path in E from h to every node in S.

3) $G_S = (S,E_S,h)$ is a reducible flow graph.†

Proof. Omitted. □

The following lemma shows that procedures T_1' and T_2' correspond to transformations T_1' and T_2' and that Reduceset has the desired effect on a flow graph.

Lemma 5.5 (Correctness of Reduceset). Let $G = (N,E,n_0)$ and $G_S = (S,E_S,h)$ be reducible flow graphs such that $S \subseteq N$, $E_S = \{(x,y) \in E | x,y \in S\}$ and no nontrivial fronds in E enter nodes in S other than h. After Reduceset(S,h) is carried out, let N' be the resulting set of nodes and E' be the resulting set of edges. Let E_S' be the set of edges (x,y) in E' such that x and y are in S. Then

1) Every execution of procedures T_1' and T_2' satisfies the conditions for transformations T_1' and T_2'.

2) T_1' and T_2' are called exactly as many times as the number of edges (x,y) in E_S, such that $x \neq h$. (Consequently Reduceset always terminates.)

3) After completion of Reduceset(S,h), $G' = (N',E',n_0)$ is a reducible flow graph, all edges in E_S' leave h, E' contains no new nontrivial fronds, and $|E'| \leq |E|$.

Proof. Omitted. □

As a consequence of Lemma 5.5, Algorithm A could be rewritten so that set T is computed only once, at the beginning of execution, and after each call of Reduceset, h is removed from T.

Having established that Reduceset is correct, the correctness of Algorithm A easily follows.

Theorem 5.1. Algorithm A terminates and reduces any reducible flow graph $G = (N,E,n_0)$ to a graph with the single node n_0.

Proof. Omitted. □

Now we are ready to analyze the number of T_1', T_2', and T_3' transformations necessary to reduce a reducible flow graph to a single node. We will obtain our bound by choosing a particular ordering on the edges in executing Reduceset and then showing that that ordering is inessential. To carry out this analysis, we must introduce a few more concepts.

Definitions. A flow graph $G = (N,E,n_0)$ is a tree rooted at n_0 if for every node x in $N - \{n_0\}$ there is exactly one node in E which enters x. If $G = (N,E,n_0)$ is a tree rooted at n_0 and $G_1 = (N_1,E_1,n_1)$ is a tree rooted at n_1 such that $N_1 \subseteq N$ and $E_1 \subseteq E$, then G_1 is a subtree of G.

Let $G = (N,E,n_0)$ be a flow graph. A spanning tree (for G) is a flow graph $G' = (N,E',n_0)$, $E' \subseteq E$ such that G' is a tree rooted at n_0. A cross-link of G is an edge (x,y) of E such that x does not dominate y and y does not dominate x. The definition of a frond outside the domain of reducible graphs is dependent on a particular spanning tree of a graph. A frond is an edge (x,y) such that y dominates x in the given spanning tree. A reverse frond is an edge (x,y) such that x dominates y. If a graph is reducible then the fronds are the same no matter which spanning tree is used.

Let x, y be nodes of a tree such that x dominates y. Let $u_0,u_1,u_2,\ldots,u_k$ be the path from x to y in the tree where $x = u_0$, $y = u_k$. (There can be only one such path.) The transformation cfind(x,y) replaces edges (u_1,u_2), $(u_2,u_3),\ldots,(u_{k-1},u_k)$ by edges $(x,u_2),(x,u_3),\ldots,(x,u_k)$, thereby transforming the tree to another tree. The cost of cfind(x,y) is $k-1$, which is the number of edges changed.

Paterson [15] states a less general theorem but his proof supports the following:

Theorem (Paterson's Theorem). If a tree has less than or equal to e edges and less than or equal to e c-finds are performed then the sum of the costs of the c-finds is no more than $O(e \log e)$.

We will show that Algorithm A is equivalent to performing no more than e c-finds on a spanning tree of a flow graph $G = (N,E,n_0)$.

We will obtain a bound on the number of T_1' and T_2' transformations, by appealing to Paterson's theorem. In order to invoke Paterson's theorem, which is about trees, we will show how Algorithm A transforms a spanning tree of a flow graph as it transforms the graph. For that purpose we need the following two lemmas.

Lemma 5.6. Let $G = (N,E,n_0)$ and $G_S = (S,E_S,h)$ be reducible flow graphs such that $S \subseteq N$, $E_S = \{(x,y) \in E | x,y \in S\}$, no nontrivial frond of E enters a node of S other than h, and there is a path in G_S from every node to h. Then any spanning tree for G contains a subtree rooted at h which is a spanning tree for G_S.

Proof. Omitted. □

Lemma 5.7. Let $G = (N,E,n_0)$ and $G_S = (S,E_S,h)$ be reducible flow graphs such that $S \subseteq N$, $E_S = \{(x,y) \in E | x,y \in S\}$, no nontrivial frond of E enters a node of S other than h, G_S contains a path from every node in S to h, and G_S contains at least one nontrivial cycle. Let $G' = (N,E',n_0)$ be a spanning tree for G. Then there is some node x in $S - \{h\}$ such that

†G_S is a region in the sense of [9].

1) E contains a cross-link or a non-trivial frond which leaves x and enters a node in S.

2) There is a path $u_0,u_1,u_2,\ldots,u_k$ in G' where $h = u_0$, $x = u_k$, and for $1 \leq i \leq k$, $u_i \in S$.

3) For $0 < i \leq k$, no cross-link or non-trivial frond in E enters u_i.

Proof. Omitted. □

We next define a sequence of T_1' and T_2' transformations called a collapse. It will turn out that a collapse on a flow graph is equivalent to a cfind on a spanning tree of the graph. We then show that we can reduce any reducible flow graph to a fan graph by a sequence of collapses.

Definition. Let $G = (N,E,n_0)$ be a flow graph and let $G' = (N,E',n_0)$ be any spanning tree for G. Let x, y be nodes in N, $x \neq y$, such that either there exists a path in G' from x to y or, for some z in N, there exists a path in G' from x to z and an edge (z,y) in E. In either case let $u_0,u_1,u_2,\ldots,u_k$ be the specified path from x to y, where $x = u_0$ and $y = u_k$. The transformation collapse(x,y) is a sequence of applications of T_2' and, if necessary T_1', to the edges $(u_1,u_2),(u_2,u_3),\ldots,(u_{k-1},y)$ which replaces them respectively by $(x,u_2),(x,u_3),\ldots,(x,y)$. It is defined only if the sequence of applications of T_1' and T_2' is defined.

Lemma 5.8. Let G, G_S, and G' be defined as in Lemma 5.7. Then there exists a sequence of choices of nodes and edges in the while loop of Reduceset which corresponds to a sequence of collapses such that each collapse eliminates a nontrivial frond or a cross-link of G.

Proof. Omitted. □

Lemma 5.9. Let $G = (N,E,n_0)$ be a reducible flow graph. G can be reduced to a graph with no nontrivial fronds in $|E|$ collapses.

Proof. Follows from Lemmas 5.5 and 5.8. □

Since collapses on flow graphs correspond to cfinds on their spanning trees we get

Theorem 5.2. Let $G = (N,E,n_0)$ be a reducible flow graph. The number of T_1', T_2', and T_3' transformations carried out by Algorithm A to reduce G to a flow graph with the single node n_0 is no more than $|N|$, $O(|E|\log_2|E|)$, and $|N|$ respectively.

Sketch of Proof. It follows from Lemma 5.9, Paterson's theorem, and showing the equivalence of numbers of collapses and numbers of cfinds that all nontrivial cycles can be eliminated in $O(|E|\log_2|E|)$ T_2' transformations. It follows from Lemma 5.5 that this bound holds independent of the order in which edges are chosen within Reduceset. It can be shown that a graph with no nontrivial cycles can be reduced in $O(|E|)$ T_2' transformations and at most $|N|$ T_3' transformations. A total of at most $|N|$ T_1' transformations is needed. □

Combining this with the results of section 4 we get

Theorem 5.3. Let IP = (G,F,X,M) be an information propagation problem, where $G = (N,E,n_0)$. If F is fast, then we can find an acceptable assignment to IP in $|N|$ applications of functions, at most $|E|$ intersections of functions, and $O(|E|\log_2|E|)$ compositions of functions. If F is not fast, then we can find an acceptable assignment to IP in $|N|$ applications, at most $|E| + |N|$ intersections of functions, and $O(|E|\log_2|E| + |N|\log_2|X|)$ compositions.

Proof. Follows from Theorem 4.1, Lemma 4.5 and Theorem 5.2. □

We can expect that a typical collection of program flow graphs will tend to exhibit a more special set of characteristics than the total set of reducible flow graphs we have just studied, particularly in view of the current ideas about well-structured programs. Consequently we again analyze the number of transformations of Algorithm A, this time with respect to programming language issues.

We define two characterizations of iteration loops in programs. We then show that the number of T_1', T_2', and T_3' transformations needed to reduce a reducible flow graph is no more than $O(|E|$ + the number of exits from program loops). Finally, we discuss the implications of this result.

Definition. Let $G = (N,E,n_0)$ be a reducible flow graph. For each n in N, p-loop(n) = {v ∈ N| n dominates v and there is a nontrivial path p from v to n containing only nodes dominated by n}. c-loop(n) = {v ∈ N| n dominates v and there is a nontrivial path p from v to n containing only nodes dominated by n and passing through only one frond}. c-number(n) is the number of nodes in c-loop(n) which are left by edges which enter nodes not in p-loop(n) and e-number(n) is the number of nodes n' such that $n \in$ c-loop(n') and n is left by an edge which enters a node not in p-loop(n').

Intuitively if non-empty, p-loop(n) corresponds to a program loop starting from the statement represented by n. Clearly for any node n, c-loop(n) ⊆ p-loop(n). If non-empty, c-loop(n) corresponds to a program loop without its inner loops that do not pass through n. For any node n, c-number(n) is the number of exit nodes from the program loop dominated by n and e-number(n) is the number of program loops exited from node n.

Figure 6 contains an example illustrating these concepts.

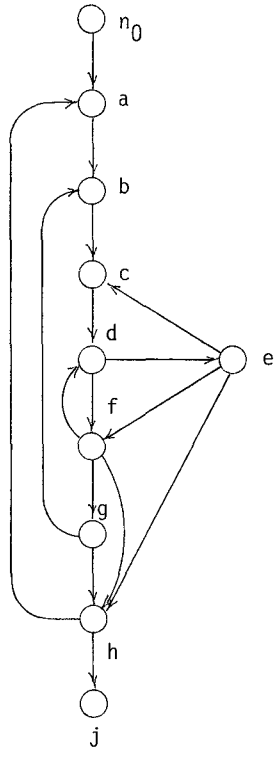
We obtain the following consequences from these definitions.

Lemma 5.10. Let $G = (N,E,n_0)$ be a reducible flow graph. Then $\sum_{n\in N}$ c-number(n) = $\sum_{n\in N}$ e-number(n).

Proof. Follows easily from definitions. □

Lemma 5.11. Let $G = (N,E,n_0)$ be a reducible flow graph. The number of T_2' transformations carried out by Algorithm A to reduce G is at most $|E| + \sum_{n\in N}$ c-number(n).

Sketch of Proof. Induction on $\sum_{n\in N}$ c-number(n). □



	p-loop	c-loop	c-number	e-number
n_o	--	--	0	0
a	abcdefgh	abcdefgh	1	0
b	bcdefg	bcdefg	3	0
c	cdef	cde	1	0
d	def	def	2	0
e	--	--	0	3
f	--	--	0	2
g	--	--	0	1
h	--	--	0	1
j	--	--	0	0

Figure 6. Example

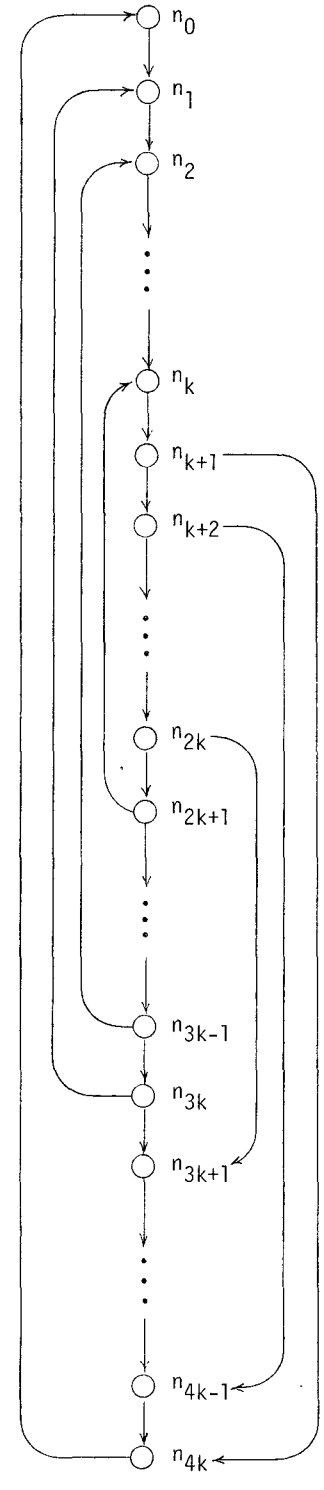


Figure 7. Example for Lemma 5.11

Since we have shown in Theorem 5.3 that it is only the number of T_2^1 transformations that can be non-linear in the size of the flow-graph, Lemma 5.11 provides a convenient measure of the time needed for global flow analysis for particular classes of programs, even though, as Figure 7 illustrates, the measure is somewhat coarse.

Consider the flow graph shown in Figure 7. If $|N|$ is the number of nodes and $|E|$ is the number of edges, then $|E| < 2|N|$. However since there are approximately $|N|/4$ exit nodes, each from approximately $|N|/4$ program loops, it follows from Lemma 5.11 that the number of T_2^1 transformations needed by Algorithm A is no more than $O(|N|^2)$. However, it is easily seen that Algorithm A performs only $O(|E|)$ transformations on this graph.

Using notions of graph grammars similar to those found in [7], we can analyze the forms of program graphs obtained from various combinations of programming language control structures.

For example suppose we consider a programming language containing assignment statements, possibly-nested conditional statements, while statements, repeat-until statements, case statements and for statements. (This is essentially what is found in PASCAL [22], omitting goto statements and procedures.) It can be rather easily shown that for such a language 1) the number of edges in a program flow graph is proportional to the number of nodes, 2) every program flow graph is reducible, and 3) there is (at most) one exit node from every program loop. Consequently the time required for global flow analysis is proportional to the number of nodes (i.e. roughly to the size of the program).

If we add a halt statement, or any statement which effectively causes an exit from the entire program, such a statement appears in a program flow graph as a node with no successors. Consequently, it turns out that such nodes cannot affect the non-linearity of Algorithm A. It is only when a programming language includes an unrestricted go-to, a labelled exit as is found in Bliss [23], or some equivalent facility to make multiple jumps out of nested loops that the possibility of non-linearity occurs. Even for such a language, we can probably expect, in practice, that the number of such jumps be reasonably small relative to the size of the program. Furthermore, their effect on the complexity of the algorithm is additive rather than multiplicative.

6. Implementation

We have shown that the number of transformations to reduce a reducible flow graph is $O(|E|\log_2|E|)$ where $|E|$ is the number of edges in the graph and that consequently the number of function operations for many global flow problems also has this bound. We have also argued informally that for actual programs the number of operations is approximately linear in the size of the program (i.e. the number of nodes). In order to argue that the method is indeed of practical interest, we now indicate how one might implement the algorithm so that the time required on a random access machine for finding the transformations and doing other "bookkeeping" is on the order of the number of transformations. Figure 8 summarizes the data structures used in the implementation.

We will represent the flow graph $G = (N,E,n_0)$ by an adjacency structure; namely, with each node x we will associate an unordered adjacency list of the nodes y such that (x,y) is an edge of the graph. The nodes are numbered from 1 to $|N|$ and can be addressed directly.

The key to an efficient implementation is the following notion.

Definition. An s-numbering of the nodes of a graph is an assignment of integers to the nodes such that for any non-looping edge (x,y), if (x,y) is a frond, then s-number(y) < s-number(x) and, if not, then s-number(x) < s-number(y).

Tarjan [18] gives an algorithm which produces an s-numbering of the nodes in a graph in time proportional to the number of edges. He also shows the usefulness of adjacency structures and s-numbering for a variety of graph algorithms. An s-numbering has the property that for any nodes x, y, if $x \neq y$ and x dominates y, then s-number(x) < s-number(y).

The first step in the implementation is to produce an s-numbering of the nodes. We list the nodes in their s-number ordering. The ordered list can be constructed in linear time by a radix sort, creating an array of pointers to the nodes and adjacency lists, indexed by s-numbers.

Next we create a reverse adjacency structure in which each node x points to two lists. The first is a list of nodes y such that (y,x) is an edge which is a frond; the second is a list of nodes y such that (y,x) is an edge which is not a frond. Thus for each node x, the lists to which x points indicate the edges entering x. Since a frond can be determined by the fact that it goes from a node with a higher s-number to one with a lower s-number, the reverse adjacency structure can be created in linear time.

The next step is to find the set S and node h which are arguments to Reduceset. We scan the list of nodes in decreasing s-number order. We let h be the first node encountered which is entered by a nontrivial frond. This h cannot dominate any node with an entering frond, since such a node would have to have a higher s-number.

Having found h, we find the subgraph $G_S = (S,E_S,h)$ by what is essentially a depth-first search. Recall that for any frond (x,h), x must be in S since by definition of frond, h dominates x, and the edge (x,h) provides a path from x to h. Also, for any node x in $S - \{h\}$, any node entering x leaves a node in S. We mark each node as its membership in S is discovered. Beginning with h and using the reverse adjacency structure, after adding a node x to S, we follow one of the edges entering x. If that edge leaves a node not already in S we add the node to S and follow one of its edges, otherwise we backtrack and follow another edge from a previously encountered node. The number of steps in this process is the number of edges in the resulting E_S, since each edge is traversed once. Additionally, an "edge-from-h" bit is associated with each node, which is set if an edge from h to that node is traversed. Also, as edges are traversed, construct

an adjacency structure for the subgraph G_S.

Next, in time proportional to the number of edges in E_S, we do an s-numbering on the nodes of S. As before, we list the nodes of S in order of the new s-numbering. This new list has the same order as the old list but has no "gaps" caused by nodes not in S. (The purpose of the new s-numbering is only to obtain this list. The new s-numbering can then be discarded. This method of obtaining the list is in general theoretically faster than obtaining the sublist of nodes by scanning the original list starting at h.)

Now we are ready to carry out the computation of Reduceset, by again exploiting the properties of s-numbering. Since the graph has no fronds entering nodes of $S-\{h\}$, all entering edges of nodes in $S-\{h\}$ must leave nodes with smaller s-numbers. The node h has the smallest s-number of any node in S. Consequently the node following h in the new list has entering edges only from h and possibly itself. If the node has a looping edge, apply T_1'. Then, using the adjacency structure for the subgraph, apply T_2' to all edges leaving that node. At the completion of the T_2' transformations, the third node of S in the s-number ordering will have entering edges only from h and possibly itself. Repeat until all nodes in S have been processed.

Application of T_1' and T_2' requires changing edges in the original adjacency structure and the reverse adjacency structure. By keeping a pointer from each entry in the adjacency structure for G_S to the corresponding entry in the original adjacency structure, edges are easily deleted. By checking the edge-from-h bit before adding an edge, duplicate entries for edges leaving h can be avoided. If the edge is added, the appropriate edge-from-h bit is set. Since the reverse adjacency structure is not needed during the Reduceset computation, when a node in $S-\{h\}$ is processed, its non-frond reverse adjacency list can be replaced immediately by the single entry for h. The frond reverse adjacency list for h can be replaced by the single entry for the looping edge to h. The adjacency structure for G_S need not be updated.

Application of T_1' or T_2' does not change the s-numbering of the nodes. Since finding each subgraph G_S is done in time proportional to the number of edges in E_S, it can be shown that the time for finding all such subgraphs is proportional to the number of transformations necessary to remove all non-trivial cycles. Once the nontrivial cycles have been eliminated, the rest of the computation can be done in a straightforward way by processing the remaining nodes in increasing s-number order. The time for the total computation is proportional to the total number of transformations.

The implementation just described illustrates that our global flow analysis algorithm can be realized on a random access machine within the theoretical time bound. However, we make no claims that what we have described is in any sense the best implementation. In particular one is likely, in practice, to put an upper bound on the size of a graph to be analyzed. In light of such a bound, and the empirical characteristics of flow graphs for actual programs, other realizations might well be more efficient. In addition, we have made no attempt to save space, instead creating new doubly-linked lists at will.

Data structures include:

An array containing a record for each node with fields for
1) pointer to doubly-linked adjacency list
2) pointer to each linked reverse adjacency list
3) s-number
4) "membership-in-S" bit
5) "edge-from-h" bit
6) pointer to linked adjacency list for subgraph G_S

An array containing nodes of G ordered by s-number.

An array containing nodes of G_S ordered by s-number.

Auxiliary stacks and temporaries, adjacency lists, et al.

Figure 8. Data representation for implementation

7. Discussion

We have presented a global flow analysis algorithm for reducible flow graphs and have analyzed its time complexity in number of function operations both in general and for special classes of program flow graphs.

The algorithm presented here is a major modification and generalization of interval analysis. Like interval analysis, the algorithm works only on reducible graphs and requires composition and intersection of functions. The iteration approach to global flow analysis works on all graphs and requires only functional application. However, almost all programs written yield reducible graphs, and reducibility has been proposed as a necessary condition for a "well-formed" or structured program [16]. In practice, all functions traditionally used in code optimization have been as easy to compose and intersect as they were to apply.

It should be noted that Algorithm A is easily modified to incorporate a test for reducibility of any flow graph. We are presently extending the algorithm to include irreducible graphs by relaxing the unique-entering-edge condition for T_2'. The complexity appears to be favorable compared with the node-splitting techniques for achieving reducibility.

For a flow graph $G = (N,E,n_0)$, both of the previously known methods have a worst case time of $O(|E|\cdot|N|)$ function operations. Their worst cases occur in loops which are nested deeply. The methods are often linear in practice because nesting level is almost independent of the length of the program. However, the nesting level, typically about 3, appears as a multiplicative factor (this is an oversimplification in the case of interval analysis). This increases the running time by a factor of 3. This does not occur in our algorithm. We expect that the running time of our algorithm will usually be significantly less than the running time of the previously known algorithms.

We have expressed our class of global flow problems as information propagation problems for sets of "facts". We could instead have used the more general bounded lattice-theoretic framework. The distributive law frequently required would be replaced by the weaker monotonicity requirement. We used the set formulation for ease of understanding by the reader. After analyzing the algorithm for fast functions, we indicated how to extend the algorithm to non-fast functions and then to infinite sets together with an information propagation space of functions which map finite sets to finite sets.

Our algorithm is easily generalized to handle global flow problems such as live-dead analysis for which one analyzes the reverse of the flow graph. We are also studying methods for increased computational efficiency. For many programming languages, such as the PASCAL subset mentioned in section 5, program loops can be analyzed before the entire program has been scanned. Consequently we can incorporate composition and intersection operations of global flow analysis in the parsing "semantics", leaving only a stack of function applications to be carried out after parsing is completed. For large programs, this technique can reduce the use of secondary storage for intermediate results during compilation. All of the above-mentioned extensions will be included in [21].

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.

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

[3] Angluin, D., Private communication, July 1974.

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

[5] Floyd, R.W., "Assigning Meanings to Programs," Proceedings American Mathematical Society Symposia in Applied Mathematics, Vol. 19, 1967, pp. 19-32.

[6] Fong, A., Kam, J. and Ullman, J.D., "Applications of Lattice Algebra to Loop Optimization," in these Proceedings.

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

[8] Hecht, M.S. and Ullman, J.D., "Analysis of a Simple Algorithm for Global Flow Problems," Proceedings ACM Symposium on Principles of Programming Languages, October 1973, pp. 207-217.

[9] Hecht, M.S. and Ullman, J.D., "Characterizations of Reducible Flow Graphs," Journal of the ACM, Vol. 21, No. 3, July 1974, pp. 367-375.

[10] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," Communications of the ACM, Vol. 12, No. 10, October 1969, pp. 576-583.

[11] Kam, J. and Ullman, J.D., "Global Optimization Problems and Iterative Algorithms," TR-146, Department of Electrical Engineering, Princeton University, January 1974.

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

[13] Kennedy, K., "A Comparison of Algorithms for Global Data Flow Analysis," Rice Technical Report 476-093-1, Rice University, February 1974.

[14] Kildall, G.A., "A Unified Approach to Global Program Optimization," Proceedings ACM Symposium on Principles of Programming Languages, October 1973, pp. 194-206.

[15] Paterson, M., unpublished memorandum, University of Warwick, Coventry, England, April 1972. See also [19].

[16] Peterson, W., Kasami, T. and Tokura, N., "On the Capabilities of While, Repeat, and Exit Statements," Communications of the ACM, Vol. 16, No. 8, August 1973, pp. 503-512.

[17] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, Englewood Cliffs, N.J., 1973.

[18] Tarjan, R.E., "Depth-first Search and Linear Graph Algorithms," SIAM Journal of Computing, Vol. 1, No. 2, September 1972, pp. 146-160.

[19] Tarjan, R.E., "Efficiency of a Good But Not Linear Set Union Algorithm," to appear in Journal of the ACM.

[20] Ullman, J.D., "Fast Algorithms for the Elimination of Common Subexpressions, Acta Informatica, Vol. 2, No. 3, December 1973, pp. 191-213.

[21] Wegman, M., Ph.D. Dissertation, in progress.

[22] Wirth, N., "The Programming Language PASCAL," Acta Informatica, Vol. 1, No. 1, 1971, pp. 35-63.

[23] Wulf, W.A., "A Case Against the GOTO," SIGPLAN Notices, Vol. 7, No. 11, November 1972, pp. 63-69.