TESTING FLOW GRAPH REDUCIBILITY

Robert Tarjan

TR 73 - 159

January 1973

Department of Computer Science
Cornell University
Ithaca, New York 14850

TESTING FLOW GRAPH REDUCIBILITY

Robert Tarjan

Department of Computer Science
Cornell University
Ithaca, New York

Abstract

Many problems in program optimization have been solved by applying a technique called interval analysis to the flow graph of the program. A flow graph which is susceptible to this type of analysis is called reducible. This paper describes an algorithm for testing whether a flow graph is reducible. The algorithm uses depth-first search to reveal the structure of the flow graph and a good method for computing disjoint set unions to determine reducibility from the search information. When the algorithm is implemented on a random access computer, it requires $O(E \log^* E)$ time to analyze a graph with E edges, where $\log^* x = \min\{i \mid \log^i x \leq 1\}$. The time bound compares favorably with the $O(E \log E)$ bound of a previously known algorithm.

Key words and phrases:

Algorithm, code optimization, complexity, depth-first search, directed graph, flow analysis, flow graph, interval analysis, program optimization, reducibility, set union algorithm, tree.

TESTING FLOW GRAPH REDUCIBILITY

Robert Tarjan

Department of Computer Science
Cornell University
Ithaca, New York

Introduction

Many code optimization methods model the flow of control in a computer program by a directed graph, called a flow graph. In order for some of these methods to work, the flow graph must have a special property called reducibility. Such methods include algorithms for finding dominators [1], finding common subexpressions [2,3], finding active variables [4,5], determining constant progagation [6], finding useless definitions [6], and solving other problems [7,8]. Some interesting classes of computer programs, such as "go-to-less-programs," give rise to flow graphs which are necessarily reducible [9], and most programs may be modelled by a reducible flow graph using a process of "node splitting" [10]. However, this can be computationally expensive. We would like a fast algorithm for determining whether these optimization methods can be applied to any given program; that is, an algorithm for determining whether a flow graph is reducible.

A "reducible" flow graph is a flow graph to which a technique called "interval analysis" may be applied to determine the graph's structure. Cocke [2] and Allen [7]

were the original formulators of this notion. Hecht and Ullman [9] simplified the definition of reducibility, giving two simple transformations which characterize the class of reducible graphs. They also gave a structural characterization of reducibility. Hopcroft and Ullman have constructed an algorithm which tests a graph for reducibility according to Hecht and Ullman's transformational definition. This algorithm has a running time of $O(E \log E)$ steps to test a graph with E edges and V vertices. In most applications, E is proportional to V, and this algorithm is an improvement over the obvious way to apply the definition, which requires $O(V^2)$ time.

This paper gives an algorithm which is asymptotically faster than Hopcroft and Ullman's and which always runs at least as fast as the obvious algorithm. The algorithm tests for Hecht and Ullman's structural condition. It happens that the algorithm simultaneously tests the transformational condition, which is more useful for applications. The method uses depth-first search [11,12] to reveal the structure of the flow graph and a good set union algorithm [13,14,15] to test reducibility using the search information. The exact running time of the algorithm depends upon the exact running time of the set union algorithm, which is unknown. However, a good bound on this running time is known, and the reducibility algorithm requires $O(\min\{E \log^* E, V \log V + E\})$ time to test a graph with V vertices and E edges, where $\log^* x = \min\{i \mid \log^i x \leq 1\}$. If $E > V \log V$, the algorithm requires

O(E) time and is optimal to within a constant factor, since every edge must be examined to determine reducibility. The new algorithm was independently discovered by Ullman [18].

Basic Notions

To study a graph algorithm we need first a model of computation and second some terminology from graph theory. We will assume that algorithms are to be implemented on some sort of random-access computer; data storage and retrieval, arithmetic operations, comparisons, and logical operations are assumed to require fixed times. A memory cell is allowed to hold integers proportional to the size of the problem (proportional to V, if the problem graph has V vertices). Cook [16] gives a formal model of this sort of computer. We ignore constant factors in time and space requirements; if f and g are functions of x we say "f(x) is O(g(x))" if, for some constants k_1 and k_2, $|f(x)| \leq k_1|g(x)|+k_2$ for all x.

The basic graphical definitions used here are common; see for instance [17]. A directed graph $G = (\mathcal{V},\mathcal{E})$ is an ordered pair consisting of a set of vertices $\mathcal{V}$ and set of edges $\mathcal{E}$. Each edge is an ordered pair (v,w) of vertices. We say the edge leaves v and enters w. An edge (v,v) is a loop. A graph $G_1 = (\mathcal{V}_1,\mathcal{E}_1)$ is a subgraph of a graph $G_2 = (\mathcal{V}_2,\mathcal{E}_2)$ if $\mathcal{V}_1 \subseteq \mathcal{V}_2$ and $\mathcal{E}_1 \subseteq \mathcal{E}_2$. A path p from v to w in a graph G is a sequence of vertices and edges leading from v to w. There is a path of no edges from any vertex to itself. A vertex w is reachable from a vertex v if there is a path from v to w. A flow graph (G,s) is a graph G with a distinguished

vertex s such that every vertex in G is reachable from s. A path is simple if all its edges are distinct. A path from a vertex to itself is a closed path. A closed path from v to v is a cycle if all its edges are distinct and the only vertex to appear twice is v, which appears exactly twice. A cycle contains at least one edge. Two cycles which are cyclic permutations of each other are considered to be the same cycle.

A (directed, rooted) tree T is a graph with one distinguished vertex, called the root r, such that every vertex in T is reachable from r, no edges enter r, and exactly one edge enters every other vertex in T. The relation "(v,w) is an edge in T" is denoted by $v \rightarrow w$. The relation "there is a path from v to w in T" is denoted by $v \overset{*}{\rightarrow} w$. If $v \rightarrow w$, v is the father of w and w is a son of v. If $v \overset{*}{\rightarrow} w$, v is an ancestor of w and w is a descendant of v. Every vertex is an ancestor and a descendant of itself. If $v \overset{*}{\rightarrow} w$ and $v \neq w$, v is a proper ancestor of w and w is a proper descendant of v. If T_1 is a tree and T_1 is a subgraph of a tree T_2, then T_1 is a subtree of T_2. If T is a tree which is a subgraph of a directed graph G and T contains all the vertices of G, then T is a spanning tree of G.

Given a flow graph (G,s), a depth-first search of G is an exploration of G which proceeds in the following way: We start at vertex s and choose an edge leaving s to explore. Traversing this edge leads to a vertex, either new or already reached. In general we continue the search by selecting and traversing an unexplored edge from the most recently reached vertex which still has unexplored edges. If G has V vertices

and E edges, we may carry out a depth-first search in $O(V+E)$ time if we are initially given a list of the edges in the graph. References [11] and [12] discuss an implementation of this algorithm. Other calculations, such as numbering the vertices from 1 to V in the order they are reached, may be performed during the search.

If a depth-first search of G is carried out from s and the vertices are numbered in search order, then every vertex in G is numbered [11]. Furthermore, the search partitions the edges of G into four classes [11,12]:

(1) A set of tree arcs defining a spanning tree T of G. These are the edges which lead to a new vertex during the search. A tree arc (v,w) has $v < w$ if we identify vertices by their number.

(2) A set of edges (v,w) with $w \overset{*}{\rightarrow} v$ in T, called <u>fronds</u>.

(3) A set of edges (v,w) with $v \overset{*}{\rightarrow} w$ in T, called <u>reverse fronds</u>.

(4) A set of edges (v,w) with neither $v \overset{*}{\rightarrow} w$ nor $w \overset{*}{\rightarrow} v$, called <u>cross-links</u>. A cross-link has $w < v$.

Figures 1 and 2 show the application of depth-first search to a graph. Depth-first search simplifies the structure of the paths in a graph. The following lemma is implicit in [11] and proved in [12].

<u>Lemma 1</u>: Let (G,s) be a flow graph which is explored using depth-first search. Let T be the generated spanning tree of G. If p is a path from v to w and $v < w$, then p passes through some common ancestor of v and w in T.

Flow Graphs and Reducibility

Consider the following two transformations on a flow graph (G,s):

T_1: Delete a loop (v,v) in G.

T_2: If (v,w) is the only edge entering w and $w \neq s$, delete vertex w. For every old edge (w,x) add a new edge (v,x); for every old edge (x,w) add a new edge (x,v). Delete duplicate edges. This transformation is called collapsing vertex w into vertex v.

A flow graph is reducible if it can be transformed into the graph consisting only of vertex s by repeated application of T_1 and T_2. This definition is Hecht and Ullman's simplification of Cocke and Allen's notion. If G' is obtained from G by repeated application of T_1 and T_2, then G' is a reduction of G. Any reduction of a reducible graph is reducible [9]; thus the order of applying transformations doesn't matter in a test for reducibility. If G' is a reduction of G, each vertex v in G' corresponds to a set of vertices in G; namely those collapsed into v. We say v represents this set of vertices.

The obvious way to test reducibility is to try applying T_1 and T_2 to the graph G to be tested. We make one pass over the graph, deleting all loops and counting the number of edges entering each vertex. Then we find a vertex with only one entering edge and apply T_2, updating the number of edges entering other vertices. We repeat until we reduce the graph completely or we get stuck. Each application of T_2

requires O(V) time, so this algorithm has an $O(V^2)$ time bound. Hopcroft and Ullman have improved this algorithm to O(E log E) by applying a clever method of updating information after T_2 is applied. Hopcroft and Ullman's algorithm is only an improvement if E is small relative to V^2, but this is almost always true in any flow graph representing a real program.

Hecht and Ullman give a structural characterization of reducible graphs. It is from this characterization that we shall build a faster reducibility algorithm. Later we shall see that the algorithm also tests reducibility according to the definition.

Lemma 2: Let (G,s) be a flow graph. G is reducible if and only if there do not exist distinct vertices $v \neq s$ and $w \neq s$, paths p_1 from s to v and p_2 from s to w, and a cycle c containing v and w, such that c has no edges and only one vertex in common with each of p_1 and p_2.

Proof: See [9].

To use this characterization effectively, we need to strengthen it somewhat. To test the reducibility of G, we explore G using depth-first search. This process generates a numbering of the vertices, a spanning tree T, and sets of fronds, reverse fronds, and cross-links. The search requires O(V+E) time if G has V vertices and E edges. Henceforth we shall identify vertices by their number. Lemma 2 becomes:

Lemma 3: G is reducible if and only if G contains no simple path p from s to some point v such that v is a proper ancestor of some other point on p.

Proof: Suppose G is not reducible. Then vertices v, w and paths c, p_1, p_2 exist satisfying the condition in Lemma 2. Without loss of generality assume v < w. Let c_1 be the part of c from v to w. By Lemma 1, c_1 contains some common ancestor u of v and w. Let p be the path consisting of p_2 followed by the part of c from w to u. Then p satisfies the condition in the lemma.

Conversely, suppose p satisfies the condition in the lemma. Let v be the first vertex on p which is a proper ancestor of some earlier vertex on the path. Let w be the first vertex on p which is a descendant of v. Then v and w satisfy the condition in Lemma 2 for suitable paths c, p_1, and p_2, and G is not reducible.

For any vertex v in G, let HIGHPT(v) be the highest numbered proper ancestor of v such that there is a path p from v to HIGHPT(v) and p includes no proper ancestors of v except HIGHPT(v). By convention HIGHPT(v) = 0 if there is no path from v to a proper ancestor of v.

Lemma 4: G is reducible if and only if there is no vertex v with an entering edge (u,v) such that if w is the highest common ancestor of u and v, w < HIGHPT(v).

Proof: Suppose the condition holds. Then (u,v) is either a reverse frond (w = u) or (u,v) is a cross-link. Let p_1 be a path from v to HIGHPT(v) which passes through no proper ancestors of v except HIGHPT(v). Let p be the path of tree arcs from s to u followed by edge (u,v) followed by p_1. Then p is simple and satisfies the condition in Lemma 3, so G is not reducible.

Conversely, suppose G is not reducible. Then by Lemma 3 there is a path p from s to v with v a proper ancestor of some other vertex on p. Choose p as short as possible. Let w be the first vertex on p which is a descendant of v. Then w , HIGHPT(w), and the edge of p entering w satisfy the condition above.

To test G for reducibility, we calculate HIGHPT(v) for each vertex and apply Lemma 4. For cross-links, we check the condition in Lemma 4 during the HIGHPT calculation. For reverse fronds, we check the condition in Lemma 4 after the HIGHPT calculation by testing each reverse frond. This last step requires O(V+E) time. We may ignore reverse fronds in the HIGHPT calculation since if p is a path from v to w and p contains no ancestors of v except v and w, we may substitute a path of tree arcs for each reverse frond in p and still have a path from v to w which contains no ancestors of v except v and w.

To calculate HIGHPT values, we order the fronds (u,v) by the number of v. This requires O(V+E) time using a radix sort and V buckets. Then we process the fronds (u,v) in order, from highest v to lowest v. Initially all vertices are unlabelled.

To process (u,v) we proceed down the tree path from u to v, labelling each currently unlabelled vertex with label v. (We don't label v itself). If a vertex w gets labelled, we examine all cross-links entering w. If (z,w) is such a cross-link, we proceed down the tree path from z to v labelling each unlabelled vertex with label v. If z is not an ancestor of v then G is not reducible by Lemma 4 and the calculation stops. We continue labelling until we run out of cross-links entering just-labelled vertices; then we process the next frond. When all fronds are processed, the labels give the HIGHPT values of the vertices. Each unlabelled vertex has HIGHPT 0. The algorithm is given in Algol-like notation below.

```
comment procedure to calculate HIGHPT(v) for each v;
for i: = 1 until V do
    begin comment initialization;
       HIGHPT(i): = 0;
       BUCKET(i): = the empty list;
    end;
a: for each frond (u,w) in G do add (u,w) to BUCKET(w);
b: for w: = V_1 step - 1 until 1 do
   while BUCKET (w) is not empty do
       begin
         delete (u,w) from BUCKET(w);
         CHECK: = {u};
         while CHECK is not empty do
            begin
               delete u from CHECK;
               c: if ¬(w →* u) then go to not reducible;
               d: while u ≠ w do
                  begin
```

```
        e: if HIGHPT(u) = 0 then
           begin
              HIGHPT(u):= w;
              for each cross-link (v,u) do add v to CHECK;
           end;
        f: u:= FATHER(u);
     end;
  end;
end;
comment if G is reducible then the program reaches
   this point;
```

Lemma 5: If G with reverse fronds deleted is reducible, then the algorithm above finishes, and it calculates HIGHPT values correctly. If G with reverse fronds deleted is not reducible, then test c succeeds at some time and the algorithm doesn't finish.

Proof: If p is a path from v to HIGHPT(v) and p contains no proper ancestors of v except HIGHPT(v), then p ends with a frond. This follows from Lemma 1. Let us first assume that G with reverse fronds deleted is reducible. We prove by induction on w that after all fronds entering w have been processed, all vertices v with HIGHPT(v) $\geq$ w will be correctly labelled and all other vertices will be unlabelled. This is certainly true initially. Suppose it is true after all fronds entering w+1 have been processed. Then each vertex u labelled during processing of a frond entering w has HIGHPT(u) $\geq$ w, since there is a path of the proper type from u to w.

If HIGHPT(u) $>$ w, u will have been previously labelled. Thus HIGHPT(u) = w.

Suppose HIGHPT(u) = w. Let p be a path from u to w which contains no proper ancestors of u except w. We know p contains no reverse fronds. Further, p can contain only one frond since G is reducible. If any vertex on p is labelled, then G is not reducible, so every vertex on p must be unlabelled. It follows that u gets labelled when a frond entering w is processed.

Now suppose G with reverse fronds deleted is not reducible. Then G satisfies the condition in Lemma 4 for some v with (u,v) a cross-link. The calculation will proceed correctly until a cross-link satisfying this condition is found. Then test c will succeed. This may be proved rigorously by induction as above.

Once we have calculated the HIGHPT values using the algorithm above, we need to check the condition in Lemma 4 only with respect to the reverse fronds, since the HIGHPT calculation checks the condition with respect to the cross-links. This last step is straightforward, and requires O(V+E) time. Two parts of the HIGHPT calculation are a little tricky. First, test c requires that we be able to determine whether a vertex w is a descendant of another vertex u. Let ND(u) be the number of descendants of vertex u in T. Then $u \overset{*}{\rightarrow} w$ if and only if $u \leq w < u + ND(u)$ [12]. We can calculate ND(u) during the depth-first search in a straightforward fashion [12]. This gives an ancestry test.

We also need to avoid examining vertices which have already been labelled if we are to get an efficient algorithm

We borrow a method used in [12], based on a good algorithm for computing disjoint set unions. We shall have sets numbered 1 to V. A vertex $w \neq 1$ will be in set v if v is the highest numbered, unlabelled proper ancestor of w. Since vertex 1 never gets labelled, each vertex is always in a set; initially a vertex is in the set whose number is its father in T. To carry out step f, we find the number u' of the set containing u and let that be the new u. We also combine the sets numbered u and u' to form a new set numbered u'; when u becomes labelled, u' becomes the highest numbered, unlabelled proper ancestor of any vertex in the old set u. The necessary program modifications are:

```
comment initialization;
for i: = 1 until V do SET(i): = the empty set;
for i: = 2 until V do add i to SET(FATHER(i));

   comment modified step d;
d: while ¬ (u →* w) do
   begin
      g: u':= the number of the set containing u;
      e: if HIGHPT(u) = 0 then
         begin
            HIGHPT(u): = w;
            for each cross-link(v,w) do add v to CHECK:
            SET(u'):= SET(u) ∪ SET(u');
         end;
      f: u:= u';
   end;
```

Lemma 6: The HIGHPT calculation modified as above works correctly and requires O(V) set unions, O(V+E) executions of step g, and O(V+E) time exclusive of set operations.

Proof: We may verify that the algorithm works correctly by showing that each SET(v) contains exactly the vertices w such that v is the highest numbered, unlabelled proper ancestor of w. This is straightforward. Consider the running time of the algorithm. The while loop is entered from the top O(E) times. Test e can only fail if the loop is entered from the top; otherwise u must be unlabelled because u is defined in step g. Each vertex is labelled only once. Thus there are O(V) set unions and O(V+E) executions of step g. It is easy to see that the total time required by other steps is O(V+E).

Corollary: The HIGHPT calculations require $O(\min\{V \log V + E, E \log^* E\})$ time if a good algorithm for computing disjoint set unions is used. (Note: $\log^* x = \min\{i \mid \log^i x \leq 1\}$.)

Proof: If we use the algorithm described in [13,14,15] for computing the set unions and performing step g, the bound follows from the bounds on the set operations [12,14,15] and from Lemma 6.

Corollary: The reducibility algorithm requires $O(\min\{V \log V + E, E \log^* E\})$ time and O(V+E) space to test a graph with E edges and V vertices.

Proof: The depth-first search and associated calculations require O(V+E) time. The last step, to check the condition in Lemma 4 for reverse fronds, requires O(V+E) time. The HIGHPT calculations are the slow part of the algorithm, with time bound given above. Combining gives the total time bound. It is clear that the algorithm requires O(V+E) space.

Reducing a Reducible Graph

The algorithm described here is fast but non-constructive; that is, it does not tell us what sequence of transformations will reduce a reducible graph G. However, we can get this information out of the calculations the algorithm performs. Suppose G' is a reduction of G. Then it is clear that each vertex v in G' represents a subtree of the spanning tree T of G, and that v is the root of this subtree.

We can assign numbers, called SNUMBER's, to the vertices of G so that tree arcs (v,w) satisfy SNUMBER(v) < SNUMBER(w) and cross-links (v,w) also satisfy SNUMBER(v) < SNUMBER(w). This can be done during a depth-first search of G[12], and corresponds to traversing the spanning tree of G using depth-first search and proceeding to highest numbered vertices first. Suppose we apply the reducibility algorithm, and each time we label a vertex we associate with it a pair (HIGHPT(v), SNUMBER(v)). When the calculation is finished, we order the vertices so that a vertex labelled (x_1,y_1) appears before

a vertex labelled (x_2,y_2) if and only if $x_1 > x_2$ or $x_1 = x_2$ and $y_1 < y_2$. This order of vertices is called reduction order. Note that an unlabelled vertex v has an associated pair (0, SNUMBER(v)).

Lemma 7: If G is reducible, then we may collapse the vertices of G in reduction order using T_2 (interspersed with applications of T_1).

Proof: We prove the lemma by induction on the number of vertices collapsed. Suppose all the vertices up to v in reduction order may be collapsed. This creates a graph G' which is a reduction of G. Consider vertex v. If v is not the start vertex, a single tree arc enters v in G. If G contains a frond (u,w) with $v \overset{*}{\rightarrow} w$, all vertices x on the tree path from u to w will have been collapsed before v, since HIGHPT(x) $\geq$ w and HIGHPT(v) < v $\leq$ w. If G contains a reverse frond (u,w) with $v \overset{*}{\rightarrow} w$, then HIGHPT(w) $\leq$ u by Lemma 4. Furthermore HIGHPT(x) $\geq$ HIGHPT(w) and SNUMBER(x) $\leq$ SNUMBER(w) for all vertices x on the tree path from u to w. It follows that all vertices on the tree path from u to w have been collapsed before w. Suppose G contains a cross-link (u,w) with $v \overset{*}{\rightarrow} w$. Let x be the least common ancestor of u and w. Then HIGHPT(w) $\leq$ x by Lemma 4, and all vertices y on the tree paths from x to w and from x to u satisfy HIGHPT(y) $\geq$ HIGHPT(w) and SNUMBER(y) $\leq$ SNUMBER(w). Thus all vertices on these tree paths have been collapsed before w. It follows that in G' vertex v can have only one edge entering it, and we may collapse v. By induction the lemma holds.

Thus our originally non-constructive reducibility test actually gives us a reduction order for any reducible graph. The SNUMBER calculations may be done during the depth-first search and require only O(V+E) time, so the constructive reducibility test has the same time bound as the non-constructive version. Figure 3 gives HIGHPT values and a reduction order for the graph in Figure 2.

Conclusions:

This paper has presented an algorithm with an almost-linear time bound for determining whether a flow graph is reducible. The algorithm may be used to determine a way to reduce the graph if such a way exists. The method uses depth-first search and a good algorithm for computing disjoint set unions, and it improves upon a previously published algorithm for determining reducibility. The algorithm may be used as a basic subroutine for various code optimization procedures [1,2,3,4,5,6,7,8]. Many of these procedures use non-linear algorithms, some of which may be improvable using the methods applied here.

[15] R. Tarjan, "On the Efficiency of a Good but Not Linear Set Union Algorithm," TR 72-148, Department of Computer Science, Cornell University (November, 1972).

[16] S.A. Cook, "Linear Time Simulation of Two-Way Pushdown Automata," Proc. IFIP Congress, 1971, TA-2, North Holland Publishing Co., Amsterdam, 1972, 174-179.

[17] R.G. Busacker and T.L. Saaty, Finite Graphs and Networks: An Introduction with Applications, McGraw-Hill Book Co., New York, 1965.

[18] J.D. Ullman, private communication, December, 1972.

References:

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "On Finding the Least Common Ancestors in Trees," submitted to the 1973 ACM Symposium on Theory of Computing, Austin, Texas (April 30-May 2, 1972).

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

[3] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions," 13th Annual Symposium on Switching and Automata Theory, IEEE (October, 1972), 161-176.

[4] R. Kennedy, "A Global Flow Analysis Algorithm," International J. Computer Mathematics, Vol. 3, No. 1 (December, 1971), 5-16.

[5] M. Shaeffer, "A Mathematical Theory of Global Program Analysis," unpublished notes, System Development Corporation, Santa Monica, Calif., 1971.

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

[7] F.E. Allen, "Control Flow Analysis," SIGPLAN Notices, Vol. 5 (1970), 1-19.

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

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

[10] J. Cocke and R.E. Miller, "Some Analysis Techniques for Optimizing Computer Programs," Proc. Second International Conference on System Sciences, Honolulu, Hawaii, 1969.

[11] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," SIAM J. Computing, Vol. 1, No. 2(June, 1972), 146-159.

[12] R. Tarjan, "Finding Dominators in Directed Graphs," unpublished, Cornell University, (December, 1972).

[13] M. Fischer, "Efficiency of Equivalence Algorithms," Complexity of Computer Computations, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, New York, 1972, 153-168.

[14] J.E. Hopcroft and J.D. Ullman, "Set Merging Algorithms," submitted to SIAM J. Computing.

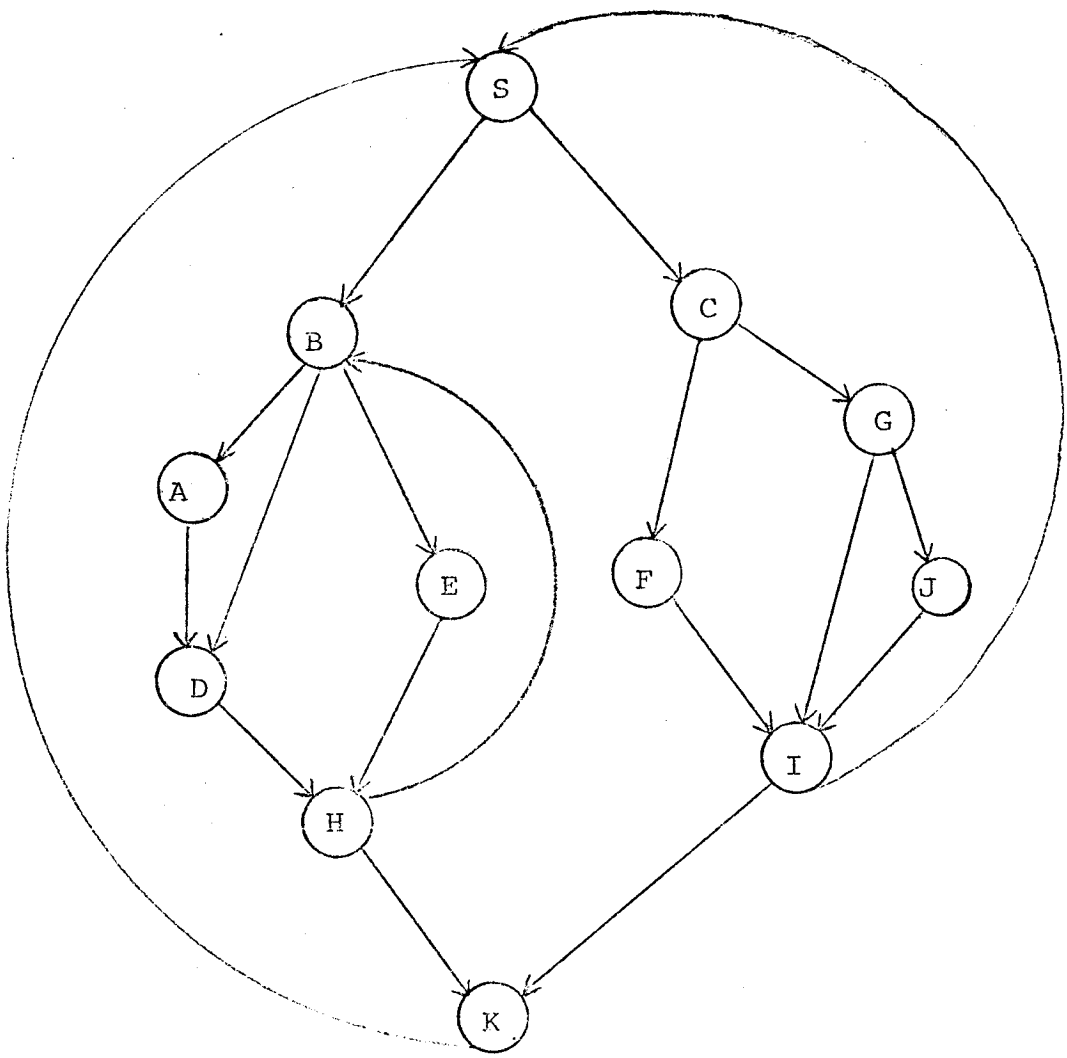

Figure 1: A flow graph. Is this graph reducible?

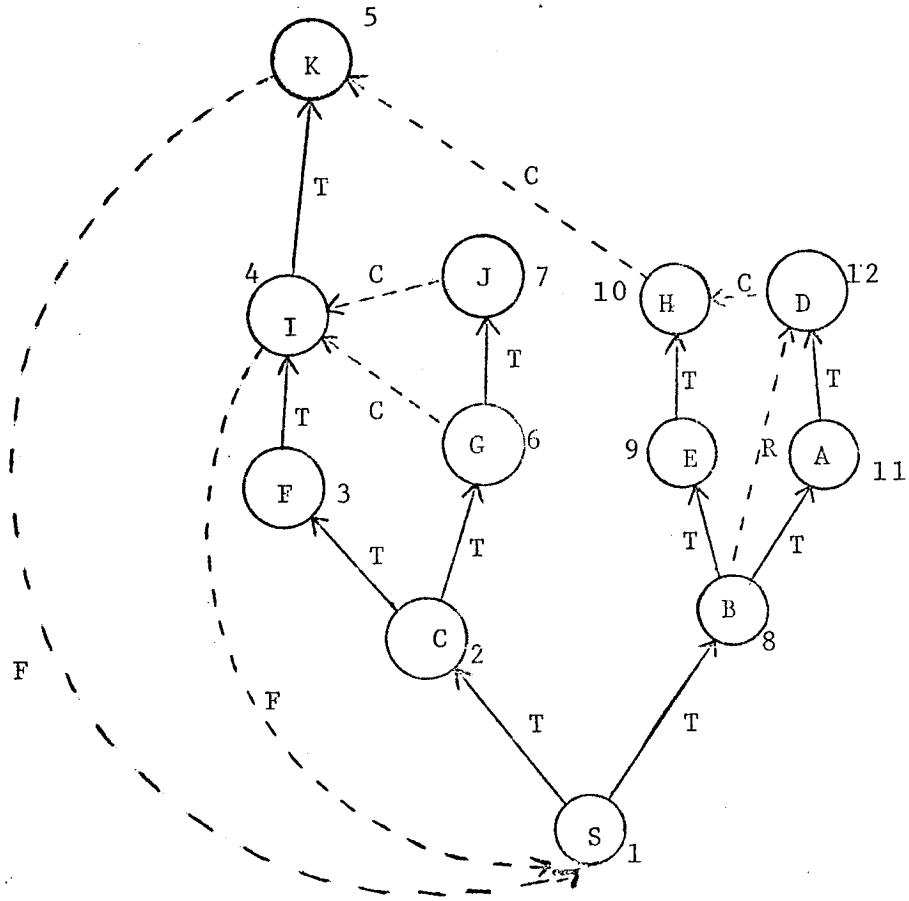

Figure 2: Depth-first search of the graph in Figure 1. Vertices are numbered in search order. Tree arcs are labelled T, fronds F, reverse fronds R, and cross-links C.

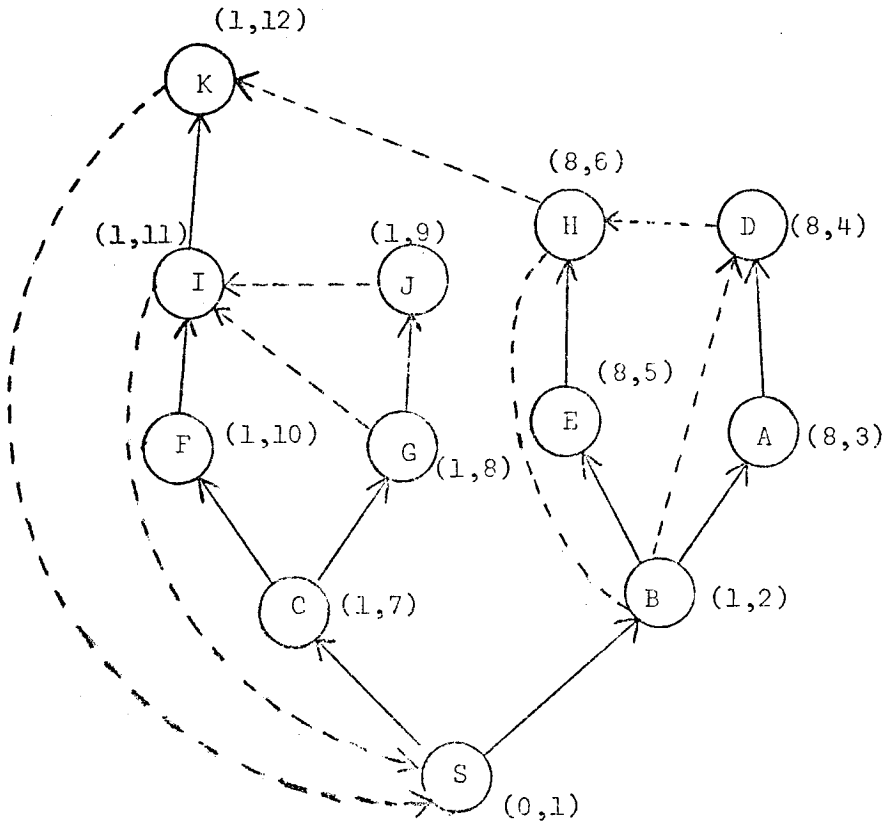

Figure 3: HIGHPT and SNUMBER values (in parentheses) for the graph in Figure 2. A reduction order is: A,D,E,H,B,C,G,J,F,I,K.