Depth-First Search Techniques and Efficient Methods for Creating Test Paths *

Leon Osterweil

CU-CS-077-75

* This work supported by NSF grant GJ-3646.

Depth-First Search Techniques and Efficient Methods for Creating Test Paths*

by

Leon Osterweil
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

TR #CU-CS-077-75 August 1975

*This work supported by NSF Grant GJ-36461

Introduction

This paper is intended as a tutorial in the use of search techniques to construct constrained paths through program flow graphs. Currently it seems that incidence, or adjacency, matrices are widely used by practitioners in analyzing flow graphs. In this paper, however, it shall be shown that sizable gains in both speed and storage utilization are possible through the use of search techniques in place of matrix manipulation.

The popularity of incidence matrices as tools for graph investigation seems due largely to some early work of Ramamoorthy [1] and Prosser [2]. More recently, others such as Hopcroft and Tarjan [3] have developed search techniques for performing diverse and powerful graph analysis. Through the use of searching it has been possible to create a wide variety of useful graph manipulative algorithms whose speeds and storage requirements are improved, often by orders of magnitude, over comparable algorithms using incidence matrices (see, for example [4,5,6,7,8]). Hence searching algorithms are currently extremely interesting to researchers in algorithmic complexity. Unfortunately, it appears that actual practitioners who routinely deal with graphs in the course of program analysis have often continued to use incidence matrices in their work.

The goal of this paper is to illustrate the power, versatility, and efficiency of searching by presenting some algorithms as examples. These examples are all related to the problem of generating paths through a graph, a problem of significant interest in program testing and validation (see, for example, [9]). Most of the examples are rather straightforward adaptations of one basic, well-known algorithm. The author believes, however, that these useful example algorithms have never been presented before, and are therefore not widely known or used among practitioners.

The presentation of these actual algorithms should therefore be of direct benefit to some practitioners. It is hoped, moreover, that this illustration of the adaptability and efficiency of the basic searching concept may serve as a stimulus to other practitioners to use the concept in performing necessary graph manipulations arising in other areas of program analysis.

The paper is concluded with the presentation of some recent results concerning the construction of paths under important and more challenging constraints.

The Edge-List Representation of a Graph

In the remainder of this paper it shall be implicitly assumed that the graphs under consideration are all program flow graphs. Hence the graphs are all assumed to be directed. Moreover, because the nodes of a program flow graph represent the program's basic blocks, the term block may be used interchangeably with the term node. It is important to note, however, that despite this terminology and the consistent use of flow graphs as examples, the representations and algorithms described in this paper are fully applicable to all directed graphs.

The search procedures described below all rely upon the representation of a directed graph by means of an edge-list, as opposed to an incidence matrix. The edge list representation employs two vectors, which we shall call HEAD and FIRST.

The vector HEAD contains an entry for each edge in the graph. The actual entry made in HEAD as a representation of a given edge, say from block A to block B, is simply the node number B - the destination, or head, node of the edge. Hence HEAD is a vector of the heads of all graph edges.

The vector FIRST has an entry for every block in the graph. The entry made in FIRST for a given block is the location in HEAD of the first representation of an edge which originates from the given block - that is FIRST [A] is the location in HEAD of the first edge whose tail is A. In a graph having v blocks it is useful for the purposes of the algorithms to maintain an entry FIRST [v+1] which points to the first location in HEAD which follows the last edge represented by HEAD.

As an example, consider the flow graph in Figure 1.

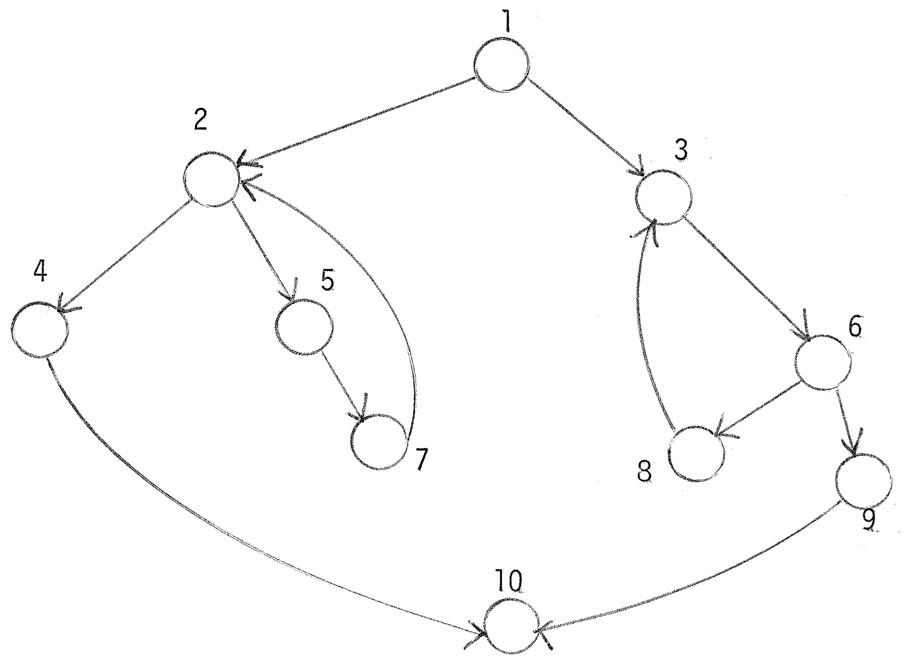


Figure 1 - A Sample Flow Graph

FIRST

1	1
2	3
3	5
4	6
5	7
6	8
7	10
8	11
9	12
10	13
11	13

HEAD

1	2
2	3
3	4
4	5
5	6
6	10
7	7
8	8
9	9
10	2
11	3
12	10
13	----

Figure 2 - An Edge List Representation of the Flow Graph in Figure 1.

An edge list representation of the graph in Figure 1 is given in Figure 2.

Note that all the blocks directly reachable from block i, $1 \leq i \leq v$, are listed from HEAD[FIRST[i]] to (but not including) HEAD[FIRST[i+1]]. Thus, for example, the blocks directly reachable from block 2 are listed starting in HEAD[FIRST[2]] (or HEAD[3], namely 4) and going up to HEAD[FIRST[3] -1](or HEAD[5-1] that is HEAD[4] or 5). Thus from block 2 there are direct transfers to blocks HEAD[3] and HEAD[4], i.e., blocks 4 and 5. The case of block 10 is slightly awkward. The first block reachable from 10 should be listed in HEAD[13], but the last block reachable should be in HEAD[12]. In fact, in an edge list representation of a graph, this is a reliable indication that the block has no out-edges. That is, if FIRST[i] = FIRST[i+1], block i has no successors (it is presumably a stop block). The algorithms which use this representation are, of course, designed to recognize and exploit this.

An immediate virtue of this representation is its compactness. Note that FIRST is v+1 entries long, and HEAD is e entries long, where v is the number of program basic blocks and e is the number of edges. Hence the edge list representation requires O(v+e) entries. Because a basic block rarely has more than 2 out-edges (the three out-edges in the case of a FORTRAN arithmetic IF, and perhaps even more in the case of a CASE statement or FORTRAN computed GO TO, are exceptions to this), e in the case of an actual program flow graph is generally O(v), and is almost always bounded by 2v. Hence this representation requires, for virtually all programs, O(v+2v) = O(v) storage. Storage for an incidence matrix requires, of course, $O(v^2)$ entries. To be totally fair, these v^2 entries can be single bits, but note that edge-list entries also tend to be small integers, requiring no more that $\lceil \log_2 v \rceil$ bits for HEAD entries and $\lceil \log_2 e \rceil$ bits for FIRST entries. Hence the entries for an edge list representation can likewise be packed several to a word.

One last note about the edge list representation should be made. For clarity, the edge-lists are represented here as being sequential lists. It is sometimes not practical, however, to build such a representation as the analysis of actual source text is proceeding, owing to the fact that the block numbers or even the very existence of some graph edges may suddenly become known at awkward times during the analysis. For this reason, the edge-lists for the various blocks are often represented as linked lists. This requires that HEAD be an $e \times 2$ matrix, where one column contains head block numbers as before, but the other column contains links, each link pointing to another head reachable from the given tail block. Figure 3 illustrates this representation. Zero entries (both in FIRST and column 2 of HEAD) indicate the absence of additional (or any) destination blocks.

Although the linked edge list representation is often preferable to the original sequential edge representation, the remainder of this paper will assume the sequential representation, primarily for reasons of algorithmic simplicity. Adaptations of the algorithms which operate on the linked representation are not hard. They do, however, require auxiliary vectors whose lengths, nevertheless, are $O(v)$.

FIRST

1	1
2	2
3	6
4	7
5	8
6	9
7	12
8	10
9	11
10	0

HEAD

	heads	next heads (links)
1	2	5
2	4	3
3	5	0
4	9	0
5	3	0
6	6	0
7	10	0
8	7	0
9	8	4
10	3	0
11	10	0
12	2	0

Figure 3 - Linked Edge List Representation of the Graph in Figure 1

The Basic Path Generation Algorithm

The path generation algorithm presented here is actually an adapted depth first search procedure. A search procedure on a graph may be characterized as a procedure which causes all of the nodes and edges of the graph to be visited in a systematic way. A search always proceeds from one node to another by following an edge of the graph. Two important types of search are in common use - the breadth first search and the depth first search. A breadth first search is one in which the nodes are visited one at a time, with all edges emanating from one node being examined before consideration of the next node. In a depth first search, whenever a new node, N_1, is visited, a single out-edge is traversed in search of another new node, N_2. A second out-edge from N_1 will not be traversed until all out-edges from N_2 have been traversed. In both types of search a special mark is used to allow visited nodes to be distinguished from unvisited nodes. In this way, needless searching is avoided. The depth first search seems to be a likely basis for a path generation algorithm, because by its very nature, it burrows deeply into a graph rapidly and directly.

The basic path generation algorithm can now be stated. This algorithm differs only superficially from others in the literature such as the connected component algorithm in [3], attributed there to Shirey [10], and the classical garbage collection marking algorithm (see Algorithm B, p. 145 of [11]) which has been known for quite some time.

In addition to the arrays described in the previous section, some additional arrays are needed for the algorithm:

STACK - a vector of length v, which will be used as a stack. This vector will keep track of the blocks currently being explored by the search. Hence the desired path will be built in this vector.

MARK - a vector of v bits which is to contain the marking information needed to insure the efficient, non-repetitious visitation of blocks.

CURRENTROW - a vector of length v, whose ith entry is the row in HEAD of the edge out of block i which is currently being used as part of the evolving path.

The following simple variables are also used by this algorithm:

PTR - the stack pointer.

NEWHEAD - a block which may be used to extend the currently evolving path.

CURRENTBLOCK - the block to which the evolving path currently extends.

V - the number of basic blocks in the graph.

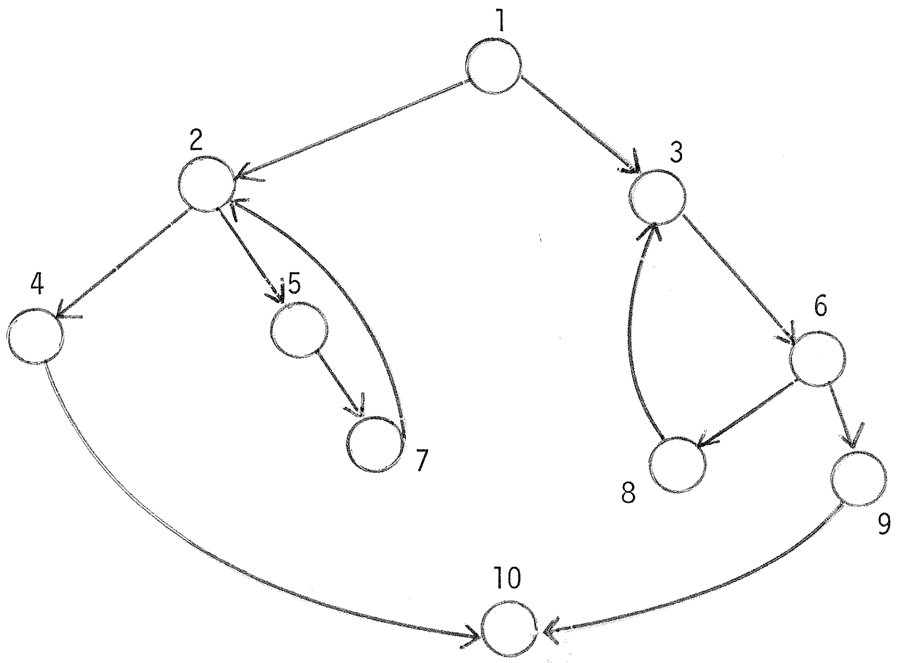
Note that the algorithms stated here use the notation [ ] to denote subscripting and $\leftarrow$ to denote assignment.

Algorithm P (Path tracer): Assume that BEGINBLOCK is a variable containing the number of the block at which the path is to start, and ENDBLOCK is a variable containing the number of the block at which the path is to stop.

```
1   procedure P;
2     begin
3     procedure NEXTEDGE;
        comment NEXTEDGE checks whether CURRENTROW[CURRENTBLOCK]
                         indicates an actual edge out of CURRENTBLOCK.
                         If so, it returns.  If not, it backtracks to
                         previously stacked blocks until finding an
                         actual edge;
4       begin
5       while CURRENTROW[CURRENTBLOCK]=FIRST[CURRENTBLOCK+1]do
            comment no further edges out of CURRENTBLOCK, back up;
6           begin
7           PTR←PTR-1;
8           if PTR=0 then stop
            comment the desired path does not exist;
9             else begin
10            CURRENTBLOCK←STACK[PTR];
11            CURRENTROW[CURRENTBLOCK]←CURRENTROW[CURRENTBLOCK]+1
12            end;
13          end;
14      end NEXTEDGE;
      comment initialize P;
15    for I←1 to V do MARK[I]←0;
16    PTR←0;
      comment initialize the search;
17    CURRENTBLOCK←BEGINBLOCK:
18    while CURRENTBLOCK≠ ENDBLOCK do
        comment continue the search until ENDBLOCK is encountered;
19      begin
        comment mark and stack newly encountered block;
20      MARK[CURRENTBLOCK]←1;
21      PTR←PTR+1;
22      STACK[PTR]←CURRENTBLOCK;
        comment get an edge out of the newly encountered block and
                follow it to its head;
23      CURRENTROW[CURRENTBLOCK]←FIRST[CURRENTBLOCK];
24      NEXTEDGE;
25      NEWHEAD←HEAD[CURRENTROW[CURRENTBLOCK]];
26      while MARK[NEWHEAD]=1 do
            comment if head has been visited, get the next edge,
                    examine its head, continue until an unvisited
                    head is found;
27          begin
28          CURRENTROW[CURRENTBLOCK]←CURRENTROW[CURRENTBLOCK]+1;
29          NEXTEDGE;
30          NEWHEAD←HEAD[CURRENTROW[CURRENTBLOCK]];
31          end;
        comment a new unvisited node is found;
32      CURRENTBLOCK←NEWHEAD;
33      end;
      comment ENDBLOCK has been reached;
34    PTR←PTR+1;
```

```
      STACK[PTR]←ENDBLOCK;
      comment print path and terminate successfully;
      for I←1 to PTR do print STACK[I];
      stop
  end P;
```

It seems useful to show how this algorithm works in finding a path from 3 to 9 for the graph in Figure 1. Hence, in Figures 4a - 4g, the status of the algorithm and its key variables are shown as the algorithm executes. In these figures a marked block is indicated by coloring it black, while NEWHEAD is indicated by a □.



BEGINBLOCK: 3

ENDBLOCK: 9

Figure 4a: Algorithm starts

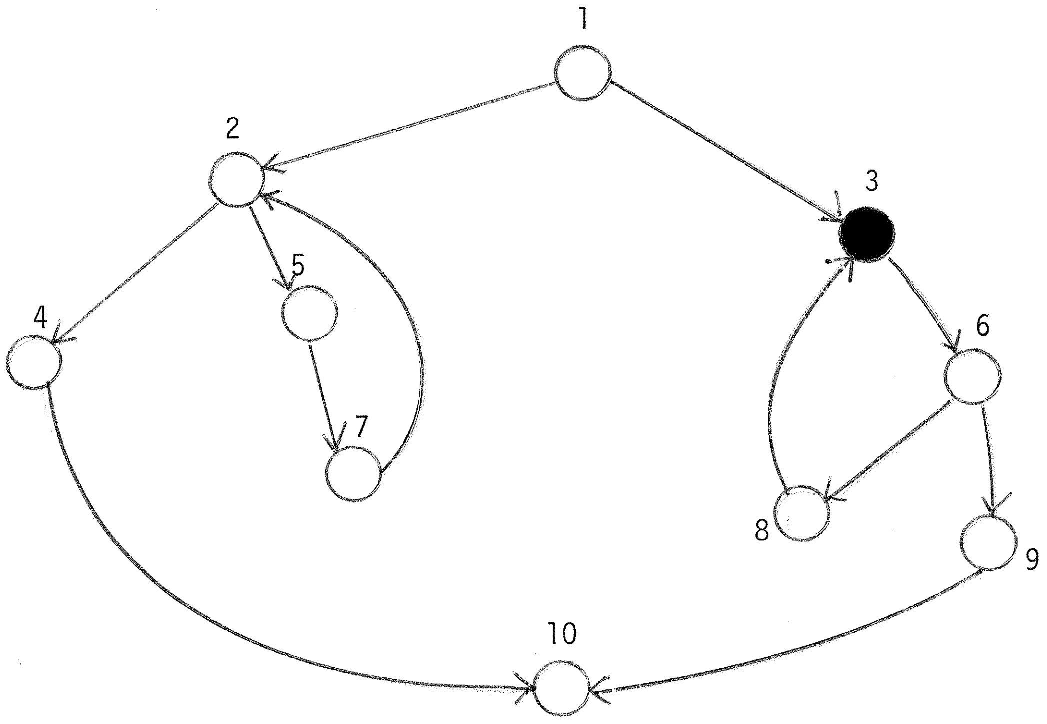


Figure 4b: After execution of line 22

STACK: 3

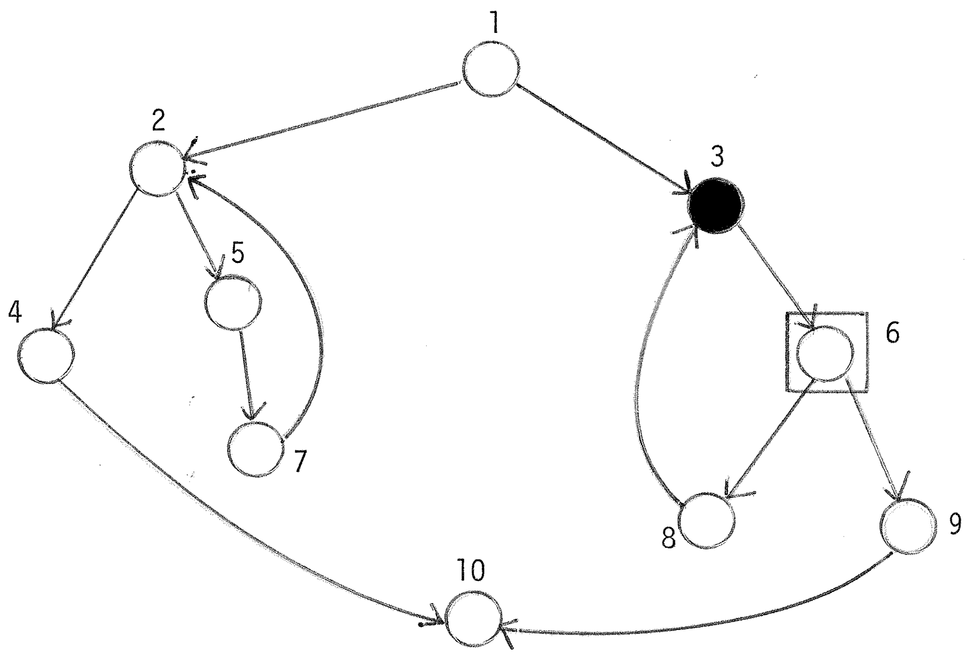


Figure 4c: After the first execution of line 32:

STACK: 3
NEWHEAD: 6
CURRENTBLOCK: 6
CURRENTROW [3]: 5

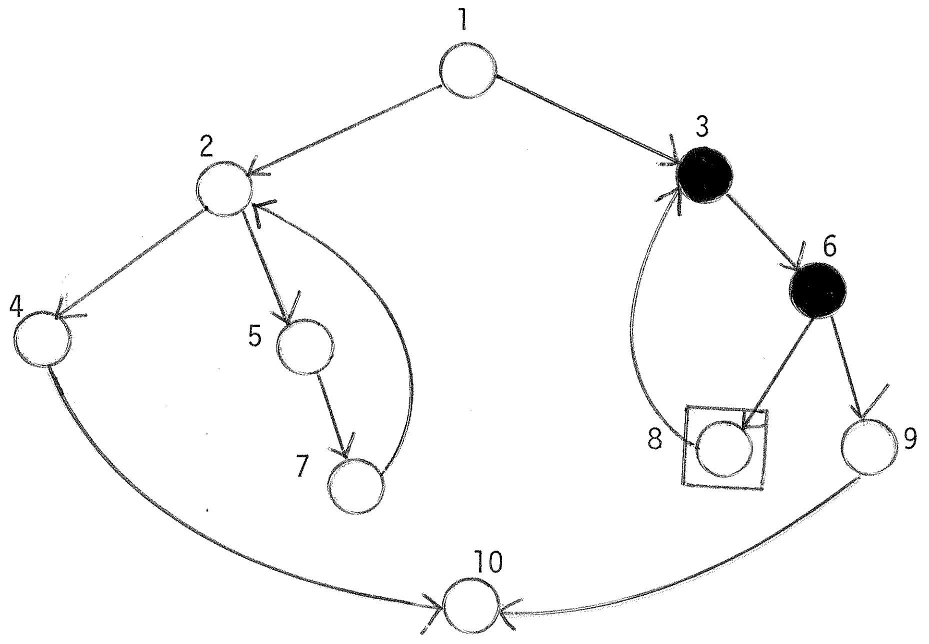


Figure 4d: After another iteration of the major loop, line 32 is again executed, and the status is:

STACK: 3,6
NEWHEAD: 8
CURRENTBLOCK: 8
CURRENTROW[3]: 5
CURRENTROW[6]: 8

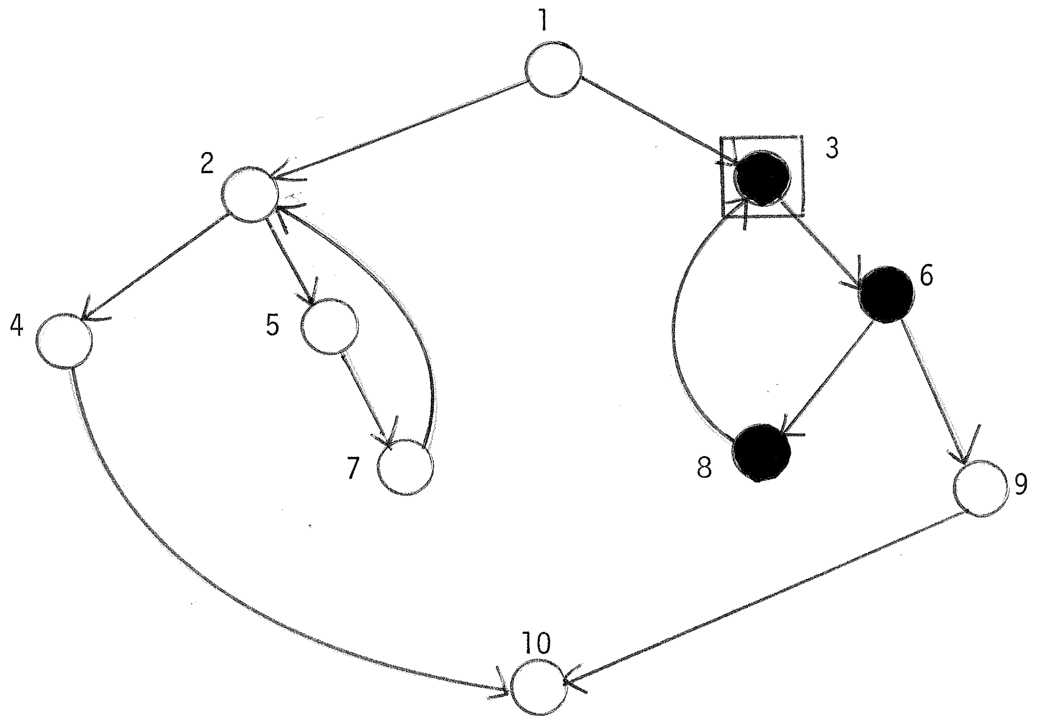


Figure 4e: On the next iteration of the major loop, the test at line 26 will succeed. Just before this point: STACK: 3, 6, 8
NEWHEAD: 3
CURRENTROW[8]: 11

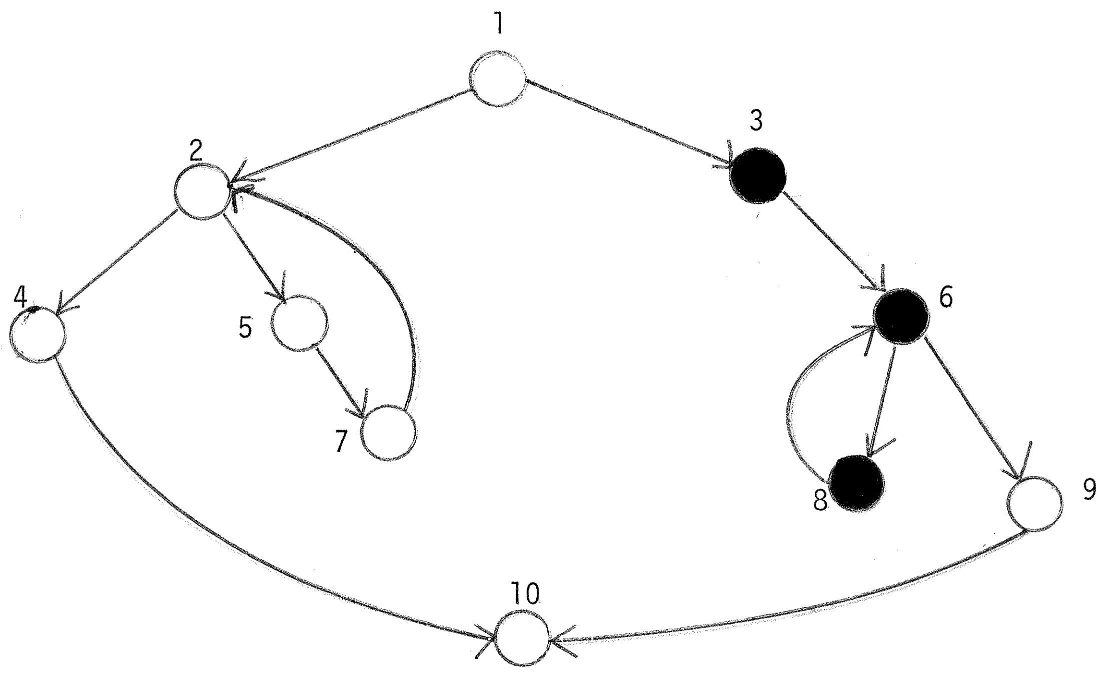


Figure 4f: After execution of line 11: STACK: 3, 6
CURRENTBLOCK: 6
CURRENTROW[3]: 5
CURRENTROW[6]: 9

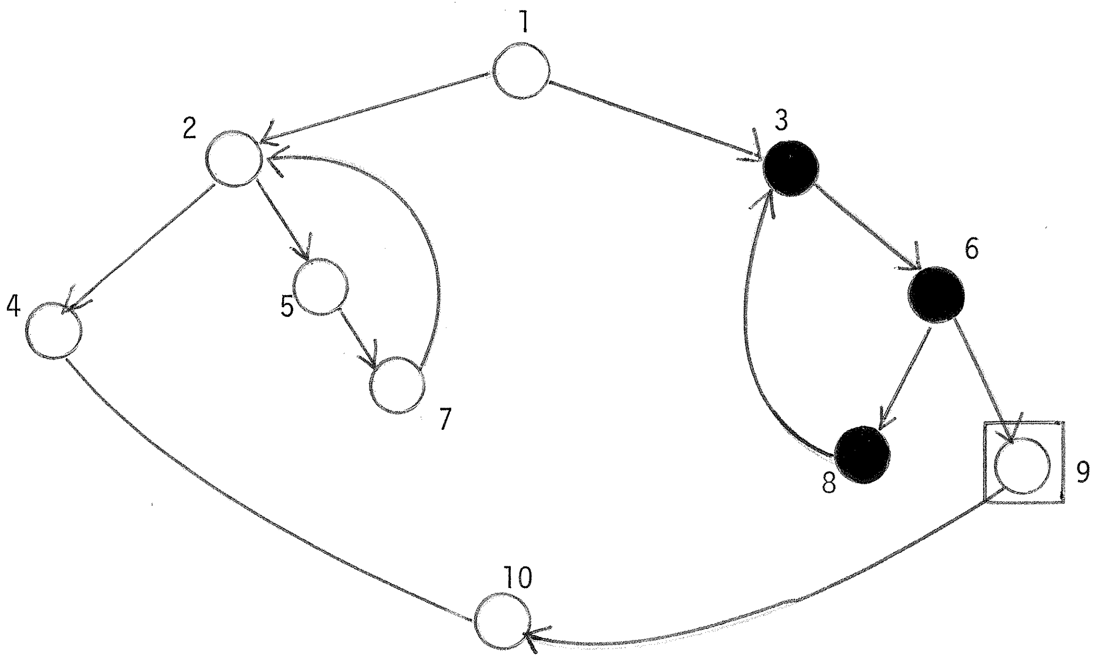


Figure 4g: After the next execution of line 25:
STACK: 3, 6
NEWHEAD: 9
CURRENTBLOCK: 9
CURRENTROW[3]: 5
CURRENTROW[6]: 9

Figure 4a shows the status of the algorithm at the start of execution. The algorithm begins by executing lines 1,2 and 15 through 22, resulting in the status shown in Figure 4b. Then line 23 is executed; procedure NEXTEDGE is invoked and the test at line 5 fails, causing an immediate exit. Line 25 is executed, and the test in line 26 then fails causing a skip to line 32. The status at this point is shown in Figure 4c. The algorithm then loops back up to line 18 executes through line 23, invokes NEXTEDGE, where the test at line 5 fails causing an immediate return to line 25. The test at line 26 again fails, and line 32 is executed next. At this point the status is as shown in Figure 4d. Again there is a loop back to 18, execution sequences down through 23, NEXTEDGE is entered, the test at line 5 fails, returning control to line 24, and line 25 is then executed. At this point the status is as shown in Figure 4e.

Note that the algorithm now threatens to go off on a wild goose chase. Because block 3 is marked, it is pointless to revisit it. All paths out of any marked node either have been explored or are being explored (as is the case here). Thus the algorithm should not go to block 3, but instead should seek an alternate way out of block 8.

The algorithm does in fact detect the futility of going to 3 by noting (line 26) that 3 is already marked. In the WHILE loop beginning at line 26, it seeks an alternate way out of block 8. If there exist alternates, one would have to be on row CURRENTROW[8]+1 (namely row 12) of HEAD. In line 28 this row number is computed. The execution of line 29 then causes an invocation of NEXTEDGE to determine whether the proposed alternate, the edge at CURRENTROW[CURRENTBLOCK] of HEAD, is actually an edge out of CURRENTBLOCK. In this case, the test at line 5 shows that the proposed edge is actually the edge at FIRST[CURRENTBLOCK + 1]. Hence the proposed edge is actually an out-edge from block CURRENTBLOCK + 1, indicating that there are no further edges to be explored out of block 8 (CURRENTBLOCK). The success of the test causes the execution of lines 7 and 8. The test at line 8 fails, and thus the ELSE clause beginning on line 9 is executed. Lines 10 and 11 are executed, resetting CURRENTBLOCK and CURRENTROW[CURRENTBLOCK]. The status of the algorithm now is shown in Figure 4f.

The algorithm has backed out block 8, the blind alley. Note that block 8 <u>remains marked</u>. It is known to be a blind alley, hence if on some future exploration it were to be reached again, the mark would prevent it from being reexplored. Next the end of the WHILE loop begun at line 5 is reached. Now CURRENTROW[CURRENTBLOCK] is 9 and FIRST[CURRENTBLOCK + 1] is 10. Thus the test at line 5 fails and NEXTEDGE is exited. Lines 25 and 26 are then executed; the test at line 26 fails causing a skip to line 32. The status of the algorithm at this point is shown in Figure 4g. Now, upon reaching line 33, the end of the WHILE clause begun on line 18, it is discovered that CURRENTBLOCK=ENDBLOCK. Hence the loop is exited and lines 34 through 37 are executed. The path 3, 6, 9 is printed and the algorithm terminates.

Hopefully this simple example illustrates enough elements of the search procedure to allow the reader to create and follow new, more complex examples. Through such examples insight into the search algorithm should be gained.

Such insight should also convince the reader that at no time does the algorithm ever traverse any edge of the graph more than once. Hence the running speed of the algorithm is $O(e)$, which as noted earlier for program flow graphs is invariably $O(v)$. This is a sizable improvement over the running speeds of comparable algorithms using incidence matrices. (Note, for example, that a single matrix multiplication alone requires $O(v^3)$ time).

In addition, it is worthwhile to observe that any path generated by algorithm P must be a simple path (i.e., must have no repeated edges), and hence can contain no more than v vertices.

Some Applications of the Basic Algorithm To Path Generation Problems

An immediate application of algorithm P is to use it to generate a path from the start node of a program to the stop node. This is easily done by setting BEGINBLOCK to the block number of the start node, ENDBLOCK to the block number of the stop node, and invoking algorithm P. This will yield a complete program test path.

The generation of test paths under more challenging constraints is also possible. It is important to observe that marking a node assures that it will never be explored subsequently, and hence will never be put on the stack and become a member of a resulting path. Hence by arbitrarily marking a node, or set of nodes, before executing P (or even during P's execution) one can assure that those arbitrarily marked nodes will be excluded from the result path.

Thus, for example, suppose a path from block S to block T is sought which does not include any of the blocks $\{B_1, B_2, \ldots, B_i\}$. Such a path is easily found (if it exists) by marking B_1, B_2, ..., B_i , setting BEGINBLOCK $\leftarrow$ S, ENDBLOCK $\leftarrow$ T and invoking algorithm P.

Having dealt with the problem of generating a path which must avoid certain blocks, it is now reasonable to consider the generation of a path which must include a certain block. It is easy to generate a path from S to T which necessarily includes a given block, say A. Such a path necessarily consists of two concatenated subpaths - a path from S to A, and a path from A to T. Hence to find the desired path we execute the following:

```
BEGINBLOCK ← S
ENDBLOCK ← A
invoke  P
BEGINBLOCK ← A
ENDBLOCK ← T
invoke  P
```

The resulting block sequence will, upon deletion of the repeated block A, be the desired path. Clearly the execution speed of this algorithm is O(v) for a flow graph.

It is worth noting that the path generated may not be a simple path. Because algorithm P resets the entire MARK vector to zero as part of its initialization, the path generated from A to T may include blocks which were also included in the path from S to A. Because each subpath is simple, however, the concatenated path will contain each node at most twice. If a simple path from S through A to T is desired, the marks set by the first invocation of P should not be reset to 0 by the second invocation of P. Unfortunately, this procedure may fail to find a simple path even though one may exist. For example in Figure 5, if the first invocation of P were to produce the path S,B,C,D,A then the next invocation of P could find no path from A to T, because C would retain its mark, set by the first invocation of P. Of course, the path S,A,C,T has the desired characteristics. On the other hand, the insistence upon a simple path is probably unwise anyway. Figure 6 shows a flow graph containing a loop. Clearly there is no simple path from S to T through A; yet it is most desirable to be able to create a path from S to T through A. This is easily done by concatenating a simple path from S to A with a simple path from A to T.

The generation of edge constrained paths - paths which must necessarily include or omit specified edges - can be accomplished in similar ways. Perhaps the most straightforward way to produce a path from node S to

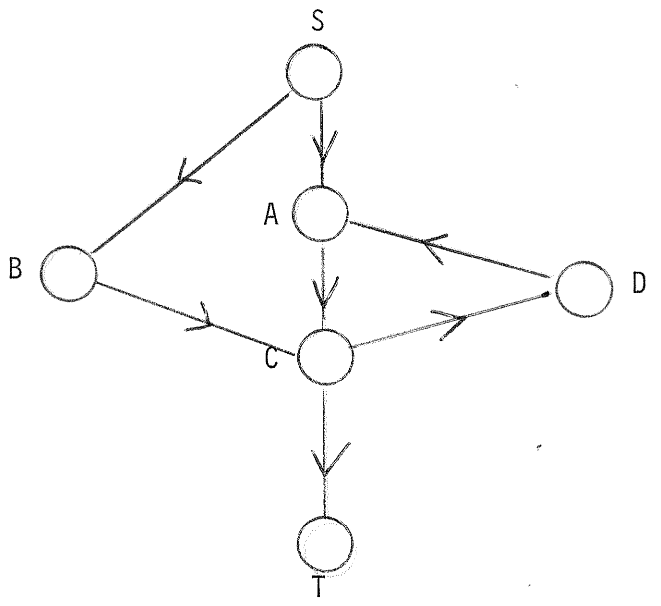


Figure 5: A graph where a simple path from S to T through A cannot always be found by concatenating an arbitrary simple path from S to A with an arbitrary simple path from A to T.

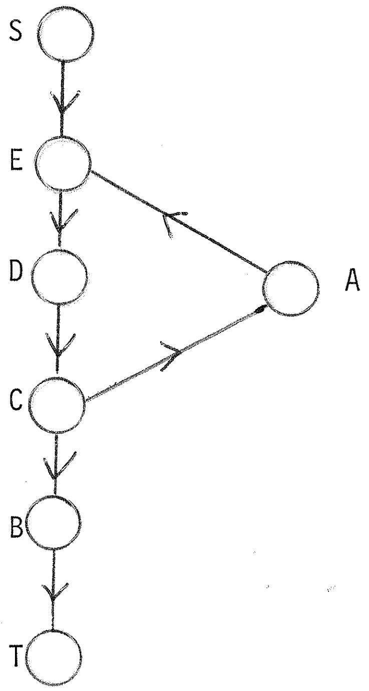


Figure 6: A program flow graph for which there is no simple path from S to T through A.

node T which omits edges e_1, e_2, ..., e_i is to delete the specified edges from the edge-list representation of the graph and invoke Algorithm P. This deletion of edges is easily done when the edge lists are represented by linked lists, but a bit clumsy if the edge lists are represented sequentially.

The generation of a path containing a single specified edge is also straightforward. If a path from S to T which passes through the edge (A,B) is desired, it can be found by the following procedure:

```
BEGINBLOCK ← S
ENDBLOCK ← A
invoke  P
BEGINBLOCK ← B
ENDBLOCK ← T
invoke  P
```

Here again, it is important to observe that the path generated in this way will contain no node more than twice, but may not be simple. In order to assure that any path generated will be simple, it suffices to retain the marks set during the first invocation of P and not reset them before the second invocation. Here too, however, this adapted procedure may be unable to find a simple path even though one may exist. Moreover, by insisting on a simple path, the possibility of generating a path which includes the closing edge of a loop (e.g., a Fortran DO loop) is ruled out. Hence insistence upon simple paths is often unwise.

An alternative approach to creating edge constrained graphs is to first build the edge graph of the original directed graph, and then apply to it the algorithms for creating node-constrained paths which have already been presented.

We construct an edge graph in the following way. Suppose G is a flow graph. To build its edge graph, G_e, first add to G a new start node and an edge directed from it to the original start node. Similarly add to

G a new stop node and an edge directed from the original stop node to the new stop node. Call this new graph G'. The node set of G_e is the edge set of G'. The edge set of G_e consists of all pairs of nodes of G_e (edges of G'), (e_1', e_2'), such that there exists a vertex v' in the vertex set of G' such that $e_1' = (a,v')$ $e_2' = (v',b)$ for some a,b, vertices of G'. As an example, Figure 7 shows the edge graph of the graph in Figure 1. In Figure 7 node a represents edge (1,2), b represents (1,3), c is (2,4),d is (2,5), e is (7,2), f is (8,3), g is (3,6), h is (5,7), i is (6,8), j is (6,9), k is (4,10), m is (9,10), E is the new edge directed to node 1, and F is the new edge directed from node 10.

A path (i.e., node sequence) in G_e corresponds to a sequence of edges in the original graph, G. Hence a node constrained path through G_e corresponds to an edge constrained path through G. Thus for example, in order to find a path from S to T in G which includes the edge (A,B) one can proceed as follows. First produce G_e from G. Suppose (A,B) corresponds to node D in G_e, and that the node S_e in G_e corresponds to any edge in G' which is an in-edge to node S in G', and that the node T_e in G_e corresponds to any edge in G' which is an out-edge from node T in G'. Then:

```
BEGINBLOCK ← $S_e$
ENDBLOCK ← D
invoke P (for the graph $G_e$)
BEGINBLOCK ← D
ENDBLOCK ← $T_e$
invoke P (for the graph $G_e$)
```

produces a list of edges in G'. The repetition of edge D must be removed from this list; then the list can be easily transformed into a list of nodes in G'. Finally, by removing the first and last nodes from this list, the desired path through G is obtained as a sequence of nodes in G.

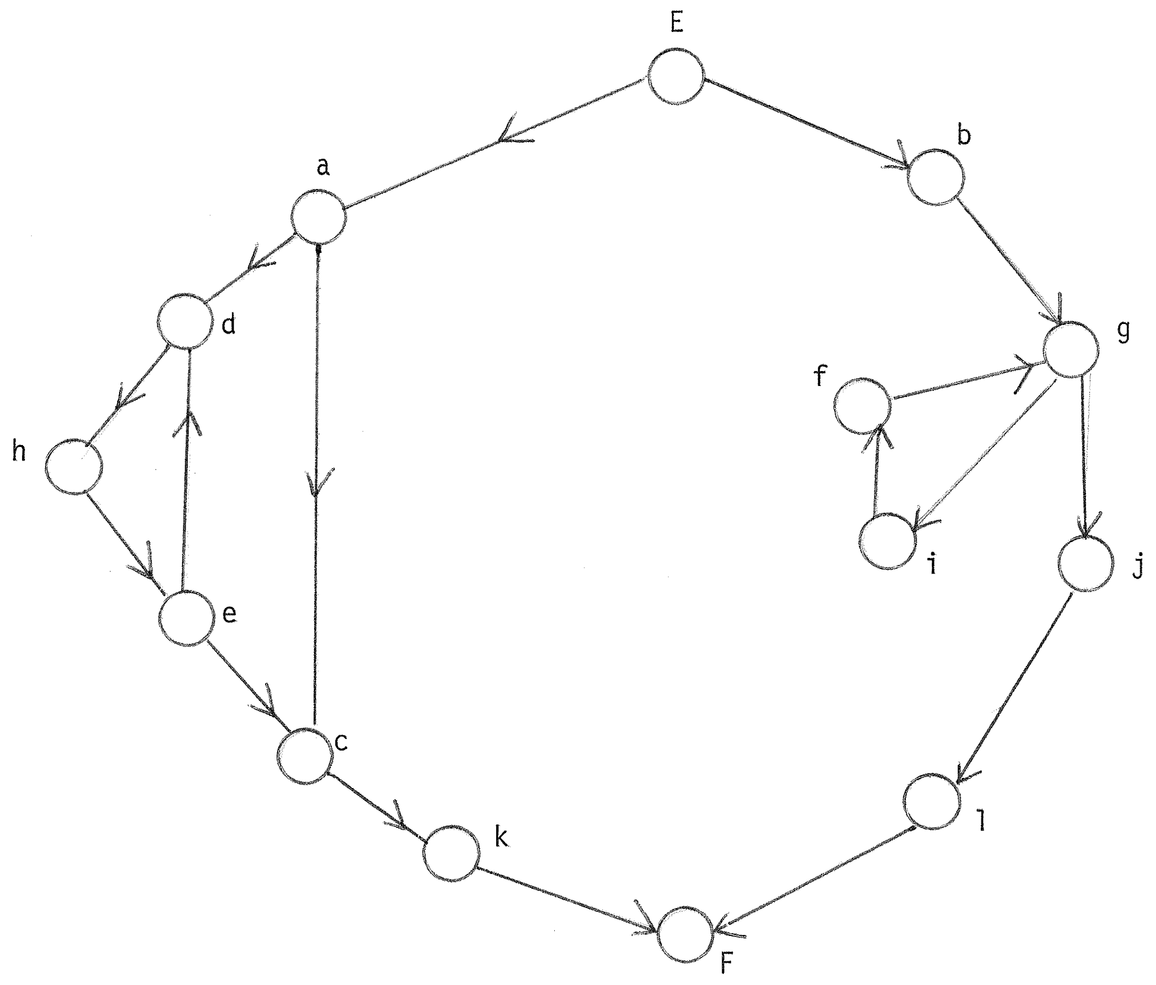


Figure 7: The edge graph of the flow graph in Figure 1.

In closing this section, it seems appropriate to demonstrate that algorithm P can easily be transformed into an algorithm for producing the transitive closure or reachability matrix of a graph. This matrix has proven useful in a wide variety of program analyses.

The reachability matrix of a graph, G, is a (v x v) bit matrix such that the (i, j)th entry of the matrix is a 1 if and only if there is a path in the graph from node i to node j. Hence the ith row of this matrix is a bit vector containing 1's only in those bit positions corresponding to nodes, j, for which there is a path from i to j. Algorithm P is easily altered to produce this bit vector as follows:

1) Change line 18 to

   `WHILE 1=1 DO;`

   (i.e., change line 5 to a "REPEAT FOREVER")

2) Delete lines 34 through 36.

3) Change line 1 to

   `PROCEDURE R(BEGINBLOCK).`

The resulting algorithm, Algorithm R, will STOP at line 8 only after all nodes reachable from BEGINBLOCK have been visited and marked. Hence upon termination of Algorithm R, the vector MARK will be the bit vector required as the BEGINBLOCK th row of the reachability matrix. Thus in order to compute REACH, the reachability matrix of a graph G having v vertices, it suffices to execute:

```
FOR I ← 1 TO V DO
     BEGIN
     invoke R(I);
     FOR J ← 1 TO V DO REACH[I,J]←MARK[J];
     END;
```

Algorithm R executes in $O(v)$ time for a program flow graph because, like Algorithm P, it does not visit any edge of G more than once. Hence it is easy to see that for a program flow graph the reachability matrix can be computed as shown above in $O(v^2)$ time. It should be noted that an algorithm for computing the reachability matrix using incidence matrices was presented by Warshall [12]. Improvements to his algorithm have been subsequently proposed by Purdom [13], and Warren [14]. The execution speed of the search algorithm proposed here seems comparable to these incidence matrix algorithms.

It seems worth noting, however, that the storage required for the search algorithm is $O(v)$ edge representations for a program flow graph, but v^2 bits for the incidence matrix algorithms. Considering that an edge representation is actually a node number, it is clear that an edge representation requires $\lceil \log_2 v \rceil$ bits. Hence the search algorithm requires $O(v \log v)$ bits of storage, a saving over the v^2 bits required by incidence matrix algorithms.

The Generation of Paths as Part of a Comprehensive Program Testing Strategy

Considerable attention has recently been focused on the problem of devising a strategy for thoroughly testing a program (see for example [9], [15], [16]). Although no strategy has yet gained general acceptance, it is nevertheless widely agreed ([9], [15], [17]) that no program can be considered thoroughly tested unless all lines of code have been executed, and that, moreover, each exit from each of a program's branch statements should be executed as well. The problem of hypothesizing a set of program executions which, if carried out, would cause all lines of code to be executed can be approached by first generating a set of paths through the program flow graph which covers all graph nodes, i.e., a path set such that every graph node lies on at least one path. Similarly the problem of exercising all branches is approachable by generating a path set which covers all edges of the flow graph.

It is straightforward to use the results of the previous section to produce algorithms for generating such coverings of the flow graph. For example, suppose a node covering is sought for the graph G, suppose P is a (possibly empty) set of paths through G and U is the set of nodes of G which are not covered by P. We have seen that it is easy to construct a path P_u, which is constrained to pass through any of the nodes $u \in U$. By augmenting P by P_u a better cover is created. By iteratively creating such node constrained paths, U is eventually reduced to the null set and P is augmented to a node covering. An edge covering for G is readily constructed in a similar way by iteratively creating edge constrained paths.

Unfortunately, it is becoming quite clear that the construction of flow graph coverings represents only a modest first step in designing a program test regimen. Test path generation algorithms must take program semantics into account if they are to generate interesting and challenging paths and avoid unexecutable and hence worthless paths.

Some semantic program analysis systems currently under development (e.g., DAVE [18]) seem useful in hypothesizing interesting and revealing program executions. Systems such as DAVE are capable of identifying path segments through programs which seem particularly vulnerable to such errors as uninitialized variables. It is clear that Algorithm P is directly useful in constructing the additional path segments necessary to build such vulnerable segments into full test paths.

Another reasonable approach to generating interesting test executions is to determine combinations of blocks which, if executed as part of a single program execution, seem likely to cause anomalous program behavior. If such block combinations were available, then the remaining problem would be to construct a path through the corresponding program flow graph which is constrained to pass through the specified block set.

The problem of finding such a multiple node constrained path is a logical extension of the single node constrained path problem discussed in the previous section. The solution is more difficult, however. Suppose a path is desired which starts at S, ends at T, and visits all of $\{B_1, B_2, \ldots, B_n\}$. It is easy enough to attempt to find a path from S to B_1, then from B_1 to $B_2, \ldots,$ then from B_n to T, by finding the indicated n+1 subpaths. Unfortunately, such a path may not exist, even though there may still be a path from S to T through $\{B_1, B_2, \ldots, B_n\}$. For example, suppose that we seek a path through the graph in Figure 6 from S to T through {A, B, C, D, E}. No path from S to A to B to C exists. Yet there is clearly a path from S to E to D to C to A to B to T.

An efficient algorithm for finding a multiple node constrained path can be found in [19]. This algorithm can in $O(v)$ time for a flow graph, determine whether or not the desired path exists, and in case the path is found to exist, produce a representation for it. The algorithm requires at worst $O(v^2)$ time to print out the actual path, because in the worst case the length of the path may necessarily be at least $O(v^2)$. No algorithm with a better time bound can be found.

It should also be noted that the path produced by this algorithm may not be simple. The problem of generating a simple multiple node constrained path is far harder and is easily shown to be equivalent to the Hamiltonian Path Problem, which is an NP-Complete problem. Hence no efficient algorithm for generating a simple multiple node constrained path is known, and the problem is recognized to be equivalent to a host of celebrated problems for which no efficient algorithms are known either [20].

Some work has also been done on the problem of suppressing the generation of unexecutable paths. Smith, Krause, and Goodwin have proposed [9] that it is sometimes possible to identify mutually unexecutable pairs of edges of a program's flow graph due to semantic incompatibilities. (Recent work by Clarke [21] offers hope that this identification can usually be made quite reliably.) They propose that it is unreasonable to generate any test path containing both edges of such an impossible pair. Hence it is interesting and useful to consider the problem of generating a test path which does not anywhere contain both of the edges named in any of a set of impossible pairs, where the set of impossible pairs has been specified before the algorithm begins.

A straightforward solution based upon algorithm P is possible but inefficient. Suppose there are t impossible pairs, (r_1, s_1), (r_2, s_2), ..., (r_t, s_t), and suppose that the t pairs are partitioned into two sets, E and F, of cardinality ℓ and $t-\ell$ respectively, where we shall denote $E = \{(r_{e_i}, s_{e_i})\}_{i=1}^{\ell}$ and $F = \{(r_{f_j}, s_{f_j})\}_{j=1}^{t-\ell}$. Using Algorithm P it is possible in $O(v)$ time to either build, or disprove the existence of, a path from the start of G to its stop which excludes $r_{e_1}, r_{e_2}, \ldots, r_{e_\ell}$ and $s_{f_1}, s_{f_2}, \ldots, s_{f_{t-\ell}}$. This is done by removing from the flow graph those edges which are to be excluded and then invoking Algorithm P. If this is done for all of the 2^t possible combinations of impossible pairs then either an impossible pairs constrained path will have been found, or the existence of such a path will have been disproven. Unfortunately, this algorithm requires at worst 2^t invocations of algorithm P, and hence takes $O(2^t \cdot v)$ time.

It has recently been shown however [19] that the problem of generating an impossible pairs constrained path is also NP-Complete. Hence, here too, no algorithm significantly faster than the one just given is available.

CONCLUSIONS

It is hoped that this exposition has convinced the reader that depth first search techniques provide a flexible and efficient tool for test path creation. The applications presented here should be construed as examples designed to illustrate the flexibility of this technique. The examples were selected, however, from the author's observations of the kinds of paths in which current investigators seem interested.

This paper does not address the broader and more important question of how one should go about creating path constraints which assure that a program is adequately tested. All that is presented is a mechanism for synthesizing paths which are consistent with such given constraints.

References

[1] Ramamoorthy, C.V., Analysis of Graphs by Connectivity Considerations, JACM 2 (April 1966), pp. 211-222.

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

[3] Hopcroft, John and Robert Tarjan, Efficient Algorithms for Graph Manipulation, CACM 16 (June 1973), pp. 372-378.

[4] Johnson, Donald B., Finding All the Elementary Circuits of a Directed Graph, SIAM J. Computing 4, (March 1975), pp. 77-84.

[5] Tarjan, R., Depth First Search and Linear Graph Algorithms, SIAM J. Computing 1 (June 1972), pp. 146-160.

[6] Ullman, J.D., Fast Algorithms for the Elimination of Common Subexpressions, Acta Informatica 2 (December 1973), pp. 191-213.

[7] Allen, F.E., A Basis for Program Optimization, Proc. IFIP Conf., 71, Amsterdam: North Holland 1972, pp. 385-390.

[8] Tarjan, R., An Efficient Planarity Algorithm, Rep. STAN-CS-244-71, Computer Science Department, Stanford University, Stanford California, 1971.

[9] Krause, K.A., R.W. Smith and M.A. Goodwin, Optimal Software Test Planning Through Auotmated Network Analysis, 1973 IEEE Symposium on Computer Software Reliability, IEEE #73 C40741-9CSR, New York, pp. 18-22.

[10] Shirey, R.W., Implementation and Analysis of Efficient Graph Planarity Testing, Ph.D. Dissertation, Computer Science Department, University of Wisconsin, Madison, Wisconsin, 1969.

[11] Knuth, Donald E., The Art of Computer Programming, vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass, 1968.

[12] Warshall, Stephen, A Theorem on Boolean Matrices, JACM 1 (January 1962), pp. 11-12.

[13] Purdom, Paul, Jr., A Transitive Closrue Algorithm, BIT 10 (1970) pp. 76-94.

[14] Warren, Henry S., Jr., A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations, CACM 18 (April 1975), pp. 218-220.

[15] Goodenough, John B., and Susan L. Gerhart, Toward a Theory of Test Data Selection, 1975 International Conference on Reliable Software, SIGPLAN Notices 10 (June 1975), pp. 493-510.

[16] Howden, William, Methodology for the Generation of Program Test Data, IEEE Transactions on Computers, C-24 (May 1975),pp.554-560.

[17] Boehm, B., Software and its Impact: A Quantitative Assessment, Datamation 19 (May 1973), pp. 48-59.

[18] Osterweil, L.J., and Fosdick, L.D.,Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Dectection, Department of Computer Science, University of Colorado, Technical Report #CU-CS-055-74, September 1974.

[19] Gabow, H.N., Maheshwari, S., and Osterweil, L.J., Some Results in the Construction of Constrained Program Test Paths, Department of Computer Science, University of Colorado, Technical Report (to appear).

[20] Karp, R.M., Reducibility Among Combinatorial Problems, in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher, Eds., Plenum, New York, 1972, pp. 85-103.

[21] Clarke, Lori, A System to Generate Test Data and Symbolically Execute Programs, Department of Computer Science, University of Colorado, Technical Report #CU-CS-060-75 (February, 1975).