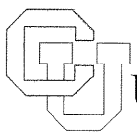
On Two Problems in the Generation of Program Test Paths *

Harold N. Gabow
Sachindra N. Maheshwari
Leon J. Osterweil

CU-CS-081-75



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work supported by NSF Grant DCR75-90072

On Two Problems in the Generation of Program Test Paths *

by

Harold N. Gabow
Sachindra N. Maheshwari
Leon J. Osterweil

Department of Computer Science
University of Colorado
Boulder, Colorado 80302

TR #CU-CS-081-75 September 1975

* This work supported by NSF Grant DCR75-90072

ABSTRACT

In this paper we analyze the complexity of algorithms for two problems that arise in automatic test path generation for programs: the problem of building a path through a specified set of flow graph nodes and the problem of building a path which satisfies impossible-pairs restrictions in a flow graph. We give a highly efficient algorithm for the first problem, and show that the second problem is NP-complete in the sense of Cook and Karp.

INTRODUCTION

The ability to construct constrained paths through a program is useful in program testing and validation. It is clear, for example, that if it is possible to construct a path which is constrained to pass through a given arbitrary basic block of a program, then this capability can be used to build a set of test paths such that every block (and thus every statement) of the program is covered (i.e. visited) by at least one test path. Moreoever, this capability can, with minor modification, be used to construct a set of test paths which covers every exit from every branching statement. The desirability of such test path sets seems well agreed upon ([1],[2],[3]), and discussions of how to construct them can be found in the literature ([1],[4],[5],[6]).

Criticism of such schemes for building test path sets can be levelled in at least two ways. First, while the test path set may cover all lines of code and all branches, no effort is made to insure that any single path covers a potentially revealing or vulnerable combination of blocks or branches. Hence the test path set may be bland and unchallenging. Second, no effort is made to use program semantics to suppress the generation of paths which are unexecutable, and hence useless in any testing procedure.

In response to the first criticism, useful criteria for determining how to select revealing and troublesome combinations of blocks are now being sought ([7], [8], [9]), and it seems reasonable to expect that automated systems for doing this may be available before long ([8], [9]). The notion of impossible pairs, has been advanced by Krause, Smith, and Goodwin [1] as a reasonable step towards meeting the second criticism. They suggest that pairs of mutually unexecutable edges in a program be iden-

tified through analysis of program branching conditions, and that every test path generated be constrained to contain no more than one edge of each impossible pair. Recent program analysis systems ([7], [9], [10]) offer hope that non-trivial impossible pair sets can be automatically generated.

In this paper we assume the existence of program analysis schemes for generating troublesome combination conditions and impossible pair conditions. We address only the graph analytic problems of generating paths constrained by these conditions. Our goal is to determine which of these problems are combinatorially tractable, and hence which types of program analysis seem worthy of continued effort.

We make the conventional assumption that a program is represented by a program flow graph - a digraph in which each node represents a basic block of the program and in which the edges represent possible transfers of control. Hence the paths which we seek are flow graph paths in the usual graph theoretic sense. In particular a path may be a non-simple path, i.e. it may contain repeated nodes. We assume that V is the flow graph's node set, E is its edge set, that it has a single start or source node s, and a single stop or sink node t. Definitions of other graph theoretic terminology used here can be found in [11].

THE GENERATION OF MULTIPLE NODE CONSTRAINED PATHS

In this section we discuss the multiple node constrained path problem. Given a flow graph G, with source s and sink t, the problem is to either find a path from s to t which passes through a specified subset V* of V, or show that no such path exists. We solve the problem by first reducing G to an acyclic equivalent and then determining a multiple node constrained path through this simpler structure.

For the flow graph G let $G_1, G_2, \ldots, G_k$ be the strongly connected components (SCC) of G (i.e. each G_i is a maximal subgraph of G such that any node of G_i is reachable from any other node in G_i by a path in G_i). Now consider the digraph $G_r = (V_r, E_r)$ for which each node in the node set V_r represents a SCC of G, and (v_i, v_j) is in the edge set E_r if and only if there is an edge in G directed from some node in the SCC corresponding to v_i to some node in the SCC corresponding to v_j. We call G_r the reduced graph of G. It is easy to see that G_r is acyclic, and has a source s_r corresponding to s and a sink t_r corresponding to t. Moreover, because each node in G corresponds to a unique node in G_r there is a correspondence between the paths in G and the paths in G_r. Now denote by V_r^* the subset of V_r which corresponds to V* in V. Clearly a path in G from s to t through all nodes of V* exists if and only if there exists a path in G_r from s_r to t_r which passes through all nodes of V^*_r.

Hence the problem of determining the existence of a multiple node constrained path in an arbitrary digraph is reduced to the problem of determining the existence of such a path in an acyclic digraph. In the process of solving this latter problem an actual path in the acyclic digraph will be produced if one exists. It will be shown that the

desired path in the original digraph is easily generated from this path in the acyclic digraph.

We will now present an efficient algorithm for finding a multiple node constrained path (if one exists) through a single source, single sink acyclic digraph. Suppose G is such an acyclic diagraph, s and t are its source and sink respectively, and V* is a subset of nodes of G through which the desired path must go. The algorithm relies upon the following observations. First, the desired path exists if and only if for all pairs of nodes v_i, $v_j \in V^* \cup \{s,t\}$, either v_i is reachable from v_j, or v_j is reachable from v_i. A node pair is said to be unrelated if neither is reachable from the other. Thus, for example, any pair of source nodes in an arbitrary acyclic diagraph is unrelated. Second, the deletion of a source and all edges emanating from it in an acyclic digraph results in another acyclic digraph. This process can be continued until the trivial graph with one node results. This successive removal of sources creates a sequence of node-deleted subdigraphs, some of which may have multiple sources. The order in which nodes are deleted can, as shall be shown, be contrived to assure that if there are two unrelated nodes in V*, then there will be a subdigraph in the node-deleted sequence for which some pair of sources will both be members of V*. This condition is easily tested for, and hence the algorithm is readily usable to disprove the existence of a multiple node constrained path. This successive reduction of the acyclic digraph, ultimately to a single node can, moreover, be viewed as a traversal of the digraph, where the traversal is the order in which the nodes are deleted. This traversal easily yields the desired multiple node constrained path through G.

The following lemma leads directly to the algorithm:

Lemma 1. If u and v are a pair of unrelated nodes in an acyclic digraph G, then there exists a traversal in which at some stage both u and v are sources in the remaining node-deleted subdigraph.

Proof. Without loss of generality consider a traversal in which, after nodes $v_{i_1}, v_{i_2}, \ldots, v_{i_r}$ are traversed, the subdigraph of undeleted nodes and edges contains both u and v, and has u as a source. If v is also a source then we are done. Therefore assume that v is not a source. Since u and v are unrelated, and v is not a source there must be another source in the subdigraph, say $v_{i_{r+1}}$, from which v is reachable. Let $v_{i_{r+1}}$ be the next node to be traversed. In the resultant untraversed subdigraph u will continue to be a source, and if v is not a source then the same argument can be used to choose other nodes to traverse until both u and v become sources. □

The algorithm for determining the existence of a multiple node constrained path works just as the proof of the lemma, namely by traversing a nonconstraining source in a node-deleted subdigraph first, whenever a choice exists. It will be shown that the successful traversal of G implies the existence of a path from s to t that passes through all of the nodes of V*. The information about the path itself is built in the vector PATH, a rooted father tree (see [12] for the definition of this term) which shall be called the traversal tree. Initially the source of G is made the root of the traversal tree. At each successive stage of traversal any new source created is added to the tree by linking it to the node most recently traversed. The algorithm uses sets SET1 and SET2 to store those sources of the node-deleted subdigraph which are nonconstraining and constraining nodes respectively.

The information about the indegrees of nodes is stored in the array INDEG. For the purposes of the following algorithm, $V^* \cup \{s,t\}$ is considered to be the set of constraining nodes.

```
procedure CONSTRAINTPATH;
        begin
          SET1←{s};
          SET2← φ;
A.  TRAVERSE:  while SET1 is non-empty and |SET2| ≤ 1 do
            begin
              delete a node x from SET1;
              for each node w in the adjacency list of x do
                begin
                  INDEG(w) ← INDEG(w) - 1;
                  if INDEG(w) = 0 then
                    begin
                      PATH(w) ← x;
                      if w is a constraining node then add w to SET2
                      else add w to SET1;
                    end
                end
            end
B.        if |SET2| > 1 then output "no path exists"
          else if t ∉ SET2 then
            begin
              delete the node from SET2 and add it to SET1;
              go to TRAVERSE;
            end
          else output PATH;
        end;
```

The following lemma shows how a multiple node constrained path through G can be produced from PATH.

Lemma 2: If CONSTRAINTPATH terminates by placing t into SET2, then the reverse of the node sequence t, PATH(t), PATH(PATH(t)), ..., s, denoted by $P = p_1, p_2, \ldots, p_\ell$, is a path in G which contains all the nodes in V*.

Proof: The proof is by induction on the depth of the traversal tree, depth of a tree being defined as the length of the longest path from the root to a leaf. The claim is trivially true for traversal trees of depth 1. Assume it is true for all traversal trees of depth $k \leq n$. Now consider a digraph for which CONSTRAINTPATH terminates by placing t in SET2, and the depth of the traversal tree represented by PATH is n+1. Let u be the first constraining node after s to be added to SET2 by CONSTRAINTPATH. From lines A and B, and Lemma 1 it follows that u is the only source of the node deleted subdigraph G_u when u is traversed. Also, since CONSTRAINT-PATH stops with t in SET2, V* is a subset of the node set of G_u. Therefore, under the condition that V* - {u} are the contraining nodes for G_u, the subtree rooted at u in the traversal tree represented by PATH is the traversal tree for G_u. Since the depth of this subtree is not greater than n, it follows from the induction hypothesis that the reverse of the node sequence t, PATH(t), PATH(PATH(t)), ... u is a path in G_u which contains all the nodes in V* - {u}. Therefore the reverse of the node sequence t, PATH(t), ..., u, PATH(u), ...,s is a path in G which contains all the nodes in V*. □

It is easy to see that CONSTRAINTPATH requires only $O(|E|)$ (where $|E|$ denotes the cardinality of set E) steps to build the traversal tree. This is because CONSTRAINTPATH involves essentially the deletion, one by one, of the edges and nodes of G, and creation, node by node, of PATH.

In order to determine whether a multiple node constrained path exists in a flowgraph G=(V,E), the reduced graph $G_r=(V_r,E_r)$ must be constructed and CONSTRAINTPATH must then be run on G_r. G_r can be constructed from flowgraph G=(V,E) in $O(|E|)$ steps by using an algorithm of Tarjan [13]. As stated above CONSTRAINTPATH takes $O(|E_r|)$ steps to build the entire traversal tree of G_r. Hence, since $|E_r| \leq |E|$, the existence of a multiple node constrained path can be determined in $O(|E|)$ steps. It is often assumed (see [14], for example) that $|E| = O(|V|)$ for a program flow graph. Hence existence can be determined in $O(|V|)$ steps for a program flow graph.

If such a path exists, then its representation in G_r is easily obtained in $O(|V_r|)$ steps by determining P as in the proof of Lemma 2. In order to construct a multiple node constrained path in G, P is first expanded by replacing each p_i, $1 \leq i \leq \ell$, by any arbitrary permutation of the vertices in the SCC of G corresponding to p_i. This results in a sequence of nodes in G such that each consecutive pair of the sequence can be connected by a simple path in G, thereby forming the desired multiple node constrained path in G. In this sense, this sequence is a representation of the desired path. This sequence in G can be obtained in $O(|V|)$ steps for a program flow graph.

In order to obtain the actual path in G, a depth first search can be used to build subpaths between each pair of adjacent nodes in the representation. Each subpath can be computed in $O(|V|)$ steps. Since there may be as many as $|V|-1$ subpaths to be determined, the total effort is bounded by $O(|V|^2)$. This bound holds for all digraphs, regardless of whether $|E| = O(|V|)$ or not.

It should be noted that the path produced by this algorithm may not be simple. It is not hard to see that the problem of producing a simple multiple node constrained path is equivalent to the problem of finding a hamiltonian path and thus is an NP-complete problem [15].

It should also be noted that a digraph G and a constrained node set V* can be constructed which force the required path to have $O(|V|^2)$ length (see figure 1). Hence if the process of listing a node is defined to require one unit of effort, then no algorithm for listing a multiple node constrained path can have complexity less than the one given here.

It should be stressed, however, that the method outlined here can be used to determine a useful representation of such a path (if a path exists) with $O(|V|)$ effort, for a program flow graph. The work of Clarke [16] seems to indicate, moreover, that such a representation may be more useful than a completely specified path in an actual program testing situation. This is because a mechanical subpath generator is inherently incapable of producing subpaths which are known to be executable. Preliminary work by Clarke, however, offers hope that symbolic execution methods can be used to build an executable path between two given flow graph nodes.

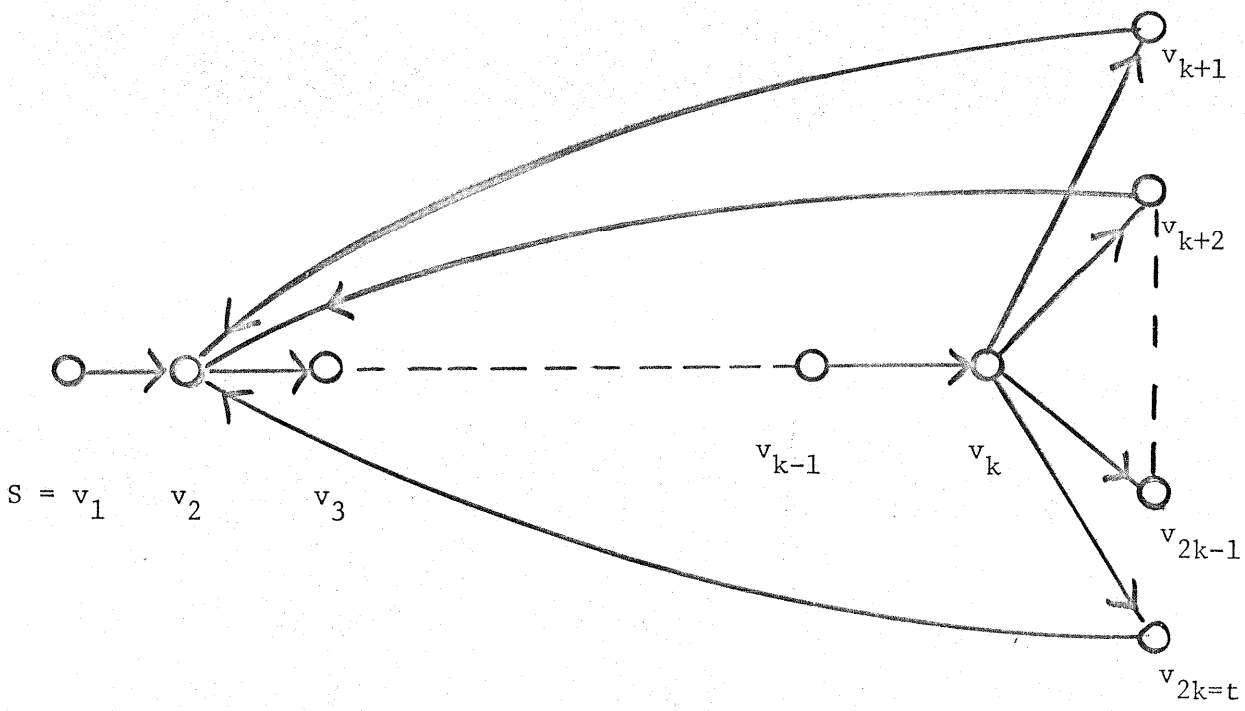


Figure 1: Assume $|V| = 2k$. If the constrained node set $V^* = \{v_{k+1}, v_{k+2}, \ldots, v_{2k-1}\}$, then the length of the path from s to t that passes through the vertices in V^* must be at least $O(k^2)$.

THE CONSTRUCTION OF IMPOSSIBLE PAIRS CONSTRAINED PATHS

In this section we discuss the impossible pairs constrained path problem. We show that the existential impossible pairs problem--i.e., the problem of determining the existence of an impossible pairs constrained path--is NP-complete. We show, moreover, that the problem is NP-complete even when the graph is acyclic and all indegrees and outdegrees are bounded by two.

We begin by describing a straightforward algorithm for constructing an impossible pairs constrained path, if one exists. Suppose G is a flow graph with node set V and edge set E. We solve the problem of finding a path from s to t containing at most one node from each pair of nodes (a_i,b_i) in the set

$$M=\{(a_i,b_i)\}_{i=1}^{n}$$

It can be easily shown that this problem is equivalent to the problem stated in [1], in which the impossible pairs are edge pairs.

Let $I=\{1,2,\ldots,n\}$ be the set of indices of M.

```
procedure DETPATH;
    begin for all subsets J of I do
        begin form the set S={a_i | i∈J}∪{b_i | i∉J};
            form the graph G' by deleting from G the nodes in S and
            their incident edges;
            if a path from s to t exists in G' then go to Y;
        end;
    output "path does not exist"; go to DONE;
    Y:  output "path exists";
    DONE:  end;
```

The existence of a path from s to t can be determined in $O(|V|)$ time (if G is a flow graph) by using a depth-first search. Hence the generation and searching of each of the 2^n graphs formed in DETPATH requires $O(|V|)$ time. Hence this algorithm requires $O(|V|*2^n)$ time. (Another exhaustive search algorithm, requiring $O(|V|^{|V|})$ time, is easily constructed.)

Note that the above algorithm can be thought of as a non-deterministic polynomial time bounded algorithm, because once the correct choice for J has been made, finding the desired path requires only $O(|V|)$ time. Thus the impossible pairs problem is in NP.

In order to finish the proof of the NP-completeness, it is shown that an NP-complete problem is transformable in polynomial time to the existential impossible pairs problem. The satisfiability problem for a Boolean expression in conjunctive normal form is used for this.

A Boolean expression B is in conjunctive normal form if it is written as

$$B=(p_{11}\vee p_{12}\vee\ldots p_{1n_1})\wedge(p_{21}\vee p_{22}\ldots\vee p_{2n_2})\wedge\ldots\ (p_{m1}\vee p_{m2}\vee\ldots\vee p_{mn_m})$$

where each p_{ij} represents either a Boolean variable, x_k, or the negation of a Boolean variable, $\bar{x}_k$. B is <u>satisfiable</u> if the x_k can be assigned Boolean values so B is true. The problem of determining the satisfiability of an arbitrary expression in conjunctive normal form is NP-complete [15].

Lemma 3: The satisfiability problem is polynomially transformable into the existential impossible pairs problem.

Proof: Using the given Boolean expression B, a digraph G_B is constructed. V_B, the node set of G_B, consists of a source s, a sink t and a node v_{ij} for each literal p_{ij} in B. E_B, the edge set of G_B, is the set of pairs of vertices corresponding to pairs of literals in consecutive disjuncts of B, where s is considered the first disjunct of B and t the last disjunct in B. Thus

$$E_B=\{(s,v_{1j}) \mid 1\leq j\leq n_1\}$$
$$\cup\{v_{ij},v_{i+1,k}) \mid 1\leq i<m, 1\leq j\leq n_i, 1\leq k\leq n_{i+1}\}$$
$$\cup\{(v_{mj},t) \mid 1\leq j\leq n_m\}$$

An illustration of this construction is given in Figure 2.

M_B, the impossible pairs list for G_B, is the set of all pairs of nodes representing pairs of literals which are negations of each other. Thus

$$M_B=\{(v_{ij},v_{k\ell}) \mid p_{ij}=\overline{p_{k\ell}}\}$$

Hence, for the example illustrated in Figure 2,

$$M_B=\{(v_{11}, v_{21}),(v_{12},v_{31}),(v_{22},v_{31}),(v_{23},v_{32})\}$$

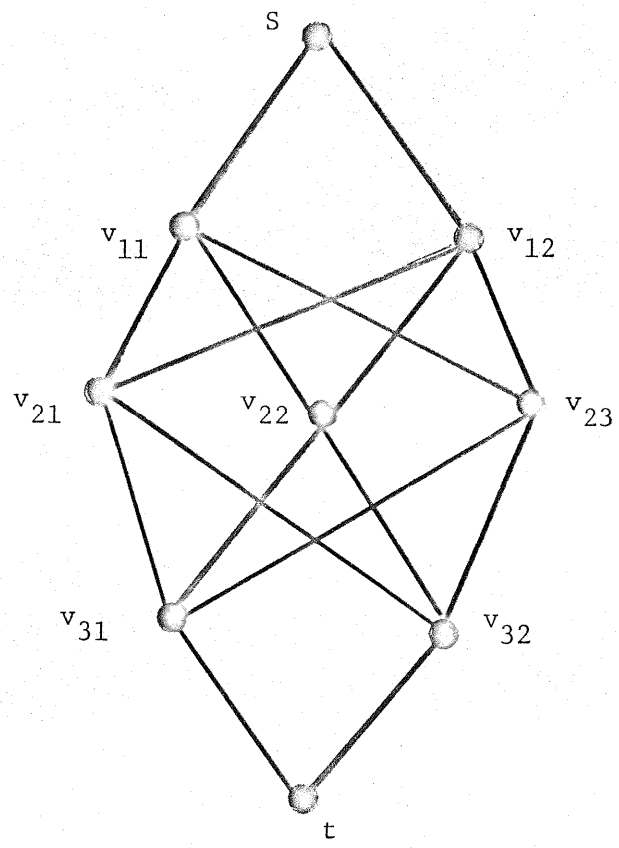


Fig. 2: The graph G_B constructed from the Boolean expression $B=(p\vee\bar{q})\wedge(\bar{p}\vee q\vee\bar{r})\wedge(q\vee r)$.

It is easy to verify that if a constrained path exists in G_B, then B is satisfiable. Let $P_B=\{s,v_{i\ell_1},v_{2\ell_2},\ldots,v_{m\ell_m},t\}$ be such a constrained path. By making $p_{1\ell_1}$, $p_{2\ell_2}$,... and $p_{m\ell_m}$ all true in B, the value of B is true. Because P_B is impossible pairs constrained, moreover, these requirements are guaranteed to be non-conflicting. Hence the existence of P_B implies the satisfiability of B.

Conversely if B is satisfiable, then there exists a sequence

$$\{\ell_1\ell_2\ldots,\ell_m\}$$

such that $p_{1\ell_1}$, $p_{2\ell_2}$, ..., and $p_{m\ell_m}$ are all true in B. Clearly the assignment of true to all of these literals cannot result in a conflicting definition of any Boolean variable, x_k. Hence s, $v_{1\ell_1}$, $v_{2\ell_2}$, ..., $v_{m\ell_m}$, t is a path through G_B which is impossible pairs constrained. Thus the satisfiability of B implies the existence of P_B. □

It is clear that G_B and M_B can be created from B in polynomial time. Thus the problem of determining the existence of an impossible pairs constrained path in a digraph is NP-complete.

It is interesting to note that G_B is acyclic for all B, implying that the problem of determining the existence of an impossible-pairs constrained path in an acyclic digraph is NP-complete.

It is not hard to see, moreover, that this result holds for acyclic digraphs for which all vertices have indegree and outdegree less than or equal to two. The demonstration of this is the same as the previous demonstration, except that now, in place of G_B, a G'_B must be constructed which satisfies the above degree constraints. In order to build G'_B, first arrange the nodes of V_B, the node set of G_B, in m+2 rows, each row corresponding to the literals of a single disjunct of B; the edges of G_B make

consecutive rows of V_B into complete bipartite graphs. G'_B is constructed by introducing new nodes between the rows of G_B, and altering the edge set of G_B, so the following two properties hold. First, any two nodes in consecutive rows of V_B are joined by a path through new nodes. Second, all nodes have the desired degree constraints. This construction can always be done in polynomial time.

Fig. 3 shows how G'_B can be constructed from G_B in Fig. 2. It is now easy to see that the problem of determining the satisfiability of a Boolean expression in conjunctive normal form is transformable to the impossible pairs problem for G'_B and M_B.

We have thus shown that the problem of finding impossible pairs constrained paths is NP-complete for a highly restricted class of graphs. Clearly this restricted class is included in the class of program flow graphs, even assuming that programs are written with highly disciplined use of control flow structures. Hence this result is directly applicable to the problem of finding paths in real-world programs.

It should be noted in closing, however, that the impossible pairs problem is not NP-complete for the very restricted, yet important, class of program flow graphs which are trees. A linear algorithm is constructed as follows. Let T be a flow graph which is a tree. Form G=(V,E) from T by adding a sink node, t, to the node set of T and by adding, for each leaf of T, an edge directed from the leaf to t. Use a depth-first search to number the nodes of T in such a way that all descendants of a node are consecutively numbered, and each node is labelled with the range of numbers borne by its descendants. This can be done in $O(|E|)$ time [13]. For each impossible pair use the node numbering to determine whether one node of the pair is a descendent of the other. If so, remove the single in-edge

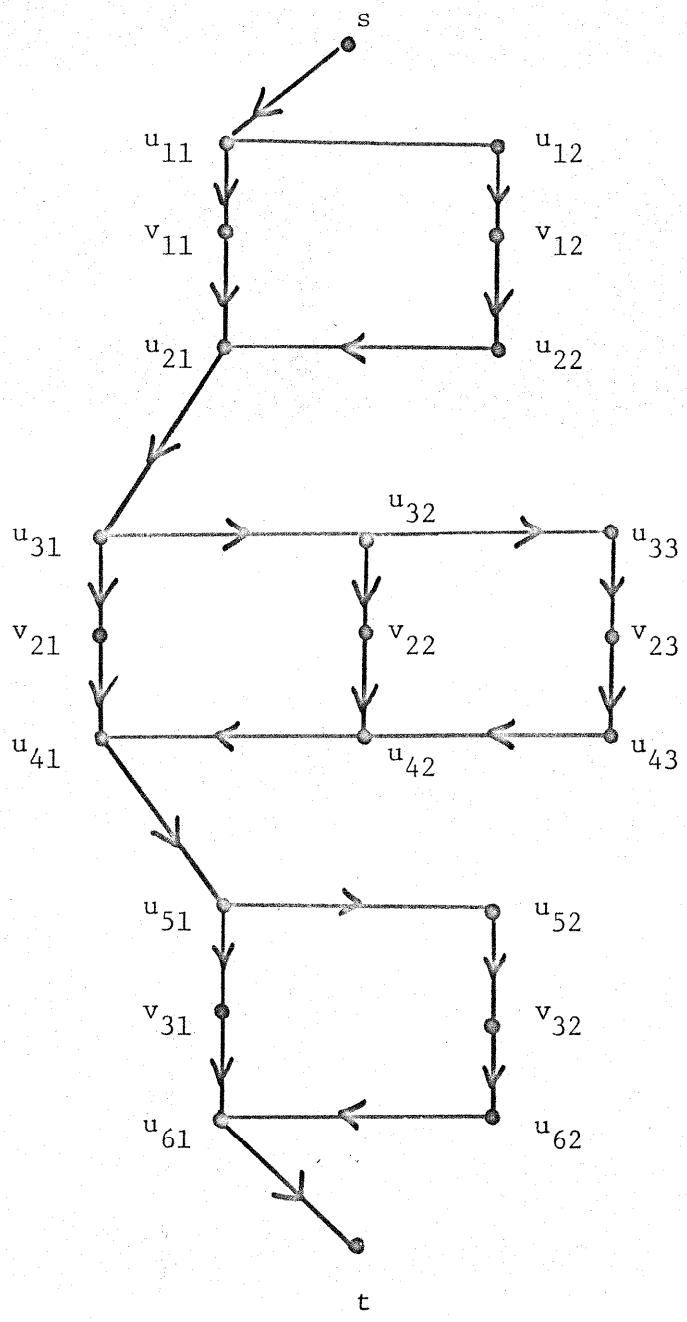


Fig. 3: The graph G_B' constructed from the graph G_B in Figure 2. For each row of nodes v_{ij} in G_B, two rows of nodes $u_{2i-1,j}$ and $u_{2i,j}$ have been added.

into the descendent node from the edge set of G. This determination and deletion (if necessary) requires constant time for one pair, hence $O(n)$ time for all pairs. Finally use a depth first-search to attempt to build a path from s to t in G. This search requires $O(|E|)$ time. Any path built is an impossible pairs constrained path. If no such path exists then there is no impossible pairs constrained path. Hence the impossible pairs problem can be solved in $O(\max(n, |E|))$ time for the class of flow graphs which are trees.

CONCLUSIONS

We have analyzed two different schemes for generating test paths through programs. For the first scheme, in which a path through a specified set of nodes is desired, we were able to produce a highly efficient algorithm. It appears that it will be quite useful and rewarding to press forward efforts in the automatic generation of interesting combinations of program blocks. The second scheme, involves finding a path which never includes both nodes of any of a given set of node pairs. Our results are unencouraging for this scheme. Even if the program graph is acyclic and even if no node has more than two inedges or outedges, the best known algorithm for finding such a path is exponential in the number of impossible pairs. Hence it appears that it is reasonable to attempt the generation of impossible pairs constrained paths only if the number of pairs is small or if good heuristics can be discovered.

References

1 Krause, K.A., Smith, R.W., and Goodwin, M.A. "Optimal Software Test Planning Through Automated Network Analysis," 1973 IEEE Symposium on Computer Software Reliability, IEEE #73 C 40741-9CSR, New York, pp. 18-22.

2 Ramamoorthy, C.V., and Ho, S.-B.F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Transaction on Software Engineering, SE-1, (March 1975), pp. 46-58.

3 Boehm, B. "Software and Its Impact: A Quantitative Assessment," Datamation 19 (May 1973) pp. 48-59.

4 Osterweil, L., "Depth First Search Techniques and Efficient Methods for Creating Test Paths," University of Colorado Dept. of Computer Sci. Technical Report #CU-CS-077-75 (August 1975).

5 Hoffman, R.H. and Smith, R.W., "Advanced Techniques in the Generation of Connectivity Matrices for Software Network Analysis," TRW Systems Group, Houston, Texas, 1975.

6 Huang, J.C., "Program Testing," University of Houston, Dept. of Computer Science, Houston, Texas, May 1974.

7 Clarke, L., "A System to Generate Test Data and Symbolically Execute Programs," University of Colorado Dept. of Computer Sci. Technical Report #CU-CS-060-75, (February 1975)

8 Howden, W., "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, C-24 (May 1975), pp. 554-560.

9 Osterweil, L., and Fosdick, L.D. "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation and Error Detection," University of Colorado, Dept. of Computer Sci. Technical Report #CU-CS-055-74 (September 1974).

10 King, J.C., "A New Approach to Program Testing," 1975 International Conference on Reliable Software, IEEE Cat. No. 75 CHO 940-7CSR pp. 228-233.

11 Harary, F., Graph Theory, Addison-Wesley, Reading, Mass., 1969.

12 Knuth, D.E., The Art of Computer Programming, v.1, Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

13 Tarjan, R.E., "Depth First Search and Linear Graph Algorithms," SIAM Journal on Computing, **1** (June 1972) pp. 146-160.

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

15 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.

References cont'd.

16 Clarke, L. "Automatic Generation of Test Data for Computer Programs," Ph.D. Dissertation, Dept. of Computer Sci. University of Colorado, Boulder, Colorado (to appear).