BIT 11 (1971), 94–106

A NEW DERIVATION OF FRISCH'S ALGORITHM FOR CALCULATING VERTEX-PAIR CONNECTIVITY

KENNETH STEIGLITZ and JOHN BRUNO

Abstract.

In this paper we give a new and conceptually simple version of Frisch's algorithm for calculating the vertex connectivity of a graph. We show how this algorithm is obtained immediately from the Ford and Fulkerson labelling algorithm by using a "2-ply" scanning step. Data structures are introduced which lead to efficiencies in execution, and the final algorithm is presented in a "go-to-less" notation.

1. Introduction.

In a recent paper [1] Frisch describes an algorithm for finding the number of vertex disjoint paths between a pair of vertices in a graph. He establishes the validity of this algorithm by making a correspondence between the Ford-Fulkerson labelling algorithm (FFLA) applied to an augmented graph and his own algorithm applied to the original graph. In this paper we give a new and conceptually simpler version of Frisch's algorithm and show how it is obtained immediately from the FFLA by using a "2-ply" scanning step. Moreover, we present our results in an algorithmic notation and introduce data structures which lead to efficiencies in execution of the algorithm.

2. Statement of the Problem.

Let G be an undirected graph without self-loops or multiple edges, and with edges $E(G)$ and N vertices $V(G)$. Two vertices $I1$, $I2 \in V(G)$ are distinguished as source and terminal vertices. Our problem is to find the maximum number $NF(I1,I2)$ of vertex-disjoint paths from vertex $I1$ to $I2$.

It is well-known [2] that the Ford and Fulkerson labelling algorithm (FFLA) can be used to find $NF(I1,I2)$ if it is applied to an augmented

Received January 12, 1970; revised August 20, 1970.

This work was supported in part by U.S. Army Research Office, Durham under contract DA HC04 69 C 0012, and NSF Grant GJ-965.

graph G^+ which is obtained from G. G^+ is a capacitated directed graph defined as follows:

1. For every vertex $v \in V(G)$ let there be two vertices of G^+, denoted by v' and v''.
2. For every vertex $v \in V(G)$, $E(G^+)$ contains the edge (v'',v').
3. For every edge $(i,j) \in E(G)$, $E(G^+)$ contains the edges (i',j'') and (j',i'').
4. Every edge of G^+ has capacity 1; that is, if f is the flow in the direction of orientation on any edge of G^+, then $0 \leqq f \leqq 1$.

$NF(I1,I2)$ is found by finding the maximum flow from $I1'$ to $I2''$ in G^+.

We proceed by writing a program for the FFLA applied to G^+ which takes advantage of its special structure. By so doing we shall arrive at an algorithm which has the storage and time requirements of a flow calculation on a graph with N rather than $2N$ vertices.

3. Data Structures and Programming Style.

In this section we define the arrays which will be used in our program:

$NADJ(1:N)$	— a list which contains in position i the valence of vertex i in G. This entry is also equal to the out-valence of vertex i' and the in-valence of vertex i'' in G^+.
$NEAR(1:N,\ 1:N)$	— a matrix which describes the vertex to vertex adjacency of G. If i is a vertex of G, then $NEAR(i,k)$ for $k=1,\ldots,NADJ(i)$ is a list of all the vertices adjacent to i in G. This list is identical to the successor vertices of vertex i' in G^+, (with a double-prime assumed); as well as the predecessor vertices of vertex i'' in G^+ (with a prime assumed).
$F(1:N,\ 1:N)$	— a matrix which describes a flow in the directed graph G^+. The diagonal element $F(i,i)$ is 1 or 0 depending on whether there is or is not unit flow in the edge from vertex i'' to vertex i', respectively. The off-diagonal term $F(i,j)$ $(i \neq j)$ is 1 or 0 depending on whether there is or is not unit flow in the edge from vertex i' to vertex j''. Notice that there is a one-to-one correspondence between the entries of F and possible edges of G^+.
$INFLOW(1:N)$	— This list will allow us to scan doubleprime vertices of G^+ without searching the $NEAR$ array. If there

is a unit flow in an edge (j',i'') of G^+, then $INFLOW(i)=j$. Note that there is at most one such edge for each doubleprime vertex of G^+.

When the *FFLA* is applied to G^+, it is only necessary to use one label on each vertex: the number of an adjacent vertex from which an incre-

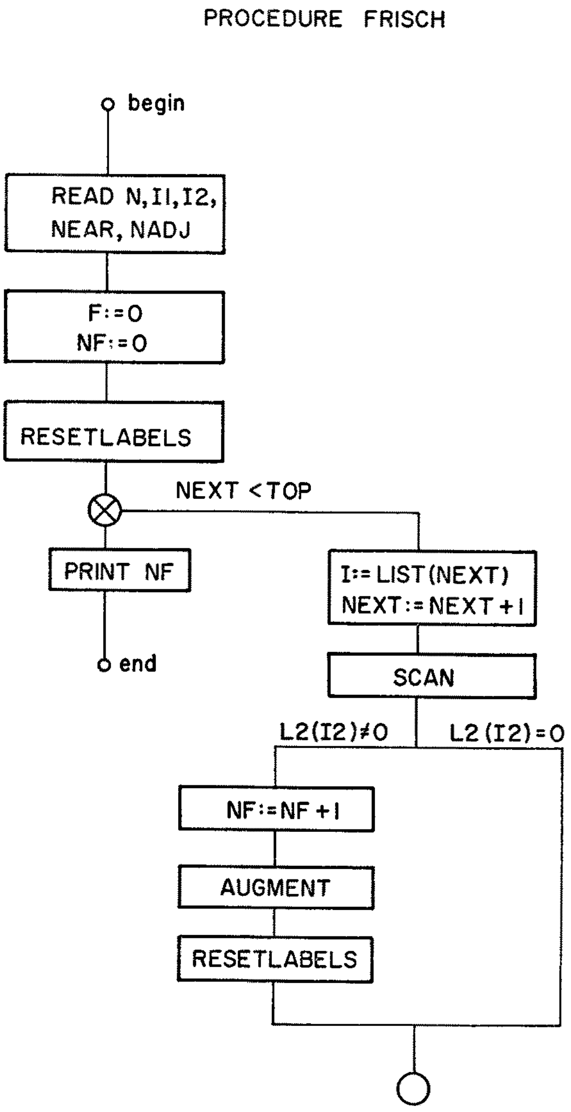

Fig. 1. Procedure *FRISCH*.

ment of flow can be brought. It is not necessary to record the size of the flow increment, since this is always 1. Furthermore, since a prime vertex can be labelled only from a double prime vertex, and *vice-versa,* it is not necessary to record whether a label is prime or double-prime.

$L1(1:N)$	— This list contains in position i the label of vertex i' in G^+, or 0 if vertex i' is unlabelled.
$L2(1:N)$	— This list contains in position i the label of vertex i'' in G^+, or 0 if vertex i'' is unlabelled.
$LIST(1:N)$	— This is a list of labelled but unscanned vertices of G^+. Since double-prime vertices will be scanned as soon as they are labelled, this list contains only prime vertices. Hence *LIST* need only have a capacity of N, and it is not necessary to record whether an entry is prime or double-prime. *LIST* will be managed as a stack, where *LIST(NEXT)* is the next vertex to be scanned, and *LIST(TOP)* is the next available space. Thus when $NEXT = TOP$, there are no more labelled, unscanned vertices in G^+.

Having described the arrangement of data for the flow calculation in G^+, we now turn to the program. Fig. 1 shows a diagram of main program *FRISCH.* The execution starts at the top and proceeds down-

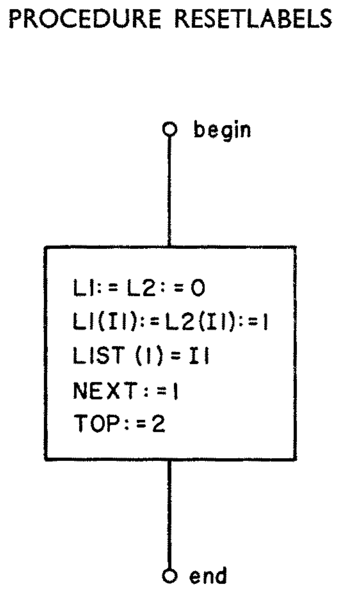

Fig. 2. Procedure *RESETLABELS.*

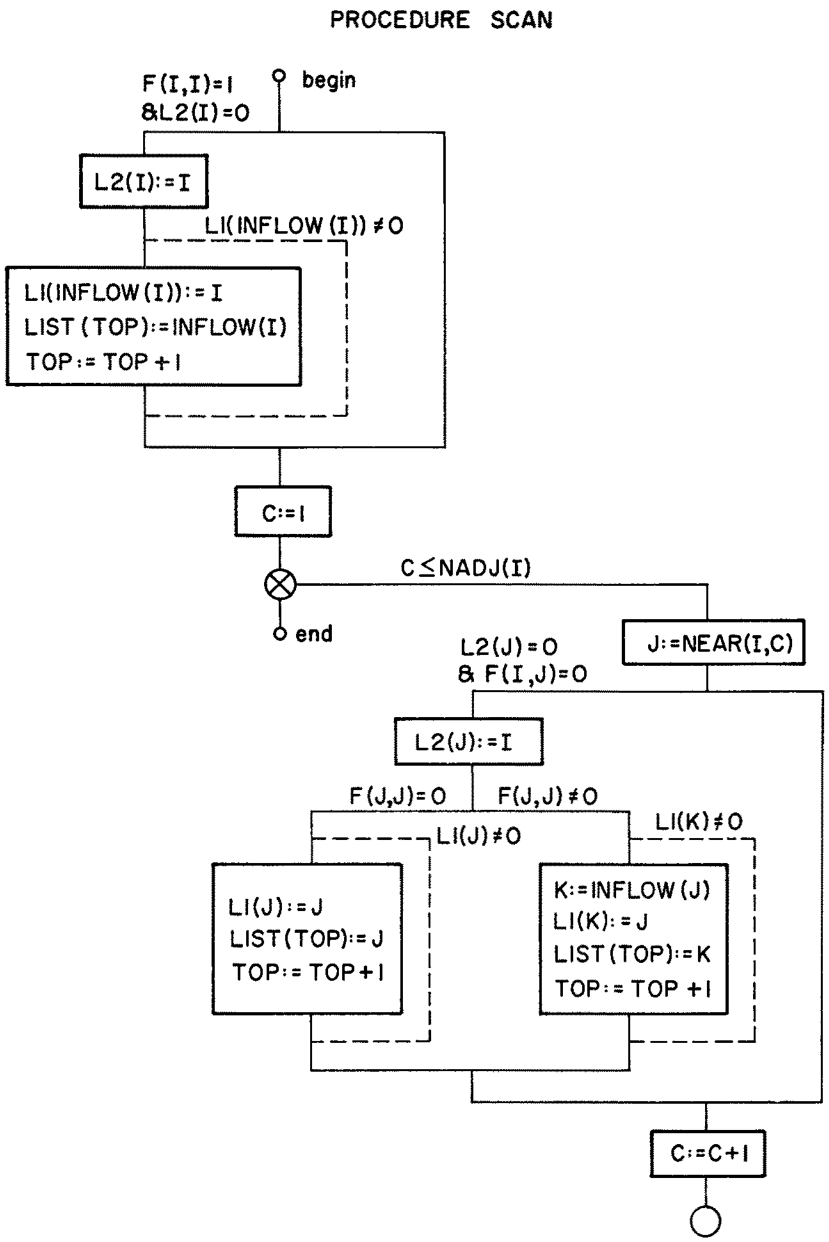

Fig. 3. Procedure *SCAN*.

ward, with straight-line code or other procedures enclosed in boxes. The junction indicated by "⊗" indicates a "while-do" control statement; the branch to the right is executed repeatedly as long as the condition on the branch, "*NEXT* < *TOP*", is satisfied. When the condition is violated for the first time, control continues downward from the "⊗". The 2-way branch following the box enclosing the procedure *SCAN* indicates an "if-then-else" control statement, with the condition indicated on the appropriate branch. This programming style has been advocated by Dijkstra [3] and has the distinction of not using "go to" statements. It has the advantage of indicating quite clearly exactly what conditions hold at each point, and the diagram can be translated

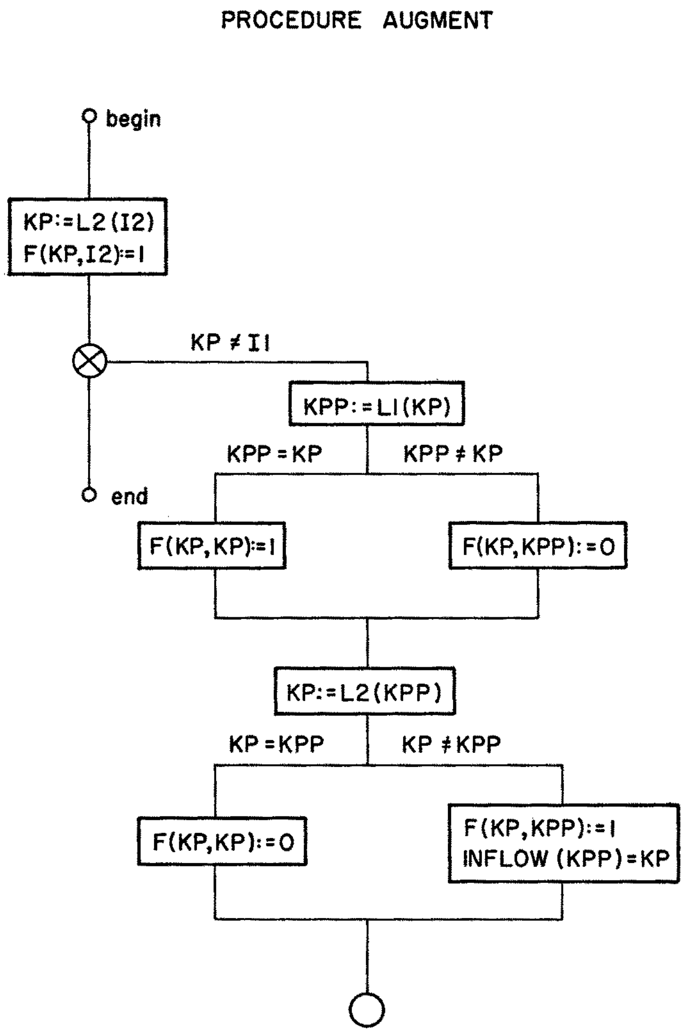

Fig. 4. Procedure *AUGMENT*.

into languages such as FORTRAN, ALGOL, PL1, and PL360 in a mechanical way.

Procedure *FRISCH* is a straightforward version of the FFLA, using the stack *LIST* described above, and procedures *RESETLABELS*, *SCAN*, and *AUGMENT*.

Procedure *RESETLABELS*, shown in Fig. 2, simply sets all the labels to 0 except those on $I1'$ and $I1''$ and places $I1'$ on *LIST*. This is used at the beginning of *FRISCH*, and after each flow augmentation.

Procedures *SCAN* and *AUGMENT*, shown in Figs. 3 and 4, respectively, are specially suited to the structure of G^+ and are described in the next two sections.

4. 2-Ply Scanning and Procedure *SCAN*.

Procedure *SCAN* scans vertex I' in G^+. First the predecessor vertex I'' is examined. If vertex I'' is unlabelled, and if there is a flow from I'' to I', then I'' can be labelled by setting $L2(I)=I$. At this point, I'' becomes labelled and unscanned, and in the general FFLA would be put on the stack *LIST* together with other vertices in this state. However, I'' can be scanned immediately as follows: since a unit flow exists from I'' to I', this flow has been brought to I'' from the vertex $INFLOW(I)'$. Furthermore, $INFLOW(I)'$ cannot at this point be already labelled, since it could have been labelled only from $INFLOW(I)''$ or I'', both of which are impossible. Hence $INFLOW(I)'$ can be labelled with I, and $L1(INFLOW(I))$ need not be tested. The unnecessary test is shown in Fig. 3 as a dotted branch in the program.

We call this process of labelling from prime vertex to double-prime vertex to prime vertex, "2-ply" scanning.

Similarly, we can label forward from I' to a successor vertex $J''=NEAR(I,c)''$, provided that $L2(J)=0$ and $F(I,J)=0$. Vertex J'' can be scanned immediately, and furthermore the label of the prime vertex to be labelled need not be tested first since we can show that it cannot have been labelled before. The two possible unnecessary tests are shown in Fig. 3 as dotted branches.

We thus can scan the vertex I', adding only prime vertices to *LIST*.

5. 2-ply Backtracking and Procedure *AUGMENT*.

When $I2''$ becomes labelled, a breakthrough has been found, and the flow from $I1'$ to $I2''$ can be augmented by 1. This is accomplished by Procedure *AUGMENT*. This procedure backtracks from vertex to vertex by labels, in the same way as the general FFLA, except that it proceeds in a 2-ply step from prime vertex to double-prime vertex to prime vertex. At each point we must determine whether or not we are backtracking from a vertex to its "companion" vertex (that is—prime-to-double-prime or double-prime-to-prime), and this tells us whether the flow should be augmented or cancelled. In one case, if we augment the flow from a prime vertex to a double-prime vertex, we need to set the appropriate entry of *INFLOW*.

6. An Algol Program.

Fig. 5 shows an ALGOL program with the procedure *FRISCH*, *RESETLABELS*, *SCAN* and *AUGMENT*. The three unnecessary tests in *SCAN* have not, of course, been programmed. Notice that the "while-do" control statement has been programmed as a **for** clause, with a dummy **for-list** as follows:

for *N* := *N* **while** (*condition*) **do**

Fig. 5. The complete ALGOL procedure *FRISCH*. The parameters and variables are described in Sections 2 and 3.

```
integer procedure FRISCH(N, I1, I2, NEAR, NADJ);
value N, I1, I2;
integer N, I1, I2;
integer array NEAR, NADJ;
begin
  integer NODE, NEXT, TOP, NF, I, J, K, KP, KPP, C;
  integer array F[1:N, 1:N], L1, L2, LIST, INFLOW[1:N];

  procedure RESETLABELS;
  begin
    for J := 1 step 1 until N do L1[J] := L2[J] := 0;
    L1[I1] := L2[I1] := 1;
    LIST[1] := I1;
    NEXT := 1;
    TOP := 2
  end RESETLABELS;

  procedure SCAN;
  begin
    if F[I, I] = 1 ∧ L2[I] = 0 then
    begin
      L2[I] := I;
      $L1[INFLOW[I]] := I$;
      $LIST[TOP] := INFLOW[I]$;
      $TOP := TOP+1$
    **end** *if*;
    $C := 1$;
    **for** $N := N$ **while** $C \leqq NADJ[I]$ **do**
    **begin**
      $J := NEAR[I, C]$;
      **if** $L2[J] = 0 \wedge F[I, J] = 0$ **then**
      **begin**
        $L2[J] := I$;
        **if** $F[J, J] = 0$ **then**
        **begin**
          $L1[J] := J$;
          $LIST[TOP] := J$;
          $TOP := TOP+1$
        **end**
      **else**
        **begin**
          $K := INFLOW[J]$;
          $L1[K] := J$;
          $LIST[TOP] := K$;
          $TOP := TOP+1$
        **end**
      **end** *if*;
      $C := C+1$
    **end** *for*
  **end** *SCAN*;

  **procedure** *AUGMENT*;
  **begin**
    $KP := L2[I2]$;
    $F[KP, I2] := 1$;
    **for** $N := N$ **while** $KP \neq I1$ **do**
    **begin**
      $KPP := L1[KP]$;
      **if** $KPP = KP$ **then** $F[KP, KP] := 1$
                     **else** $F[KP, KPP] := 0$;
      $KP := L2[KPP]$;
      **if** $KP = KPP$ **then** $F[KP, KP] := 0$
                     **else** **begin**
```

```
                              F[KP, KPP] := 1;
                              INFLOW[KPP] := KP
                           end
    end for
  end AUGMENT;

  comment Start of FRISCH;
  for J := 1 step 1 until N do
  for K := 1 step 1 until N do
  F[J, K] := 0;
  NF := 0;
  RESETLABELS;
  for N := N while NEXT < TOP do
  begin
    I := LIST[NEXT];
    NEXT := NEXT+1;
    SCAN;
    if L2[I2] ≠ 0 then
    begin
      NF := NF+1;
    AUGMENT;
      RESETLABELS
    end if
  end for;
  FRISCH := NF
end FRISCH;
```

7. Analysis of the Algorithm.

Dijkstra's "go-to-less" programming style has the advantage that the number of times each part of the resulting program is executed can be measured in a very straightforward way. We need only place counters in the initial branch, at the beginning of each "while-do" branch, and in one of each of the pairs of "if-then-else" branches. Fig. 6 shows the skeleton of the entire connectivity algorithm, with the counters indicated by N_i, $i=1,\ldots,10$. By convention we always place the counter in the left branch of an "if-then-else" branch.

As an example the connectivity was calculated between every pair of the 58-vertex 6-connected graph given in [4]. In this graph every vertex has a valence of 6. The results are shown in Fig. 6. These numbers have the following interpretation. First,

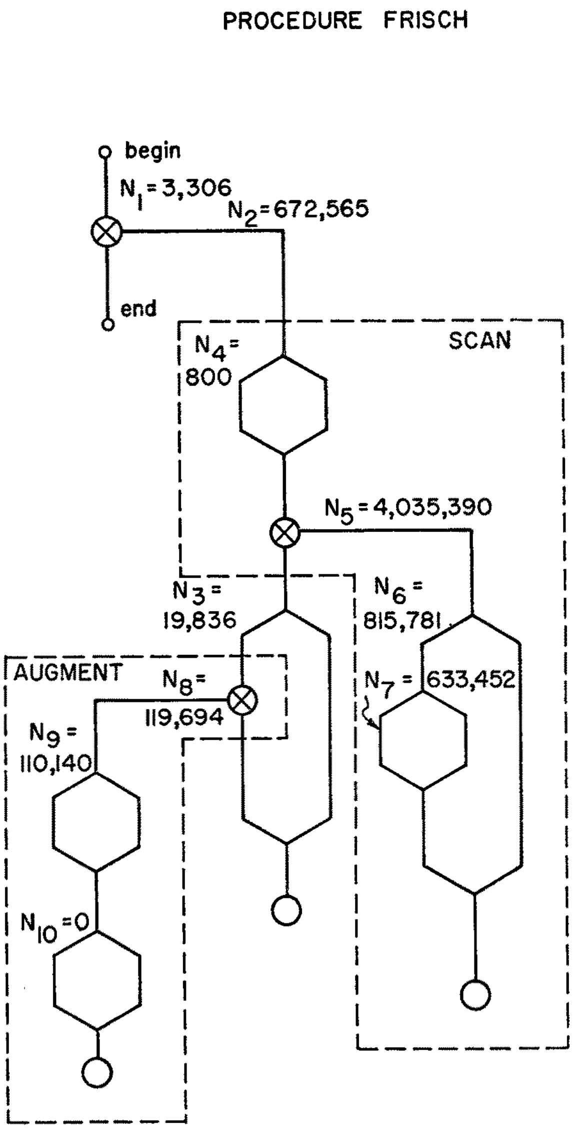

Fig. 6. Analysis of the algorithm for all vertex-pairs of a 58-vertex 6-connected graph.

$$N_1 = \text{number of vertex-pairs} = (57)(58)\,.$$

Next,

$$N_2 = N_1 \cdot NF \cdot S, \text{ where } S = \text{average number of vertices scanned per breakthrough.}$$

We can calculate $S = 33.9$ vertices/breakthrough. Next,

$$N_5 = N_2 \cdot V, \text{ where } V = \text{average valence} = 6\,.$$

Further,

$$N_3 = \text{total number of breakthroughs} = N_1 \cdot NF\,,$$

and

$$N_8 = N_3 \cdot A, \text{ where } A = \text{average number of 2-ply backtracking steps per augmentation path.}$$

Here, we can calculate

$$A = 6.0\,.$$

The ratio $N_4/N_2 = .12\%$ represents the probability of labelling back from I' to I'' when I' is being scanned, and is quite small for this graph. $N_6/N_5 = 20.0\%$ represents the probability of labelling forward to J''. $N_7/N_6 = 77.5\%$ represents the probability of labelling forward from J'' to J' as opposed to labelling back to $INFLOW(J)'$.

Turning to *AUGMENT*, $N_9/N_8 = 92.2\%$ represents the probability of backtracking from K' to K'' (augmenting flow) as opposed to backtracking from K' to some other double-prime vertex (cancelling flow). $N_{10}/N_8 = 0$ represents the probability of backtracking from K'' to K' (cancelling flow) as opposed to backtracking from K'' to some other prime vertex (augmenting flow). This last branch is never executed in this relatively highly connected graph. Fig. 7 shows a graph in which the

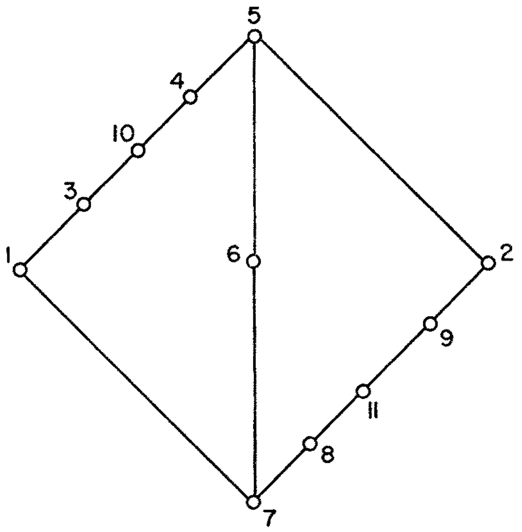

Fig. 7. A test case which exercises all the branches in the program. The connectivity between vertices 5 and 7 is 3, while the connectivity between all other pairs is 2.

branch labelled N_{10} is executed, in the flow calculation for vertex pair 1–2, for example. This occurs because the initial 1–2 path requires a subsequent backup through vertex 6.

The running time of the algorithm depends, of course, on how much code is in each branch, as well as the number of times each branch is executed. We may use the largest N_i, N_5, as an estimate of the asymptotic dependence of the running time on the graph parameters:

$$N_5 = N_1 \cdot NF \cdot V \cdot S\,.$$

Thus, we may expect running time per vertex-pair to be proportional to $NF \cdot V \cdot S$. Since S is of the order of N, and V is of the order of NF, this yields $N \cdot (NF)^2$.

8. Comments.

The algorithm can be used to calculate the connectivity of directed graphs by defining the arrays $NADJ$ and $NEAR$ to correspond to the appropriate G^+.

Furthermore, the idea of 2-ply scanning and 2-ply backtracking can be extended to graphs with arbitrary branch and vertex capacities, by virtue of the special structure of G^+.

Actual running time for the 58-vertex 6-connected example using a Fortran IV coding with counting, was about 27.5 vertex-pairs per second on the IBM 360/91.

9. Acknowledgment.

The authors wish to thank Dr. Ivan T. Frisch for helpful discussions about this problem, and the referee for improving the ALGOL program.

REFERENCES

1. I. T. Frisch, *An Algorithm for Vertex-Pair Connectivity*, Internatl. J. Control, vol. 6, no. 6, pp. 579–593; 1967.
2. L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, New Jersey; 1962.
3. E. W. Dijkstra, *Go To Statement Considered Harmful*, (Letter-To-The-Editor), Comm. of the ACM, vo.. 11, no. 3, pp. 147–8; March 1968.
4. K. Steiglitz, P. Weiner, and D. J. Kleitman, *The Design of Minimum-Cost Survivable Networks*, IEEE Trans. on Circuit Theory, vol. CT-16, no. 4, pp. 455–460; Nov. 1969.

DEPARTMENT OF ELECTRICAL ENGINEERING
PRINCETON UNIVERSITY
PRINCETON, NEW JERSEY 08540
USA