Computing 21, 183—194 (1979)

Computing

An Algorithm for Finding a Minimal Equivalent Graph of a Strongly Connected Digraph*

S. Martello, Bologna

Received March 13, 1978; revised July 5, 1978

Abstract — Zusammenfassung

An Algorithm for Finding a Minimal Equivalent Graph of a Strongly Connected Digraph. The paper presents a new branch and bound algorithm for removing the maximum number of edges from a strongly connected digraph without affecting its reachability properties.

A FORTRAN IV implementation is given.

The efficiency of the algorithm is analyzed through computational comparison with the methods of Moyles-Thompson and Hsu.

Ein Algorithmus, der zu einem stark zusammenhängenden Digraph einen äquivalenten kantenminimalen Digraph bestimmt. Es wird ein neuer Branch-and-Bound-Algorithmus vorgestellt, der aus einem stark zusammenhängenden Digraphen eine maximale Anzahl von Kanten entfernt, ohne die Zusammenhangsverhältnisse zu verändern.

Eine FORTRAN IV Version des Algorithmus ist beigefügt.

Das Verhalten des Algorithmus wird durch Abschätzung der Komplexität und durch Vergleich mit den Algorithmen von Moyles-Thompson und Hsu verdeutlicht.

1. Introduction

Let $G=(V, E)$ be a *directed graph* (or *digraph*), where $V=\{1, 2, \ldots, n\}$ is the set of all the *vertices* and E is the set of all the *edges* (i, j) in G. Let $[A_{i,j}]$ be the $(n \times n)$ *adjacency matrix* defined by

$$\begin{cases} A_{i,j}=1 & \text{if } (i,j) \in E; \\ A_{i,j}=0 & \text{otherwise.} \end{cases}$$

Define the following sets:

$$\Gamma^+(i)=\{j \mid (i,j) \in E\} \quad \text{for } i=1, \ldots, n;$$
$$\Gamma^-(i)=\{j \mid (j,i) \in E\} \quad \text{for } i=1, \ldots, n.$$

* An earlier version of this paper was presented at the ORSA/TIMS Joint National Meeting, Atlanta (November 1977).

0010-485X/79/0021/0183/$ 02.40

A sequence of $h+1$ vertices

$$i_0, i_1, i_2, \ldots, i_h$$

such that

$$i_k \in \Gamma^+(i_{k-1}) \quad \text{for } k=1,\ldots,h$$

is a *path* from i_0 to i_h and the $(n \times n)$ *path matrix* $[P_{i,j}]$ is defined by

$$\begin{cases} P_{i,j}=1 & \text{if either } i=j \text{ or a path exists from } i \text{ to } j; \\ P_{i,j}=0 & \text{otherwise.} \end{cases}$$

The problem of finding a *minimal equivalent graph* (MEG) of $G=(V, E)$ consists in removing the maximum number of edges from E without affecting the path matrix, that is in finding a graph $G^\circ=(V, E^\circ)$ such that

$$\begin{aligned} & E^\circ \subseteq E; \\ & P^\circ_{i,j}=P_{i,j} \quad \text{for } i,j=1,\ldots,n; \\ & |E^\circ| \quad \text{is a minimum.} \end{aligned}$$

Without loss of generality, we can suppose that

$$(i, i) \notin E \quad \text{for } i=1,\ldots,n.$$

$G=(V, E)$ is said to be *strongly connected* iff

$$P_{i,j}=1 \quad \text{for } i,j=1,\ldots,n.$$

Any graph $G=(V, E)$ can be subdivided into $m\,(1 \leq m \leq n)$ strongly connected subgraphs $S_i=(V_i, E_i)$ so that

$$\begin{gathered} V=V_1 \cup V_2 \cup \ldots \cup V_m; \\ V_i \cap V_j=\emptyset \text{ if } i \neq j, \text{ for } i,j=1,\ldots,m; \\ E_i \subseteq E \quad \text{for } i=1,\ldots,m. \end{gathered}$$

The *condensed graph* of G is defined as a graph $G^*=(V^*, E^*)$ where

$$V^*=\{1, 2, \ldots, m\};$$

$$(i,j) \in E^* \text{ iff } \exists\, (\bar{i}, \bar{j}) \in E \mid \bar{i} \in S_i \text{ and } \bar{j} \in S_j.$$

Moyles and Thompson [2], who first discussed the problem, showed that an MEG of G can be obtained through the following steps:

1. Find the strongly connected subgraphs S_i of G and build the condensed graph G^*.
2. Find an MEG for each S_i.
3. Find an MEG for G^*.

Most of the computing time is normally taken by step 2. In [2] two different algorithms (called $A2$ and $A2'$) have been proposed for this step; Hsu [1] showed that algorithm $A2$ contains some mistakes and proposed a different approach to the solution of the step. Neither in [1] nor in [2] results are given about the computational performance of the proposed methods.

This paper presents a new branch and bound algorithm for finding the minimal equivalent graph of a strongly connected digraph. The method's computational performance is analyzed through both theoretical and experimental comparisons with the algorithms of Moyles-Thompson and Hsu.

2. Preliminaries to the Algorithm

In what follows we will suppose that $G=(V, E)$ is a strongly connected digraph of n vertices. The enumerative scheme we will consider tries all possible ways of eliminating edges from G, cutting out from the decision tree those branches which cannot lead to a solution improving on a previously obtained current solution.

Suppose the edges in E are labelled by the integers $(1, 2, \ldots, e)$.

The algorithm starts by consecutively analyzing all the edges: an edge (i, j) is eliminated from E whenever an alternative path — not including previously eliminated edges — from i to j exists. After this phase is exhausted, the resulting graph $G^\circ=(V, E^\circ)$ is assumed as the first current solution; let $\bar{E}$ be a set containing all the removed edges.

The core of the algorithm is an iterated execution of backtracking and forward moves. A *backtracking move* consists in removing from $\bar{E}$ the highest labelled edge, say the k-th, and in re-inserting it in E; a *forward move* is then executed, consisting in consecutively considering all the edges of E with labels greater than k, removing from E those for which an alternative path exists and inserting them in $\bar{E}$. Whenever a forward move is exhausted, if $|E|<|E^\circ|$, the current solution $G^\circ=(V, E^\circ)$ is updated; in any case, a backtracking move follows. The algorithm stops when no further backtracking is possible ($\bar{E}=\emptyset$) or when $|E^\circ|=n$.

The test for the existence of an alternative path from i to j is obtained through a procedure based on Dijkstra's algorithm as modified by Yen [3]; since such a search normally takes a large part of the total computing time required by the algorithm, it is worthwhile to avoid it whenever possible. So, at each forward step, when considering an edge (i, j), the algorithm first tests whether

$$\Gamma^+(i)=\{j\}$$

or

$$\Gamma^-(j)=\{i\};$$

in this case it is immediately concluded that no alternative path can exist from i to j (note that the sets $\Gamma^+(i)$, $\Gamma^-(j)$ have obviously to be updated whenever an edge (i, j) is removed from E or re-inserted in it).

A bound is imposed on each phase of the forward move: a new edge, say the k-th, is considered only if:

$$|E|-(|S|-|\bar{S}|)<|E^\circ|,$$

with

$S=\{s \mid s\geq k,\ s\in E\}$;

$\bar{S}=$ set of edges $\in S$ which it is impossible to remove.

The size of $\bar{S}$ depends on the utilized scanning technique and will be discussed in the next section.

3. The Algorithm

Algorithm $\mathcal{M}$ scans the edges by analyzing the adjacency matrix $[A_{i,j}]$ by rows. Relative to the current situation, the meaning of the utilized variables is the following:

$$VR_i = |\Gamma^+(i)|;$$
$$VC_j = |\Gamma^-(j)|;$$
$$r = |\bar{E}|;$$
$$t = |S|.$$

e gives the initial value of $|E|$, r° the value of r in the current optimal solution $[A^\circ_{i,j}]$. In $[A_{i,j}]$ the removed 1's are set to -1, so the algorithm returns a solution $[A^\circ_{i,j}]$ where all the negative values have to be set to 0.

Algorithm $\mathcal{M}$

1. Compute $\left(VR_i = \sum_{j=1}^{n} A_{i,j} \text{ for } i=1,\ldots,n\right)$, $\left(VC_j = \sum_{i=1}^{n} A_{i,j} \text{ for } j=1,\ldots,n\right)$, $e = \sum_{i=1}^{n} VR_i$.
   If $e=n$, stop.
   Set $r=0$, $i=j=1$.
2. If $A_{i,j}=0$, or $VR_i=1$, or $VC_j=1$, go to 3.
   If, in the hypothesis $A_{i,j}=0$, an alternative path from i to j exists, set $A_{i,j}=-1$, $VR_i=VR_i-1$, $VC_j=VC_j-1$, $r=r+1$.
3. If $j<n$, set $j=j+1$ and go to 2.
   If $i<n$, set $i=i+1$, $j=1$ and go to 2.
   If $r=0$, set $r^\circ=0$, $(A^\circ_{u,v}=A_{u,v}$ for $u,v=1,\ldots,n)$ and stop.
   Set $t=0$.
4. Set $r^\circ=r$, $(A^\circ_{u,v}=A_{u,v}$ for $u,v=1,\ldots,n)$.
   If $e-r^\circ=n$, stop.
5. If $A_{i,j}=0$, go to 6.
   If $A_{i,j}=1$, set $t=t+1$ and go to 6.
   Set $A_{i,j}=1$, $VR_i=VR_i+1$, $VC_j=VC_j+1$, $r=r-1$.
   If $r+t-(n-i)>r^\circ$, go to 10.
   Set $t=t+1$.
6. If $j>1$, set $j=j-1$ and go to 5.
   If $i>1$, set $i=i-1$, $j=n$ and go to 5.
   Stop.

7. If $A_{i,j}=0$, go to 10.
   Set $t=t-1$.
   If $VR_i=1$ or $VC_j=1$, go to 8.
   If, in the hypothesis $A_{i,j}=0$, an alternative path from i to j exists,
   set $A_{i,j}=-1$, $VR_i=VR_i-1$, $VC_j=VC_j-1$, $r=r+1$ and go to 9.
8. If $r+t-(n-i)\leq r^\circ$, set $t=t+1$ and go to 6.
9. If $t-(n-i)=0$, go to 4.
10. If $j<n$, set $j=j+1$ and go to 7.
    If $i<n$, set $i=i+1, j=1$ and go to 7.
    Go to 4.

For what concerns the upper bound computed before a forward move is executed (steps 5 and 8), it is assumed that

$$|\bar{S}|=n-i;$$

in fact, if we perform a forward move starting from element $A_{i,j}$, at least one edge for each row following the i-th remains in the final solution. For the same reason the forward moves are stopped when we are sure that no further element could be removed, that is when, at step 9,

$$t-(n-i)=0.$$

4. Existence of an Alternative Path

The existence of an alternative path from i to j (steps 2 and 7 of algorithm $\mathscr{M}$) can be tested by means of the procedure described in the present section, obtained by modifying Yen's improvement [3] of Dijkstra's algorithm for shortest paths.

The efficiency of this algorithm, whose iterated executions normally take a large part of the computing time required by $\mathscr{M}$, is high if a boolean adjacency matrix is given; that is the main reason why in algorithm $\mathscr{M}$ too the boolean matrix $[A_{i,j}]$ has been employed. A different approach, utilizing $\Gamma^+(i)$ and/or $\Gamma^-(i)$ instead of $[A_{i,j}]$, would require an appropriate algorithm to efficiently compute $P_{i,j}$ and would give the computational advantage of avoiding the tests on the null elements of the adjacency matrix (steps 2, 5 and 7 of $\mathscr{M}$).

Algorithm $\mathscr{Y}$

1. Set $F_i=0$, $(F_{j'}=\infty$ for $j'=1,\ldots,n, j'\neq j)$, $H_i=n$, $(H_{i'}=i'$ for $i'=1,\ldots,n-1, i'\neq i)$, $k=n-1, s=i, i'=1$.
2. Set $j'=H_{i'}$.
   If $A_{s,j'}=1$, set $F_{j'}=0$.
   If $F_{j'}=0$, set $i''=i'+1$ and go to 3.
   If $i'<k$, set $i'=i'+1$ and repeat 2.
   Go to 6.

3. If $i'' > k$, go to 4.
   Set $j'' = H_{i''}$.
   If $A_{s,j''} = 1$, set $F_{j''} = 0$.
   Set $i'' = i'' + 1$ and repeat 3.

4. If $F_j = 0$, go to 5.
   Set $H_{i'} = H_k$, $k = k - 1$, $s = j'$.
   If $k > 1$, set $i' = 1$ and go to 2.
   If $A_{s,j} \neq 1$, go to 6.

5. An alternative path from i to j exists: return.

6. No alternative path from i to j exists: return.

The main difference between $\mathscr{Y}$ and the algorithm in [3] is that an adjacency matrix is here utilized instead of a distance matrix: so, if we assume that the length of an edge (p, q) is 0 if $(p, q) \in E$ and ∞ if $(p, q) \notin E$, it obviously follows that the shortest paths from i to all the vertices, given by $F_{j'}$ $(j' = 1, \ldots, n)$, take on the value 0 if a path from i to j' exists, the value ∞ otherwise (steps 1 to 3).

It should be noted that, when no vertex having a null distance from i has been found at step 2, it is immediately concluded that no further vertex could have a null distance from i. Also note that, at each iteration on step 4, it is tested whether the required alternative path has been found ($F_j = 0$).

5. Computational Complexity

This section analyzes the number of iterations required, in the worst and in the best case, by the algorithms for finding an MEG of a strongly connected digraph.

Let n be the number of vertices and e the number of edges in the graph.

Moyles-Thompson's algorithm $A2'$ tries all possible ways of constructing a *complete sequence* (that is a path which has the same initial and terminal vertex and goes through every vertex at least once) utilizing n edges; if that fails, it tries all possible ways of constructing a complete sequence utilizing $n+1$ edges; etc.

The number of iterations for each value of t (length of the complete sequence the algorithm tries to construct) is bounded by $(n-1)^{t-n}(n-1)!$. In the worst case, the values $t = n, n+1, \ldots, e-1$ have to be tried (we do not consider the value $t = e$ since, in this case, we can immediately conclude that the MEG coincides with the given graph), so the number of iterations is bounded by

$$\frac{1-(n-1)^{e-n}}{1-(n-1)}(n-1)!$$

In the best case, the first iteration ($t = n$) finds a Hamiltonian Circuit, i.e. an MEG of the graph.

In the average case, the algorithm can be expected to have a good efficiency when a high probability exists that a complete sequence can be found by utilizing a low number of edges, that is when the ratio e/n is sufficiently large.

The Hsu algorithm can be shortly described as follows:

1. Find an *acyclic* subgraph $G1$ of G (that is a graph $G1=(V, E1)$ having a path-matrix $[P1_{i,j}]$ such that if $P1_{i,j}=1$ then $P1_{j,i}=0$ for $i,j=1,\ldots,n, i\neq j$).
2. Use an appropriate algorithm to obtain an MEG $G2=(V, E2)$ of $G1$ and define $G3=(V, E3)$, with $E3=E2\cup(E-E1)$.
   Set $p=q=1$.
3. Assume that an edge $(i,j)\in E3$ is *superfluous* if between vertices i and j there is a path consisting of p number of edges from $E-E1$ and q number of edges from $E2$; try to remove as many superfluous edges of $G3$ as possible.
4. If $p<|E-E1|$, set $p=p+1$ and go to 3.
   If $q<|E2|$, set $q=q+1$, $p=1$ and go to 3.
   Stop.

Apart from the determination of an acyclic subgraph and its reduction (steps 1 and 2), which take a comparatively small number of operations, this algorithm performs the main step 3 $e^2/4$ times in the worst case (i.e. when $|E-E1|=|E2|=e/2$) and $n-1$ times in the best case (i.e. when $|E-E1|=n-1$, $|E2|=1$ or when $|E-E1|=1$, $|E2|=n-1$); step 3 tries to remove as many superfluous edges of $G3$ as possible, so it requires e complete iterations in the worst case and n in the best case. Thus the Hsu algorithm involves $e^3/4$ complete iterations in the worst case and $n(n-1)$ in the best case.

It should be noted that the Hsu algorithm's complexity is only apparently polynomial, since each complete iteration requires a nonpolynomial number of iterations.

In the average case, the algorithm will generally require a nearly complete enumeration of all the possible paths in $G3$; it can thus be expected to have an average performance affected by comparatively small variations, for the same values of n, when the "density" of the graph varies.

Algorithm $\mathscr{M}$ is based on a branch and bound technique, so, in the worst case (complete enumeration), it gives a binary tree with e levels ($2^{e+1}-1$ nodes), where $2^e/2+1$ nodes of the last level are not explored: a total of $O(\frac{3}{2}2^e)$ nodes. Half of these nodes, generated by forward steps 2 and 7, consist in trying to remove an edge from the graph and, in the worst case, require the use of procedure $\mathscr{Y}$; the remaining half, generated by backtracking step 5, consist in simply re-inserting in the graph a previously removed edge. We can conclude that the number of iterations of the algorithm is bounded by a value $O(\frac{3}{4}2^e)$.

In the best case ($r=n$) no iterations are needed.

In the average case, the number of iterations will strictly depend on the value of e, so the method appears particularly suitable for edge-sparse graphs.

6. Computational Experience

Neither Moyles and Thompson [2] nor Hsu [1] give experimental results about the performance of their methods for finding an MEG of a digraph. Computational results are here given, limited to algorithms for finding an MEG of a strongly connected digraph: we will compare algorithm $\mathcal{M}$ with algorithm $A2'$ of Moyles and Thompson (here referred to as $\mathcal{M}-\mathcal{T}$) and with algorithm $A2$ of Hsu (here referred to as $\mathcal{H}$).

All the algorithms have been coded in FORTRAN IV and tested on a CDC-6600 computer. Random graphs have been obtained by generating the values $A_{i,j}$ from a uniform probability distribution so as to obtain a given value of the average out-degree of the vertices; after a graph had been generated, if it was not strongly connected, it was discarded and a new graph was generated.

The tables of this section relate to three different values of the resulting average out-degree. For each table 6 sets of 10 strongly connected digraphs each have been generated for increasing values of n; each algorithm had a maximum of 256 seconds running time assigned to solve the 60 resulting problems. The entries in the tables give the average running times — and, in brackets, the maximum times — expressed in seconds; for the cases where the time limit was not enough, only the number of problems solved is given.

Table 1 shows that $\mathcal{M}$ is clearly better than the other methods for sparse graphs (average out-degree = 1.5).

Table 1. *Average out-degree of the vertices = 1.5. 10 problems for each n. Entries: Average time (maximum time) in CDC-6600 seconds*

n	$\mathcal{M}-\mathcal{T}$	$\mathcal{H}$	$\mathcal{M}$
5	0.002 (0.004)	0.016 (0.036)	0.002 (0.003)
10	0.033 (0.081)	0.186 (0.243)	0.009 (0.016)
15	14.631 (129.776)	1.126 (2.113)	0.033 (0.150)
20	—	3.946 (5.567)	0.301 (1.248)
25	—	11.857 (18.223)	0.788 (2.353)
30	—	3 problems	1.297 (3.321)

Table 2 relates to an average out-degree = 2.0: $\mathcal{M}$ was the fastest method for $n \leq 20$, $\mathcal{H}$ for $n \geq 25$, but no algorithm solved problems with more than 25 vertices.

Table 2. *Average out-degree of the vertices = 2.0. 10 problems for each n. Entries: Average time (maximum time) in CDC-6600 seconds*

n	$\mathcal{M}-\mathcal{T}$	$\mathcal{H}$	$\mathcal{M}$
5	0.002 (0.006)	0.015 (0.025)	0.002 (0.009)
10	0.227 (0.923)	0.283 (0.407)	0.020 (0.049)
15	12.246 (51.098)	1.774 (2.607)	0.300 (0.591)
20	—	5.438 (8.137)	2.114 (9.295)
25	—	16.285 (25.022)	8 problems
30	—	—	—

Table 3 (average out-degree = 2.5) shows an improvement in the relative performance of $\mathcal{H}$, which was better than the other methods for $n \geq 20$, while $\mathcal{M}$ was the fastest method for $n \leq 15$; no algorithm solved the complete set of problems for $n = 25$.

Table 3. *Average out-degree of the vertices = 2.5. 10 problems for each n. Entries: Average time (maximum time) in CDC-6600 seconds*

n	$\mathcal{M}-\mathcal{T}$	$\mathcal{H}$	$\mathcal{M}$
5	0.004 (0.011)	0.014 (0.023)	0.003 (0.006)
10	0.265 (1.169)	0.316 (0.396)	0.157 (0.492)
15	2.280 (10.173)	1.704 (3.183)	1.427 (4.691)
20	1 problem	8.222 (16.919)	7 problems
25	—	6 problems	—
30	—	—	—

Relative to the three considered cases, $\mathcal{M}$ was always better than $\mathcal{M}-\mathcal{T}$; $\mathcal{H}$ was better than $\mathcal{M}-\mathcal{T}$ for $n \geq 15$ and worse for $n \leq 10$.

These differences in the relative performances of the algorithms when n and/or the average out-degree of the vertices vary tend to confirm the results of the theoretical analysis of section 5: we saw that, in the worst case, algorithm $\mathcal{H}$ involves a number of iterations which is clearly smaller than that involved by the other two methods considered; the opposite holds for the best case. The computational results confirm that the number of iterations of algorithm $\mathcal{H}$ lies in a comparatively small range; they also seem to show that the average number of iterations of both algorithms $\mathcal{M}-\mathcal{T}$ and $\mathcal{M}$ is generally far from that computed for the worst case and that it depends on the density of the graph.

It should be noted that, since the three available algorithms utilize completely different approaches, their performance for each single graph is absolutely unpredictable; consider for example Table 4, where details are given about the first 3 problems generated in Table 3 for the case $n = 15$ (the average running times resulted, for this case, quite similar for the three algorithms). Problem 1 was very difficult for $\mathcal{M}-\mathcal{T}$, medium for $\mathcal{H}$ and quite easy for $\mathcal{M}$; problem 2 was very difficult for $\mathcal{M}$, slightly difficult for $\mathcal{H}$ and easy for $\mathcal{M}-\mathcal{T}$; problem 3 was difficult for $\mathcal{H}$ and very easy both for $\mathcal{M}-\mathcal{T}$ and for $\mathcal{M}$.

Table 4. *Average out degree of the vertices = 2.5. Single trials. n = 15. Running times in CDC-6600 seconds*

Problem number	$\lvert E \rvert$	$\lvert E^0 \rvert$	$\mathcal{M}-\mathcal{T}$	$\mathcal{H}$	$\mathcal{M}$
1	34	17	5.540	1.443	0.453
2	40	15	0.332	1.923	4.691
3	39	15	0.103	3.122	0.078

7. FORTRAN Implementation of Algorithm $\mathcal{M}$

```
      SUBROUTINE MEGBB(A,N)
C
C THIS SUBROUTINE DETERMINES, THROUGH THE BRANCH AND BOUND
C ALGORITHM OF MARTELLO, THE MINIMAL EQUIVALENT GRAPH OF A
C STRONGLY CONNECTED DIGRAPH OF  N  VERTICES.
C
C PARAMETER  A  MUST CONTAIN, IN INPUT, THE (NXN) ADJACENCY
C MATRIX OF THE GRAPH. A  IS RETURNED WITH ALL THE  1
C CORRESPONDING TO THE REMOVED EDGES SET TO  - 1 .
C N  IS RETURNED INTACT.
C
      INTEGER A(40,40),ABAR(40,40),VR(40),VC(40)
      INTEGER FF,R,T,RBAR
C STEP 1
      DO 10 I=1,N
        VR(I) = 0
        VC(I) = 0
   10 CONTINUE
      NARCS = 0
      DO 30 I=1,N
        DO 20 J=1,N
          IF ( A(I,J) .EQ. 0 ) GO TO 20
          VR(I) = VR(I) + 1
          VC(J) = VC(J) + 1
          NARCS = NARCS + 1
   20   CONTINUE
   30 CONTINUE
      IF ( NARCS .EQ. N ) RETURN
      R = 0
      DO 60 I=1,N
        DO 50 J=1,N
C STEP 2
          IF ( A(I,J) .EQ. 0 ) GO TO 50
          IF ( VR(I) .EQ. 1 ) GO TO 50
          IF ( VC(J) .EQ. 1 ) GO TO 50
          A(I,J) = - 1
          CALL PATHY(A,N,I,J,FF)
          IF ( FF .EQ. 1 ) GO TO 40
          A(I,J) = 1
          GO TO 50
   40     VR(I) = VR(I) - 1
          VC(J) = VC(J) - 1
          R = R + 1
C STEP 3
   50   CONTINUE
   60 CONTINUE
      IF ( R .EQ. 0 ) RETURN
      T = 0
      II = N
      JJ = N
C STEP 4
   70 DO 90 I=1,N
        DO 80 J=1,N
          ABAR(I,J) = A(I,J)
   80   CONTINUE
   90 CONTINUE
      RBAR = R
      IF ( NARCS - RBAR .EQ. N ) GO TO 200
C STEP 5
  100 IF ( A(II,JJ) .EQ. 0 ) GO TO 120
      IF ( A(II,JJ) .EQ. 1 ) GO TO 110
      A(II,JJ) = 1
      VR(II) = VR(II) + 1
      VC(JJ) = VC(JJ) + 1
      R = R - 1
      IF ( R + T - (N - II)  .GT. RBAR ) GO TO 180
  110 T = T + 1
C STEP 6
  120 IF ( JJ .EQ. 1 ) GO TO 130
      JJ = JJ - 1
      GO TO 100
  130 IF ( II .EQ. 1 ) GO TO 200
      II = II - 1
      JJ = N
      GO TO 100
C STEP 7
  140 IF ( A(II,JJ) .EQ. 0 ) GO TO 180
      T = T - 1
      IF ( VR(II) .EQ. 1 ) GO TO 160
      IF ( VC(JJ) .EQ. 1 ) GO TO 160
      A(II,JJ) = - 1
```

```
      CALL PATHY(A,N,II,JJ,FF)
      IF ( FF .EQ. 0 ) GO TO 150
      VR(II) = VR(II) - 1
      VC(JJ) = VC(JJ) - 1
      R = R + 1
      GO TO 170
  150 A(II,JJ) = 1
C STEP 8
  160 IF ( R + T - (N - II) .GT. RBAR ) GO TO 170
      T = T + 1
      GO TO 120
C STEP 9
  170 IF ( T - (N - II) .EQ. 0 ) GO TO 70
C STEP 10
  180 IF ( JJ .EQ. N  ) GO TO 190
      JJ = JJ + 1
      GO TO 140
  190 IF ( II .EQ. N ) GO TO 70
      II = II + 1
      JJ = 1
      GO TO 140
C STORE ABAR IN A
  200 DO 220 I=1,N
        DO 210 J=1,N
          A(I,J) = ABAR(I,J)
  210   CONTINUE
  220 CONTINUE
      RETURN
      END
```

8. FORTRAN Implementation of Algorithm $\mathscr{Y}$

```
      SUBROUTINE PATHY(A,N,II,JJ,FF)
C
C THIS SUBROUTINE TESTS, THROUGH A MODIFIED VERSION OF THE
C DIJKSTRA - YEN ALGORITHM, THE EXISTENCE OF A PATH FROM
C VERTEX  II  TO VERTEX  JJ  IN THE STRONGLY CONNECTED DIGRAPH
C OF  N  VERTICES WHOSE ADJACENCY MATRIX IS  A .
C
C PARAMETER  FF  IS RETURNED WITH THE VALUE  1  IF THE PATH
C EXISTS, WITH THE VALUE  0  OTHERWISE.
C A, N, II  AND  JJ  ARE RETURNED INTACT.
C
      INTEGER A(40,40),F(40),H(40)
      INTEGER FF
C STEP 1
      DO 10 J=1,N
        F(J) = 999
        H(J) = J
   10 CONTINUE
      F(II) = 0
      H(II) = N
      K = N - 1
      L = II
C STEP 2
   20 DO 30 I=1,K
        J = H(I)
        IF ( A(L,J) .EQ. 1 ) F(J) = 0
        IF ( F(J) .EQ. 0 ) GO TO 40
   30 CONTINUE
      GO TO 80
C STEP 3
   40 IS = I + 1
      IF ( IS .GT. K ) GO TO 60
      DO 50 IN=IS,K
        JS = H(IN)
        IF ( A(L,JS) .EQ. 1 ) F(JS) = 0
   50 CONTINUE
C STEP 4
   60 IF ( F(JJ) .EQ. 0 ) GO TO 70
      H(I) = H(K)
      L = J
      K = K - 1
      IF ( K .GT. 1 ) GO TO 20
      IF ( A(L,JJ) .NE. 1 ) GO TO 80
C STEP 5
   70 FF = 1
      RETURN
C STEP 6
   80 FF = 0
      RETURN
      END
```

Acknowledgement

I would like to acknowledge Professor Walter Knödel for his valuable suggestions and comments.

References

[1] Hsu, H. T.: An Algorithm for Finding a Minimal Equivalent Graph of a Digraph. Journal of ACM *22* (1975).

[2] Moyles, D. M., Thompson, G. L.: An Algorithm for Finding a Minimum Equivalent Graph of a Digraph. Journal of ACM *16* (1969).

[3] Yen, J. Y.: Finding the Lenghts of all Shortest Paths in N-Node Nonnegative Distance Complete Networks Using $^1/_2\ N^3$ Additions and N^3 Comparisons. Journal of ACM *19* (1972).

Dr. S. Martello
Istituto di Automatica
University of Bologna
Viale Risorgimento, 2
I-40136 Bologna
Italy