Chapter I

SHORTEST PATHS

Annals of Operations Research 13(1988)3 – 79

SHORTEST PATH ALGORITHMS*

Giorgio GALLO

Department of Informatics, University of Pisa, 40, Corso Italia, I-56100 Pisa, Italy

and

Stefano PALLOTTINO

I.A.C., National Research Council, 137, viale del Policlinico, I-00161 Roma, Italy☆

Abstract

The *shortest path problem* is considered from a computational point of view. Eight algorithms which solve the *shortest path tree problem* on directed graphs are presented, together with the results of wide-ranging experimentation designed to compare their relative performances on different graph topologies. The focus of this paper is on the implementation of the different data structures used in the algorithms. A "Pidgin Pascal" description of the algorithms is given, containing enough details to allow for almost direct implementation in any programming language. In addition, Fortran codes of the algorithms and of the graph generators used in the experimentation are provided on the diskette.

1. Introduction

In this paper, we deal with the *shortest path problem* from a computational point of view. As is well known, this problem is a fundamental component in real-life large-scale network models. This explains why, although the problem itself is quite simple and widely studied, new contributions keep appearing in the scientific literature (see, for one example, the paper by Glover et al. [20]).

For an extended survey on the subject, we refer to Gallo and Pallottino [16], where the shortest path methods are presented in a unified framework. There, all the algorithms are shown to derive from a single prototype procedure, the main difference between them being in the particular data structures used to implement the set of candidate nodes. Here, from among the ones presented there, we have selected eight algorithms which solve the *shortest path tree problem* on directed graphs. In this selection, we followed three main criteria: historical importance, practical computational relevance and simplicity of implementation. Thus, we have left out a certain number of important algorithms, either because their interest is more theoretical

*Research carried out as part of the SOFMAT (Mathematical Software) activities of the Italian National Research Council (C.N.R.) "Progetto Finalizzato Informatica".
☆Present affiliation: Department of Informatics, University of Pisa, I-56100 Pisa, Italy.

© J.C. Baltzer AG, Scientific Publishing Company

than practical (see, for instance, Johnson [25]) or because their use implies rather complex implementation such as some of the algorithms presented in Denardo and Fox [5].

The focus of the paper is on the implementation of the different data structures used in the algorithms. Although Fortran codes are provided on the diskette, the "Pidgin Pascal" description of the algorithms contains enough details to allow for almost direct implementation in any other language. In fact, there is a one-to-one correspondence between Fortran codes and Pascal descriptions of the algorithms.

The results of wide-ranging experimentation with a large number of types of graphs are reported. The source codes of the graph generators used in the experimentation are contained on the diskette.

Although the aim of the paper is to provide well tested and efficient algorithms for the shortest path tree problem, some attention has also been reserved for the *all pairs* problem. In sect. 9, an efficient algorithm for this problem is described, in such a way that its implementation can easily be obtained making use of the shortest path tree algorithms on the diskette as subroutines.

2. Definitions and notation

Let us state the main assumptions and the notation we shall use throughout. $G = (N, A; l)$ is a *directed graph* with $n = |N|$ *nodes*, $m = |A|$ *arcs* and a *length function* $l{:}A \to R$. We shall denote the *length* of arc (i, j) by l_{ij}. The length of a path (respectively, of a cycle) is the sum of the lengths of its arcs. The arc lengths may be either positive or negative; the only assumptions we need to make is that there is no *directed cycle* with negative length in G. A further possible assumption is that G is strictly connected, i.e. for each pair on nodes u and v, a *directed path* exists from u to v. Note that this assumption is not binding; in fact, the connection can always be achieved by the insertion of arcs with a very high length $(+\infty)$. In the following, instead of directed path (directed cycle), we shall simply use *path* (*cycle*).

Given a node r, which we call the *origin*, a *shortest path tree* (**spt**), $T^*(r)$, is a spanning directed tree of G, rooted at r, which, for each $v \in N$, contains a shortest path from r to v (see Lawler [30]). We shall be dealing with the problem of finding a shortest path tree.

We shall assume that the graph be given in the form of *arc-lists*. That is, for each node u, the list is given of those arcs (u,j) which have u as the first node. The set of such arcs, $\mathrm{FS}(u) = \{(u, j) \in A\}$, is called the *forward star* of node u. When the graph is sparse, which is often the case in applications, it is much better to use this data structure than the matrix of arc lengths; this is because it requires less computer storage and usually allows the implementation of more efficient algorithms.

G, then, is represented by n lists, one for each forward star (see fig. 1), accessed by one array of pointers.

An efficient implementation of the arc-lists is to put them consecutively in a pair of arrays, ND[.] and LNGT[.]. The pointers are stored in array $A[.]$; each component points to the first element of the corresponding arc-list, i.e. $A[u] = w$ if (u, v) is the first arc in the forward star of u where $v = \text{ND}[w]$ and $l_{uv} = \text{LNGT}[w]$. By convention, we set $A[u] := A[u+1]$ when $\text{FS}(u) = \emptyset$, and $A[n+1] := m+1$.

Thus, the information relative to FS(u) is stored in ND[.] and LNGT[.], from position $A[u]$ to $A[u+1] - 1$ (see fig. 2). The storage requirement for this implementation is $2m + n + 1$. An example of this technique of representing graphs is given in fig. 3.

The scanning of FS(u) can be implemented as:

```
if A[u] < A[u+1] then
   for j := A[u] to A[u+1] - 1 do
      begin
         v   := ND[j];
         l_uv := LNGT[j];
         .
         .
         .
      end;
```

In the following, such a sequence of operations shall be denoted by the single compact statement

foreach $(u, v) \in \text{FS}(v)$ **do** ...

In order to represent a tree, we shall use the *predecessor list,* that is, a vector p, where p_v is the predecessor of node v in the tree. The predecessor list p is implemented by means of an n-array $P[.]$ in such a way that:

(i) $P[r] = 0 \leftrightarrow r$ is the root of T,

(ii) $P[j] = i \leftrightarrow (i, j)$ is an arc of T.

Together with the vectors defined so far, an n-array $D[.]$ is used, which will be returned by the algorithms as the shortest distance vector.

3. A prototype shortest path tree algorithm

Virtually all the shortest path tree algorithms can be viewed as performing the following operations:

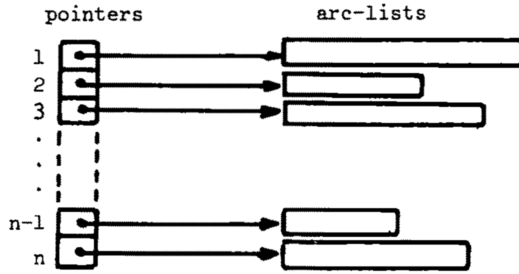


Fig. 1. A record in the lists contains the ending node of one arc and its length.

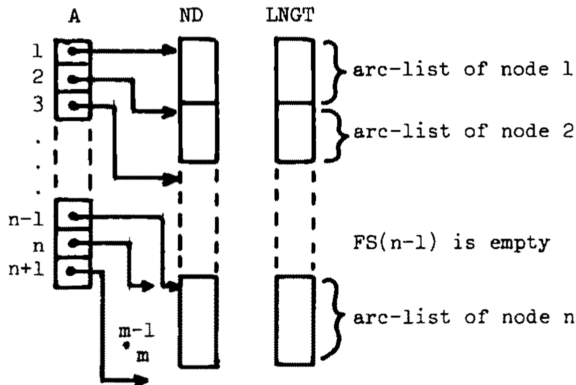
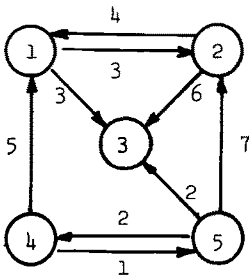


Fig. 2. Graph storage.



	1	2	3	4	5	6
A	1	3	5	5	7	10

	1	2	3	4	5	6	7	8	9
ND	2	3	1	3	1	5	2	3	4
LNGT	3	3	4	6	5	1	7	2	2

Fig. 3. A graph with five nodes and nine arcs.

1. Initialize a directed tree T rooted at r and, for each $v \in N$, let d_v be the length of the path from r to v in T;
2. Let $(i, j) \in A$ be an arc for which $d_i + l_{ij} - d_j < 0$, then adjust the vector d by setting $d_j = d_i + l_{ij}$, and update the tree T by replacing the current arc incident into node j by the new arc (i, j);
3. Repeat step (2) until the optimality conditions

   $$d_i + l_{ij} \geqslant d_j, \quad \forall (i, j) \in A$$

   are satisfied.

Note that, during the execution of the procedure, d_v is greater than, or equal to, the distance on the current tree from r to v, $v \in N$; only at termination is equality bound to hold. We shall call d_v the *label* of node v.

A crucial point in the implementation of this procedure is how to select arcs at step 2.

Since $n \leqslant m$, it seems quite reasonable to select nodes rather than arcs, then once a node u has been selected, the operations of step 2 are performed on all arcs of FS(u). This choice has the further advantage of being able to exploit reasonably well the arc-list representation of the graph.

Although a few algorithms have been developed in which only one arc of the forward star is considered for each selected node, in the large majority of the algorithms presented in the literature, when a node is selected the whole forward star is considered in one go.

Algorithm SPT, which is given next, is a rather general implementation of the procedure, based on the exploration of complete forward stars.

```
Procedure SPT(r);
  begin
    TINIT(r); QINIT(r);
    repeat
      QOUT(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin QIN(v); TUPDATE(u, v) end
    until Q = ∅
  end;

Procedure TINIT(r);
  begin
    for i := 1 to n do begin P[i] := r; D[i] := +∞ end;
    P[r] := 0; D[r] := 0
  end;
```

```
Procedure QINIT(r);
   begin
      Q := {r}
   end;

Procedure TUPDATE(u, v);
   begin
      P[v] := u; D[v] := D[u] + l_uv
   end;

Procedure QOUT(u);
   begin
      select u ∈ Q; Q := Q − {u}; update Q
   end;

Procedure QIN(v);
   begin
      if v ∉ Q then Q := Q ∪ {v};
      update Q
   end;
```

The initial tree in TINIT is a star-shaped tree, with one dummy arc (r, v) for each $v \in N - \{r\}$; these dummy arcs are assigned a length equal to $+\infty$.

At this point, no assumption is made about how, in QOUT, a node u is selected from the set of *candidate nodes* Q. This is crucial. In fact, almost all the **spt**-algorithms of practical interest are derived from SPT by properly defining the operation of *selection* and, consequently, the particular data structure which is used to implement the set Q.

In the following section, different *selection rules* and different data structures for Q are described. For each implementation of SPT, the storage requirements and the time complexity are given. As a theoretical measure of time complexity, the worst-case running time on a random-access machine is used (Tarjan [43]).

We now give a general expression of the complexity of SPT. Here, the complexity is given as a function of the operations performed: from this expression, we shall derive the complexity of the different implementations of SPT as a function of n and m. Let q_0, q_1 and q_2 denote the complexity of QINIT, QOUT and QIN, respectively. Let c_1 and c_2 denote the number of times QOUT and QIN are performed, respectively.

Let $\tilde{c}$ be the maximum number of selections of a single node. Then we may state the complexity of SPT as $O(q_0 + q_1 . c_1 + q_2 . c_2)$. Note that $\tilde{c} \geqslant 1$, $n \leqslant c_1 \leqslant n . \tilde{c}$, $m \leqslant c_2 \leqslant \min\{n . c_1, m . \tilde{c}\}$, and hence $c_1 \leqslant c_2$ (Gallo and Pallottino [16]).

4. Selection rules

The choice of the selection rule in QOUT affects the way in which the graph G is explored to check whether the optimality conditions are satisfied: each selection rule induces a particular *search strategy* on G. Three of the most commonly used search strategies are the *breadth-first search,* the *depth-first search* and the *best-first search* (Aho et al. [1], Tarjan [42]).

In the *breadth-first search* (also known as FIFO, First-In-First-Out), at each iteration the oldest element in Q is selected, i.e. the element which was inserted first,

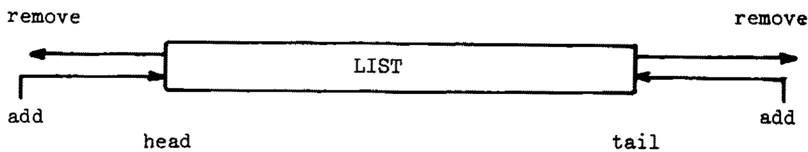


Fig. 4. A list.

whereas in the *depth-first search* (or LIFO, Last-In-First-Out), the element of Q which is selected is the newest one, i.e. the element which was inserted last.

A characteristic common to the breadth-first search, the depth-first search and all search strategies derived from them, is that to implement them properly, lists need to be used. A *list* is a sequence of elements; the first element is its *head* and the last element is its *tail.*

Typical operations on a list are: *adding* an element to form the new head of the list (making the old head the second element); adding an element to form the new tail; *removing* (retrieving and deleting) the head of the list; *removing* the tail of the list.

Other list operations include: *concatenating* two lists (making the tail of the first list point to the head of the second one), *inserting* an element after an element whose location in the list is known, and *deleting* an element whose location in the list is known (Tarjan [42]).

In the *best-first search,* we assume that a real valued label is associated with each element, and the element to be selected is the minimum label element currently in Q. Although a list might also be used to implement this search strategy, an efficient implementation calls for more sophisticated data structures. The data structure most commonly used in this case is the priority queue. A *priority queue* is a collection of elements, each with an associated numerical value (label), on which the following operations are efficiently performed: *adding* a new element, *removing* the minimum value element and *correcting* the label of an element whose location is known.

In the following section, we shall call *list-search algorithms* the **spt** algorithms which make use of either a breadth-first or a depth-first search (or any other search strategy derived from them). We shall call *shortest-first search algorithms* those **spt**

algorithms which make use of a search strategy derived from the best-first one. The name of the latter class of algorithms comes from the fact that the label of node v, d_v, represents the distance of a path from r to v in G at each iteration. Then the best label element is the one, out of those of Q, which is at the shortest distance from r, at least as far as we can ascertain at that stage of the computations.

5. List implementation

The two simplest and most common types of lists are the *queue* and the *stack*. A *queue* is a list in which additions are allowed only at the tail and deletions are allowed only at the head; the queue is used to implement the breadth-first search strategy. A *stack* is a list with addition and deletion allowed only at the head; it is used to implement the depth-first search strategy.

Two other types of lists which are relevant in the implementation of shortest path algorithms are the *deque* and the *2queue* (Horowitz and Sahni [21], Knuth [28], Pallottino [35]).

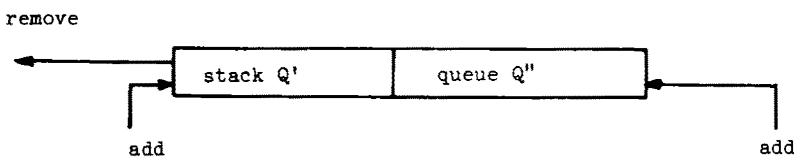


Fig. 5. Deque Q.

A *deque*, or double-ended queue, is a list in which additions and deletions are possible at either end. In the deque Q, used in the following, additions are made at both ends, while deletions are made at the head. Deque can be interpreted as a stack Q' and a queue Q'' connected in series, in such a way that the tail of the stack points to the head of the queue.

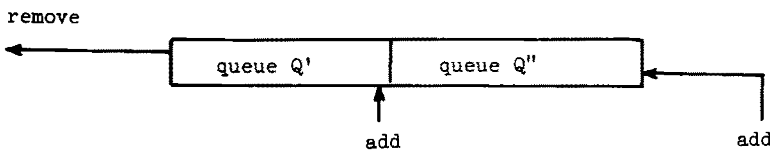


Fig. 6. 2queue Q.

2queue differs from deque in that the elements are inserted into Q' at the tail instead of being inserted at the head; thus, Q can be interpreted as two queues, Q' and Q'', connected in series.

Queues, stacks, deques and 2queues are efficiently implemented by means of linked-lists.

5.1. LINKED LIST

The *linked-list* we deal with is a sequence of elements, $l_1, l_2, \ldots, l_n$, each of which is one of the nodes in the graph ($l_i \in N$).

Each element is linked by means of a pointer to the next one. Two special elements, *frst* and *last* (*frst* $= l_1$, *last* $= l_n$), are used for quick retrieval of the head and the tail of the list, respectively.

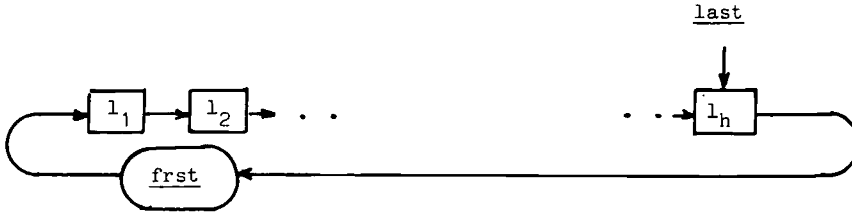


Fig. 7. Linked-list.

In practice, it is advantageous to implement the linked-list in a circular fashion, making the last element l_n point to the dummy element *frst*. The general structure of this kind of list is depicted in fig. 7.

Since the nodes are represented by the first n integers, the linked-list is efficiently implemented by means of an $n + 1$ array $Q[.]$, where:

$$Q[i] = \begin{cases} 0 & \text{if } i \notin Q, \\ j & (0 < j \leqslant n) \text{ if } i \text{ precedes } j \text{ in } Q, \\ n+1 & \text{if } i \text{ is the last element;} \end{cases}$$

$$Q[n+1] = frst = \text{the head of } Q.$$

An additional element, *last*, is needed so that the total storage requirement is $n + 2$.

The single vector $Q[.]$ can be used to represent two (or more) linked-lists connected in series, such as deque and 2queue, provided that these lists are disjoint.

In particular, the general implementation described above can be used for deque, with the following proviso that the last element of queue Q'' points to $n + 1$ and $Q[n + 1]$ contains the first element (the head) of stack Q'.

Note that there is no need to know the connection point between the stack and the queue. On the other hand, this point must be known in order to perform the insertion operation (at the tail of the queue Q') in 2queue; so, in this case an additional pointer, *pntr*, is used.

Since $v \in Q$ if and only if $Q[v] \neq 0$, and $Q = \emptyset$ if and only if $Q[n + 1] = n + 1$, the questions "is v in Q?" and "is Q empty?" can be answered in constant time.

The complexity of procedure QINIT is $O(n)$:

```
Procedure QINIT(r);
  begin
    for i := 1 to n do Q[i] := 0;
    Q[n + 1] := r; Q[r] := n + 1; last := r;
    pntr := n + 1; comment: only for 2queue
  end;
```

Procedure QOUT, described next, is common to all the lists consiedered; it runs in constant time.

```
Procedure QOUT(u);
  begin
    u := Q[n + 1]; Q[n + 1] := Q[u]; Q[u] := 0;
    if last = u then last := n + 1;
    if pntr = u then pntr := n + 1; comment: only for 2queue
  end;
```

As for the addition of a new element, $Q := Q \cup \{v\}$, we must distinguish between the case of addition at the head and the case of addition at the tail:

addition at the head

```
Q[v] := Q[n + 1];
Q[n + 1] := v;
if last = n + 1 then last := v;
if pntr = n + 1 then pntr := v;
                comment: only for 2queue;
```

addition at the tail

```
Q[last] := v;
Q[v] := n + 1;
last := v;
```

Hence, procedure QIN is different for each of the four lists considered. In deque and 2queue, the choice of the list to which the new element v must be added is commanded by a logical variable, *cond*, which can assume either of two values: *true* or *false*. All these procedures run in constant time.

```
Procedure QIN(v);
  begin comment: version for queue;
    if Q[v] = 0 then begin Q[last] := v; Q[v] := n + 1; last := v end
  end;
```

```
Procedure QIN(v);
  begin comment: version for stack;
    if Q[v] = 0 then begin Q[v] := Q[n + 1]; Q[n + 1] := v end
  end;
```

```
Procedure QIN(v);
  begin comment: version for deque;
    if Q[v] = 0 then if cond
      then
        begin comment: tail insertion;
          Q[last] := v; Q[v] := n + 1; last := v
        end
      else
        begin comment: head insertion;
          Q[v] := Q[n + 1]; Q[n + 1] := v;
          if last = n + 1 then last := v
        end
  end;

Procedure QIN(v);
  begin comment: version for 2queue;
    if Q[v] = 0 then if cond
      then
        begin comment: tail insertion in Q";
          Q[last] := v; Q[v] := n + 1; last := v
        end
      else
        begin comment: tail insertion in Q';
          Q[v] := Q[pntr]; Q[pntr] := v;
          if last = pntr then last := v;
          pntr := v
        end
  end;
```

5.2. TWO-WAY LINKED-LIST

Two-way linked-lists are used when elements are either deleted from, or inserted into, the list at an arbitrary position. In this case, two $n + 1$ arrays are used, UP[.] and DOWN[.], where:

$$\begin{aligned}
&\mathrm{UP}[i] = \mathrm{DOWN}[i] = 0 && \text{if } i \notin Q,\\
&\mathrm{UP}[\mathrm{DOWN}[i]] = \mathrm{DOWN}[\mathrm{UP}[i]] = i && \text{if } i \in Q,\\
&\mathrm{DOWN}[n+1] = \mathit{frst} = \text{the head of } Q,\\
&\mathrm{UP}[n+1] = \mathit{last} = \text{the tail of } Q.
\end{aligned}$$

DOWN[i] points to the element following i in the list, while UP[i] points to the preceding element. Note that, since a two-way linked-list has an inherent

symmetry, it is just a matter of convention to call one of the two ends of the list *head* and the other one *tail*: the roles of tail and head can be interchanged.

The initialization of the two arrays UP[.] and DOWN[.] is similar to the initialization of the array Q[.] described above.

Characteristic of the two-way linked-lists are the following elementary operations:

```
Procedure DELETE(x);
  begin
    comment: deletion of element x from Q;
    UP[DOWN[x]] := UP[x]; DOWN[UP[x]] := DOWN[x];
    DOWN[x] := UP[x] := 0
  end;

Procedure INSERT(x, y);
  begin
    comment: insertion of element x to Q immediately after y;
    DOWN[x] := DOWN[y]; UP[DOWN[x]] := x;
    DOWN[y] := x; UP[x] := y
  end;
```

5.3. MULTIPLE LISTS

A *multiple list* is a collection of h disjoint linked-lists (one or two ways) $L_1, L_2, \ldots, L_h$.

A single array Q[.] (or, in the case of two ways, a pair of arrays UP[.] and DOWN[.]) can be used to implement a multiple list. Since each list needs two pointers *frst* and *last*, two additional arrays of h elements FRST[.] and LAST[.] are needed. Thus, the overall storage requirement is $n + 2h$ for multiple linked-lists and $2n + 2h$ for multiple two-way linked-lists.

Two-way linked-lists and multiple lists will be used to implement priority queues (see sect. 7).

6. List search algorithms

We now present four **spt** algorithms which make use of linked-lists in order to implement the set Q.

Remembering the general complexity bound presented in sect. 3, $O(q_0 + c_1 q_1 + c_2 q_2)$, where $c_2 \geqslant c_1 \geqslant n$, it is easy to verify that these algorithms run in $O(c_2)$; in fact, q_0 is $O(n)$, q_1 and q_2 are constant.

Algorithm **L-queue**

The queue seems to be a very natural choice when implementing SPT. We shall call such an algorithm **L-queue.** It represents an efficient implementation of a well-known shortest path method which is often credited to Bellman [3], Ford [12] and Moore [32].

```
Procedure LQUEUE(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞; Q[i] := 0 end;
    P[r] := D[r] := 0; Q[n + 1] := last := r; Q[r] := n + 1;
    repeat
      comment: QOUT;
      u := Q[n + 1]; Q[n + 1] := Q[u]; Q[u] := 0;
      if last = u then last := n + 1;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
          if Q[v] = 0 then begin Q[last] := last := v; Q[v] := n + 1 end;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until Q[n + 1] = n + 1
  end;
```

Since each node cannot be inserted in the queue more than n times (Lawler [30]), $\tilde{c} \leqslant n$ and, as $c_2 \leqslant m\tilde{c}$, the complexity of **L-queue** is $O(nm)$ ($O(n^3)$ for complete graphs).

The space requirement is $4n + 2m$: $n + 2m$ for the input data, n for the queue, and $2n$ for the arrays $P[.]$ and $D[.]$.

Algorithm **L-deque**

A deque is used in the well-known D'Esopo–Pape algorithm (see Pape [36–38]), where the insertion policy is the following:

> the first time a node is to be inserted into Q, it is added to Q'' at the tail; this corresponds to a breadth-first search strategy. When, later on, the same node, after being removed from Q, again becomes a candidate for insertion, it is added to Q' at the head: from now on the node is processed on the basis of a depth-first search strategy.

The rationale for using this rather peculiar policy is that every time a label d_u is updated (decreased), except the first time, it is worth trying to decrease the labels of the successors of u in the current tree as well: this is the aim of the depth-first search phase.

We call this algorithm **L-deque.**

```
Procedure LDEQUE(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞; Q[i] := 0 end;
    P[r] := D[r] := 0; Q[n + 1] := last := r; Q[r] := n + 1;
    repeat
      comment: QOUT;
      u := Q[n + 1]; Q[n + 1] := Q[u]; Q[u] := 0;
      if last = u then last := n + 1;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
            if Q[v] = 0 then if D[v] = +∞
              then begin Q[last] := last := v; Q[v] := n + 1 end
              else begin Q[v] := Q[n + 1]; Q[n + 1] := v;
                         if last = n + 1 then last := v end;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until Q[n + 1] = n + 1
  end;
```

In this case, $\tilde{c}$ and c_1 are bounded by $O(2^n)$ and, as $c_2 \leqslant nc_1$, the complexity of **L-deque** is $O(n \cdot 2^n)$ (Kershenbaum [27], Shier and Witzgall [39]).

Although characterized by a rather high worst-case complexity, the algorithm **L-deque** is very efficient in practice, mainly when dealing with sparse, almost planar, graphs (Dial et al. [7], Gallo et al. [17], Van Vliet [44]). An interesting experimental finding is that, for sparse graphs, $\tilde{c}$ is independent of n ($\tilde{c} \cong 1.5$) (see table 5 in sect. 10).

The space requirement is $4n + 2m$, as for **L-queue**.

Algorithm **L-2queue**

L-2queue is the algorithm obtained when Q is implemented by means of a 2queue and the insertion policy is similar to the one used in **L-deque**.

```
Procedure L2QUEUE(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞; Q[i] := 0 end;
    P[r] := D[r] := 0; Q[n + 1] := last := r; Q[r] := pntr := n + 1;
    repeat
      comment: QOUT;
      u := Q[n + 1]; Q[n + 1] := Q[u]; Q[u] := 0;
      if pntr = u then pntr := n + 1;
      if last = u then last := n + 1;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
            if Q[v] = 0 then if D[v] = +∞
              then begin Q[last] := last := v; Q[v] := n + 1 end
              else begin Q[v] := Q[pntr]; Q[pntr] := v;
                         if last = pntr then last := v;
                         pntr := v end;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until Q[n + 1] = n + 1
  end;
```

The main difference between **L-deque** and **L-2queue** is in the worst-case computational complexity; in fact, as $\tilde{c} = 0(n^2)$, the complexity for **L 2queue** is $O(n^2 m)$ (Pallottino [34,35]). In practice, the two algorithms behave quite similarly.

Since **L-2queue** (at least in our experimentation) has always proved to be almost as good as **L-deque** without the risk of bad behavior in pathological cases, it can be recommended to risk-averse users.

The space requirement of **L-2queue** is the same as for **L-queue**.

Algorithm **L-threshold**

The interesting idea behind **L-deque** and **L-2queue** is the partitioning of the set of nodes into two subsets and the different processing of the nodes according to which of the two subsets they belong to. The partitioning is dynamically updated at each iteration.

In the papers by Glover et al. [19, 20], the use of the threshold value s is suggested for partitioning Q into two subsets, Q' and Q'': deletions are made at the head of Q', which contains only nodes with labels less than, or equal to, s. Many

different algorithms can be obtained, depending on the choice of: the threshold, the policy for updating its value, the implementation of Q' and Q'', and the way elements are moved from Q' to Q''.

The algorithm **L-threshold** is derived from the threshold algorithm proposed by Glover et al. [19]. The set of candidate nodes Q is partitioned into two subsets Q' and Q'', where Q' is a queue and Q'' is a linked-list. Note that, while in **L-deque** and in **L-2queue** Q' and Q'' were implemented as two sections of a single list, here Q' and Q'' are two distinct linked-lists.

The queue Q' contains all the nodes of Q whose labels are less than, or equal to, the current threshold value *thrs*; the remaining nodes are maintained in Q''.

The procedure QOUT removes the head of Q' if $Q' \neq \emptyset$; when Q' is emptied, the threshold value *thrs* is updated (increased) and Q'' is scanned to move all the nodes with label $\leqslant thrs$ to Q' and then the new head of Q' is removed. A node v is inserted in Q', at the tail, only if $d_v \leqslant thrs$. When a node v currently belonging to Q'' is assigned a new label $d_v \leqslant thrs$, v is moved to Q'. In practice it has been proved to be computationally advantageous to leave a copy of v in Q''; when Q'' is scanned to refresh Q', all the copies are deleted.

To evaluate and update the threshold value, two parameters are computed in procedure QINIT (Glover et al. [19]):

$$s = \min\{m/n,\ 35\},$$

$$t = \begin{cases} x2.\,lmax & \text{if } s \leqslant 7, \\ 7.x2.\,lmax/s & \text{otherwise}; \end{cases}$$

where *lmax* is the maximum arc length and where the value of $x2$ is chosen on the basis of the topology of the graph: Glover et al. suggest $x2 = 1.5$ for grid graphs and $x2 = 0.25$ for random graphs.

```
Procedure LTHRESHOLD(r);
  begin
    comment: TINIT and QINIT;
    for i := to n do begin P[i] := r; D[i] := +∞; Q1[i] := Q2[i] := 0 end;
    P[r] := D[r] := 0; Q1[n + 1] := last := r; Q2[n + 1] := Q1[r] := n + 1;
    s := MIN(m/n,35); t := x2 * lmax; if s > 7 then t := t * 7/s; thrs := -1;
    repeat
      comment: QOUT;
      if Q1[n + 1] = n + 1 then
        begin
          min := +∞; t1 := thrs + t + 1; i := n + 1; j := Q2[i];
          comment: scan Q2[.], compute min, remove copies of nodes
                   already removed, and move nodes from Q2[.] to Q1[.];
```

```
        while j ≠ n + 1 do if D[j] > t1
           then begin comment: update min;
                    if D[j] < min then min := D[j];
                    i := j; j := Q2[i] end
           else begin comment: remove j from Q2[.];
                    Q2[i] := Q2[j]; Q2[j] := 0;
                    if D[j] ≤ thrs then begin Q1[last] := last := j;
                                             Q1[j] := n + 1 end;
                    j := Q2[i] end;
        if Q1[n + 1] ≠ n + 1 then thrs := t1 else
           begin comment: if Q2 = ∅ STOP;
                 if Q2[n + 1] = n + 1 then stop;
                 thrs := min + t; i := n + 1; j := Q2[i];
                 while j ≠ n + 1 do if D[j] > thrs
                    then begin i := j; j := Q2[i] end
                    else begin Q2[i] := Q2[j]; Q2[j] := 0;
                               Q1[last] := last := j; Q1[j] := n + 1;
                               j := Q2[i] end
           end
      end;
    u := Q1[n + 1]; Q1[n + 1] := Q1[u]; Q1[u] := 0;
    if last = u then last := n + 1;
    comment: scan FS(u);
    foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
      begin
        comment: QIN;
        if D[u] + l_uv ≤ thrs
           then if Q1[v] = 0 then begin Q1[last] := last := v;
                                        Q1[v] := n + 1 end
           else if Q2[v] = 0 then begin Q2[v] := Q2[n + 1];
                                        Q2[n + 1] := v end;
        comment: TUPDATE;
        P[v] := u; D[v] := D[u] + l_uv
      end
  until Q1[n + 1] = Q2[n + 1] = n + 1
end;
```

In QINIT we set $thrs = -1$, and in QOUT, when needed, the threshold value is updated on the basis of

$$thrs = \begin{cases} thrs + t + 1 & \text{if } dmin \leqslant thrs + t + 1, \\ dmin + t & \text{otherwise,} \end{cases}$$

where $dmin = \min\{d_v : v \in Q''\}$.

As for complexity, q_0 is $O(n)$ and q_2 is constant; q_1, the cost of QOUT, is a constant if Q' is not empty, otherwise it is $O(n)$ since the linked-list Q'' must be scanned. Note that, since *thrs* cannot decrease and the node labels cannot increase, when a label d_v goes below *thrs* its node v can no longer be inserted in Q''. Then the number of times that Q'' is scanned to refresh Q' is bounded by n, and hence the total cost of "refreshing" operations is bounded by n^2.

Between two successive refreshing operations, **L-threshold** behaves like **L-queue** on a smaller graph, the partial graph (N, A'), with $A' = \{(i, j) \in A : d_i \leqslant thrs\}$. Hence, the number of extractions from Q' for each node is bounded by n; then $\tilde{c} \leqslant n^2$ and the overall cost of QOUT is $O(n^3)$.

The total cost of QIN is $O(m\tilde{c}) = O(n^2 m)$. We may conclude that the complexity of **L-threshold** is $O(n^2 m)$.

Since to implement Q' and Q'' we need two distinct n-arrays, the space requirement for **L-threshold** is $5n + 2m$.

7. Priority queues implementation

As mentioned in sect. 4, a priority queue Q is a collection of elements with each of which is associated a real valued label. We shall denote the label of element $i \in Q$ by $D[i]$. The procedure QOUT returns a minimum label element out of the one currently in the priority queue Q.

The elements of Q can be maintained either in order (*ordered priority queue*) or out of order (*unordered priority queue*). This choice is crucial to the complexity of the algorithms; in fact, when an ordering of the elements is maintained, any operation involving individual elements (change of the label values, insertion or deletion) requires, at least in principle, the updating of the whole priority queue.

For the sake of computational efficiency, we shall consider only implementations of priority queues in which the questions "is v in Q?" and "is Q empty?" can be answered in constant time*.

7.1. UNORDERED LINKED-LIST

The simplest way to implement a priority queue is the *linked-list* (see sect. 5), in which the insertion is made at the head so that only one pointer *frst* is needed. With this implementation, all operations performed on Q but QOUT are the same as the corresponding operations described for the stacks.

QOUT, described next, requires the full scanning of Q in order to select the minimum label element u.

*After the writing of this paper was completed, the paper by M.L. Fredman and R.E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms* (J. ACM 34(1987) 596) has been brought to our attention. In this paper, a new data structure is described which allows a very efficient implementation of priority queues for shortest path algorithms.

```
Procedure QOUT(u);
  begin
    min := +∞; i := n + 1; x := i;
    while Q[i] ≠ n + 1 do
      begin
        if D[Q[i]] < min then begin x := i; min := D[Q[i]] end;
        i := Q[i]
      end;
    u := Q[x]; Q[x] := Q[u]; Q[u] := 0
  end;
```

For a better understanding of the procedure, note that to remove an element u, the element x preceding u in the list must be known.

Let n_q be the maximum cardinality of Q; then QOUT runs in $O(n_q)$ time.

7.2. ORDERED LINKED-LIST

Q can easily be implemented by means of a two-way linked-list UP[.] end DOWN[.], where the elements are sorted on the basis of a non-decreasing order of the label values:

$$D[j] \geqslant D[i] \quad \text{if } j = \text{DOWN}[i],$$

and

$$D[frst] = \min\{d_j : j \in Q\}, \quad \text{where } frst = \text{DOWN}[n+1].$$

QINIT runs in $O(n)$ time and elementary operations DELETE and INSERT have constant cost (see sect. 5.2).

Procedure QOUT, which is described next, has constant complexity.

```
Procedure QOUT(u);
  begin
    u := DOWN[n + 1]; DELETE(u)
  end;
```

Much more expensive is procedure QIN, since we have to determine the position at which the new element is to be inserted.

```
Procedure QIN(v);
  begin
    if DOWN[v] = 0 then pntr := UP[n + 1]
      else begin pntr := UP[v]; DELETE(v) end;
    while D[pntr] > D[v] and pntr ≠ n + 1 do pntr := UP[pntr];
    INSERT (v, pntr)
  end;
```

If $n_q \leqslant n$ is an upper bound on the cardinality of Q, then QIN runs in $O(n_q)$ time.

7.3. BUCKETS

Let f be a monotonic non-decreasing integer function which maps the set of labels onto the set of integers $1, \ldots, K$. We shall assume that the function $f(d_v)$ can be evaluated in unit time.

The kth bucket is defined to be the subset of all the nodes $v \in Q$ such that $f(d_v) = k$ (Denardo and Fox [5], Knuth [29], Tarjan [42]).

A priority queue structured by buckets can be implemented by means of a K-array of pointers $Q[.]$, where $Q[k]$ points to the kth bucket. Each bucket is

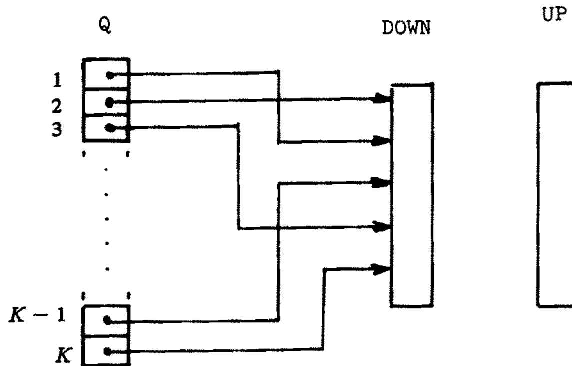


Fig. 8. A buckets data structure.

implemented as an unordered two-way linked-list; since the buckets are disjoint, they can be implemented as just one multiple list by means of a pair of n-arrays UP[.] and DOWN[.]. Note that $Q[.]$ plays the role of the array FRST[.] described in sect. 5.3, while the array LAST[.] is not needed since the additions in our implementation are performed only at the heads of the lists. An additional pointer *pntr* is used, which contains the index of the first non-empty bucket. The overall storage requirement is $K + 2n + 1$.

Procedure QINIT runs in $O(K + n)$ time:

```
Procedure QINIT(r);
  begin
    for i := 1 to K do Q[i] := n + 1;
    for i := 1 to n do UP[i] := DOWN[i] := 0;
    pntr := f(0); Q[pntr] := r; UP[r] := DOWN[r] := n + 1
  end;
```

In the procedure QOUT, we update *pntr* by scanning the array $Q[.]$ to find the first non-empty bucket; then the bucket is scanned to retrieve and remove the minimum label element. Since the algorithms using buckets are intended for graphs with non-negative lengths, we may assume that the minimum labels are monotonically non-decreasing and hence that *pntr* cannot decrease from one iteration to the next in the algorithm. The complexity of QOUT can be split into two parts: the cost of updating *pntr*, plus the cost of retrieving the minimum label element in the bucket.

```
Procedure QOUT(u);
  begin
    while Q[pntr] = n + 1 and pntr < K do pntr := pntr + 1;
    u := Q[pntr]; min := D[u]; i := DOWN[u];
    while i ≠ n + 1 do
      begin if D[i] < min then begin min := D[i]; u := i end;
            i := DOWN[i] end;
    DELETE(u)
  end;
```

Procedure QIN runs in constant time.

```
Procedure QIN(v);
  begin
    if UP[v] ≠ 0 then DELETE(v);
    k := f(D[v]); INSERT(v, k)
  end;
```

Note that DELETE and INSERT are the operations described in sect. 5.2 with only minor modifications due to the fact that here we deal with multiple linked-lists.

7.4. BINARY HEAPS

The binary heap (Williams [46]) is a balanced binary tree where each node points to one of the elements of a set of labeled elements, in such a way that the label of the element pointed to by node i is less than, or equal to, the labels of the

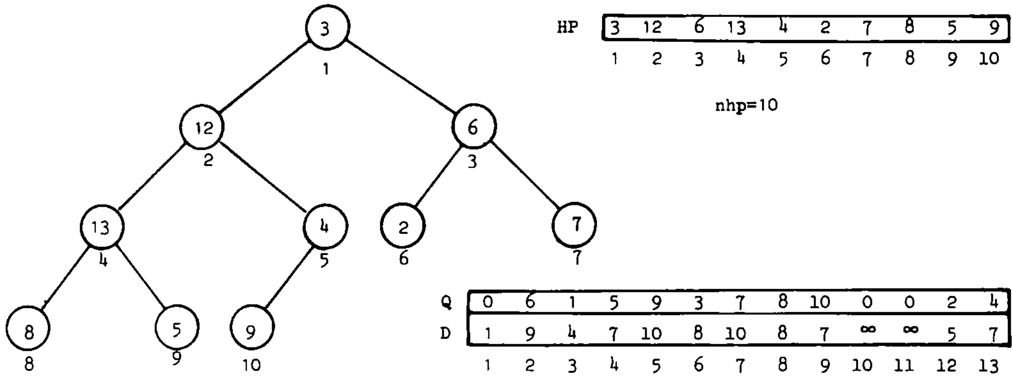


Fig. 9. The binary heap HP[.] and its dictionary Q[.].

elements pointed to by its descendants. Clearly, the root of the tree, node 1, points to a minimum label element.

Binary heaps can be implemented as a pair of n-arrays HP[.] and Q[.], where HP[i] is the element pointed to by node i and $Q[v]$ is the index of the node which points to the element v; if $v \notin Q$, then $Q[v] = 0$. An additional integer, $nhp = |Q|$, is used. The storage requirement is $2n + 1$.

As usual, the nodes of the heap are numbered in such a way that the two sons of a node i are nodes $2i$ and $2i + 1$ (see fig. 9).

```
Procedure QINIT(r);
  begin for i := 1 to n do Q[i] := HP[i] := 0;
        Q[r] := 1; HP[1] := r; nhp := 1
  end;
```

Procedure QINIT runs in O(n) time.

The two procedures QOUT and QIN run in O(log n_q) time, where n_q is the maximum cardinality of Q.

In the former, the element u = HP[1] is removed and the heap is updated: the last element HP[nhp] is placed at the root and is then *moved down* to restore the order of the labels.

In the latter, first we check if v is not already in Q, in which case v is placed in the first available position $nhp + 1$, then v is *moved up* until it reaches its correct position in the heap.

```
Procedure QOUT(u);
  begin
    u := HP[1]; Q[u] := 0; h := HP[nhp]; nhp := nhp - 1;
    if nhp > 0 then
```

```
            begin
               k := 1;
               repeat
                  k2 := 2 * k;
                  if k2 ≤ nhp then
                     begin
                        if k2 < nhp and D[HP[k2 + 1]] < D[HP[k2]]
                           then k2 := k2 + 1;
                        i := HP[k2];
                        if D[i] < D[h] then begin HP[k] := i; Q[i] := k;
                                                  k := k2 end
                     end
               until k ≠ k2;
               HP[k] := h; Q[h] := k
            end
      end;

   Procedure QIN(v);
      begin
         if Q[v] = 0 then begin nhp := nhp + 1; HP[nhp] := v; Q[v] := nhp end;
         k := Q[v]; k2 := INT(k/2);
         while k2 > 0 and D[v] < D[HP[k2]] do
            begin HP[k] := HP[k2]; Q[HP[k2]] := k;
                  k := k2; k2 := INT(k/2) end;
         HP[k] := v; Q[v] := k
      end;
```

8. Shortest-first search algorithms

The first algorithm to use a shortest-first search strategy can be credited both to Dijkstra [8] and to Moore [32], although it was the former who stated formally its properties; actually, any algorithm which uses this strategy can be considered a particular implementation of Dijkstra's original method.

The basic property of these algorithms is the following:

Proposition 1. If $l_{ij} \geqslant 0, \forall (i, j) \in A$, then each node is removed from (and hence inserted into) Q exactly once.

This is due to the fact that, at each step, if u is a minimum label element of Q, then d_u is the shortest distance from r to u, provided that no arc has a negative length. In this case, we have $\tilde{c} = 1$, $c_1 = O(n)$ and $c_2 = O(m)$.

So, the complexity for the shortest-first algorithms when the lengths are non-negative is $O(q_0 + nq_1 + mq_2)$.

Unfortunately, such a nice property no longer holds when the arc lengths are not restricted to being non-negative. In this case, it can be shown (Johnson [24]) that the algorithm might pass through step 2 exponentially many times, and then $c_1 = O(2^n)$ and $c_2 = O(n \cdot 2^n)$.

In the following, we shall restrict ourselves to the case of non-negative lengths.

Algorithm **S-Dijkstra**

S-Dijkstra represents the simplest implementation of Dijkstra's original idea.

Q is an unordered linked-list. As shown in sects. 5.1 and 7.1, q_0 is $O(n)$, q_1 is $O(n_q) \leq O(n)$ and q_2 is constant. In this case, the algorithm runs in $O(n^2)$ time.

Its space requirement is $4n + 2m$.

```
Procedure SDIJKSTRA(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞; Q[i] := 0 end;
    P[r] := D[r] := 0; Q[n + 1] := r; Q[r] := n + 1;
    repeat
      comment: QOUT;
      min := +∞; i := n + 1; x := i;
      while Q[i] ≠ n + 1 do
        begin
          if D[Q[i]] < min then begin x := i; min := D[Q[i]] end;
          i := Q[i]
        end;
      u := Q[x]; Q[x] := Q[u]; Q[u] := 0;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
          if Q[v] = 0 then begin Q[v] := Q[n + 1]; Q[n + 1] := v end;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until Q[n + 1] = n + 1
  end;
```

Algorithm **S-ord**

The algorithm **S-ord** makes use of an ordered two-way linked-list. To improve its efficiency, a pre-processing phase is performed where the forward stars are sorted in decreasing order of arc lengths (Simeone [40]). In doing so, we obtain the two following advantages:

(i) to perform the operations QIN on the ending nodes of the arcs in FS(u), the list must be scanned only once from the tail (highest label element) to the head;

(ii) let (u, v) and (u, w) be successive arcs in FS(u), and assume that, after having performed operation QIN(v) we also update the label of node w from the old value d_w to the new value $d'_w = d_u + l_{uw}$. Then, since FS(u) has been sorted, $d'_w \leq \min\{d_v, d_w\}$.

These facts can be exploited to "skip" uninteresting portions of the list when QIN(w) is performed.

The complexity of the pre-processing phase is $O(mn)$; in fact, in our implementation each forward star is sorted by means of a bubble-sort algorithm.

Now we derive the complexity of the algorithm without considering the pre-processing phase. This is of interest when the algorithm must be used many times on the same set of data. The cost q_0 is $O(n)$ and q_1 is constant. Since all the QIN operations relative to the same forward star cost $O(n)$ because of (i), the overall cost due to QIN is $O(n^2)$. Hence, the complexity of **S-ord** is $O(n^2)$.

The space requirement is $5n + 2m$.

```
Procedure SORD(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞;
                             UP[i] := DOWN[i] := 0 end;
    P[r] := D[r] := 0; UP[n + 1] := DOWN[n + 1] := r;
                       UP[r] := DOWN[r] := n + 1;
    D[n + 1] := -∞;
    repeat
      comment: QOUT;
      u := DOWN[n + 1];
      UP[DOWN[u]] := n + 1; DOWN[n + 1] := DOWN[u];
      DOWN[u] := UP[u] := 0; pntr := UP[n + 1];
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
          if DOWN[v] ≠ 0 then
```

```
                begin UP[DOWN[v]] := UP[v]; DOWN[UP[v]] := DOWN[v];
                      if D[v] < D[pntr] then pntr := UP[v] end;
             delta := D[u] + l_uv;
             while D[pntr] > delta do pntr := UP[pntr];
             DOWN[v] := DOWN[pntr]; UP[DOWN[v]] := v;
             DOWN[pntr] := v; UP[v] := pntr;
             comment: TUPDATE;
             P[v] := u; D[v] := delta
          end
    until DOWN[n + 1] = n + 1
end;
```

Algorithm **S-Dial**

The algorithm **S-Dial** (Dial [6]) makes use of the buckets to implement Q. Without loss of generality, we can consider the arc-length (non-negative) integers. Let $lmax = \max\{l_{ij} : (i,j) \in A\}$.

The function $f(d_v)$ is: $f(d_v) = d_v + 1$, so that the first non-empty bucket contains only minimum label nodes; hence, in operation QOUT, the scanning of the bucket is not necessary.

Moreover, the dimension of the vector $Q[.]$, which is crucial to the storage requirement of the algorithm, can be reduced by means of the following consideration: if u is a minimum label element of Q, then, for each $v \in Q$, $d_u \leqslant d_v \leqslant d_u + lmax$. Hence, the function f introduced can be replaced by the new function $f' = f.\mathrm{mod}(lmax + 1)$, which is implemented by means of:

$$f'(d_v) = \begin{cases} f'(d_u) + l_{uv} & \text{if } f'(d_u) + l_{uv} \leqslant lmax + 1, \\ f'(d_u) + l_{uv} - lmax - 1 & \text{otherwise;} \end{cases}$$

and the dimension of $Q[.]$ is reduced to $K = lmax + 1$. The effect of using the latter function f' is that each bucket is used several times.

The overall complexity of QOUT is $O(dmax)$, where $dmax = \max\{d_v : v \in N\} \leqslant n.\,lmax$.

As $q_0 = O(n + lmax)$ and q_2 is constant, the complexity of **S-Dial** is $O(m + n.\,lmax)$.

The space requirement is $5n + 2m + lmax$.

```
Procedure SDIAL(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞;
                             UP[i] := DOWN[i] := 0 end;
    for i := 1 to lmax + 1 do Q[i] := n + 1;
    P[r] := D[r] := 0; Q[1] := r; UP[r] := -1; DOWN[r] := n + 1;
    cntr := pntr := 1;
    repeat
      comment: QOUT;
      while Q[pntr] = n + 1 do if pntr > lmax then pntr := 1
                                              else pntr := pntr + 1;
      u := Q[pntr]; Q[pntr] := DOWN[u];
      UP[DOWN[u]] := -pntr; UP[u] := DOWN[u] := 0; cntr := cntr - 1;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
          if DOWN[v] := 0 then cntr := cntr + 1
            else
              begin
                if UP[v] > 0 then DOWN[UP[v]] := DOWN[v]
                             else Q[-UP[v]] := DOWN[v];
                UP[DOWN[v]] := UP[v]
              end;
          x := pntr + l_uv;
          if x > lmax + 1 then x := x - lmax - 1;
          DOWN[v] := Q[x]; UP[DOWN[v]] := v;
          Q[x] := v; UP[v] := -x;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until cntr = 0
  end;
```

Algorithm **S-heap**

In **S-heap**, the priority queue is a partially ordered set implemented by means of a *binary heap* (see sect. 7.4). The operations QOUT and QIN run in $O(\log n_q)$. Then the complexity of **S-heap** is $O(m \,.\log n)$ (D.B. Johnson [25] and E.L. Johnson [26]).

```
Procedure SHEAP(r);
  begin
    comment: TINIT and QINIT;
    for i := 1 to n do begin P[i] := r; D[i] := +∞;
                             HP[i] := Q[i] := 0 end;
    P[r] := D[r] := 0; Q[r] := nhp := 1; HP[1] := r;
    repeat
      comment: QOUT;
      u := HP[1]; Q[u] := 0; h := HP[nhp]; nhp := nhp − 1;
      if nhp > 0 then
        begin
          k := 1;
          repeat
            k2 := 2 * k;
            if k2 ≤ nhp then
              begin
                if k2 < nhp and D[HP[k2 + 1]] < D[HP[k2]]
                  then k2 := k2 + 1;
                i := HP[k2];
                if D[i] < D[h] then
                  begin HP[k] := i; Q[i] := k; k := k2 end
              end
          until k ≠ k2;
          HP[k] := h; Q[h] := k
        end;
      comment: scan FS(u);
      foreach (u, v) ∈ FS(u) do if D[u] + l_uv < D[v] then
        begin
          comment: QIN;
          if Q[v] = 0 then begin nhp := nhp + 1; HP[nhp] := v;
                                 Q[v] := nhp end;
          k := Q[v]; k2 := INT(k/2);
          while k2 > 0 and D[v] < D[HP[k2]] do
            begin HP[k] := HP[k2]; Q[HP[k2]] := k;
                  k := k2; k2 := INT(k/2) end;
          HP[k] := v; Q[v] := k;
          comment: TUPDATE;
          P[v] := u; D[v] := D[u] + l_uv
        end
    until nhp = 0
  end;
```

Note that quite often in sparse graphs, m is $O(n)$, and the complexity becomes $O(n \,.\log n)$.

The space requirement is $5n + 2m$.

9. The all pairs problem

In most real-life models, the requirement is to find the shortest distances between all pairs of nodes. Algorithms exist which solve this problem directly. One of them, very ingenious indeed, is due to Floyd [11] and is based on a result by Warshall [45]. The underlying idea is rather simple: let $d_{ij}^{(h)}$ be the length of the shortest path from i to j, subject to the condition that the path does not pass through any of the nodes $h, h+1, \ldots, n$ (i and j except); then we have

$$d_{ij}^{(1)} = \begin{cases} l_{ij}, & \text{if } (i,j) \in A, \\ 0, & \text{if } i = j, \\ +\infty, & \text{otherwise;} \end{cases}$$

$$d_{ij}^{(h+1)} = \min\left\{d_{ij}^{(h)},\ d_{ih}^{(h)} + d_{hj}^{(h)}\right\}.$$

The array of predecessor nodes, where p_{ij} is the predecessor of node j in the shortest path from i to j, is (Hu [22]):

$$p_{ij}^{(1)} = \begin{cases} i, & \text{if } (i,j) \in A, \\ 0, & \text{otherwise;} \end{cases}$$

$$p_{ij}^{(h+1)} = \begin{cases} p_{ij}^{(h)}, & \text{if } d_{ij}^{(h+1)} = d_{ij}^{(h)}, \\ p_{hj}^{(h)}, & \text{otherwise.} \end{cases}$$

Clearly, $d_{ij}^{(n+1)}$ is the shortest distance between nodes i and j. The complexity of this algorithm is $O(n^3)$, whatever the density of the graph.

```
Procedure FLOYD(n);
  begin
    comment: initialization;
    for i := 1 to n do for j := 1 to n do
      if (i, j) ∈ A then begin D[i, j] := l_ij; P[i, j] := i end
                    else begin D[i, j] := +∞; P[i, j] := 0 end;
    for i := 1 to n do D[i, i] := 0;
    comment: paths computation;
    for h := 1 to n do for i := 1 to n do for j := 1 to n do
      if D[i, j] > D[i, h] + D[h, j] then
        begin D[i, j] := D[i, h] + D[h, j]; P[i, j] := P[h, j] end
  end;
```

A major drawback of this algorithm is its high storage requirement, $O(n^2)$, which prevents it being used in large-scale models. The need for a storage space of the order of n^2 is a characteristic shared by all direct approaches to the *all-pairs shortest path problem* (Dantzig [4], Tabourier [41]).

This explains why the approach most often used in practice consists of carrying out a sequence of n distinct **spt** computations. This *sequential approach* may seem rather redundant, but can in fact be implemented efficiently (Bazaraa and Langley [2]).

The problem of finding the **spt** rooted at $s \neq r$, $T^*(s)$, once $T^*(r)$ is known, is simply a particular reoptimization problem, as shown in the paper by Gallo [13]. An efficient sequential solution procedure can be devised making use of the transformation

$$\tilde{l}_{ij} = l_{ij} + d_{ri} - d_{rj} \geqslant 0, \quad (i, j) \in A;$$

where d_{ri} is the shortest distance from the old root r to node i.

It is easy to see that such a transformation does not affect the shortest path tree $T(s)$, since its effect is to change the length of each path by a constant (Edmonds and Karp [9], Gallo [14], Nemhauser [33]). Once the shortest distances with transformed lengths have been found, the true distances are determined by means of the inverse transformation

$$d_{sv} = \tilde{d}_{sv} - d_{rs} + d_{rv}, \quad v \in N,$$

where $\tilde{d}_{sv}$ is the shortest distance from s to v with the transformed lengths l.

The resulting procedure is:

Procedure SA (Sequential All pairs) (Gallo and Pallottino [15])

Step 1. Find $T^*(1)$ and d_{1v}, $v \in N$, making use of any list-search algorithm. Set $k = 1$.

Step 2. Set $\tilde{l}_{ij} = l_{ij} + d_{ki} - d_{kj}, (i, j) \in A$; set $k = k + 1$.

Step 3. Find $T^*(k)$ and $\tilde{d}_{kv}$, $v \in N$, making use of any shortest-first search algorithm.

Step 4. Set $d_{kv} = \tilde{d}_{kv} - d_{(k-1)k} + d_{(k-1)v}$, $v \in N$.
If $k < n$, then go to step 2; else STOP.

Unlike the Floyd–Warshall method, this procedure needs a storage space linear with m and n. If **L-queue** and **S-heap** are used at step 1 and at step 3, respectively, the complexity of SA is $O(m \cdot n \cdot \log n)$ (Lawler [31]), which for sparse graphs can be less than the complexity of the Floyd–Warshall method. Moreover, while the actual number of operations for this method is always of the order of n^3, this is not the case for SA for which, in practical problems, the complexity is often far below its worst-case figure.

Note that at step 3, an **spt** problem is solved where the distribution of the arc lengths is very peculiar indeed. In fact, a large number of arcs have zero length (at least all the $n - 1$ arcs of $T^*(k-1)$ since for such arcs $d_{(k-1)j} - d_{(k-1)i} = l_{ij}$), while some others may have rather large lengths depending on the relative value of $d_{(k-1)i}$ and $d_{(k-1)j}$. These facts can be exploited to speed up the algorithm.

Moreover, a bound to the longest shortest path can easily be determined:

Proposition 2 (Gallo [13])

$$D_k = \max\{d_{kv} : v \in N\} = \tilde{d}_{k(k-1)} = d_{k(k-1)} + d_{(k-1)k}.$$

D_k, at least for graphs without zero length cycles, is a norm and can be considered as a measure of the "distance" between the kth and $(k-1)$th problems. Then, the computational complexity of step 3 is $O(m + D_k)$.

Similar results, although slightly more intricate, can be obtained with the other shortest-first search algorithms.

From these considerations, it follows that the ordering of the nodes (origins) is not immaterial to the computational complexity of SA.

To maximize the efficiency of SA, one should try to order the nodes in such a way that nodes $k - 1$ and k be as close as possible. This can be done either with an "apriori" ordering, or in an "adaptive" way, i.e. by selecting at each step, on the basis of the current shortest distances, the node to be considered next. An "apriori" ordering calls for a pre-processing of the graph, which can be done by means of any heuristic algorithm for the travelling saleman problem; this is particularly advantageous when many all-pairs problems are to be solved on the same graph.

In the following, an implementation of SA is described (Gallo and Pallottino [15]).

Algorithm PSA (Primal Sequential All pairs)

This algorithm is based on two facts which derive from the considerations developed before and in proposition 2:

(i) in the execution of algorithm SPT, at step 3 of SA, it is likely that more than one element with a minimum label exists in Q (this happens, for instance, when a node u is selected, whose forward star contains more than one arc with zero transformed length);

(ii) a node $v \in Q$ with label $d_v > D_k$ cannot be a candidate for selection at step 2 of SPT.

Then we may partition Q into three sets Q', Q'' and Q''', and the insertion of a node v at step 3 of SPT is carried out according to the following rule, where D is an estimate of D_k:

(1) if $d_v = d_u$ (i.e. $\tilde{l}_{uv} = 0$), then v is inserted into Q' (and deleted from Q'' or from Q''' if already in Q);

(2) if $d_u < d_v \leqslant D$, then v is inserted into Q'' (and deleted from Q''' if necessary);

(3) if $d_v > D$, then v is inserted into Q'''.

The sets Q' and Q''' are implemented as linked-lists, whereas the set Q'' is implemented as a priority queue. Each time the set Q' is empty, all the minimum label elements of Q'' are moved into Q'. When Q' and Q'' are empty and Q''' is not, D is increased and all the elements $v \in Q'''$ with d_v less than, or equal to, the new estimate D are moved into Q' or into Q''. A reasonable initial estimate of D_k is $2d_{(k-1)k}$ (Gallo [13], Gallo and Pallottino [15]).

Note that when node $k-1$ (the previous origin) is inserted into Q', the exploration of Q can be interrupted. In fact, any node v not yet inserted into Q' can be reached from $k-1$ at zero (transformed) distance, and a shortest path from k to v passes through node $k-1$. Hence, provided that additional bookkeeping is carried out to update the predecessor vector, it is easy to determine the new **spt** $T^\star(k)$. The main advantage of this fact is that each shortest path tree computation in SA (except for the first one) does not necessarily pass through the examination of all the arcs; thus, its complexity is $\Omega(n)$ and not $\Omega(m)$ as in the general case.

The computational complexity of this implementation of SA depends on the particular data structure chosen to implement the priority queue, and in any case it can be bounded by $O(n^3)$.

The application of PSA and an analogous algorithm based on a dual approach (Florian et al. [10]) to some real urban transportation problems has shown that about 10% of the nodes and arcs do not need to be examined at each iteration; moreover, up to 90% of the examined nodes are inserted directly into Q'. The solution was attained with computer times from 20% to 35% lower than the times needed by **L-deque** or by **S-Dial**.

10. Numerical experimentation

Now we describe the experimentation performed on a rather broad set of test problems, in order to achieve a deeper understanding of the algorithms' behavior and to assess their relative efficiency.

The experimentation reported refers to the **spt** algorithms described in sects. 6 and 8 and is part of wider ranging experimentation illustrated in Gallo et al. [17].

The codes we used are an almost "one-to-one" FORTRAN-ANSI implementation of the pidgin Pascal description of the algorithms. The only modifications, which regard minor points in the implementation of the data structures, were made for reasons of computational efficiency. For this reason, we do not include in the text the source lists, which of course can be obtained together with the test problem generators from the enclosed diskette.

The graphs used in the experimentation can be partitioned into two major classes: *complete* and *sparse* graphs.

Due to the high growth rate of the storage requirement, only relatively small-size (up to a maximum of 175 nodes) complete graphs were generated.

Three types of sparse graphs were considered: *random graphs, k-linear graphs* and *grid graphs.* Graphs going from 1000 of nodes and 10 000 arcs up to 3000 and 30 000 arcs were generated.

In random graphs, the arcs were generated randomly, without repetitions, in order to achieve assigned density values.

k-linear graphs (see sect. 10.3) are particularly structured sparse graphs which represent real-life models quite well, such as a large class of transportation networks.

The grid graphs, which have been widely used in previous experimentations (Gilsinn and Witzgall [18], Dial et al. [7]), can be regarded as a particular case of k-linear graphs.

Integer arc lengths were generated randomly, with a uniform distribution between 0 and an assigned maximum positive value *lmax*.

As expected, the only algorithm whose behavior is noticeably affected by the arc-length range is **S-Dial** which, out of the shortest-first algorithms, is the fastest for small values of *lmax* (100 in our experimentation). When *lmax* increases, **S-Dial** slows down strongly.

Out of the list-search algorithms, **L-threshold** turns out to be the fastest. Out of the other algorithms in this class, **L-deque** and **L-2queue,** which behave very much alike, out-perform **L-queue** for well structured graphs, k-linear and grid graphs, while **L-queue** is usually faster for complete and random graphs. It can be noted that these experimental findings go against what is suggested by the worst-case computational complexity.

The efficiency of **L-threshold** is mainly due to the choice of the parameters s, t and $x2$ (see sect. 6), determined by the authors (Glover et al. [19]) by means of a very great deal of experimentation. Since the parameter values depend on the particular

topology of the graph and on *lmax*, a tuning-up phase might be needed in some applications to achieve the maximum efficiency of the algorithm.

Although slower, **L-deque** and **L-2queue**, whose behavior is quite stable with respect to the input data, are fast enough to be considered a good choice in most applications. Out of the two, risk-averse users might prefer the latter, which is polynomially bounded.

Strong points for some users are the simplicity of implementation and the storage requirement.

L-queue, L-deque and **L-2queue**, which are based on linked lists with simple handling operations, are easy to implement with a small number of statements. More complicated are **S-Dijkstra** and **S-ord** because of the operations performed on the linked-lists.

S-heap, S-Dial and **L-threshold** require more sophisticated data structures and are the most complex to implement out of the **spt** algorithms described in this paper. Hence, their implementations are characterized by a rather large number of statements.

Table 1

Memory requirement

Storage requirement	Algorithms
$4n + 2m$	S-Dijkstra, L-queue, L-deque, L-2queue
$5n + 2m$	S-ord, S-heap, L-threshold
$5n + 2m + lmax$	S-Dial

In table 1, the storage requirement of the different algorithms is summarized. Remember that in assessing overall memory occupation one should consider, in addition to the space for the data, given in the table, the in-core memory needed by the program itself, which usually increases with the number of statements.

We now give a detailed description of the experimentation. In the tests, two ranges, from which the arc lengths have been uniformly drawn, were used: [0 ÷ 100] and [0 ÷ 10 000]. Since all the algorithms but **S-Dial** are unaffected by *lmax*, we report only the results obtained with *lmax* = 100 except for **S-Dial**, for which we fully report on all the tests performed.

The CPU times in $\sec \cdot 10^{-3}$ given in the tables and in the figures are the mean values over ten runs, each differing from the others only as far as the origin of the paths is concerned.

The computer used was an IBM 3033 N08 under VM/CMS Operating System and a FORTGI compiler.

10.1. COMPLETE GRAPHS

Four different complete graphs were generated with $n = 25, 75, 125$ and 175.

We ran **L-threshold** with different values of x, obtaining the CPU time reported in table 2, which suggests that $x = 0.25$ is a good choice for complete graphs.

Table 2

CPU times for $n = 175$

$x2$	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
time	72	90	100	104	108	109	112	113

Table 3

Complete graphs

Algorithms	$n = 25$	$n = 75$	$n = 125$	$n = 175$
S-Dijkstra	2.2	20	55	107
S-ord	2	52	217	573
S-heap	2	14	37	71
S-Dial ($lmax = 10^2$)	2	14	37	71
S-Dial ($lmax = 10^4$)	20.8	31	54	89
L-queue	2.2	23	62	125
L-deque	2.4	32	83	186
L-2queue	2.4	31	81	179
L-threshold ($lmax = 10^2$) ($x2 = 0.25$)	1.8	14	38	72
L-threshold ($lmax = 10^4$) ($x2 = 0.25$)	1.8	15	40	94

The results of the experimentation reported in table 3 show that (only) in this case the fastest algorithm is **S-heap**.

The behavior of the algorithms on the basis of the data of table 3 is summarized in fig. 10.

The relatively disappointing performance of the list-search algorithms (except **L-threshold**) can be credited to the following fact: the number of label updates performed in these algorithms increases with the number of alternative paths with different lengths from the root to each node. A large number of such paths may produce too many trials before hitting on the right label. Of course, it is in complete graphs that we have a maximum number of alternative paths.

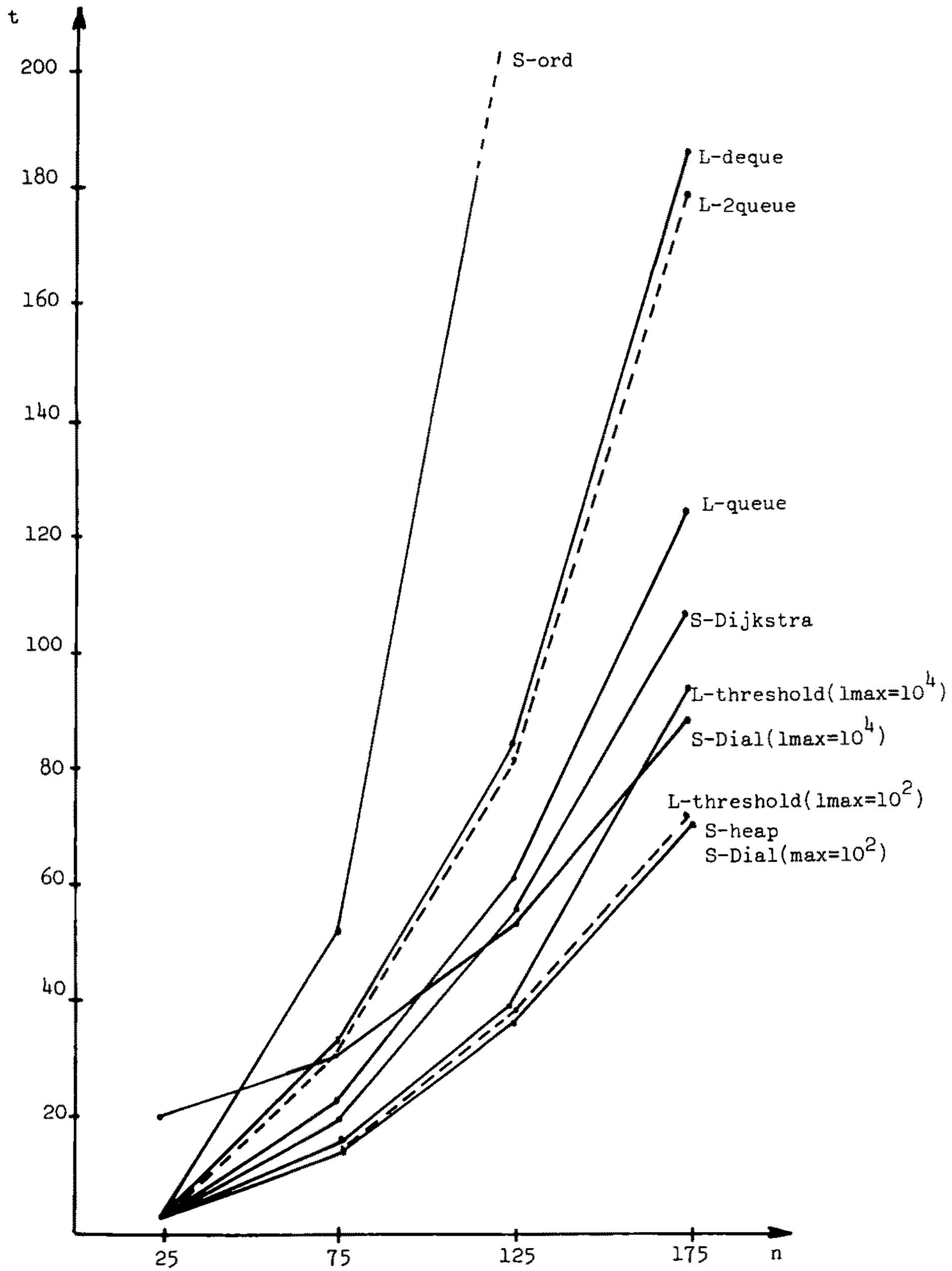


Fig. 10. Complete graphs.

Table 4

Complete graphs: $lmax = n^2$

Algorithms	$n = 25$	$n = 75$	$n = 125$	$n = 175$
S-Dijkstra	2.4	20	55	108
S-ord	2.2	53	219	576
S-heap	2.2	15	39	72
S-Dial	3	24	64	123
L-queue	2.2	29	73	162
L-deque	2.4	40	109	284
L-2queue	2.6	39	105	258
L-threshold ($x2 = 0.25$)	1.8	15	40	94

These considerations are backed up by the experimental findings reported in table 4, where we ran the same problems as before with increasing *lmax* values, namely $lmax = n^2$. Clearly, the effect of increasing the arc-length range is to increase the probability of distinct paths having different lengths. In any case, the effect of the arc-length range on the behavior of list-search algorithms is relatively small.

10.2. RANDOM GRAPHS

Four different random graphs were generated with n = 1000, 3000 and m = 10 000, 30 000; ($lmax$ = 100 for all the algorithms except for **S-Dial**, for which the value 10 000 was also used).

Table 5

Random graphs

Algorithms	n = 1000, m = 10 000	n = 1000, m = 30 000	n = 3000, m = 10 000	n = 3000, m = 30 000
S-Dijkstra	1180	1420	6017	9506
S-ord	163	333	597	1106
S-heap	72	121	171	226
S-Dial ($lmax = 10^2$)	39	88	64	112
S-Dial ($lmax = 10^4$)	76	238	153	163
L-queue	55	161	70	183
L-deque	57	186	65	193
L-2queue	58	186	64	193
L-threshold ($x2 = 0.25$)	36	83	64	105

From the results reported in table 5, the following considerations derive:

- as anticipated, **L-threshold** is the fastest algorithm;
- the behavior of algorithms **S-Dijkstra** and **S-ord** is more affected by the number of the nodes than by the number of the arcs, which is reasonable since their theoretical complexity is $O(n^2)$;
- the other algorithms are more affected by the number of the arcs; this is particularly true for the list-search algorithms.

10.3. k-LINEAR GRAPHS

k-linear graphs are layered graphs where the nodes are partitioned into k subsets, $N_1, N_2, \ldots, N_k$, with arcs connecting either nodes of the same subset or nodes belonging to adjacent subsets.

In the experimentation, we always chose

$$2|N_1| = |N_2| = \ldots = |N_{k-1}| = 2|N_k| = n/(k-1).$$

As for arcs, which were randomly generated, two types of k-linear graphs were used:

(i) the *stable* ones, where no arc exists between nodes in the same subset;
(ii) the *unstable* ones, where such arcs are allowed.

More details on the way these arcs were generated can be obtained from the comments of the generators source lists in the diskette.

For each of these two types of graphs, four different topologies were generated with 2500 nodes, 30 000 arcs and $k = 10, 50, 100, 200$. Again, we have set $lmax = 100$, and $lmax = 10\,000$ only for **S-Dial**.

From the results reported in table 6, one can observe that:

- **L-threshold** strictly dominates all the other algorithms;
- except for **L-queue**, the list-search algorithms are unaffected by k;
- as backed up also be experimental findings not reported here, **L-queue** slows down when the diameter of the graph increases, which in this case is bounded from below by $k-1$;
- except for **S-Dial**, the effect of increasing k is to decrease (although only slightly) the running times of the shortest-first search algorithms; this is due to the fact that as the diameter increases, the maximum number of elements currently in the priority queue decreases (for a given size of graphs);
- the behavior of **S-Dial** is explained by that fact that, as reported in sect. 8, the number of operations performed linearly increases with $dmax$, which in turn increases with k.

Table 6

k-linear graphs: n = 2500, m = 30 000

Algorithms	k = 10	k = 50	k = 100	k = 200
S-Dijkstra	6145	1282	649	349
S-ord	734	243	186	160
S-heap	206	176	164	152
S-Dial ($lmax$ = 10^2)	113	106	105	107
S-Dial ($lmax$ = 10^4)	157	185	243	371
L-queue (stable graphs)	194	241	297	292
L-queue (unstable graphs)	196	434	360	1208
L-deque	159	117	110	105
L-2queue	159	118	111	106
L-threshold ($x2$ = 0.25)	104	98	97	97

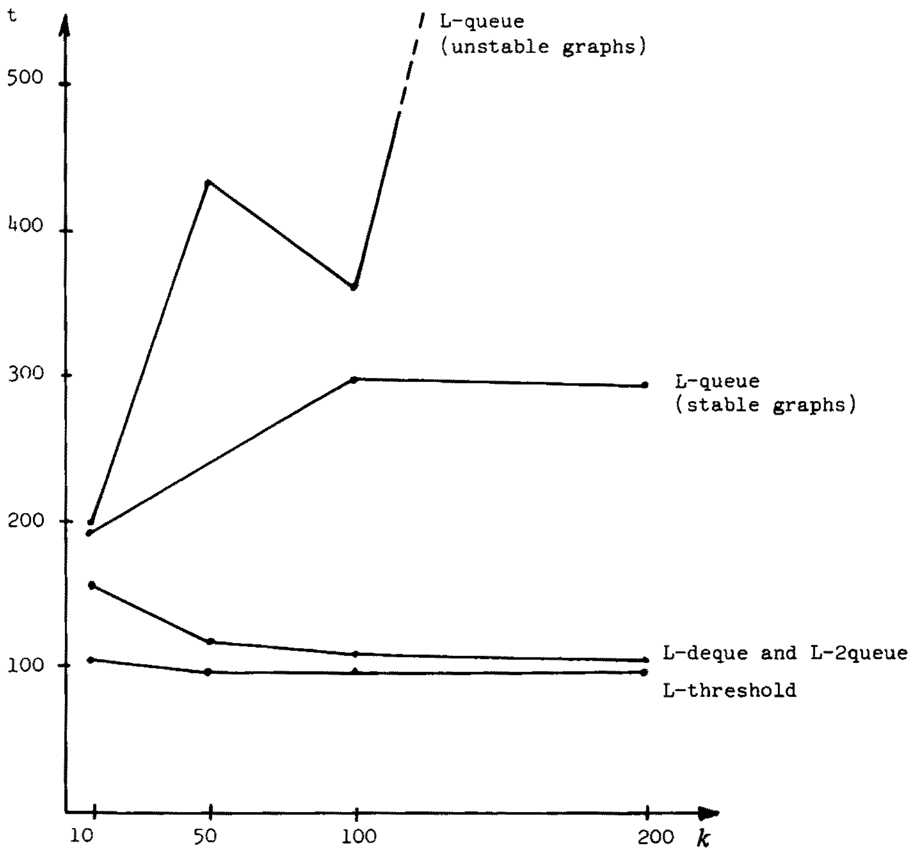


Fig. 11. k-linear graphs.

Table 7

CPU times for n = 2500, m = 30 000, $lmax$ = 10 000, k = 200

$x2$	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
time	98	103	111	122	130	142	142	149

In table 6, the results on unstable graphs have been listed only for **L-queue** because all others are almost unaffected by the stability of the graph.

The behavior of the list-search algorithms is summarized in fig. 11. In table 7, the experimentation which led to the choice of the table value of $x2$ is reported.

10.4. GRID GRAPHS

Four grid graphs with 2500 nodes and about 10 000 arcs were generated by varying the number of rows (nr) and columns (nc) of the grid.

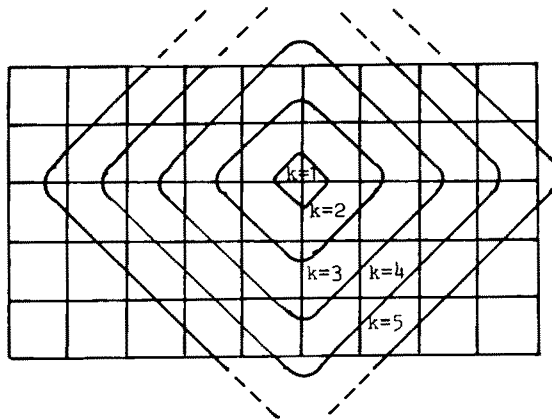


Fig. 12. A grid graph.

The results (see table 8) are consistent with the results obtained with k-linear graphs. In fact, as illustrated by fig. 12, a grid is a particularly stable k-linear graph, where k increases with $\max\{nc, nr\}$.

In addition to the considerations of the previous section, we only note that:

- the effect of $lmax$ on the behavior of **S-Dial** is particularly evident;
- the behavior of **L-deque**, **L-2queue** and **L-threshold** is almost indistinguishable.

Here, the value of $x2$ in **L-threshold** has been set equal to 1, according to the experimental results of table 9.

Table 8

Grid graphs: $n = 2500$

Algorithms	$nr = 50$ $nc = 50$	$nr = 25$ $nc = 100$	$nr = 10$ $nc = 250$	$nr = 5$ $nc = 500$
S-Dijkstra	582	407	176	108
S-ord	111	92	67	59
S-heap	110	104	88	75
S-Dial ($lmax = 10^2$)	53	53	57	67
S-Dial ($lmax = 10^4$)	348	464	1054	2094
L-queue	95	146	336	557
L-deque	43	46	48	45
L-2queue	44	46	49	44
L-threshold ($x2 = 1$)	48	47	45	42

Table 9

CPU times for $n = 2500$

$x2$	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
$nr = 50, nc = 50$	50	47	47	48	48	47	49	50
$nr = 5, nc = 500$	47	44	43	42	43	43	43	44

Acknowledgements

The authors are indebted to Carlo Ruggeri for his valuable contribution in carrying out the numerical experimentation.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Mass., 1974).

[2] M.S. Bazaraa and R.W. Langley, A dual shortest path algorithm, SIAM J. Appl. Math. 26, 3(1974)496.

[3] R. Bellman, On a routing problem, Quart. Appl. Math. 16(1958)88.

[4] G.B. Dantzig, All shortest routes in a graph, Theory of Graphs, Int. Symp., Rome 1966 (Dunod, Paris, 1967) pp. 91–92.

[5] E.V. Denardo and B.L. Fox, Shortest-route methods. I: Reaching, pruning, and buckets, Oper. Res. 27, 1(1979)161.

[6] R.B. Dial, Algorithm 360: Shortest path forest with topological ordering, Commun. A.C.M. 12, 11(1969)632.

[7] R.B. Dial, F. Glover, D. Karney and D. Klingman, A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees, Networks 9, 3(1979)215.
[8] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1(1959)269.
[9] J. Edmonds and R.M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J. A.C.M. 10, 2(1972)248.
[10] M. Florian, S. Nguyen and S. Pallottino, A dual simplex algorithm for finding all shortest paths, Networks 11 (1981)367.
[11] R.W. Floyd, Algorithm 97: Shortest path, Commun. A.C.M. 5(1962)345.
[12] L.R. Ford, Jr., Network flow theory, Rand Corporation Report No. P-293 (1956).
[13] G. Gallo, Reoptimization procedures in shortest path problems, Rivista di Matematica e di Scienze Economiche e Sociali 3, 1(1980)3.
[14] G. Gallo, Updating shortest paths in large-scale networks, paper presented at the Int. Workshop on Advances in Linear Optimization Algorithms and Software, Pisa, Italy (1980).
[15] G. Gallo and S. Pallottino, A new algorithm to find the shortest paths between all pairs of nodes, Discr. Appl. Math. 4(1982)23.
[16] G. Gallo and S. Pallottino, Shortest path methods: A unifying approach, Mathematical Programming Study 26(1986)38.
[17] G. Gallo, S. Pallottino, C. Ruggeri and G. Storchi, Metodi ed algoritmi per la determinazione di cammini minimi, Monografie di Software Matematico, N.29 (1984).
[18] J. Gilsinn and C. Witzgall, A performance comparison of labeling algorithms for calculating shortest path trees, Natl. Bureau of Standards, Technical Note N.772 (1973).
[19] F. Glover, R. Glover and D. Klingman, Computational study of an improved shortest path algorithm, Networks 14(1984)25.
[20] F. Glover, D. Klingman and N. Phillips, A new polynomially bounded shortest path algorithm, Oper. Res. 33(1985)65.
[21] E. Horowitz and S. Sahni, *Fundamentals of Data Structures* (Pitman, London, 1976).
[22] T.C. Hu, Revised matrix algorithms for shortest paths, SIAM J. Appl. Math. 15(1967)207.
[23] D.B. Johnson, Algorithms for shortest paths, Ph.D. Dissertation, Cornell University, Report No. tr-73-169 (1973).
[24] D.B. Johnson, A note on Dijkstra's shortest path algorithm, J. A.C.M. 20, 3(1973)385.
[25] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. A.C.M. 24 1(1977)1.
[26] E.L. Johnson, On shortest paths and sorting, Proc. 25th A.C.M. Annual Conference (1972) pp. 510–517.
[27] A. Kershenbaum, A note on finding shortest path trees, Networks 11(1981)399.
[28] D.E. Knuth, *The Art of Computer Programming,* Vol. 1: *Fundamental Algorithms* (Addison-Wesley, Reading, Mass., 1968).
[29] D.E. Knuth, *The Art of Computer Programming,* Vol. 3: *Sorting and Searching* (Addison-Wesley, Reading, Mass., 1973).
[30] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids* (Holt, Rinehart and Winston, New York, 1976).
[31] E.L. Lawler, Shortest path and network flow algorithms, Ann. Discr. Maths. 4(1979)251.
[32] E.F. Moore, The shortest path through a maze, Proc. Int. Symp. on Theory of Switching, part 2, Harvard University Press (1959) pp. 285–292.
[33] G.L. Nemhauser, A generalized permanent label setting algorithm for the shortest path between specified nodes, J. Math. Analysis and Appl. 38(1972)328.
[34] S. Pallottino, Adaptation de l'algorithme de d'Esopo-Pape pour la determination de tous les chemins les plus courts: ameliorations et simplifications, CRT, University of Montreal, Publ. No. 136 (1979).

[35] S. Pallottino, Shortest path methods: Complexity, interrelations and new propositions, Networks 14(1984)257.
[36] U. Pape, Implementation and efficiency of Moore algorithms for the shortest route problem, Math. Progr. 7(1974)212.
[37] U. Pape, Algorithm 562: Shortest path lengths, A.C.M. Transactions on Mathematical Software 6(1980)450.
[38] U. Pape, Remark on algorithm 562, A.C.M. Transactions on Mathematical Software 9, 2(1983)260.
[39] D.R. Shier and C. Witzgall, Properties of labeling methods for determining shortest path trees, J. Res. Natl. Bureau of Standards 86, 3(1981)317.
[40] B. Simeone, Private communication, Bologna, Italy (1980).
[41] Y. Tabourier, All shortest distances in a graph. An improvement to Dantzig's inductive algorithm, Discr. Math. 4(1973)83.
[42] R.E. Tarjan, Complexity of combinatorial algorithms, SIAM Review 20, 3(1978)457.
[43] R.E. Tarjan, Data structures and network algorithms, CBMS-NSF 44 (SIAM, Philadelphia, 1983).
[44] D. Van Vliet, Improved shortest path algorithms for transport networks, Trans. Res. 12, 1(1978)7.
[45] S. Warshall, A theorem on Boolean matrices, J. A.C.M. 9(1962)11.
[46] J.W.J. Williams, Algorithm 232: Heapsort, Commun. A.C.M. 7(1964)347.

Appendix

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE LQUEUE *******
C      ***     (SHORTEST PATH PROBLEM)                          ***
C      ***                                                      ***
C      ***     THE PROGRAM IS BASED ON THE PAPER                ***
C      ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-     ***
C      ***       RITHMS",                                       ***
C      ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME     ***
C      ***                                                      ***
C      ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN      ***
C      ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE       ***
C      ***       PFORT VERIFIER.                                ***
C      ***                                                      ***
C      ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO     ***
C      ***     S. PALLOTTINO AND C. RUGGERI                     ***
C      ***     C.N.R.-I.A.C., ROMA, ITALY.                      ***
C      ************************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH NODE ORIGIN, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                        Q(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF Q(.) AND P(.) ARRAYS IS MADE IN THE MAIN
C PROGRAM, (THE SUBROUTINE RESETS THEM AT THE END OF EACH ITERATION)
C
C EXTERNALS:
C READ
C LQUEUE
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*********************************************************************
      DIMENSION A(3001),ND(31000),LNGT(31000),RAD(100),D(3001),P(3001),
     1Q(3001)
      INTEGER A,D,P,Q,R,RAD,RMAX
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      DO 10 I=1,N
        Q(I) = 0
        P(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
        R = RAD(I)
```

```
          CALL LQUEUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
          WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20  CONTINUE
       STOP
   30  FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
       END
C
C
      SUBROUTINE LQUEUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
C***********************************************************************
C
C     ROUTINE LQUEUE
C
C 1)  FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C     DISTANCES
C 2)  IS BASED ON THE FORD-BELLMAN-MOORE METHOD, WITH THE SET Q
C     IMPLEMENTED AS A QUEUE Q(.)
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), Q(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)     = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = LIST OF THE CANDIDATE NODES; Q(I)= 0 IF I IS NOT IN Q
C                                         = J IF I PRECEDES NODE J IN  Q
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL FOR NODE V
C LAST = POINTER TO THE LAST NODE OF Q(.)
C
C ALL THE PARAMETERS ARE INTEGER
C
C***********************************************************************
      INTEGER A,D,P,Q,R,U,V,DV
      DIMENSION D(NMAX),P(NMAX),Q(NMAX),A(NMAX),ND(MMAX),LNGT(MMAX)
C
C INITIALIZE
C
      DO 10 I=1,N
        D(I) = INF
   10 CONTINUE
```

```
      D(R) = 0
      P(R) = 0
      NN = N + 1
      Q(NN) = NN
      LAST = NN
      U = R
C
C EXPLORE THE FORWARD STAR OF U
C
   20 INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( INIT .GT. IFIN ) GO TO 40
      DO 30 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHEK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 30
        D(V) = DV
        P(V) = U
C
C IF V IS NOT IN Q, IT  IS INSERTED AT THE TAIL OF Q
C
        IF ( Q(V) .GT. 0 ) GO TO 30
        Q(LAST) = V
        Q(V) = NN
        LAST = V
   30 CONTINUE
C
C REMOVE THE NEW CURRENT NODE U
C
   40 U = Q(NN)
      Q(NN) = Q(U)
      Q(U) = 0
      IF ( LAST .EQ. U ) LAST = NN
C
C CHECK WHETHER THE QUEUE IS EMPTY
C
      IF ( U .LE. N ) GO TO 20
      RETURN
      END

      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C**********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C*********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C     ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE LDEQUE *******
C     ***      (SHORTEST PATH PROBLEM)                          ***
C     ***                                                       ***
C     ***     THE PROGRAM IS BASED ON THE PAPER                 ***
C     ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-      ***
C     ***       RITHMS",                                        ***
C     ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME      ***
C     ***                                                       ***
C     ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN       ***
C     ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE        ***
C     ***       PFORT VERIFIER.                                 ***
C     ***                                                       ***
C     ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO      ***
C     ***     S. PALLOTTINO AND C. RUGGERI                      ***
C     ***     C.N.R.-I.A.C., ROMA, ITALY.                       ***
C     ***********************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH NODE ORIGIN, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                     Q(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =  100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF P(.) ARRAY IS MADE IN THE MAIN PROGRAM, (THE
C SUBROUTINE RESETS IT AT THE END OF EACH ITERATION)
C
C EXTERNALS:
C  READ
C  LDEQUE
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*********************************************************************
      DIMENSION A(3001),ND(31000),LNGT(31000),RAD(100),D(3001),P(3001),
     1Q(3001)
      INTEGER A,D,P,Q,R,RAD,RMAX
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      DO 10 I=1,N
        P(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
      R = RAD(I)
      CALL LDEQUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
      WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20 CONTINUE
      STOP
   30 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
      END
```

```
C
C
      SUBROUTINE LDEQUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
C***********************************************************************
C
C    ROUTINE LDEQUE
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON THE D'ESOPO-PAPE METHOD, WITH THE SET Q IMPLEMENTED AS
C    DOUBLE-ENDED-QUEUE Q(.)
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), Q(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)     = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = LIST OF CANDIDATE NODES; Q(I)= -1 IF I IS NOT IN Q AND IT HAS
C                                        ALREADY BEEN SCANNED
C                                     = 0 IF I IS NOT IN Q AND IT HAS
C                                         NOT BEEN SCANNED
C                                     = J IF I PRECEDES NODE J IN THE
C                                              LIST
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL OF NODE V
C LAST = POINTER TO THE LAST NODE OF Q(.)
C
C ALL THE PARAMETERS ARE INTEGER
C
C***********************************************************************
      INTEGER A,D,P,Q,R,U,V,DV
      DIMENSION D(NMAX),P(NMAX),Q(NMAX),A(NMAX),ND(MMAX),LNGT(MMAX)
C
C INITIALIZE
C
      DO 10 I=1,N
        Q(I) = 0
        D(I) = INF
```

```
   10  CONTINUE
       Q(R) = - 1
       D(R)=0
       P(R) = 0
       NN = N + 1
       Q(NN) = NN
       LAST = NN
       PNTR = NN
       U = R
C
C EXPLORE THE FORWARD STAR OF U
C
   20 INIT = A(U)
       IFIN = A(U+1) - 1
       IF ( INIT .GT. IFIN ) GO TO 60
       DO 50 J=INIT,IFIN
         V = ND(J)
         DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
         IF ( D(V) .LE. DV ) GO TO 50
         D(V) = DV
         P(V) = U
         IF ( Q(V) ) 30,40,50
C
C IF V IS NOT IN Q AND IT HAS ALREADY BEEN SCANNED, IT IS INSERTED  AT
C THE POSITION POINTED BY PNTR
C
   30    Q(V) = Q(PNTR)
         Q(PNTR) = V
         IF ( LAST .EQ. PNTR ) LAST=V
         PNTR = V
         GO TO 50
C
C IF V IS NOT IN Q AND IT WAS NEVER SCANNED, IT IS INSERTED AT THE TAIL
C OF Q
C
   40    Q(LAST) = V
         Q(V) = NN
         LAST = V
   50 CONTINUE
C
C REMOVE THE NEW CURRENT NODE U
C
   60 U = Q(NN)
       Q(NN) = Q(U)
       Q(U) = - 1
       IF ( LAST .EQ. U ) LAST = NN
       IF ( PNTR .EQ. U ) PNTR = NN
C
C CHECK WHETHER THE LIST IS EMPTY
C
       IF ( U .LE. N ) GO TO 20
       RETURN
       END

       SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
```

```
C**********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C*********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C     ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE L2QUE  *******
C     ***     (SHORTEST PATH PROBLEM)                          ***
C     ***                                                      ***
C     ***     THE PROGRAM IS BASED ON THE PAPER                ***
C     ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-     ***
C     ***       RITHMS",                                       ***
C     ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME     ***
C     ***                                                      ***
C     ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN      ***
C     ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE       ***
C     ***       PFORT VERIFIER.                                ***
C     ***                                                      ***
C     ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO     ***
C     ***     S. PALLOTTINO AND C. RUGGERI                     ***
C     ***     C.N.R.-I.A.C., ROMA, ITALY.                      ***
C     ************************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH NODE ORIGIN, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                        Q(.)
C       MMAX =31000 FOR ND(.), LNGT(.)
C       RMAX =   100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF P(.) ARRAY IS MADE IN THE MAIN PROGRAM, (THE
C SUBROUTINE RESETS IT AT THE END OF EACH ITERATION).
C
C EXTERNALS:
C READ
C L2QUE
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*****************************************************************
      INTEGER A,D,P,Q,R,RAD,RMAX
      DIMENSION A(3001),ND(31000),LNGT(31000),RAD(100),D(3001),P(3001),
     1Q(3001)
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      DO 10 I=1,N
        P(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
        R = RAD(I)
        CALL L2QUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
```

```
        WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20 CONTINUE
      STOP
   30 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
      END


C
C
      SUBROUTINE L2QUE(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
C*********************************************************************
C
C    ROUTINE L2QUE
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON THE D'ESOPO-PAPE METHOD WITH THE SET Q IMPLEMENTED
C    AS A DOUBLE QUEUE Q(.)
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), Q(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)     = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = LIST OF CANDIDATE NODES; Q(I) = -1 IF I IS NOT IN Q AND IT HAS
C          ALREADY BEEN SCANNED
C        = 0 IF I IS NOT IN Q AND IT HAS
C          NOT BEEN SCANNED
C        = J IF I PRECEDES NODE J IN THE
C          LIST
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL OF NODE V
C LAST = POINTER TO THE LAST NODE OF Q(.)
C PNTR = POINTER TO THE LAST NODE OF THE FIRST QUEUE OF Q(.)
C
C ALL THE PARAMETERS ARE INTEGER
C
C*********************************************************************
      INTEGER A,D,P,Q,R,U,V,DV,PNTR
      DIMENSION D(NMAX),P(NMAX),Q(NMAX),A(NMAX),ND(MMAX),LNGT(MMAX)
C
C INITIALIZE
```

```
      DO 10 I=1,N
        Q(I) = 0
        D(I) = INF
   10 CONTINUE
      Q(R) = - 1
      D(R) = 0
      P(R) = 0
      NN = N + 1
      Q(NN) = NN
      LAST = NN
      U = R
C
C EXPLORE THE FORWARD STAR OF U
C
   20 INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( INIT .GT. IFIN ) GO TO 60
      DO 50 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 50
        D(V) = DV
        P(V) = U
        IF ( Q(V) ) 30,40,50
C
C IF V IT NOT IN Q AND IT HAS ALREADY BEEN SCANNED, IT IS INSERTED AT
C THE HEAD OF Q
C
   30   Q(V) = Q(NN)
        Q(NN) = V
        IF ( LAST .EQ. NN ) LAST = V
        GO TO 50
C
C IF V IS NOT IN Q AND IT WAS NEVER SCANNED, IT IS INSERTED AT THE
C TAIL OF Q
C
   40   Q(LAST) = V
        Q(V) = NN
        LAST = V
   50 CONTINUE
C
C REMOVE THE NEW CURRENT NODE U
C
   60 U = Q(NN)
      Q(NN) = Q(U)
      Q(U) = - 1
      IF ( LAST .EQ. U ) LAST = NN
C
C CHECK WHETHER THE LIST IS EMPTY
C
      IF ( U .LE. N ) GO TO 20
      RETURN
      END
```

```
      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C***********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C**********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40)  (A(I),I=1,N1)
      READ (5,50)  (ND(I),LNGT(I),I=1,M)
      READ (5,40)  (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE LTHRS   *******
C      ***     (SHORTEST PATH PROBLEM)                            ***
C      ***                                                        ***
C      ***     THE PROGRAM IS BASED ON THE PAPER                  ***
C      ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-       ***
C      ***       RITHMS",                                         ***
C      ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME       ***
C      ***                                                        ***
C      ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN        ***
C      ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE         ***
C      ***       PFORT VERIFIER.                                  ***
C      ***                                                        ***
C      ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO       ***
C      ***     S. PALLOTTINO AND C. RUGGERI                       ***
C      ***     C.N.R.-I.A.C., ROMA, ITALY.                        ***
C      ************************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH NODE ORIGIN, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.), P(.),
C                                                    Q1(.),Q2(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =  100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF Q1(.),Q2(.) AND P(.) ARRAYS IS MADE IN THE MAIN
C PROGRAM, (THE SUBROUTINE RESETS THEM AT THE END OF EACH ITERATION)
C
C EXTERNALS:
C  READ
C  THINCR
C  LTHRS
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*********************************************************************
      INTEGER A,D,P,Q1,Q2,T,R,RAD,RMAX
      DIMENSION A(3001),ND(31000),LNGT(31000),RAD(100),D(3001),P(3001),
     1Q1(3001),Q2(3001)
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      CALL THINCR(N,M,LMAX,T)
      DO 10 I=1,N
        P(I) = 0
        Q1(I) = 0
```

```
        Q2(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
        R = RAD(I)
        CALL LTHRS(A,ND,LNGT,D,P,Q1,Q2,NMAX,MMAX,N,INF,T,R)
        WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20 CONTINUE
      STOP
   30 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9/))
      END
```

```
      SUBROUTINE LTHRS (A,ND,LNGT,D,P,Q1,Q2,NMAX,MMAX,N,INF,T,R)
C***********************************************************************
C
C    ROUTINE LTHRS
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON THRESHOLD METHOD PROPOSED BY F.GLOVER, R.GLOVER AND D.
C    KLINGMAN, WITH INSERTION POLICY DEPENDING ON A THRESHOLD VALUE
C    AND WITH Q IMPLEMENTED AS A PAIR OF LISTS: Q1(.) AS A QUEUE AND
C    Q2(.) AS A LINKED-LIST.
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)    = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)   = ENDING NODE OF ARC J
C LNGT(J) = LENGTH OF ARC J, J=1,2,...,M                      ,2,...,M
C NMAX    = DIMENSION OF ARRAYS A(.), D(.), P(.), Q1(.), Q2(.)
C MMAX    = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N       = NUMBER OF NODES
C INF     = VERY LARGE INTEGER VALUE (INFINITY)
C R       = ROOT
C T       = INCREMENT FOR THE THRESHOLD
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)    = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)    = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C Q1(I) = LIST OF CANDIDATE NODES       Q1(I) = 0 IF I IS NOT IN Q1(.)
C         HAVING THEIR LABEL LESS THAN        = J IF I PRECEDES NODE J
C         OR EQUAL TO THE CURRENT               IN THE LIST
C         THRESHOLD
C
C Q2(I) = LIST OF THE OTHER CANDIDATE   Q2(I) = 0 IF I IS NOT IN Q2(.)
C         NODES AND OLD COPIES OF NODES       = J IF I PRECEDES NODE J
C         INSERTED IN Q1(.)                     IN THE LIST
C
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL FOR NODE V
C LAST = POINTER TO THE LAST NODE OF Q1(.)
```

```
C THRS = CURRENT THRESHOLD VALUE
C T1   = TENTATIVE NEW THRESHOLD VALUE
C MIN  = MINIMUM LABEL VALUE OF NODES IN Q2(.)
C
C  ALL THE PARAMETERS ARE INTEGER
C
C**********************************************************************
      DIMENSION A(NMAX),D(NMAX),P(NMAX),Q1(NMAX),Q2(NMAX),ND(MMAX),LNGT(
     *MMAX)
      INTEGER A,D,P,Q1,Q2,T,THRS,T1,U,V,DV,R
C
C INITIALIZE
C
      DO 10 I=1,N
        D(I) = INF
   10 CONTINUE
      D(R) = 0
      P(R) = 0
      NN = N + 1
      Q1(NN) = NN
      Q2(NN) = NN
      THRS = -1
      LAST = NN
      U = R
C
C EXPLORE THE FORWARD STAR OF U
C
   20 INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( INIT .GT. IFIN ) GO TO 60
      DO 50 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 50
        IF ( DV .GT. THRS ) GO TO 30
        IF ( Q1(V) .GT. 0 ) GO TO 40
C
C INSERT V AT THE TAIL OF Q1(.)
C
        Q1(LAST) = V
        Q1(V) = NN
        LAST = V
        GO TO 40
   30   IF ( Q2(V) .GT. 0 ) GO TO 40
C
C IF V IS NOT IN Q2(.), IT IS INSERTED AT THE HEAD OF Q2(.)
C
        Q2(V) = Q2(NN)
        Q2(NN) = V
   40   D(V) = DV
        P(V) = U
   50 CONTINUE
C
C CHECK WHETHER Q1(.) IS EMPTY
C
   60 IF ( LAST .EQ. NN ) GO TO 80
C
```

```
C REMOVE THE NEW CURRENT NODE U FROM THE HEAD OF Q1(.)
C
  70   U = Q1(NN)
       Q1(NN) = Q1(U)
       Q1(U) = 0
       IF ( LAST .EQ. U ) LAST = NN
       GO TO 20
C
C CHECK WHETHER ALSO Q2(.) IS EMPTY
C
  80   IF ( Q2(NN) .EQ. NN ) RETURN
C
C COMPUTE THE NEW TENTATIVE THRESHOLD VALUE T1
C
       MIN = INF
       T1 = THRS + 1 + T
       I = NN
       J = Q2(I)
C
C SCAN Q2(.) IN ORDER TO COMPUTE MIN AND TO REMOVE COPIES OF NODES
C ALREADY REMOVED
C
  90   IF ( D(J) .LE. T1 ) GO TO 100
C
C UPDATE MIN
C
       MIN = MINO(MIN, D(J))
       I = J
       GO TO 110
C
C  REMOVE J FROM Q2(.)
C
 100   Q2(I) = Q2(J)
       Q2(J) = 0
C
C CHECK WHETHER J MUST BE INSERTED IN Q1(.)
C
       IF ( D(J) .LE. THRS ) GO TO 110
       Q1(LAST) = J
       Q1(J) = NN
       LAST = J
 110   J = Q2(I)
       IF ( J .NE. NN ) GO TO 90
C
C UPDATE THE THRESHOLD VALUE
C
       THRS = T1
C
C CHECK WHETHER Q1(.) IS STILL EMPTY
C
       IF ( LAST .NE. NN ) GO TO 70
C
C IF Q2(.) WAS EMPTIED THEN RETURN
C
       IF ( Q2(NN) .EQ. NN ) RETURN
C
C INCREASE THE THRESHOLD VALUE THRS AND SCAN AGAIN Q2(.)
C
       THRS = MIN + T
       I = NN
```

```
      J = Q2(I)
 120  IF ( D(J) .GT. THRS ) GO TO 130
C
C  MOVE J FROM Q2(.) TO Q1(.)
C
      Q2(I) = Q2(J)
      Q2(J) = 0
      Q1(LAST) = J
      Q1(J) = NN
      LAST = J
      GO TO 140
C
C  CONTINUE THE SCANNING OF Q2(.)
C
 130  I = J
 140  J = Q2(I)
      IF ( J .NE. NN ) GO TO 120
      GO TO 70
      END


      SUBROUTINE THINCR(N,M,LMAX,T)
C*********************************************************************
C  COMPUTE THE INCREMENT OF THE THRESHOLD AS PROPOSED IN "F.GLOVER; R.
C  GLOVER; D.KLINGMAN, CENTER FOR CYBERNETIC STUDIES, UNIVERSITY OF
C  TEXAS, AUSTIN TX, USA. RESEARCH REPORT CCS 419, JUNE 1981.
C*********************************************************************
      INTEGER T,S
      READ (5,100) X2
      S = MIN0(35,M/N)
      ALMAX = LMAX
      T = X2 * ALMAX
      AT = T
      AS = S
      IF ( S .GT. 7 ) T = AT * 7. / AS
      IF ( T .LE. 0 ) T = 1
      RETURN
 100  FORMAT(F4.2)
      END


C
      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C*********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C*********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N,M,NR,LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE SDKSTR *******
C      ***      (SHORTEST PATH PROBLEM)                           ***
C      ***                                                        ***
C      ***      THE PROGRAM IS BASED ON THE PAPER                 ***
C      ***      G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-      ***
C      ***        RITHMS",                                        ***
C      ***        ANNALS OF OPERATIONS RESEARCH, THIS VOLUME      ***
C      ***                                                        ***
C      ***      ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN       ***
C      ***        STANDARD FORTRAN AND ARE ACCEPTED BY THE        ***
C      ***        PFORT VERIFIER.                                 ***
C      ***                                                        ***
C      ***      QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO      ***
C      ***      S. PALLOTTINO AND C. RUGGERI                      ***
C      ***      C.N.R.-I.A.C., ROMA, ITALY.                       ***
C      ************************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH ORIGIN NODE, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                       Q(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =  100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF  Q(.), AND P(.) ARRAYS IS MADE IN THE MAIN
C PROGRAM, (THE SUBROUTINE RESETS THEM AT THE END OF EACH ITERATION)
C
C EXTERNALS:
C  READ
C  SDKSTR
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C***************************************************************
      INTEGER A,D,P,Q,R,HP,RAD,Y,X,T2,RMAX
      DIMENSION A(3001),ND(31000),RAD(100),D(3001),P(3001),LNGT(31000),
     1Q(3001)
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      DO 10 I=1,N
        Q(I) = 0
        P(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
```

```
        R = RAD(I)
        CALL SDKSTR(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
        WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20 CONTINUE
      STOP
   30 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9/))
      END



C
C
      SUBROUTINE SDKSTR(A,ND,LNGT,D,P,Q,NMAX,MMAX,N,INF,R)
C**********************************************************************
C
C    ROUTINE  SDKSTR
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON DIJKSTRA'S METHOD, WITH PRIORITY QUEUE Q IMPLEMENTED
C    AS AN UNORDERED LIST
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    =ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), Q(.), HP(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)     = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = LIST OF THE CANDIDATE NODES: = 0 IF NODE I IS NOT IN Q
C                                       J IF NODE I PRECEDES NODE J IN Q
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL OF NODE V
C
C ALL THE PARAMETERS ARE INTEGER
C
C**********************************************************************
       INTEGER A,D,P,Q,R,U,V,DV,VAL
       DIMENSION A(NMAX),D(NMAX),P(NMAX),Q(NMAX),ND(MMAX),LNGT(MMAX)
C
C INITIALIZE
C
      DO 10 I=1,N
        D(I) = INF
   10 CONTINUE
```

```
      D(R)  = 0
      P(R)  = 0
      NN = N + 1
      Q(NN)  = NN
      U = R
C
C EXPLORE THE FORWARD STAR OF  U
C
  20  INIT = A(U)
      IFIN = A(U+1)  - 1
      IF ( IFIN .LT. INIT ) GO TO 40
      DO 30 J=INIT,IFIN
        V = ND(J)
        DV = D(U)  + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 30
        D(V)  = DV
        P(V)  = U
C
C IF V IS NOT IN Q, INSERT V AT THE HEAD OF Q
C
        IF ( Q(V) .NE. 0 ) GO TO 30
          Q(V)  = Q(NN)
          Q(NN)  = V
   30 CONTINUE
C
C CHECK WHETHER THE LIST IS EMPTY
C
   40 IF ( Q(NN) .EQ. NN ) GO TO 70
C
C SEARCH THE MINIMUM LABEL NODE IN Q
C
      K = 0
      I = NN
      VAL = INF
   50 JJQ = Q(I)
      IF ( D(JJQ) .GE. VAL ) GO TO 60
      VAL = D(JJQ)
      K = I
   60 I = Q(I)
      IF ( Q(I) .NE. NN ) GO TO 50
C
C REMOVE THE NEW CURRENT NODE U FROM Q
C
      U = Q(K)
      Q(K)  = Q(U)
      Q(U)  = 0
      GO TO 20
   70 CONTINUE
      RETURN
      END


      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C**********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
```

```
C***********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE SORD2  *******
C      ***      (SHORTEST PATH PROBLEM)                           ***
C      ***                                                        ***
C      ***      THE PROGRAM IS BASED ON THE PAPER                 ***
C      ***      G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-      ***
C      ***        RITHMS",                                        ***
C      ***        ANNALS OF OPERATIONS RESEARCH, THIS VOLUME      ***
C      ***                                                        ***
C      ***      ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN       ***
C      ***        STANDARD FORTRAN AND ARE ACCEPTED BY THE        ***
C      ***        PFORT VERIFIER.                                 ***
C      ***                                                        ***
C      ***      QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO      ***
C      ***      S. PALLOTTINO AND C. RUGGERI                      ***
C      ***      C.N.R.-I.A.C., ROMA, ITALY.                       ***
C      ************************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH ORIGIN NODE, I=1,2,...,NR
C IVER     = BUBBLE SORT PARAMETER; SET IVER = 1 FOR INCREASING ORDER AND
C            IVER = -1 FOR DECREASING ORDER
C
C ALL THE PARAMETERS ARE INTEGER
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                   UP(.), DOWN(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =  100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C THE INITIALIZATION OF  UP(.), DOWN(.), AND P(.) ARRAYS IS MADE IN THE
C MAIN PROGRAM, (THE SUBROUTINE RESET THEM AT THE END OF EACH
C  ITERATION)
C
C EXTERNALS:
C   READ
C   BUBBLE
C   SORD2
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*********************************************************************
      DIMENSION A(3001),ND(31000),RAD(100),D(3001),P(3001),UP(3001),
     1DOWN(3001),LNGT(31000)
      INTEGER A,D,P,R,UP,RAD,DOWN,RMAX
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      IVER = - 1
      DO 10 I=1,N
C
```

```
C  SORT THE FORWARD STAR OF NODE I
C
         INIT = A(I)
         IFIN = A(I+1) - 1
         IF ( INIT .GE. IFIN ) GO TO 10
         CALL BUBBLE(INIT,IFIN,LNGT,ND,IVER)
   10 CONTINUE
      DO 20 J=1,N
        UP(J) = 0
        DOWN(J) = 0
        P(J) = 0
  20  CONTINUE
      DO 30 I=1,NR
        R = RAD(I)
        CALL SORD2(A,ND,LNGT,D,P,UP,DOWN,NMAX,MMAX,N,INF,R)
        WRITE(6,40)R,(J,P(J),D(J),J=1,N)
  30  CONTINUE
      STOP
   40 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
      END


      SUBROUTINE SORD2(A,ND,LNGT,D,P,UP,DOWN,NMAX,MMAX,N,INF,R)
C***********************************************************************
C
C    ROUTINE  SORD2
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON DJKSTRA'S METHOD, WITH THE PRIORITY QUEUE Q IMPLEMENTED
C    AS AN ORDERED TWO-WAY LINKED LIST
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)    = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)   = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J) = LENGTH OF ARC J, J=1,2,...,M
C NMAX    = DIMENSION OF ARRAYS A(.), D(.), P(.), UP(.), DOWN(.)
C MMAX    = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N       = NUMBER OF NODES
C INF     = VERY LARGE INTEGER VALUE (INFINITY)
C R       = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)    = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)    = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C UP(I)= 0 IF I DOES NOT BELONG TO THE LIST
C      = J IF J PRECEDES I IN THE LIST
C DOWN(I)= 0 IF I DOES NOT BELONG TO THE LIST
C          J IF J FOLLOWS I IN THE LIST
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
```

```
C PNTR = POINTER TO THE PORTION OF THE LIST TO BE SCANNED. IT MOVES
C         BOTTOM-UP
C DV    = TENTATIVE LABEL OF NODE V
C
C ALL THE PARAMETERS ARE INTEGER
C
C*************************************************************************
      INTEGER A,D,P,R,U,V,UP,DOWN,DV,PNTR
      DIMENSION D(NMAX),P(NMAX),UP(NMAX),DOWN(NMAX),A(NMAX),ND(MMAX),
     1LNGT(MMAX)
C
C  INITIALIZE
C
       DO 10 I=1,N
         D(I) = INF
   10  CONTINUE
       D(R) = 0
       P(R) = 0
       NN = N + 1
       D(NN) = - 1
       UP(NN) = NN
       DOWN(NN) = NN
       U = R
C
C EXPLORE OF THE FORWARD STAR OF U
C
   20 INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( INIT .GT. IFIN )  GO TO 80
C
C  RESET PNTR TO THE BOTTOM OF THE LIST
C
      PNTR = UP(NN)
      DO 70 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 70
C
C  RESET THE POINTER
C
        IF ( D(V) .LT. D(PNTR) .AND. UP(V) .GT. 0 ) PNTR = V
C
C  FIND THE INSERTION POINT FOR V
C
   30   IF ( D(PNTR) .LE. DV ) GO TO 40
        PNTR = UP(PNTR)
        GO TO 30
  40    IF ( DOWN(PNTR) .EQ. V ) GO TO 60
C
C  REMOVE V FROM UP(.) AND DOWN(.)‘IF NECESSARY
C
        IF ( UP(V) .EQ. 0 ) GO TO 50
        IUV = UP(V)
        IDV = DOWN(V)
        DOWN(IUV) = IDV
        UP(IDV) = IUV
C
```

```
C   INSERT V INTO UP(.) AND DOWN(.)
C
   50    IDV = DOWN(PNTR)
         DOWN(V) = IDV
         DOWN(PNTR) = V
         UP(IDV) = V
         UP(V) = PNTR
   60    D(V) = DV
         P(V) = U
   70 CONTINUE
C
C   REMOVE THE NEW CURRENT NODE U
C
   80 U = DOWN(NN)
      IDU = DOWN(U)
      DOWN(NN) = IDU
      UP(IDU) = NN
      UP(U) = 0
C
C CHECK WHETHER THE LIST IS EMPTY
C
      IF ( U .LE. N ) GO TO 20
      RETURN
      END


C
C
      SUBROUTINE BUBBLE(INIT,IFIN,IV1,IV2,IVER)
C*************************************************************************
C SORTS A PORTION OF THE ARRAYS IV1(.) AND IV2(.) ACCORDING TO THE
C DECREASING (IVER=-1) OR INCREASING (IVER=1) ORDER OF THE ELEMENTS OF
C IV1(.). THE ELEMENTS TO BE SORTED HAVE INDEX J BETWEEN INIT AND IFIN.
C A BUBBLE SORTING TECHNIQUE IS USED.
C*************************************************************************
      INTEGER PNTR
      DIMENSION IV1(IFIN),IV2(IFIN)
      LAST = IFIN
   10 PNTR = LAST - 1
      LAST = 0
      DO 20 J=INIT,PNTR
        IF ( (IV1(J) - IV1(J+1))*IVER .LE. 0 ) GO TO 20
        IBUF = IV1(J)
        IV1(J) = IV1(J+1)
        IV1(J+1) = IBUF
        IBUF = IV2(J)
        IV2(J) = IV2(J+1)
        IV2(J+1) = IBUF
        LAST = J
   20 CONTINUE
      IF ( LAST .GT. 1 ) GO TO 10
      RETURN
      END


      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C**********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
```

```
C LIST.
C********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE SDIAL  *******
C      ***     (SHORTEST PATH PROBLEM)                          ***
C      ***                                                      ***
C      ***     THE PROGRAM IS BASED ON THE PAPER                ***
C      ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-     ***
C      ***       RITHMS",                                       ***
C      ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME     ***
C      ***                                                      ***
C      ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN      ***
C      ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE       ***
C      ***       PFORT VERIFIER.                                ***
C      ***                                                      ***
C      ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO     ***
C      ***     S. PALLOTTINO AND C. RUGGERI                     ***
C      ***     C.N.R.-I.A.C., ROMA, ITALY.                      ***
C      **********************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH NODE ORIGIN, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                  UP(.), DOWN(.)
C                                    MMAX =31000 FOR ND(.), LNGT(.)
C                                    RMAX =   100 FOR RAD(.)
C                                    QMAX =10001 FOR Q(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET TO A
C  LARGE INTEGER VALUE
C
C
C THE INITIALIZATION OF  Q(.), UP(.), DOWN(.), AND P(.) ARRAYS IS MADE
C IN THE MAIN PROGRAM, (THE SUBROUTINE RESETS THEM AT THE END OF EACH
C  ITERATION).
C
C EXTERNALS:
C  SDIAL
C  READ
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*****************************************************************
      DIMENSION A(3001),ND(31000),RAD(100),D(3001),P(3001),UP(3001),
     1DOWN(3001),Q(10001),LNGT(31000)
      INTEGER A,D,P,Q,R,UP,DOWN,RAD,QMAX,RMAX
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/,QMAX/10001/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      NN = N + 1
      LMAX = LMAX + 1
      DO 10 I=1,LMAX
        Q(I) = NN
```

```
   10   CONTINUE
        DO 20 J=1,N
          UP(J) = 0
          DOWN(J) = 0
          P(J) = 0
   20    CONTINUE
         DO 30 I=1,NR
           R = RAD(I)
           CALL SDIAL(A,ND,LNGT,D,P,Q,UP,DOWN,NMAX,MMAX,QMAX,N,INF,R,LMAX)
           WRITE(6,40)R,(J,P(J),D(J),J=1,N)
   30   CONTINUE
        STOP
    40 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
        END



C
C
        SUBROUTINE SDIAL(A,ND,LNGT,D,P,Q,UP,DOWN,NMAX,MMAX,QMAX,N,INF,R,
      *LMAX)
C*************************************************************************
C
C     ROUTINE SDIAL
C
C 1)  FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C     DISTANCES
C 2)  IS BASED ON DIAL'S METHOD, WITH THE PRIORITY QUEUE Q IMPLEMENTED AS
C     AN ADDRESS ARRAY AND A TWO-WAY LINKED LIST
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), UP(.), DOWN(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C QMAX     = DIMENSION OF ARRAY Q(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C LMAX     = LENGTH OF THE ACTIVE PART OF ARRAY Q(.)
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C P(I)     = PREDECESSOR NODE OF I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = J IF J IS THE FIRST ELEMENT OF THE I-TH LIST
C      = NN IF THE I-TH LIST IS EMPTY
C UP(J)= 0 IF J DOES NOT BELONG TO ANY LIST
C        K IF K PRECEDES J IN THE SAME LIST
C       -I IF J IS THE FIRST ELEMENT OF THE I-TH LIST (Q(I)=J)
C DOWN(J)= 0 IF J DOES NOT BELONG TO ANY LIST
C          K IF K FOLLOWS J IN THE SAME LIST
C          NN IF J IS THE LAST ELEMENT OF THE LIST
C NN    = N+1
C U     = CURRENT NODE
```

```
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL OF NODE V
C PNTR = POINTER TO THE LAST SCANNED POSITION OF Q(.)
C PREC= POINTER FOR UP(.) AND DOWN(.) ARRAYS
C       IF PREC < 0 THEN -PREC IS A POINTER TO Q(.) ARRAY
C SEG = POINTER FOR UP(.) AND DOWN (.) ARRAYS
C ADDR= POINTER TO THE LIST IN WHICH V MUST BE INSERTED
C
C ALL THE PARAMETERS ARE INTEGER
C
C***********************************************************************
      INTEGER A,D,P,Q,R,U,V,DV,SEG,PREC,ADDR,PNTR,UP,DOWN,QMAX
      DIMENSION A(NMAX),D(NMAX),P(NMAX),UP(NMAX),DOWN(NMAX),ND(MMAX),
     1LNGT(MMAX),Q(QMAX)
C
C INITIALIZE
C
      DO 10 I=1,N
        D(I) = INF
   10 CONTINUE
      D(R) = 0
      P(R) = 0
      NN = N + 1
      UP(NN) = NN
      DOWN(NN) = NN
      PNTR = 1
      U = R
C
C EXPLORE THE FORWARD STAR OF U
C
   20 INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( IFIN .LT. INIT ) GO TO 60
      DO 50 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 50
        D(V) = DV
        P(V) = U
        IF ( UP(V) .EQ. 0 ) GO TO 40
C
C REMOVE V FROM UP(.) AND DOWN(.) IF NECESSARY
C
        PREC = UP(V)
        SEG = DOWN(V)
        IF ( PREC .LT. 0 ) GO TO 30
        DOWN(PREC) = SEG
        UP(SEG) = PREC
        GO TO 40
   30   IP = - PREC
        Q(IP) = SEG
        UP(SEG) = PREC
C
C COMPUTE THE POINTER TO TE LIST IN WHICH V MUST BE INSERTED
C
```

```
   40    ADDR = DV + 1 - DV/LMAX*LMAX
         SEG = Q(ADDR)
         DOWN(V) = SEG
         Q(ADDR) = V
         UP(V) = - ADDR
         UP(SEG) = V
   50 CONTINUE
C
C REMOVE THE NEW CURRENT NODE U
C
   60 ADDR = PNTR
   70 IF ( Q(PNTR) .EQ. NN ) GO TO 80
      U = Q(PNTR)
      IDU = DOWN(U)
      Q(PNTR) = IDU
      UP(IDU) = - PNTR
      UP(U) = 0
      GO TO 20
   80 PNTR = PNTR + 1
      IF ( PNTR .GT. LMAX ) PNTR = 1
C
C CHECK WHETER Q(.) IS EMPTY
C
      IF ( PNTR .NE. ADDR ) GO TO 70
      RETURN
      END


      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C***********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C**********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```

```
C      ******* SAMPLE CALLING PROGRAM FOR SUBROUTINE SHEAP  *******
C      ***     (SHORTEST PATH PROBLEM)                          ***
C      ***                                                      ***
C      ***     THE PROGRAM IS BASED ON THE PAPER                ***
C      ***     G. GALLO, S. PALLOTTINO "SHORTEST PATH ALGO-     ***
C      ***       RITHMS",                                       ***
C      ***       ANNALS OF OPERATIONS RESEARCH, THIS VOLUME     ***
C      ***                                                      ***
C      ***     ALL THE SUBROUTINES ARE WRITTEN IN AMERICAN      ***
C      ***       STANDARD FORTRAN AND ARE ACCEPTED BY THE       ***
C      ***       PFORT VERIFIER.                                ***
C      ***                                                      ***
C      ***     QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO     ***
C      ***     S. PALLOTTINO AND C. RUGGERI                     ***
C      ***     C.N.R.-I.A.C., ROMA, ITALY.                      ***
C      **********************************************************
C
C MEANING OF THE MAIN PARAMETERS NOT EXPLAINED IN THE SUBROUTINE:
C
C RAD(I)   = I-TH ORIGIN NODE, I=1,2,...,NR
C
C ALL THE PARAMETERS ARE INTEGER
C
C
C AT PRESENT THE SIZE DIMENSIONS ARE NMAX = 3001 FOR A(.), D(.). P(.),
C                                                    HP(.), Q(.)
C                                   MMAX =31000 FOR ND(.), LNGT(.)
C                                   RMAX =  100 FOR RAD(.)
C
C THE ONLY MACHINE DEPENDENT CONSTANT USED IS INF: MUST BE SET T
C
C
C THE INITIALIZATION OF  Q(.), AND P(.) ARRAYS IS MADE IN THE MAIN
C PROGRAM, (THE SUBROUTINE RESETS THEM AT THE END OF EACH ITERATION)
C
C EXTERNALS:
C  READ
C  SHEAP
C
C
C THIS WORK WAS SUPPORTED BY C.N.R., ITALY.
C
C*********************************************************************
      INTEGER A,D,P,Q,R,HP,RAD,Y,X,T2,RMAX
      DIMENSION A(3001),ND(31000),RAD(100),D(3001),P(3001),HP(3001),
     1LNGT(31000),Q(3001)
      DATA NMAX/3001/,MMAX/31000/,RMAX/100/,INF/999999999/
      CALL READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
      DO 10 I=1,N
        Q(I) = 0
        P(I) = 0
   10 CONTINUE
      DO 20 I=1,NR
        R = RAD(I)
```

```
      CALL SHEAP(A,ND,LNGT,D,P,Q,HP,NMAX,MMAX,N,INF,R)
      WRITE(6,30)R,(J,P(J),D(J),J=1,N)
   20 CONTINUE
      STOP
   30 FORMAT(/7H ROOT =,I4//5H NODE,4X,1HP,7X,1HD/(2I5,I9))
      END



C
C
      SUBROUTINE SHEAP(A,ND,LNGT,D,P,Q,HP,NMAX,MMAX,N,INF,R)
C***********************************************************************
C
C    ROUTINE  SHEAP
C
C 1) FINDS A SHORTEST PATH TREE ROOTED AT NODE R AND THE SHORTEST
C    DISTANCES
C 2) IS BASED ON DIJKSTRA'S METHOD, WITH PRIORITY QUEUE Q IMPLEMENTED
C    AS A BINARY HEAP
C
C MEANING OF THE INPUT PARAMETERS:
C
C A(I)     = POINTER TO ARC-LIST OF NODE I, I=1,2,...,N+1
C ND(J)    = ENDING NODE OF ARC J, J=1,2,...,M
C LNGT(J)  = LENGTH OF ARC J, J=1,2,...,M
C NMAX     = DIMENSION OF ARRAYS A(.), D(.), P(.), Q(.), HP(.)
C MMAX     = DIMENSION OF ARRAYS ND(.), LNGT(.)
C N        = NUMBER OF NODES
C INF      = VERY LARGE INTEGER VALUE (INFINITY)
C R        = ROOT
C
C MEANING OF THE OUTPUT PARAMETERS:
C
C D(I)     = SHORTEST DISTANCE FROM R TO I, I=1,2,...,N
C                I IN THE SHORTEST PATH TREE, I=1,2,...,N
C
C MEANING OF THE MAIN INTERNAL PARAMETERS:
C
C Q(I) = DICTIONARY OF THE HEAP: Q(I) GIVES THE POSITION OF NODE
C        I IN THE HEAP HP(.), I=1,2,...,N
C HP(I)= I-TH NODE IN THE HEAP, I=1,2,...,NHP
C NHP  = NUMBER OF NODES IN THE HEAP (NHP<=N)
C NN   = N+1
C U    = CURRENT NODE
C V    = ENDING NODE OF THE CURRENT ARC
C INIT = START-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C IFIN = END-POINTER TO THE ARC-LIST OF THE CURRENT NODE
C DV   = TENTATIVE LABEL OF NODE V
C
C ALL THE PARAMETERS ARE INTEGER
C
C***********************************************************************
      INTEGER A,D,P,Q,R,U,V,DV,HP,DP1,HP1,HP2,HP3
      DIMENSION A(NMAX),D(NMAX),P(NMAX),Q(NMAX),HP(NMAX),ND(MMAX),
     1LNGT(MMAX)
C
C INITIALIZE
C
      DO 10 I=1,N
```

```
        D(I) = INF
  10  CONTINUE
      NHP = 0
      D(R) = 0
      P(R) = 0
      NN = N + 1
      U = R
C
C EXPLORE THE FORWARD STAR OF  U
C
  20  INIT = A(U)
      IFIN = A(U+1) - 1
      IF ( IFIN .LT. INIT ) GO TO 70
      DO 60 J=INIT,IFIN
        V = ND(J)
        DV = D(U) + LNGT(J)
C
C CHECK WHETHER THE LABEL OF V CAN BE IMPROVED
C
        IF ( D(V) .LE. DV ) GO TO 60
        D(V) = DV
        P(V) = U
        IF ( Q(V) .NE. 0 ) GO TO 30
C
C INSERT NODE V INTO THE HEAP
C
        NHP = NHP + 1
        Q(V) = NHP
C
C UPDATE THE HEAP
C
   30   K = Q(V)
   40   K2 = K/2
        IF ( K2 .LE. 0 ) GO TO 50
        HP2 = HP(K2)
        IF ( DV .GE. D(HP2) ) GO TO 50
        HP(K) = HP2
        Q(HP2) = K
        K = K2
        GO TO 40
   50   HP(K) = V
        Q(V) = K
   60 CONTINUE
C
C REMOVE THE NEW CURRENT NODE U FROM THE HEAP
C
  70  U = HP(1)
      Q(U) = 0
      NHP = NHP - 1
C
C CHECK WHETHER THE HEAP IS EMPTY
C
      IF ( NHP ) 130,20,80
C
C UPDATE  THE HEAP
C
  80  HP1 = HP(NHP+1)
      DP1 = D(HP1)
      K = 1
   90 K2 = 2*K
```

```
      HP2 = HP(K2)
      IF ( K2-NHP ) 100,110,120
  100 HP3 = HP(K2+1)
      IF ( D(HP2) .LT. D(HP3) ) GO TO 110
      HP2 = HP3
      K2 = K2 + 1
  110 IF ( DP1 .LE. D(HP2) ) GO TO 120
      HP(K) = HP2
      Q(HP2) = K
      K = K2
      GO TO 90
  120 HP(K)=HP1
      Q(HP1) = K
      GO TO 20
  130 CONTINUE
      RETURN
      END


      SUBROUTINE READ(N,M,NR,LMAX,A,ND,LNGT,RAD,NMAX,MMAX,RMAX)
C**********************************************************************
C READS THE GRAPH DATA (STORED AS AN ADJACENCE LIST) AND THE ORIGINS
C LIST.
C*********************************************************************
      INTEGER A,RAD,RMAX
      DIMENSION A(NMAX),ND(MMAX),LNGT(MMAX),RAD(RMAX)
      READ (5,30) N, M, NR, LMAX
      N1 = N + 1
      READ (5,40) (A(I),I=1,N1)
      READ (5,50) (ND(I),LNGT(I),I=1,M)
      READ (5,40) (RAD(I),I=1,NR)
      RETURN
   30 FORMAT(4I6)
   40 FORMAT(10I6)
   50 FORMAT(12I6)
      END
```