

Chapter 8

□ SHORTEST PATH ALGORITHMS

8.1 INTRODUCTION

8.1.1 Problem

Given a directed graph $G = (N, A)$, whose arcs $(u, v) \in A$ have lengths d_{uv} , and two nodes r and t find the path from r to t whose length is a minimum.

This problem is one of the most fundamental and important in combinatorial optimization. Many practical optimization problems can be formulated as shortest path problems. Also, many other algorithms call a shortest path algorithm as a procedure.

8.1.2 History

This problem has been studied extensively since the late 1950s. It appears in many forms and variations and hundreds of algorithms have been devised to solve it. The problem is easy in the sense that a shortest path in a graph of n nodes and m arcs can be found in $O(n^2)$ time.

Despite the simplicity of the problem and the amount of research effort spent on it, many people (some of the best) are still working it. The table below shows the history of *theoretical* improvements that have been made to the original $O(n^2)$ algorithm of Dijkstra:

TABLE I. SHORTEST-PATH ALGORITHMS.

Algorithm	Complexity
Dijkstra (1959)	$O(n^2)$
Williams (1964)	$O(m \log_2 n)$
Johnson (1977)	$O(m \log_d n)$, $d = \max(2, \frac{m}{n})$
Boas, <i>et al.</i> (1977)	$O(c + m \log_2 c \log_2 \log_2 c)$
Johnson (1982)	$O(m \log_2 \log_2 c)$
Fredman & Tarjan (1984)	$O(m + n \log_2 n)$
Gabow (1985)	$O(m \log_d c)$, $d = \max(2, \frac{m}{n})$
Tarjan, <i>et al.</i> (1989)	$O(m + n \sqrt{\log_2 c})$

8.2 THEORY

Two extensions of the standard problem (single pair) are common:

- *Single Source* : Given a source r find the shortest paths from r to all other nodes u in G
- *All Pairs* : For every pair of nodes u and v in G find the shortest path from u to v .

All known algorithms for solving the single pair problem must solve all or part of the single source problem. The all-pairs problem can be viewed as n single source problems and solved accordingly. For these reasons the fundamental problem is the single source problem, i.e. find the shortest paths from r to all other nodes.

8.2.1 Spanning Tree Solutions

The optimum solution to the single-source shortest path problem is a set of $n - 1$ shortest paths and their lengths. Such a solution can be succinctly described by a spanning tree $T(r)$ whose root is r . The unique path from r to any u in $T(r)$ is the shortest path from r to u in G . The length of any path from r to u in $T(r)$ is the shortest distance from r to u in G and is denoted by D_u . We call such a tree a *Shortest Path Spanning Tree*.

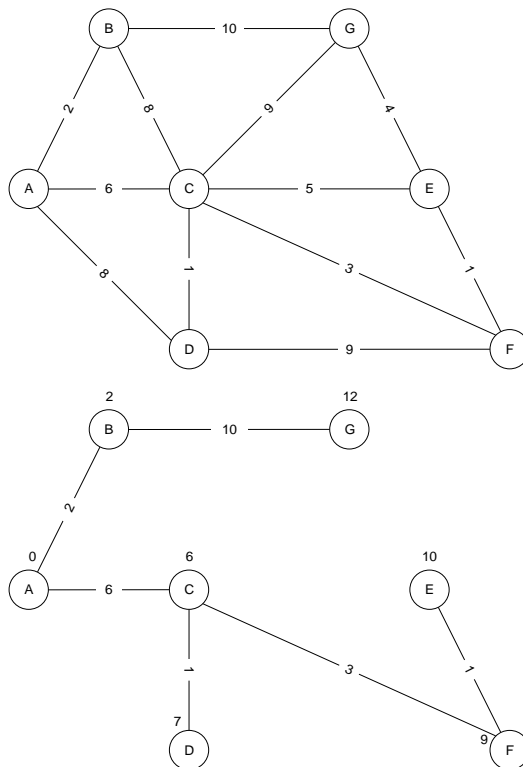


Figure 8.1. A Graph and its Shortest Path Spanning Tree.

Theorem 8.1 (OPTIMUM SPANNING TREE). $T(r)$ is a shortest-path spanning tree of G if and only if

$$D_v \leq D_u + d_{uv}, \quad \forall (u, v) \in A,$$

where D_u and D_v are the lengths of the tree paths from r to u and v , respectively. \square

Theorem 1 allows us to test in $O(m)$ time if a given spanning tree is a shortest path spanning tree.

8.2.2 Bellman's Equations

An alternative to Theorem 1 is to apply Bellman's *Principle of Optimality* to the length of the shortest path to any node v . This path must have a final arc (u, v) , for some node u . Thus the shortest path from r to v is a path from r to u followed by an arc from u to v . Bellman's principle says that the path from r to u must also be a shortest path from r to u . This means that the shortest path distances must satisfy the following equations :

$$D_r = 0,$$

$$D_v = \min_{u \neq v} \{D_u + d_{uv}\}, \quad v \in N - \{r\}.$$

8.2.3 Linear Programming Interpretation

If we label the nodes with the integers $1, \dots, n$, giving node r the number 1, then Bellman's equations become

$$D_1 = 0,$$

$$D_j = \min_{k \neq j} \{D_k + d_{kj}\}, \quad j = 2, \dots, n.$$

These equations imply that the following set of $n - 1$ inequalities must hold for each j

$$D_j \leq D_k + d_{kj}, \quad k = 1, \dots, j - 1, j + 1, \dots, n.$$

The shortest path problem can be solved by solving the following linear program

$$\text{Maximize } \sum_{i=1}^n D_i$$

subject to

$$D_1 = 1,$$

$$D_j - D_i \leq d_{ij}, \quad i = 1, \dots, n, \quad j = 2, \dots, n, \quad i \neq j.$$

We note that the *dual* of this linear program is a *Minimum Cost Flow* problem, whose formulation is:

$$\text{Minimize } \sum_{(i,j) \in \mathcal{A}} d_{ij} x_{ij}$$

subject to

$$\sum_{j:(1,j) \in \mathcal{A}} x_{1j} - \sum_{j:(j,1) \in \mathcal{A}} x_{j1} = n - 1$$

$$\sum_{j:(i,j) \in \mathcal{A}} x_{ij} - \sum_{j:(j,i) \in \mathcal{A}} x_{ji} = -1, \quad \text{for } i = 2, 3, \dots, n$$

$$0 \leq x_{ij} \leq n - 1, \quad \text{for all } (i, j) \in \mathcal{A}$$

$$x_{ij} \in \{0, 1\}.$$

In this formulation $x_{ij} = 1$ if arc (i, j) is in the shortest path spanning tree.

The optimum solution to this problem sends unit flow from the root to every other node along a shortest path. Thus the flow out of node 1 (root) is $n - 1$ and the flow out of every other node is -1 .

8.2.4 A Relaxation Algorithm

The shortest path problem can be solved by the following very general relaxation algorithm. We may view this algorithm as (i) finding a tree $T(r)$ that satisfies Theorem 1, or (ii) solving Bellman's equations, or (iii) the *dual Simplex method* applied to the linear program. Most shortest path algorithms are based on this algorithm. The tree T is represented by an array, $p[1..n]$, of parent pointers, and the distance from u to the root r in the current tree is $D[u]$.

```

algorithm SPathRelax ( $r, G, T$ )
  for each node  $u \in N$ 
     $D[u] := \infty$ ;  $p[u] := r$  {Initialize}
  endfor
   $D[r] := 0$ ;  $p[r] := \text{null}$ 
  while  $T(r)$  is not an SP Tree do
    for each arc  $(u, v) \in A$  do
      if  $D[v] > D[u] + d_{uv}$  then
         $D[v] := D[u] + d_{uv}$  {relax inequality}
         $p[v] := u$ 
      endif
    endfor
  endwhile
endalg {SPathRelax}

```

The main problem with this algorithm is that it may require $O(2^m)$ iterations. Nonetheless, it is guaranteed to find the shortest path spanning tree in a finite number of steps.

8.3 A PROTOTYPE SHORTEST PATH ALGORITHM

We now develop some shortest path algorithms that will, in general, require fewer steps than the relaxation algorithm. We will show that most shortest path algorithms can be derived from a very simple prototype algorithm due to Gallo & Pallottino, 1986. This has the great benefit that it unifies the study of these algorithms and makes very clear the differences between them.

The only difficulty with the relaxation algorithm above is how to find and select an arc (u, v) that violates Theorem 1. The set of arcs A can be scanned in $O(m)$ time and this will give a subset of arcs that violate the theorem. The question then is: which arc from this subset do we choose? It has been shown that certain sequences of choices can cause the relaxation algorithm to take $O(2^m)$ steps. For this reason we must be very careful in the selection of arcs.

Before we present the prototype algorithm we need to define the abstract data structures that it will use. We give only a high-level description of these structures, leaving the details of their implementation till later.

8.3.1 Graph and Tree Representation

We represent the graph G by n *adjacency lists*, i.e., each node u has a list of nodes adjacent to it and their associated arc distances. We denote the adjacency list of node u by $Adj(u)$. We represent the tree $T(r)$ by an array of parent pointers $p[1..n]$ and an array of distances $D[1..n]$. Thus the parent of any node u is $p[u]$ and its distance from the root r is $D[u]$, with $D[r] = 0$. Finally we use a *selection set* S which will contain a subset of the nodes N .

The prototype algorithm will select nodes from the selection set S and then check the arcs of the selected node's adjacency list to see if they violate Theorem 1.

```

algorithm Proto-SPathTree ( $r, G, p, D$ )
  for each node  $u \in N$ 
     $D[u] := \infty$ ;  $p[u] := r$   {Initialize}
  endfor
   $S := \{r\}$ ;  $D[r] := 0$ ;  $p[r] := \text{null}$ 
  while  $S$  is not empty
     $u := \text{Select}(S)$ 
    for each  $v \in \text{Adj}(u)$ 
      if  $D[v] > D[u] + d_{uv}$  then
         $D[v] := D[u] + d_{uv}$ 
         $p[v] := u$ 
         $\text{Insert}(v, S)$ 
      endif
    endfor
  endwhile
endalg {Proto-SPathTree}

```

There are two crucial and related decisions to be considered in this prototype:

1. The order in which the nodes of S are to be selected.
2. The representation (structure) of S and the complexity of the operations on S .

8.3.2 Selection Rules and S -Structures

The following is a table of selection rules and their S structures.

TABLE II. SELECTION RULES & STRUCTURES

SELECTION RULE	S -STRUCTURE	INVENTOR
Best First	Unordered List	Dijkstra
	d -Heap	Johnson
	Buckets	Dial
	R -Heap	Tarjan <i>et al.</i>
Breadth First	Queue	Bellman, Ford, Moore
Depth First	Stack	
Topological	Loop	
Hybrid 1	Deque	Pape
Hybrid 2	2 Queues	Glover, Klingman
Hybrid 3	2 Queues	Goldfarb, <i>et al.</i>

8.4 BEST FIRST ALGORITHMS

The selection rule for these algorithms is

Select $u \in S$ with smallest $D[u]$

The most famous of this type is *Dijkstra's Shortest Path Algorithm*. Indeed, all algorithms in this section are variations of Dijkstra's original 1959 algorithm.

Theorem 8.2 (). *If G has positive arc distances then when any u is selected from S its distance $D[u]$ is optimal and it is never added to S again.* □

8.4.1 Dijkstra's Algorithm using an Unordered List

This is the original form of Dijkstra's shortest path algorithm.

```

algorithm Dijkstra-SPathTree ( $r, G, p, D$ )
{ $S$  is represented as an unordered list}
Initialize ( $r, G, p, D$ )
 $S := N$ ;  $D[r] := 0$ ;  $p[r] := \text{null}$ 
while  $S$  is not empty do
     $u := \text{DeleteMin}(S)$ 
    for each  $v \in \text{Adj}(u)$  do
        if  $D[v] > D[u] + d_{uv}$  then
             $D[v] := D[u] + d_{uv}$ 
             $p[v] := u$ 
        endif
    endfor
endwhile
endalg {Dijkstra-SPathTree}

```

Complexity : $O(n^2)$ for positive arcs,

Notice that the $\text{Insert}(v, S)$ operation has been omitted above because of Theorem 8.2. At any stage of this algorithm the nodes in S are called *temporary* because they have not got their final (optimal) $D[\cdot]$ value. The nodes not in S are called *permanent* and these have their optimal $D[\cdot]$ value.

The most costly operation in Dijkstra's algorithm is the $\text{DeleteMin}(S)$ operation. Dijkstra originally used an unordered array for S and finding and extracting the minimum u required $O(n)$ operations.

8.4.2 Dijkstra's Algorithm using a Heap

If we use a *Min-Heap* to represent S then all operations on S require $O(\log n)$ time.

Theorem 8.3 (). *The sequence of $D[u]$ values for any node u decreases monotonically from ∞ to $D^*[u]$, the optimal value.* □

We now give Tarjan's version of the Best First algorithm which uses a heap for S . It also uses Theorem 8.3 to save some work. In this clever version of the best-first shortest path algorithm there is no need to explicitly make a heap at the start — it is built up node-by-node by the Insert operation.

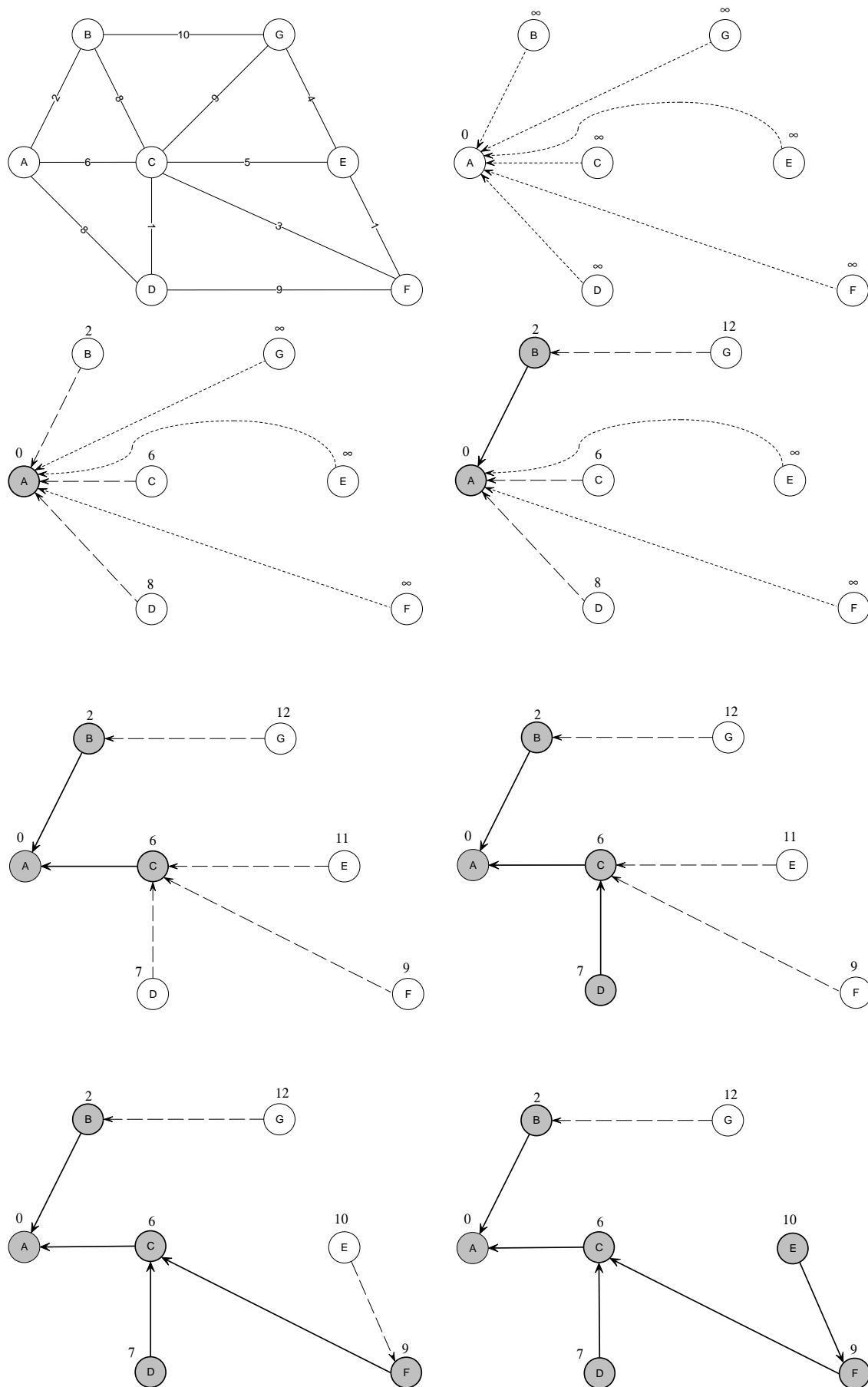


Figure 8.2. Dijkstra's Algorithm.

```

algorithm Tarjan-SPathTree ( $r, G, p, D$ )
{ $S$  is represented as a Heap}
Initialize ( $r, G, p, D$ )
 $S := \text{empty}; D[r] := 0; p[r] := \text{null}$ 
 $u := r$ 
while  $u \neq \text{null}$  do
  for each  $v \in \text{Adj}(u)$  do
    if  $D[v] > D[u] + d_{uv}$  then
       $D[v] := D[u] + d_{uv}$ 
       $p[v] := u$ 
      if  $v \notin S$  then
        Insert ( $v, S$ )
      else
        SiftUp ( $v, S$ )
      endif
    endif
  endfor
   $u := \text{DeleteMin}(S)$ 
endwhile
endalg Tarjan-SPathTree.

```

Complexity : $O(m \log_2 n)$.

If a d -heap is used then SiftUp is $O(\log_d n)$ and SiftDown is $O(d \log_d n)$. This gives a complexity of $O(dn \log_d n + m \log_d n) = O((dn + m) \log_d n)$. If there are many more arcs than nodes there will be more SiftUps than SiftDowns.

Exercise 8.4.1 : What is the best value for d as a function of m and n ? □

8.5 BREADTH FIRST ALGORITHMS

The original Breadth First algorithm was due to Bellman, Ford and Moore. Bellman stated necessary conditions for optimality of paths as follows :

$$D_r = 0, \quad D_v = \min_{u \neq v} \{D_u + d_{uv}\}, \quad v \in N - \{r\}.$$

These are known as *Bellman's Equations* and are equivalent to the condition of Theorem 1 above. These conditions are also sufficient if there are no negative cycles in G . Bellman proposed that these equations be solved by *successive approximations* as follows :

Initialize the D 's

$$\begin{aligned} D_r^{(0)} &= 0, \\ D_u^{(0)} &= \infty, \quad u \neq r, \end{aligned}$$

and then compute the $k + 1$ st approximation

$$D_v^{(k+1)} = \min\{D_v^{(k)}, \min_{u \neq v} (D_u^{(k)} + d_{uv})\}.$$

It is obvious that, for each u ,

$$D_u^{(1)} \geq D_u^{(2)} \dots \geq D_u^{(n-1)}.$$

Bellman showed that the successive approximations $D_u^{(k)}$ converge to their optimum shortest path values after $n - 1$ iterations, i.e., $D_u^{(n-1)} = D_u^*$, the length of the shortest path from r to u .

We can now state Bellman's Shortest Path Algorithm.

algorithm *BellmanSP* (r, G, p, D)

```

Initialize( $r, G, p, D$ )
for  $i := 1$  TO  $n - 1$  do
  for each arc  $(u, v) \in G$  do
    if  $D[v] > D[u] + d_{uv}$  then
       $D[v] := D[u] + d_{uv}$ 
       $p[v] := u$ 
    endif
  endfor
endfor
endalg {BellmanSP}

```

The outer loop is performed $n - 1$ times and the inner loop is performed m times for each iteration of the outer loop. Hence the complexity is $O(mn)$. The algorithm works with negative arcs and if there is a negative cycle this can be detected by executing the outer loop for the n th time. If a D value changes in this pass, the graph has a negative cycle. Thus we can modify Bellman's algorithm to detect negative cycles in $O(mn)$ time.

Exercise 8.5.1 : NEGATIVE CYCLES. Modify Bellman's algorithm to detect and then *identify* a negative cycle, i.e., find the nodes in the negative cycle. \square

As it stands, Bellman's algorithm is not very good for most graphs. If G is dense and has no negative cycles it takes $O(n^3)$ time, while Dijkstra's takes $O(n^2)$ time.

Exercise 8.5.1 : EARLY STOPPING OF BELLMAN'S ALGORITHM. It can be seen that the **for** -loop in Bellman's algorithm is a direct implementation of the optimality test in Theorem 1. Hence, as soon as this test has been passed we may stop the algorithm (we have constructed a spanning tree that passes the Theorem 1 test). Show how to modify Bellman's algorithm so that it stops as soon the test of Theorem 1 has been passed. \square

Bellman's algorithm scans each arc of G each time the outer loop is executed and there is no defined order in the arc scanning. Let us scan the arcs in adjacency list order : take each node and scan its adjacency list. Now, if during one pass through all the arcs the $D[u]$ of node u does not change, then in the next pass we must have $D[v] \leq D[u] + d_{uv}$ for every $v \in Adj(u)$. Hence we do not need to scan the arcs out of u in this pass. We can avoid unnecessary scanning by keeping a list of nodes whose D 's have changed since we last looked at them, and scanning the arcs of these nodes only. This list is, of course the selection set S .

The selection rule for these algorithms is

Select the 'oldest' $u \in S$

This rule means that we select the node u that has spent the longest time in S . This is also called the 'first-in, first-out', or FIFO rule. This rule is easily implemented if we operate S as a *Queue*.

Algorithms that use this rule, or variations of it, are called *label correcting* algorithms. Bellman, Ford, and Moore independently suggested this rule for the solution of the Bellman equations. In these algorithms nodes are not guaranteed to get their optimal $D[\cdot]$ values until the final iteration. Also, these algorithms are more robust because they run in polynomial time even if G has negative arcs.

```
algorithm BFM-SPathTree ( $r, G, p, D$ )
```

```
{ $S$  is represented as a Queue}  
Initialize ( $r, G, p, D$ )  
 $S :=$  empty;  
 $u := r$   
while  $u \neq$  null do  
  for each  $v \in \text{Adj}(u)$  do  
    if  $D[v] > D[u] + d_{uv}$  then  
       $D[v] := D[u] + d_{uv}$   
       $p[v] := u$   
      if  $v \notin S$  then EnQueue ( $v, S$ )  
    endif  
  endfor  
   $u :=$  DeQueue ( $S$ )  
endwhile  
endalg BFM-SPathTree.
```

Complexity : The outer loop is performed $n - 1$ time. The inner loop is perform $O(m)$ times. Hence this algorithm is $O(mn)$ for positive or negative arcs.

We now use a small example to see the workings of the BFM algorithm.

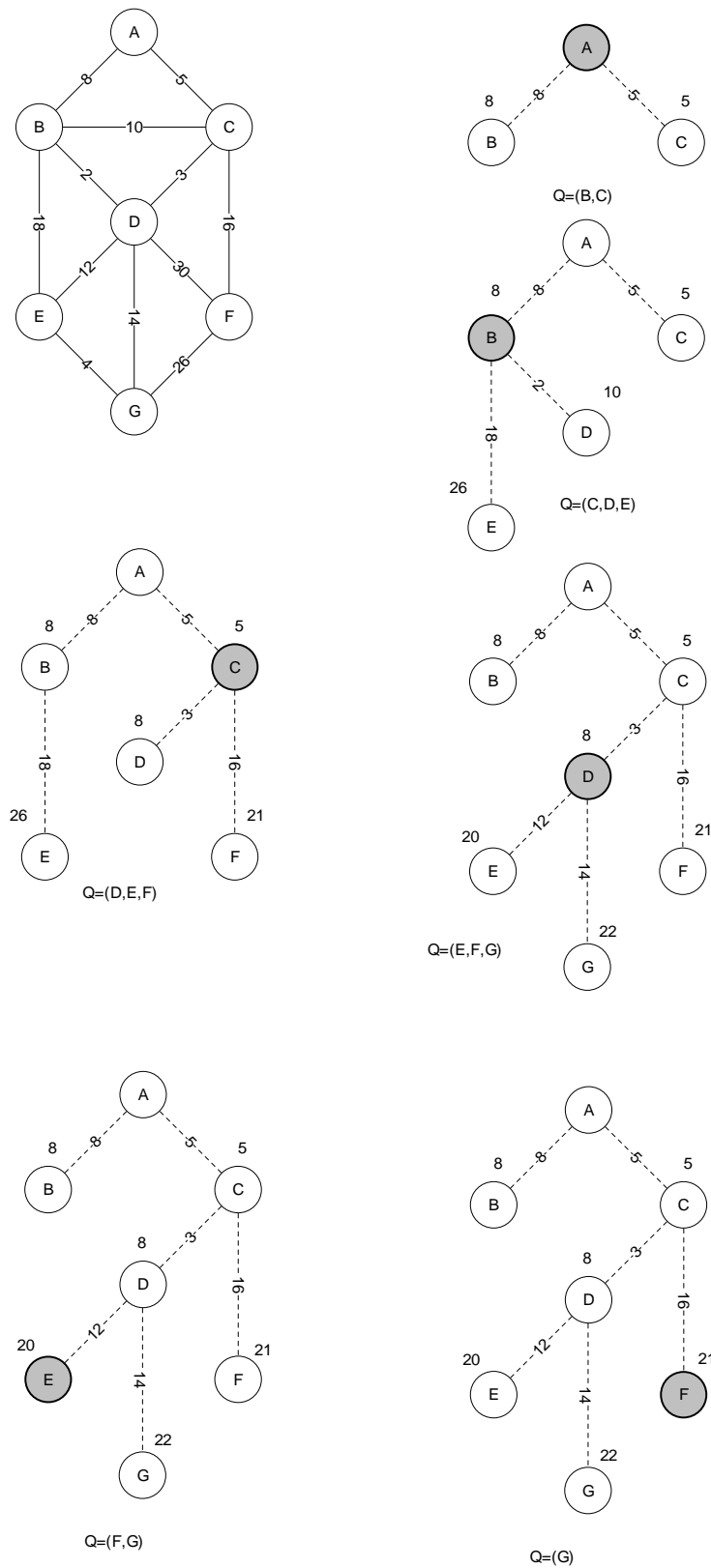


Figure 8.3. BFM-SpathTree Algorithm.

8.5.1 D'Esopo-Pape Modification

Pape's modification of the BFM algorithm uses an output restricted *Deque* to represent S . This structure is the same as a queue but, in addition, allows nodes to be inserted at the front. This means that a deque acts as a stack and a queue.

Pape's rule:

If a node has never been on S then add it to the back of S ; otherwise add it to the front of S .

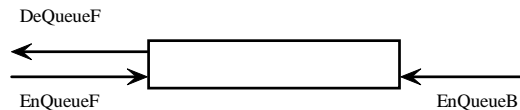


Figure 8.4. An Output-Restricted Deque.

```

algorithm Pape-SPathTree ( $r, G, p, D$ )
  { $S$  is represented as an output-restricted Deque}
  Initialize ( $r, G, p, D$ )
   $S := \text{empty}; D[r] := 0; p[r] := \text{null}$ 
   $u := r$ 
  while  $u \neq \text{null}$  do
    for each  $v \in \text{Adj}(u)$  do
      if  $D[v] > D[u] + d_{uv}$  then
        if  $v \notin S$  then
          if  $D[v] = \infty$  then
            EnQueueB ( $v, S$ )
          else
            EnQueueF ( $v, S$ )
          endif
           $D[v] := D[u] + d_{uv}$ 
           $p[v] := u$ 
        endif { $v$ }
      endif { $D[v]$ }
    endfor
     $u := \text{DeQueue}(S)$ 
  endwhile
endalg Pape-SPathTree.

```

Complexity : $O(n2^n)$ for positive or negative arcs. A slight modification of Pape's algorithm, suggested by Gallo & Pallottino 1986, reduces this exponential complexity to $O(mn)$.

8.6 IMPLEMENTATION & TESTING

As we have seen, Shortest Path algorithms are extremely simple. They contain 2 loops : a **for**-loop nested inside a **while**-loop. Because of this simplicity careful implementation is vital. Even small variations between two different implementations can cause great differences in performance, especially when solving large problems.

Most of the published performance results seem to be based on programs written in Fortran and, of necessity, using arrays only for the data structures. Also, there seems to be a myth in the Operations Research-Engineering folklore that says Fortran is a fast and efficient language. This myth is very convenient for those who know only Fortran and know nothing about dynamic memory allocation. We hope to undermine this myth with the experimental results below.

Because the algorithms are simple the main implementation issues are the choice and design of the data structures used for the selection set S .

8.6.1 Data Structures

We wish to solve shortest path problems on large sparse networks. Real networks are large and sparse with $n = [1000 - 10000]$ and $m = O(n)$. For example the road networks used in Section 5 below range in size from $(n, m) = (981, 2654)$ to $(4900, 11200)$.

Large sparse networks cannot be stored using matrices because matrix storage requires $O(n^2)$ space. For this reason we must use linked storage for large sparse structures. Other structures such as vectors that are static and dense can and should be stored as arrays.

The data structures used to implement the algorithms above are as follows :

- Graph G : An array $[1..n]$ of pointers to singly-linked adjacency lists. Size $n + O(m)$.
- SP Tree $T(r)$: A parent array $p[1..n]$ to store the tree structure (paths), and a distance array $D[1..n]$ to hold the path lengths. Size : $2n$
- S -Structure :
 - Heap : Two arrays $H[1..n]$ and $Hpos[1..n]$. Size : $2n$.
 - Queue : Singly-linked list. Size $O(n)$.
 - Buckets : Array $B[0..\max(d_{uv})]$ pointing to doubly-linked lists and an array $Addr[1..n]$ of pointers to nodes in buckets. Size : $n + O(n) + O(\max(d_{uv}))$.

8.6.2 Computational Tests

Language-Compiler-Machine. The algorithms and data structures above were implemented in Turbo Pascal 5.5 and run on a 10MHz AT micro-computer.

Test Data. Proper testing requires that the algorithms be tested on as wide a range of problem type as possible. Randomly generated problems can only test general features of the algorithm. Ultimately any algorithm must be tested on problems it is intended to solve, not on synthetic problems.

The following problem parameters should be considered when testing Shortest Path algorithms.

1. Large or small arc lengths
2. Real or Integer arc lengths
3. Sparse or dense graph, i.e., $m = O(n)$ or $O(n^2)$.
4. Special structure, e.g., lattice, Euclidean.

The algorithms were tested on the following sets of networks all of which had non-negative integer arc lengths:

1. *Road Networks* : Name & Sizes (n, m) : Sri Lanka (981,2654), Ireland (1797,5620), Dublin (1830, 6278), Mallow (4923, 11358).
2. *Random Networks* : (500,5000-15000), (1000,5000-15000), (5000, 10000-15000)
3. *Grid Networks* : 50 x 50, 60 x 60, 70 x 70.

ShortestPath Algorithms Annals OR, Vol13.

Time (secs) - 10MHz AT

	SL981	IRL1797	DUB1830	MAL4923	TOTAL	RATD
BFM-Deque FORT	0.720	1.360	1.581	3.383	7.044	4.37
Dijk-BktFORT	0.973	2.076	1.790	4.048	8.887	5.51
BFM-Deque PAS	0.166	0.310	0.346	0.790	1.612	1.00
Dijk-BktPAS	0.179	0.333	0.367	0.828	1.707	1.06

NOTE : FORTRAN programs compiled with IBM RM ProFortver1.0
Pascal programs compiled with Turbo Pascal5.5

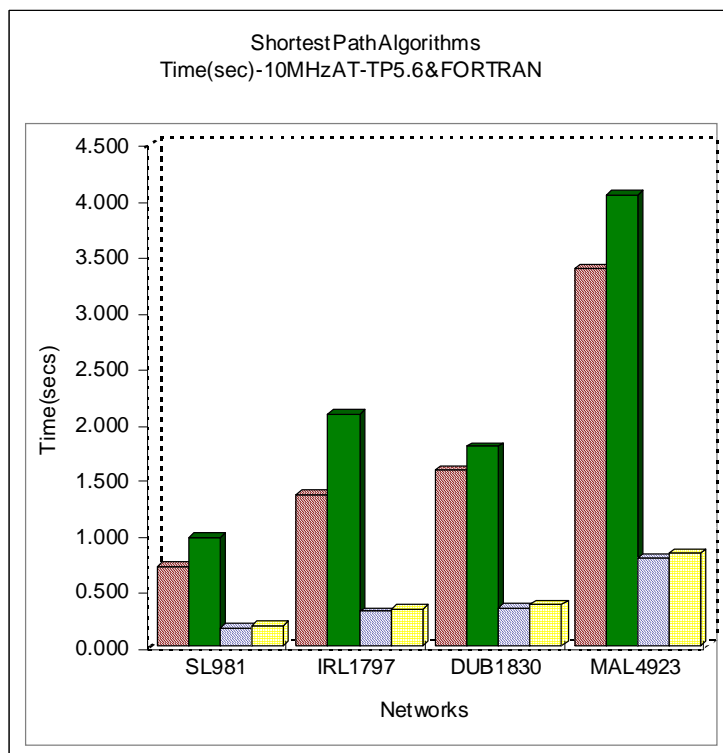


Figure 8.5. .