Chapter 7

▪ MINIMUM SPANNING TREES

7.1 INTRODUCTION

Given a weighted graph $G = (N, A, d)$ in which each arc $(u, v) \in A$ has a weight d_{uv}, find a spanning tree of minimum total weight, i.e., find a spanning tree T that minimizes

$$D(T) = \sum_{(u,v) \in T} d_{uv}$$

Minumum Spanning Tree Property : *Given any partition of the nodes N into two sets U and V, then a minimum spanning tree contains the shortest arc connecting a node $u \in U$ to a node $v \in V$.*
Algorithms for minimum spanning trees fall into two types : (1) those that build a tree from a collection of subtrees and (2) those that maintain a single subtree. Both types use the MST property to add a smallest arc to the existing subtree(s), as shown in Figure 2.17.

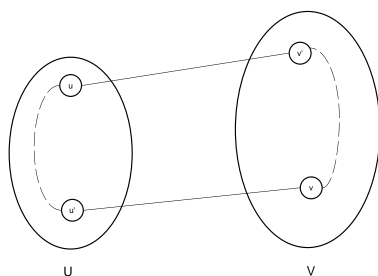


Figure 7.1. Constructing Minimum Spanning Trees.

7.2 KRUSKAL'S MST ALGORITHM

This algorithm starts with an unconnected set of nodes N and an empty tree T. The main step is : add the smallest available arc to T that does not form a cycle. The algorithm stops when $n - 1$ arcs have been added to T.

Copyright © 2001, by Derek R. O'Connor.

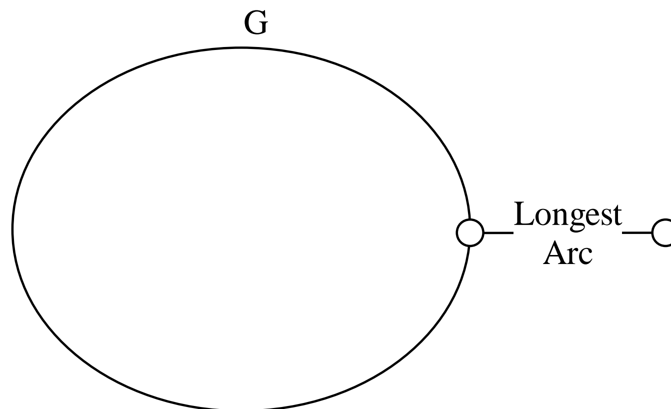


Figure 7.2. The Longest Arc must be in the MST.

algorithm *Kruskal1* $(G = (N, A), T)$

```
        T := ∅
        WHILE |T| < n − 1 DO
            (u, v) := DeleteMinArc(A)
            IF (u, v) forms no cycle in T THEN
                  Add((u, v), T)
            ENDIF
        ENDWHILE
ENDALG Kruskal1
```

Figure ?? shows an example of Kruskal's algorithm in operation

Implementation of Kruskal's Algorithm. The while-loop contains two important operations : DeleteMinArc and cycle-checking. The while-loop is performed a minimum of $n-1$ times and a maximum of m times because the longest arc could be in the minimum spanning tree, as shown in Figure ??.

The DeleteMinArc operation is $O(1)$ if the arcs are pre-sorted, at a cost of $O(m \log_2 m)$. The cycle-checking operation can be implemented using the *UFSet* ADT. This is done by maintaining each subtree as a disjoint set and if both nodes of (u, v) are in the same subtree then a cycle is formed. See Figure 2.19.

Copyright © 2001, by Derek R. O'Connor.

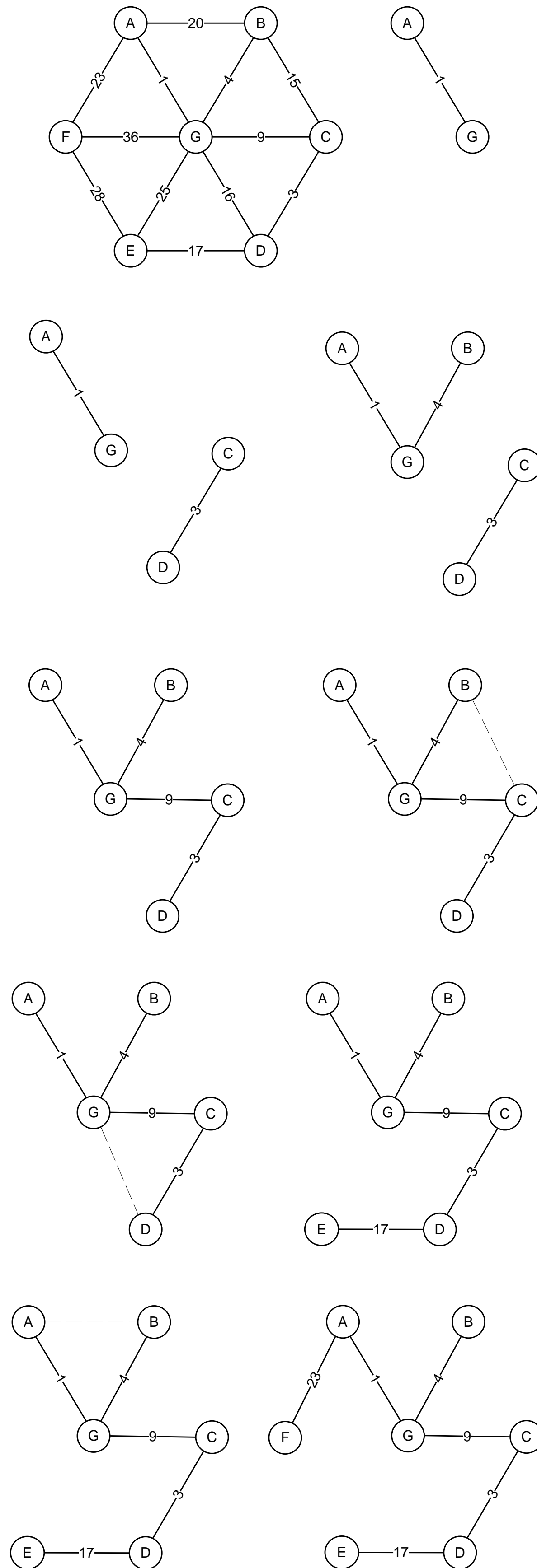


Figure 7.3. Kruskal's Algorithm.

Copyright © 2001, by Derek R. O'Connor.

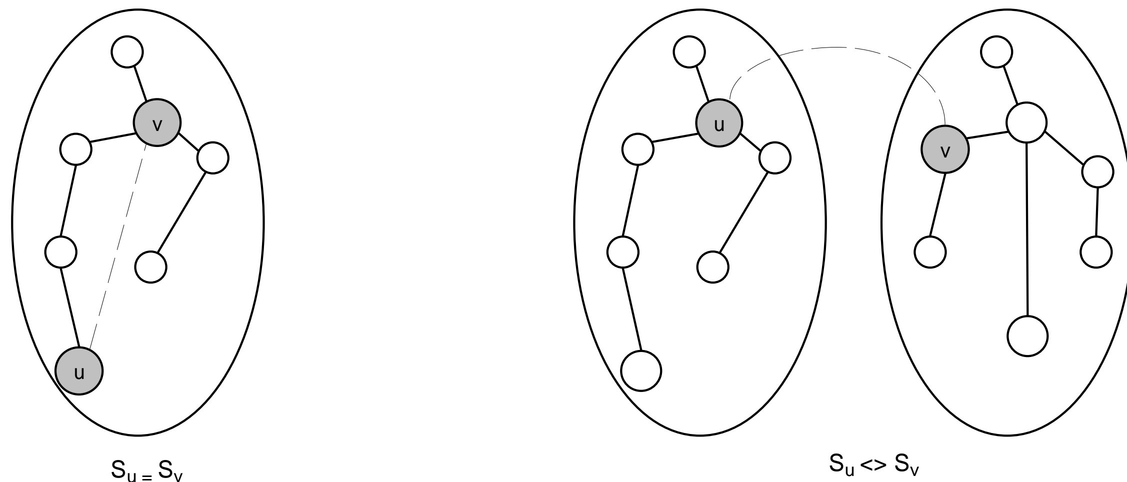


Figure 7.4. Cycle Checking in Kruskal's Algorithm.

algorithm *Kruskal2* $(G = (N, A), T)$

```
    T := ∅
    Sort(A)
    MakeUFSet(N)
    WHILE |T| < n − 1 DO
        (u, v) := DeleteMinArc(A)
        S_u := Find(u)
        S_v := Find(v)
        IF S_u ≠ S_v THEN
              Union(S_u, S_v)
              Add((u, v), T)
        ENDIF
    ENDWHILE
ENDALG Kruskal2
```

Analysis. Sorting m arcs requires $O(m \log_2 m)$ operations. MakeUFSet is $O(n)$.DeleteMinArc, Find, and Union are $O(1)$. These are performed $O(m)$ times in the while-loop. Hence the total is $O(m \log_2 m) + O(n) + mO(1) = O(m \log_2 m)$.

Exercise 7.2.1 : *Kruskal with Heaps* An alternative to pre-sorting the arcs is to form a heap initially and then use the heap DeleteMin to extract the next arc. This may be better than sorting if the graph is dense and the while-loop is performed $O(n)$ times. Write the pseudo-code for this modification and analyse it.

Exercise 7.2.1 : An On-Line MST Algorithm. Devise an algorithm that constructs an MST with the arcs given one-at-a-time in any order. [HINT : You must check for and *identify* cycles]

Copyright © 2001, by Derek R. O'Connor.

7.3 PRIM'S MST ALGORITHM

This algorithm maintains a single subtree T and a subset of unconnected nodes U. The main step is : add the smallest arc that connects the subtree to an unconnected node.

algorithm *Prim1* $(G = (N, A), T)$

```
        T := ∅
        S := any node s ∈ N
        U := N − {s}
        WHILE |S| < n DO
            (s, u) := MinArc(s ∈ S, u ∈ U)
            Add(u, S)
            Delete(u, U)
            Add((s, u), T)
        ENDWHILE
ENDALG Prim1
```

Our initial implementation of Prim's algorithm maintains two sets : S are the nodes in the subtree and $U = N - S$, i.e., the unconnected nodes. The main step moves a node from U to S until $|S| = n$.

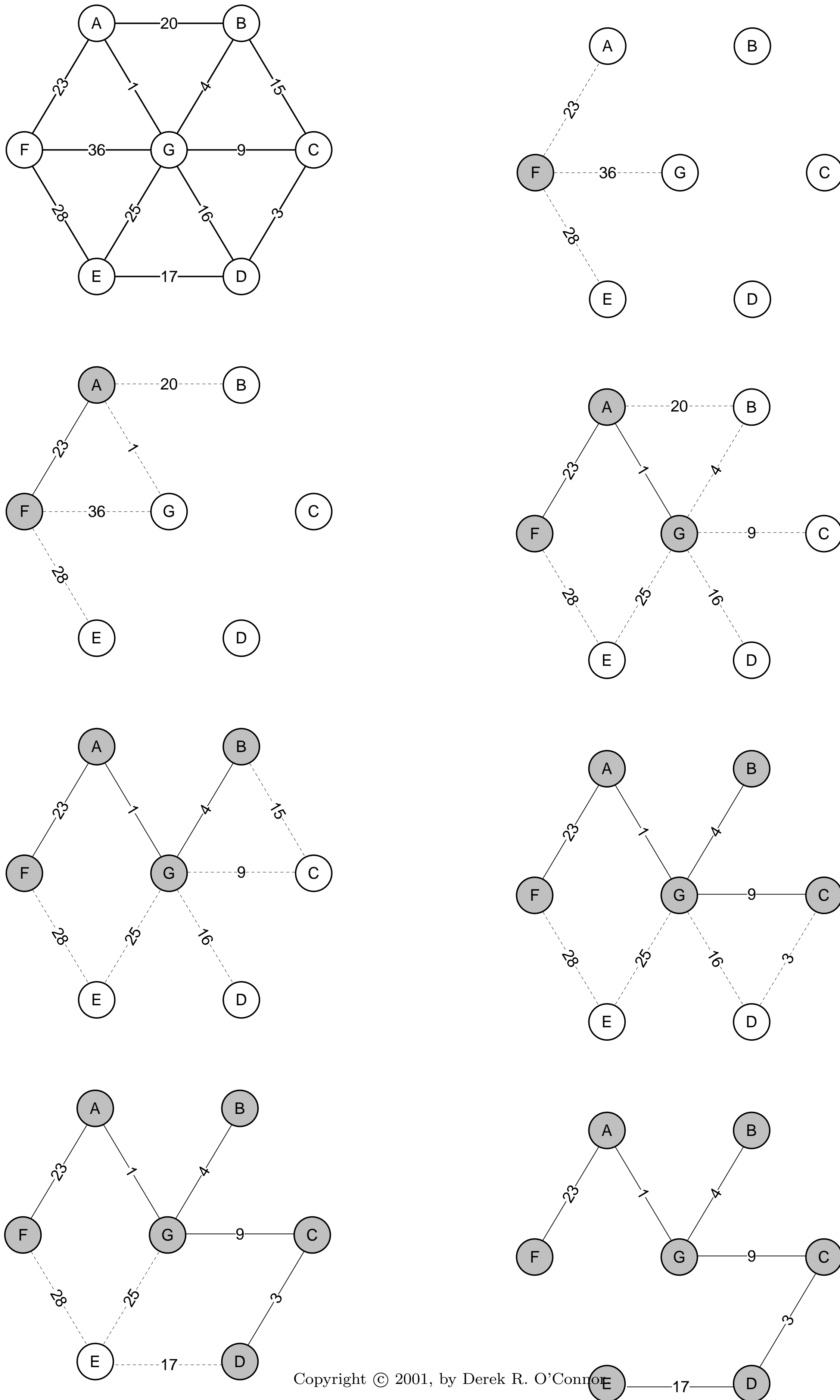
The while-loop is performed $n - 1$ times. The sets S,U, and T can be represented by arrays and the set operations can then be implemented in $O(1)$ time. The MinArc$(s \in S, u \in U)$ is expensive because we need to every $u \in U$ for every $s \in S$, i.e., every arc. Hence MinArc is $O(m)$ and the total running time is $O(mn)$. This is $O(n^3)$ if the graph is dense.

We can speed up Prim1 if we maintain two extra pieces of information for each node $u \in U$:

1. $p[u]$: the node in S nearest to u, and
2. $D[u]$: the length of the arc $(u, p[u])$.

This information is stored in arrays $p[1..n]$ and $D[1..n]$ and is shown in Figure ??. It can be seen that the arrays $p[1..n]$ and $D[1..n]$ need to be updated at every step.

Copyright © 2001, by Derek R. O'Connor.



Copyright © 2001, by Derek R. O'Connor

Figure 7.5. Prim's Algorithm in Operation.

```
algorithm Prim2 (G = (N, A), T)

      T := ∅
      S := any node s ∈ N
      U := N − {s}
      FOR each u ∈ U DO
            p[u] := s
            D[u] := d_su
      ENDFOR
      WHILE |S| < n DO
         s := MinNode(u ∈ U)
         Add(s, S)
         Delete(s, U)
         Add((s, p[s]), T)
         FOR each node u ∈ U DO
              IF D[u] > d_su THEN
                  D[u] := d_su
                  p[u] := s
              ENDIF
         ENDFOR
      ENDWHILE
ENDALG Prim2
```

Analysis of Prim2. Initialization of $p[1..n]$ and $D[1..n]$ is $O(n)$. The while-loop is performed $n-1$ times. The MinNode operation is $O(n)$ and the set operations are $O(1)$. The for-loop that updates p and D is $O(n)$. Hence the total time is $O(n)+(n-1)(O(n)+O(1)+O(n)) = O(n^2)$.

The most expensive operation in this version of Prim is MinNode. It is $O(n)$ and is performed $O(n)$ times. This dominates the running time of the algorithm. We can speed up this operation if we implement U as a heap, ordered by $D[u]$. The MinNode operation becomes the heap operation DeleteMin which is $O(\log_2 n)$. The heap U needs to be updated each time the IF-statement is executed. This requires a SiftUp operation because $D[u]$ can only *decrease* in value. We now use these ideas and others as given in Tarjan's monograph, page 77. This is a very elegant and efficient implementation of Prim's algorithm.

Copyright © 2001, by Derek R. O'Connor.

algorithm *Prim3* $(G = (N, A), p, D)$

```
        FOR each u ∈ N DO
              D[u] := ∞
        ENDFOR
        s := any node in N
        U := empty heap
        WHILE s ≠ NULL DO
            D[s] := −∞
            FOR each node u ∈ Adj(s) DO
                 IF D[u] > d_su THEN
                     D[u] := d_su
                     p[u] := s
                     IF u ∉ U THEN
                           Insert(u, U)
                     ELSE
                           SiftUp(u, U)
                     ENDIF
                 ENDIF
            ENDFOR
            s := DeleteMin(U)
        ENDWHILE
ENDALG Prim3
```

Analysis of Prim3. Initialization of $D[1..n]$ is $O(n)$. The while-loop is performed $n-1$ times because each node is inserted and therefore deleted once and only once. The FOR-loop traverses the adjacency list of each node once. Hence the statements in the FOR- loop are performed a total of $O(m)$ times. The heap operations Insert, SiftUp, and DeleteMin are all $O(\log_2 n)$. Hence the total time is $O(n) + mO(\log_2 n) = O(m \log_2 n)$.

Implementation Details. The statement $D[s] := -\infty$ is necessary to exclude nodes on $Adj(s)$ that are already in the connected subtree. The heap operations need to be implemented with an additional array Pos$[1..n]$ because the heap operations need to know the position of node u in the heap. No MakeHeap operation is necessary : it is built a node-at-a-time by the Insert operation. The spanning tree is stored in the parent array $p[1..n]$. The following example illustrates these ideas.

Copyright © 2001, by Derek R. O'Connor.