Prim's Minimum Spanning Tree Algorithm

Discussion

Prim's minimum spanning tree algorithm is a greedy algorithm in which the minimum spanning tree grows from some initial vertex by adding the smallest edge that is adjacent to the existing tree at each step.

Initially all vertices are designated as unexplored except the starting vertex, which is designated as a tree vertex. The vertices adjacent to the starting vertex are classified as belong to the fringe and their distance from the starting vertex is recorded. Then the smallest edge from the starting vertex to one of the fringe vertices is added to the tree and is labeled as a tree vertex. Once an edge is added to the tree, vertices adjacent to the added vertex become fringe vertices and their distance from the tree are recorded. It is important to note that each time a new edge and vertex is added the distance from the tree to any of the fringe vertices may change and must be updated. This process continues until all of the vertices have been labeled as tree vertices.

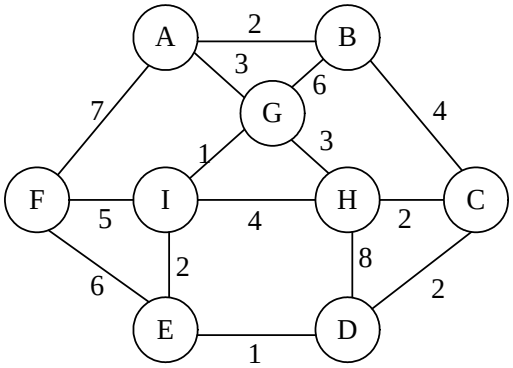
The pseudocode for Prim's algorithm is shown below where `G` is the weighted graph and `s` is the starting vertex.

```
prims (G, s)
  initialize all vertices except the starting vertex as unexplored
  make the starting vertex a tree vertex
  make all vertices adjacent to the starting vertex fringe vertices and
    record their distance
  while fringe vertices remain
    choose the closest fringe vertex
    for all adjacent vertices
      if vertex is unexplored
        make it a fringe vertex and record its distance from the tree
      else if it is already in the fringe
        if the weight for the new edge is less than current distance
          update the distance to be the lower weight
```

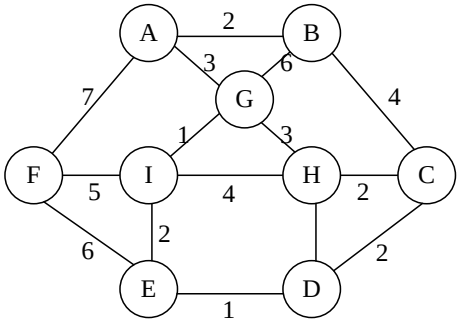
The data structure used to implement the list of fringe vertices conceptually is a priority queue, but the most efficient implementation for this algorithm is not using a heap. Instead an array of vertices is maintained that contains a field that indicates the current status of each vertex, whether it is unexplored, in the fringe or a part of the tree. In addition, a field in needed to record the current closest distance from every fringe vertex to the tree. Choosing the closest fringe vertex involves a linear search of the array. If we wish to output the edges of the tree it is necessary to also maintain the emanating vertex of the edge associated with each vertex in the array.

Sample Problem

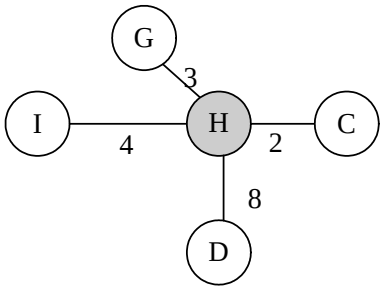
Execute Prim's minimum spanning tree algorithm by hand on the graph below showing how the data structures evolve specifically indicating when the distance from a fringe vertex to the tree is updated. Clearly indicate which edges become part of the minimum spanning tree and in which order. Start at vertex H.



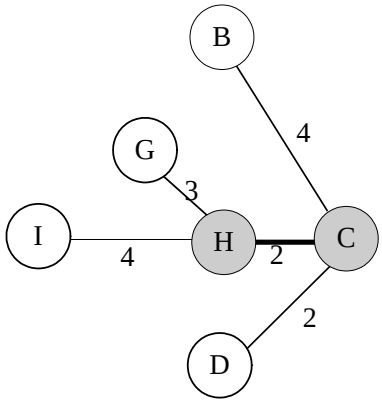
Solution



A	U		
B	U		
C	U		
D	U		
E	U		
F	U		
G	U		
H	F	0	minimum
I	U		

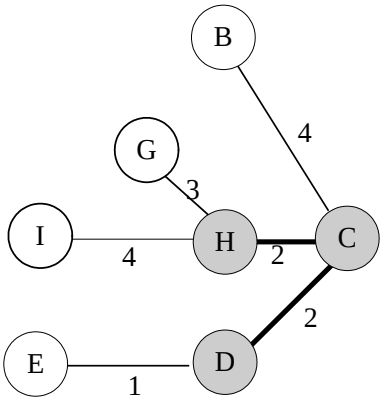


A	U		
B	U		
C	F	2	minimum
D	F	8	
E	U		
F	U		
G	F	3	
H	T		
I	F	4	



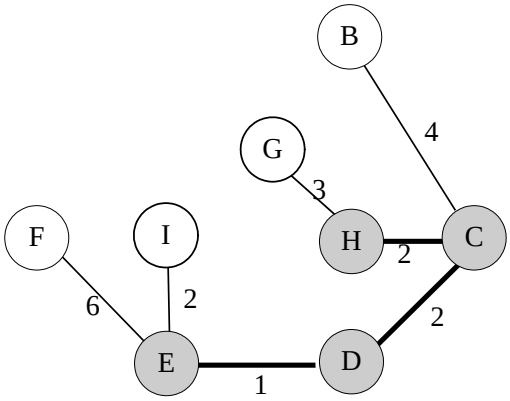
A	U	
B	F	4
C	T	
D	F	2
E	U	
F	U	
G	F	3
H	T	
I	F	4

minimum, updated



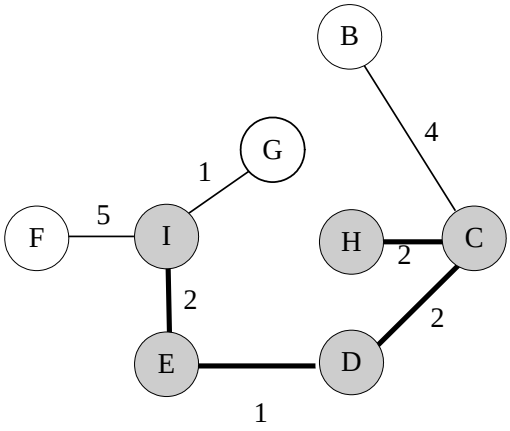
A	U	
B	F	4
C	T	
D	T	
E	F	1
F	U	
G	F	3
H	T	
I	F	4

minimum



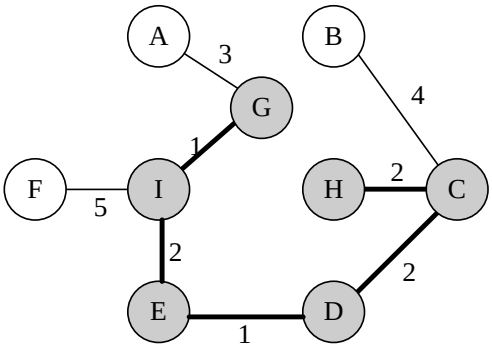
A	U	
B	F	4
C	T	
D	T	
E	T	
F	F	6
G	F	3
H	T	
I	F	2

minimum, updated



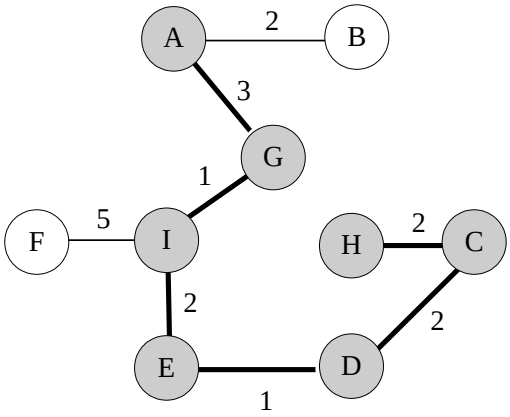
A	U	
B	F	4
C	T	
D	T	
E	T	
F	F	5
G	F	1
H	T	
I	T	

updated
updated, minimum



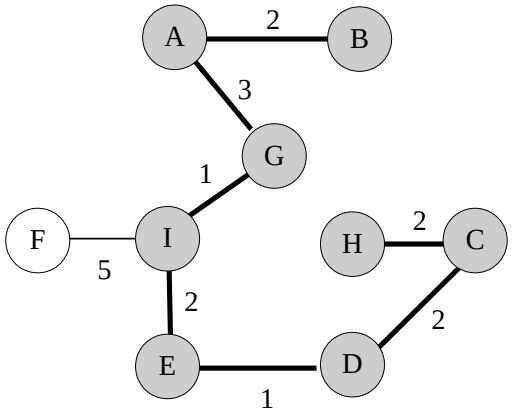
A	F	3
B	F	4
C	T	
D	T	
E	T	
F	F	5
G	T	
H	T	
I	T	

minimum



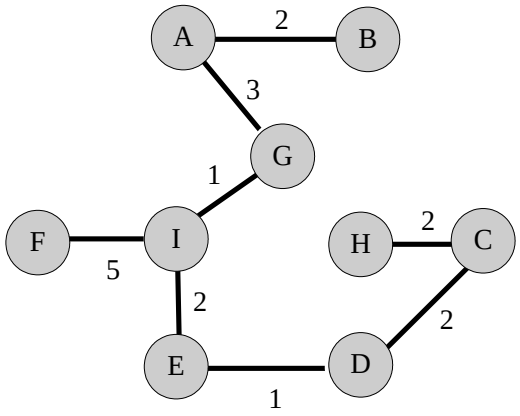
A	T	
B	F	2
C	T	
D	T	
E	T	
F	F	5
G	T	
H	T	
I	T	

updated, minimum



A	T	
B	T	
C	T	
D	T	
E	T	
F	F	5
G	T	
H	T	
I	T	

minimum



A	T	
B	T	
C	T	
D	T	
E	T	
F	T	
G	T	
H	T	
I	T	

Kruskal's Minimum Spanning Tree Algorithm

Discussion

Like Prim's algorithm, Kruskal's algorithm is also a greedy algorithm. It constructs the minimum spanning tree in a different manner however. At each step of Prim's algorithm the edges being added form a single tree, but with Kruskal's algorithm, the added edges form a forest of trees. It is only at the final step that we are guaranteed to have a single tree.

With this algorithm all the edges of the graph are added to a priority where the weight of an edge is the priority and lowest weights are considered to be highest priority. A dynamic equivalence relation (or disjoint union) data structure is initialized so that each vertex is in its own equivalence class. Then the edges are successively dequeued. A check is made to determine whether the two vertices of the edge belong to the same dynamic equivalence class. If they are in the same class, adding the edge would form a cycle, so that edge is skipped. Otherwise the edge is added to the forest and the equivalence classes to which the two vertices of that edge belong are combined into a single equivalence class. Once the dynamic equivalence relation contains a single equivalence class the minimum spanning tree is complete.

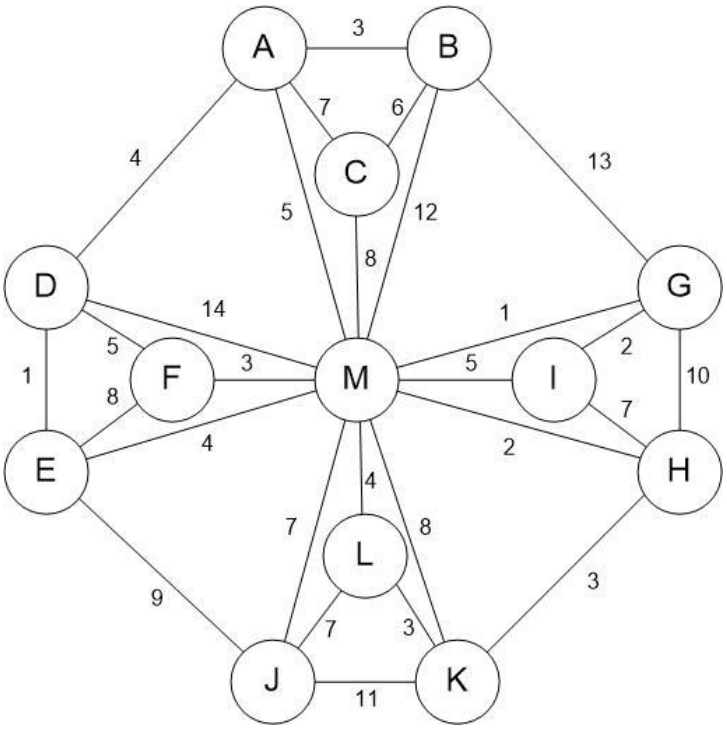
The pseudocode for Kruskal's algorithm is shown below where `G` is the weighted graph.

```
kruskals (G)
  add every edge of the graph to the priority queue
  initialize the dynamic equivalence relation so each vertex is in its own
    class
  while the equivalence relation has more than one class
    dequeue the edge of lowest weight
    if the vertices of that edge belong to the same equivalence class
      skip that edge
    else
      form the union of the two equivalence classes
      add that edge to the forest
```

Unlike Prim's algorithm, the priority queue used by this algorithm can be the customary implementation using a heap. The dynamic equivalence relation used is implemented with a forest of in-trees. The union operation simply makes the tree with the fewest nodes a subtree of the other so it executes in constant time. The find operation must perform path compression when it executes to ensure that it too runs in constant time.

Sample Problem

Execute Kruskal's algorithm on the weighted tree shown below. Assume that edges of equal weight will be in the priority queue in alphabetical order. Clearly show what happens each time an edge is removed from the priority queue and how the dynamic equivalence relation changes on each step and show the final minimum spanning tree that is generated.

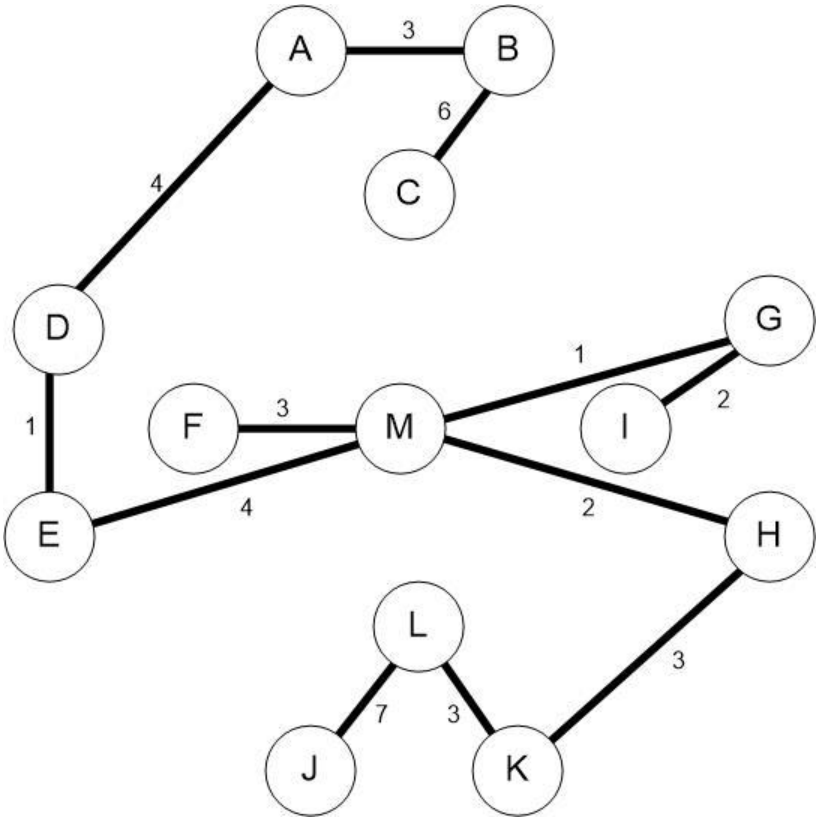


Solution

Below is the action each time an edge is removed from the priority queue and the state of the dynamic equivalence relation after that edge has been processed. Only the equivalence classes that contain more than a single vertex are shown:

Choose DE 1 (D, E)
Choose GM 1 (D, E) (G, M)
Choose GI 2 (D, E) (G, I, M)
Choose HM 2 (D, E) (G, H, I, M)
Choose AB 3 (A, B) (D, E) (G, H, I, M)
Choose FM 3 (A, B) (D, E) (F, G, H, I, M)
Choose HK 3 (A, B) (D, E) (F, G, H, I, K, M)
Choose KL 3 (A, B) (D, E) (F, G, H, I, K, L, M)
Choose AD 4 (A, B, D, E) (F, G, H, I, K, L, M)
Choose EM 4 (A, B, D, E, F, G, H, I, K, L, M)
Skip LM 4
Skip AM 5
Skip DF 5
Skip IM 5
Choose BC 6 (A, B, C, D, E, F, G, H, I, K, L, M)
Skip AC 7
Skip HI 7
Choose JL 7 (A, B, C, D, E, F, G, H, I, J, K, L, M)

Below is the minimum spanning tree that is produced by Kruskal's algorithm:



Minimum Spanning Tree Property

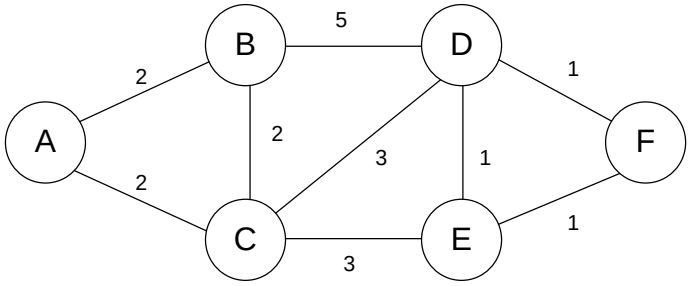
Discussion

A spanning tree of an undirected weighted graph has the *minimum spanning tree property* if any edge added to the tree is the maximum edge on the cycle it creates. Every minimum spanning tree has this property and every spanning tree with this property is a minimum spanning tree.

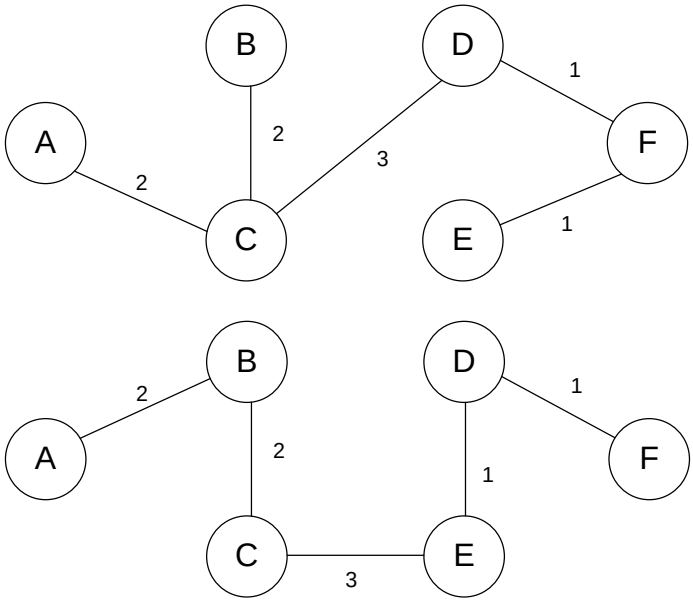
When an undirected weighted graph has two minimum spanning trees that differ by *n* edges, we can always construct a sequence of minimum spanning trees in which each adjacent pair in the sequence differs by only one edge.

Sample Problem

Given the following weighted undirected graph:



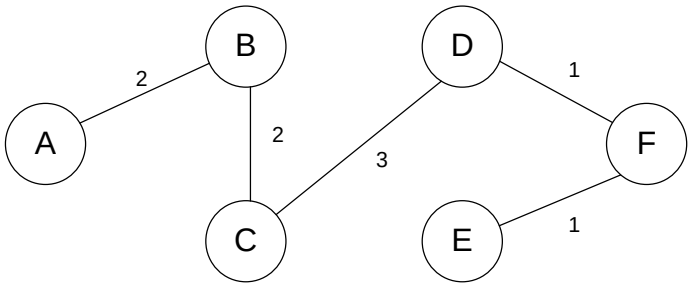
Consider the following two minimum spanning trees of that graph which differ in three edges, referring to the first tree as T_1 and the second as T_4:



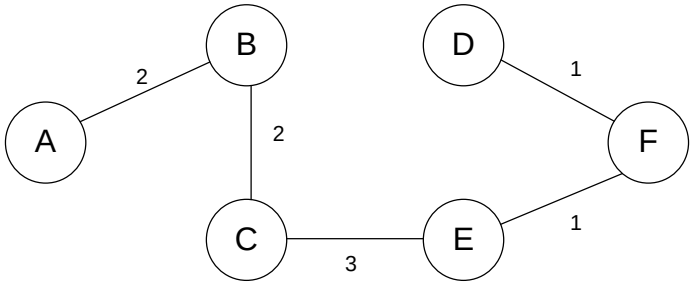
Construct two intermediate minimum spanning trees T_2, and T_3 so that the adjacent pairs in the sequence differ by only one edge.

Solution

The first intermediate tree T_2 is shown below. Edge AC has been replaced by AB:



The second intermediate tree T_3 is shown below. Edge CD has been replaced by CE.



By replacing edge EF with DE we get the tree T_4.

Dijkstra's Single Source Shortest Path Algorithm

Discussion

Like both Prim's and Kruskal's algorithm, Dijkstra's single source shortest path algorithm is a greedy algorithm. Unlike those algorithms it used on directed weighted directed graphs rather than weighted undirected graphs. Although its objective is to find shortest paths rather than a minimum spanning tree, it is quite similar to Prim's algorithm. As the algorithm executes it creates a single tree not a forest. It also uses the same data structure, which is a priority queue implemented as an array indexed by the vertex. The primary difference is that the distance that it associates with each vertex that is in the fringe is the distance from the starting vertex, not the distance from the tree.

The pseudocode for Dijkstra's algorithm is shown below where `G` is the weighted directed graph and `s` is the starting vertex.

```
dijkstras (G, s)
  make vertex s a fringe vertex with distance 0
  while fringe vertices remain
    choose the closest fringe vertex v and make it a tree vertex
    for all vertices w adjacent to v
      calculate distance to w as distance from s to v plus weight of vw
      if it is unexplored
        make it a fringe vertex with the calculated distance
      else if it is a fringe and the newly calculated distance is closer
        update the distance with the newly calculated distance
```

The process of updating the distance to a particular vertex with a shorter distance from a different predecessor is often referred to a *relaxing the edge*. If we wish to output the edges of the tree, we must also maintain the predecessor vertex with each vertex which defines the edge.

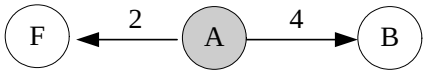
Sample Problem

Given the following adjacency lists (with edge weights in parentheses) for a directed graph:

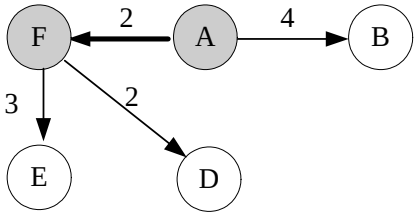
A: B(4), F(2)
B: A(1), C(3), D(4)
C: A(6), B(3), D(7)
D: A(6), E(2)
E: D(5)
F: D(2), E(3)

Execute Dijkstra's single source shortest-path algorithm by hand on this graph, showing how the data structures evolve, with *A* as the starting vertex. Clearly indicate which edges become part of the shortest path and in which order.

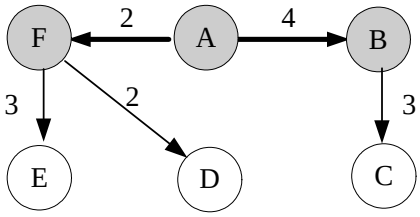
Solution



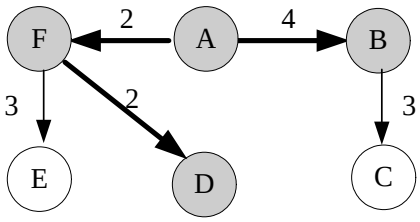
A	T	
B	F	4
C	U	
D	U	
E	U	
F	F	2 minimum



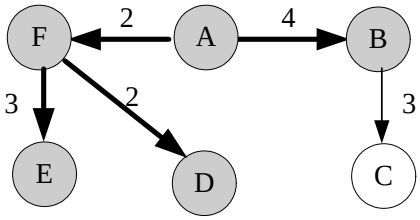
A	T	
B	F	4 minimum
C	U	
D	F	4
E	F	5
F	T	



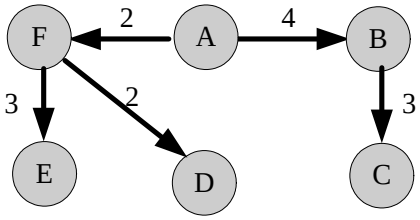
A	T	
B	T	
C	F	7
D	F	4 minimum
E	F	5
F	T	



A	T	
B	T	
C	F	7
D	T	
E	F	5 minimum
F	T	



A	T	
B	T	
C	F	7 minimum
D	T	
E	T	
F	T	



A	T	
B	T	
C	T	
D	T	
E	T	
F	T	