Minimum spanning tree problem

Consider a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$ that assigns a weight $w(e)$ to each edge e. Our task is to find the minimum spanning tree (MST) of G, the spanning tree T which minimizes the cost function

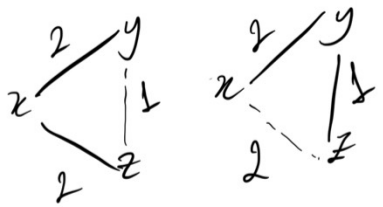
$$w(T) = \sum_{e \in T} w(e).$$

This captures a common problem in communication networks and circuit design, requiring connecting together a set of nodes by a network of minimal total length. In general, MST of a graph is not unique (it is only unique if all the weights differ, which we will not prove). In the following, we assume that the edge weights are distinct.

Kruskal's algorithm and Prim's algorithm find an MST of an undirected, connected, weighted graph. (What about a disconnected graph? Or a directed graph?) Kruskal's sorts all the edges from low weight to high, takes the edge with the lowest weight, and adds it to the existing spanning tree (forest, to be more precise), without allowing cycles. Prim's algorithm picks a starting vertex, and looks at all edges connecting to the vertex, and chooses one with the lowest weight, and adds it to the tree. It proceeds by looking at all edges connected to the tree that do not contain both vertices in the tree, and adds such lowest weighted edge to the tree.

Although Prim's algorithm is very similar to the Dijkstra's algorithm we studided last week, MST and shortest-path tree problems are different.

Question 0. Starting with vertex x, which figure would illustrate the behavior of Prim's (respectively, Djikstra's) algorithm?



Question 3: Prove that an MST contains every safe edge and no useless edges (Jeff Erickson, Algorithms)

Question 4: How to find the maximum spanning tree of an undirected, weighted graph G with n vertices, which would be the spanning tree of largest total weight?

The following pages are referenced from Michael Lecet's algorithm note page 43, 45-47, 51-54.

Definition 58. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediate spanning forest of G. Let $e = \{u, v\}$ be an edge of G such that $e \notin \mathcal{F}$.

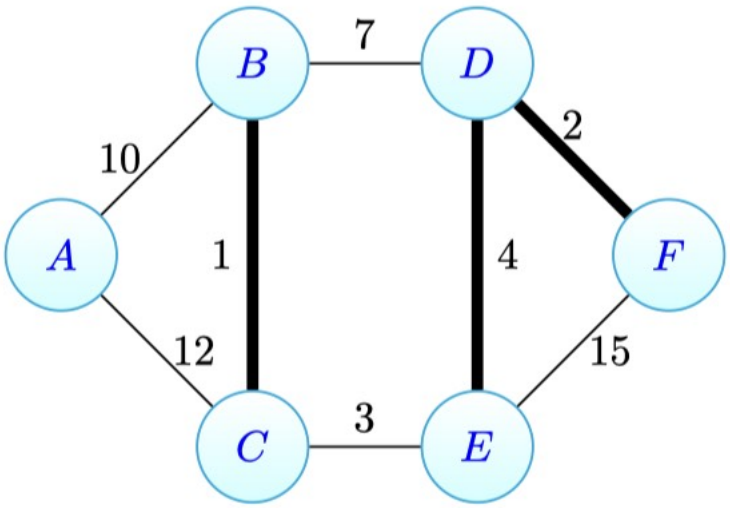
(a) We say that e is *safe* with respect to $\mathcal{F}$ if $\mathcal{F} \cup e$ is a subgraph of some minimum-weight spanning tree of G.

(b) Let $S \subseteq V(G)$ be a set of vertices such that every edge $\{x, y\}$ in $\mathcal{F}$ has either $x, y \in S$ or $x, y \in V(G) \setminus S$. We say that e is a *light* edge e is a minimum weight edge with one endpoint in S and the other endpoint in $V(G) \setminus S$.

(c) We say that e is *useless* with respect to $\mathcal{F}$ if both u and v lie on the same tree in $\mathcal{F}$. Note that in this case, adding e to $\mathcal{F}$ creates a cycle.

(d) We say that e is *undecided* with respect to $\mathcal{F}$ if e is neither safe nor useless.

Corollary 61. Let $G(V, E, w)$ be a weighted graph, and let $\mathcal{F}$ be an intermediary spanning forest. Fix a tree T_i, and tet $e \in E(G)$ be a light edge with exactly one endpoint in T_i. Then e is safe.



We have the following.

- $\{A, B\}$ is the minimum-weight edge incident to $\{A\}$. Therefore, as $\{A, B\}$ is a light edge with exactly one endpoint belonging to $\{A\}$, we have by Corollary 61 that $\{A, B\}$ is **safe** with respect to $\mathcal{F}$.
- $\{C, E\}$ is the minimum-weight edge with exactly one endpoint in the component $\{B, C\}$ (as well as the minimum-weight edge with exactly one endpoint in the component $\{D, E, F\}$). Therefore, we have by Corollary 61 that $\{C, E\}$ is **safe** with respect to $\mathcal{F}$.
- While the edge $\{A, C\}$ connects the components $\{A\}$ and $\{B, C\}$, $\{A, C\}$ is not a minimum-weight edge doing so. Therefore, $\{A, C\}$ is **undecided** with respect to $\mathcal{F}$.
- While the edge $\{B, D\}$ connects the components $\{B, C\}$ and $\{D, E, F\}$, $\{B, D\}$ is not a minimum-weight edge doing so. Therefore, $\{B, D\}$ is **undecided** with respect to $\mathcal{F}$.
- The edge $\{E, F\}$ creates has both endpoints in the component $\{D, E, F\}$. So $\{E, F\}$ is **useless** with respect to $\mathcal{F}$.

4.3 Kruskal's Algorithm

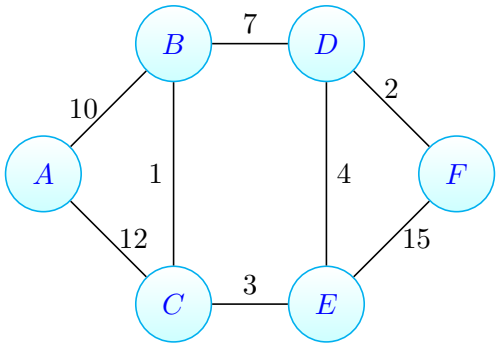
We briefly introduced Kruskal's algorithm in Section 4.1. In this section, we will examine Kruskal's algorithm in more detail. Recall that Kruskal's algorithm places the edges of the input graph into a priority queue. It then polls the edges one at a time, adding the edge e currently being considered to the intermediate spanning forest precisely if e connects two disjoint components. As the edges are sorted from lowest weight to highest weight, it follows that e is added precisely if there exists a component T where e is a light edge with exactly one endpoint in T. So by Corollary 61, e is added precisely if e is safe.

We formalize Kruskal's algorithm below.

Algorithm 6 Kruskal's Algorithm

```
1: procedure KRUSKAL(ConnectedWeightedGraph G(V, E, w))
2:     F ← (V(G), ∅)                    ▷ Initialize the Intermediate Spanning Forest to contain no edges.
3:     PriorityQueue Q ← []
4:     Q.addAll(E(G))
5:     while F.numEdges() < |V(G)| − 1 do
6:         {u, v} ← Q.poll()                    ▷ Poll an edge and call the endpoints u and v
7:         if u and v are on different components of F then
8:             F.addEdge({u, v})
       return F
```

Example 64. We now work through Kruskal's algorithm on the following graph.

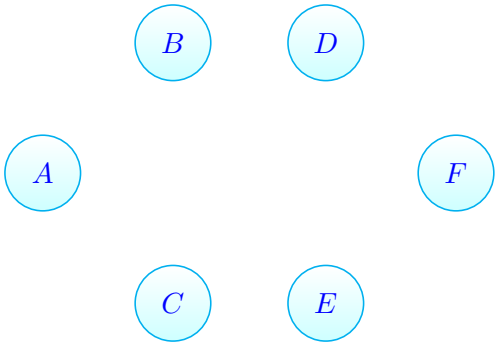


We proceed as follows.

1. We initialize the intermediate spanning forest $\mathcal{F}$ to be the empty graph (the graph on no edges). We also place the edges of G into a priority queue, which we call Q. So:

   $$Q = [(\{B, C\}, 1), (\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)].$$

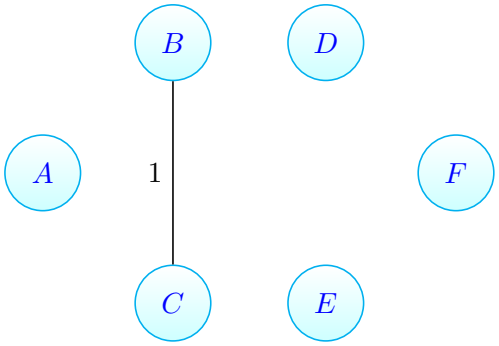
   Here, $(\{B, C\}, 1)$ indicates the edge $\{B, C\}$ has weight 1. The intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll from Q, which returns the edge $\{B, C\}$. Note that $w(\{B, C\}) = 1$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{B, C\}$ to $\mathcal{F}$. So:

   $$Q = [(\{D, F\}, 2), (\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

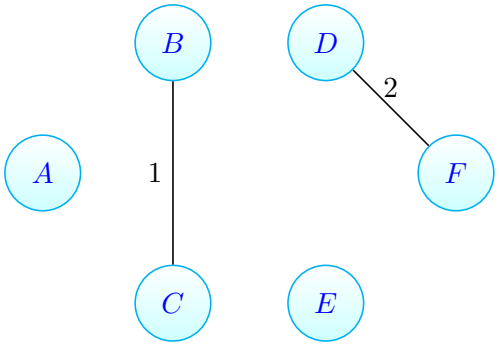
   and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



3. We poll from Q, which returns the edge $\{D, F\}$. Note that $w(\{D, F\}) = 2$. As B and C are on different components of $\mathcal{F}$, we add the edge $\{D, F\}$ to $\mathcal{F}$. So:

$$Q = [(\{C, E\}, 3), (\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

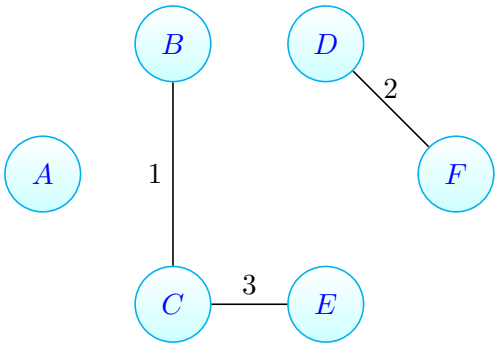
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll from Q, which returns the edge $\{C, E\}$. Note that $w(\{C, E\}) = 3$. As C and E are on different components of $\mathcal{F}$, we add the edge $\{C, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{D, E\}, 4), (\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

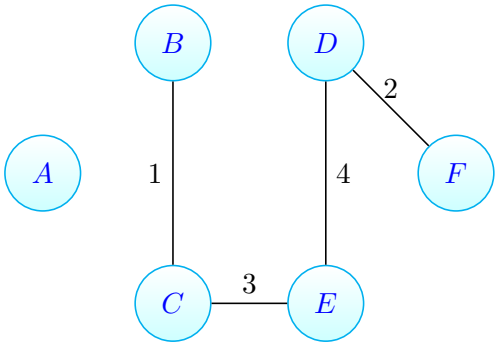
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll from Q, which returns the edge $\{D, E\}$. Note that $w(\{D, E\}) = 4$. As D and E are on different components of $\mathcal{F}$, we add the edge $\{D, E\}$ to $\mathcal{F}$. So:

$$Q = [(\{B, D\}, 7), (\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll from Q, which returns the edge $\{B, D\}$. Note that $w(\{B, D\}) = 7$. As B and D are on the same component of $\mathcal{F}$, we do **not** add the edge $\{B, D\}$ to $\mathcal{F}$. So:

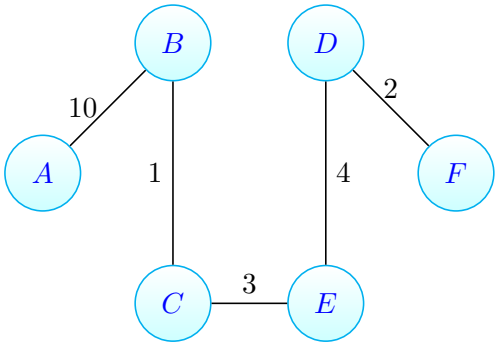
$$Q = [(\{B, A\}, 10), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the intermediate spanning forest $\mathcal{F}$ remains unchanged from the previous iteration.

7. We poll from Q, which returns the edge $\{B, A\}$. Note that $w(\{B, A\}) = 10$. As B and A are on different components of $\mathcal{F}$, we add the edge $\{B, A\}$ to $\mathcal{F}$. So:

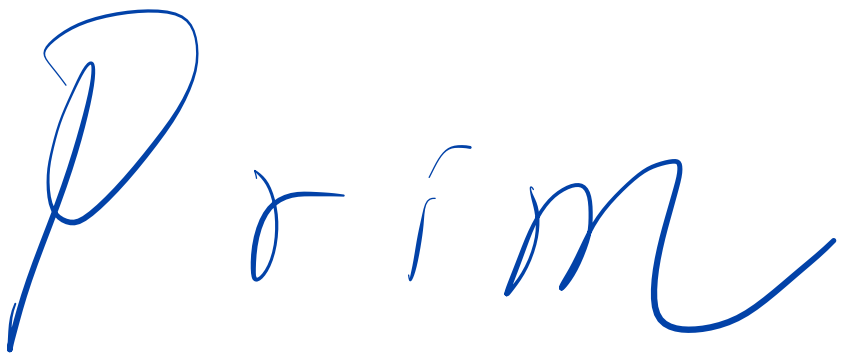
$$Q = [(\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



As there are 6 vertices and $\mathcal{F}$ has 5 edges, Kruskal's algorithm terminates and returns $\mathcal{F}$, which is our minimum-weight spanning tree.

Remark 65. Now that we have worked through an example of Kruskal's algorithm, we wish to comment a bit about the algorithm provided (Algorithm 6). On line 7 of Algorithm 6, there is an `if` statement checking whether two vertices belong to the same connected component. For the purposes of this class, we will not examine the details associated with implementing this functionality. In practice, a Union-Find data structure is used to manage the intermediate spanning forest. We direct the reader to [CLRS09, Chapter 21] for details regarding the Union-Find data structure.



4.4 Prim's Algorithm

In this section, we examine a second technique to construct minimum-weight spanning trees; namely, Prim's algorithm. We again start with the intermediate spanning forest $\mathcal{F}$ that contains all the vertices of our input graph $G(V, E, w)$, but none of the edges. While Kruskal's algorithm determines which edges to add to $\mathcal{F}$ by examining the entire graph, Prim's algorithm takes a more local perspective. We provide as input a specified source vertex $s \in V(G)$. Let T^* be the component of $\mathcal{F}$ that contains s. Prim's algorithm examines the edges of G that have exactly one endpoint in T^* and select a light edge e from these to add to $\mathcal{F}$. As e has exactly one endpoint in T^*, e connects two distinct components of $\mathcal{F}$. So by Corollary 61, e is a safe edge with respect to $\mathcal{F}$. This is the key observation in establishing that Prim's algorithm returns a minimum-weight spanning tree.

We now turn to formalizing Prim's algorithm.

Algorithm 7 Prim's Algorithm

```
1:  procedure PRIM(ConnectedWeightedGraph G(V, E, w), Vertex source)
2:      F ← (V(G), ∅)                    ▷ Initialize the Intermediate Spanning Forest to contain no edges.
3:      PriorityQueue Q ← []

4:      for each edge e ∈ E(G) do
5:          e.processed ← false

6:      for each edge e incident to source do
7:          Q.add(e)
8:          e.processed ← true

9:      while F.numEdges() < |V(G)| − 1 do
10:         {u, v} ← Q.poll()                ▷ Poll an edge and call the endpoints u and v
11:         T_u ← F.componentContaining(u)
12:         T_v ← F.componentContaining(v)

13:         if T_u ≠ T_v then                ▷ Check that u and v belong to different components
14:             F.addEdge({u, v})

15:             if source ∈ T_u then         ▷ If v was added to the component containing source
16:                 for each unprocessed edge e incident to v do
17:                     Q.add(e)             ▷ Then add to Q each unprocessed edge incident to v

18:             else                         ▷ If u was added to the component containing source
19:                 for each unprocessed edge e incident to u do
20:                     Q.add(e)             ▷ Then add to Q each unprocessed edge incident to u

        return F
```

We associate to each edge an attribute `processed` to indicate whether that edge has been placed into the priority queue. This ensures that each edge is considered at most once, which helps ensure that the algorithm will terminate.

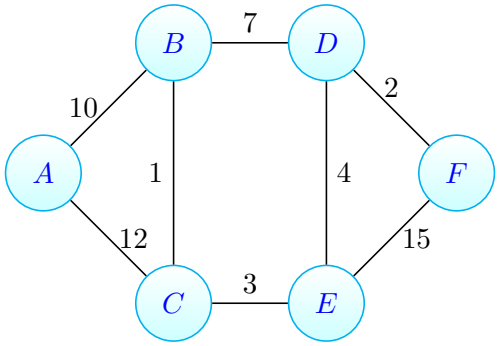
Now at lines 6-8, we initialize the priority queue to contain only edges that are incident to the source vertex. This ensures that the first edge placed into the intermediate spanning forest is incident to the source vertex. Now by adding an edge to $\mathcal{F}$, we introduce a new vertex v to the component containing our source vertex. Prim's algorithm then adds to the prioritiy queue the edges incident to v, provided such edges have not already been polled from the queue. So the `while` loop at line 9 preserves the invariant that every edge in the priority queue has at least one endpoint in the component containing our source vertex.

Prim's algorithm only adds an edge if it connects two components. Such an edge e is polled from the priority queue, and so (i) has an endpoint in the component containing the source vertex, and (ii) is a minimum-weight edge connecting two distinct components. Therefore, e is a safe edge.

4.4.1 Prim's Algorithm: Example 1

We now work through an example of Prim's algorithm.

Example 71. Consider the following graph $G(V, E, w)$ pictured below. Suppose we select the source vertex A.

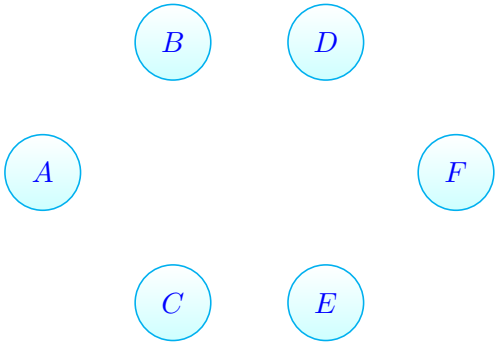


Prim's algorithm proceeds as follows.

1. We initialize the intermediate spanning forest to contain all the vertices of G, but no edges. We then initialize the priority queue to contain the edges incident to our source vertex A. So:

$$Q = [(\{A, B\}, 10), (\{A, C\}, 12)],$$

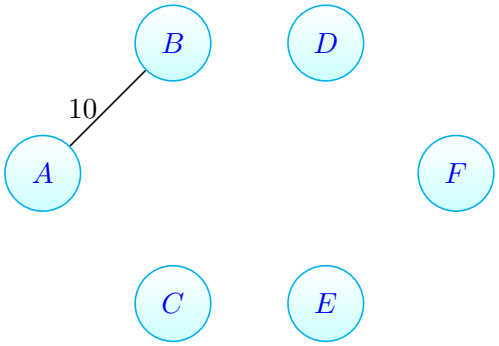
and our intermediate spanning forest $\mathcal{F}$ is pictured below.



2. We poll the edge $\{A, B\}$ from the queue and mark $\{A, B\}$ as processed. Note that $w(\{A, B\}) = 10$. As $\{A, B\}$ has exactly one endpoint on the component containing A (which is the isolated vertex A), we add $\{A, B\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to B. So:

$$Q = [(\{B, C\}, 1), (\{B, D\}, 7), (\{A, C\}, 12)],$$

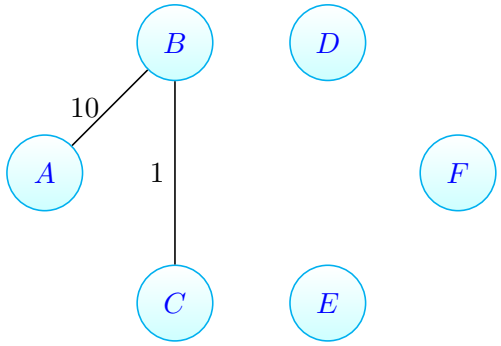
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



3. We poll the edge $\{B, C\}$ from the queue and mark $\{B, C\}$ as processed. Note that $w(\{B, C\}) = 1$. As $\{B, C\}$ has exactly one endpoint on the component containing A (which is the isolated vertex $\{A, B\}$), we add $\{B, C\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to C (provided said edges are not already in the priority queue). So:

$$Q = [(\{C, E\}, 3), (\{B, D\}, 7), (\{A, C\}, 12)],$$

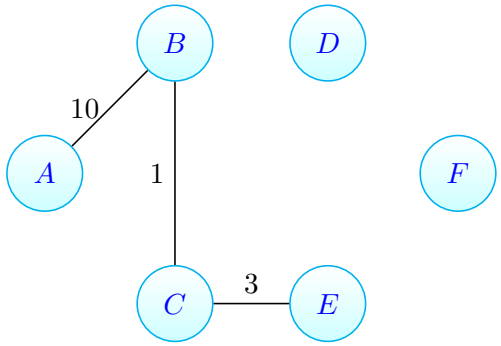
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



4. We poll the edge $\{C, E\}$ from the queue and mark $\{C, E\}$ as processed. Note that $w(\{C, E\}) = 3$. As $\{C, E\}$ has exactly one endpoint on the component containing A, we add $\{C, E\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to E (provided said edges are not already in the priority queue). So:

$$Q = [(\{E, D\}, 4), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

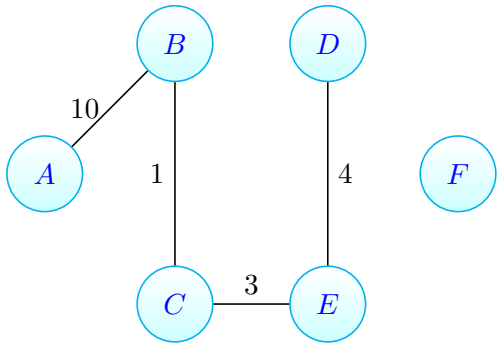
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



5. We poll the edge $\{E, D\}$ from the queue and mark $\{E, D\}$ as processed. Note that $w(\{E, D\}) = 4$. As $\{E, D\}$ has exactly one endpoint on the component containing A, we add $\{E, D\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{D, F\}, 2), (\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

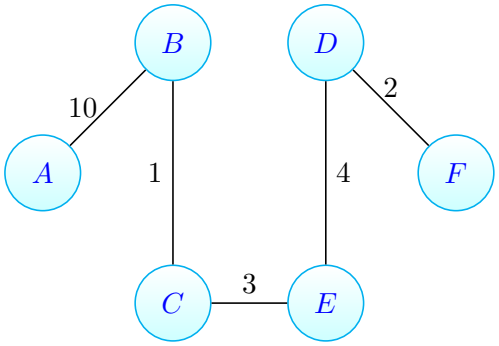
and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



6. We poll the edge $\{D, F\}$ from the queue and mark $\{D, F\}$ as processed. Note that $w(\{D, F\}) = 2$. As $\{D, F\}$ has exactly one endpoint on the component containing A, we add $\{D, F\}$ to $\mathcal{F}$. We then push into the priority queue the unprocessed edges incident to D (provided said edges are not already in the priority queue). So:

$$Q = [(\{B, D\}, 7), (\{A, C\}, 12), (\{E, F\}, 15)],$$

and the updated intermediate spanning forest $\mathcal{F}$ is pictured below.



7. As $\mathcal{F}$ has $|V(G)| - 1 = 6 - 1 = 5$ edges, the algorithm terminates and returns $\mathcal{F}$, pictured in Step 6 immediately above.

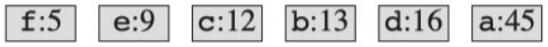
Remark 72. We note that the minimum-weight spanning tree constructed by Kruskal's algorithm in Example 64 is the same tree that Prim's algorithm constructed in Example 71. For this input graph, the edge weights were distinct. Therefore, the graph had only one minimum-weight spanning tree. In general, Prim's algorithm and Kruskal's do not construct the same minimum-weight spanning tree.

The coming peges are refered from Introduction to Algorithm in page 431, 432.

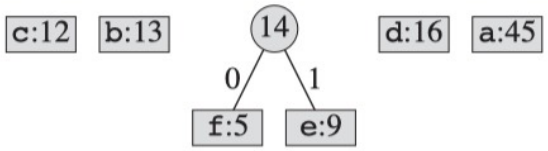
Huffman codes

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

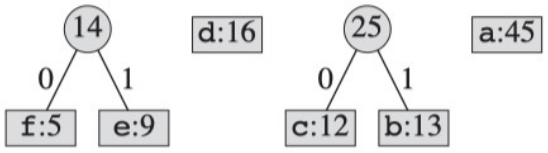
(a)



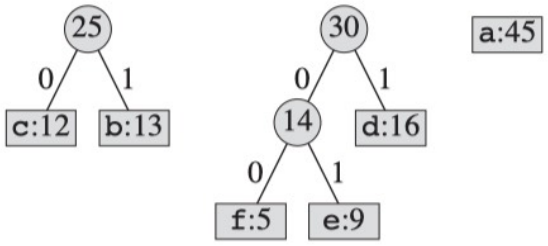
(b)



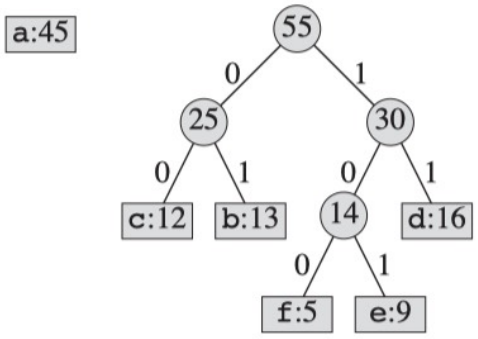
(c)



(d)



(e)



(f)