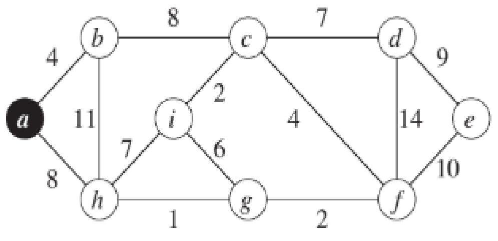
# Minimum Spanning Tree

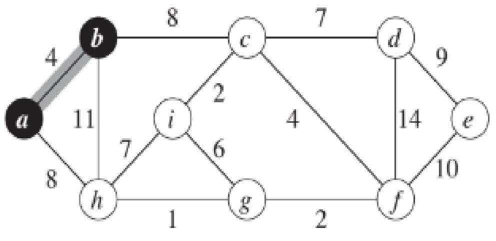
Minimum Spanning tree: A tree that covers all the vertices of the graph in minimum cost. Remember the tree properties (no cycle and each vertex must have only single parent node). Don't confuse MST with SSSP (single source shortest path). In SSSP the main objective is to find the shortest path from the source vertex to all other vertices whereas in MST the objective is to cover the entire network in minimum cost. While an MST may offer the shortest path from the source to all other vertices, it cannot guarantee providing the shortest path from the source to all other vertices. However it is also possible that in some cases both MST (minimum spanning tree) and SPT (Shortest path tree) are same.

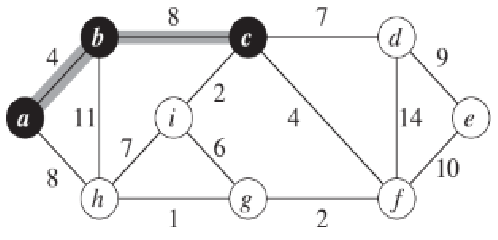
```
MST Prim's (G, src)
{
   A = {} //MST solution set (optional)
   for each v belongs to G.V   ➔ (V)
      v. cost = ∞
      v. PI = NULL
   src. key = 0
   create a Min-heap "Q"
   for each v belongs to G.V
   {
      Q. insert(v)  ➔ (V* log V)
   }
   while (Q != {})
   {
      v = Q. ExtMin()  ➔ (V*logV)
      A. insert(v) // insert the vertex in MST solution set (optional)
      for each u belongs to G. Adj[v]
      {
         if (u belongs to Q and u. cost > w(u,v))
         {
            u. cost = w(u,v)
            u. PI = v
         // Since we have a heap in which the key of one of the vertices has been updated. So, we need
         // to modify the heap to ensure that we can obtain the correct minimum value by using ExtMin
         // function in the subsequent iteration.
            Q. decrease Key(u)    ➔ (E*logV)
        //decrease key is equivalent to heapify function call to update the heap contents. It is very
        // Important to update heap contents otherwise we may get inaccurate value in the next getMin
        // function call
         }
      }
   }
   Return A
}
```

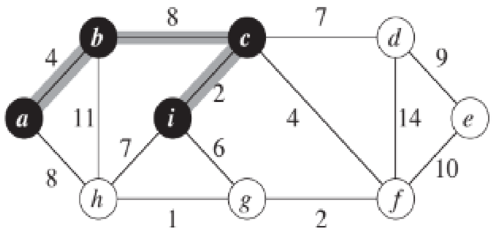
**Time complexity:** T(V,E) = V* log V + E* log V
Since E ~ V^2 (In worst case scenario) so **T(V,E) = O(V^2* log V)**

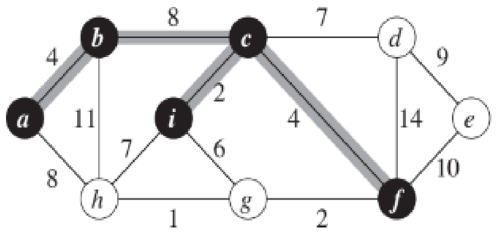
**Prim's algorithm uses heap data structure, and it is preferred for dense graph. It expands the MST by including the vertices in the solution set i.e., vertex of lightest safe edge will be included in MST solution set.** Sample Example of Prim's Algorithm with vertex **"a"** as source**.** Highlighted grey edges in the last figure i.e., (i) are basically representing the MST.

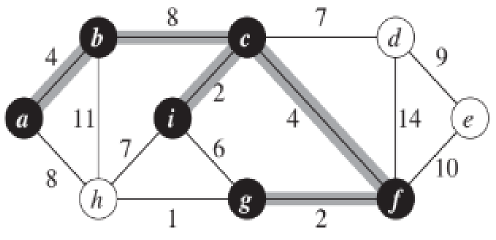
(a)


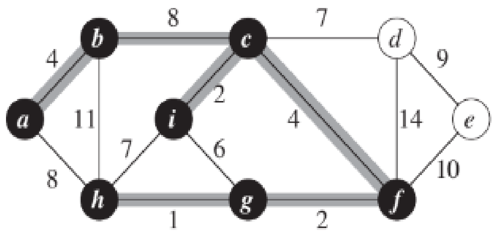
(b)


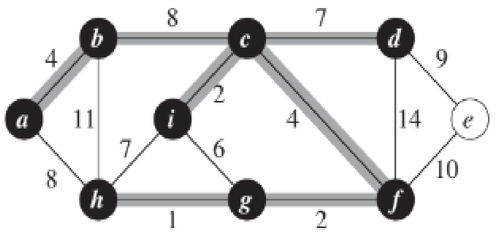
(c)


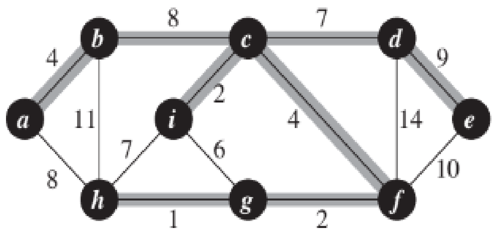
(d)


(e)


(f)


(g)


(h)


(i)


# Kruskal Method

Kruskal method is preferred in sparse graph. It expands MST by including the edges in the solution set whereas Prim's expands the MST by including the vertices in the solution set. Prim's use heap data structure whereas Kruskal use disjoint set data structure.

**Main idea of Kruskal:** Create singleton set of each vertex in graph. A unique id will be assigned to each set. Now create a single list and store all the edges of graph in this list. Sort the list. Now iterate for each edge of the graph and select the edges from sorted list. Check whether the vertices of the selected edge are disjoint sets or not. Use "**find**" function for this purpose. "**find**" will return the set-id. If the set-id's are different then they are disjoint set. Incase of disjoint sets then insert the edge in the MST solution set and merge the sets into the same so that we can avoid the cycle in MST.

**Kruskal (G, w)**

```
{
  A = {} // MST solution set

  //Create a singleton set of each vertex. Each set will be given a unique id.
  for each vertex "V" belongs to G.V
  {
    Make-set(V)
  }

  Create a single list and store all the G.E (edges of graph) in this list.
  sort the list of edges in the increasing order of weights "w" of edges.

  for each edge (u,v) taken from the sorted list of edges in order
  {
//find function will return the set-id. if the id's of "u" and "v" are different then they are disjoint sets.
    if(find(u) != find(v))
    {
// if "u" and "v" are disjoint then insert this edge into the solution set and merge these sets into the
same set.
       A. insert ((u,v)) //insert the edge(u,v) in the MST solution set
       Union (u,v) // take union to merge the disjoint sets into the same set (to avoid cycle)
    }
  }
  return A
}
```

**Time complexity:** Sorting requires $N * logN$ time. We are sorting edges so it will become $E * \ logE$
T(V,E) = $E * \ log \ E$
Since E ~ V^2 (In worst case scenario) so **T(V,E) =** $O(V^2 * \log V^2)$
Apply the Property of log i.e., $\log V^2 = 2 * logV$
So overall time complexity = $O(V^2 * logV)$

Store all the edges in single list and then sort the list in the increasing order of weights. Select the edges from sorted list, make sure it will not create any cycle and then include the edge in the MST solution set.

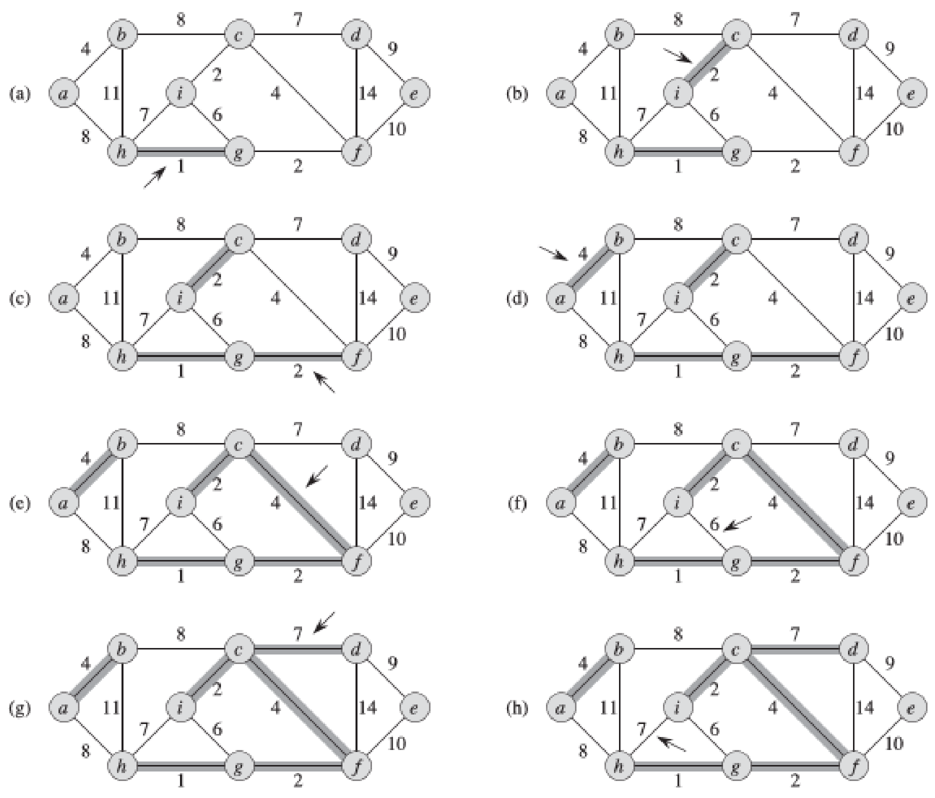


**Figure 23.4** The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest $A$ being grown. The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

checks, for each edge $(u, v)$, whether the endpoints $u$ and $v$ belong to the same tree. If they do, then the edge $(u, v)$ cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, line 7 adds the edge $(u, v)$ to $A$, and line 8 merges the vertices in the two trees.