CSE 373: Data Structures and Algorithms

Lecture 24: Graphs VI

Minimum Spanning Tree Problem

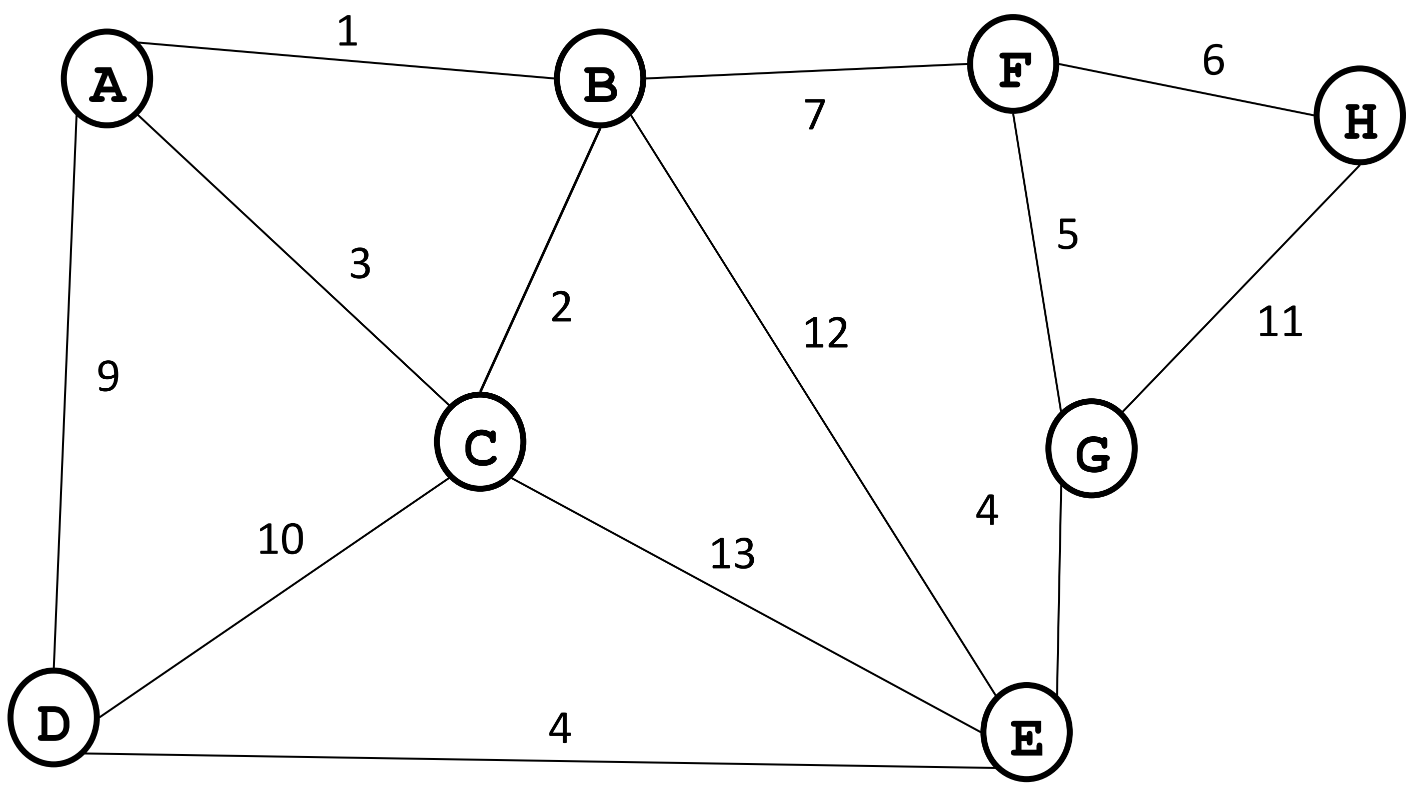
- Input: Undirected Graph $G = (V, E)$ and a cost function C from E to non-negative real numbers. $C(e)$ is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

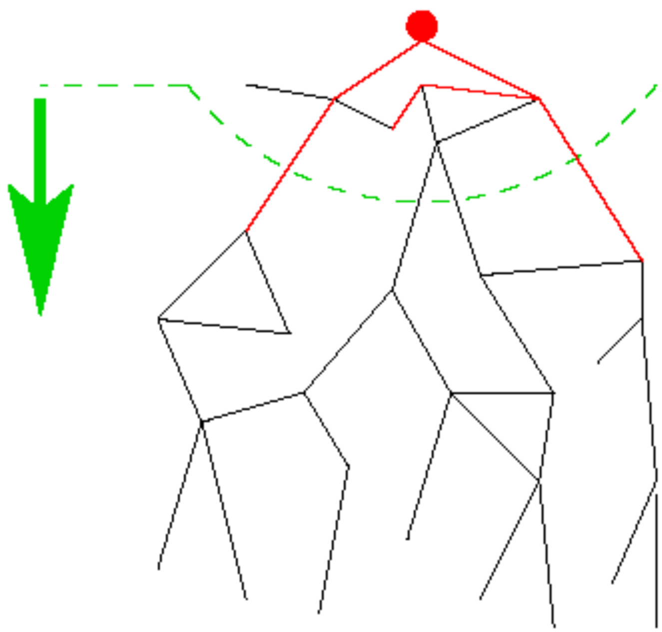
Observations about Spanning Trees

- For any spanning tree T, inserting an edge e_{new} not in T creates a cycle
- But
  - Removing any edge e_{old} from the cycle gives back a spanning tree
  - If e_{new} has a lower cost than e_{old} we have progressed!

Find the MST

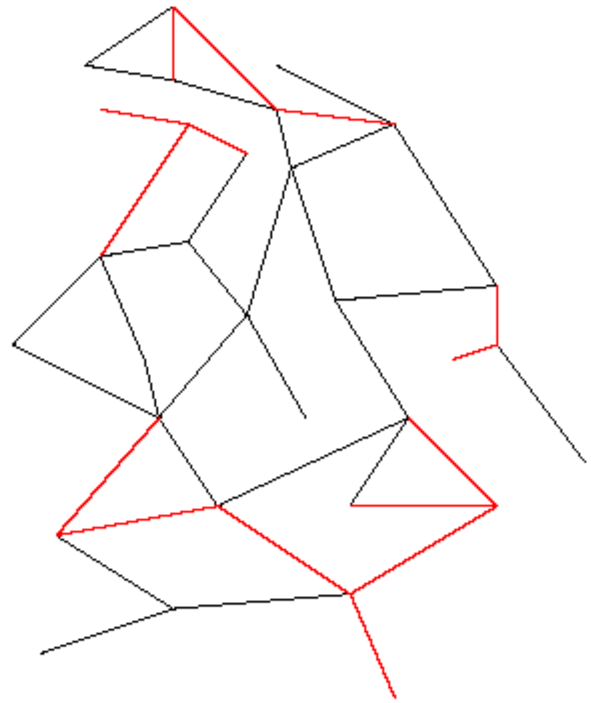


Two Different Approaches



Prim's Algorithm

Looks familiar!

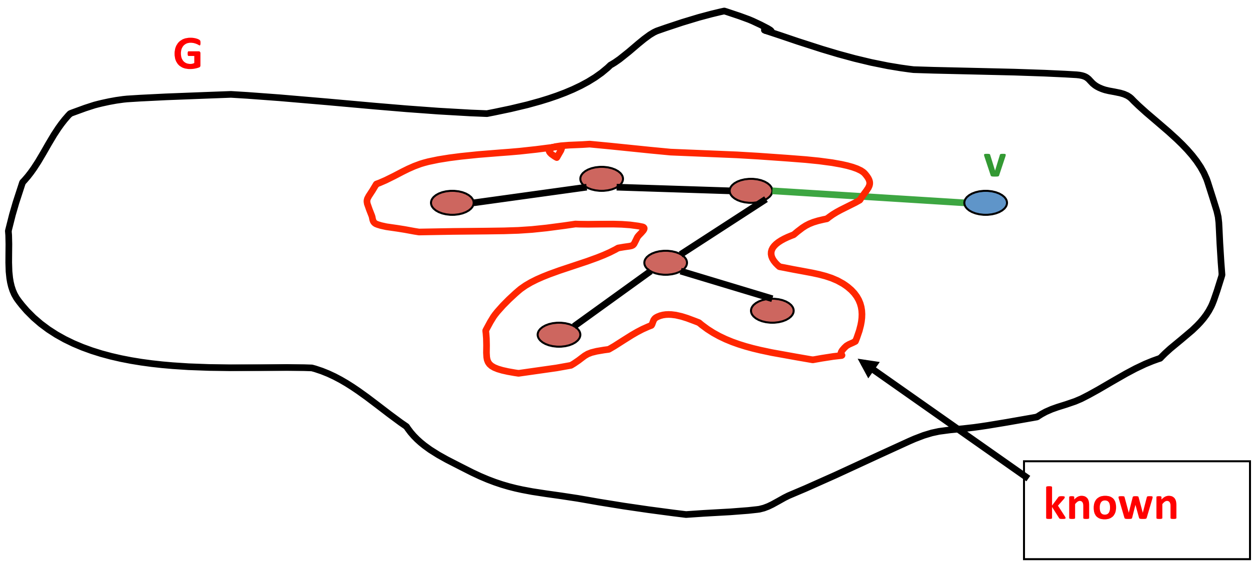


Kruskals's Algorithm

Completely different!

Prim's algorithm

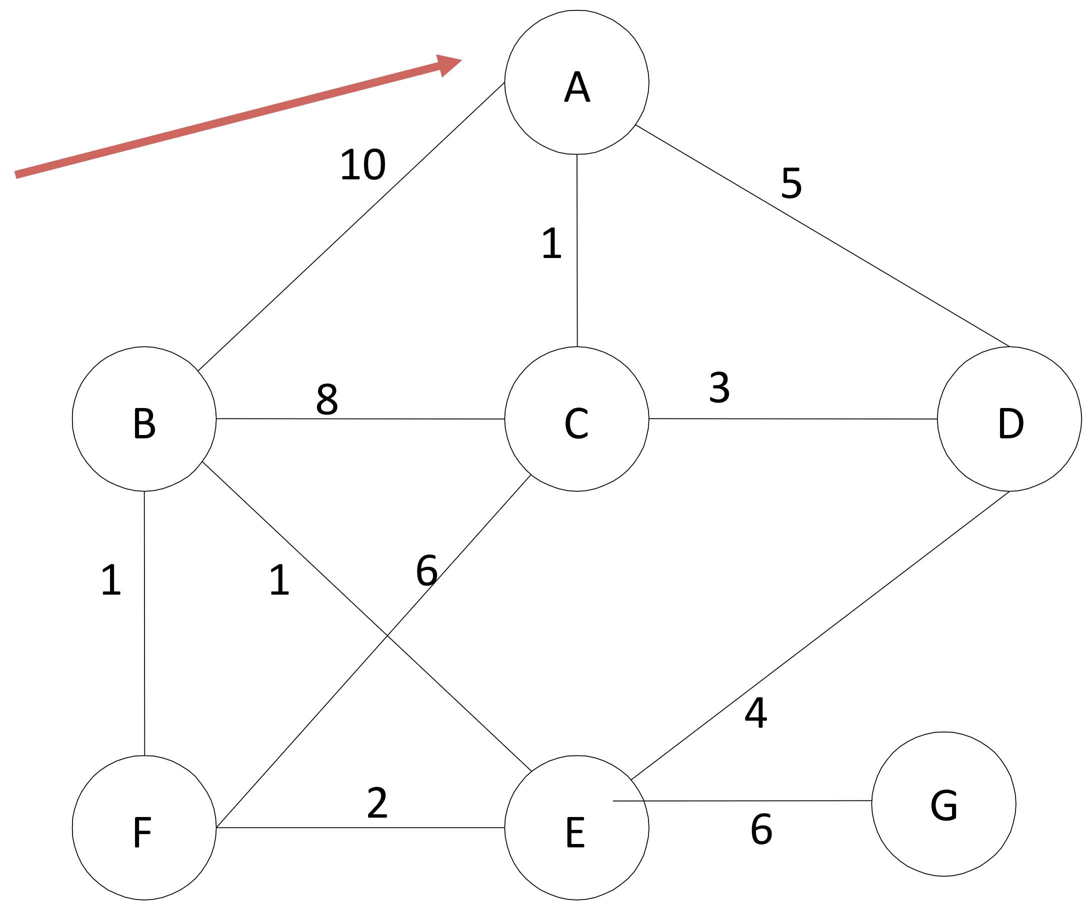
Idea: Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices. Pick the edge with the smallest weight.



Prim’s algorithm

Starting from empty *T*, choose a vertex at random and initialize

V = {A}, *T* ={}

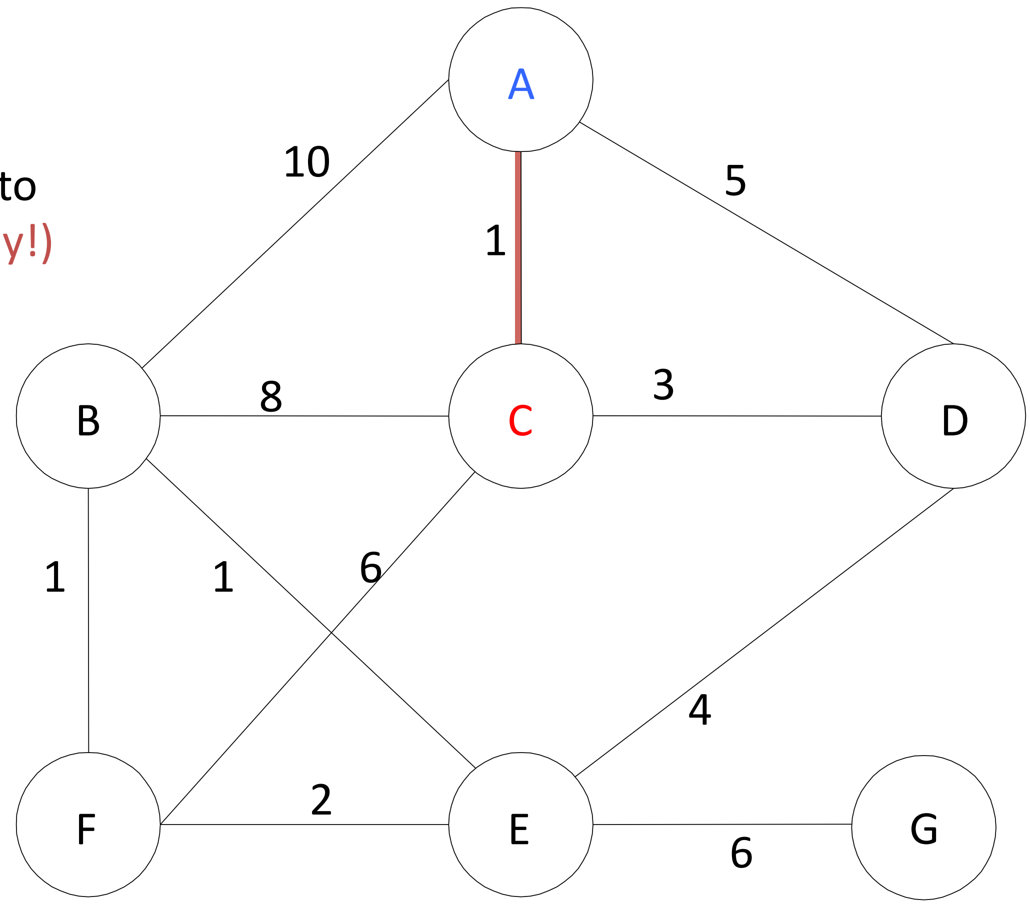


Prim's algorithm

Choose the vertex u not in *V* such that edge weight from u to a vertex in *V* is minimal (greedy!)

V = {A,C}

T = { (A,C) }

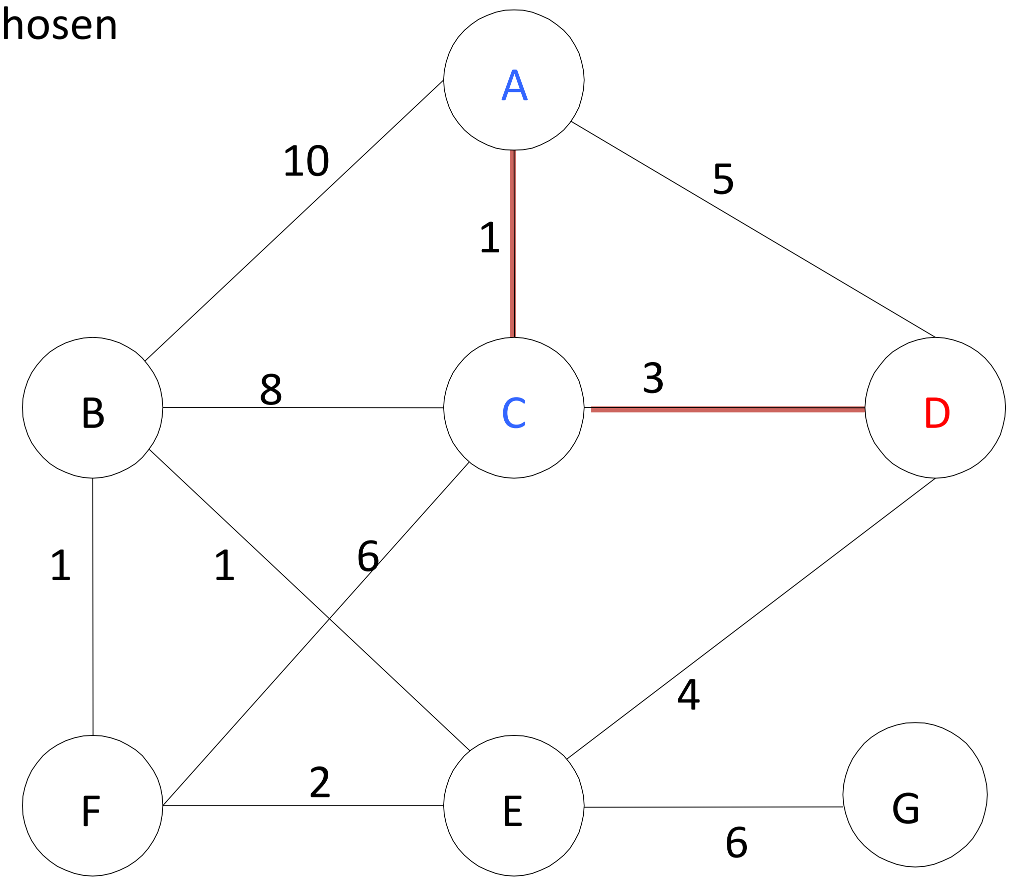


Prim’s algorithm

Repeat until all vertices have been chosen

V = {A,C,D}

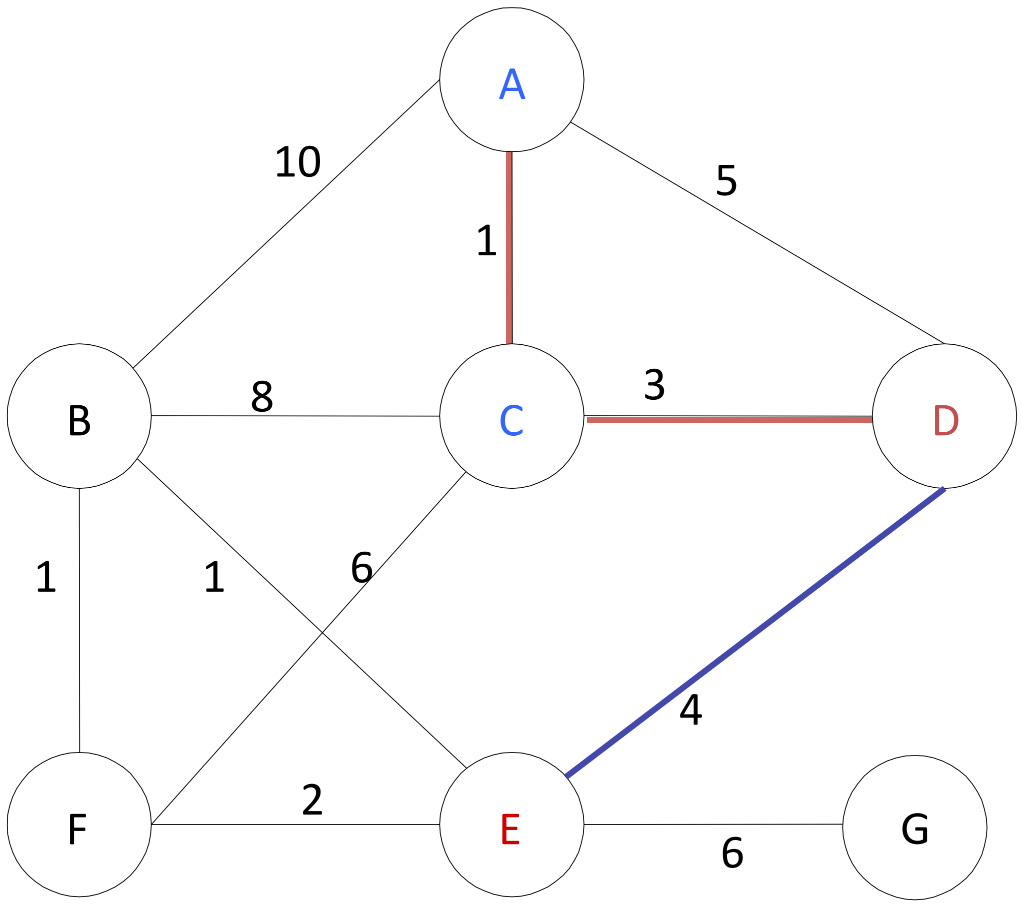
T= { (A,C), (C,D)}



Prim's algorithm

V = {A,C,D,E}

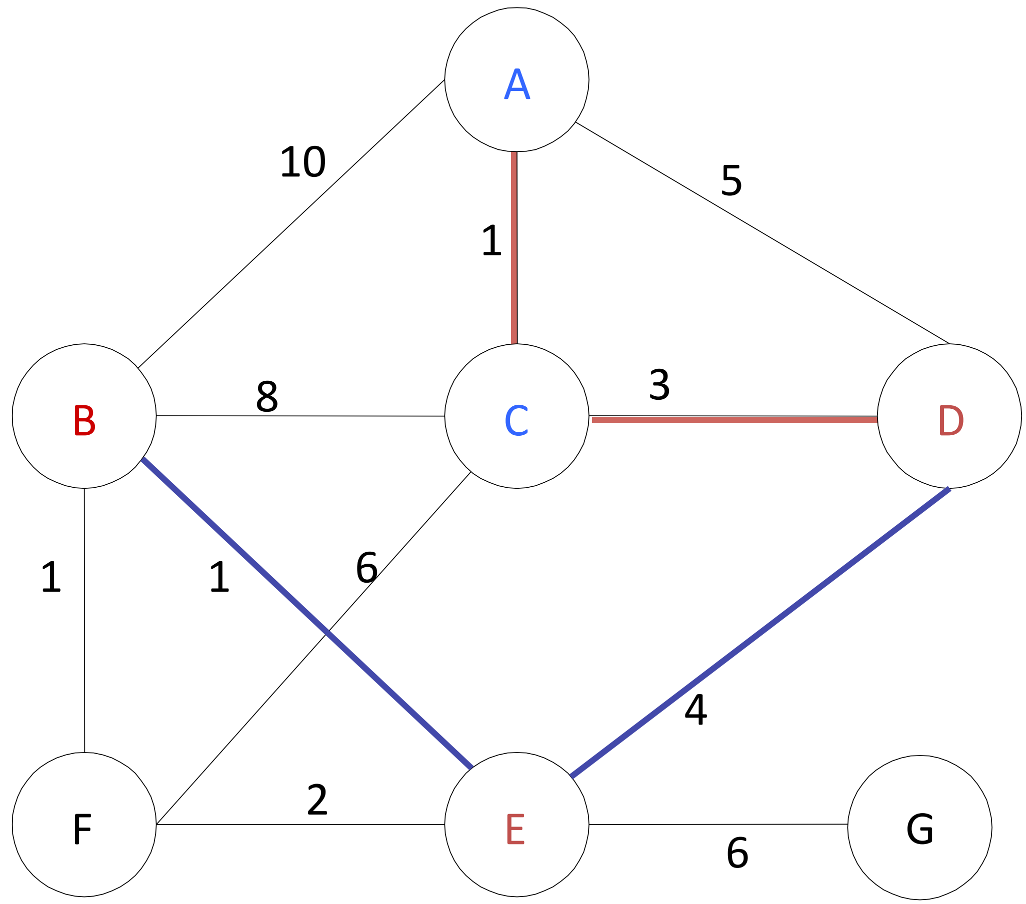
T = { (A,C), (C,D), (D,E) }



Prim's algorithm

V = {A,C,D,E,B}

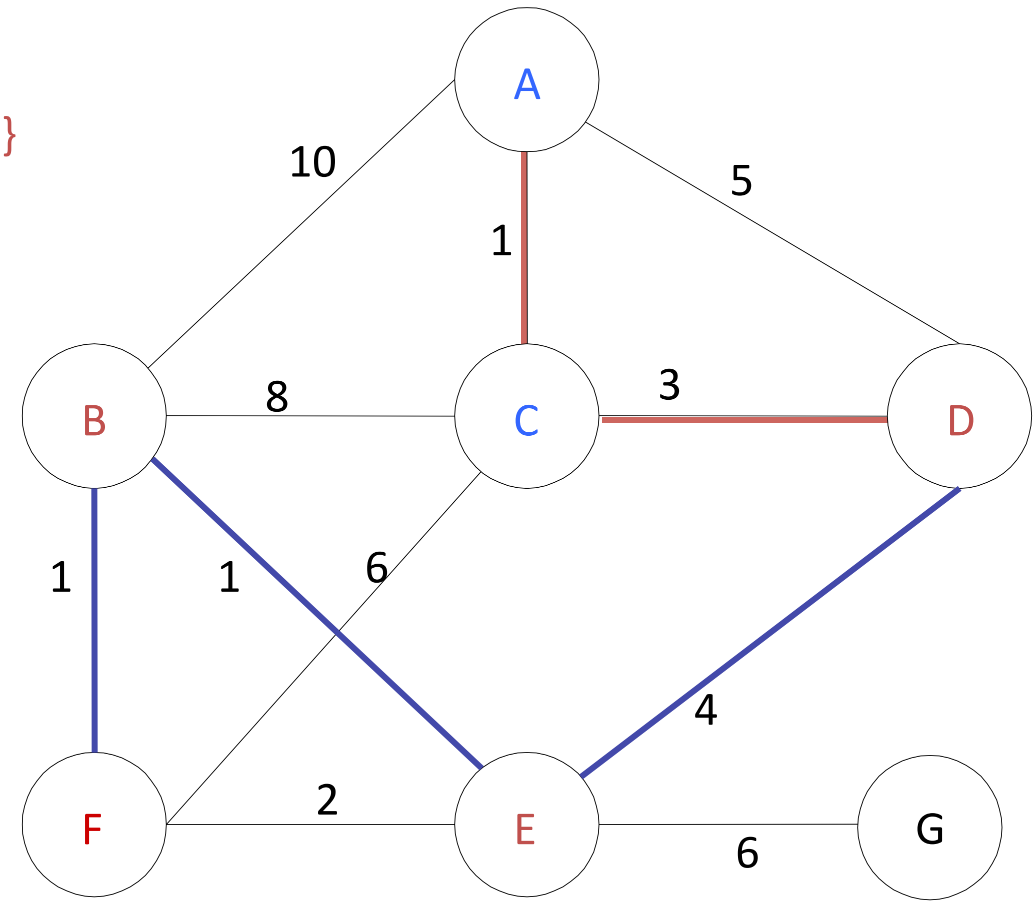
T = { (A,C), (C,D), (D,E), (E,B) }



Prim’s algorithm

V = {A,C,D,E,B,F}

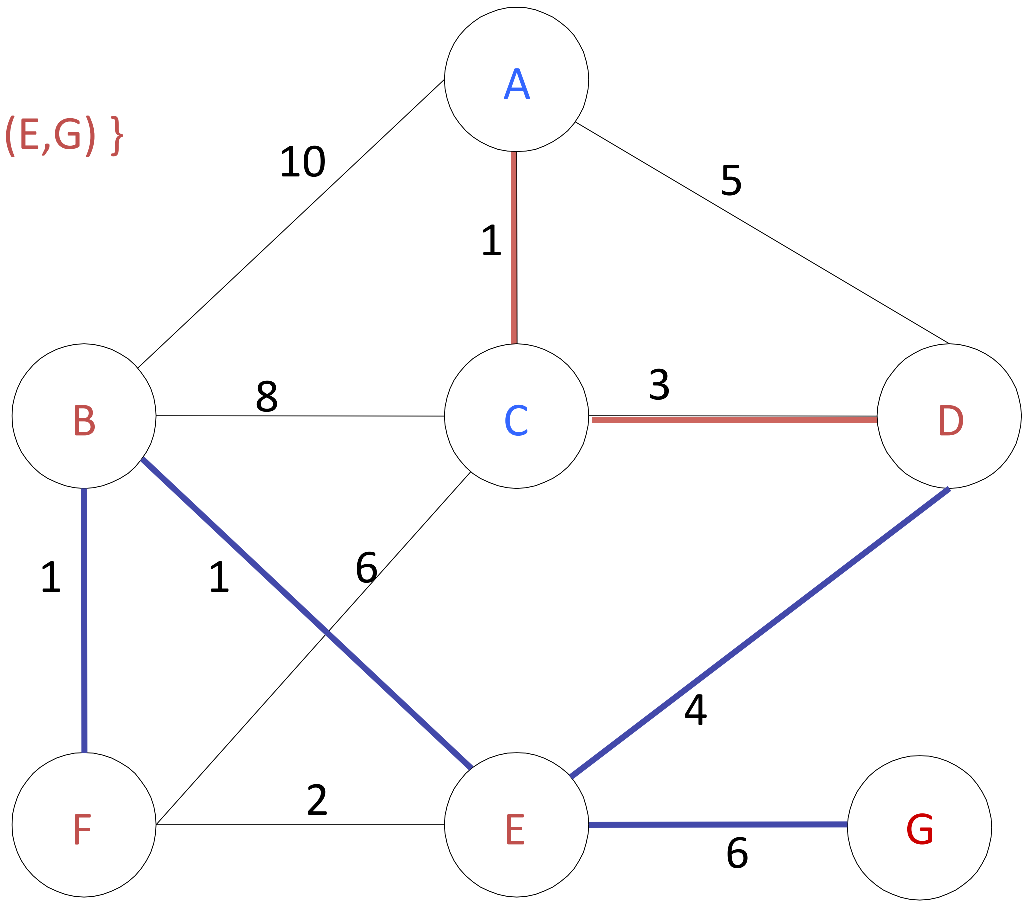
T = { (A,C), (C,D), (D,E), (E,B), (B,F) }



Prim's algorithm

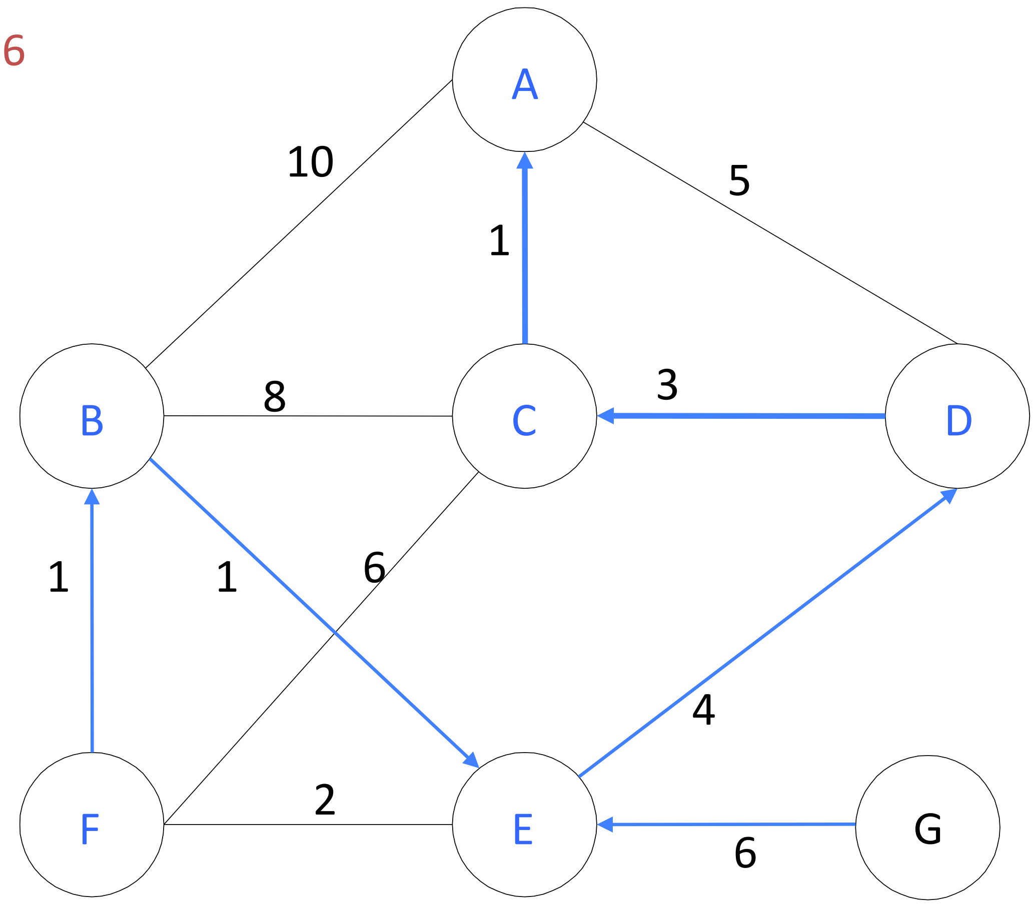
V = {A,C,D,E,B,F,G}

T = { (A,C), (C,D), (D,E), (E,B), (B,F), (E,G) }



Prim’s algorithm

Final Cost: 1 + 3 + 4 + 1 + 1 + 6 = 16



Prim's Algorithm Implementation

```
Prim():
   for each vertex v:                        // Initialization
       v's distance := infinity.
       v's previous := none.
       mark v as unknown.
   choose random node v1.
   v1's distance := 0.
   List := {all vertices}.
   T := {}.

   while List is not empty:
       v := remove List vertex with minimum distance.
       add edge {v, v's previous} to T.
       mark v as known.
       for each unknown neighbor n of v:
           if distance(v, n) is smaller than n's distance:
               n's distance := distance(v, n).
               n's previous := v.

   return T.
```

Prim’s algorithm Analysis

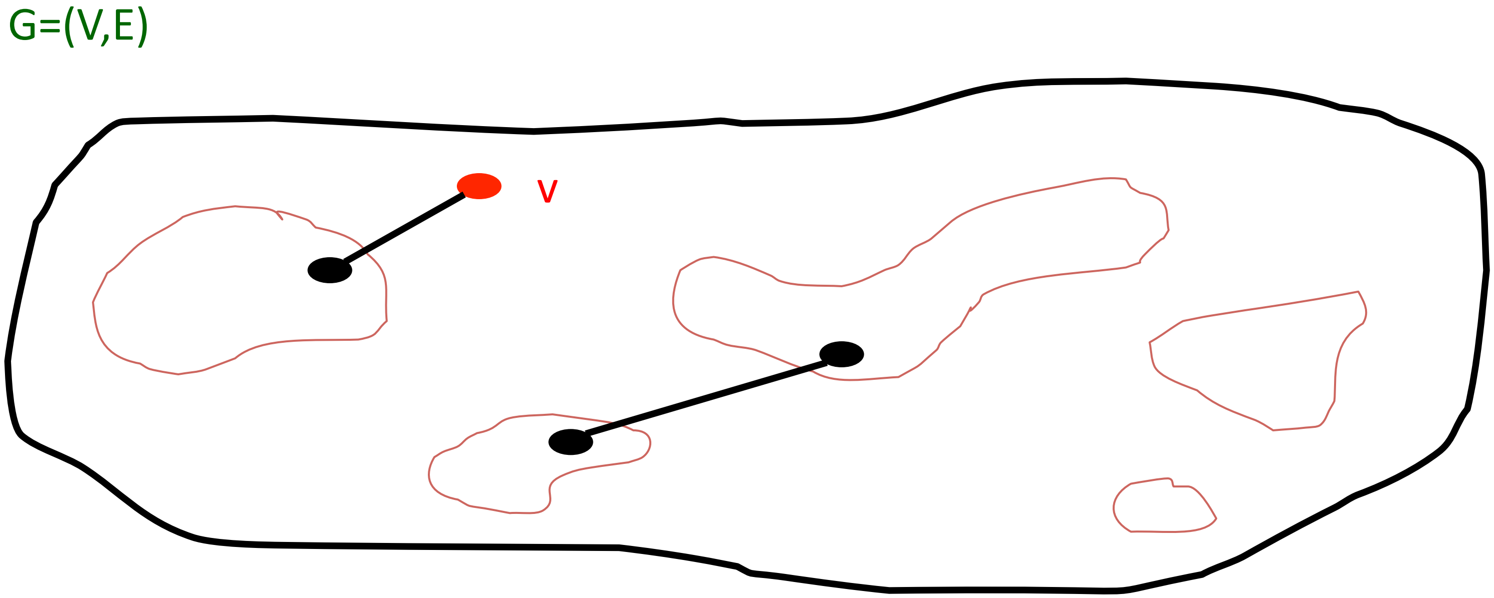
- How is it different from Djikstra's algorithm?

- If the step that removes unknown vertex with minimum distance is done with binary heap the running time is:

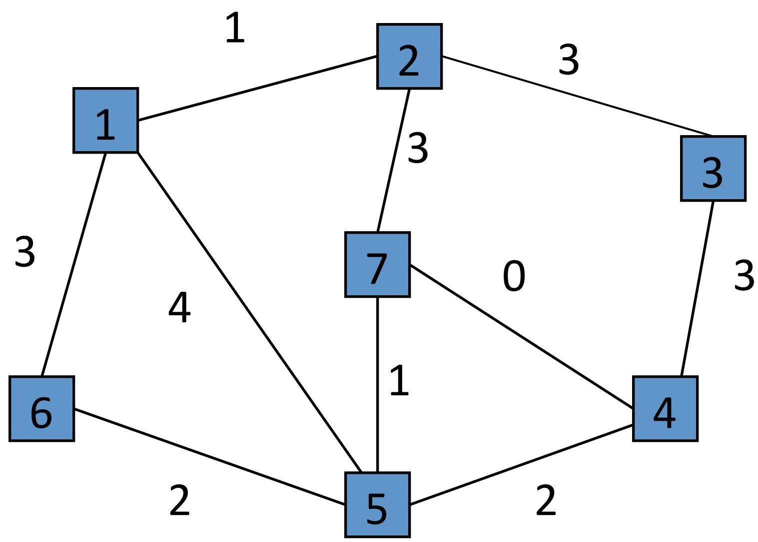
  $O(|E|\log |V|)$

Kruskal's MST Algorithm

Idea: Grow a forest out of edges that do not create a cycle. Pick an edge with the smallest weight.

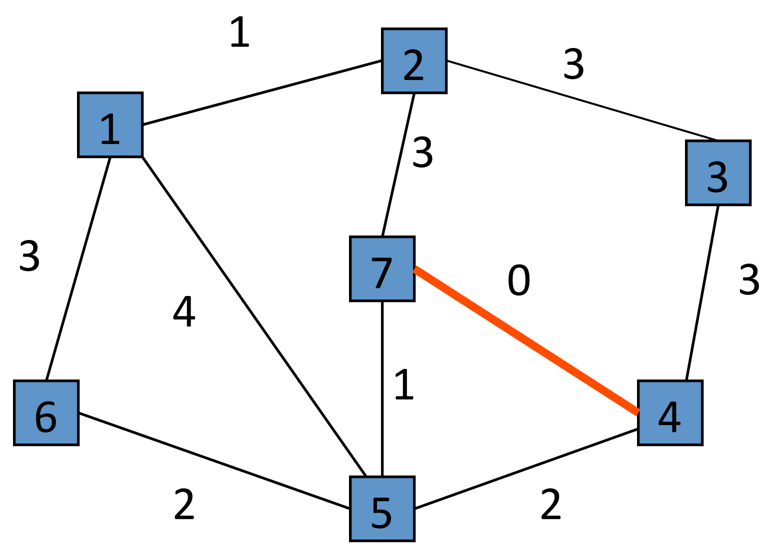


Example of Kruskal 1



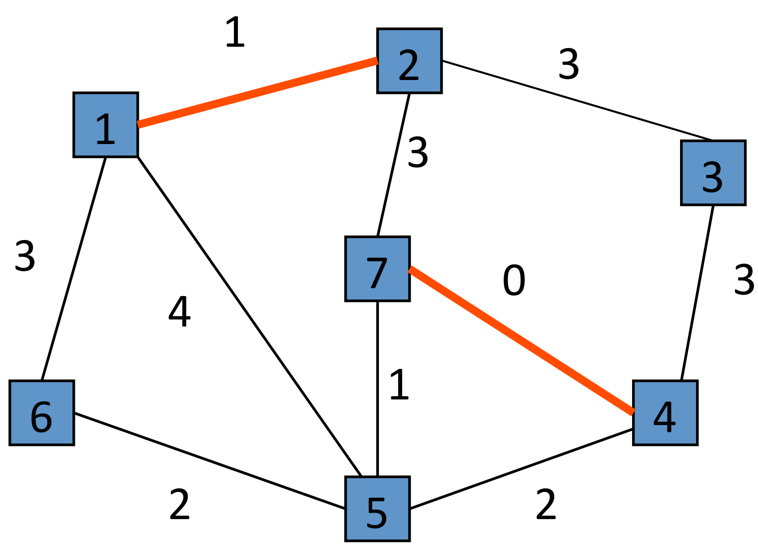
{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Example of Kruskal 2



{7,4} {2,1} {7,5} {5,6} {5,4} {1,6} {2,7} {2,3} {3,4} {1,5}
0 1 1 2 2 3 3 3 3 4

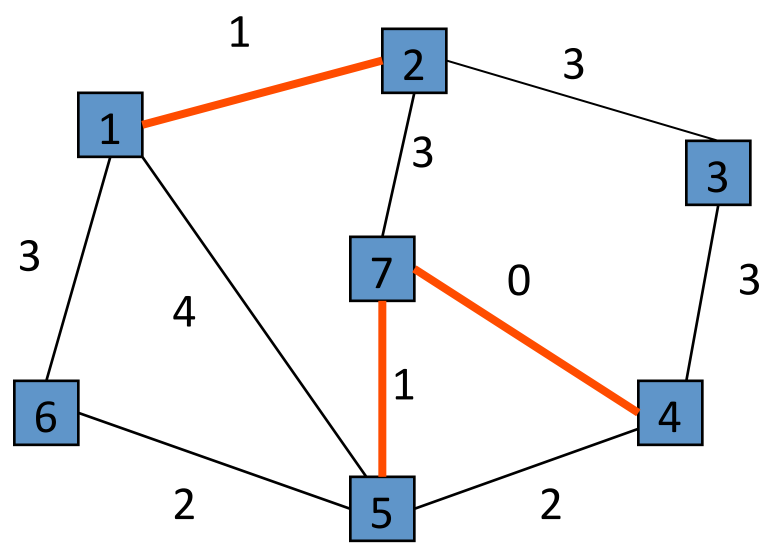
Example of Kruskal 2



{7,4} {2,1} {7,5} {5,6} {5,4} {1,6} {2,7} {2,3} {3,4} {1,5}

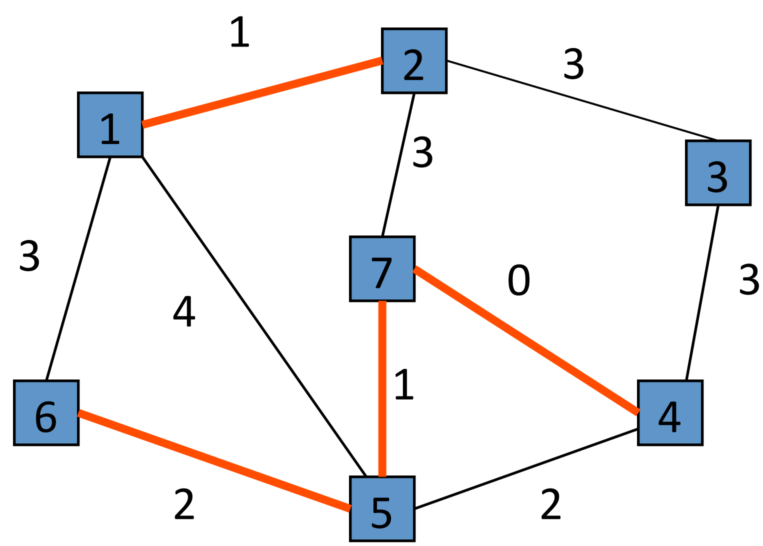
0 1 1 2 2 3 3 3 3 4

Example of Kruskal 3



{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

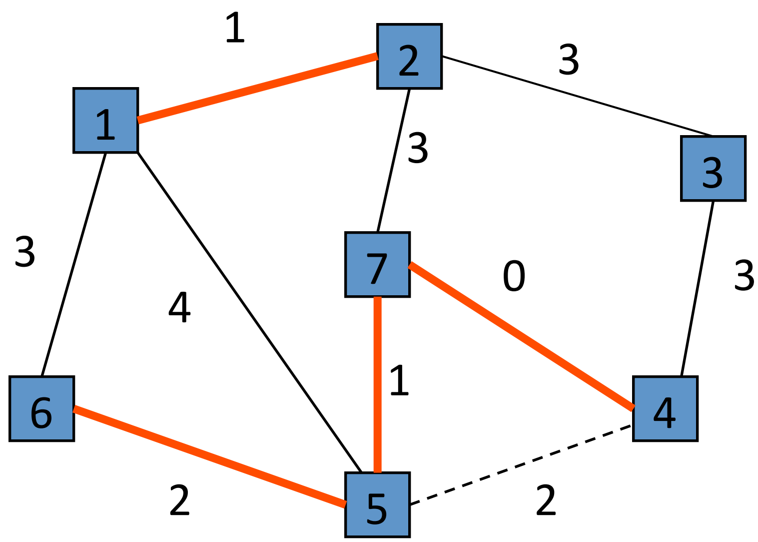
Example of Kruskal 4



{7,4} {2,1} {7,5} {5,6} {5,4} {1,6} {2,7} {2,3} {3,4} {1,5}

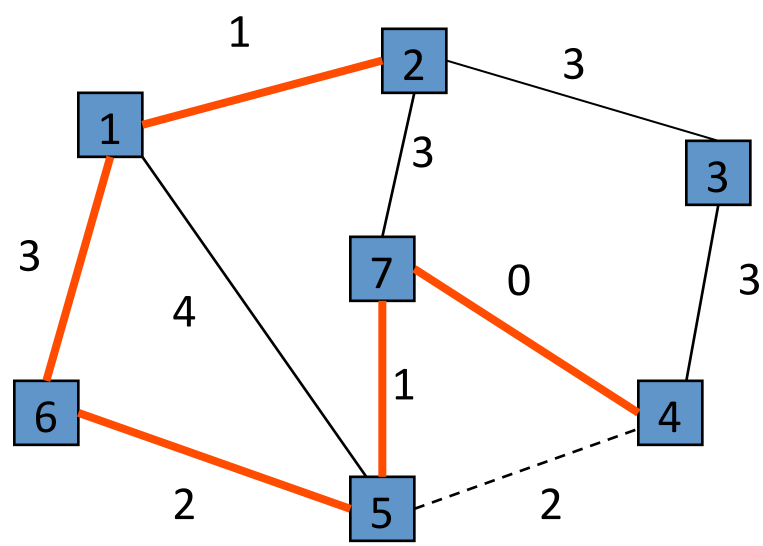
0 1 1 2 2 3 3 3 3 4

Example of Kruskal 5



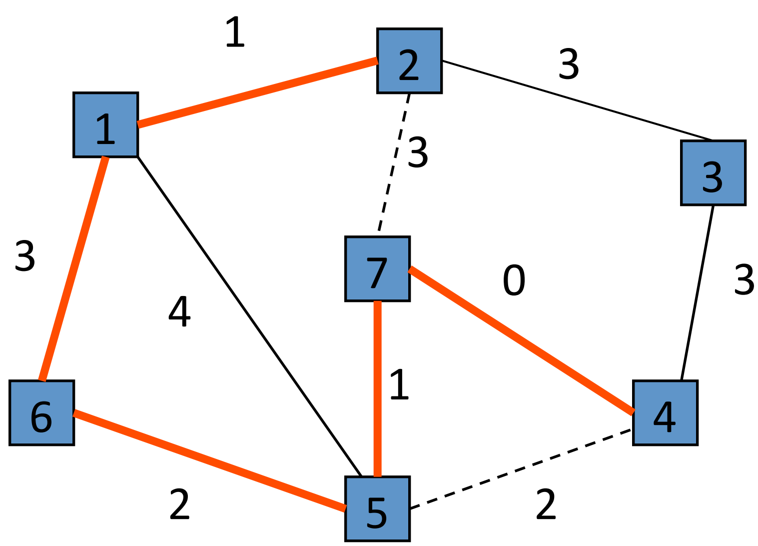
~~{7,4}~~	~~{2,1}~~	~~{7,5}~~	~~{5,6}~~	~~{5,4}~~	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
~~0~~	~~1~~	~~1~~	~~2~~	~~2~~	3	3	3	3	4

Example of Kruskal 6



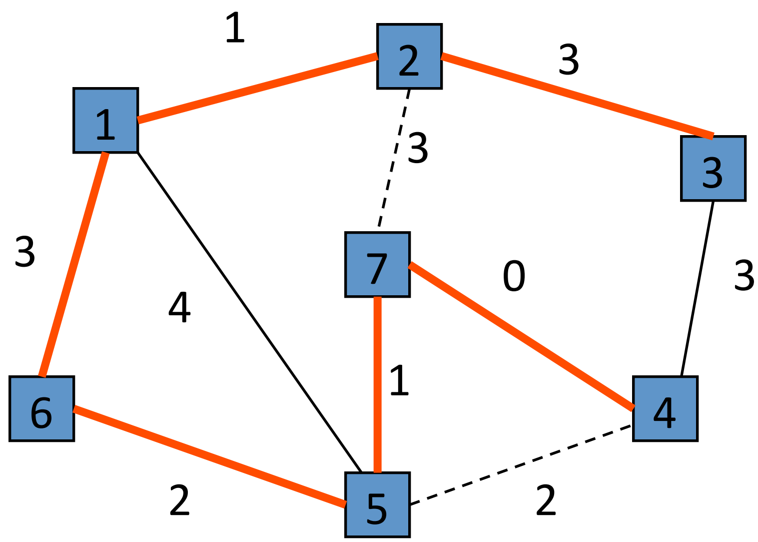
{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Example of Kruskal 7



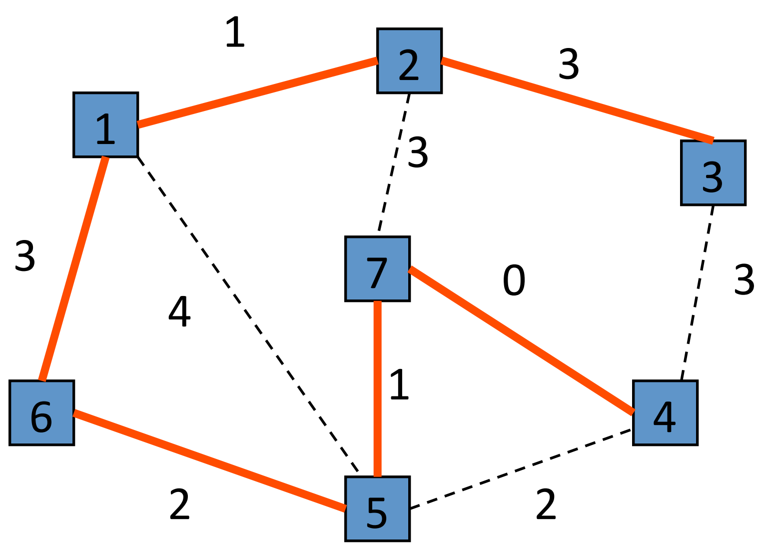
{7,4} {2,1} {7,5} {5,6} {5,4} {1,6} {2,7} {2,3} {3,4} {1,5}
0 1 1 2 2 3 3 3 3 4

Example of Kruskal 7



{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Example of Kruskal 8,9



{7,4}	{2,1}	{7,5}	{5,6}	{5,4}	{1,6}	{2,7}	{2,3}	{3,4}	{1,5}
0	1	1	2	2	3	3	3	3	4

Kruskal's Algorithm Implementation

Kruskals():
   sort edges in increasing order of length (e_1, e_2, e_3, ..., e_m).

   T := {}.

   for i = 1 to m
      if e_i does not add a cycle:
         add e_i to T.

   return T.

- But how can we determine that adding e_i to *T* won't add a cycle?