Minimum Spanning Trees

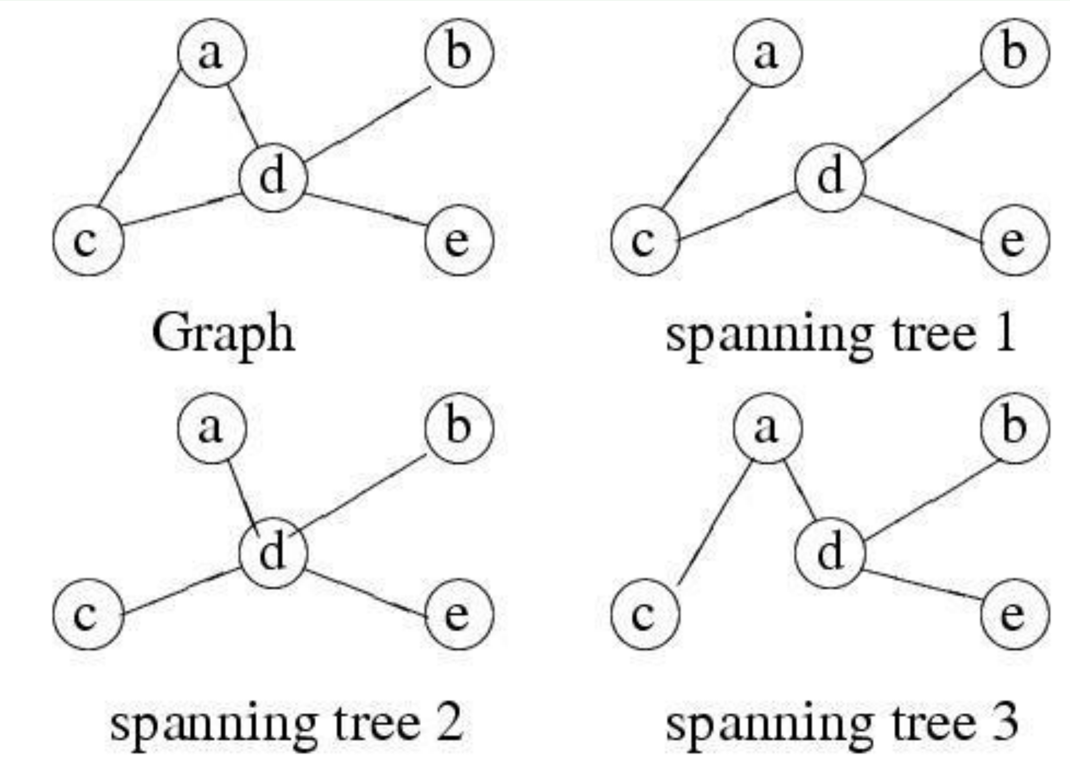
Prim’s Algorithm

Spanning Trees

Definition

A subgraph T of a undirected graph $G = (V, E)$ is a spanning tree of G if it is a tree and contains every vertex of G

Example



Graph

spanning tree 1

spanning tree 2

spanning tree 3

Spanning Trees

Theorem

Every connected graph has a spanning tree.

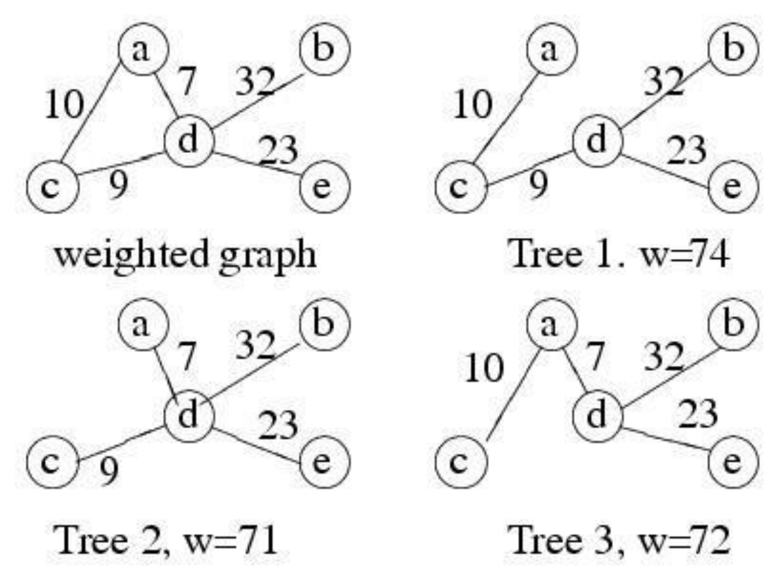
- Spanning Tree has exactly n-1 edges, one more edge will definitely create a cycle, one less edge means at least one vertex is not connected to tree.

Weighted Graph

Definition

A weighted graph is a graph, in which each edge has a weight (some real number) Could denote length, time, strength, etc.

Example



Definition

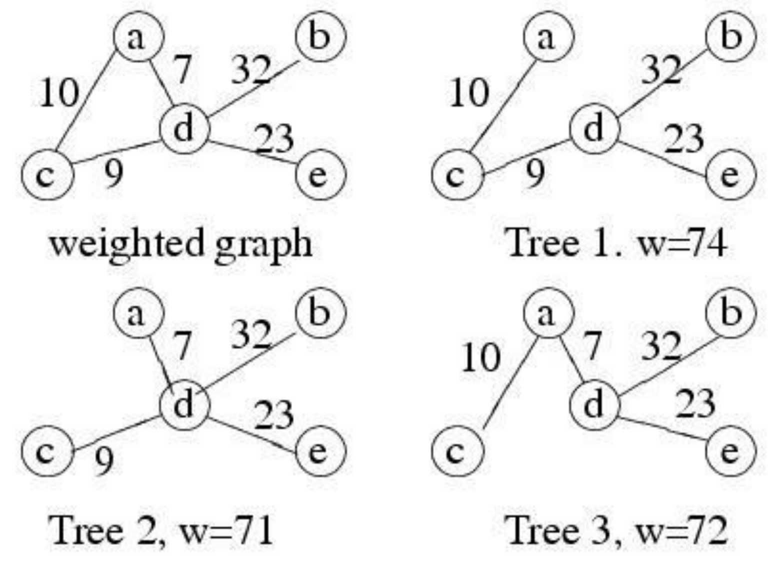
Weight of a graph: The sum of the weights of all edges

Minimum Spanning Trees

Definition

A Minimum spanning tree (MST) of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

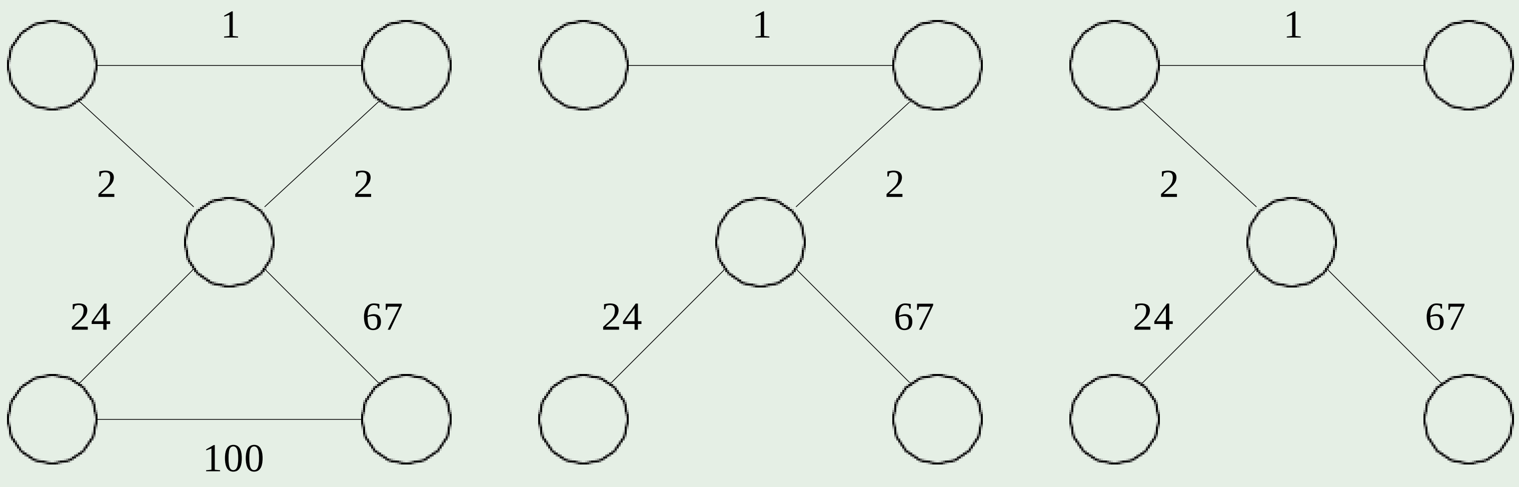
Example



Minimum Spanning Trees

The minimum spanning tree may not be unique

Example



weighted graph MST1 MST2

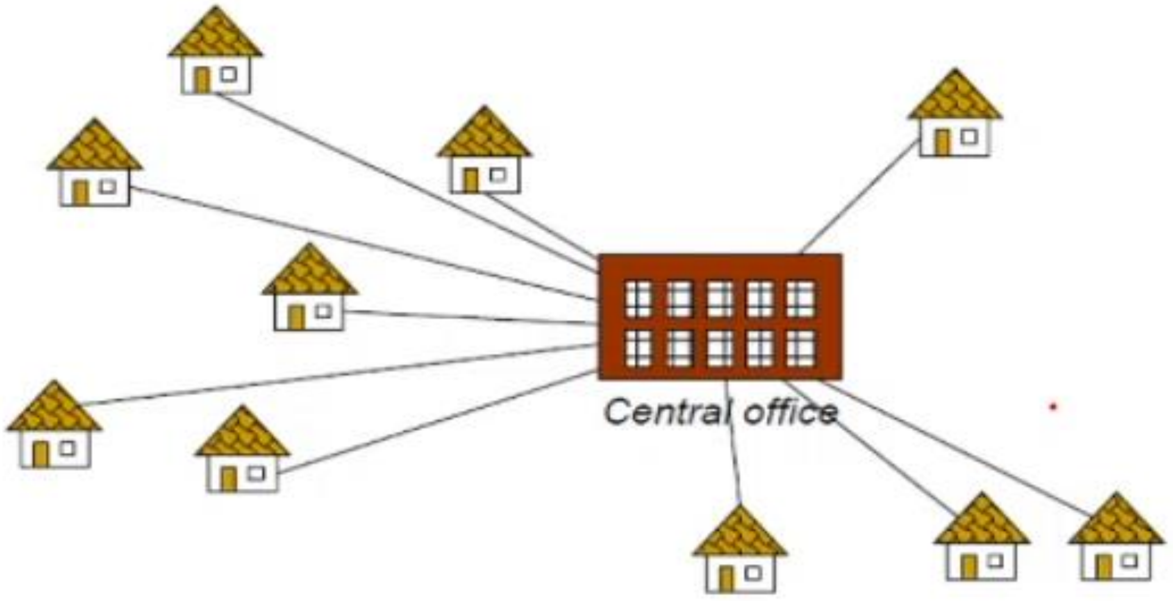
Note: if the weights of all the edges are distinct, MST is unique

Minimum Spanning Tree Problem

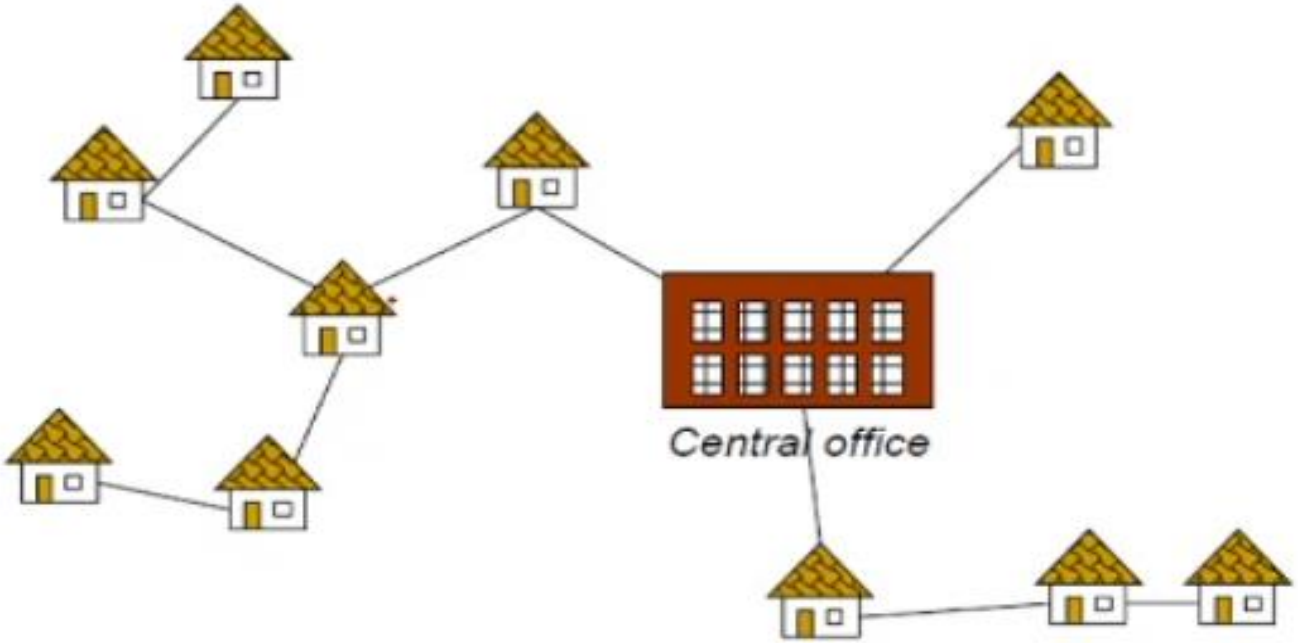
Definition (MST Problem)

Given a connected weighted **undirected** graph G, design an algorithm that outputs a minimum spanning tree (MST) of G.

Wiring: A Naïve Approach



Wiring: A Better Approach



Minimum Spanning Tree Problem

Input: A graph G = (V, E) and a cost c_e for each edge e.

- Adjacency list representation of graph
- Edges can have negative weights

Output: A minimum cost tree T ⊆ E, that spans all vertices.

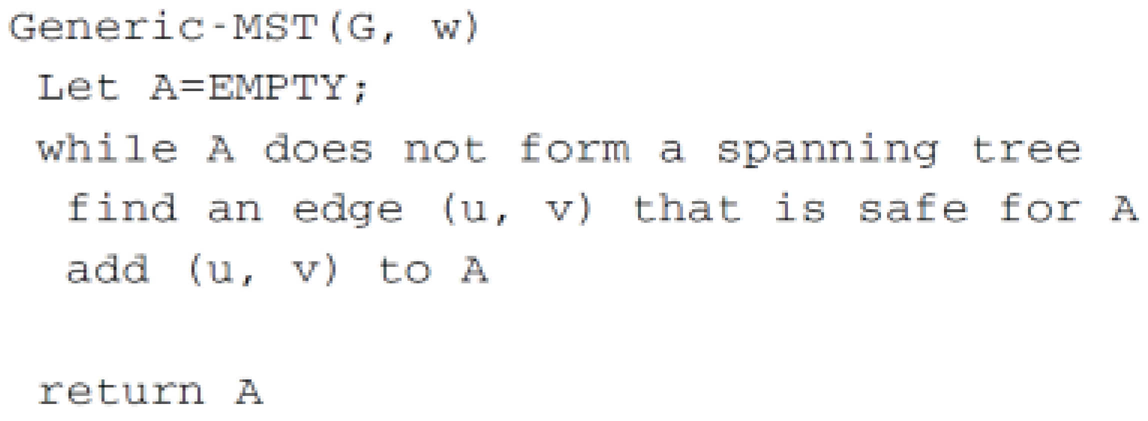
- T has no cycle
- The subgraph (V, T) is connected i.e. contains path between each pair of vertices

Approach

- We know tree is an acyclic graph.
- Idea:
  - Start with an empty graph and try to add edges one at a time, always making sure that what is built is an acyclic graph. And if we are making sure every time that the resulting graph is always a subset of some minimum spanning tree, we are done.
- Question: How to find this safe edge that satisfies the above conditions?

Generic Prim’s Algorithm for MST problem

- Let A be the set of edges such that $A \subseteq T$, where T is a MST. An edge (u, v) is a safe edge for A if $A \cup (u, v)$ is also a subset of some MST
- If at each step we can find such an edge, we can grow MST



```
Generic-MST(G, w)
 Let A=EMPTY;
 while A does not form a spanning tree
  find an edge (u, v) that is safe for A
  add (u, v) to A

 return A
```

Question: How to find this safe edge?

Some Important Definitions

- Let G(V, E) be a weighted, undirected graph

- **Cut:** A cut (S, V\S) is a partition of V

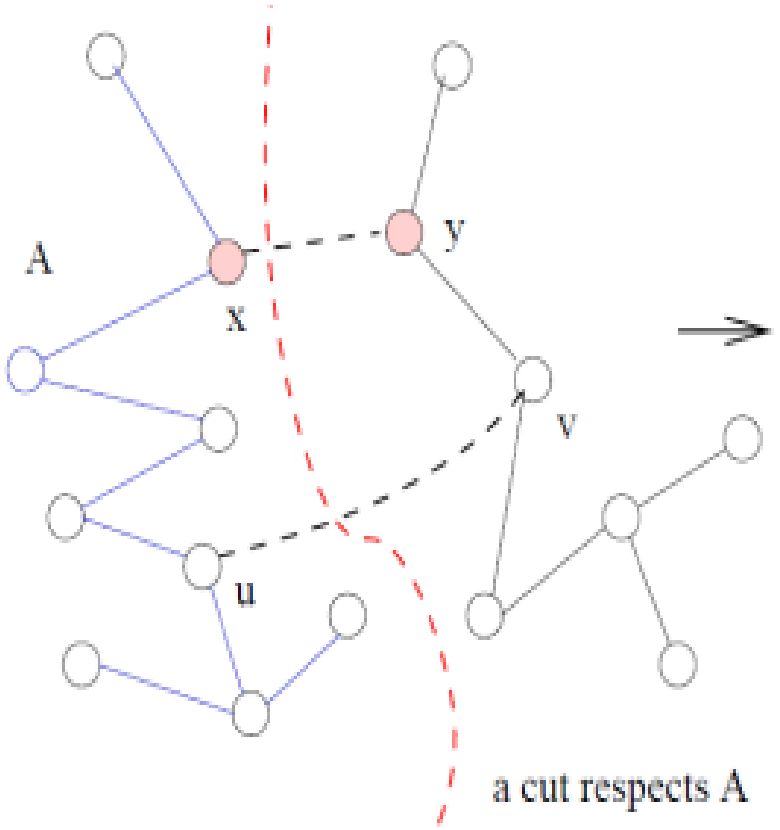
- **Cross Edge:** An edge (u, v) is a cross edge of a Cut (S, V\S) if one of its end point is in S and other in V\S

- **Light edge:** An edge is a **light edge** crossing a cut if its weight is the minimum of any edge crossing the cut

- A cut **RESPECTS** a set A of edges if no edge in A is a cross edge of the cut

How to find a Safe Edge?

- Lemma 1: Let G(V, E) be a connected, weighted, undirected graph. Let A be a subset of E that is included in some MST of G. Also let (S, V\S) be any cut of G that respects A and (u, v) be a light edge crossing the cut (S, V\S) then (u, v) is a safe edge.
- The Idea is to first find a cut that respects A and then find the light edge that crosses the cut. This light edge will be the safe edge

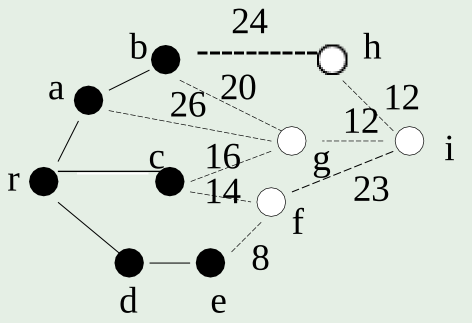


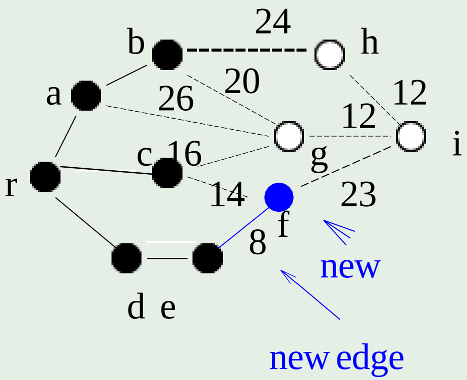
Prim’s Algorithm

- Prim's algorithm for finding an MST is a greedy algorithm.
- Start by selecting an arbitrary vertex, include it into the current MST.
- The *Prim’s* algorithm makes a natural choice of the cut in each iteration – it grows a single tree and adds a light edge in each iteration.
- Grow the current MST by selecting the safe edges.
- Cut makes two partitions(1st partition: vertices in MST, 2nd partition: Vertices not in current MST)
- Find the light edge for this Cut i.e. find the cross edge with minimum weight and add it to the current MST

Prim’s Algorithm

Example





Step 0:

- Choose any element r; set $S = \{r\}$ and $A = \emptyset$.
- (Take r as the root of our spanning tree.)

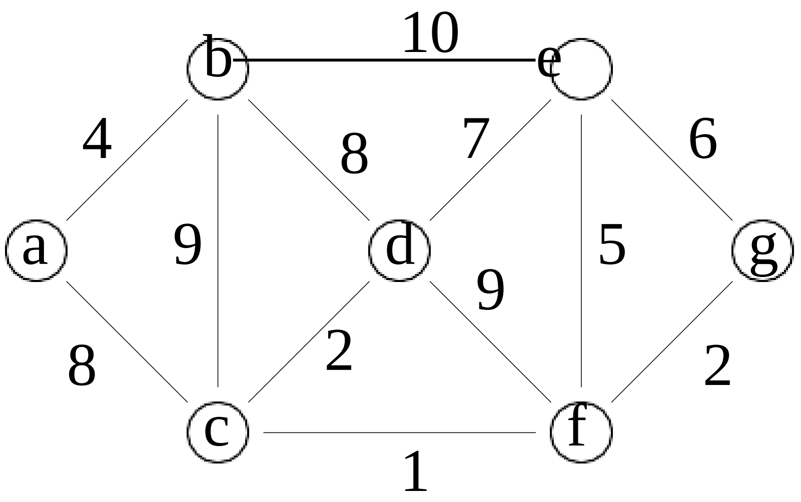
Step 1:

- Find a lightest edge such that one endpoint is in S and the other is in $V \setminus S$.
- Add this edge to A and its (other) endpoint to S.

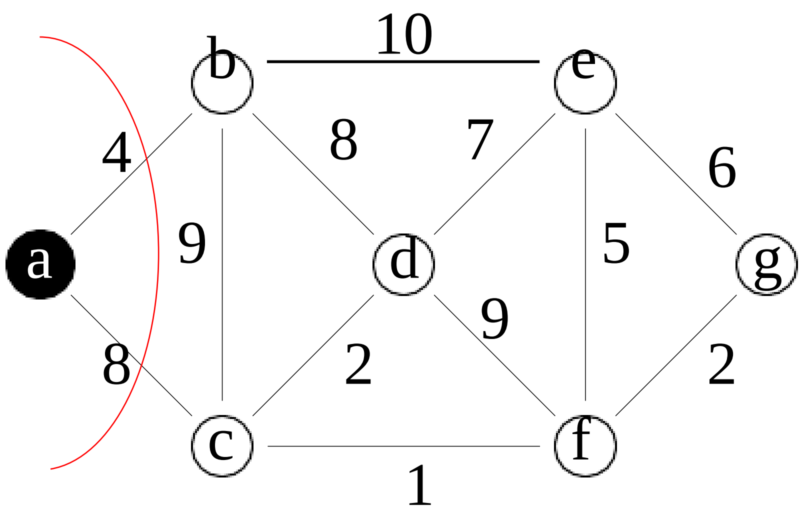
Step 2:

- If $V \setminus S = \emptyset$, then stop and output (minimum) spanning tree (S, A); Otherwise, go to Step 1.

Worked Example



Connected graph



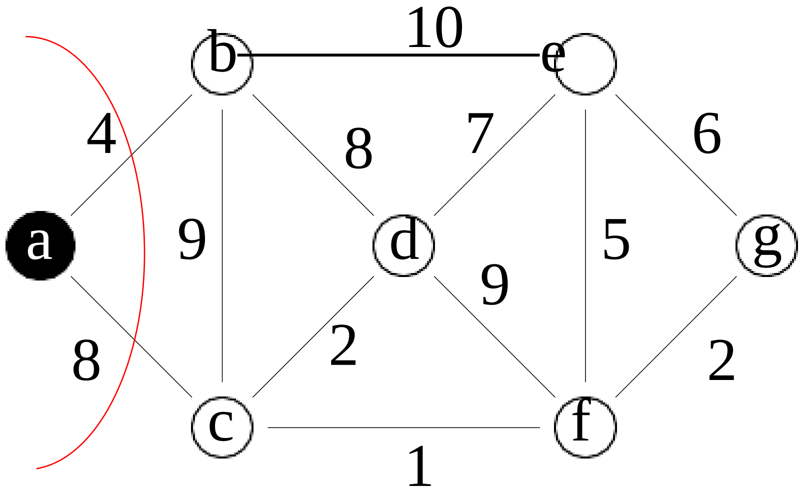
Step 0

S={a}

V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

Worked Example



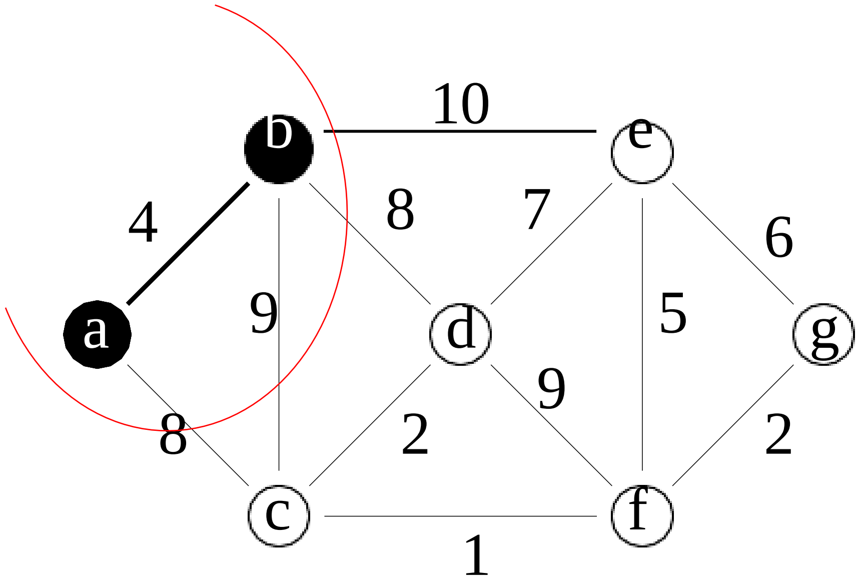
Step 1.1 before

S={a}

V \ S = {b,c,d,e,f,g}

A={}

lightest edge = {a,b}



Step 1.1 after

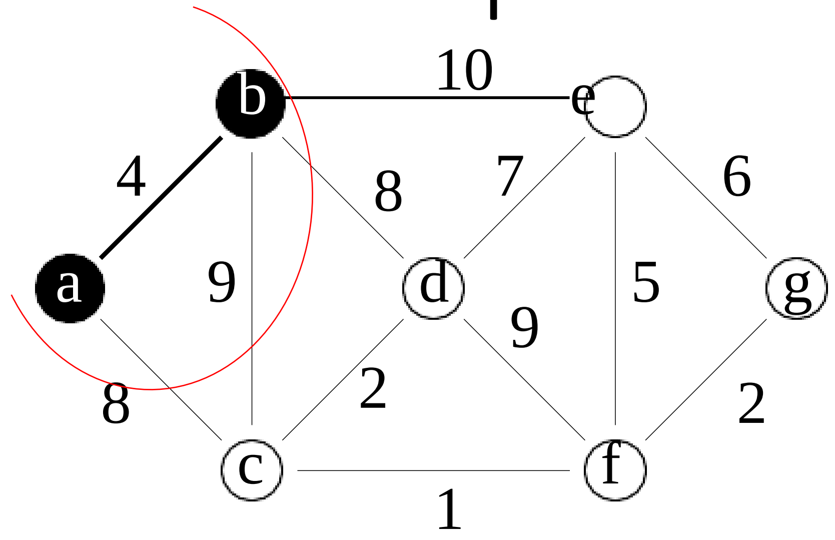
S={a,b}

V \ S = {c,d,e,f,g}

A={{a,b}}

lightest edge = {b,d}, {a,c}

Worked Example



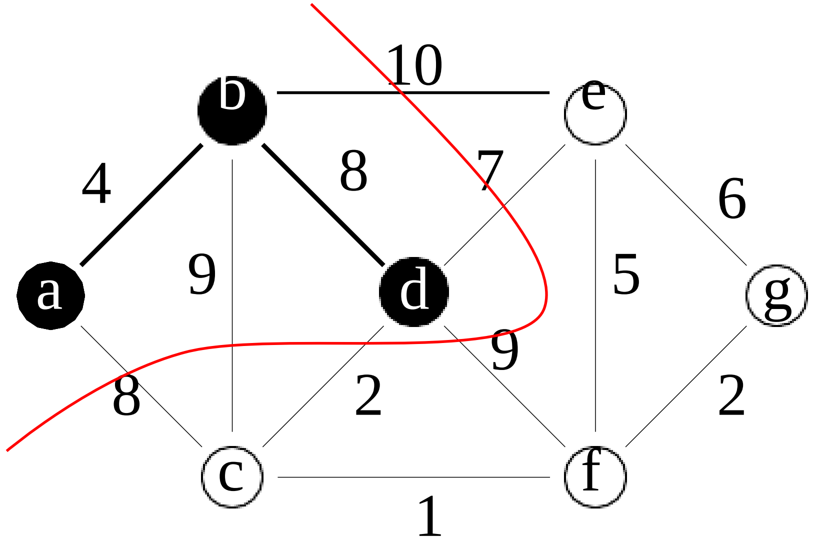
Step 1.2 before

S={a,b}

V \ S = {c,d,e,f,g}

A={{a,b}}

lightest edge = {b,d}, {a,c}



Step 1.2 after

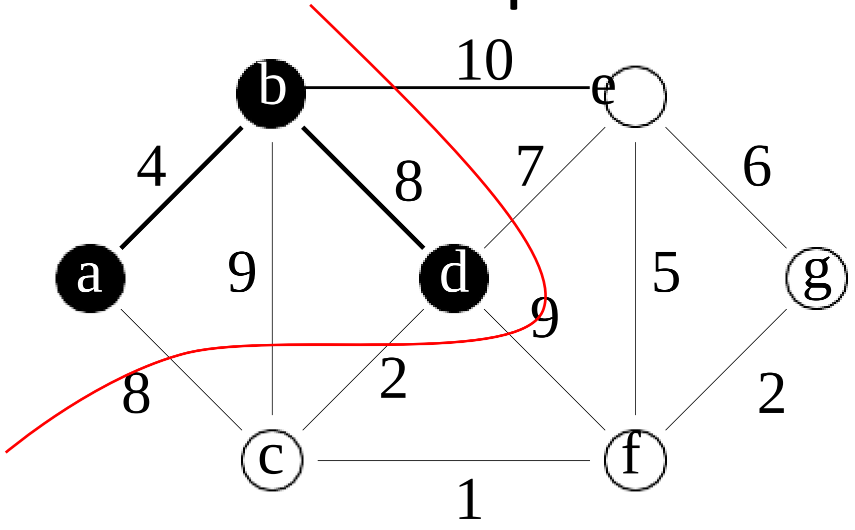
S={a,b,d}

V \ S = {c,e,f,g}

A={{a,b},{b,d}}

lightest edge = {d,c}

Worked Example



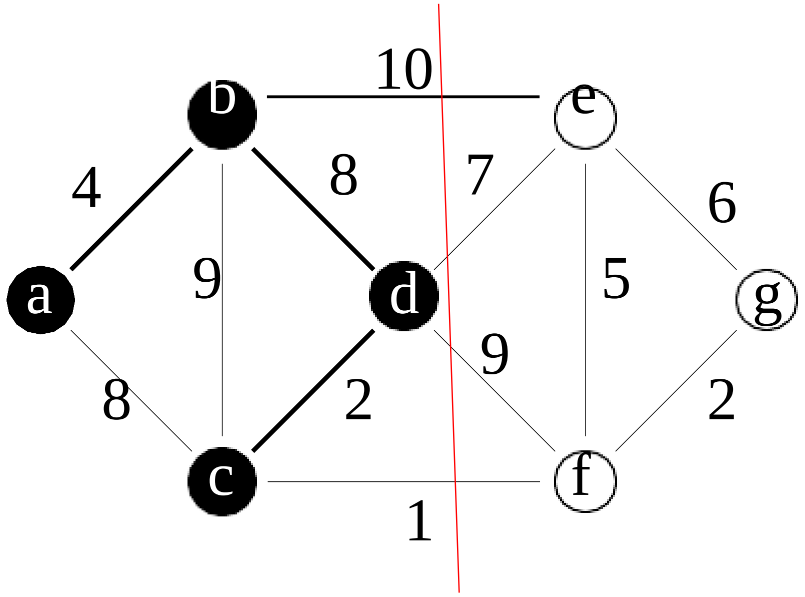
Step 1.3 before

S={a,b,d}

V \ S = {c,e,f,g}

A={{a,b},{b,d}}

lightest edge = {d,c}



Step 1.3 after

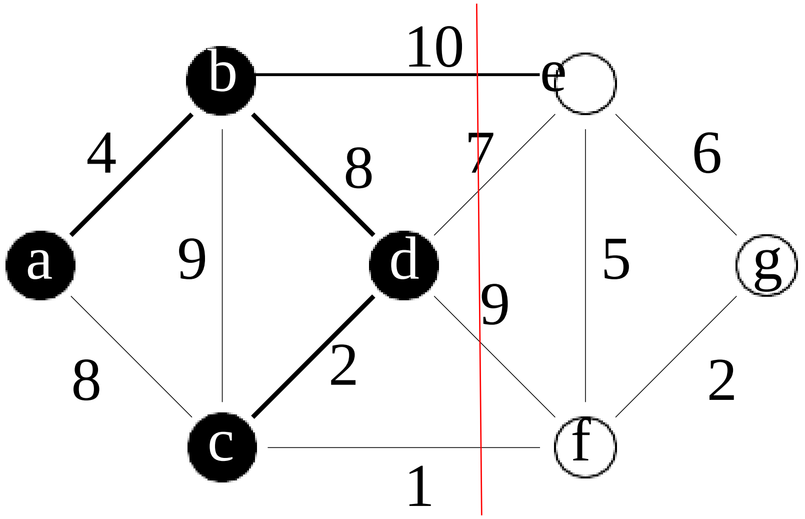
S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}

Worked Example



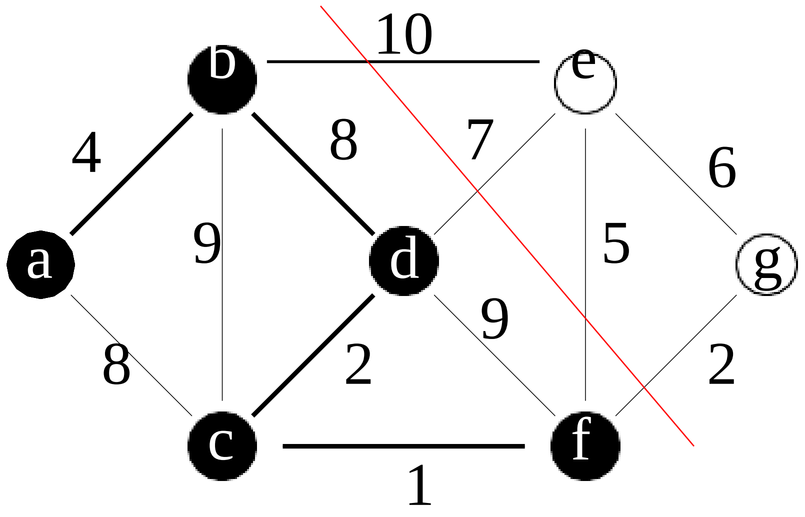
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



Step 1.4 after

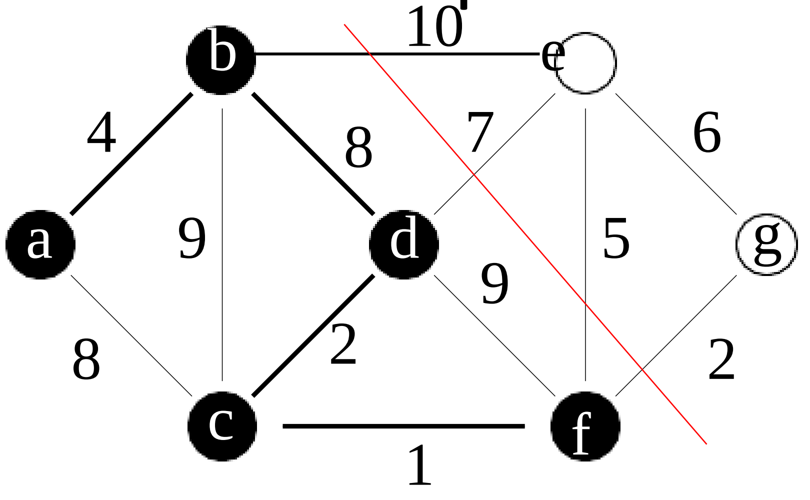
S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}

Worked Example



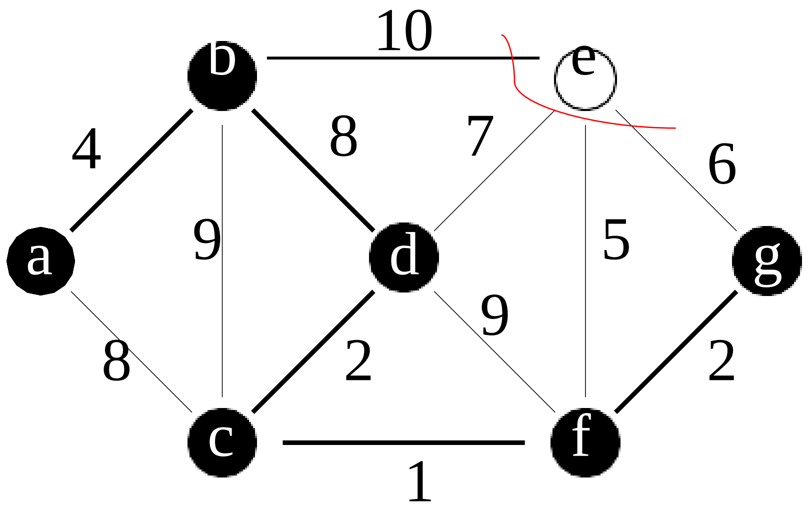
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

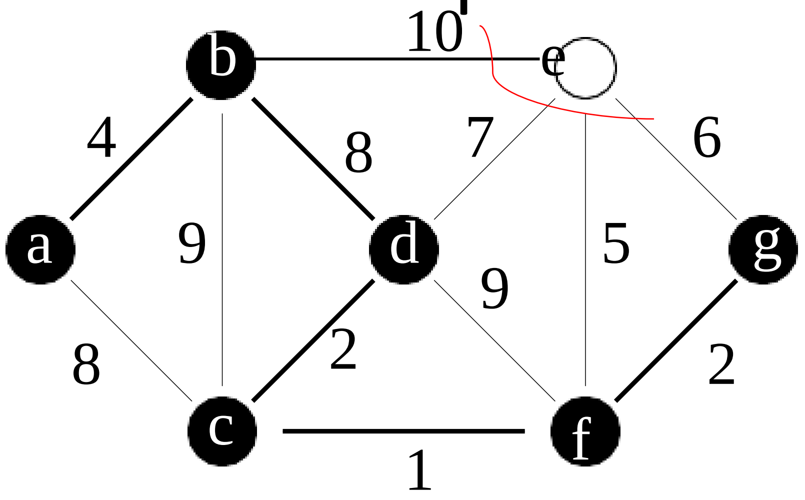
S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

Worked Example



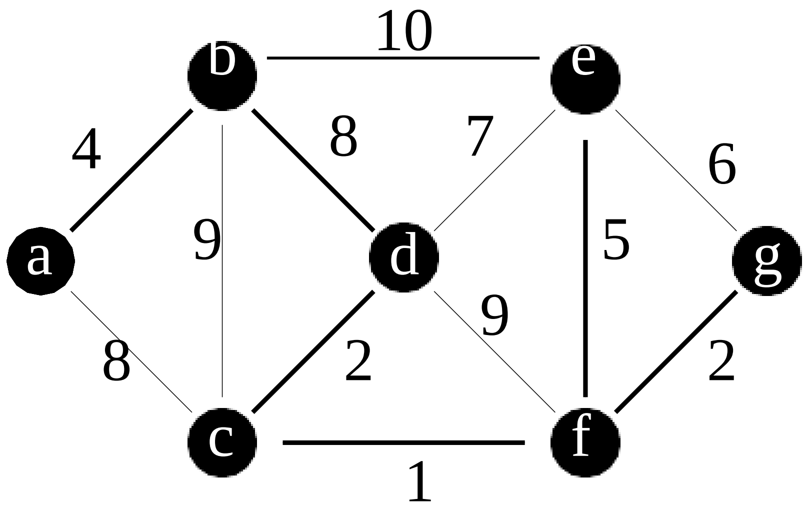
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed

Efficient Implementation

- We are doing repeated minimum computations
i.e. selecting cheapest edge
- Which data structure to use when you need to do
repeated minimum computations?

Efficient Implementation (Use MinHeap)

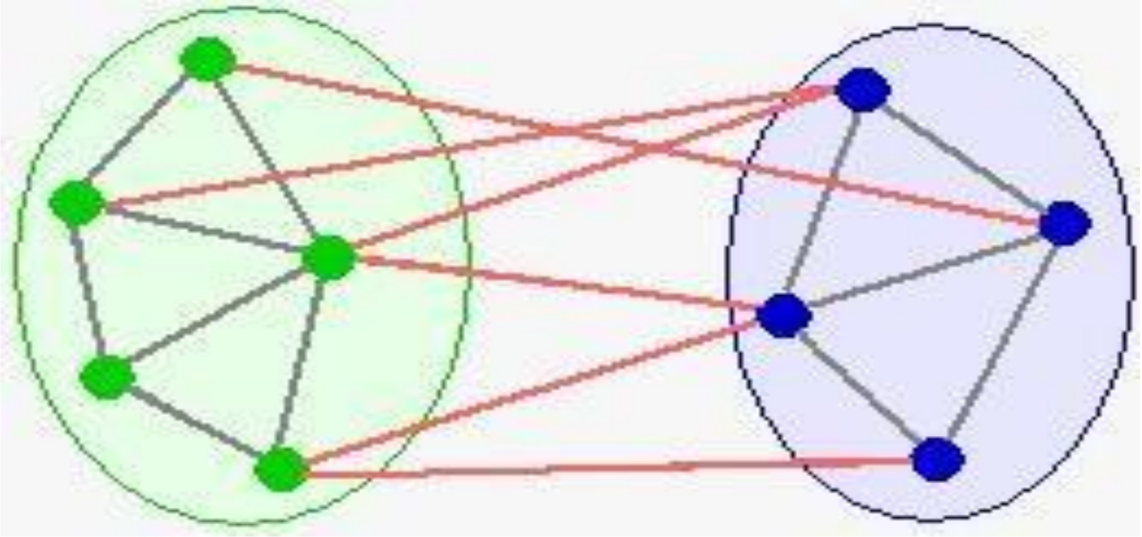
- What to store in minHeap?
- We can use edge costs as keys and store edges in heap.
- Initialize heap with m edges in O(m) time using BuildHeap()
- In every iteration use extractMin() to get edge with minimum cost.
- Running time will be O(m lg m) or O(m lg n)
- m is at most n^2 so lg m = lg n^2 = 2lgn = O(lg n)

Efficient Implementation

- A more efficient implementation will be using vertices in heap instead of edges. (it will give asymptotically same time but better constants since number of vertices is less or equal to number of edges)

- But what should be key of vertices in heap ?

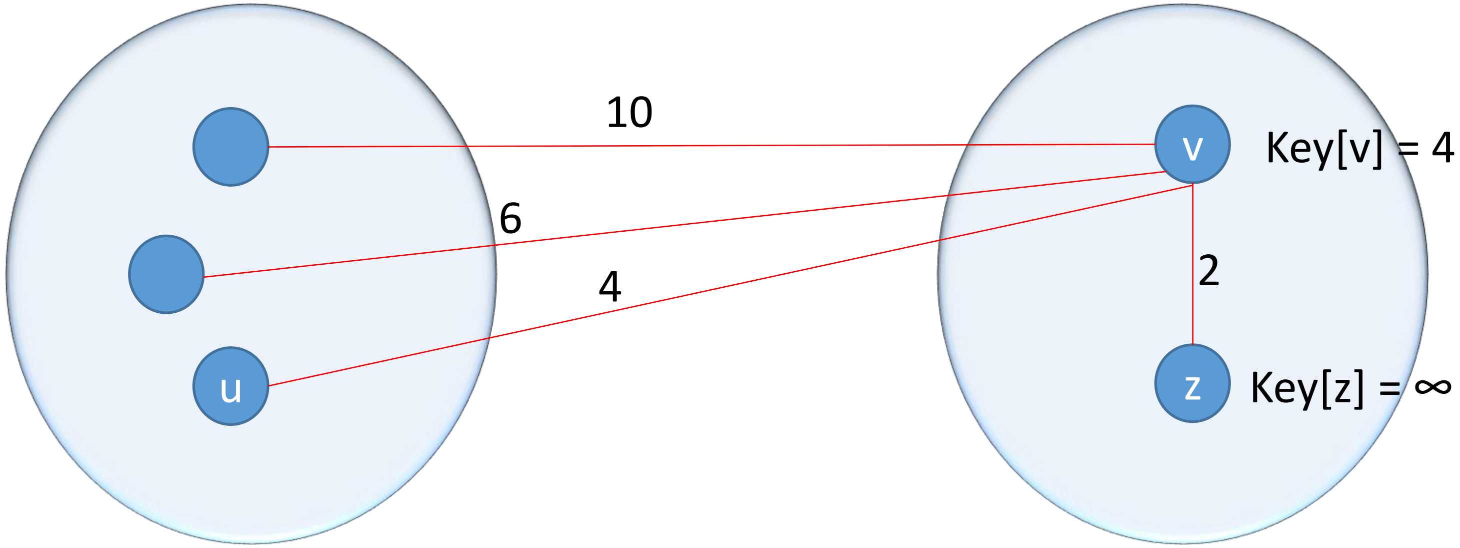
Efficient Implementation



Vertices in MST
(set: S)

Vertices in MinHeap
(set: V – S)

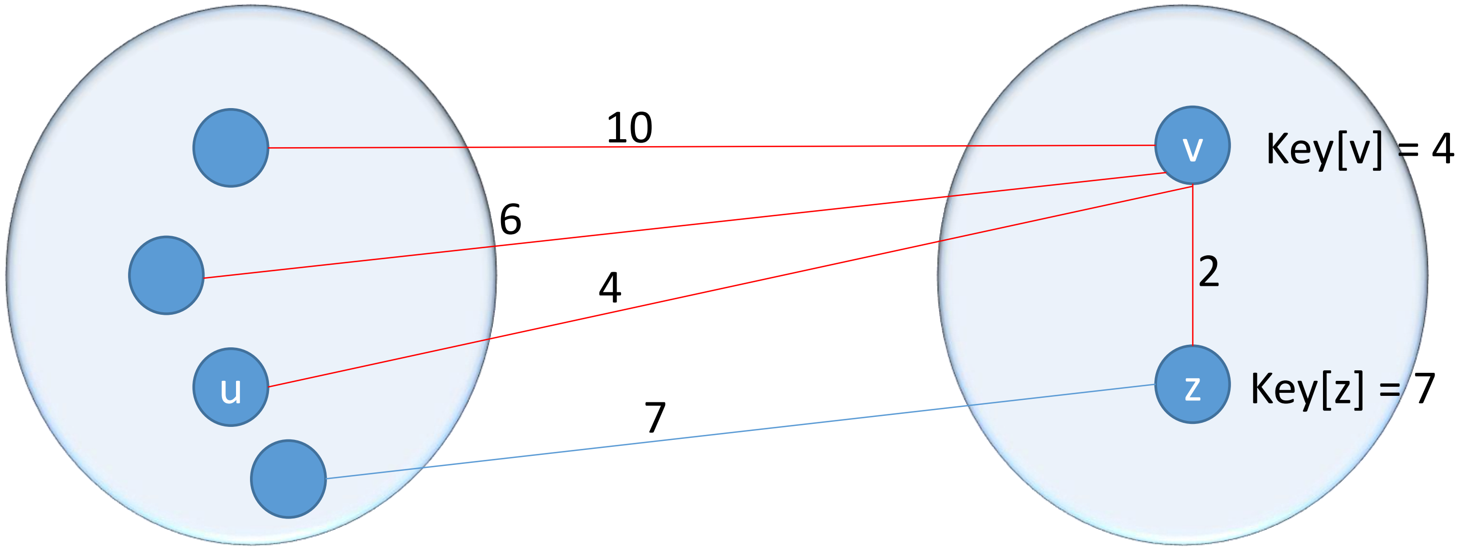
Efficient Implementation



Vertices in MST
(set: S)

Vertices in MinHeap
(set: V – S)

Efficient Implementation

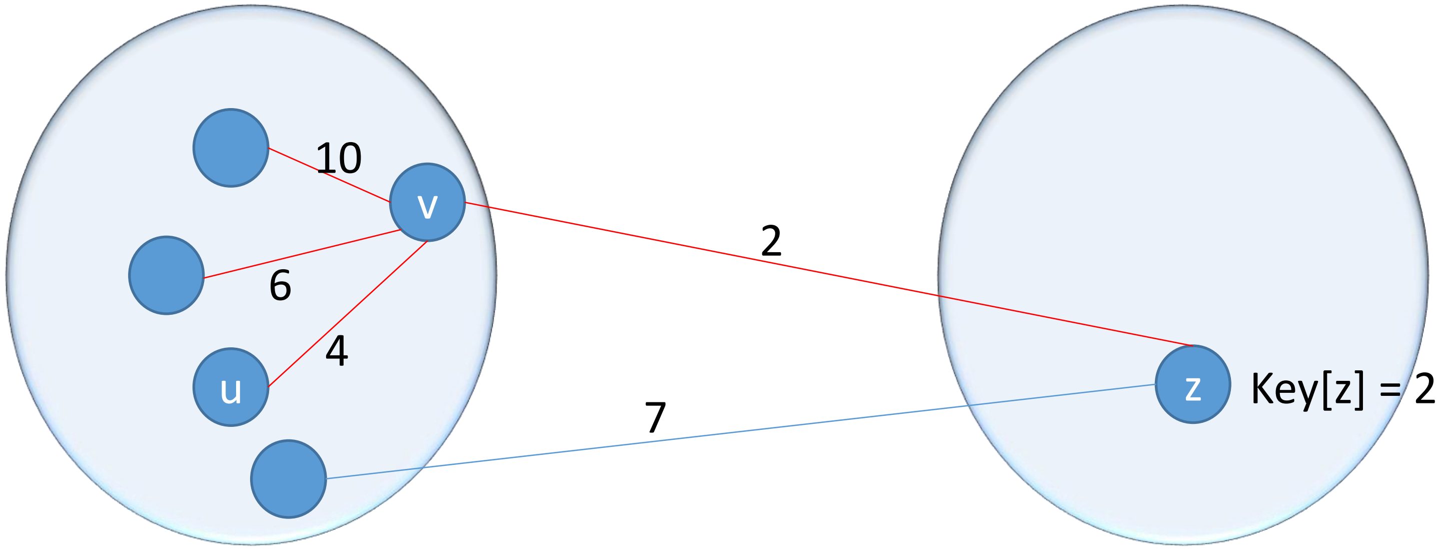


For each vertex v in MinHeap, the key will be cheapest edge (u,v) such that u ∈ S

Vertices in MST
(set: S)

Vertices in MinHeap
(set: V – S)

Efficient Implementation



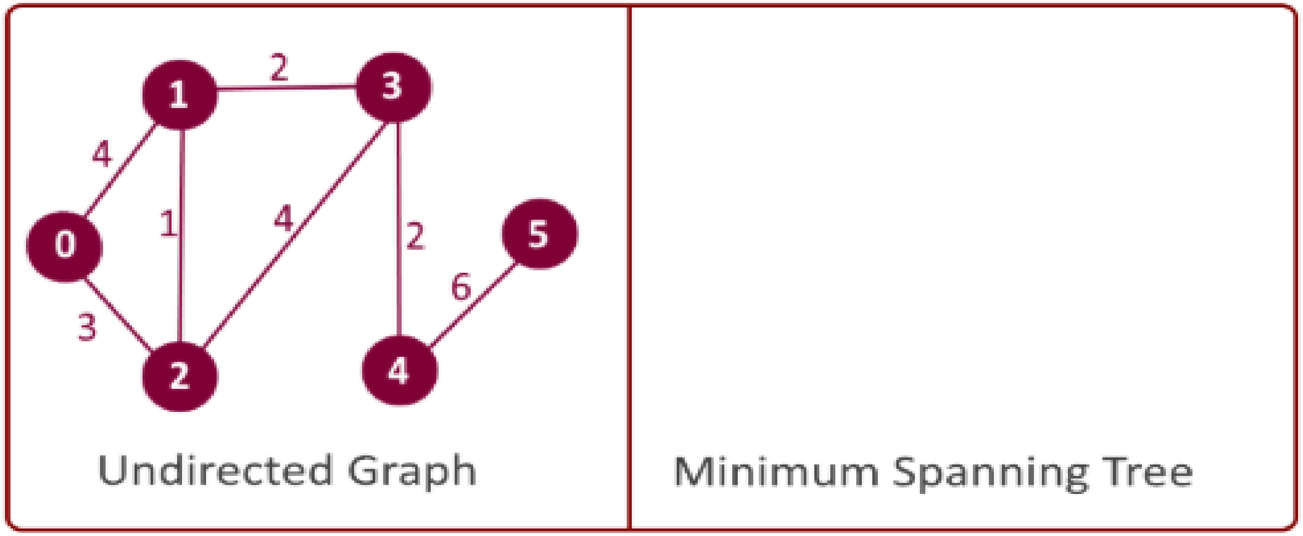
Vertices in MST
(set: S)

Vertices in MinHeap
(set: V – S)

Prim’s Algorithm – MinHeap

```
1) T = $\emptyset$   [invariant: S = vertices spanned by tree-so-far T]
2) for each $u \in V$
        Key[u] = $\infty$
3) Key[s] = 0 // select any random vertex and make its cost 0
4) Heap Q is initialized with all vertices
5) While heap $\neq \emptyset$
   •  u = ExtractMin from Heap
   •  Add u to S
   •  for each $v \in$ adj[u]
           DecreaseKey(Q, v, cost[u,v])    // DecreaseKey will update key of
                                              vertex v only if cost[u,v] < key [v]
```

Example

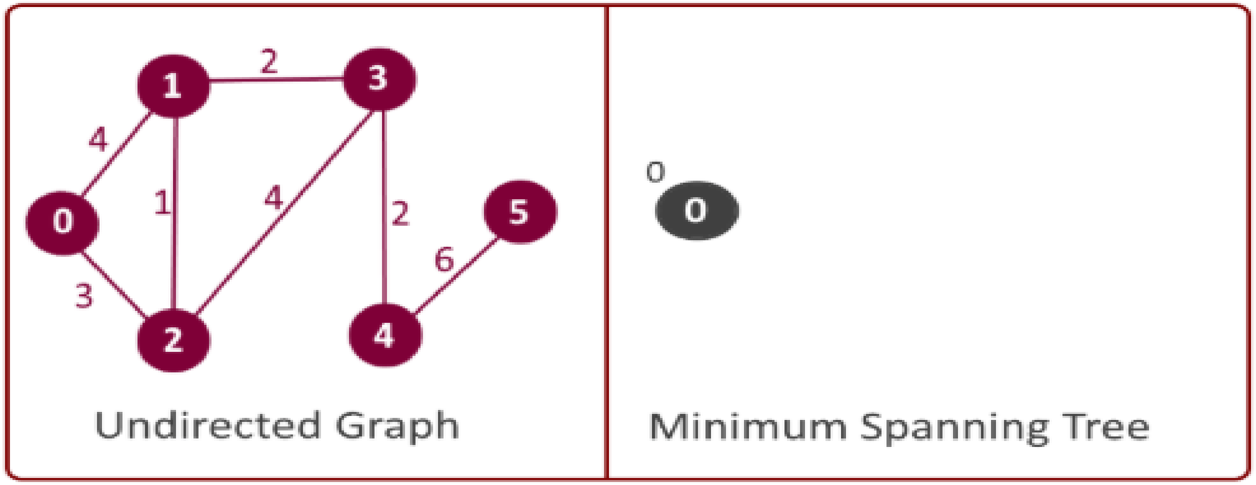


Undirected Graph

Minimum Spanning Tree

Min Heap	0	1	2	3	4	5
	0	+∞	+∞	+∞	+∞	+∞

Example



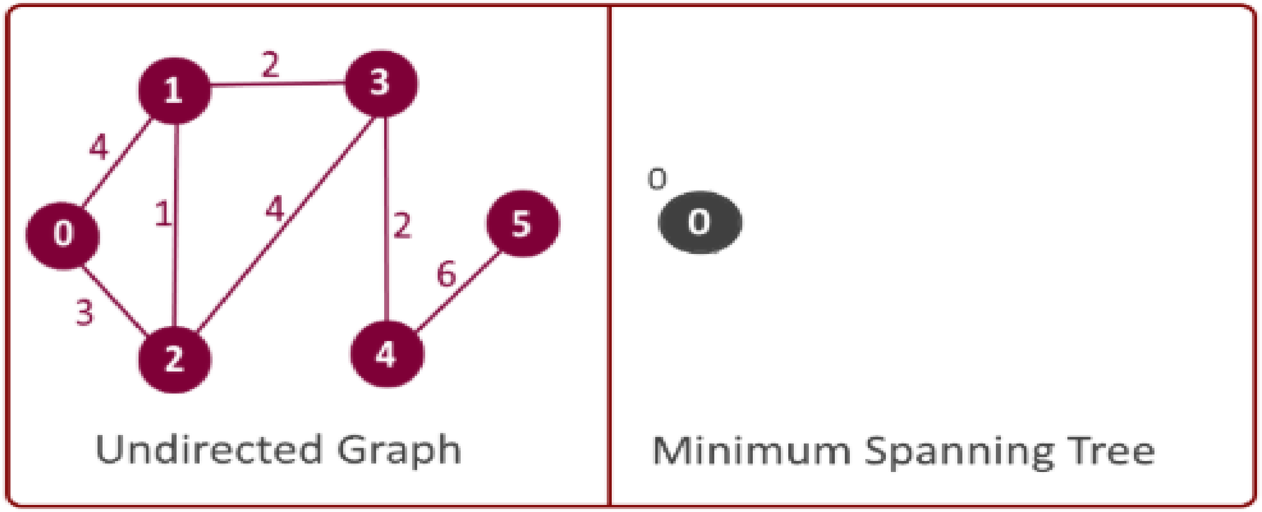
Min Heap	0	1	2	3	4	5
	0	+∞	+∞	+∞	+∞	+∞

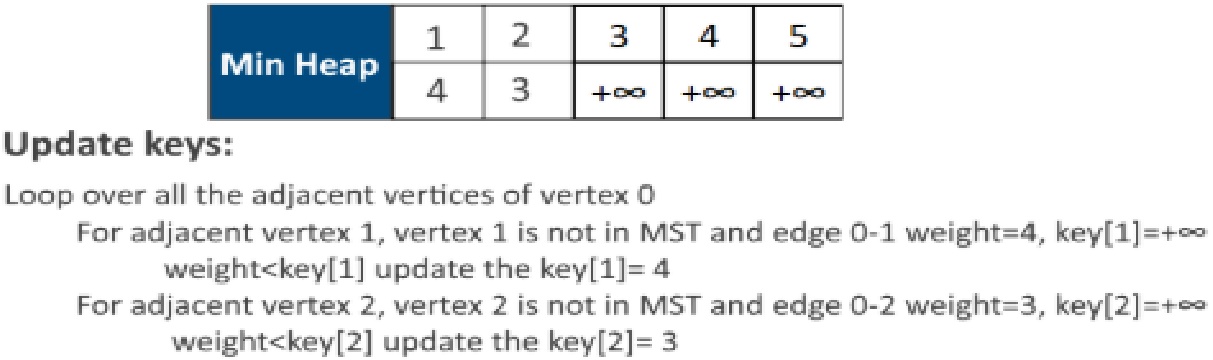
Extract-Min:

Pick the vertex has minimum Key: vertex 0

Include vertex 0 in MST

Example





Min Heap	1	2	3	4	5
	4	3	+∞	+∞	+∞

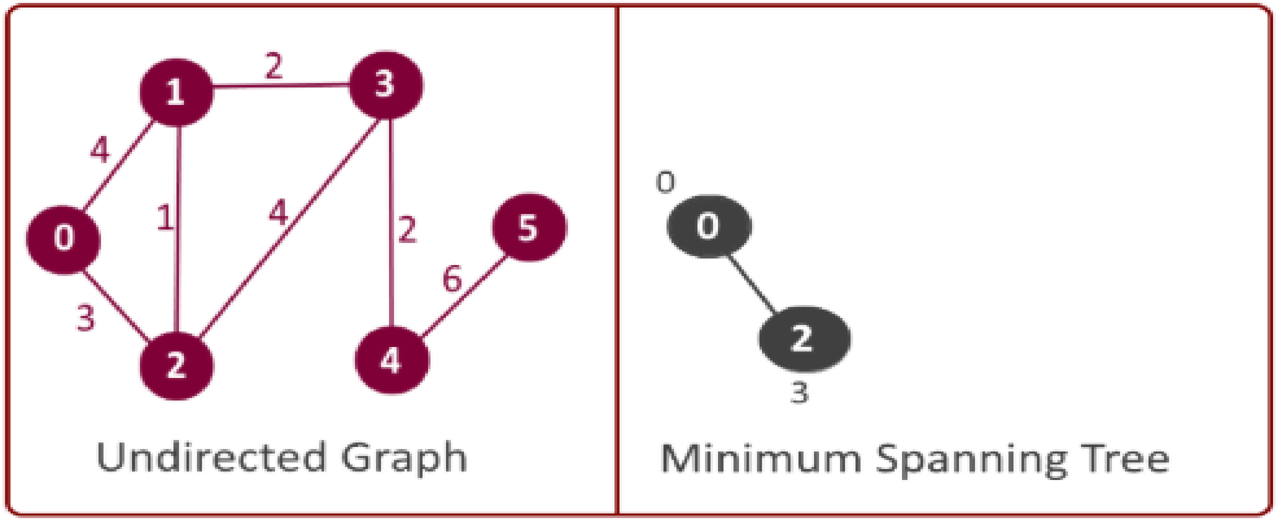
Update keys:

Loop over all the adjacent vertices of vertex 0

For adjacent vertex 1, vertex 1 is not in MST and edge 0-1 weight=4, key[1]=+∞
weight<key[1] update the key[1]= 4

For adjacent vertex 2, vertex 2 is not in MST and edge 0-2 weight=3, key[2]=+∞
weight<key[2] update the key[2]= 3

Example



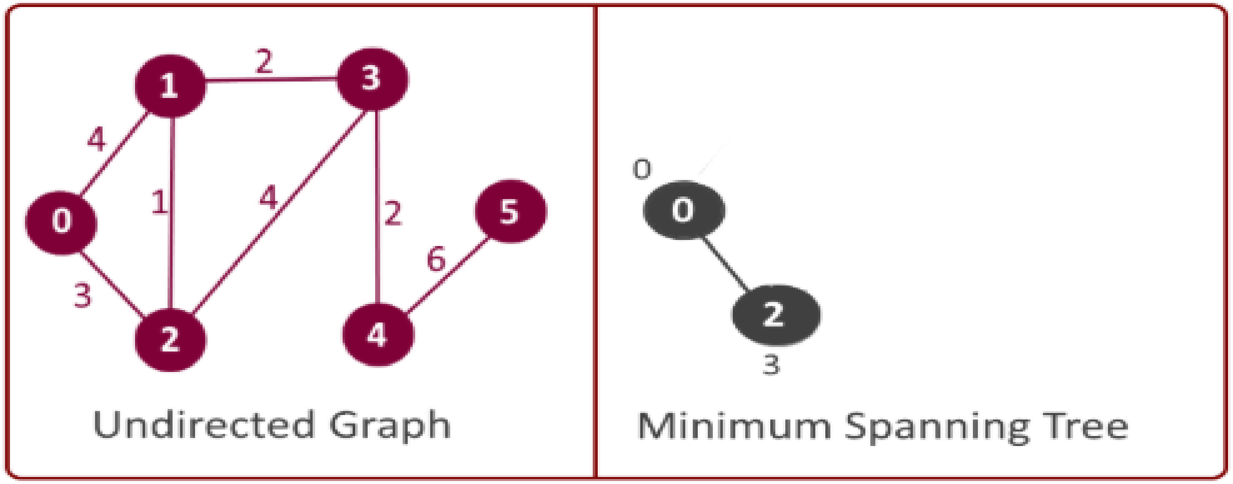
Min Heap	1	2	3	4	5
	4	3	+∞	+∞	+∞

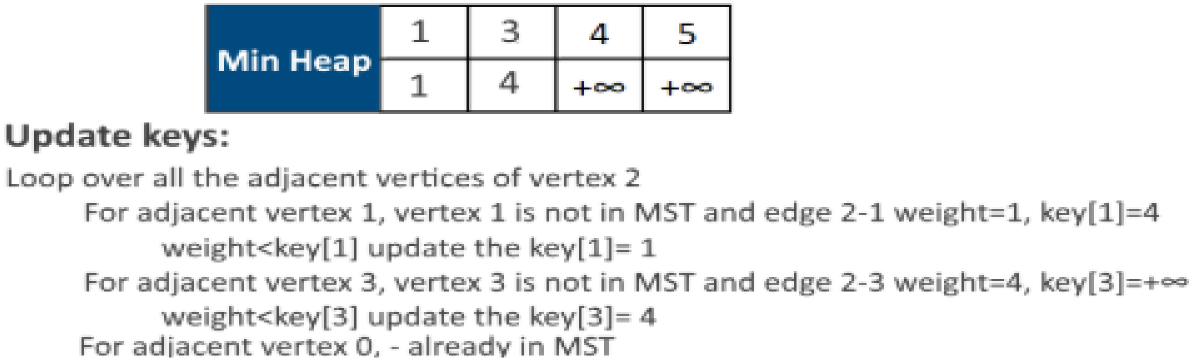
Extract-Min:

Pick the vertex which has minimum Key: vertex 2

Include vertex 2 in MST

Example





Min Heap	1	3	4	5
	1	4	+∞	+∞

Update keys:

Loop over all the adjacent vertices of vertex 2

For adjacent vertex 1, vertex 1 is not in MST and edge 2-1 weight=1, key[1]=4

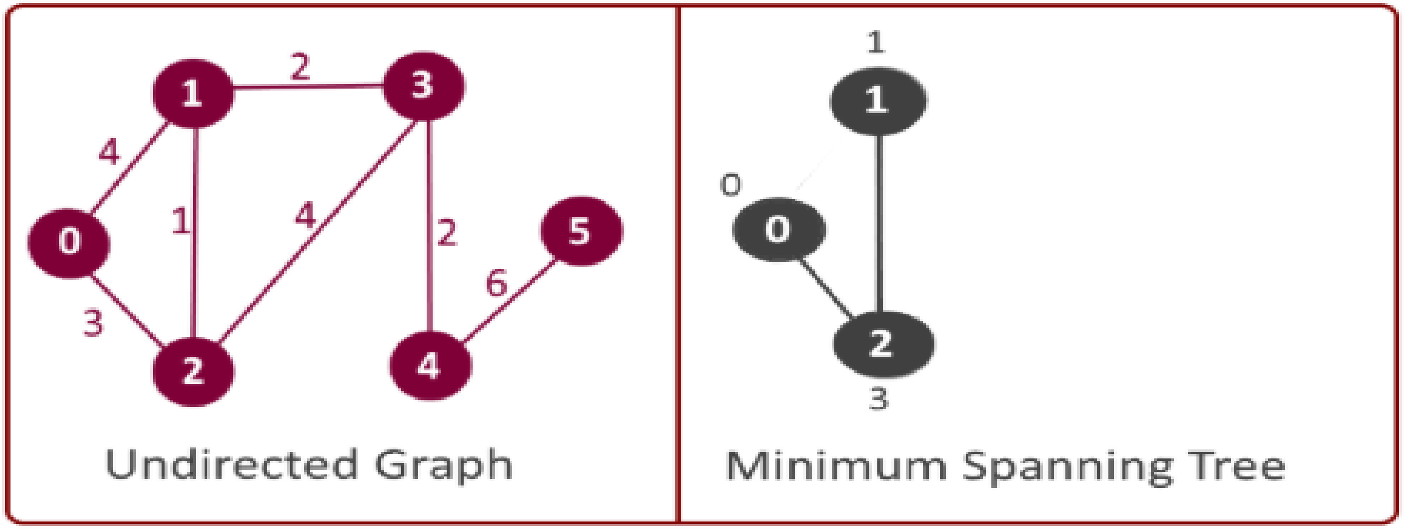
weight<key[1] update the key[1]= 1

For adjacent vertex 3, vertex 3 is not in MST and edge 2-3 weight=4, key[3]=+∞

weight<key[3] update the key[3]= 4

For adjacent vertex 0, - already in MST

Example



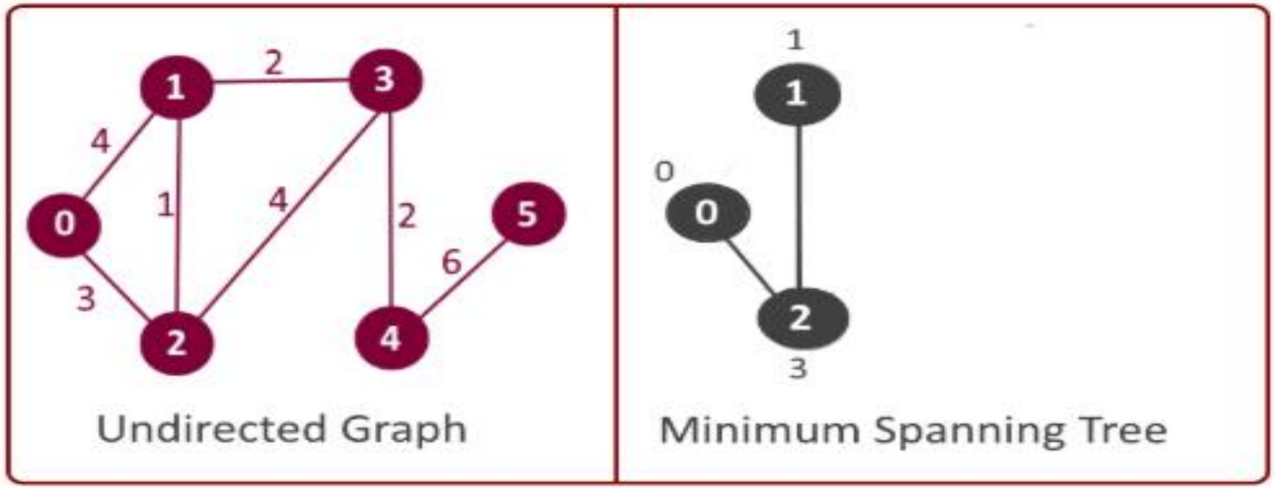
Min Heap	1	3	4	5
	1	4	+∞	+∞

Extract-Min:

Pick the vertex has minimum Key: vertex 1

Include vertex 1 in MST

Example



Undirected Graph

Minimum Spanning Tree

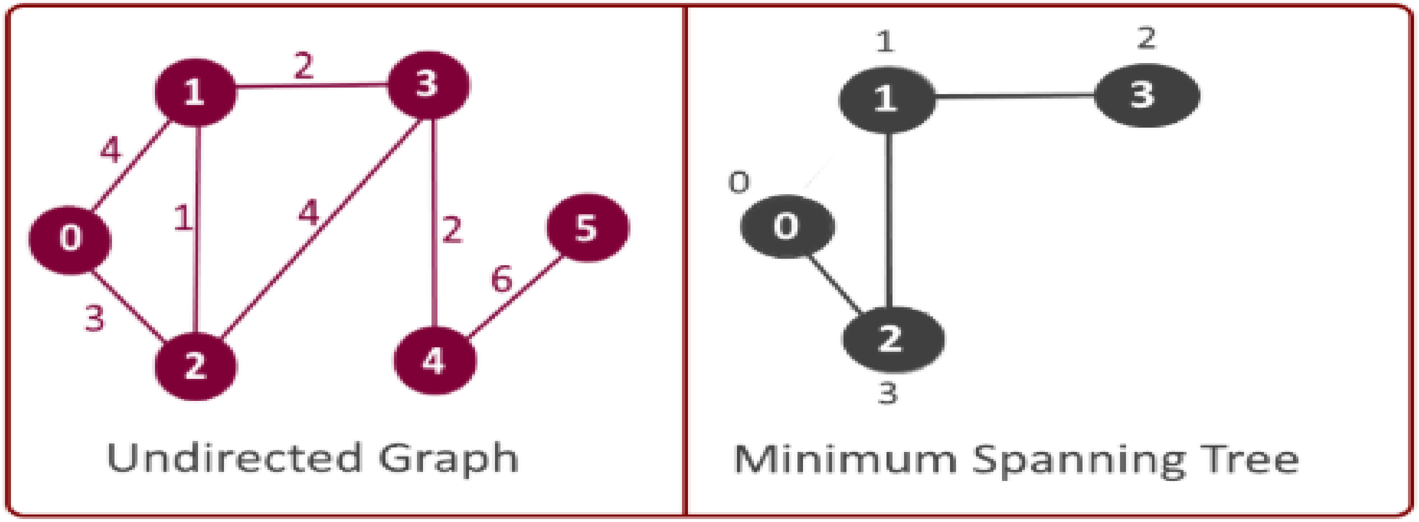
Min Heap	3	4	5
	2	+∞	+∞

Update keys:

Loop over all the adjacent vertices of vertex 1

- For adjacent vertex 2, - already in MST
- For adjacent vertex 3, vertex 3 is not in MST and edge 1-3 weight=2, key[3]=4
  - weight<key[3] update the key[3]= 2
- For adjacent vertex 0, - already in MST

Example



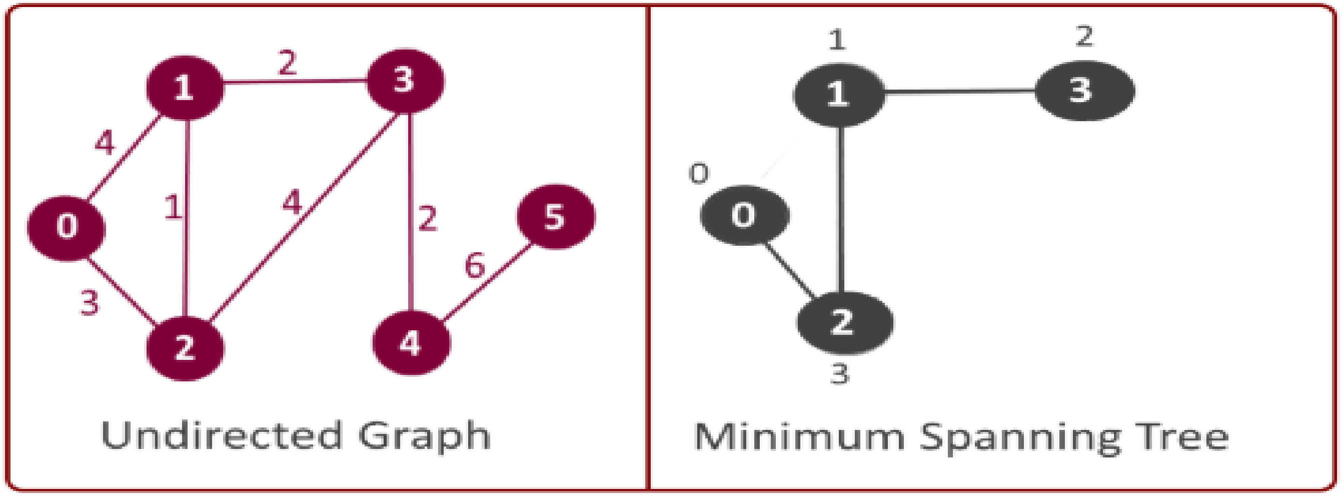
Min Heap	3	4	5
	2	+∞	+∞

Extract-Min:

Pick the vertex has minimum Key: vertex 3

Include vertex 3 in MST

Example



Min Heap	4	5
	2	+∞

Update keys:

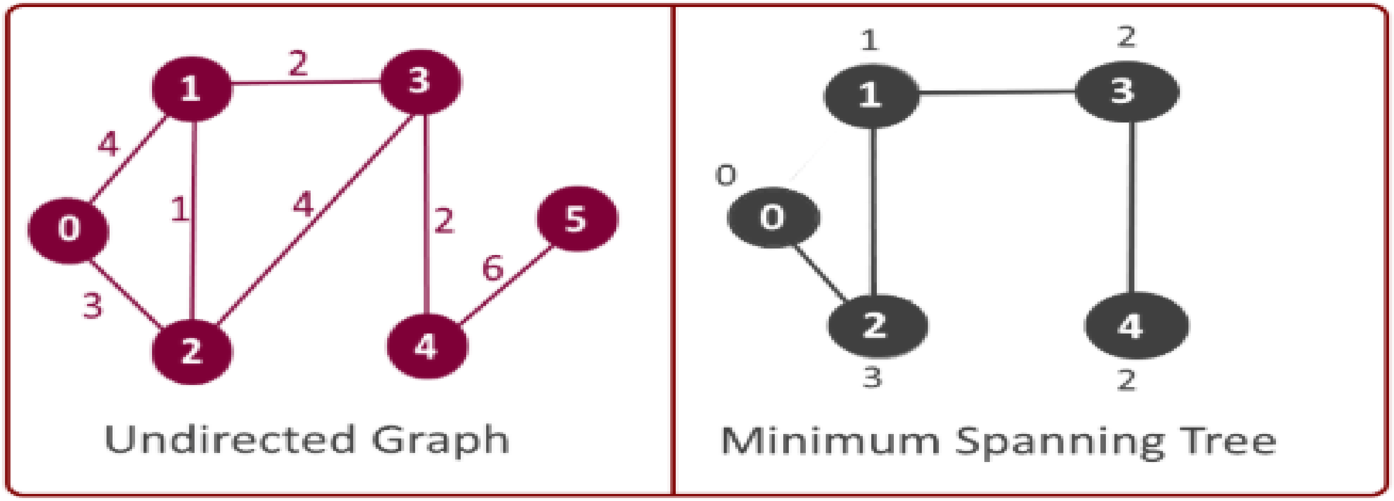
Loop over all the adjacent vertices of vertex 3

For adjacent vertex 2, - already in MST

For adjacent vertex 4, vertex 4 is not in MST and edge 3-4 weight=2, key[4]=+∞
weight<key[4] update the key[4]= 2

For adjacent vertex 1, - already in MST

Example



Undirected Graph

Minimum Spanning Tree

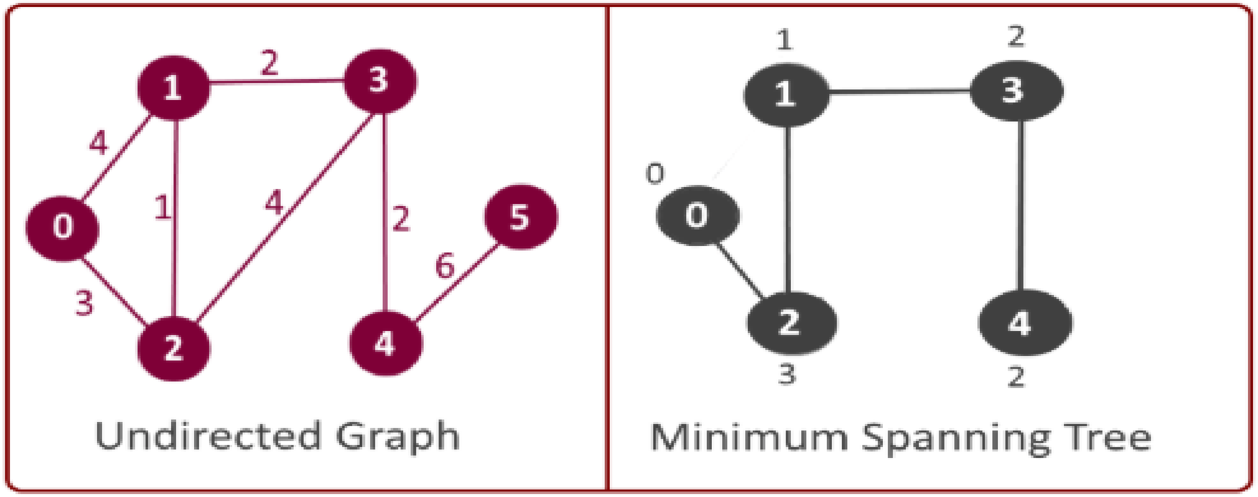
Min Heap	4	5
	2	+∞

Extract-Min:

Pick the vertex has minimum Key: vertex 4

Include vertex 4 in MST

Example



Undirected Graph

Minimum Spanning Tree

Min Heap	5
	6

Update keys:

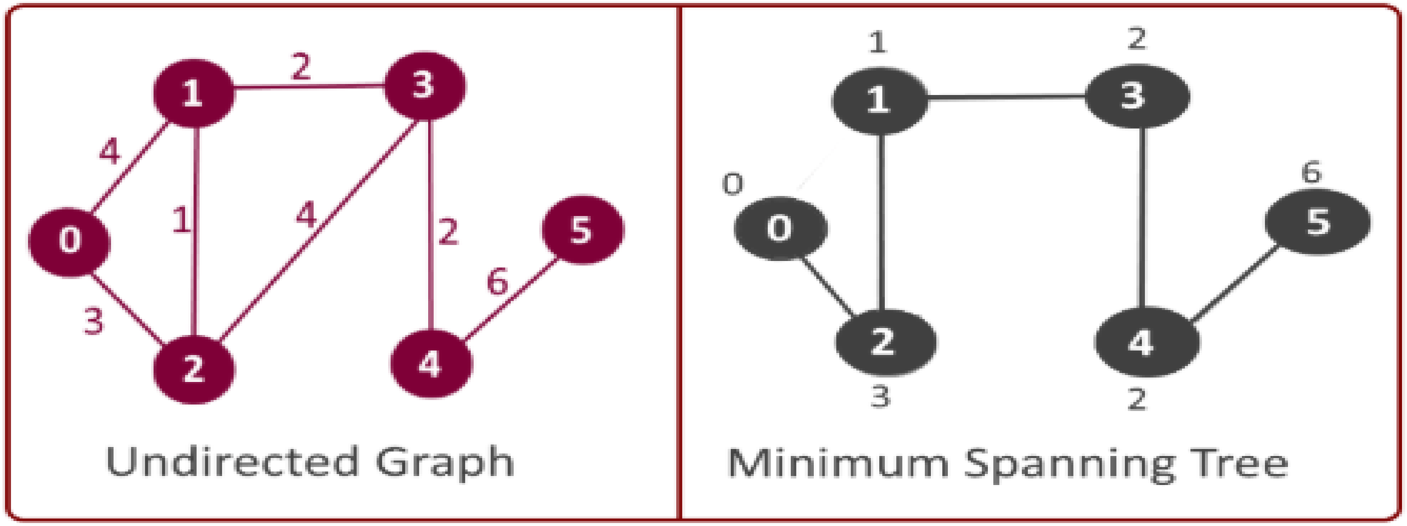
Loop over all the adjacent vertices of vertex 4

For adjacent vertex 5, vertex 5 is not in MST and edge 4-5 weight=6, key[5]=+∞

weight<key[5] update the key[5]= 6

For adjacent vertex 3, - already in MST

Example



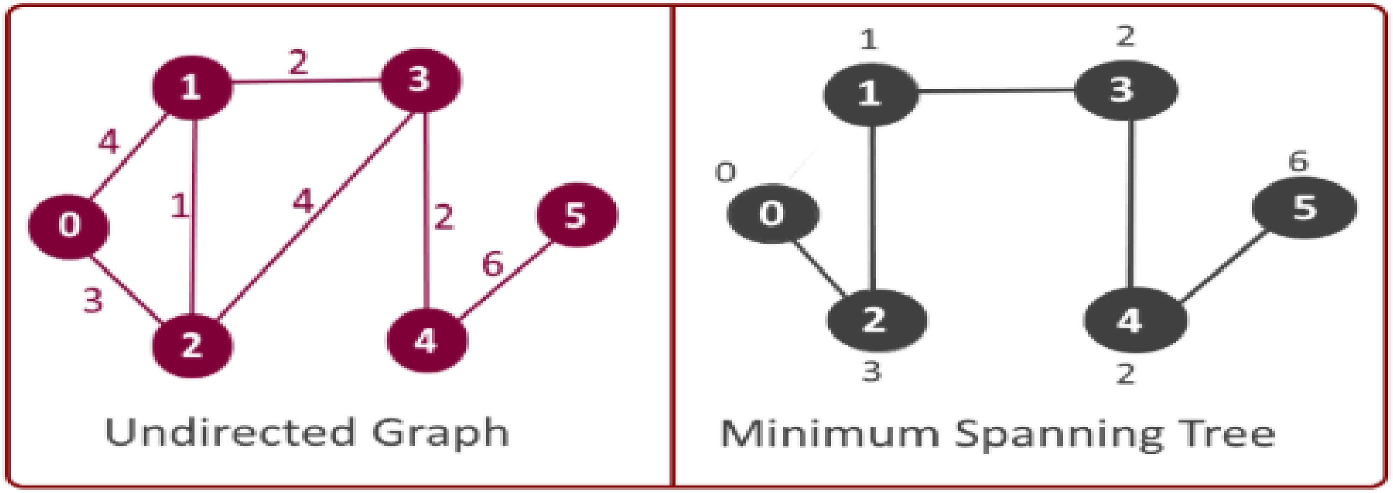
Min Heap	5
	6

Extract-Min:

Pick the vertex has minimum Key: vertex 5

Include vertex 5 in MST

Example



Min Heap NULL

Minimum Spanning Tree is Found
Total Weight : 14

Prim’s Algorithm – MinHeap

```
1) $T = \emptyset$   [invariant: S = vertices spanned by tree-so-far T]
2) for each $u \in V$   O(n)
        $Key[u] = \infty$
3) $Key[s] = 0$ // select any random vertex and make its cost 0
4) Heap Q is initialized with all vertices   O(nlogn)
5) While heap $\neq \emptyset$   O(n)
    •  u = ExtractMin from Heap   O(logn)
    •  Add u to S
    •  for each $v \in adj[u]$
            DecreaseKey(Q, v, cost[u,v])
```

O(mlogn) time for all iterations of outer While loop

total running time = n + nlogn + n (logn) + mlogn = O(mlogn)

Lemma 1 Proof

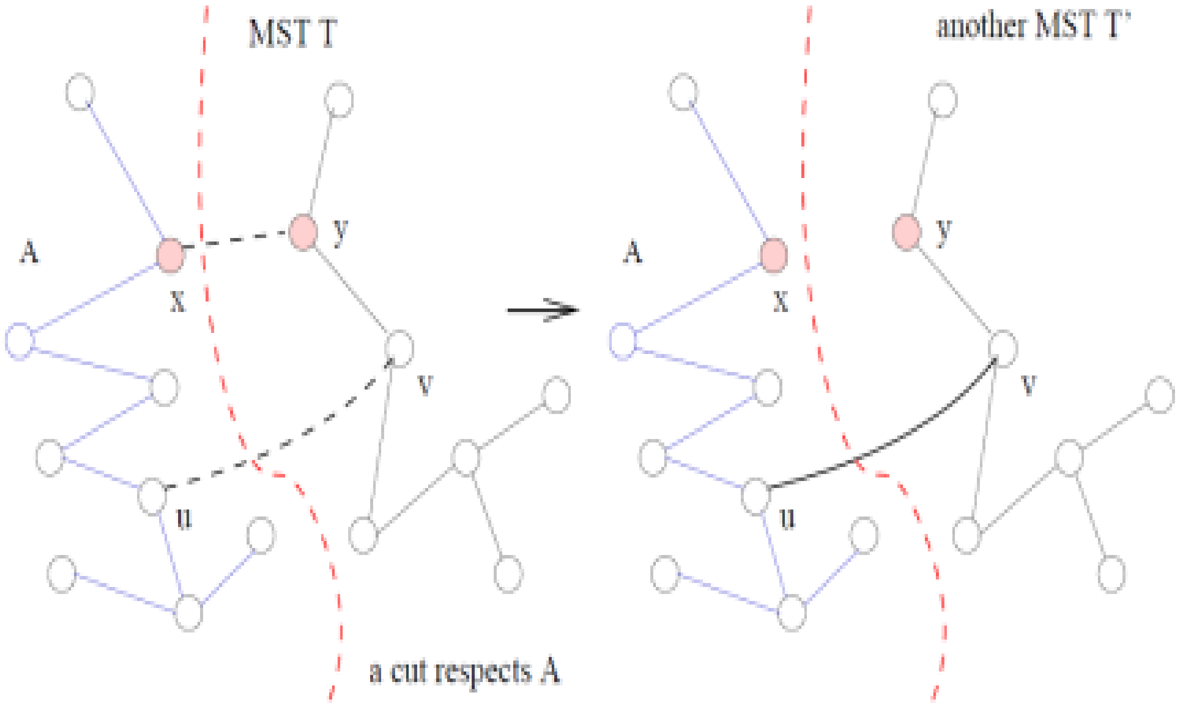
- Lemma 1: Let G(V, E) be a connected, weighted, undirected graph. Let A be a subset of E that is included in some MST of G. Also let (S, V\S) be any cut of G that respects A and (u, v) be a light edge crossing the cut (S, V\S) then (u, v) is a safe edge.

- Proof: By contradiction!!
- Let A is a subset of T where T is a MST but $(u, v) \notin T$
- Since u and v are on opposite side of the cut, there must be at least one edge on the path from u to v in T that crosses the cut.
- Let (x, y) be that edge
- Since cut respects A, $(x, y) \notin A$
- Also (u, v) is light edge so w(u,v) <= w(x,y)

Lemma 1 Proof

- Adding (u,v) to T, will create a cycle now.
- By removing (x,y) we create a new tree T’.
- W(T’) = W(T)-w(x,y)+w(u,v)
- W(T’)<=W(T)

Contradiction!!



Slide Credits

- COMP 3711H Design and Analysis of AlgorithmsFall 2016
- https://algorithms.tutorialhorizon.com/prims-minimum-spanning-tree-mst-using-adjacency-list-and-min-heap/