Minimum Spanning Tree

Outline

- Spanning trees and minimum spanning trees
- Prim’s algorithm for the MST problem.
  - The idea
  - The algorithm
  - Analysis

Overview

- Informal Goal: Connect a bunch of points together as cheaply as possible.
- Applications: Clustering, networking.

- Blazingly Fast Greedy Algorithms:
    - Prim's Algorithm [1957]
    - Kruskal's algorithm [1956]

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

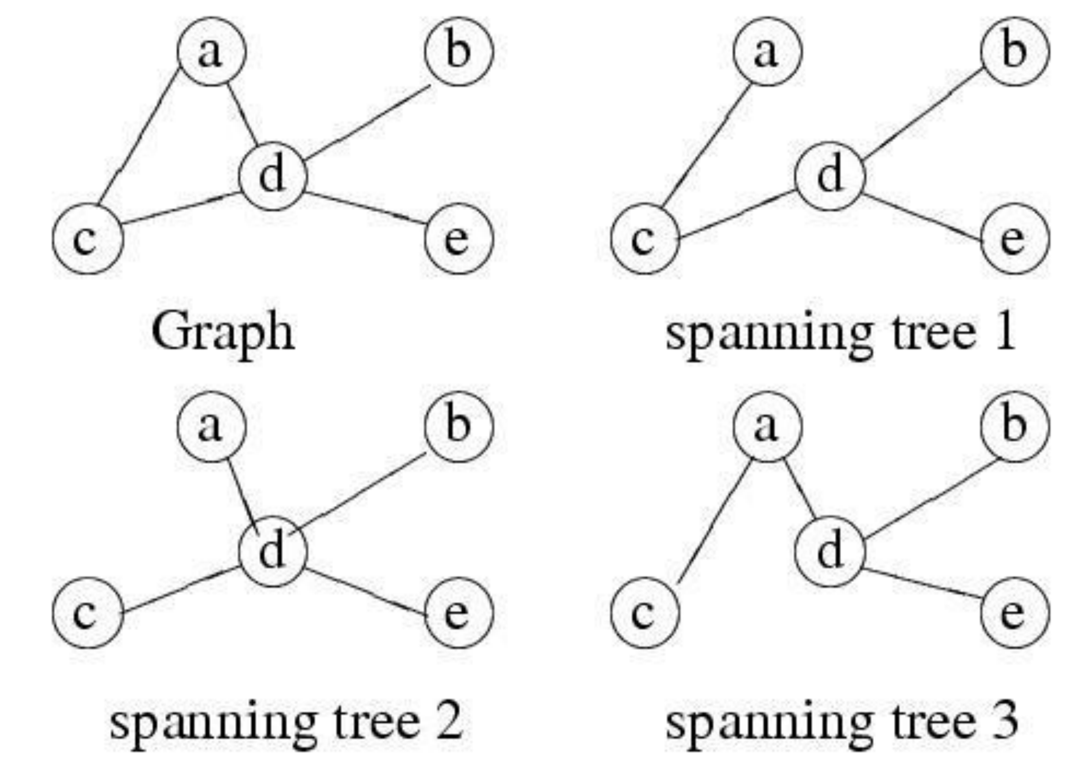
A tree is an acyclic graph

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



Spanning Tree

- How many edges are in a spanning tree of Graph with n vertices?

a) n^2
b) n-1
c) n
d) n choose 2

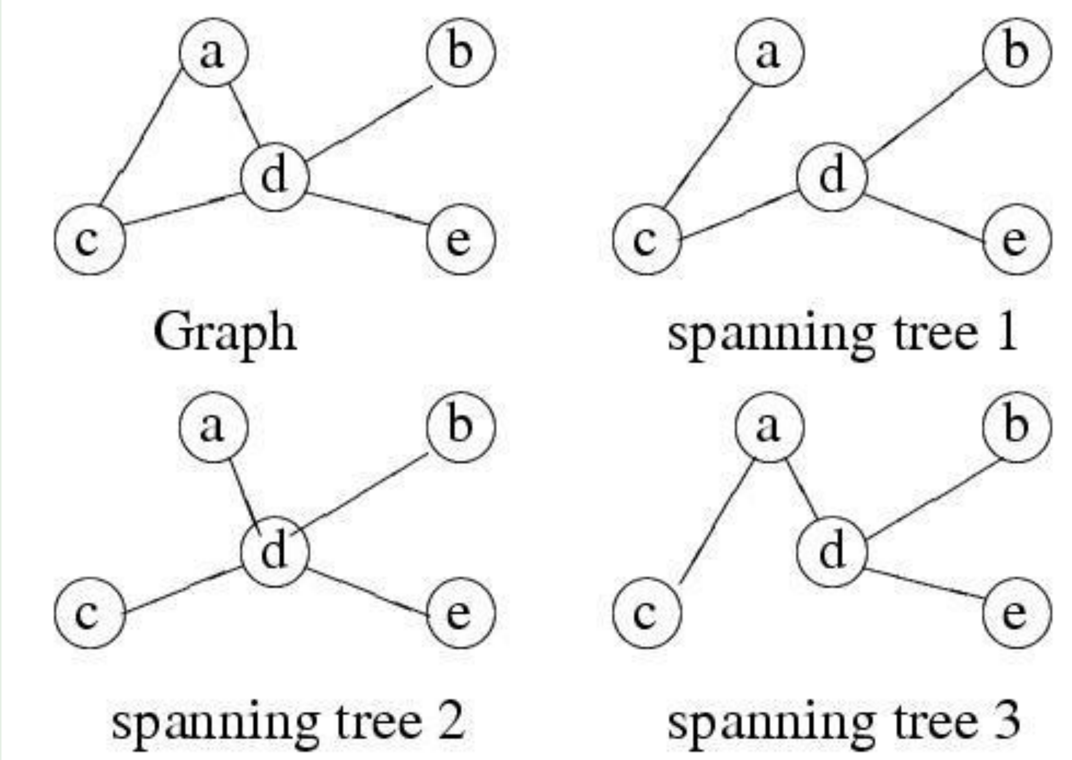
Spanning Tree

- How many edges are in a spanning tree of Graph with n vertices?

a) n^2
b) n-1
c) n
d) n choose 2

- Correct answer: option b (spanning tree has exactly n-1 edges)

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



Theorem

Every connected graph has a spanning tree.

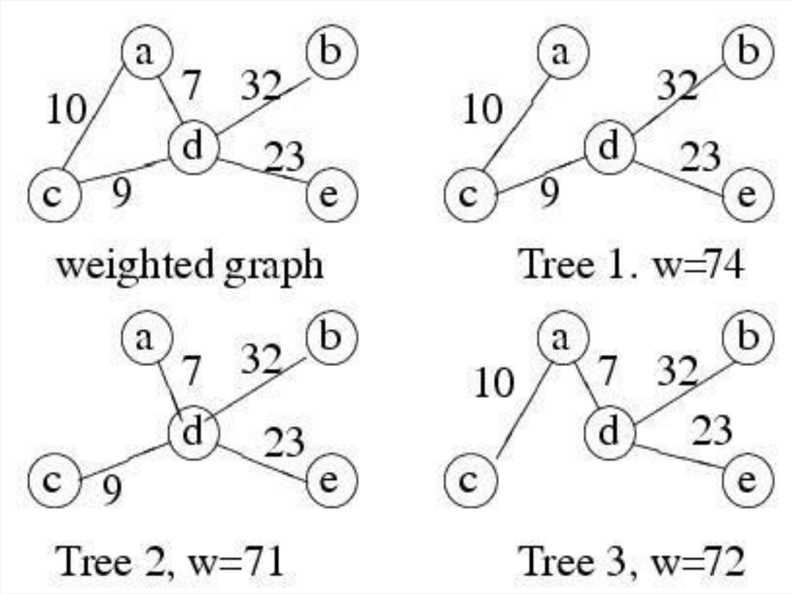
Definition

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

Weighted Graphs

Definition

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

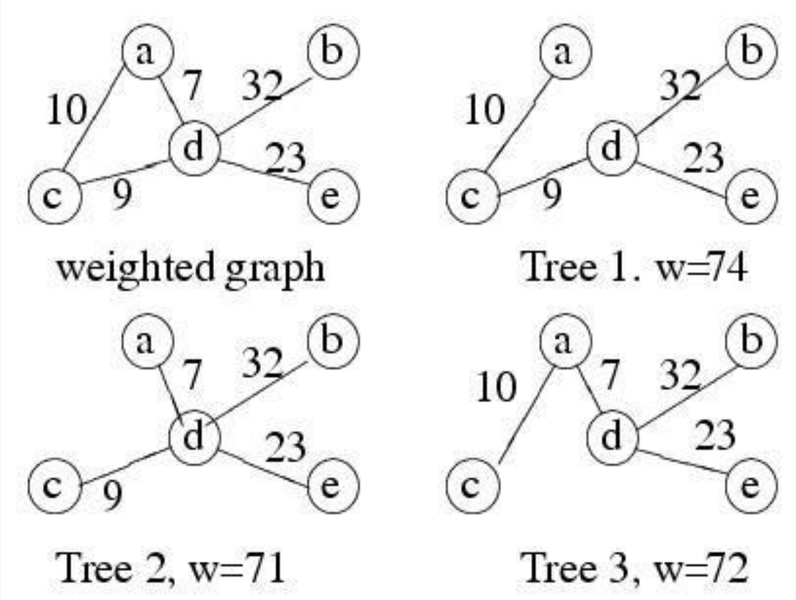


Weighted Graphs

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

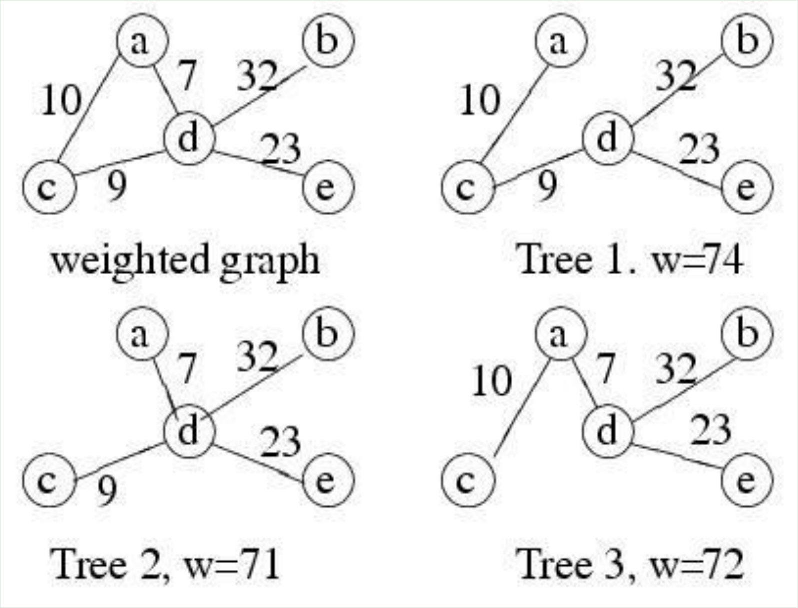
A Minimum spanning tree (MST) of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

Minimum Spanning Trees

Definition

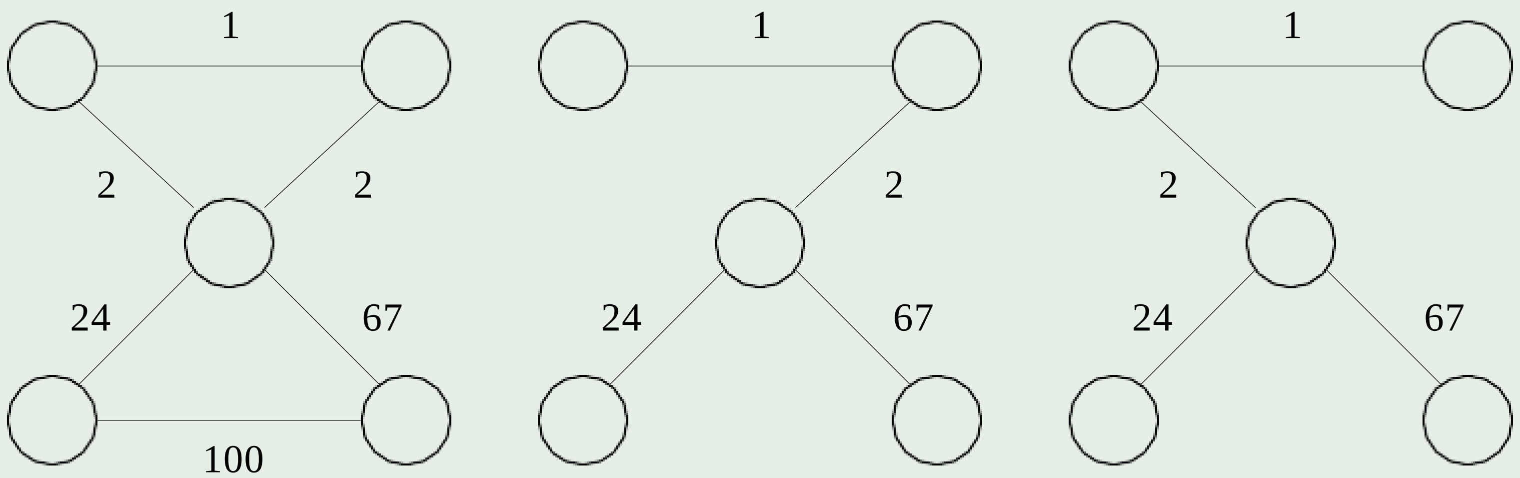
A Minimum spanning tree (MST) of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

Example



Remark

The minimum spanning tree may not be unique

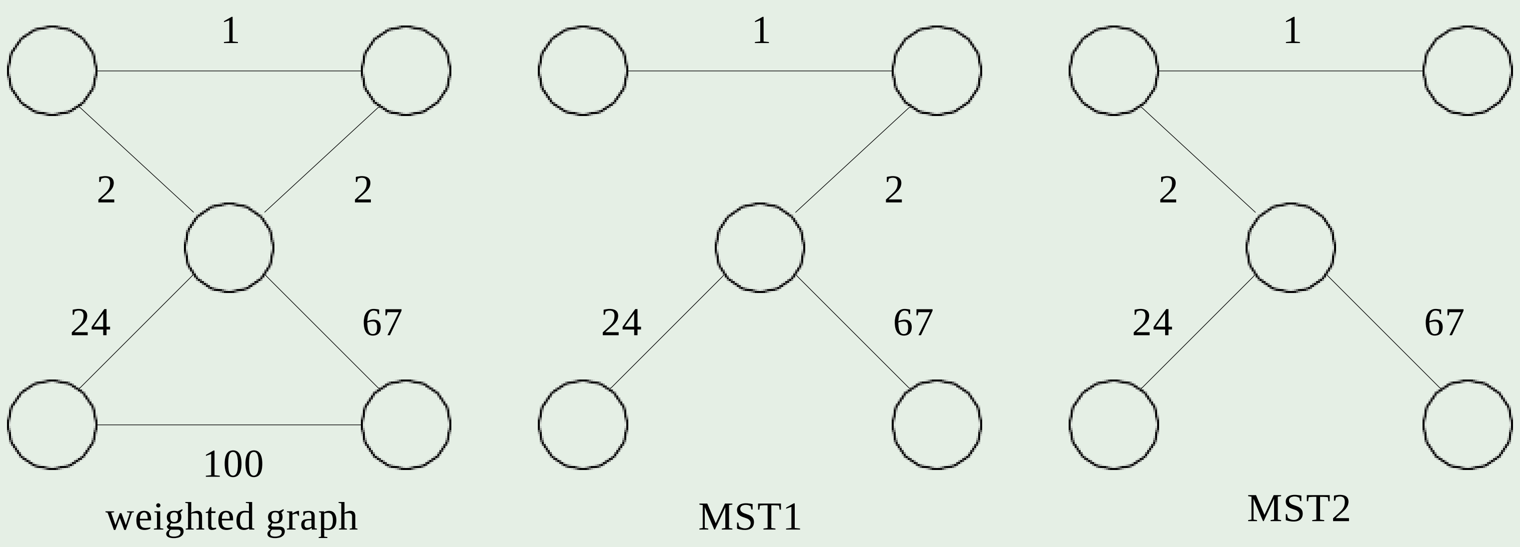


weighted graph MST1 MST2

Remark

The minimum spanning tree may not be unique

Example



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

Definition (MST Problem)

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

vertices   edges

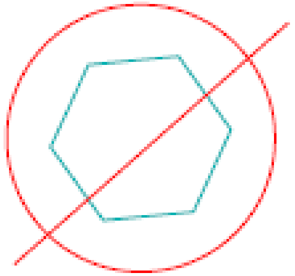
Input: Undirected graph $G = (V, E)$ and a cost c_e for each edge $e \in E$.

- Assume adjacency list representation
- OK if edge costs are negative

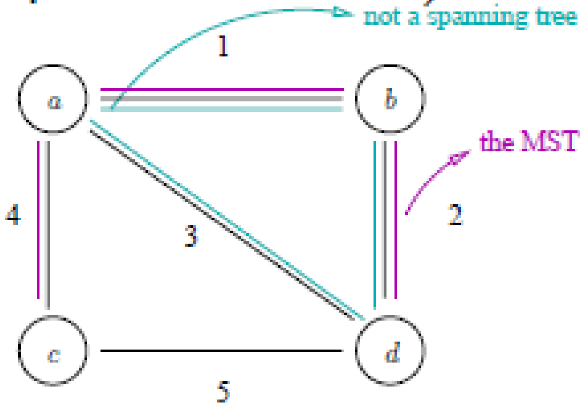
Output: minimum cost tree $T \subseteq E$ that spans all vertices.

i.e., sum of edge costs

I.e.: (1) T has no cycles, (2) the subgraph (V, T) is connected (i.e., contains path between each pair of vertices).



(disallowed)



Standing Assumptions

Assumption #1: Input graph G is connected.

- Else no spanning trees.
- Easy to check in preprocessing (e.g., depth-first search).

Assumption #2: Edge costs are distinct.

- Prim + Kruskal remain correct with ties (which can be broken arbitrarily).
- Correctness proof a bit more annoying (will skip).

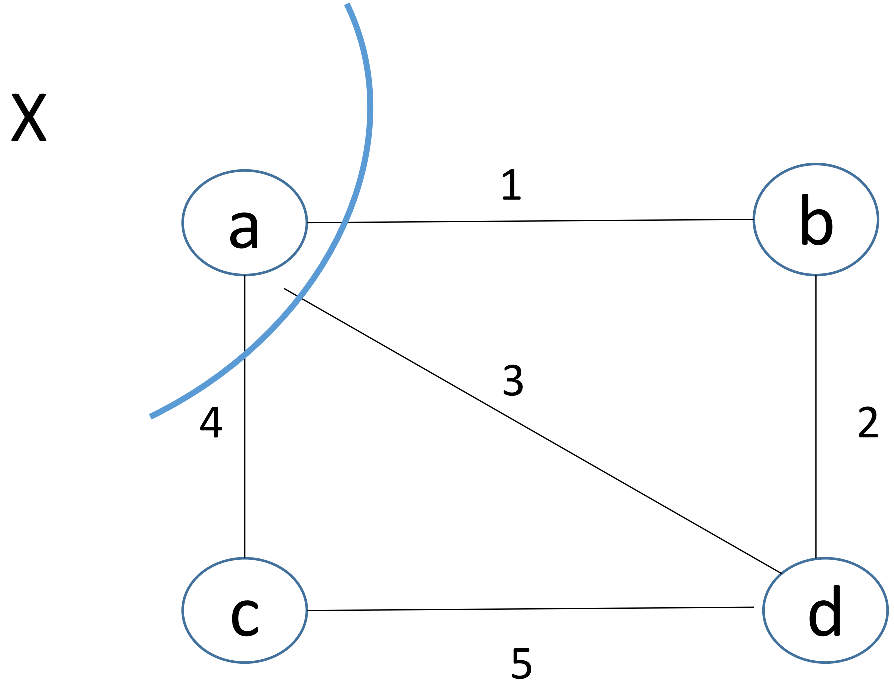
Prim’s Algorithm

Prim's Algorithm (create two sets of vertices X and V)

Initialize $X = \{s\}$ $\{s \in V$ chosen arbitrarily$\}$

- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
    - Let $e = (u; v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
    - Add e to T
    - Add v to X.

Prim’s Algorithm Dry Run with sets X and V



Initialization:

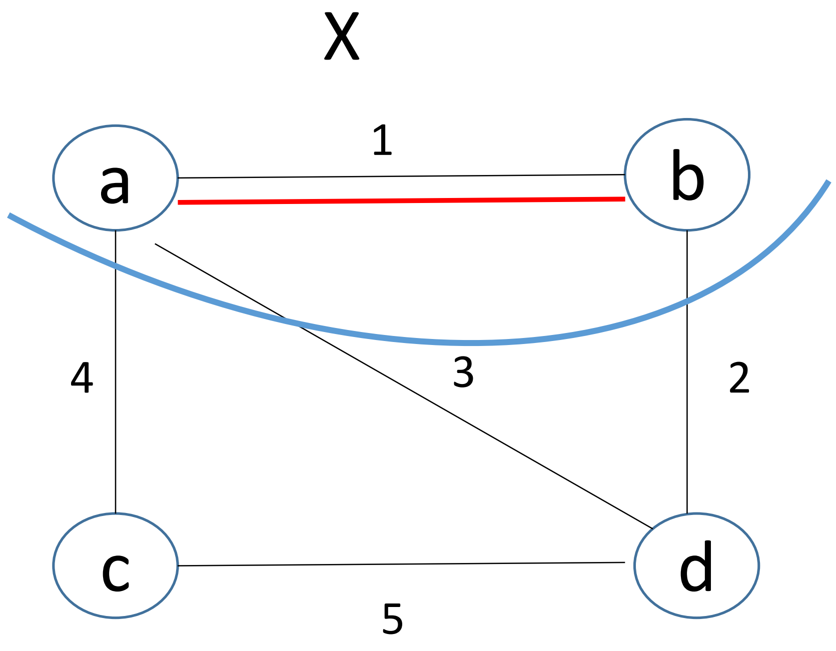
X = {a} ,   T {}

V = {a, b , c , d}

Initialize X = {s} {s ∈ V chosen arbitrarily}

- T = Ø [invariant: X = vertices spanned by tree-so-far T]

Prim's Algorithm Dry Run with sets X and V



Initialization:

X = {a} , T {}

V = {a, b , c , d}

First Iteration:

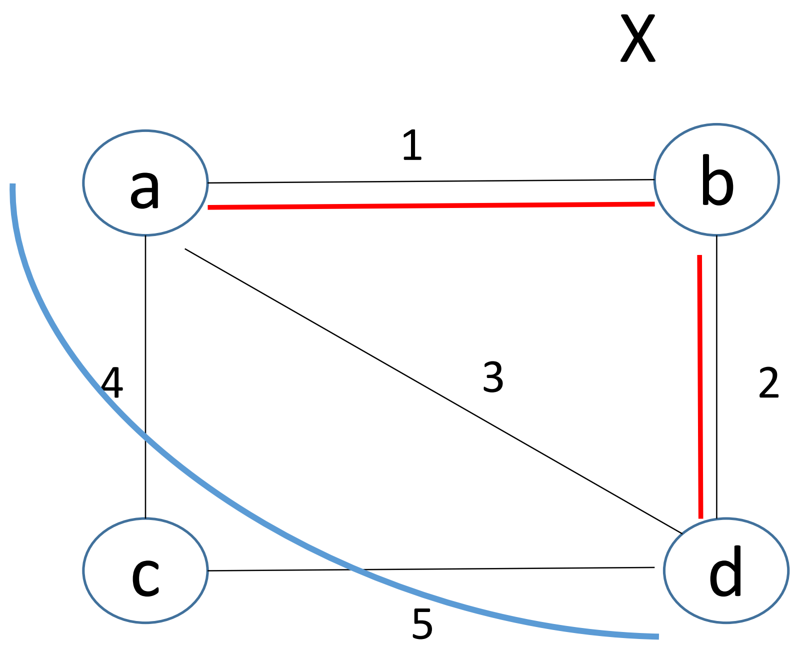
X = {a, b}, T = {1}

V = {a, b , c , d}

While X ≠ V

- Let e = (u; v) be the cheapest edge of G with u ∈ X, v ∉ X.
- Add e to T
- Add v to X.

Prim’s Algorithm Dry Run with sets X and V



Initialization:

X = {a} ,   T {}

V = {a, b , c , d}

First Iteration:

X = {a, b},   T = {1}

V = {a, b , c , d}

Second Iteration:

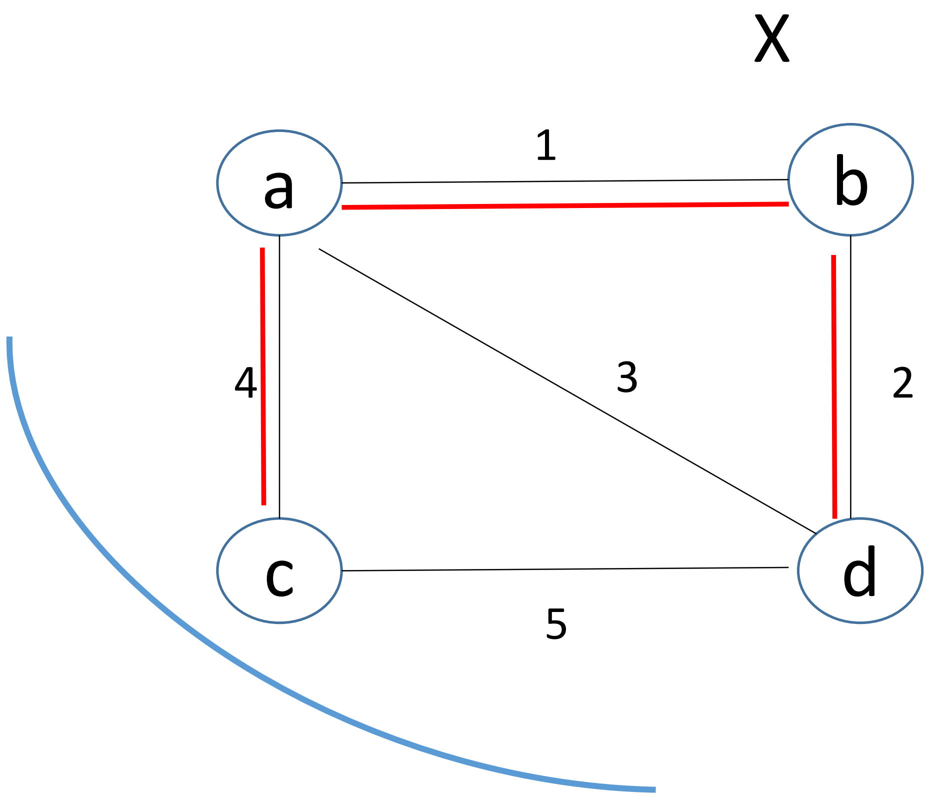
X = {a, b, d},   T = {1, 2}

V = {a, b , c , d}

While X ≠ V

- Let e = (u; v) be the cheapest edge of G with u ∈ X, v ∉ X.
- Add e to T
- Add v to X.

Prim's Algorithm Dry Run with sets X and V



Initialization:

X = {a} ,   T {}

V = {a, b , c , d}

First Iteration:

X = {a, b},   T = {1}

V = {a, b , c , d}

Second Iteration:

X = {a, b, d},   T = {1, 2}

V = {a, b , c , d}

Third Iteration:

X = {a, b, d, c},   T = {1, 2, 4}

V = {a, b , c , d}

While X ≠ V

- Let e = (u; v) be the cheapest edge of G with u ∈ X, v ∉ X.
- Add e to T
- Add v to X.

Prim's Algorithm (create two sets of vertices X and V)

Initialize X = {s} {s ∈ V chosen arbitrarily}

- T = Ø [invariant: X = vertices spanned by tree-so-far T]
- While X ≠ V
  - Let e = (u; v) be the cheapest edge of G with u ∈ X, v ∉ X.
  - Add e to T
  - Add v to X.

What is time complexity?

O(mn) since while loop runs n times and in each iteration m operations to find cheapest edge

Prim’s Algorithm
Efficient Version

Prim's Algorithm (create two sets of vertices X and V)

Initialize $X = \{s\}$ $\{s \in V$ chosen arbitrarily$\}$

- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u; v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T
  - Add v to X.

- Running time of straightforward implementation:
- - $O(n)$ iterations [where n = # of vertices]
- - $O(m)$ time per iteration [where m = # of edges]
- $O(mn)$ time
- BUT CAN WE DO BETTER?

Can we do better than O(mn) in Prims Algorithm?

- We are doing repeated minimum computations, select minimum edge
- Which data structure comes to your mind when you need to do repeated minimum computations?

- **MinHeap**

What should be stored in heap?

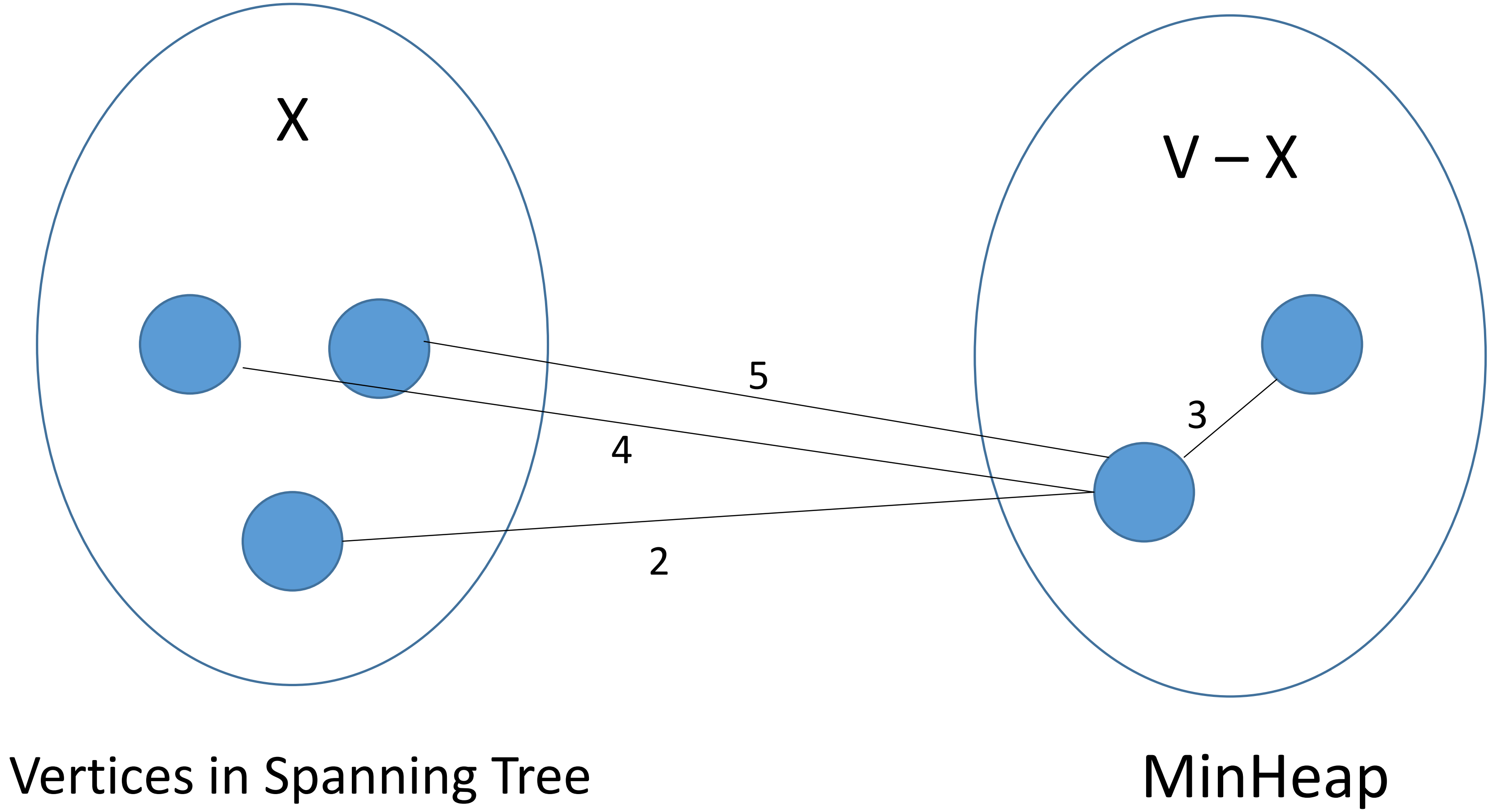
- We can use edge costs as keys and store edges in heap.
- Running time will be O(m lg n)
- Note that m is at most n^2 so lg m = lg n^2 = 2lgn = O(lg n)

Prim’s Algorithm (MinHeap)

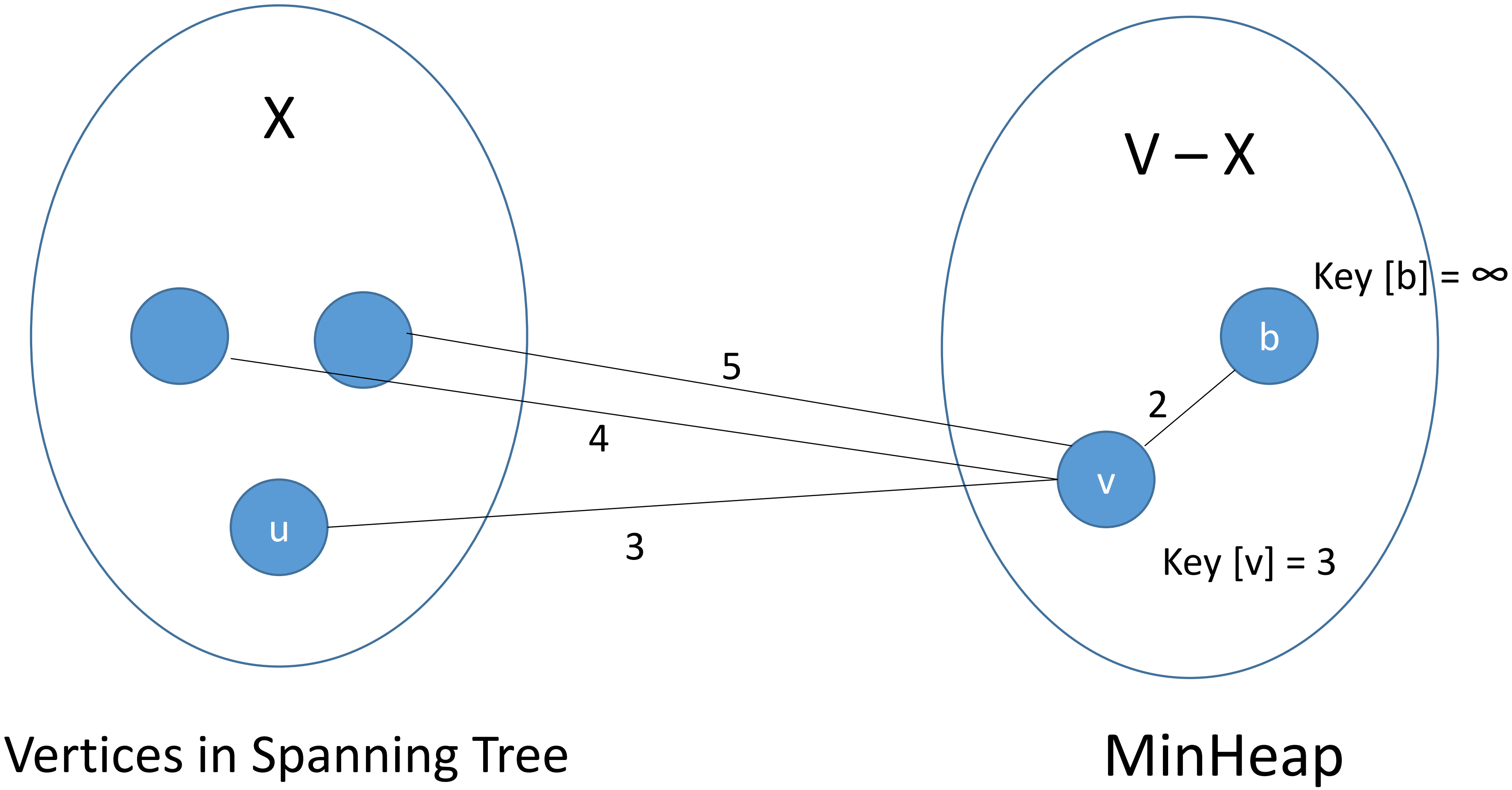
- A more efficient implementation will be using vertices in heap instead of edges. (it will give asymptotically same time but better constants since number of vertices is less or equal to number of edges)

- But what should be key of vertices in heap ?

Prim's Algorithm with MinHeap
(which set of vertices should be stored in MinHeap?)



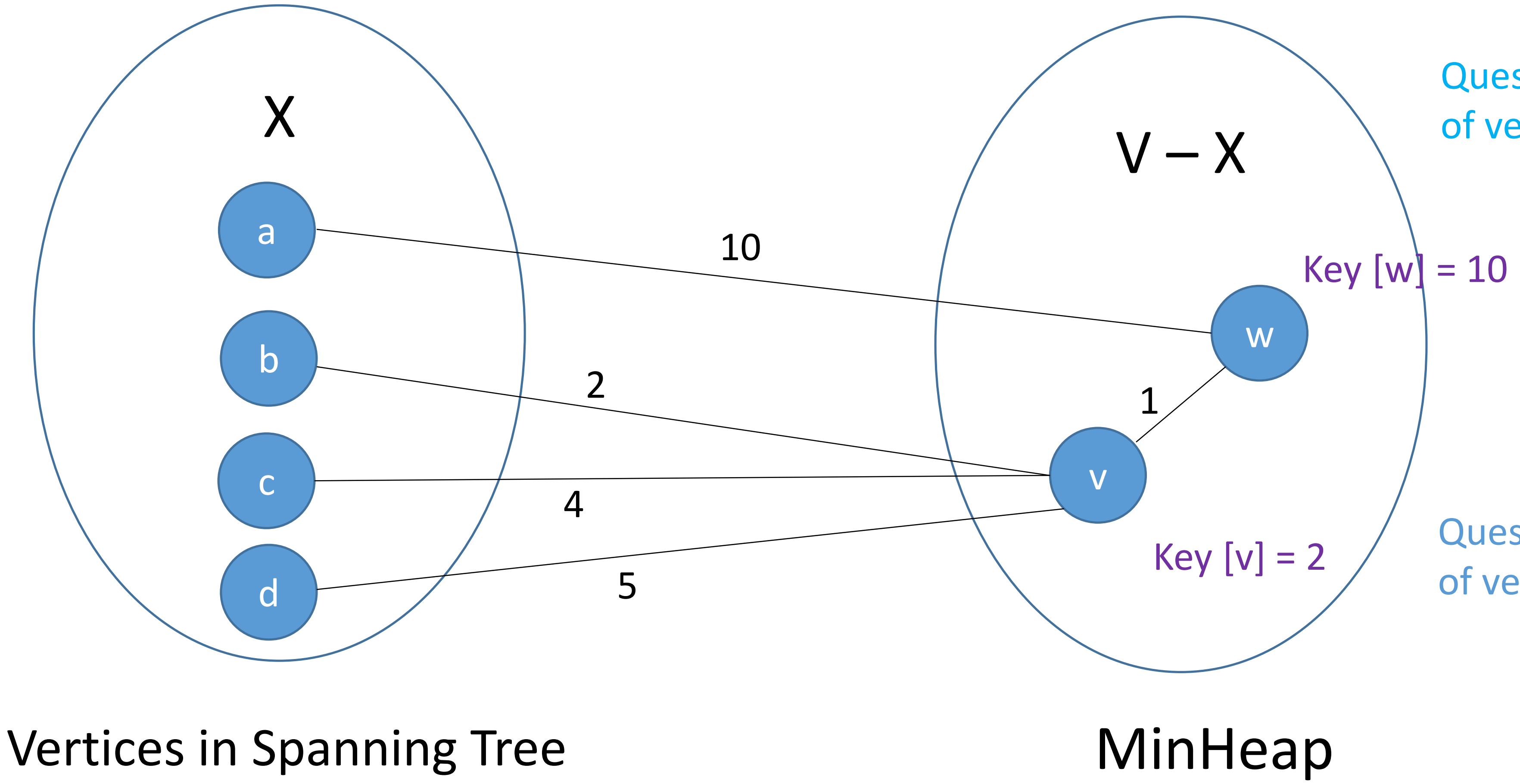
Prim's Algorithm with MinHeap
(What should be key of vertices in MinHeap?)



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

Prim’s Algorithm with MinHeap

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



Question 1: What will be key of vertex w ?

Question 2: What will be key of vertex v ?

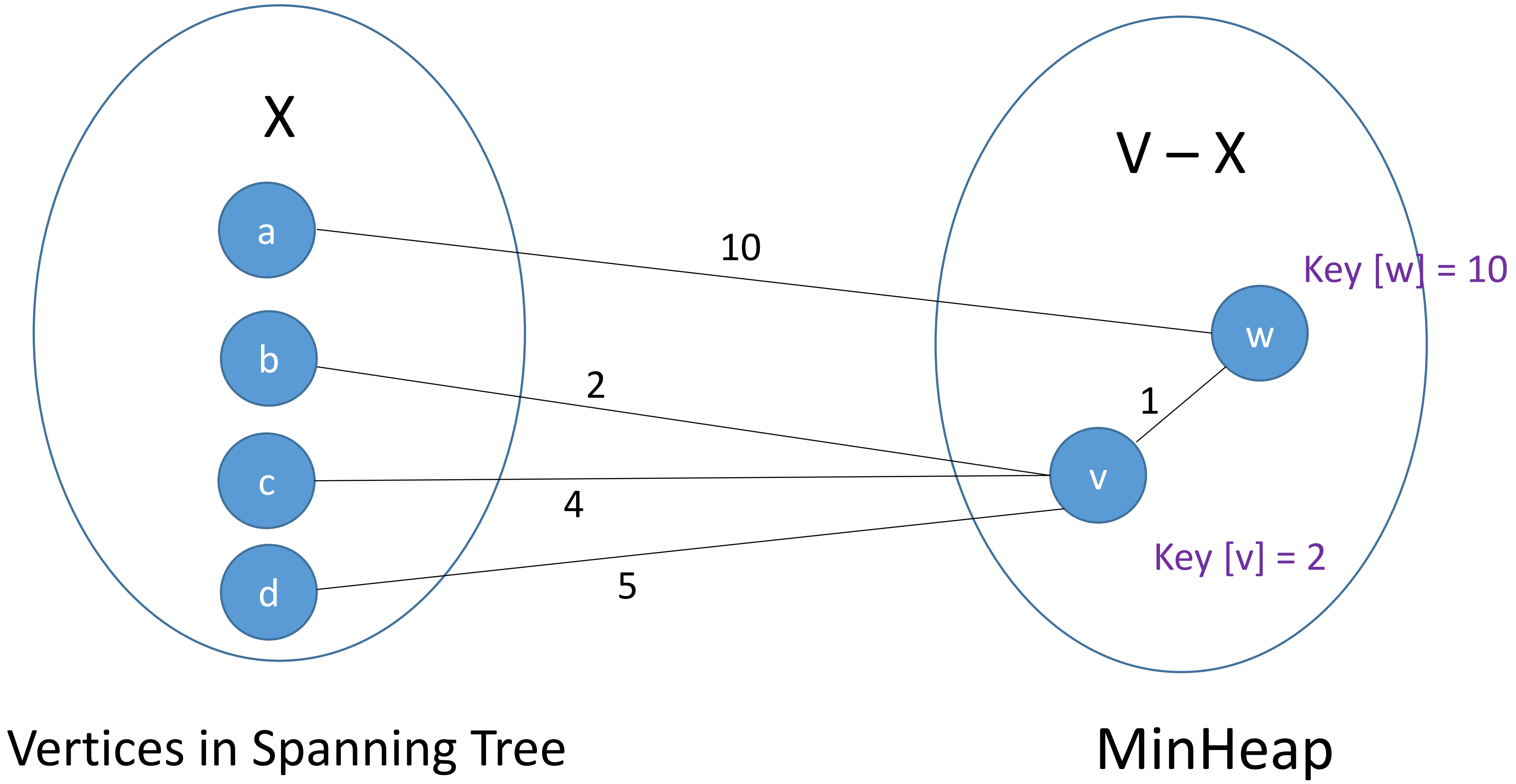
Prim’s Algorithm (MinHeap with vertices)

- T = Ø [invariant: X = vertices spanned by tree-so-far T]
- Initialize Heap Q with vertices
- While X ≠ V
    - ExtractMin from Heap Q to get vertex v with cheapest cost edge e
    - Add e to T
    - Add v to X.

Incomplete Algorithm

Prim's Algorithm with MinHeap

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

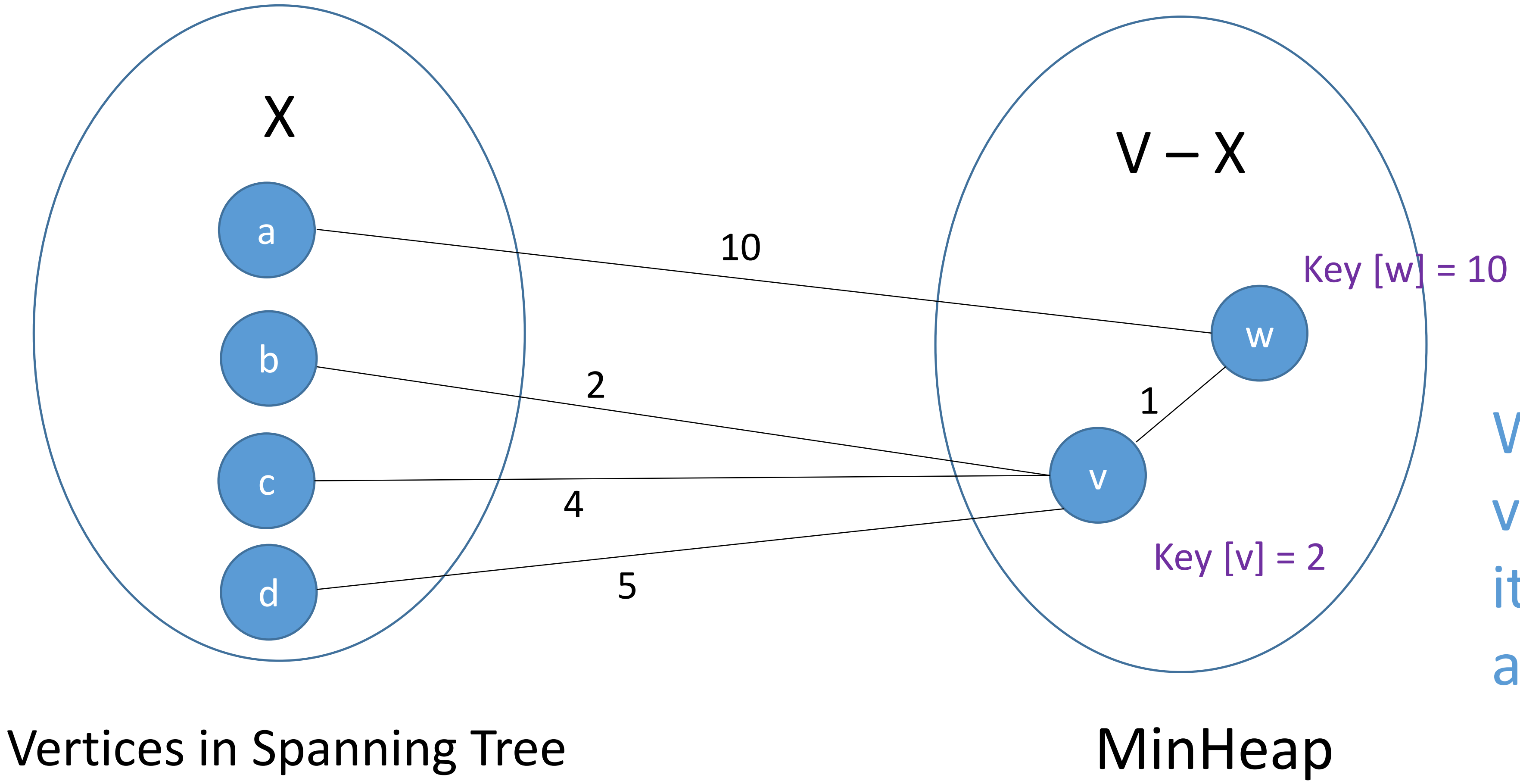


Which vertex will be selected to be added to set X in next iteration of Prims algorithm ?

Vertex v

Prim's Algorithm with MinHeap

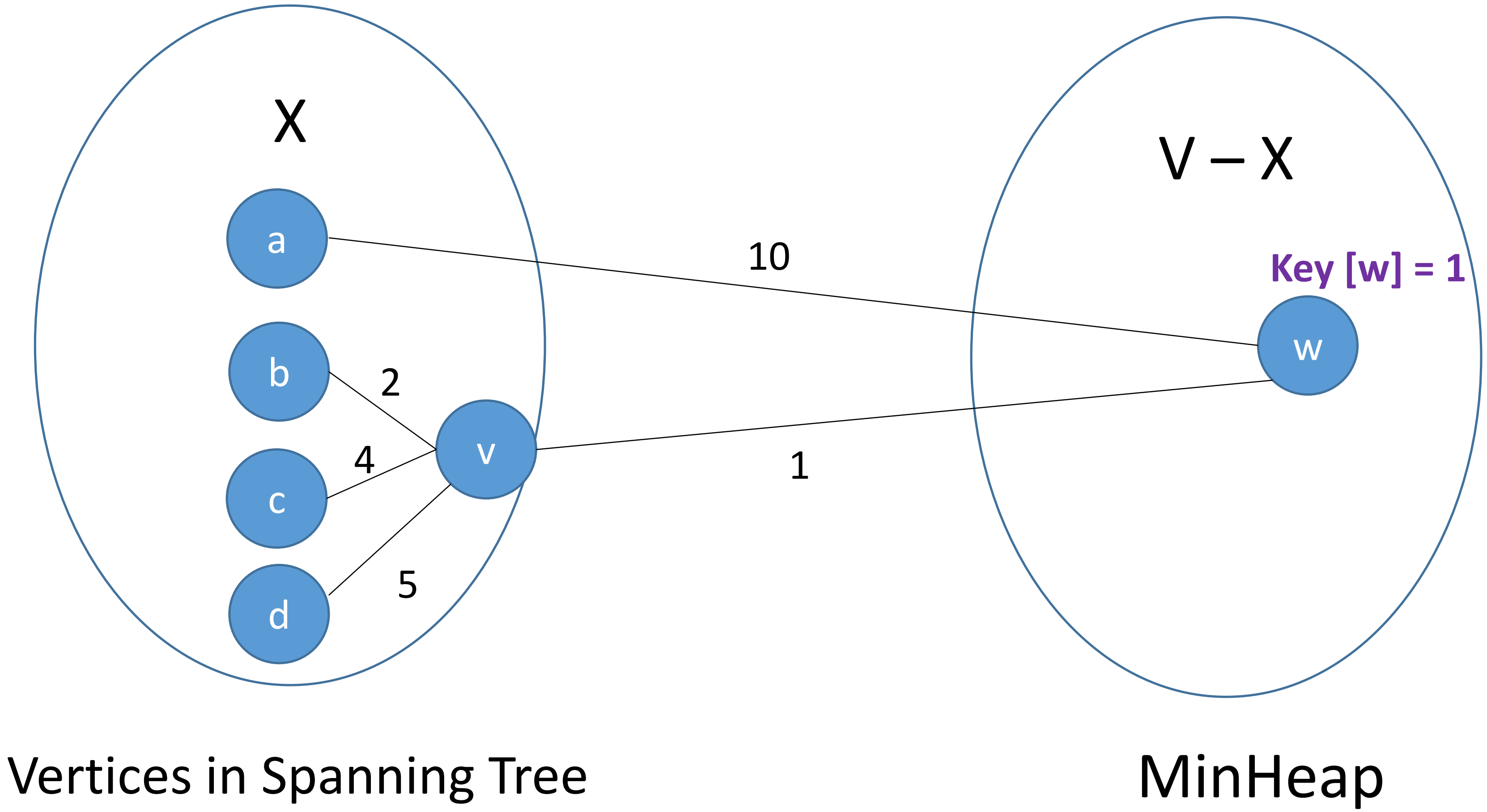
For each vertex v in MinHeap, the key will be cheapest edge (u,v) such that u ∈ X



What will be key of vertex w after one iteration of Prims algorithm ?

Prim’s Algorithm with MinHeap

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



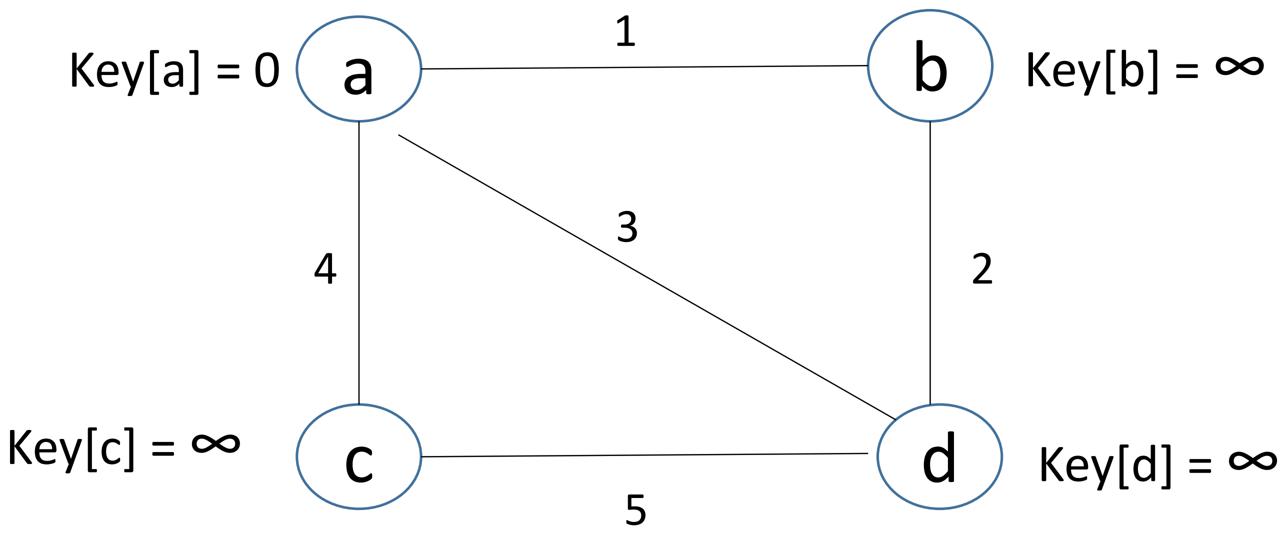
What will be key of vertex w after one iteration of Prims algorithm ?

- So we need to re-compute keys after each extractMin()

Prim's Algorithm (MinHeap with vertices)

- T = $\emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- for each u $\in$ V
    - Key[u] = ∞
- Key[s] = 0 // select any random vertex and make its cost 0
- Heap Q is initialized with all vertices
- While heap $\neq \emptyset$
    - u = ExtractMin from Heap
    - Add u to X
    - 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]

Prim’s Algorithm Dry Run MinHeap

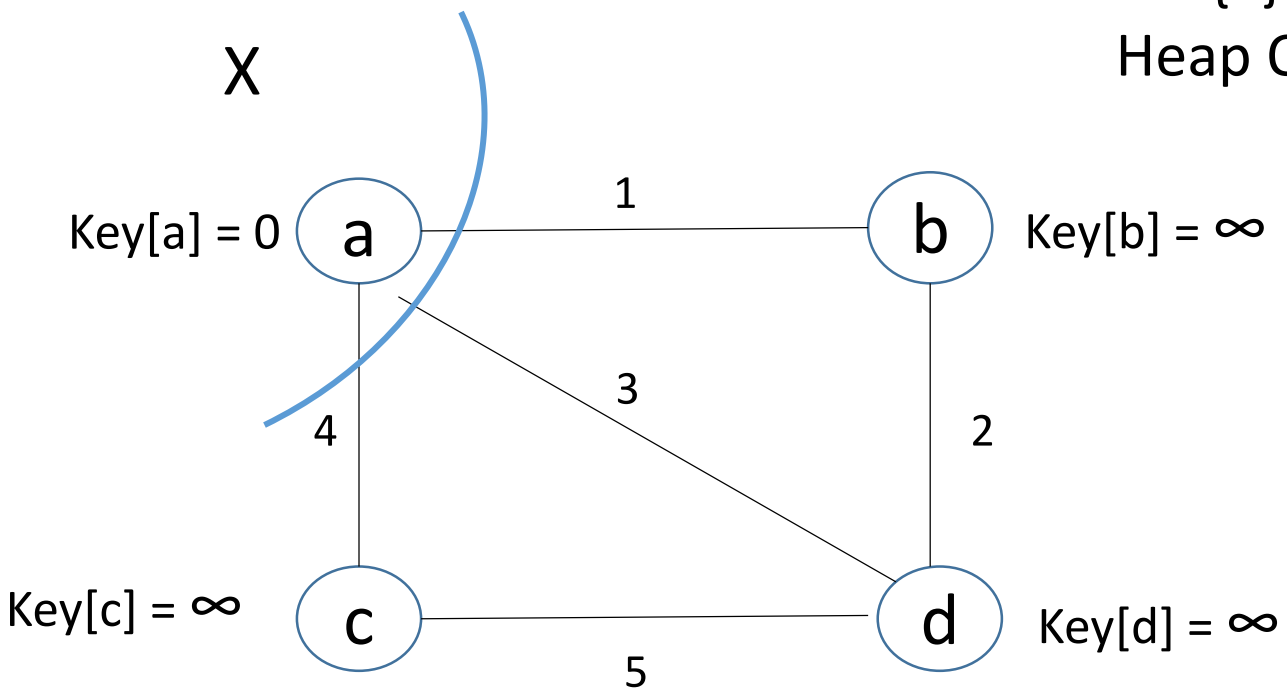


Initialization:

X = {} ,

Heap Q = {a[0], b[∞] , c[∞] , d[∞]}

Prim’s Algorithm Dry Run MinHeap



First Iteration:

X = {a} ,

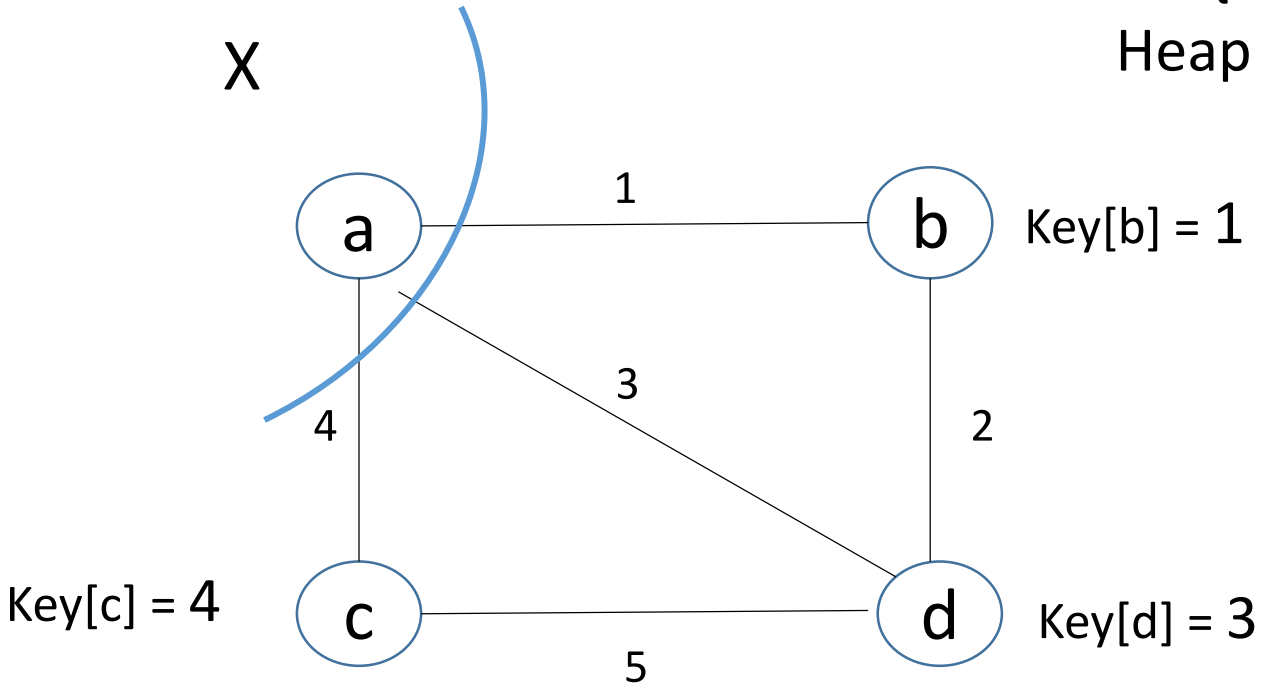
Heap Q = { b[∞] , c[∞] , d[∞]}

Prim’s Algorithm Dry Run MinHeap

First Iteration:

X = {a} ,

Heap Q = { b[1] , c[4] , d[3]}



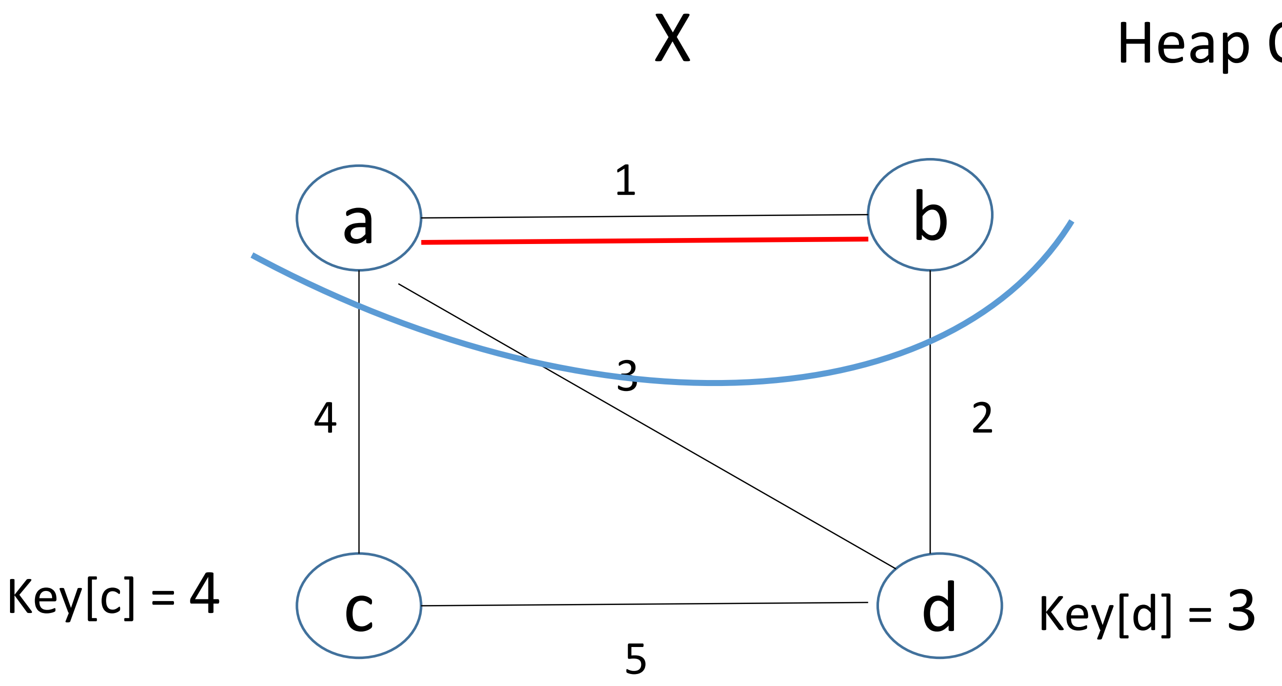
Keys are updated

Prim’s Algorithm Dry Run MinHeap

Second Iteration:

X = {a , b} ,

Heap Q = { c[4] , d[3]}



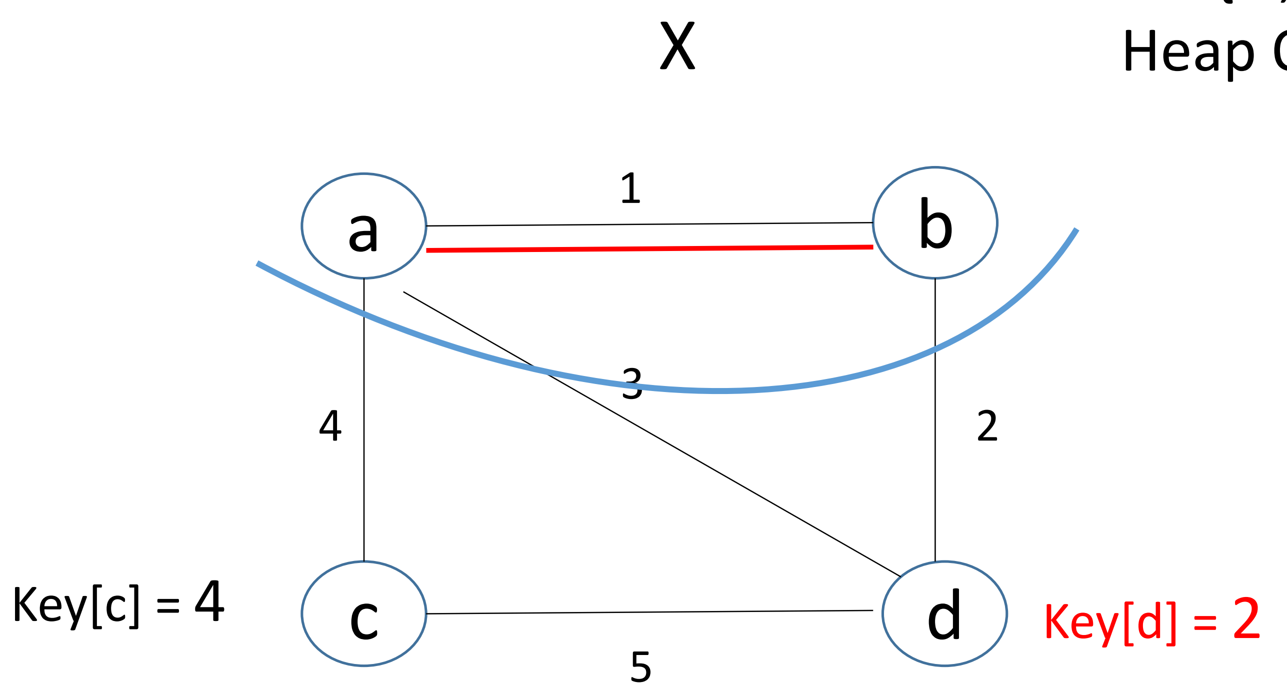
Vertex b is extracted with least cost from heap

Prim's Algorithm Dry Run MinHeap

Second Iteration:

X = {a , b} ,

Heap Q = { c[4] , d[2]}



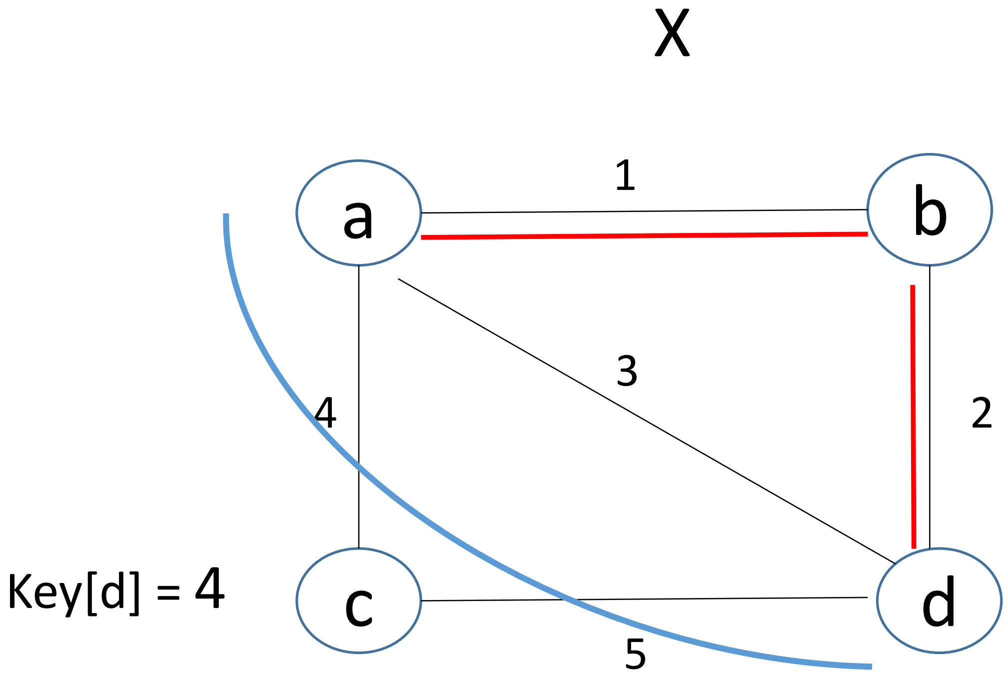
Keys are updated

Prim’s Algorithm Dry Run MinHeap

Third Iteration:

X = {a , b, d} ,

Heap Q = { c[4]}



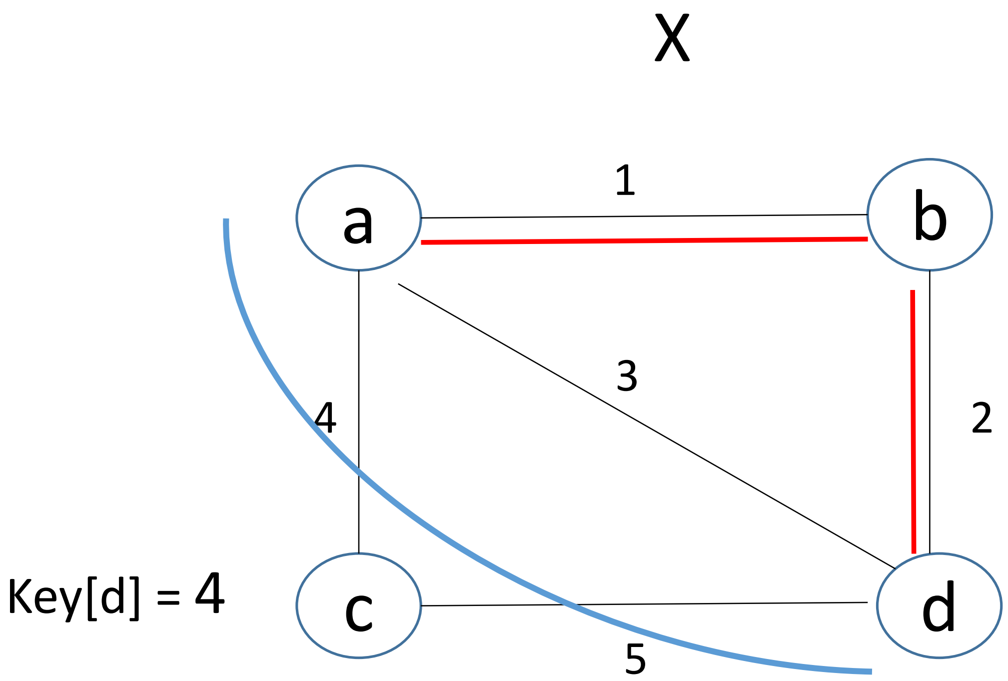
Vertex d is extracted with least cost from heap

Prim's Algorithm Dry Run MinHeap

Third Iteration:

X = {a , b, d} ,

Heap Q = { c[4]}



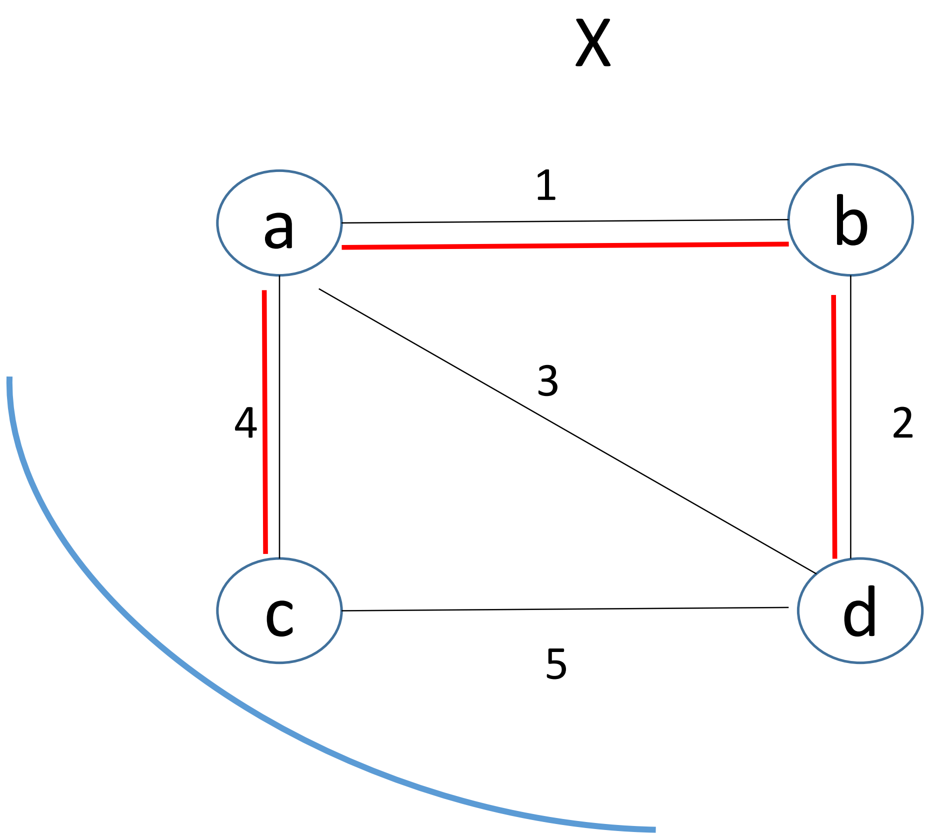
Keys are updated

Prim’s Algorithm Dry Run MinHeap

Fourth Iteration:

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

Heap Q = { }



Heap is empty, algorithm terminates

Prim's Algorithm (MinHeap with vertices) Time Complexity

- T = Ø [invariant: X = vertices spanned by tree-so-far T]
- for each u ∈ V → O (n)
  - Key[u] = ∞
- Key[s] = 0 // select any random vertex and make its cost 0
- Heap Q is initialized with all vertices → O (n lg n)
- While heap ≠ Ø → O (n)
  - u = ExtractMin from Heap → O (lg n)
  - Add u to X
  - for each v ∈ adj[u]
    - DecreaseKey(Q, v, cost[u,v])

O (m lg n) time for all iterations of outer while

Overall running time = n + nlgn + n (lgn) + mlgn = O(mlgn)