Design and Analysis of Algorithms
Part IV: Graph Algorithms
Lecture 11: Minimum Spanning Trees

Yongxin Tong (童咏昕)

School of CSE, Beihang University

yxtong@buaa.edu.cn

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim's algorithm
  - The idea
  - The algorithm
  - Analysis for Prim's algorithm
- Kruskal's algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal's algorithm

Introduction to Part IV

- In Part IV, we will illustrate several graph algorithm problems using several examples:
  - Basic Concepts of Graphs (图的基本概念)
  - Breadth-First Search [BFS] (广度优先搜索)
  - Depth-First Search [DFS] (深度优先搜索)
  - Topological Sort (拓扑排序)
  - Strongly Connected Components (强联通分量)
  - Minimum Spanning Trees (最小生成树)
  - Shortest Path (最短路径)
  - All-Pairs Shortest Paths (所有结点对的最短路径)
  - Maximum/Network Flows (最大流/网络流)

Introduction to Part IV

- In Part IV, we will illustrate several graph algorithm problems using several examples:
  - Basic Concepts of Graphs (图的基本概念)
  - Breadth-First Search [BFS] (广度优先搜索)
  - Depth-First Search [DFS] (深度优先搜索)
  - Topological Sort (拓扑排序)
  - Strongly Connected Components (强联通分量)
  - Minimum Spanning Trees (最小生成树)
  - Shortest Path (最短路径)
  - All-Pairs Shortest Paths (所有结点对的最短路径)
  - Maximum/Network Flows (最大流/网络流)

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Spanning Trees

Definition

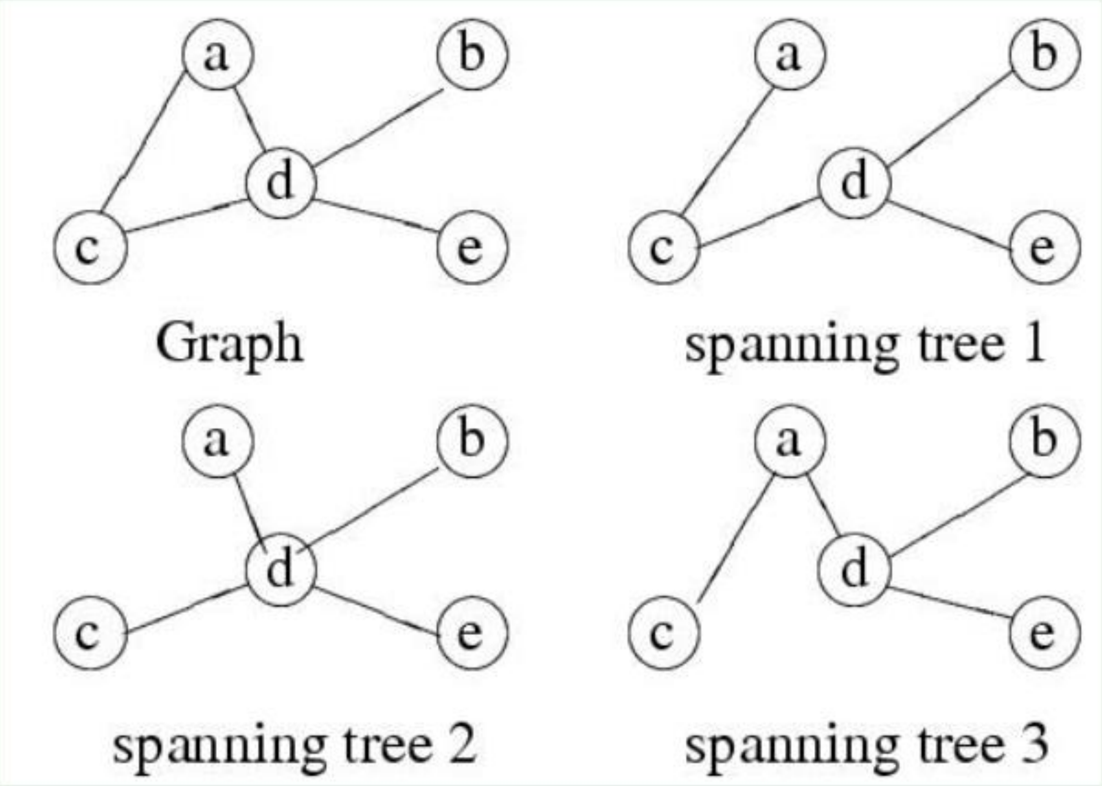
A subgraph T of a undirected graph $G = (V, E)$ is a spanning tree of G if it is a tree and contains every vertex of G

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 Trees

Theorem

Every connected graph has a spanning tree.

Spanning Trees

Theorem

Every connected graph has a spanning tree.

Question

Why is this true?

Spanning Trees

Theorem

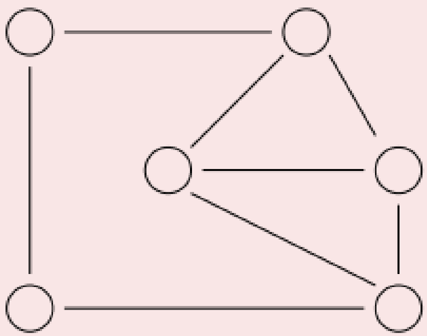
Every connected graph has a spanning tree.

Question

Why is this true?

Question

Given a connected graph G, how can you find a spanning tree of G?



Weighted Graphs

Definition

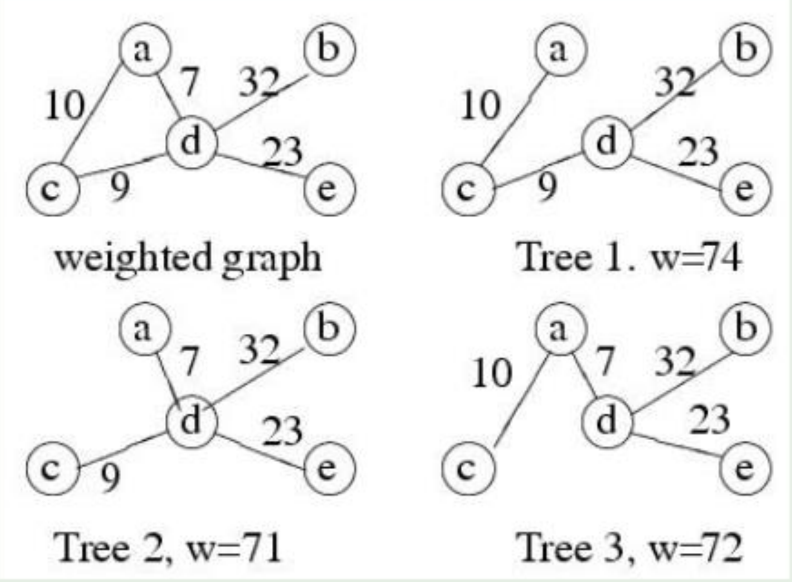
A weighted graph is a graph, in which each edge has a weight (some real number)

Weighted Graphs

Definition

A weighted graph is a graph, in which each edge has a weight (some real number)

Example

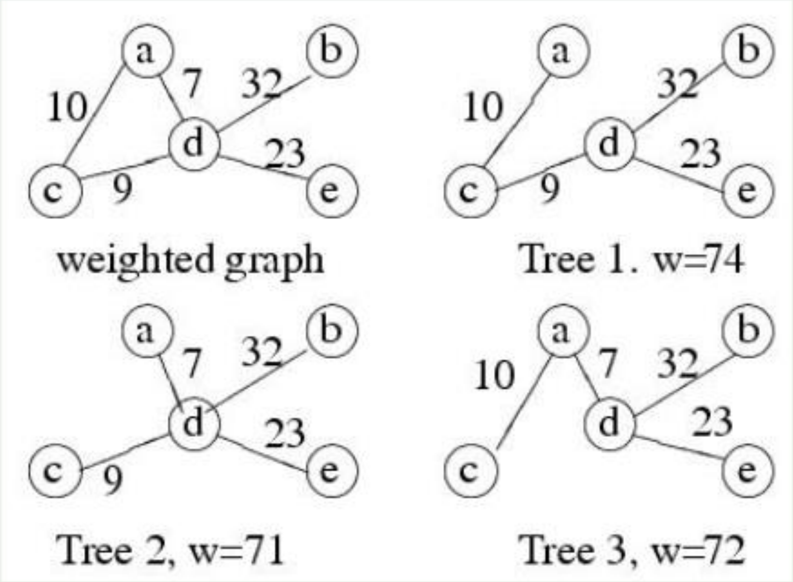


Weighted Graphs

Definition

A weighted graph is a graph, in which each edge has a weight (some real number)

Example



Definition

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

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum Spanning Trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Minimum Spanning Trees

Definition

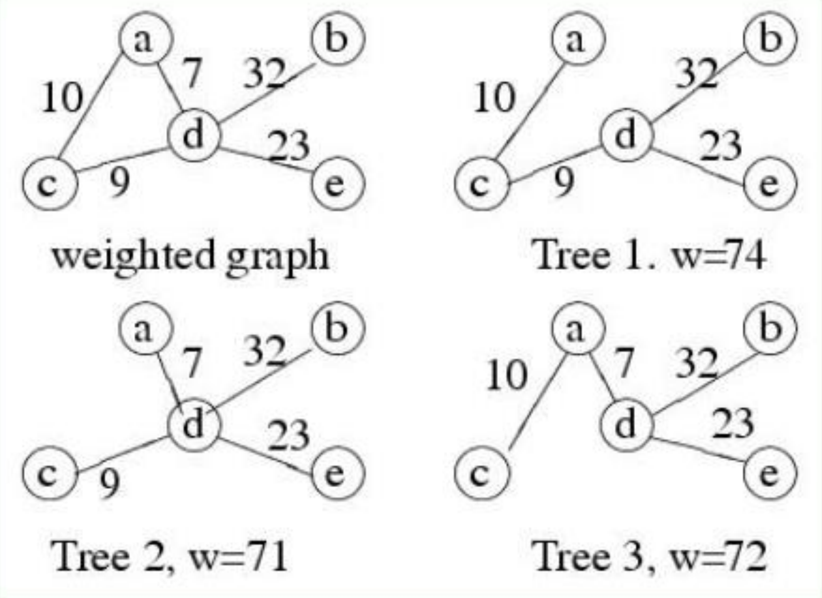
A Minimum spanning tree of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

Minimum Spanning Trees

Definition

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

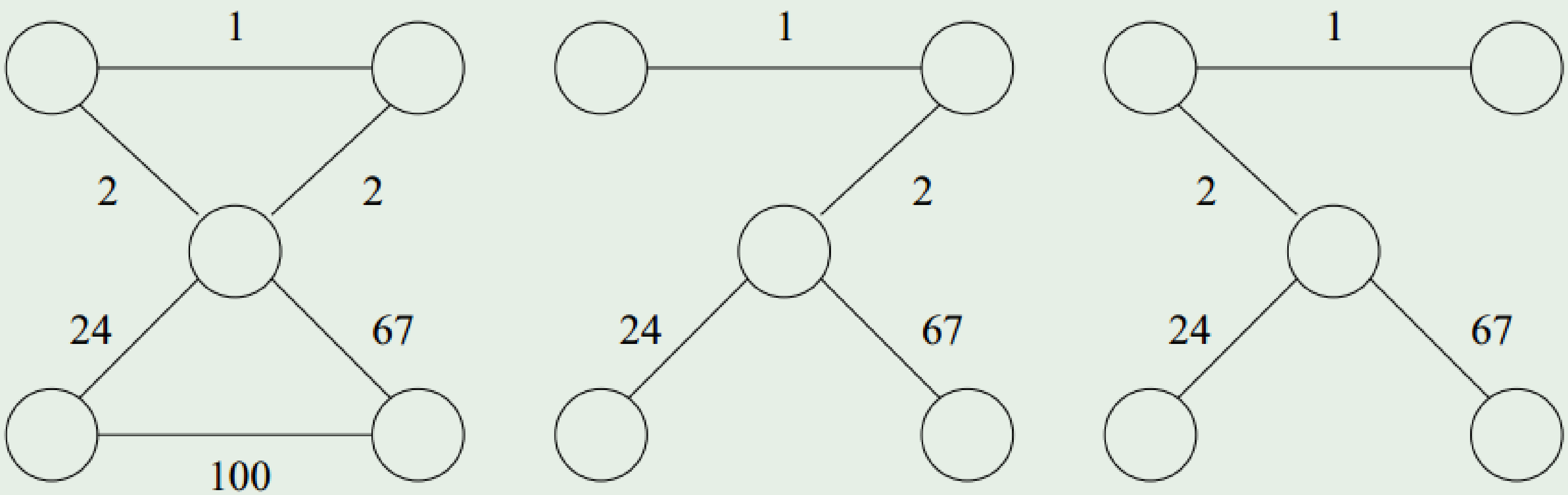
Example



Remark

The minimum spanning tree may not be unique

Example

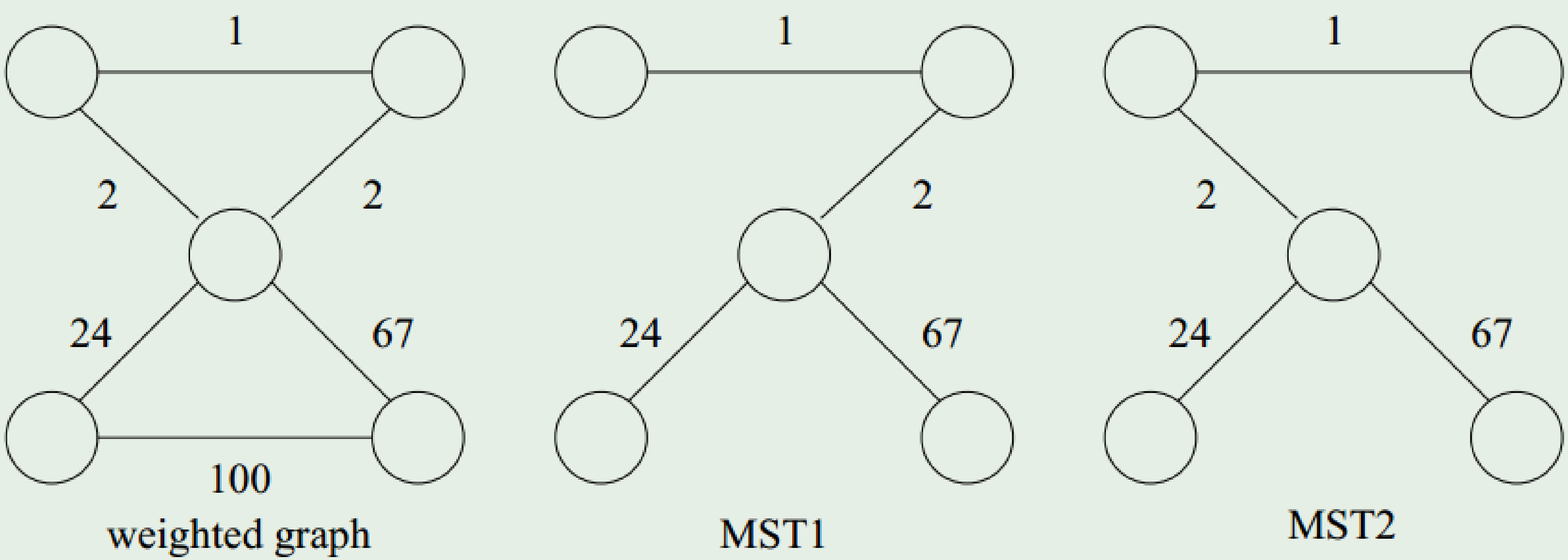


weighted graph MST1 MST2

Remark

The minimum spanning tree may not be unique

Example



However, if the weights of all the edges are distinct, it is indeed unique (we won't prove this now)

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.

General strategy for solving the MST Problem

A tree is an acyclic graph

- start with an empty graph.

General strategy for solving the MST Problem

A tree is an acyclic graph

- start with an empty graph.
- try to add edges one at a time, always making sure that what is built remains acyclic.

General strategy for solving the MST Problem

A tree is an acyclic graph

- start with an empty graph.
- try to add edges one at a time, always making sure that what is built remains acyclic.
- if after adding each edge we are sure that the resulting graph is a subset of some minimum spanning trees, we are done.

Generic Algorithm for MST problem

Definition

Let A be a set of edges such that $A \subseteq T$, where T is a MST.

Generic Algorithm for MST problem

Definition

Let A be a 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

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic-MST(G)

Input: A graph $\boldsymbol{G}$
Output: $\boldsymbol{A}$ is the MST of G
$A \leftarrow$ EMPTY;

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic-MST(G)

Input: A graph $\boldsymbol{G}$
Output: $\boldsymbol{A}$ is the MST of G
$A \leftarrow$ EMPTY;
while A *does not form a spanning tree* **do**

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic-MST(G)

Input: A graph $\boldsymbol{G}$
Output: $\boldsymbol{A}$ is the MST of G
$A \leftarrow$ EMPTY;
while A *does not form a spanning tree* **do**
  find an edge(u, v) that is safe for A;

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic-MST(G)

```
Input: A graph G
Output: A is the MST of G
A ← EMPTY;
while A does not form a spanning tree do
    find an edge(u, v) that is safe for A;
    add (u, v) to A;
```

Generic Algorithm for MST problem

Definition

Let A be a 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 a safe edge (u, v), we can grow a MST

Generic-MST(G)

```
Input: A graph G
Output: A is the MST of G
A ← EMPTY;
while A does not form a spanning tree do
    find an edge(u, v) that is safe for A;
    add (u, v) to A;
end
return A;
```

Some Definitions

Definition

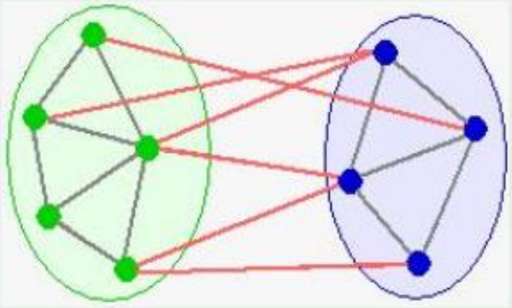
Let $G = (V, E)$ be a connected and undirected graph. A cut $(S, V - S)$ of G is a partition of V.

Some Definitions

Definition

Let $G = (V, E)$ be a connected and undirected graph. A cut $(S, V - S)$ of G is a partition of V.

Example

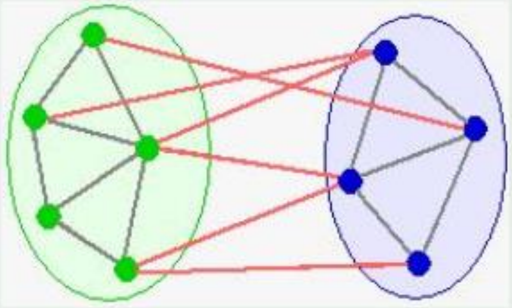


Some Definitions

Definition

Let $G = (V, E)$ be a connected and undirected graph. A cut $(S, V - S)$ of G is a partition of V.

Example



Definition

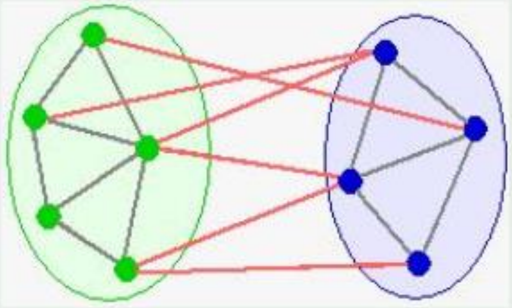
An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.

Some Definitions

Definition

Let $G = (V, E)$ be a connected and undirected graph. A cut $(S, V - S)$ of G is a partition of V.

Example



Definition

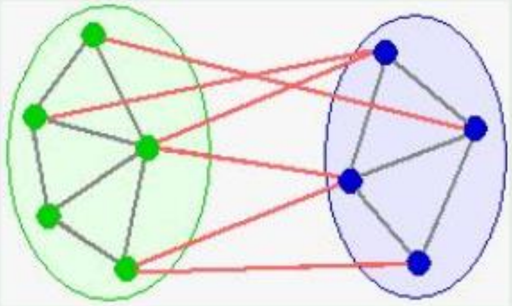
An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.
A cut respects a set A of edges if no edge in A crosses the cut.

Some Definitions

Definition

Let $G = (V, E)$ be a connected and undirected graph. A cut $(S, V - S)$ of G is a partition of V.

Example



Definition

An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.
A cut respects a set A of edges if no edge in A crosses the cut.
An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

How to Find a Safe Edge?

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*

How to Find a Safe Edge?

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

How to Find a Safe Edge?

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*

How to Find a Safe Edge?

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

How to Find a Safe Edge?

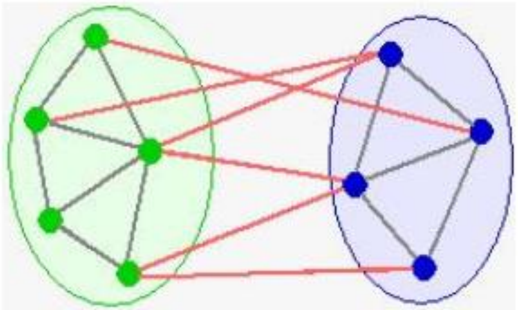
Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.



How to Find a Safe Edge?

Lemma

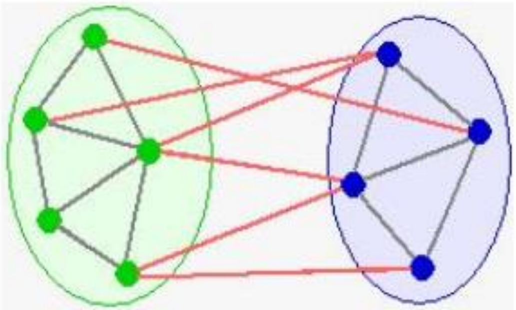
- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.

It means that we can find a safe edge by



How to Find a Safe Edge?

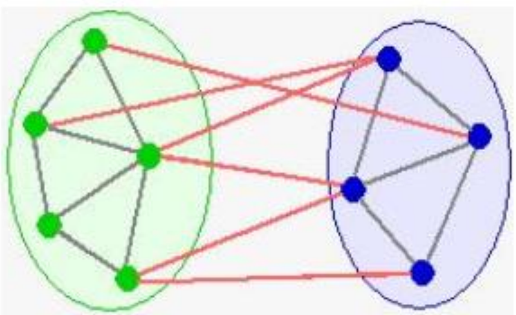
Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.



It means that we can find a safe edge by

1. first finding a cut that respects A,

How to Find a Safe Edge?

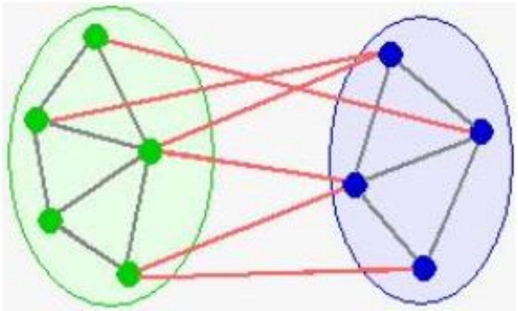
Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.



It means that we can find a safe edge by

1. first finding a cut that respects A,
2. then finding the light edge crossing that cut.

That light edge is a safe edge.

Proof

- Let $A \subseteq T$, where T is a MST.

Proof

- Let A ⊆ T, where T is a MST.
- Case 1: (u, v) ∈ T

Proof

- Let $A \subseteq T$, where T is a MST.
- Case 1: $(u, v) \in T$
  - $A \cup \{(u, v)\} \subseteq T$.
  - Hence (u, v) is safe for A.

Proof (cont'd)

- Case 2: $(u, v) \notin T$

Proof (cont'd)

- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.

Proof (cont'd)

- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.
  - Consider a path P in T from u to v.

Proof (cont'd)

- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.
  - Consider a path P in T from u to v.
  - Since u and v are on opposite sides of the cut (S, V−S),
    - There is at least one edge in P that crosses the cut.

Proof (cont'd)

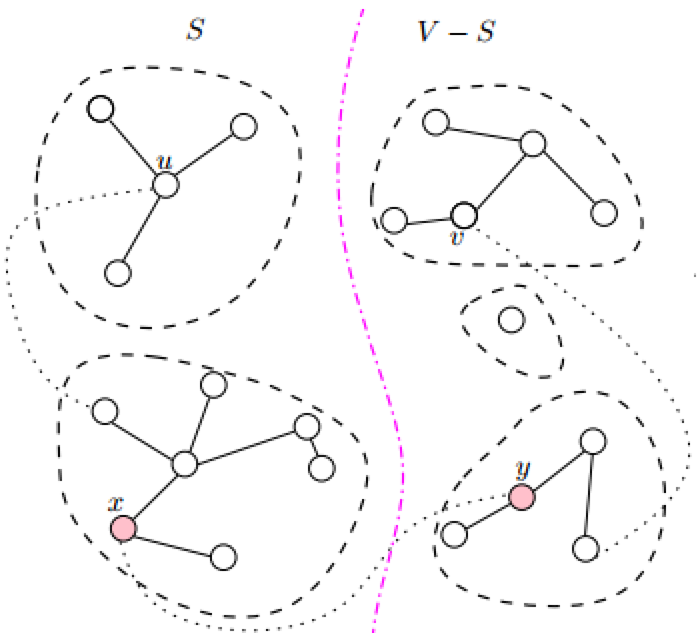
- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.
  - Consider a path P in T from u to v.
  - Since u and v are on opposite sides of the cut (S, V−S),
    - There is at least one edge in P that crosses the cut.
    - Let (x, y) be such an edge.

Proof (cont'd)

- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.
  - Consider a path P in T from u to v.
  - Since u and v are on opposite sides of the cut (S, V−S),
    - There is at least one edge in P that crosses the cut.
    - Let (x, y) be such an edge.
  - Since the cut respects A, $(x, y) \notin A$.

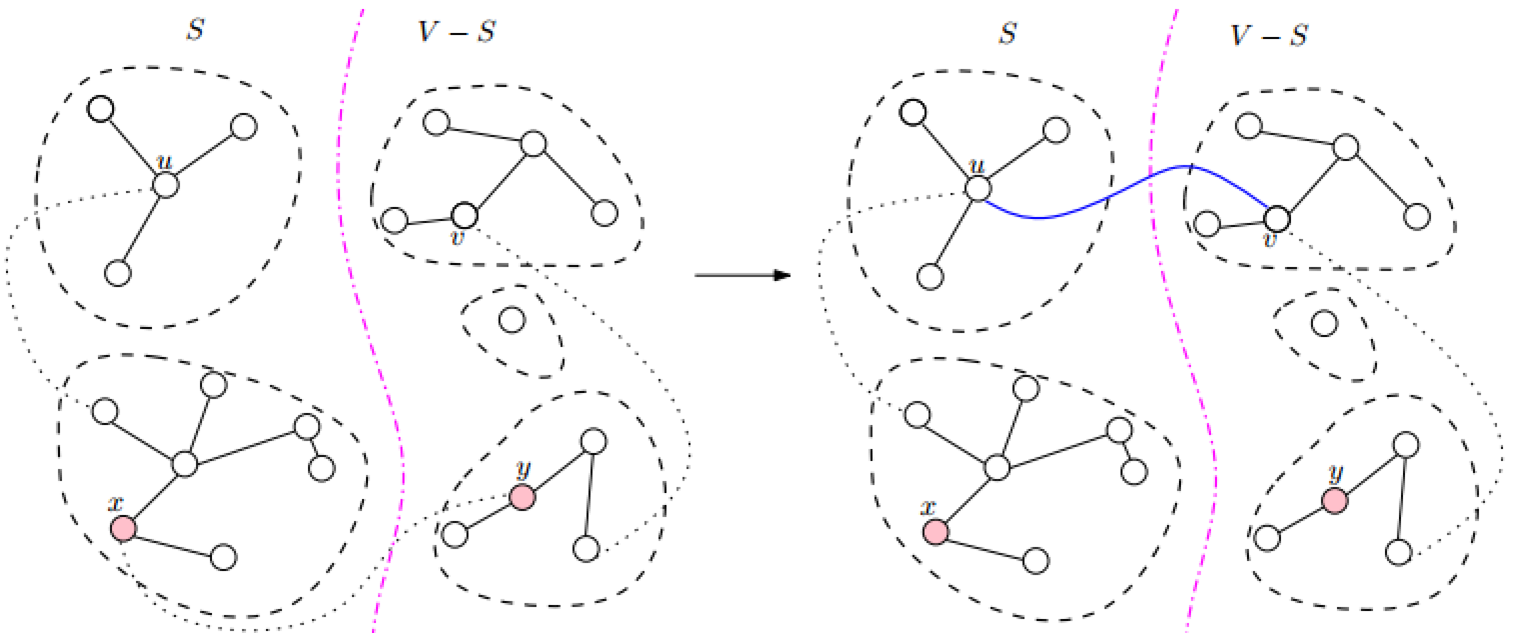
Proof (cont'd)

- Case 2: $(u, v) \notin T$
  - Idea: construct another MST T' s.t. $A \cup \{(u, v)\} \subseteq T'$.
  - Consider a path P in T from u to v.
  - Since u and v are on opposite sides of the cut (S, V−S),
    - There is at least one edge in P that crosses the cut.
    - Let (x, y) be such an edge.
  - Since the cut respects A, $(x, y) \notin A$.
  - Since (u, v) is a light edge crossing the cut, we have $w(x, y) \geq w(u, v)$.



Proof (cont'd)

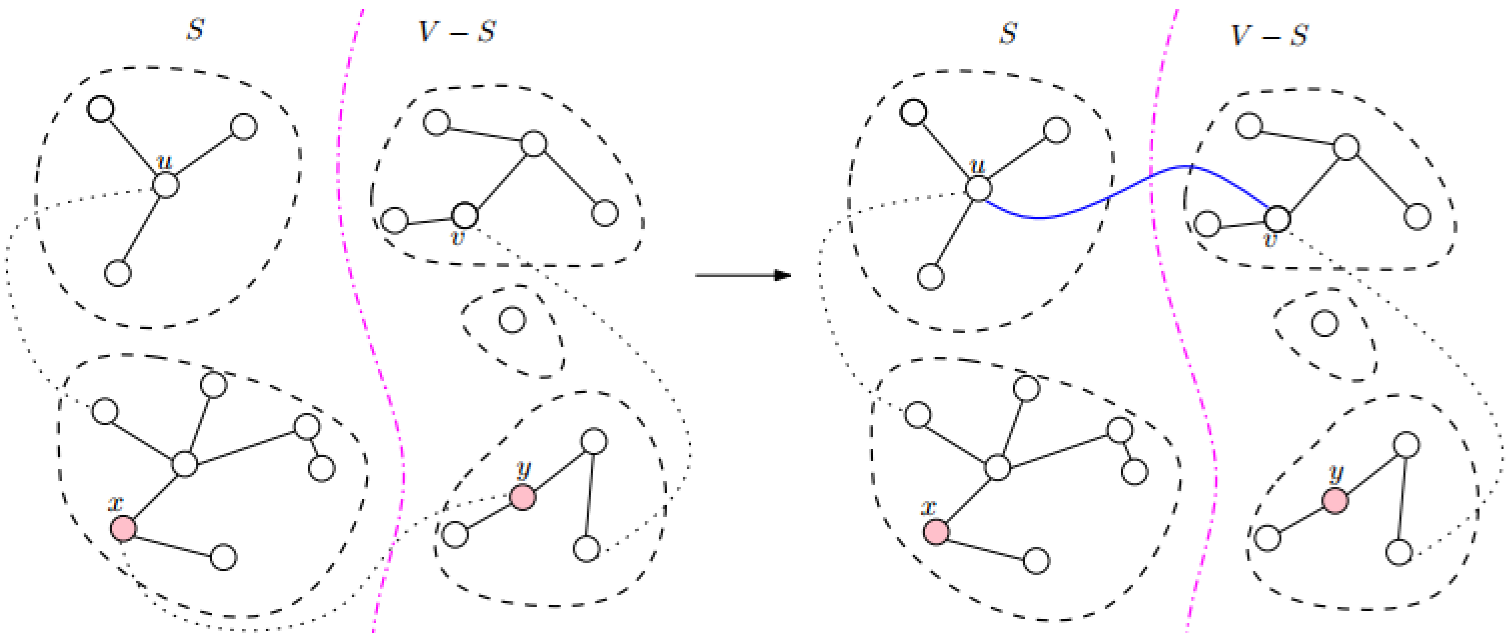
- Add (u, v) to T, it creates a cycle. By removing an edge from the cycle, it becomes a tree again. In particular, we remove (x, y) (∉ A) to make a new tree T'.



Proof (cont'd)

- Add (u, v) to T, it creates a cycle. By removing an edge from the cycle, it becomes a tree again. In particular, we remove (x, y) (∉ A) to make a new tree T'.
- The weight of T' is

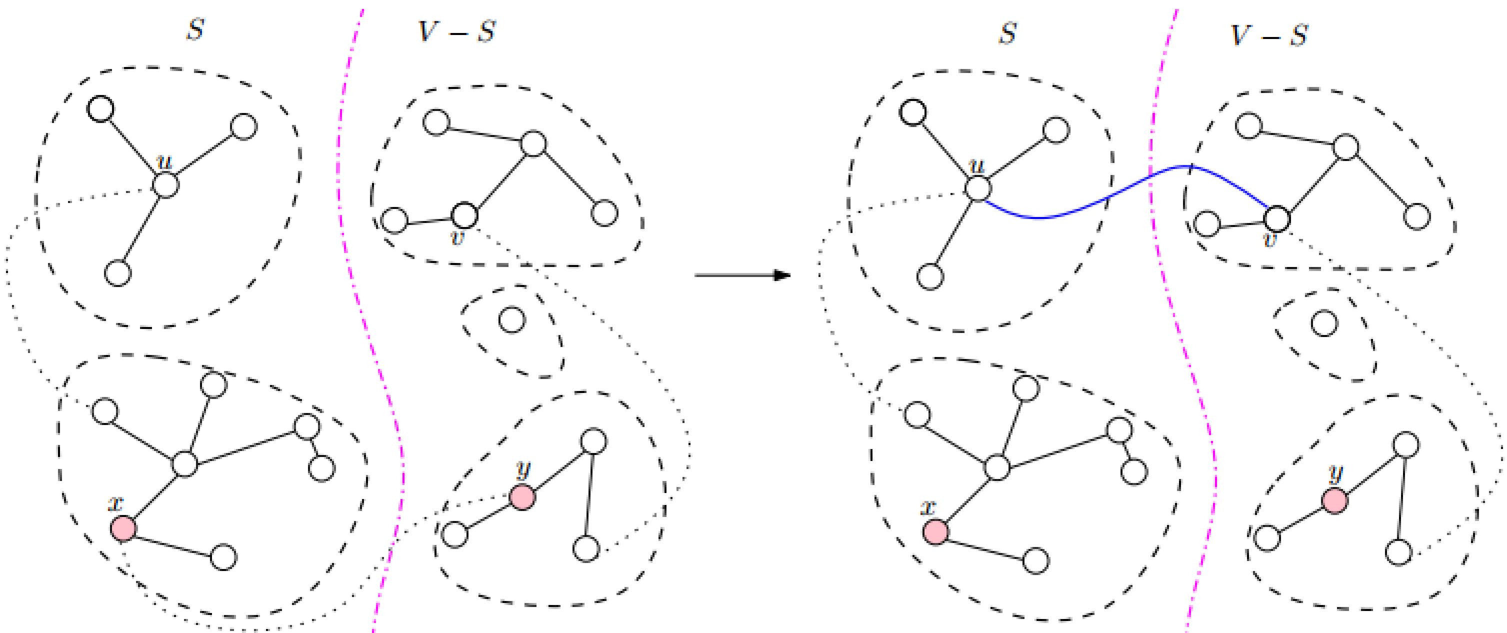
$$w(T') = w(T) - w(x, y) + w(u, v)$$



Proof (cont'd)

- Add (u, v) to T, it creates a cycle. By removing an edge from the cycle, it becomes a tree again. In particular, we remove (x, y) (∉ A) to make a new tree T'.
- The weight of T' is

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

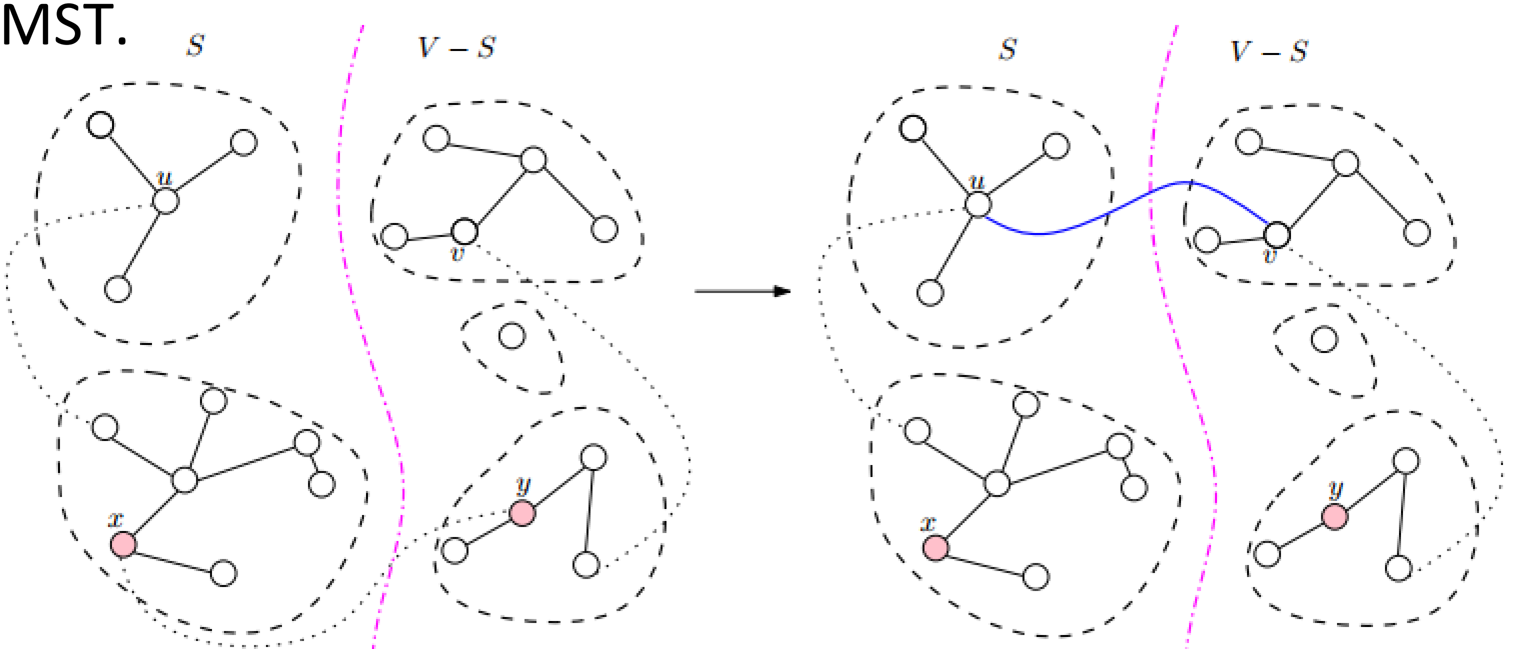


Proof (cont'd)

- Add (u, v) to T, it creates a cycle. By removing an edge from the cycle, it becomes a tree again. In particular, we remove (x, y) (∉ A) to make a new tree T'.
- The weight of T' is

$$\begin{aligned} w(T') &= w(T) - w(x,y) + w(u,v) \\ &\le w(T) \end{aligned}$$

- Since T is a MST, we must have w(T) = w(T'), hence T' is also a MST.

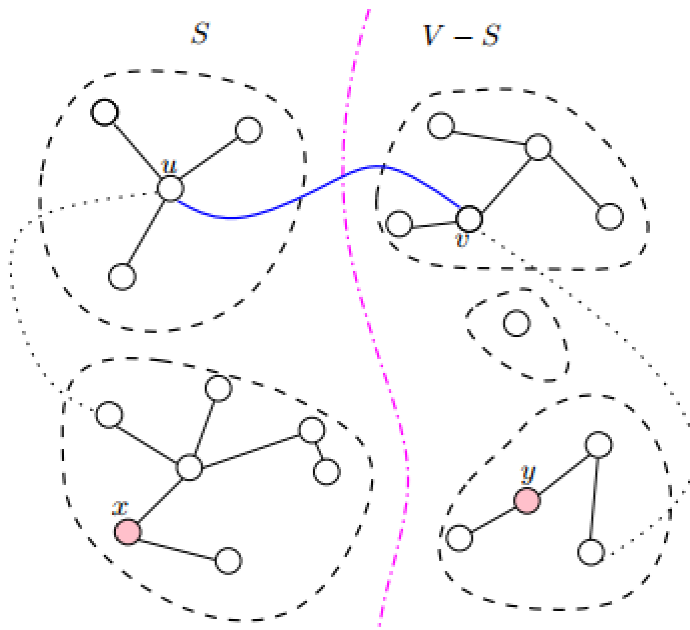


Proof (cont'd)

- Add (u, v) to T, it creates a cycle. By removing an edge from the cycle, it becomes a tree again. In particular, we remove (x, y) ($\notin$ A) to make a new tree T'.
- The weight of T' is

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) \end{aligned}$$

- Since T is a MST, we must have w(T) = w(T'), hence T' is also a MST.
- Since A $\cup$ {(u,v)} $\subseteq$ T', (u, v) is safe for A.
- The Lemma is proved.



Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration
  - grows a single tree and adds a light edge in each iteration.

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration
  - grows a single tree and adds a light edge in each iteration.
  - Grow a tree

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration
  - grows a single tree and adds a light edge in each iteration.
  - Grow a tree
    - Start by picking any vertex r to be the root of the tree.

Prim’s Algorithm

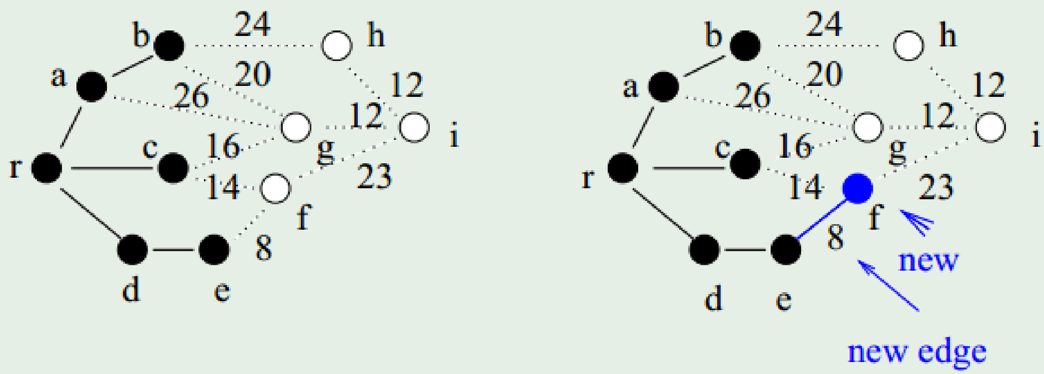
- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration
  - grows a single tree and adds a light edge in each iteration.
  - Grow a tree
    - Start by picking any vertex r to be the root of the tree.
    - While the tree does not contain all vertices in the graph:

Prim’s Algorithm

- The generic algorithm gives us an idea how to ’grow’ a MST.
  - If you read the theorem and the proof carefully, you will notice that the choice of a cut (and hence the corresponding light edge) in each iteration is arbitrary.
  - We can select any cut (that respects the selected edges) and find the light edge crossing that cut to proceed.
- Prim’s algorithm makes a nature choice of the cut in each Iteration
  - grows a single tree and adds a light edge in each iteration.
  - Grow a tree
    - Start by picking any vertex r to be the root of the tree.
    - While the tree does not contain all vertices in the graph: find shortest edge leaving the tree and add it to the tree.

More Details

Example

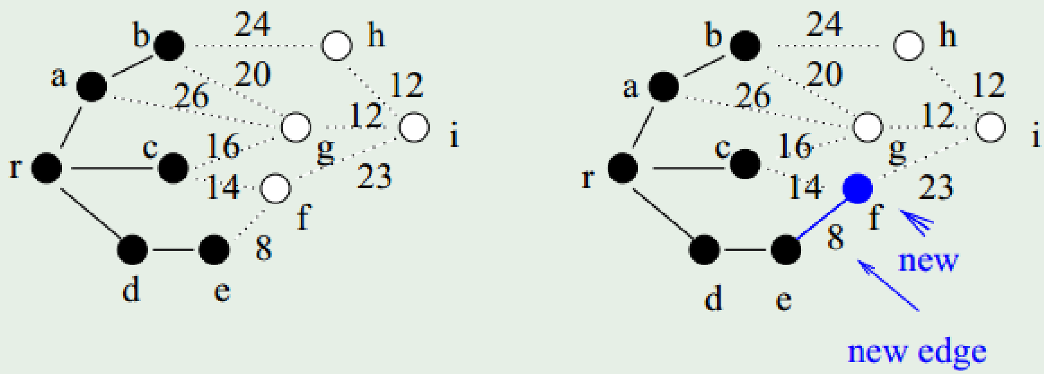


Step 0:

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

More Details

Example



Step 0:

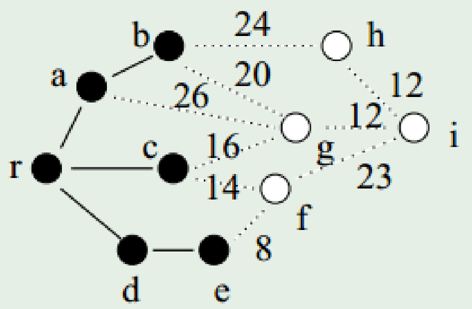
- Choose any element r; set $S = \{r\}$ and $A = \emptyset$.
- (Take r as the root of our spanning tree.)

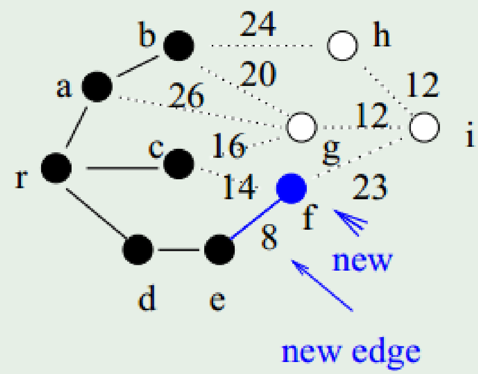
Step 1:

- Find a lightest edge such that one endpoint is in S and the other is in $V \setminus S$.

More Details

Example





Step 0:

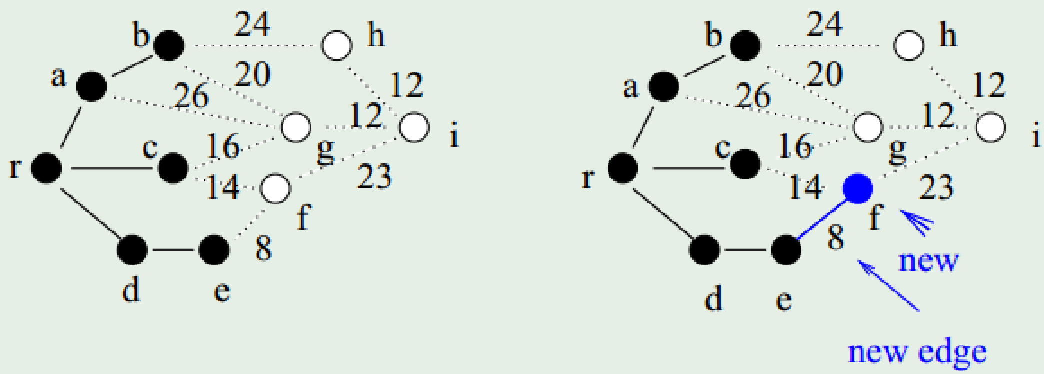
- Choose any element r; set $S = \{r\}$ and $A = \emptyset$.
- (Take r as the root of our spanning tree.)

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.

More Details

Example



Step 0:

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

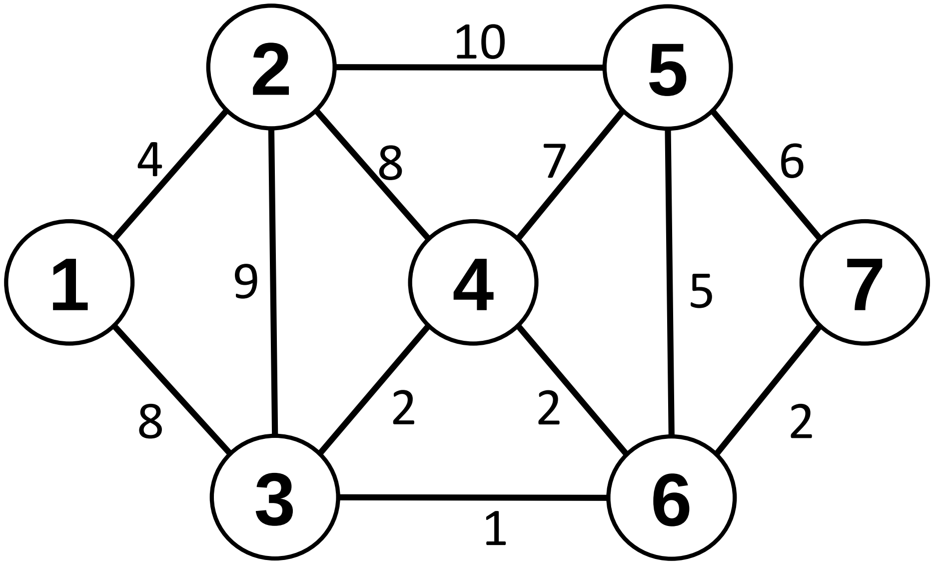
Step 1:

- 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.

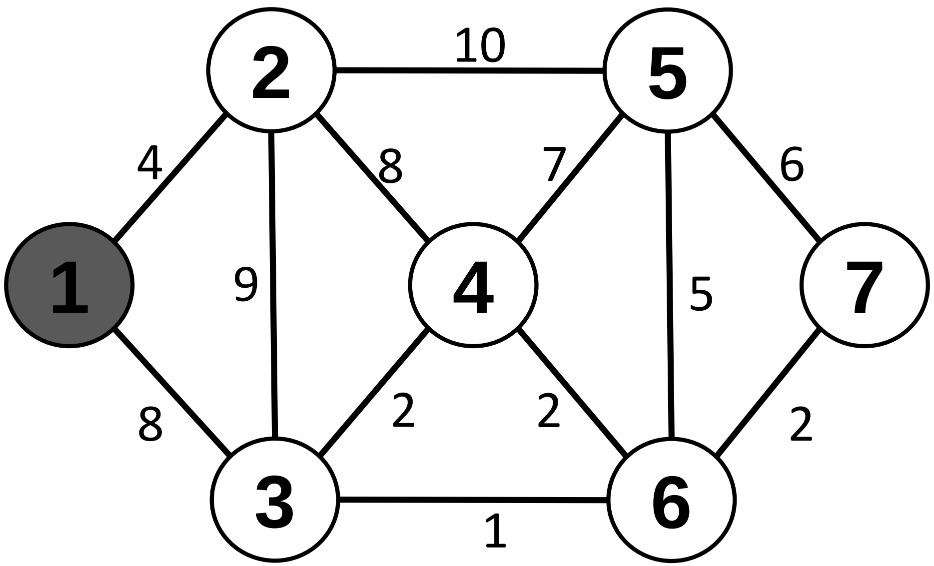
Step 2:

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

Prim's Example



Connected graph



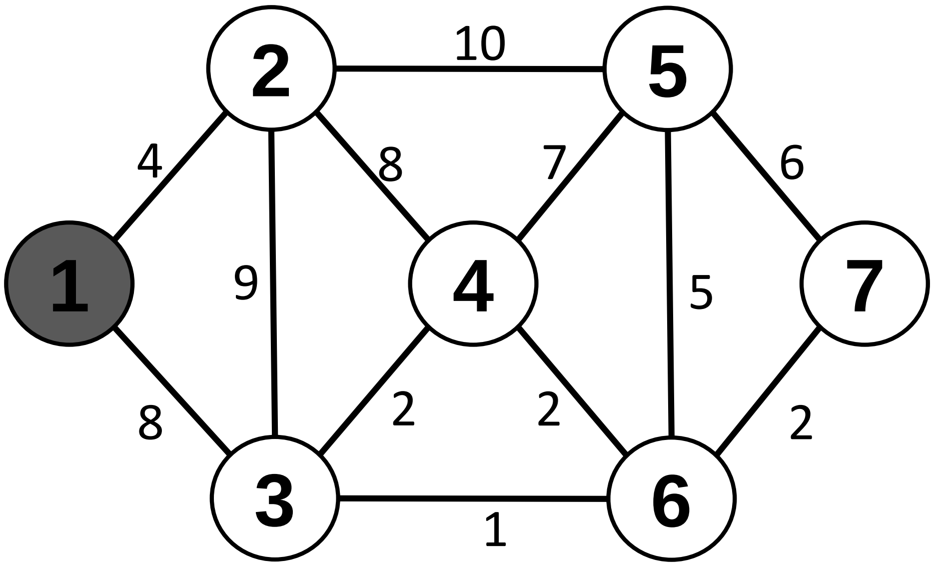
Step 0

S = {1}

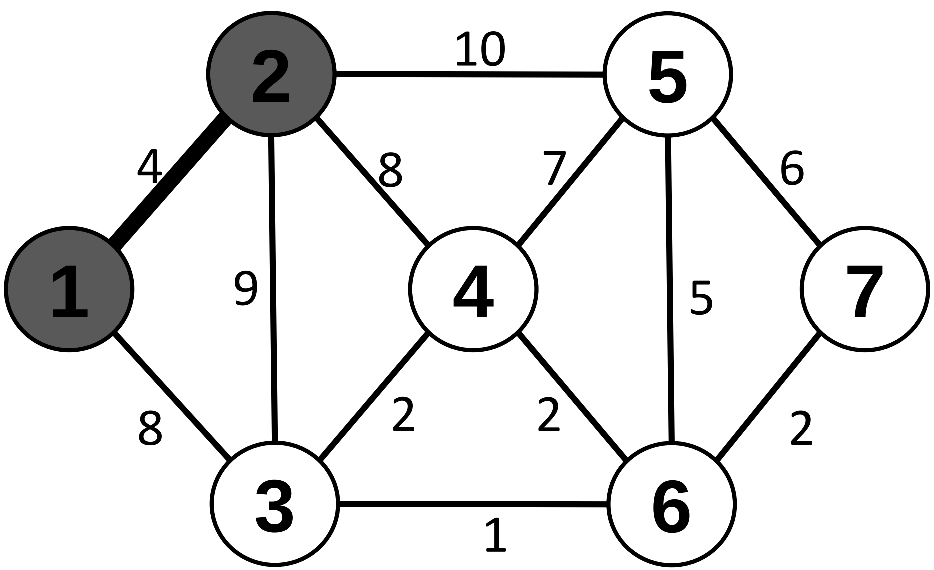
V \ S = {2,3,4,5,6,7}

lightest edge = {1,2}

Prim's Example

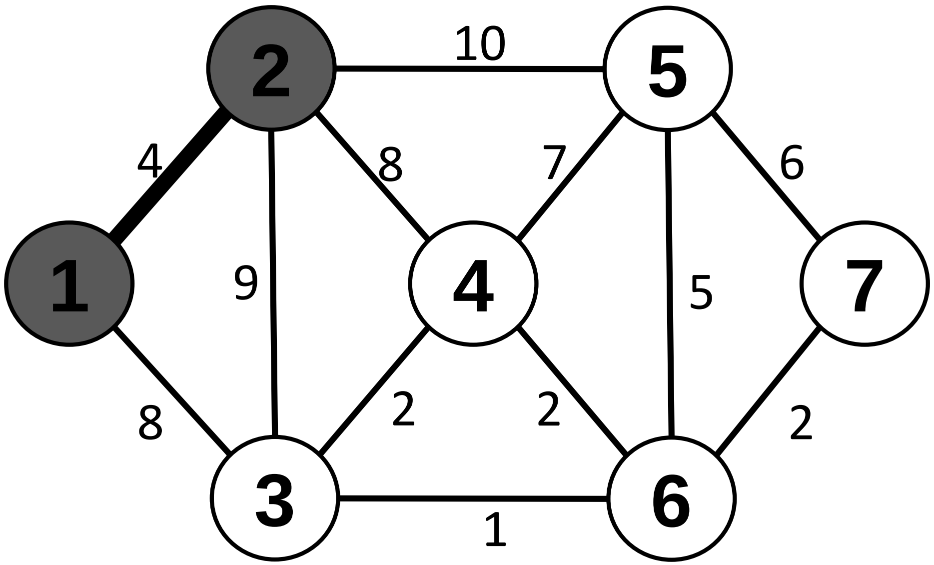


Step 1.1 before
S = {1}
V \ S = {2,3,4,5,6,7}
A={}
lightest edge = {1,2}

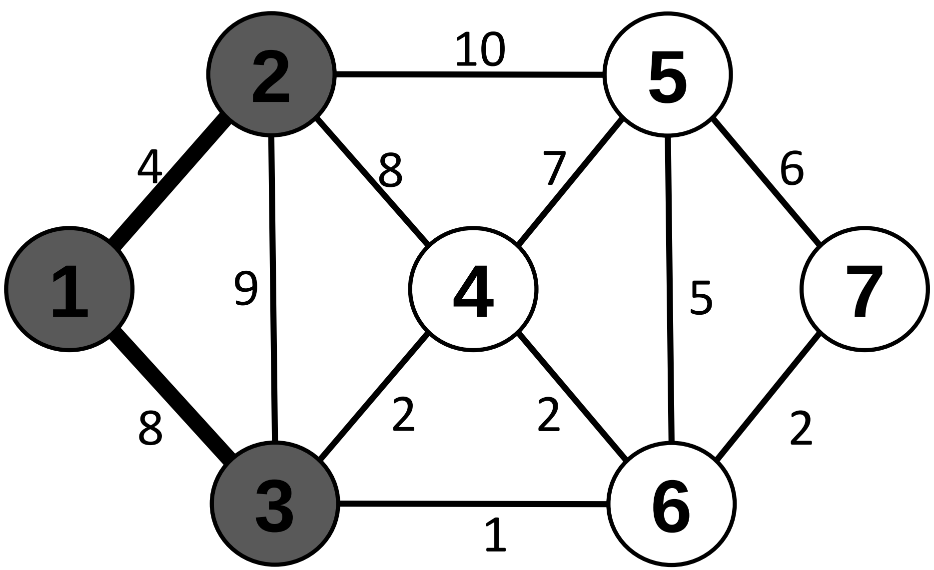


Step 1.1 after
S = {1,2}
V \ S = {3,4,5,6,7}
A = { {1,2} }
lightest edge = {1,3},{2,4}

Prim's Example

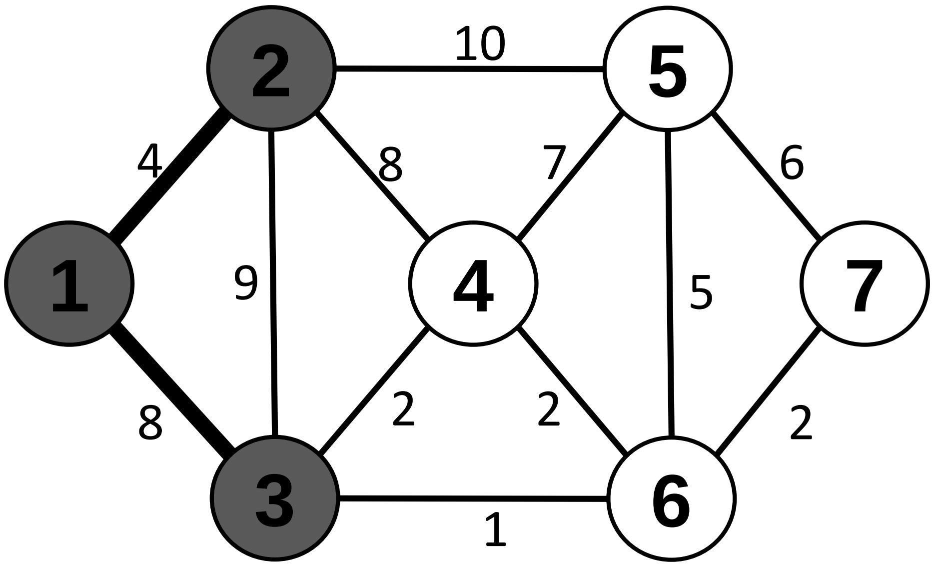


Step 1.2 before
S = {1,2}
V \ S = {3,4,5,6,7}
A = { {1,2} }
lightest edge = {1,3},{2,4}

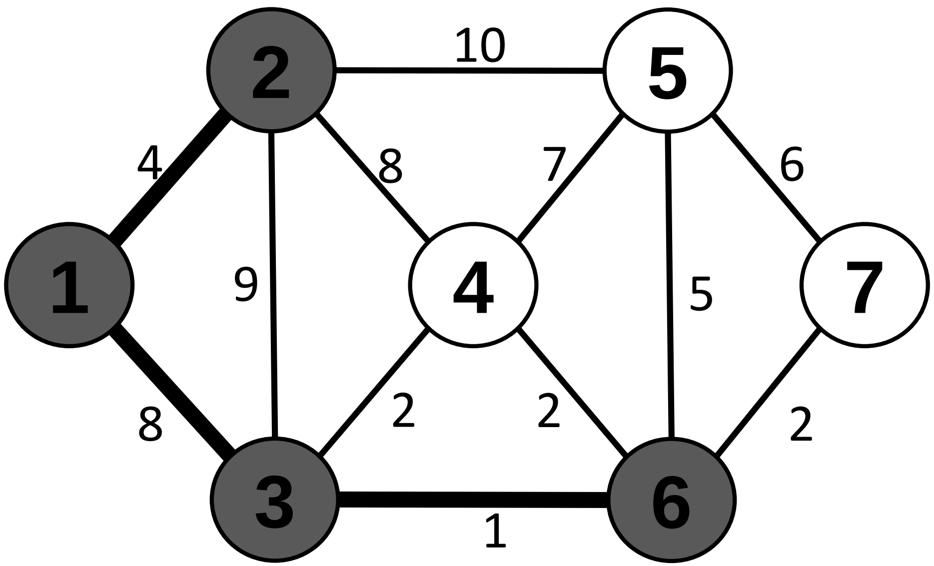


Step 1.2 after
S = {1,2,3}
V \ S = {4,5,6,7}
A = { {1,2},{1,3} }
lightest edge = {3,6}

Prim's Example

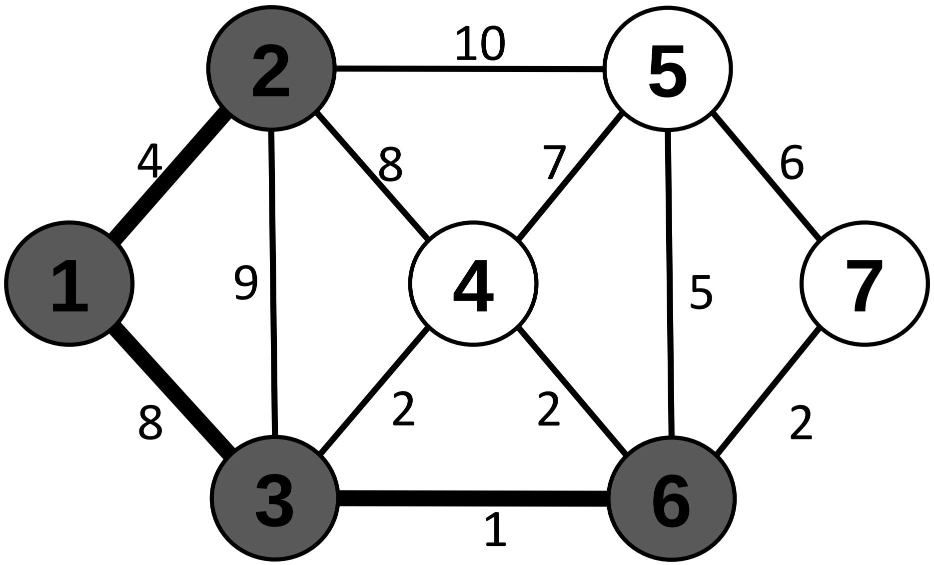


Step 1.3 before
S = {1,2,3}
V \ S = {4,5,6,7}
A = { {1,2},{1,3} }
lightest edge = {3,6}

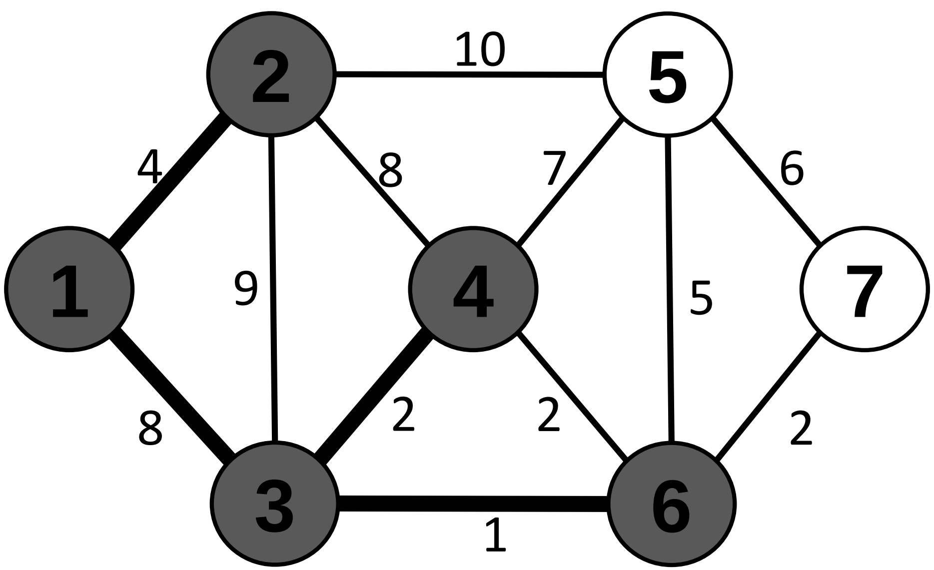


Step 1.3 after
S = {1,2,3,6}
V \ S = {4,5,7}
A = { {1,2},{1,3},{3,6} }
lightest edge = {3,4},{6,4},{6,7}

Prim's Example

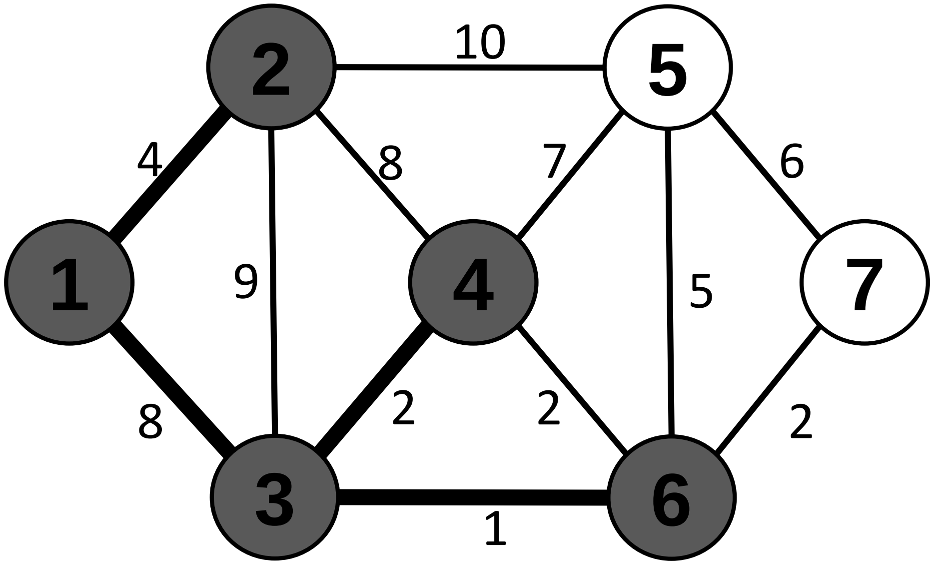


Step 1.4 before
S = {1,2,3,6}
V \ S = {4,5,7}
A = { {1,2},{1,3},{3,6} }
lightest edge = {3,4},{6,4},{6,7}

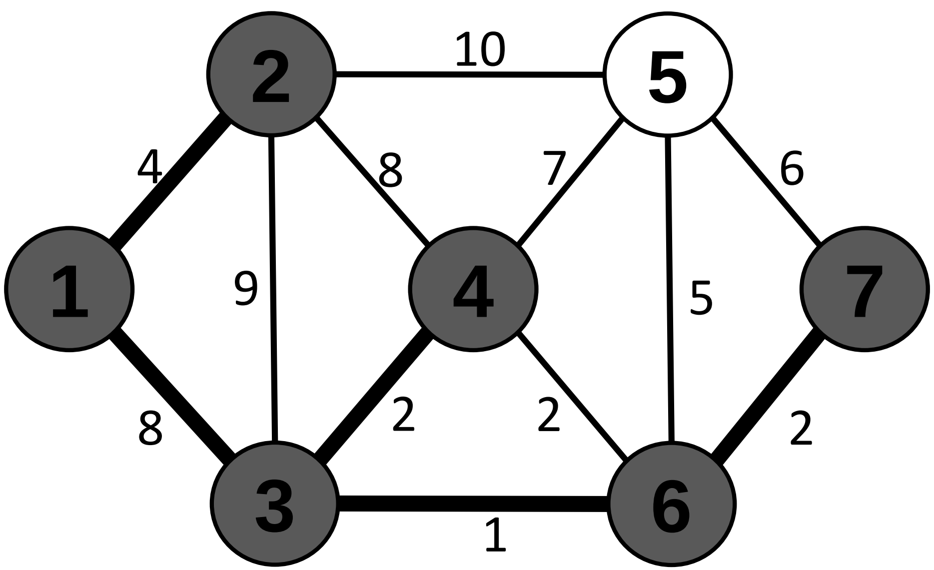


Step 1.4 after
S = {1,2,3,4,6}
V \ S = {5,7}
A = { {1,2},{1,3},{3,6},{3,4} }
lightest edge = {6,7}

Prim's Example

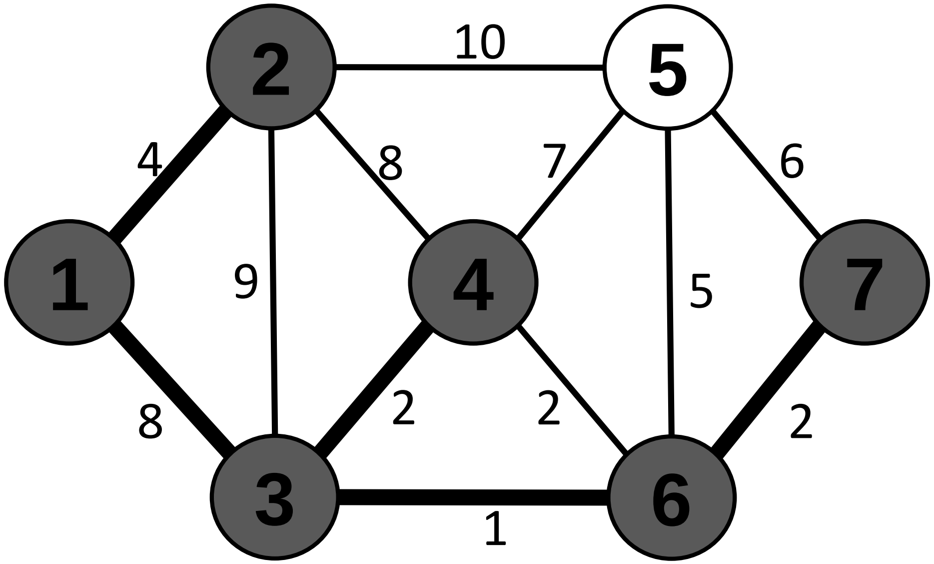


Step 1.5 before
S = {1,2,3,4,6}
V \ S = {5,7}
A = { {1,2},{1,3},{3,6},{3,4} }
lightest edge = {6,7}

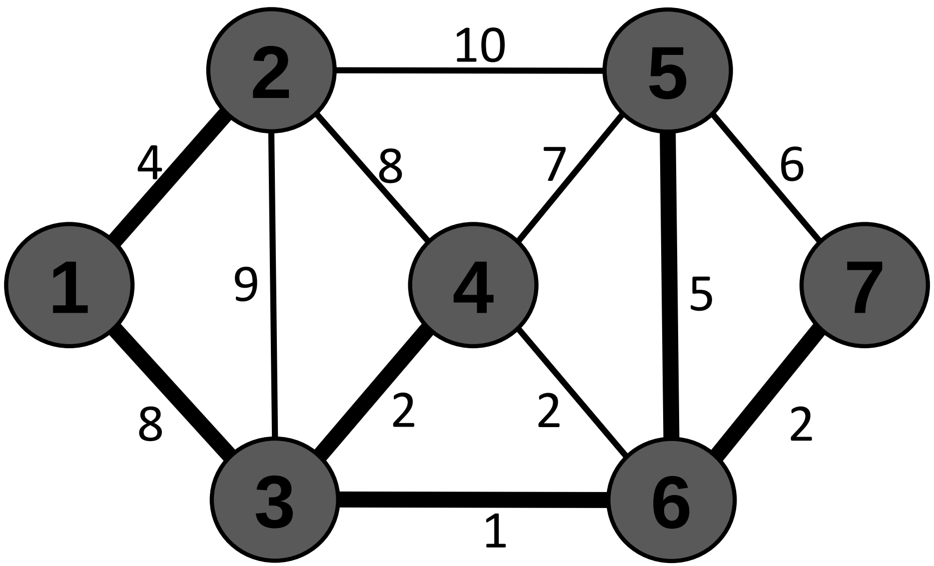


Step 1.5 after
S = {1,2,3,4,6,7}
V \ S = {5}
A = { {1,2},{1,3},{3,6},{3,4},{6,7} }
lightest edge = {6,5}

Prim's Example



Step 1.6 before
S = {1,2,3,4,6,7}
V \ S = {5}
A = { {1,2},{1,3},{3,6},{3,4},{6,7} }
lightest edge = {6,5}



Step 1.6 after
S = {1,2,3,4,5,6,7}
V \ S = {}
A = { {1,2},{1,3},{3,6},{3,4},{6,7}, {6,5} }
MST completed

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Recall Idea of Prim's Algorithm

Step 0: Choose any element r and set S = {r} and A = ∅. (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 \ S. Add this edge to A and its (other) endpoint to S.

Step 2: If V \ S = ∅, then stop and output the minimum spanning tree (S,A); Otherwise go to Step 1.

Recall Idea of Prim's Algorithm

Step 0: Choose any element r and set S = {r} and A = ∅. (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 \ S. Add this edge to A and its (other) endpoint to S.

Step 2: If V \ S = ∅, then stop and output the minimum spanning tree (S,A); Otherwise go to Step 1.

Questions:

- How does the algorithm update S efficiently?
- How does the algorithm find the lightest edge and update A efficiently?

Prim’s Algorithm

Question

How does the algorithm update S efficiently?

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.
- Change the color to black when the vertex is moved to S.

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.
- Change the color to black when the vertex is moved to S.
- Use color$[v]$ to store color.

Prim’s Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.
- Change the color to black when the vertex is moved to S.
- Use color$[v]$ to store color.

Question

How does the algorithm find the lightest edge and update A efficiently?

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.
- Change the color to black when the vertex is moved to S.
- Use color[v] to store color.

Question

How does the algorithm find the lightest edge and update A efficiently?

Answer:

1. Use a priority queue to find the lightest edge.

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.
- Change the color to black when the vertex is moved to S.
- Use color$[v]$ to store color.

Question

How does the algorithm find the lightest edge and update A efficiently?

Answer:

1. Use a priority queue to find the lightest edge.
2. Use pred$[v]$ to update A.

Reviewing Priority Queues

Priority Queue is a data structure

Reviewing Priority Queues

Priority Queue is a data structure

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Supports the following operations:

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Supports the following operations:

Insert(u, key): Insert u with the key value key in Q.

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Supports the following operations:

Insert(u, key): Insert u with the key value key in Q.

u = Extract-Min(): Extract the item with minimum key value.

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Supports the following operations:

Insert(u, key): Insert u with the key value key in Q.

u = Extract-Min(): Extract the item with minimum key value.

Decrease-Key(u, new-key): Decrease u’s key value to new-key.

Reviewing Priority Queues

Priority Queue is a data structure

- can be implemented as a heap

Supports the following operations:

Insert(u, key): Insert u with the key value key in Q.

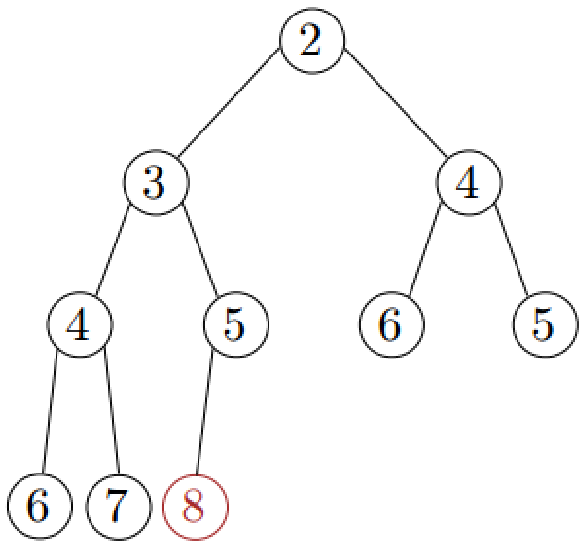
u = Extract-Min(): Extract the item with minimum key value.

Decrease-Key(u, new-key): Decrease u's key value to new-key.

Remark: Priority Queues can be implemented so that each operation takes time O(log |Q|). See Lecture 5 & Chapter 6.5 CLRS.

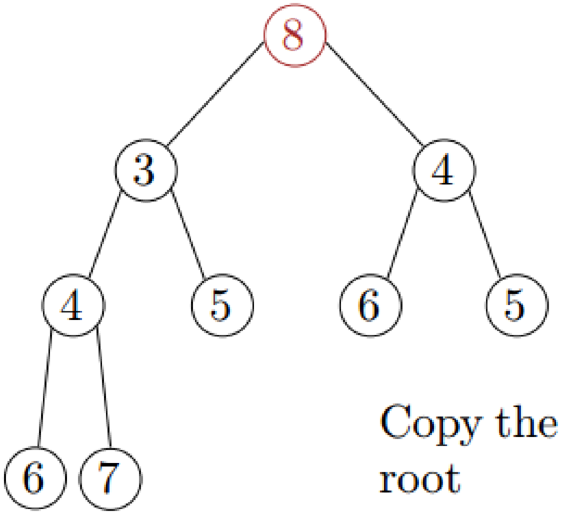
Reviewing Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Reviewing Extract-Min

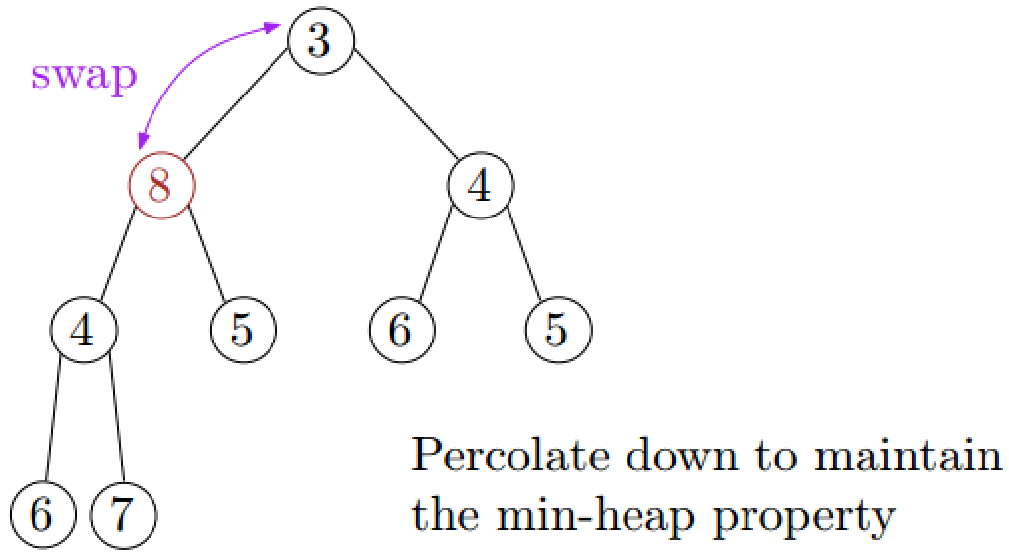
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Copy the last element to the root

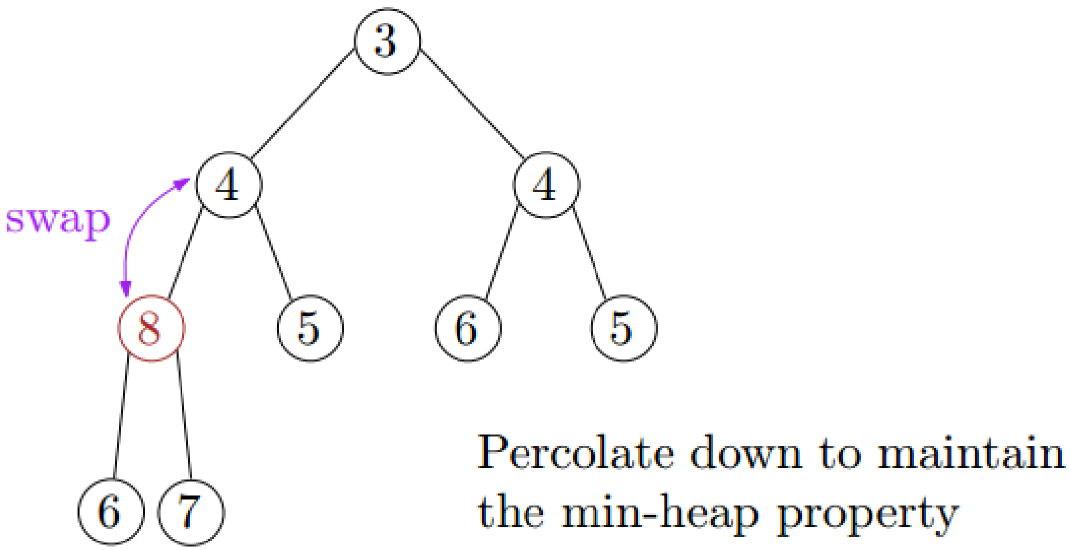
Reviewing Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



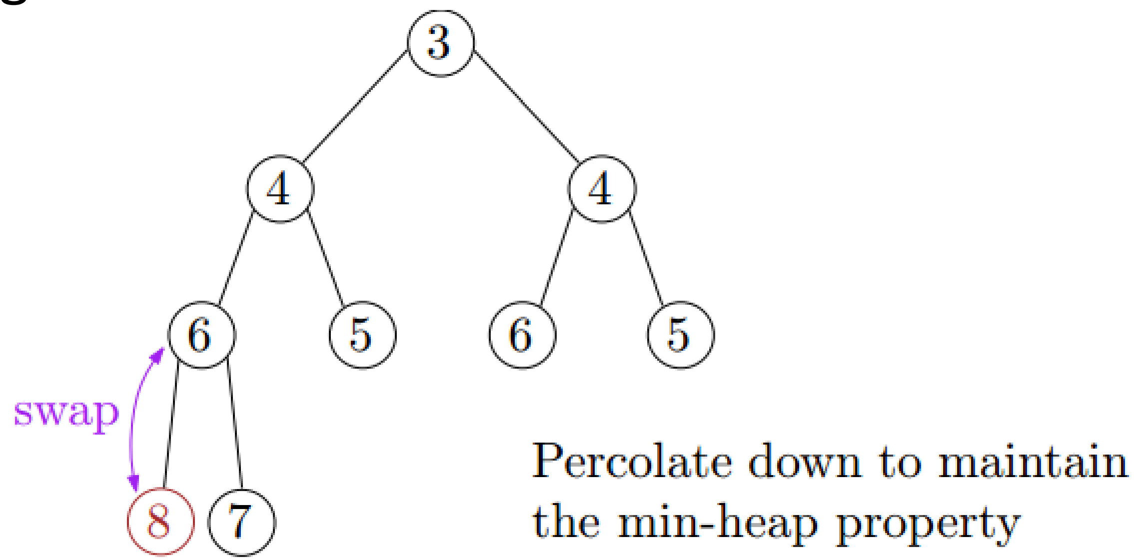
Reviewing Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Reviewing Extract-Min

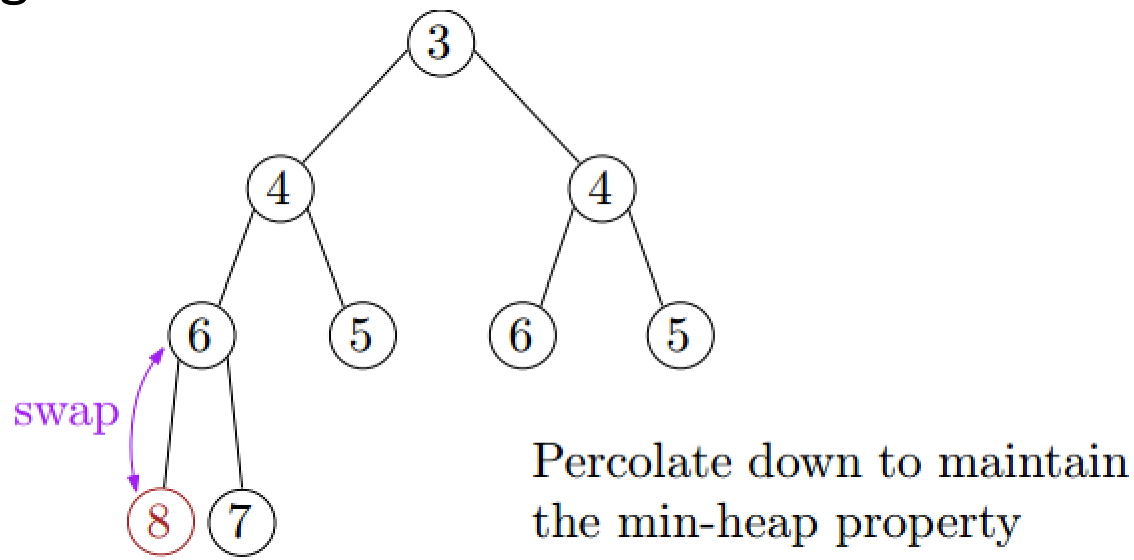
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)

Reviewing Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)
- Time complexity = O(height) = O(log n)

Using a Priority Queue to Find the Lightest Edge

Each item of the queue is a pair (u, key[u]),

Using a Priority Queue to Find the Lightest Edge

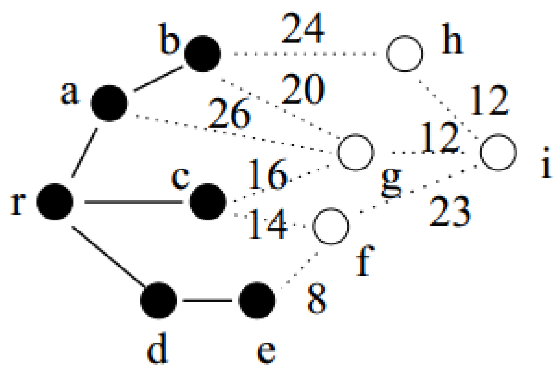
Each item of the queue is a pair (u, key[u]), where

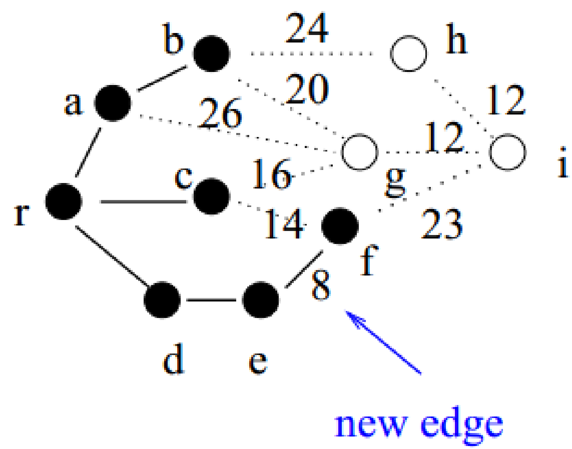
- u is a vertex in V\S,

Using a Priority Queue to Find the Lightest Edge

Each item of the queue is a pair (u, key[u]), where

- u is a vertex in V\S,
- key[u] is the weight of the lightest edge from u to any vertex in S. (The endpoint of this edge in S is stored in pred[u], which is used to build the MST tree.)





key[f] = 8, pred[f] = e

key[i] = infinity, pred[i] = nil

key[g] = 16, pred[g] = c

key[h] = 24, pred[h] = b

→ f has the minimum key

key[i] = 23, pred[i] = f

After adding the new edge and vertex f, update the key[v] and pred[v] for each vertex v adjacent to f

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

| $color[u] \leftarrow$

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

| $color[u] \leftarrow$ WHITE,$key[u] \leftarrow$

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

| $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

| $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

 | $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex

$Q \leftarrow$ new PriQueue(V);// put vertices in Q

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex

$Q \leftarrow$ new PriQueue(V);// put vertices in Q

while Q *is nonempty* **do**

  $u \leftarrow Q$.Extract-Min();// lightest edge

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex

$Q \leftarrow$ new PriQueue(V);// put vertices in Q

while Q *is nonempty* **do**

  $u \leftarrow Q$.Extract-Min();// lightest edge

  for $v \in adj[u]$ **do**

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

```
Let color[1...|V|], key[1...|V|], pred[1...|V|] be new arrays;
for u ∈ V do
    color[u] ← WHITE, key[u] ← +∞; // Initialize
end
key[r] ← 0, pred[r] ← NULL; // Start at root vertex
Q ← new PriQueue(V); // put vertices in Q
while Q is nonempty do
    u ← Q.Extract-Min(); // lightest edge
    for v ∈ adj[u] do
        if (color[v] ← WHITE)&&(w[u,v] < key[v]) then
            key[v] ← w[u,v]; // new lightest edge
```

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

```
Let color[1...|V|], key[1...|V|], pred[1...|V|] be new arrays;
for u ∈ V do
|  color[u] ← WHITE, key[u] ← +∞; // Initialize
end
key[r] ← 0, pred[r] ← NULL; // Start at root vertex
Q ← new PriQueue(V); // put vertices in Q
while Q is nonempty do
|  u ← Q.Extract-Min(); // lightest edge
|  for v ∈ adj[u] do
|  |  if (color[v] ← WHITE)&&(w[u, v] < key[v]) then
|  |  |  key[v] ← w[u, v]; // new lightest edge
|  |  |  Q.Decrease-Key(v, key[v]);
```

Description of Prim's Algorithm

Prim(G,w,r)

Input: A graph ***G***, a matrix ***w*** representing the weights between vertices in G, the algorithm will start at root vertex ***r***

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

$\quad color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// Initialize

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;// Start at root vertex

$Q \leftarrow$ new PriQueue(V);// put vertices in Q

while Q *is nonempty* **do**

$\quad u \leftarrow Q$.Extract-Min();// lightest edge

$\quad$ **for** $v \in adj[u]$ **do**

$\quad\quad$ **if** $(color[v] \leftarrow WHITE)\&\&(w[u,v] < key[v])$ **then**

$\quad\quad\quad key[v] \leftarrow w[u,v]$;// new lightest edge

$\quad\quad\quad Q$.Decrease-Key$(v, key[v])$;

$\quad\quad\quad pred[v] \leftarrow u$;

Description of Prim's Algorithm

Prim(G,w,r)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, the algorithm will start at root vertex r
Output: None
Let color[1...|V|], key[1...|V|], pred[1...|V|] be new arrays;
for u ∈ V do
|   color[u] ← WHITE,key[u] ← +∞;// Initialize
end
key[r] ← 0, pred[r] ← NULL;// Start at root vertex
Q ← new PriQueue(V);// put vertices in Q
while Q is nonempty do
|   u ← Q.Extract-Min();// lightest edge
|   for v ∈ adj[u] do
|   |   if (color[v] ← WHITE)&&(w[u, v] < key[v]) then
|   |   |   key[v] ← w[u, v];// new lightest edge
|   |   |   Q.Decrease-Key(v, key[v]);
|   |   |   pred[v] ← u;
|   |   end
|   end
|   color[u] ← BLACK;
end
```

Prim's Example

color

W	**W**	**W**	**W**	**W**	**W**	**W**

pred

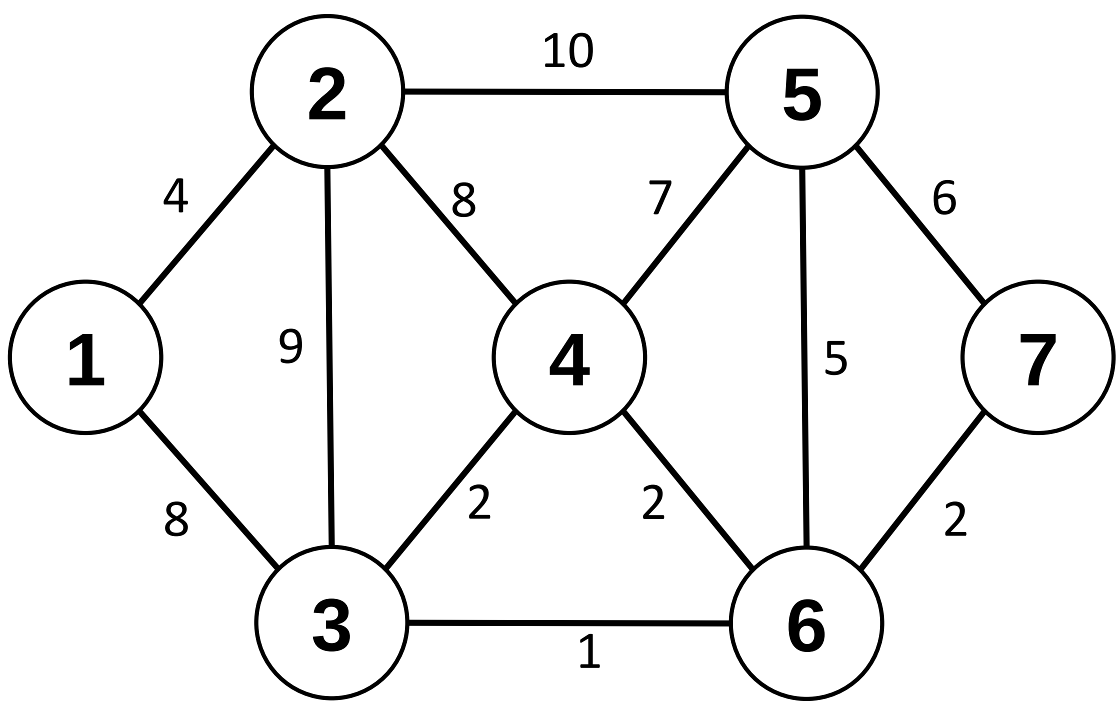
N	**N**	**N**	**N**	**N**	**N**	**N**

key

∞	**∞**	**∞**	**∞**	**∞**	**∞**	**∞**

Q

1,∞	**2,∞**	**3,∞**	**4,∞**	**5,∞**	**6,∞**	**7,∞**



Prim's Example

color

W	**W**	**W**	**W**	**W**	**W**	**W**

pred

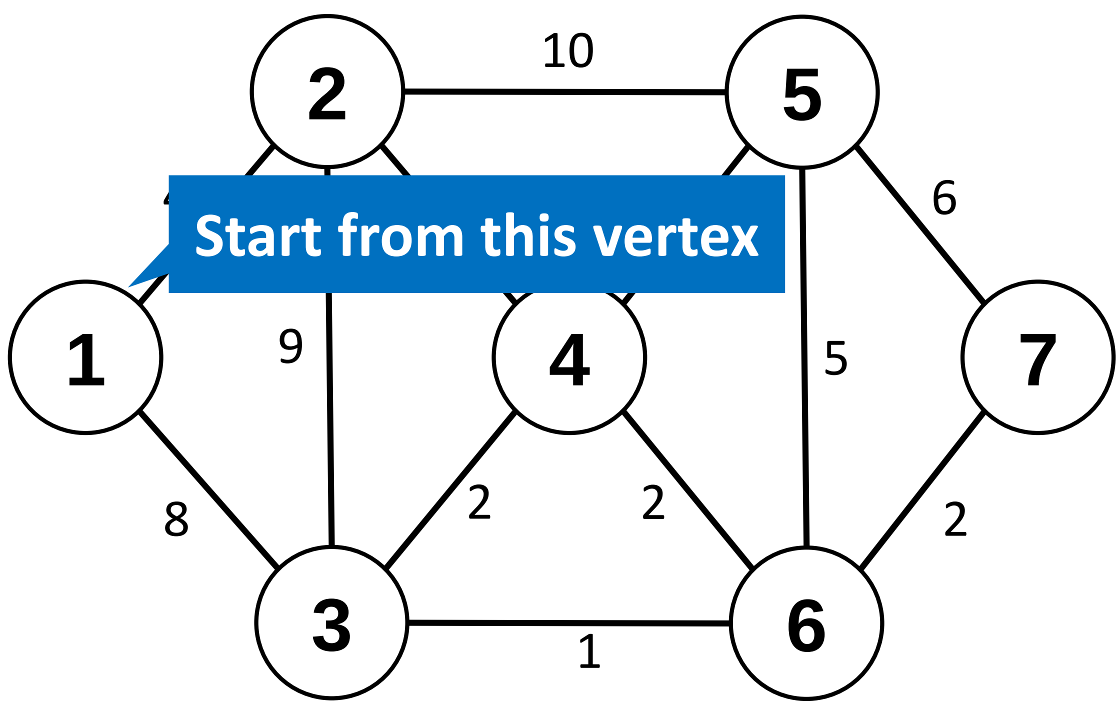
N	**N**	**N**	**N**	**N**	**N**	**N**

key

∞	∞	∞	∞	∞	∞	∞

Q

1,∞	2,∞	3,∞	4,∞	5,∞	6,∞	7,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

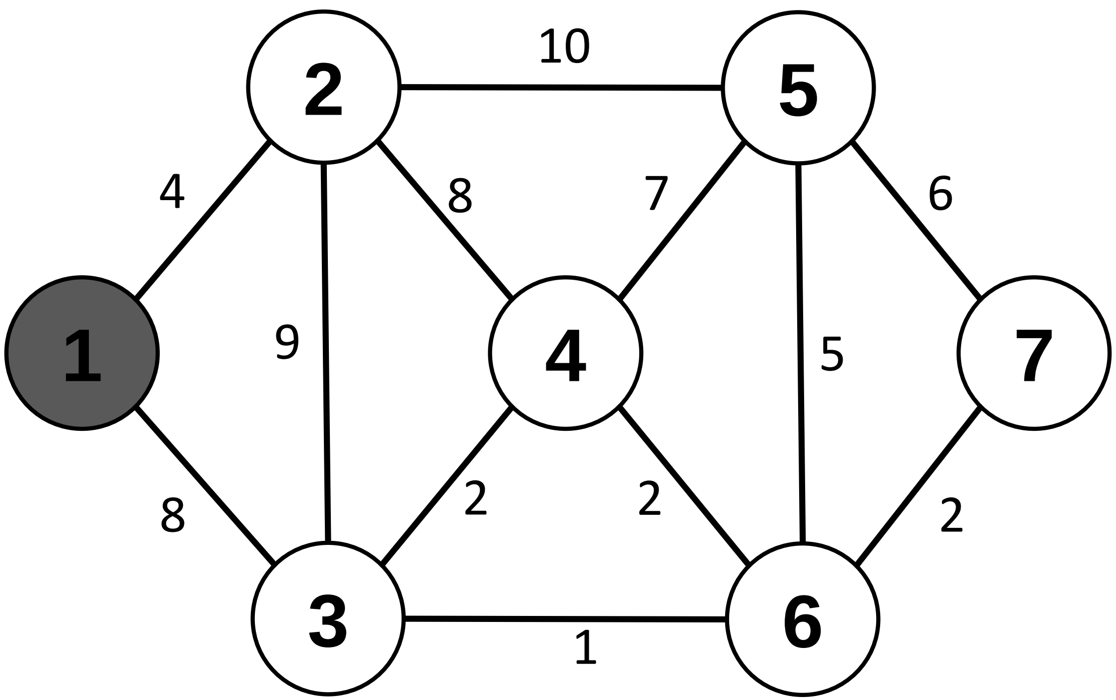
N	N	N	N	N	N	N

key

0	∞	∞	∞	∞	∞	∞

Q

1,0	2,∞	3,∞	4,∞	5,∞	6,∞	7,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

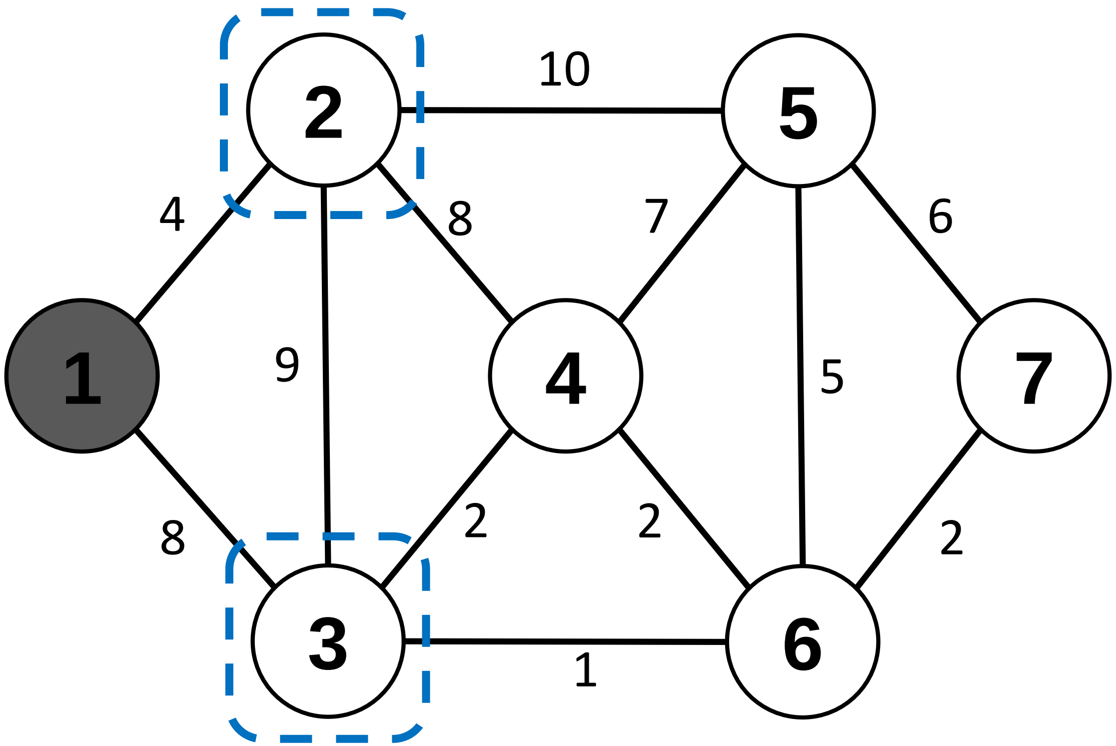
N	N	N	N	N	N	N

key

0	∞	∞	∞	∞	∞	∞

Q

7,∞	2,∞	3,∞	4,∞	5,∞	6,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

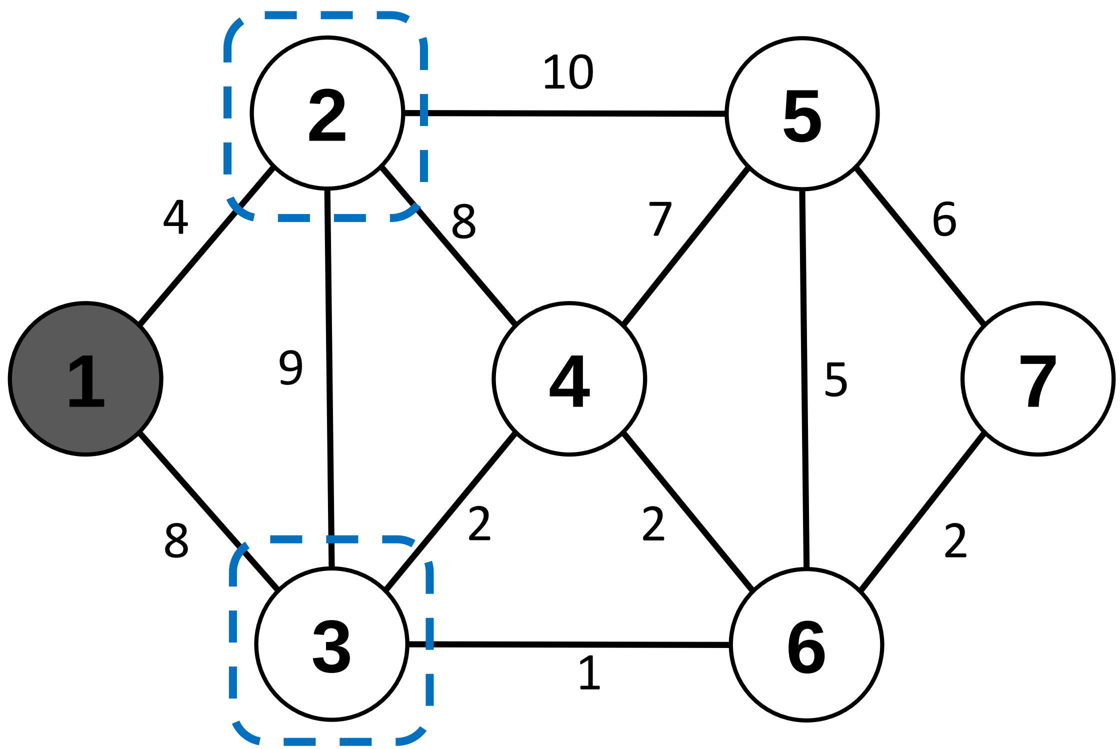
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

7,∞	2,4	3,8	4,∞	5,∞	6,∞



Prim's Example

color

B	W	W	W	W	W	W

pred

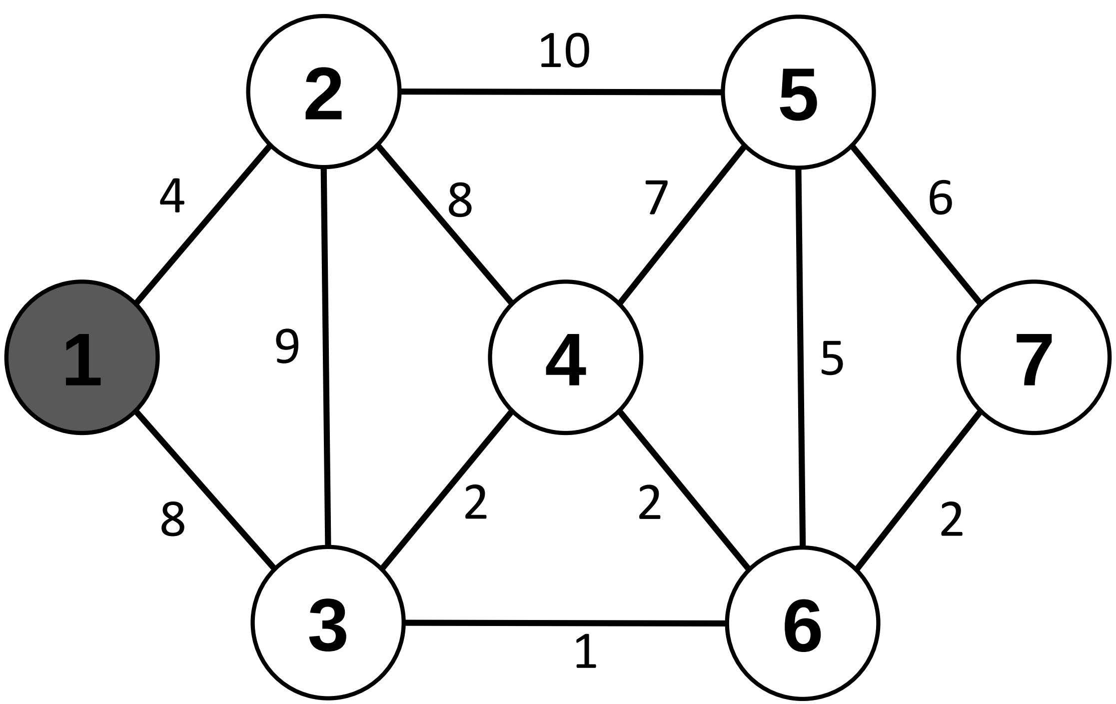
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

2,4	7,∞	3,8	4,∞	5,∞	6,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

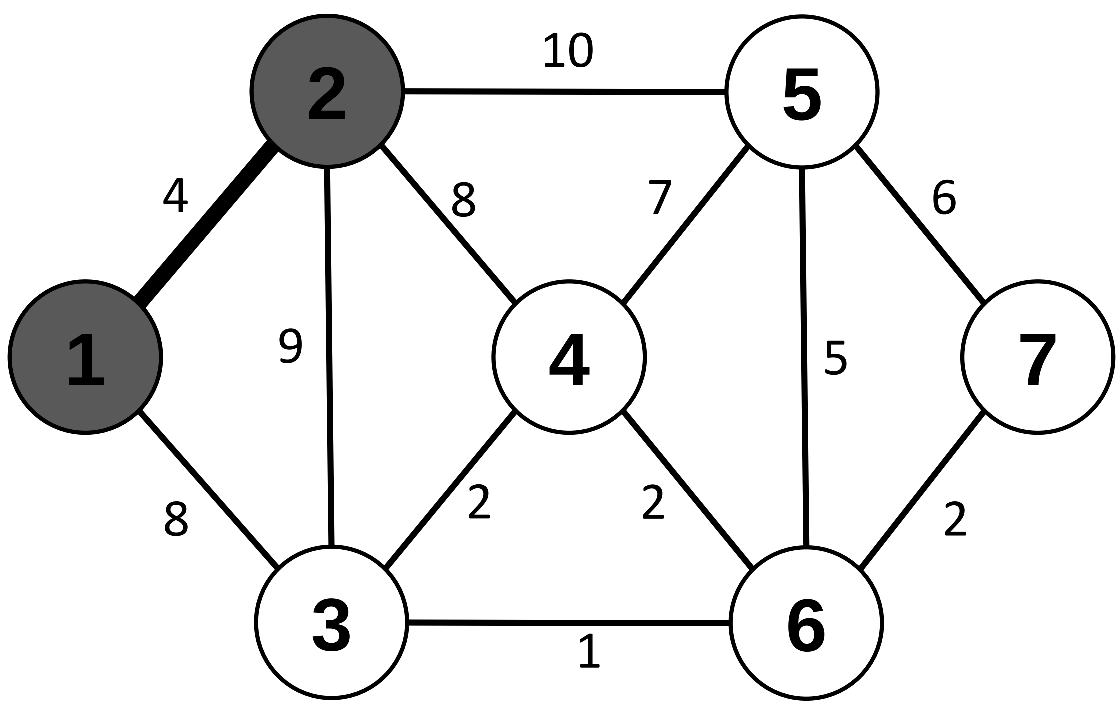
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

3,8	7,∞	6,∞	4,∞	5,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

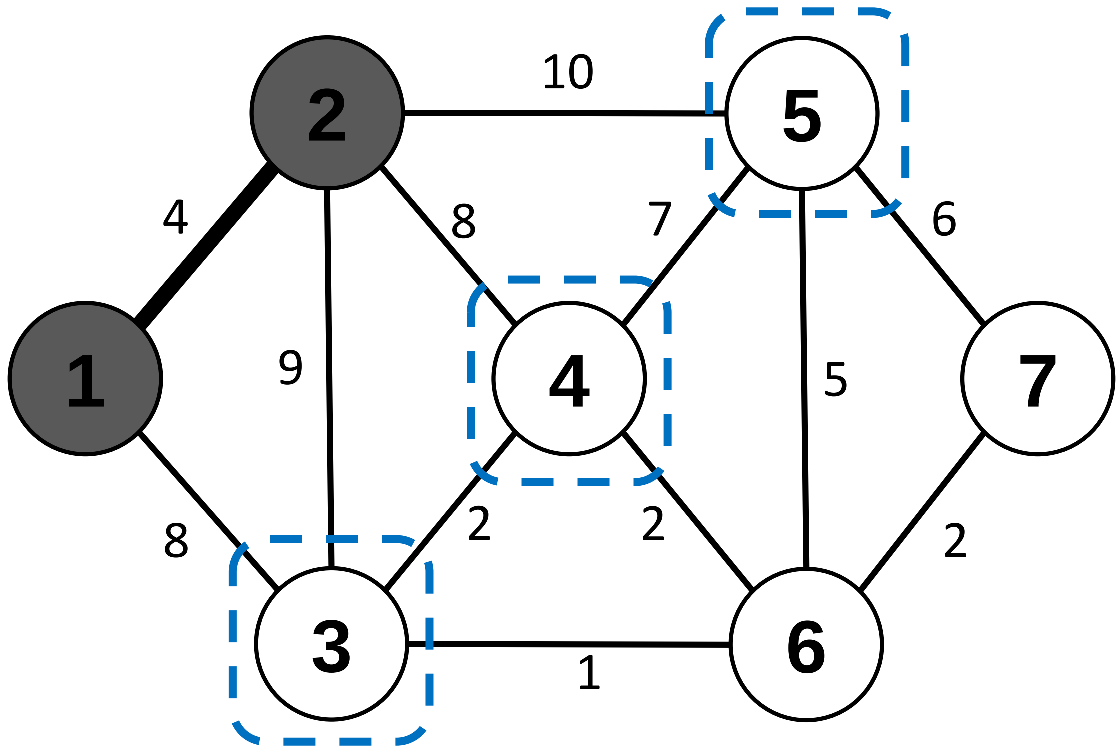
N	1	1	N	N	N	N

key

0	4	8	∞	∞	∞	∞

Q

3,8	7,∞	6,∞	4,∞	5,∞



Prim's Example

color

B	B	W	W	W	W	W

pred

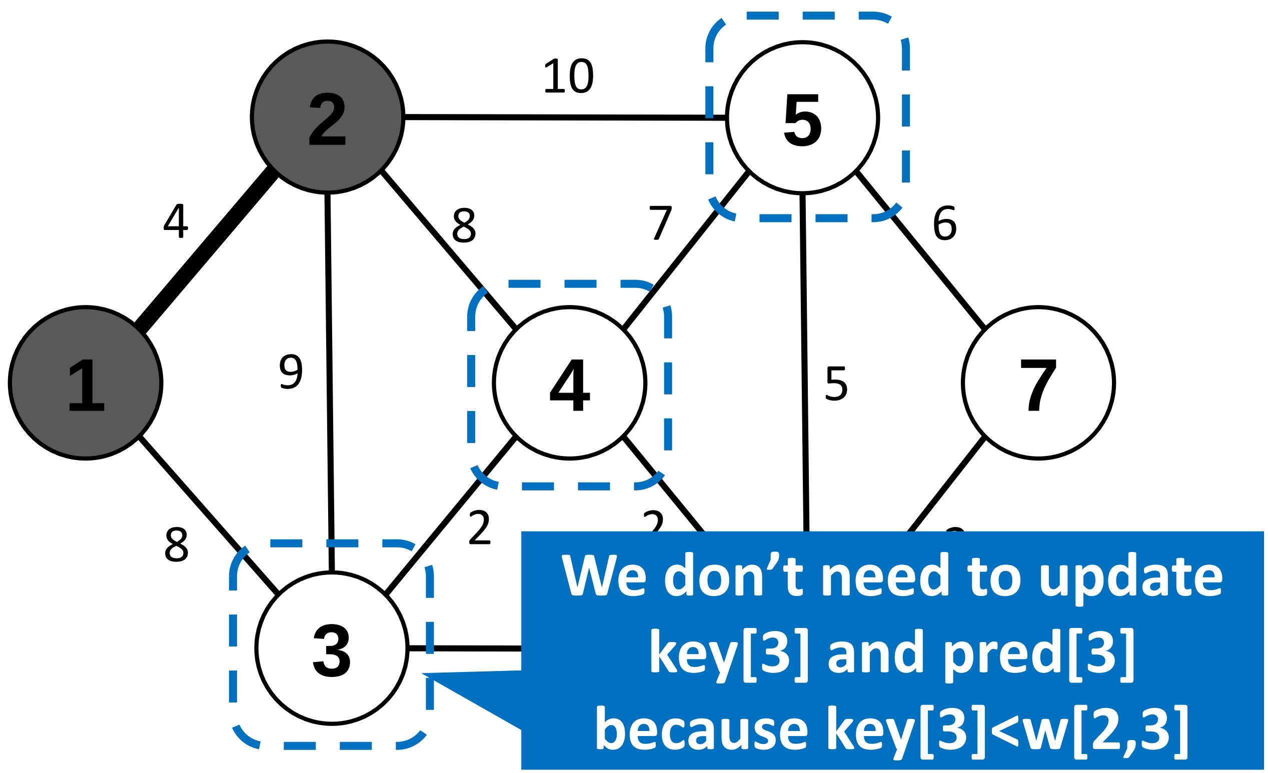
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

3,8	7,∞	6,∞	4,8	5,10



Prim's Example

color

B	B	W	W	W	W	W

pred

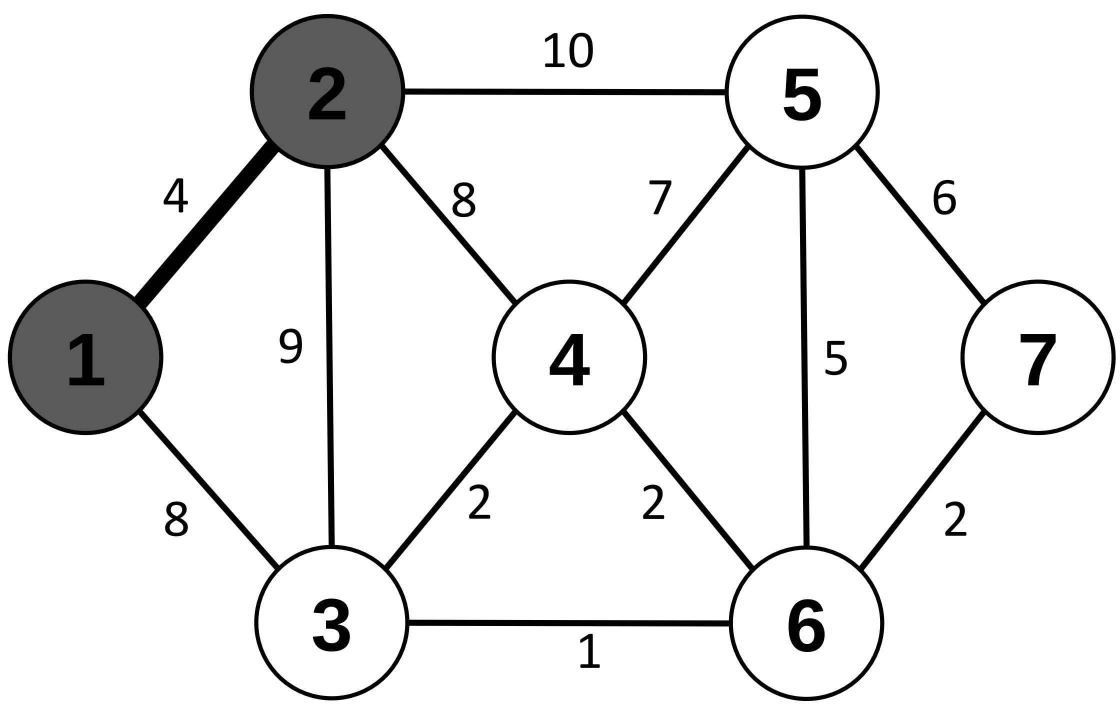
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

3,8	4,8	6,∞	7,∞	5,10



Prim's Example

color

B	B	B	W	W	W	W

pred

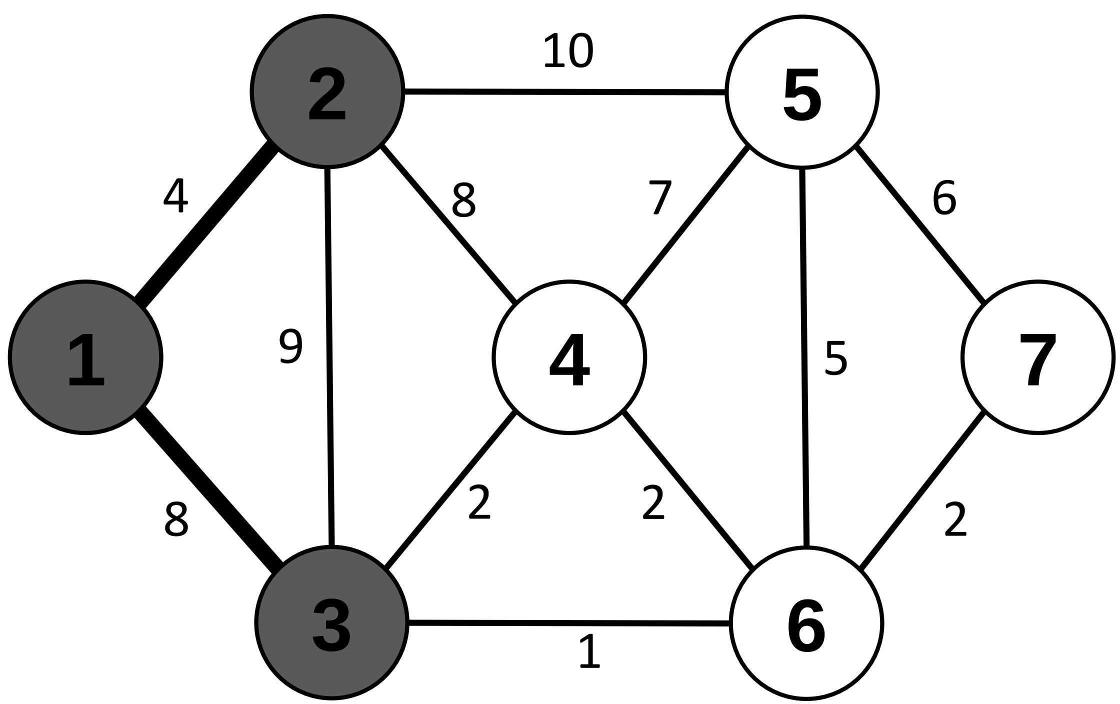
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

4,8	5,10	6,∞	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

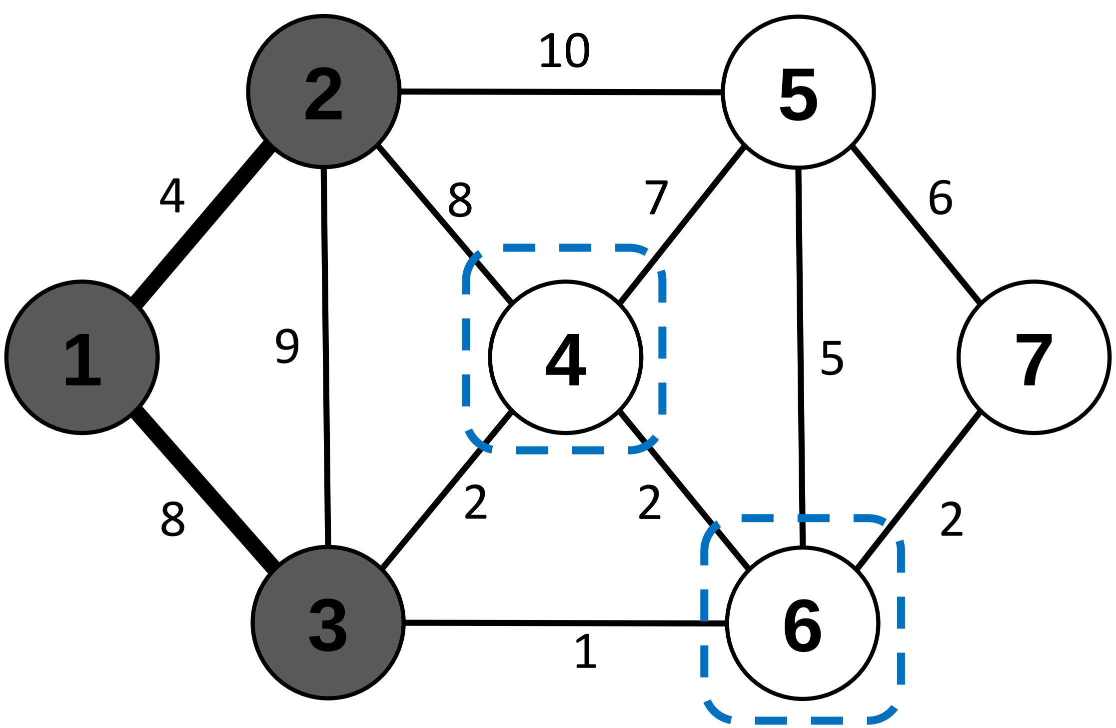
N	1	1	2	2	N	N

key

0	4	8	8	10	∞	∞

Q

4,8	5,10	6,∞	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

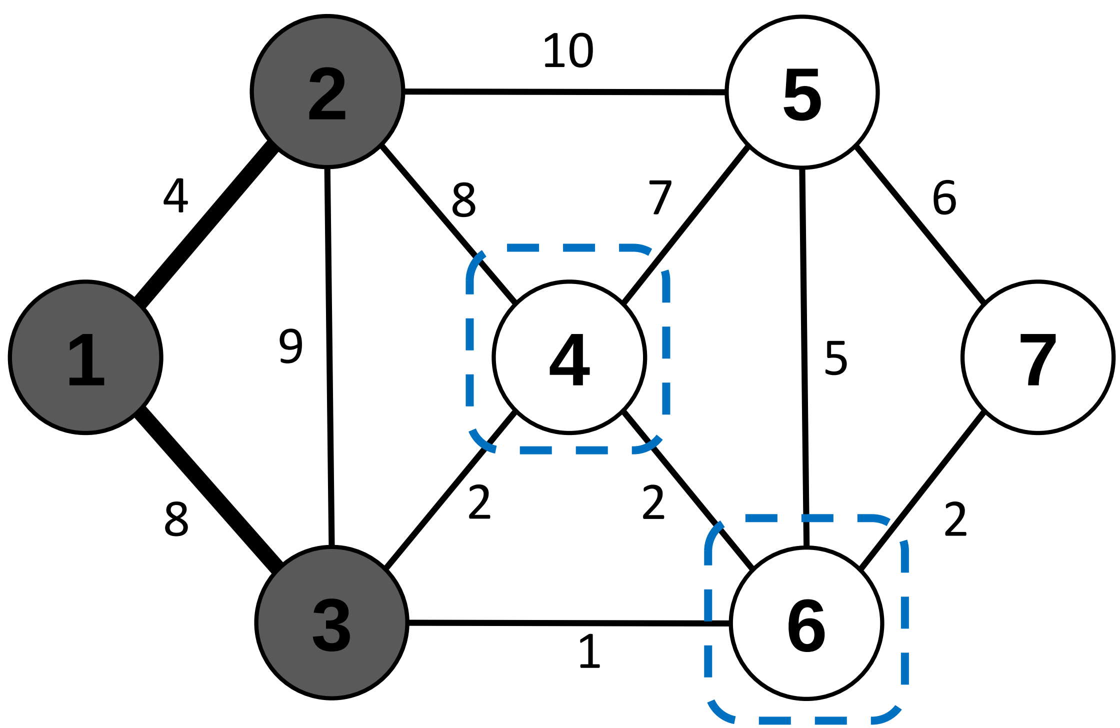
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	6,1	7,∞



Prim's Example

color

B	B	B	W	W	W	W

pred

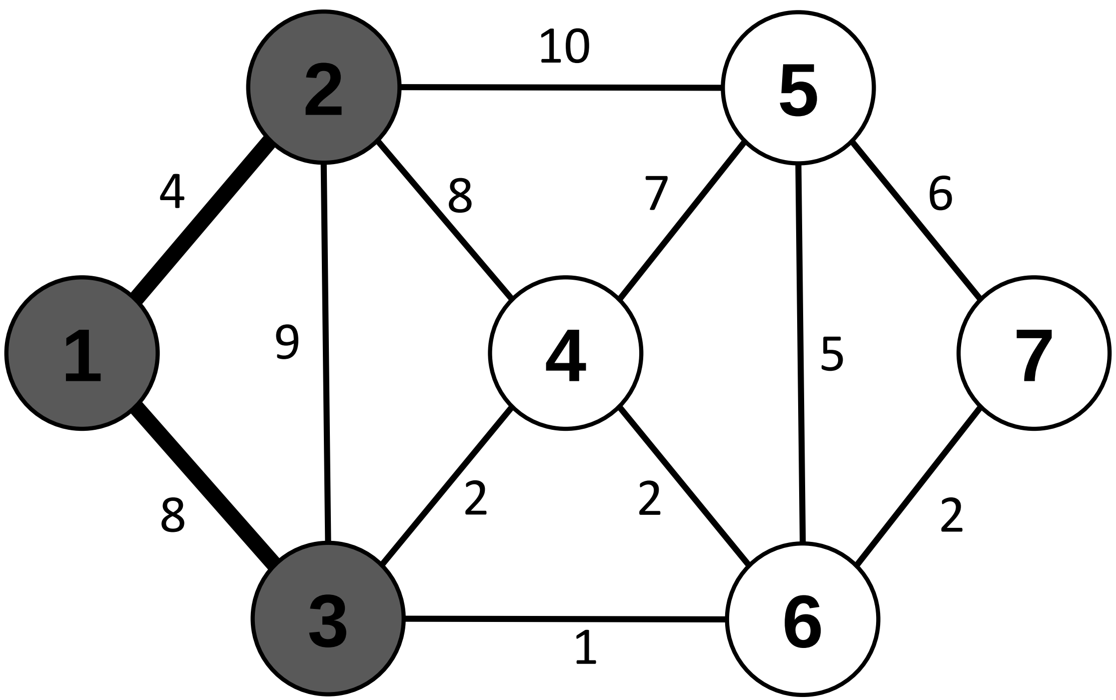
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

6,1	5,10	4,2	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

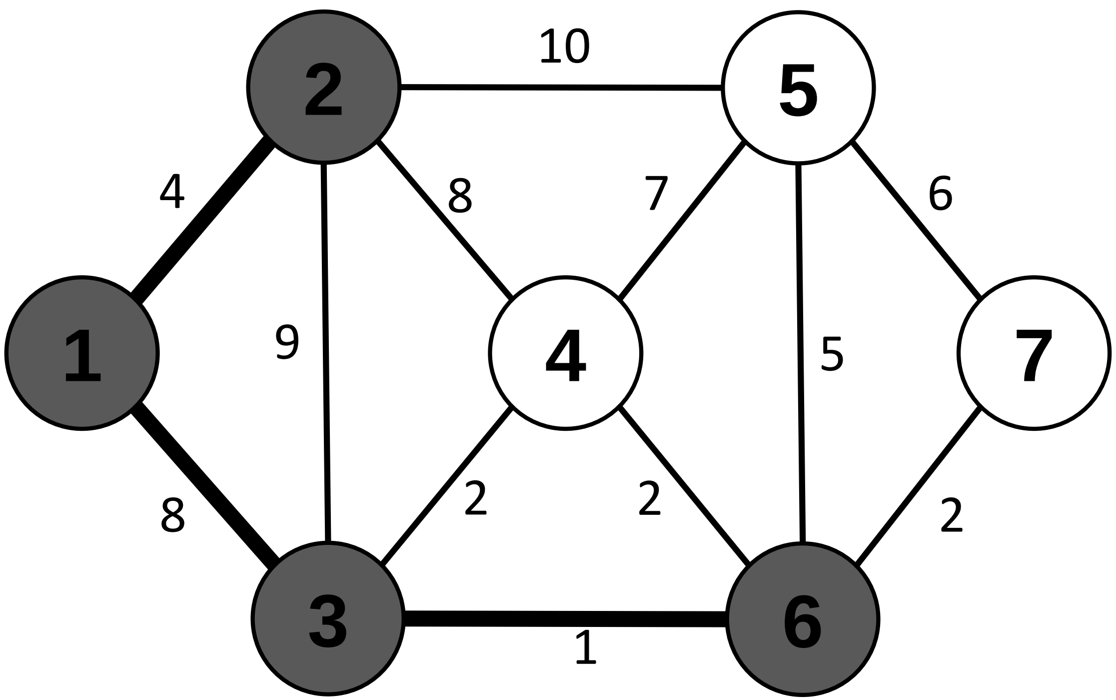
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

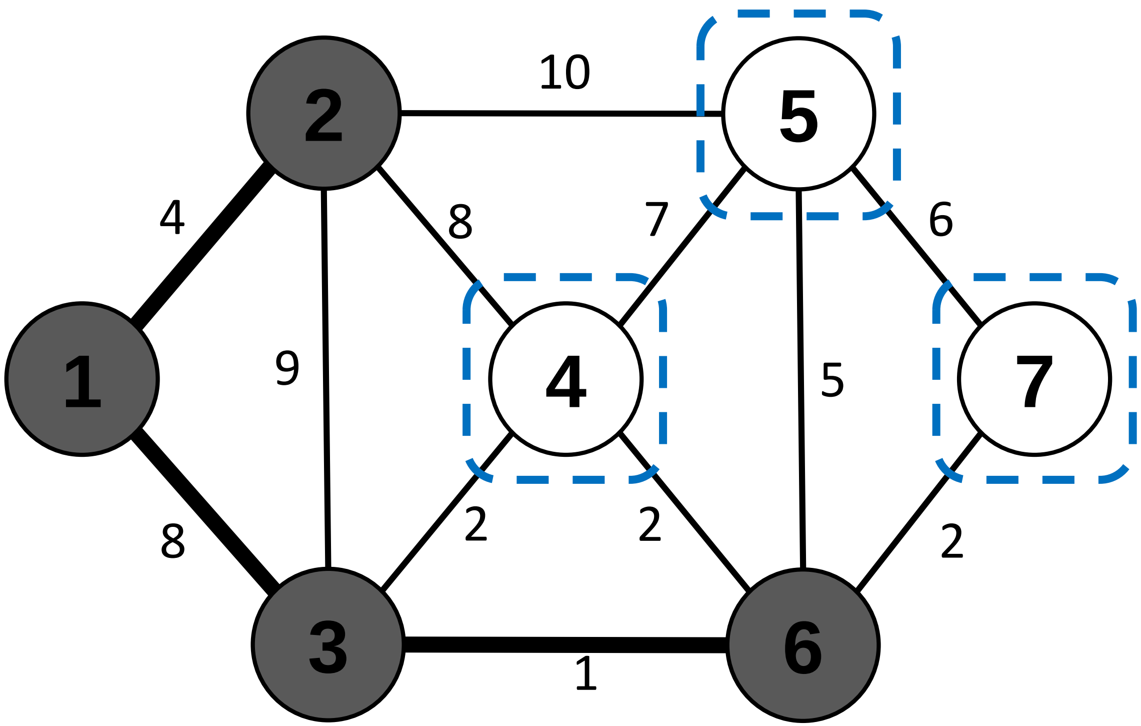
N	1	1	3	2	3	N

key

0	4	8	2	10	1	∞

Q

4,2	5,10	7,∞



Prim's Example

color

B	B	B	W	W	B	W

pred

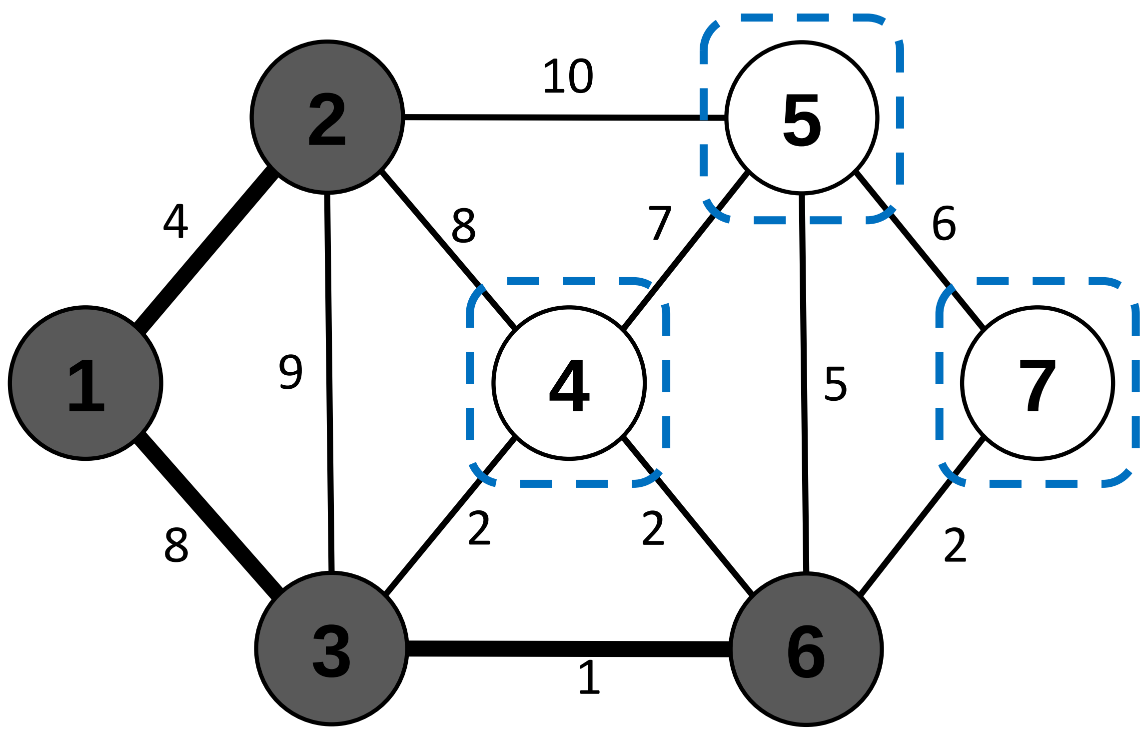
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

4,2	5,5	7,2



Prim's Example

color

B	B	B	W	W	B	W

pred

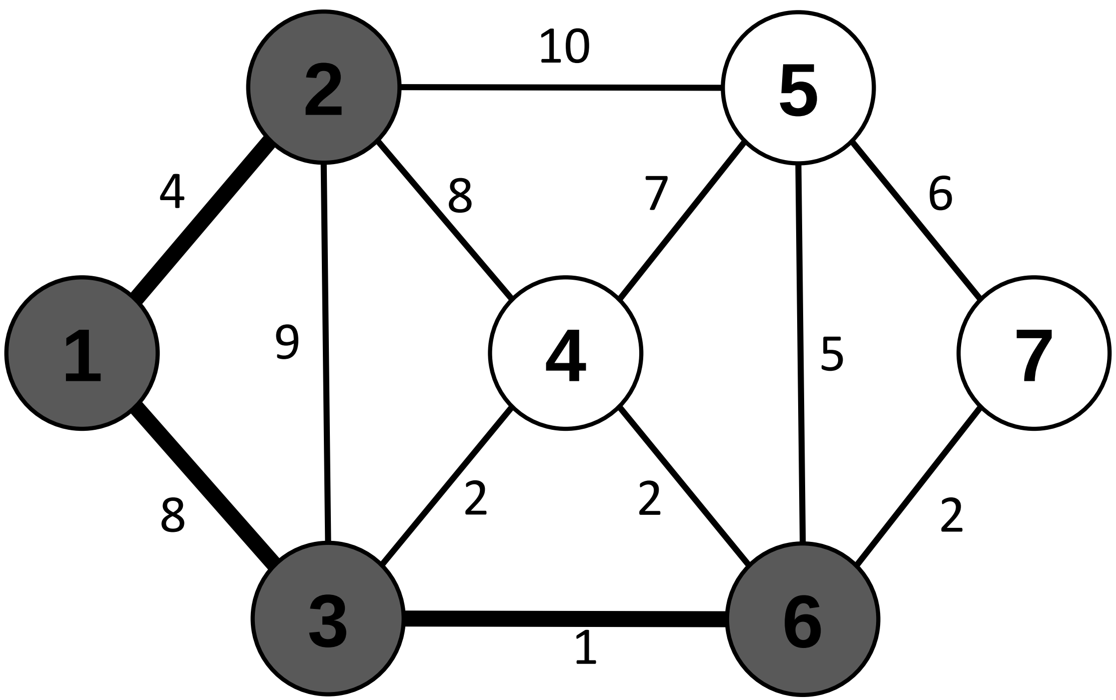
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

4,2	5,5	7,2



Prim's Example

color

B	B	B	B	W	B	W

pred

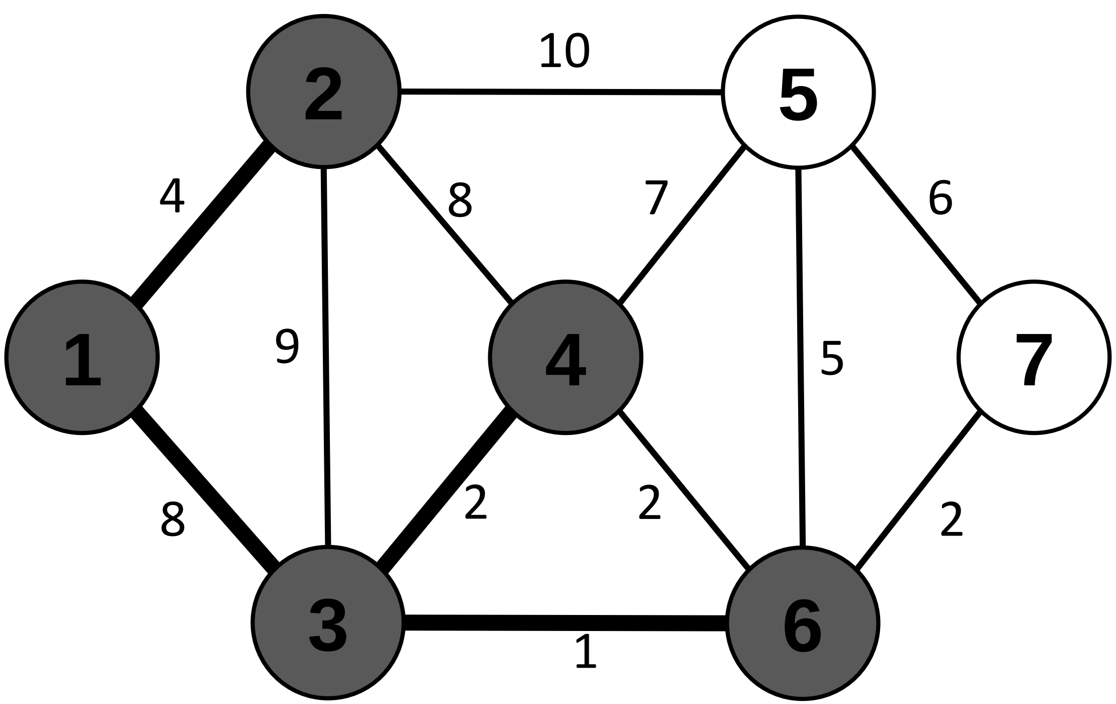
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



Prim's Example

color

B	B	B	B	W	B	W

pred

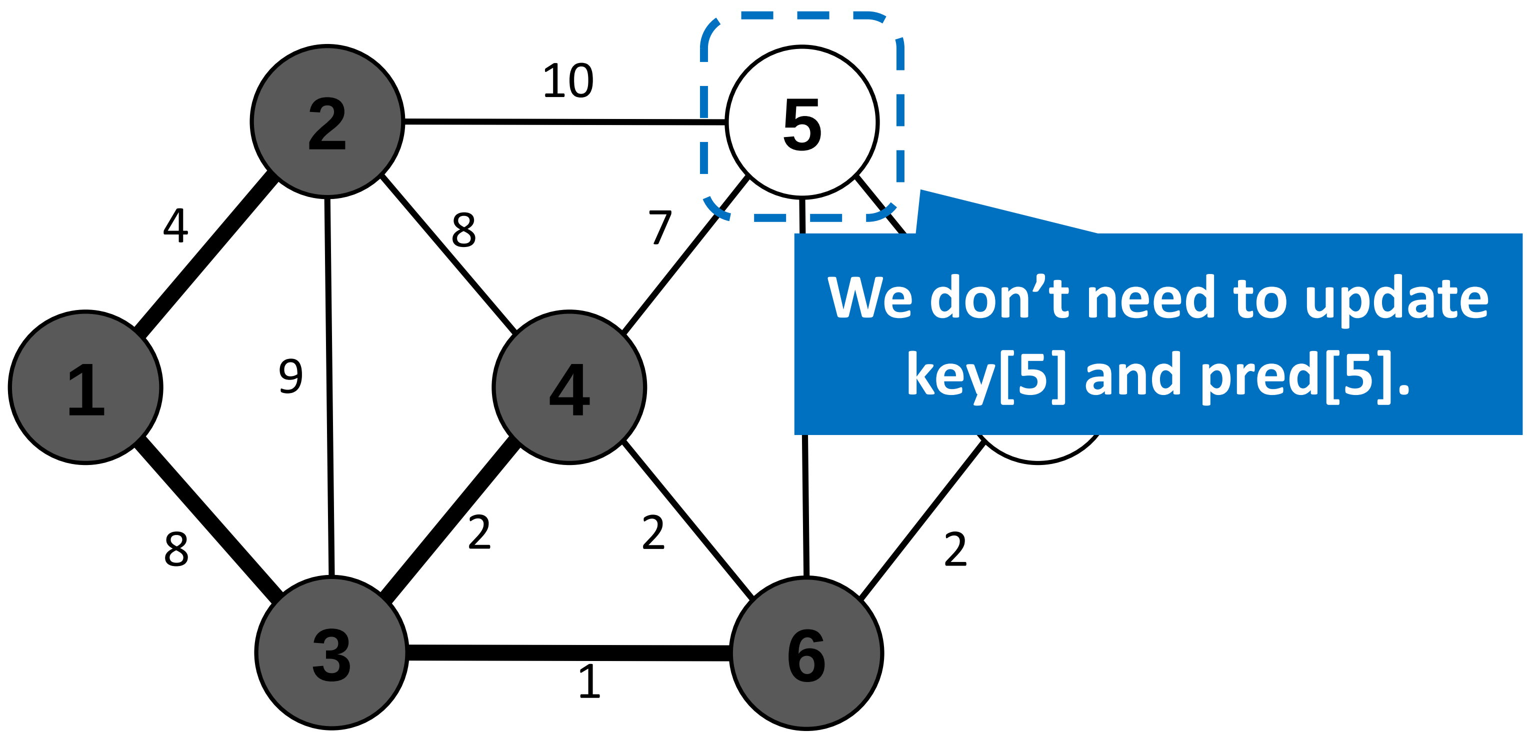
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



Prim's Example

color

B	B	B	B	W	B	W

pred

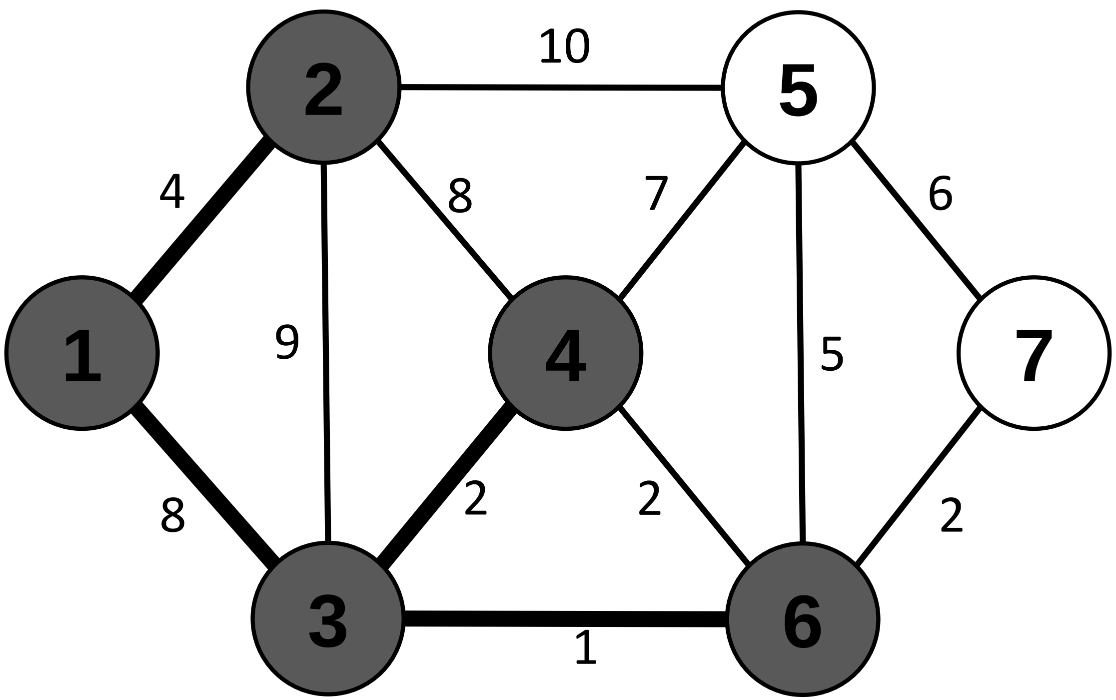
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

7,2	5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

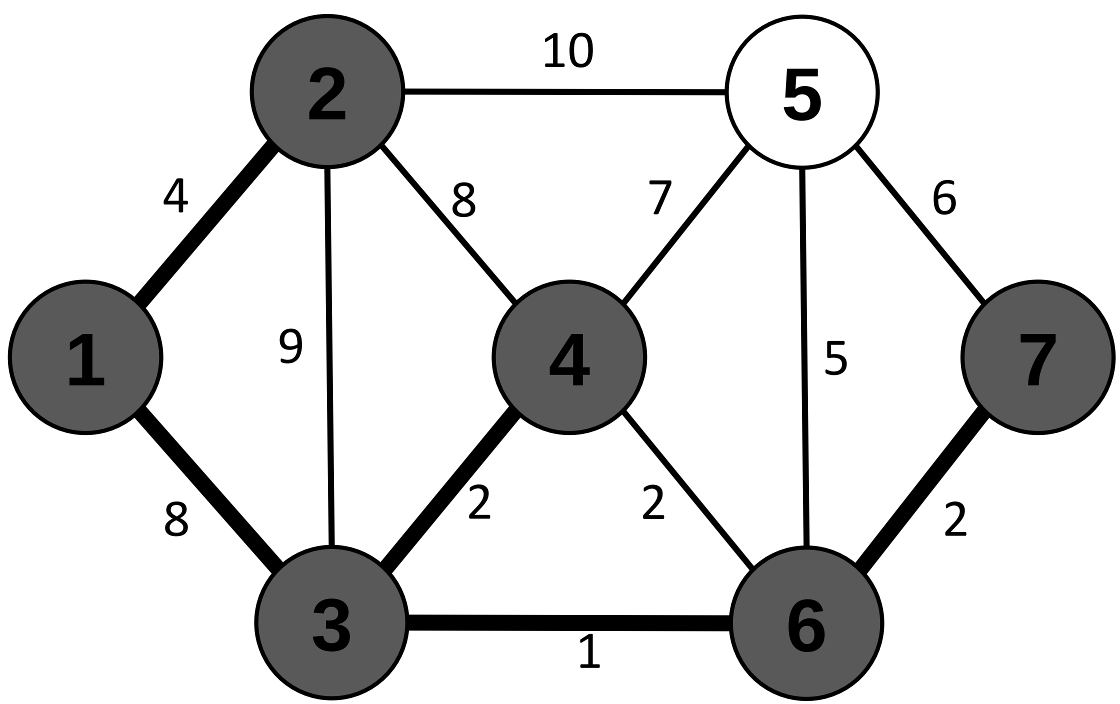
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

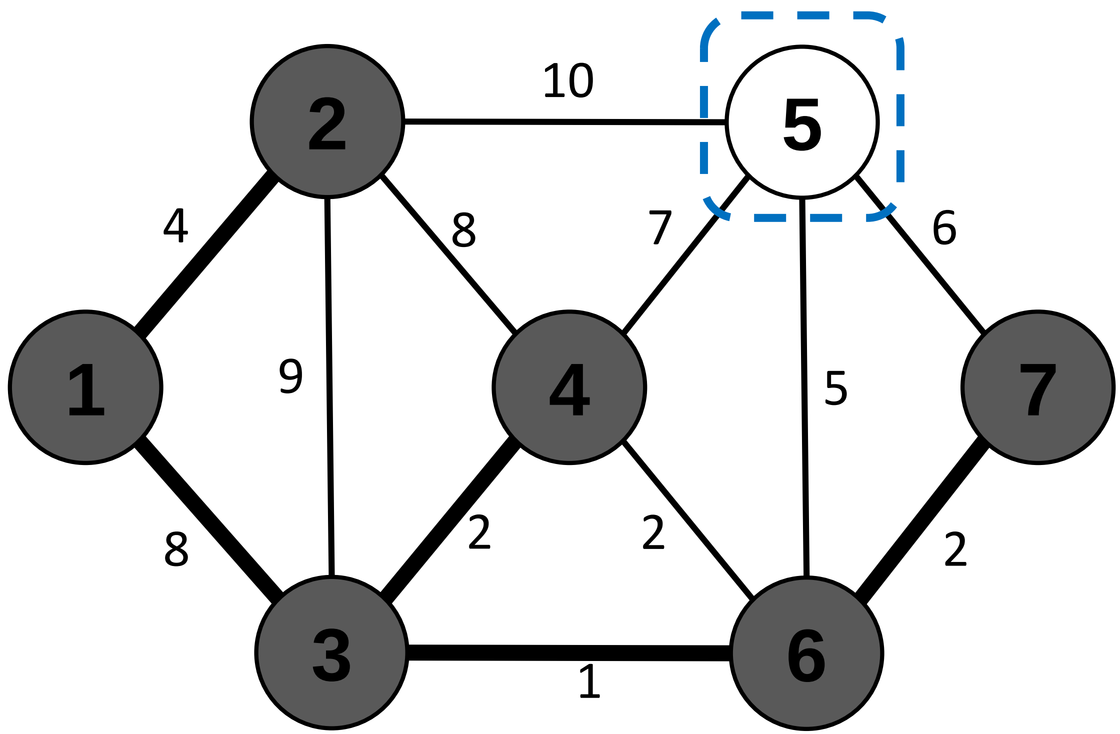
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	B	B	B	W	B	B

pred

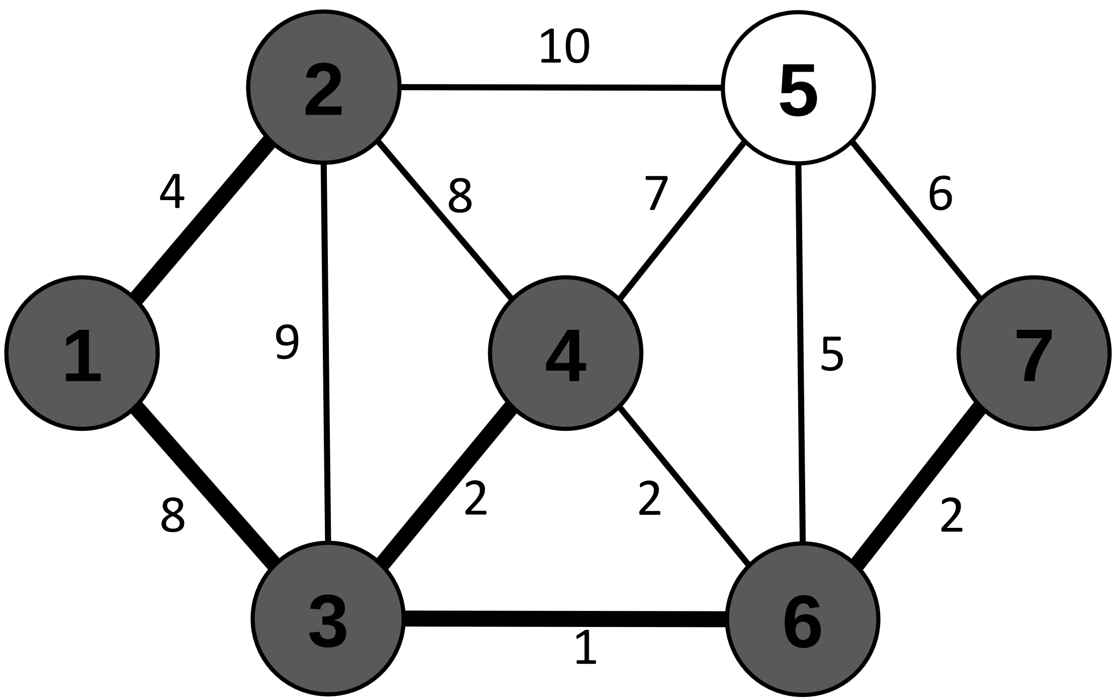
N	1	1	3	6	3	6

key

0	4	8	2	5	1	2

Q

5,5



Prim's Example

color

B	**B**	**B**	**B**	**B**	**B**	**B**

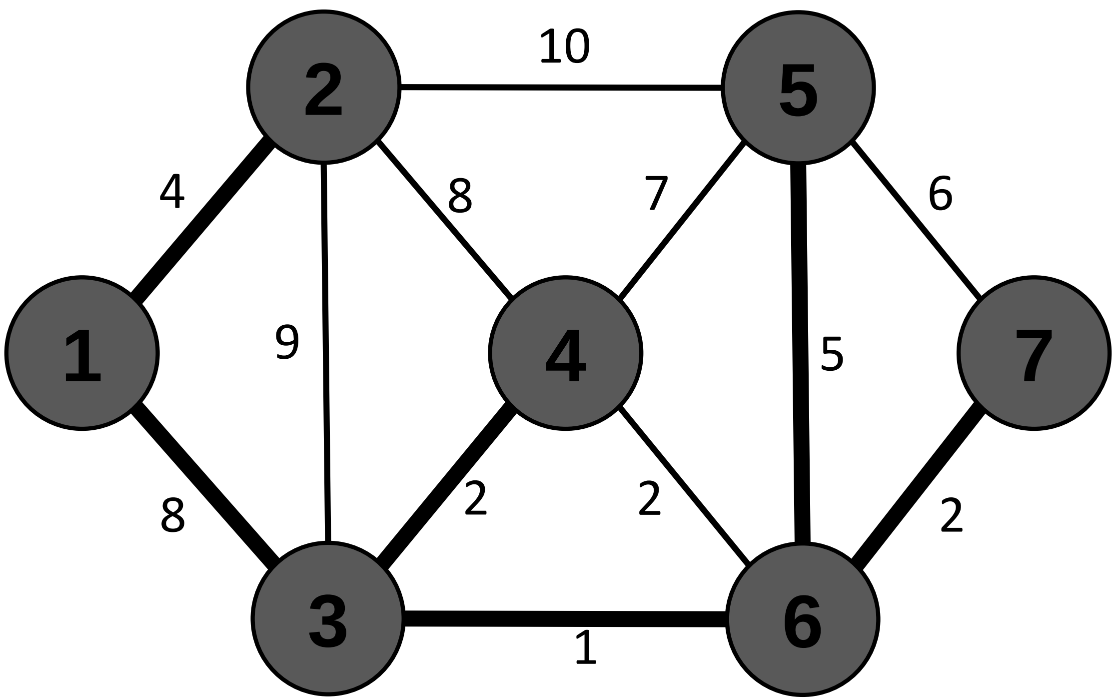
pred

N	**1**	**1**	**3**	**6**	**3**	**6**

key

0	4	8	2	5	1	2

Q



Prim's Example

color

B	**B**	**B**	**B**	**B**	**B**	**B**

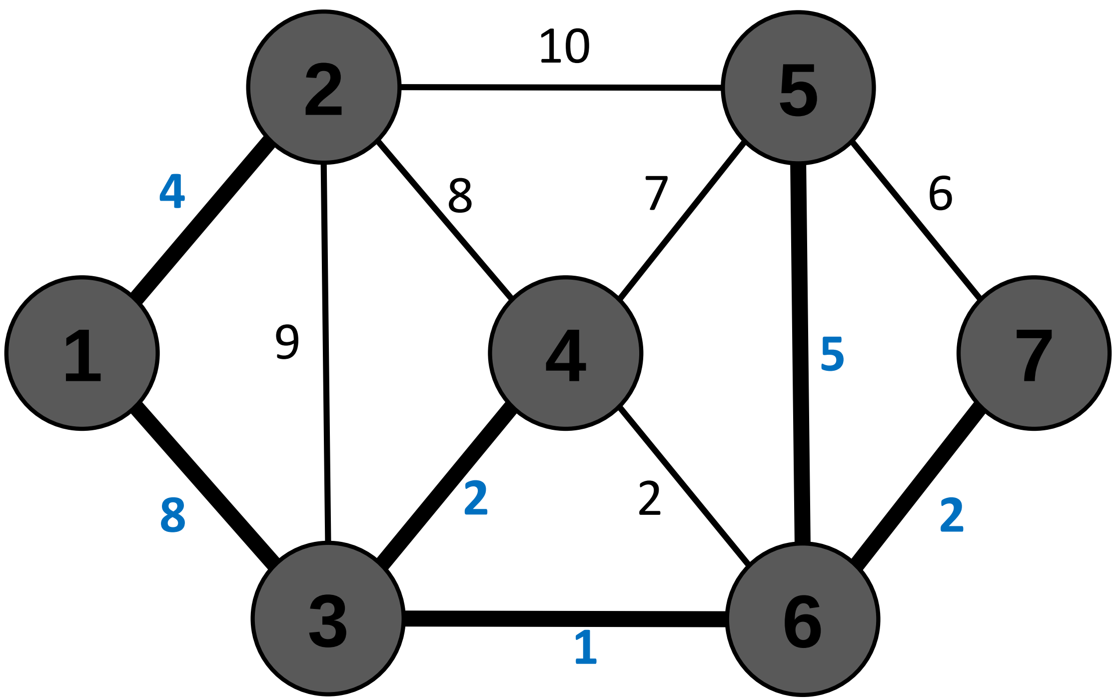
pred

N	**1**	**1**	**3**	**6**	**3**	**6**

key

0	**4**	**8**	**2**	**5**	**1**	**2**

Weight of MST = **22**



Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim's algorithm
  - The idea
  - The algorithm
  - Analysis for Prim's algorithm
- Kruskal's algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal's algorithm

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// $O(V)$

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

| $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// $O(V)$

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// $O(V)$

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;

$Q \leftarrow$ new PriQueue(V);// $O(V)$

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;

for $u \in V$ **do**

  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// $O(V)$

end

$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;

$Q \leftarrow$ new PriQueue(V);// $O(V)$

while Q *is nonempty* **do**

  $u \leftarrow Q$.Extract-Min();// $O(logV)$

Analysis of Prim's Algorithm...

Prim(G,w,r)

```
Input: A graph G, a matrix w representing the weights between vertices
       in G, the algorithm will start at root vertex r
Output: None
Let color[1...|V|], key[1...|V|], pred[1...|V|] be new arrays;
for u ∈ V do
 |  color[u] ← WHITE, key[u] ← +∞; // O(V)
end
key[r] ← 0, pred[r] ← NULL;
Q ← new PriQueue(V); // O(V)
while Q is nonempty do
 |  u ← Q.Extract-Min(); // O(logV)
 |  for v ∈ adj[u] do
 |   |  if (color[v] ← WHITE)&&(w[u, v] < key[v]) then
 |   |   |  key[v] ← w[u, v];
 |
```

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$
Output: None

Let $color[1...|V|], key[1...|V|], pred[1...|V|]$ be new arrays;
for $u \in V$ **do**
  $color[u] \leftarrow$ WHITE,$key[u] \leftarrow +\infty$;// $O(V)$
end
$key[r] \leftarrow 0$, $pred[r] \leftarrow$ NULL;
$Q \leftarrow$ new PriQueue(V);// $O(V)$
while Q *is nonempty* **do**
  $u \leftarrow Q$.Extract-Min();// $O(logV)$
  for $v \in adj[u]$ **do**
    if $(color[v] \leftarrow WHITE)\&\&(w[u,v] < key[v])$ **then**
      $key[v] \leftarrow w[u,v]$;
      Q.Decrease-Key($v, key[v]$);// $O(logV)$

Analysis of Prim's Algorithm...

Prim(G,w,r)

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G, the algorithm will start at root vertex $\boldsymbol{r}$

Output: None

```
Let color[1...|V|], key[1...|V|], pred[1...|V|] be new arrays;
for u ∈ V do
|   color[u] ← WHITE, key[u] ← +∞; // O(V)
end
key[r] ← 0, pred[r] ← NULL;
Q ← new PriQueue(V); // O(V)
while Q is nonempty do
|   u ← Q.Extract-Min(); // O(logV)
|   for v ∈ adj[u] do
|   |   if (color[v] ← WHITE)&&(w[u, v] < key[v]) then
|   |   |   key[v] ← w[u, v];
|   |   |   Q.Decrease-Key(v, key[v]); // O(logV)
|   |   |   pred[v] ← u;
|   |   end
|   end
|   color[u] ← BLACK;
end
```

Analysis of Prim's Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .

Analysis of Prim’s Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .
  Total cost: O(V log V)

Analysis of Prim's Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .
  Total cost: O(V log V)
- O(log V) time for Decrease-Key on a PriQueue of size at most E.

Analysis of Prim's Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .
  Total cost: O(V log V)
- O(log V) time for Decrease-Key on a PriQueue of size at most E.
  Total cost: O(E log V).

Analysis of Prim's Algorithm

The data structure PriQueue (heap) supports the following two operations: (See CLRS)

- O(log V) for Extract-Min on a PriQueue of size at most V .
  Total cost: O(V log V)
- O(log V) time for Decrease-Key on a PriQueue of size at most E.
  Total cost: O(E log V).

Total cost is then O((V + E) log V) = O(E log V)

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Recalling the Generic Algorithm

- Start with an empty graph.
- Try to add edges one at a time, always making sure that what is built remains acyclic.
- If we are sure at each step that the resulting graph is a subset of some minimum spanning tree, we are done.

Lemma

- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *Let A be a subset of E that is included in some minimum spanning tree for G.*

Let

- *$(S, V - S)$ be any cut of G that respects A*
- *(u, v) be a light edge crossing the cut $(S, V - S)$*

Then, edge (u, v) is safe for A.

Idea of Kruskal’s Algorithm

- Kruskal’s Algorithm is based directly on the generic algorithm.

Idea of Kruskal's Algorithm

- Kruskal's Algorithm is based directly on the generic algorithm.
- Unlike Prim's algorithm, which grows one tree, Kruskal's algorithm grows a collection of trees (a forest).

Idea of Kruskal’s Algorithm

- Kruskal’s Algorithm is based directly on the generic algorithm.
- Unlike Prim’s algorithm, which grows one tree, Kruskal’s algorithm grows a collection of trees (a forest).
- Initially, this forest consists of the vertices only (no edges).

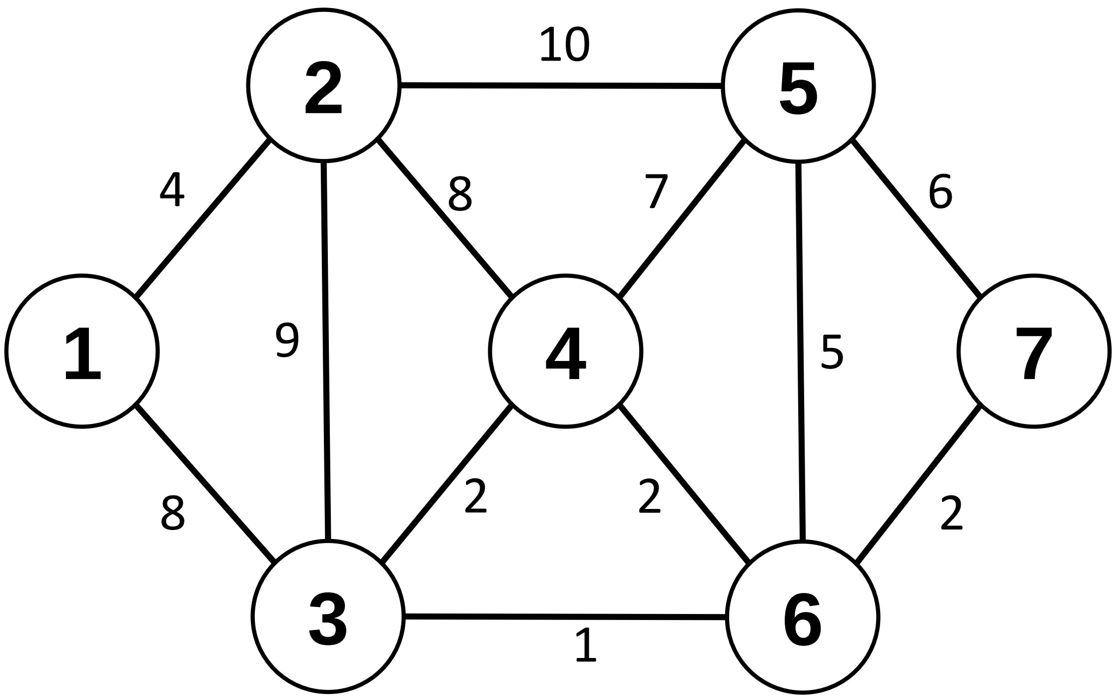
Idea of Kruskal’s Algorithm

- Kruskal’s Algorithm is based directly on the generic algorithm.
- Unlike Prim’s algorithm, which grows one tree, Kruskal’s algorithm grows a collection of trees (a forest).
- Initially, this forest consists of the vertices only (no edges).
- In each step the cheapest edge that does not create a cycle is added.

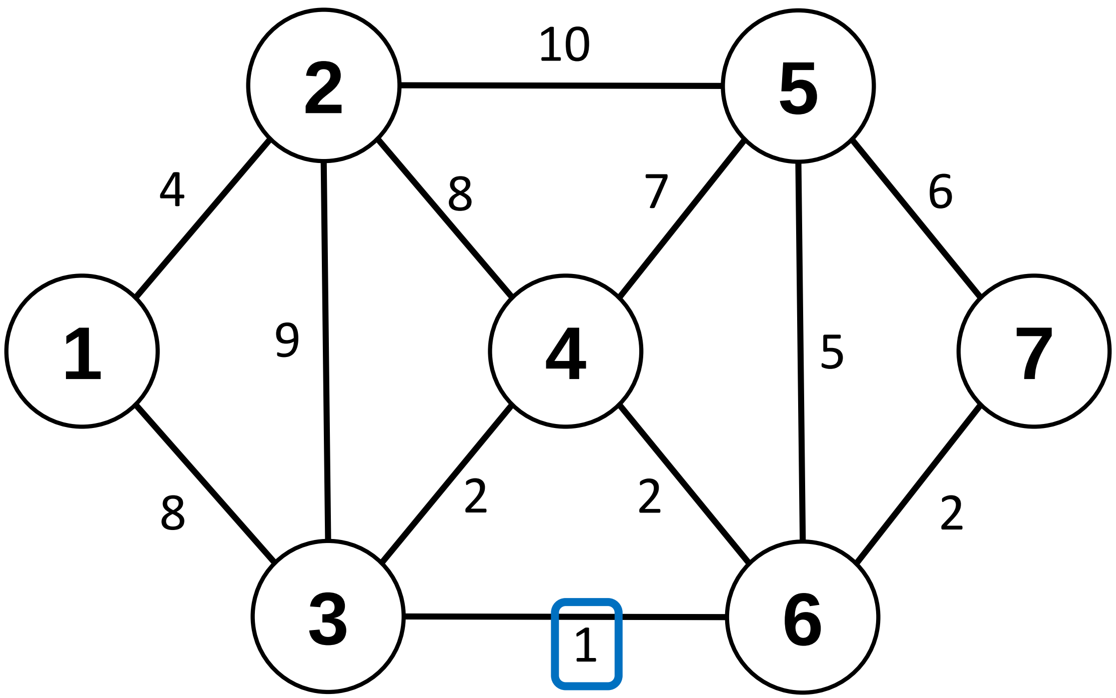
Idea of Kruskal's Algorithm

- Kruskal's Algorithm is based directly on the generic algorithm.
- Unlike Prim's algorithm, which grows one tree, Kruskal's algorithm grows a collection of trees (a forest).
- Initially, this forest consists of the vertices only (no edges).
- In each step the cheapest edge that does not create a cycle is added.
- Continue until the forest 'merges into' a single tree.

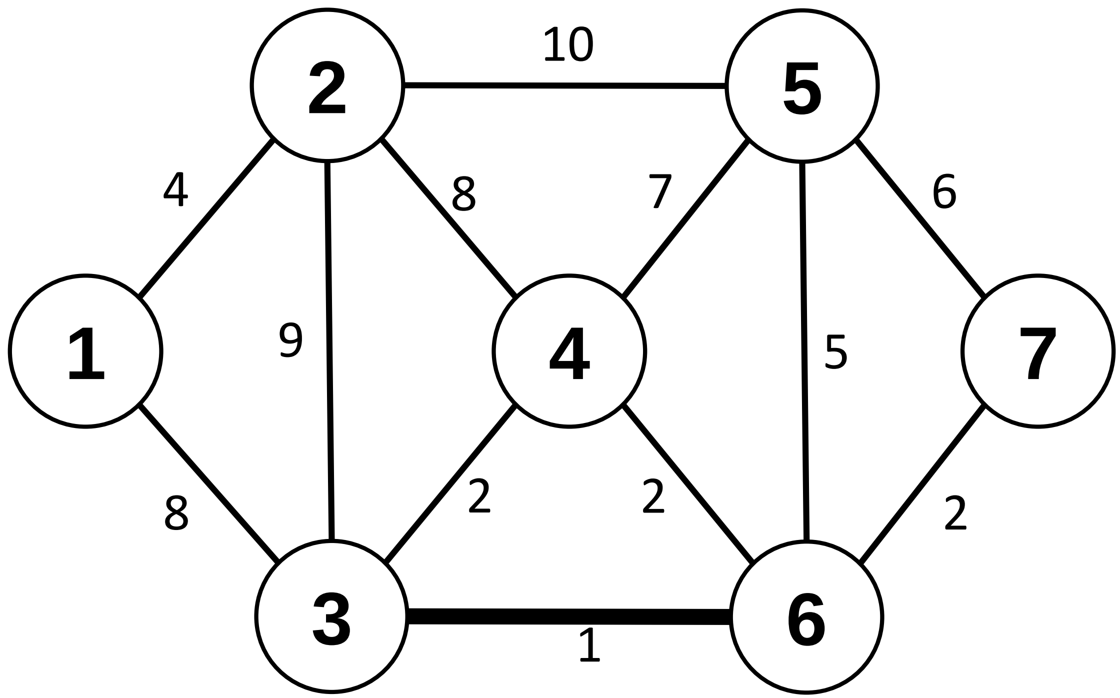
Kruskal's Example



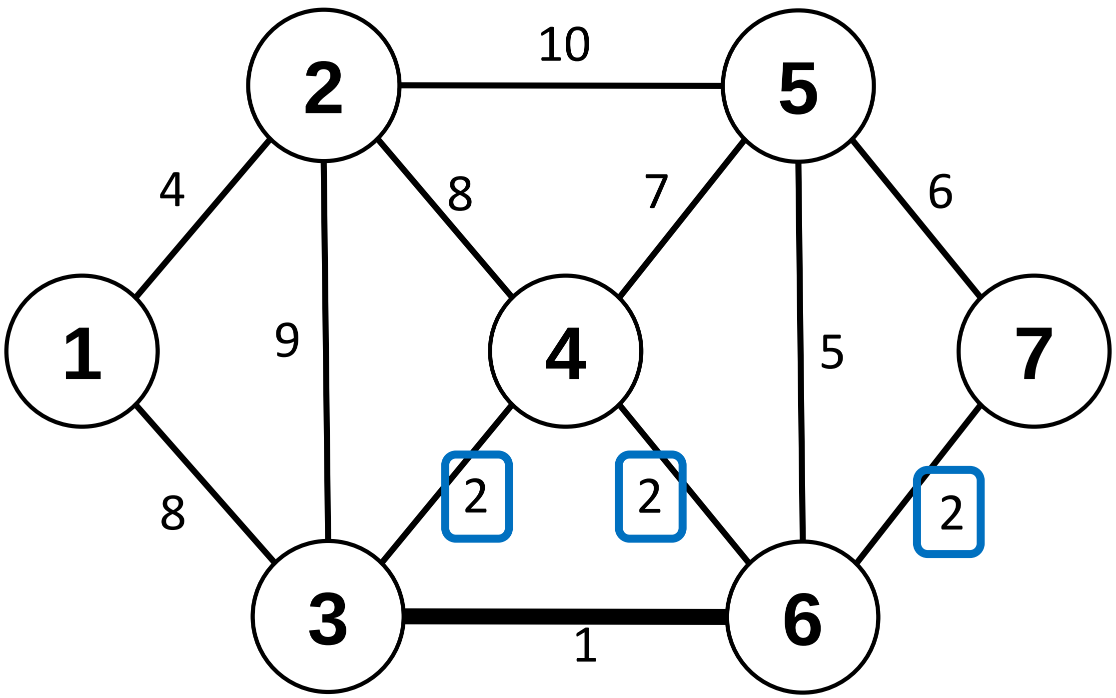
Kruskal's Example



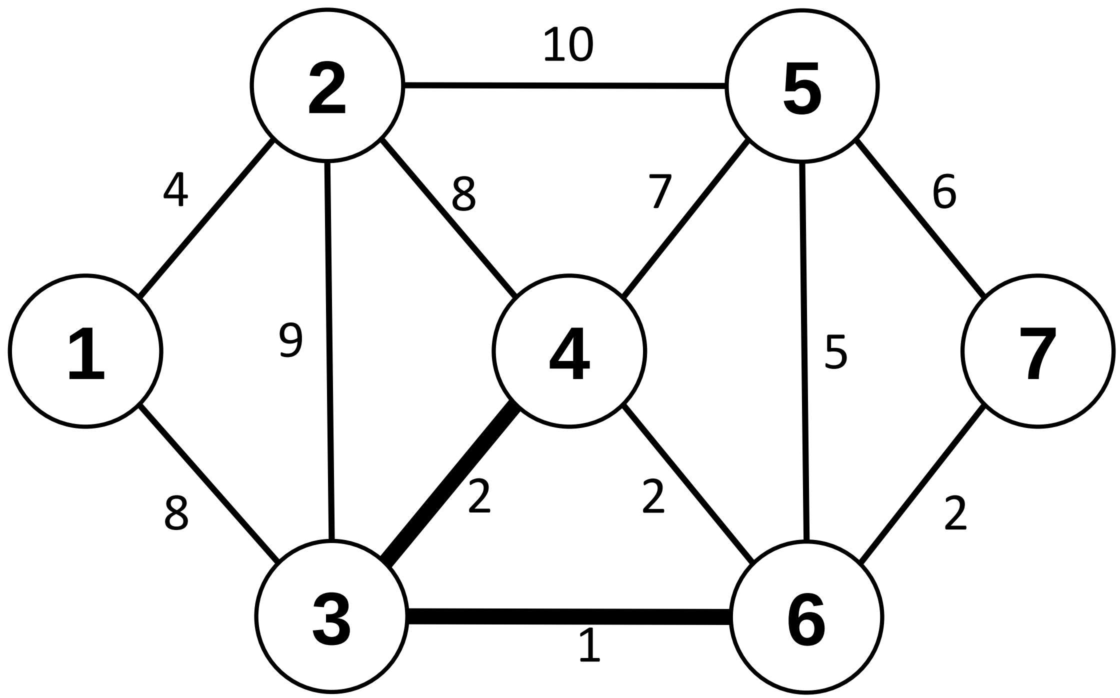
Kruskal's Example



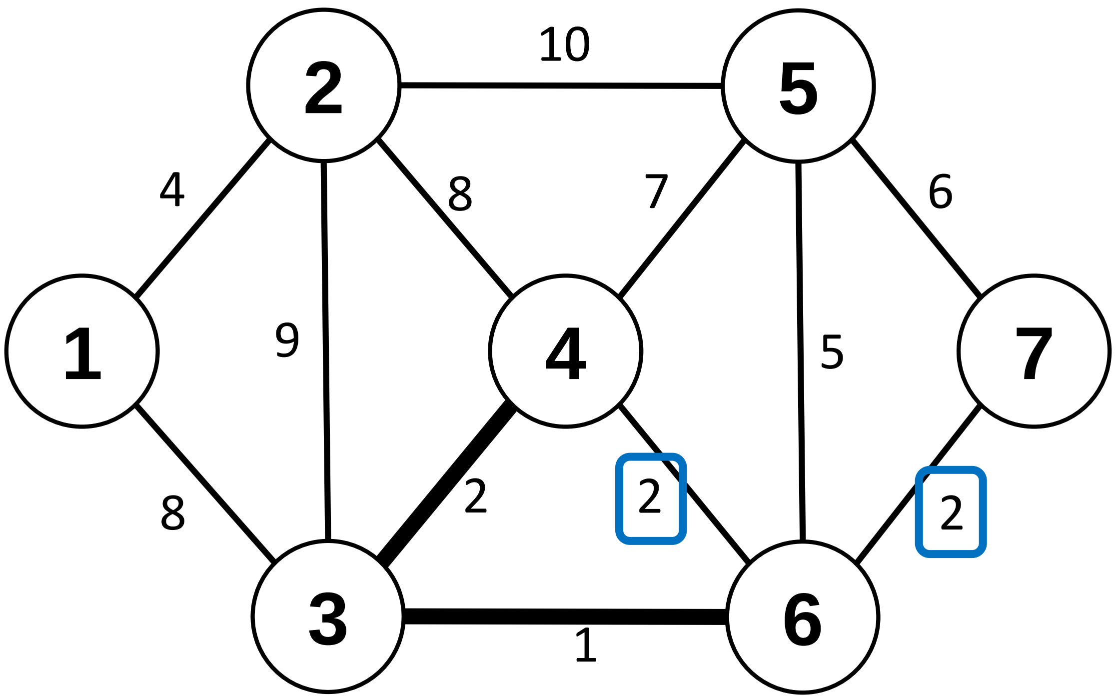
Kruskal's Example



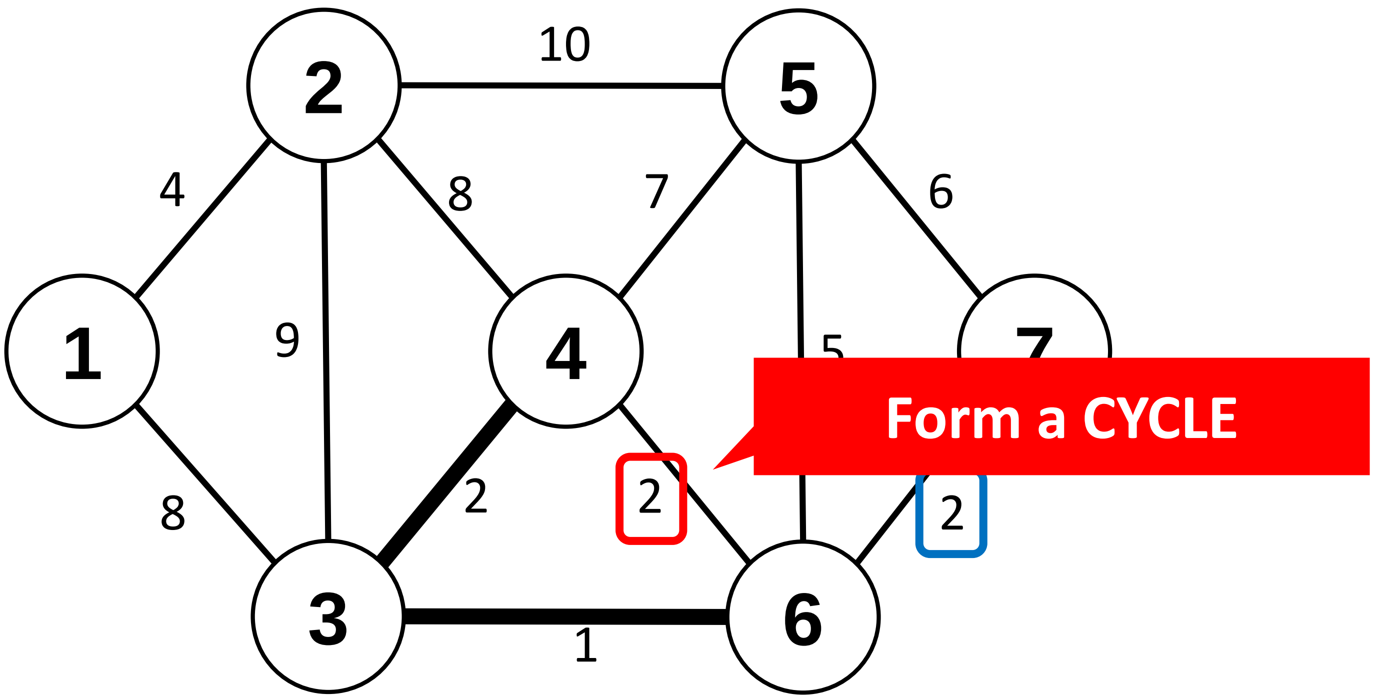
Kruskal's Example



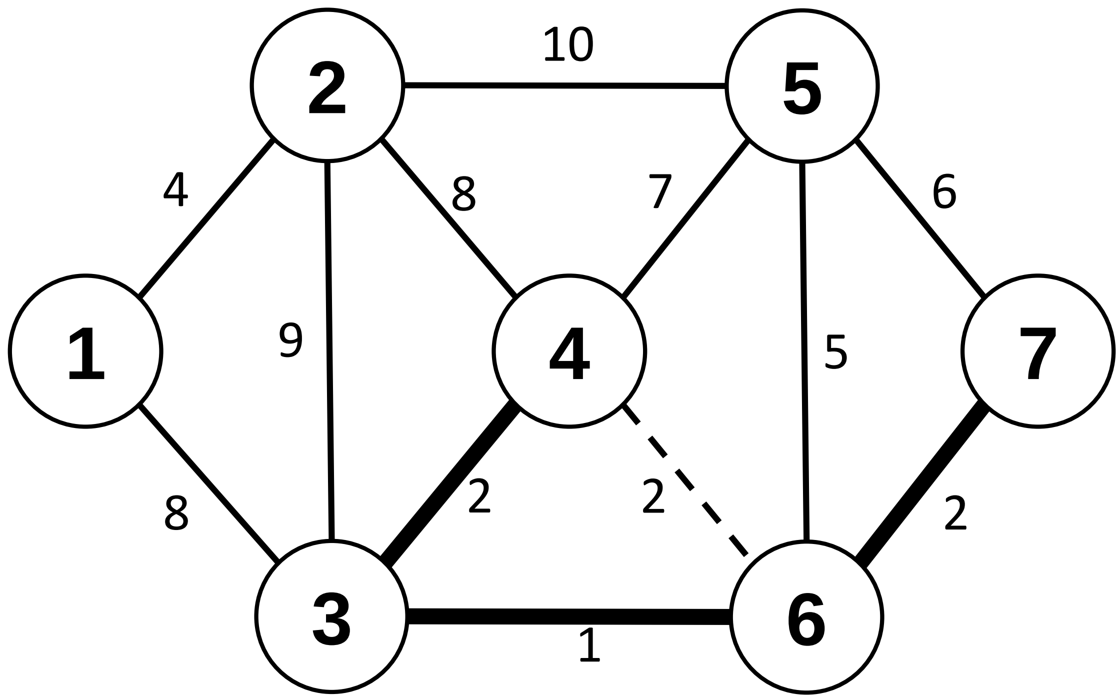
Kruskal's Example



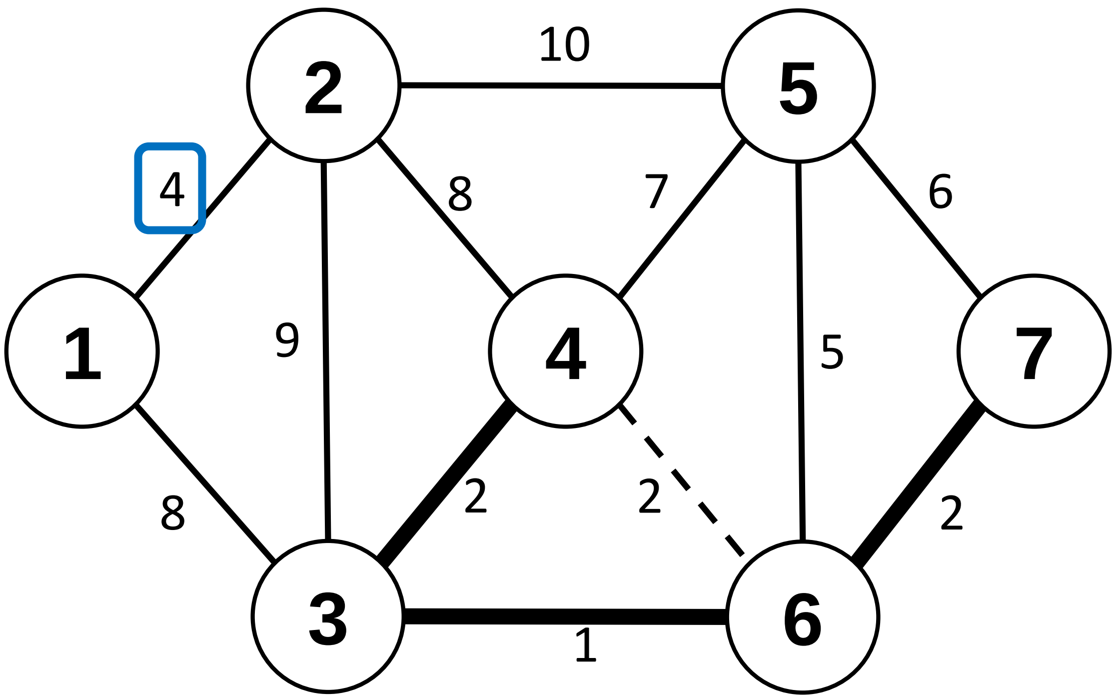
Kruskal's Example



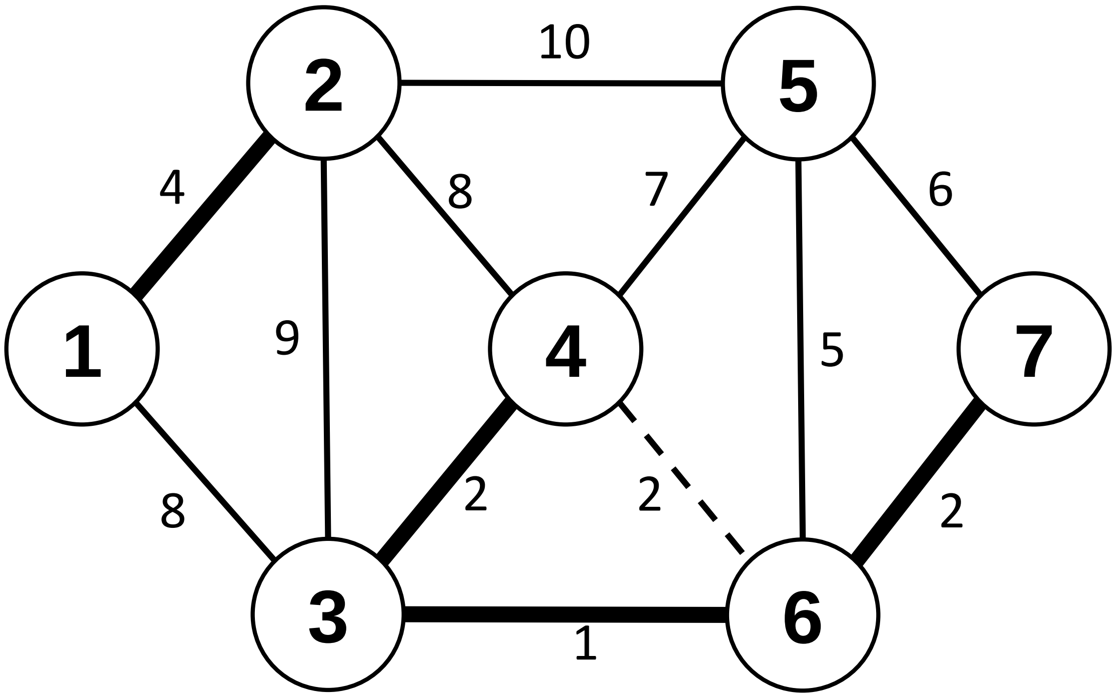
Kruskal's Example



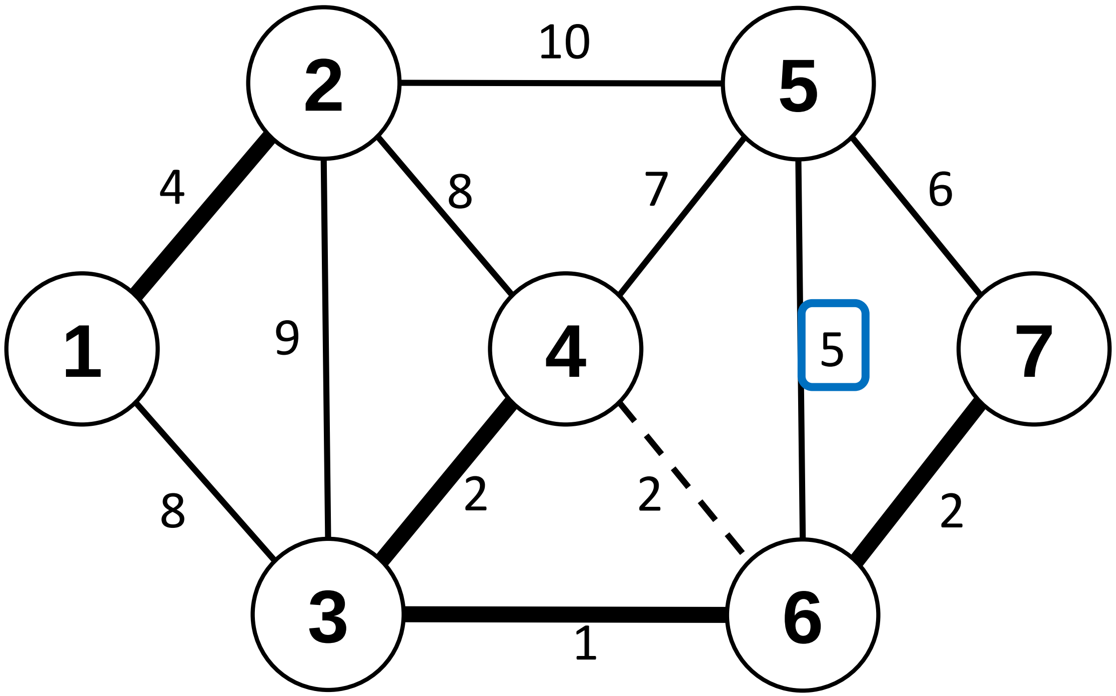
Kruskal's Example



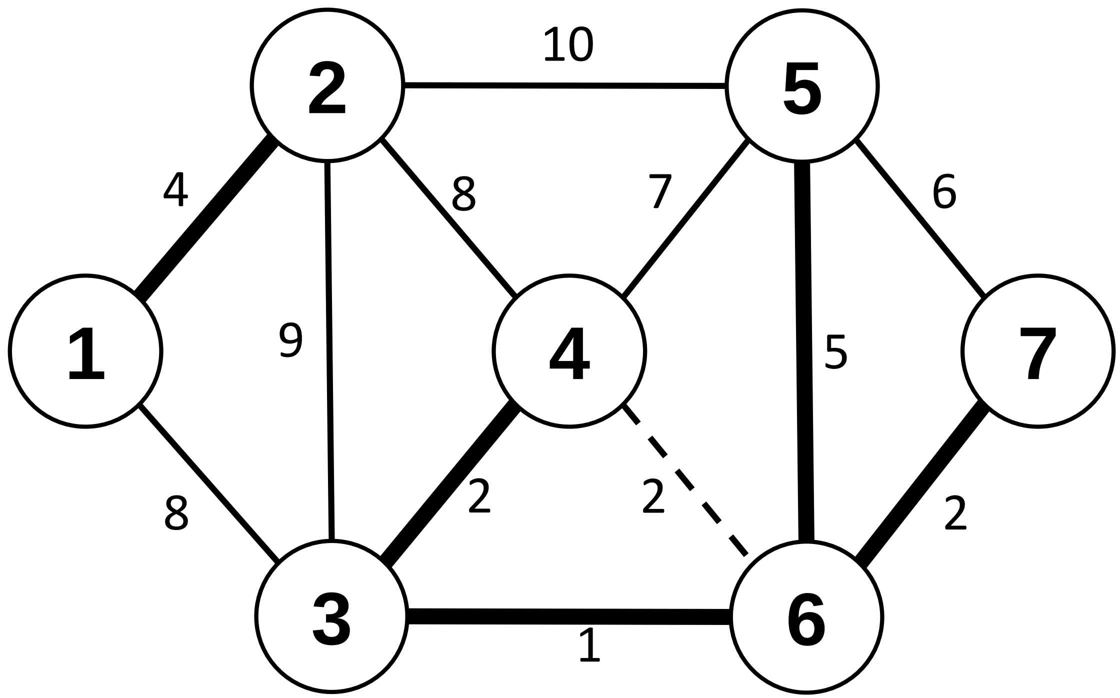
Kruskal's Example



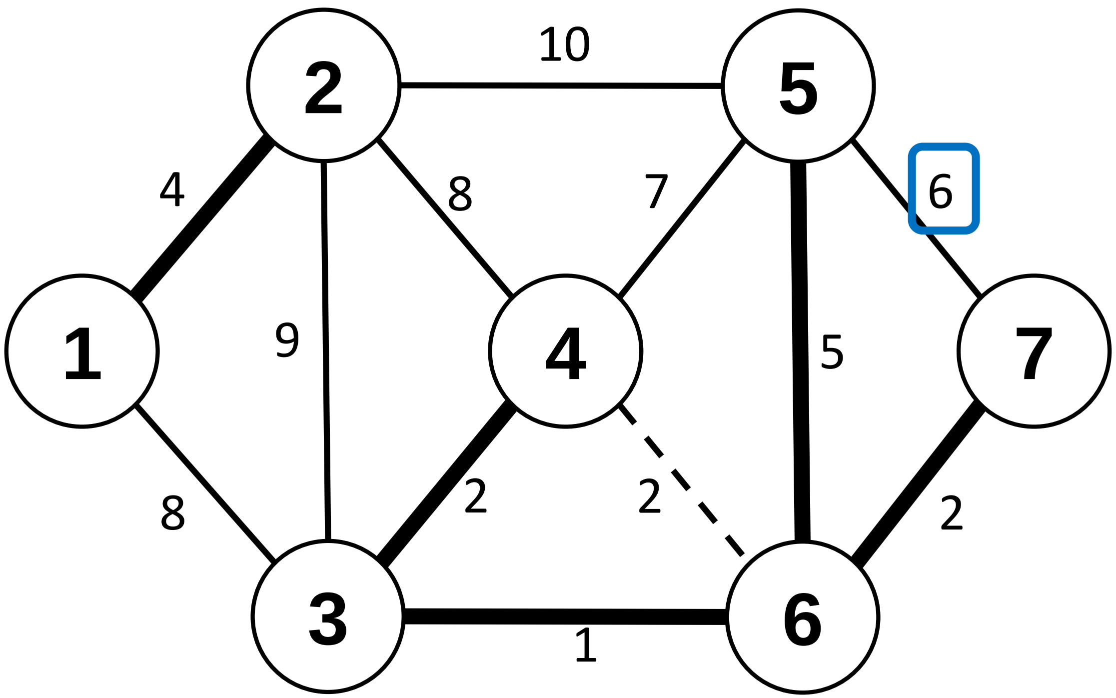
Kruskal's Example



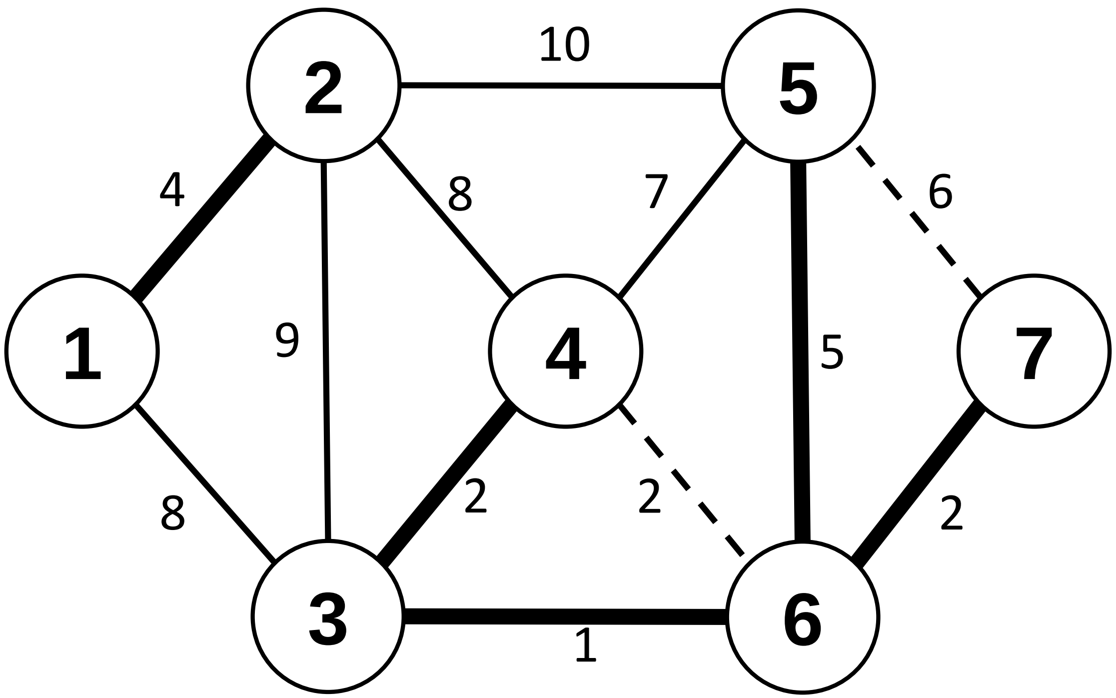
Kruskal's Example



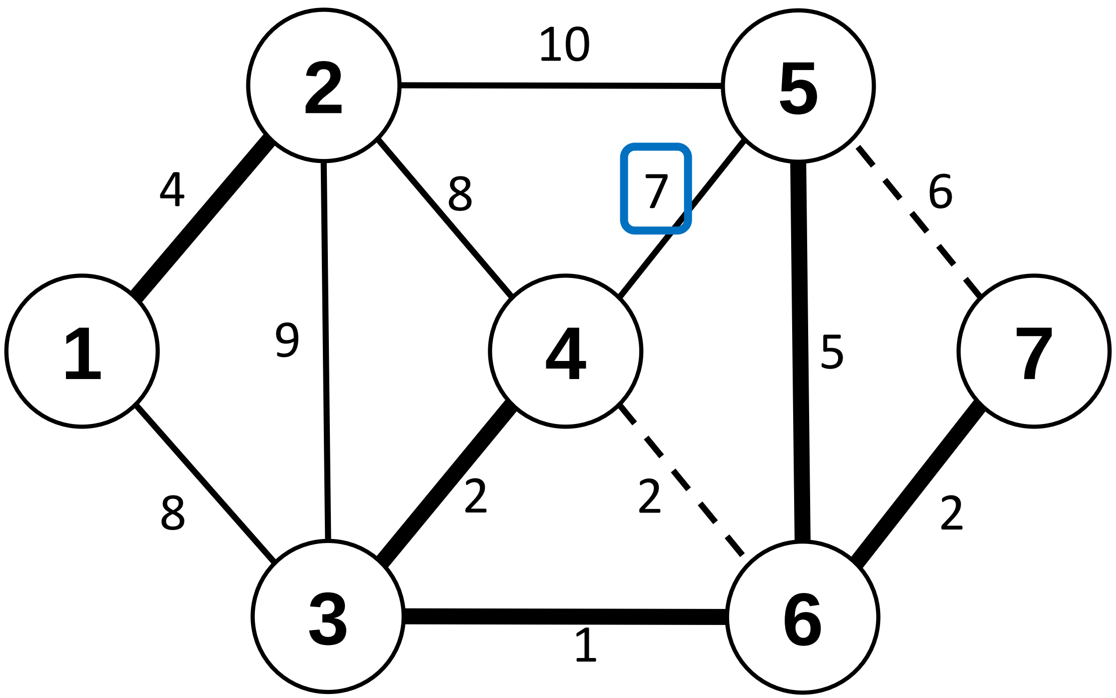
Kruskal's Example



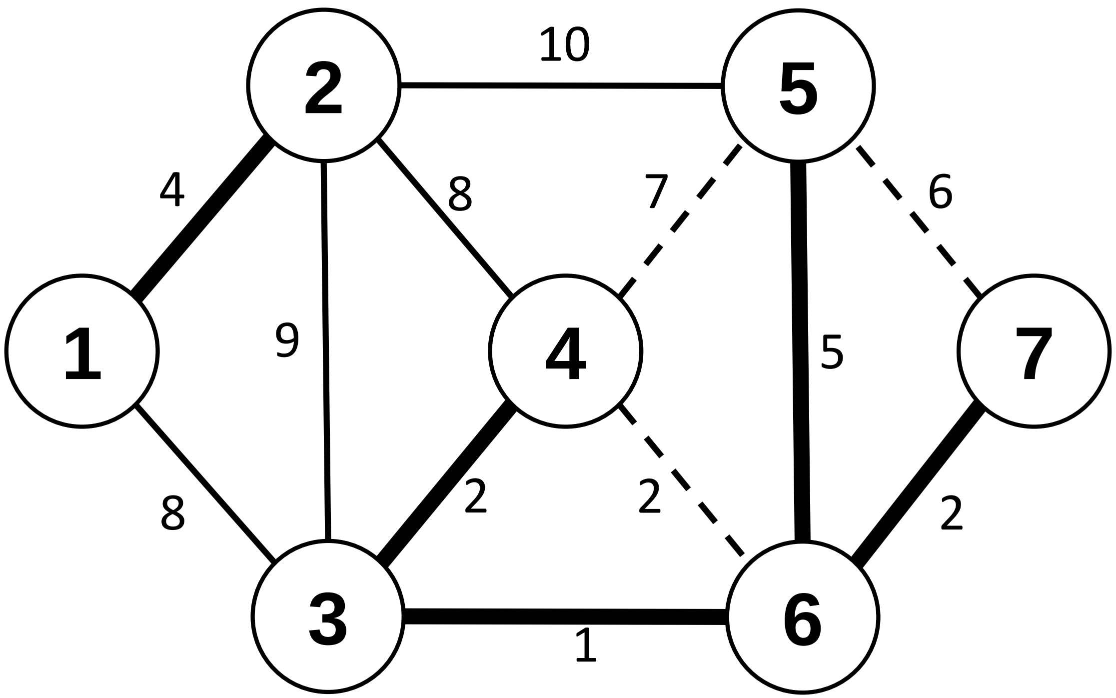
Kruskal's Example



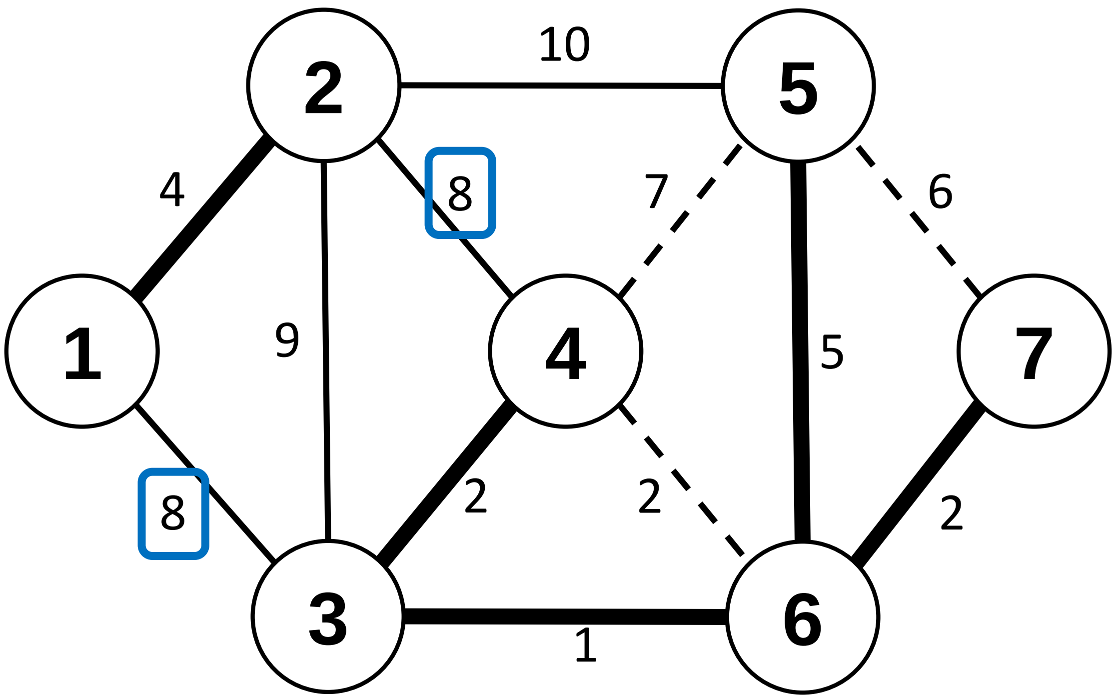
Kruskal's Example



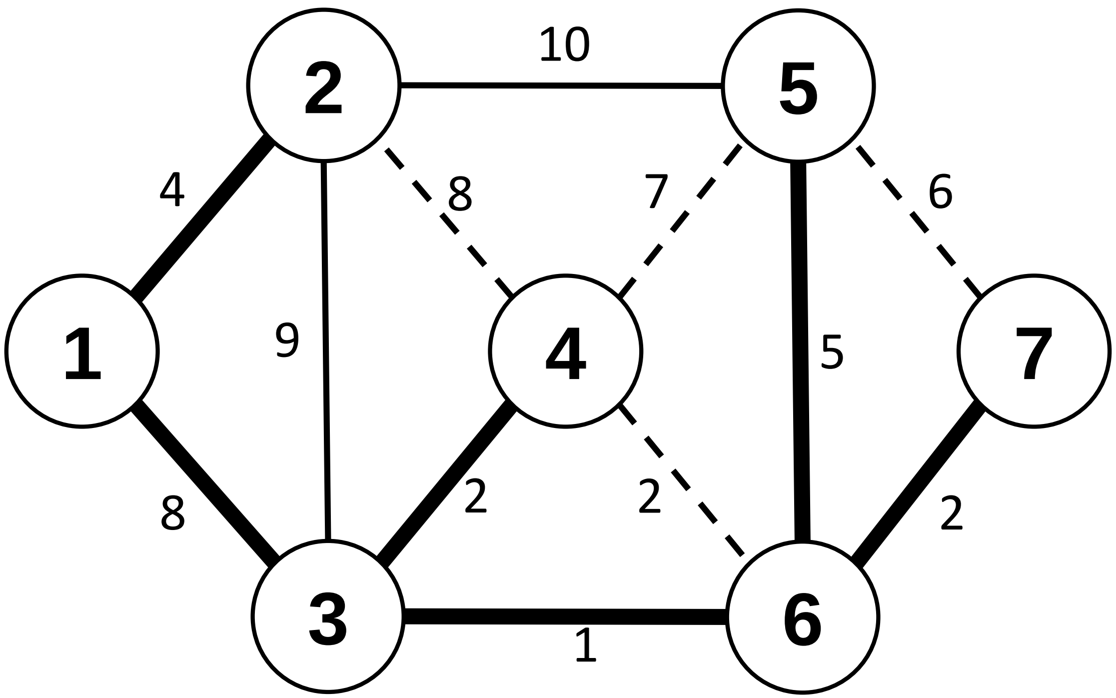
Kruskal's Example



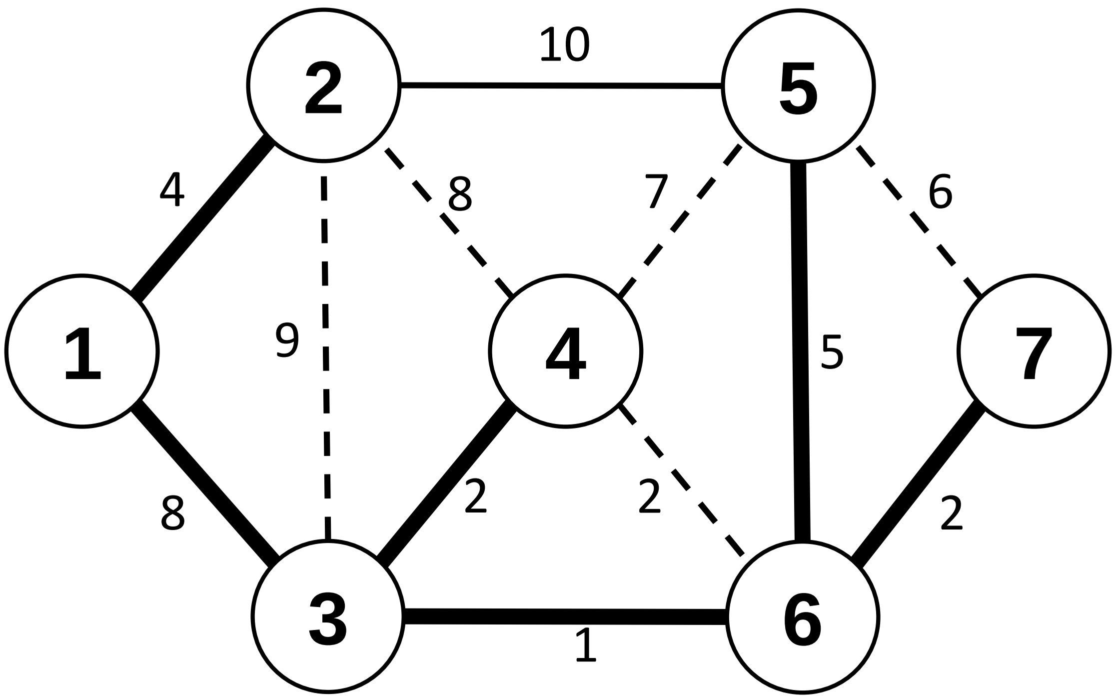
Kruskal's Example



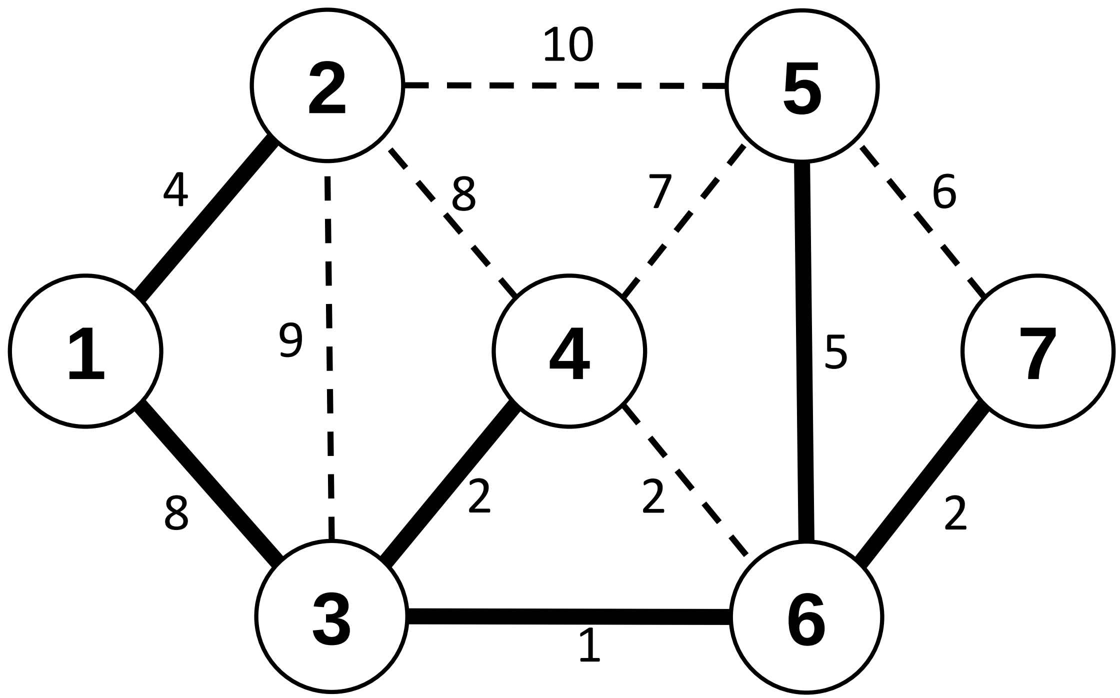
Kruskal's Example



Kruskal's Example



Kruskal's Example



Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

The Framework of Kruskal's Algorithm

- Set A = Ø and F = E, the set of all edges.

The Framework of Kruskal's Algorithm

- Set A = Ø and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.

The Framework of Kruskal’s Algorithm

- Set A = ∅ and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
  - If "yes", remove e from F.

The Framework of Kruskal's Algorithm

- Set A = ∅ and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
  - If "yes", remove e from F.
  - If "no", move e from F to A.

The Framework of Kruskal’s Algorithm

- Set A = Ø and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
  - If "yes", remove e from F.
  - If "no", move e from F to A.
- If F = Ø, stop and output the minimal spanning tree (V,A).

The Framework of Kruskal’s Algorithm

- Set A = Ø and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
  - If "yes", remove e from F.
  - If "no", move e from F to A.
- If F = Ø, stop and output the minimal spanning tree (V,A). Otherwise go to Step 2.

The Framework of Kruskal's Algorithm

- Set A = Ø and F = E, the set of all edges.
- Choose an edge e in F of minimum weight, and check whether adding e to A creates a cycle.
  - If "yes", remove e from F.
  - If "no", move e from F to A.
- If F = Ø, stop and output the minimal spanning tree (V,A). Otherwise go to Step 2.

Questions

- How does algorithm choose edge $e \in F$ with minimum weight?
- How does algorithm check whether adding e to A creates a cycle?

How to Choose the Edge of Least Weight?

Question

How does algorithm choose edge $e \in F$ with minimum weight?

How to Choose the Edge of Least Weight?

Question

How does algorithm choose edge $e \in F$ with minimum weight?

Answer:

- Start by sorting edges in E in order of increasing weight.

How to Choose the Edge of Least Weight?

Question

How does algorithm choose edge $e \in F$ with minimum weight?

Answer:

- Start by sorting edges in E in order of increasing weight.
- Walk through the edges in this order.

How to Choose the Edge of Least Weight?

Question

How does algorithm choose edge $e \in F$ with minimum weight?

Answer:

- Start by sorting edges in E in order of increasing weight.
- Walk through the edges in this order.
- (Once edge e causes a cycle it will always cause a cycle, so it can be thrown away.)

How to Check for Cycles?

Observations:

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

- Vertices in a tree are considered to be in same set.

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

- Vertices in a tree are considered to be in same set.
- Test if Find-Set(u) = Find-Set(v)?

How to Check for Cycles?

Observations:

- At each step of the framework algorithm, (V,A) is acyclic so it is a forest.
- If u and v are in the same tree, then adding edge {u,v} to A creates a cycle.
- If u and v are not in the same tree, then adding edge {u,v} to A does not create a cycle.

Question

How to test whether u and v are in the same tree?

High-Level Answer: Use a disjoint-set data structure

- Vertices in a tree are considered to be in same set.
- Test if Find-Set(u) = Find-Set(v)?

Low-Level Answer:

- The Union-Find data structure implements this

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

The Union-Find Data Structure

Union-Find supports three operations on collections of disjoint sets over some universe U. Let n = |U|. For any u,v∈U:

The Union-Find Data Structure

Union-Find supports three operations on collections of disjoint sets over some universe U. Let n = |U|. For any u,v∈U:

- Create-Set(u): Create a set containing the single element u.
  - O(1) time

The Union-Find Data Structure

Union-Find supports three operations on collections of disjoint sets over some universe U. Let n = |U|. For any u,v∈U:

- Create-Set(u): Create a set containing the single element u.
  - O(1) time
- Find-Set(u): Find the set containing the element u. (Say each set has a unique ID)
  - O(log n) time

The Union-Find Data Structure

Union-Find supports three operations on collections of disjoint sets over some universe U. Let $n = |U|$. For any $u, v \in U$:

- Create-Set(u): Create a set containing the single element u.
  - O(1) time
- Find-Set(u): Find the set containing the element u. (Say each set has a unique ID)
  - O(log n) time
- Union(u, v): Merge the sets containing u and v respectively into a common set.
  - O(log n) time

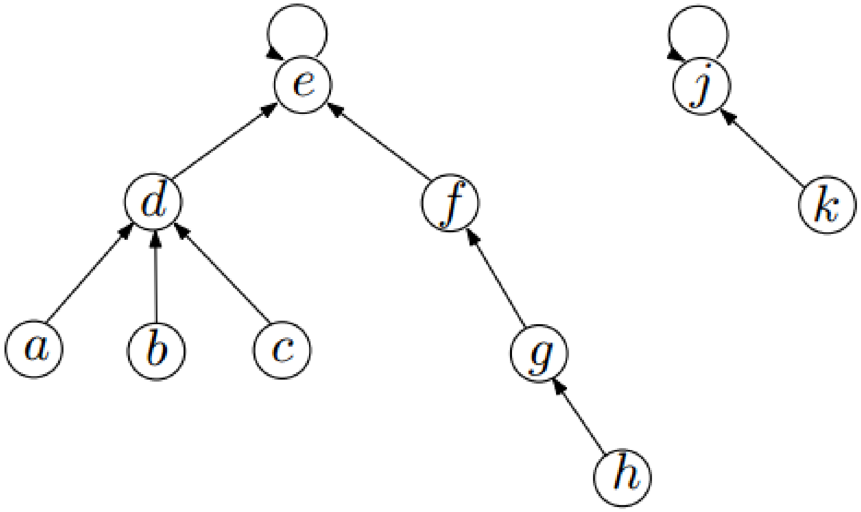
The Union-Find Data Structure

Union-Find supports three operations on collections of disjoint sets over some universe U. Let n = |U|. For any u,v∈U:

- Create-Set(u): Create a set containing the single element u.
  - O(1) time
- Find-Set(u): Find the set containing the element u. (Say each set has a unique ID)
  - O(log n) time
- Union(u, v): Merge the sets containing u and v respectively into a common set.
  - O(log n) time

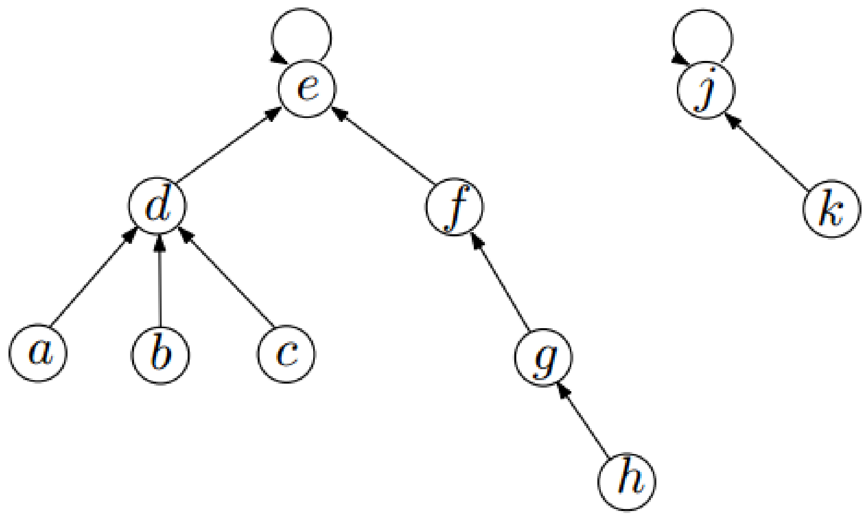
For now we treat Union-Find as a black box. We will present its implementation.

Up-Tree Implementation



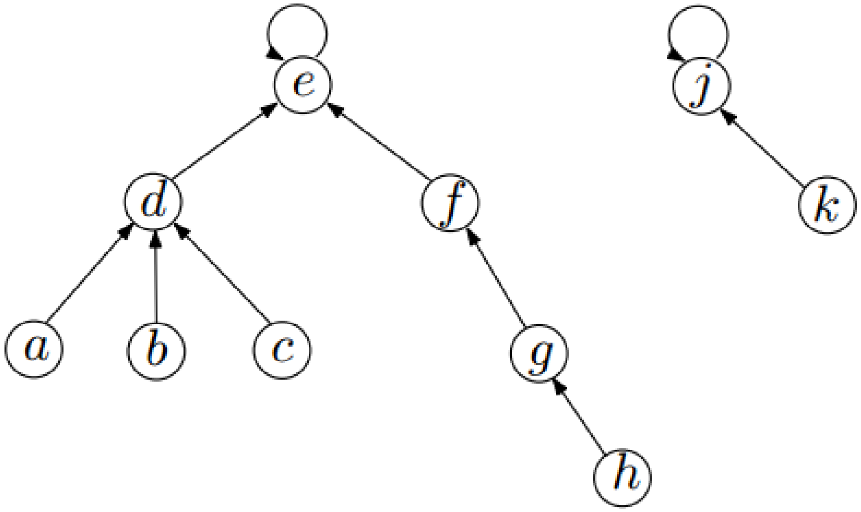
- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)

Up-Tree Implementation



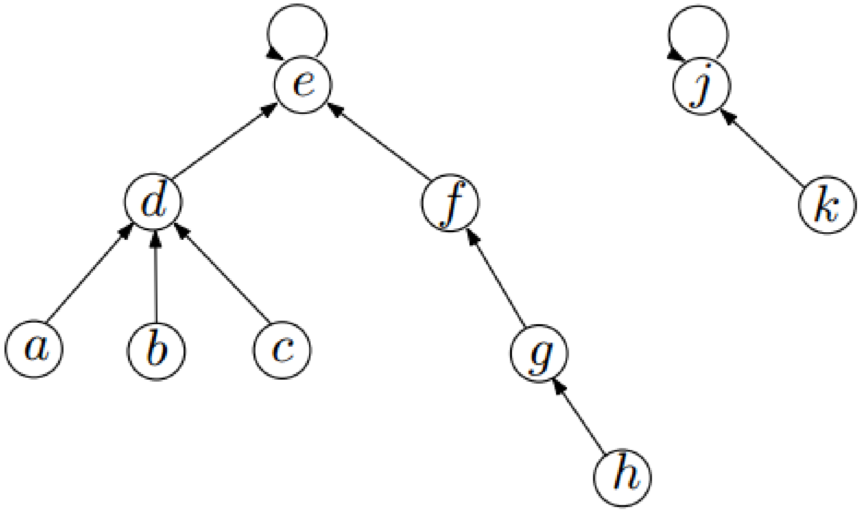
- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)
- The root of the tree is the representative item of all items in that tree

Up-Tree Implementation



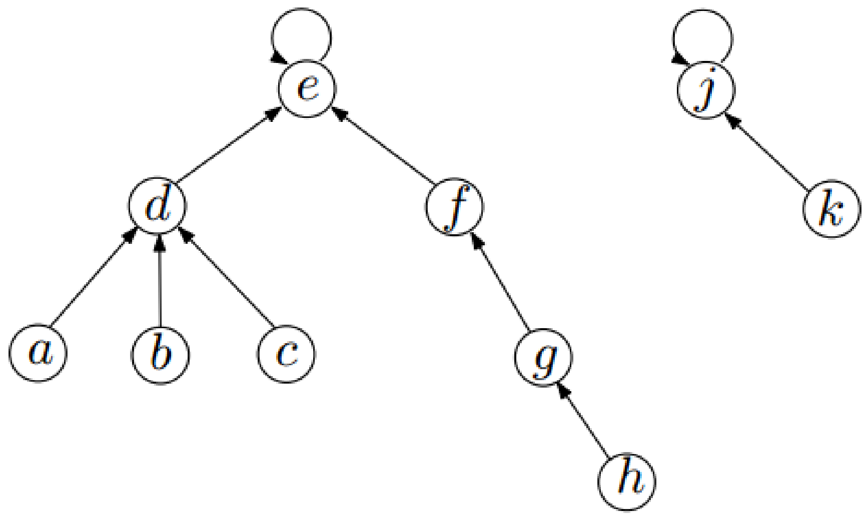
- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)
- The root of the tree is the representative item of all items in that tree
  - i.e., the root of the tree represents the whole items.

Up-Tree Implementation



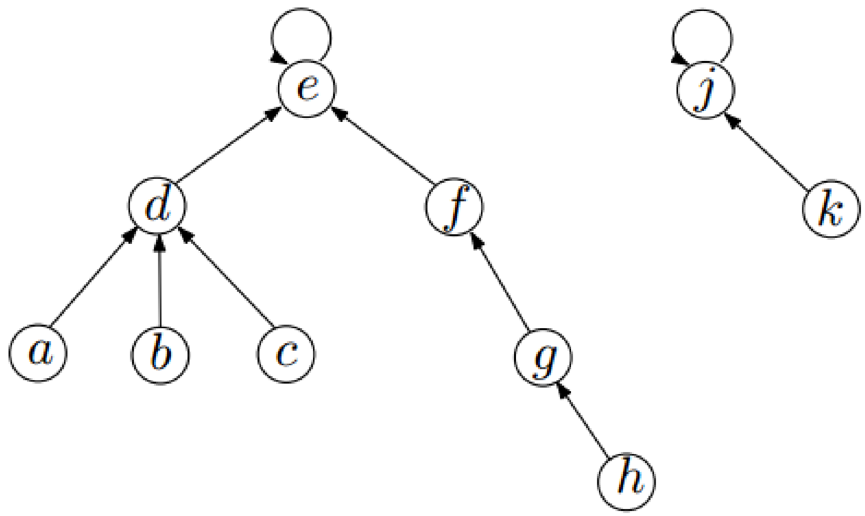
- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)
- The root of the tree is the representative item of all items in that tree
  - i.e., the root of the tree represents the whole items.
  - use the root's ID as the unique ID of the set.

Up-Tree Implementation



- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)
- The root of the tree is the representative item of all items in that tree
  - i.e., the root of the tree represents the whole items.
  - use the root's ID as the unique ID of the set.
- In this up-tree implementation, every node (except the root) has a pointer pointing to its parent.

Up-Tree Implementation



- Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal's algorithm.)
- The root of the tree is the representative item of all items in that tree
  - i.e., the root of the tree represents the whole items.
  - use the root's ID as the unique ID of the set.
- In this up-tree implementation, every node (except the root) has a pointer pointing to its parent.
  - The root element has a pointer pointing to itself.

Create-Set(x) and Find-Set(x)

Create-Set(x): easy

$$x.\text{parent} \leftarrow x;$$

Create-Set(x) and Find-Set(x)

Create-Set(x): easy

$x.\text{parent} \leftarrow x;$

Find-Set(x): also easy

- simply trace the parent point until we hit the root, then return the root element.

Create-Set(x) and Find-Set(x)

Create-Set(x): easy

x.parent $\leftarrow x$;

Find-Set(x): also easy

- simply trace the parent point until we hit the root, then return the root element.

while $x \neq x.parent$ **do**
$\quad x \leftarrow x$.parent;
end

Union(x, y)

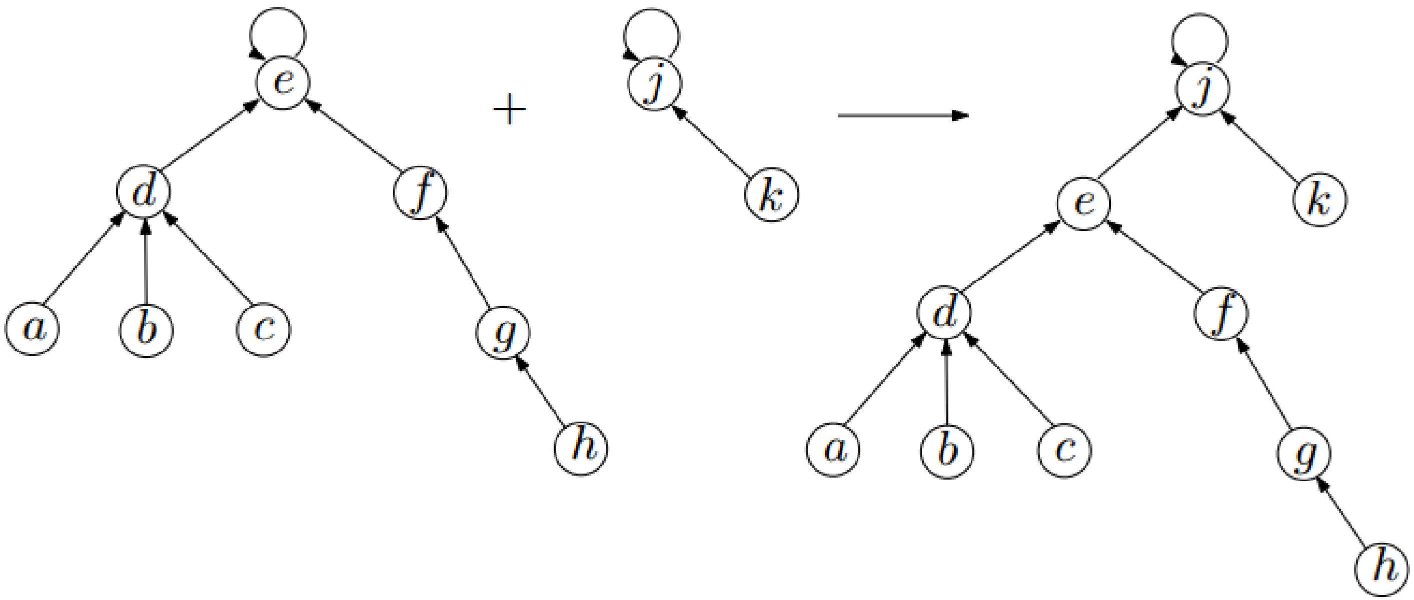
Naive solution:

- put the parent pointer of the representation of x pointing to the representation of y.

Union(x, y)

Naive solution:

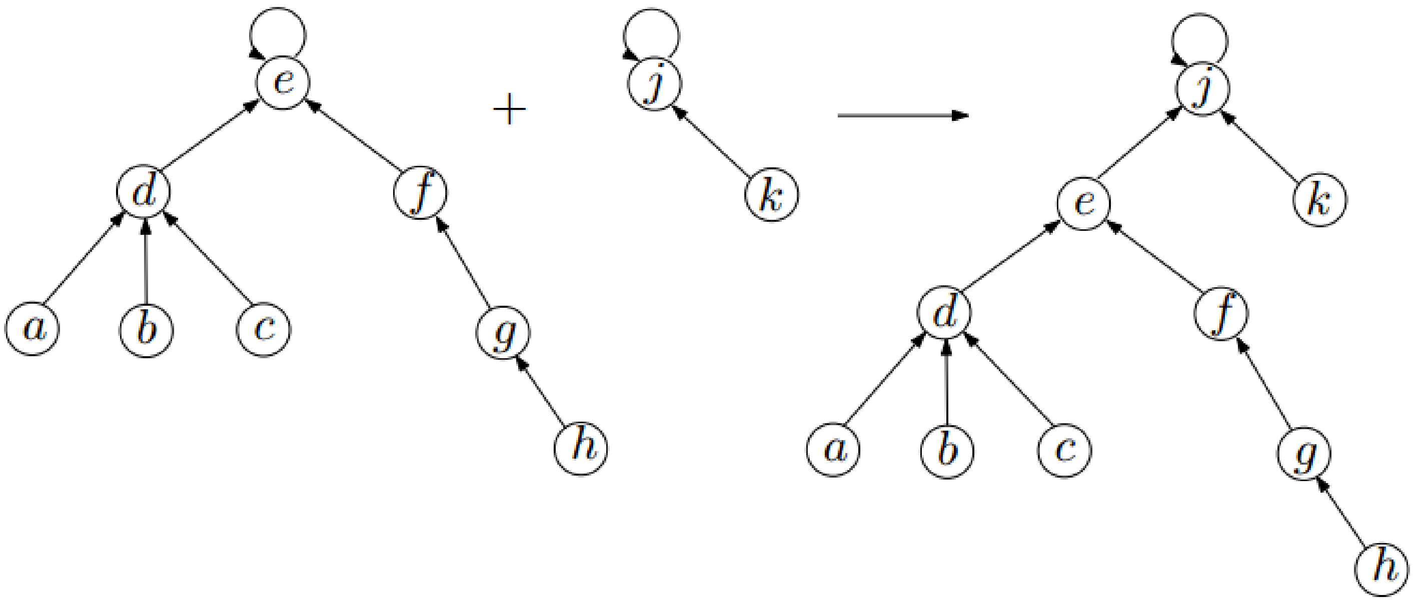
- put the parent pointer of the representation of x pointing to the representation of y.



Union(x, y)

Naive solution:

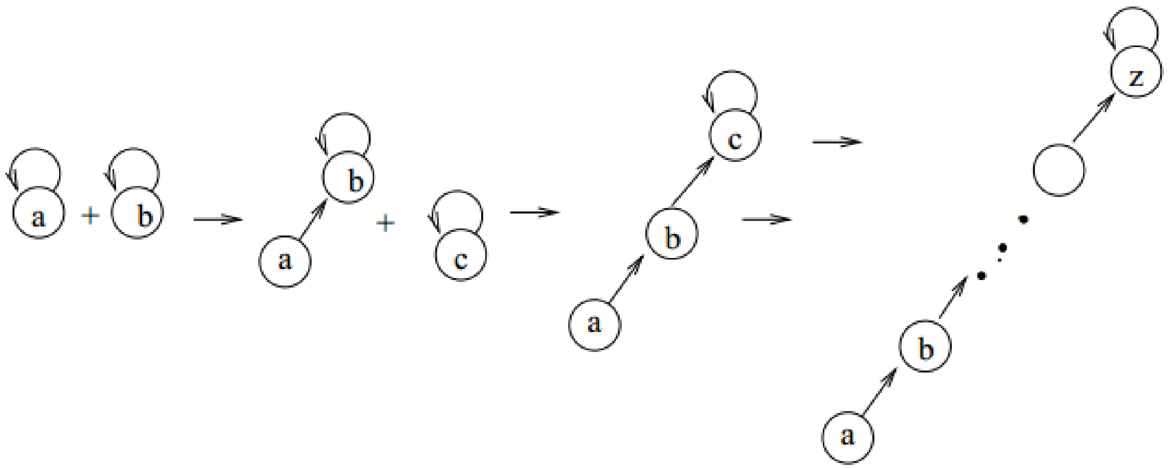
- put the parent pointer of the representation of x pointing to the representation of y.



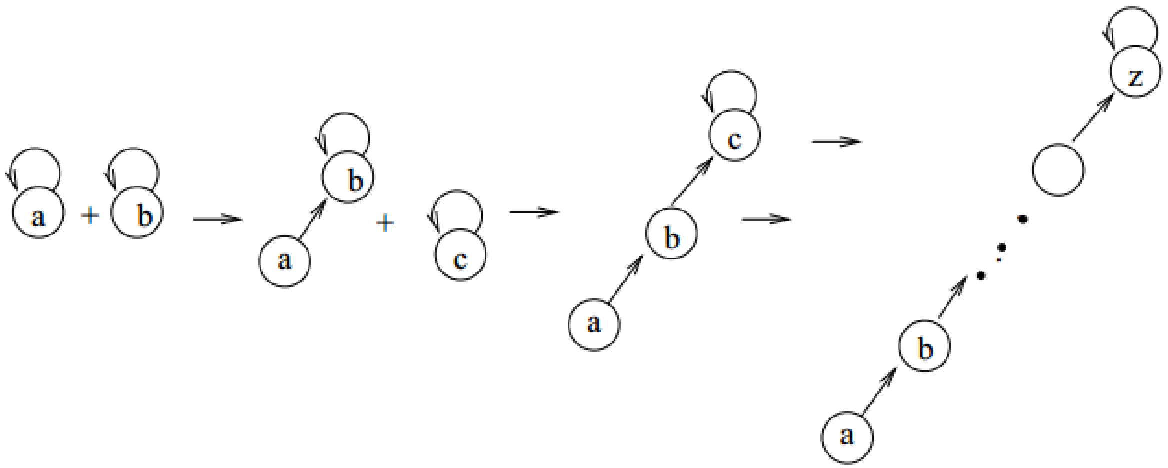
Question

Is it a good idea?

Union(x, y)

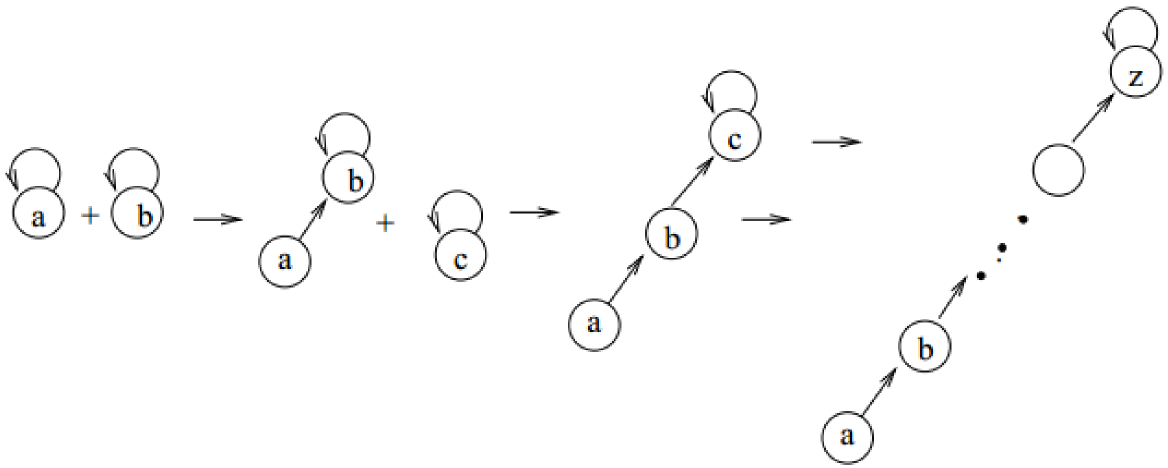


Union(x, y)



May become a linked-list at the end! Hence it is not efficient.

Union(x, y)

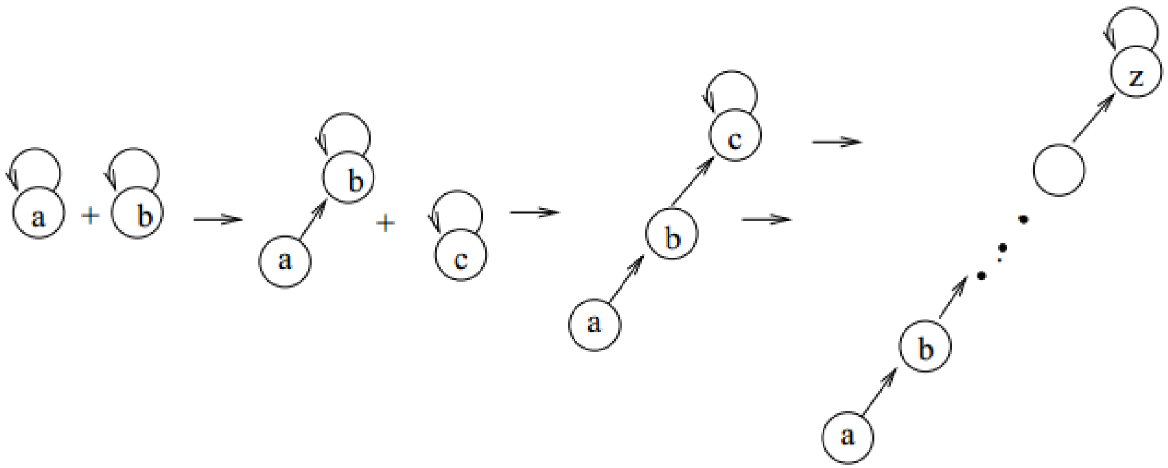


May become a linked-list at the end! Hence it is not efficient.

Question

Can we do better?

Union(x, y)



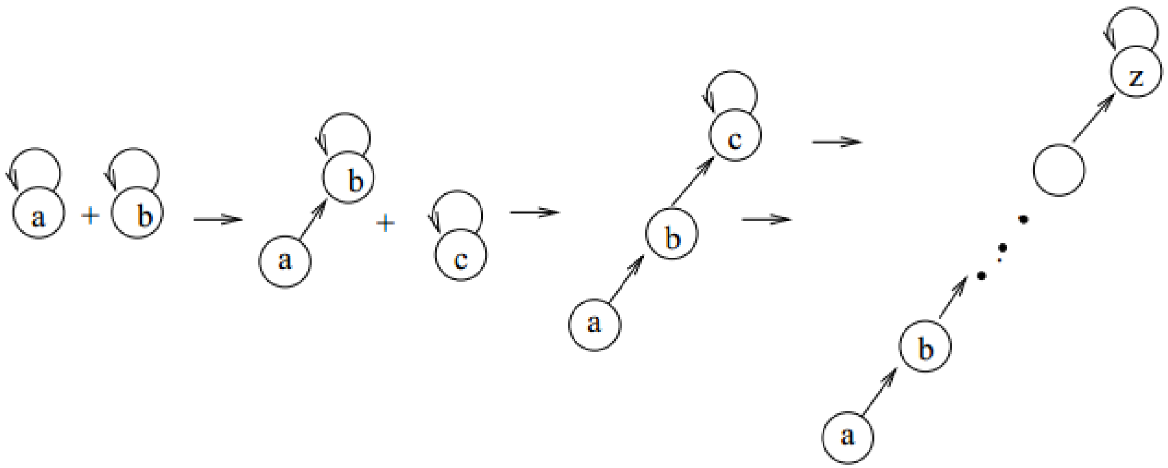
May become a linked-list at the end! Hence it is not efficient.

Question

Can we do better?

Simple trick (Union by height):

Union(x, y)



May become a linked-list at the end! Hence it is not efficient.

Question

Can we do better?

Simple trick (Union by height):

- when we union two trees together, we always make the root of the taller tree the parent of shorter tree.

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second.

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None
$a \leftarrow$Find-Set(x);
$b \leftarrow$Find-Set(y);

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None
$a \leftarrow$Find-Set(x);
$b \leftarrow$Find-Set(y);
if $a.height \leq b.height$ **then**
    if *a.height is equal to b.height* **then**

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None
$a \leftarrow$Find-Set(x);
$b \leftarrow$Find-Set(y);
if $a.height \leq b.height$ **then**
  if *a.height is equal to b.height* **then**
    b.height++;
  end

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None
$a \leftarrow$Find-Set(x);
$b \leftarrow$Find-Set(y);
if *a.height* $\leq$ *b.height* **then**
    if *a.height is equal to b.height* **then**
        b.height++;
    end
    a.parent $\leftarrow b$;
end
else

Up-Tree Implementation : Union by Height

- The root of every tree also holds the height of the tree.
- In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union(x,y)

Input: Two elements $\boldsymbol{x}, \boldsymbol{y} \in U$
Output: None
$a \leftarrow$Find-Set(x);
$b \leftarrow$Find-Set(y);
if $a.height \leq b.height$ **then**
  if *a.height is equal to b.height* **then**
    b.height++;
  end
  a.parent $\leftarrow b$;
end
else
  b.parent $\leftarrow a$;
end

Lemma

Lemma

For the root x of any tree, let $size(x)$ *denote the number of nodes and* $h(x)$ *be the height of the tree. Then* $size(x) \geq 2^{h(x)}$.

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before $\text{Union}(x, y)$.

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

$$size(x') =$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

$$size(x') = size(x) + size(y) \geq$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

$$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

$$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} =$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

$$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$
   - $h(x) = h(y)$, we have
     $$size(x') =$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before $\text{Union}(x, y)$. Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
   $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$
   - $h(x) = h(y)$, we have
   $$size(x') = size(x) + size(y) \geq$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$

   - $h(x) = h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} =$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$

   - $h(x) = h(y)$, we have

     $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} =$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
   $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$
   - $h(x) = h(y)$, we have
   $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}.$$

Lemma

Lemma

For the root x of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.
2. Suppose the assumption is true for any x and y before Union(x, y). Let the size and height of the resulting tree be $size(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
   $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.$$
   - $h(x) = h(y)$, we have
   $$size(x') = size(x) + size(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}.$$
   - $h(x) > h(y)$, it is similar to the first case

$\square$

Lemma

Lemma

For n items, the running time of

- *Create-Set is* $O(1)$,

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Proof.

- Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x).

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Proof.

- Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x).
- Since the running time of Find-Set(x) depends on the height of the tree. From previous lemma, for any tree, we have

$$n \geq 2^h \quad \Rightarrow \quad h$$

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Proof.

- Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x).
- Since the running time of Find-Set(x) depends on the height of the tree. From previous lemma, for any tree, we have

$$\begin{aligned} n \geq 2^h \quad &\Rightarrow \quad h \leq \log n \\ &\Rightarrow \quad h = O(\log n) \end{aligned}$$

Lemma

Lemma

For n items, the running time of

- *Create-Set is $O(1)$,*
- *Find-Set is $O(\log n)$, and*
- *Union is $O(\log n)$*

respectively.

Proof.

- Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x).
- Since the running time of Find-Set(x) depends on the height of the tree. From previous lemma, for any tree, we have

$$\begin{aligned} n \geq 2^h \quad &\Rightarrow \quad h \leq \log n \\ &\Rightarrow \quad h = O(\log n) \end{aligned}$$

Hence we have Find-Set(x) $= O(\log n)$. $\square$

Outline

- Review to Part IV
- Minimum Spanning Trees
  - Spanning trees
  - Minimum spanning trees
- Prim’s algorithm
  - The idea
  - The algorithm
  - Analysis for Prim’s algorithm
- Kruskal’s algorithm
  - The idea
  - The algorithm
  - The Disjoint Set Union-Find data structure
  - Analysis for Kruskal’s algorithm

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and

Why Kruskal's Algorithm is Correct?

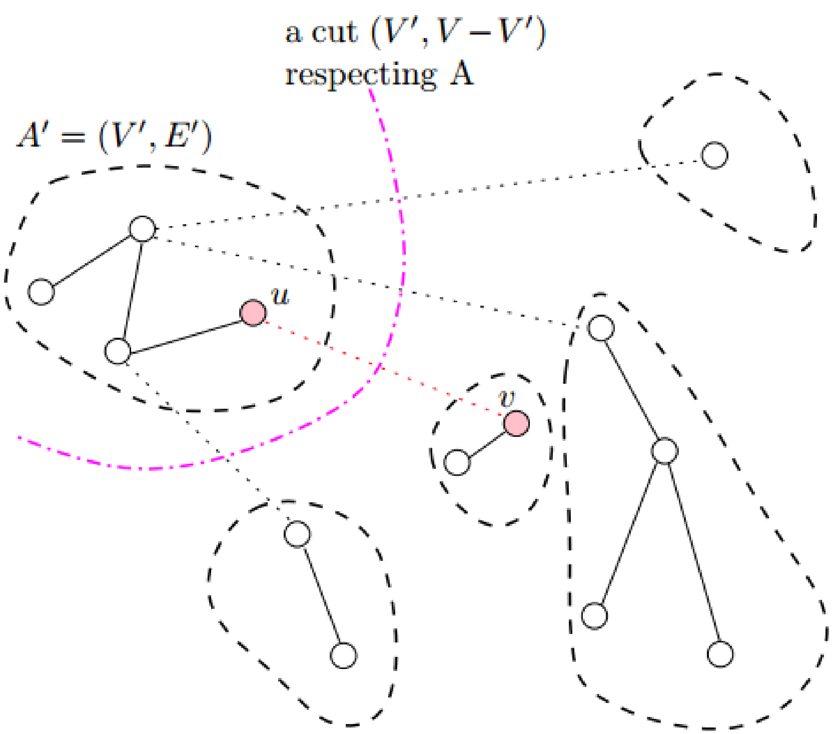
Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut

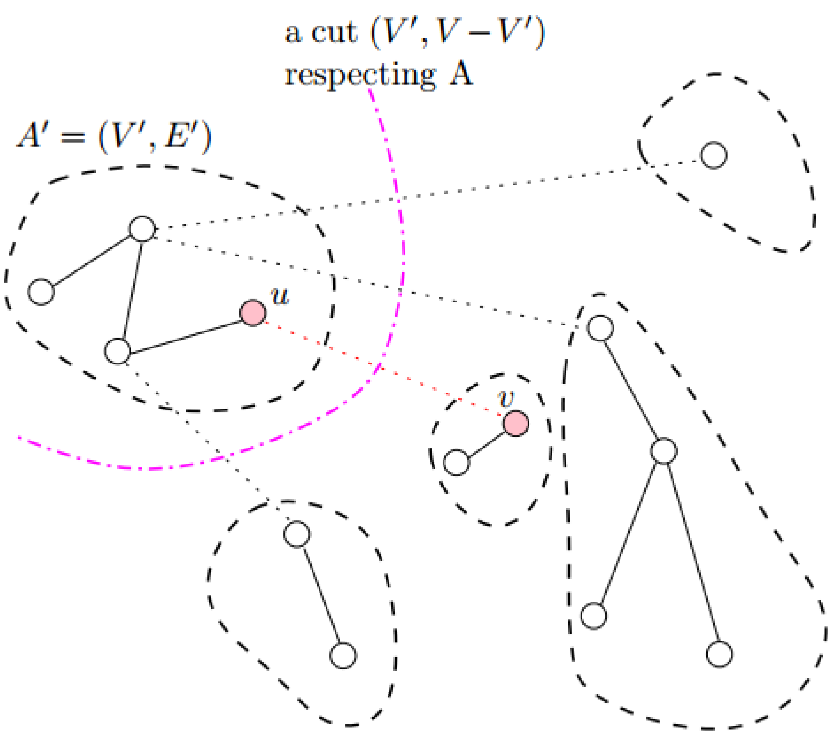


- Let $A' = (V', E')$ denote the tree of the forest A that contains u.

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut

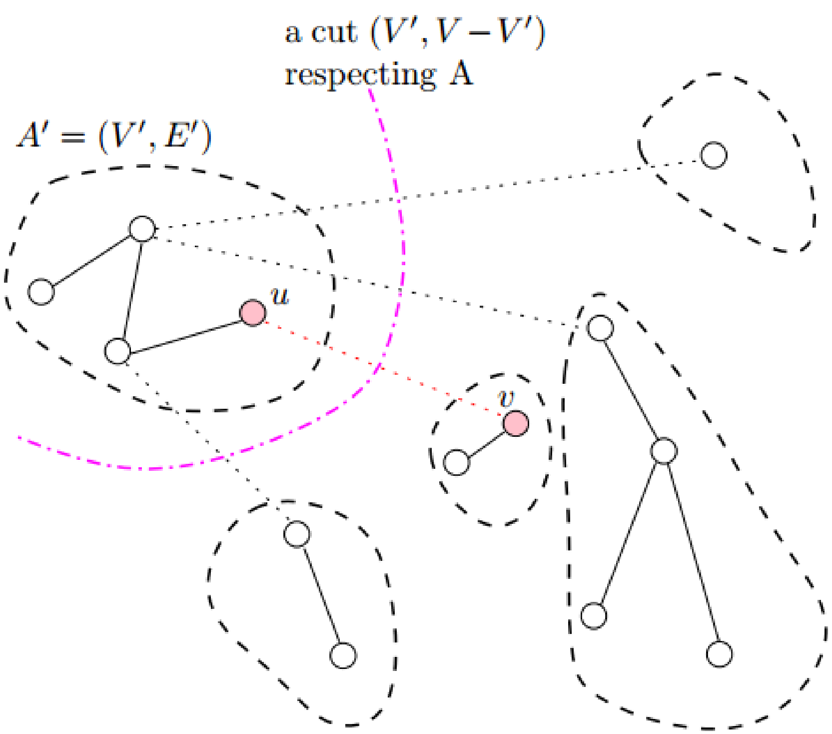


- Let $A' = (V', E')$ denote the tree of the forest A that contains u. Consider the cut $(V', V - V')$.

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut

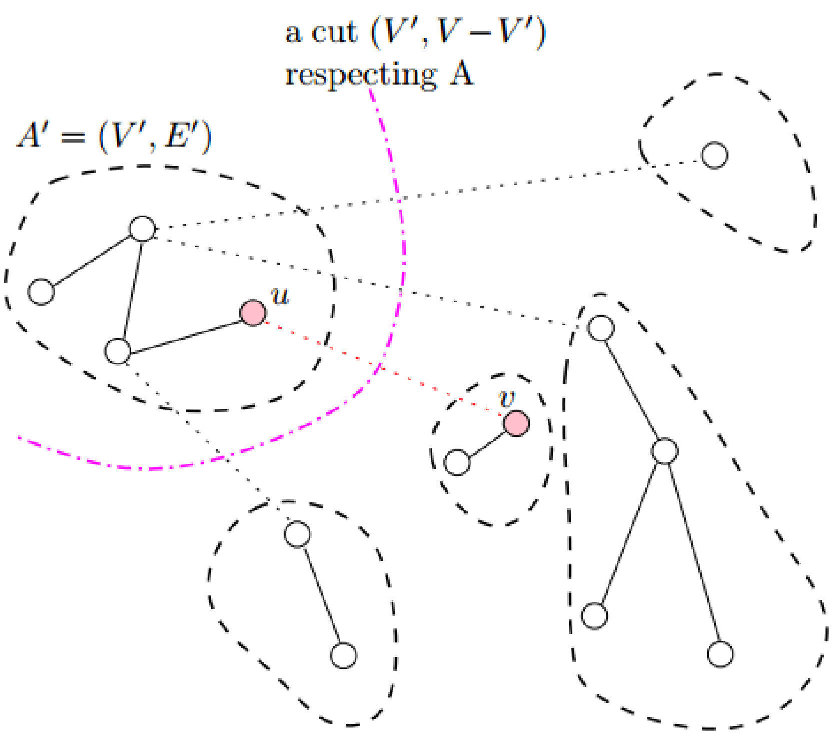


- Let $A' = (V', E')$ denote the tree of the forest A that contains u. Consider the cut $(V', V - V')$.
- There is no edge in A that crosses this cut, so the cut respects A.

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut

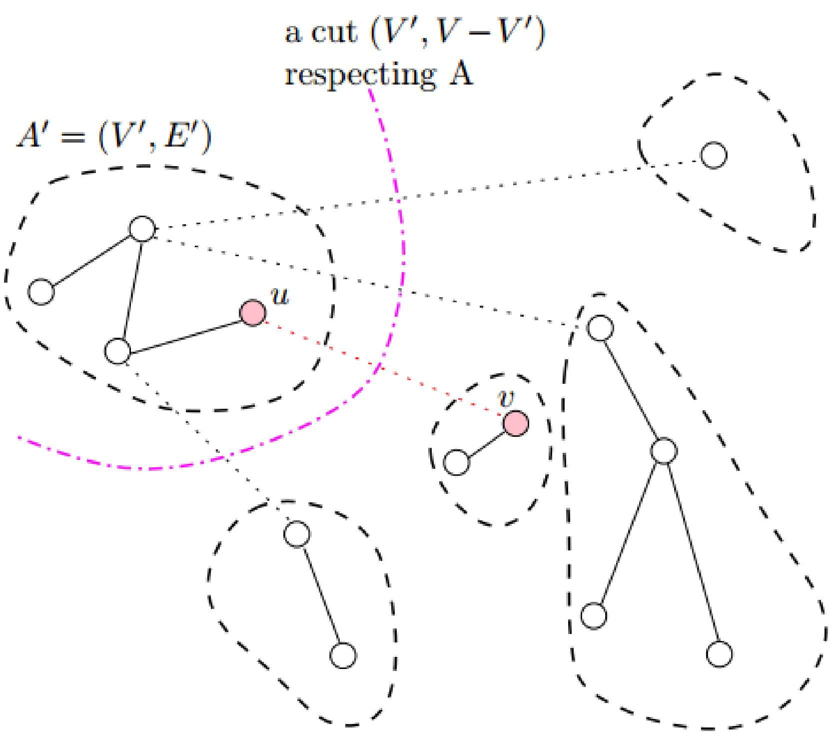


- Let $A' = (V', E')$ denote the tree of the forest A that contains u. Consider the cut $(V', V - V')$.
- There is no edge in A that crosses this cut, so the cut respects A.
- Since adding (u, v) to A' does not induce a cycle, (u, v) crosses the cut.

Why Kruskal's Algorithm is Correct?

Let A be the edge set selected by Kruskal's Algorithm, and let (u, v) be the edge to be added next. It suffices to show that

- there is a cut that respects A, and
- (u, v) is the light edge crossing this cut



- Let $A' = (V', E')$ denote the tree of the forest A that contains u. Consider the cut $(V', V - V')$.
- There is no edge in A that crosses this cut, so the cut respects A.
- Since adding (u, v) to A' does not induce a cycle, (u, v) crosses the cut.
- Moreover, since (u, v) is currently the smallest edge, (u, v) is the light edge crossing the cut.

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$

$A \leftarrow \{\}$;

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle\{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\}\rangle$

$A \leftarrow \{\}$;

for $u \in V$ **do**

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$

$A \leftarrow \{\}$;

for $u \in V$ **do**

  Create-Set(u);

end

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E| \log |E|)$
// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
  Create-Set(u);// $O(|V|)$
end

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E| \log |E|)$

// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$

$A \leftarrow \{\}$;

for $u \in V$ **do**

  Create-Set(u);// $O(|V|)$

end

for $e_i \in E$ **do**

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E|\log|E|)$
// After sorting $E = \langle\{u_1, v_1\}, \{u_2, v_2\}, \ldots, \{u_{|E|}, v_{|E|}\}\rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
    Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
    // $O(|E|\log|V|)$
    if *Find-Set*(u_i)$\neq$*Find-Set*(v_i) **then**

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E|\log|E|)$
// After sorting $E = \langle\{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\}\rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
   Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
   // $O(|E|\log|V|)$
   if *Find-Set*$(u_i)\neq$*Find-Set*(v_i) **then**
      add $\{u_i, v_i\}$ to A;

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E| \log |E|)$
// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
  Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
  // $O(|E| \log |V|)$
  if *Find-Set*$(u_i) \neq$ *Find-Set*(v_i) **then**
    add $\{u_i, v_i\}$ to A;
    Union(u_i, v_i);
  end
end

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E| \log |E|)$
// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
  Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
  // $O(|E| \log |V|)$
  if *Find-Set*$(u_i) \neq$*Find-Set*(v_i) **then**
    add $\{u_i, v_i\}$ to A;
    Union(u_i, v_i);
  end
end
return

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G
Sort E in increasing order by weight w;// $O(|E| \log |E|)$
// After sorting $E = \langle \{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\} \rangle$
$A \leftarrow \{\}$;
for $u \in V$ **do**
  Create-Set(u);// $O(|V|)$
end
for $e_i \in E$ **do**
  // $O(|E| \log |V|)$
  if *Find-Set*$(u_i) \neq$*Find-Set*(v_i) **then**
    add $\{u_i, v_i\}$ to A;
    Union(u_i, v_i);
  end
end
return A;

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G
Output: MST of G

```
Sort E in increasing order by weight w;// O(|E| log |E|)
// After sorting E = ⟨{u_1, v_1}, {u_2, v_2}, ..., {u_|E|, v_|E|}⟩
A ← {};
for u ∈ V do
 | Create-Set(u);// O(|V|)
end
for e_i ∈ E do
 | // O(|E| log |V|)
 | if Find-Set(u_i)≠Find-Set(v_i) then
 | | add {u_i, v_i} to A;
 | | Union(u_i, v_i);
 | end
end
return A;
```

Remark: With a proper implementation of Union-Find, Kruskal's algorithm has running time $O(|E|\log|E|) = O(|E|\log|V|)$.

Time Complexity of Kruskal's Algorithm

Input: A graph $\boldsymbol{G}$, a matrix $\boldsymbol{w}$ representing the weights between vertices in G

Output: MST of G

Sort E in increasing order by weight w;// $O(|E|\log|E|)$

// After sorting $E = \langle\{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_{|E|}, v_{|E|}\}\rangle$

$A \leftarrow \{\}$;

for $u \in V$ **do**

  Create-Set(u);// $O(|V|)$

end

for $e_i \in E$ **do**

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

  if *Find-Set*$(u_i)\neq$*Find-Set*(v_i) **then**

    add $\{u_i, v_i\}$ to A;

    Union(u_i, v_i);

  end

end

return A;

$\log|E| \leq \log|V|^2 = 2\log|V| = O(\log|V|)$

Remark: With a proper implementation of Union-Find, Kruskal's algorithm has running time $O(|E|\log|E|) = O(|E|\log|V|)$.

Summary

- Prim's algorithm always grows one tree.
- Kruskal's algorithm grows a collection of trees, namely a forest.
- Both Prim's algorithm and Kruskal's algorithm take O($|E|\log|V|$) time, but they adopt different data structures.


thank you
danke
gracias
merci
obrigado
謝謝
teşekkür ederim
ngiyabonga
go raibh maith agat
mochchakkeram
dziękuje
bedankt
sukriya
kop khun krap
terima kasih
tapadh leat
dank je
arigatô
dakujem
grazie
спасибо
Баярлалаа
рахмат
хвала
мерси
감사합니다
ありがとう
ευχαριστώ
tack
takk
welalin
barka
mersi
faafetai lava
vinaka
blagodaram
misaotra
matondo
paldies
grazzi
mahalo
asante
manana
obrigada
tenki
murakoze
chokrane
mamnun
дякую
djiere dieuf
tau
kiitos
dankie
dhanyavad
nanni
nandri
mauruuru
köszönöm
hvala
gràçie
bayarlalaa
enkosi
sobodi
dĕkuji
sagolun
mèsi
didi madloba
najis tuke
kam sah hamnida
rahmat
chnorakaloutioun
gratias ago
gràcies
sulpáy
taiku
tanemirt
rahmet
xièxie
diolch
dhanyavadagalu
shukriya
mercé
trugarez
dankon
akun
ačiū
spas
kia ora
спасибі