Minimum Spanning Trees and Prim's Algorithm

Version of September 23, 2016

Outline

- Spanning trees and minimum spanning trees (MST).

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.
- Prim's algorithm for the MST problem.
  - The idea
  - The algorithm
  - Analysis

Spanning Trees

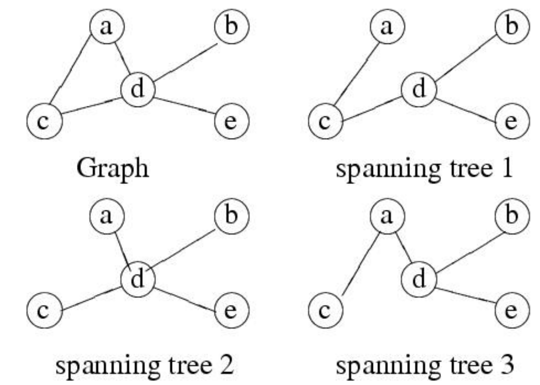
Definition

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

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.

Theorem

Every connected graph has a spanning tree.

Question

Why is this true?

Spanning Trees

Theorem

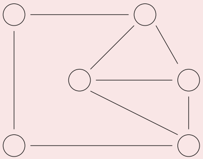
Every connected graph has a spanning tree.

Question

Why is this true?

Question

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



Weighted Graphs

Definition

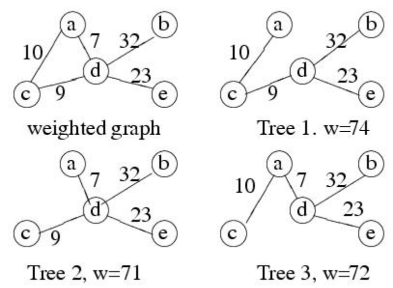
A weighted graph is a graph, in which each edge has a weight (some real number) Could denote length, time, strength, etc.

Weighted Graphs

Definition

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

Example

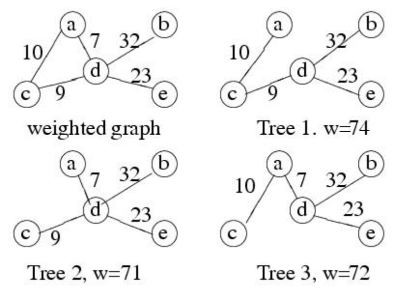


Weighted Graphs

Definition

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

Example



Definition

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

Minimum Spanning Trees

Definition

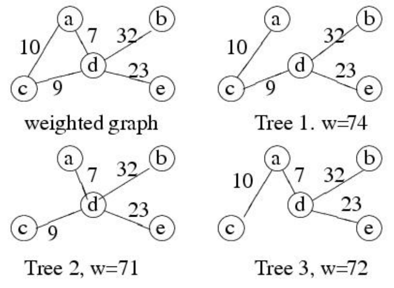
A Minimum spanning tree (MST) of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

Minimum Spanning Trees

Definition

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

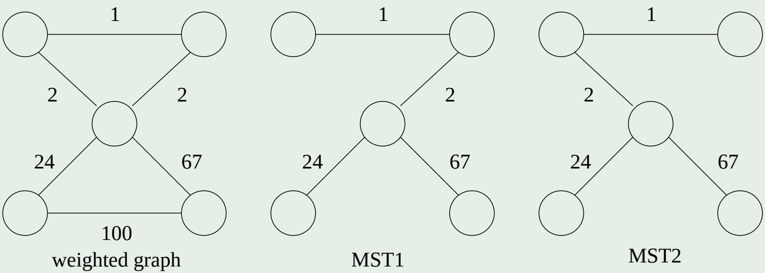
Example



Remark

The minimum spanning tree may not be unique

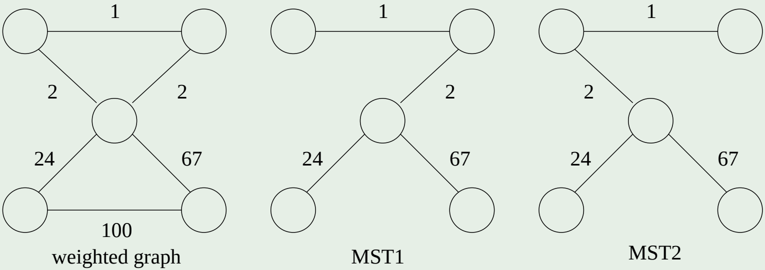
Example



Remark

The minimum spanning tree may not be unique

Example



Note: if the weights of all the edges are distinct, MST is provably unique (proof will follow from later results).

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.

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.

Outline

- Spanning trees and minimum spanning trees (MST).

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.
- Prim's algorithm for the MST problem.
    - The idea
    - The algorithm
    - Analysis

A tree is an acyclic graph

A tree is an acyclic graph

1. start with an empty graph

Tools for solving the MST Problem

A tree is an acyclic graph

1. start with an empty graph
2. try to add edges one at a time, subject to not creating a cycle

Tools for solving the MST Problem

A tree is an acyclic graph

1. start with an empty graph
2. try to add edges one at a time, subject to not creating a cycle
3. if after adding each edge we are sure that the resulting graph is a subset of some minimum spanning tree, then, after $n-1$ steps we are done.

Tools for solving the MST Problem

A tree is an acyclic graph

1. start with an empty graph
2. try to add edges one at a time, subject to not creating a cycle
3. if after adding each edge we are sure that the resulting graph is a subset of some minimum spanning tree, then, after $n - 1$ steps we are done.

Hard part is ensuring (3)!

Generic Algorithm for MST problem

Definition

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

Generic Algorithm for MST problem

Definition

Let A be a set of edges such that $A \subseteq T$, where T is some MST. Edge (u, v) is 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 some MST. Edge (u, v) is 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

Generic Algorithm for MST problem

Definition

Let A be a set of edges such that $A \subseteq T$, where T is some MST. Edge (u, v) is 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 some MST. Edge (u, v) is 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, w)

```
begin
    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
end
```

Some Definitions

Definition

Let $G = (V, E)$ be a connected and undirected graph.

Definition

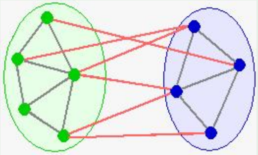
Let $G = (V, E)$ be a connected and undirected graph.
A cut $(S, V - S)$ of G is a partition of V.

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

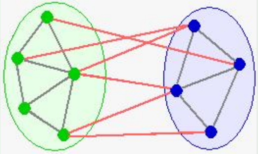


Some Definitions

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

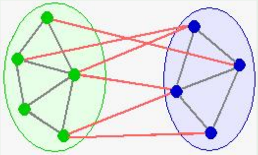
An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.

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

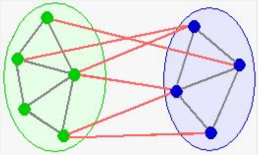
An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S, and the other is in $V - S$.
A cut respects a set A of edges if no edge in A crosses the cut.

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?

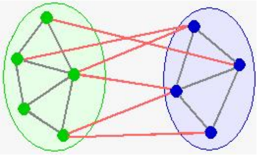
Lemma

- *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

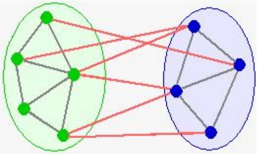
- *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E*
- *A be a subset of E that is included in some minimum spanning tree for G.*

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.

This implies we can find a safe edge by



How to Find a Safe Edge?

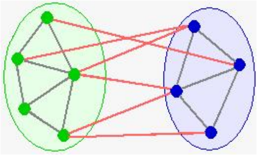
Lemma

- *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.



This implies we can find a safe edge by

1. first finding a cut that respects A,

How to Find a Safe Edge?

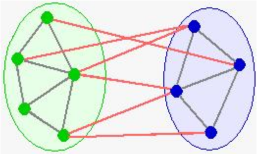
Lemma

- *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.



This implies we can find a safe edge by

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

That light edge is a safe edge.

Proof

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

Proof

- Let $A \subseteq T$, where T is a MST.
- Case 1: $(u, v) \in T$

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 the unique 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 the unique 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)

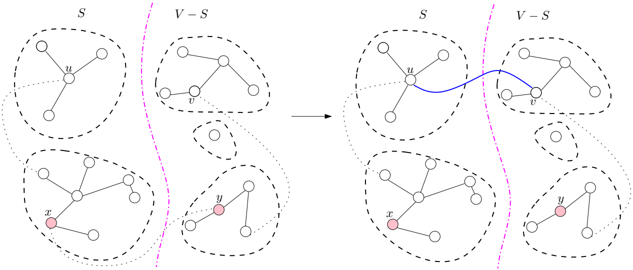
- Case 2: $(u, v) \notin T$
  - *Idea: construct another MST $T^{'}$ s.t. $A \cup \{(u, v)\} \subseteq T'$.*
  - Consider the unique path P in T from u to v.
  - Since u and v are on opposite sides of the cut $(S, V - S)$,
    - There is at least one edge in P that crosses the cut.
    - Let (x, y) be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



- Adding (u, v) to T, creates a cycle with P.

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.
  In particular, adding (u, v) and removing (x, y) creates a new tree $T^{'}$.

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.
  In particular, adding (u, v) and removing (x, y) creates a new tree $T^{'}$.
- The weight of $T^{'}$ is

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.
  In particular, adding (u, v) and removing (x, y) creates a new tree $T^{'}$.

- The weight of $T^{'}$ is

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

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.
  In particular, adding (u, v) and removing (x, y) creates 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, $W(T) \le W(T')$ so $W(T') = W(T)$ and T is also an MST.

- Adding (u, v) to T, creates a cycle with P.
  Removing any edge from this cycle gives a tree again.
  In particular, adding (u, v) and removing (x, y) creates 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, $W(T) \leq W(T')$ so $W(T') = W(T)$ and T is also an MST.
- But $A \cup \{(u, v)\} \subseteq T^{'}$, so (u, v), is safe for A.
- The Lemma is proved.

Outline

- Spanning trees and minimum spanning trees (MST).

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.

Outline

- Spanning trees and minimum spanning trees (MST).
- Tools for solving the MST problem.
- Prim's algorithm for the MST problem.
  - The idea
  - The algorithm
  - Analysis

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 proof carefully, you will notice that the choice of a cut (and hence a 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 proof carefully, you will notice that the choice of a cut (and hence a corresponding light edge) in each iteration is arbitrary.
- We can select any cut (that respects current edge set A) and find a 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 proof carefully, you will notice that the choice of a cut (and hence a corresponding light edge) in each iteration is arbitrary.
- We can select any cut (that respects current edge set A) and find a light edge crossing that cut to proceed.
- Different ways of chosing cuts correspond to different algorithms.
- The two major ones are Prim's algorithm and Kruskal's algorithm,

Prim's Algorithm

Prim's algorithm

- grows a tree, adding a new light edge in each iteration, creating a new tree.

Prim's Algorithm

Prim's algorithm

- grows a tree, adding a new light edge in each iteration, creating a new tree.

Growing a tree

Prim's Algorithm

Prim's algorithm

- grows a tree, adding a new light edge in each iteration, creating a new tree.

Growing a tree

- Start by picking any vertex r to be the root of the tree.

Prim's algorithm

- grows a tree, adding a new light edge in each iteration, creating a new tree.

Growing 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

- grows a tree, adding a new light edge in each iteration, creating a new tree.

Growing 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 tree and add it to the tree.

Prim's algorithm

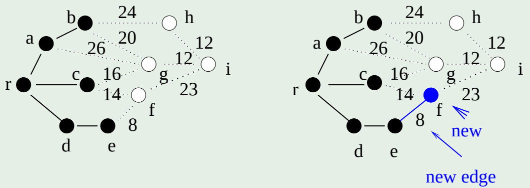
- grows a tree, adding a new light edge in each iteration, creating a new tree.

Growing 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 tree and add it to the tree.

We will show that these steps can be implemented in total $O(E \cdot \log V)$.

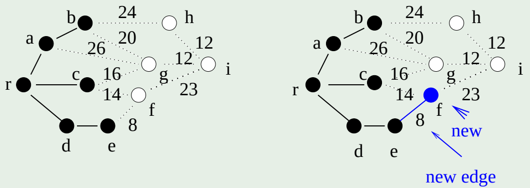
Example



Step 0:

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

Example



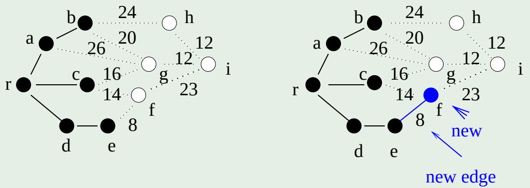
Step 0:

- 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 $V \setminus S$.

Example



Step 0:

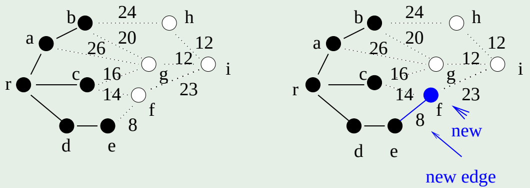
- Choose any element r; set $S = \{r\}$ and $A = \emptyset$.
- (Take r as the root of our spanning tree.)

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.)

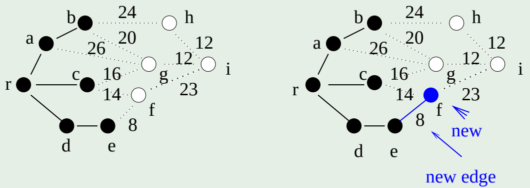
Step 1:

- 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);

Example



Step 0:

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

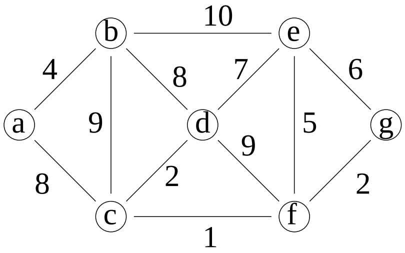
Step 1:

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

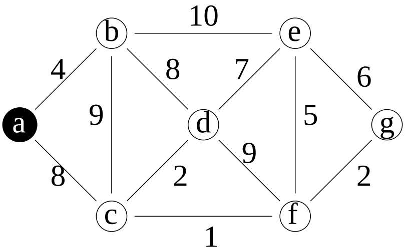
Step 2:

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

Worked Example



Connected graph



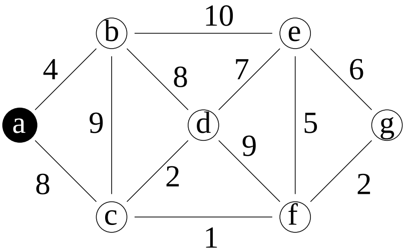
Step 0

S={a}

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

lightest edge = {a,b}

Prim's Example – Continued



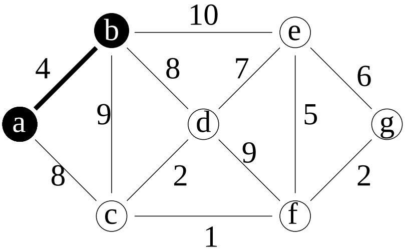
Step 1.1 before

S={a}

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

A={}

lightest edge = {a,b}



Step 1.1 after

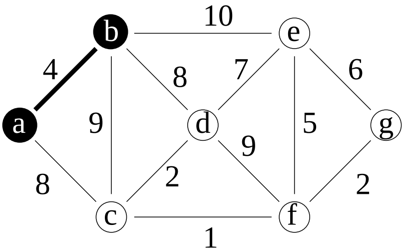
S={a,b}

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

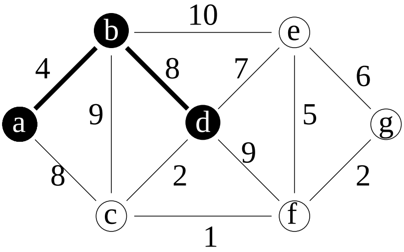
A={{a,b}}

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

Prim's Example – Continued

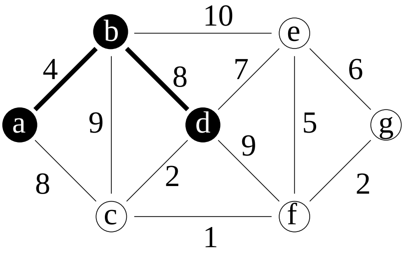


Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

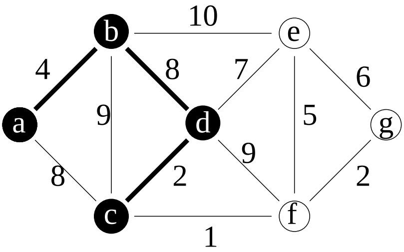


Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

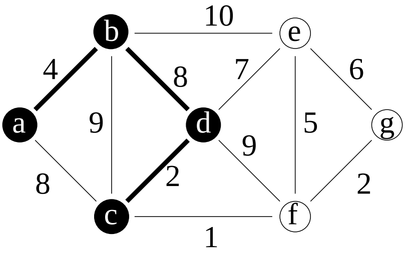
Prim's Example – Continued



Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}



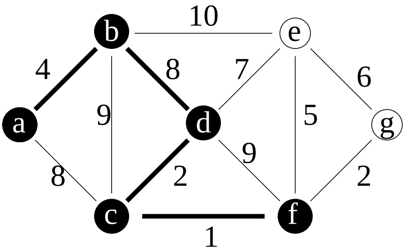
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

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

lightest edge = {c,f}



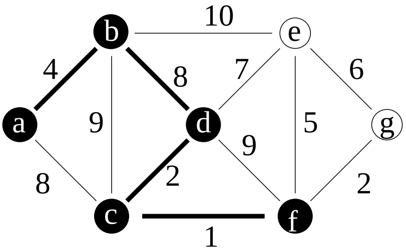
Step 1.4 after

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

V \ S = {e,g}

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

lightest edge = {f,g}



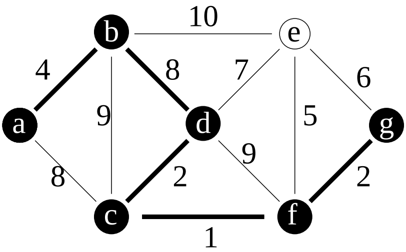
Step 1.5 before

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

V \ S = {e,g}

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

lightest edge = {f,g}



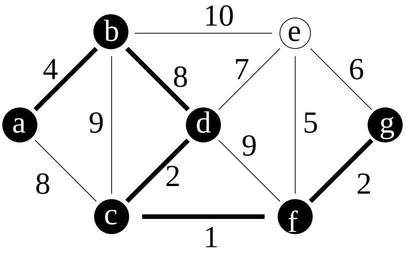
Step 1.5 after

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

V \ S = {e}

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

lightest edge = {f,e}



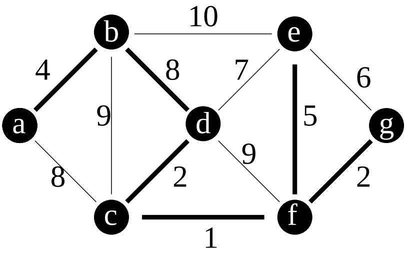
Step 1.6 before

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

V \ S = {e}

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

lightest edge = {f,e}



Step 1.6 after

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

V \ S = {}

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

MST completed

- Spanning trees and minimum spanning trees (MST).

Outline

- Spanning trees and minimum spanning trees (MST).
- Strategy for solving the MST problem.

Outline

- Spanning trees and minimum spanning trees (MST).
- Strategy for solving the MST problem.
- Prim's algorithm for the MST problem.
  - The idea
  - The algorithm
  - Analysis

Recall Idea of Prim's Algorithm

Step 0: Choose any element r and 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 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 $A = \emptyset$.
(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 the minimum spanning tree (S, A); Otherwise go to Step 1.

Questions

1. Why does this produce a minimum spanning tree?
2. How does the algorithm find the lightest edge and update A efficiently?
3. How does the algorithm update S efficiently?

Prim's Algorithm

Question

How does the algorithm update S efficiently?

Question

How does the algorithm update S efficiently?

Answer: Color the vertices.

- Initially all are white.

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 a lightest edge and update A efficiently?

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 a 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 a lightest edge and update A efficiently?

Answer:

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

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($u, new\text{-}key$): Decrease u's key value to $new\text{-}key$.

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, $new\text{-}key$): Decrease u's key value to $new\text{-}key$.

Remark: We already saw how to implement Insert and Extract-Min (and Delete) in $O(\log |Q|)$ time.
Same ideas can also be used to implement Decrease-Key in $O(\log |Q|)$ time.
Alternatively, can implement Decrease-Key using Delete followed by Insert.

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 $V \setminus S$,

Using a Priority Queue to Find the Lightest Edge

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

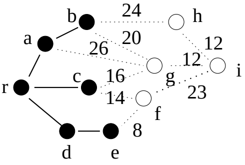
- u is a vertex in $V \setminus S$,
- $key[u]$ is the weight of the lightest edge from u to any vertex in S.

(The endpoint of this edge in S is stored in $pred[u]$, which is used to build the MST tree.)

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

- u is a vertex in $V \setminus S$,
- $key[u]$ 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.)



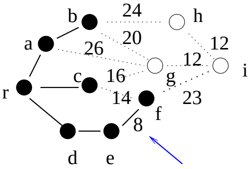
key[f] = 8, pred[f] = e

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

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

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

→ f has the minimum key



new edge

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

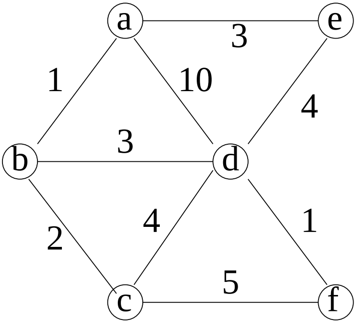
```
begin
    foreach u ∈ V do
        color[u] = WHITE; key[u] = +∞; // initialize
    end
    key[r] = 0; pred[r] = NIL; // start at root
    Q = new PriQueue(V); // put vertices in Q
    while Q is nonempty do
        u= Q.Extract-Min(); // lightest edge
        foreach v ∈ adj[u] do
            if (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
end
```

Description of Prim's Algorithm...

When the algorithm terminates, $Q = \emptyset$ and the MST is

$$T = \{\{v, pred[v]\} : v \in V \setminus \{r\}\}.$$

- The pred pointers define the MST as an inverted tree rooted at r.



u	a	b	c	d	e	f
key[u]						
pred[u]						

Outline

- Spanning trees and minimum spanning trees (MST).

Outline

- Spanning trees and minimum spanning trees (MST).
- Strategy for solving the MST problem.

Outline

- Spanning trees and minimum spanning trees (MST).
- Strategy for solving the MST problem.
- Prim's algorithm for the MST problem.
    - The idea
    - The algorithm
    - Analysis

```
begin
    foreach u ∈ V do
        key[u] = +∞; color[u] = WHITE; // O(V)
    end
    key[r] = 0; pred[r] = NIL;
    Q = new PriQueue(V); // O(V)
    while Q is nonempty do
        u= Q.Extract-Min(); // Do this for each vertex
        foreach v ∈ adj[u] do
            // Do the following for each edge twice
            if (color[v] = WHITE)&&(w[u, v] < key[v]) then
                key[v] = w[u, v]; pred[v] = u;
                Q.Decrease-Key(v, key[v]); // This is bottleneck
            end
        end
        color[u] = BLACK;
    end
end
```

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

- ($O(|V|)$ for creating new Priority Queue
- $O(\log V)$ 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 V.
  Total cost: $O(E \log V)$.

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

Going Further

A more advanced Priority Queue implementation called *Fibonnaci Heaps* allow

- $O(1)$ for inserting each item
- $O(\log |V|)$ for Extract-Min
- **O(1) (amortized) for each Decrease-Key**

Since algorithm performs $|V|$ Inserts, $|V|$ Extract-Mins and at most E Decrease-Keys this leads to a $O(|E| + |V| \log |V|)$ algorithm, improving upon the $O(E \log V)$ more naive implementation.