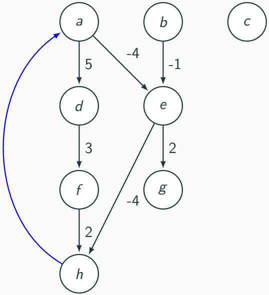
Pre-lecture brain teaser

You are given a directed acyclic graph (DAG) $G = (V, E)$ that contains positive and negative edges with $|V| = n$ and $|E| = m$. You are able to place one edge (weight=0) with the aim of creating smallest cycle possible. Describe an algorithm (lowest running time possible) to produce this min cost cycle.



ECE-374-B: Lecture 18 - Minimum spanning trees

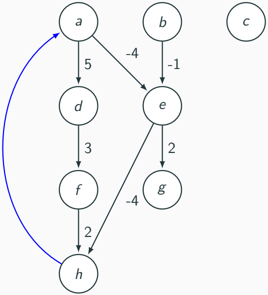
Instructor: Abhishek Kumar Umrawal

April 02, 2024

University of Illinois at Urbana-Champaign

Pre-lecture brain teaser

You are given a directed acyclic graph (DAG) $G = (V, E)$ that contains positive and negative edges with $|V| = n$ and $|E| = m$. You are able to place one edge (weight=0) with the aim of creating smallest cycle possible. Describe an algorithm (lowest running time possible) to produce this min cost cycle.



Minimum Spanning Tree

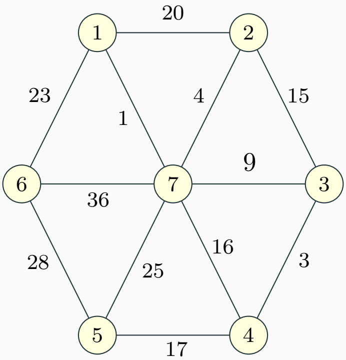
The Problem

Minimum Spanning Tree

Input Connected graph $G = (V, E)$ with edge costs

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest

- T is the minimum spanning tree (MST) of G

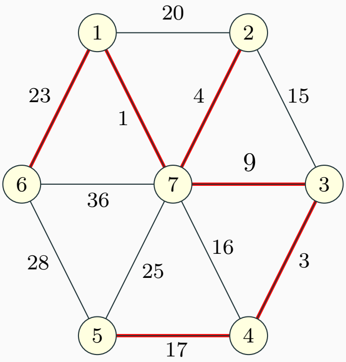


Minimum Spanning Tree

Input Connected graph $G = (V, E)$ with edge costs

Goal Find $T \subseteq E$ such that (V, T) is connected and total cost of all edges in T is smallest

- T is the minimum spanning tree (MST) of G



Applications

- Network Design
  - Designing networks with minimum cost but maximum connectivity
- Approximation algorithms
  - Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
- Cluster Analysis

Some history

The first algorithm for MST was first published in 1926 by Otakar Borůvka as a method of constructing an efficient electricity network for Moravia. From his memoirs:

> *My studies at poly-technical schools made me feel very close to engineering sciences and made me fully appreciate technical and other applications of mathematics. Soon after the end of World War I, at the beginning of the 192Os, the Electric Power Company of Western Moravia, Brno, was engaged in rural electrification of Southern Moravia. In the framework of my friendly relations with some of their employees, I was asked to solve, from a mathematical standpoint, the question of the most economical construction of an electric power network. I succeeded in finding a construction-as it would be expressed today-of a maximal connected subgraph of minimum length, which I published in 1926 (i.e., at a time when the theory of graphs did not exist).*

There is some work in 1909 by a Polish anthropologist Jan Czekanowski on clustering, which is a precursor to MST.

Some graph theory

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.
- Subgraph H of G is <u>spanning</u> for G, if G and H have same connected components.

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.
- Subgraph H of G is <u>spanning</u> for G, if G and H have same connected components.
- A graph G is connected $\iff$ it has a spanning tree.

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.
- Subgraph H of G is spanning for G, if G and H have same connected components.
- A graph G is connected $\iff$ it has a spanning tree.
- Every tree has a leaf (i.e., vertex of degree one).

Some basic properties of Spanning Trees

- Tree = undirected graph in which any two vertices are connected by exactly one path.
- Tree = a connected graph with no cycles.
- Subgraph H of G is spanning for G, if G and H have same connected components.
- A graph G is connected $\iff$ it has a spanning tree.
- Every tree has a leaf (i.e., vertex of degree one).
- Every spanning tree of a graph on n nodes has $n-1$ edges.

Exchanging an edge in a spanning tree

Lemma
$T = (V, E_T)$: a spanning tree of $G = (V, E)$. For every non-tree edge $e \in E \setminus E_T$ there is a unique cycle C in $T + e$. For every edge $f \in C - \{e\}$, $T - f + e$ is another spanning tree of G.

Safe and unsafe edges

Assumption
Edge costs are distinct, that is no two edge costs are equal.

Cuts

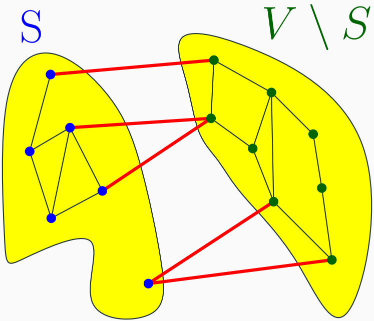
Definition
Given a graph $G = (V, E)$, a cut is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

Cuts

Definition
Given a graph $G = (V, E)$, a cut is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

Edges having an endpoint on both sides are the edges of the cut.

A cut edge is crossing the cut.



$$(S, V \setminus S) = \{uv \in E \mid u \in S, v \in V \setminus S\}.$$

Safe and Unsafe Edges

Definition
An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Safe and Unsafe Edges

Definition
An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Definition
An edge $e = (u, v)$ is an unsafe edge if there is some cycle C such that e is the unique maximum cost edge in C.

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$.

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

- If x, y in same connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

- If x, y in same connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
  $\implies$ e is unsafe.

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

- If x, y in same connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
  $\implies$ e is unsafe.
- If x and y are in diff connected component of $G_{<w(e)}$,

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

- If x, y in same connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
  $\implies$ e is unsafe.
- If x and y are in diff connected component of $G_{<w(e)}$,
  Let S the vertices of connected component of $G_{<w(e)}$ containing x.
  The edge e is cheapest edge in cut $(S, V \setminus S)$.

Every edge is either safe or unsafe

Proposition
If edge costs are distinct then every edge is either safe or unsafe.

Proof.
Consider any edge $e = uv$.

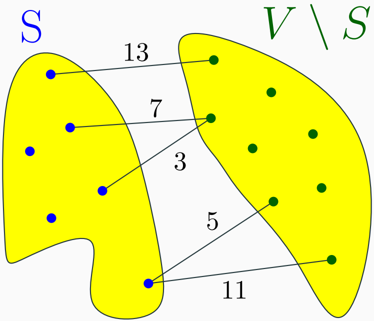
Let $G_{<w(e)} = (V, \{xy \in E \mid w(xy) < w(e)\})$. (Observe that $e \notin E\left(G_{<w(e)}\right)$.)

- If x, y in same connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
  $\implies$ e is unsafe.
- If x and y are in diff connected component of $G_{<w(e)}$,
  Let S the vertices of connected component of $G_{<w(e)}$ containing x.
  The edge e is cheapest edge in cut $(S, V \setminus S)$.
  $\implies$ e is safe.

$\square$

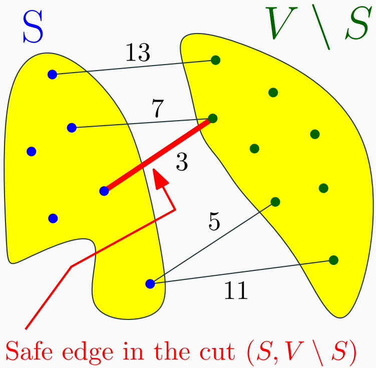
Safe edge - Example...

Every cut identifies one safe edge...



Safe edge - Example...

Every cut identifies one safe edge...

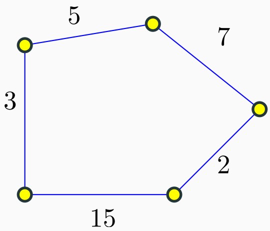


...the cheapest edge in the cut.

Note: An edge e may be a safe edge for many cuts!

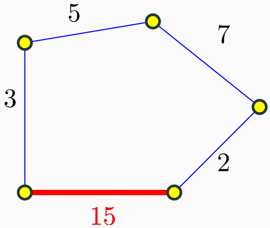
Unsafe edge - Example...

Every cycle identifies one <u>unsafe</u> edge...



Unsafe edge - Example...

Every cycle identifies one unsafe edge...



...the most expensive edge in the cycle.

Example

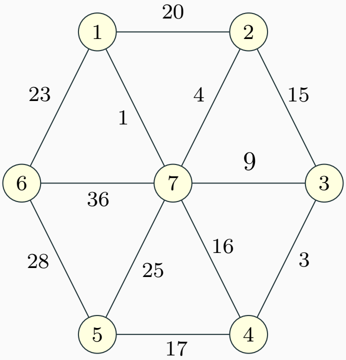


Figure 1: Graph with unique edge costs. Safe edges are red, rest are unsafe.

Example

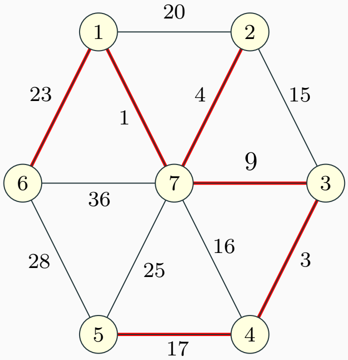


Figure 1: Graph with unique edge costs. Safe edges are red, rest are unsafe.

Example

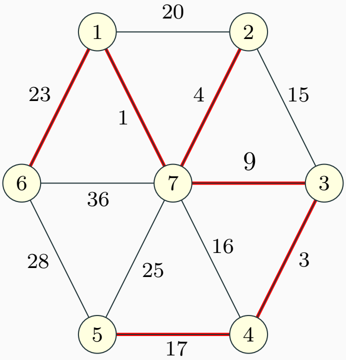


Figure 1: Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...

Some key observations

Lemma
If e is a safe edge then **every** *minimum spanning tree contains e.*

Lemma
If e is an unsafe edge then no MST of G contains e.

Why do we care about safety?

Safe edges must be in the MST

Correctness of MST Algorithms

- Many different MST algorithms
- All of them rely on some basic properties of MSTs, in particular the Cut Property to be seen shortly.

Key Observation: Cut Property

Lemma
If e is a safe edge then **every** *minimum spanning tree contains e.*

Key Observation: Cut Property

Lemma
If e is a safe edge then **every** *minimum spanning tree contains e.*

Proof.

- Suppose (for contradiction) e is not in MST T.
- Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
- Since $c_f > c_e$, $T' = (T \setminus \{f\}) \cup \{e\}$ is a spanning tree of lower cost!

Key Observation: Cut Property

Lemma
If e is a safe edge then **every** *minimum spanning tree contains e.*

Proof.

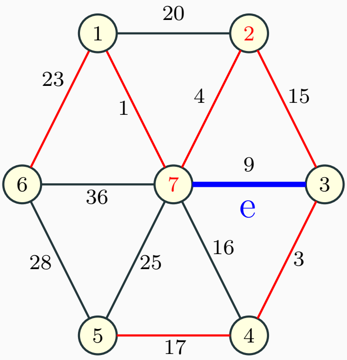
- Suppose (for contradiction) e is not in MST T.
- Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
- Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
- Since $c_f > c_e$, $T' = (T \setminus \{f\}) \cup \{e\}$ is a spanning tree of lower cost! Error: T' may not be a spanning tree!!

$\square$

Error in Proof: Example

Problematic example. $\mathbf{S} = \{\mathbf{1}, \mathbf{2}, \mathbf{7}\}$, $\mathbf{e} = (\mathbf{7}, \mathbf{3})$, $\mathbf{f} = (\mathbf{1}, \mathbf{6})$.

$\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.



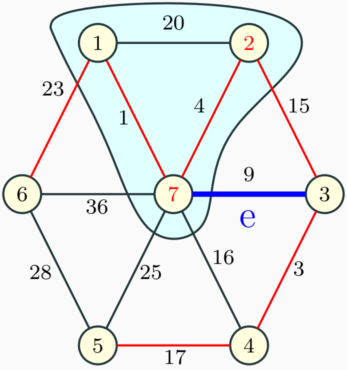
(A)

(A) Consider adding the edge f.

Error in Proof: Example

Problematic example. $\mathbf{S} = \{\mathbf{1}, \mathbf{2}, \mathbf{7}\}$, $\mathbf{e} = (\mathbf{7}, \mathbf{3})$, $\mathbf{f} = (\mathbf{1}, \mathbf{6})$.

$\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.



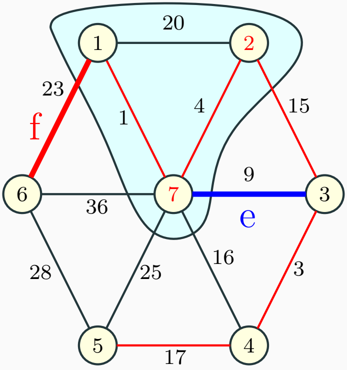
(B)

(A) Consider adding the edge f.

(B) It is safe because it is the cheapest edge in the cut.

Error in Proof: Example

Problematic example. $\mathbf{S} = \{\mathbf{1}, \mathbf{2}, \mathbf{7}\}$, $\mathbf{e} = (\mathbf{7}, \mathbf{3})$, $\mathbf{f} = (\mathbf{1}, \mathbf{6})$.
$\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.



(C)

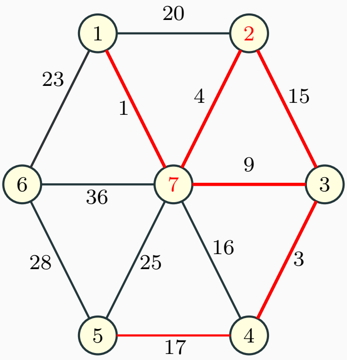
(A) Consider adding the edge f.

(B) It is safe because it is the cheapest edge in the cut.

(C) Lets throw out the edge e currently in the spanning tree which is more expensive than f and is in the same cut. Put it f instead...

Error in Proof: Example

Problematic example. $\mathbf{S} = \{\mathbf{1}, \mathbf{2}, \mathbf{7}\}$, $\mathbf{e} = (\mathbf{7}, \mathbf{3})$, $\mathbf{f} = (\mathbf{1}, \mathbf{6})$.
$\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.



(D)

(A) Consider adding the edge f.

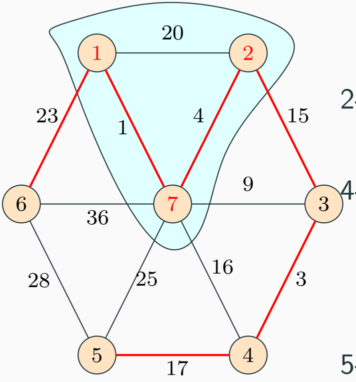
(B) It is safe because it is the cheapest edge in the cut.

(C) Lets throw out the edge e currently in the spanning tree which is more expensive than f and is in the same cut. Put it f instead...

(D) New graph of selected edges is not a tree anymore. BUG.

Proof of Cut Property

Proof.



- Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

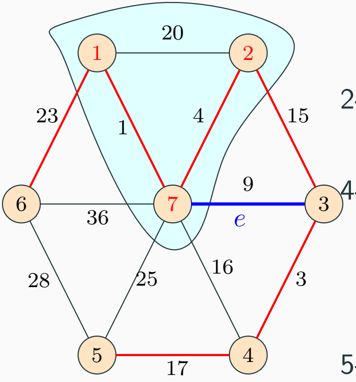
2- T is spanning tree: there is a unique path P from v to w in T

4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

5- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) $\square$

Proof of Cut Property

Proof.



- Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

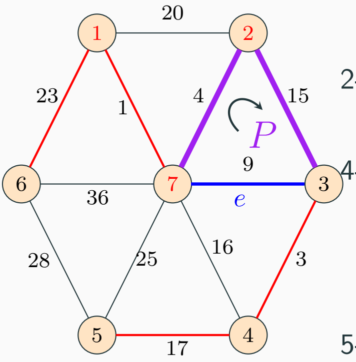
2- T is spanning tree: there is a unique path P from v to w in T

4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

5- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) $\square$

Proof of Cut Property

Proof.



- Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

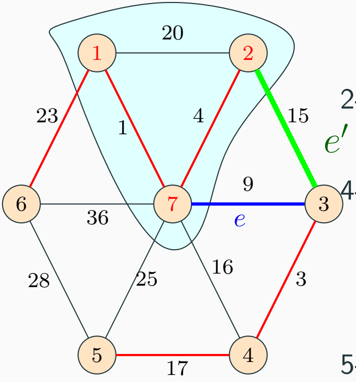
2- T is spanning tree: there is a unique path P from v to w in T

4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

5- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) $\square$

Proof of Cut Property

Proof.



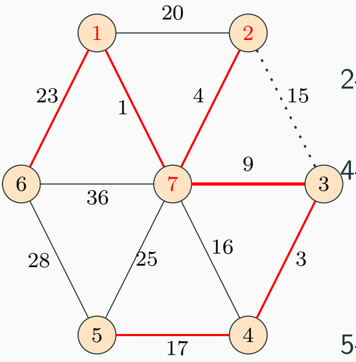
- Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

2- T is spanning tree: there is a unique path P from v to w in T

4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

5- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) $\square$

Proof.



- Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.

2- T is spanning tree: there is a unique path P from v to w in T

4- Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$

5- $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) $\square$

Proof of Cut Property (contd)

Observation
$T' = (T \setminus \{e'\}) \cup \{e\}$ *is a spanning tree.*

Proof.
T' is connected.

2- Removed $e' = (v', w')$ from T but v' and w' are connected by the path $P - f + e$ in T'. Hence T' is connected if T is.

T' is a tree

3- T' is connected and has $n - 1$ edges (since T had $n - 1$ edges) and hence T' is a tree

$\square$

The safe edges form the MST

Safe Edges form a connected graph

Lemma
Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.

Proof.

- Suppose not. Let S be a connected component in the graph induced by the safe edges.
- Consider the edges crossing S, there must be a safe edge among them since edge costs are distinct and so we must have picked it.

$\square$

Safe Edges do not contain a cycle

Lemma
Let G be a connected graph with distinct edge costs, then the set of safe edges does not contain a cycle.

Safe Edges form an MST

Corollary
Let G be a connected graph with distinct edge costs, then set of safe edges form the unique MST of G.

Safe Edges form an MST

Corollary
Let G be a connected graph with distinct edge costs, then set of safe edges form the unique MST of G.

Consequence: Every correct MST algorithm when G has unique edge costs includes exactly the safe edges.

The unsafe edges are NOT in the MST

Cycle Property

Lemma
If e is an unsafe edge then no MST of G contains e.

Cycle Property

Lemma
If e is an unsafe edge then no MST of G contains e.

Proof.
Exercise. □

Note: Cut and Cycle properties hold even when edge costs are not distinct. Safe and unsafe definitions do not rely on distinct cost assumption.

Borůvka's Algorithm

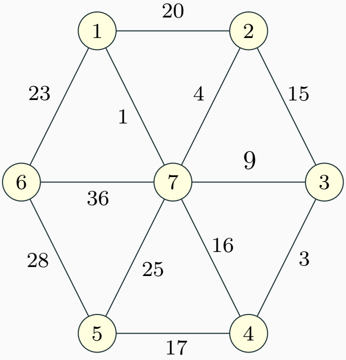
Borůvka's Algorithm

Simplest to implement. See notes.

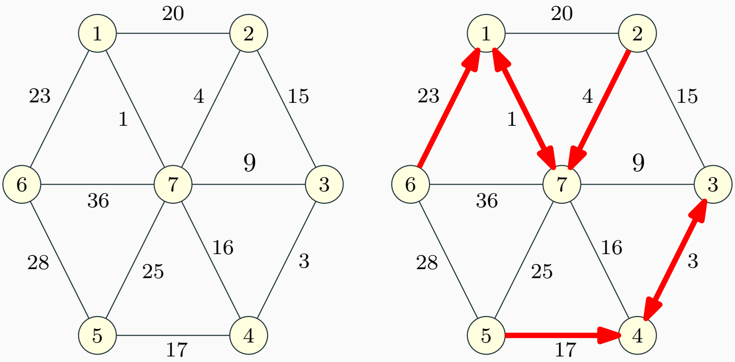
Assume G is a connected graph.

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

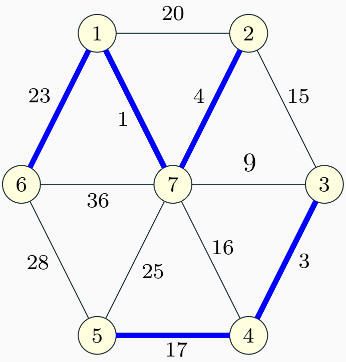
Borůvka's Algorithm



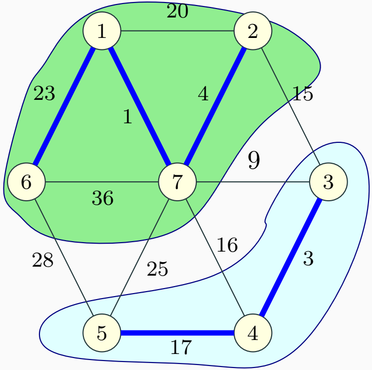
Borůvka's Algorithm



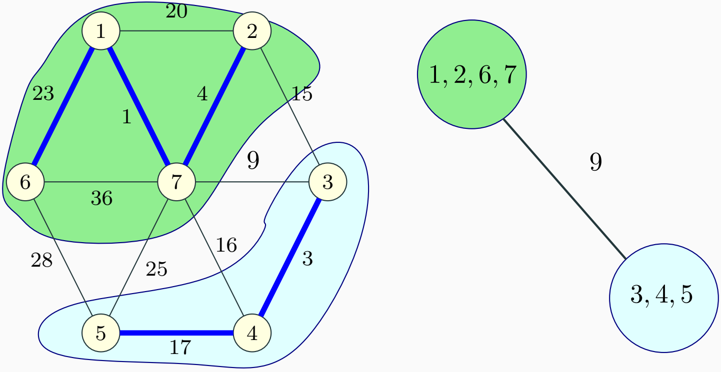
Borůvka's Algorithm



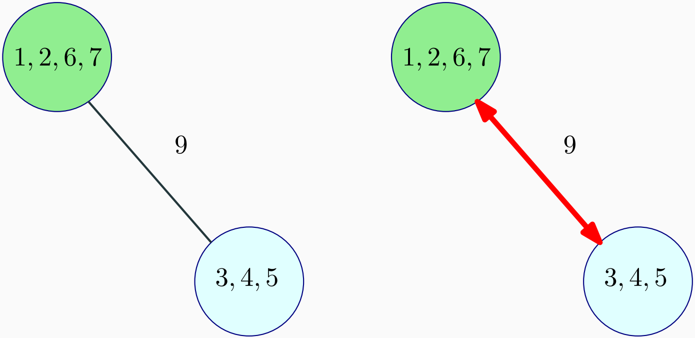
Borůvka's Algorithm



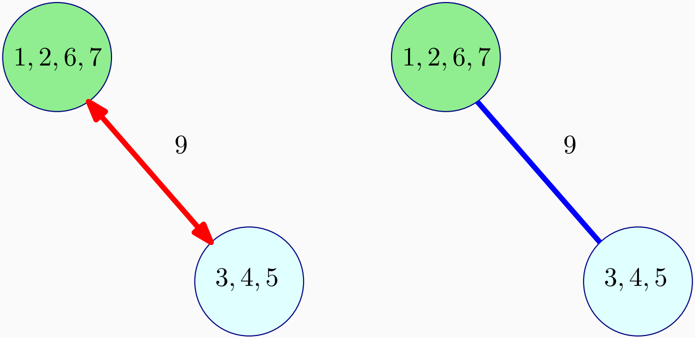
Borůvka's Algorithm



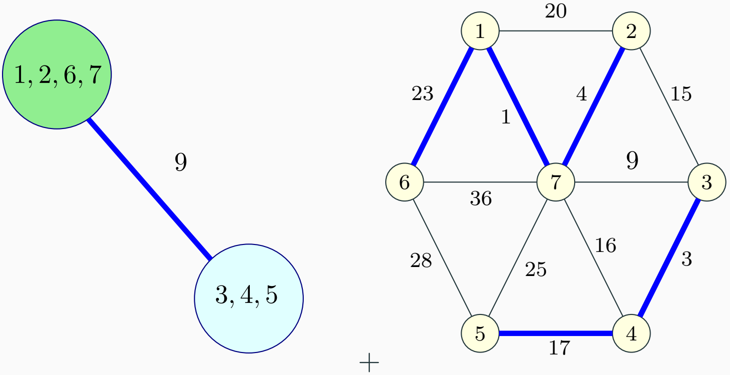
Borůvka's Algorithm



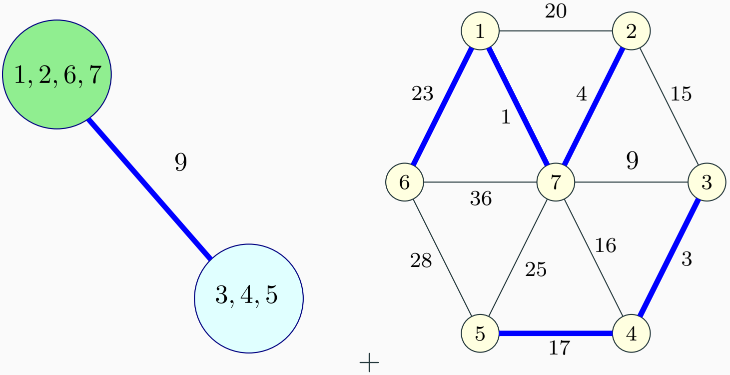
Borůvka's Algorithm



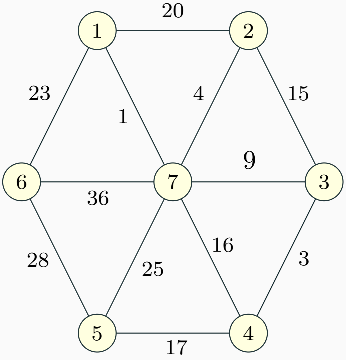
Borůvka's Algorithm

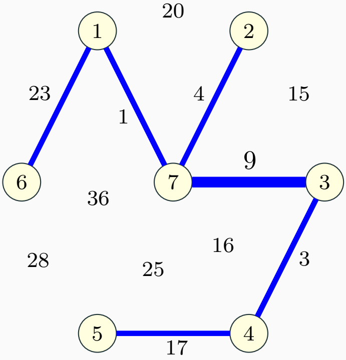


Borůvka’s Algorithm



Borůvka's Algorithm





Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

- $O(\log n)$ iterations of while loop. Why?

Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

- $O(\log n)$ iterations of while loop. Why? Number of connected components shrink by at least half since each component merges with one or more other components.
- Each iteration can be implemented in $O(m)$ time.

Running time:

Implementing Borůvka's Algorithm

No complex data structure needed.

```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

- $O(\log n)$ iterations of while loop. Why? Number of connected components shrink by at least half since each component merges with one or more other components.
- Each iteration can be implemented in $O(m)$ time.

Running time: $O(m \log n)$ time.

Kruskal's Algorithm

Greedy Template

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty do
    choose e ∈ E
    remove e from E
    if (e satisfies condition)
        add e to T
return the set T
```

Main Task: In what order should edges be processed? When should we add edge to spanning tree?

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

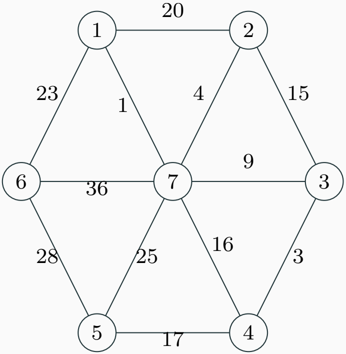


Figure 2: Graph G

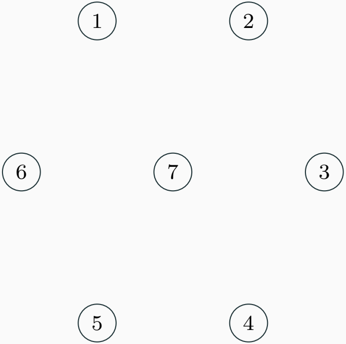


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

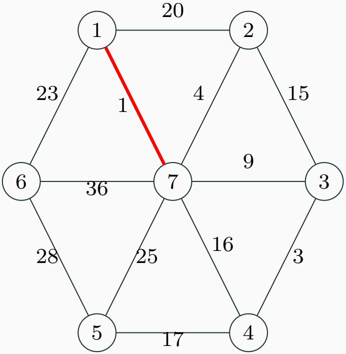


Figure 2: Graph G

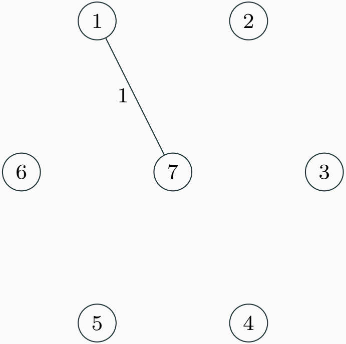


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

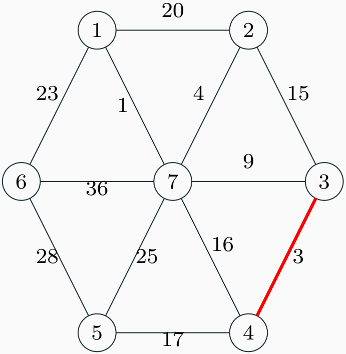


Figure 2: Graph G

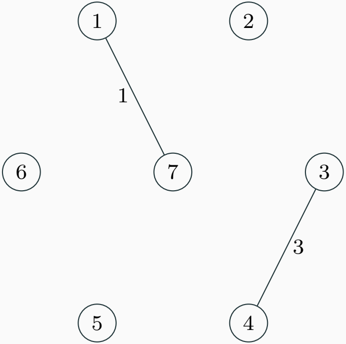


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

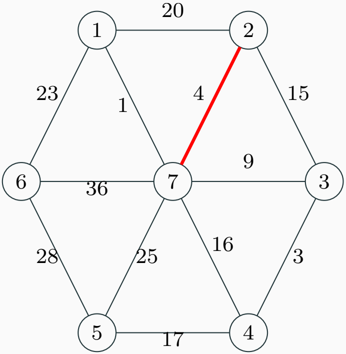


Figure 2: Graph G

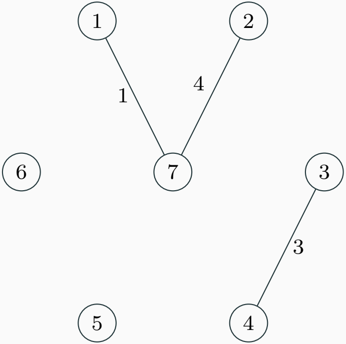


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

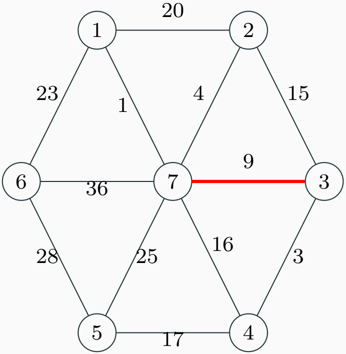


Figure 2: Graph G

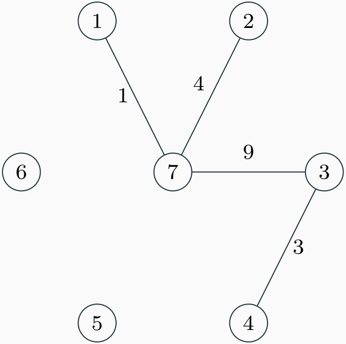


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

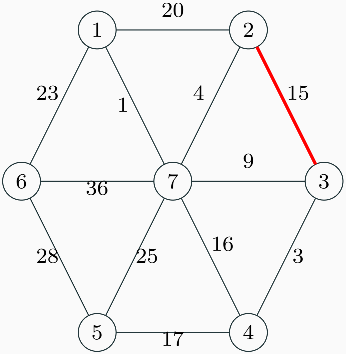


Figure 2: Graph G

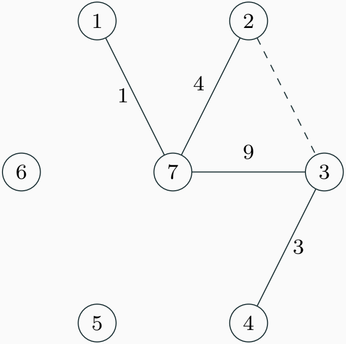


Figure 3: MST of G

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

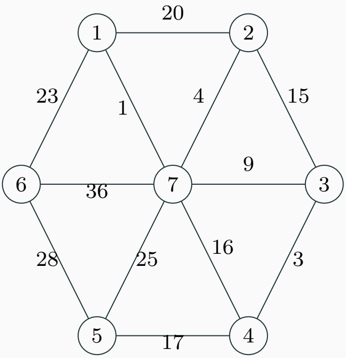


Figure 2: Graph G

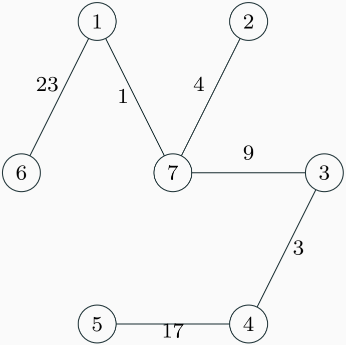


Figure 3: MST of G

Correctness of Kruskal's Algorithm

Kruskal's Algorithm
Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof of correctness.

- If $e = (u, v)$ is added to tree, then e is safe
  - When algorithm adds e let S and S' be the connected components containing u and v respectively
  - e is the lowest cost edge crossing S (and also S').
  - If there is an edge e' crossing S and has lower cost than e, then e' would come before e in the sorted order and would be added by the algorithm to T
- Set of edges output is a spanning tree

□

Implementing Kruskal's Algorithm

Kruskal's Algorithm

Kruskal_ComputeMST

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty do
    choose e ∈ E of minimum cost
    if (T ∪ {e} does not have cycles)
        add e to T
return the set T
```

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

- Presort edges based on cost. Choosing minimum can be done in $O(1)$ time

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

- Presort edges based on cost. Choosing minimum can be done in $O(1)$ time

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

- Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
- Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

- Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
- Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
- Total time $O(m \log m) + O(mn) = O(mn)$

Implementing Kruskal's Algorithm Efficiently

Kruskal_ComputeMST

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty do
    pick e = (u, v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

Implementing Kruskal's Algorithm Efficiently

```
Kruskal_ComputeMST
    Sort edges in E based on cost
    T is empty (* T will store edges of a MST *)
    each vertex u is placed in a set by itself
    while E is not empty do
        pick e = (u, v) ∈ E of minimum cost
        if u and v belong to different sets
            add e to T
            merge the sets containing u and v
    return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

Implementing Kruskal's Algorithm Efficiently

Kruskal_ComputeMST

```
Sort edges in E based on cost
T is empty (* T will store edges of a MST *)
each vertex u is placed in a set by itself
while E is not empty do
    pick e = (u, v) ∈ E of minimum cost
    if u and v belong to different sets
        add e to T
        merge the sets containing u and v
return the set T
```

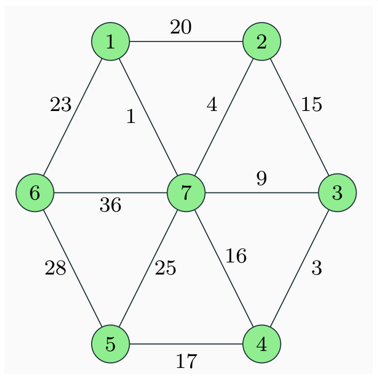
Need a data structure to check if two elements belong to same set and to merge two sets.

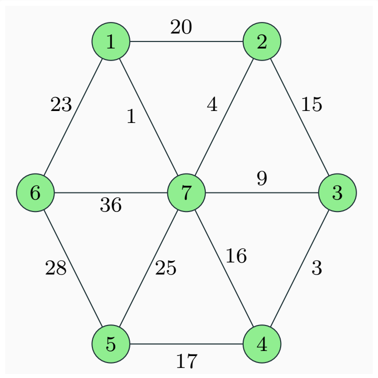
Using Union-Find (disjoint-set) data structure can implement Kruskal's algorithm in $O((m+n)\log m)$ time.

Prim’s Algorithm

Prim's Algorithm: Animation

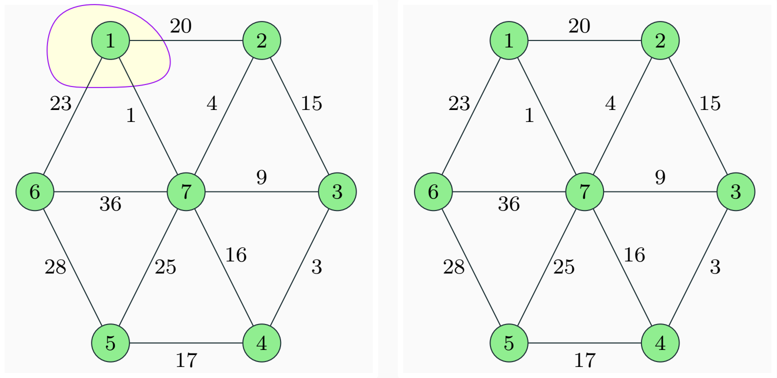
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.





Prim's Algorithm: Animation

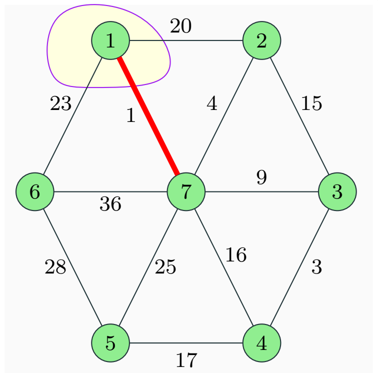
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.

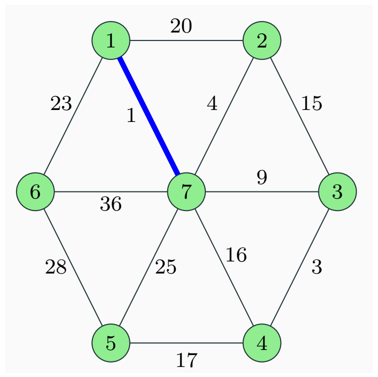


Prim's Algorithm: Animation

T maintained by algorithm will be a tree. Start with a node in T.

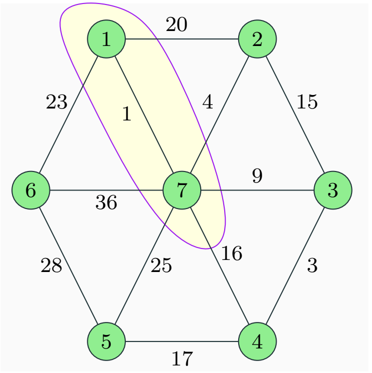
In each iteration, pick edge with least attachment cost to T.

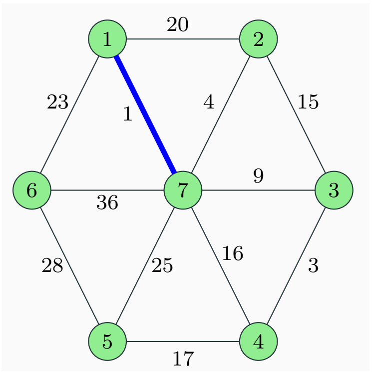




Prim's Algorithm: Animation

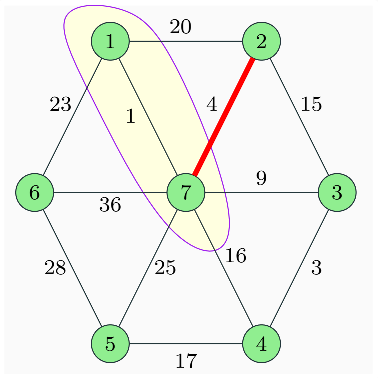
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.

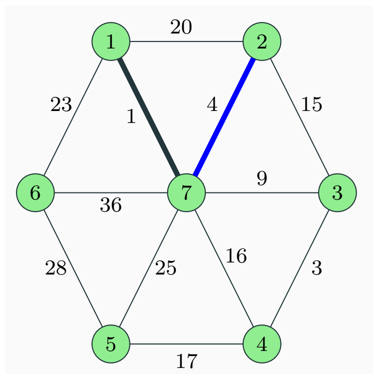




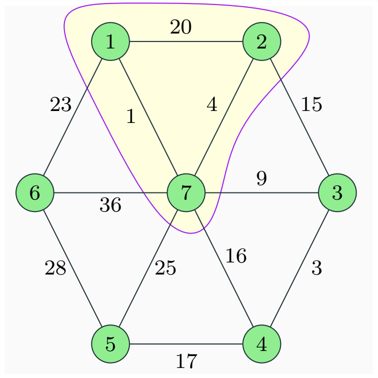
Prim's Algorithm: Animation

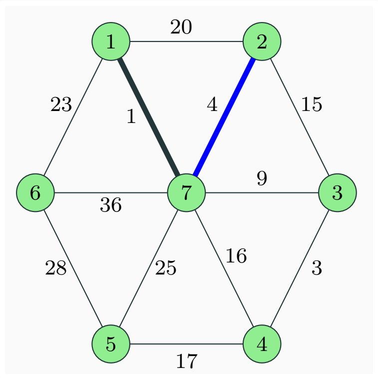
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.





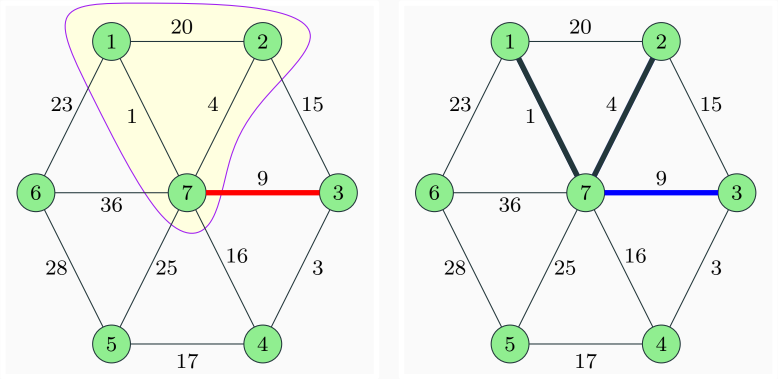
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.



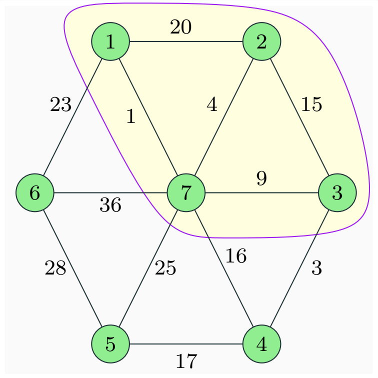


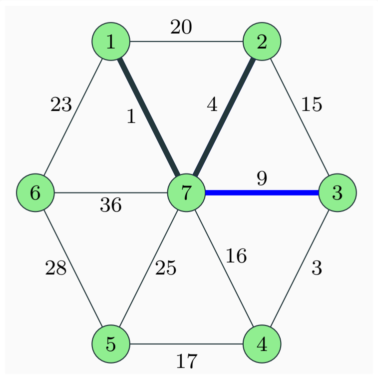
Prim's Algorithm: Animation

T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.



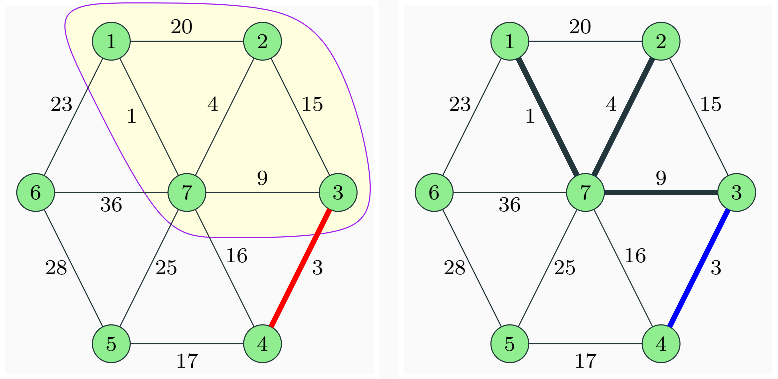
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.





Prim's Algorithm: Animation

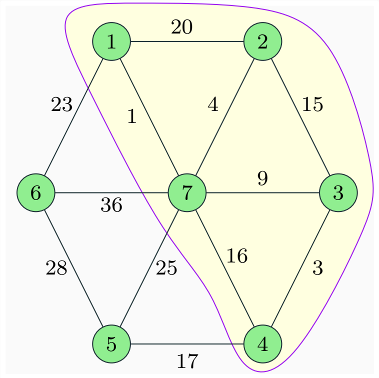
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.

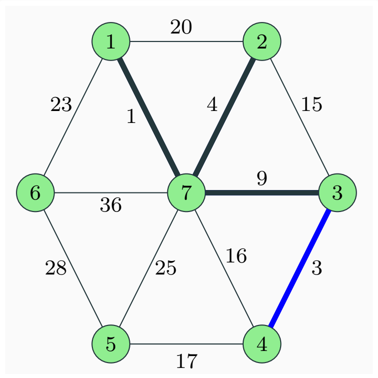


Prim's Algorithm: Animation

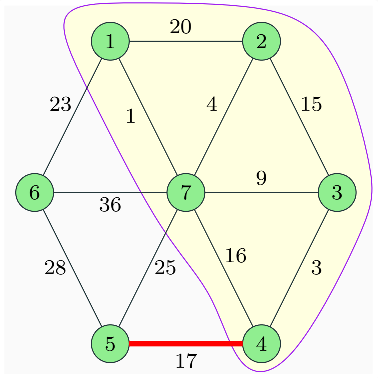
T maintained by algorithm will be a tree. Start with a node in T.

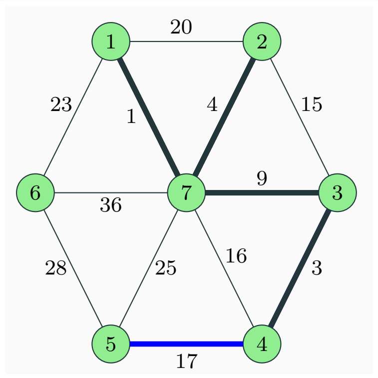
In each iteration, pick edge with least attachment cost to T.



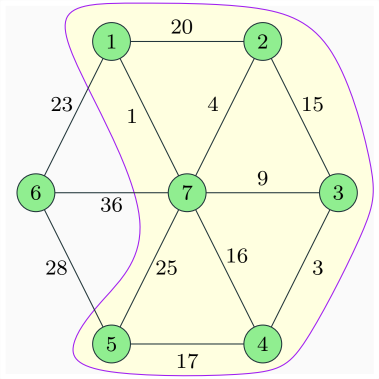


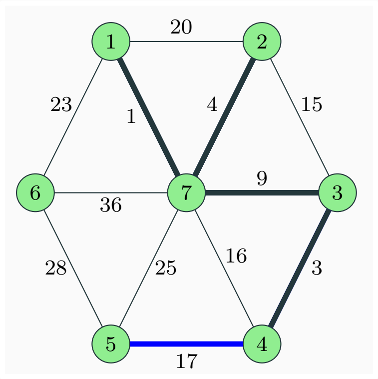
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.





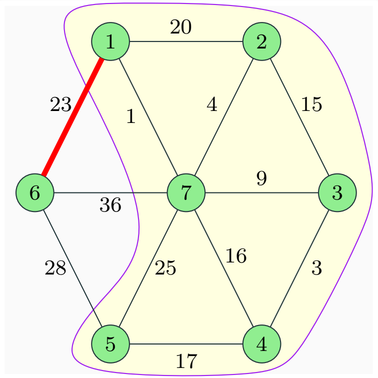
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.

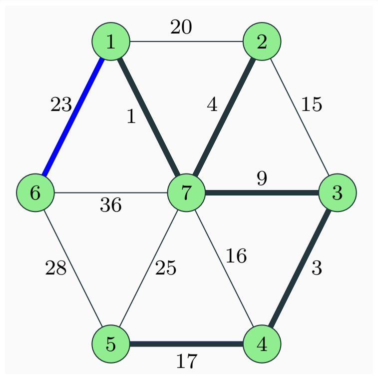




Prim's Algorithm: Animation

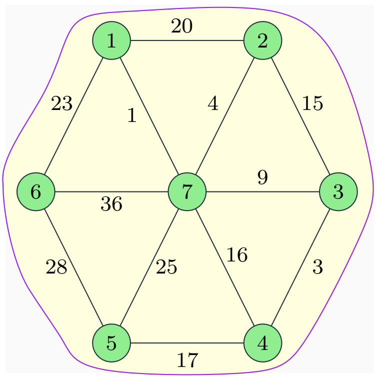
T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.

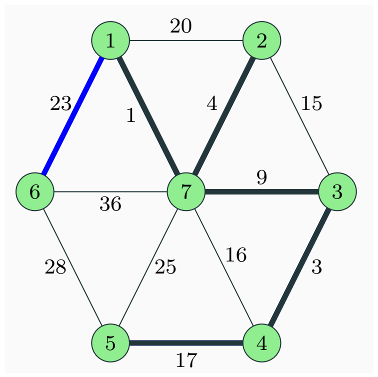




T maintained by algorithm will be a tree. Start with a node in T.

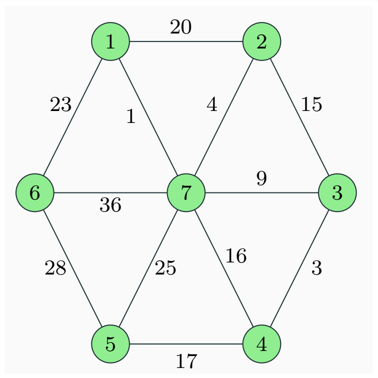
In each iteration, pick edge with least attachment cost to T.

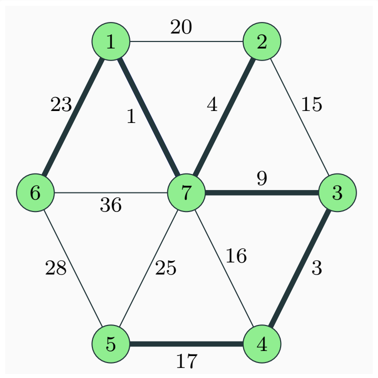




Prim's Algorithm: Animation

T maintained by algorithm will be a tree. Start with a node in T.
In each iteration, pick edge with least attachment cost to T.





Correctness of Prim's Algorithm

Prim's Algorithm
Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

- If e is added to tree, then e is safe and belongs to every MST.
    - 2- Let S be the vertices connected by edges in T when e is added.
    - 3- e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
- Set of edges output is a spanning tree
    - 4- Set of edges output forms a connected graph: by induction, S is connected in each iteration and eventually $S = V$.
    - 5- Only safe edges added and they do not have a cycle $\square$

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

```
    E is the set of all edges in G
    S = {1}
    T is empty (* T will store edges of a MST *)
    while S ≠ V do
        pick e = (v, w) ∈ E such that
            v ∈ S and w ∈ V \ S
            e has minimum cost
        T = T ∪ e
        S = S ∪ w
    return the set T
```

Analysis

Implementing Prim's Algorithm

Prim_ComputeMST

```
    E is the set of all edges in G
    S = {1}
    T is empty (* T will store edges of a MST *)
    while S ≠ V do
        pick e = (v, w) ∈ E such that
            v ∈ S and w ∈ V \ S
            e has minimum cost
        T = T ∪ e
        S = S ∪ w
    return the set T
```

Analysis

- Number of iterations $= O(n)$, where n is number of vertices

Implementing Prim's Algorithm

Prim_ComputeMST

E is the set of all edges in G
$S = \{1\}$
T is empty (* T will store edges of a MST *)
while $S \neq V$ **do**
  pick $e = (v, w) \in E$ such that
    $v \in S$ and $w \in V \setminus S$
    e has minimum cost
  $T = T \cup e$
  $S = S \cup w$
return the set T

Analysis

- Number of iterations $= O(n)$, where n is number of vertices
- Picking e is $O(m)$ where m is the number of edges

Implementing Prim's Algorithm

Prim_ComputeMST

E is the set of all edges in G
$S = \{1\}$
T is empty (* T will store edges of a MST *)
while $S \neq V$ **do**
    pick $e = (v, w) \in E$ such that
        $v \in S$ and $w \in V \setminus S$
        e has minimum cost
    $T = T \cup e$
    $S = S \cup w$
return the set T

Analysis

- Number of iterations $= O(n)$, where n is number of vertices
- Picking e is $O(m)$ where m is the number of edges
- Total time $O(nm)$

Prim's relation to Djikstra

Prim_ComputeMSTv1

```
E is the set of all edges in G
S ← {1}
T is empty
(* T will store edges of a MST *)
for v ∉ S, d(v) = min_{x∈S} c(xv)
for v ∉ S, p(v) = arg min_{x∈S} c(xv)
while S ≠ V do
    pick v ∈ V \ S with minimum d(v)
    e ← vp(v)
    T ← T ∪ {e}
    S ← S ∪ {v}
    update arrays d and p
return the set T
```

Prim's relation to Djikstra

Prim_ComputeMSTv1

```
E is the set of all edges in G
S ← {1}
T is empty
(* T will store edges of a MST *)
for v ∉ S, d(v) = min_{x∈S} c(xv)
for v ∉ S, p(v) = arg min_{x∈S} c(xv)
while S ≠ V do
    pick v ∈ V \ S with minimum d(v)
    e ← vp(v)
    T ← T ∪ {e}
    S ← S ∪ {v}
    update arrays d and p
return the set T
```

Prim_ComputeMSTv2

```
T ← ∅, S ← ∅, s = 1
∀v ∈ V(G) : d(v) ← ∞
∀v ∈ V(G) : p(v) ← Nil
d(s) ← 0
while S ≠ V do
    pick v ∈ V \ S with minimum d(v)
    e ← vp(v)
    T ← T ∪ {e}
    S ← S ∪ {v}
    update arrays d and p
return T
```

Prim's relation to Djikstra

Prim_ComputeMSTv2

$T \leftarrow \emptyset,\ S \leftarrow \emptyset,\ s = 1$
$\forall v \in V(G) : d(v) \leftarrow \infty$
$\forall v \in V(G) : p(v) \leftarrow \texttt{Nil}$
$d(s) \leftarrow 0$
while $S \neq V$ **do**
  pick $v \in V \setminus S$ with minimum $d(v)$
  $e \leftarrow vp(v)$
  $T \leftarrow T \cup \{e\}$
  $S \leftarrow S \cup \{v\}$
  update arrays d and p
return T

Prim_ComputeMSTv3

$T \leftarrow \emptyset,\ S \leftarrow \emptyset,\ s = 1$
$\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
$d(s) \leftarrow 0$
while $S \neq V$ **do**
  $v \leftarrow \arg\min_{u \in V \setminus S} d(u)$
  $T \leftarrow T \cup \{vp(v)\}$
  $S \leftarrow S \cup \{v\}$
  for each u in $\mathrm{Adj}(v)$ **do**
    $d(u) \leftarrow \min \begin{cases} d(u) \\ c(vu) \end{cases}$
    if $d(u) = c(vu)$ **then**
      $p(u) \leftarrow v$
return T

Maintain vertices in $V \setminus S$ in a priority queue with key $d(v)$.

Prim's relation to Djikstra

Prim_ComputeMSTv3

> $T \leftarrow \emptyset,\ S \leftarrow \emptyset,\ s = 1$
> $\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
> $d(s) \leftarrow 0$
> **while** $S \neq V$ **do**
> $v \leftarrow \arg\min_{u \in V \setminus S} d(u)$
> $T \leftarrow T \cup \{vp(v)\}$
> $S \leftarrow S \cup \{v\}$
>   **for** `each` u `in` $\mathrm{Adj}(v)$ **do**
> $d(u) \leftarrow \min \begin{cases} d(u) \\ c(vu) \end{cases}$
>   **if** $d(u) = c(vu)$ **then**
> $p(u) \leftarrow v$
> **return** T

Dijkstra(G, s):

> $\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
> $S \leftarrow \emptyset,\ d(s) \leftarrow 0$
> **while** $S \neq V$ **do**
> $v \leftarrow \arg\min_{u \in V \setminus S} d(u)$
> $S \leftarrow S \cup \{v\}$
>   **for** `each` u `in` $\mathrm{Adj}(v)$ **do**
> $d(u) \leftarrow \min \begin{cases} d(u) \\ d(v) + \ell(v, u) \end{cases}$
>   **if** $d(u) = d(v) + \ell(v, u)$ **then**
> $p(u) \leftarrow v$
>
> **return** $d(V)$

Maintain vertices in $V \setminus S$ in a priority queue with key $d(v)$.

Prim's relation to Djikstra

Prim_ComputeMSTv3

$T \leftarrow \emptyset,\ S \leftarrow \emptyset,\ s = 1$
$\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
$d(s) \leftarrow 0$
while $S \neq V$ **do**
  $v \leftarrow \arg\min_{u \in V \setminus S} d(u)$
  $T \leftarrow T \cup \{vp(v)\}$
  $S \leftarrow S \cup \{v\}$
  for each u in $\mathrm{Adj}(v)$ **do**
    $d(u) \leftarrow \min \begin{cases} d(u) \\ c(vu) \end{cases}$
    if $d(u) = c(vu)$ **then**
      $p(u) \leftarrow v$
return T

Dijkstra(G, s):

$\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
$S \leftarrow \emptyset,\ d(s) \leftarrow 0$
while $S \neq V$ **do**
  $v \leftarrow \arg\min_{u \in V \setminus S} d(u)$
  $S \leftarrow S \cup \{v\}$
  for each u in $\mathrm{Adj}(v)$ **do**
    $d(u) \leftarrow \min \begin{cases} d(u) \\ d(v) + \ell(v, u) \end{cases}$
    if $d(u) = d(v) + \ell(v, u)$ **then**
      $p(u) \leftarrow v$

return $d(V)$

Maintain vertices in $V \setminus S$ in a priority queue with key $d(v)$.**Prim's algorithm is essentially Dijkstra's algorithm!**

Implementing Prim's algorithm with priority queues

Priority Queues

Data structure to store a set S of n elements where each element $v \in S$ has an associated real/integer key $k(v)$ such that the following operations

- **makeQ**: create an empty queue
- **findMin**: find the minimum key in S
- **extractMin**: Remove $v \in S$ with smallest key and return it
- **add**$(v, k(v))$: Add new element v with key $k(v)$ to S
- **Delete**(v): Remove element v from S
- **decreaseKey** $(v, k'(v))$: decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
- **meld**: merge two separate priority queues into one

Prim's using priority queues

Prim_ComputeMSTv3

$T \leftarrow \emptyset,\ S \leftarrow \emptyset,\ s \leftarrow 1$
$\forall v \in V(G) : d(v) \leftarrow \infty, p(v) \leftarrow \texttt{Nil}$
$d(s) \leftarrow 0$
while $S \neq V$ **do**
  $v = \arg\min_{u \in V \setminus S} d(u)$
  $T = T \cup \{vp(v)\}$
  $S = S \cup \{v\}$
  for `each` u `in` $Adj(v)$ **do**
    $d(u) \leftarrow \min \begin{cases} d(u) \\ c(vu) \end{cases}$
    if $d(u) = c(vu)$ **then**
      $p(u) \leftarrow v$
return T

Maintain vertices in $V \setminus S$ in a priority queue with key $d(v)$

- 2- Requires $O(n)$ **extractMin** operations
- 3- Requires $O(m)$ **decreaseKey** operations

Running time of Prim's Algorithm

$O(n)$ **extractMin** operations and $O(m)$ **decreaseKey** operations

- Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m+n)\log n)$
- Using Fibonacci Heaps, $O(\log n)$ for **extractMin** and $O(1)$ (amortized) for **decreaseKey**. Total: $O(n \log n + m)$.

Running time of Prim's Algorithm

$O(n)$ **extractMin** operations and $O(m)$ **decreaseKey** operations

- Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m+n)\log n)$
- Using Fibonacci Heaps, $O(\log n)$ for **extractMin** and $O(1)$ (amortized) for **decreaseKey**. Total: $O(n \log n + m)$.
- Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?

Running time of Prim's Algorithm

$O(n)$ **extractMin** operations and $O(m)$ **decreaseKey** operations

- Using standard Heaps, **extractMin** and **decreaseKey** take $O(\log n)$ time. Total: $O((m+n)\log n)$
- Using Fibonacci Heaps, $O(\log n)$ for **extractMin** and $O(1)$ (amortized) for **decreaseKey**. Total: $O(n \log n + m)$.
- Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?
- Prim's algorithm = Dijkstra where length of a path π is the weight of the heaviest edge in π. (Bottleneck shortest path.)

MST algorithm for negative weights, and non-distinct costs

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

- $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
- Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or $(c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A)$.
- Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

- $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
- Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or $(c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A)$.
- Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

- $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
- Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or ($c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A$).
- Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's and Kruskal's Algorithms are optimal with respect to lexicographic ordering.

Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
- Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
- Can compute maximum weight spanning tree by negating edge costs and then computing an MST.

Edge Costs: Positive and Negative

- Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
- Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
- Can compute maximum weight spanning tree by negating edge costs and then computing an MST.
  Question: Why does this not work for shortest paths?

MST: An epilogue

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $O(n \log n + m)$.
If m is $O(n)$ then running time is $\Omega(n \log n)$.

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $O(n \log n + m)$.

If m is $O(n)$ then running time is $\Omega(n \log n)$.

Question

Is there a linear time ($O(m + n)$ time) algorithm for MST?

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $O(n \log n + m)$.
If m is $O(n)$ then running time is $\Omega(n \log n)$.

Question
Is there a linear time ($O(m + n)$ time) algorithm for MST?

- $O(m \log^* m)$ time **[Fredman and Tarjan 1987]**
- $O(m + n)$ time using bit operations in RAM model **[Fredman, Willard 1994]**
- $O(m + n)$ expected time (randomized algorithm) **[Karger, Klein, Tarjan 1995]**
- $O((n + m)\alpha(m, n))$ time **[Chazelle 2000]**
- Still open: Is there an $O(n + m)$ time deterministic algorithm in the comparison model?

Fin