Algorithms for Minimum Spanning Trees

Lecture 20
Thursday, November 14, 2024

LaTeXed: November 14, 2024 10:18

20.1
Minimum Spanning Tree

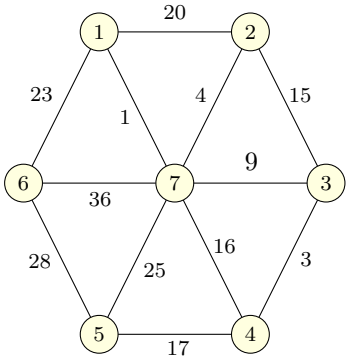
20.1.1
The Problem

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

1. T is the minimum spanning tree (MST) of G

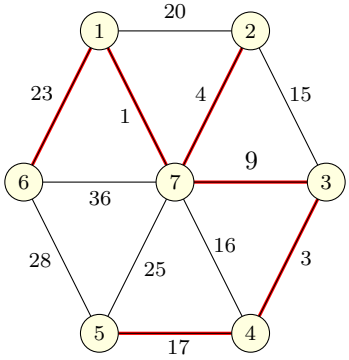


Minimum Spanning Tree

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

1. T is the minimum spanning tree (MST) of G



Applications

1. Network Design
   1.1 Designing networks with minimum cost but maximum connectivity
2. Approximation algorithms
   2.1 Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
3. 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.

20.1.2
Some graph theory

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.

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.

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.

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.

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.

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 20.1.

$T = (V, E_T)$: 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.

20.2
Safe and unsafe edges

Assumption

And for now ...

Assumption 20.1.

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

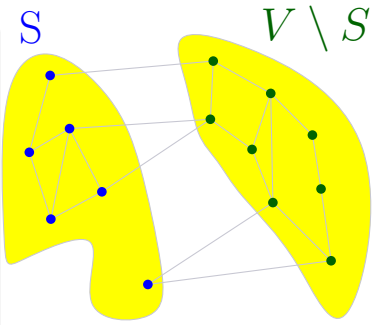
Cuts

Definition 20.2.

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.



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

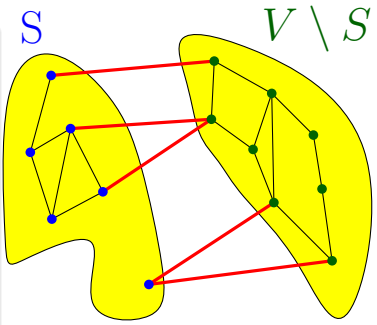
Cuts

Definition 20.2.

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 20.3.

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 20.4.

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.

Safe and Unsafe Edges

Definition 20.3.

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 20.4.

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 20.5.

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 $e \notin \mathbf{E}(G_{<w(e)})$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. 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.

Every edge is either safe or unsafe

Proposition 20.5.

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 $e \notin \mathbf{E}\big(G_{<w(e)}\big)$.

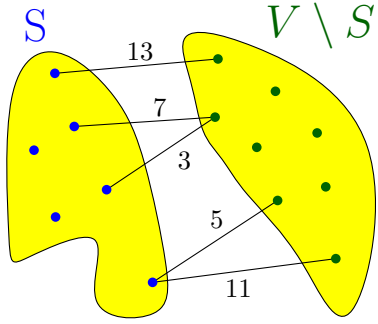
1. If x, y in some connected component of $G_{<w(e)}$, then $G_{<w(e)} + e$ contains a cycle where e is most expensive.
   $\implies$ e is unsafe.
2. If x and y are in diff connected component of $G_{<w(e)}$,
   Let S the vertices of connected component of $G_{<w(e)}$ containing x.
   The edge e is cheapest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

□

Safe edge

Example...

Every cut identifies one safe edge...



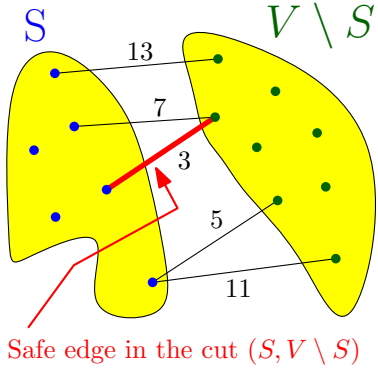
...the cheapest edge in the cut.

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

Safe edge

Example...

Every cut identifies one safe edge...



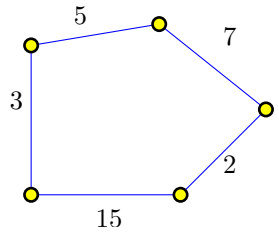
...the cheapest edge in the cut.

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

Unsafe edge

Example...

Every cycle identifies one **unsafe** edge...

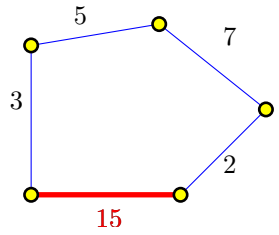


...the most expensive edge in the cycle.

Unsafe edge

Example...

Every cycle identifies one **unsafe** edge...



...the most expensive edge in the cycle.

Example

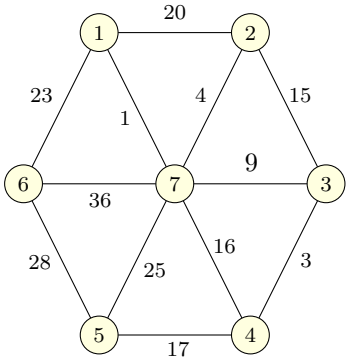


Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

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

Example

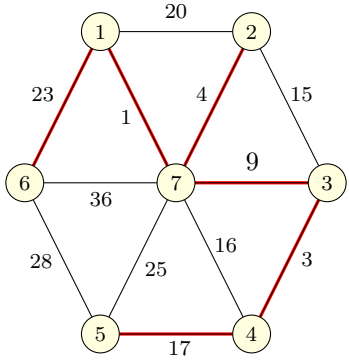


Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

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

Example

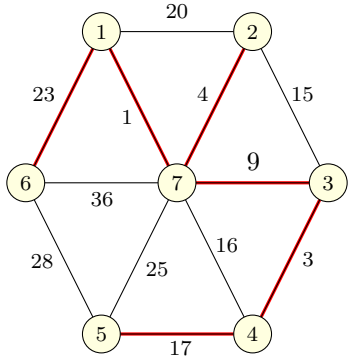


Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

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

Some key observations

Proofs later

Lemma 20.6.

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

Lemma 20.7.

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

20.3
The Algorithms for computing MST

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?

KA PA RD

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.

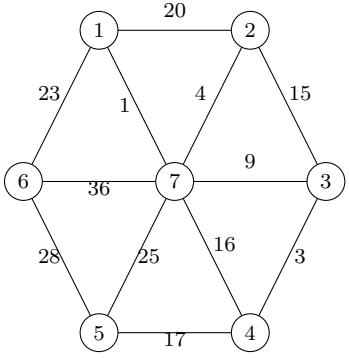


Figure: Graph G

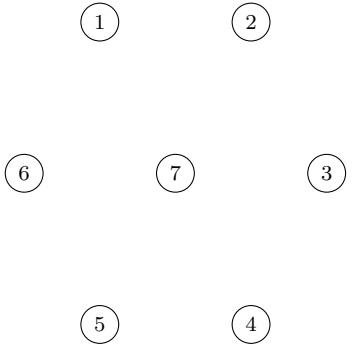


Figure: MST of G

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.

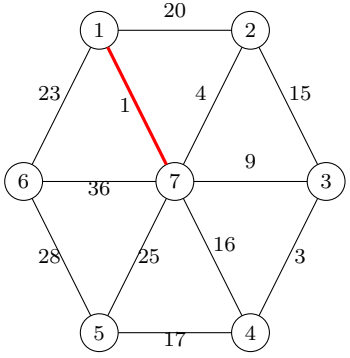


Figure: Graph G

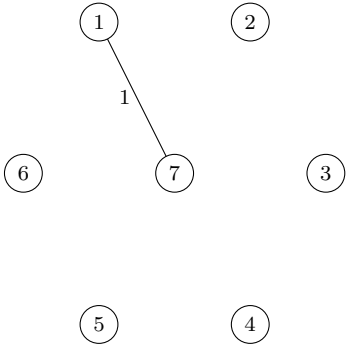


Figure: MST of G

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.

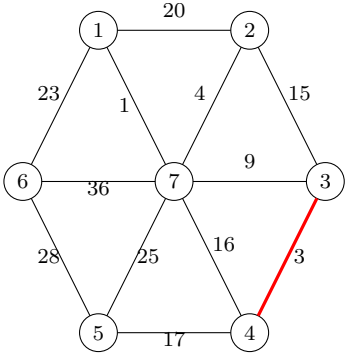


Figure: Graph G

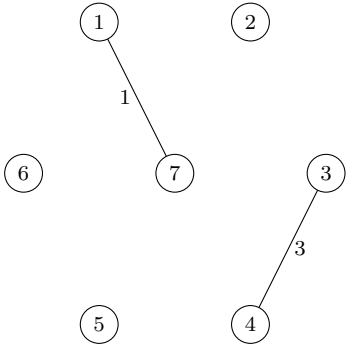


Figure: MST of G

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.

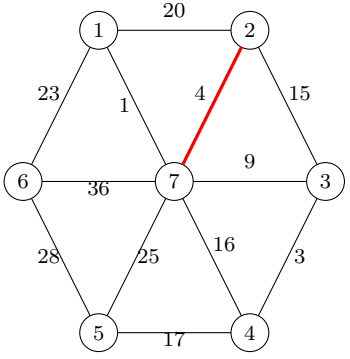


Figure: Graph G

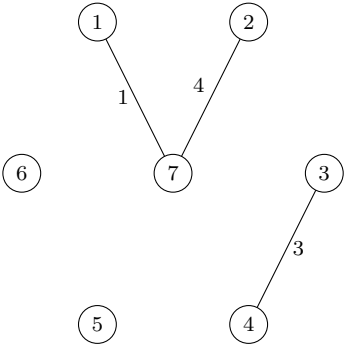


Figure: MST of G

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.

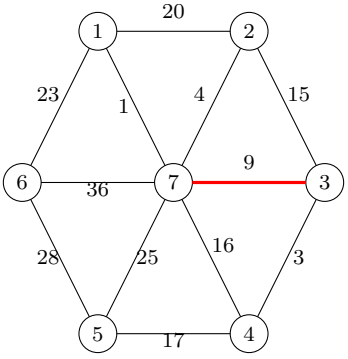


Figure: Graph G

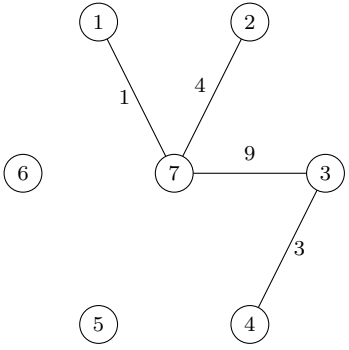


Figure: MST of G

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.

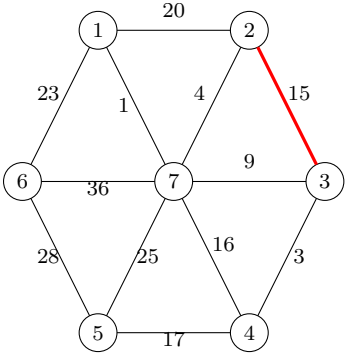


Figure: Graph G

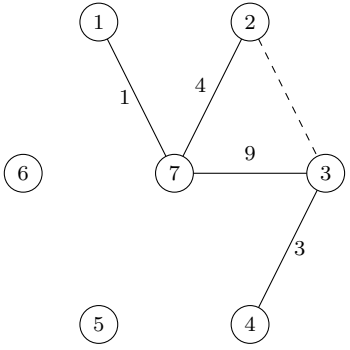


Figure: MST of G

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.

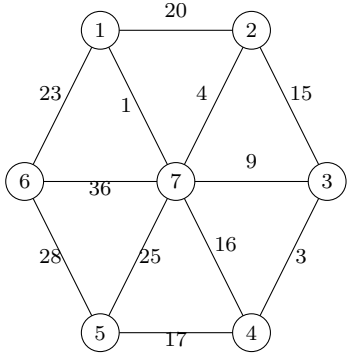


Figure: Graph G

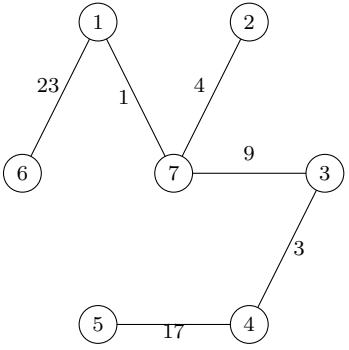
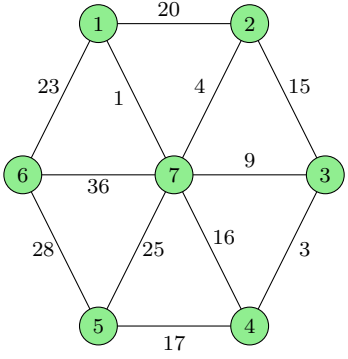
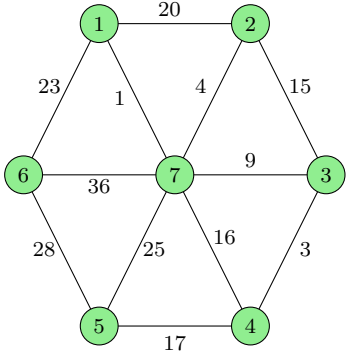


Figure: MST of G

Prim's Algorithm: Animation

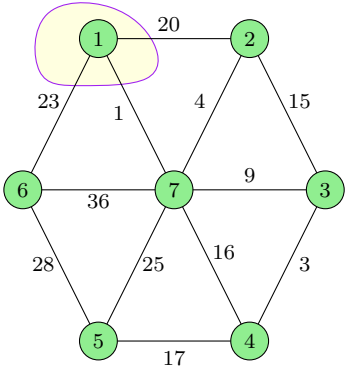
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

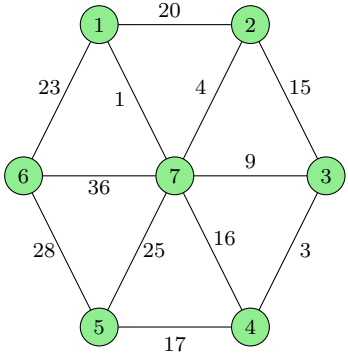




Prim's Algorithm: Animation

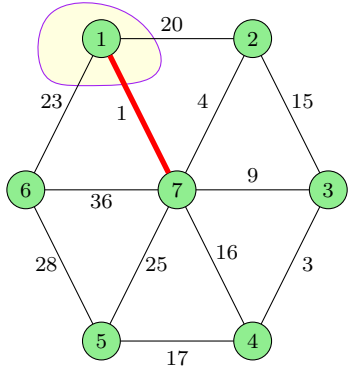
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

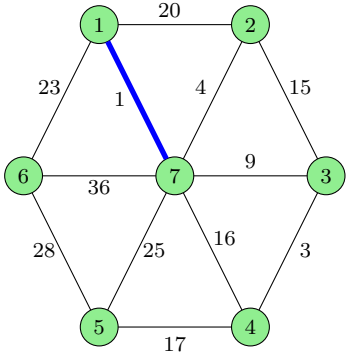




Prim's Algorithm: Animation

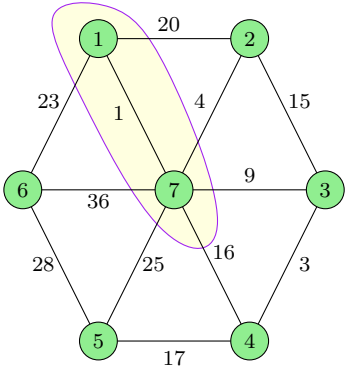
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

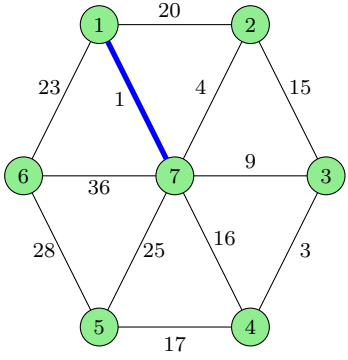




Prim's Algorithm: Animation

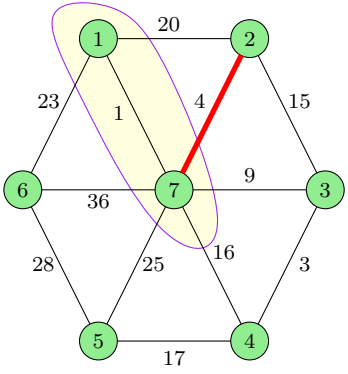
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

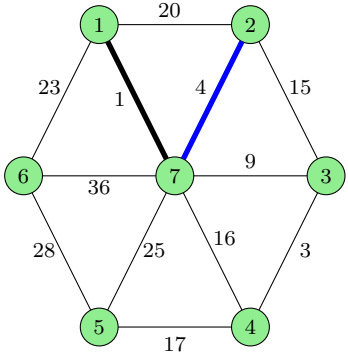




Prim's Algorithm: Animation

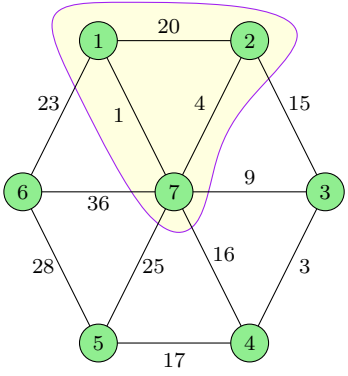
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

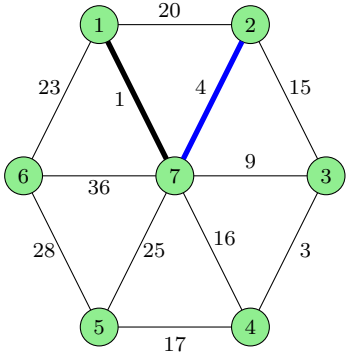




Prim's Algorithm: Animation

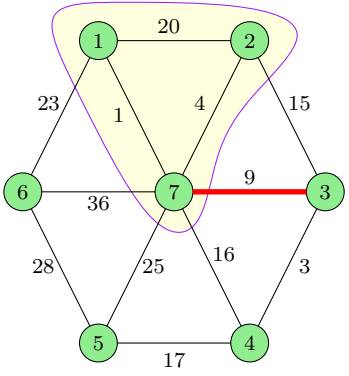
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

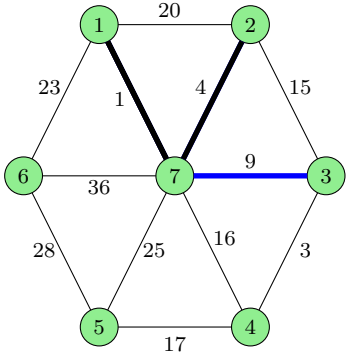




Prim's Algorithm: Animation

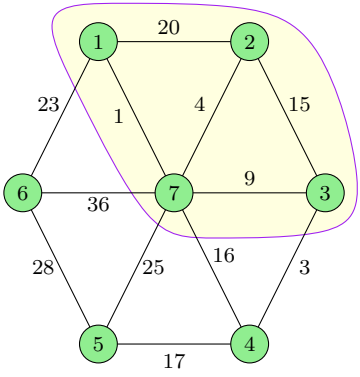
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

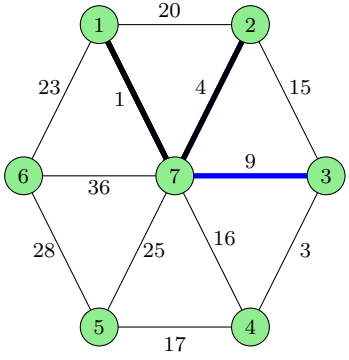




Prim's Algorithm: Animation

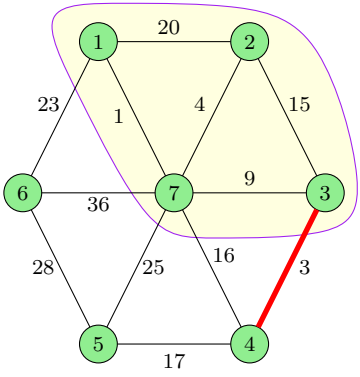
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

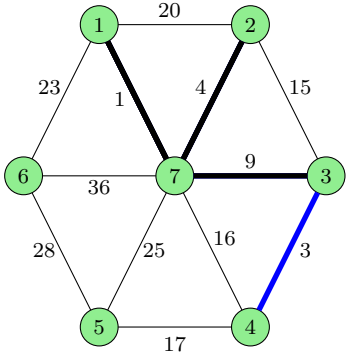




Prim's Algorithm: Animation

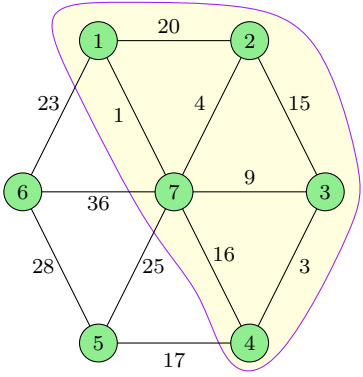
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

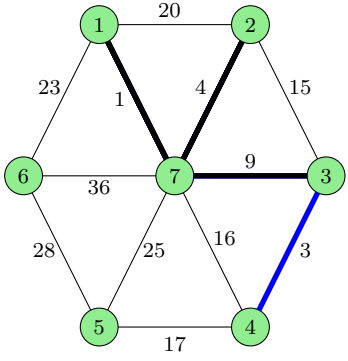




Prim's Algorithm: Animation

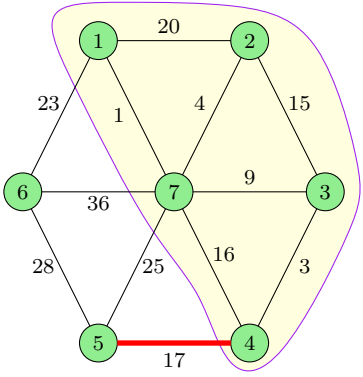
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

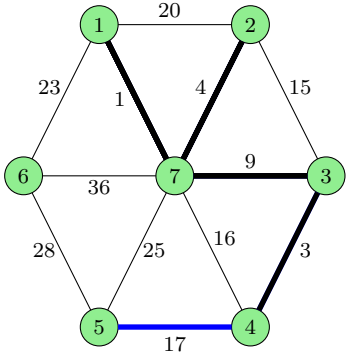




Prim's Algorithm: Animation

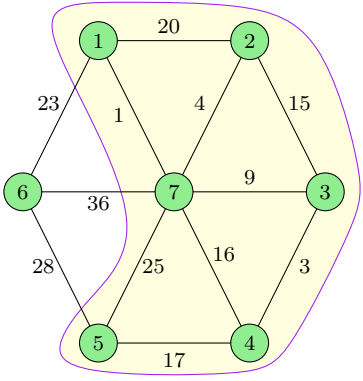
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

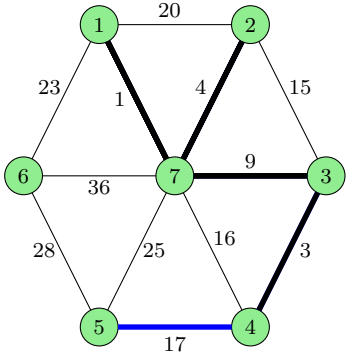




Prim's Algorithm: Animation

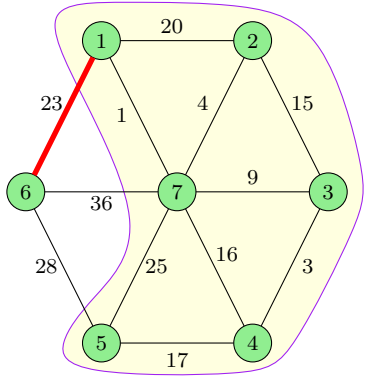
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

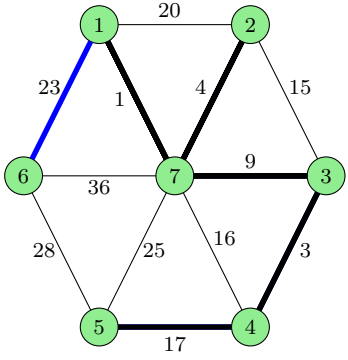




Prim's Algorithm: Animation

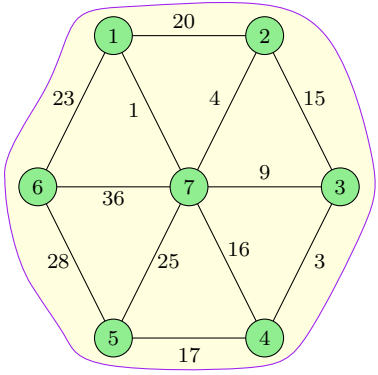
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

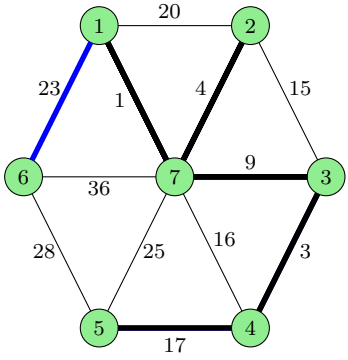




Prim's Algorithm: Animation

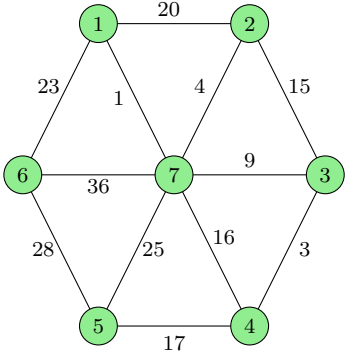
T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

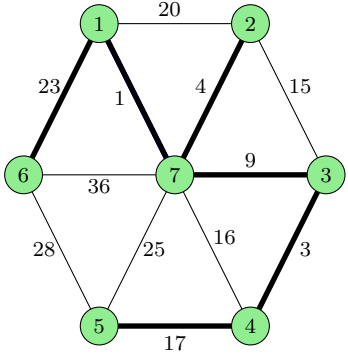




Prim's Algorithm: Animation

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.





Reverse Delete Algorithm

```
Initially Z is the set of all edges in G
T ⇐ Z (* T will store edges of a MST *)
while Z is not empty do
    choose e ∈ Z of largest cost
    remove e from Z
    if removing e does not disconnect T then
        remove e from T
return the set T
```

Returns a minimum spanning tree.

Back

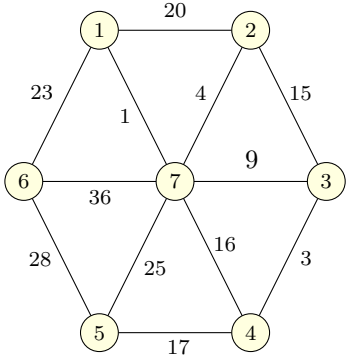
Borůvka's Algorithm

Simplest to implement. See notes.

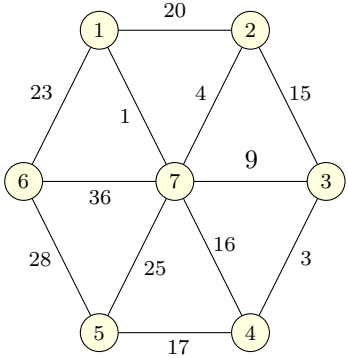
Assume G is a connected graph.

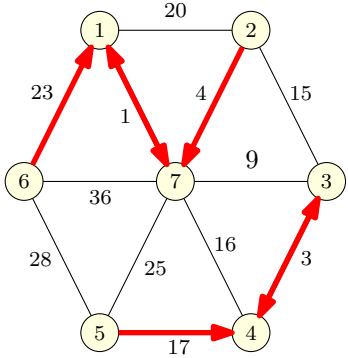
```
T is ∅ (* T will store edges of a MST *)
while T is not spanning do
    X ← ∅
    for each connected component S of T do
        add to X the cheapest edge between S and V \ S
    Add edges in X to T
return the set T
```

Borůvka's Algorithm

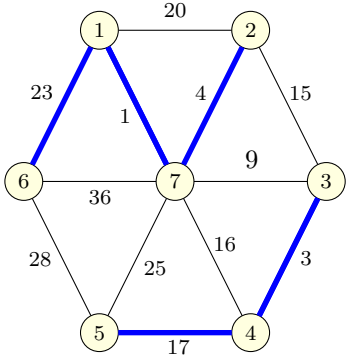


Borůvka's Algorithm

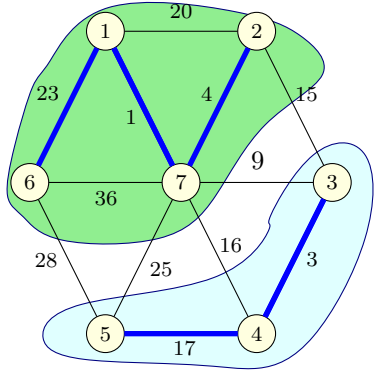




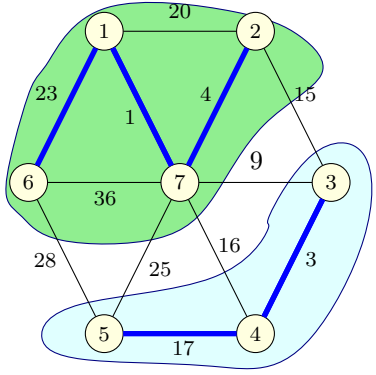
Borůvka's Algorithm

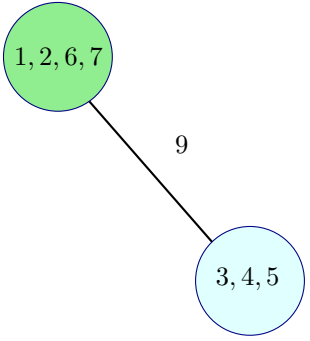


Borůvka's Algorithm

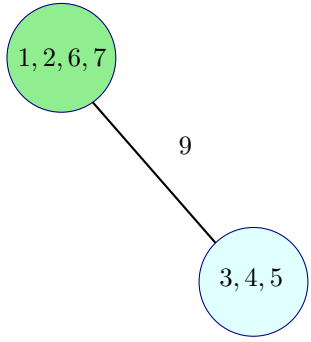


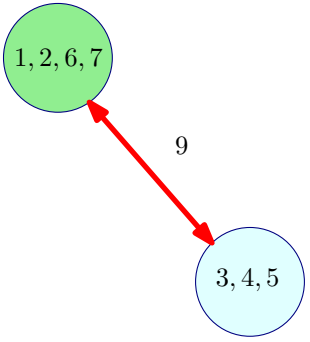
Borůvka's Algorithm



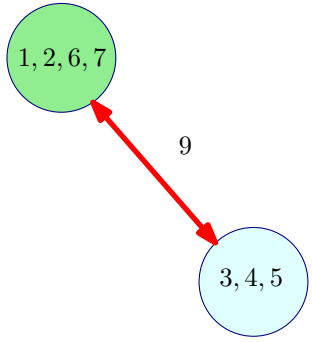


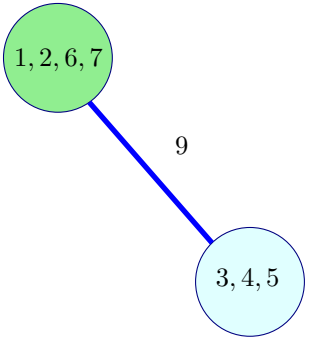
Borůvka's Algorithm



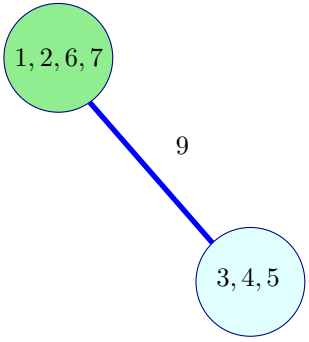


Borůvka's Algorithm

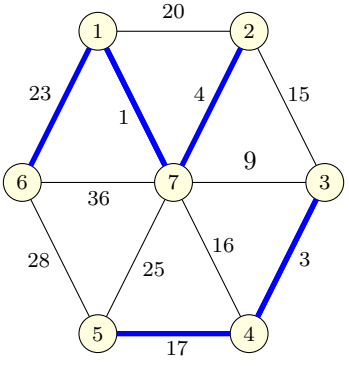




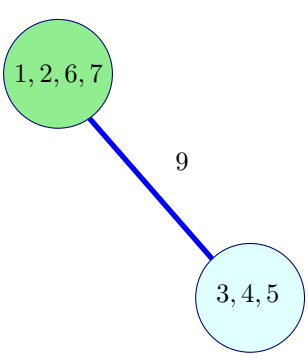
Borůvka's Algorithm



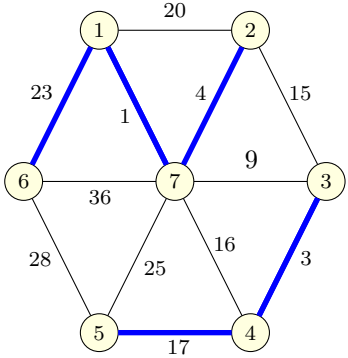
+

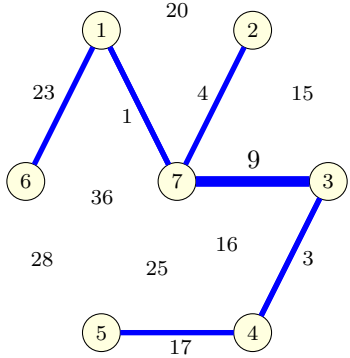


Borůvka's Algorithm

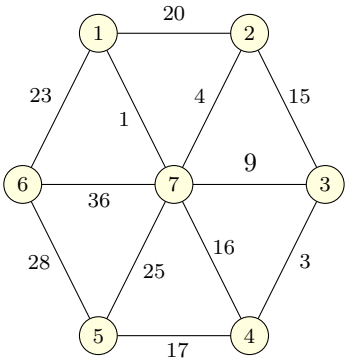


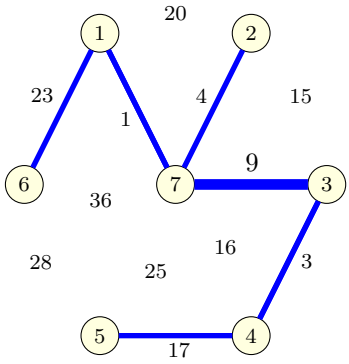
+





Borůvka's Algorithm





20.4
Correctness of the MST algorithms

20.4.1
Safe edges must be in the MST

Correctness of MST Algorithms

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

Key Observation: Cut Property

Lemma 20.1.

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

Proof.

1. Suppose (for contradiction) e is not in MST T.
2. Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
3. Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
4. 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!!

Key Observation: Cut Property

Lemma 20.1.

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

Proof.

1. Suppose (for contradiction) e is not in MST T.
2. Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
3. Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
4. 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!!

Key Observation: Cut Property

Lemma 20.1.

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

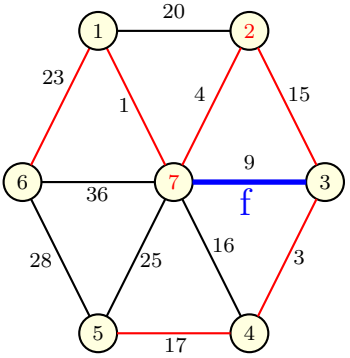
Proof.

1. Suppose (for contradiction) e is not in MST T.
2. Since e is safe there is an $S \subset V$ such that e is the unique min cost edge crossing S.
3. Since T is connected, there must be some edge f with one end in S and the other in $V \setminus S$
4. 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!!

□

Error in Proof: Example

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

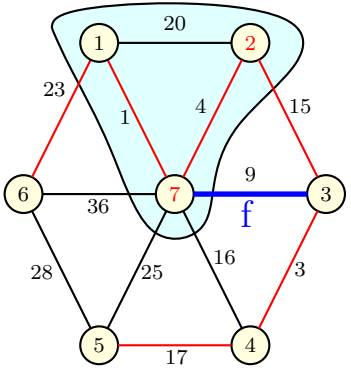


(A)

(A) Consider adding the edge $\boldsymbol{f}$.

Error in Proof: Example

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



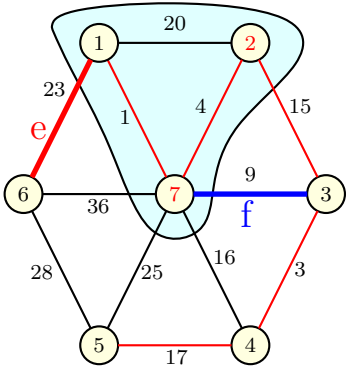
(B)

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



(C)

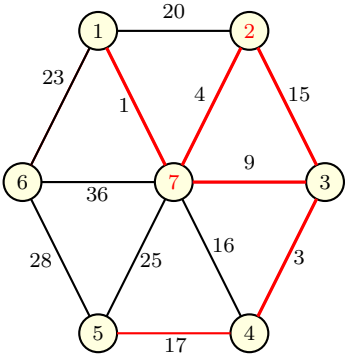
(A) Consider adding the edge f.

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



(D)

(A) Consider adding the edge f.

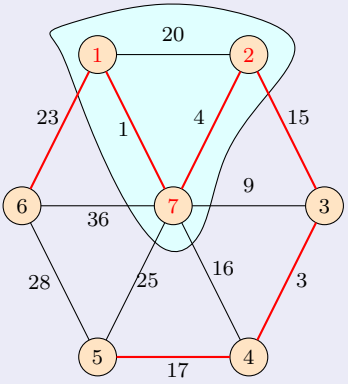
(B) It is safe because it is the cheapest edge in the cut.

(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

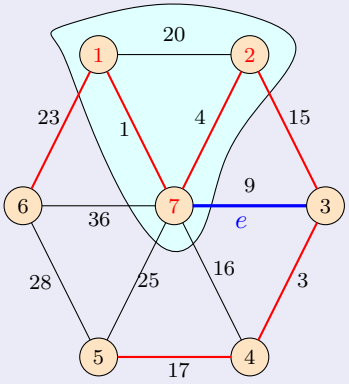
Proof.



1. 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
3. 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 $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

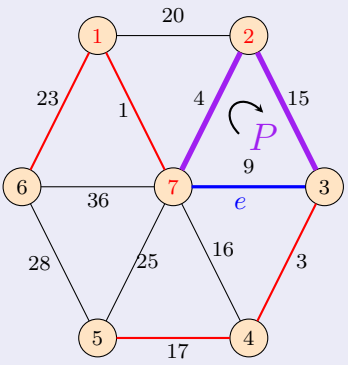
Proof.



1. 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
3. 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 $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

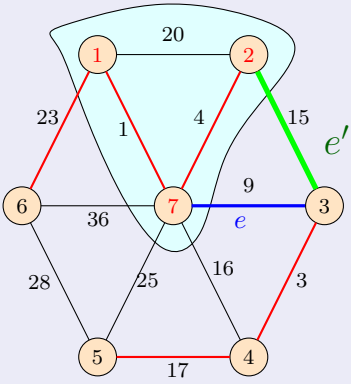
Proof.



1. 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
3. 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 $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

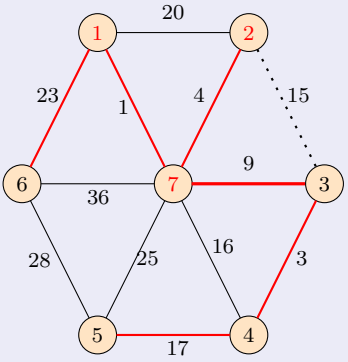
Proof.



1. 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
3. 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 $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

Proof.



1. 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
3. 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 $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property (contd)

Observation 20.2.

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

Proof.

T' is connected.

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

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

□

Proof of Cut Property (contd)

Observation 20.2.

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

Proof.

T' is connected.

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

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

□

Proof of Cut Property (contd)

Observation 20.2.

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

Proof.

T' is connected.

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

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

□

20.4.2
The safe edges form the MST

Safe Edges form a connected graph

Lemma 20.3.

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

Proof.

1. Suppose not. Let S be a connected component in the graph induced by the safe edges.
2. 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.

□

Safe Edges do not contain a cycle

Lemma 20.4.

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

Proof.

Proposition 20.5 : proved every edge in graph is either safe or unsafe. If $\exists$ cycle, then by definition the most expensive edge in the cycle is unsafe. Contradiction. □

Safe Edges do not contain a cycle

Lemma 20.4.

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

Proof.

Assume false, and let π a cycle made of safe edges.

e: Most expensive edge in the cycle π.

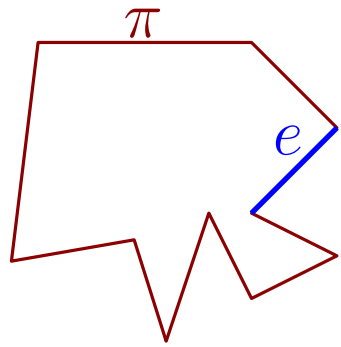
$\mathcal{C} = (S, V \setminus S)$: Cut that e is safe for.

π must have at least two edges in $\mathcal{C}$.

f: cheapest edge in $\pi \cap \mathcal{C}$.

e is not cheapest edge in $\mathcal{C}$.

A contradiction. □



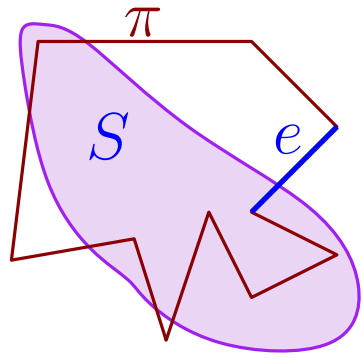
Safe Edges do not contain a cycle

Lemma 20.4.

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

Proof.

Assume false, and let π a cycle made of safe edges.
e: Most expensive edge in the cycle π.
$\mathcal{C} = (S, V \setminus S)$: Cut that e is safe for.
π must have at least two edges in $\mathcal{C}$.
f: cheapest edge in $\pi \cap \mathcal{C}$.
e is not cheapest edge in $\mathcal{C}$.
A contradiction. □



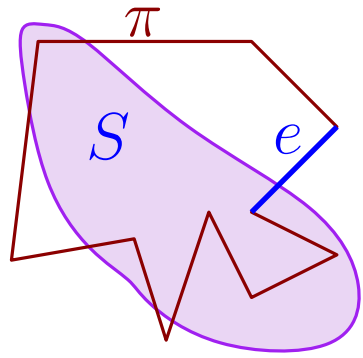
Safe Edges do not contain a cycle

Lemma 20.4.

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

Proof.

Assume false, and let π a cycle made of safe edges.
e: Most expensive edge in the cycle π.
$\mathcal{C} = (S, V \setminus S)$: Cut that e is safe for.
π must have at least two edges in $\mathcal{C}$.
f: cheapest edge in $\pi \cap \mathcal{C}$.
e is not cheapest edge in $\mathcal{C}$.
A contradiction. □



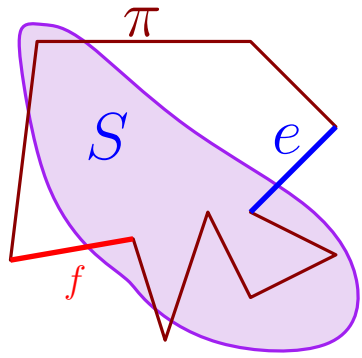
Safe Edges do not contain a cycle

Lemma 20.4.

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

Proof.

Assume false, and let π a cycle made of safe edges.
e: Most expensive edge in the cycle π.
$\mathcal{C} = (S, V \setminus S)$: Cut that e is safe for.
π must have at least two edges in $\mathcal{C}$.
f: cheapest edge in $\pi \cap \mathcal{C}$.
e is not cheapest edge in $\mathcal{C}$.
A contradiction. □



Safe Edges form an MST

Corollary 20.5.

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.

Safe Edges form an MST

Corollary 20.5.

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.

20.4.3
The unsafe edges are NOT in the MST

Cycle Property

Lemma 20.6.

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

Cycle Property

Lemma 20.6.

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.

20.4.4
Correctness of the various MST algorithms

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.

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

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.

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

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.

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

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.

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

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.

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

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.

1. If $e = (u, v)$ is added to tree, then e is safe
   - 1.1 When algorithm adds e let S and S' be the connected components containing u and v respectively
   - 1.2 e is the lowest cost edge crossing S (and also S').
   - 1.3 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
2. Set of edges output is a spanning tree : exercise

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.

1. If $e = (u, v)$ is added to tree, then e is safe
   - 1.1 When algorithm adds e let S and S' be the connected components containing u and v respectively
   - 1.2 e is the lowest cost edge crossing S (and also S').
   - 1.3 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
2. Set of edges output is a spanning tree : exercise

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.

1. If $e = (u, v)$ is added to tree, then e is safe
   - 1.1 When algorithm adds e let S and S' be the connected components containing u and v respectively
   - 1.2 e is the lowest cost edge crossing S (and also S').
   - 1.3 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
2. Set of edges output is a spanning tree : exercise

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.

1. If $e = (u, v)$ is added to tree, then e is safe
   - 1.1 When algorithm adds e let S and S' be the connected components containing u and v respectively
   - 1.2 e is the lowest cost edge crossing S (and also S').
   - 1.3 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
2. Set of edges output is a spanning tree : exercise

Correctness of Borůvka's Algorithm

Proof of correctness.

Argue that only safe edges are added. □

Correctness of Reverse Delete Algorithm

Reverse Delete Algorithm

Consider edges in decreasing cost and remove an edge if it does not disconnect the graph

Proof of correctness.

Argue that only unsafe edges are removed. □

20.5
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

Formal argument: Order edges lexicographically to break ties

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or $(c(e_i) = c(e_j)$ and $i < j)$
2. 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 $B \setminus A)$.
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

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

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

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. 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 $B \setminus A$).
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

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

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

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. 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 $B \setminus A$).
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

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

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

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. 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 $B \setminus A$).
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

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

Edge Costs: Positive and Negative

1. Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
2. 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?
3. Can compute maximum weight spanning tree by negating edge costs and then computing an MST.

   Question: Why does this not work for shortest paths?

Edge Costs: Positive and Negative

1. Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
2. 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?
3. Can compute maximum weight spanning tree by negating edge costs and then computing an MST.
   Question: Why does this not work for shortest paths?

20.6
Data Structures for MST: Priority Queues and Union-Find

20.6.1
Implementing 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? 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.

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.

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.

20.6.2
Implementing Prim's Algorithm

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

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

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

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

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

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

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

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

Making Prim's Algorithm Great Again

By making it look like Dijkstra's algorithm

// $c(e)$: Cost of the edge e

Prim_ComputeMSTv1

E is the set of all edges in G
$S \leftarrow \{1\}$
T is empty
(* T will store edges of a MST *)
for $v \notin S$, $d(v) = \min_{x \in S} c(xv)$
for $v \notin S$, $p(v) = \arg\min_{x \in S} c(xv)$
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 the set T

Making Prim's Algorithm Great Again

By making it look like Dijkstra's algorithm

// $c(e)$: Cost of the edge e

Prim_ComputeMSTv1

E is the set of all edges in G
$S \leftarrow \{1\}$
T is empty
(* T will store edges of a MST *)
for $v \notin S$, $d(v) = \min_{x \in S} c(xv)$
for $v \notin S$, $p(v) = \arg\min_{x \in S} c(xv)$
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 the set T

Prim_ComputeMSTv2

$T \leftarrow \emptyset$, $S \leftarrow \emptyset$, $s = 1$
$\forall v \in V(G) : d(v) \leftarrow \infty$
$\forall v \in V(G) : p(v) \leftarrow$ 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

Making Prim's Algorithm Great Again

By making it look like Dijkstra's algorithm

// $c(e)$: Cost of the edge e

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

Making Prim's Algorithm Great Again

By making it look like Dijkstra's algorithm

// $c(e)$: Cost of the edge e

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 $d(V)$

Making Prim's Algorithm Great Again

By making it look like Dijkstra's algorithm

// $c(e)$: Cost of the edge e

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 $d(V)$

Prim's algorithm is essentially Dijkstra's algorithm!

20.6.3
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

1. **makeQ**: create an empty queue
2. **findMin**: find the minimum key in S
3. **extractMin**: Remove $v \in S$ with smallest key and return it
4. **add**(v, $k(v)$): Add new element v with key $k(v)$ to S
5. **Delete**(v): Remove element v from S
6. **decreaseKey** (v, $k'(v)$): decrease key of v from $k(v)$ (current key) to $k'(v)$ (new key). Assumption: $k'(v) \leq k(v)$
7. **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` $\mathbf{Adj}(v)$ **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 $d(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

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 $\mathbf{Adj}(v)$ **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 $d(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

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 $\mathbf{Adj}(v)$ **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 $d(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

Running time of Prim's Algorithm

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

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

Running time of Prim's Algorithm

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

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

Running time of Prim's Algorithm

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

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

20.6.4
Union-find data-structure

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Requirements from the union-find data-structure

1. Maintain a collection of sets.
2. **makeSet**(x) - creates a set that contains the single element x.
3. **find**(x) - returns the set that contains x.
4. **union**(A, B) - returns set = union of A and B. That is $A \cup B$.
   ... merges the two sets A and B and return the merged set.

Pseudo-code of Union-Find using reverse tree...

```
makeSet(x)
  p̄(x) ← x
  rank(x) ← 0
```

```
find(x)
  if x ≠ p̄(x) then
    p̄(x) ← find(p̄(x))
  return p̄(x)
```

```
union(x, y )
  A ← find(x)
  B ← find(y)
  if rank(A) > rank(B) then
    p̄(B) ← A
  else
    p̄(A) ← B
    if rank(A) = rank(B) then
      rank(B) ← rank(B) + 1
```

Union-find

Theorem 20.1.

For a sequence of m *operations over* n *elements, the overall running time of the* **UnionFind** *data-structure is* $O((n+m)\log^* n)$.

(Better analysis is known – to involved to explain here.)

20.6.5
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
```

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

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
```

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

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
```

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

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
```

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

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
```

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

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
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
3. 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
```

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

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

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 data structure can implement Kruskal's algorithm in $O((m + n) \log m)$ time.

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 data structure can implement Kruskal's algorithm in $O((m + n) \log m)$ time.

20.7
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 $\Omega(n \log n)$.

Question

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

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

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?

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

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?

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