Algorithms & Models of Computation
CS/ECE 374, Fall 2017

Algorithms for Minimum Spanning Trees
Lecture 20
April 6
Part I

Algorithms for Minimum Spanning Tree
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

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

1. Network Design
   - Designing networks with minimum cost but maximum connectivity

2. Approximation algorithms
   - Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.

3. Cluster Analysis
Some basic properties of Spanning Trees

- A graph $G$ is connected iff it has a spanning tree.
- Every spanning tree of a graph on $n$ nodes has $n - 1$ edges.
- Let $T = (V, E_T)$ be 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 $G$. 
Some basic properties of Spanning Trees

- A graph $G$ is connected iff it has a spanning tree.
- Every spanning tree of a graph on $n$ nodes has $n - 1$ edges.
- Let $T = (V, E_T)$ be 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 $G$. 
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 $e \in E$
    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: 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

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

![Graph G](image1)

![MST of G](image2)

Figure: Graph $G$

Figure: MST of $G$
**Prim’s Algorithm**

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

![Graph G](image1.png)

**Figure: Graph G**

![MST of G](image2.png)

**Figure: MST of G**
**Prim’s Algorithm**

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

![Graph $G$](image1)

![Minimum Spanning Tree (MST) of $G$](image2)

**Figure: Graph $G$**

**Figure: MST of $G$**
Prim’s Algorithm

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

![Figure: Graph $G$](image1)

![Figure: MST of $G$](image2)
Prim’s Algorithm

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

**Figure: Graph $G$**

**Figure: MST of $G$**
Prim’s Algorithm

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

Figure: Graph $G$

Figure: MST of $G$
Prim’s Algorithm

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

![Graph G](image1.png)

![MST of G](image2.png)

Figure: Graph $G$

Figure: MST of $G$
Reverse Delete Algorithm

Initially $E$ is the set of all edges in $G$
$T$ is $E$ (* $T$ will store edges of a MST *)

while $E$ is not empty do
    choose $e \in E$ of largest cost
    if removing $e$ does not disconnect $T$ then
        remove $e$ from $T$
    return the set $T$

Returns a minimum spanning tree.
Borůvka’s Algorithm

Simplest to implement. See notes.
Assume $G$ is a connected graph.

$$
T \text{ is } \emptyset \text{ (* } T \text{ will store edges of a MST *)}
$$

while $T$ is not spanning do

$X \leftarrow \emptyset$

for each connected component $S$ of $T$ do

add to $X$ the cheapest edge between $S$ and $V \setminus S$

Add edges in $X$ to $T$

return the set $T$
Borůvka’s Algorithm
Correctness of MST Algorithms

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

And for now . . .

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 $(S, V \setminus S)$. Edges having an endpoint on both sides are the edges of the cut.

A cut edge is crossing the cut.
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.
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 $C$.

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

Proof.
Exercise.
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 \( C \).

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

**Proof.**
Exercise.
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 \( C \).

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

Proof.
Exercise.
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!
Every cut identifies one safe edge...

...the cheapest edge in the cut.

Note: An edge $e$ may be a safe edge for many cuts!
Every cycle identifies one **unsafe** edge...

...the most expensive edge in the cycle.
Every cycle identifies one **unsafe** edge...

...the most expensive edge in the cycle.
Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the **MST** in this case...
Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...
Figure: Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...
Key Observation: Cut Property

**Lemma**

*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!!
Lemma

If \( e \) is a safe edge then every minimum spanning tree contains \( e \).

Proof.

1. Suppose (for contradiction) \( e \) is not in \( \text{MST} \ T \).
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

\textit{If }\ e \ \textit{is a safe edge then every minimum spanning tree contains }\ e.\n
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! \textbf{Error: }\ T' \ \textbf{may not be a spanning tree}!!
Error in Proof: Example

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

(A) Consider adding the edge \( f \).
Error in Proof: Example

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

1. (A) Consider adding the edge \( f \).
2. (B) It is safe because it is the cheapest edge in the cut.
Error in Proof: Example

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

(A) Consider adding the edge \( f \).

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

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

1. (A) Consider adding the edge \( f \).
2. (B) It is safe because it is the cheapest edge in the cut.
3. (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...
4. (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 \( \text{MST } T \) 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

\[ T' = (T \setminus \{e'\}) \cup \{e\} \text{ is a spanning tree.} \]

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.
Observation

\[ T' = (T \setminus \{e'\}) \cup \{e\} \text{ is a spanning tree.} \]

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

\[ T' = (T \setminus \{e'\}) \cup \{e\} \text{ is a spanning tree.} \]

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.
Lemma

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 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.
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.
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.
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.
   - Let $S$ be the vertices connected by edges in $T$ when $e$ is added.
   - $e$ 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
   - Set of edges output forms a connected graph: by induction, $S$ is connected in each iteration and eventually $S = V$.
   - 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.
   - Let $S$ be the vertices connected by edges in $T$ when $e$ is added.
   - $e$ 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
   - Set of edges output forms a connected graph: by induction, $S$ is connected in each iteration and eventually $S = V$.
   - 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. Let $S$ be the vertices connected by edges in $T$ when $e$ is added.
   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
   1. Set of edges output forms a connected graph: by induction, $S$ is connected in each iteration and eventually $S = V$.
   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. Let \( S \) be the vertices connected by edges in \( T \) when \( e \) is added.
   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
   1. Set of edges output forms a connected graph: by induction, \( S \) is connected in each iteration and eventually \( S = V \).
   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. Let $S$ be the vertices connected by edges in $T$ when $e$ is added.
   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
   1. Set of edges output forms a connected graph: by induction, $S$ is connected in each iteration and eventually $S = V$.
   2. Only safe edges added and they do not have a cycle.
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
   - 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 $T$

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. When algorithm adds \( e \) let \( S \) and \( S' \) be the connected components containing \( u \) and \( v \) respectively
   2. \( e \) is the lowest cost edge crossing \( S \) (and also \( S' \)).
   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. When algorithm adds \( e \) let \( S \) and \( S' \) be the connected components containing \( u \) and \( v \) respectively
   2. \( e \) is the lowest cost edge crossing \( S \) (and also \( S' \)).
   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. When algorithm adds \( e \) let \( S \) and \( S' \) be the connected components containing \( u \) and \( v \) respectively
   
   2. \( e \) is the lowest cost edge crossing \( S \) (and also \( S' \)).
   
   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
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.
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 \textbf{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 $(c(e_i) = c(e_j) \text{ and } i < j)$

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 $(c(A) = c(B) \text{ and } A \setminus B \text{ has a lower indexed edge than } B \setminus A)$

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. \( e_i \preceq e_j \) 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 \( A \preceq B \) 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.
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 **MST**s 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?
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?
Part II

Data Structures for MST: Priority Queues and Union-Find
Implementing Borůvka’s Algorithm

No complex data structure needed.

\[ T \text{ is } \emptyset (* \ T \text{ will store edges of a MST } *) \]

\[ \text{while } T \text{ is not spanning do} \]

\[ X \leftarrow \emptyset \]

\[ \text{for each connected component } S \text{ of } T \text{ do} \]

\[ \text{add to } X \text{ the cheapest edge between } S \text{ and } V \setminus S \]

\[ \text{Add edges in } X \text{ to } T \]

\[ \text{return the set } T \]

- \( O(\log n) \) 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 \( \emptyset \) (* \( T \) will store edges of a MST *)

while \( T \) is not spanning do

\[ X \leftarrow \emptyset \]

for each connected component \( S \) of \( T \) do

add to \( X \) the cheapest edge between \( S \) and \( V \setminus S \)

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.

\[
\begin{align*}
T & \text{ is } \emptyset \text{ (* } T \text{ will store edges of a MST *)} \\
\text{while } T \text{ is not spanning do} & \\
X & \leftarrow \emptyset \\
\text{for each connected component } S \text{ of } T \text{ do} & \\
& \text{add to } X \text{ the cheapest edge between } S \text{ and } V \setminus S \\
& \text{Add edges in } X \text{ to } T \\
\text{return the set } T
\end{align*}
\]

- \(O(\log n)\) 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 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
\[ v \in S \text{ and } w \in V - S \]
\( e \) has minimum cost

\[ T = T \cup e \]
\[ S = S \cup w \]

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

**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\n
- \( v \in S \) and \( w \in V - S \)
- \( e \) has minimum cost

\( T = T \cup e \)

\( S = S \cup w \)

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

**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 \( w \in V - S \)
  - \( e \) has minimum cost

- **\( T = T \cup e \)**
- **\( S = S \cup w \)**

**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

Prim\_ComputeMST

\[ E \text{ is the set of all edges in } G \]
\[ S = \{1\} \]
\[ T \text{ is empty } (* T \text{ will store edges of a MST } *) \]

while \( S \neq V \) do

pick \( e = (v, w) \in E \) such that

\[ v \in S \text{ and } w \in V - S \]
\[ e \text{ has minimum cost} \]

\[ T = T \cup e \]
\[ S = S \cup w \]

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

More Efficient Implementation

Prim_ComputeMST

1. \( E \) is the set of all edges in \( G \)
2. \( S = \{1\} \)
3. \( T \) is empty (* \( T \) will store edges of a \( MST \) *)
4. for \( v \not\in S \), \( a(v) = \min_{w \in S} c(w, v) \)
5. for \( v \not\in S \), \( e(v) = w \) such that \( w \in S \) and \( c(w, v) \) is minimum
6. while \( S \neq V \) do
   - pick \( v \) with minimum \( a(v) \)
   - \( T = T \cup \{(e(v), v)\} \)
   - \( S = S \cup \{v\} \)
   - update arrays \( a \) and \( e \)
7. return the set \( T \)

Maintain vertices in \( V \setminus S \) in a priority queue with key \( a(v) \).
**Prim’s Algorithm**

More Efficient Implementation

**Prim**

ComputeMST

- $E$ is the set of all edges in $G$
- $S = \{1\}$
- $T$ is empty (* $T$ will store edges of a MST *)
- for $v \not\in S$, $a(v) = \min_{w \in S} c(w, v)$
- for $v \not\in S$, $e(v) = w$ such that $w \in S$ and $c(w, v)$ is minimum
- while $S \neq V$ do
  - pick $v$ with minimum $a(v)$
  - $T = T \cup \{(e(v), v)\}$
  - $S = S \cup \{v\}$
  - update arrays $a$ and $e$
- return the set $T$

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

---

Sariel Har-Peled (UIUC)  
CS374  
Fall 2017  
35 / 41
Implementing Prim’s Algorithm

More Efficient Implementation

```
Prim_ComputeMST
E is the set of all edges in G
S = \{1\}
T is empty (* T will store edges of a MST *)
for v \notin S, a(v) = \min_{w \in S} c(w, v)
for v \notin S, e(v) = w such that w \in S and c(w, v) is minimum
while S \neq V do
    pick v with minimum a(v)
    T = T \cup \{(e(v), v)\}
    S = S \cup \{v\}
    update arrays a and e
return the set T
```

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

\[ E \text{ is the set of all edges in } G \]
\[ S = \{1\} \]
\[ T \text{ is empty (} T \text{ will store edges of a MST }) \]
for \( v \not\in S \), \( a(v) = \min_{w \in S} c(w, v) \)
for \( v \not\in S \), \( e(v) = w \) such that \( w \in S \) and \( c(w, v) \) is minimum
while \( S \neq V \) do
  pick \( v \) with minimum \( a(v) \)
  \[ T = T \cup \{(e(v), v)\} \]
  \[ S = S \cup \{v\} \]
  update arrays \( a \) and \( e \)
return the set \( T \)

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

1. Requires \( O(n) \) \textbf{extractMin} operations
2. Requires \( O(m) \) \textbf{decreaseKey} operations
Prim’s using priority queues

\[ E \text{ is the set of all edges in } G \]
\[ S = \{1\} \]
\[ T \text{ is empty (* } T \text{ will store edges of a MST *)} \]

for \( v \not\in S \), \( a(v) = \min_{w \in S} c(w, v) \)
for \( v \not\in S \), \( e(v) = w \) such that \( w \in S \) and \( c(w, v) \) is minimum

while \( S \neq V \) do
pick \( v \) with minimum \( a(v) \)
\( T = T \cup \{(e(v), v)\} \)
\( S = S \cup \{v\} \)
update arrays \( a \) and \( e \)

return the set \( T \)

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

1. Requires \( O(n) \) \texttt{extractMin} operations
2. Requires \( O(m) \) \texttt{decreaseKey} operations
Prim’s using priority queues

\[ E \] is the set of all edges in \( G \)
\[ S = \{1\} \]
\[ T \] is empty (* \( T \) will store edges of a MST *)
for \( v \notin S \), \( a(v) = \min_{w \in S} c(w, v) \)
for \( v \notin S \), \( e(v) = w \) such that \( w \in S \) and \( c(w, v) \) is minimum
while \( S \neq V \) do
    pick \( v \) with minimum \( a(v) \)
    \[ T = T \cup \{(e(v), v)\} \]
    \[ S = S \cup \{v\} \]
    update arrays \( a \) and \( e \)
return the set \( T \)

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

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: \(O(n \log n + m)\).

Prim’s algorithm and Dijkstra’s algorithms are similar. Where is the difference?
Running time of Prim’s Algorithm

\( O(n) \) \textbf{extractMin} operations and \( O(m) \) \textbf{decreaseKey} operations

1. Using standard Heaps, \textbf{extractMin} and \textbf{decreaseKey} take \( O(\log n) \) time. Total: \( O((m + n) \log n) \)

2. Using Fibonacci Heaps, \( O(\log n) \) for \textbf{extractMin} and \( O(1) \) (amortized) for \textbf{decreaseKey}. Total: \( O(n \log n + m) \).

Prim’s algorithm and Dijkstra’s algorithms are similar. Where is the difference?
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 \( e \in E \) of minimum cost
    if \( (T \cup \{e\} \) 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

\[ \text{Kruskal}\_\text{ComputeMST} \]

Initially \( E \) is the set of all edges in \( G \)
\( T \) is empty (* \( T \) will store edges of a MST *)

\[ \text{while } E \text{ is not empty do} \]

\[ \text{choose } e \in E \text{ of minimum cost} \]

\[ \text{if } ( T \cup \{e\} \text{ does not have cycles}) \]

\[ \text{add } e \text{ to } T \]

\[ \text{return the set } T \]

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

\begin{algorithm}
\caption{Kruskal\_ComputeMST}
Initially \( E \) is the set of all edges in \( G \)
\( T \) is empty (* \( T \) will store edges of a MST *)
\begin{algorithmic}
\While {\( E \) is not empty}
\State choose \( e \in E \) of minimum cost
\If {\( \{ T \cup \{ e \} \) does not have cycles}
\State add \( e \) to \( T \)
\EndIf
\EndWhile
\State return the set \( T \)
\end{algorithmic}
\end{algorithm}

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

\[
\textbf{Kruskal ComputeMST} \\
\text{Initially } E \text{ is the set of all edges in } G \text{ } \\
T \text{ is empty (* } T \text{ will store edges of a MST *)} \\
\textbf{while } E \text{ is not empty do} \\
\quad \text{choose } e \in E \text{ of minimum cost} \\
\quad \textbf{if } (T \cup \{e\} \text{ does not have cycles}) \\
\quad \quad \text{add } e \text{ to } T \\
\textbf{return} \text{ the set } T
\]

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 $e \in E$ of minimum cost

if $(T \cup \{e\}$ 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 $e \in E$ of minimum cost
  if $(T \cup \{e\}$ 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 (\textit{* T will store edges of a MST *})
each vertex $u$ is placed in a set by itself

while $E$ is not empty do
    pick $e = (u, v) \in E$ 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 \textbf{Union-Find} data structure can implement Kruskal’s algorithm
in $O((m + n) \log m)$ time.
Implementing Kruskal’s Algorithm Efficiently

```plaintext
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 $e = (u, v) \in E$ 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.
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, 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, 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?
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, 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?