Greedy Algorithms for Minimum Spanning Trees

Lecture 13
March 5, 2015
Part I

Greedy Algorithms: 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 $G$
The Minimum Spanning Tree problem involves finding a spanning tree in a connected weighted graph such that the total weight of the tree is as small as possible. Formally, given a connected graph $G = (V, E)$ with edge costs, the goal is to find a subset $T \subseteq E$ such that $(V, T)$ is connected and the total cost of all edges in $T$ is smallest.

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

In the diagram, we see a graph with weighted edges, and the minimum spanning tree is highlighted in red. The edges are labeled with their respective weights.
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 \( G \)

\[
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
20 & & & & & 23 & 1 & 4 & 15 \\
& & & & & 36 & & & 9 \\
& & & & & & 28 & 25 & 16 \\
& & & & & & & 17 & 3 \\
\end{array}
\]
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**
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**
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
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
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
Initially $E$ is the set of all edges in $G$ and $T$ is empty (* $T$ will store edges of a MST *)

while $E$ is not empty do

    choose $i \in E$

    if ($i$ satisfies condition)
        add $i$ to $T$

return the set $T$

Main Task: In what order should edges be processed? When should we add edge to spanning tree?
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 $i \in E$
  if ($i$ satisfies condition) add $i$ 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 $T$.

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](image)

![MST of G](image)

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 $T$.

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 $T$.

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 $G$
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 $i \in E$ of largest cost
    if removing $i$ does not disconnect $T$ then
        remove $i$ from $T$
    return the set $T$

Returns a minimum spanning tree.
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 $i \in E$ of largest cost

  if removing $i$ does not disconnect $T$ then

    remove $i$ from $T$

return the set $T$

Returns a minimum spanning tree.
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.
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.
Assumption

And for now . . .

**Assumption**

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

**Definition**

1. $G = (V, E)$: graph. A **cut** is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.

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

3. A cut edge is **crossing** the cut.
Definition

1. \( G = (V, E) \): graph. A cut is a partition of the vertices of the graph into two sets \((S, V \setminus S)\).

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

3. A cut edge is crossing the cut.
Cuts

Definition

1. $G = (V, E)$: graph. A cut is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.
2. Edges having an endpoint on both sides are the edges of the cut.
3. A cut edge is crossing the cut.
Cuts

**Definition**

1. $G = (V, E)$: graph. A **cut** is a partition of the vertices of the graph into two sets $(S, V \setminus S)$.
2. Edges having an endpoint on both sides are the **edges of the cut**.
3. A cut edge is **crossing** the cut.
**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...

1. Every cut identifies one safe edge...

2. ...the cheapest edge in the cut.

3. 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 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 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...

1. Every cycle identifies one unsafe edge...

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

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

2. ...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...
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!
5. Error: \( T' \) may not be a spanning tree!!
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!
5. 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 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!
5. Error: \( T' \) may not be a spanning tree!!
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!
5. Error: $T'$ may not be a spanning tree!!
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!

5. Error: \( T' \) may not be a spanning tree!!
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!
5. Error: $T'$ may not be a spanning tree!!
Error in Proof: Example

Problematic example. \( S = \{1, 2, 7\}, \ e = (7, 3), \ f = (1, 6). \ \ T - f + e \) 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.

1. (A) Consider adding the edge $f$.
2. (B) It is safe because it is the cheapest edge in the cut.
3. (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) 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...
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 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.

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

\[ T' = (T \setminus \{e'\}) \cup \{e\} \] 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.**

\[ T' \text{ is connected.} \]

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.

\[ T' \text{ is a tree} \]

\( T' \) 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 \( P \to f \to e \) 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. See text book.

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.
   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.
   - 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 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
   - 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
   - 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
   - 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
   - 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 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 (see text book).
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.
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 \text{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 **MST**s but not for shortest paths?

3. Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.
Part II

Data Structures for MST: Priority Queues and Union-Find
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

PrimComputeMST

- $E$ 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* 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

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

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 $V \setminus S$ in a priority queue with key $a(v)$. 
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 $v \not\in S$, $a(v) = \min_{w \in S} c(w, v)$

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 $a(v)$. 
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 $v \not\in S$, $a(v) = \min_{w \in S} c(w, v)$
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 $a(v)$. 
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 \( O(n) \) extractMin operations
2. Requires \( O(m) \) decreaseKey operations
Prim's using priority queues

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

Maintain vertices in \(\textbf{V} \setminus \textbf{S}\) 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 $v \not\in S$, $a(v) = \min_{w \in S} c(w, v)$
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 $O(n)$ extractMin operations
2. Requires $O(m)$ decreaseKey operations
Running time of Prim’s Algorithm

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

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

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

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

\( O(n) \) extractMin operations and \( O(m) \) decreaseKey operations

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?
Kruskal’s Algorithm

\textbf{Kruskal\_ComputeMST}

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

\textbf{while} $E$ is not empty \textbf{do}
\begin{itemize}
  \item choose $e \in E$ of minimum cost
  \item if ($T \cup \{e\}$ does not have cycles) add $e$ to $T$
\end{itemize}
\textbf{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 $O(m \log m) + O(mn) = O(mn)$
Kruskal’s Algorithm

\[\text{Kruskal ComputeMST}\]

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 $O(m \log m) + O(mn) = O(mn)$
Kruskal’s Algorithm

\textbf{Kruskal\_ComputeMST}

Initially $E$ is the set of all edges in $G$

$T$ is empty (* $T$ will store edges of a MST *)

\textbf{while} $E$ is not empty \textbf{do}

\hspace{10pt} choose $e \in E$ of minimum cost

\hspace{10pt} \textbf{if} ($T \cup \{e\}$ does not have cycles)

\hspace{20pt} add $e$ to $T$

\textbf{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

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

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 \( O(m \log m) + O(mn) = O(mn) \)
### 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.
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.
Implementing Kruskal’s Algorithm Efficiently

\[
\text{Kruskal\_ComputeMST}
\]

Sort edges in \( E \) based on cost

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

each vertex \( u \) is placed in a set by itself

\textbf{while} \( E \) is not empty \textbf{do}

\textbf{pick} \( e = (u, v) \in E \) of minimum cost

\textbf{if} \( u \) and \( v \) belong to different sets

\textbf{add} \( e \) to \( T \)

\textbf{merge} the sets containing \( u \) and \( v \)

\textbf{return} the set \( T \)

Need a data structure to check if two elements belong to same set and to merge two sets.
Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. \textbf{makeUnionFind}(S) returns a data structure where each element of S is in a separate set

2. \textbf{find}(u) returns the name of set containing element u. Thus, u and v belong to the same set if and only if \textbf{find}(u) = \textbf{find}(v)

3. \textbf{union}(A, B) merges two sets A and B. Here A and B are the names of the sets. Typically the name of a set is some element in the set.
Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. \text{makeUnionFind}(S) \text{ returns a data structure where each element of } S \text{ is in a separate set}

2. \text{find}(u) \text{ returns the name of set containing element } u. \text{ Thus, } u \text{ and } v \text{ belong to the same set if and only if } \text{find}(u) = \text{find}(v)

3. \text{union}(A, B) \text{ merges two sets } A \text{ and } B. \text{ Here } A \text{ and } B \text{ are the names of the sets. Typically the name of a set is some element in the set.}
Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. **makeUnionFind**\((S)\) returns a data structure where each element of \(S\) is in a separate set

2. **find**\((u)\) returns the *name* of set containing element \(u\). Thus, \(u\) and \(v\) belong to the same set if and only if \(\text{find}(u) = \text{find}(v)\)

3. **union**\((A, B)\) merges two sets \(A\) and \(B\). Here \(A\) and \(B\) are the names of the sets. Typically the name of a set is some element in the set.
Implementing Union-Find using Arrays and Lists

Using lists

1. Each set stored as list with a name associated with the list.
2. For each element \( u \in S \) a pointer to the its set. Array for pointers: \( \text{component}[u] \) is pointer for \( u \).
3. \text{makeUnionFind} (S) takes \( O(n) \) time and space.
Example
1. `find(u)` reads the entry `component[u]`: $O(1)$ time

2. `union(A, B)` involves updating the entries `component[u]` for all elements $u$ in $A$ and $B$: $O(|A| + |B|)$ which is $O(n)$
Implementing Union-Find using Arrays and Lists

1. \textbf{find}(u) reads the entry component\([u]\): \(O(1)\) time
2. \textbf{union}(A, B) involves updating the entries component\([u]\) for all elements \(u\) in \(A\) and \(B\): \(O(|A| + |B|)\) which is \(O(n)\)
Implementing Union-Find using Arrays and Lists

1. \textbf{\texttt{find}}(u) reads the entry component\([u]\): \(O(1)\) time

2. \textbf{\texttt{union}}(A, B) involves updating the entries component\([u]\) for all elements \(u\) in \(A\) and \(B\): \(O(|A| + |B|)\) which is \(O(n)\)
New Implementation

As before use component\( [u] \) to store set of \( u \).

Change to \texttt{union}(A, B):

1. with each set, keep track of its size
2. assume \(|A| \leq |B|\) for now
3. Merge the list of \( A \) into that of \( B \): \( O(1) \) time (linked lists)
4. Update \texttt{component}[u] only for elements in the smaller set \( A \)
5. Total \( O(|A|) \) time. Worst case is still \( O(n) \).

\texttt{find} still takes \( O(1) \) time
New Implementation

As before use component$[u]$ to store set of $u$. Change to $\text{union}(A, B)$:

1. with each set, keep track of its size
2. assume $|A| \leq |B|$ for now
3. Merge the list of $A$ into that of $B$: $O(1)$ time (linked lists)
4. Update component$[u]$ only for elements in the smaller set $A$
5. Total $O(|A|)$ time. Worst case is still $O(n)$.

$\text{find}$ still takes $O(1)$ time
## New Implementation

As before use `component[u]` to store set of `u`.

Change to `union(A, B)`:  

1. with each set, keep track of its size  
2. assume `|A| \leq |B|` for now  
3. Merge the list of `A` into that of `B`: \( O(1) \) time (linked lists)  
4. Update `component[u]` only for elements in the smaller set `A`  
5. Total \( O(|A|) \) time. Worst case is still \( O(n) \).

`find` still takes \( O(1) \) time.
The smaller set (list) is appended to the largest set (list)
Improving the List Implementation for Union

Question
Is the improved implementation provably better or is it simply a nice heuristic?

Theorem
Any sequence of \( k \) union operations, starting from \( \text{makeUnionFind}(S) \) on set \( S \) of size \( n \), takes at most \( O(k \log k) \).

Corollary
Kruskal’s algorithm can be implemented in \( O(m \log m) \) time.

Sorting takes \( O(m \log m) \) time, \( O(m) \) finds take \( O(m) \) time and \( O(n) \) unions take \( O(n \log n) \) time.
Improving the List Implementation for Union

Question
Is the improved implementation provably better or is it simply a nice heuristic?

Theorem
Any sequence of $k$ union operations, starting from makeUnionFind($S$) on set $S$ of size $n$, takes at most $O(k \log k)$.

Corollary
Kruskal’s algorithm can be implemented in $O(m \log m)$ time.

Sorting takes $O(m \log m)$ time, $O(m)$ finds take $O(m)$ time and $O(n)$ unions take $O(n \log n)$ time.
Improving the List Implementation for Union

**Question**
Is the improved implementation provably better or is it simply a nice heuristic?

**Theorem**
Any sequence of \( k \) union operations, starting from \( \text{makeUnionFind}(S) \) on set \( S \) of size \( n \), takes at most \( O(k \log k) \).

**Corollary**
Kruskal’s algorithm can be implemented in \( O(m \log m) \) time.

Sorting takes \( O(m \log m) \) time, \( O(m) \) finds take \( O(m) \) time and \( O(n) \) unions take \( O(n \log n) \) time.
Why does theorem work?

**Key Observation**

\( \text{union}(A, B) \) takes \( O(|A|) \) time where \( |A| \leq |B| \). Size of new set is \( \geq 2|A| \). Cannot double too many times.
Proof of Theorem

Proof.

1. Any union operation involves at most 2 of the original one-element sets; thus at least $n - 2k$ elements have never been involved in a union.

2. Also, maximum size of any set (after $k$ unions) is $2k$.

3. $\text{union}(A, B)$ takes $O(|A|)$ time where $|A| \leq |B|$.

4. Charge each element in $A$ constant time to pay for $O(|A|)$ time.

5. How much does any element get charged?

6. If component $[v]$ is updated, set containing $v$ doubles in size.

7. Component $[v]$ is updated at most $\log 2k$ times.

8. Total number of updates is $2k \log 2k = O(k \log k)$. 

\hfill $\square$
Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root’s name.

1. **find**(\(u\)): Traverse from \(u\) to the root
2. **union**(\(A, B\)): Make root of \(A\) (smaller set) point to root of \(B\).

Takes \(O(1)\) time.
Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root’s name.

1. **find**(*u*): Traverse from *u* to the root

2. **union**(*A*, *B*): Make root of *A* (smaller set) point to root of *B*. Takes \( O(1) \) time.
Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root’s name.

1. **find** (*u*): Traverse from *u* to the root
2. **union** (*A, B*): Make root of *A* (smaller set) point to root of *B*. Takes \(O(1)\) time.
Details of Implementation

Each element \( u \in S \) has a pointer \( \text{parent}(u) \) to its ancestor.

\[
\text{makeUnionFind}(S) \\
\quad \text{for each } u \text{ in } S \text{ do} \\
\quad \quad \text{parent}(u) = u
\]

\[
\text{find}(u) \\
\quad \text{while } (\text{parent}(u) \neq u) \text{ do} \\
\quad \quad u = \text{parent}(u) \\
\quad \text{return } u
\]

\[
\text{union}(\text{component}(u), \text{component}(v)) \\
\quad \text{(* parent}(u) = u \& \text{parent}(v) = v \text{ *)} \\
\quad \text{if } (|\text{component}(u)| \leq |\text{component}(v)|) \text{ then} \\
\quad \quad \text{parent}(u) = v \\
\quad \text{else} \\
\quad \quad \text{parent}(v) = u \\
\quad \text{set new component size to } |\text{component}(u)| + |\text{component}(v)|
\]
Details of Implementation

Each element $u \in S$ has a pointer $\text{parent}(u)$ to its ancestor.

\begin{verbatim}
makeUnionFind(S)
    for each u in S do
        parent(u) = u

find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u

union(component(u), component(v))
    (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
\end{verbatim}
Details of Implementation

Each element $u \in S$ has a pointer $\text{parent}(u)$ to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
    (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```
Each element $u \in S$ has a pointer $\text{parent}(u)$ to its ancestor.

```
\text{makeUnionFind}(S)
\quad \text{for each } u \text{ in } S \text{ do}
\quad \quad \text{parent}(u) = u
```

```
\text{find}(u)
\quad \text{while } (\text{parent}(u) \neq u) \text{ do}
\quad \quad u = \text{parent}(u)
\quad \text{return } u
```

```
\text{union}(\text{component}(u), \text{component}(v))
\quad \text{(* parent}(u) = u \text{ & parent}(v) = v \text{ *)}
\quad \text{if } (|\text{component}(u)| \leq |\text{component}(v)|) \text{ then}
\quad \quad \text{parent}(u) = v
\quad \text{else}
\quad \quad \text{parent}(v) = u
\quad \text{set new component size to } |\text{component}(u)| + |\text{component}(v)|
```
Analysis

Theorem

The forest based implementation for a set of size \( n \), has the following complexity for the various operations: makeUnionFind takes \( O(n) \), union takes \( O(1) \), and find takes \( O(\log n) \).

Proof.

1. \textbf{find}(u) depends on the height of tree containing \( u \).
2. Height of \( u \) increases by at most 1 only when the set containing \( u \) changes its name.
3. If height of \( u \) increases then size of the set containing \( u \) (at least) doubles.
4. Maximum set size is \( n \); so height of any tree is at most \( O(\log n) \).
Further Improvements: Path Compression

Observation
Consecutive calls of \(\text{find}(u)\) take \(O(\log n)\) time each, but they traverse the same sequence of pointers.

Idea: Path Compression
Make all nodes encountered in the \(\text{find}(u)\) point to root.
Further Improvements: Path Compression

Observation

Consecutive calls of \texttt{find}(u) take $O(\log n)$ time each, but they traverse the same sequence of pointers.

Idea: Path Compression

Make all nodes encountered in the \texttt{find}(u) point to root.
Path Compression: Example

Before find(u):

After find(u):

```
    r
   / 
  v   w
 /     
```

```
    r
   /   
  v     w
     /   
      u
```
Path Compression

\[ \text{find}(u): \]
\[ \text{if } (\text{parent}(u) \neq u) \text{ then} \]
\[ \text{parent}(u) = \text{find}(\text{parent}(u)) \]
\[ \text{return } \text{parent}(u) \]

Question

Does Path Compression help?

Yes!

Theorem

With Path Compression, \( k \) operations (\text{find} and/or \text{union}) take \( O(k\alpha(k, \min\{k, n\})) \) time where \( \alpha \) is the inverse Ackermann function.
Path Compression

\[ \text{find}(u): \]
\[ \text{if } (\text{parent}(u) \neq u) \text{ then} \]
\[ \text{parent}(u) = \text{find}(\text{parent}(u)) \]
\[ \text{return } \text{parent}(u) \]

**Question**

Does Path Compression help?

Yes!

**Theorem**

*With Path Compression, \( k \) operations (\text{find} and/or \text{union}) take \( O(k \alpha(k, \min\{k, n\})) \) time where \( \alpha \) is the inverse Ackermann function.*
find\( (u) \):
    if \( \text{parent}(u) \neq u \) then
        \text{parent}(u) = \text{find}(\text{parent}(u))
    \text{return} \text{parent}(u)

Question
Does Path Compression help?
Yes!

Theorem
With Path Compression, \( k \) operations (find and/or union) take \( O(k\alpha(k, \min\{k, n\})) \) time where \( \alpha \) is the inverse Ackermann function.
Ackermann and Inverse Ackermann Functions

Ackermann function $A(m, n)$ defined for $m, n \geq 0$ recursively

$$A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, n) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}$$

$A(3, n) = 2^{n+3} - 3$
$A(4, 3) = 2^{65536} - 3$

$\alpha(m, n)$ is inverse Ackermann function defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

For all practical purposes $\alpha(m, n) \leq 5$
Ackermann and Inverse Ackermann Functions

Ackermann function $A(m, n)$ defined for $m, n \geq 0$ recursively

$$A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0
\end{cases}$$

$A(3, n) = 2^{n+3} - 3$
$A(4, 3) = 2^{65536} - 3$

$\alpha(m, n)$ is inverse Ackermann function defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

For all practical purposes $\alpha(m, n) \leq 5$
Ackermann and Inverse Ackermann Functions

Ackermann function $A(m, n)$ defined for $m, n \geq 0$ recursively

$$A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}$$

$A(3, n) = 2^{n+3} - 3$
$A(4, 3) = 2^{65536} - 3$

$\alpha(m, n)$ is inverse Ackermann function defined as

$$\alpha(m, n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}$$

For all practical purposes $\alpha(m, n) \leq 5$
Amazing result:

**Theorem (Tarjan)**

For **Union-Find**, any data structure in the pointer model requires \( \Omega(m \alpha(m, n)) \) time for \( m \) operations.
Running time of Kruskal’s Algorithm

Using Union-Find data structure:

1. $O(m)$ find operations (two for each edge)
2. $O(n)$ union operations (one for each edge added to $T$)
3. Total time: $O(m \log m)$ for sorting plus $O(m \alpha(n))$ for union-find operations. Thus $O(m \log m)$ time despite the improved Union-Find data structure.
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 and Willard [1994].
3. $O(m + n)$ expected time (randomized algorithm) Karger et al. [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 and Willard [1994].
3. \( O(m + n) \) expected time (randomized algorithm) Karger et al. [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 and Tarjan [1987].
2. \( O(m + n) \) time using bit operations in RAM model Fredman and Willard [1994].
3. \( O(m + n) \) expected time (randomized algorithm) Karger et al. [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?

