6.3: Minimum Spanning Tree

Frank Stajano

Thomas Sauerwald

Lent 2016
Minimum Spanning Tree Problem

- **Given:** undirected, connected graph $G = (V, E, w)$ with non-negative edge weights.

---

Minimum Spanning Tree Problem

Street Networks, Wiring Electronic Components, Laying Pipes

Weights may represent distances, costs, travel times, capacities, resistance etc.

Applications

6.3: Minimum Spanning Tree T.S. 2
Minimum Spanning Tree Problem

- **Given:** undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- **Goal:** Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

Applications

- Street Networks
- Wiring Electronic Components
- Laying Pipes

Weights may represent distances, costs, travel times, capacities, resistances, etc.
Minimum Spanning Tree Problem

- **Given:** undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- **Goal:** Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

Must be necessarily a tree!
Minimum Spanning Tree Problem

- Given: undirected, connected graph $G = (V, E, w)$ with non-negative edge weights
- Goal: Find a subgraph $\subseteq E$ of minimum total weight that links all vertices

Applications

- Street Networks, Wiring Electronic Components, Laying Pipes
- Weights may represent distances, costs, travel times, capacities, resistance etc.
Generic Algorithm

0: def minimum spanningTree(G)
1:   A = empty set of edges
2:   while A does not span all vertices yet:
3:     add a safe edge to A
Generic Algorithm

0: def minimum spanningTree(G)
1:   A = empty set of edges
2:   while A does not span all vertices yet:
3:     add a safe edge to A

Definition

An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.
0: def minimum spanningTree(G)
1:   A = empty set of edges
2:   while A does not span all vertices yet:
3:     add a safe edge to A

Definition
An edge of $G$ is safe if by adding the edge to $A$, the resulting subgraph is still a subset of a minimum spanning tree.

How to find a safe edge?
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
Finding safe edges

Definitions

- a cut is a partition of \( V \) into at least two disjoint sets
- a cut respects \( A \subseteq E \) if no edge of \( A \) goes across the cut
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut

Definitions

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.
Finding safe edges

**Definitions**
- a **cut** is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut
Finding safe edges

Definitions

- a cut is a partition of $V$ into at least two disjoint sets
- a cut respects $A \subseteq E$ if no edge of $A$ goes across the cut

Theorem

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.
**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**
Proof of Theorem

Theorem
Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:
- Let $T$ be a MST containing $A$
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$
Proof of Theorem

**Theorem**

Let \( A \subseteq E \) be a subset of a MST of \( G \). Then for any cut that respects \( A \), the lightest edge of \( G \) that goes across the cut is safe.

**Proof:**

- Let \( T \) be a MST containing \( A \)
- Let \( e_\ell \) be the lightest edge across the cut
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is **safe**.

**Proof:**

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces a cycle crossing the cut through $e_\ell$ and another edge $e_x$.

Consider the tree $T \cup e_\ell \setminus e_x$:
- This tree must be a spanning tree.
- If $w(e_\ell) < w(e_x)$, then this spanning tree has a smaller cost than $T$ (can't happen!)
- If $w(e_\ell) = w(e_x)$, then $T \cup e_\ell \setminus e_x$ is a MST.
Proof of Theorem

Theorem
Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

Proof:
- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
Proof of Theorem

**Theorem**
Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**
- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the \textit{lightest edge} of $G$ that goes across the cut is \textit{safe}.

**Proof:**

- Let $T$ be a MST containing $A$.
- Let $e_\ell$ be the \textit{lightest} edge across the cut.
- If $e_\ell \in T$, then we are done.
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces a cycle.
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$. 
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$.
- Let $e_\ell$ be the lightest edge across the cut.
- If $e_\ell \in T$, then we are done.
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle.
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$. 
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$: 

![Diagram of a minimum spanning tree with edges $e_\ell$ and $e_x$]
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is **safe**.

**Proof:**

- Let $T$ be a MST containing $A$.
- Let $e_\ell$ be the **lightest** edge across the cut.
- If $e_\ell \in T$, then we are done.
- If $e_\ell \not\in T$, then adding $e_\ell$ to $T$ introduces cycle.
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$.
- Consider now the tree $T \cup e_\ell \setminus e_x$: 

![Diagram](image.png)
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
Proof of Theorem

Let \( A \subseteq E \) be a subset of a MST of \( G \). Then for any cut that respects \( A \), the lightest edge of \( G \) that goes across the cut is safe.

Proof:

- Let \( T \) be a MST containing \( A \)
- Let \( e_\ell \) be the lightest edge across the cut
- If \( e_\ell \in T \), then we are done
- If \( e_\ell \not\in T \), then adding \( e_\ell \) to \( T \) introduces cycle
- This cycle crosses the cut through \( e_\ell \) and another edge \( e_x \)
- Consider now the tree \( T \cup e_\ell \setminus e_x \):
  - This tree must be a spanning tree
  - If \( w(e_\ell) < w(e_x) \), then this spanning tree has smaller cost than \( T \) (can’t happen!)
Proof of Theorem

**Theorem**

Let $A \subseteq E$ be a subset of a MST of $G$. Then for any cut that respects $A$, the lightest edge of $G$ that goes across the cut is safe.

**Proof:**

- Let $T$ be a MST containing $A$
- Let $e_\ell$ be the lightest edge across the cut
- If $e_\ell \in T$, then we are done
- If $e_\ell \notin T$, then adding $e_\ell$ to $T$ introduces cycle
- This cycle crosses the cut through $e_\ell$ and another edge $e_x$
- Consider now the tree $T \cup e_\ell \setminus e_x$:
  - This tree must be a spanning tree
  - If $w(e_\ell) < w(e_x)$, then this spanning tree has smaller cost than $T$ (can't happen!)
  - If $w(e_\ell) = w(e_x)$, then $T \cup e_\ell \setminus e_x$ is a MST.
Glimpse at Kruskal’s Algorithm

Basic Strategy

Let $A \subseteq E$ be a forest, initially empty.

At every step, given $A$, perform:

Add lightest edge to $A$ that does not introduce a cycle.

Use Disjoint Sets to keep track of connected components!
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step,

![Graph](image)

Use Disjoint Sets to keep track of connected components!
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:

![Graph diagram with edge weights]
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:

![Diagram of a graph with labeled edges and vertices]
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:

  **Add lightest edge to $A$ that does not introduce a cycle**

![Graph](image-url)
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:
  
  Add lightest edge to $A$ that does not introduce a cycle

Use Disjoint Sets to keep track of connected components!
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:
  
  Add lightest edge to $A$ that does not introduce a cycle
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let $A \subseteq E$ be a forest, initially empty
- At every step, given $A$, perform:
  
  Add lightest edge to $A$ that does not introduce a cycle

Use Disjoint Sets to keep track of connected components!
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let \( A \subseteq E \) be a forest, initially empty
- At every step, given \( A \), perform:
  
  Add lightest edge to \( A \) that does not introduce a cycle

Use Disjoint Sets to keep track of connected components!
Glimpse at Kruskal’s Algorithm

Basic Strategy

- Let \( A \subseteq E \) be a forest, initially empty
- At every step, given \( A \), perform:
  
  Add lightest edge to \( A \) that does not introduce a cycle

Use Disjoint Sets to keep track of connected components!
Execution of Kruskal's Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal's Algorithm

T.S. 7
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree

T.S.
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Execution of Kruskal’s Algorithm

6.3: Minimum Spanning Tree
Details of Kruskal’s Algorithm

```python
0: def kruskal(G)
1:   Apply Kruskal’s algorithm to graph G
2:   Return set of edges that form a MST
3:
4:   A = Set()  # Set of edges of MST; initially empty.
5:   D = DisjointSet()
6:   for v in G.vertices():
7:       D.makeSet(v)
8:   E = G.edges()
9:   E.sort(key=weight, direction=ascending)
10:
11:  for edge in E:
12:      startSet = D.findSet(edge.start)
13:      endSet = D.findSet(edge.end)
14:      if startSet != endSet:
15:         A.append(edge)
16:         D.union(startSet, endSet)
17:   return A
```

Consider the cut of all connected components (disjoint sets). Line 14 ensures that we extend $A$ by an edge that goes across the cut. This edge is also the lightest edge crossing the cut (otherwise, we would have included a lighter edge before).
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:    Apply Kruskal’s algorithm to graph G
2:    Return set of edges that form a MST
3:    
4: A = Set() # Set of edges of MST; initially empty.
5: D = DisjointSet()
6: for v in G.vertices():
7:    D.makeSet(v)
8: E = G.edges()
9: E.sort(key=weight, direction=ascending)
10: 
11: for edge in E:
12:    startSet = D.findSet(edge.start)
13:    endSet = D.findSet(edge.end)
14:    if startSet != endSet:
15:        A.append(edge)
16:        D.union(startSet, endSet)
17: return A

Time Complexity

Consider the cut of all connected components (disjoint sets)
L. 14 ensures that we extend A by an edge that goes across the cut
This edge is also the lightest edge crossing the cut (otherwise, we
would have included a lighter edge before)

Correctness

6.3: Minimum Spanning Tree
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:
4:     A = Set()  # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:
11:    for edge in E:
12:        startSet = D.findSet(edge.start)
13:        endSet = D.findSet(edge.end)
14:        if startSet != endSet:
15:            A.append(edge)
16:            D.union(startSet, endSet)
17:    return A

Time Complexity

Consider the cut of all connected components (disjoint sets)
L. 14 ensures that we extend A by an edge that goes across the cut
This edge is also the lightest edge crossing the cut (otherwise, we would have included a lighter edge before)

Correctness

6.3: Minimum Spanning Tree
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:
4:     A = Set() # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:
11:     for edge in E:
12:         startSet = D.findSet(edge.start)
13:         endSet = D.findSet(edge.end)
14:         if startSet != endSet:
15:             A.append(edge)
16:             D.union(startSet, endSet)
17:     return A

Time Complexity

- **Initialisation** (l. 4-9): $O(V + E \log E)$
Details of Kruskal’s Algorithm

0: def kruskal(G)
1: Apply Kruskal’s algorithm to graph $G$
2: Return set of edges that form a MST
3:
4: $A = \text{Set()}$ # Set of edges of MST; initially empty.
5: $D = \text{DisjointSet()}$
6: for $v$ in $G$.vertices():
7:   $D$.makeSet($v$)
8: $E = G$.edges()
9: $E$.sort(key=weight, direction=ascending)
10:
11: for edge in $E$:
12:   $\text{startSet} = D$.findSet(edge.start)
13:   $\text{endSet} = D$.findSet(edge.end)
14:   if startSet != endSet:
15:     $A$.append(edge)
16:     $D$.union(startSet, endSet)
17: return $A$

---

Time Complexity

- **Initialisation** (l. 4-9): $\mathcal{O}(V + E \log E)$
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:     
4:     A = Set() # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:     
11:     for edge in E:
12:         startSet = D.findSet(edge.start)
13:         endSet = D.findSet(edge.end)
14:         if startSet != endSet:
15:             A.append(edge)
16:             D.union(startSet,endSet)
17:     return A

Time Complexity

- Initialisation (l. 4-9): $\mathcal{O}(V + E \log E)$
- Main Loop (l. 11-16): $\mathcal{O}(E \cdot \alpha(n))$
Details of Kruskal’s Algorithm

```python
0: def kruskal(G)
1:    Apply Kruskal’s algorithm to graph G
2:    Return set of edges that form a MST
3:
4:    A = Set()  # Set of edges of MST; initially empty.
5:    D = DisjointSet()
6:    for v in G.vertices():
7:        D.makeSet(v)
8:    E = G.edges()
9:    E.sort(key=weight, direction=ascending)
10:
11:   for edge in E:
12:       startSet = D.findSet(edge.start)
13:       endSet = D.findSet(edge.end)
14:       if startSet != endSet:
15:           A.append(edge)
16:           D.union(startSet,endSet)
17:   return A
```

**Time Complexity**

- **Initialisation** (l. 4-9): $O(V + E \log E)$
- **Main Loop** (l. 11-16): $O(E \cdot \alpha(n))$

$\Rightarrow$ **Overall:** $O(E \log E) = O(E \log V)$
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:    Apply Kruskal’s algorithm to graph G
2:    Return set of edges that form a MST
3:
4:    A = Set() # Set of edges of MST; initially empty.
5:    D = DisjointSet()
6:    for v in G.vertices():
7:        D.makeSet(v)
8:    E = G.edges()
9:    E.sort(key=weight, direction=ascending)
10:
11:   for edge in E:
12:       startSet = D.findSet(edge.start)
13:       endSet = D.findSet(edge.end)
14:       if startSet != endSet:
15:           A.append(edge)
16:           D.union(startSet, endSet)
17:   return A

Time Complexity

- Initialisation (l. 4-9): $O(V + E \log E)$
- Main Loop (l. 11-16): $O(E \cdot \alpha(n))$

⇒ Overall: $O(E \log E) = O(E \log V)$

If edges are already sorted, runtime becomes $O(E \cdot \alpha(n))!$
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:      
4:     A = Set() # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:      
11:     for edge in E:
12:         startSet = D.findSet(edge.start)
13:         endSet = D.findSet(edge.end)
14:         if startSet != endSet:
15:             A.append(edge)
16:             D.union(startSet,endSet)
17:     return A

Correctness

6.3: Minimum Spanning Tree
Details of Kruskal’s Algorithm

0: def kruskal(G):
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:
4:     A = Set()  # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:
11:     for edge in E:
12:         startSet = D.findSet(edge.start)
13:         endSet = D.findSet(edge.end)
14:         if startSet != endSet:
15:             A.append(edge)
16:             D.union(startSet,endSet)
17:     return A

Correctness

- Consider the cut of all connected components (disjoint sets)
Details of Kruskal’s Algorithm

0: def kruskal(G)
1:     Apply Kruskal’s algorithm to graph G
2:     Return set of edges that form a MST
3:     
4:     A = Set() # Set of edges of MST; initially empty.
5:     D = DisjointSet()
6:     for v in G.vertices():
7:         D.makeSet(v)
8:     E = G.edges()
9:     E.sort(key=weight, direction=ascending)
10:    
11:     for edge in E:
12:         startSet = D.findSet(edge.start)
13:         endSet = D.findSet(edge.end)
14:         if startSet != endSet:
15:             A.append(edge)
16:             D.union(startSet,endSet)
17:     return A

Correctness

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend A by an edge that goes across the cut
Details of Kruskal’s Algorithm

```python
0: def kruskal(G):
1:     # Apply Kruskal’s algorithm to graph G
2:     # Return set of edges that form a MST
3:     A = Set() # Set of edges of MST; initially empty.
4:     D = DisjointSet()
5:     for v in G.vertices():
6:         D.makeSet(v)
7:     E = G.edges()
8:     E.sort(key=weight, direction=ascending)
9:     for edge in E:
10:        startSet = D.findSet(edge.start)
11:        endSet = D.findSet(edge.end)
12:        if startSet != endSet:
13:            A.append(edge)
14:            D.union(startSet, endSet)
15:     return A
```

Correctness

- Consider the cut of all connected components (disjoint sets)
- L. 14 ensures that we extend A by an edge that goes across the cut
- This edge is also the lightest edge crossing the cut (otherwise, we would have included a lighter edge before)
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex

![Graph diagram with labeled edges and vertices]

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Basic Strategy**

- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to A that does not yield cycle

---

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
**Prim’s Algorithm**

**Basic Strategy**
- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle

---

We computed the same MST as Kruskal, but in a completely different order! Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
**Prim’s Algorithm**

**Basic Strategy**
- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle

![Diagram of a graph with labeled edges and vertices](image)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

---

6.3: Minimum Spanning Tree
Prim’s Algorithm

**Basic Strategy**

- **Start growing a tree** from a designated root vertex
- **At each step**, add **lightest edge** linked to $A$ that does not yield cycle

![Graph](image)

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to \( A \) that does not yield cycle

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
### Prim’s Algorithm

**Basic Strategy**

- **Start** growing a tree from a designated root vertex
- At each step, **add lightest edge** linked to \( A \) that does not yield cycle

---

**Diagram**

We computed the same MST as Kruskal, but in a completely different order! The final MST is given implicitly by the pointers!
Prim’s Algorithm

Basic Strategy
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

We computed the same MST as Kruskal, but in a completely different order! The final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Basic Strategy**

- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle

---

![Graph Diagram](attachment:graph.png)

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Basic Strategy

- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle
Prim’s Algorithm

Basic Strategy

- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to A that does not yield cycle

Implementation will be based on vertices!

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Basic Strategy

- Start **growing a tree** from a designated root vertex
- At each step, **add lightest edge** linked to $A$ that does not yield cycle

Assign every vertex not in $A$ a **key** which is **at all stages** equal to the smallest weight of an edge connecting to $A$.
Prim’s Algorithm

Basic Strategy
- Start growing a tree from a designated root vertex
- At each step, add lightest edge linked to $A$ that does not yield cycle

Assign every vertex not in $A$ a key which is at all stages equal to the smallest weight of an edge connecting to $A$

Use a Priority Queue!

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has **key** and **pointer** of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph Diagram]

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph of Prim's Algorithm](image-url)
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph showing Prim's Algorithm](image)

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
**Prim’s Algorithm**

**Implementation**

- Every vertex in \( Q \) has **key** and **pointer** of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key \( \iff \) **safe edge of cut** \( (V \setminus Q, Q) \)
  2. update keys and pointers of its neighbors in \( Q \)

---

[Diagram of a graph with labeled edges and vertices showing the process of Prim's Algorithm.]

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key ⇔ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

6.3: Minimum Spanning Tree
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph Example](image)
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has **key** and **pointer** of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!
Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$.
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$.

We computed the same MST as Kruskal, but in a completely different order! Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!
Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!
Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in \( Q \) has key and pointer of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key ⇔ safe edge of cut \( (V \setminus Q, Q) \)
  2. update keys and pointers of its neighbors in \( Q \)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in \( Q \) has key and pointer of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key ⇔ safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

![Diagram of Prim's Algorithm](image-url)
Prim’s Algorithm

Implementation

- Every vertex in \( Q \) has key and pointer of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key \(\Leftrightarrow\) safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph diagram]

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

[Diagram of a weighted graph with vertices and edges labeled with weights. The explanation matches the diagram.]

---

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given implicitly by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

**Implementation**

- Every vertex in \( Q \) has **key** and **pointer** of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key ⇔ **safe edge** of cut \( (V \setminus Q, Q) \)
  2. update keys and pointers of its neighbors in \( Q \)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Diagram of a graph with Prim's algorithm applied](image-url)
Prim’s Algorithm

**Implementation**

- Every vertex in $Q$ has **key** and **pointer** of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

---

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\Leftrightarrow$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

![Graph representation of Prim's Algorithm](image)
Prim’s Algorithm

Implementation

- Every vertex in \( Q \) has key and pointer of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key ⇔ safe edge of cut \((V \setminus Q, Q)\)
  2. update keys and pointers of its neighbors in \( Q \)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

Diagram of graph with weights and connections.
Prim’s Algorithm

Implementation

- Every vertex in \( Q \) has key and pointer of least-weight edge to \( V \setminus Q \)
- At each step:
  1. extract vertex from \( Q \) with smallest key ⇔ safe edge of cut (\( V \setminus Q, Q \))
  2. update keys and pointers of its neighbors in \( Q \)

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$
Prim’s Algorithm

Implementation

- Every vertex in $Q$ has key and pointer of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ safe edge of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed the same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!
Prim’s Algorithm

### Implementation

- Every vertex in $Q$ has **key** and **pointer** of least-weight edge to $V \setminus Q$
- At each step:
  1. extract vertex from $Q$ with smallest key $\iff$ **safe edge** of cut $(V \setminus Q, Q)$
  2. update keys and pointers of its neighbors in $Q$

We computed same MST as Kruskal, but in a completely different order!

Final MST is given (implicitly) by the pointers!

6.3: Minimum Spanning Tree
Details of Prim’s Algorithm

```python
0: def prim(G,r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:     
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:     Q.insert(v)
13:     
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u,v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:                 Q.decreaseKey(item=v, newKey=w)
```

---

**Time Complexity**

<table>
<thead>
<tr>
<th>Method</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prim’s Algorithm</td>
<td>(O(V \log V + E))</td>
</tr>
<tr>
<td>Fibonacci Heaps</td>
<td>(O(V))</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>(O(V \cdot \log V))</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>(O(E \cdot 1))</td>
</tr>
</tbody>
</table>

Fibonacci Heaps:
- Init (l. 6-13): \(O(V)\),
- ExtractMin (15): \(O(V \cdot \log V)\),
- DecreaseKey (16-20): \(O(E \cdot 1)\),
- Overall: \(O(V \log V + E)\)

Binary/Binomial Heaps:
- Init (l. 6-13): \(O(V)\),
- ExtractMin (15): \(O(V \cdot \log V)\),
- DecreaseKey (16-20): \(O(E \cdot \log V)\),
- Overall: \(O(V \log V + E \log V)\)
Details of Prim’s Algorithm

```python
0: def prim(G, r):
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:         
5:         Q = MinPriorityQueue()
6:         for v in G.vertices():
7:             v.predecessor = None
8:             if v == r:
9:                 v.key = 0
10:            else:
11:                v.key = Infinity
12:         Q.insert(v)
13:     
14:         while not Q.isEmpty():
15:             u = Q.extractMin()
16:             for v in u.adjacent():
17:                 w = G.weightOfEdge(u, v)
18:                 if Q.hasItem(v) and w < v.key:
19:                     v.predecessor = u
20:                     Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- **Fibonacci Heaps**:
  - **Init**: $O(V)$,
  - **ExtractMin**: $O(V \cdot \log V)$,
  - **DecreaseKey**: $O(E \cdot 1)$,
  - **Overall**: $O(V \log V + E)$

- **Binary/Binomial Heaps**:
  - **Init**: $O(V)$,
  - **ExtractMin**: $O(V \cdot \log V)$,
  - **DecreaseKey**: $O(E \cdot \log V)$,
  - **Overall**: $O(V \log V + E \log V)$
Details of Prim’s Algorithm

```
0:   def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:       MST induced by the .predecessor fields
4:
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:       v.predecessor = None
8:       if v == r:
9:         v.key = 0
10:      else:
11:         v.key = Infinity
12:     Q.insert(v)
13:
14:   while not Q.isEmpty():
15:     u = Q.extractMin()
16:     for v in u.adjacent():
17:       w = G.weightOfEdge(u, v)
18:       if Q.hasItem(v) and w < v.key:
19:         v.predecessor = u
20:     Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- **Fibonacci Heaps:**

  - Init (l. 6-13): $O(V)$
  - ExtractMin (15): $O(V \cdot \log V)$
  - DecreaseKey (16-20): $O(E)$
  - Overall: $O(V \log V + E)$

- **Binary/Binomial Heaps:**

  - Init (l. 6-13): $O(V)$
  - ExtractMin (15): $O(V \cdot \log V)$
  - DecreaseKey (16-20): $O(E \cdot \log V)$
  - Overall: $O(V \log V + E \log V)$
Details of Prim’s Algorithm

```python
0: def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:     
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:     Q.insert(v)
13:     
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u, v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:                 Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- **Fibonacci Heaps:**
  - Init (l. 6-13): $O(V)$,
Details of Prim’s Algorithm

```python
0: def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:     
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:        Q.insert(v)
13:     
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u, v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:                 Q.decreaseKey(item=v, newKey=w)
```

**Time Complexity**

- **Fibonacci Heaps:**
  - **Init** (l. 6-13): $\mathcal{O}(V)$,
Details of Prim’s Algorithm

```python
def prim(G, r):
    Apply Prim’s Algorithm to graph G and root r
    Return result implicitly by modifying G:
    MST induced by the .predecessor fields

Q = MinPriorityQueue()
for v in G.vertices():
    v.predecessor = None
    if v == r:
        v.key = 0
    else:
        v.key = Infinity
Q.insert(v)

while not Q.isEmpty():
    u = Q.extractMin()
    for v in u.adjacent():
        w = G.weightOfEdge(u, v)
        if Q.hasItem(v) and w < v.key:
            v.predecessor = u
    Q.decreaseKey(item=v, newKey=w)
```

**Time Complexity**

- **Fibonacci Heaps:**
  - **Init** (l. 6-13): $O(V)$,
  - **ExtractMin** (15): $O(V \cdot \log V)$,
Details of Prim’s Algorithm

```python
0: def prim(G, r):
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:     Q.insert(v)
13:
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u, v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Fibonacci Heaps:
- **Init (l. 6-13):** $\mathcal{O}(V)$,
- **ExtractMin (15):** $\mathcal{O}(V \cdot \log V)$,
- **DecreaseKey (16-20):** $\mathcal{O}(E \cdot 1)$,
  \[ \Rightarrow \] Overall: $\mathcal{O}(V \log V + E)$

Binary/Binomial Heaps:
- **Init (l. 6-13):** $\mathcal{O}(V)$,
- **ExtractMin (15):** $\mathcal{O}(V \cdot \log V)$,
- **DecreaseKey (16-20):** $\mathcal{O}(E \cdot \log V)$,
  \[ \Rightarrow \] Overall: $\mathcal{O}(V \log V + E \log V)$
Details of Prim’s Algorithm

```python
0: def prim(G, r)
1:   Apply Prim’s Algorithm to graph G and root r
2:   Return result implicitly by modifying G:
3:   MST induced by the .predecessor fields
4:
5:   Q = MinPriorityQueue()
6: for v in G.vertices():
7:     v.predecessor = None
8:   if v == r:
9:     v.key = 0
10:   else:
11:     v.key = Infinity
12:   Q.insert(v)
13:
14: while not Q.isEmpty():
15:   u = Q.extractMin()
16:   for v in u.adjacent():
17:     w = G.weightOfEdge(u,v)
18:     if Q.hasItem(v) and w < v.key:
19:       v.predecessor = u
20:       Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- **Fibonacci Heaps:**
  - Init (l. 6-13): $\mathcal{O}(V)$
  - ExtractMin (15): $\mathcal{O}(V \cdot \log V)$
  - DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
Details of Prim’s Algorithm

```
0: def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:     
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:     Q.insert(v)
13:     
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u, v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:             Q.decreaseKey(item=v, newKey=w)
```

Time Complexity

- **Fibonacci Heaps:**
  - Init (l. 6-13): $\mathcal{O}(V)$,
  - ExtractMin (15): $\mathcal{O}(V \cdot \log V)$,
  - DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$

Amortized Cost
Details of Prim’s Algorithm

```
0: def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:         MST induced by the .predecessor fields
4:     Q = MinPriorityQueue()
5:     for v in G.vertices():
6:         v.predecessor = None
7:         if v == r:
8:             v.key = 0
9:         else:
10:            v.key = Infinity
11:     Q.insert(v)
12:     while not Q.isEmpty():
13:         u = Q.extractMin()
14:         for v in u.adjacent():
15:             w = G.weightOfEdge(u, v)
16:             if Q.hasItem(v) and w < v.key:
17:                 v.predecessor = u
18:                 Q.decreaseKey(item=v, newKey=w)
```

**Time Complexity**

- **Fibonacci Heaps:**
  - **Init** (l. 6-13): $O(V)$, **ExtractMin** (15): $O(V \cdot \log V)$, **DecreaseKey** (16-20): $O(E \cdot 1)$
  - $\Rightarrow$ **Overall:** $O(V \log V + E)$
Details of Prim’s Algorithm

0: def prim(G, r)
1:     Apply Prim’s Algorithm to graph G and root r
2:     Return result implicitly by modifying G:
3:     MST induced by the .predecessor fields
4:     
5:     Q = MinPriorityQueue()
6:     for v in G.vertices():
7:         v.predecessor = None
8:         if v == r:
9:             v.key = 0
10:        else:
11:            v.key = Infinity
12:        Q.insert(v)
13:     
14:     while not Q.isEmpty():
15:         u = Q.extractMin()
16:         for v in u.adjacent():
17:             w = G.weightOfEdge(u, v)
18:             if Q.hasItem(v) and w < v.key:
19:                 v.predecessor = u
20:                 Q.decreaseKey(item=v, newKey=w)

Time Complexity

- **Fibonacci Heaps:**
  - Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot 1)$
  - $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E)$

- **Binary/Binomial Heaps:**
  - Init (l. 6-13): $\mathcal{O}(V)$, ExtractMin (15): $\mathcal{O}(V \cdot \log V)$, DecreaseKey (16-20): $\mathcal{O}(E \cdot \log V)$
  - $\Rightarrow$ Overall: $\mathcal{O}(V \log V + E \log V)$
Summary (Kruskal and Prim)

**Generic Idea**
- Add **safe edge** to the current MST as long as possible
- **Theorem:** An edge is **safe** if it is the lightest of a cut respecting $A$

6.3: Minimum Spanning Tree
Summary (Kruskal and Prim)

Generic Idea
- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting $A$

Kruskal’s Algorithm
- Gradually transforms a forest into a MST by merging trees
- invokes disjoint set data structure
- Runtime $O(E \log V)$
Summary (Kruskal and Prim)

Generic Idea
- Add safe edge to the current MST as long as possible
- Theorem: An edge is safe if it is the lightest of a cut respecting \( A \)

Kruskal’s Algorithm
- Gradually transforms a forest into a MST by merging trees
- Invokes disjoint set data structure
- Runtime \( \mathcal{O}(E \log V) \)

Prim’s Algorithm
- Gradually extends a tree into a MST by adding incident edges
- Invokes Fibonacci heaps (priority queue)
- Runtime \( \mathcal{O}(V \log V + E) \)
Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$
Outlook: Reverse-Delete Algorithm

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Basic Idea

Can be implemented in time $O(E \log V \left(\log \log V\right)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

![Graph with edges and weights](image)

Can be implemented in time $O(E \log V (\log \log V))$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

---

Can be implemented in time $O(E \log V (\log \log V))$. [Thorup, 2000]

---

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

![Graph with labeled edges showing a minimum spanning tree](image)

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V))$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

![Graph Diagram]

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Let $A$ be initially the set of all edges
Consider all edges in decreasing order of their weight
Remove edge from $A$ as long as all vertices are connected by $A$
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V))$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

---

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V))^3$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

**Basic Idea**

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(\text{E} \log \text{V} (\log \log \text{V}))$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V \left(\log \log V \right)^3)$. [Thorup, 2000]

6.3: Minimum Spanning Tree
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

Basic Idea

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Outlook: Reverse-Delete Algorithm

- Let $A$ be initially the set of all edges
- Consider all edges in decreasing order of their weight
- Remove edge from $A$ as long as all vertices are connected by $A$

Can be implemented in time $O(E \log V (\log \log V)^3)$. [Thorup, 2000]
Current State-of-the-Art

Does a linear-time MST algorithm exist?

Randomised MST algorithm with expected runtime $O(E)$ based on Boruvka’s algorithm (from 1926).

Karger, Klein, Tarjan, JACM’1995

Deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$.

Chazelle, JACM’2000

Deterministic MST algorithm with asymptotically optimal runtime however, the runtime itself is not known...

Pettie, Ramachandran, JACM’2002
Current State-of-the-Art

Does a linear-time MST algorithm exist?

- Karger, Klein, Tarjan, JACM’1995
  - randomised MST algorithm with expected runtime $O(E)$
  - based on Boruvka’s algorithm (from 1926)
Current State-of-the-Art

Does a linear-time MST algorithm exist?

- **Karger, Klein, Tarjan, JACM'1995**
  - randomised MST algorithm with expected runtime $O(E)$
  - based on Boruvka’s algorithm (from 1926)

- **Chazelle, JACM’2000**
  - deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$
Current State-of-the-Art

Does a linear-time MST algorithm exist?

- Karger, Klein, Tarjan, JACM’1995
  - randomised MST algorithm with expected runtime $O(E)$
  - based on Boruvka’s algorithm (from 1926)

- Chazelle, JACM’2000
  - deterministic MST algorithm with runtime $O(E \cdot \alpha(n))$

- Pettie, Ramachandran, JACM’2002
  - deterministic MST algorithm with asymptotically optimal runtime
  - however, the runtime itself is not known...