Input:

- An undirected and connected graph $G = (V, E)$ with $n$ vertices and $m$ edges.
- A weight function $w : E \rightarrow \mathbb{R}$ on the edges.
Minimum Spanning Trees (MST)

Input:
- An undirected and connected graph $G = (V, E)$ with $n$ vertices and $m$ edges.
- A weight function $w : E \rightarrow \mathbb{R}$ on the edges.

A spanning tree: A connected sub-graph with $n - 1$ edges.

A minimum spanning tree (MST): A spanning tree for which the sum of the weights of the $n - 1$ edges is minimum.
Minimum Spanning Trees (MST)

- **Input:**
  - An undirected and connected graph \( G = (V, E) \) with \( n \) vertices and \( m \) edges.
  - A weight function \( w : E \to \mathbb{R} \) on the edges.

- **A spanning tree:** A connected sub-graph with \( n - 1 \) edges.

- **A minimum spanning tree (MST):** A spanning tree for which the sum of the weights of the \( n - 1 \) edges is minimum.

- **Two greedy algorithms:**
  - Kruskal - \( O(m \log m) \)-time, a distributed algorithm.
  - Prim - \( O(m + n \log n) \)-time, a centralized algorithm.
Example: A weighted Graph

![Weighted Graph](image-url)
Example: A BFS Spanning Tree

\[ W(T) = 21 \]
Example: A DFS Spanning Tree

\[
W(T) = 14
\]
Example: An MST

\[ W(T) = 13 \]
Example: Another MST

\[ W(T) = 13 \]
A cut $\langle S, (V - S) \rangle$ in a graph is a partition of the set of vertices $V$ into two disjoint sets: $V = S \cup (V - S)$.

An edge $(u, v)$ crosses a cut $\langle S, (V - S) \rangle$ if $(u \in S \& v \in (V - S))$ or $(v \in S \& u \in (V - S))$.

An edge $(u, v)$ is contained in a cut $\langle S, (V - S) \rangle$ if $(u, v) \in S$ or $(u, v) \in (V - S)$.

A set $A$ of edges is contained in a cut $\langle S, (V - S) \rangle$ if all of the edges of $A$ are contained in the cut.
Example: A Cut

An \( \langle \{ABDE\}, \{CF\} \rangle \) cut
Example: Another Cut

An \( \{\{A, E, F\}\}, \{\{B, C, D\}\} \) cut

\[\text{An } \langle \{AEF\}, \{BCD\} \rangle \text{ cut}\]
Minimal Forests

- A set $A$ of edges is a **minimal forest** if it is possible to add edges to $A$ to become a **minimal spanning tree**.

- An **empty set** is in particular a minimal forest.

- A **minimum spanning tree** is in particular a minimal forest.
A Minimal Forest

A weighted graph:

```
A
  
B
  1
  
C
  1
  
D
  1
  
```

Minimal forests:

```
A
  
B
  1
  
C
  1
  
D
  1
  
```
A Minimal Forest

- A weighted graph:

```
   A
  /|
 / | 
B---2---C
  | |
  | 1|
  | A
  |   
  |   
D---1---1
     
```

- Not a minimal forest:

```
   A
   |
   |
B---2---C
   |
   |
D---1
```
The Greedy Lemma

**Input:**

- A weighted connected graph $G = (V, E)$.
- A minimal forest $A \subset E$.
- A cut $\langle S, (V - S) \rangle$ that contains $A$.
- An edge $(u, v)$ crossing $\langle S, (V - S) \rangle$ with minimum weight.

**Lemma:** $A \cup \{(u, v)\}$ is also a minimal forest.
The Greedy Lemma

Input:
- A weighted connected graph $G = (V, E)$.
- A minimal forest $A \subset E$.
- A cut $\langle S, (V - S) \rangle$ that contains $A$.
- An edge $(u, v)$ crossing $\langle S, (V - S) \rangle$ with minimum weight.

Lemma: $A \cup \{(u, v)\}$ is also a minimal forest.
Proof

An MST $T$ that contains $A$ and $(u, v) \not\in T$.

$(x, y) \in T$ crossing $\langle S, (V - S) \rangle \Rightarrow w(u, v) \leq w(x, y)$.

$T' = T - \{(x, y)\} \cup \{(u, v)\} \Rightarrow T'$ is also an MST.

$\Rightarrow A \cup \{(u, v)\}$ is a minimal forest.
A Schematic Greedy Algorithm

(1) \( A = \emptyset \)
(2) while \( |A| < n - 1 \) do
(3) \( \text{find} \ (u, v) \) s.t. \( A \cup \{(u, v)\} \) is a minimal forest
(4) \( A = A \cup \{(u, v)\} \)
(5) return(\(A\))

Correctness: Due to the lemma, step (3) is always possible. By definition, \(A\) is always a minimal forest. At the end \(A\) has \(n - 1\) edges. A forest with \(n - 1\) edges is a tree.

Complexity: Depends on the implementation of step (3).
A Schematic Greedy Algorithm

1. $A = \emptyset$
2. while $|A| < n - 1$ do
3. find $(u, v)$ s.t. $A \cup \{(u, v)\}$ is a minimal forest
4. $A = A \cup \{(u, v)\}$
5. return $(A)$

**Correctness:**
- Due to the lemma, step (3) is always possible.
- By definition, $A$ is always a minimal forest.
- At the end $A$ has $n - 1$ edges.
- A forest with $n - 1$ edges is a tree.
A Schematic Greedy Algorithm

1. \( A = \emptyset \)
2. while \(|A| < n - 1\) do
3. find \((u, v)\) s.t. \(A \cup \{(u, v)\}\) is a minimal forest
4. \(A = A \cup \{(u, v)\}\)
5. return \(A\)

Correctness:
- Due to the lemma, step (3) is always possible.
- By definition, \(A\) is always a minimal forest.
- At the end \(A\) has \(n - 1\) edges.
- A forest with \(n - 1\) edges is a tree.

Complexity: Depends on the implementation of step (3).
Kruskal Algorithm

- Sort the edges from the lightest to the heaviest.
- Consider the edges following this order.
- Start with an empty minimal forest.
- Add an edge to the minimal forest if it doesn’t close a cycle.
- Terminate when there are $n - 1$ edges in the minimal forest.
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm

Minimum Spanning Trees
Example: Kruskal Algorithm

Minimum Spanning Trees
Example: Kruskal Algorithm

Minimum Spanning Trees
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Kruskal Algorithm – Data Structure

- A collection $S$ of disjoint sets of vertices: $\{S_1, S_2, \ldots, S_k\}$
  - $S_i \cap S_j = \emptyset$ for all $1 \leq i \neq j \leq k$.
  - Initially, the collection is empty: $S = \emptyset$.
  - At the end $S$ contains one set of all the vertices: $S = \{V\}$.

- **Make-Set($x$)**: Creates a new set containing only $x$ and adds it to
  the collection $S$: $S = S \cup \{\{x\}\}$.

- **Find-Set($x$)**: Finds the set in the collection $S$ that contains $x$:
  $S_i \in S$ such that $x \in S_i$.

- **Union($S_i, S_j$)**: replaces in the collection $S$ the two sets $S_i$ and $S_j$
  with their union: $S = S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$. 

Amotz Bar-Noy (CUNY)
Variables:
- $A$: A minimal forest.
- Each tree in $A$ is represented by a set of vertices.

Algorithm:
1. $A = \emptyset$
2. for all $v \in V$ do Make-Set($v$)
3. Sort($E$) all edges from $\text{min}$ to $\text{max}$
4. for each edge $(u, v)$ in sorted order do
5. if Find-Set($u$)$ \neq $Find-Set($v$) then
6. $A = A \cup \{(u, v)\}$
7. Union(Find-Set($u$), Find-Set($v$))
8. return (A)
Assume to the contrary that the output set $A$ is not an MST.

Let $(u, v)$ be the first edge that was added to $A$ such that $A \cup \{(u, v)\}$ is not a minimal forest.

In particular, at that time, $A$ is a minimal forest.

Let $C = \langle S, (V - S) \rangle$ be a cut for the set $u \in S$.

$(u, v)$ crosses $C$ since $v$ belongs to another set.
Kruskal Algorithm – Correctness

Any lighter edge \((x, y)\) is contained in \(C\):

- If \((x, y) \notin A\), then both \(x\) and \(y\) belong to the same set before \((x, y)\) was examined.
- If \((x, y) \in A\), then both \(x\) and \(y\) belong to the same set after \((x, y)\) was examined.

Sorting \(\Rightarrow (u, v)\) is a minimal crossing edge for the cut \(C\).

Greedy lemma \(\Rightarrow A \cup \{(u, v)\}\) is a minimal forest.

A contradiction.
Kruskal Algorithm – Complexity

- **Sorting complexity:** \( O(m \log m) \).

- **Set operations complexity:**
  - There are \( n \) Make-Set operations.
  - There are \( O(m) \) Find-Set operations.
  - There are \( n - 1 \) Union operations.
  - Possible to implement in time: \( O(m \cdot \alpha(m, n)) \).
    - \( \alpha(m, n) \) is the inverse of the Ackerman’s function.
    - The \( \alpha(m, n) \) function grows very slowly.
    - For example, \( m, n \approx 10^{80} \Rightarrow \alpha(m, n) \leq 4 \).

- **Overall complexity:** \( O(m \log m) \).
The Ackerman’s Function

\[ A_k(n) = \begin{cases} 
2n & \text{for } k = 0 \text{ and } n \geq 0 \\
A_{k-1}(1) & \text{for } k \geq 1 \text{ and } n = 1 \\
A_{k-1}(A_k(n - 1)) & \text{for } k \geq 1 \text{ and } n \geq 2 
\end{cases} \]
The Ackerman’s Function

\[ A_k(n) = \begin{cases} 
2n & \text{for } k = 0 \text{ and } n \geq 0 \\
A_{k-1}(1) & \text{for } k \geq 1 \text{ and } n = 1 \\
A_{k-1}(A_k(n - 1)) & \text{for } k \geq 1 \text{ and } n \geq 2 
\end{cases} \]

- \( A_0(n) = 2 + \cdots + 2 = 2n \) – The \textbf{multiply-by-2} function.
- \( A_1(n) = 2 \times \cdots \times 2 = 2^n \) – The \textbf{power-of-2} function.
- \( A_2(n) = 2^2 \cdot 2^2 \cdots 2^2 \) – With \( n \) 2’s, the \textbf{tower-of-2} function.
- Each recursive level does the previous level’s operation \( n \) times.
The Ackerman’s Function

\[ A_k(n) = \begin{cases} 
2n & \text{for } k = 0 \text{ and } n \geq 0 \\
A_{k-1}(1) & \text{for } k \geq 1 \text{ and } n = 1 \\
A_{k-1}(A_k(n - 1)) & \text{for } k \geq 1 \text{ and } n \geq 2
\end{cases} \]

- \( A_0(n) = 2 + \cdots + 2 = 2n \) – The multiply-by-2 function.
- \( A_1(n) = 2 \times \cdots \times 2 = 2^n \) – The power-of-2 function.
- \( A_2(n) = 2^{2^{\cdots^2}} \) – With \( n \) 2’s, the tower-of-2 function.

Each recursive level does the previous level’s operation \( n \) times.

- \( A_2(4) = 2^{2^{2^2}} = 2^{16} = 65536. \)
- Already \( A_3(4) \) and \( A_4(4) \) must be extremely large!
Very Slow Growing Functions

- \( \log^* n \) – the inverse of the **tower-of-2** function – is the least \( x \) such that \( 2^{2^{\ldots^{2^x}}} \) \( x \) times is greater or equal to \( n \).

- For example, \( \log^* (2^{65536}) = 5 \).

- \( \alpha(n) \) – the inverse Ackerman’s function – is the least \( x \) such that \( A_x(x) \geq n \).

- \( \alpha(n) \) is **much** slower than \( \log^* n \).
A set is represented by the following **linked list**:
A set is represented by the following **linked list**: 

![Diagram of a linked list with set names and set sizes]
The head of the set contains two fields: the name and the size of the set.

The head of the set has two pointers: to the head and the tail of a linked list of vertices.

An array of $n$ vertices each has two pointers: to the head of its set and to the next vertex in its linked list.
**Kruskal Algorithm – A Simple Implementation**

- **Make-Set**(x) ⇒ $O(1)$ complexity.
Find-Set($x$) $\Rightarrow O(1)$ complexity.
**Kruskal Algorithm – A Simple Implementation**

- **Union**($R, S$) $\Rightarrow O(s)$ complexity.
The **Union** Operation Implementation

- **Worst case complexity:**
  - Connect the larger set to the smaller set.
  - Consider the $n - 1$ **Union** operations: $\text{Union}(S_2, S_1), \text{Union}(S_3, S_2), \ldots, \text{Union}(S_n, S_{n-1})$
  - The cost of $\text{Union}(S_{i+1}, S_i)$ is $\Omega(i)$.
  - Total cost for the **Union** operations: $\Omega(1) + \Omega(2) + \cdots + \Omega(n - 1) = \Omega(n^2)$. 
The **Union** Operation Implementation

- **Worst case complexity:**
  - Connect the larger set to the smaller set.
  - Consider the \( n - 1 \) **Union** operations: 
    \[
    \text{Union}(S_2, S_1), \text{Union}(S_3, S_2), \ldots, \text{Union}(S_n, S_{n-1})
    \]
  - The cost of **Union** \((S_{i+1}, S_i)\) is \(\Omega(i)\).
  - Total cost for the **Union** operations: 
    \[
    \Omega(1) + \Omega(2) + \cdots + \Omega(n - 1) = \Omega(n^2).
    \]

- **Modification:**
  - Connect the smaller set to the larger set.
  - The pointer of each vertex is changed at most \(\log n\) times, since after each **Union** operation the pointer points to a set whose size is at least twice the size of the previous set.
  - All together, for the \( n - 1 \) **Union** operations, for all vertices, \(O(n \log n)\) complexity.
Implementation Complexity

- $n - 1$ **Union** operations: $O(n \log n)$.

- $n$ **Make-Set** operations: $n \cdot \Theta(1) = \Theta(n)$.

- $O(m)$ **Find-Set** operations: $O(m) \cdot \Theta(1) = \Theta(m)$.

- Sorting complexity: $\Theta(m \log m) = \Theta(m \log n)$. 

Kruskal algorithm complexity: $\Theta(m \log m)$. 

Amotz Bar-Noy (CUNY)
Implementation Complexity

- $n - 1$ **Union** operations: $O(n \log n)$.

- $n$ **Make-Set** operations: $n \cdot \Theta(1) = \Theta(n)$.

- $O(m)$ **Find-Set** operations: $O(m) \cdot \Theta(1) = \Theta(m)$.

- Sorting complexity: $\Theta(m \log m) = \Theta(m \log n)$.

**Kruskal algorithm complexity**: $\Theta(m \log m)$. 
Prim Algorithm

- Start with an arbitrary vertex as a singleton tree.
- Maintain a minimal forest initially set to be this tree.
- Find the closest vertex to the minimal forest.
- Add this vertex and its closest edge to the minimal forest.
- Continue until all vertices are in the minimal forest.
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm

The diagram illustrates a graph with weighted edges representing a Minimum Spanning Tree. The algorithm starts with an initial node and iteratively adds the lowest-weight edge that connects a new node to the growing tree, ensuring that no cycles are formed.

Key steps in Prim's Algorithm:
1. Select an arbitrary node as the starting point.
2. Find the edge with the lowest weight that connects a node in the tree to a node outside the tree.
3. Add this edge and the new node to the tree.
4. Repeat steps 2 and 3 until all nodes are included in the tree.

The graph shown includes nodes A, B, C, D, E, F, G, H, and I, with weighted edges connecting them.
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Prim Algorithm – Implementing the Greedy Idea

Data structure:
- A minimal tree (forest) $A$ that will be the MST.
- A starting vertex $r$ that is the root of the MST.
- A priority queue $Q$ for vertices not yet in $A$.
- A distance key $\text{key}(v)$ from $A$ for vertices in $Q$.
- A candidate edge $(v, \Pi(v))$ for any vertex $v$ in $Q$. 

The greedy idea:
Repeat adding to $A$ the edge $(u, \Pi(u))$ for the vertex $u$ that has the minimum value for $\text{key}(\cdot)$ in $Q$.
Update, if necessary, the distance key $\text{key}(v)$ and the candidate edge $(v, \Pi(v))$ for each neighbor $v$ of $u$ that is still in $Q$. 

Amotz Bar-Noy (CUNY)
Prim Algorithm – Implementing the Greedy Idea

**Data structure:**
- A minimal tree (forest) $A$ that will be the MST.
- A starting vertex $r$ that is the root of the MST.
- A priority queue $Q$ for vertices not yet in $A$.
- A distance $key(v)$ from $A$ for vertices in $Q$.
- A candidate edge $(v, \Pi(v))$ for any vertex $v$ in $Q$.

**The greedy idea:**
- **Repeat** adding to $A$ the edge $(u, \Pi(u))$ for the vertex $u$ that has the minimum value for $key(\cdot)$ in $Q$.
- **Update**, if necessary, the distance $key(v)$ and the candidate edge $(v, \Pi(v))$ for each neighbor $v$ of $u$ that is still in $Q$. 
Prim Algorithm – Code

(1) initialize $\Pi(r) = \text{nil}; \ A = \emptyset; \ Q = V - \{r\}$
(2) for all $u \in Q$ do $\text{key}(u) = \infty$
(3) $u = r$
(4) Repeat
(5) for each neighbor $v$ of $u$ do
(6) if ($v \in Q$) and $w(u, v) < \text{key}(v)$ then
(7) $\text{key}(v) = w(u, v); \ \Pi(v) = u$
(8) $u = \text{Extract-Min}(Q)$
(9) $A = A \cup \{(u, \Pi(u))\}$
(10) until $Q = \emptyset$
(11) return ($A$)
Assume to the contrary that the output set $A$ is not an MST.

Let $(u, v)$ be the first edge that was added to $A$ such that $A \cup \{(u, v)\}$ is not a minimal forest.

In particular, at that time, $A$ is a minimal forest.

Let $C = \langle Q, (V - Q) \rangle$ be a cut.
Assume to the contrary that the output set $A$ is not an MST.

Let $(u, v)$ be the first edge that was added to $A$ such that $A \cup \{(u, v)\}$ is not a minimal forest.

In particular, at that time, $A$ is a minimal forest.

Let $\mathcal{C} = \langle Q, (V - Q) \rangle$ be a cut.

The algorithm guarantees that $\mathcal{C}$ contains $A$.

The priority queue implies that $(u, v)$ is a minimal crossing edge for the cut $\mathcal{C}$.

Greedy lemma $\Rightarrow A \cup \{(u, v)\}$ is a minimal forest.

A contradiction.
Queue operations:
- One time building a priority queue.
- $n - 1$ times the operation Extract-Min.
- At most $m$ times updating a value of a key.
Queue operations:
- One time building a priority queue.
- \( n - 1 \) times the operation Extract-Min.
- At most \( m \) times updating a value of a key.

An implementation with an unsorted array:
- \( \Theta(n) \) to build a queue.
- \( \Theta(n) \) for the Extract-Min operation.
- \( \Theta(1) \) for the Update-Queue operation.

Unsorted array total complexity:
\[
1 \times \Theta(n) + (n - 1) \times \Theta(n) + \Theta(m) \times \Theta(1) = \Theta(n^2)
\]
Queue operations:
- One time building a priority queue.
- \( n - 1 \) times the operation Extract-Min.
- At most \( m \) times updating a value of a key.

An implementation with a sorted array:
- \( \Theta(n) \) to build a queue.
- \( \Theta(1) \) for the Extract-Min operation.
- \( \Theta(n) \) for the Update-Queue operation.

Sorted array total complexity:
- \( 1 \times \Theta(n) + (n - 1) \times \Theta(1) + \Theta(m) \times \Theta(n) = \Theta(nm) \)
Queue operations:
- One time building a priority queue.
- \( n - 1 \) times the operation Extract-Min.
- At most \( m \) times updating a value of a key.

An implementation with a heap:
- \( \Theta(n) \) to build a queue.
- \( \Theta(\log n) \) for the Extract-Min operation.
- \( \Theta(\log n) \) for the Update-Queue operation.

Heap total complexity:
\[
1 \times \Theta(n) + (n - 1) \times \Theta(\log n) + \Theta(m) \times \Theta(\log n) = \Theta(m \log n)
\]
Queue operations:
- One time **building** a priority queue.
- $n - 1$ times the operation **Extract-Min**.
- At most $m$ times **updating** a value of a key.

An implementation with a Fibonacci heap:
- $\Theta(n)$ to **build** a queue.
- $\Theta(\log n)$ for the **Extract-Min** operation.
- $\Theta(1)$ (**amortized**) for the **Update-Queue** operation.

Fibonacci heap total complexity:
- $1 \times \Theta(n) + (n - 1) \times \Theta(\log n) + \Theta(m) \times \Theta(1) = \Theta(m + n \log n)$
Kruskal vs. Prim

Complexity:

- Kruskal is an $O(m \log m)$ algorithm.
- Prim is an $O(m + n \log n)$ algorithm.
Kruskal vs. Prim

**Complexity:**
- Kruskal is an $O(m \log m)$ algorithm.
- Prim is an $O(m + n \log n)$ algorithm.

**Implementation:**
- Kruskal is a **distributed** algorithm.
- Prim is a **centralized** algorithm.