Outline

1. Introduction
2. Kruskal Algorithm
3. Prim Algorithm
4. Conclusions
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.

Definitions
- 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.

Output
- A Minimum Spanning Tree of $G$.

Two greedy algorithms
- Kruskal - $\Theta(m \log m)$-time, a distributed algorithm.
- Prim - $\Theta(m + n \log n)$-time, a centralized algorithm.
Example: A weighted Graph
Example: A BFS Spanning Tree

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

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

$W(T) = 13$
Example: Another Optimal MST

\[ W(T) = 13 \]
Kruskal Algorithm

High level description

- Sort the edges from the lightest to the heaviest.
- Start with a forest of $n$ singleton trees.
- Consider the edges following this order.
- If the current edge closes a cycle in one of the trees in the forest, ignore it.
- Otherwise, add the edge to the forest.
- Each added edge combines two trees in the forest and reduces the number of trees in the forest by one.
- Terminate when there are $n - 1$ edges or equivalently when the forest has only one tree.
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm

Kruskal Algorithm

A
B
H
C
E
D
I
G
F

4
8
8 7
9
10
21
7 6
4
14
2
11

Amotz Bar-Noy (CUNY)
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
Example: Kruskal Algorithm
### Variables

- 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 $S = \{V\}$ contains one set which is the set of all vertices.

### Set Operations

- **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: $S = S - \{S_i, S_j\} \cup \{S_i \cup S_j\}$.
Kruskal Algorithm – Implementation

The forest
- $\mathcal{F}$: A forest of trees.
- Each tree in $\mathcal{F}$ is represented by a set of vertices.

Algorithm

1. $\mathcal{F} = \emptyset$
2. for each $v \in V$ 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 $\text{Find-Set}(u) \neq \text{Find-Set}(v)$ then
6. $\mathcal{F} = \mathcal{F} \cup \{(u, v)\}$
7. Union($\text{Find-Set}(u), \text{Find-Set}(v)$)
8. return ($\mathcal{F}$)
Outline

Prove by induction that at any stage of the algorithm the following two properties hold:

(I) Each tree in the forest is an MST for its vertices.
(II) The forest can be extended to be an MST of the whole graph with the already edges in it.

Initially, it is true:

(I) The forest contains only singleton trees and a singleton tree is trivially an MST.
(II) A collection of singleton trees can be extended to any of the MSTs of the graph.

Crux of the proof: Show that combing two trees with the minimum available weight edge preserves the two properties.

* Relatively easier if all the weights are distinct.

At the end, the forest contains one tree which is the minimum spanning tree of the whole graph.
Kruskal Algorithm – Complexity

Sorting complexity
- $\Theta(m \log m)$.

Set operations complexity
- There are $n$ Make-Set operations.
- There are $\Theta(m)$ Find-Set operations in the worst case.
  - Assuming the graph is represented by adjacency lists.
- There are $n - 1$ Union operations.
- Possible to implement in time: $\Theta(m \cdot \alpha(m, n))$.
  - $\alpha(m, n)$ grows extremely slowly.
  - For example, $m, n \approx 10^{80} \Rightarrow \alpha(m, n) \leq 4$.

Overall complexity
- $\Theta(m \log m) = \Theta(m \log n)$. 
The Ackerman’s Function

One of the definitions

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

Small \( k \)

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

Examples

- \( A_2(4) = 2^{2^{2^2}} = 2^{16} = 65536. \)
- 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 
  \[ 2^2 \cdot 2 \cdot \cdots \cdot 2^x \]  
  \( x \) 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 \( \log^* n \).
Sets

- A set is represented by the following **linked list**:

```
<table>
<thead>
<tr>
<th>Set Name</th>
<th>Set Size</th>
</tr>
</thead>
</table>
```

Links

- 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($x$) $\implies$ $\Theta(1)$ complexity.
Kruskal Algorithm – A Simple Implementation

\[ \text{Find-Set}(x) \quad \rightarrow \quad \Theta(1) \text{ complexity.} \]
Kruskal Algorithm – A Simple Implementation

- **Union**($R, S$) $\Rightarrow \Theta(s)$ 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 **Union**($S_{i+1}, S_i$) is $\Omega(i)$.
- Total cost for all the **Union** operations: 
  \[ \Omega(1) + \Omega(2) + \cdots + \Omega(n - 1) = \Omega(n^2). \]

**Modification**
- Always, 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, $\Theta(n \log n)$ complexity.
Simple Implementation Complexity

**Set operations**
- $n - 1$ **Union** operations: $\Theta(n \log n)$.
- $n$ **Make-Set** operations: $n \cdot \Theta(1) = \Theta(n)$.
- $\Theta(m)$ **Find-Set** operations: $\Theta(m) \cdot \Theta(1) = \Theta(m)$.
- Sorting complexity: $\Theta(m \log m)$.

**Kruskal algorithm complexity**
- Overall complexity $\Theta(m \log m) = \Theta(m \log n)$. 
Prim Algorithm

High level description

- Start with an arbitrary vertex as a singleton tree.
- Grow this tree in \( n - 1 \) rounds.
- The distance between a vertex and a tree is the shortest distance between this vertex and any of the vertices in the tree.
- In each round, find the closest vertex to the tree.
- Add to the tree this vertex and its closest edge to the tree.
- Continue until all vertices are in the tree.
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm

Prim Algorithm

MST

Fall 2020
Example: Prim Algorithm
Example: Prim Algorithm

Graph with edges and weights: A-B (4), A-H (8), B-C (8), C-D (7), C-E (9), E-F (10), F-G (2), G-I (6), I-H (7), H-D (11), D-B (8).
Example: Prim Algorithm
Example: Prim Algorithm
Example: Prim Algorithm
Prim Algorithm – Data Structure

**Variables**

- $\mathcal{T}$: A tree that will be the MST.
- $r$: A starting vertex that is the root of $\mathcal{T}$.
- $(u, \Pi(u))$: A candidate edge from $u \in Q$ to a potential parent $\Pi(u) \in \mathcal{T}$.
- $\text{dist}_\mathcal{T}(u)$: For $u \in Q$, the weight of the edge $(u, \Pi(u))$.
- $Q$: A priority queue among all $\text{dist}_\mathcal{T}(u)$ for $u \notin \mathcal{T}$.

**The greedy idea**

- **Repeat** adding to $\mathcal{T}$ the edge $(u, \Pi(u))$ for the vertex $u$ that has the minimum value for $\text{dist}_\mathcal{T}$ in $Q$.
- **Update**, if necessary, the distance $\text{dist}_\mathcal{T}(v)$ and the candidate edge $(v, \Pi(v))$ for each neighbor $v \in Q$ of $u$. 
Prim Algorithm – Implementation

(01) $T = \{r\}$

(02) $\Pi(r) = \textit{nil}$

(03) for all $v \neq r$ do $\text{dist}_T(v) = \infty$

(04) Build-Queue($Q$, $V - \{r\}$)

(05) $u = r$

(06) repeat

(07) for each neighbor $v$ of $u$ do

(08) if ($v \in Q$) and $w(u, v) < \text{dist}_T(v)$ then

(09) $\text{dist}_T(v) = w(u, v)$

(10) Update($Q, \text{dist}_T(v)$)

(11) $\Pi(v) = u$

(12) $u =$ Extract-Min($Q$)

(13) $T = T \cup \{(u, \Pi(u))\}$

(14) until $Q = \emptyset$

(15) return ($T$)
Prim Algorithm – Correctness

Outline

- Prove by induction that at any stage of the algorithm the following two properties hold:
  1. $\mathcal{T}$ is an MST for its vertices.
  2. $\mathcal{T}$ can be extended to be an MST of the whole graph with the already edges in it.

- Initially, it is true:
  1. $\mathcal{T}$ is a singleton tree which is trivially an MST.
  2. A singleton tree can be extended to any of the MSTs of the graph.

- **Crux of the proof:** Show that after adding a vertex to the tree the two properties are preserved.
  * Relatively **easier** if all the weights are **distinct**.

- At the end, $\mathcal{T}$ spans all the vertices and therefore it is the minimum spanning of the whole graph.
Queue operations
- One time **building** a priority queue.
- \( n - 1 \) times the operation **Extract-Min**.
- At most \( m \) times **updating** a value of some \( dist_T(v) \).

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
- \( 1 \times \Theta(n) + (n - 1) \times \Theta(n) + \Theta(m) \times \Theta(1) = \Theta(n^2) \)
Prim Algorithm – Complexity II

Queue operations
- One time **building** a priority queue.
- \( n - 1 \) times the operation **Extract-Min**.
- At most \( m \) times **updating** a value of some \( \text{dist}_T(v) \).

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 some \( \text{dist}_T(v) \).

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

Binary 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 some \( dist_T(v) \).

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 $\Theta(m \log m)$ algorithm.
- Prim is an $\Theta(m + n \log n)$ algorithm.

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

Remark
- The complexity of Kruskal could be $\Theta(m \alpha(n, m))$ if sorting can be done in linear time.
- The function $\alpha$ grows so slow implying that the complexity is basically $\Theta(m)$ in this case.