

Minimum Spanning Trees

Josh Wilcox (jw14g24@soton.ac.uk)

March 11, 2025

Contents

1	What Are Minimum Spanning Trees	2
2	Kruskal's Algorithm	2
2.1	Checking for a cycle	2
2.1.1	<i>Union-Find Operations</i>	2
2.1.2	<i>Cycle Detection in Kruskal's Algorithm</i>	2
2.1.3	<i>Why This Works</i>	2
2.2	Pseudocode	3
2.3	Time Complexity	3
3	Prim's Algorithm	3
3.1	The process	3
3.2	Pseudocode	3
3.3	Optimizing Edge Selection with Priority Queues	4
3.3.1	<i>Fixing Step 1</i>	4

1 What Are Minimum Spanning Trees

- Spanning Trees
 - A tree $T \subseteq E$ that **Spans** G
 - * Every node is reachable in the graph
 - * Trees have **no cycles**
- Minimum
 - The weight of the whole tree is as little as possible

2 Kruskal's Algorithm

- Consider each edge in the graph, smallest weight first
- Build the MST edge-by-edge
 - Only add an edge if it **does not** create a cycle

2.1 Checking for a cycle

- We can use the **union-find** data structure to efficiently detect cycles
- The union-find data structure maintains a collection of disjoint sets
- For Kruskal's algorithm, each set represents a connected component of our partial MST

2.1.1 Union-Find Operations

Given a graph $G = (V, E)$ and any vertex $v \in V$, the union-find data structure supports three key operations:

- **MAKE-SET(v)**: Creates a new set whose only member is vertex v
- **FIND-SET(v)**: Returns a pointer to the representative of the set containing vertex v
- **UNION(u,v)**: Combines the sets containing vertices u and v into a new set

2.1.2 Cycle Detection in Kruskal's Algorithm

To detect if adding an edge (u, v) would create a cycle:

- Before adding any edges, call **MAKE-SET(v)** for each vertex $v \in V$
- When considering an edge (u, v) :
 - Check if **FIND-SET(u) = FIND-SET(v)**
 - If equal, adding edge (u, v) would create a cycle (reject this edge)
 - If not equal, adding edge (u, v) is safe (add it to the MST)
 - After adding edge (u, v) , call **UNION(u, v)** to merge their components

2.1.3 Why This Works

- Each set in the union-find structure represents a connected component in our partial MST
- If two vertices have the same representative (same **FIND-SET** result), they're already connected
- Adding an edge between already-connected vertices would create a cycle
- When we add a valid edge, we merge the components with **UNION**

2.2 Pseudocode

Algorithm 1 Kruskal's Algorithm

```

 $T \leftarrow \emptyset$ 
for all  $v \in V$  do
    MAKE-SET( $v$ )
end for
 $F \leftarrow \text{SORT-EDGES}(E)$ 
for all  $(u, v) \in F$  do
    if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ ) then
         $T \leftarrow T \cup \{(u, v)\}$ 
        UNION( $u, v$ )
    end if
end for

```

2.3 Time Complexity

3 Prim's Algorithm

- Kruskal's is **Edge-Based** ;)
- Prim's is **Vertex Based**

3.1 The process

- Prim's maintains a **set** S of vertices, initially containing a single vertex s which could be any vertex in V
 - At each iteration consider the sets S and $V - S$
 - Find the edge (u, v) with **minimum weight** such that $u \in S$ and $v \in V - S$
 - Add v to S and (u, v) to T
1. Find $x \in S, y \in V - S$ such that $w(x, y) \leq w(u, v)$ for all $v \in S, u \in V - S$
 2. Remove y from $V - S$ and add it to S . Also add edge (x, y) to T
 3. Repeat until $V - S = \emptyset$

3.2 Pseudocode

Algorithm 2 Prim's Algorithm

```

for all  $v \in V$  do
     $v.\text{key} \leftarrow \infty, v.p \leftarrow \text{NULL}$ 
end for
 $s.\text{key} \leftarrow 0, T \leftarrow \emptyset$ 
 $Q \leftarrow V$ 
while  $Q \neq \emptyset$  do
     $u \leftarrow \text{DELETE-MIN}(Q)$ 
     $T \leftarrow T \cup \{(u, u.p)\}$ 
    for all  $v \in \text{adj}[u]$  do
        if  $w(u, v) < v.\text{key}$  then
             $v.\text{key} \leftarrow w(u, v)$ 
             $v.p \leftarrow u$ 
        end if
    end for
end while

```

3.3 Optimizing Edge Selection with Priority Queues

3.3.1 Fixing Step 1

- Finding the minimum weight edge between sets S and $V - S$ naively is inefficient:
 - The naive approach would require checking all edges (u, v) where $u \in S$ and $v \in V - S$ in each iteration
 - This would require $O(|E|)$ time per iteration, resulting in $O(|V| \cdot |E|)$ overall complexity
- Efficient implementation uses a priority queue to track potential edges:
 - For each vertex $v \in V - S$, maintain a value $v.key$ representing the minimum weight edge connecting v to any vertex in S
 - Initially, set $v.key = \infty$ for all vertices except the starting vertex s , which has $s.key = 0$
 - Also track the source vertex in S that gives this minimum weight using $v.p$
- The algorithm maintains these invariants:
 - For any vertex $v \in V - S$, $v.key$ is the weight of the lightest edge connecting v to some vertex in S
 - When we extract the vertex y with minimum key from $V - S$, we add the edge $(y.p, y)$ to our MST
- After adding a new vertex y to S :
 - We need to update the keys of remaining vertices in $V - S$
 - For each neighbor $v \in adj[y]$ that is still in $V - S$:
 - If $w(y, v) < v.key$, update $v.key = w(y, v)$ and set $v.p = y$
 - This "relaxes" the edge and ensures $v.key$ represents the minimum weight connection to S
- Why this works:
 - When we extract a vertex from the priority queue, we're guaranteed to get the vertex with minimum connection cost to S
 - This follows the greedy choice property of Prim's algorithm
 - The key updates ensure we always have accurate information about the cheapest way to extend S
 - With a binary heap implementation, the overall time complexity becomes $O(|E| \log |V|)$