CS 2500: Algorithms

Lecture 21: Greedy Algorithms: Minimum Spanning Tree

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 29, 2024

Problem

- **Task:**
  - An telephone company wants to lay lines connecting some cities $x_1, x_2, x_3, \ldots x_n$.
  - Let x_{ij} be the cost of laying the line from x_i to x_j.
- **Objective:** Cheapest possible network serving all the towns in question.
  - **Constraint:** Only direct links between towns can be used.
- Problems like this can be modelled using Graphs, and solved using Minimum Spanning Tree Algorithms.

Spanning Tree

Let $G = (N, A)$ be an undirected connected graph. A subgraph $t = (N, A')$ of G is a *spanning tree* of G if and only if t is a tree.

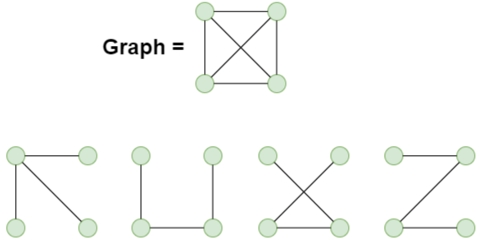


Figure: A graph with four of its spanning trees.

Minimum Spanning Trees

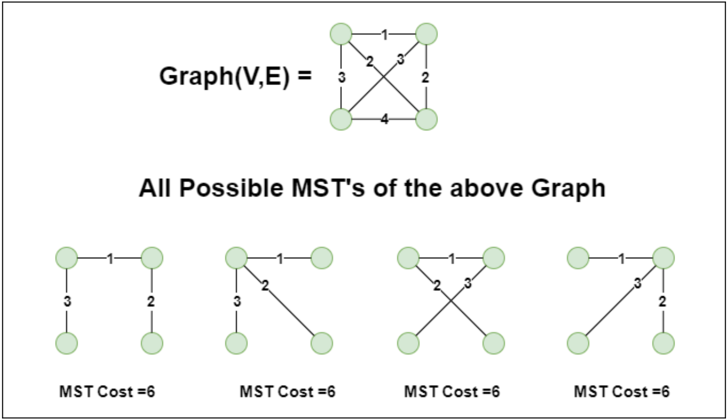
- A mimimum spanning tree is a minimal subgraph G' of G such that $N(G') = N(G)$ and G' is connected.
- We want to find $T \subset A$ such that:
  1. All nodes N in G remain connected when only edges in T are used.
  2. Sum of the cost of the edges in T is as small as possible.

Minimum Spanning Tree

- Let $G' = (N, T)$ be the partial graph formed by the nodes of G and the edges in T, where N has n nodes.
- A connected graph with n nodes must have at least $n - 1$ edges for minimal connectivity.
- Any graph with more than $n - 1$ edges would contain at least one cycle.
- If G' is connected and T has more than $n - 1$ edges, removing an edge in a cycle without disconnecting G' will:
  - Either decrease the total length of the edges in T, or
  - Leave the total length the same while reducing the edge count.
- Thus, a set T with n or more edges cannot be optimal for a spanning tree.
- Conclusion: T must have exactly $n - 1$ edges and, since G' is connected, it must form a tree.

Weighted Graphs:

- In practical situations, edges may have weights representing cost, length, etc.
- The goal is to find a spanning tree with the minimum total cost or length.



Subset Paradigm

Subset Paradigm:

- Finding a minimum spanning tree involves selecting a subset of edges with the minimum sum of weights.
- This selection avoids cycles and ensures connectivity, fitting the *subset paradigm*.

Greedy Approach for Minimum Spanning Tree

Approaches:

- Two main strategies for a greedy algorithm:
  - Start with an empty set T and select the shortest edge that hasn't been chosen or rejected, regardless of position.
  - Choose a node and build a tree from there, selecting the shortest available edge that extends the tree to a new node.
- **Both approaches work for finding a minimum-cost spanning tree**.

Key Concepts:

- **Candidates:** The edges in G.
- **Solution Set:** A set of edges that forms a spanning tree for the nodes in N.
- **Feasibility:** A set of edges is feasible if it does not contain a cycle.
- **Selection Function:** Varies based on the chosen greedy algorithm.
- **Objective:** Minimize the total length of the edges in the solution.

Minimum Spanning Tree: Kruskal's Algorithm

How the Algorithm Works: The Big Idea

- Start with an empty set T.
- While T is not a spanning tree:
  - Examine edges of G in increasing order of length.
  - If an edge connects nodes in different components, add it to T.
  - If an edge connects nodes in the same component, reject it to avoid cycles.
- The algorithm stops when only one connected component remains, forming a minimum spanning tree for all nodes.

Example:

- Consider a graph with nodes connected by edges in increasing order of length: (1, 2), (2, 3), (4, 5), (6, 7), (1, 4), (2, 5), (4, 7), (3, 5), (2, 4), (3, 6), (5, 7), (5, 6)
- Initial connected components: {1}, {2}, {3}, {4}, {5}, {6}, {7}.

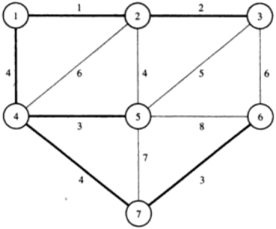


Figure: A graph with its MST.

Minimum Spanning Tree: Kruskal's Algorithm

Step	Edge Considered	Connected Components
Initialization		{1}, {2}, {3}, {4}, {5}, {6}, {7}
1	(1, 2)	{1, 2}, {3}, {4}, {5}, {6}, {7}
2	(2, 3)	{1, 2, 3}, {4}, {5}, {6}, {7}
3	(4, 5)	{1, 2, 3}, {4, 5}, {6}, {7}
4	(6, 7)	{1, 2, 3}, {4, 5}, {6, 7}
5	(1, 4)	{1, 2, 3, 4, 5}, {6, 7}
6	(2, 5)	Rejected
7	(4, 7)	{1, 2, 3, 4, 5, 6, 7}
8	(3, 5)	Rejected
9	(2, 4)	Rejected
10	(3, 6)	Rejected
11	(5, 7)	Rejected
12	(5, 6)	Rejected

Figure: Kruskal's Algorithm on given graph

Result: The minimum spanning tree includes edges $(1,2),(2,3),(4,5),(6,7),(1,4),(4,7)$ with a total length of 17.

Minimum Spanning Tree: Kruskal's Algorithm

How the Algorithm Works: Implementation

Kruskal's Algorithm is implemented using a data structure called **Disjoint Sets** or **Union-Find**.

1. Sort all edges in the graph by weight.
2. Initialize an empty forest, where each vertex is a separate tree.
3. Repeat until the forest has a single tree:
    - Pick the smallest edge. If it connects two different trees, add it to the MST.
    - Use the Union-Find data structure to efficiently check if the edge creates a cycle.

Union-Find Data Structure

- Union-Find is a data structure used to keep track of elements in disjoint sets.
- It supports two primary operations:
  - **Find**: Determine which subset a particular element is in.
  - **Union**: Join two subsets into a single subset.

Union-Find Data Structure

Union-Find Operations:

- **MakeSet(x)**: Create a new set containing x.
- **Find(x)**: Returns the representative of the set containing x.
- **Union(x, y)**: Merge the sets containing x and y.

Rank:

- Estimate of the "depth" of a tree in the Union-Find data structure.
- Helps decide which tree (or set) should become a subtree of the other. The set with the lower rank is attached under the set with the higher rank.

Union-Find Data Structure

Path Compression: Technique to make the `Find` operation faster by flattening the structure of the tree.

- During `Find(x)`, we make each node on the path from `x` to the root point directly to the root.
- This reduces the depth of the tree, so subsequent `Find` operations become faster.

Effect: Path Compression ensures that the tree remains flat, which, combined with Union by Rank, leads to nearly constant time complexity for both `Union` and `Find`.

Union-Find Data Structure

Union by Rank: Heuristic used in the `Union` operation to keep the tree balanced.

- **Rank** represents an estimate of the tree depth for each root.
- When performing `Union(x, y)`:
  - Find the roots of `x` and `y`, say `rootX` and `rootY`.
  - Attach the tree with the lower rank to the tree with the higher rank.
  - If ranks are equal, make one root the parent of the other and increase its rank by 1.
- This approach keeps the trees as shallow as possible, making future `Find` operations faster.

Union-Find Data Structure

Algorithm MakeSet(x)

1: Set $parent(x) \leftarrow x$ ▷ Set the pointer to point to itself
2: Set $rank(x) \leftarrow 0$ ▷ Initialize the rank of x as 0
3: **Return**

Union-Find Data Structure

Algorithm Find(x)

```
if x ≠ parent(x) then
    parent(x) ← Find(parent(x))        ▷ Path compression
end if
Return parent(x)
```

Union-Find Data Structure

Algorithm Union(x, y)

```
rootX ← Find(x)
rootY ← Find(y)
if rootX ≠ rootY then
    if rank(rootX) > rank(rootY) then
        parent(rootY) ← rootX
    else if rank(rootX) < rank(rootY) then
        parent(rootX) ← rootY
    else
        parent(rootY) ← rootX
        rank(rootX) ← rank(rootX) + 1
    end if
end if
```

Minimum Spanning Tree: Kruskal's Algorithm

Algorithm Kruskal(G)

for each vertex $n \in N$ **do**
    MakeSet(n)
end for
Sort edges A in non-decreasing order by weight
$T \leftarrow \emptyset$
for each edge $(u, v) \in A$ in sorted order **do**
    if Find(u) $\neq$ **Find**(v) **then**
        Union(u, v)
        $T \leftarrow T \cup u, v$
    end if
end for
Return T $\triangleright$ T is the MST of G

Step 1: Sorting Edges

- The first step in Kruskal's Algorithm is to sort all edges by weight.
- Sorting a edges takes $\Theta(a \log a)$.
- Since $n - 1 \leq a \leq \frac{n(n-1)}{2}$, this is approximately $\Theta(a \log n)$.

Conclusion

Sorting contributes $\Theta(a \log n)$ to the overall time complexity.

Kruskal's Algorithm: Time Complexity Analysis

Step 2: Initializing Disjoint Sets

- Each node in the graph is initially placed in its own set.
- Using the Union-Find data structure, this initialization step takes $\Theta(n)$ time.

Conclusion

Initializing disjoint sets contributes $\Theta(n)$ to the time complexity.

Kruskal's Algorithm: Time Complexity Analysis

Step 3: Union-Find Operations

- In the Union-Find data structure, `Find` and `Union` operations can vary in time, but their average cost over multiple operations is close to constant.
- How do we say this mathematically?
  - We use the inverse Ackermann function $\alpha(n)$

The Inverse Ackermann Function $\alpha(n)$

- The inverse Ackermann function, $\alpha(n)$, is a very slowly growing function used in theoretical computer science.
- It appears in the analysis of Union-Find with path compression and union by rank.
- $\alpha(n)$ grows so slowly that for any practical input size, $\alpha(n) \leq 5$.

Relevance to Union-Find

With path compression and union by rank, the `Find` and `Union` operations have an amortized complexity of $O(\alpha(n))$, close to constant for real-world applications.

Amortized Analysis: Helps us find the *average* time per operation over a sequence of operations, rather than analyzing the worst-case time for each individual operation.

Kruskal's Algorithm: Time Complexity Analysis

Step 3: Union-Find Operations

- The total time complexity for all `Find` and `Union` operations is $\Theta(2a \cdot \alpha(2a, n))$.
- Here, α is the inverse Ackermann function, which grows very slowly.

Conclusion

Union-Find operations contribute $\Theta(2a \cdot \alpha(2a, n))$, which is very efficient and close to $O(a)$.

Kruskal's Algorithm: Time Complexity Analysis

Remaining Operations

- Additional operations (such as comparisons) contribute at most $\Theta(a)$ to the total complexity.
- This term is insignificant compared to the other terms in the analysis.

Conclusion

The remaining operations do not affect the overall time complexity significantly.

Kruskal's Algorithm: Time Complexity Analysis

Overall Time Complexity

- The total time complexity of Kruskal's Algorithm is dominated by the sorting and Union-Find steps.
- Thus, the **overall time complexity** is:

$$\Theta(a \log n)$$

- Since $\alpha(2a, n)$ is almost constant, it does not significantly impact the complexity.

Kruskal's Algorithm: Proof of Correctness

- **Promising Set:**
  - A feasible set of edges is *promising* if it can be extended to form an optimal solution.
  - The empty set is always promising, as an optimal solution always exists.
  - If a promising set is already a solution, it must be optimal.
- **Edge Leaves a Set:** An edge *leaves* a set of nodes if exactly one end is in the set.
- **Lemma:**
  - Let $G = (N, A)$ be a connected undirected graph where the length of each edge is given.
  - Let $B \subset N$ be a strict subset of the nodes of G.
  - Let $T \subseteq A$ be a promising set of edges such that no edge in T leaves B.
  - Let v be the shortest edge that leaves B (or one of the shortest if ties exist).
  - Then $T \cup \{v\}$ is promising.

Kruskal's Algorithm: Proof of Correctness

Proof Outline (Induction on the Number of Edges in T):

- **Basis:**
  - The empty set is promising because G is connected, and a solution must exist.
- **Induction Step:**
  - Assume T is promising just before adding a new edge $e = \{u, v\}$.
  - The edges in T divide G into two or more connected components; u is in one component and v in another.
  - Let B be the set of nodes in the component containing u.
    - B is a strict subset of the nodes of G.
    - T is promising, with no edge in T leaving B.
    - e is one of the shortest edges leaving B, satisfying Lemma.
  - By Lemma, $T \cup \{e\}$ is also promising.
- **Conclusion:**
  - When the algorithm stops, T is a solution and is promising, hence optimal.

Two main strategies for a greedy algorithm:

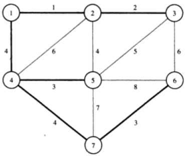
- **Kruskal:** Start with an empty set T and select the shortest edge that hasn't been chosen or rejected, regardless of position.
- **Prim:** Choose a node and build a tree from there, selecting the shortest available edge that extends the tree to a new node.

Minimum Spanning Tree: Prim's Algorithm

Algorithm Prim(G, length)

$T \leftarrow \emptyset$
Choose an arbitrary node u and initialize $B = \{u\}$
while $B \neq N$ **do**
    Find the edge $\{u, v\}$ of minimum length such that $u \in B$ and $v \in N \setminus B$
    Add v to B and $\{u, v\}$ to T
end while
Return T as the minimum spanning tree

Minimum Spanning Tree: Prim's Algorithm



Example (same as Slide 13):

Step	Edge $\{u, v\}$ Considered	B (Nodes in MST)
Initialization		$\{1\}$
1	$\{1, 2\}$	$\{1, 2\}$
2	$\{2, 3\}$	$\{1, 2, 3\}$
3	$\{1, 4\}$	$\{1, 2, 3, 4\}$
4	$\{4, 5\}$	$\{1, 2, 3, 4, 5\}$
5	$\{4, 7\}$	$\{1, 2, 3, 4, 5, 7\}$
6	$\{7, 6\}$	$\{1, 2, 3, 4, 5, 6, 7\}$

Result: The MST includes edges $(1, 2), (2, 3), (1, 4), (4, 5), (4, 7), (7, 6)$.

Quick Assignment 5: Prim's Algorithm: Proof of Correctness

Proof Outline:

- The proof is by mathematical induction on the number of edges in the set T.
- We shall show that if T is promising at any stage of the algorithm, then it is still promising when an extra edge has been added.
- When the algorithm stops, T gives a solution to our problem; since it is also promising, this solution is optimal.
- Similar to how we proved correctness of Kruskal's Algorithm.