Minimum Spanning Trees

1 Minimum Spanning Trees (MSTs)

Given an undirected, connected graph $\mathcal{G}$ with edge costs c_e, find a minimum cost subset $E' \subseteq E$ such that (V, E') is connected. In this example, $\text{cost}(E') = \sum_{e \in E'} c_e$.

Assume without loss of generality that $c_e \geq 0$, because the negative edges (if they exist) can be preprocessed out. For simplicity, assume distinct edge weights. With these conditions, the optimum solution to this problem is always a spanning tree.

2 MST Algorithms

Algorithm 1 Kruskal's Algorithm

```
1: Sort edges by non-decreasing c_e
2: for edge e in E do
3:    if e does not create a cycle in E' then
4:       Add e to E'
5:    else
6:       Discard e
7:    end if
8: end for
```

Algorithm 2 Prim's Algorithm

```
1: Let r be an arbitrary root node, then S = [r]
2: while S ≠ V do
3:    Add the cheapest edge (s, v) crossing from S → S̄
4:    Add v to S
5: end while
```

3 Algorithm Analyses

3.1 Proofs of Correctness

Theorem 1. *Both Kruskal's Algorithm and Prim's Algorithm produce MSTs.*

Proof. This is a 3-part proof; we must ensure that each algorithm creates an optimal solution that (1) is acyclic, (2) is connected, and (3) is minimum cost.

(1). No cycles:

Kruskal's Algorithm: holds trivially.

Prim's Algorithm: the algorithm never adds an edge between two members of S or $\bar{S}$.

(2). Connectedness:

Kruskal's Algorithm: assume for the sake of contradiction that there are at least two connected components at termination time (i.e., no edges cross $(S, \bar{S})$). However $\mathcal{G}$ is connected, so there must be some cheapest edge connecting $(S, \bar{S})$. Thus, Kruskal's would added that edge, resulting in a contradiction.

Prim's Algorithm: if the algorithm terminates with $S \neq V$, then by definition there is no edge crossing $(S, \bar{S})$in $\mathcal{G}$. This means that $\mathcal{G}$ is disconnected, which is a contradiction to our initial condition.

(3). Minimum cost:

Lemma 2. *For every cut $(S, \bar{S})$, the cheapest edge across that cut is in the MST.*

Proof. For the sake of contradiction, let $(S, \bar{S})$ be a cut. Let e be the cheapest edge crossing $(S, \bar{S})$, but e is not in the MST. Adding e to the MST creates a cycle which contains one or more edges e' also crossing $(S, \bar{S})$. But because e is cheapest, $c_e < c_{e'}$. Thus, e would have been in the MST in the first place. □

Corollary 3. *If e is the most expensive edge in a cycle C, then e is not in the MST.*

This corollary suggests the **reverse-delete algorithm**, where one begins with all edges E and removes the most expensive edge in C until the graph is acyclic.

Claim. Every edge included by Kruskal's Algorithm or Prim's Algorithm is cheapest for some cut.

Kruskal's Algorithm: when some edge e is added to the MST, it merges two components S_i and S_j by crossing (S_i, S_j). Because no edge has yet been added across (S_i, S_j), e must be the cheapest edge across that cut.

Prim's Algorithm: holds trivially.

Because every edge included by the algorithms is cheapest for some cut, and for every cut, the cheapest edge across that cut is in the MST, every edge included by the algorithms is in the MST. □

3.2 Runtime

Kruskal's Algorithm:

- $\Theta(n^2)$ via running BFS or DFS n times to find the next cheapest edge
- $\mathcal{O}(m \log m + n \log n)$ via a basic union-find data structure
- $\mathcal{O}(m \log m + n \log^* n) = \mathcal{O}(m \log m)$ via union-find with path compression

Prim's Algorithm:

- $\Theta(n^2)$ via naive implementation
- $\Theta(m \log n)$ via a minheap with priority metric c_e
- $\Theta(m + n \log n)$ via a Fibonacci heap

4 Union-Find Data Structure

A **union-find data structure** returns the set (representing the connected component) that an element e belongs to, and allows these sets to be combined (representing Kruskal's Algorithm selecting an edge which connects two connected components).

Union-find utilizes a forest structure where each e has a size counter and a parent pointer (which, if null, means that e is the root of its tree). Initially, all size counters are equal to one and all parent pointers are null.

Operation	Runtime	Description
union(e, e')	$\mathcal{O}(\log n)$	Find the roots of e and e'; point the smaller root to the larger. Update the size counters to the sum of all elements in the resultant tree.
find(e)	$\mathcal{O}(\log n)$	Follow the parent of e to the root; return the root.

Key step in analysis: how deep is each tree? For each union, the size of the set at least doubles, since the smaller root links to the larger. The size of a particular set can double at most $\log n$ times; thus the operations run in $\mathcal{O}(\log n)$.

These operations can be further optimized by utilizing **path compression**. When running find, after finding the root of e, also point all elements on the path to the root directly to the root. This speeds up all future find operations on those elements, resulting in an amortized runtime of $\mathcal{O}(\alpha(n)) \leq \mathcal{O}(\log^* n)$[1].

5 Open Questions

Is there a deterministic MST algorithm which runs in $\mathcal{O}(m)$? For randomized algorithms, Karger's Algorithm has expected runtime $\mathcal{O}(m)$.

What about if the edges are already sorted? Kruskal's Algorithm is pretty good at $\mathcal{O}(n \log^* n)$ post-sort, but that isn't quite constant.

6 Amortized Analysis

Amortized analysis is the analysis of algorithm runtime over multiple operations that gives better bounds than assuming each operation takes worst-case time. It exploits the fact that oftentimes, expensive operations are mutually exclusive or cannot occur consecutively.

Kitchen sink analogy: let putting a dish in the sink be an $\mathcal{O}(1)$ operation and washing the dishes be an $\mathcal{O}(n)$ operation, where n is the number of dishes in the sink. If I have n kitchen operations, it seems that the worst-case time is $\mathcal{O}(n^2)$ if all the operations were dishwashing, but this is actually impossible. So we can take an amortization over n realistic kitchen operations:

$$\frac{n-1 \text{ operations of } \mathcal{O}(1) + 1 \text{ operation of } \mathcal{O}(n)}{n \text{ total kitchen operations}} = \mathcal{O}(n) \text{ amortized runtime}$$

[1] $\alpha(n)$ is the inverse Ackermann function, $\log^* n$ is the iterated logarithm. Both are bounded by 5 for typical inputs.