Spanning Trees: Efficient Algorithms

Minimum spanning tree (MST)

Input: A connected, undirected graph $G = (V, E)$ with positive edge weights

Output: A subset of edges $E' \subseteq E$ of minimum total weight such that the graph (V, E') is connected

Remark: E' always forms a tree

Properties of Trees

- A tree is an undirected graph that is connected and acyclic
- A tree on n vertices has $n - 1$ edges
- Any connected undirected graph $G(V, E)$ with $|E| = |V| - 1$ is a tree
- An undirected graph is a tree if and only if there is a unique path between any pair of its vertices

Kruskal's algorithm

Idea: repeatedly add the next lightest edge if this doesn't produce a cycle

Prim's algorithm

Idea: repeated attach a new vertex to the current tree by a lightest edge

Cut Property

Let $X \subseteq E$ be a part of a MST of $G(V, E)$, $S \subseteq V$ be such that no edge of X crosses between S and $V - S$, and $e \in E$ be a lightest edge across this partition. Then, $X + \{e\}$ is a part of some MST.

Implementation details of Kruskal's algorithm

- Disjoint sets data structure. Initially, each vertex lies in a separate set
- Each set is the set of vertices of a connected component
- To check whether the current edge $\{u, v\}$ produces a cycle, we check whether u and v belong to the same set

```
Kruskal(G):
for all u in V:
    MakeSet(v)
X <- empty set
sort the edges E by weight
for all {u, v} in E in non-decreasing weight order:
    if Find(u) != Find(v):
        add {u, v} to X
        Union(u, v)
return X
```

Running time

- Sorting edges:
  $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$
- Processing edges:
  $2|E| \cdot T(\text{Find}) + |V| \cdot T(\text{Union}) = O((|E| + |V|) \log |V|) = O(|E| \log |V|)$
- Total running time: $O(|E| \log |V|)$

Implementation details of Prim's algorithm

- X is always a subtree which grows by one edge at each iteration
- we add a lightest edge between a vertex of the tree and a vertex not in the tree
- It is very similar to Dijkstra's algorithm

```
Prim(G):
for all u in V:
    cost[u] <- infinity
    parent[u] <- nil
pick any initial vertex u[0]
cost[u[0]] <- 0
PrioQ <- MakeQueue(V) {priority is cost}
while PrioQ is not empty:
    v <- ExtractMin(PrioQ)
    for all {v, z} in E:
        if z in PrioQ and cost[z] > w(v, z):
            cost[z] <- w(v, z)
            parent[z] <- v
            ChangePriority(PrioQ, z, cost[z])
```

Running time

Using binary heap, the cost is $O((|E| + |V|) \log |V|) = O(|E| \log |V|)$