Minimum Spanning Trees

- Spanning subgraph
  - Subgraph of a graph \( G \) containing all the vertices of \( G \)

- Spanning tree
  - Spanning subgraph that is itself a (free) tree

- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks

Example:

- Graph with cities: ORD, PIT, ATL, DFW, STL, DCA
- Edges with weights: e.g., ORD-PIT = 10, DFW-STL = 8, STL-DCA = 3
Cycle Property:

Let $T$ be a minimum spanning tree of a weighted graph $G$.

Let $e$ be an edge of $G$ that is not in $T$ and let $C$ be the cycle formed by $e$ with $T$.

For every edge $f$ of $C$, $\text{weight}(f) \leq \text{weight}(e)$.

Proof:

By contradiction.

If $\text{weight}(f) > \text{weight}(e)$, we can get a spanning tree of smaller weight by replacing $e$ with $f$.

Partition Property:

Consider a partition of the vertices of $G$ into subsets $U$ and $V$.

Let $e$ be an edge of minimum weight across the partition.

There is a minimum spanning tree of $G$ containing edge $e$.

Proof:

Let $T$ be an MST of $G$.

If $T$ does not contain $e$, consider the cycle $C$ formed by $e$ with $T$ and let $f$ be an edge of $C$ across the partition.

By the cycle property, $\text{weight}(f) \leq \text{weight}(e)$.

Thus, $\text{weight}(f) = \text{weight}(e)$.

We obtain another MST by replacing $f$ with $e$. 

Replacing $f$ with $e$ yields a better spanning tree.

Replacing $f$ with $e$ yields another MST.
Prim-Jarnik’s Algorithm

- Similar to Dijkstra’s algorithm
- We pick an arbitrary vertex $s$ and we grow the MST as a cloud of vertices, starting from $s$
- We store with each vertex $v$ label $d(v)$ representing the smallest weight of an edge connecting $v$ to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex $u$ outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to $u$

Input: A weighted directed graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$

Output: The distance from vertex 1 to every other vertex in $G$

1. $X = \{1\}; \ Y \leftarrow V - \{1\}; \ D[1] \leftarrow 0$
2. for $y \leftarrow 2$ to $n$
   3. if ($y$ is adjacent to 1) { $D[y] \leftarrow \text{length}[1, y]; \ p[y] \leftarrow 1$ }
   4. else $D[y] \leftarrow \infty$
5. for $j \leftarrow 2$ to $n$
6. Let $y \in Y$ be such that $D[y]$ is minimum;
7. $X \leftarrow X \cup \{y\};$ // add vertex $y$ to $X$
8. $Y \leftarrow Y - \{y\};$ //delete vertex $y$ from $Y$
9. for each edge $(y, w)$
10. if ($w \in Y$ and $D[w] + \text{length}[y, w] < D[w]$)
11. { $D[w] \leftarrow D[w] + \text{length}[y, w]; \ p[w] \leftarrow y; \}$
Example

Example (contd.)
Kruskal’s Approach

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge “closest” clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

Kruskal’s Algorithm

```
Algorithm KruskalMST(G):
    Input: A simple connected weighted graph G with n vertices and m edges
    Output: A minimum spanning tree T for G
    for each vertex v in G do
        Define an elementary cluster C(v) ← {v}.
    Let Q be a priority queue storing the edges in G, using edge weights as keys
    T ← ∅   // T will ultimately contain the edges of the MST
    while T has fewer than n − 1 edges do
        (u, v) ← Q.removeMin()
        Let C(v) be the cluster containing v
        Let C(u) be the cluster containing u
        if C(v) ≠ C(u) then
            Add edge (v, u) to T
            Merge C(v) and C(u) into one cluster, that is, union C(v) and C(u)
    return tree T
```
Example of Kruskal’s Algorithm

Example (contd.)
Data Structure for Kruskal’s Algorithm
- The algorithm maintains a forest of trees
- Sort all edges into non-decreasing order
- An edge is accepted if it connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - `find(u)`: return the set storing u
  - `union(A, B)`: replace sets A and B with their union

Implementation with Union-Find
- **Kruskal’s Algorithm**
  - Cluster merges as unions
  - Cluster locations as finds
- **Running time** $O(m \log n)$
  - Sorting: $O(m \log m) = O(m \log n)$
  - Union-Find operations: (practically) $O(n + m)$
Baruvka’s Algorithm

- Like Kruskal’s Algorithm, Baruvka’s algorithm grows many clusters at once and maintains a forest $T$.
- Each iteration of the while loop halves the number of connected components in forest $T$.
- The running time: $O(m \log n)$

**Algorithm** $BaruvkaMST(G)$

$T \leftarrow V$ [just the vertices of $G$]

while $T$ has fewer than $n - 1$ edges do

for each connected component $C$ in $T$ do

Let edge $e$ be the smallest-weight edge from $C$ to another component in $T$

if $e$ is not already in $T$ then

Add edge $e$ to $T$

return $T$

Example of Baruvka’s Algorithm (animated)