Lecture 18: Minimum Spanning Trees
Outline

- Minimum Spanning Trees
- Prim’s algorithm
- Kruskal’s algorithm
Minimum Spanning Trees

- Spanning subgraph is a subgraph of $G$ containing all vertices of $G$
- Spanning tree (for connected graph) is a spanning subgraph that is a free tree
- Unless graph is a tree itself, many spanning trees are possible
- Any tree graph has $n - 1$ edges, where $n$ is number of vertices
- Any connected graph with $n - 1$ edges and $n$ vertices is a tree
- Minimum spanning tree (MST)
  - Spanning tree of a weighted graph with minimum total edge weight
- Applications: communications networks, transportation networks
Prim’s Algorithm: Basic Idea

- Greedily grow MST from some start vertex \( s \)
- Partition vertices into green and blue clouds
  - blue cloud starts with one vertex \( s \)
  - green cloud has all the other vertices
- Iterate until blue cloud has all vertices
  - find smallest weight edge across green/blue partition
  - mark this edge as part of the MST
  - move green endpoint of this edge into blue cloud

At the end of the algorithm

- blue vertices are connected by marked edges
- there are exactly \( n - 1 \) marked edges, so marked edges (and all vertices) form a spanning tree
- Is this a minimum spanning tree, however?
  - YES, will show this later
Prim’s Algorithm: Efficient Implementation

- Directly searching for the smallest edge which crosses the green/blue partition is inefficient
- More efficient approach is similar to Dijkstra’s algorithm
- Store with each vertex v a label \( d[v] \) = smallest weight of an edge connecting v to a vertex in blue cloud

At each iteration

- add to blue cloud vertex u outside blue cloud with smallest \( d[u] \)
- for any vertex w not in the blue cloud and adjacent to u
  - if \( d[w] > \text{weight}(u,w) \), set \( d[w] = \text{weight}(u,w) \)
  - this is new distance of w to blue cloud
Prim’s Algorithm

- priority queue stores vertices outside the blue cloud
  - **VISITED** vertices are in blue cloud
  - **UNVISITED** vertices are in green cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - $insert(k, e)$ returns a locator
  - $replaceKey(l, k)$ changes the key of an item
- Store with each vertex
  - distance
  - parent edge in MST
  - locator in priority queue

Algorithm Prim(G)

\[
\begin{align*}
Q & \leftarrow \text{new heap-based priority queue} \\
s & \leftarrow \text{a vertex of } G \\
\text{for all } v & \in G.\text{vertices()} \\
\quad \text{if } v = s & \quad v.\text{setDistance}(0) \\
\quad \text{else} & \quad v.\text{setDistance}(\infty) \\
\quad v.\text{setParent}(\emptyset) & \\
\text{mark } v & \text{ UNVISITED} \\
l & \leftarrow Q.\text{insert}(v.\text{getDistance}(), v) \\
v.\text{setLocator}(l) & \\
\text{while } Q.\text{isEmpty}() & \\
u & \leftarrow Q.\text{removeMin()} \\
\text{mark } u & \text{ VISITED} \\
\text{for all } e & \in G.\text{incidentEdges}(u) \\
z & \leftarrow G.\text{opposite}(u, e) \\
\quad \text{if } z \text{ is UNVISITED} & \\
r & \leftarrow e.\text{weight()} \\
\quad \text{if } r < z.\text{getDistance()} & \\
z.\text{setDistance}(r) & \\
z.\text{setParent}(u) & \\
Q.\text{replaceKey}(z.\text{getLocator}(), r) & \end{align*}
\]
Example
Example
**Prim’s Algorithm: Correctness Proof**

**Theorem:** after each iteration, marked edges are part of some MST

**Proof (by contradiction):**

- Let \((u, v)\) be first marked edge after which theorem is false
  - before adding \((u, v)\), marked edges were a part of some MST \(T\), after adding \((u, v)\), marked edges are not part of any MST
- Consider blue cloud right before \(u\) was added to it
- In \(T\), there must be path between \(u\) and \(v\), it does not include edge \((u, v)\)
- Let \(w\) be first blue vertex after \(u\) on this path
- Let \(t\) be the green vertex before \(w\) on this path
- Get \(T^*\) from \(T\) by remove edge \((w, t)\) adding \((u, v)\)
  - \(T^*\) is spanning tree, it has all marked edges and \((u, v)\)
  - weight of \(T^*\) \(\leq\) weight of \(T\) because \((u, v)\) is the smallest weight edge weight out of the blue cloud
  - \(T^*\) is a MST which has marked edges after marking \((u, v)\)
- **CONTRADICTION!**
Prim’s Algorithm: Complexity Analysis

- Edge weights **can** be negative for Prim’s algorithm.
- Running time analysis is exactly like that for the Dijkstra’s algorithm.
- Assume
  - setting/getting a label takes $O(1)$ time,
  - graph is represented by the adjacency list structure.
- Prim algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure.
Kruskal’s Algorithm

- A tree with \( n \) vertices must have exactly \( n - 1 \) edges
- A priority queue stores the edges outside the cloud
  - key: weight
  - element: edge
- At the end of the algorithm
  - one cloud that encompasses the MST is left
  - a tree \( T \) which is MST

Algorithm KruskalMST(G)

```plaintext
for each vertex \( v \) in \( G \) do
    \( \text{cloud}(v) = \{v\} \)

let \( Q \) be a priority queue
insert all edges into \( Q \) using their weights as the key

\( T \leftarrow \emptyset \)
while \( T \) has fewer than \( n-1 \) edges do
    edge \( e = Q\).removeMin()
    let \( u, v \) be endpoints of \( e \)
    if \( \text{cloud}(v) \neq \text{cloud}(u) \) then
        add edge \( e \) to \( T \)
        merge \( \text{cloud}(v) \) and \( \text{cloud}(u) \)
return \( T \)
```
Data Structure for Kruskal Algorithm

- Algorithm maintains a forest of trees
- An edge is accepted if it connects distinct trees
- Need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the operations:
  - **find**(u): return the set storing u
  - **union**(u,v): replace the sets storing u and v with their union
Partition Representation

- Each set is stored in a sequence, sequence head is the set name.
- Each sequence element has a reference to the sequence head:
  - operation `find(u)` is $O(1)$, returns head of the sequence $u$ belongs to.
  - operation `union(u, v)` moves elements of smaller set to larger set and updates their head references.
  - operation `union(u, v)` takes time proportional to $\min(n_u, n_v)$, where $n_u$ and $n_v$ are sizes of sets storing $u$ and $v$.
- Whenever an element is moved, it goes into a set of size at least twice larger, hence each element is moved at most $\log n$ times.
Kruskal’s Algorithm

```
for each vertex v in G do
    cloud(v) = {v}

let Q be a priority queue
insert all edges into Q using their weights as the key
T ← ∅
while T has fewer than n-1 edges do
    edge e = Q.removeMin()
    let u, v be endpoints of e
    if cloud(v) ≠ cloud(u) then
        add edge e to T
        merge cloud(v) and cloud(u)

return T
```

- each element is moved at most \( \log n \) times
- takes constant time to move 1 element
- there are \( n \) elements, time spent moving them is at most \( n \log n \)
- total running time is \( n \log n + 3n + m \log n + 3m + 1 \), which is \( O((m+n) \log n) \)
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example

Diagram showing a network of cities with distances between them. The cities are labeled as SFO, LAX, DFW, ORD, MIA, JFK, BOS, PVD, and BWI. Distances are indicated by numbers on the edges connecting the cities.
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example
Kruskal Example