Minimum Spanning Trees

Outline and Reading
- Minimum Spanning Trees (§7.3)
  - Definitions
  - A crucial fact
- The Prim-Jarnik Algorithm (§7.3.2)
- Kruskal's Algorithm (§7.3.1)
- Baruva's Algorithm (§7.3.3)

Minimum Spanning Trees
- 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 $T$.
  - By the cycle property, $\text{weight}(e) \leq \text{weight}(f)$.
  - Thus, $\text{weight}(f) = \text{weight}(e)$.
  - We obtain another MST by replacing $f$ with $e$.

Prim-Jarnik’s Algorithm
- Similar to Dijkstra’s algorithm (for a connected graph).
- We pick an arbitrary vertex $x$ and we grow the MST as a cloud of vertices, starting from $x$.
- We store with each vertex $v$ a label $d(v) = \text{smallest weight of an edge connecting } x \text{ to a vertex in the cloud}$.
- 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 $u$.
### Prim-Jarnik’s Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - insert(u,v) returns a locator
  - replaceKey(LA) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue

#### Algorithm: Prim-JarnikMST(G)

```
Q ← new heap-based priority queue
s ← a vertex of G
for all v ∈ G.vertices()
  if v ≠ s
    setDistance(v, 0)
    setParent(v, 0)
    setLocator(v, 0)
    insert(Q, v)

while ¬Q.isEmpty()
  u ← Q.removeMin()
  for all e ∈ G.incidentEdges(u)
    r ← G.opposite(e)
    if r ≠ setParent(e)
      w ← setDistance(r)
      setDistance(r) ← getDistance(u, e) + w
      if ¬Q.insert(Q, r)
  Q.replaceKey(Q, Locator(u, w))
```

### Example

- **Example (contd.)**

### Analysis

- **Graph operations**
  - Method incidentEdges is called once for each vertex
- **Label operations**
  - We set/get the distance, parent and locator labels of vertex \( t \) in \( O(\text{deg}(t)) \) time
  - Setting/getting a label takes \( O(1) \) time
- **Priority queue operations**
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes \( O(\log m) \) time
  - The key of a vertex \( w \) in the priority queue is modified at most \( \text{deg}(w) \) times, where each key change takes \( O(\log m) \) time
  - Prim-Jarnik’s algorithm runs in \( O(m + m \log m) \) time provided the graph is represented by the adjacency list structure
  - Recall that \( \sum \text{deg}(v) = 2m \)
  - The running time is \( O(m \log m) \) since the graph is connected

### Data Structure for Kruskal Algorithm

- The algorithm maintains a forest of trees
- 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 the operations:
  - `find(u)`: return the set storing \( u \)
  - `union(u, v)`: replace the sets storing \( u \) and \( v \) with their union
Representation of a Partition

- Each set is stored in a sequence
- Each element has a reference back to the set
- Operation `find(u)` takes O(1) time, and returns the set of which u is a member.
- In operation `union(u,v)`, we move the elements of the smaller set to the sequence of the larger set and update their references
- The time for operation `union(u,v)` is min(n_u, n_v), where n_u and n_v are the sizes of the sets storing u and v.
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times.

Partition-Based Implementation

- A partition-based version of Kruskal’s Algorithm performs cloud merges as unions and tests as finds.

Algorithm Kruskal(G):

- **Input:** A weighted graph G.
- **Output:** An MST T for G.
- Let P be a partition of the vertices of G, where each vertex forms a separate set.
- Let Q be a priority queue storing the edges of G, sorted by their weights.
- Let T be an initially-empty tree.
- While Q is not empty:
  - (u,v) ← Q.removeMinElement()
  - If P.find(u) ≠ P.find(v):
    - Add (u,v) to T
    - P.union(u,v)
- Return T.

Running time: O((n+m)log n)

Example

<table>
<thead>
<tr>
<th>JFK</th>
<th>BOS</th>
<th>MIA</th>
<th>ORD</th>
<th>LAX</th>
<th>DFW</th>
<th>SFO</th>
<th>BWI</th>
<th>PVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>867</td>
<td>2704</td>
<td>187</td>
<td>849</td>
<td>1258</td>
<td>849</td>
<td>1391</td>
<td>184</td>
<td>946</td>
</tr>
<tr>
<td>1464</td>
<td>1090</td>
<td>1391</td>
<td>184</td>
<td>946</td>
<td>1391</td>
<td>184</td>
<td>946</td>
<td>1090</td>
</tr>
<tr>
<td>1235</td>
<td>2342</td>
<td>1235</td>
<td>1121</td>
<td>1258</td>
<td>1121</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
</tr>
<tr>
<td>337</td>
<td>802</td>
<td>1090</td>
<td>1121</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>JFK</th>
<th>BOS</th>
<th>MIA</th>
<th>ORD</th>
<th>LAX</th>
<th>DFW</th>
<th>SFO</th>
<th>BWI</th>
<th>PVD</th>
</tr>
</thead>
<tbody>
<tr>
<td>867</td>
<td>2704</td>
<td>187</td>
<td>849</td>
<td>1258</td>
<td>849</td>
<td>1391</td>
<td>184</td>
<td>946</td>
</tr>
<tr>
<td>1464</td>
<td>1090</td>
<td>1391</td>
<td>184</td>
<td>946</td>
<td>1391</td>
<td>184</td>
<td>946</td>
<td>1090</td>
</tr>
<tr>
<td>1235</td>
<td>2342</td>
<td>1235</td>
<td>1121</td>
<td>1258</td>
<td>1121</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
</tr>
<tr>
<td>337</td>
<td>802</td>
<td>1090</td>
<td>1121</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
<td>1258</td>
</tr>
</tbody>
</table>
Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

Algorithm \( \text{BaruvkaMST}(G) \)

1. \( T \leftarrow \langle \text{the vertices of } G \rangle \)
2. While \( T \) has fewer than \( n-1 \) edges do
   1. For each connected component \( C \) in \( T \) do
      1. Let edge \( e \) be the smallest-weight edge from \( C \) to another component in \( T \).
      2. If \( e \) is not already in \( T \) then
         1. Add edge \( e \) to \( T \)
3. Return \( T \)

- Each iteration of the while-loop halves the number of connected components in \( T \).
  - The running time is \( O(m \log n) \).