MST and Prim’s Algorithm

Underlying Principles
- Finding the MST has been studied long before development of modern data structures
- New algorithms differ from old ones in the use of these data structures
- Find MST for graphs with billions of edges

Basic Properties
- Adding an edge to any tree creates a unique cycle
  - Basis for fundamental MST properties

Cycle Property
- Given a graph G, consider the graph G’ defined by adding an edge E to G. Adding E to a MST of G and deleting a maximal edge on the resulting cycle gives an MST of G’

Property
- Used to build a minimum spanning tree
- Assume that you have already selected some edges for the minimum spanning tree
- Explanation
  - Cut - graph partition of vertices into two disjoint sets
  - Crossing edge - an edge that connects a vertex in one set with a vertex in the other
• Suppose edges $X$ are part of a minimum spanning tree $G=(V,E)$. Pick any subset of nodes $S$ for which $X$ does not cross between $S$ and $V-S$, and let $e$ be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

• Proof by contradiction
  - Suppose $e$ is a minimal crossing edge that it is not in any MST, let’s say $T$
  - If $e$ is added to $T$, then there is a cycle and at least one other crossing edge, $f$
  - Since $e$ is a minimal crossing edge, $f$ must be equal or higher weight
  - New spanning tree created of equal or lower weight by removing $f$
  - This contradicts the minimality of $T$

### MST Algorithms

• Prim’s algorithm
  - Start with any vertex as a single-vertex MST
  - Add V-1 edges to it
    - Always take a minimal edge that connects a vertex on MST to a vertex no yet on MST

• Krusgal’s algorithm
  - Process edges in order of their length (shortest first)
  - Add edge if it does not form a cycle with edges previously added
  - Stop after V-1 edges

• Boruvvka’s algorithm
  - Add all edges that connect each vertex to its closest neighbor
    - Results in a forest of MST subtrees
  - Add to the MST the edges that connect each tree to a closest neighbor
  - Iterate until there is just one tree

### High-level operations

• Find a minimal edge connecting two subtrees
• Determine whether adding an edge would create a cycle
• Delete the longest edge on a cycle

• Develop algorithms and data structures to support operations

### Prim’s Algorithm

• Simplest MST algorithm
• Method of choice for dense graphs
• Idea
  - Maintain a cut of the graph comprised of tree vertices and nontree vertices
  - To begin, put any vertex on the MST
  - Add minimal crossing edge
  - Repeat V-1 times to put all vertices on the tree
• Always interested in the shortest edge that goes from a tree vertex to a nontree vertex
**Incremental Change**

- Adding a vertex to the MST is an incremental change
- When adding vertex v, the only possible change is that the addition of v made w (each nontree vertex) closer to the tree
- Data Structure Requirements:
  - o
  - o
  - o
  - o

**Simplest Implementation**

- After adding each edge
  - o Check whether new edge brought any nontree vertex closer
  - o Find the next edge
- How? Pass through the incident edges to new vertex
  - o Update weights
  - o Determine the next closest edge
- Running Time:

**Pseudo Code**

- Initialize three arrays
  - o spanTree of size V initialized to not visited
  - o closestEdge of size V initialized to itself
  - o weightArray of size V+1 initialized to SENTINAL_WEIGHT
- Put vertex zero in the tree
  - o spanTree[0] ← 0
  - o chosenVertex ← 0
- Iterate over the vertices while there is still a vertex to add
  - o previousVertex ← chosenVertex
  - o spanTree[chosenVertex] = closestEdge[chosenVertex]
  - o chosenVertex ← V
  - o Iterate over each vertex
    - If it (possibleVertex) is not in the spanning tree
      - If it is adjacent to previousVertex and this edge makes it closer to the Tree
        - o weightArray[possibleVertex] ← edge weight
        - o closestEdge[possibleVertex] ← previousVertex
      - If weight of possibleVertex is less than the weight of the chosenVertex
        - o chosenVertex ← possibleVertex
**Priority First Implementation**

- When graph is sparse, there are fewer than V steps to perform each operation
- Each step add potential edges to the edge collection (fringe)
- Algorithm
  - Begin with self loop
  - Move minimal edge from the edge collection to the tree
  - Visit the vertex edge leads to
  - Add incident edges that lead to nontree vertices to collection
    - Replace longer edge when two edges in the collection point to the same vertex
- What data structure should be used for collection?
- Running time:

**Psuedo Code**

- Create three arrays and initialize two
  - spanTree of size V to not visited
  - closestEdge of size V to not visited
  - weight of size V
- Make zero vertex the root of the tree
  - closestEdge[0] ← 0
  - Insert into priority queue(0,0) for (id, weight)
- While the priority queue is not empty
  - chosenVertex ← dequeue priority queue
  - spanTree[chosenVertex] ← closestEdge[chosenVertex]
  - Iterate over vertices adjacent to the chosenVertex
    - If candidateVertex is not adjacent to the tree
      - Insert into priority queue(candidateVertex, edge weight)
      - weight[candidateVertex] ← edge weight
      - closestEdge[candidateVertex] ← chosenVertex
    - Else if candidate vertex is not in the spanning tree and the edge make it closer to the tree
      - Update priority queue
        (candidateVertex, edge weight)
      - weight [candidateVertex] ← edge weight
      - closestEdge [candidateVertex] ← chosenVertex