Lesson-17 – Minimum Spanning Tree

Algorithm ***GenericMST***(***G***)
    T ← an Empty tree
    while ***T does not form a spanning tree*** **do**
        (***u, v***) ← a safe edge of ***G***
        T ← (***u, v***) ∪ ***T***
    return ***T***

Two MST algorithms Prim's and Kruskal's discussed in this section. They each use a specific rule to determine a safe edge of GENERIC-MST. In Kruskal's algorithm, the set A is a forest. The safe edge added to A is always a least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set A forms a single `tree`. The safe edge added to A is always a least-weight edge connecting the `tree` to a vertex not in the tree.

Prim's Algorithm –PQ

- Similar to Dijkstra's shortest path algorithm (for a connected graph)
- We pick an arbitrary vertex ***s*** and we grow the MST as a tree of vertices, starting from ***s***
- We store with each vertex ***v*** a parent edge ***parent***(***v***) with the smallest weight of any edge connecting ***v*** to a vertex in the tree
- At each step:
    - We add to the tree, the vertex ***u*** outside the tree with the parent edge with the smallest/minimum weight
    - We update the parent edges of the vertices adjacent to ***u***

Run time Analysis

Prim-Jarnik's Algorithm (cont.)

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*, *e*) returns a locator
  - *replaceKey*(*l*, *k*) changes the key of an item
- We store three labels with each vertex:
  - Distance
  - Parent edge in MST
  - Locator in priority queue
- Correction in red inspired by Bereket Chalew (May 2014)

```
Algorithm PrimJarnikMST(G)
    PQ ← new priority queue
    for all v ∈ G.vertices() do // n times
        setParent(v, ∅) // O(1)
        d ← ∞
        p ← PQ.insertItem(d, v) // log n
        setLocator(v, p) // O(1)
    s ← G.aVertex () // get a start vertex
    p ← getLocator (s)
    PQ.replaceKey(p, 0)
    while ! PQ.isEmpty() do // O(n)
        u ← PQ.removeMin() // log n
        setLocator(u, ∅) {u is now in MST}
        for all e ∈ G.incidentEdges(u) do // 2m
            z ← G.opposite(u, e) // O(1)
            r ← weight(e) // O(1)
            p ← getLocator (z) {p=(weight, vertex)}
            if p ≠ ∅ {z not yet in MST}
                and r < p.key() then
                setParent(z, e)
                PQ.replaceKey(p, r) // log n
```


- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
- The running time is $O(m \log n)$ since the graph is connected

setParent(v, ∅): Each vertex v is initially given a parent value of ∅, indicating no parent because they're not yet part of the MST.

setLocator(u, ∅): Since u is now part of the MST, its locator is set to ∅.

setLocator(v, p): Each vertex is given a locator that references its position in the priority queue, allowing for constant-time updates (O(1)) of its key.

setParent(z, e): Set the parent of z to be the edge e, indicating that e is now part of the MST.

PQ.replaceKey(p, r): Update the key of z in the priority queue to the weight of e, as it's the new minimum weight edge connecting z to the MST.

Kruskal's Procedure

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal's Runtime Analysis

```
Algorithm KruskalMST(G)
  for each vertex v in G do      // O(n)
    define a Cloud(v) ← {v}
  Q ← new heap-based priority queue.
  for each edge e in G.edges()
    Q.insert(weight(e), e) // O(log m)
  T ← Empty tree
  while T has fewer than n-1 edges do  // O(n-1)
    e ← Q.removeMin()  // O(log m)
    (u, v) ← G.endVertices(e) // O(1)
   // Performance of this step depends on the data structure you used.
    if Cloud(v) != Cloud(u) then
      Add edge e to T
      Merge Cloud(v) and Cloud(u)
  return T
```

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure to check whether two vertices belong to the same tree. They are called Disjoint Subsets and Union-Find Algorithms.

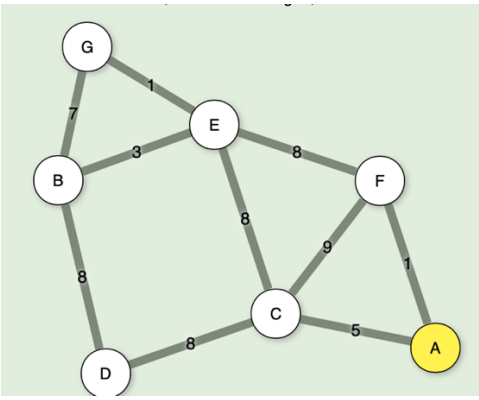
find(x) – returns the subset Xi to which x belongs

union(A,B) – replaces the subsets A, B in C with A U B.

With this approach runtime is O(m log n).

Review Questions

1. What is Minimum Spanning Tree(MST)?
2. About Cycle Property for MST and how do you prove it?
3. Write the generic MST algorithm?
4. How does the Prim's algorithm works to obtain a MST?
5. Able to write the pseudo code of Prim's and analyse each step.
6. Manually execute the Prim's procedure for the given tree. Start Vertex A.



7. How does the Kruskal's algorithm works to obtain a MST?
8. Able to write the pseudo code of Kruskal's algorithm.
9. Manually execute the Kruskal's procedure for the given tree. Start Vertex A.