1 Minimum Spanning Trees

- Given a graph G, we want to find the tree (with a connected set of \(|V| - 1\) edges from G) that has the minimum weight
- The Minimum Spanning Tree should reach every node, and the total sum of the edge weights should be the minimum possible to connect all the nodes
- Since the MST is a tree, there is only one path from any node to any other

2 Prim’s Algorithm

- Fundamentally the same as Dijkstra’s algorithm, except we keep track of the distance of nodes to the “cloud” of nodes that are in the MST (the minimum distance to connect any node that is not in the cloud to any node that is in the cloud)
- We are not interested in the minimum distance from any particular node to any other, just a way to link all the nodes together with the minimum weight edges
- Runs in \(O((|V| + |E|) \log |V|)\)

Given a graph \(G = (V, E)\), compute the Minimum Spanning Tree

Choose any node \(s\)
\[ D[s] = 0 \]
For all \(u \in V, u \neq s\)
\[ D[u] = \infty \]

Construct a priority queue \(Q\) with the array \(D\) as keys for (node, edge) pairs
Construct an empty tree \(T\)
while (!\(Q\).isEmpty()) {
  \((u, e) = Q\).removeMin()\]
  Add \((u, e)\) to \(T\)
  For each \(v\) in \(Q\), \[(u, v) \in E\]
  if \(w(u, v) < D[v]\)
  \[ D[v] = w(u, v) \]
  Set \(v\) in \(Q\) to be \((v, (u, v))\)
  Update \(Q\) (decrease-key)
}

3 Kruskal’s Algorithm

- A simpler algorithm (that is about equally fast) is Kruskal’s algorithm
- Instead of looking at nodes and travelling outwards, treat all nodes (initially) as a set and merge sets with edges of increasing weight until everything is in one set
Given a graph $G = (V, E)$, compute the Minimum Spanning Tree

For all $u \in V$
    Create a set with u in it

Construct a priority queue Q with all edges in E with weights as keys

Construct an empty tree T

while $|T| < |V| - 1$
    $(u,v) = Q.removeMin()$
    If $(u$ and $v$ are not in the same set)
        Add $(u,v)$ to $T$
        Union u’s set with v’s set

4 Eulerian and Hamiltonian Paths

- An Eulerian path is a path (a sequence of edges) through a graph such that each edge is visited once
- A Hamiltonian path is a path through a graph such that each vertex is visited once
- A slight variation are Eulerian and Hamiltonian cycles, in which the path must terminate where it began in the graph

5 Announcements

- Reading: Data Structures (Into Java) – Ch. 12; Goodrich and Tamassia – Ch. 12

6 Disjoint Sets

- A group of sets, each of which store elements
- Two operations: union and find
- A find operation given any element of any set returns the “canonical element” of its set – the element by which we identify the set
- A union operation given two sets merges the sets into one

7 Disjoint Sets: list-based find

- We want to do find operations in $O(1)$ time – we should somehow know which set every element belongs to (what the canonical element is)
- Store sets as linked lists and store pointers from each element to the canonical element
8 Disjoint Sets: list-based union (slow)

- We want to be able to merge two sets with the union operation
- We can simply take all elements from one list, add them to the other list and change all their pointers to point to the canonical element of the first list
- This operation is slow – $O(n)$

9 Disjoint Sets: tree-based

- Instead of representing the union/find structure as a list, we represent it as a forest of trees
- Each item is in its own tree, and through union operations is joined with other trees
- The root of each tree is the canonical element (all elements of the tree point to their parents, etc... all the way up to the root)

10 Disjoint Sets: tree-based union (fast)

- A union operation merges two trees by making one tree a child of the root of the other – $O(1)$ given the roots of the sets to be merged
- Now find operations are slower – we must travel up the tree to the root, which may get further and further away as more union operations are done
- Using the heuristic union by height, we union trees based on their heights – the root of each tree stores the height of the tree and when a union operation is performed, the shorter tree is made the child of the root of the taller tree

11 Disjoint Sets: tree-based find (fast)

- Find operations are slow if unioned trees grow tall – if we perform find on a deep node repeatedly, we waste time travelling through the tree
- Using the heuristic path compression, every time find is called, we move the elements along the way (up to the root) making them children of the root element – this way, future find calls will immediately find the root
- We don’t want to have to recompute the height of the tree (for the union by height heuristic), so we consider the rank of the tree to be the same as the height had we not done path compression
- With these heuristics, union and find operations in the average case (which we won’t analyze) are in almost constant time