**Weighted Graphs**

**Overview**

- Spatial layouts *were* irrelevant but now we want to reason about bandwidth, distance, capacity, etc. of real-world things

- We can represent this information with edge weights

- We can store edge weights with an adjacency matrix or an adjacency list; adjacency list is linked list of neighbors with weight field

- Now need to find *minimum* spanning tree and *weighted* shortest path

**Minimum Spanning Tree**

- Minimum Spanning Tree (MST) is the spanning tree that has the minimum sum of the weights of its edges

- There are a max of V2 trees; naively, we could generate a lot of trees using BFS and DFS but this is impractical

- Prim’s Algorithm is guaranteed to generate \*an\* MST by looking at all neighbors of each node **(Example in notes)**:

1. Initialize tree T to contain the starting vertex (T will eventually become an MST)

2. While there are vertices not in T:

a. Find the minimum-weight edge that connects a vertex in T to a vertex not yet in T

b. Add edge with its vertex to T

- At each step of Prim’s algorithm, all possible edges are checked; overall runtime evaluates to 𝞱(v3)

- However, we don’t need to look through all remaining edges; rather, we only need to consider the best edge for each vertex **(Example in notes)**

- For each vertex we add to T, we need to check all neighbors to check for edges to add to T next; ends up being 𝞱(v2) for either adjacency list or adjacency matrix

- An even faster way to pick the best edge uses a priority queue, as it removes min value in 𝞱(lg n) and adds in 𝞱(lg n); this approach is called Lazy Prim’s **(Example in notes)**:

1. Visit a vertex and look at its neighbors

2. Add edges coming out of that vertex to a PQ

3. While there are unvisited vertices, pop from the PQ for next vertex to visit & repeat

- Lazy Prim’s is not as much of an improvement as the graph becomes more dense, since every edge still has to be inserted; we end up with 𝞱(e lg e) as the runtime

- The eager Prim’s approach only stores cheapest edge for each vertex, but an indirection table is needed; every edge is still considered, but eager Prim’s has a runtime of 𝞱(e lg v)

- Comparison of Prim’s implementations:

a. Parent/best-edge array Prim’s: dense graphs, adjacency matrix; runtime is 𝞱(v2), while memory is 𝞱(v)

b. Lazy Prim’s: sparse graph, adjacency list; runtime is 𝞱(e lg e), while memory is 𝞱(e)

c. Eager Prim’s: sparse graph, adjacency list; runtime is 𝞱(e lg v), while memory is 𝞱(v) – the main benefit of eager Prim’s is the space reduction

**Weighted Shortest Path**

- The weighted shortest path is the minimum sum of all edges from a start vertex to a destination vertex

- Dijkstra’s Algorithm **(Example in notes)**:

0. Set a distance value of MAX\_INT for all vertices except start, set cur = start

1. While destination is not visited

a. For each unvisited neighbor of cur, compute tentative distance from start to the unvisited neighbor through cur and update any vertices for which a lesser distance is computed

b. Mark cur as visited

c. Let cur be the unvisited vertex with the smallest tentative distance from start

- Spanning tree created by path represents weighted shortest path from 0 to any other vertex

- Dijkstra’s Analysis:

- Implemented as a best path/parent array, the runtime is 𝞱(v2) for matrix and for list because you need to go through distance array to find cheapest

- Implemented as an indexable priority queue is very similar to eager Prim’s, but storing paths instead of edges; runtime is 𝞱(e log v), same as eager Prim’s

**One More Minimum Spanning Tree Algorithm**

**-** Kruskal’s MST Algorithm **(Example in notes)**:

1. Insert all edges into a PQ

2. Grab the minimum edge from the PQ that doesn’t create a cycle in the MST **\*\***

3. Remove it from the PQ and add it to the MST

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph

- However, we need to efficiently implement cycle detection and connectedness checking

- This is the motivating question for creating the union-find data structure