Algorithms for Data Science
Lab 9
Greedy Shortest Path and MST

Below is a Python implementation of Dijkstra's shortest path algorithm. It takes a weighted graph and produces the shortest distance from a given vertex to every other vertex in the graph. This particular implementation is hard-coded to compute the distance from vertex 0, but it is a very simple change to specify a different starting vertex.

Dijkstra's shortest path algorithm is a greedy algorithm and very similar to Prim's minimal spanning tree (MST) algorithm. The vertices are divided into two sets: the vertices still to be processed and the vertices where we have already figured out the shortest distance. At each step, we add one vertex to the processed set. It is the vertex with the minimal distance that we have not processed yet. Adding this vertex in turn potentially updates the distances to its neighbors. The distance to any neighbor is the distance to the vertex plus the cost of the edge to that neighbor. If this is a better distance than what the neighbor already has, then we update it. Therefore, at each step we are selecting the next vertex and locking it in. The greedy choice property holds for the same reasons it does for Prim's MST, so this leads to optimal distances in the end.

Note that we store the vertices with priorities in a simple Python list and then search for the min each step. We could have thought about using a priority queue for this, such as heapq, demonstrated earlier in the course. However, note that we potentially change the priorities of elements after we add them to the heap. Most standard heap implementations, including heapq, do not allow you to change priorities of elements that have already been added. There are some relatively simple and efficient ways to upgrade a heap so that this can happen, but we will not do so in this lab because the focus is on the greedy method rather than heaps.

Run the algorithm a few times, perhaps uncommenting the prints or changing the starting vertex. After you are comfortable with how it works, you can move on to your task for this lab. You are to use this code as a blueprint for implementing Prim's MST algorithm described in the videos/PowerPoint slides. Note that there is pseudo-code provided in the videos/PowerPoint slides as well as an extensive walkthrough of an example. The graph in the code below is actually the same one as in the videos/Power Point slides. You should call your method *mst* rather than *shortest*, but use similar data structures for representing the graph, vertices, and so on. The output from your code should be a list of edges that form the MST. Note the 0→-1 edge just represents which vertex is the root of the MST:

```
[[0, -1], [1, 0], [3, 1], [2, 3], [5, 3], [4, 5], [7, 1], [6, 0]]
```

```
# Dijkstra's shortest path greedy algorithm
```

```python
# Find the min priority vertex from the list of given vertices
# Each vertex in the form of a list with priority as the first
# element returns the min vertex and removes it from the list
def extractMin(verts):
    minIndex = 0
    for v in range(1,len(verts)):
        if verts[v][1] < verts[minIndex][1]:
            minIndex = v
    return verts.pop(minIndex)

# Dijkstra's shortest path algorithm
def shortest(g):

    # Create a list of vertices and their current shortest distances
    # from vertex 0
    # [vertNum, dist]
    nVerts = len(g)
    vertsToProcess = [[i, float("inf")] for i in range(nVerts)]

    # Start at vertex 0 - it has a current shortest distance of 0
    vertsToProcess[0][1] = 0

    # Start with an empty list of processed edges
    vertsProcessed = []

    while len(vertsToProcess) > 0:
        u = extractMin(vertsToProcess)
        vertsProcessed.append(u)
        #print("to process:",vertsToProcess)
        #print("   processed:",vertsProcessed)

        # Examine all potential verts remaining
        for v in vertsToProcess:
            # Only care about the ones that are adjacent to u
            if g[u[0]][v[0]] > 0:
                # Update the distances if necessary
                if u[1] + g[u[0]][v[0]] < v[1]:
                    v[1] = u[1] + g[u[0]][v[0]]

    print(vertsProcessed)

# Adjacency matrix representation of a graph
# This particular graph is the one from the videos
# The vertices didn't have labels in the videos
#  so I'm using the following vertex labels:
#    2
#   / \
#  3---1--7
#  |\  |
#  4 | 0--6
#   \|/
#    5
graph = [[0, 7, 0, 0, 0, 10, 15, 0],
         [7, 0, 12, 5, 0, 0, 0, 9],
         [0, 12, 0, 6, 0, 0, 0, 0],
         [0, 5, 6, 0, 14, 8, 0, 0],
         [0, 0, 0, 14, 0, 3, 0, 0],
         [10, 0, 0, 8, 3, 0, 0, 0],
         [15, 0, 0, 0, 0, 0, 0, 0],
         [0, 9, 0, 0, 0, 0, 0, 0]]

shortest(graph)
```