Programming Assignment 2

Due November 23, 2020 at 11:59 PM

In this assignment, you will implement Disjktra's algorithm for computing shortest paths, and Kruskal's algorithm for minimum spanning trees (including a union-find data structure).

1 (20 points) Implementing Dijktra and Kruskal (autograded on Gradescope)

The first part of the assignment is to implement the two algorithms. We have provided starter code that reads the graphs into memory and

Dijkstra's algorithm The input is the adjacency list representation of a **directed** graph with a nonnegative cost on each edge, together with a a source node.

Kruskal's algorithm The input is the adjacency list representation of an **undirected** graph with costs on each edge (not necessarily positive).

The details of the file format are in Appendix A below. Your grade is based on the autograder score, but we will also inspect your code to make sure you have implemented the correct algorithms (as opposed to Prim's, for the case of MSTs). You will only receive points for implementation of the correct algorithm.

2 (15 points) Experiments with SP trees and MST's (submitted as a PDF on Gradescope)

Suppose you need to choose a set of edges in the graph that do well with respect to two different objectives: their total weight (for which the MST is the best choice), and the length of the paths from s to all other vertices. For example, suppose we need to maintain a set of roads in the winter time. We want to choose roads that allow us to get from s (think a fire station, or other central resource) to other important points (the nodes of G). But the total cost of plowing a road (which we'll take to be proportional to their length) should also be small.

In the second part of the assignment, you will run experiments to compare the kinds of spanning trees that arise when optimizing the two different objectives (path lengths, in the case of Dijkstra's algorithm, and total weight, in the case of the MST). These trees can look pretty different! Figure 1 shows the edges in one particular graph. The nodes in this graph correspond to locations in the real plane (that is (x, y) pairs); the cost of an edge is the actual length of the straight line segment connecting its endpoints. We've included all edges below a certain length.

We will provide several **undirected** graphs as input. (Specifically, we will make sure that each edge appears twice, once in each orientation. You can also think of these graphs as having two

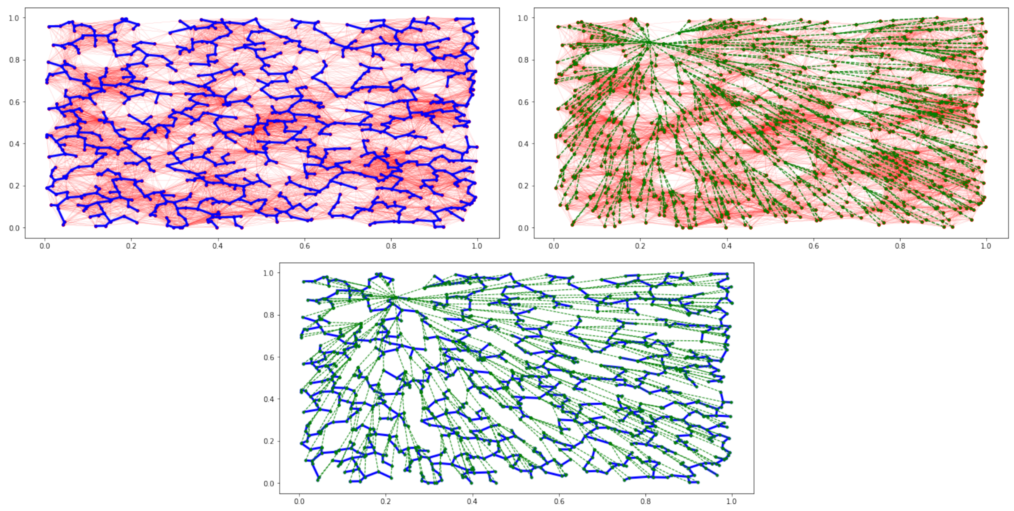


Figure 1: The top two pictures show all the edges of a graph (red) together with the edges of the MST (blue) and the edges of the shortest-paths tree (green) from a particular vertex. The bottom picture shows only the MST and shortest-paths trees.

directed edges for each edge (u, v): one from u to v and v to u. Therefore, if you write an algorithm that works on directed edges, you can still use it on an undirected graph.) Each graph will come with a special source node s, which is to be the root of the shortest paths tree.

For each graph, we will ask: how short are paths from s in the MST? What is the weight of the shortest paths tree? Specifically, let T_{MST} be the minimum-cost spanning tree and T_{SP} be the tree of shortest paths (from the root s we specify). Use your implementations from part 1 to compute two quantities:

- The total weight ratio: $TWR = \frac{\mathsf{cost}(T_{SP})}{\mathsf{cost}(T_{MST})}$
- The maximum ratio of distance. Given a tree T and two nodes u and v, let $d_T(u, v)$ be the length of the path from u to v using the edges in T. Let's define the maximum distance ratio as how much any particular distance from s gets increased if one uses the MST paths instead of the actual shortest path trees: $MDR = \max_{u \neq s} \frac{d_{T_{MST}}(s,u)}{d_{T_{SP}}(s,u)}$.

Doing this requires processing the outputs of the two algorithms you coded up in part 1. Deciding how to go about that is part of the assignment.

Please submit a PDF providing:

1. Results: namely, the pairs (TWR, MDR) for each of the input graphs). These should be printed in a table, and depicted in a scatter plot.
2. Discussion: Suppose you had to choose only one of the two trees for each graph, but needed to take into account both criteria. For each graph, which tree would be better? Is there a clear winner or does it depend on the relative importance of the two objectives?
3. Code: please clearly separate the code for part 1 from the code for part 2 (which may call the functions in your part 1 code).

3 Optional: Optimization (submitted as a PDF on Gradescope)

In this part, you will try to find "better" trees than either T_{MST} or T_{SP}.

This part will be due later; we will provide further details soon.

Appendix

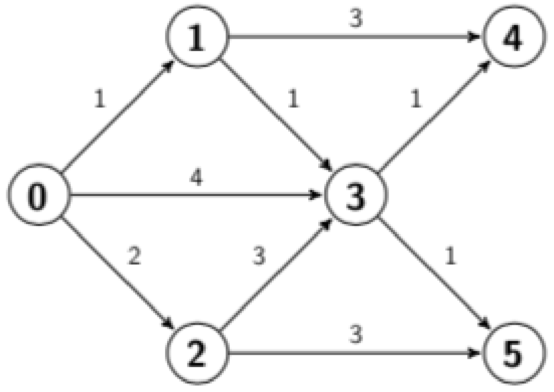
A Graph file information

Since we are now dealing with graphs where the edges have weights, the input format differs from the previous programming assignment. The graph files are all text files that contain $m + 3$ lines. The first three lines will specify the number of vertices N, the number of edges m, and the source node s from which to start Dijkstra's algorithm. The vertices are indexed $0, 1, \ldots, n - 1$. Every row after the first three will specify an edge by giving the starting node, the ending node, and the weight. The weights will all be integers. You may assume that there is a path from s to every node in the graph.

The lines of `input` will contain the following information:

1. n, the number of vertices.
2. m, the number of edges.
3. s, the index of where the paths should originate from.
4. The starting node of edge 0, a comma, the ending node of edge 0, a comma, the weight on edge 0.
5. The starting node of edge 1, a comma, the ending node of edge 1, a comma, the weight on edge 1.
6. ...
7. The starting node of edge m, a comma, the ending node of edge m, a comma, the weight on edge m.

We provide you with a test graph file called `input`, which you may use to help debug your code. It contains the information for the following graph:



Note: passing this test case does not guarantee your code is correct!

B Starter Code

We provide you with a starter code file called `starter_pa2.py`. In this code, you will find two methods to complete: one for Dijkstra's algorithm and one for Kruskal's algorithm. Once you are finished, you may submit your code to the autograder on Gradescope called **Programming Assignment 2, Part 1**. When you submit to Gradescope, **please do not change the name of this file, or add any print() statements to the code, as this will confuse the autograder.** You may resubmit your code to the autograder as many times as you like until the due date.

Now let's talk about the two methods you need to write:

C Implementing Dijkstra's algorithm

In the `dikstra` method, we give you four variables to work with:

- `N`: an integer for the number of nodes in the graph
- `m`: an integer for the number of edges in the graph
- `s`: an integer that is the name of the source node
- `adj_list`: a list of lists of size N, where each index represents a node n. The sublist at index n has a list of two-tuples, each of which represents an outgoing edges from n as: (adjacent node as an integer, weight of edge as a float). **NOTE:** *If a node has no outgoing edges, it is represented by an empty list.*

For the example graph above, the variable `adj_list` is :

$$[[(1, 1.0), (3, 4.0), (2, 2.0)], [(4, 3.0), (3, 1.0)], [(3, 3.0), (5, 3.0)], [(4, 1.0), (5, 1.0)], [], []]$$

The method should return two dictionaries, in which each key represents a node n:

- `distances`: the length of the shortest path from s to n
- `parents`: the previous node before n on the shortest path

In the example graph above, the code should return the following:

- `distances`: $\{0 : 0, 1 : 1, 2 : 2, 3 : 2, 4 : 3, 5 : 3\}$
- `parents`: {0:0, 1:1.0, 2:2.0, 3:2.0, 4:3.0, 5:3.0}

Since the focus of this exercise is on the algorithm, you may import a built-in priority queue, such as Python's `heapq` or Java's `PriorityQueue`. Note that these implementations may not have all the methods the textbooks assumes. You are also free to implement your own priority queue. The textbook discusses using heaps as priority queues on page 64.

D Implementing Kruskal's algorithm

In the `kruskal` method, you will perform Kruskal's algorithm on the same graph as in Dijkstra's algorithm, **except this time you will consider each edge to be undirected**. Therefore, **each edge will be represented twice in your adjacency list.** We give you the following three variables as inputs:

- `N`: an integer for the number of nodes in the graph
- `m`: an integer for the number of edges in the graph
- `undirected_adj_list`: a list of lists of size N, where each index represents a node n. The sublist at index n has a list of two-tuples, each of which represents an edges from n as: (adjacent node as an integer, weight of edge as a float). **NOTE:** *Since the graph is undirected, each edge* (u, v) *is now represented twice in this adjacency list: once at index* u *and once at index* v.

For the example graph above, the variable `undirected_adj_list` would be:

$$[[(1, 1.0), (2, 2.0), (3, 4.0)], [(0, 1.0), (3, 1.0), (4, 3.0)], [(0, 2.0), (3, 3.0), (5, 3.0)],$$
$$[(0, 4.0), (1, 1.0), (2, 3.0), (4, 1.0), (5, 1.0)], [(1, 3.0), (3, 1.0)], [(2, 3.0), (3, 1.0)]]$$

The method `Kruskal` outputs the minimum spanning tree as `mst_adj_list`, which should be formatted in exactly the same way as `undirected_adj_list`. Again, since the MST will be undirected, each edge will be represented twice. For the example graph above, `mst_adj_list` would be:

$$[[(1, 1.0), (2, 2.0)], [(0, 1.0), (3, 1.0)], [(0, 2.0)], [(1, 1.0), (4, 1.0), (5, 1.0)], [(3, 1.0)], [(3, 1.0)]]$$

HINT:. When you are writing `kruskal`, you will want to think carefully about how to the implement `union-find` algorithm. You will find it helpful to write subroutines for this method, such as `find` and `merge`. Revisit **Lab 6** for a detailed discussion of `union-find`.

E Implementing the experiments for Part 2

The three graphs that we you to run these experiments on are in the following files:

- `g_randomEdges.txt`
- `g_donutEdges.txt`
- `g_zigzagEdges.txt`

You can work on Part 2 using the starter code for part 1. To input these specific graphs, follow the instructions in the comments of the `main` fuction of `pa2_starter.py`.

F Implementing Part 3 (optional)

The location files for this part are:

- `g_randomLocations.txt`
- `g_donutLocations.txt`
- `g_zigzagLocations.txt`

You only need to use these files if you choose to do part 3.