CS 330 – Dijkstra's Algorithm for Shortest Path in a Graph

deadline: 10/11/2019 at 11:59 pm

In this exercise you will create an implementation of Dijkstra's Algorithm for shortest paths in a directed graph with weighted edges. The algorithm is in Section 4.4 of *Algorithm Design*, with pseudocode on 138. **Important:** in this exercise we extend the pseudocode and require that you output path information. Your code must run in $O(m \log n)$ time, where m is the number of edges and n is the number of vertices. We've set rough upper bounds on execution time; if your code exceeds these, you'll see the limit and your code's running time.

Submission Format

On Gradescope submit a file (either `dijkstra.py` or `dijkstra.java`, names must match exactly) that reads a text file `input` and creates a text file `output` containing your solution. **Change from previous exercises:** you may submit other files containing additional functions and classes. The autograder will put your code in the same directory as `input` and call it from the command line. Your code must create and write `output` in that directory.

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.

Input

Since we are now dealing with a graph where the edges have weights, the input format differs from previous graphs. The file `input` contains $m + 3$ lines. The first three lines will specify the number of vertices, the number of edges, and the node from which to start. 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. $\ldots$
7. The starting node of edge m, a comma, the ending node of edge m, a comma, the weight on edge m.

We have uploaded files corresponding to the visible test cases on Gradescope. They may help you test and debug locally. Note: passing these test cases does not guarantee your code is correct!

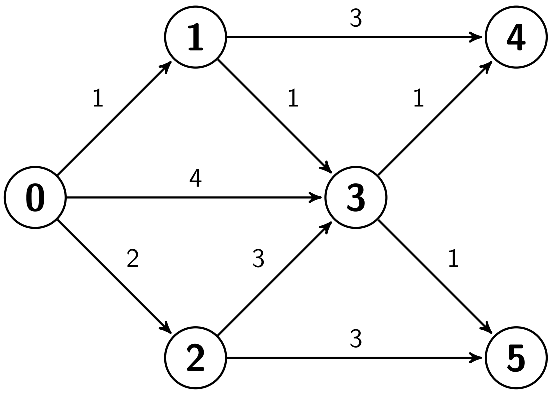
Output

The output must be a text file `output` with n lines specifying the lengths of paths **and** the node from which the shortest path reaches that node, the "parent" node in the path. Keeping track of the parents will allow us to quickly determine the explicit shortest paths - do you see how? The first line will be the length of the shortest path from s to vertex 0, a comma, and the parent node of 0 in the path from s to

0. The n-th row will be the length of the shortest path from s to vertex $n-1$, a comma, and the parent node of $n-1$. Note that the length of the shortest path from s to itself is 0. On the line corresponding to s, write a '`-`' in place of a parent.

Example

Consider the following directed graph, taken from Figure 4.7:



If we calculate the paths from vertex 0, one possible input file follows. Note that the edges may be in any order.

```
input:

6
9
0
0,1,1
0,3,4
0,2,2
2,3,3
2,5,3
3,4,1
1,4,3
1,3,1
3,5,1
```

The output is unique and lists the distances from 0 to every node as well as the path information. The last line, for instance, shows that the shortest path from 0 to 5 is of length 3 and that the final edge in that path begins at node 3.

```
output:

0,-
1,0
2,0
2,1
3,3
3,3
```