COMP 285 (NC A&T, Fall ‘22) Homework 6 (80 pt)

Fun with Graph Algorithms

###### Due 11/03/22 @ 11:59PM ET

Submitting

In order to submit this assignment, you must download your assignment as a ZIP file and upload it to Gradescope. There will be a written component and a coding component.

For the written component, you will write your answers in the corresponding `.txt` files.

For the coding component, you will write your solution in `answers.cpp`. We will use an autograder to test your solutions - the results of which can be seen after submitting to Gradescope. Note that you have unlimited tries to submit. In order to receive full credit, you must also provide documentation on your approach, see the commented sections in the `answers.cpp` file.

Note: This homework, again, requires you to input your Banner ID as a string in the getBannerID method in `answers.cpp`.

Question 1: More Graph Traversals (30 pts)

Please fill in the `getBannerId` function with your Banner ID in `answers.cpp`. When you hit run on replit, you will receive an auto-generated adjacency matrix such as this example:

```
 ##################
##### PART 2 #####
##################
Your adjacency matrix:
[null,    1, null,   10,    8,    9, null]
[   1, null,   15,    6,    6, null, null]
[null,   15, null, null,   14, null, null]
[  10,    6, null, null,    9, null,    9]
[   8,    6,   14,    9, null, null,    4]
[   9, null, null, null, null, null,    3]
[null, null, null,    9,    4,    3, null]
Run Dijktra's starting at node 0
```

Answer the following questions with the graph provided.

a. Drawing the Weighted Graph (5 pt) Represent your weighted graph as a picture using the same template as above. You can use the same template.

For reference, the graph representation of the above is as follows

Upload your picture as `weighted-graph.png`.

b. Weighted Adjacency List (5 pt) Take the generated adjacency matrix and translate it into an adjacency list. Put that list in the `getWeightedAdjacencyList` method in `answers.cpp`. For example, the answer for the example graph above would look like this:

```
  return {
    {{1, 1}, {3, 10}, {4, 8}, {5, 9}},
    {{0, 1}, {2, 15}, {3, 6}, {4, 6}},
    {{1, 15}, {4, 14}},
    {{0, 10}, {1, 6}, {4, 9}, {6, 9}},
    {{0, 8}, {1, 6}, {2, 14}, {3, 9}, {6, 4}},
    {{0, 9}, {6, 3}},
    {{3, 9}, {4, 4}, {5, 3}}
  };
```

Hint: Look at the `WeightedEdge` struct in `answers.h`

Write your solution in `getWeightedAdjacencyList` in `answers.cpp`

HIGHLY RECOMMENDED TIPS FOR PART C AND D: - *Before doing this question, submit your work to GradeScope at this point to verify that your part b adjacency list is correct. This will help sanity check that your drawing / understanding of what your graph looks like is correct *before* you do all the work to run Dijkstra in part c and d. It would be a bummer to run Dijkstra manually then have to redo your work because of a faulty initial graph.* - *Before running Dijkstra manually, **make sure you understand the example below**. Fully understanding that example before doing your own will reduce the amount of times you have to submit to GradeScope and wait, and save you a lot of time!- Before moving on to doing part d, make sure that your part c is working as expected by submitting to the autograder (as your answers will be very related). This should also save you time!*

c. Dijkstra's Algorithm Dist (10 pt) Now we manually run Dijkstra's on your graph starting at the node described by your output ("Run Dijktra's starting at node 0"). Record the changes of `dist` as you run this algorithm into the `getDijkstraDist` method in `answers.cpp`. * Note that your starting node may be different depending on your Banner ID. Not every student will have the same answer. * **Every time you examine an edge (regardless of whether you update `dist`), you should add a row to your answer.** * Like in class, we visit the neighbor with the smallest label first.

For example, here is the answer for the example graph:

```
  return {
    {inf, inf, inf, inf, inf, inf, inf},  // initial
    {0, inf, inf, inf, inf, inf, inf},  // dist[s] = 0
    {0, 1, inf, inf, inf, inf, inf},  // edge: 0 1 update
    {0, 1, inf, 10, inf, inf, inf},  // edge: 0 3 update
    {0, 1, inf, 10, 8, inf, inf},  // edge: 0 4 update
    {0, 1, inf, 10, 8, 9, inf},  // edge: 0 5 update
    {0, 1, inf, 10, 8, 9, inf},  // edge: 1 0
    {0, 1, 16, 10, 8, 9, inf},  // edge: 1 2 update
    {0, 1, 16, 7, 8, 9, inf},  // edge: 1 3 update
    {0, 1, 16, 7, 7, 9, inf},  // edge: 1 4 update
    {0, 1, 16, 7, 7, 9, inf},  // edge: 3 0
    {0, 1, 16, 7, 7, 9, inf},  // edge: 3 1
    {0, 1, 16, 7, 7, 9, inf},  // edge: 3 4
    {0, 1, 16, 7, 7, 9, 16},  // edge: 3 6 update
    {0, 1, 16, 7, 7, 9, 16},  // edge: 4 0
    {0, 1, 16, 7, 7, 9, 16},  // edge: 4 1
    {0, 1, 16, 7, 7, 9, 16},  // edge: 4 2
    {0, 1, 16, 7, 7, 9, 16},  // edge: 4 3
    {0, 1, 16, 7, 7, 9, 11},  // edge: 4 6 update
    {0, 1, 16, 7, 7, 9, 11},  // edge: 5 0
    {0, 1, 16, 7, 7, 9, 11},  // edge: 5 6
    {0, 1, 16, 7, 7, 9, 11},  // edge: 6 3
    {0, 1, 16, 7, 7, 9, 11},  // edge: 6 4
    {0, 1, 16, 7, 7, 9, 11},  // edge: 6 5
    {0, 1, 16, 7, 7, 9, 11},  // edge: 2 1
    {0, 1, 16, 7, 7, 9, 11}  // edge: 2 4
  };
```

We'll be using the following Pseudocode for Dijkstra's:

```
algorithm dijkstras
  Input:
    Graph G = (V, E) where each edge (a, b) ∈ E has cost(a, b) > 0
    int s, the label to indicate the source node that we start on
  Output:
    int[] dist where dist[u] = cost of shortest path from s to u
    int[] prev where prev[u] = node label of the previous node on the shortest
      path from s to u

  dist = new int array of size |V|
  prev = new int array of size |V|
  for v in V
    dist[v] = infinity
    prev[v] = null
  dist[s] = 0

  H = new min priorty queue of the elements in V, ordered by dist
  while H.size() > 0
    u = extractMin(H)
    for each edge (u, v) in E
      if dist[v] > dist[u] + cost(u, v)
        dist[v] = dist[u] + cost(u, v)
        prev[v] = u
        decreaseKey(H, v, dist[v])

  return dist, prev
```

Write your solution in `getDijkstraDist` `answers.cpp`.

d. Dijkstra's Algorithm Prev (10 pt)

Now we manually run Dijkstra's on your graph. Record the changes of `prev` as you run this algorithm into the `getDijkstraPrev` method in `answers.cpp`. **Every time you examine an edge (regardless of whether you update `prev`), you should add a row to your answer.**

```
 // We use -1 as a special marker to indicate "no predecessor node".
 return {
   {-1, -1, -1, -1, -1, -1, -1},  // initial
   {-1, 0,  -1, -1, -1, -1, -1},  // edge: 0 1 update
   {-1, 0,  -1, 0, -1, -1, -1},  // edge: 0 3 update
   {-1, 0,  -1, 0, 0, -1, -1},  // edge: 0 4 update
   {-1, 0,  -1, 0, 0, 0, -1},  // edge: 0 5 update
   {-1, 0,  -1, 0, 0, 0, -1},  // edge: 1 0
   {-1, 0,  1, 0, 0, 0, -1},  // edge: 1 2 update
   {-1, 0,  1, 1, 1, 0, -1},  // edge: 1 4 update
   {-1, 0,  1, 1, 0, 0, -1},  // edge: 1 3 update
   {-1, 0,  1, 1, 1, 0, -1},  // edge: 3 0
   {-1, 0,  1, 1, 1, 0, -1},  // edge: 3 1
   {-1, 0,  1, 1, 1, 0, -1},  // edge: 3 4
   {-1, 0,  1, 1, 1, 0, 3},  // edge: 3 6 update
   {-1, 0,  1, 1, 1, 0, 3},  // edge: 4 0
   {-1, 0,  1, 1, 1, 0, 3},  // edge: 4 1
   {-1, 0,  1, 1, 1, 0, 3},  // edge: 4 2
   {-1, 0,  1, 1, 1, 0, 3},  // edge: 4 3
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 4 6 update
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 5 0
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 5 6
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 6 3
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 6 4
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 6 5
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 2 1
   {-1, 0,  1, 1, 1, 0, 4},  // edge: 2 4
 };
```

Write your solution in `getDijkstraPrev` `answers.cpp`.

Question 2: Kruskal's Algorithm (20 pt)

In class, we showed an abbreviated version of Kruskal's pseudocode. The actual pseudocode looks like this:

```
algorithm kruskal(G, w)
 Input: A connected undirected graph G = (V, E) with edge weights w(u, v) for all (u,v) in E
 Output: A minimum spanning tree defined by the edges X

 for all u ∈ V:
   makeset(u)
 X = {}
 Sort the edges E by weight
 for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) != find(v):
     add edge {u, v} to X
     union(u, v)
 return X
```

Recall during the Kruskal's lecture how we would make sure that adding an edge between two nodes would not create a cycle. A data structure called disjoint sets or union-find sets allow us to do this efficiently by keeping track of / updating a "set" for each connected component.

We have 3 functions for this data structure: - `makeset(x)`, which takes in a node x and creates a new set with just x in it. - `find(x)`, which takes in a node and returns which set it belongs to. - `union(x, y)`, which takes in two sets, and combines them into one big set.

a. (5 pt) How many times will we call `makeset` within Kruskal's? Put your answer in terms of |V| and |E|.

Write your answer in `q2.txt`.

b. (5 pt) How many times will we call `find` within Kruskal's? Put your answer in terms of |V| and |E|.

Write your answer in `q2.txt`.

c. (5 pt) How many times will we call `union` within Kruskal's? Put your answer in terms of |V| and |E|.

Write your answer in `q2.txt`.

d. (5 pt) Describe using your own words why `find(u) != find(v)` is the same as "if this edge doesn't cause a cycle".

Write your answer in `q2.txt`.

Question 3: Sibling Rivalry (20 pt)

Two siblings, Alice and Bob, dislike each other so much that they refuse to walk to school together from their house. Not only that, but they refuse to walk along any block that their sibling has stepped on that day. They are okay for their route to cross at corners (and luckily, both their house and the school are on corners).

a. (15 pt) We are given a directed graph G with vertices representing different corners and directed edges between them representing blocks (for simplicity, let's assume they are "one-way" blocks). We want to design an algorithm to determine whether it will be possible for Alice and Bob to get to school using max-flow. If it is possible, we want to generate the two paths. Solve this problem using a network flow network. Be sure to address each of the following in your answer:

- What is the source?
- What is the sink?
- What are the edge capacities?
- How will we know it is possible for them to each go to school?
- Assuming it's possible to find two such paths, how will we use the flow graph to find the paths they should take?

Write your answer in `q3.txt`.

b. (5 pt) The relationship tension has only increased with a few coincidental run-ins at corners. Now, they refuse to even cross corners that their sibling has crossed on that day (aside from the house and school). What graph transformation can we apply to our approach in Q3.A to represent this new constraint? Be sure to explain in detail which vertices or edges will be modified and how they will be modified.

Write your answer in `q3.txt`.

Question 4: Network Flow Cuts (10 pt)

Remember from lecture that an s-t cut of a flow network is a partitioning of nodes into two groups, one which contains the source s and the other which contains the sink t.

The image above represents the cut `{s, v} / {u, t}`. The cut capacity is the sum of all edge capacities that go from the set containing s to the set containing t. So the cut capacity shown above is `7 + 8 = 15` (note that we do not include `3`).

a. (5 pt) Define all 4 s-t cuts in the graph above and calculate their capacities. Remember that the Max-Flow Min-Cut Theorem states that the max flow in a network is equal to the capacity of the s-t cut with minimum capacity. Using this, what is the max flow of this network?

Write your answer in `q4.txt`.

b. (5 pt) Find and list flow values for each edge that will give this max-flow. What do you notice about the flows of the edges along the minimum s-t cut? Use this observation to explain why the max-flow min-cut theorem makes intuitive sense.

Write your answer in `q4.txt`**.**