Assignment 5: Spanning Trees, Shortest Path, Maximum Flow

In this assignment, you will solve problems on directed and undirected graphs and submit the solution as a single PDF through Github.

You will push all of your solutions to the Assignment 4 repository through the Github Classroom. Organize your repository in such a way that it contains:

Root – Assignment 4 directory

- Assignment 4.PDF

Note: You do not have to submit any runnable code.

There is no restriction on the programming language that you use.

For the problems, you will be implementing your code as an API through which the graders will make function calls.

Include a README.md file for the GitHub repository that has your Name as the title and a description of what is inside the repository, generally.

!! INVITATION LINK: https://classroom.github.com/a/YFsfpjNg !!

For anyone new to Github, we encourage you to use **Github Desktop** as a simple GUI for interacting: https://help.github.com/desktop/guides/getting-started-with-github-desktop/

For people who want to be more advanced and up to industry standards, you can use the Git command line:

https://git-scm.com/downloads

A very barebones introduction to the command line (very readable, I recommend it):

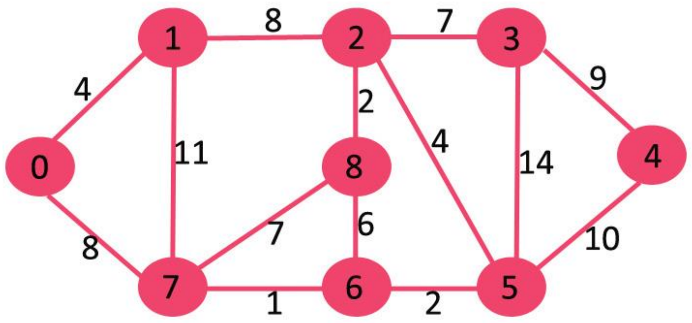
http://rogerdudler.github.io/git-gu ide/

1. For the graph pictured below, complete two things:
    a. The list of edges, in tabular form, of the MST for Kruskal's Algorithm.

edge	6,7	6,5	2,8	0,1	2,5	2,3	1,2	3,4
weight	1	2	2	4	4	7	8	9

    b. The list of edges, in tabular form, of the MST for Prim's Algorithm. Start from vertex 8.

edge	2,8	2,5	5,6	6,7	2,3	1,2	0,1	3,4
weight	2	4	2	1	7	8	4	9



2. Answer:
    a. How does Kruskal's Algorithm implement cycle detection?
        i. **The algorithm uses a union-find for cycle detection. Kruskal's Algo considers edges in ascending order of weight, without consideration to which vertices have already been observed in the spanning tree. We use union find to determine whether or not the vertices in the graph are part of the MST set. If a cycle is detected through union find, the edge is not added to the MST.**
    b. How does Prim's Algorithm implement cycle detection?
        i. **Prim's Algorithm implements cycle detection by checking whether a vertex has been visited before adding the edge containing that vertex to the spanning tree. This works because we grow outwards from a single vertex, and can check whether a vertex has been visited (or considered a candidate) in the process of building the tree to prevent a cycle.**
    c. What would happen if you tried to use Prim's Algorithm cycle detection in Kruskal's Algorithm? Demonstrate by completing the above graph for Kruskal's Algorithm but use Prim's Algorithm cycle detection. Show every step of the algorithm and show where the errors in the MST occurs.
        i. **Kruskal's will fail to generate a MST by, likely, undercounting the number of edges needed to build the tree. Every time we add an edge, we will consider both of the vertices added to be 'visited', and then jump to another part of the graph. In the process of building a tree, we will avoid adding edges that won't create cycles but will be considered to do so, creating a disjoint set for the spanning tree.**

3. What happens if you apply Kruskal and Prim's algorithm to a directed graph? In the event that one of them does not work, reproduce the algorithm and highlight where the error occurs and use it to explain your answer.
    a. **MST for directed graphs are often difficult if not impossible to generate for directed graphs, since it is entirely possible that nodes are unreachable. So we won't be able to generate a MST generally for the digraph. Going forward, if we apply Kruskal's Algorithm we will not be able to properly implement cycle detection because a union find function call on a edge vertex won't be able to tell if a vertex is reachable.**

```
public KruskalMST(EdgeWeightedGraph G) {
         // more efficient to build heap by passing array of edges
         MinPQ<Edge> pq = new MinPQ<Edge>();
         for (Edge e : G.edges()) {
             pq.insert(e);
         }

           // run greedy algorithm
           UF uf = new UF(G.V());
           while (!pq.isEmpty() && mst.size() < G.V() - 1) {
               Edge e = pq.delMin();
               int v = e.either();
               int w = e.other(v);
               if (!uf.connected(v, w)) { // v-w does not create a cycle
                   uf.union(v, w);   // merge v and w components
                   mst.enqueue(e);   // add edge e to mst
                   weight += e.weight();
               }
           }

           // check optimality conditions
           assert check(G);
     }
```

b. For Prim's Algorithm, this requires that every vertex is reachable from every other vertex. Prim's will fail if it starts at a vertex whose degree is entirely indegree with no outdegree. As such, it will create MSTs of singleton values from those vertices. Moreover, because it cannot see out from a vertex of outdegree 0, it will not be able to update potential smaller weights on the priority queue for unvisited vertices.

```
// run Prim's algorithm in graph G, starting from vertex s
private void prim(EdgeWeightedGraph G, int s) {
    distTo[s] = 0.0;
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        scan(G, v);
    }
}

// scan vertex v
private void scan(EdgeWeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v)) {
        int w = e.other(v);
        if (marked[w]) continue;         // v-w is obsolete edge
        if (e.weight() < distTo[w]) {
            distTo[w] = e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
}
```

So, if we visit a vertex with outdegree 0 from some other point in the graph, we can end up missing a spanning tree connection of smaller weight to that vertex if we happen to discover one through a different directed path from the source.

4. Reproduce the code for Prim's Algorithm and Eager Prim's Algorithm. Explain the differences in these two implementations by highlighting the changed parts(from naïve to eager) and explain how this affects space and time complexities. Explain all of the change, including the switch from vertices to edges.
    a. **Lazy Prim inserts edges of the graph starting from a given vertex to a priority queue. This requires us to store potentially every edge in the queue. Eager Prim will observe whether there are smaller possible edge weights to be updated in the priority queue depending on the most recently added vertex, instead of growing the priority queue to V-1.**

    Eager Prim only adds the edge of minimum weight to a needed vertex in the heap. This requires the heap to only have to store vertices, and vertices are *usually* much smaller than edges, reducing the size. The deletion of edges from the queue costs ElogE; the deletion of vertices from the queue costs ElogV. V << E for dense graphs, so this Is better time complexity. The space difference is O(E) for Lazy and O(V) for Eager.

```
Eager:
// run Prim's algorithm in graph G, starting from vertex s
    private void prim(EdgeWeightedGraph G, int s) {
        distTo[s] = 0.0;
        pq.insert(s, distTo[s]);
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            scan(G, v);
        }
    }

    // scan vertex v
    private void scan(EdgeWeightedGraph G, int v) {
        marked[v] = true;
        for (Edge e : G.adj(v)) {
            int w = e.other(v);
            if (marked[w]) continue;         // v-w is obsolete edge
            if (e.weight() < distTo[w]) {
                distTo[w] = e.weight();
                edgeTo[w] = e;
                if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
                else                pq.insert(w, distTo[w]);
            }
        }
    }

Lazy:

    // run Prim's algorithm
    private void prim(EdgeWeightedGraph G, int s) {
        scan(G, s);
        while (!pq.isEmpty()) {                        // better to stop when
mst has V-1 edges
            Edge e = pq.delMin();                      // smallest edge on pq
            int v = e.either(), w = e.other(v);        // two endpoints
            assert marked[v] || marked[w];
            if (marked[v] && marked[w]) continue;      // lazy, both v and w
already scanned
            mst.enqueue(e);                            // add e to MST
            weight += e.weight();
            if (!marked[v]) scan(G, v);               // v becomes part of
tree
            if (!marked[w]) scan(G, w);               // w becomes part of
tree
        }
    }

    // add all edges e incident to v onto pq if the other endpoint has not
yet been scanned
    private void scan(EdgeWeightedGraph G, int v) {
        assert !marked[v];
        marked[v] = true;
        for (Edge e : G.adj(v))
            if (!marked[e.other(v)]) pq.insert(e);
    }
```

5. Imagine a client presents to you a graph, G, and a MST, already generated, as a variable. This variable would contain the collection of vertices that form the spanning tree. Write pseudocode for linear algorithms (in terms of V /E/V and E) algorithm that can validate whether the tree is still a Minimum Spanning Tree in these events:
   a. An edge is added to the graph G

First thing we can do is add the edge directly to the spanning tree and create a cycle somewhere in the tree. We'll then do something like this:

   i. **Run DFS on the MST to find the current path to the new vertex added. We'll get updated edgeTo recursive paths.**
   ii. **Check the weight of this vertex as compared to every other weight in the spanning tree..**
   iii. **If the weight is larger than any other weight on the MST path, then this is still a MST.**
   iv. **If the weight is smaller than some other weight on the recursive path to that vertex:**
      1. **The MST is invalid.**
      2. **To fix it, delete the edge in the newly created cycle that has smaller weight than the one we added.**
   v. **If the weight is larger than the weight of any other edge in the spanning tree then the tree remains the same**

The complexity of this is O(V) at most, because we are searching for paths inside the spanning tree only. The normal complexity is O(V+2E), but in this case E = (V-1), so totally O(V+2(V-1)) = O(3V) ~ O(V).

   b. An edge not in the tree in made smaller

An edge not in the tree, but still in the graph, being made smaller will invalidate the tree. We will follow the same logic as in the previous problem:

   i. **Add the edge to the MST.**
   ii. **Run DFS to find the cycle in the new MST / the path to that edge.**
   iii. **Whichever edge is smaller in the cycle, delete it.**

The complexity for this is O(V), again.

   c. An edge in the tree is made larger

The problem here is that there is now an edge in the graph G that should be in the MST, and an edge in the MST that needs to be removed. One solution is to just re-generate the MST, but that is too time consuming. Instead:

   i. **Remove the edge from the MST.**
   ii. **Run DFS from one end point of those edges, obtaining a subtree of one portion of the MST.**
   iii. **Initialize the MST with the first edge not in that subtree -> minimum.**
   iv. **From every other edge in the graph:**
      1. **If an edge is not in that subtree && is smaller than minimum**
         a. **Remove minimum, add this edge to the tree**

This could, in the worst case, result in a disjoint graph of 1 vertex, and all other vertices in the other set. Running DFS on either and then iterating over the remaining costs you O(V+2E) in the worst case, because edges have to be checked.

6. For the graph below, complete the following for a shortest path from A to B:

   a. Bellman Ford Algorithm. Produce the contents of the distTo[] and edgeTo[] matrix for each pass of the algorithm. If the algorithm runs 3 times, you need 6 tables. Format them horizontally. Assume that the adjacency list is formatted in increasing order. This means a path out from A would have 1,2,3 in that order of consideration for its adjacency list; similarly, 1 would prioritize 3 and 4.

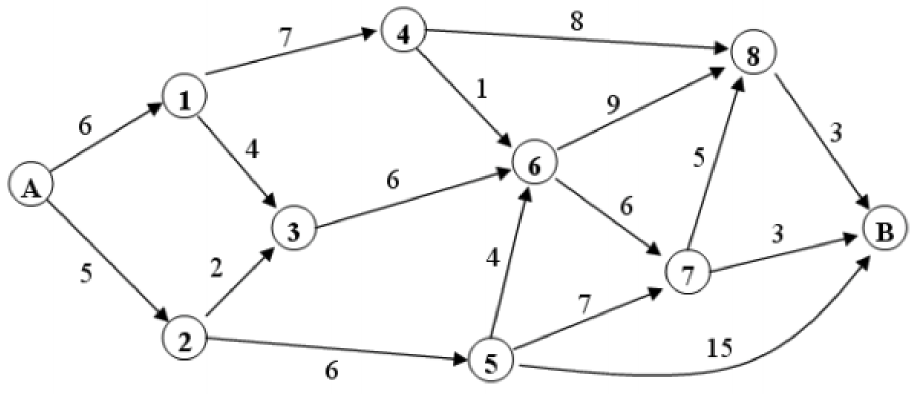
vertex	A	1	2	3	4	5	6	7	8	B
distTo	-	6	5	7	13	11	13	18	21	21
edgeTo	-	A	A	2	1	2	3	5	4	7

After this, there are no more changes in the graph since there are no cycles.

   b. Djikstra's Algorithm. For this, produce three tables: the distTo and edgeTo for the shortest path from A to B, and another table relaxCount that keeps track of how many times each edge is relaxed. For example, each edge is relaxed at least once (from infinity to the edge weight). As you pass through Djikstra's Algorithm, continually track the total # of times an edge is relaxed. Use the same adjacency list structure as above.

vertex	A	1	2	3	4	5	6	7	8	B
distTo	-	6	5	7	13	11	13	18	21	21
edgeTo	-	A	A	2	1	2	3	5	4	7
relaxCount	-	1	1	1	1	1	1	1	1	2

   c. Topological Sort Algorithm. Follow the same instructions as in Part B.



Run DFS from vertex A, assuming adjacency list structure in ascending order to get postorder:
B 8 7 6 3 4 1 5 2 A
Reversing this gives topological ordering:
A 2 5 1 4 3 6 7 8 B
We'll run our topological sort algo on this:

vertex	A	1	2	3	4	5	6	7	8	B
distTo	-	6	5	7	13	11	13	18	21	21
edgeTo	-	A	A	2	1	2	3	5	4	7
relaxCount	-	1	1	1	1	1	3	1	1	2

7. Explain why the Topological Sort Algorithm requires a DAG. What happens when you run the algorithm on a non-DAG? Reproduce the code and highlight where the error occurs.
    a. **Topological sort is not possible for graphs that contain cycles. There is no way to arrange vertices that have all outgoing edges to a later vertex.**

```
public AcyclicSP(EdgeWeightedDigraph G, int s) {
    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];

    validateVertex(s);

    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    // visit vertices in topological order
    Topological topological = new Topological(G);
    if (!topological.hasOrder())
        throw new IllegalArgumentException("Digraph is not acyclic.");
    for (int v : topological.order()) {
        for (DirectedEdge e : G.adj(v))
            relax(e);
    }
```

There is a check for topological ordering. If there wasn't, the topological.order() collection wouldn't return anything of value.

8. Why do we not implement a cycle detection for Djikstra's Algorithm, despite the fact that Prim's Algorithm for the MST -- which is nearly the same as Djikstra's – does?
    a. **Shortest path is never a cycle. There is no logic in the way Djikstra considers edge weights – it will never generate a cycle because it is trying to find the shortest path. A shortest path would never contain a cycle. However, if there are net negative cycles then the algorithm gets stuck in an infinite loop.**
9. Why does Djikstra's Algorithm fail for negative edge weights? Reproduce the primary code and highlight what causes the error.
    a. **To be more exact: Djikstra fails for negative cycles. This happens if there is a cycle of net negative weight; it would get stuck trying to continually updating the path through adding in the edge weight of the negative cycle.**

```
    // relax edge e and update pq if changed
    private void relax(DirectedEdge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
            else                pq.insert(w, distTo[w]);
        }
    }
```

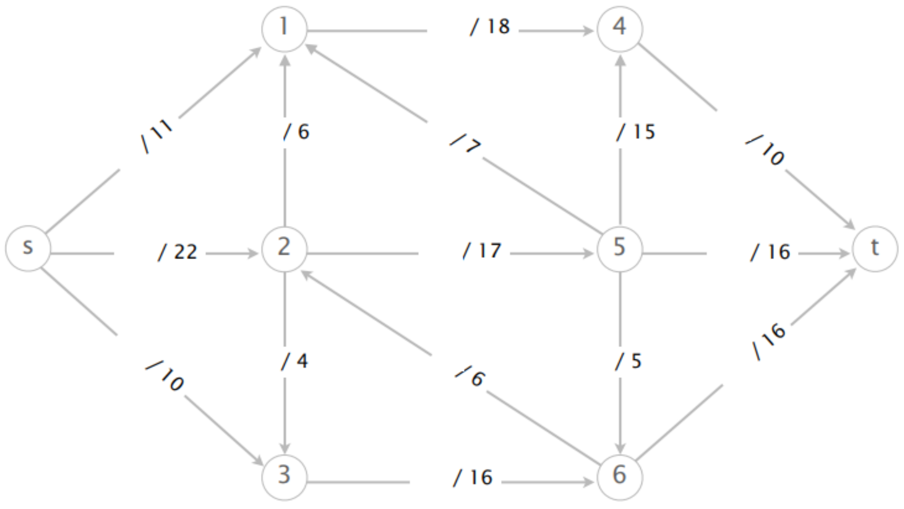
So even if a vertex has been deleted from the priority queue, if a negative cycle occurs it will be continually readded and updated.

20. Given the graph below, do the following:

a. Find the Maximum Flow from s to t using the Ford Fulkerson Algorithm. Assume the same adjacency list structure rules apply as in #6. For each pass of the algorithm, produce the value of the current maxflow and the current path(s) in the graph used to compute the flow. Format the paths as a table from the source to the sink in tabular for for each pass of the algorithm. For example, the first pass of the algorithm will produce a single flow contribution to t and one path. This should look like:

S	1	4	t
-	10/11	10/18	10/10

This is one example of a path. At the first run, we update the value of the flow and the capacity for that edge S->1->4->t. Produce a table like this for every path among every iteration of the algorithm.



vertex	s	1	4	t
flow	-	"10/11"	"10/18"	"10/10"

vertex	s	2	5	t
flow	-	"16/22"	"16/17"	"16/16"

vertex	s	3	6	t
flow	-	"10/22"	"10/16"	"10/16"

vertex	s	2	3	6	t
flow	-	"20/22"	"4/4"	"14/16"	"14/16"

vertex	s	2	5	6	t
flow	-	"21/22"	"17/17"	"1/5"	"15/16"

Maximum flow is 6-t + 5-t + 4-t -> 10+16+15 = 41.

b. What is the mincut of the above graph? Give the answer in terms of the set.

Typically you would do recursion on the residual graph network. If we consider the residual graph and run traversal, we get a postorder:
4 1 2 s -> The mincut sets are {s,1,2,4} and {3,5,6,t}.

c. What is the capacity of the mincut?

It is the same as the maximum flow. For completeness, though, this is 4->t, 1->5, 2->5, 2->6, 2->3, s->3. Since there is no flow possible from 4->5 or 2->6 or 1->5, this becomes:
4->t+2->5+2->3+s->3 = 10+17+ 4 + 10 = 41.