asatish2@ncsu.edu HW-4 200536723
Aakarsh Satish

1) **1. Purpose: Learn about Euler tours (12 points). Please solve Problem 20-3 of our textbook.**

Q1.
a) An Euler tour in a directed graph is characterized by the requirement that, in each of its cycles, every vertex must have both an incoming and outgoing edge, forming a complete cycle. This can be visualized as a combination of individual simple cycles. In a simple cycle, each vertex typically has one incoming and one outgoing edge, resulting in in-degree(v) = out-degree(v) = 1 for regular cases. Isolated vertices, on the other hand, have in-degree(v) = out-degree(v) = 0. The summation of in-degrees and out-degrees across all edges shows that in a graph G with an Euler tour, the equality in-degree(v) = out-degree(v) holds true for every vertex v in the vertex set V.

On the other hand, if for every vertex v in the set V, the in-degree of v equals the out-degree of v, we can establish the existence of an Euler tour in graph G. Now we will select a vertex u such that in-degree(u) = out-degree(u).

First we shall select an edge departing from u and reaching a vertex v where in-degree(v)=out-degree(v). At the same time, we shall follow an untraversed outgoing edge from v in a cyclic manner, ensuring that each selected edge is traversed only once. The equality in-degree(v)=out-degree(v) ensures the existence of an unvisited outgoing edge from v, ultimately leading back to ‘u’ and finishing the cycle.

This proves the presence of a vertex u in v from which initiating traversal through any edge allows for exploration of untraversed edges, resulting in a return to u after covering all edges. In the event of remaining unvisited edges and graph G being connected, these may lead to intermediary vertices, forming new cycles. These additional cycles, such as u→v→u and v→x→v, can be traversed, uncovering new edges until the entire graph is traversed.

The satisfaction of in-degree(v)=out-degree(v) for every vertex in v ensures a exploration starting from a vertex u, guaranteeing the traversal of all edges.
This proves that G has an Euler tour when in-degree(v) = out-degree(v) for each vertex v belongs to V.

b)

Algorithm:

EulerTour(G)

Input: Directed graph G = (V, E)

Output: If Euler Tour exists return Euler tour T, else return "No Euler tour"

For each vertex v in V:

    If in-degree(v) is not equal to out-degree(v):

    Return "No Euler Tour"

Choosing an arbitrary starting vertex u in V.

Find a cycle C starting from u without repetitive edges.

Let T be the initial cycle C.

Remove the edges of C from E.

While E is not empty:

    Find a vertex u in T such that there is an edge (u, v) is in E.

    Find a cycle C starting with (u, v) without repetitive edges in G = (V, E).

    Remove the edges of C from E.

    Update T to include the newly found cycle C.

End while loop

Return T.

Explanation:

- We first choose an arbitrary starting vertex u from the vertex set V.
- Next we find a cycle C starting from u without repetitive edges.
- Let T be the initial cycle C.
- Now it removes the edges of C from the edge set E.
- While the edge set E is not empty, we perform the following :
- Find a vertex u in the current circuit T such that there is an edge (u,v) in the remaining edge set E.
- Then find a cycle C starting with the edge (u,v) without repetitive edges in the current graph G=(V,E).
- Next we remove the edges of C from the edge set E.
- Finally update the circuit T to include the newly found cycle C. We continue this process until the edge set E is empty.
- The algorithm returns T as the final result.

Example: Let us take an example with graph S->U, U->V,V->W,W->S.

In this case all the incoming edges and outgoing edges of these vertices S, U,V and W are equal with 1.

Lets consider a cycle in this and it comes out to be S-U-V-W-S. Now append this to T and remove the edges.

Finally the path we get is the one which would have travelled every vertex exactly once and will be a path S->U->V->W->S. This forms the cycle and is a euler trial.

Proof of correctness:

Loop Invariant:

At the beginning of each iteration of the while loop, the graph G induced by V and E has a trail that covers all edges exactly once, and T represents a prefix of this trail.

Initialization:

- If the algorithm returns "No Euler Tour" if the ‘if’ condition of degrees fail, the loop invariant holds true because a graph with unequal in-degrees and out-degrees does not have an Eulerian trail.

Maintenance:

- In each iteration, it finds a vertex u in T with an unused outgoing edge, forms a cycle C starting with (u,v) without repetition, and then removes C's edges from E, and updates T to include C. This maintains the property, and the updated T remains a valid prefix.

Termination:

- The loop terminates when E becomes empty, implying that the graph G has an Euler trail that covers all edges exactly once. At termination, the loop invariant holds, and T represents the full Eulerpath.

Time Complexity of the algorithm:

It is O(E) as it traverse through all the edges presentin the graph. All the edges will be visited once.

2) **Purpose: Reinforce your understanding of MST algorithms and practice algorithm design (14 points). Let T be the Minimum Spanning Tree of a graph G=(V, E, w). Suppose g is connected, |E|≥|V|, and all edge weights are distinct. Denote T* the MST of G and ST(G) be the set of all spanning trees of G. A second-best MST is a spanning tree T such that w(T) =min{w(T): T ϵ ST(G)-{T*}}.**

a) (4 points) Show that T* is unique but that the second-best MST T2 need not be unique.

Suppose we have a graph on four vertices {a, b, c, d} with the following edge weights:

To prove let us take ana example with the following Adjacency matrix:

	a	b	c	d
a	-	6	9	8
b	6	-	10	7
c	9	10	-	11
d	8	7	11	-

(a-a) = -

(a-b) = 6

(a-c) = 9

(a-d) = 8

(b-b) = -

(b-c) = 10

(b-d) = 7

(c-c) = -

(c-d) = 11

(d-d) = -

The minimum spanning tree (MST) for this graph has a weight of 22. One possible MST is {(a, b), (b, d), (a, c)}.

6+7+9 = 22

Now, let's explore two different spanning trees with the second-best weight of 23:

1. Spanning Tree 1: Remove edge (a, c) from the MST, and connect the components with the edge (b, c). Weight: 6 + 7 + 10 = 23
2. Spanning Tree 2: Remove edge (b, d) from the MST, and connect with the edge (a, d). Weight: 6 + 9 + 8 = 23

Therefore, in this example, there are two different spanning trees with the second-best weight of 23.

According to the example above, two of them have the same weights on the graph which proves that second minimum spanning tree, T* might not be unique.

2 b) **(4 points) Prove that G contains an edge (u,v) ϵ T* and another edge (s,t) ∉ T* such that (T*-{(u,v)}) ∪ {(s,t)} is a second-best minimum spanning tree of G.**

In graph G, the second-best minimum spanning tree T* is formed by altering a single edge from the original minimum spanning tree T. This implies the existence of an edge $(u,v) \in T^*$ and an edge $(s,t) \notin T^*$.

In the process of constructing a minimum spanning tree from graph G, we traverse edges with the minimum weight. In the second-best MST, only one edge undergoes a change, while all other edges remain the same. This suggests the possibility of a cut for a vertex in T where an edge has a low weight. If this was the edge processed and had a lower weight then that tree would be considered as MST and not second MST.

For the MST T, the second-best MST T* is $T^* = (T - (u,v)) \cup (s,t)$. This says that T initially had a light edge (u,v), and T* retains the same set of edges but with the exchange of (s,t) for (u,v). Similarly, T* includes (s,t) but not (u,v).

In cases where the determination of the second-best MST T* involves exchanging two or more edges, the edge chosen for another cut may not be the one with the lowest weight. This would result in an increased weight of T compared to T*, rendering T* no longer the second-best MST.

Hence, utilizing a proof by contradiction, we can assert that to construct the second-best MST T*, we must select exactly one non-light edge, and the remaining edges should align with the structure of MST T.

2 c) **(6 points) Please use Kruskal's algorithm to design an efficient algorithm to compute G's second-best MST.**

Second_Kruskal_MST(G):

sort edges based on weights

Find MST T

for e in edges in MST edge list:

    Remove e from MST edge list

    Do not consider e

    Calculate MST using remaining edges (E - e) by Krushkals Algorithm

Second best MST, T* = MST with minimum weight

Explaination:

1. Sort edges based on weights:

- The first step is to arrange all edges of the graph in ascending order based on their weights. This sorting process creates a list of edges starting from the smallest weight to the largest.

2. Find MST T:

- Using an MST algorithm, the algorithm computes the Minimum Spanning Tree T for the given graph.

3. Iterate through MST edges and remove e from MST edge list and not consider e:

- The algorithm now iterates through each edge in the MST edge list. For each iteration, it temporarily removes the current edge e from the MST edge list, exploring and by excluding this particular edge from the MST.

- By excluding the edge e, the algorithm proceeds to the next step without considering this specific edge in the next MST calculation.

4. Calculate MST using remaining edges (E - e) by Kruskal's Algorithm:

- Now applying Kruskal's algorithm to the modified graph without edge e, the algorithm computes a candidate MST. This involves finding the smallest set of edges that connect all vertices in the modified graph.

5. Second best MST, T*:

The algorithm keeps track of the candidate MST obtained in step 4 if its weight is smaller than the current second-best MST (T*). This comparison is crucial for identifying the second-best MST as the algorithm explores all possibilities of removing one edge at a time from the original MST.

Once all the edges are covered, the MST tree with minimum weight is considered as the second best MST.

Example:

	a	b	c	d
a	-	6	9	8
b	6	-	10	7
c	9	10	-	11
d	8	7	11	-

The minimum spanning tree (MST) for this graph has a weight of 22. One possible MST is {(a, b), (b, d), (a, c)}.

6+7+9 = 22

The second Best MST can be calculated with the above algorithm and we shall find two second Best MST with the edges ({a,b}, {b,d} , {b,c}) and another one with the edges ({a,b} , {a,c} , {a,d}).

Time Complexity of the algorithm:

1) Sorting edges will take a time complexity of O(Elog(E)).
2) Finding Kruskal for MST takes O(E log V)
3) Removing an edge will leave V-1 edges and then using these edges to create MST using kruskal's Algorithm will take $O((V-1)E)) \approx VE$
4) Total time complexity = $O(ElgE + E\log V + VE) = O(VE)$

Correctness of the algorithm:

Loop Invariant:

For each iteration of the loop, the algorithm maintains the following invariant:

1. Initialization: Before the initial loop iteration, the algorithm has already computed the Minimum Spanning Tree (MST), denoted as T0, utilizing Kruskal's algorithm.
2. Maintenance: After each iteration of the loop, the algorithm has temporarily removed one edge (e) from the MST (T), calculated the Minimum Spanning Tree (T′) for the modified graph (excluding e), and compared T′ with the current second-best MST (T*). If T′ has a smaller weight than T*, T* is updated to be equal to T′.
3. Termination: When the loop completes all iterations, the algorithm has explored the removal of each edge from the initial MST and has updated T* to represent the second-best MST.

3) a) **Reinforce your understanding of Dijkstra's shortest path algorithm and practice algorithm design (16 points). Suppose you have a weighted, undirected graph G with positive edge weights and a start vertex s.**
a) (6 points) Describe a modification of Dijkstra's algorithm that runs (asymptotically) as fast as the original algorithm and assigns a binary value usp[u] to every vertex u in G, so that usp[u]=1 if and only if there is a unique shortest path from s to u. In addition to your modification, be sure to provide arguments for both the correctness and time-bound of your algorithm and an example.

Algorithm: Modified Dijkstra's Algorithm with USP Calculation

```
Initialization(G, s):
  For every vertex v in G:
        cost_src_v[v] = INF   # Cost to reach v from the source location
        pred[v] = null        # Predecessor of v on the shortest path
        usp[v] = 1            # Initialize unique shortest path count to 1
  cost_src_v[s] = 0           # Cost to the source vertex is 0
```

```
USP_calc(v1, v2, v3):
  If cost_src_v[v2] > cost_src_v[v1] + edge_cost[v1, v2]:
        cost_src_v[v2] = cost_src_v[v1] + edge_cost[v1, v2]
        pred[v2] = v1
        usp[v2] = usp[v1]
  Else if cost_src_v[v2] = cost_src_v[v1] + edge_cost[v1, v2]:
        usp[v2] = 0              # Set USP to 0 if there is an alternative path

Dijkstra(G, v3, s):
  Initialization(G, s)
  S = {}                  # Set of processed vertices
  While there are vertices not in S:
        v1 = Extract_Min(V - S)   # Extract the vertex with the minimum cost from the remaining
vertices
        S = S Union {v1}
        for each vertex v in the adjacency list of v1:
              If v is not in S:
                    USP_calc(v1, v, v3)
```

Explanation:

1. Initialization (G, s):

   For each vertex v in the graph G:

   Set cost_src_v[v] to ∞.

   Set pred[v] to null. Which is the predecessor array.

   Set usp[v] to 1.

   Set cost_src_v[s] to 0.This is the cost to reach the same vertex..

2. USP_calc(v1, v2, v3):

   If the cost to reach v2 which is cost_src_v[v2] is greater than the cost to reach v1 plus the cost of the edge from v1 to v2 which is edge_cost[v1, v2]:

   Then we update cost_src_v[v2] to the sum of cost_src_v[v1] and edge_cost[v1, v2].

   Then we set pred[v2] to v1.

   Set usp[v2] to the same value as usp[v1].

Else if the cost to reach v2 is equal to the previous cost:

Set usp[v2] to 0 (indicating there is an alternative path).

3. Dijkstra(G, v3, s):

    Call Initialization(G, s).

    Initialize a set S to keep track of processed vertices.

    While there are vertices not in S:

        Extract the vertex with the minimum cost from the set of remaining vertices (V - S) and call it v1.

        Add v1 to S.

        For each vertex v in the adjacency list of v1 that is not in S:

            Call USP_calc(v1, v, v3).

The algorithm begins by setting up the required data structures, initializing initial costs, and keeping track of unique shortest path counts. It then proceeds iteratively, selecting the vertex with the minimum cost, updating costs and paths to its neighboring vertices, and marking the selected vertex as processed. The USP_calc function plays a role in adjusting the unique shortest path count for each vertex based on specific conditions. This iterative process persists until all vertices have been processed. As a result, the algorithm provides the shortest paths from the source vertex to every other vertex in the graph. Additionally, it calculates the corresponding counts of unique shortest paths for each vertex.

Time Complexity:

1. Initialization (G, s):

    O(V), where V is the number of vertices.

2. USP_calc (v1, v2, v3):

    O(1) for each vertex.

3. Dijkstra(G, v3, s):

    Use a priority queue to efficiently get the minimum element: O((V+E)logV) in the worst case.

4. Overall Time Complexity:

   Considering the dominant factor, the overall time complexity is O((V+E)logV).

Correctness of the algorithm :

The modified algorithm closely resembles the original algorithm, with modifications primarily focused on the calculations of the unique shortest path counts (uspusp). Specifically, in the Initialization function, usp[v]usp[v] is initialized to 1, a constant-time operation. Similarly, in the Calculate_USP function, basic operations are performed, each taking constant time. These adjustments do not alter the overall complexity of the original algorithm.

Using Loop Invariant:

Initialization: Before the loop, S is empty, satisfying the invariant.

Maintenance: Assuming the invariant at the start of an iteration, the algorithm correctly processes v1 by adding it to S. The subsequent iteration through v1's neighbors maintains the correctness of shortest paths and unique shortest path counts.

Termination: Upon loop termination, S includes all vertices, and the algorithm has correctly determined shortest paths and unique shortest path counts for each vertex.

Hence the algorithm is correct.

Example: Let us take an example where
a -> b has weight of 7,
a-> c has a weight of 9,
b-> c has a weight of 6,
b->d has a weight of 10
c->d has a weight of 8.

Initially we initialise all the required data structures
where pred[a]=pred[b]=pred[c]=pred[d] will be null,
usp[a]=usp[b]=usp[c]=usp[d] will be true and
cost_src_v[a]=cost_src_v[b]=cost_src_v[c] will be inf for and cost_src_v[d]=0.

We shall start with A while computing the djikstra's algorithm.
For B the weight will be 7 which is the distance from a to b and the pred[b] will be a. Now for C it will be 9 which is the distance from a to c and pred[c] will be a.
Now traverse again and choose b.

For the vertex C i.e B to C. We shall calculate this only if the updated value is lower than the current value. cost_src_v[c] > cost_src_v[b] + edge_cost[b,c]: So now this will be 7+6=13 and now pred[c] = b.
For D it will be Infinite.
Next we choose vertex c.
For D it will be 13+8 = 21 and pred[d] will be c.
As all vertex are explored, the program ends.

Now calculate USP: Now the path A->B->C->D is the path with a distance of 21.

3 b) In the programming file :

```
[asatish2@engr-ras-202 ~]$ cd DijkstraFramework
[asatish2@engr-ras-202 DijkstraFramework]$ ls
DijkstraFramework   __MACOSX
[asatish2@engr-ras-202 DijkstraFramework]$ cd DijkstraFramework/
[asatish2@engr-ras-202 DijkstraFramework]$ cd python/
[asatish2@engr-ras-202 python]$ python3 DijkstraTest.py -v
testcase1 (__main__.TestCases) ... ok
testcase2 (__main__.TestCases) ... ok
testcase3 (__main__.TestCases) ... ok
testcase4 (__main__.TestCases) ... ok
testcase5 (__main__.TestCases) ... ok
testcase6 (__main__.TestCases) ... ok
testcase7 (__main__.TestCases) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.001s

OK
```