Report on NFT Graph Data Analytics
CS 5413 Assignment 2

Project by Program 1 1(Group 1)

(Satya Darapureddy, Dileep Domakonda, Monisha Sydam, Parker Hague, Vasavi Gannina, Navyasree Sangaraju, Mohammed Mukkaram, Dinesh Vadlamudi, Poovi Hosur)

Query 4:

The directed acyclic graph of buyer IDs will provide a visualization of the NFT transaction history. The directed edges between buyers will show the changes of ownership of the NFT, and the weights assigned to these edges will correspond to the prices paid for the NFT. The time tags associated with each buyer vertex will provide information on when the NFT transaction occurred, allowing for an analysis of the NFT market over time. This graph can also be used to draw conclusions about the behavior of individual buyers, such as which buyers are consistently paying higher prices or buying more NFTs. The resulting graph would represent the history of ownership of the NFT and the prices it has been sold for over time. Each vertex would represent the buyer ID of the current owner, and the edges would represent the transfer of ownership from one buyer to another. The weight associated with each edge would represent the price of the NFT at the time of the transfer. The time tag associated with each vertex would represent the time of the transfer. This graph could be used to analyze the history and trends in the price of the NFT over time as well as the buyers that have held the NFT.

Language Used: Python

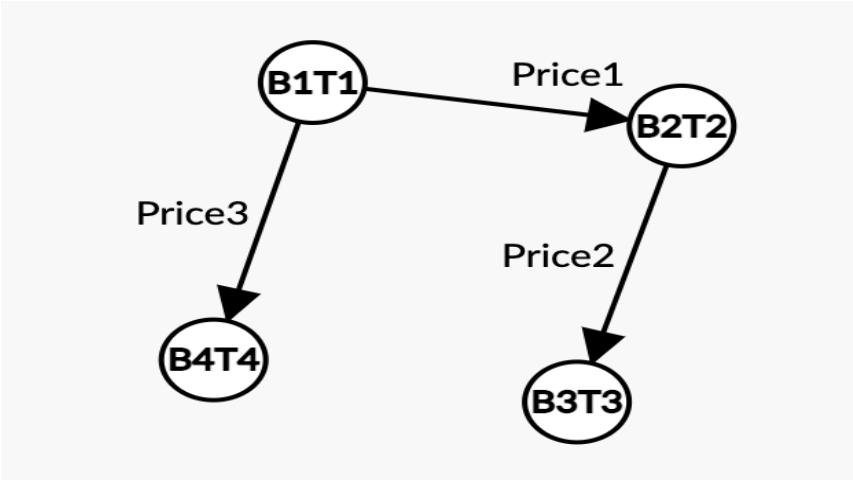
Input:

Give the input file as CSV which contains 92000 records.

Output:

Graph is generated as output which contains vertices as buyerID+UnixTimestamp and edges are built based on weights of transaction price for the buyers that are connected with token ID and NFT.

Expected output Graph:



Above is the sample graph for four buyers B1,B2,B3 and B4 with timestamps T1,T2,T3 and T4 respectively. Edge is constructed with transactions for NFT and token ID between two buyers.

Let's take B1=0x4549134864309380066061298379900077840065 (T1 = 1656633600, Price1=$153.41)

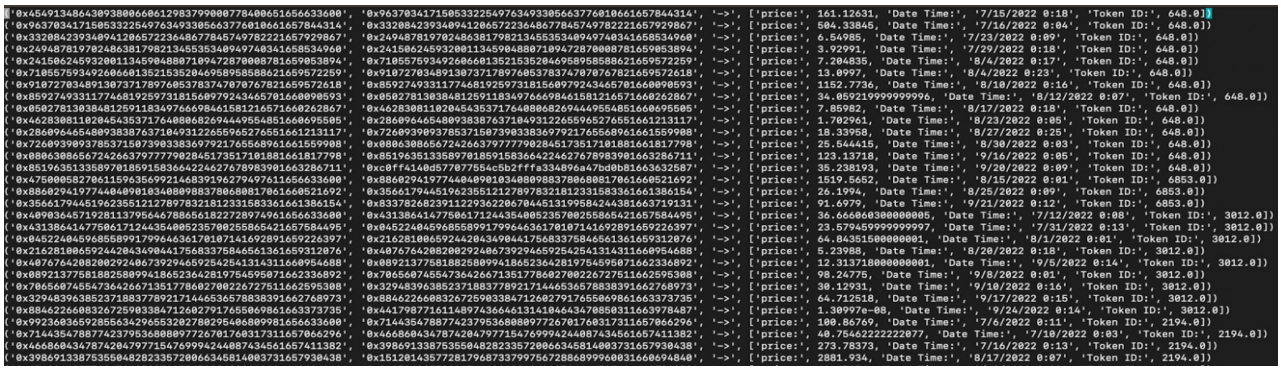
B2 = 0x475000582706115963569921468391962794976 1 (T2 = 1656633603,Price2=$150.25)

B3 = 0x40903645719281137956467886561822729 7496, (T3 = 1656633619,Price3=$26.45)

B4 = 0x992360365928556342965532027802954068099 8 (T4 = 1656633639,Price4 =$6.77)

For all the edges above, price1, price2, Price3 and price4 are the weights of edges. NFT and Token ID for above edges are WizNFT and 648 respectively. NFT and token ID will be the same for all the edges since they are connected with each other.

Expected output Adjacency List:



From the above adjacency list, we can see that two two vertices (buyer ID's) are connected with an edge of weight as transaction price with respective token ID, timestamp and price.

Adjacency matrix for the graph:

		0x4549134864309380066061298379900077840065	0x4750005827061159635699214683919627949761	0xe3502772cd69f996a82429b5a86c2f397c450f47	0x773391201767014061999775298425087731884
2	0x4549134864309380066061298379900077840065	0.0	0.14201	0.0	0.0
3	0x4750005827061159635699214683919627949761	0.0	0.0	0.205	0.0
4	0xe3502772cd69f996a82429b5a86c2f397c450f47	0.0	0.0	0.0	0.28
5	0x773391201767014061999775298425087731884	0.0	0.0	0.0	0.0
6	0x828274928371326359795237902916961367417	0.0	0.0	0.0	0.0
7	0x288034822403145655277417152250615446969	0.0	0.0	0.0	0.0
8	0x241489419565865731331897331015175252140	0.0	0.0	0.0	0.0
9	0x058390300558703079198034485768054830457	0.0	0.0	0.0	0.0
10	0x154011284048036699850748320762309834105	0.0	0.0	0.0	0.0
11	0x630837969461012198139538175391905543545	0.0	0.0	0.0	0.0
12	0x964351959652038428075067958353068519660	0.0	0.0	0.0	0.0
13	0x366075615819002484792204960535738911366	0.0	0.0	0.0	0.0
14	0x689853266099440357396997399748072979897	0.0	0.0	0.0	0.0
15	0x224521789338008582094963248651670005932	0.0	0.0	0.0	0.0
16	0x788377651954042778189209250825856860742	0.0	0.0	0.0	0.0
17	0x398830890155087749257658750391977210092	0.0	0.0	0.0	0.0
18	0x715577231058682578786497403569038515168	0.0	0.0	0.0	0.0
19	0x360045974535270775940480273291545395782	0.0	0.0	0.0	0.0

Above image represents the adjacency matrix with rows and columns as buyers ID's and the corresponding values as the price of the transactions. '0' represents that there is no edge between two corresponding buyers.

Time taken to run the code:

```
mukarram@Mohammeds-Air-3 DS-2-assignment2 % python3 query_4.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/Users/mukarram/DS-2-Final/DS-2-assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists

Starting the code

Creating the graph
The time of execution for generating the graph is: 115518133000 nano seconds
```

Time taken for generating the graph is 115518133000 nanoseconds.

Approach:

- We considered each vertex to be Buyer ID and respective timestamp to make the vertices unique.
- Edges are formed from buyer B1 to buyer B2 if only if there was a transaction taking place and change in ownership of NFT and token ID from B1 to B2. The weight of the edge will be the price of the transaction paid by the current buyer at that timestamp.
- We considered NFT and token ID should be present in common for an edge to be formed where ownership of NFT and token ID would be transferable at their respective timestamp.
- By considering token ID along with NFT, we should be able to track the buyers who are responsible for doing the price manipulation for a specific NFT.

- The buyers which do not have change of ownership of NFT will not form an edge and hence we give the value as ‘0’ to them.

Algorithm for Graph:

- Read the input csv file with 92000 records.
- Take NFT, Token ID and Buyer ID in a list.
- Convert the prices of NFT into dollars

  do get_final_price(price) //conversion of currency

- While (buyer ID+Timestamp)

  Price = TransactionPrice(Buyer1_Timestamp1)

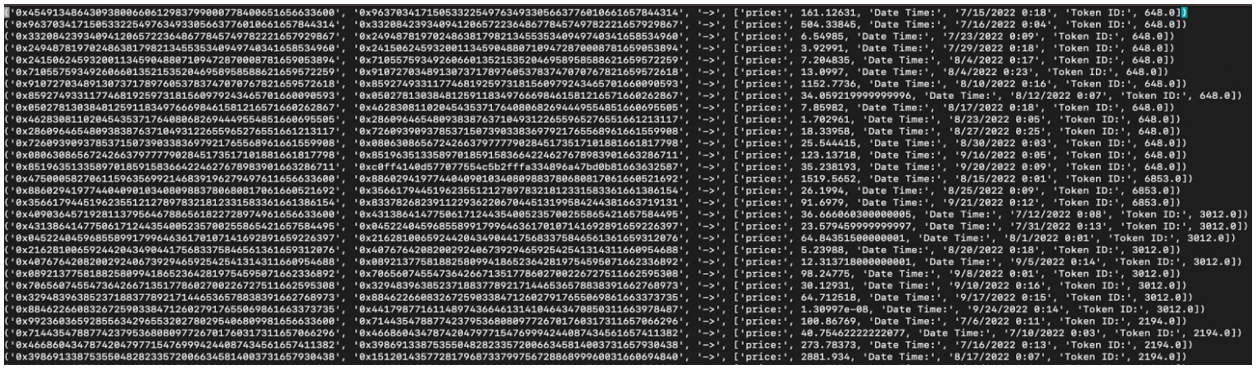
```
#storing the data based on timestamp
while i<len(input) - 1:
  temp = []
  tempnft = []
  j = i+1

  #converting the NFT price to dollars
  price = get_final_price(input[i][9])
  temp.append([str(input[i][4])+str(input[i][1]), input[i][6], price, input[i][2]])
  while(input[i][1] == input[j][1] and j < len(input)-1):
    price = get_final_price(input[j][9])
    temp.append([str(input[j][4])+str(input[i][1]), input[j][6], price, input[j][2]])
    j += 1
  data[input[i][1]] = temp
  i = j
m = 0

#checking for similar toke IDs
while m < len(tokenID):
  nft = tokenID[m]
  buyerlist = []
  for key, value in data.items():
    for i in range(0, len(value)):
      if nft == value[i][1] and any(value[i][0] in sublist for sublist in buyerlist) == False:
        buyerlist.append([value[i][0], value[i][2], value[i][1], value[i][3]])
  k = 0
  l = 1
  while k < len(buyerlist) and l < len(buyerlist):
    graph.add_edge(buyerlist[k][0], buyerlist[l][0], weight = buyerlist[l][1])
    outputdict.append([buyerlist[k][0], buyerlist[l][0], buyerlist[l][1], buyerlist[l][3], buyerlist[l][2]])
    k = l
    l += 1
  m += 1
```

Images for Input and Output:

Input Format:

```
mmukarr@csx1:~/DS2/Assignment2$ python3 query_4.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/home/mmukarr/DS2/Assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists
```

Output Format:



Query 5:

Perform DFS, topological sort and then strongly connected components. Then, output the acyclic component graph of the strongly connected components in a form of adjacency matrix.

Give your interpretation of the results in the context of buyers/NFT's/prices/time.

Input:

Merged CSV files with 92000 records will be taken as input.

Output:

The output prints the list of strongly connected components of the buyer node along with a timestamp.

Language Used: Python

Approach:

- The first step would be to derive a graph which has been implemented on query 4 and for that some portion of the 4th query has been included in query 5. Next implement the Depth First Search (DFS) algorithm to traverse the graph. This would involve starting with a given node and 'visiting' each of its adjacent nodes while keeping track of the order in which they are visited. The algorithm would continue until all nodes have been visited. Below is the code snippet for performing the Depth First Search.
- In our code we considered an empty directed graph and appended edges by performing DFS edge by edge and returning the final output graph. Below is the pseudo code for implementing DFS.

```
def dfs_get_tree(G, source=None, depth_limit=None):
    T = nx.DiGraph()
    if source is None:
        T.add_nodes_from(G)
    else:
        T.add_node(source)
    T.add_edges_from(dfs_get_edges(G, source, depth_limit))
    return T
```

```
def dfs_get_edges(G, source=None, depth_limit=None):
    if source is None:
        # edges for all components
        nodes = G
    else:
        # edges for components with source
        nodes = [source]
    visited = set()
    if depth_limit is None:
        depth_limit = len(G)
    for start in nodes:
        if start in visited:
            continue
        visited.add(start)
        stack = [(start, depth_limit, iter(G[start]))]
        while stack:
            parent, depth_now, children = stack[-1]
            try:
                child = next(children)
                if child not in visited:
                    yield parent, child
                    visited.add(child)
                    if depth_now > 1:
                        stack.append((child, depth_now - 1, iter(G[child])))
            except StopIteration:
                stack.pop()
```

Pseudo Code for DFS:

DFS(V ,E)

For each u ∈ V

Do color[u] ← WHITE

Time ← 0

For each u ∈ V

Do if color[u] = WHITE

Then DFS-VISIT(u)

DFS-VISIT(u)

color[u] ← Gray

Time ← Time + 1

D[u] ← Time

For each v ∈ Adj[u]

Do if color[v] = WHITE

Then DFS-VISIT(v)

Color[u] ← Black

Time ← time + 1

F[u] ← time

- The next step would be to perform a topological sort. This would involve arranging the nodes of the graph in a linear order such that if there is a path from node A to node B, node A appears before node B in the order. This would provide a logical order for the nodes that can be used to determine the relationships between them. Below is the snippet of the code which implements topological sort on The DFS performed graph.

```
def topological_sort(G):
    for generation in topological_generations(G):
        yield from generation
```

```
def topological_generations(G):
    multigraph = G.is_multigraph()
    indegree_map = {v: d for v, d in G.in_degree() if d > 0}
    zero_indegree = [v for v, d in G.in_degree() if d == 0]

    while zero_indegree:
        this_generation = zero_indegree
        zero_indegree = []
        for node in this_generation:
            if node not in G:
                raise RuntimeError("Graph changed during iteration")
            for child in G.neighbors(node):
                try:
                    indegree_map[child] -= len(G[node][child]) if multigraph else 1
                except KeyError as err:
                    raise RuntimeError("Graph changed during iteration") from err
                if indegree_map[child] == 0:
                    zero_indegree.append(child)
                    del indegree_map[child]
        yield this_generation

    if indegree_map:
        raise nx.NetworkXUnfeasible(
            "Graph contains a cycle or graph changed during iteration"
        )
```

Pseudo code for Topological Sort:

Topological_sort(G) //G is the graph obtained after calling DFS(G)

Return the vertices of decreasing order to the list

- The last step would be to find the strongly connected components of the graph. This would involve grouping nodes together that are connected to each other. The result would be a set of components, each of which would have nodes that are connected to each other but not to nodes in other components. Below is the code snippet to generate the strongly connected components.

```
def strongly_connected_components(G):
    preorder = {}
    lowlink = {}
    scc_found = set()
    scc_queue = []
    i = 0  # Preorder counter
    neighbors = {v: iter(G[v]) for v in G}
    for source in G:
        if source not in scc_found:
            queue = [source]
            while queue:
                v = queue[-1]
                if v not in preorder:
                    i = i + 1
                    preorder[v] = i
                done = True
                for w in neighbors[v]:
                    if w not in preorder:
                        queue.append(w)
                        done = False
                        break
                if done:
                    lowlink[v] = preorder[v]
                    for w in G[v]:
                        if w not in scc_found:
                            if preorder[w] > preorder[v]:
                                lowlink[v] = min([lowlink[v], lowlink[w]])
                            else:
                                lowlink[v] = min([lowlink[v], preorder[w]])
                    queue.pop()
                    if lowlink[v] == preorder[v]:
                        scc = {v}
                        while scc_queue and preorder[scc_queue[-1]] > preorder[v]:
                            k = scc_queue.pop()
                            scc.add(k)
                        scc_found.update(scc)
                        yield scc
                    else:
                        scc_queue.append(v)
```

- The output of this process would be an adjacency List that represents the acyclic component graph. This matrix would show the relationships between the nodes in a visual form. The rows of the matrix would represent the nodes, while the columns would represent the edges connecting them. Each cell in the matrix would contain a value indicating whether or not an edge exists between two nodes.

Images of Input and Output:

Input Format:

```
mmukarr@csx1:~/DS2/Assignment2$ python3 query_5.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/home/mmukarr/DS2/Assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists
```

Output Format:

Output on Console:

Just saves the output in a file called "query5_output.txt" as shown below

```
mmukarr@csx1:~/DS2/Assignment2$ python3 query_4.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/home/mmukarr/DS2/Assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists

Starting the code

Creating the graph
Exporting the output

The output file query4_output.txt has been generated at /home/mmukarr/DS2/Assignment2

mmukarr@csx1:~/DS2/Assignment2$ ls
nft_dataset.csv  query4_output.txt  query_4.py  query_5.py  query_7.py
```

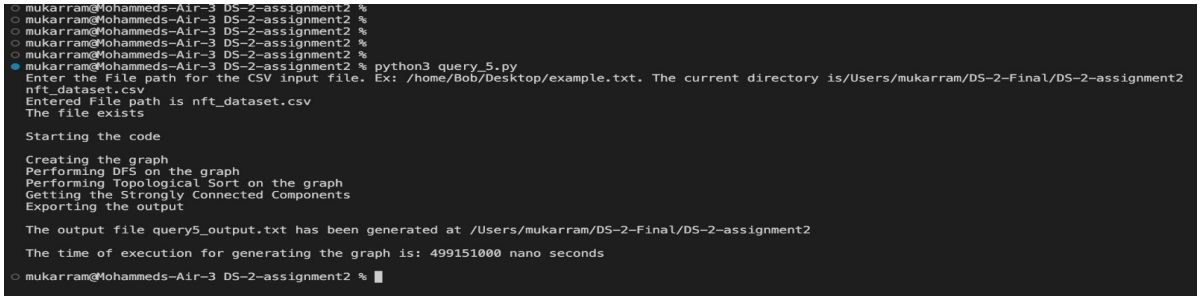
Below is the Output screenshot saved to the file "query5_output.txt". The output prints the list of strongly connected components of the buyer node along with a timestamp. After performing the DFS and topological sort to perform SCC we get individual nodes as strongly connected components.

```
Scc 0
("{'0xc0ff4140d577077554c5b2fffa334896a47bd0b8', 'Unix timestamp:', '1663632587'}")
Scc 1
("{'0x8519635133589701859158366422462767898390', 'Unix timestamp:', '1663286711'}")
Scc 2
("{'0x0806308656724266379777779028451735171018 8', 'Unix timestamp:', '1661817798'}")
Scc 3
("{'0x726093909378537150739033836979217655 6896', 'Unix timestamp:', '1661559908'}")
Scc 4
("{'0x2860964654809383876371049312265596527655', 'Unix timestamp:', '1661213117'}")
Scc 5
("{'0x4628308110204543537176408068269444955485', 'Unix timestamp:', '1660695505'}")
Scc 6
("{'0x0502781303848125911834976669846158121657', 'Unix timestamp:', '1660262867'}")
Scc 7
("{'0x8592749331177468192597318156097924346570', 'Unix timestamp:', '1660090593'}")
Scc 8
("{'0x9107270348913073717897605378374707076782', 'Unix timestamp:', '1659572618'}")
Scc 9
("{'0x7105575934926066013521535204695895858862', 'Unix timestamp:', '1659572259'}")
Scc 10
("{'0x2415062459320011345904880710947287000878', 'Unix timestamp:', '1659053894'}")
Scc 11
("{'0x5007798565741762542481027068938698507136', 'Unix timestamp:', '1664497067'}")
Scc 12
("{'0x8778341410900168768056767877039262292471', 'Unix timestamp:', '1664064683'}")
Scc 13
("{'0x6798233648189564281584628712569216315432', 'Unix timestamp:', '1663546631'}")
Scc 14
("{'0x665084f8c21bff391bf38f290ad912b643ee6cd5', 'Unix timestamp:', '1662509465'}")
Scc 15
("{'0x9773711181739653448106635685223173639158', 'Unix timestamp:', '1661126593'}")
```

Time taken for the execution of query 5:

Below image shows the execution time for query5.

Time taken for the execution time for query5 is 49915100 nanoseconds.



Query 6:

Ignore the directions on the edges, and then identify a minimum spanning tree and a maximum spanning tree. Then, output each tree in a form of adjacency matrix along with min or max total and the NFTs involved.

Give your interpretation of the results in the context of buyers/NFT's/prices/time.

Input:

The input is the single merged csv file of 92000 records.

Output:

The Minimum Spanning Tree and a Maximum Spanning Tree costs are shown in the output. Here, the buyers serve as nodes/vertices, and the transaction prices between the two buyers which are termed as vertices serve as edges by connecting the two vertices.

Language Used: JAVA

Approach:

- At first, we have given a merged csv file of 92000 records as input and then generated the directed acyclic graph and adjacency list based on the query4 criteria (which is basically the output of query4).
- From the adjacency list, we have constructed an undirected graph and then found out the connected components based on Token ID as the obtained undirected graph has many disconnected subgraphs.
- The algorithm we have used to find out the Minimum and Maximum Spanning Tree for each connected component is Kruskal's. We have chosen this algorithm instead of Prims because Prims

doesn't work if the input graph is a disconnected graph and doesn't give the minimum/maximum spanning tree.

- In order to find the edges with the least weight for the lowest spanning tree and the edges with the most weight for the maximum spanning tree in each connected component, we implemented Kruskal's. For each connected component, there will be one Minimum Spanning Tree and one Maximum Spanning Tree.
- Inside the Kruskal's algorithm, Quick Sort is used to get the least weight edge in order to find the Minimum Spanning Tree and the total cost of it.
- We have used the reverse sorted edge on the above result to get the maximum weight edge in order to find the Maximum Spanning Tree and the total cost of it.
- At last, we will get the Minimum and Maximum Spanning Trees costs for each connected component in the undirected graph.

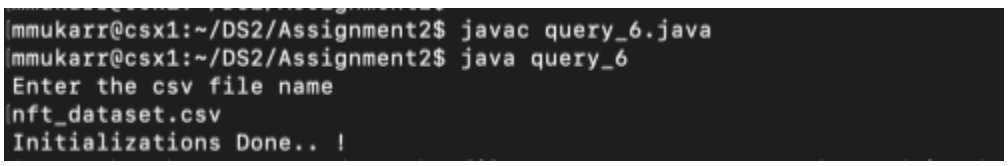
Pseudo Code to find Minimum and Maximum Spanning tree:

Kruskal(V,E,w)

```
A <- ∅
For each vertex v ∈ V
        do MAKE-SET(v)
Sort E into nondecreasing order by weight w
for each (u,v) taken from the sorted list
        do if FIND-SET(u) ≠ FIND-SET(v)
                then A <- A ∪ {(u,v)}
                        UNION(u,v)
return A
```

Images of Input and Output:

Input Format:



Output Format:

Output on Console:

Displaying the minimum and maximum spanning tree with their respective costs as shown below on the console.

```
mmukarr@csx1:~/DS2/Assignment2$ javac query_6.java
mmukarr@csx1:~/DS2/Assignment2$ java query_6
Enter the csv file name
nft_dataset.csv
Initializations Done.. !
0x303810239855306008971452122029346962247 4    0.0
0x685035217134399535578138464303683066729 5    528864.0
Output has been exported to the file query6_output.txt and saved in the current directory
```

The above highlighted output screenshot indicates that the two vertices represent the root nodes of the minimum and maximum spanning trees respectively along with their costs.

Also, output for query 6 has been saved in a text file(query6_output.txt). Below screenshot represents the minimum and spanning tree costs for the respective buyers and it is the saved to the file “query6_output.txt”.

vertex	min span tree cost	max span tree cost
0x8e41d80e78980e2da7dc06eda2a3da2009446c0c	102.17766	102.17766
0x76459039153382118382869289818452789946 98	51.08883	51.08883
0x3aa2481045fdcd12157a26532e3205a6c4ceb0be	5.763868	5.763868
0x3ad353e2845b96fb1b58c1ca6cda46b870de8d6f	13.2216	13.2216
0xa03bc3a6812d3ba3ea549dccc1469fa19320fc45	5.23988	5.23988
0x33d5cc43debe407d20dd360f4853385135f97e9d	286.88343000000003	286.88343000000003
0x32d22632e346058773778c2ed9ba9207d2c819d2	392.991	392.991
0x2fdd813ea3e49efed854a77a61fc7743959720b1	301.29310000000004	301.29310000000004
0xb8ae3d632cca3a1f5568ebccbbab47d6e8df5778	10.47976	10.47976
0x4560bb2b9000751286bf9adf9d9c33be00a654d5	108.41712000000001	108.41712000000001
0x7840677643951913596962797950298238522278	320.94265	320.94265
0x6d0b84a01e582d5c1f95660add37155bda42ec72	610.924	610.924
0x928e87ae4af2883b342962b317a1b98cd401fcc6	708.3080762533835	708.3080762533835
0xf19e64b88f2240c453e61d1c8d814a7c2c91e1a5	1.30997	1.30997
0x163485912909556191167179223077213580404 8	98.11675299999999	98.11675299999999
0x39c0b6d007bcd54814af586f138c0a1414b38918	157.1964	157.1964
0x581342da6b9b46ead49b09b5d3601d8543361b71	13.0997	13.0997
0x73d604e0493a09dca4b705f6205f4a7bc64f813e	716.6265625338351	716.6265625338351

The above output screenshot indicates the minimum and maximum spanning tree cost for each source vertex(which are connected components).

Query 7:

Identify a shortest path tree from an arbitrary buyer. Then, output the shortest path tree in the form of an adjacency matrix filled with the shortest weight sum and the NFT’s on the path. Give your interpretation of the results in the context of buyers/NFT’s/prices/time.

Input:

Previously generated graphs with nodes as Buyer IDs connected with edges and the weight of edges as transaction price is taken as input to generate the shortest path tree.

Output:

The shortest path tree for an arbitrary(random) buyer with the lowest edge weight of transaction price and the NFTs is derived.

Language Used: Python

Approach:

The algorithm used to find the shortest path tree is Dijkstra's. The above-generated graph is provided as an input which has the vertex as Buyer ID and the weight of the edge as the price of the transaction paid by the buyer. Initially, we chose a source to traverse through the graph to find the shortest path. The edge with the lowest edge weight is tracked and updated every time the graph is traversed. One finding the lowest edge weight between the Buyer IDs, the path is stored. The process continues until we visit all the buyer IDs and keep track. Thus, we derive the final path that connects the source node and the destination node, possibly the shortest way to reach each buyer. Please find the below code snippet.

```
def single_source_dijkstra(G, source, target=None, cutoff=None, weight="weight"):
    return multi_source_dijkstra(
        G, {source}, cutoff=cutoff, target=target, weight=weight
    )
```

```
def multi_source_dijkstra(G, sources, target=None, cutoff=None, weight="weight"):
    if not sources:
        raise ValueError("sources must not be empty")
    for s in sources:
        if s not in G:
            raise nx.NodeNotFound(f"Node {s} not found in graph")
    if target in sources:
        return (0, [target])
    weight = weightFunction(G, weight)
    paths = {source: [source] for source in sources}  # dictionary of paths
    dist = _dijkstra_multisource(
        G, sources, weight, paths=paths, cutoff=cutoff, target=target
    )
    if target is None:
        return (dist, paths)
    try:
        return (dist[target], paths[target])
    except KeyError as err:
        raise nx.NetworkXNoPath(f"No path to {target}.") from err
```

```python
def _dijkstra_multisource(G, sources, weight, pred=None, paths=None, cutoff=None, target=None):
    G_succ = G._adj  # For speed-up (and works for both directed and undirected graphs)
    push = heappush
    pop = heappop
    dist = {}  # dictionary of final distances
    seen = {}
    # fringe is heapq with 3-tuples (distance,c,node)
    # use the count c to avoid comparing nodes (may not be able to)
    c = count()
    fringe = []
    for source in sources:
        seen[source] = 0
        push(fringe, (0, next(c), source))
    while fringe:
        (d, _, v) = pop(fringe)
        if v in dist:
            continue  # already searched this node.
        dist[v] = d
        if v == target:
            break
        for u, e in G_succ[v].items():
            cost = weight(v, u, e)
            if cost is None:
                continue
            vu_dist = dist[v] + cost
            if cutoff is not None:
                if vu_dist > cutoff:
                    continue
            if u in dist:
                u_dist = dist[u]
                if vu_dist < u_dist:
                    raise ValueError("Contradictory paths found:", "negative weights?")
                elif pred is not None and vu_dist == u_dist:
                    pred[u].append(v)
            elif u not in seen or vu_dist < seen[u]:
                seen[u] = vu_dist
                push(fringe, (vu_dist, next(c), u))
                if paths is not None:
                    paths[u] = paths[v] + [u]
                if pred is not None:
                    pred[u] = [v]
            elif vu_dist == seen[u]:
                if pred is not None:
                    pred[u].append(v)
    return dist
```

Pseudo Code to find the shortest path tree:

DIJKSTRA(V, E, w, s)

Init-Single-Source(V, s)
S ← ∅
Q ← V
while Q ≠ ∅
    Do u ← Extract-Min (Q)
        S ← S U {u}
        For each vertex v ∈ Adj[u]
            Do Relax (u, v ,w)

Images of Input and Output:

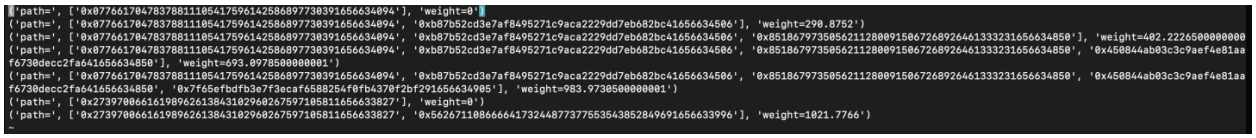
Input Format:

```
mmukarr@csx1:~/DS2/Assignment2$ python3 query_7.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/home/mmukarr/DS2/Assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists

Starting the code

Creating the graph
```

Output on console:

```
mmukarr@csx1:~/DS2/Assignment2$ python3 query_7.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/home/mmukarr/DS2/Assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists

Starting the code

Creating the graph
Finding the shortest path by using a arbitrary Node 0x07766170478378811105417596142586897730391656634094
Exporting the output

The output file query7_output.txt has been generated at /home/mmukarr/DS2/Assignment2
```

Also, output for query 7 has been saved in a text file(query7_output.txt). Below is the output screenshot saved to the file "query7_output.txt".



Time taken to run the code:

```
mukarram@mohammeds-air-3 DS-2-assignment2 % python3 query_7.py
Enter the File path for the CSV input file. Ex: /home/Bob/Desktop/example.txt. The current directory is/Users/mukarram/DS-2-Final/DS-2-assignment2
nft_dataset.csv
Entered File path is nft_dataset.csv
The file exists

Starting the code

Creating the graph
Finding the shortest path by using a arbitrary Node 0x84157675839900247711785183009464972390821657238828
Exporting the output

The output file query7_output.txt has been generated at /Users/mukarram/DS-2-Final/DS-2-assignment2

The time of execution for generating the graph is: 223000 nano seconds

mukarram@mohammeds-air-3 DS-2-assignment2 %
```

Time taken for execution of query 7 is 223000 nanoseconds.