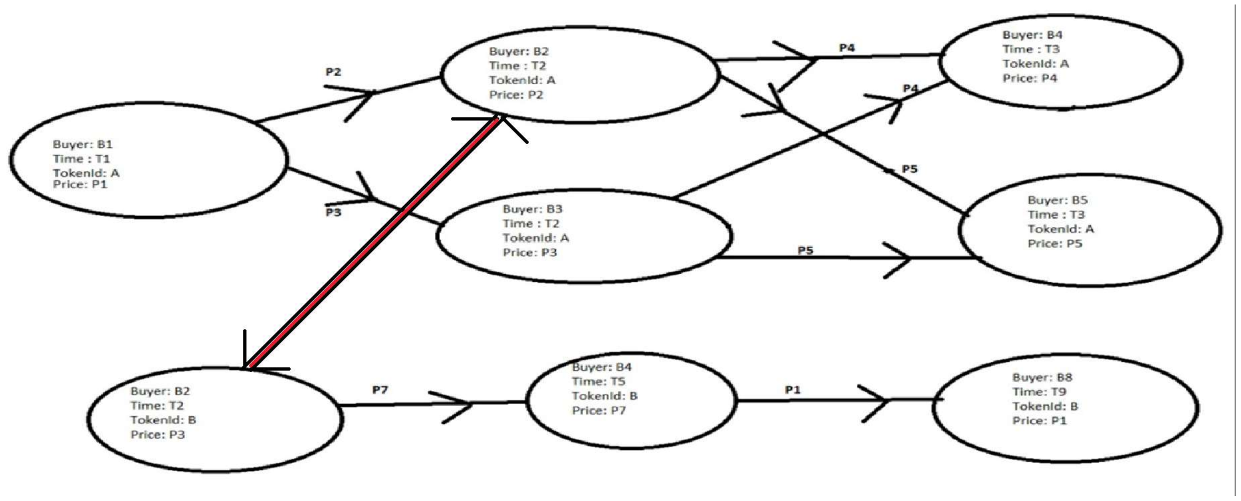
4. A directed acyclic graph of buyer ID's as vertices is to be built. A directed edge from a buyer *i* to another buyer *j* is to be established if there was a transaction to trigger a change of the ownership of the NFT from *i* to *j* at the price of the NFT to be assigned as a weight to the edge. Each buyer vertex will record the price of the current buyer paid and carry a time (not the finish time in DFS) tag so no cycles are formed.

Output the graph in the form of an adjacency matrix.

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

Approach :

We have BuyerId, Timestamp and TokenId to consider for establishing unique connections between nodes. We will consider the price of an NFT as the weight of an edge. From the given data we will have multiple graphs for every BuyerId and Timestamp connection among nodes. Firstly, we find one buyerId Timestamp pair and connect all the nodes with similar tokenId's and assign its price as the weight of its edge. There are cases where one buyerId timestamp pair is connected to nodes with different tokenId. From below picture we have a node with buyer: B1 and time: T1 connected to a node with buyer: B2 and time: T2 and to a node with buyer: B3 and time: T2, both connected nodes had tokenId same as parent node i.e. TokenId: A, We can also see that this parent node is connected to a node with buyer: B4 and time: T5 and tokenId: B which is different from the tokenId of the parent node. Hence we are taking the node uniqueness with respect to Buyer + timestamp.



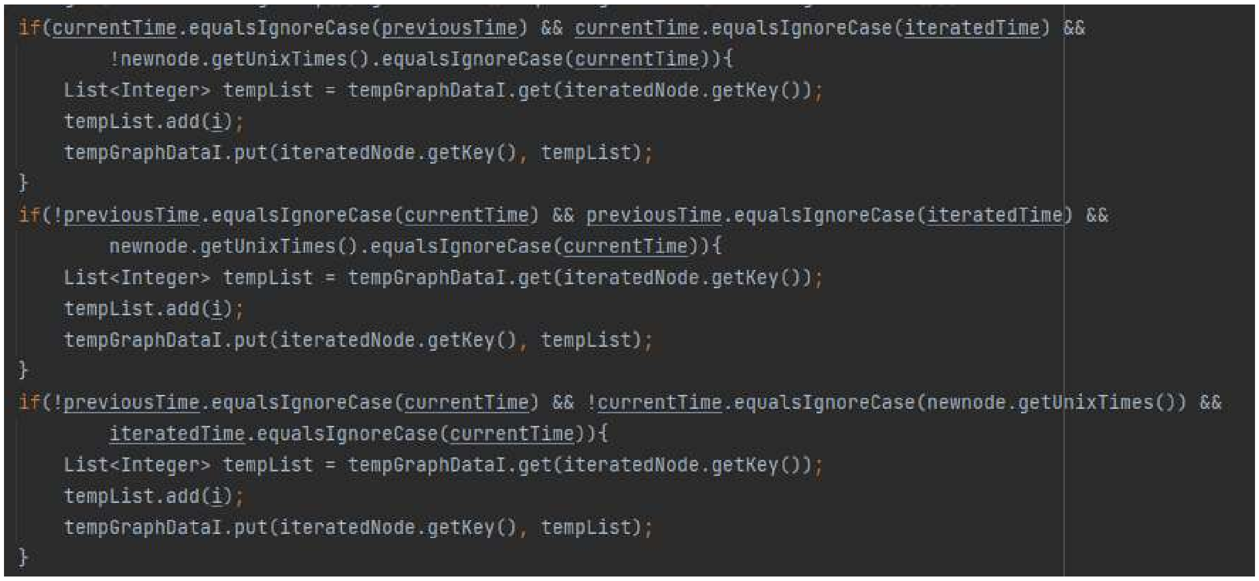
From above picture we can say that B2T2 (Buyer2_TimeStamp2) is connected to the nodes that belongs to different tokenIDs i.e. TokenId A and TokenId B. We depicted this scenario with the red line that indicates the same node(B2T2) connecting to nodes with different TokenIds (B4T3, B5T3, B4T5).

Code:

First we are taking the whole data into a list. By traversing the list, we are adding nodes first based on tokenId to hashmap and while adding the nodes, we are checking the following conditions to form an edge between 2 nodes

We are forming edge between two nodes if the nodes satisfy one of the given conditions taken are:

we will have currentTimeStamp which states the latest timestamp of respective tokenId, previous timestamp which is before timestamp of current timestamp of its respective tokenId and the timestamp of new node to be added.

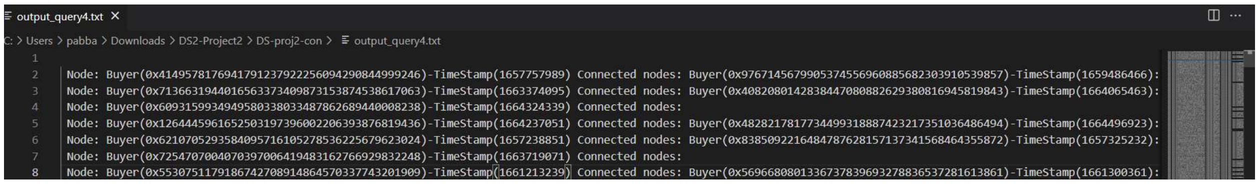


```java
if(currentTime.equalsIgnoreCase(previousTime) && currentTime.equalsIgnoreCase(iteratedTime) &&
        !newnode.getUnixTimes().equalsIgnoreCase(currentTime)){
    List<Integer> tempList = tempGraphDataI.get(iteratedNode.getKey());
    tempList.add(i);
    tempGraphDataI.put(iteratedNode.getKey(), tempList);
}
if(!previousTime.equalsIgnoreCase(currentTime) && previousTime.equalsIgnoreCase(iteratedTime) &&
        newnode.getUnixTimes().equalsIgnoreCase(currentTime)){
    List<Integer> tempList = tempGraphDataI.get(iteratedNode.getKey());
    tempList.add(i);
    tempGraphDataI.put(iteratedNode.getKey(), tempList);
}
if(!previousTime.equalsIgnoreCase(currentTime) && !currentTime.equalsIgnoreCase(newnode.getUnixTimes()) &&
        iteratedTime.equalsIgnoreCase(currentTime)){
    List<Integer> tempList = tempGraphDataI.get(iteratedNode.getKey());
    tempList.add(i);
    tempGraphDataI.put(iteratedNode.getKey(), tempList);
}
```

If it satisfies one of the conditions, we are setting the edge between the node and the new node.

Now, we get tokenId independent graphs. There might be buyers at the same timestamp having different tokenIds. As we consider node uniqueness as “Buyer + Timestamp”, we traverse each tokenId independent graph and combine the nodes based on occurrence of same buyer + timestamp for different tokenIds.

Output:



output_query4.txt
C: > Users > pabba > Downloads > DS2-Project2 > DS-proj2-con > output_query4.txt

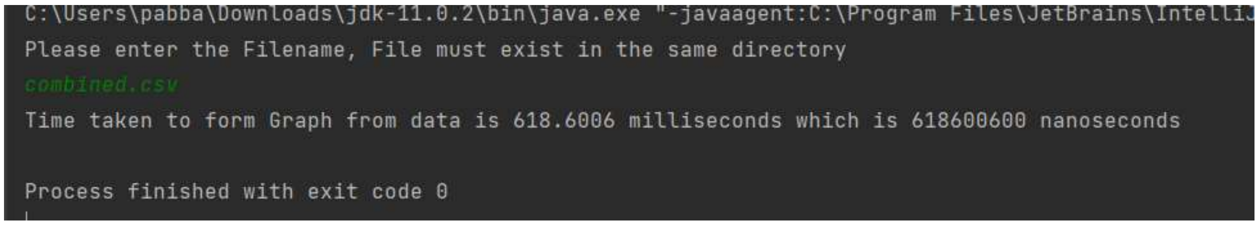
```

Node: Buyer(0x4149578176941791237922256094290844999246)-TimeStamp(1657757989) Connected nodes: Buyer(0x9767145679905374556960885682303910539857)-TimeStamp(1659486466):
Node: Buyer(0x7136631944016563373409873153874538617063)-TimeStamp(1663374095) Connected nodes: Buyer(0x4082080142838447080882629380816945819843)-TimeStamp(1664065463):
Node: Buyer(0x6093159934949580338033487862689440008238)-TimeStamp(1664324339) Connected nodes:
Node: Buyer(0x1264445961652503197396002206393876819436)-TimeStamp(1664237051) Connected nodes: Buyer(0x4828217817734499318887423217351036486494)-TimeStamp(1664496923):
Node: Buyer(0x6210705293584095716105278536225679623024)-TimeStamp(1657238851) Connected nodes: Buyer(0x8385092216484787628157137341568464355872)-TimeStamp(1657325232):
Node: Buyer(0x7254707004070397006419483162766929832248)-TimeStamp(1663719071) Connected nodes:
Node: Buyer(0x5530751179186742708914864570337743201909)-TimeStamp(1661213239) Connected nodes: Buyer(0x5696680801336737839693278836537281613861)-TimeStamp(1661300361):
```

For the output we are creating a text file “output_query4.txt” that consists of an adjacency list that displays the buyerId timestamp pair and price with which it is connected to the parent node. For every parent node we print buyerId timestamp and the price of the edge. Below is an example format of the adjacency list.

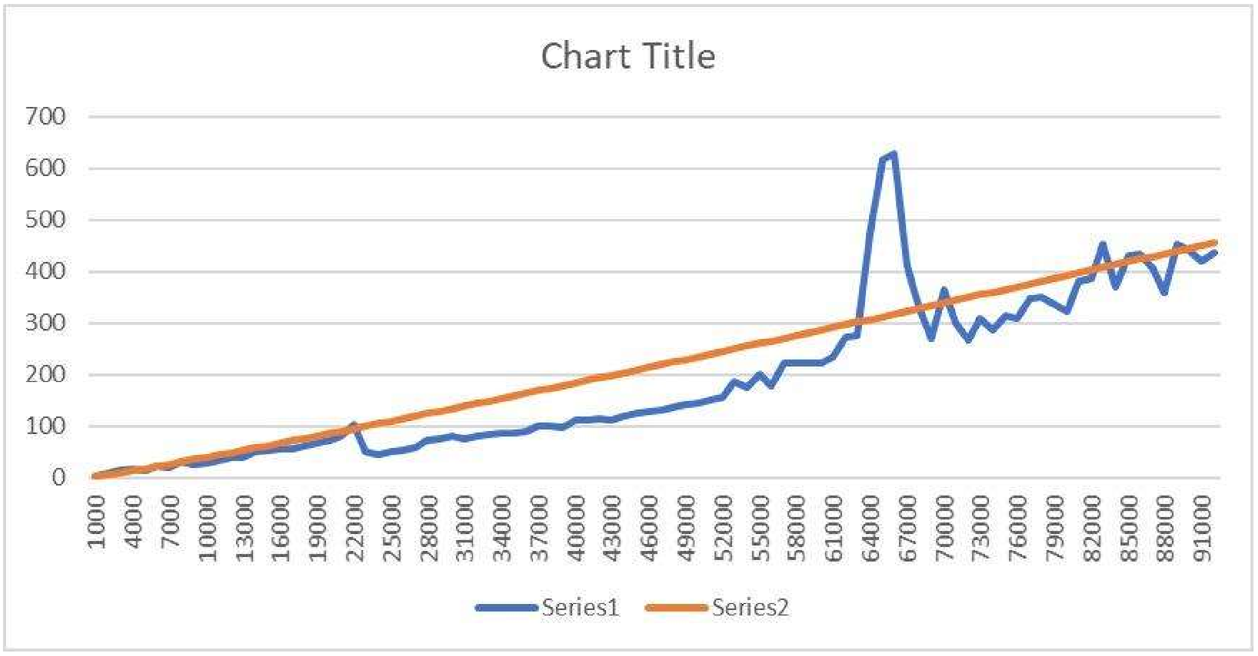
B1( )T1( ) connected nodes: B2( )T2( )p2( ), B3( )T2( )p4( )

From above B1, B2, B3 are buyerId's and T1,T2,T3 are timestamp and P1,P2,P3 are prices.

Time complexity:



```
C:\Users\pabba\Downloads\jdk-11.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
Please enter the Filename, File must exist in the same directory
combined.csv
Time taken to form Graph from data is 618.6006 milliseconds which is 618600600 nanoseconds

Process finished with exit code 0
```



Above picture depicts a graph on asymptotic run time on NFT transaction dataset for every 1000 lines increment.

Blue line indicates actual time and the orange line indicates the asymptotic run time.

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.

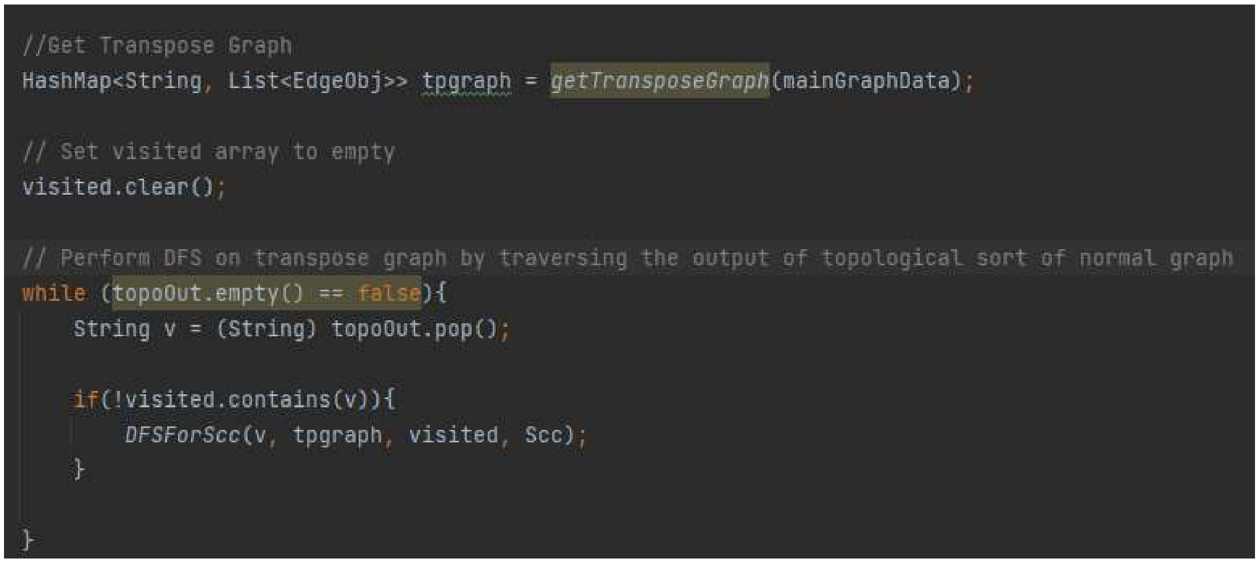
Approach:

DFS : We have performed the Depth First Search on given data and are displaying the path in the output.

Topological sort : We have performed Topological sort on given data and are displaying the path in the output.

Strongly connected components : We performed DFS on transpose graph data by taking the root nodes from the output of topological sort (formed from input graph) to extract strongly connected components and displaying it in the form of buyerId and TimeStamp pairs.

Code:



```
//Get Transpose Graph
HashMap<String, List<EdgeObj>> tpgraph = getTransposeGraph(mainGraphData);

// Set visited array to empty
visited.clear();

// Perform DFS on transpose graph by traversing the output of topological sort of normal graph
while (topoOut.empty() == false){
    String v = (String) topoOut.pop();

    if(!visited.contains(v)){
        DFSForScc(v, tpgraph, visited, Scc);
    }

}
```

The above code snippet explains how we extracted strongly connected components.

Output:

After performing Depth First search on given data we are printing the path as an output in the form of buyerId and timestamp. In the same way we have performed Topological sort and are printing the output path in the form of buyerId and timestamp.(ex; B1T1 -> B2T2 -> B3T2 -> …. Where B1, B2, B3 are buyerId's and T1,T2 are timestamps). As for strongly connected components we are printing all the nodes in the form of buyerId and timestamp (ex; B1T1 where B1 is the buyerId and T1 is the timestamp).

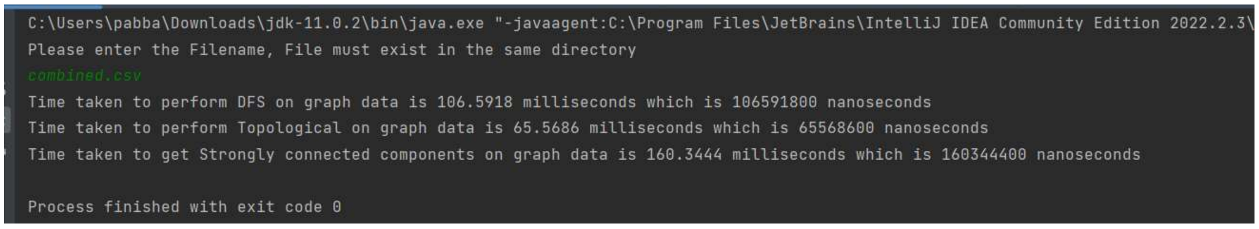
Running the code will produce three text files :

-> output_query5_DFS.txt text file consists of the output path produced by Depth first search algorithm.

-> output_query5_Topological.txt file consists of the output path produced by sorting the dataset with topological sort algorithm.

-> output_quert5_SCC.txt file consists of all strongly connected components from the NFT transaction dataset. Each line in the output file will represent one strongly connected component in the form of Buyer( ) - TimeStamp( ).

Time complexity:

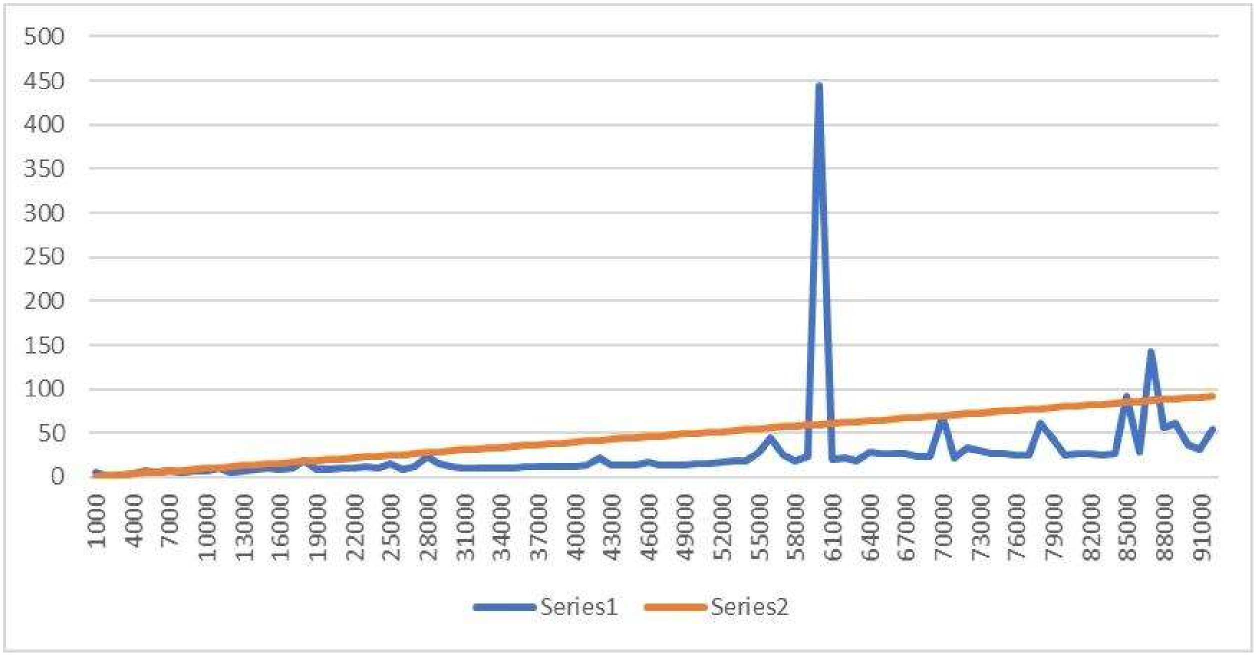


```
C:\Users\pabba\Downloads\jdk-11.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\
Please enter the Filename, File must exist in the same directory
combined.csv
Time taken to perform DFS on graph data is 106.5918 milliseconds which is 106591800 nanoseconds
Time taken to perform Topological on graph data is 65.5686 milliseconds which is 65568600 nanoseconds
Time taken to get Strongly connected components on graph data is 160.3444 milliseconds which is 160344400 nanoseconds

Process finished with exit code 0
```

DFS: O(V+E) where E is the edge and V is the vertices of the graph.

Below graph represents asymptotic run time on NFT transaction dataset for every 1000 lines increment while performing DFS algorithm.

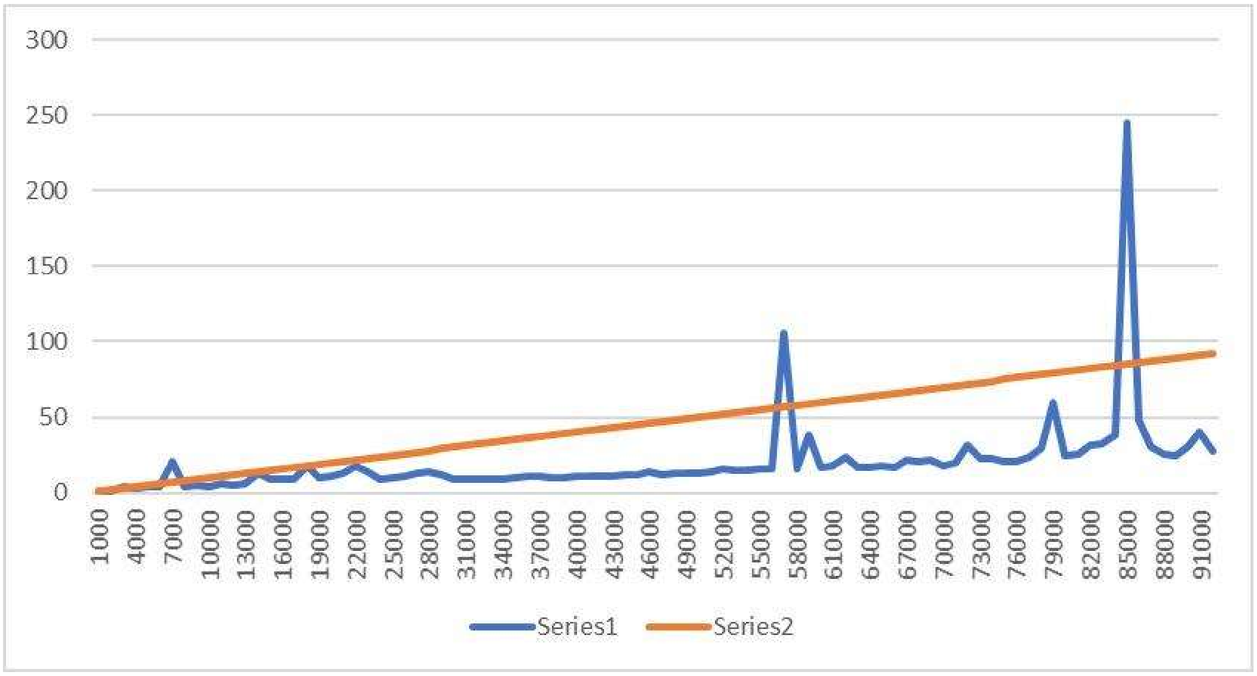
Blue line indicates actual time and the orange line indicates the asymptotic run time.



Topological sort: O(V+E) where E is the edge and V is the vertices of the graph.

Below graph represents asymptotic run time on NFT transaction dataset for every 1000 lines increment while performing Topological sort algorithm.

Blue line indicates actual time and the orange line indicates the asymptotic run time.



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.

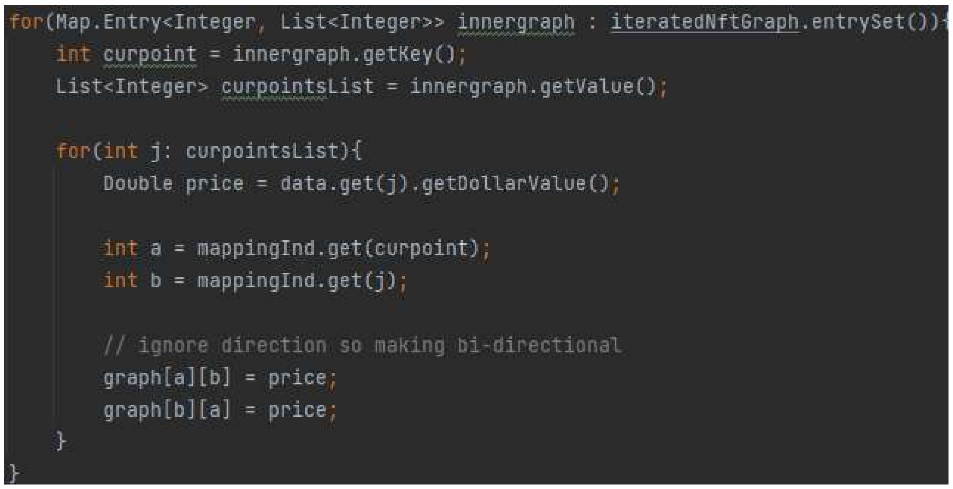
Give your interpretation of the results in the context of buyers/NFT's/prices/time.

Approach:

We can perform minimum spanning tree and maximum spanning tree algorithms on connected graphs. As we know our graph is not connected, so we are generating graphs that are connected and are performing minimum and maximum spanning tree algorithms on these graphs and printing out an adjacency matrix for every graph. We used prims algorithm for minimum spanning tree and maximum spanning tree.

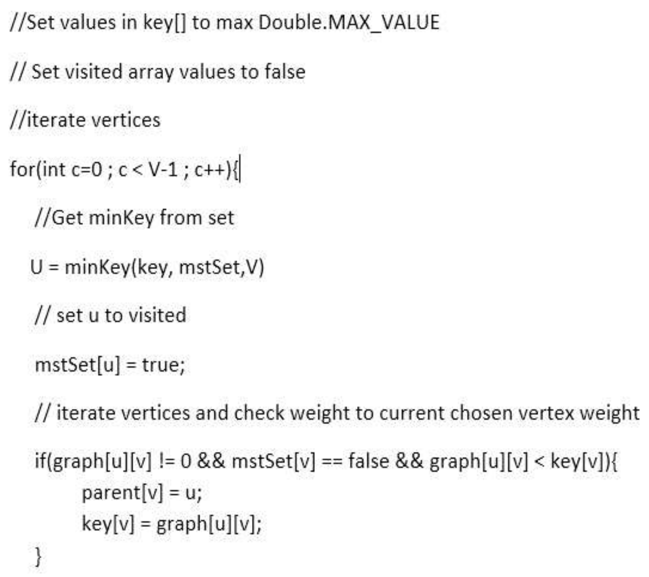
Code:

For each connected graph, get the adjacency matrix and ignore the directions(making bi-directional for each edge)



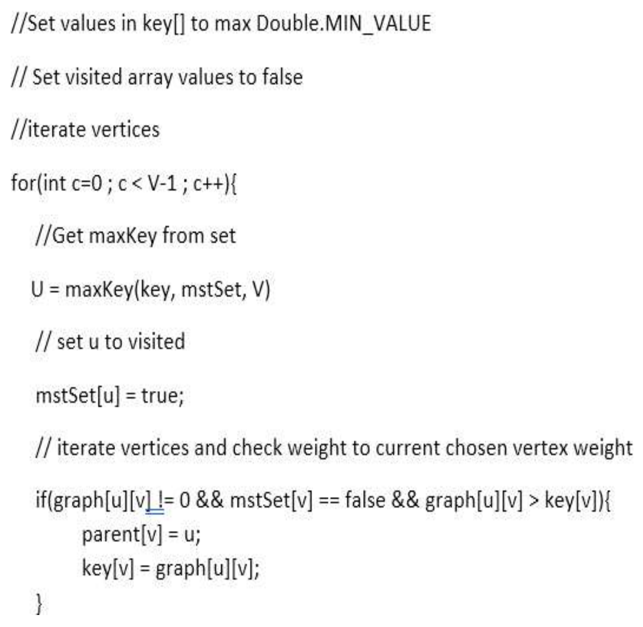
```java
for(Map.Entry<Integer, List<Integer>> innergraph : iteratedNftGraph.entrySet()){
    int curpoint = innergraph.getKey();
    List<Integer> curpointsList = innergraph.getValue();

    for(int j: curpointsList){
        Double price = data.get(j).getDollarValue();

        int a = mappingInd.get(curpoint);
        int b = mappingInd.get(j);

        // ignore direction so making bi-directional
        graph[a][b] = price;
        graph[b][a] = price;
    }
}
```

Pseudocode for Minimum spanning tree on each connected graph using Prim's algorithm:



```
//Set values in key[] to max Double.MAX_VALUE

// Set visited array values to false

//iterate vertices

for(int c=0 ; c < V-1 ; c++){

  //Get minKey from set

  U = minKey(key, mstSet,V)

  // set u to visited

  mstSet[u] = true;

  // iterate vertices and check weight to current chosen vertex weight

  if(graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v]){
        parent[v] = u;
        key[v] = graph[u][v];
  }
```

Pseudocode for Maximum Spanning tree on each connected graph using Prim's algorithm



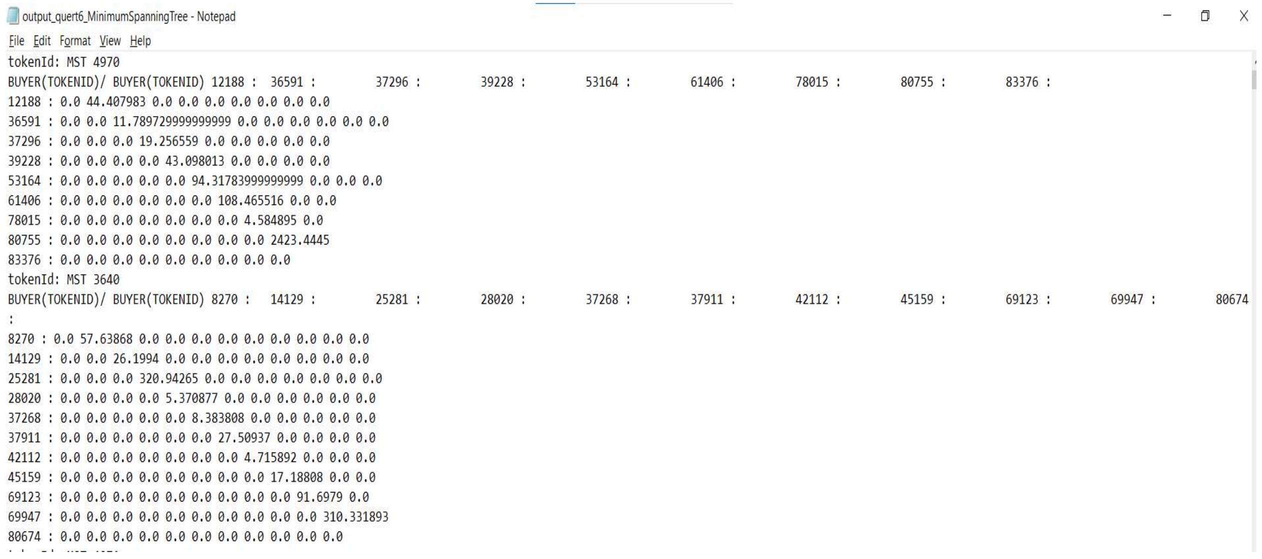
```
//Set values in key[] to max Double.MIN_VALUE

// Set visited array values to false

//iterate vertices

for(int c=0 ; c < V-1 ; c++){

   //Get maxKey from set

  U = maxKey(key, mstSet, V)

   // set u to visited

   mstSet[u] = true;

   // iterate vertices and check weight to current chosen vertex weight

   if(graph[u][v] != 0 && mstSet[v] == false && graph[u][v] > key[v]){
          parent[v] = u;
          key[v] = graph[u][v];
   }
```

Output:

Firstly we generated multiple connected graphs and performed minimum and maximum spanning tree algorithms on every connected graph. For every graph we are displaying the output in the form of an adjacency matrix.

Output format: Running the algorithm will produce two text files i.e. “output_quert6_MinimumSpanningTree.txt” and “output_quert6_MaximumSpanningTree.txt” where in each file are printing adjacency matrices of every graph.

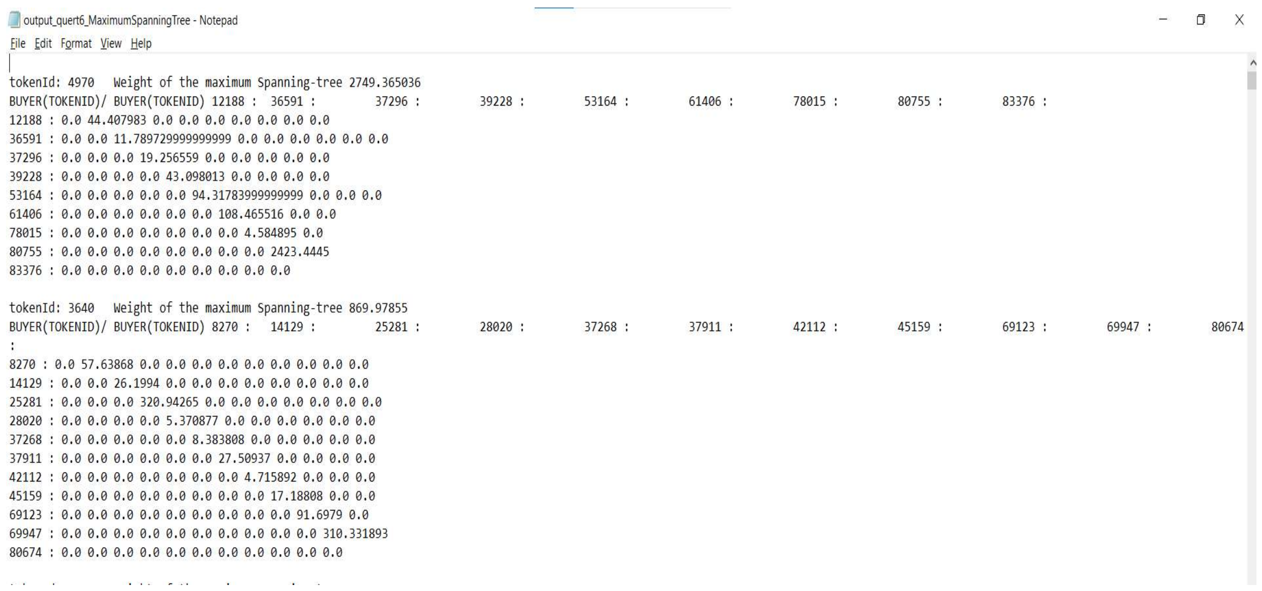
Below is the output of adjacency matrices from running minimum spanning tree algorithms.



output_quert6_MinimumSpanningTree - Notepad

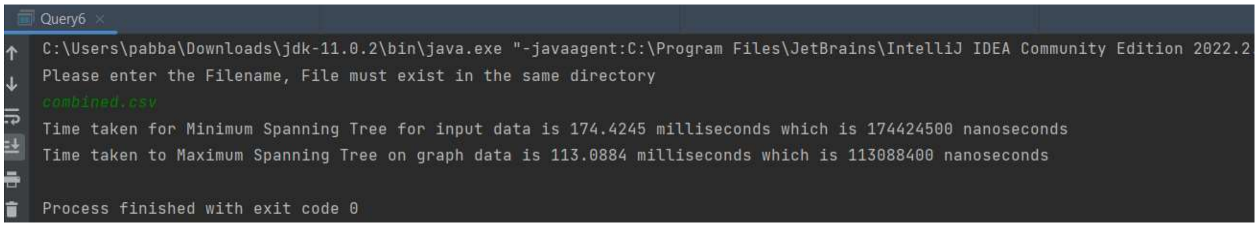
```
tokenId: MST 4970
BUYER(TOKENID)/ BUYER(TOKENID) 12188 :  36591 :         37296 :         39228 :         53164 :         61406 :         78015 :         80755 :         83376 :
12188 : 0.0 44.407983 0.0 0.0 0.0 0.0 0.0 0.0 0.0
36591 : 0.0 0.0 11.789729999999999 0.0 0.0 0.0 0.0 0.0 0.0
37296 : 0.0 0.0 0.0 19.256559 0.0 0.0 0.0 0.0 0.0
39228 : 0.0 0.0 0.0 0.0 43.098013 0.0 0.0 0.0 0.0
53164 : 0.0 0.0 0.0 0.0 0.0 94.31783999999999 0.0 0.0 0.0
61406 : 0.0 0.0 0.0 0.0 0.0 0.0 108.465516 0.0 0.0
78015 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 4.584895 0.0
80755 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 2423.4445
83376 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
tokenId: MST 3640
BUYER(TOKENID)/ BUYER(TOKENID) 8270 :   14129 :         25281 :         28020 :         37268 :         37911 :         42112 :         45159 :         69123 :         69947 :         80674
:
8270 : 0.0 57.63868 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
14129 : 0.0 0.0 26.1994 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
25281 : 0.0 0.0 0.0 320.94265 0.0 0.0 0.0 0.0 0.0 0.0 0.0
28020 : 0.0 0.0 0.0 0.0 5.370877 0.0 0.0 0.0 0.0 0.0 0.0
37268 : 0.0 0.0 0.0 0.0 0.0 8.383808 0.0 0.0 0.0 0.0 0.0
37911 : 0.0 0.0 0.0 0.0 0.0 0.0 27.50937 0.0 0.0 0.0 0.0
42112 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 4.715892 0.0 0.0 0.0
45159 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 17.18808 0.0 0.0
69123 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 91.6979 0.0
69947 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 310.331893
80674 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Below is the output of adjacency matrices from running maximum spanning tree algorithms.



```
tokenId: 4970   Weight of the maximum Spanning-tree 2749.365036
BUYER(TOKENID)/ BUYER(TOKENID) 12188 :  36591 :         37296 :         39228 :         53164 :         61406 :         78015 :         80755 :         83376 :
12188 : 0.0 44.407983 0.0 0.0 0.0 0.0 0.0 0.0 0.0
36591 : 0.0 0.0 11.789729999999999 0.0 0.0 0.0 0.0 0.0 0.0
37296 : 0.0 0.0 0.0 19.256559 0.0 0.0 0.0 0.0 0.0
39228 : 0.0 0.0 0.0 0.0 43.098013 0.0 0.0 0.0 0.0
53164 : 0.0 0.0 0.0 0.0 0.0 94.31783999999999 0.0 0.0 0.0
61406 : 0.0 0.0 0.0 0.0 0.0 0.0 108.465516 0.0 0.0
78015 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 4.584895 0.0
80755 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 2423.4445
83376 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

tokenId: 3640   Weight of the maximum Spanning-tree 869.97855
BUYER(TOKENID)/ BUYER(TOKENID) 8270 :   14129 :         25281 :         28020 :         37268 :         37911 :         42112 :         45159 :         69123 :         69947 :         80674
:
8270 : 0.0 57.63868 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
14129 : 0.0 0.0 26.1994 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
25281 : 0.0 0.0 0.0 320.94265 0.0 0.0 0.0 0.0 0.0 0.0 0.0
28020 : 0.0 0.0 0.0 0.0 5.370877 0.0 0.0 0.0 0.0 0.0 0.0
37268 : 0.0 0.0 0.0 0.0 0.0 8.383808 0.0 0.0 0.0 0.0 0.0
37911 : 0.0 0.0 0.0 0.0 0.0 0.0 27.50937 0.0 0.0 0.0 0.0
42112 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 4.715892 0.0 0.0 0.0
45159 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 17.18808 0.0 0.0
69123 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 91.6979 0.0
69947 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 310.331893
80674 : 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
```

Time complexity:



```
C:\Users\pabba\Downloads\jdk-11.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2
Please enter the Filename, File must exist in the same directory
combined.csv
Time taken for Minimum Spanning Tree for input data is 174.4245 milliseconds which is 174424500 nanoseconds
Time taken to Maximum Spanning Tree on graph data is 113.0884 milliseconds which is 113088400 nanoseconds

Process finished with exit code 0
```

Minimum spanning tree : O(V^2) where E is the edge and V is the vertex of the graph.

Maximum spanning tree: O(V^2) where V is the vertex of the graph.

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.

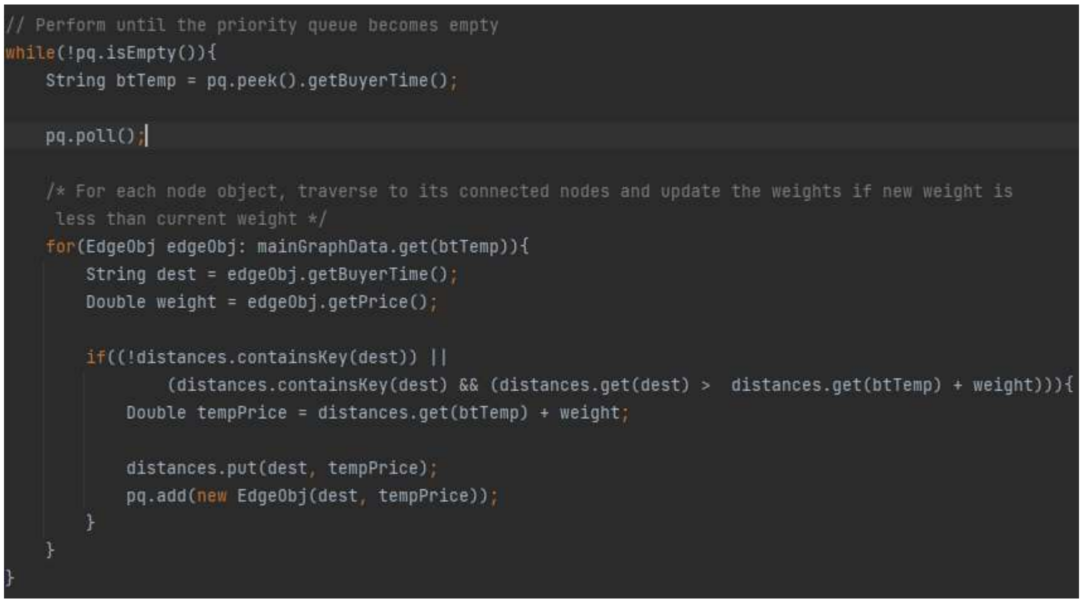
Give your interpretation of the results in the context of buyers/NFT's/prices/time.

Approach:

To find the shortest path from an arbitrary buyer we have performed dijkstra's algorithm on the given data. From the graph where we have formed connections between nodes of buyerId timestamp pairs we are running dijkstra's algorithm and printing out the shortest path in the form of an adjacency list.

Code:

Used Priority Queue, getting the shortest path from source node to all the reachable buyers.



```
// Perform until the priority queue becomes empty
while(!pq.isEmpty()){
    String btTemp = pq.peek().getBuyerTime();

    pq.poll();

    /* For each node object, traverse to its connected nodes and update the weights if new weight is
     less than current weight */
    for(EdgeObj edgeObj: mainGraphData.get(btTemp)){
        String dest = edgeObj.getBuyerTime();
        Double weight = edgeObj.getPrice();

        if((!distances.containsKey(dest)) ||
                (distances.containsKey(dest) && (distances.get(dest) >  distances.get(btTemp) + weight))){
            Double tempPrice = distances.get(btTemp) + weight;

            distances.put(dest, tempPrice);
            pq.add(new EdgeObj(dest, tempPrice));
        }
    }
}
```

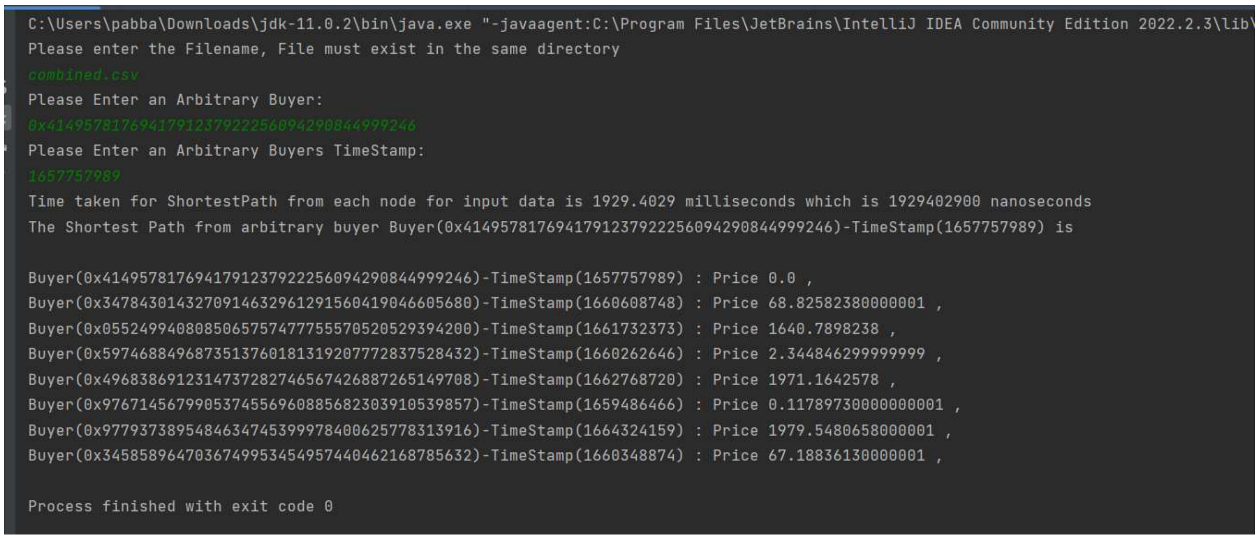
Output:

For the output we are creating a text file "output_quert7.txt" that displays an adjacency list. For every node in the shortest path we print buyerId timestamp and the minimum price to reach that respective node. Below is an example format of the adjacency list.

Source B1( )T1( ) -> B2( )T2( )p2( ), B3( )T2( )p4( )

From above B1, B2, B3 are buyerId's and T1,T2,T3 are timestamp and P1,P2,P3 are prices.

In the console we input BuyerId and TimeStamp and then the code would display all the Buyers and timestamps that belong to the shortest path of given input. If we input the wrong BuyerId and TimeStamp pair the console displays ”The given Buyer and TimeStamp combination is not found from the input data set” but will create “output_query7.txt” with shortest paths from every buyer timestamp pair.

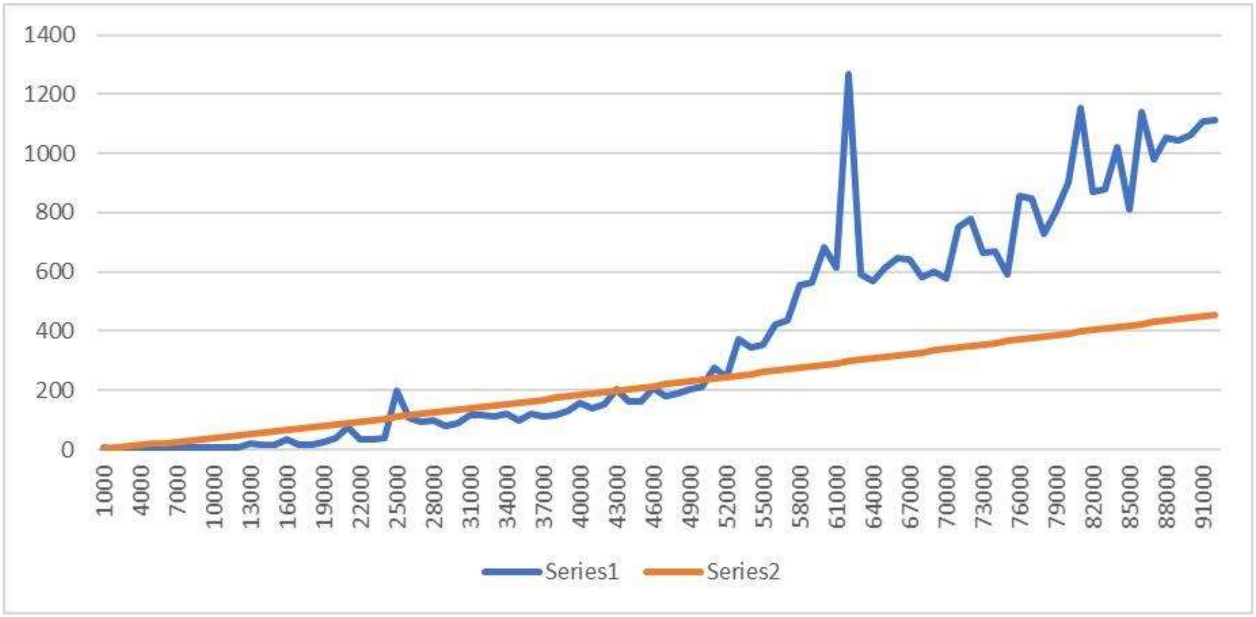


```
C:\Users\pabba\Downloads\jdk-11.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.3\lib\
Please enter the Filename, File must exist in the same directory
combined.csv
Please Enter an Arbitrary Buyer:
0x4149578176941791237922256094290844999246
Please Enter an Arbitrary Buyers TimeStamp:
1657757989
Time taken for ShortestPath from each node for input data is 1929.4029 milliseconds which is 1929402900 nanoseconds
The Shortest Path from arbitrary buyer Buyer(0x4149578176941791237922256094290844999246)-TimeStamp(1657757989) is

Buyer(0x4149578176941791237922256094290844999246)-TimeStamp(1657757989) : Price 0.0 ,
Buyer(0x3478430143270914632961291560419046605680)-TimeStamp(1660608748) : Price 68.82582380000001 ,
Buyer(0x0552499408085065757477755570520529394200)-TimeStamp(1661732373) : Price 1640.7898238 ,
Buyer(0x5974688496873513760181319207772837528432)-TimeStamp(1660262646) : Price 2.344846299999999 ,
Buyer(0x4968386912314737282746567426887265149708)-TimeStamp(1662768720) : Price 1971.1642578 ,
Buyer(0x9767145679905374556960885682303910539857)-TimeStamp(1659486466) : Price 0.11789730000000001 ,
Buyer(0x9779373895484634745399978400625778313916)-TimeStamp(1664324159) : Price 1979.5480658000001 ,
Buyer(0x3458589647036749953454957440462168785632)-TimeStamp(1660348874) : Price 67.18836130000001 ,

Process finished with exit code 0
```

Time Complexity: O(E* log V) where E is the edge and V is the vertex of the graph.

Below is the graph of asymptotic time from running Dijkstra’s algorithm for finding the shortest path for each node from NFT transaction dataset for every 1000 lines increment.

Blue line indicates actual time and the orange line indicates the asymptotic run time.


Series1
Series2