Name: Mohinish Chatterjee
UIN: 329006958

Project Report

1. Implementation

1.1 Graph Class

To represent and work with graphs a Graph class was created. A Graph class has the following important class variables:

- *numVertices* – Stores number of vertices in the graph.
- *maxWeightAllowed* – Maximum possible weight of an edge in the graph.
- *adjacencyList* – Adjacency list representation of the graph.
- *edgeSet* – Weighted edge representation of the graph.

Once a graph object is created, we generate a random graph using one of the two possible ways; based on average vertex degree or based on percent of adjacent neighbors. To support these two ways of generating graphs, the Graph class has following methods:

- *generateWithAvgDegree(**int** avgVertexDegree)* – Generates a random graph with a given average vertex degree. It ensures that the graph generated is connected by creating a cycle containing all the vertices. Then it randomly adds rest of the edges required to maintain the average vertex degree. The number of edges required to maintain the average vertex degree is calculated using the relation, $degree_{avg} = 2|E|/|V|$. If this method is called multiple times, it overwrites the previous graph representation.

- *generateWithAdjNeighbors(**int** percentNeighbors)* – Generates a random graph with a given percentage of vertices that can be adjacent to a vertex. It ensures that the graph generated is connected by creating a cycle containing all the vertices. Then it goes vertex by vertex, adding edges until number of neighbors are around 20%. To keep number of neighbors random enough, I have kept an offset of 2%. So, number of neighbors for a vertex can be anywhere between 900 (20% - 2%) to 1100 (20% + 2%), considering

total vertices to be 5000. If this method is called multiple times, it overwrites the previous graph representation.

- *initializeConnectedGraph()*– Initializes a connected graph with all the vertices connected in a cycle. It first generates a random order of vertices and connects them all in a cycle.

The class also uses following helper methods:

- *convertToEdgeSet()*– Creates weighted edge set by traversing the adjacency list and clears the adjacency list. This is needed for Kruskal's algorithm, as it expects graph in the form of edge set.

- *convertToAdjList()*– Creates adjacency list by traversing the edge set and clears the edge set. This is needed for switching between the graph representation, as Dijkstra's algorithm needs graph in adjacency format and Kruskal in edge weighted format.

  (I am maintaining only one representation of the graph at any given point. If we try to maintain both the representations, then the program is running out of heap memory)

- *printGraphToFile(**String** filename)*– Utility method to print the graph in adjacency list format in a text file specified by the file name.

1.2 Routing Class

A Routing Class is implemented which has static methods that implement the routing algorithms. Each of the routing algorithms return the *dad[], wt[]* and the *execution time* of the algorithm. The class also contains some helper methods to support these algorithms. Following methods are implemented in this class:

- *MAX_BANDWIDTH_PATH_NO_HEAP(**Graph** G, **int** s, **int** t)*– Finds maximum bandwidth path between source '*s*' and target '*t*' in the given Graph, G. It uses modified Dijkstra's algorithm to find the maximum bandwidth path. It returns *dad[]* and *wt[]* arrays which contains, parent values and bandwidth values for each vertex respectively. Following helper methods are needed for implementation:

- *thereAreFringes(**String[]** status)*– Helper method that checks the status array to see if there are any fringes left to be explored. The status can have the values, "*unseen*", "*in-tree*" and "*fringe*". Since this method might have to loop over all the vertices it takes O(*n*) time.

- *getBestFringe(**int**[] wt, **String[]** status)*– Helper method that returns the next best fringe to explore. It loops over the status array and finds the fringe with maximum bandwidth so far. Since this method also might have to loop over all the vertices, it takes O(*n*) time.

The initialization step of the algorithm takes around O(*n*) time. Then the algorithm loops for roughly (n-1) times (no of fringes) and gets the next best fringe to explore in O(*n*) time. Thus, total time taken by the algorithm would be O(n^2).

- *MAX_BANDWIDTH_PATH_WITH_HEAP(**Graph** G, **int** s, **int** t)***–*** Finds maximum bandwidth path between source '*s*' and target '*t*' in the given Graph, G. It uses modified Dijkstra's algorithm to find the maximum bandwidth path. It returns *dad[]* and *wt[]* arrays which contains, parent values and bandwidth values for each vertex respectively. Fringes are stored in a Max-Heap, which allows finding the next best fringe in O(*1*) time. The Max-Heap (Fringe Heap) is implemented as a class that maintains three arrays namely H[], D[] and pos[]. The H[] is the heap array that stores the vertex ids. The vertices bandwidths are stored in D[] array and the pos[] array stores the position of vertices in the heap array. This class provides a *MAXIMUM()* method that extracts the 1st or the vertex with maximum bandwidth and fixes the heap. It calls the *fixHeap(**int** index)* method that maintains the heap structure by swapping elements in the heap as required. It does so in O(*log(n)*) time. Thus, the extracting maximum element totally takes O(*log(n)*) time. The algorithm loops over all the fringes one by one and takes O(*log(n)*) time to get the next best fringe. Overall the algorithm loops over the edges and takes additional O(*log(n)*) time to fix the heap whenever needed. Therefore, the total running time of this algorithm would be O((*n+m).log(n)*).

- *MAX_BANDWIDTH_PATH_KRUSKAL(**Graph** G, **int** s, **int** t)***–*** Finds maximum bandwidth path between source '*s*' and target '*t*' in the given Graph, G. It

implements Kruskal's Algorithm to find the Maximum Spanning Tree. Then it returns the path from 's' to 't' in that maximum spanning tree using Depth-First-search. The algorithm uses Heap-sort to sort the edges. A Min-Heap (Edge Heap) class was implemented to sort the edges in descending order. The algorithm further uses following helper methods and class(es):

- *addEdgeToTree(**Tree** T, **Edge** e)*– Helper method that adds the given edge in the adjacency list of the maximum spanning tree. Performs the task in O(*1*) time.

- *DEPTH_FIRST_SEARCH(**Tree** T, **int** s, **int** t)*– Performs depth first search on maximum spanning tree T and returns the path from 's' to 't' and the maximum bandwidth. Since it can end up exploring all the vertices, it runs in O(*n*) time.

- **Class** MAKESET_UNION_FIND– Implements Makeset, Union and Find operations to keep track of connected components.

The edges are sorted in O(*m.log(n)*) time using heap sort and the rest of the spanning tree generation takes O(*m.log(n)*) time using union by rank. Therefore, the total run time of the algorithm is O(*m.log(n)*).

2. Testing

To do the testing, following methods were implemented in the *Main class:*

- *generateGraphPairs(**int** numPairsGraph)***–*** This helper method takes number of pairs to be generated as input and generates the pairs of graphs as per average degree or percent neighbors of each vertex. The graph objects are also stored in the local memory in order to verify the results if needed. For storing graph objects, this method uses another helping method, *storeGraph(**Graph** G, **String** fileName)* which stores the graph using *ObjectOutputStream* class from *java.io* package.

- *executeTesting(**int** numPairsGraph, **int** numTestingPairs)*– Executes testing of Maximum bandwidth algorithms for each pair of graphs generated using randomly chosen source and target vertices. It takes input, the number of pairs of graphs and the number of source and target pairs it needs to test these graphs on. First, it loads the graphs from the local memory (which were generated by the method above) using the helper method *loadGraph(**String** fileName*) and then for each graph, it generates number of '*s*' and '*t*' pair to test the graph on. It makes note of the time each version of the algorithm takes and averages them at the end, for each type of graph. It also prints the time taken by the algorithm for a single graph averages over all the (s, t) pairs. Further, it prints the path and maximum bandwidth as derived from the dad[] array and the wt[] array respectively. To help with the printing following method was implemented:

    - *printPathAndBandwidth(**Object[]** ans, **int** s, **int** t)*– This helper method takes as input the *dad[], wt[]* as contained in *ans[]* and 's' and 't'. It traces the dad array to find the parent of target and then recursively finds the parents of parent till source is reached. It pushes the parents in stack and finally when source is reached, it pops the vertices until it's empty and prints them. Getting the maximum bandwidth is straight forward as we the value can be found in *wt[t].*

3. Results and Discussion

A sample output of each algorithm run on a (s, t) pairs for a sparse graph looks like this:

```
p1_graph1
Finding Maximum bandwidth path between source = 1697 and target = 2922.

      Found Maximum bandwidth path WITHOUT heap in 0.119 seconds
      No. of Hops: 71
      Path: 1697 -> 1407 -> 1459 -> 4610 -> 221 -> 907 -> 1912 -> 416 -> 1586
      -> 2122 -> 556 -> 2728 -> 3565 -> 2718 -> 786 -> 1860 -> 1772 -> 2307 -
      > 3366 -> 1581 -> 702 -> 812 -> 39 -> 2191 -> 930 -> 2515 -> 2263 ->
      2779 -> 91 -> 2700 -> 616 -> 1142 -> 1958 -> 1256 -> 2020 -> 2008 ->
      3066 -> 606 -> 1181 -> 1712 -> 285 -> 1458 -> 2404 -> 2762 -> 3323 ->
      348 -> 1731 -> 1049 -> 1396 -> 3164 -> 275 -> 1706 -> 251 -> 446 ->
      1091 -> 1151 -> 1038 -> 3277 -> 2575 -> 357 -> 1059 -> 3055 -> 63 -> 86
      -> 844 -> 257 -> 229 -> 1760 -> 1951 -> 2019 -> 746 -> 2922
```

Maximum Bandwidth: 3427

Found Maximum bandwidth path WITH heap in **0.048 seconds**
No. of Hops: 32
Path: 1697 -> 1407 -> 1459 -> 4610 -> 221 -> 907 -> 1912 -> 416 -> 1586 -> 2122 -> 556 -> 2728 -> 3565 -> 4407 -> 4455 -> 3483 -> 303 -> 70 -> 4077 -> 4840 -> 1788 -> 1663 -> 4015 -> 1199 -> 101 -> 1285 -> 2168 -> 3363 -> 1003 -> 4849 -> 1066 -> 746 -> 2922
Maximum Bandwidth: 3427

Found Maximum bandwidth path using Kruskal's Algorithm in **0.067 seconds**
No. of Hops: 79
Path: 1697 -> 1407 -> 1459 -> 4610 -> 221 -> 579 -> 2453 -> 2557 -> 80 -> 4448 -> 3206 -> 2920 -> 1176 -> 282 -> 3916 -> 4865 -> 930 -> 1208 -> 3118 -> 664 -> 4968 -> 1953 -> 4572 -> 4502 -> 3263 -> 2758 -> 4662 -> 1997 -> 404 -> 3379 -> 1410 -> 2900 -> 4661 -> 910 -> 2279 -> 4352 -> 3399 -> 1857 -> 3971 -> 3742 -> 1675 -> 2175 -> 3088 -> 4766 -> 2038 -> 3196 -> 1337 -> 3499 -> 2727 -> 2887 -> 2500 -> 4115 -> 1448 -> 76 -> 3589 -> 820 -> 4013 -> 2263 -> 2779 -> 2180 -> 3112 -> 2248 -> 3997 -> 2391 -> 137 -> 332 -> 4873 -> 1119 -> 303 -> 3483 -> 2600 -> 308 -> 4372 -> 2588 -> 4243 -> 85 -> 63 -> 2334 -> 746 -> 2922
Maximum Bandwidth: 3427

As expected, Dijkstra's Algorithm with Heap (O((*n+m).log(n)*)) and kruskal's algorithm (O(*m.log(n)*)) out-perform normal Dijkstra's algorithm without heap (O(n^2)). However, for dense graph the performance of Kruskal seems to deteriorate. As noted below.

A sample output for each algorithm run on a (s, t) pair for a dense graph looks like this:

p1_graph2
Finding Maximum bandwidth path between **source = 1998** and **target = 989.**

Found Maximum bandwidth path WITHOUT heap in **0.156 seconds**
No. of Hops: 43
Path: 1998 -> 3356 -> 4236 -> 3266 -> 3685 -> 1180 -> 3818 -> 1239 -> 137 -> 890 -> 3528 -> 4384 -> 2607 -> 4637 -> 3459 -> 3418 -> 329 -> 139 -> 4899 -> 998 -> 3856 -> 581 -> 83 -> 429 -> 652 -> 2876 -> 2530 -> 2300 -> 3743 -> 1281 -> 3340 -> 796 -> 1023 -> 2526 -> 3261 -> 3366 -> 268 -> 3918 -> 2610 -> 2457 -> 3049 -> 2940 -> 4333 -> 989
Maximum Bandwidth: 5976

Found Maximum bandwidth path WITH heap in **0.098 seconds**
No. of Hops: 69
Path: 1998 -> 3356 -> 4236 -> 3266 -> 3685 -> 1180 -> 3818 -> 1239 -> 137 -> 890 -> 3528 -> 4384 -> 2607 -> 4637 -> 3459 -> 3418 -> 329 -> 139 -> 4899 -> 998 -> 3856 -> 581 -> 83 -> 443 -> 489 -> 4052 -> 2653 -> 3987 -> 2597 -> 4737 -> 923 -> 4133 -> 4949 -> 29 -> 4837 -> 765 -> 4720 -> 2005 -> 3838 -> 1357 -> 4191 -> 4718 -> 715 -> 3253 -> 918 -> 3179 -> 2284 -> 3311 -> 590 -> 4648 -> 4495 -> 2961 -> 4343 -> 1385 -> 2052 -> 742 -> 4319 -> 2314 -> 1376 -> 4250 -> 209 -> 1865 -> 268 -> 3918 -> 2610 -> 2457 -> 3049 -> 2940 -> 4333 -> 989

Maximum Bandwidth: 5976

Found Maximum bandwidth path using Kruskal's Algorithm in **2.638 seconds**
No. of Hops: 46
Path: 1998 -> 3356 -> 4236 -> 3266 -> 3685 -> 1180 -> 3818 -> 1239 -> 137 -> 890 -> 3528 -> 4384 -> 2607 -> 4637 -> 3459 -> 3418 -> 329 -> 139 -> 4899 -> 4265 -> 4575 -> 1219 -> 3465 -> 2759 -> 414 -> 3163 -> 3168 -> 4171 -> 1204 -> 2214 -> 974 -> 1373 -> 1445 -> 2728 -> 4145 -> 4308 -> 1790 -> 2372 -> 3236 -> 961 -> 1420 -> 621 -> 2788 -> 4325 -> 2101 -> 4333 -> 989
Maximum Bandwidth: 5976

We notice here that, Dijkstra's Algorithm with Heap still performs the best, but Kruskal's Algorithm performs worse than even the Dijkstra's Non-heap version. This result can be attributed to the fact that complexity of Kruskal's algorithm involves large constant values, which significantly affect its performance when the graph becomes denser.

A typical execution summary of a sparse graph is observed like so:

```
EXECUTION SUMMARY:
Average time taken by Dijkstra with no heap: 0.1044
Average time taken by Dijkstra with heap: 0.0174
Average time taken by Kruskal's Algorithm: 0.0234
```

A typical execution summary of a dense graph is observed like so:

```
EXECUTION SUMMARY:
Average time taken by Dijkstra with no heap: 0.1904
Average time taken by Dijkstra with heap: 0.121199995
Average time taken by Kruskal's Algorithm: 2.6890001
```

The overall summary for the entire run of testing is obtained as follows:

```
OVERALL SUMMARY:

Average time taken by Dijkstra with no heap for sparse graph: 0.09608
Average time taken by Dijkstra with no heap for dense graph: 0.18072

Average time taken by Dijkstra with heap for sparse graph: 0.006
Average time taken by Dijkstra with heap for dense graph: 0.115959994

Average time taken by Kruskal's Algorithm for sparse graph: 0.00836
Average time taken by Kruskal's Algorithm for dense graph: 2.5963602
```

- END -