# PROJECT
# CSCE 629: ANALYSIS OF ALGORITHMS
# NETWORK OPTIMIZATION

CHINMAY AJIT SAWKAR

UIN:633002213

# INTRODUCTION

Network optimization is well researched area in Computer Science and Computer Engineering where many advancements in the field of optimized algorithms have been made. The aim of this project is to practically implement a efficient network routing protocol using various data structures and graph algorithms and analyse their performance in terms of time complexity. The goal of this project is to find the maximum bandwidth path in the routing network considered as a graph with the weights as the maximum bandwidth of the edge or the connection between two nodes. In this project, the well-know algorithms like Dijkstra's and Kruskal Maximum Spanning Tree are modified to solve for the maximum bandwidth path in the network. The Dijkstra's algorithm has been implemented in two ways i.e. using heap and without using heap data structure. All the algorithms have been tested on two types of randomly generated graphs: a sparse graph and a dense graph. The comparison of the average execution time of the three algorithmic approaches is done to analyse the theoretical and practical complexities of this algorithms.

# IMPLEMENTATION

***** The code was written in python 3.8 **without** use of any high level libraries or predefined data structures (like dictionaries, heaps, stack etc.) for implementing algorithms or defining the data structures. (only simple array is used for whole implementation)

## Graph design and storage

A graph is data structure with vertices or nodes and edges. In this routing problem we have a undirected and weighted graph where the weight of the edge represents the bandwidth of the edge. There are various ways in which we can represent a graph like Adjacency Matrix or Adjacency List. Both the ways have their own benefit. The adjacency matrix stores the weight of the graph in a square matrix of length equal to number of vertexes and is efficient for dense graphs where we can visit the weight of any edge (u,v) in O(1) time. But in case of sparse graphs most of the matrix will be empty and this will be very inefficient in terms of space and to traverse edges from a node, insert of delete new edges. The adjacency list is better in such cases when the graph is sparse and works by storing vertices which are connected to a node in a separate linked list, thus using this linked list we can traverse the nodes connect to a particular node efficiently compared to adjacency matrix especially in case of sparse graphs. As in our case

of routing problem we are given two scenarios of generating graphs one where the randomly created graph should have maximum degree of six. This makes the graph sparse enough, thus the adjacency matrix approach is not feasible for building graph. In this project, used a adjacency list approach in which we have a list of neighbours for each node in the graph and with the neighbour node we also store the weight associated with the edge as a tuple of (neighbouring node, weight of the edge). This is a efficient use memory and time. Therefore we can now traverse through the list of neighbours efficiently without checking 5000 vertices each time as in adjacency matrix.

## Random Graph Generation

In this project to test the algorithms we generate two random graphs of 5000 vertices.

a. Sparse graph

In the first graph G1, the average vertex degree is six. This is implemented by initialising an array of the graph in the class graph. Initially we create 5000 nodes one by one and add an edge between the current edge and the previous one. Thus we create 5000 vertexes and 5000 edges making an fully connected graph in a cycle. Then we randomly add edges in the graph between two random vertices such that degree of each vertex is 6 in the end. This is achieved by randomly adding another 10000 unique edges to the graph as we already have 5000 edges. I do this by generating two random vertices check if the edge between them already exist, if not then the edge is added to the graph , else we sample two different random vertices. Thus repeating this process to generate 10000 edges we will have the average vertex degree of the graph exactly as six and total number of edges being 15000.

b. Dense graph

For creating the dense graph G2 we use the same approach, first we create 5000 vertices and connect them sequentially such that all vertices are connected and then we add edges randomly between any two vertices such that an vertex is connected to approximately 20% of vertexes (i.e. 1000 other random vertices). To do this every vertex must have degree around 1000. To achieve an average degree of 1000 we need to add another (5000 *1000)/2 -5000 = 2495000 edges. This can be done similarly as above with slight modification that the degree of each vertex that we are going to add must me around 1000. For this I select randomly the degree of each vertex to be around 1000 in range of (995, 1005). This will ensures that the degree of vertices are approximately around 1000. Using this principle I use a loop for 2495000 times to generate two random vertexes and check if the edge between them already exists or if degree of any of the vertex is within the degree limit for that vertex generated randomly earlier. (allowed vertex degree limit). If both the vertices satisfy the conditions we add the edge to the graph else we generate two different random vertices and repeat the check. Thus dense graph G2 has 2500000 edges and 5000 vertex, with average vertex degree equal to 1000 with minimum degree around 940, and maximum degree around 1005. For faster checking of vertices if a edge exits I make a adjacency matrix and mark in the matrix [u][v] as 1 whenever I add a edge (u,v). This make generation of dense graph much faster.

We can delete the adjacency matrix after graph generation as we will use adjacency list created separately. Below is the pseudocode for generating the sparse graph G2.

PSEUDO CODE:

```
def DenseGraphGenerator(self):
     max_degree = [assign random weights to all vertexes]
     remaining_vertices=(5000*1000)//2-5000
     //initialise graph G with 5000 vertices and 2D empty adjacency matrix
     G=graph(5000), adj_mat=[[0]]
     //form chain by adding edges sequentially with randomweights
     for v in vertices{
        add_edge(v, v+1, random_weight)
        adj_mat[v,v+1]=1
        adj_mat[v+1,V]=1
     }


     for loop from 0 to remaining vertices{
        v1,v2=random()  //generate 2 random vertices
        //generate new random vertices until conditions are met
        while (v1==v2 || adj_mat[v1][v2]==1 || if deg(v1),deg(v2)> respective
        max_degree){
            v1,v2=random()  //generate 2 different random vertices
        }
        add_edge(v, v+1, random_weight)
        adj_mat[v,v+1]=1
        adj_mat[v+1,V]=1
      }
```

## Modified Dijkstra's algorithm for maximum Bandwidth problem

The Dijkstra's algorithm finds the maximum bandwidth pass using the Dijkstra's greedy method when all the weighs are positive. As in our graph all the weights are positive we can say that the Dijkstra's algorithm will work to find the maximum bandwidth problem. The Dijkstra algorithm is modified to find the Max bandwidth as follows:

PSEUDO CODE:

```
def Dijkstra(G, source, target):
    for all vertexes:
       initialize bw = [0] , parent = [-1] , status = 'not_visted'

    status[source] = 'in_tree', bw[source] = infinty
```

```
    for every neighbour v of s:
        bw[v] = wt[v]
        parent[v] = source
        status[v] = 'fringe'

    while there are no fingers left:
        v= find fringe with maxium bandwidth.
        status[v] = 'in_tree'

        for every neighbour w of v:
            if status[w] == 'not_visted':
                bw[w] = min(bw[v], wt[w])
                parent[w] = v
                status[w] = 'fringe'
            else if (status[w] == 'fringe' and bw[w] < min(bw[v], wt[w])):
                bw[w] = min(bw[v], wt[w])
                parent[w] = v

    return bw[target],parent
```

## Heap

The heap is implemented using the heap functions get_max(),insert(), delete(), sift_up(), sift_down() similar to what we did in the class just that we use recursion insstead of while loop. The max heap is implemented using array as we did in class with modification that we keep a position array (P) and another array and update it during insert and delete operations whenever we swap a two elements in the heap. With this we can keep track of the index of all the elements and in O(1) can access.

## Modified Dijkstra's algorithm with Heap

The Dijkstra's algorithm can be modified to run more efficiently if we add a heap implementation. A heap can be used to get the fringe with maximum bandwidth in O(1) time. Thus we just update the above Dijkstra's algorithm with changes to incorporate the heap implementation as stated above. We form a position array and keep track of positions of the

PSEUDO CODE:

```
def Dijkstra_With_Heap(G, source, target):
    for all vertexes:
        //initialize the variables bandwidth, parent, status, position array
         bw = [0] , parent = [-1] , status = 'not_visted', H = Heap(),P=[-1]
```

```
status[source] = 'in_tree', bw[source] = infinty
for every neighbour w of s:
    bw[v] = wt[v]
    parent[v] = source
    status[v] = 'fringe'
    H.insert(v, bw[v]) //insert into heap

while there are fingers left:
    v = H.get_max()  //get max from heap
    H.delete(v)      //delete from heap
    status[v] = 'in_tree'

    for every neighbour w of v:
        if status[w] == 'not_visted':
            bw[w] = min(bw[v], wt[w])
            parent[w] = v
            status[w] = 'fringe'
        else if (status[w] == 'fringe' and bw[w] < min(bw[v], wt[w])):
            H.delete(w)
            bw[w] = min(bw[v], wt[w])
            parent[w] = v
            H.insert(w,bw[w])   //insert w again with updated bw

return bw[target],parent
```

## Make Union Find

For efficient implementation of the Kruskal's algorithm we need to implement the Union Find algorithm for path compression. In this we use the find method for searching the root of the vertex and union function to join two disjoint branches of the tree. The implementation of this is done as discussed in the class with path compression. We use recursion to find the root using the parent array and while we find the root we store the vertices which we get while searching iteratively and at the end assign the root as the parent for all those vertices, this induces path compression.

## Modified Kruskal's Algorithm for Maximum Bandwidth

The Kruskal's algorithm is used to find the maximum spanning tree (MST) in a graph. In our case we use a modified Kruskal's algorithm to solve maximum Bandwidth problem as discussed in class. We first use the Kruskal's algorithm to find Maximum Spanning tree and then use DFS for find the path from source to destination in the maxium spanning tree. Instead of using the normal Kruskal's two modifications are made to make the algorithm efficient. First we implement the Heap Sort on the edges to arrange them in descending order of their weights. Second we use the MakeUnionFind algorithm to check if two vertices are in different disjoint set of vertices or not and da we use Union method to join two vertices so that all the vertices connected to these two vertices are also connected. Finally we use DFS (Depth First Search) to find the path from source to target.

PSEUDO CODE:

```
def Kruskal(G, s, t):
    edges = [array of all edges in graph G]
    HeapSort(G, edges)
    initialise bw[v]=0 for all vertices

    muf = MakeUnionFind(number_of_vertices)   // initialise MUF class and MST
    MST  = Graph()
    for each edge [v,w] in edges:
        root_u = muf.Find(u)
        root_v = muf.Find(v)
        if root_u != root_v:
            add edge [v,w] to MST
            muf.Union(root_u, root_v)
   return MST
```

# RESULTS AND ANALYSIS

The algorithms were tested on five pairs of randomly generated graphs G1(Sparse Graph) and G2 (Dense Graph). The code was written in Python 3.8 without use of any high level libraries for implementing algorithms or defining the data structures. For each of the five sparse and dense graphs five iterations were run choosing 5 different pairs of source and destination nodes. The time is measured for all the three algorithm implementations.

## Theoretical Time Complexities.

1. Modified Dijkstra's Algorithm without Heap implantation
   As we have to use array to find the fringe with maximum bandwidth we have to iterate through all the fringes for that causing O(n) time for searching. As we do this at most n times (number of vertices) the time complexity of the algorithm is O($n^2$).

2. Modified Dijkstra's Algorithm with Heap Structure
   If we implement a max heap for storing and finding the maximum bandwidth of the edges, we can reduce the time complexity of searching the fringe with maximum bandwidth to O(1) instead of O(n). But now the bottleneck is insertion and deletion in the max heap which take O(log(n)) time and is done for all the fringes and thus will take O(nlog(n)) time. Also for insertion of all egdes from the sources can take maximum (nlog(n)) time. Thus the overall time complexity for the algorithm is now O((n+m)log(n)).

3. Modified Kruskal's Algorithm for Maximum Bandwidth

   The time complexity of the Kruskal's algorithm is O(mlogn) for building the maximum spanning tree as we use the MakeUnionFind algorithm to keep track of which vertexes are connected and which are disjoint. As discussed in class the optimised version of MUF() works in time O(mlog*(n)) which is also using optimal sorting algorithm (in our case heap Sort) we can sort the edges in O(mlog(m)).

   Thus total time complexity of log the algorithm will be O(m (log(m) + log*(n)). Since for log*(n) will be very small for any practical implementations it can be considered as constant. Thus time complexity for the algorithm will be O(mlog(m)) which is equivalent to O(mlog($n^2$)) = O(mlog(n)). For finding maximum bandwidth path after generating the MST requires O(m+n) time complexity using DFS.

## Implementation Results

The testing was done on Intel Core™ i5-6300 HQ CPU @ 2.301 Mhz, 4 cores. The graphs were randomly generated with 5000 vertices and random weights between (1, 8000). All the three algorithms were tested on both 5 Spare graphs and 5 Dense graphs. The time complexity of the all the algorithms was noted.

## Time Complexity Summary: Sparse Graph

| Graph | Iteration | Dijkstra without Heap | Dijkstra with Heap | Kruskal MST Build time | Kruskal Path Find time |
|---|---|---|---|---|---|
| Sparse1 | 1 | 1.739 | 0.172 | 0.219 | 0.010 |
| Sparse1 | 2 | 0.375 | 0.056 | 0.200 | 0.010 |
| Sparse1 | 3 | 1.080 | 0.105 | 0.191 | 0.009 |
| Sparse1 | 4 | 1.641 | 0.112 | 0.242 | 0.012 |
| Sparse1 | 5 | 0.866 | 0.098 | 0.215 | 0.014 |
| Sparse2 | 1 | 1.403 | 0.135 | 0.207 | 0.009 |
| Sparse2 | 2 | 1.832 | 0.164 | 0.219 | 0.012 |
| Sparse2 | 3 | 1.798 | 0.148 | 0.232 | 0.012 |
| Sparse2 | 4 | 1.321 | 0.126 | 0.215 | 0.011 |
| Sparse2 | 5 | 1.854 | 0.154 | 0.207 | 0.012 |
| Sparse3 | 1 | 1.256 | 0.116 | 0.213 | 0.016 |
| Sparse3 | 2 | 0.233 | 0.027 | 0.206 | 0.012 |
| Sparse3 | 3 | 0.177 | 0.010 | 0.208 | 0.013 |
| Sparse3 | 4 | 1.460 | 0.073 | 0.226 | 0.010 |
| Sparse3 | 5 | 0.831 | 0.101 | 0.220 | 0.011 |
| Sparse4 | 1 | 1.866 | 0.159 | 0.260 | 0.016 |
| Sparse4 | 2 | 0.974 | 0.097 | 0.220 | 0.013 |
| Sparse4 | 3 | 1.623 | 0.147 | 0.212 | 0.012 |
| Sparse4 | 4 | 1.540 | 0.062 | 0.205 | 0.011 |
| Sparse4 | 5 | 1.080 | 0.107 | 0.187 | 0.009 |
| Sparse5 | 1 | 0.534 | 0.057 | 0.197 | 0.008 |
| Sparse5 | 2 | 1.114 | 0.163 | 0.238 | 0.009 |
| Sparse5 | 3 | 1.803 | 0.168 | 0.285 | 0.018 |
| Sparse5 | 4 | 1.599 | 0.023 | 0.210 | 0.013 |
| Sparse5 | 5 | 1.167 | 0.114 | 0.201 | 0.012 |
| **Average Time** | | **1.247** | **0.107** | **0.218** | **0.012** |

# Time Complexity Summary: Dense Graph

| Graph | Iteration | Dijkstra without Heap | Dijkstra with Heap | Kruskal MST Build time | Kruskal Path Find time |
|---|---|---|---|---|---|
| Dense1 | 1 | 4.629 | 3.043 | 68.273 | 0.010 |
| Dense1 | 2 | 3.105 | 0.871 | 65.073 | 0.010 |
| Dense1 | 3 | 1.673 | 0.970 | 64.409 | 0.009 |
| Dense1 | 4 | 4.431 | 2.891 | 65.039 | 0.012 |
| Dense1 | 5 | 1.937 | 2.139 | 64.964 | 0.014 |
| Dense2 | 1 | 5.496 | 3.319 | 68.210 | 0.009 |
| Dense2 | 2 | 4.496 | 3.051 | 68.177 | 0.012 |
| Dense2 | 3 | 3.514 | 2.421 | 74.066 | 0.012 |
| Dense2 | 4 | 6.082 | 3.072 | 76.241 | 0.011 |
| Dense2 | 5 | 6.461 | 4.470 | 79.559 | 0.012 |
| Dense3 | 1 | 4.641 | 2.883 | 67.073 | 0.016 |
| Dense3 | 2 | 5.032 | 0.927 | 65.770 | 0.012 |
| Dense3 | 3 | 3.010 | 1.772 | 67.662 | 0.013 |
| Dense3 | 4 | 5.223 | 3.314 | 66.231 | 0.010 |
| Dense3 | 5 | 0.265 | 0.683 | 65.785 | 0.011 |
| Dense4 | 1 | 2.066 | 2.202 | 77.722 | 0.016 |
| Dense4 | 2 | 5.375 | 3.688 | 67.232 | 0.013 |
| Dense4 | 3 | 6.275 | 4.081 | 68.719 | 0.012 |
| Dense4 | 4 | 4.952 | 3.150 | 69.485 | 0.011 |
| Dense4 | 5 | 1.107 | 1.344 | 70.626 | 0.009 |
| Dense5 | 1 | 6.534 | 3.867 | 73.413 | 0.008 |
| Dense5 | 2 | 3.394 | 0.446 | 70.326 | 0.009 |
| Dense5 | 3 | 6.258 | 3.809 | 73.299 | 0.018 |
| Dense5 | 4 | 6.237 | 3.482 | 71.996 | 0.013 |
| Dense5 | 5 | 4.517 | 2.239 | 71.612 | 0.012 |
| **Average Time** | | **4.268** | **2.565** | **69.639** | **0.012** |

# Sample Output results for Sparse Graph

```
PS C:\Users\Chinmay\Desktop\Tamu acad\Algo\proj> & C:/Users/Chinmay/AppData/Local/Programs/Python/Python38/python.exe "c:/Users/Chinmay/Desktop/Tamu
 acad/Algo/proj/graphG final1.py"
------------------------------------------------------------
Generating Sparse Graph version  1
Time for graph generation:  0.06183362007141113
number of vertices in (G1) Sparse Graph v1: 5000
number of edges in (G1) Sparse Graph v1: 15000
average vertex degree in (G1) Sparse Graph v1: 6

*** Iteration 1  ***
Source:  3155  Destination:  3445
Time_Dijkstra:  1.7393488883972168
Time_Dijkstra_with_heap:  0.171539306640625
Time_Krushkal_building_MST:  0.21941399574279785
Time_Kruskal_finding_MBpath:  0.010037899017333984
Max Bandwidth value using Dijkstra's:  5705
Max Bandwidth value using Dijkstra's with heap:  5705
Max Bandwidth value using Kruskals:  5705
Max Bandwidth path using Dijkstra's:  [3445, 1442, 1443, 4927, 4141, 4007, 1334, 279, 2576, 3309, 203, 202, 3424, 3425, 29, 4076, 4077, 936, 3519, 3
518, 349, 1068, 1067, 2562, 712, 711, 1722, 1723, 3377, 2578, 2577, 3534, 88, 1346, 3817, 3248, 3247, 4304, 3575, 3576, 2307, 2308, 2772, 4140, 4139
, 1940, 479, 478, 4704, 1352, 3155]
Max Bandwidth path using Dijkstra's with heap:  [3445, 1442, 1443, 4927, 4141, 4007, 1334, 279, 2576, 3309, 203, 3081, 4441, 1410, 2815, 2814, 4974,
 4975, 1747, 883, 3624, 2471, 1563, 2565, 3113, 62, 63, 887, 886, 1547, 1546, 141, 2785, 499, 498, 3710, 2984, 1898, 1897, 1790, 2697, 4136, 4137, 4
138, 3466, 1407, 762, 3241, 1432, 1433, 2309, 2310, 220, 4989, 4858, 775, 774, 2771, 2772, 4140, 4139, 1940, 479, 478, 4704, 1352, 3155]
Max Bandwidth path using Kruskals:  [3445, 1442, 1443, 4927, 4141, 3673, 3674, 4215, 2967, 2966, 1267, 3921, 3922, 3923, 2413, 4596, 3091, 3090, 308
9, 555, 1034, 3612, 3611, 2769, 2770, 2771, 2772, 4140, 4139, 1940, 479, 478, 4704, 1352, 3155]

*** Iteration 2  ***
Source:  1100  Destination:  4662
Time_Dijkstra:  0.37499523162841797
Time_Dijkstra_with_heap:  0.05585885047912598
Time_Krushkal_building_MST:  0.20045709609985352
Time_Kruskal_finding_MBpath:  0.009972095489501953
Max Bandwidth value using Dijkstra's:  6343
Max Bandwidth value using Dijkstra's with heap:  6343
Max Bandwidth value using Kruskals:  6343
Max Bandwidth path using Dijkstra's:  [4662, 1860, 1859, 3657, 2522, 1620, 1619, 1032, 582, 581, 580, 940, 4147, 4148, 855, 2647, 4029, 4030, 1788,
804, 4538, 1185, 371, 1656, 1655, 2578, 3377, 109, 1888, 647, 2494, 2030, 998, 3964, 3965, 3389, 3388, 3911, 3912, 3625, 3519, 936, 4077, 4076, 29,
3425, 3424, 202, 203, 3309, 1100]
Max Bandwidth path using Dijkstra's with heap:  [4662, 1734, 851, 852, 3033, 234, 233, 4854, 971, 4358, 4357, 2786, 2787, 84, 3265, 2888, 2887, 1904
, 4321, 3388, 3911, 3912, 3625, 3519, 936, 4077, 4076, 29, 3425, 3424, 202, 203, 3309, 1100]
Max Bandwidth path using Kruskals:  [4662, 1734, 851, 852, 3033, 234, 233, 4854, 971, 4358, 4357, 2786, 2787, 84, 3265, 2888, 2887, 1904, 4321, 3388
, 3911, 3912, 3625, 2372, 2803, 3624, 2471, 1563, 2565, 3113, 62, 63, 887, 4209, 989, 3834, 3835, 1598, 2479, 2480, 3018, 982, 983, 984, 3187, 2721,
 4191, 4190, 4189, 1400, 272, 4236, 936, 4077, 4076, 29, 3425, 3424, 202, 203, 3309, 1100]
```

# Sample Output for the Dense Graph

```
------------------------------------------------------------
generating Dense Graph version  2
Time_DenseG:  15.437718629837036
number of vertices in Dense Graph G2 v2: 5000
number of edges in Dense Graph G2 v2: 2500000
avg deg in Dense Graph G2 v2: 1000
min deg in Dense Graph G2 v2: 933
max deg in Dense Graph G2 v2: 1006

*** Iteration 1  ***
3155  :  3445
Time_D:  5.496300220489502
Time_DH:  3.3191237449645996
Time_Krushkal_building_MST:  68.2099974155426
Time_Kruskal_finding_MBpath:  0.00997304916381836
Max Bandwidth value using Dijkstra's:  7980
Max Bandwidth value using Dijkstra's with heap:  7980
Max Bandwidth value using Kruskals:  7980
Max Bandwidth path using Dijkstra's:  [3445, 2549, 303, 921, 59, 1375, 2994, 3296, 4326, 106, 3790, 4650, 3956, 500, 2386, 4252, 3385, 1158, 1110, 9
34, 3076, 1501, 1316, 1018, 2104, 1588, 2322, 4222, 1092, 1421, 3596, 4776, 4068, 2770, 3155]
Max Bandwidth path using Dijkstra's with heap:  [3445, 2549, 303, 921, 59, 1375, 2994, 3296, 4326, 2681, 4874, 2368, 3380, 2111, 3385, 1158, 4619, 1
422, 4794, 3243, 1524, 3677, 4876, 3870, 3257, 111, 739, 4336, 1872, 3266, 745, 4018, 3679, 2019, 2574, 2500, 2596, 3012, 4451, 4637, 2794, 3063, 42
61, 2747, 1931, 1421, 3596, 4776, 4068, 2770, 3155]
Max Bandwidth path using Kruskals:  [3445, 2549, 303, 921, 59, 1375, 2994, 3296, 4326, 106, 3790, 4650, 3956, 500, 2386, 4252, 3385, 1158, 4619, 142
2, 4794, 3243, 1524, 3677, 790, 4272, 1387, 4558, 3931, 521, 4734, 1194, 359, 1408, 3957, 981, 1092, 1421, 3596, 4776, 4068, 2770, 3155]

*** Iteration 2  ***
1100  :  4662
Time_D:  4.495966911315918
Time_DH:  3.0508415699005127
Time_Krushkal_building_MST:  68.17675971984863
Time_Kruskal_finding_MBpath:  0.010971546173095703
Max Bandwidth value using Dijkstra's:  7987
Max Bandwidth value using Dijkstra's with heap:  7987
Max Bandwidth value using Kruskals:  7987
Max Bandwidth path using Dijkstra's:  [4662, 305, 1200, 3344, 1497, 3472, 3409, 2152, 4079, 4133, 2316, 623, 443, 76, 2725, 2079, 2753, 2010, 3117,
3068, 2202, 2078, 1506, 3144, 1822, 1684, 130, 1406, 957, 1687, 3753, 3803, 2600, 1397, 4997, 598, 1073, 3741, 890, 319, 4950, 2591, 1100]
Max Bandwidth path using Dijkstra's with heap:  [4662, 305, 1200, 3344, 1497, 4311, 317, 306, 4765, 3086, 4999, 4473, 4018, 745, 3266, 1872, 2201, 4
980, 868, 2438, 88, 2278, 1299, 434, 2811, 2501, 3323, 1671, 1745, 2324, 4386, 51, 2857, 1377, 3172, 1684, 130, 1406, 957, 1687, 3753, 3803, 2600, 1
397, 4997, 598, 1073, 3741, 890, 319, 4950, 2591, 1100]
Max Bandwidth path using Kruskals:  [4662, 305, 1200, 3344, 1497, 3472, 3409, 2152, 4079, 4550, 3077, 1353, 306, 317, 1936, 3415, 4637, 2794, 3063,
4261, 2747, 1931, 1421, 1092, 981, 3957, 1408, 359, 1194, 4734, 521, 3931, 4558, 1387, 4272, 790, 3677, 1524, 3243, 4794, 1422, 4619, 1158, 3385, 42
52, 2386, 4941, 2264, 1981, 2216, 1913, 4923, 4096, 388, 4800, 3218, 4968, 1770, 2425, 2397, 4674, 2344, 3824, 4178, 1822, 1684, 130, 1406, 957, 168
7, 3753, 3803, 2600, 1397, 4997, 598, 1073, 3741, 890, 319, 4950, 2591, 1100]
```

## Analysis of results

1. The average vertex degree of Sparse Graph is 6 and the average vertex degree of Dense graph is 1000 (i.e 20% of the vertices).

2. From the results we can see that all the three algorithms get the same value of Maximum bandwidth value but get different paths sometimes.
3. Time complexity comparison
    a. Average Time complexity in Sparse graph:
    Dijkstra without Heap > Kruskal > Dijkstra with heap Heap
    b. Average Time complexity in Sparse graph:
    Kruskal > Dijkstra without Heap > Dijkstra with heap Heap

4. From the results we can see that the time complexity of Dijkstra without Heap is the highest among the three which is verified and acceptable according to the theoretical time complexity. Thus we can say that Dijkstra's with heap and Kruskal is better than Dijkstra without heap in all scenarios.

5. The total time complexity of Kruskal (graph generation and path finding) is lower than Dijkstra with heap for the sparse graph but higher for the Dense graph. This is because in sparse graph the number of edges are less (15000) thus the time to sort the edges O(mlog(m)) using heap sort is less while in the dense graph the number of edges are much larger (2500000) (almost 160 times the edges in the sparse graph). Thus in the case of the dense graph the time complexity of Kruskal is much larger than than the time complexity of Dijkstra with heap.

6. The time required for finding path in kruskal is very less compared to Dijkstra's in both sparse and dense graphs thus if we want to find multiple maximum bandwidths betweeen different source destination pairs, Kruskal is better algorithm.

### Improvements

We observe that time complexity increases for dense graphs in kruskal due to time consumed by heapSort. If we already have sorted edges the time to build the MST will be reduced considerably. We can optimise few functions using hashmaps like graph generation and Heap instead of array as this will improve the space complexity of the implementation giving same O(1) behaviour that is implemente by using 2D arrays with lot of space reuirement.