Fall 2017

# CSCE-629 Course Project

## Design and Analysis of Algorithms

Comparison between running time of different Maximum Bandwidth path problems.

Piyush Bhatt
UIN: 227002733

## 1. Random Graph Generation

Algorithm used for generating random graph: First, as suggested in the problem statement edges were added to form a cycle containing all the vertices (5000) in the graph which makes it a connected graph. Next I add every edge to the graph with the probability 'p', such that the expected degree of every vertex comes out to be 'd'. Moreover, I have used a Boolean (|V^2|) size matrix to ensure that the graph is simple i.e. there are no multiple edges.

Let G be a graph with a given requirement that degree should be 'd' and number of vertices should be 'n'.
Let the vertex names be 1,2,3,4 - - - - , n

```
Add an edge between vertex 1 and vertex n
for (i = 0; i < n; ++i)
   Add an edge between vertex i and vertex i+1
```

After the above step, the number of edges in our graph will be 'n' – every vertex is connected to exactly 2 other vertices.

Then, for every pair of vertices {(s, t) | such that t >= (s+2)} the probability that an edge exists between 's' and 't' in the given graph will be (d-2)/n.
For the first Graph $G_1$ with average vertex degree = 8 and number of vertices = 5000, then for every pair of vertices {(s, t) | such that t >= (s+2)} the probability that an edge exists between 's' and 't' in the given graph is equal to 6/5000.

```
for (s = 1;s<=5000;s++)
      for (t = s+2;t<=5000;t++)
         p = random number between [0,4999]
         if p<6 then add an edge between vertex s and vertex t
```

For second Graph $G_2$ in which each vertex is adjacent to 20% of other vertices and number of vertices = 5000, then for every pair of vertices {(s, t) | such that t >= (s+2)} the probability that an edge exists between 's' and 't' in the given graph is equal to 20/100.

```
for (s = 1;s<=5000;s++)
      for (t = s+2;t<=5000;t++)
         p = random number between [0,100]
         if p<20 then add an edge between vertex s and vertex t
```

## 2. Heap Structure

Binary Heap Structure implemented in this project is not the standard algorithm of heap mentioned in the textbook but rather a modified version of heap that can be called the Indexed Heap Implementation. Maximum bandwidth path algorithm based on a modification of Dikjstra's algorithm using a heap structure for fringes and Maximum bandwidth path algorithm based on a modification of Kruskal's algorithm, needed a priority queue implementation of heap which provides a feature to delete elements without providing the element's index in the heap. To achieve this, I have created a priority queue in array H[ ] keyed by their values stored in a separate array D[ ] and I[ ] storing the index of the element in H. For example, suppose element with name 20 is the largest element in the heap with its value

equal to 9000. So this element will be at the topmost position in the heap, and the corresponding values in all the arrays will be like

```
H[1] = 20                  // Name of the element stored at position 1 is 20
D[H[1]] = D[20] = 9000     // Value of Element 20 is 9000
I[H[1]]=I[20] = 1          // Element 20 is stored at position 1
```

Thus, in every function in heap which involves the position of an element to be changed we are also maintaining the index array I accordingly. For example, when swapping the first element of the heap with the last element:

```
Swap(H[1] with H[last]) // H[1]=4 & H[last]=8, now H[1]=8 & H[last]=4
I[H[1]] = 1            // now index of 8 is changed to 1
I[H[last]] = last      // now index of 4 is changed to 4
```

Similarly, whenever the position of an element is changed in the heap, we update the Index array simultaneously. If we maintain the Index array, we can always delete the element from the heap by merely knowing the name of the element.

This structure would work perfectly with the Algorithm based on the modification of Dijkstra's algorithm which uses the heap structure for maintaining fringes because the fringe vertices are all denoted by some *unique integer*, i.e. vertex names ranging from 1 to 5000, so it is convenient to store the names of the fringes in array H. But using this structure for the Algorithm using Kruskal's maximum spanning tree algorithm won't be as convenient because here we need heap for storing and sorting the edges and each edge is represented by a pair of vertices but H, being an array, cannot store multiple values in a single cell. This would create a problem because if we cannot identify each edge uniquely in H, we cannot use the array D and I for getting the weight and index of that edge.
There are two solutions to this problem-

a. Generate a function f such that it will give a unique value for every (s,t) pair where s and t lie in the range [1,5000] and the use this value to uniquely identify an edge in heap.
b. Form a structure that denotes an edge and use an array of this structure instead of the integer array to store the end vertices and weight of an edge. Here each edge is denoted by a *unique arbitrary integer* between [1, m], where m denotes the total number of edges in the graph.

I have used the second method in my project, thus we begin with the definition of the structure needed.

```
struct Node{
  int src, dest, weight;
};
```

**Using this heap for Kruskal's Algorithm**

Suppose for an edge between vertex 8 and vertex 23, and weight of this edge is 5000 which is maximum among all the edge weights in the given graph G. If this was the $12^{th}$ element inserted into the heap – then the *unique arbitrary integer* assigned to it will be 12.

Define edge[ ] as an structure array of type Node of size equal to the total number of edges.

```
H[1] = 12;                                  // Unique arbitrary integer denoting this edge is 12
edge[H[1]].src = edge[12].src = 8;
edge[H[1]].dest = edge[12].dest = 23;       // Even if src=23 and dest=8, it is still the same
edge[H[1]].weight = edge[12].weight = 5000;
I[H[1]] = I[12] = 1                         // Edge denoted by number 12 is at index 1
```

**Using this heap for Dijkstra's Algorithm**

This, heap version can be used for storing the fringes also. I don't need

Define fringe[ ] as an structure array of type Node of size equal to the total number of vertices.

```
H[1] = 20  // Destination of the fringe stored at position 1 is 20
node[H[1]].weight = node[20].weight = 9000// Weight of fringe is 9000
I[H[1]]=I[20] = 1      // Fringe 20 is stored at position 1
```

Thus, with few modification, I have implemented a heap which is suitable to be used by both the Algorithms, this will make the comparison between the two fair and true. This implementation also supports deletion and insertion in O(log n) time per operation. Heap Sort on this heap is also implemented which is bounded by $O(n.\log_2 n)$ time.

## 3. Finding Maximum Bandwidth Path

As a requirement of the project, we have to implement and compare 3 algorithms:

a. Algorithm 1: MAX-BANDWIDTH-PATH based on Dikjstra's Algorithm without heap.
b. Algorithm 2: MAX-BANDWIDTH-PATH based on Dikjstra's Algorithm with heap for fringes.
c. Algorithm 3: MAX-BANDWIDTH-PATH based on modified Kruskal and using HeapSort.

Algorithm 1 and Algorithm 2 are almost similar in implementation except for a part. They differ on how we pick the fringe with maximum bandwidth. The two ways are as follows:

a. Algorithm 1: Uses an array to maintain the fringes and whenever needed scan the array to find the fringe with maximum bandwidth. It would give a worst case time complexity of $O(n^2)$.
b. Algorithm 2: Uses heap to store fringes and whenever needed we can pop the topmost element of the heap to get the fringe with maximum bandwidth. It would give the worst case time complexity of $O(n.\log_2 n)$.

**MAX-BANDWIDTH-PATH-USING-DIJKSTRA-WITHOUT-HEAP(G,s,t)**

**Input:** Graph G, Source node s and destination node t

**Output:** Returns the parent array

```
1. for each node v in G do { status[v] = unseen; bandwidth[v] = -∞;}
2. status[v] = in-tree; bandwidth[s] = +∞;
3. for each neighbor w of s do { status[w] = fringe; bandwidth[w]=weight[s,w]; parent[w]=s;}
4. while status[t] ≠ in-tree
     pick a fringe v of maximum bandwidth;  status[v] = in-tree;
     for each neighbor w of v do
       if status[w] is unseen then
         {status[w] = fringe; bandwidth[w] = min{bandwidth[v], weight[v,w]}; parent[w]= v;}
       else if status[w] is fringe AND bandwidth[w] < min{bandwidth[v], weight[v,w]} then
         { bandwidth[w] = min{bandwidth[v], weight[v,w]}; parent[w] = v;}
5. return parent[1….n]
```

**MAX-BANDWIDTH-PATH-USING-DIJKSTRA-WITH-HEAP(G,s,t)**

We will use the same algorithm mentioned above with minor changes for implementation using heap.

The changes will be as follows:

a. Initialize an empty heap H before step 4, using the heap structure defined in last section.
b. In step 3, push all the neighbors of s to this heap H.
c. Replace the "pick a fringe v…" step by "pop the topmost element from the heap containing fringes".
d. If first 'if' condition is true, then add that fringe to the heap.
e. If the 'else if' condition is true then increase the bandwidth value of the fringe inside heap using the Index array.

For the standard heap implemented Priority Queue without Indexing, it would not have been possible to update or delete the fringe based on the name of the fringe, but because we are maintaining the index of every fringe, we can perform these operation in O(log n) time.

**MAX-BANDWIDTH-PATH-USING-KRUSKAL-WITH-HEAP(G,s,t)**

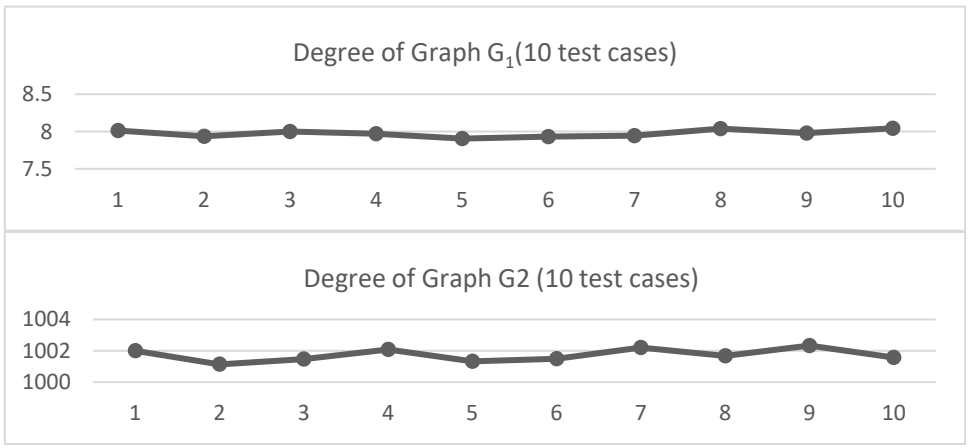
1. Construct a Maximum Spanning Tree T for G using modifications to Kruskal's Algorithm.
2. Find the s to t path on T using Depth First Search.
3. Return this path

I have implemented the modification of Kruskal's Algorithm to get maximum bandwidth path using the algorithm mentioned in the class i.e. using the Union-Find with path compression for avoiding the cycles and my own heap structure for selecting the edges in the non-increasing order of their edge weights. This function KruskalMST() returns a graph that has all the vertices of graph G and only the edges contained in the maximum spanning tree. The maximum spanning tree is then passed on to the DFS() function along with the source s and destination t, this will return the path from s to t in the form a parent array. This path is the maximum bandwidth path from s to t in Graph G.

Next, we will compare these three algorithms on the basis of running time by comparing them against each other for 5 graphs with average degree 8 and 5 graphs with 20% adjacent vertices rule. Each of these graphs are tested for 5 source (s) to destination (t) pairs by the 3 algorithms.

## 4. Testing and Comparison (Time taken in terms of clock ticks elapsed)

To test the accuracy of my random graph generating algorithms, I generated 10 random graphs of each type and calculated the average degree of each of these graphs. All of the graphs of type $G_1$ had average degree 8. For graphs of type $G_2$, each vertex is adjacent to about **(**1002/5000)*100 = 20% of other vertices. Thus, the condition stated in project is satisfied.



**Time Comparison between different algorithms for different Graphs and different s-t pairs.**

| | | s-t pair 1 | s-t pair 2 | s-t pair 3 | s-t pair 4 | s-t pair 5 | Average |
|---|---|---|---|---|---|---|---|
| **Sparse Graph 1** | **Algo 1** | 2216 | 163246 | 123705 | 57988 | 152341 | **99899.2** |
| | **Algo 2** | 431 | 4714 | 6071 | 4919 | 5857 | **4398.4** |
| | **Algo 3** | 15046 | 648 | 566 | 635 | 336 | **3446.2** |
| **Sparse Graph 2** | **Algo 1** | 3248 | 64684 | 93679 | 147950 | 11816 | **64275.4** |
| | **Algo 2** | 1331 | 3801 | 4160 | 5612 | 2988 | **3578.4** |
| | **Algo 3** | 15766 | 204 | 229 | 761 | 251 | **3442.2** |
| **Sparse Graph 3** | **Algo 1** | 65844 | 141100 | 49906 | 165426 | 52376 | **94930.4** |
| | **Algo 2** | 2575 | 4459 | 2604 | 6217 | 730 | **3317** |
| | **Algo 3** | 15046 | 648 | 566 | 635 | 336 | **3446.2** |
| **Sparse Graph 4** | **Algo 1** | 29928 | 42624 | 165650 | 161913 | 66783 | **93379.6** |
| | **Algo 2** | 1366 | 5175 | 6325 | 5969 | 5495 | **4866** |
| | **Algo 3** | 16744 | 750 | 253 | 743 | 500 | **3798** |
| **Sparse Graph 5** | **Algo 1** | 101238 | 46442 | 135238 | 141807 | 20459 | **89036.8** |
| | **Algo 2** | 4401 | 823 | 5190 | 5649 | 1246 | **3461.8** |
| | **Algo 3** | 15253 | 339 | 586 | 412 | 418 | **3401.6** |
| **Dense Graph 1** | **Algo 1** | 206063 | 253520 | 94123 | 12733 | 476077 | **208503.2** |
| | **Algo 2** | 183228 | 26391 | 84890 | 9001 | 398770 | **140456** |
| | **Algo 3** | 663433 | 386 | 420 | 200 | 688 | **133025.4** |
| **Dense Graph 2** | **Algo 1** | 608347 | 134048 | 35375 | 83154 | 10960 | **174376.8** |
| | **Algo 2** | 498824 | 192976 | 172437 | 274999 | 11829 | **230213** |
| | **Algo 3** | 672696 | 387 | 390 | 335 | 529 | **134867.4** |
| **Dense Graph 3** | **Algo 1** | 335882 | 477015 | 430695 | 562196 | 164731 | **394103.8** |
| | **Algo 2** | 285532 | 374727 | 352018 | 465801 | 173732 | **330362** |
| | **Algo 3** | 720161 | 676 | 445 | 246 | 805 | **144466.6** |
| **Dense Graph 4** | **Algo 1** | 65670 | 104190 | 449828 | 96767 | 62760 | **155843** |
| | **Algo 2** | 96831 | 337815 | 365158 | 253368 | 70854 | **224805.2** |
| | **Algo 3** | 635269 | 392 | 130 | 567 | 274 | **127326.4** |
| **Dense Graph 5** | **Algo 1** | 48876 | 224003 | 40681 | 37610 | 535210 | **177276** |
| | **Algo 2** | 24683 | 182786 | 39672 | 140499 | 424249 | **162377.8** |
| | **Algo 3** | 631218 | 677 | 511 | 613 | 478 | **126699.4** |

Time taken is shown in the from of clock ticks elapsed, calculated using difference between outputs given by std::time(), a C++ library function, before and after running an algorithm.

Algorithm 2 and Algorithm 3 are always better than the Algorithm 1. For Sparse Graphs (Graphs with degree 8), average running time for Algorithm 3 is better than that for Algorithm 2 by a very small difference but for the Dense Graphs this difference widens**. Moreover, there is something notable about the Algorithm 3** that once we find a Maximum Spanning Tree for a Graph, we can use it for any number of s-t pairs without having to find it again. That is why after the first s-t pair, time taken for next s-t pair falls by almost 95%. If we compare the time taken to find s-t paths after we have the Maximum Spanning Tree, Algorithm 3 is faster than the other two by a very big difference.

## 5. Conclusion and Practical Significance

**It is clear from the results that Algorithm 3 is the fastest among the three, irrespective of whether the graph is sparse or dense.** The behaviour of Algorithm 2 where we need to find the Maximum Spanning Tree for a Graph only once might be a very useful feature for practical purposes – We can just find the maximum spanning tree once for a network and whenever we need the maximum bandwidth path, we can use this tree to get that in a very fast time. We will only need to find the Maximum Spanning Tree for a network again only when there are any changes in the network.

## 6. Sample Output from Terminal

### a. Graph with average vertex degree of 8 ( Output for five source to destination pairs )

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 7845
Maximum bandwidth path from vertex 1195 to vertex 4140 :
4140 <-- 4139 <-- 1655 <-- 1654 <-- 2256 <-- 2476 <-- 1128 <-- 1283 <-- 1860 <-- 1861 <-- 1839 <-- 617 <-- 1123 <-- 96 <-- 2070 <-- 2823 <-- 893 <-- 38 <-- 2607 <-- 2606 <-- 2094 <-- 1129 <-- 2943 <-- 1322 <-- 1323 <-- 504 <-- 885 <-- 884 <-- 883 <-- 2189 <-- 180 <-- 179 <-- 4174 <-- 1103 <-- 73 <-- 460 <-- 14 <-- 1702 <-- 1700 <-- 1195

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=7845
Maximum bandwidth path from vertex 1195 to vertex 4140 :
4140 <-- 4139 <-- 1655 <-- 1654 <-- 1653 <-- 4162 <-- 4638 <-- 594 <-- 2855 <-- 4085 <-- 4886 <-- 4077 <-- 4730 <-- 4731 <-- 460 <-- 14 <-- 1702 <-- 1700 <-- 1195

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=7845
Maximum bandwidth path from vertex 1195 to vertex 4140 :
4140 <-- 4139 <-- 4701 <-- 4232 <-- 4233 <-- 4086 <-- 1020 <-- 1019 <-- 3375 <-- 783 <-- 3198 <-- 1965 <-- 1966 <-- 897 <-- 177 <-- 199 <-- 198 <-- 197 <-- 270 <-- 269 <-- 2606 <-- 2094 <-- 1129 <-- 2943 <-- 1322 <-- 1323 <-- 504 <-- 885 <-- 884 <-- 883 <-- 2189 <-- 206 <-- 4092 <-- 4093 <-- 4094 <-- 3136 <-- 2254 <-- 1044 <-- 1045 <-- 4413 <-- 4414 <-- 3475 <-- 2867 <-- 2868 <-- 3910 <-- 1860 <-- 1861 <-- 1839 <-- 728 <-- 313 <-- 4446 <-- 719 <-- 2247 <-- 1577 <-- 2134 <-- 4813 <-- 3337 <-- 2221 <-- 3489 <-- 4077 <-- 4730 <-- 4731 <-- 460 <-- 14 <-- 1702 <-- 1700 <-- 1195

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 8372
Maximum bandwidth path from vertex 3149 to vertex 1349 :
1349 <-- 1469 <-- 1207 <-- 3974 <-- 2385 <-- 2724 <-- 1376 <-- 4094 <-- 4093 <-- 4092 <-- 206 <-- 2189 <-- 883 <-- 884 <-- 885 <-- 504 <-- 1323 <-- 1322 <-- 2943 <-- 1129 <-- 2094 <-- 2606 <-- 269 <-- 270 <-- 197 <-- 198 <-- 2099 <-- 2299 <-- 3427 <-- 3428 <-- 1832 <-- 3084 <-- 3808 <-- 2438 <-- 1622 <-- 3642 <-- 3287 <-- 104 <-- 2122 <-- 2123 <-- 2124 <-- 1460 <-- 3629 <-- 4614 <-- 4403 <-- 4402 <-- 1815 <-- 1814 <-- 644 <-- 1905 <-- 3533 <-- 4120 <-- 3895 <-- 1022 <-- 1023 <-- 4163 <-- 3661 <-- 837 <-- 4500 <-- 4501 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 3149

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=8372
Maximum bandwidth path from vertex 3149 to vertex 1349 :
1349 <-- 1469 <-- 1207 <-- 3974 <-- 2385 <-- 2724 <-- 1376 <-- 4094 <-- 3136 <-- 2254 <-- 1044 <-- 1045 <-- 4413 <-- 4414 <-- 3475 <-- 2867 <-- 1663 <-- 1664 <-- 2284 <-- 2285 <-- 1194 <-- 534 <-- 535 <-- 536 <-- 4609 <-- 513 <-- 1383 <-- 1384 <-- 4235 <-- 4649 <-- 4546 <-- 1233 <-- 2178 <-- 1351 <-- 3345 <-- 4501 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 3149

[Cont. Next Page]

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=8372
Maximum bandwidth path from vertex 3149 to vertex 1349 :
1349 <-- 1469 <-- 1207 <-- 3974 <-- 2385 <-- 2724 <-- 1376 <-- 4094 <-- 3136 <-- 2254 <-- 1044 <-- 1045 <-- 4413 <-- 4414 <-- 3475 <-- 2867 <-- 1663 <-- 1664 <-- 2284 <-- 2285 <-- 1194 <-- 534 <-- 535 <-- 536 <-- 4609 <-- 513 <-- 1383 <-- 1384 <-- 4235 <-- 4649 <-- 4546 <-- 1233 <-- 2178 <-- 1351 <-- 3345 <-- 4501 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 3149

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 7394
Maximum bandwidth path from vertex 4964 to vertex 2942 :
2942 <-- 1732 <-- 130 <-- 1721 <-- 1722 <-- 2607 <-- 38 <-- 893 <-- 2823 <-- 2070 <-- 96 <-- 287 <-- 288 <-- 289 <-- 290 <-- 180 <-- 2189 <-- 883 <-- 884 <-- 1496 <-- 4148 <-- 620 <-- 580 <-- 541 <-- 340 <-- 341 <-- 342 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 1828 <-- 1827 <-- 1826 <-- 1825 <-- 1824 <-- 2430 <-- 2429 <-- 4964

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=7394
Maximum bandwidth path from vertex 4964 to vertex 2942 :
2942 <-- 1732 <-- 130 <-- 1721 <-- 1722 <-- 2607 <-- 38 <-- 893 <-- 2823 <-- 2070 <-- 96 <-- 1123 <-- 1596 <-- 932 <-- 3665 <-- 4679 <-- 4680 <-- 1258 <-- 1259 <-- 720 <-- 4297 <-- 4589 <-- 4891 <-- 459 <-- 2954 <-- 2955 <-- 1751 <-- 2547 <-- 4131 <-- 1145 <-- 3667 <-- 3668 <-- 823 <-- 3252 <-- 3413 <-- 3412 <-- 3347 <-- 4607 <-- 4608 <-- 2960 <-- 2961 <-- 2962 <-- 1324 <-- 1325 <-- 2124 <-- 1460 <-- 1459 <-- 3847 <-- 3084 <-- 1832 <-- 3428 <-- 3427 <-- 3397 <-- 1023 <-- 4163 <-- 3661 <-- 837 <-- 4500 <-- 4501 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 1828 <-- 1827 <-- 1826 <-- 1825 <-- 1824 <-- 2430 <-- 2429 <-- 4964

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=7394
Maximum bandwidth path from vertex 4964 to vertex 2942 :
2942 <-- 1732 <-- 130 <-- 1721 <-- 1722 <-- 2607 <-- 2606 <-- 2094 <-- 1129 <-- 2943 <-- 1322 <-- 1323 <-- 504 <-- 885 <-- 884 <-- 883 <-- 2189 <-- 206 <-- 4092 <-- 4093 <-- 4094 <-- 3136 <-- 2254 <-- 1044 <-- 1045 <-- 4413 <-- 4414 <-- 3475 <-- 2867 <-- 1663 <-- 1664 <-- 2284 <-- 2285 <-- 1194 <-- 534 <-- 535 <-- 536 <-- 4609 <-- 513 <-- 1383 <-- 1384 <-- 4235 <-- 4649 <-- 4546 <-- 1233 <-- 2178 <-- 1351 <-- 3345 <-- 4501 <-- 343 <-- 326 <-- 1399 <-- 4255 <-- 4145 <-- 3148 <-- 1828 <-- 1827 <-- 1826 <-- 1825 <-- 1824 <-- 2430 <-- 2429 <-- 4964

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 7641
Maximum bandwidth path from vertex 781 to vertex 1851 :
1851 <-- 1044 <-- 1045 <-- 1372 <-- 1459 <-- 1460 <-- 1756 <-- 948 <-- 769 <-- 100 <-- 101 <-- 1279 <-- 438 <-- 2185 <-- 2186 <-- 1194 <-- 2285 <-- 2284 <-- 2283 <-- 667 <-- 666 <-- 1248 <-- 1135 <-- 1531 <-- 2649 <-- 1399 <-- 326 <-- 343 <-- 342 <-- 341 <-- 340 <-- 541 <-- 580 <-- 233 <-- 738 <-- 2440 <-- 2441 <-- 2650 <-- 781

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=7641
Maximum bandwidth path from vertex 781 to vertex 1851 :
1851 <-- 1044 <-- 2254 <-- 3136 <-- 4094 <-- 4093 <-- 4092 <-- 206 <-- 2189 <-- 883 <-- 884 <-- 885 <-- 981 <-- 980 <-- 4985 <-- 1091 <-- 140 <-- 1012 <-- 1013 <-- 1438 <-- 4813 <-- 1309 <-- 4951 <-- 272 <-- 2440 <-- 2441 <-- 2650 <-- 781

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=7641
Maximum bandwidth path from vertex 781 to vertex 1851 :
1851 <-- 4568 <-- 1731 <-- 449 <-- 3279 <-- 3278 <-- 482 <-- 4131 <-- 1145 <-- 3667 <-- 3666 <-- 1805 <-- 4014 <-- 2336 <-- 1050 <-- 4344 <-- 4513 <-- 3857 <-- 563 <-- 562 <-- 4631 <-- 2468 <-- 2469 <-- 2278 <-- 2277 <-- 4905 <-- 1020 <-- 1019 <-- 3375 <-- 783 <-- 3198 <-- 1965 <-- 1966 <-- 897 <-- 177 <-- 199 <-- 198 <-- 197 <-- 270 <-- 269 <-- 2606 <-- 2094 <-- 1129 <-- 2943 <-- 1322 <-- 1323 <-- 504 <-- 885 <-- 884 <-- 883 <-- 2189 <-- 206 <-- 4092 <-- 4093 <-- 4094 <-- 3136 <-- 2254 <-- 1044 <-- 1045 <-- 4413 <-- 4414 <-- 3475 <-- 2867 <-- 2868 <-- 3910 <-- 1860 <-- 1861 <-- 1839 <-- 728 <-- 313 <-- 4446 <-- 719 <-- 2247 <-- 1577 <-- 2134 <-- 4813 <-- 1309 <-- 4951 <-- 272 <-- 2440 <-- 2441 <-- 2650 <-- 781

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 5404
Maximum bandwidth path from vertex 3107 to vertex 2397 :
2397 <-- 2396 <-- 2911 <-- 1123 <-- 1596 <-- 932 <-- 1031 <-- 1030 <-- 2673 <-- 2672 <-- 1198 <-- 1199 <-- 1901 <-- 1900 <-- 1899 <-- 3147 <-- 2160 <-- 2161 <-- 2596 <-- 3107

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=5404
Maximum bandwidth path from vertex 3107 to vertex 2397 :
2397 <-- 2396 <-- 2911 <-- 1123 <-- 96 <-- 4662 <-- 58 <-- 2189 <-- 206 <-- 4092 <-- 4093 <-- 1280 <-- 1279 <-- 438 <-- 1726 <-- 2839 <-- 2840 <-- 1462 <-- 1196 <-- 4731 <-- 27 <-- 859 <-- 2412 <-- 170 <-- 1609 <-- 4589 <-- 4297 <-- 720 <-- 2048 <-- 4090 <-- 3938 <-- 3790 <-- 3148 <-- 3147 <-- 2160 <-- 2161 <-- 2596 <-- 3107

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=5404
Maximum bandwidth path from vertex 3107 to vertex 2397 :
2397 <-- 2396 <-- 2911 <-- 1123 <-- 617 <-- 1839 <-- 1861 <-- 1860 <-- 3910 <-- 2868 <-- 2867 <-- 3475 <-- 4414 <-- 4413 <-- 1045 <-- 1044 <-- 2254 <-- 3136 <-- 4094 <-- 4093 <-- 4092 <-- 206 <-- 2189 <-- 883 <-- 884 <-- 885 <-- 504 <-- 1323 <-- 1322 <-- 2943 <-- 1129 <-- 2094 <-- 2606 <-- 269 <-- 270 <-- 197 <-- 198 <-- 199 <-- 177 <-- 897 <-- 1966 <-- 1965 <-- 3198 <-- 783 <-- 3375 <-- 1019 <-- 1020 <-- 4905 <-- 2277 <-- 2278 <-- 2469 <-- 2468 <-- 4536 <-- 1116 <-- 4667 <-- 4668 <-- 2369 <-- 4543 <-- 2492 <-- 3963 <-- 1532 <-- 1531 <-- 4816 <-- 717 <-- 718 <-- 4817 <-- 3426 <-- 1753 <-- 2253 <-- 1275 <-- 194 <-- 1009 <-- 2705 <-- 4156 <-- 1607 <-- 4844 <-- 4843 <-- 820 <-- 1900 <-- 1899 <-- 3147 <-- 2160 <-- 2161 <-- 2596 <-- 3107
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Dikjstra without heap
122274 71661 138032 14655 135461
average = 19283
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Dikjstra with heap
3833 2502 4368 3439 4377
average = 740
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Modification of Kruskal
11117 584 515 624 612
average = 538

**b. <u>Graph with average vertex degree of 8 ( Output for five source to destination pairs )</u>**

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 9983
Maximum bandwidth path from vertex 3551 to vertex 1522 :
1522 <-- 636 <-- 120 <-- 136 <-- 3386 <-- 663 <-- 667 <-- 110 <-- 4592 <-- 3594 <-- 992 <-- 3413 <-- 2730 <-- 2511 <-- 448 <-- 1259 <-- 3502 <-- 996 <-- 3354 <-- 3513 <-- 2174 <-- 2354 <-- 3566 <-- 2779 <-- 1601 <-- 1041 <-- 2311 <-- 674 <-- 856 <-- 1776 <-- 2986 <-- 2359 <-- 1783 <-- 2286 <-- 424 <-- 1650 <-- 1501 <-- 529 <-- 1310 <-- 314 <-- 809 <-- 2602 <-- 1155 <-- 3283 <-- 1855 <-- 1359 <-- 1215 <-- 1671 <-- 803 <-- 427 <-- 784 <-- 1521 <-- 3702 <-- 1260 <-- 2930 <-- 3481 <-- 905 <-- 398 <-- 1291 <-- 971 <-- 3664 <-- 104 <-- 4389 <-- 3306 <-- 3660 <-- 405 <-- 3152 <-- 4265 <-- 3551

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=9983
Maximum bandwidth path from vertex 3551 to vertex 1522 :
1522 <-- 636 <-- 120 <-- 136 <-- 3386 <-- 663 <-- 667 <-- 110 <-- 4592 <-- 3594 <-- 992 <-- 3413 <-- 2730 <-- 622 <-- 3725 <-- 493 <-- 1182 <-- 2077 <-- 146 <-- 3481 <-- 905 <-- 398 <-- 1291 <-- 971 <-- 3664 <-- 104 <-- 4389 <-- 3306 <-- 3660 <-- 405 <-- 3152 <-- 4265 <-- 3551

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=9983
Maximum bandwidth path from vertex 3551 to vertex 1522 :
1522 <-- 636 <-- 120 <-- 136 <-- 3386 <-- 663 <-- 667 <-- 110 <-- 4592 <-- 3594 <-- 992 <-- 841 <-- 332 <-- 4518 <-- 398 <-- 1291 <-- 971 <-- 3664 <-- 104 <-- 4389 <-- 3306 <-- 3660 <-- 405 <-- 3152 <-- 4265 <-- 3551

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 9979
Maximum bandwidth path from vertex 3771 to vertex 553 :
553 <-- 3056 <-- 2312 <-- 2665 <-- 2999 <-- 501 <-- 2588 <-- 1019 <-- 2292 <-- 607 <-- 2390 <-- 2609 <-- 1364 <-- 347 <-- 1280 <-- 516 <-- 803 <-- 427 <-- 784 <-- 2748 <-- 1381 <-- 2333 <-- 1558 <-- 679 <-- 580 <-- 2721 <-- 764 <-- 82 <-- 1853 <-- 1538 <-- 2818 <-- 1282 <-- 2981 <-- 150 <-- 2808 <-- 1486 <-- 1109 <-- 2302 <-- 1223 <-- 2154 <-- 1300 <-- 3377 <-- 3771

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=9979
Maximum bandwidth path from vertex 3771 to vertex 553 :
553 <-- 1539 <-- 4187 <-- 669 <-- 3426 <-- 4119 <-- 1393 <-- 898 <-- 1411 <-- 2200 <-- 1778 <-- 3847 <-- 1442 <-- 2232 <-- 1257 <-- 3829 <-- 3345 <-- 1872 <-- 2791 <-- 440 <-- 3991 <-- 768 <-- 4536 <-- 3164 <-- 3775 <-- 674 <-- 856 <-- 1776 <-- 2986 <-- 4033 <-- 4671 <-- 4357 <-- 806 <-- 1603 <-- 223 <-- 3191 <-- 3377 <-- 3771

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=9979
Maximum bandwidth path from vertex 3771 to vertex 553 :
553 <-- 3056 <-- 992 <-- 841 <-- 332 <-- 4518 <-- 1902 <-- 150 <-- 2808 <-- 1486 <-- 1109 <-- 2302 <-- 1223 <-- 2154 <-- 1300 <-- 3377 <-- 3771

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 9989
Maximum bandwidth path from vertex 1671 to vertex 4325 :
4325 <-- 2780 <-- 3365 <-- 2923 <-- 2606 <-- 4736 <-- 4456 <-- 1123 <-- 1541 <-- 4593 <-- 1474 <-- 346 <-- 2172 <-- 1299 <-- 2516 <-- 3009 <-- 3594 <-- 1756 <-- 2901 <-- 3685 <-- 1368 <-- 7 <-- 1783 <-- 2286 <-- 424 <-- 1650 <-- 1501 <-- 529 <-- 1310 <-- 314 <-- 809 <-- 2602 <-- 1155 <-- 3283 <-- 786 <-- 179 <-- 4144 <-- 245 <-- 1612 <-- 1990 <-- 2109 <-- 3459 <-- 4496 <-- 2779 <-- 1601 <-- 270 <-- 665 <-- 3700 <-- 4186 <-- 1031 <-- 3779 <-- 4791 <-- 3787 <-- 2282 <-- 4748 <-- 2887 <-- 502 <-- 462 <-- 1424 <-- 3833 <-- 1544 <-- 1931 <-- 1280 <-- 516 <-- 803 <-- 1671

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=9989
Maximum bandwidth path from vertex 1671 to vertex 4325 :
4325 <-- 2780 <-- 3365 <-- 2923 <-- 2606 <-- 4736 <-- 2858 <-- 4362 <-- 735 <-- 2371 <-- 2024 <-- 4771 <-- 154 <-- 529 <-- 1310 <-- 314 <-- 809 <-- 2602 <-- 1155 <-- 3283 <-- 786 <-- 179 <-- 4144 <-- 245 <-- 1612 <-- 1990 <-- 2109 <-- 3459 <-- 4496 <-- 2779 <-- 1601 <-- 270 <-- 665 <-- 3700 <-- 4186 <-- 1031 <-- 3779 <-- 4791 <-- 3787 <-- 2282 <-- 4748 <-- 2887 <-- 502 <-- 462 <-- 1424 <-- 3833 <-- 1544 <-- 1931 <-- 1280 <-- 516 <-- 803 <-- 1671

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=9989
Maximum bandwidth path from vertex 1671 to vertex 4325 :
4325 <-- 2780 <-- 3365 <-- 2923 <-- 2606 <-- 4736 <-- 2858 <-- 4362 <-- 735 <-- 2371 <-- 2024 <-- 4771 <-- 154 <-- 529 <-- 1310 <-- 314 <-- 809 <-- 2602 <-- 1155 <-- 3283 <-- 786 <-- 179 <-- 4144 <-- 245 <-- 1612 <-- 1990 <-- 2109 <-- 3459 <-- 4496 <-- 2779 <-- 1601 <-- 270 <-- 665 <-- 3700 <-- 4186 <-- 1031 <-- 3779 <-- 4791 <-- 3787 <-- 2282 <-- 4748 <-- 2887 <-- 502 <-- 462 <-- 1424 <-- 3833 <-- 1544 <-- 1931 <-- 1280 <-- 516 <-- 803 <-- 1671

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 9981
Maximum bandwidth path from vertex 107 to vertex 2817 :
2817 <-- 142 <-- 1118 <-- 141 <-- 4256 <-- 4768 <-- 449 <-- 3571 <-- 4726 <-- 2975 <-- 1101 <-- 1248 <-- 3873 <-- 2965 <-- 4629 <-- 2655 <-- 3527 <-- 1738 <-- 1490 <-- 2444 <-- 4538 <-- 1368 <-- 3685 <-- 2901 <-- 1756 <-- 3594 <-- 3009 <-- 2516 <-- 1299 <-- 2172 <-- 346 <-- 1474 <-- 4593 <-- 1541 <-- 1123 <-- 1693 <-- 4984 <-- 331 <-- 3648 <-- 3595 <-- 4617 <-- 2949 <-- 1986 <-- 3041 <-- 1884 <-- 4495 <-- 1108 <-- 107

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=9981
Maximum bandwidth path from vertex 107 to vertex 2817 :
2817 <-- 142 <-- 1118 <-- 141 <-- 4256 <-- 4768 <-- 449 <-- 3571 <-- 4726 <-- 2975 <-- 1101 <-- 1248 <-- 3873 <-- 2617 <-- 2616 <-- 4854 <-- 4218 <-- 4880 <-- 4920 <-- 2821 <-- 3512 <-- 1116 <-- 600 <-- 753 <-- 4081 <-- 3286 <-- 4041 <-- 3249 <-- 2030 <-- 2711 <-- 4659 <-- 1600 <-- 1398 <-- 379 <-- 4237 <-- 2856 <-- 4593 <-- 1541 <-- 1123 <-- 1693 <-- 4984 <-- 331 <-- 3648 <-- 3595 <-- 4617 <-- 2949 <-- 1986 <-- 3041 <-- 1884 <-- 4495 <-- 1108 <-- 107

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=9981
Maximum bandwidth path from vertex 107 to vertex 2817 :
2817 <-- 142 <-- 1118 <-- 141 <-- 4256 <-- 4768 <-- 449 <-- 3571 <-- 4726 <-- 2975 <-- 1101 <-- 1248 <-- 3873 <-- 2617 <-- 2616 <-- 4854 <-- 4218 <-- 4880 <-- 4920 <-- 2179 <-- 1614 <-- 394 <-- 3630 <-- 782 <-- 3946 <-- 4533 <-- 3122 <-- 2316 <-- 4041 <-- 3249 <-- 4348 <-- 2836 <-- 2880 <-- 1571 <-- 1539 <-- 4578 <-- 3837 <-- 281 <-- 2861 <-- 3777 <-- 1235 <-- 2978 <-- 4109 <-- 314 <-- 1310 <-- 529 <-- 154 <-- 4771 <-- 2024 <-- 2371 <-- 735 <-- 4362 <-- 2858 <-- 4736 <-- 4456 <-- 1123 <-- 1693 <-- 4984 <-- 331 <-- 3648 <-- 3595 <-- 4617 <-- 2949 <-- 1986 <-- 3041 <-- 1884 <-- 4495 <-- 1108 <-- 107

****Maximum Bandwidth using Dikjstra without heap****
Maximum bandwidth = 9986
Maximum bandwidth path from vertex 742 to vertex 3367 :
3367 <-- 2316 <-- 4041 <-- 3286 <-- 999 <-- 1227 <-- 1207 <-- 1093 <-- 1531 <-- 2130 <-- 3406 <-- 11 <-- 2232 <-- 3408 <-- 360 <-- 3819 <-- 996 <-- 3354 <-- 2902 <-- 4119 <-- 1393 <-- 4428 <-- 3387 <-- 4302 <-- 3140 <-- 1012 <-- 3381 <-- 4245 <-- 4368 <-- 742

****Maximum Bandwidth using Dikjstra with heap****
Maximum bandwidth=9986
Maximum bandwidth path from vertex 742 to vertex 3367 :
3367 <-- 2316 <-- 4041 <-- 3286 <-- 4081 <-- 753 <-- 600 <-- 1116 <-- 2200 <-- 3472 <-- 3739 <-- 2892 <-- 494 <-- 1337 <-- 4250 <-- 1601 <-- 2779 <-- 3566 <-- 167 <-- 1620 <-- 2270 <-- 1777 <-- 4109 <-- 2978 <-- 1807 <-- 846 <-- 3314 <-- 4071 <-- 2439 <-- 70 <-- 4595 <-- 2260 <-- 4428 <-- 3387 <-- 4302 <-- 3140 <-- 1012 <-- 3381 <-- 4245 <-- 4368 <-- 742

****Maximum Bandwidth based on modification of Kruskal****
Maximum Bandwidth=9986
Maximum bandwidth path from vertex 742 to vertex 3367 :
3367 <-- 2316 <-- 4041 <-- 3249 <-- 4348 <-- 2836 <-- 2880 <-- 1571 <-- 1539 <-- 4578 <-- 3837 <-- 281 <-- 2861 <-- 3777 <-- 1235 <-- 2978 <-- 4109 <-- 314 <-- 809 <-- 2602 <-- 1155 <-- 3283 <-- 786 <-- 179 <-- 4144 <-- 245 <-- 1612 <-- 1990 <-- 2109 <-- 3459 <-- 4496 <-- 2779 <-- 1601 <-- 1041 <-- 2311 <-- 674 <-- 3775 <-- 419 <-- 2325 <-- 3563 <-- 1129 <-- 3582 <-- 4271 <-- 2439 <-- 70 <-- 4595 <-- 2260 <-- 4428 <-- 3387 <-- 4302 <-- 3140 <-- 1012 <-- 3381 <-- 4245 <-- 4368 <-- 742
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Dikjstra without heap
392628 101902 95949 467738 109436
average = 46706
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Dikjstra with heap
342772 357238 90339 373727 44385
average = 48338
---
Time taken in clock ticks by : Maximum Bandwidth Path based on Modification of Kruskal
646515 731 506 378 379
average = 25940