CSCE 629-601 Analysis of Algorithms
Network Optimization

1. INTRODUCTION:

Network Optimization has been an important area in the current research in computer science and engineering. The problem statement in this project is to find maximum bandwidth between two end points in a larger network by implementing three different algorithms and analysing the time complexity variation in each of them. For diversification, two types of networks are defined. The project report is divided into three major sections namely, creation of network, implementation of the algorithms to find the maximum bandwidth path and analysis of the results. There are two types of networks defined for better analysis of performance of each approach. For simplicity, now on we define the network as *Graph* and end points in the network as *Nodes* or *Vertices*. The path between each node is defined as an *Edge*. The maximum bandwidth capacity of each path is defined as the *Weight* of the edge. The project is implemented using C++ programming language and report gives detailed implementation of graphs and algorithms. Finally, the timing analysis of results obtained is explained in detailed in order to understand the functionality of each algorithm and choose a better approach based on the type of graph.

2. Graph Implementation:

There are two types of graphs that required for the analysis each of which has 5000 nodes. The first one being a graph whose average degree is 6 and the second one is a graph with average degree 1000. The term degree in graphs analysis refers to the number of neighbouring nodes for a given node. Since this experiment is like a real-time scenario where the data packets can move from one node to another irrespective of direction (full duplex nature of networks), the graphs are also implemented without any direction. i.e., given an edge between two nodes the data can move in either direction based on the capacity of the edges. Each node and edge are defined with certain parameters to store information. The graphs are implemented as adjacency lists which stores the node and weight information to which it is connected. The graph nodes are selected at random so that there is no determinism in the bandwidth calculations. The two different models of graphs are designed as defined in the problem statement. For simplicity and as described in the projects each node is named as integers ranging from 0 to 4999. Nodes and edges are defined as *structs* since they are required to store more than one type of data. Each node consists of *index* which will store the adjacent node number*, weight* of the edge between the source and adjacent node and the pointer variable to the *adjacent* node. The edge consists of *source* and *destination* vertices, *weight* of the edge, and *edge number*. All are defined as integer variables in each *struct*. An adjacency list is defined which stores the pointer variable to each node. Since the size of each adjacency list is unknown, we define the pointer variable and dynamically allocate memory to each node as the edges are created. Graph is defined as the collection of adjacency lists and edges. Graph also stores the information related to number of nodes and edge count which is useful in determining the degree of the graph for verification in later stages.

Node Creation:

CreateNode is a function defined to create a vertex in graph with random weight to the edge between the two nodes connecting an edge. Initially pointer to the next node is initialized to NULL. The function returns node type of data which will be used to created edges.

Edge Creation:

An edge is created between two given nodes using the pointer variables in each node. Since the graph is bidirectional (or undirected), when an edge is created between source and destination, a connection is made between the destination and source nodes as well. Both the nodes consist of same weight information due to

obvious reasons. The weights are randomly assigned between 20 and 10000 to signify the bandwidth of each path. The weights are chosen from the *uniform_int_distribution* class with random device seeds. Functionality is defined in *CreateEdge* method.

Sparse Graph Generation:

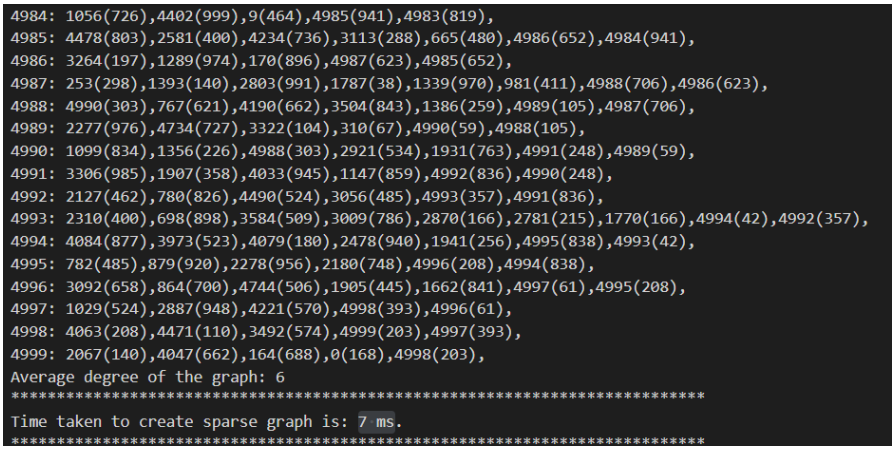
Sparse graph is defined to pick neighbours of node randomly such that the average degree of the graph remains 6. We have implemented two models where in each model has the average degree of 6. The first model is defined under the method, *GenerateSparseGraph* where in each node also has six number of neighbouring nodes at maximum chosen randomly. In order to achieve randomness, we have used the random number generator, *random device* defined in C++. This model is implemented in three stages. The first stage of implementation is to ensure that the graph is connected. An edge is created between each node and a circular connection is formed for all the 5000 edges. The second stage of implementation is to randomly choose a node among the available nodes and create a connection between them. This is achieved by using *uniform_int_distribution* class defined in random header of C++. The random number generator provides seed to the *uniform_int_distribution* class to pick values from given range. The range of values is defined as integers between 0 to 4999 as the maximum available nodes are 5000. A flag variable is created to determine if an edge exists between the randomly chosen node and the source node already. The second stage is repeated for considerable number of times for each node to add required number of neighbours. The third stage is an enhancer wherein the edges are added to all the nodes whose edge count does not match to 6. The nodes whose edge count does not satisfy the degree are picked and inserted into array named *balance_arr*. The size of the array is determined by the number of edges that are required to be added to achieve the degree condition. This stage is also repeated so that the degree is achieved as 6 for each remaining node. Finally, the graph of required degree is formed in this manner. An example of randomly generated graph with average degree 6 is present in *SparseGraph.txt* file attached along the project codes. The time taken to create this graph is *28 ms*.

```
4984: 4170(633),3449(803),1738(336),1488(998),4985(239),4983(759),
4985: 3973(785),3328(83),3066(122),1437(613),4986(188),4984(239),
4986: 2226(894),2190(917),536(652),43(802),4987(629),4985(188),
4987: 1862(869),1590(768),868(417),307(705),4988(260),4986(629),
4988: 3537(348),3388(188),2939(220),2544(65),4989(507),4987(260),
4989: 3886(906),3023(941),2700(939),694(813),4990(267),4988(507),
4990: 3648(354),2851(392),1037(834),15(135),4991(700),4989(267),
4991: 2198(35),2118(980),1671(546),149(783),4992(703),4990(700),
4992: 3716(534),3203(450),2407(474),1254(260),4993(102),4991(703),
4993: 4875(834),4480(218),2968(823),2717(153),4994(106),4992(102),
4994: 1667(884),1098(244),386(414),196(198),4995(912),4993(106),
4995: 4580(983),3434(359),3080(480),611(289),4996(694),4994(912),
4996: 1665(617),747(492),511(63),331(775),4997(699),4995(694),
4997: 1989(631),1843(131),958(477),243(760),4998(382),4996(699),
4998: 4044(213),3458(723),2911(578),2294(198),4999(71),4997(382),
4999: 3924(124),3483(102),426(373),0(595),4998(71),4983(65),
Average degree of the graph: 6
```

Sparse Graph adjacency list

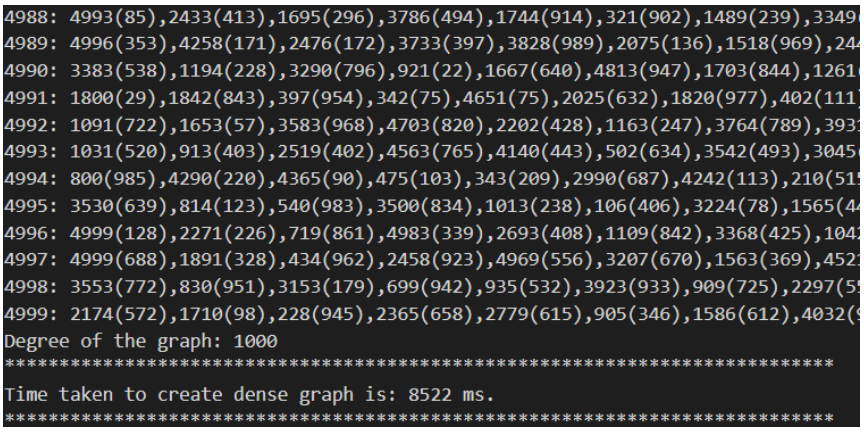
Another model of Sparse graph is also defined where the average degree is monitored to be 6. Degree of each node is not taken into consideration. This is defined in two stages, where the first stage is to ensure the connection between each node and the second stage is to loop until the degree of each node is 6. *Uniform_int_distribution* is used for determining a random node and edge is created if the obtained random node is not already a neighbouring node. Check flag is implemented to determine the list of nodes adjacent to a given source and edge is created between the nodes based on the flag outputs. If a random number

matches the existing list of neighbours, then the loop count is decremented and randomization is reiterated. Method is defined in *CreateSparseGraph_Avg* and time taken to construct graph in this method is *7 ms*.



Dense Graph Generation:

Dense graph is a network in which each node is adjacent to about 20% of the nodes in the network. Given network size is 5000 nodes and 20% neighbours say that the average degree of the graph is 1000. The dense graph generation algorithm is defined in the *CreateDenseGraph* method. The graph is developed in two stages similar to the sparse graph. Each node is connected to its neighbouring vertices to create a cycle and ensure connectivity of entire network. *Uniform_int_distribution* class is provided with random number generator and a time varying seed to randomly pick a node and create edge between a source and randomly chosen node. The method is looped until the average graph degree is achieved. If a repetition is observed the loop count is decremented and randomization is reiterated. Since the problem statement suggested that the number of neighbours can also be random, the algorithm generates the graph of average degree of 1000 without any constraints over each node. An example of dense graph is present in *DenseGraph.txt* file along with codes. The time taken to generate the such a graph is *8.5 sec*.



Adding Edge in Graph:

As said earlier, graph is defined as the combination of adjacency lists of nodes and edges which store the edge information between the nodes in adjacency lists. Since the connection of nodes is made using pointers and is stored only in the nodes, we use the edges to determine the node connections in the graph and degree. *AddEdgeInGraph* is a function that is defined to create edges between the given nodes made using the arguments of function and the edges are in turn added into the graphs to store the path information. Each node and edge are created using dynamic memory allocation function (*malloc*) and added into the graph.

3. Implementation of Algorithms:

There are three algorithms that are designed to identify the maximum bandwidth path between the source and destination vertices in a network. Implementation of each is given with a detailed explanation of the functions to achieve each step. No inbuilt or strong functions are used in the implementation of these functions. Only arrays and primitive datatypes present in C++ are used for the various functional aspects of algorithms and are defined as below:

Dijkstra's Maximum bandwidth:

The first approach is using the modified version of Dijkstra's single source shortest path algorithm to identify the maximum bandwidth from a given single source to destination. Dijkstra's algorithm is a greedy method which will choose the local optimal solution and proceeds further till the end to define the best possible path. The modified algorithm is to identify the local maximum bandwidth path from source and choose greedily the next optimum paths which are suitable for maximum bandwidth transfer between the two end points. This method is defined without using heap.

There are three statuses for each node namely, *unseen, fringe,* and *in-tree* defined as -1, 0, 1 respectively. As all the vertices are visited the status changes from unseen (-1) → fringe (0)→ in-tree (1) and process terminate, when all the vertices are marked as in-tree. We initialise a status array, parent array and bandwidth array to -*1* for each node. Data corresponding to each node is retrieved by using array indexing method from each of these arrays, as the identities of vertices are stored as integers. The first step of algorithm is to identify the source node and mark it as in-tree (*1*). Once the source node is identified we look for all vertices from source node and identify neighbours to mark their status as *0*. This is achieved using a temporary node pointer which is first pointing to the head of the adjacency list of the source node. Since the list consists of all the neighbours of source node, status of each neighbouring node is updated in the status array using array indexing where the index is the neighbour identity (of integer type) stored in *Node→index*. The bandwidth of the corresponding path is updated in the bandwidth array from *Node→weight.* Parent of each neighbouring node is marked as the source itself as the path starts from source.

A function to check if fringes exist in the network is defined as *CheckFringe* which returns true if at least one of the nodes is still a fringe. The algorithm is expected to proceed till all the fringes are marked as seen. A function *GetMaxFringe* is defined to identify the maximum available fringe from the source. Maximum fringe is a fringe that has maximum bandwidth path from source. *MinFunction* to return the minimum value of two variables is defined also defined to identify the optimum bandwidth to subsequent paths.

The functionality is to loop until all the vertices are marked as seen and optimum solution for maximum bandwidth is achieved. Hence a *while loop* is initialized to *CheckFringe* from the status array and *GetMaxFringe* to find the local maxima. This max fringe is marked as *1* and we proceed further by marking all the neighbours of maximum fringe accordingly based on their status. A nested while loop is initialized for the neighbours of local maximum fringe. If the status of each neighbour is *0* then we mark it as fringe (-*1*) and check if the bandwidth of the path is less than the minimum of existing maximum bandwidth and current path bandwidth. Since path cannot allow more than the maximum weight of that path, bandwidth of that node is updated using the *MinFunction.* Parent of that fringe is set to the local maximum fringe obtained in the previous step. If the status is already fringe (*1*) to that node, we check if the bandwidth allows more than the current maximum from source. If it allows, we keep the same bandwidth and if it does not allow, the bandwidth is again updated using the *MinFunction* defined earlier to return minimum value. This procedure is continued until all the fringes are marked as seen (*0*). The bandwidth of each node is updated with maximum allowable bandwidth.

As the status of all the nodes is marked as seen by above nested while loop, the status of destination is checked if it is marked as *0.* Ideally the connection between the source and destination may or may not exist. In our case as the entire network is connected and the updated bandwidth in each case is the maximum allowable bandwidth of the path, we identify the maximum bandwidth as the maximum allowable bandwidth to the destination from bandwidth array. This will be returned as the maximum bandwidth and the path is retrieved from the parent array updated for each iteration. Since the number of nodes is already known, the arrays are initialized statically with the respective node count. This can also be done from the node count information chosen from the graph. This algorithm takes the graph, source node and destination node as input and returns the maximum bandwidth path between the source node and destination node and each path bandwidth. The execution of this algorithm is shown in results section described later in the report.

```
Maximum Bandwidth is: 2661
Path: 119<-1178<-1183<-2261<-2674<-1039<-4162<-1759<-3614
********************************************************************************
Time taken (Dijkstra No Heap on Sparse Graph): 118 ms
********************************************************************************
```

Dijkstra's Maximum Bandwidth with Heap:

The approach in this method is to store the fringes in the heap instead of an array. This will reduce the search time complexity for the fringes and will have better performance when compared to the path finding algorithm without heap. In order to implement heap with primitive data types, there are methods defined for the functionality of heap insertion, deletion, maximum, and update functionalities. Before we proceed further to implement the algorithm, we first define the Max Heap implementation. This is the second task in the project and is implemented using the following functions.

Heap Creation and functionalities:

A heap array is created to store nodes of the graph. The index of the heap array is used to access the individual nodes. The Max heap is created from the heap array based on the bandwidth of each node. The Max heap consists of *position* pointer which is used to retrieve elements from heap, *curr_size* to store the current size and *total_size* to store the maximum size of the heap which is based on the number of nodes in the graph. Initially, max heap is created with maximum size and position pointer is pointing to -1 index of the array. This signifies that data in the heap elements is empty when the heap is created. This is created using *CreateMaxHeap* function and elements of the heap are created using the *CreateMaxHeapNode* function. This function will create nodes similar to the node structure of the graph and inserts into the heap. It is useful to create fringes in Dijkstra's algorithm. MaxHeap consists of a dynamic array which will continually increase in size based on the number of fringes and maximum bandwidth of the fringes.

Insertion:

Max heap is a heap (almost complete binary tree) that satisfies a condition in which parent of any subtree is greater than either of the two children. This heap structure is useful in determining the maximum bandwidth for each fringe in the network when the heap is created using this principle. A new element is added into the heap using the *InsertFringes* function defined to create and add elements into the heap. This function has prototypes to create nodes using *CreateMaxHeapNode* subroutine and insert elements into the heap provided as arguments. The insertion function follows *heapifyUp* and *HeapifyDown* to heapify the max heap created. These functionalities take the heap as input and perform the max heap balancing procedure from the index

position where the new element has been added. When a new node is added to the heap, the current size of the heap is increased and the position will be pointing to the last element of the heap. The final heap is returned after the heapify process.

Deletion:

To delete a specific node from the heap we can swap the specific node with the last node and initialize a pointer to store the last replaced node. The heap size is reduced to signify the deletion procedure and apply the *heapifyUp* and *heapifyDown* procedure at the index where element is replaced. Return the element that is pointed by the node pointer.

Maximum:

The maximum value of the max heap is stored at the root. Hence when we try to retrieve the max fringe, we use the deletion procedure of returning the root (element at the zeroth index). This follows the heap balancing procedures described below. *GetMaxFringe* function returns the root element if the heap is not empty and follows the procedure of updating the heap structure and this method is useful in determining maximum fringe.

Balancing the heap:

When a new node is added into the heap, *heapifyUp* will perform the balancing procedure from the position where the node is inserted till the root node and *heapifyDown* will perform the balancing from the inserted position till the leaf node. Though the heap is tree like structure in logical sense, the representation is done using the arrays where the relationship between the parent and children is defined as follows:

Parent: i; left_child = 2*i +1; right_child = 2*i+2;
i = index

heapifyUp function compares to see if the bandwidth of the parent is less than the bandwidth of children at the position where new nodes are added/updated and proceeds further till the root by swapping the elements. *HeapifyDown* does the comparison function to the lower side of the i^{th} position till the leaf node.

When a new node is added or deleted or updated based on the bandwidth values, the *heapifyUp* and *HeapifyDown* functions are called everytime to maintain the balanced structure of the heap. The condition to balance the heap could depend on the value that is being monitored in the heaps. In our case we monitor the heap using the bandwidth of each fringe. *SwapNodes* is a function defined to swap the nodes of max heap if the condition for max heap is not satisfied at every level. It takes two nodes as inputs and swaps their position in the max heap array.

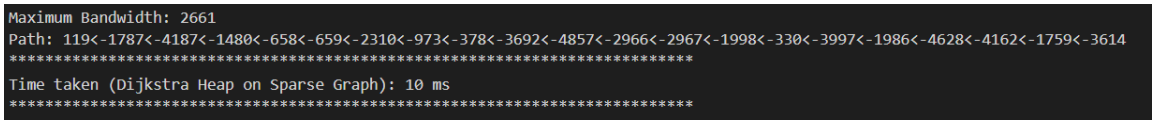
UpdateFringeHeap is a function defined specifically for the maximum bandwidth calculation as at every step we compare and update the fringe bandwidth. Once the bandwidth of the specific fringe is updated the heap array must be updated. This function calls the heap balancing methods to maintain the heap structure. *isEmpty* is a Boolean function that returns the current size of the heap. Since the Dijkstra runs until all the fringes are marked as visited, we need to check the size of the fringe heap which changes dynamically throughout the algorithm.

CSCE 629-601 Analysis of Algorithms
Network Optimization

Algorithm with Heap:

The algorithm design remains the same but the data structure used to store elements will change and has a huge performance impact. We store the elements and retrieve from max heap. Therefore, the process related to fringe will all be modelled using heap operations defined above. Three integer arrays to store the status, parent and bandwidth information of each node are initiated to -1. The status description is similar to previous model with unseen = -1; fringe = 0; in-tree = 1; The actual algorithm starts by making the source status as seen (1). Check for all the neighbours of source and insert into the max heap created with *CreateMaxHeap* function. For each new fringe inserted into the maxheap, since the algorithms run balancing functions, we need not re-run the balancing algorithms and mark the parent of these nodes as source node. Start a loop function until the size of the heap is empty i.e, there are no more fringes in the max heap. We use *isEmpty* function to determine if the max heap is empty and use *GetMaxFringe* function iteratively to obtain the local maximum bandwidth. Once the maximum fringe is identified from *GetMaxFringe* function, mark the status of maximum fringe to *seen* (*1*), create a temporary node to iterate through all the neighbours of maximum fringe and mark them all as fringes. Insert these fringes into the max heap if the status is unseen and mark the parents of these nodes as max fringe obtained in the previous step. Update the bandwidth array with *MinFunction* results to compare the actual fringes bandwidth and the maximum fringe bandwidth. While searching for the fringes if the nodes are already visited once (i.e., the nodes are already marked as fringes), compare the bandwidth achieved so far with the existing bandwidth of the fringe using *MinFunction* and again update the bandwidth array for that node. Also, apply the *UpdateFringeheap* procedure to update the bandwidth of that node in the max heap array and balance as per the max heap conditions.

Once all the nodes are marked as seen, the size of the heap is reduced to zero and the *isEmpty* function returns true making while condition false and the loop is exited. Check for the status of the destination and identify the bandwidth of the destination node. Since we keep updating the bandwidth of each node as per the maximum bandwidth allowed through the path, the maximum bandwidth will be given by the bandwidth of the destination. The path from the source to destination is given by parent array. This procedure might also return new path and is expected to be achieved in lesser time due to enhanced data structure. The only stable element in both the procedures is the *Maximum Bandwidth.* The analysis of results is described later in the report.



Kruskal's Maximum Bandwidth:

Default Kruskal's algorithm is used to construct the minimum spanning tree. The spanning tree is a structure that connects all the vertices in the graph and the minimum spanning tree is a structure that connects all the nodes with minimum weight of edges. Modification of Kruskal's algorithm results in determining the maximum bandwidth. This can be achieved by sorting the edges in *non-increasing order* instead of non-decreasing order as preferred in conventional model. Once the edges are sorted construct the maximum spanning tree and build a path using *breadth first search* in the maximum spanning tree between the two end points. As the connection is made using maximum spanning tree, the edge weight is maximum as required by the maximum bandwidth path. We implement the Kruskal's algorithm to first construct the Kruskal's Maximum Spanning Tree using *Make-Set, Find* and *Union* Operations and then implement *BFS* using *queues* to define the path from given source to destination. Various functions are defined for the BFS, *Make-set, Find* and *Union*

operations. Kruskal's algorithm requires the edge information to consider the maximum spanning tree and hence to construct an MST we need the edges to be sorted as described above and this operation is performed using a heap. To do this we define functions to create and store edges in heap. Ideally the data structures used in the implementation is an array with dynamic memory allocation.

Kruskal's Edge Heap Implementation:

CreateNewVertex is a function to create an edge that is to be stored in the edge heap. We use dynamic memory allocation function *malloc()* to create new edge structure defined specifically for the edge in heap and the edge information from the graph is stored in this vertex stored in edge-heap. Hence the source, weight and destination of actual edges in graph are pointed by the source, weight and destination of the vertex created by this function. Return type of the function is structure that stores the edge information. We use a function to create a heap which increases dynamically for every new edge observed in the graph. *CreateNewHeap* is a heterogeneous datatype that stores the pointer to heap edges, current heap size and total heap size. Since the heap size is unknown, we use the pointer to point the current vertex in the heap. Each vertex in the heap stores the edge data. *InsertEdgeInHeap* function adds the edges into the heap. It simply increases the size of the heap and uses the pointer to point the last node and new edge is inserted into raw heap. Once the edge is inserted, we perform *heapsort* which will balance the heap as per the conditions. The expectation to create a maximum spanning tree from this graph requires the edges to be sorted in non-increasing order and max heap will be ideal implementation as the data can be retrieved in non-increasing order from the max heap. Therefore, we define the *heapifyQueUp* and *HeapifyQueDown* functions to sort the heap from the point where the new edge data is inserted. The conditions for heap are like what has been given for the Heap implementation in *Dijkstra's* with parent=*i* left_child=*2*i+1* and right_child=*2*i+2.* The raw heap created by *InsertEdgeInHeap* function will be subjected to heap balancing functions. Data retrieved from the heap is then used to construct the Maximum Spanning tree. The *Maximum Spanning Tree* is a graph that connects all the nodes with only the maximum weighted edges. Since this is also a graph, we need to define the graph data type specific to MST as a struct that stores the vertices and pointers to edges. The MST graph consists of adjacency list of nodes that spans dynamically as the edges data obtained from the edge heap. *MakeSet(), Find(), Union()* operations are defined for creating MST using rank. *MakeSet* method only initialises the parent of each vertex and the corresponding rank. *Find* method checks if the given new vertex is already a part of the spanning tree using *while loop*. If not finds the vertex which can make the given new vertex as a part of the spanning tree. *Union* function changes the rank of two nodes by calculating if they are connected in the graph. Once the rank is updated by the union method, the two separate nodes are added together into the queue which is creating the maximum spanning tree. The rank value updates the parent of the either of nodes creating a logical connection between them resembling a spanning tree edge connection.

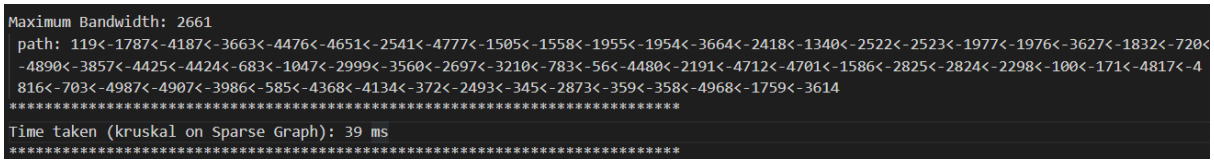
Queue Implementation for Breadth First Search:

Once the maximum spanning tree is constructed, we need to implement the BFS in order to identify the path from source to destination. We use Breadth first search since this returns the shortest path between two end points. Queue operations are defined which are basic blocks for BFS algorithm. *Que* is custom defined data structure which has current size, total size, pointers to front and rear ends of the queue. The Queue developed here will follow the same *FIFO* principle *(First In First Out). CreateQueue* is a function defined to create one such queue and called only once during program execution, which will be later updated during BFS. We define *IsQueEmpty()*, *IsQueFull()*, *Enque(), Deque()* functions for obvious reasons of queue operations.

Algorithm for Maximum bandwidth:

The Kruskal Algorithm for maximum bandwidth calculation is defined in the *Kruskal* and *BreadthFirstSearch* methods defined at the end of the *Kruskal.cpp* file. The algorithm begins with creating a heap to store edge data using *CreateEdgeHeap* and a method to create a maximum spanning tree *createMSTGraph* to create a graph of maximum spanning tree. While loop is used until all the nodes of heap are initialized and inserted as per the given graph input. This heap is subjected to sort using *HeapSort*. Once all the edges in the heap are sorted based on max heap creation, rank and parent arrays are created using *malloc()* functions. We initialize the rank and parent of each heap node (edges data present in the heap) using *MakeSet* method. We initialize two variables to store the edge count and the other to calculate and increment the rank of each element. A while loop is initialized to find and merge two pieces together that makes a connection in maximum spanning tree. This loop continues to execute until all the edges are visited and the heap size is reduced to zero. This loop will first track the rank of each node in the graph and is updated in the rank array. *Find* method is called to identify which of the node would be optimal selection for each subtree to be connected and returns the rank. *Union* method is called if the rank of any two elements is not same and a logical connection is created by the union method by updating the ranks of each node and *CreateMSTEdege* function will add these edges into the graph designed earlier using *createMSTGraph* method. By the end of this loop, we will have all the nodes connected with each other with the maximum edge weight possible. This spanning tree is passed as input to the *BreadthFirstSearch* method which will find the quickest path between two nodes. The algorithm prints the path if exists if not returns no path found.

BreadthFirstSearch is implemented using queues that are defined above. Three arrays namely parent, status, bandwidth, are initialized. The bandwidth is initially set to maximum value (*INT_MAX*) and later updated as per the weight of the edge between each node. There are two statuses for each node unseen=0, visited=1. We initialize the source node and set its status to visited (status[source]=1). Initialize a destination variable to store the identity of destination which is used to check if path is found. Add the source to queue using *Enque* method. Iterate a while loop until the queue size is empty which is checked using *IsQueEmpty* method. Pick the outgoing element from the queue and check for the status of each node connected with it. If the status is unseen, add into the queue. Update the bandwidth array and parent array with respective information. If the destination element is found anytime during the execution process for each queue element, break the loop and continue with the next part of execution of program. After the while loop execution if the status of destination element is not changed then return that path is not found between the two points. If the path exists. Print the bandwidth of the destination element from the bandwidth array as the maximum bandwidth. The path is also returned by the parent array. Since the path found is created from the maximum spanning tree, we can conclude that the bandwidth will be maximum through that path between the two nodes. Results and analysis are provided in the next section.



4. Analysis of the Algorithms:

The graph algorithms defined in the previous section are run on a random graph generated from subroutines developed earlier. The three methods to find the maximum bandwidth between two endpoints are run on 5 pairs of randomly chosen source and destination nodes. This procedure is implemented for random graph

pairs generated 5 times. There are total of 5 sparse graphs and 5 dense graphs generated and each of these graphs are run with 5 different pairs of source and destination nodes to identify the maximum bandwidth path. The results of the program execution are shown in *Results.txt* file attached along with this project code. Two files called *SparseGraph.txt* and *DenseGraph.txt* are generated which contains adjacency list representations of sparse graph and dense graph respectively. To execute the project, compile all the source codes using g++ and run the executable. A final executable is also attached along with the project named as *project.exe*. Run the executable file to observe the details:

Run: 1

Source node: 3614 Destination node: 119

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2661	118 ms
	Dijkstra's heap	2661	10 ms
	Kruskal's	2661	39 ms
Dense Graph	Dijkstra's No heap	3293	211 ms
	Dijkstra's heap	3293	64 ms
	Kruskal's	3293	472 ms

Source node: 844 Destination node: 256

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2666	111
	Dijkstra's heap	2666	9
	Kruskal's	2666	33
Dense Graph	Dijkstra's No heap	3290	181
	Dijkstra's heap	3290	53
	Kruskal's	3290	358

Source node: 107 Destination node: 4125

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	1902	112
	Dijkstra's heap	1902	6
	Kruskal's	1902	31
Dense Graph	Dijkstra's No heap	3294	183
	Dijkstra's heap	3294	50
	Kruskal's	3294	367

Source node: 4003 Destination node: 2527

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2442	120
	Dijkstra's heap	2442	8
	Kruskal's	2442	28
Dense Graph	Dijkstra's No heap	3281	192
	Dijkstra's heap	3281	49
	Kruskal's	3281	378

CSCE 629-601 Analysis of Algorithms
Network Optimization

Source node: 3326 Destination node: 3711

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3031	118
	Dijkstra's heap	3031	12
	Kruskal's	3031	28
Dense Graph	Dijkstra's No heap	3215	190
	Dijkstra's heap	3215	46
	Kruskal's	3215	358

Run: 2

Source node: 2565 Destination node: 1199

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3064	119
	Dijkstra's heap	3064	12
	Kruskal's	3064	30
Dense Graph	Dijkstra's No heap	3274	119
	Dijkstra's heap	3274	55
	Kruskal's	3274	399

Source node: 594 Destination node: 932

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2977	114
	Dijkstra's heap	2977	15
	Kruskal's	2977	33
Dense Graph	Dijkstra's No heap	3287	187
	Dijkstra's heap	3287	48
	Kruskal's	3287	379

Source node: 1252 Destination node: 160

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2897	111
	Dijkstra's heap	2897	11
	Kruskal's	2897	27
Dense Graph	Dijkstra's No heap	3287	198
	Dijkstra's heap	3287	55
	Kruskal's	3287	401

Source node: 1046 Destination node: 557

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3058	126
	Dijkstra's heap	3058	11
	Kruskal's	3058	29
Dense Graph	Dijkstra's No heap	3226	194
	Dijkstra's heap	3226	48

	Kruskal's	3226	363

Source node: 206 Destination node: 910

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2918	122
	Dijkstra's heap	2918	11
	Kruskal's	2918	29
Dense Graph	Dijkstra's No heap	3283	188
	Dijkstra's heap	3283	49
	Kruskal's	3283	392

Run: 3

Source node: 54 Destination node: 63

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2948	119
	Dijkstra's heap	2948	9
	Kruskal's	2948	30
Dense Graph	Dijkstra's No heap	3288	200
	Dijkstra's heap	3288	50
	Kruskal's	3288	405

Source node: 73 Destination node: 3244

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3018	123
	Dijkstra's heap	3018	10
	Kruskal's	3018	32
Dense Graph	Dijkstra's No heap	3296	194
	Dijkstra's heap	3296	51
	Kruskal's	3296	378

Source node: 2263 Destination node: 4147

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3012	108
	Dijkstra's heap	3012	12
	Kruskal's	3012	28
Dense Graph	Dijkstra's No heap	3248	185
	Dijkstra's heap	3248	53
	Kruskal's	3248	412

Source node: 773 Destination node: 297

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3041	115
	Dijkstra's heap	3041	10
	Kruskal's	3041	29

Dense Graph	Dijkstra's No heap	3252	193
	Dijkstra's heap	3252	78
	Kruskal's	3252	391

Source node: 1646 Destination node: 2023

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2689	121
	Dijkstra's heap	2689	11
	Kruskal's	2689	31
Dense Graph	Dijkstra's No heap	3300	192
	Dijkstra's heap	3300	54
	Kruskal's	3300	383

Run: 4

Source node: 406 Destination node: 3615

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2017	121
	Dijkstra's heap	2017	12
	Kruskal's	2017	27
Dense Graph	Dijkstra's No heap	3288	204
	Dijkstra's heap	3288	53
	Kruskal's	3288	384

Source node: 88 Destination node: 3275

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2479	134
	Dijkstra's heap	2479	11
	Kruskal's	2479	27
Dense Graph	Dijkstra's No heap	3273	194
	Dijkstra's heap	3273	51
	Kruskal's	3273	388

Source node: 1500 Destination node: 4271

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2558	120
	Dijkstra's heap	2558	9
	Kruskal's	2558	26
Dense Graph	Dijkstra's No heap	3288	191
	Dijkstra's heap	3288	50
	Kruskal's	3288	388

Source node: 779 Destination node: 4784

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)

Sparse Graph	Dijkstra's No heap	2876	113
	Dijkstra's heap	2876	11
	Kruskal's	2876	26
Dense Graph	Dijkstra's No heap	3297	177
	Dijkstra's heap	3297	49
	Kruskal's	3297	354

Source node: 3475 Destination node: 411

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	3053	112
	Dijkstra's heap	3053	12
	Kruskal's	3053	26
Dense Graph	Dijkstra's No heap	3249	214
	Dijkstra's heap	3249	50
	Kruskal's	3249	339

Source node: 1433 Destination node: 1270

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2953	123
	Dijkstra's heap	2953	14
	Kruskal's	2953	31
Dense Graph	Dijkstra's No heap	3289	201
	Dijkstra's heap	3289	50
	Kruskal's	3289	361

Source node: 3887 Destination node: 2908

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2999	112
	Dijkstra's heap	2999	10
	Kruskal's	2999	29
Dense Graph	Dijkstra's No heap	3282	184
	Dijkstra's heap	3282	53
	Kruskal's	3282	415

Source node: 2721 Destination node: 3455

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2912	112
	Dijkstra's heap	2912	9
	Kruskal's	2912	32
Dense Graph	Dijkstra's No heap	3304	192
	Dijkstra's heap	3304	51
	Kruskal's	3304	382

Source node: 1738 Destination node: 4965

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2930	116
	Dijkstra's heap	2930	10
	Kruskal's	2930	31
Dense Graph	Dijkstra's No heap	3233	196
	Dijkstra's heap	3233	50
	Kruskal's	3233	375

Source node: 1037 Destination node: 3016

Graph type	Algorithm	Maximum Bandwidth	Running time (Milli Seconds)
Sparse Graph	Dijkstra's No heap	2814	117
	Dijkstra's heap	2814	12
	Kruskal's	2814	31
Dense Graph	Dijkstra's No heap	3300	188
	Dijkstra's heap	3300	49
	Kruskal's	3300	411

5. Timing Analysis:

In Each trial run, the time taken in the Dijkstra's heap is the smallest of all three approaches. The reason for this reduction is the style of implementation. Dijkstra's algorithm time complexity for initialization of arrays takes *O(n).* While loop is executed at most n-1 times because each vertex can move from fringe and then to seen status and this step is repeated for n vertices making the time complexity of while loop as $O(n^2)$. Therefore, the worst-case time complexity of the Dijkstra's algorithm is $O(n^2)$. On the other hand, Dijkstra implementation with loop analysis is based on each stage. The array initialization still takes *O(n)* time. Insertion into heap takes *O(logn)* to find the fringes and this is executed for at most n times to make all the nodes as fringes making the total executing time for fringes insert as *O(nlogn).* The max fringe and delete fringe step also take *O(logn)* time and the loop is executed for the maximum of m edges making the step total time complexity as *O(mlogn)*. Therefore, the total time complexity of the Dijkstra's algorithm with heap implementation takes *O((m+n)logn)*.

The Kruskal's algorithm takes two steps which is implementation of maximum spanning tree and then BFS on the obtained results. The edges are added into the heap and insertion takes time of *O(mlogm). Edges are s*orted using heap sort which takes *O(mlogm)* time. Time taken for makeSet, find and Union operations is *O(logn)* since all the elements are to be initialized, iterated over the heap to *find* specific vertex of a suitable rank and merge together. Time complexity for the breadth first search is *O(m+n).* All these operations together add to the time of Kruskal's approach. Though the time complexity is similar in both Dijkstra's and Kruskal's, the sorting step must be implemented after every union which requires to the loop to run over all the connected nodes for a given vertex making it slightly higher in the sparse network. The Kruskal's implementation requires longer duration in dense network because of number of edges. The operation to sort the number of edges in dense graph takes longer duration since there are 2.5 million edges and the heap sort

contributes to the larger part of execution time. Soon after the edges are sorted and maximum spanning tree is constructed the which is later subjected to breadth first search to construct path from source to destination.

Sparse Graph timing analysis:

There are 5000 nodes in the network and 15000 edges making the Dijkstra no heap algorithm run for *O(m*n)* time. Since the Dijkstra heap approach is running only for *O((m+n)logn) and m edges and n vertices,* the time complexity is reduced significantly in the logarithmic scale due to smaller multiplication factor. In the Kruskal's approach the amount of time taken for the heap sort, addition into the heap, deletion from heap, make-set, find, union operations contribute to time. Though the time complexity for each step is *O(mlogm)* and is similar to Dijkstra's with heap, the practical time in Kruskal's implementation is larger than that of Dijkstra's implementation. Since the number of steps are more in Kruskal's implementation and the linear time contribution by breadth first search to construct path between source and node increases the time take by Kruskal's method. Hence the results described above depict the nature of algorithms with Dijkstra's heap being the smallest time, followed by Kruskal's approach and the Dijkstra's no heap which takes quadratic time to finish algorithm.

Dense Graph timing analysis:

The worst time complexities remain the same in dense network for each of the described algorithms but the number of edges increase drastically due to the connection nature of nodes in the network. Since there are 20% neighbours to each node which creates an average of 1000 degree for each node, the total number of edges in the network become 2.5 million. As the multiplication factor to the logarithm scale *increases more than quadratic value* in Dijkstra's no heap due to number of edges and the breadth first search also attributes to considerable time due to large number of edges, the practical time taken by Kruskal's approach is longer than that of other two approaches. Since the Dijkstra with heap depends on the number of nodes, the multiplication factor of log scale remains the same in sparse as well as in dense and only change occurs in the linear part of time complexity contributing to slightly higher time in dense graph approach and is ideally an expected phenomenon. On the whole, time taken by the Dijkstra's algorithm with heap is smallest in Dense graph followed by the Dijkstra's no heap approach which is followed by Kruskal's algorithm whose logarithmic multiplication factor is larger than the quadratic time factor due to number of edges in the dense network.

6. Conclusion and further developments:

The entire project is implemented using the primitive data types such as arrays and structs. Since the algorithms for maximum bandwidth calculation defined here are robust, there is less scope for improvement. Graph generation algorithms are designed to meet the requirements in the straightforward. The methods to create graphs can be improvised further to reduce the time. Randomization principles used to generate the graphs are taken from the predefined header in programming language. Time in graph creation depends on the randomization techniques implemented. Performance of graph generation algorithms can be further improved by choosing better ways to design the networks.