CSCE 629-601 Analysis of Algorithms

Course Project Report

Moeez Akmal
UIN: **431000893**

I must say that this project has been a great learning experience for me. Not only have I gained confidence in my coding skills but also have been able to link this course to real practical implementation.

Following are the details and analysis of how I implemented this Project…

Note: V is total number of vertices in my implementation (i.e. 5000).

1. Random Graph Generation

For this part, I used the `std::uniform_int_distribution` library for pseudo random generation of graphs.

This library produces random integer value *i* values uniformly distributed on the closed interval [a,b], that is, distributed according to the discrete probability function

$$P(i|a,b) = \frac{1}{b-a+1}$$

I used this library to get random unique Source and Destination vertices (i.e. random numbers between 1 to 5000 minus 1 to get vertex index) to add edges in between.

⇨ I use Adjacency List to store my Graphs because it has low complexity than Adacency Matrix. Also, it helps to achieve O(mlogn) complexity for Dijkstra's Algo with Heap Structure because to create a heap structure of fringes, it has to access adjacency list.

⇨ Before Generating Random Graph edges, I connect each adjacent vertex to generate a cycle where the last vertex links with the first.

For both types of graphs, that is, G1 with average degree ~6 & G2 with average degree ~20% (1000), I used the following method to randomly distribute degree among vertices but still keep the overall average at 6 and 20% respectively.

- I randomly add edges between vertices and keep track of the number of added edges for each vertex (through `rand_arr` struct object `adj_arr[V]` which updates the edges connected to each vertex)
- Just to be vigilant I keep the limit of added edges to each vertex at 4999 (no vertex reached this limit in any case of my testing)
- Before adding these edges, I check 2 conditions:
  - There's no edge already present between the source and destination vertices (to avoid repetition)
  - The edge is not added to the same vertex (i.e. No self-looping)

- I sum all these adjacency/connected edges for all vertices and keep this Sum divided by Total No. of Vertices (i.e. sum/V) to limit to on average 6 and 1000 respectively so that required average vertex degree is maintained for both graphs.
  (For both the Graphs the Maximum and Minimum Degree of Vertices varies.)
  - For Average Vertex Degree 6, my graph generation algorithm succeeds in adding ~15000 unique edges including initialized 5000 edges while avoiding self-loops and repeated edges.
  - For Average Vertex Degree 1000 my graph generation algorithm succeeds in adding ~2.5 Million unique edges including initialized 5000 edges while avoiding self-loops and repeated edges.

Implementation overview (Please refer to code for details)

```
//Struct for Adjacency List
struct AdjList
{
    int vertex_id; // Source Vertex
    AdjList* next; // next pointer
    int weight; // Weight of the corresponding edge
};



// Class for Pseudo Random Graph Generation
class Graph
{

    Vertices Ver[V]; // Adjacency List of Randomly Generated Graphs
    Heap K; // Heap for Kruskal's Algo

public:

    // Constructor (used for initialization)
    Graph() {…}



    // Function to Generate Pseudo Random Graph of Average Degree 6
    void rand_6() {…}

    // Function to Generate Pseudo Random Graph of Average Degree 20%
    void rand_20() {…}
```

```cpp
};

// Class to assist in Generation of Adjacency List for Random Graph Generation
/* Used Adjacency List because of lower complexity rather than higher complexity of Adjacency Matrix */
class Vertices
{

    AdjList* head; // Adjacency List Head Pointer

public:

    // Constructor (Used for Initialization)
    Vertices() {…}


    // Function to Add Edge in Adjacency List
    bool add_edge(int u, int v, Vertices* Ver, Heap& K) {…}


    // Function to add edges in MST Adjacency List of Kruskal's after Makeset, Find and Union Operations
    bool add_mst_edge(int u, int v, Vertices* mst_Ver, int wt) {…}



    // Function to assist in creation of Adjacency List
    AdjList* create_edge(int v) {…}



    // Function to assist in creation of Adjacency List
    AdjList* create_node(int x) {…}


    // Function to search in Adjacency List
    bool search(int id) {…}

};
```

⇨ I used Array of Linked List to store my Adjacency List. I made the Array size to be 5000, because I have 5000 Vertices. I used Linked List because the number of adjacent vertices may vary for each vertex.

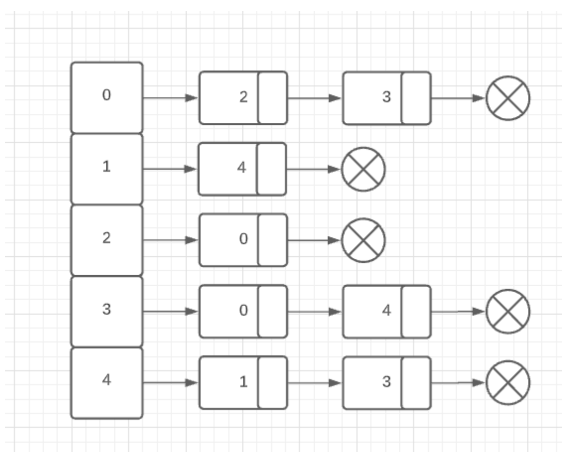


Figure 1: Just to give an idea (not actual case)

⇨ I assign all the edges random weights between [1, 1000000] during graph generation. I have defined the max weight limit via `Max_Weight_Possible` variable.

2. Heap Structure *(Heap class in my code)*

This is by far the hardest part I found. It required the most debugging and understanding. I haven't studied this concept in my undergrad so had to study it for this project.

I generate Max Heap Structures for both Dijkstra's and Kruskal's Algorithms based on the bandwidth or weight of the edge respectively.

For Basic Heap Structure to be used in Dijkstra's Algorithm I have Implemented the following functions:

- Insert()
- Delete()
- Maximum()
- Heapify_Up()
- Heapify_Down()

The `Insert()` functions adds elements to `Heap_Struct` Structure `H[V]`. This structure contains the destination vertex name and associated weight. This is because for Dijkstra's Algorithm's fringes, the source vertex remains the same so no need to keep track of it specifically.
I add every new element at the last index in heap and then I call `Heapify_Up()` function which compares it with its parent and swaps based on larger value (i.e. larger values go up). It keeps on doing this until parent is greater. At max does it for *logn* times.

I use the `Delete()` function to delete any particular element from the heap. To do this, I simply swap the last element in heap with the element that I want to delete and run `Heapify_Up()` function and `Heapify_Down()` function. Because sometimes the parents need adjustment/swapping, other times the children of that swapped node needs adjustment/swapping. `Heapify_Down()` works in exact opposite way of `Heapify_Up()`, here this function compares value of node with left and right child and swap with the largest and keeps on doing till node is greater than children. At max does it for *logn* times. Whenever delete operation is performed, only one possibility will be there i.e. either parent needs swapping or the children. This way, one Heapify_x function will run for at max 1 time and the other for logn times.

To keep track of elements' position in Heap Structure, I used array `P[V];` It helps in Deletion and Heapifying operations. I found maintaining this array the most tedious task but was finally able to use it right.

The `Maximum()` function simply returns the maximum value that is at `H[0]`. The top-most element of heap.

I use `Parent()`, `Left()` and `Right()` functions to simply return Parent, Left and Right child in heap structure respectively of a particular node reference.

I planned to use the same structure for Kruskal's Algorithm and same functions as well for sorting edges via Heap-Sort. But I realized that edges can have same weight plus one of the two i.e. source or destination can be the same as well, so one reference (i.e. only destination vertex) wasn't enough as with Dijkstra's Algo because source remained same for fringes in Dijkstra's.

Moreover, particularly for `Insert()` function, since I was Heapifying as I add edges (which works perfect for Dijkstra's as I only call it when I'm inside Dijkstra's Algo), this didn't work for Kruskal's because I was adding all the edges in Heap Structure during Graph generation for Kruskal's and I was getting already heapified structure before I could run Heap-Sort thus ignoring time of initial sort which is clearly not desired for fair analysis.

That's why I created separate functions for Kruskal's Algorithm which have exactly the same functionality but have slight difference. That are:

For `Insert_Kruskal()` I only create an unsorted array in the start and do not heapify. Once I'm in Kruskal's Algo and I call `Heap_Sort()`, I basically first build the Heap using `Build_Heap_Kruskal()` function which starts from the last parent (i.e. non-leaf) node in heap and calls `Heapify_Down_Kruskal()` for it. Then it gradually moves back through all parents calling `Heapify_Down_Kruskal()` for all. This activity of building heap takes nlogn time at max i.e. n parents and logn time for each `Heapify_Down_Kruskal()`.

`Heapify_Up_Kruskal()`, `Heapify_Down_Kruskal()` and `Delete_Kruskal()` are exactly the same as for normal Heap Implementation except one difference; I use `P_Kruskal[Source][Destination]` array to keep track of elements' position in Heap Structure for Kruskal's Algo; because as explained above, one reference was not enough. This only increases space complexity, time complexity still remains O(1) because we know source and destination.

I use `Kruskal_Struct` Structure for Kruskal's Algo which contains source vertex, destination vertex and edge weight.

The `Max_Heap_Kruskal()` function (which is alternative to Maximum()) for Kruskal's is also different because not only it returns the top-most max value but also deletes that from the structure by calling delete function.

<u>Implementation overview (Please refer to code for details)</u>

```
// Struct for Kruskal's Implementation
struct Kruskal_Struct
{
    int dest_vertex; // Destination vertex
    int edge_weight; // Weight of the corresponding edge
    int src_vertex; // Source Vertex
};

// Struct for Heap for Dijkstra's Implementation
struct Heap_Struct
{
    int vertex_name; // Destination vertex
    int vertex_weight; // Weight of the corresponding edge to above vertex
};



// Class to manage Heap Structure for Dijkstra's and Kruskal's Algos
class Heap
{

private:

    Heap_Struct H[V]; // Heap Structure for Dijkstra's
    Kruskal_Struct* H_Kruskal = new Kruskal_Struct[Limit_20]; // Heap Structure for Kruskal's
    int heap_size; // variable to keep track of heap size

    int P[V]; // Array to keep track of elements' position in Heap Structure for Dijkstra's
              // (Helps in Deletion and Heapifying)


    //Array to keep track of elements' position in Heap Structure for Kruskal's
    int P_Kruskal[V][V]; // i.e. P_Kruskal [Source] [Destination]
    // Take O(1) time because we know source and destination

    /* For Dijkstra's, it was simple because source remained the same, but for Kruskal's Algo
       edges can have same weight plus same source or destination, so one reference wasn't enough  */


public:

    // Constructor (Used for initializing above elements)
    Heap() {…}


    // For Inserting Elements (with proper references) in Heap for Dijkstra's Algo
    void Insert(AdjList* ptr, int bw) {…}

    // For Inserting Edges in Heap (with proper references) for Kruskal's Algo (Just makes a simple array
   for Build_Heap to use)
    void Insert_Kruskal(AdjList* ptr, int bw, int u) {…}



    // Builds Heap from above formed array of edges in Kruskal's Algo
```

```
    void Build_Heap_Kruskal(int length) {…}



    // Swaps last element with selected element and Heapifies (for Dijkstra's Algo)
    void Delete(int index) {…}


    // Swaps last element with selected element and Heapifies (for Kruskal's Algo)
    void Delete_Kruskal(int source, int destination) {…}


    // Returns Maximum element from Heap for Dijkstra's Algo
    Heap_Struct Maximum() {…}




    // Heapifies bottom up for Dijkstra's Algo from passed vertex reference in Heap
    void Heapify_Up(int heap_node) {…}


    // Heapifies bottom up for Kruskal's Algo from passed vertex reference in Heap
    void Heapify_Up_Kruskal(int heap_node) {…}


    // Heapifies top down for Dijkstra's Algo from passed vertex reference in Heap
    void Heapify_Down(int heap_node) {…}



    // Heapifies top down for Kruskal's Algo from passed vertex reference in Heap
    void Heapify_Down_Kruskal(int heap_node) {…}



    // Returns 'Parent' position for the particular vertex in Heap
    int Parent(int i) {…}


    // Returns 'Left Child' position for the particular vertex in Heap
    int Left(int i) {…}


    // Returns 'Right Child' position for the particular vertex in Heap
    int Right(int i) {…}



    // Returns current Heap Size
    int Curr_Heap_Size() {…}



    // Returns Maximum element from Heap for Kruskal's Algo and Deletes it which Heapifies again
    Kruskal_Struct Max_Heap_Kruskal() {…}

};
```

3. Routing Algorithms

I implemented the 3 required algorithms exactly as taught in Class.

1) For modified **Dijkstra's Algorithm without Heap** for Max-BW-Path, I use my adjacency list to span the connected nodes and change their status to Fringe in Status Array. Then I extract Max BW Fringe based on simple comparisons of all elements of array (with status == fringe).

   The Adjacency List is simply array of Linked Lists as shown and explained on Page#3. The Status Array is of size V which is the total number of vertices.

   Afterwards, I span the connected vertices to Max BW Fringe using the Adjacency List of this identified vertex and update BW as per Algorithm discussed in class and follow the Algorithm of class for rest of the steps.

   Therefore, Time Complexity remains bounded by $O(n^2)$.

Implementation overview [major portions only, some portions of code here are shortened to avoid verbosity] (Please refer to code for details)

```
    // Implemented Dijkstra's Algo taught in Class without Heap Structure
  static void Dijkstra(int s, int t, Vertices* Ver)
  {
      /*
          Status:

          unseen -> 0;
          fringe -> 1;
          in_tree -> 2;

      */


      int status[V];

      int dad[V];
      int bw[V];

      // Initialize
      for (int v = 0; v < V; v++)
      {
          status[v] = 0; //unseen

          dad[v] = -1;
          bw[v] = 0;
      }

      status[s] = 2; //in-tree


      AdjList* ptr = Ver[s].head;
```

```
    while (ptr != NULL)
    {
        status[ptr->vertex_id] = 1; //fringe

        dad[ptr->vertex_id] = s;
        bw[ptr->vertex_id] = ptr->weight;

        ptr = ptr->next;
    }

    bool fringe_exists = false; // To check if fringes exist
    int max_bw_fringe = -1; // To get Max BW Fringe
    int temp_bw = 0;

    for (int v = 0; v < V; v++)
    {
        if (status[v] == 1) //fringe
        {
            if (bw[v] > temp_bw)
            {
                max_bw_fringe = v;
                temp_bw = bw[v];
            }
            fringe_exists = true;
        }
    }

    while (fringe_exists)
    {
        status[max_bw_fringe] = 2; //in-tree

        AdjList* ptr = Ver[max_bw_fringe].head;

        while (ptr != NULL)
        {
            if (status[ptr->vertex_id] == 0) // if unseen
            {
                status[ptr->vertex_id] = 1; //fringe

                dad[ptr->vertex_id] = max_bw_fringe;
                bw[ptr->vertex_id] = std::min(bw[max_bw_fringe], ptr->weight);
            }

            // if fringe && …
            else if (status[ptr->vertex_id] == 1 && (bw[ptr->vertex_id] < (std::min(bw[max_bw_fringe],
ptr->weight))))
            {
                bw[ptr->vertex_id] = (std::min(bw[max_bw_fringe], ptr->weight));
                dad[ptr->vertex_id] = max_bw_fringe;
            }

            ptr = ptr->next;
        }

        fringe_exists = false;
        max_bw_fringe = -1;
        temp_bw = 0;

        for (int v = 0; v < V; v++)
        {
            if (status[v] == 1) // fringe
            {
                if (bw[v] > temp_bw)
                {
                    max_bw_fringe = v;
                    temp_bw = bw[v];
                }
                fringe_exists = true;
```

Using Adjacency List of s to initialize Array of Fringes by changing Status, O(n) at max

Using Initialized Array to find Max BW Fringe O(n)

<= 2m for entire execution of the code

Using Updated Array to find Max BW Fringe, O(n)

```
            }
        }

    }


    if (status[t] != 2) // if not in-tree
            no s-t path (refer to code for actual details)



    else
           Print Path using dad array! (refer to code for actual details)

}
```

2) For modified **Dijkstra's Algorithm with Heap Structure for fringes**, I used the `Insert()`, `Delete()` and `Maximum()` functions implemented in my Heap Class, to insert, delete and find maximum bw fringe in Heap Structure for fringes.

   I used exactly the same Algorithm as studied in class and replaced `Insert()`, `Delete()` and `Maximum()` functions to functions of my Heap Class.

   To get elements for the heap, I use my adjacency list to span the connected nodes to a vertex (at max O(n)) which makes sure that complexity doesn't exceed O(mlogn).

   The size of my heap array (`i.e. H[V]`) in Heap Class is `V` which is the total number of vertices but most of it remains empty here (Ref to Heap Structure for details Page#4)

   Therefore, Time Complexity remains bounded by O(mlogn).

Implementation overview [major portions only, some portions of code here are shortened to avoid verbosity] (Please refer to code for details)

```
  // Implemented Dijkstra's Algo taught in Class with Heap Structure
static void Dijkstra_Heapify(int s, int t, Vertices* Ver)
{
    /*
        Status:

        unseen -> 0;
        fringe -> 1;
        in_tree -> 2;

    */
```

```
    int status[V];

    int dad[V];
    int bw[V];

    // Initialize
    for (int v = 0; v < V; v++)
    {
        status[v] = 0; //unseen

        dad[v] = -1;
        bw[v] = 0;
    }

    status[s] = 2; //in-tree
    Heap F; //Heap Structure

    AdjList* ptr = Ver[s].head;


    while (ptr != NULL)
    {
        status[ptr->vertex_id] = 1; //fringe

        dad[ptr->vertex_id] = s;
        bw[ptr->vertex_id] = ptr->weight;

        //Insert() function from Heap Class
        F.Insert(ptr, bw[ptr->vertex_id]);

        ptr = ptr->next;
    }
```

Using Adjacency List of s to initialize Heap Structure by using Insert() Operation, O(n) at max

```
    int max_bw_fringe = -1;

    while (F.Curr_Heap_Size() >= 0)
    {

        max_bw_fringe = F.Maximum().vertex_name; //Maximum() function from Heap Class

        status[max_bw_fringe] = 2; //in-tree

        F.Delete(max_bw_fringe); //Delete() function from Heap Class

        AdjList* ptr = Ver[max_bw_fringe].head;

        while (ptr != NULL)
        {
            if (status[ptr->vertex_id] == 0) // if unseen
            {
                status[ptr->vertex_id] = 1; //fringe

                dad[ptr->vertex_id] = max_bw_fringe;
                bw[ptr->vertex_id] = std::min(bw[max_bw_fringe], ptr->weight);

                F.Insert(ptr, bw[ptr->vertex_id]);
            }

            // if fringe && ...
            else if (status[ptr->vertex_id] == 1 && (bw[ptr->vertex_id] < (std::min(bw[max_bw_fringe],
ptr->weight))))
            {
                F.Delete(ptr->vertex_id); //Delete() function from Heap Class

                bw[ptr->vertex_id] = (std::min(bw[max_bw_fringe], ptr->weight));
                F.Insert(ptr, bw[ptr->vertex_id]); //Insert() function from Heap Class
```

```
                    dad[ptr->vertex_id] = max_bw_fringe;

                }

                ptr = ptr->next;
            }


            max_bw_fringe = -1;

        }


        if (status[t] != 2) // if not in-tree
            no s-t path (refer to code for actual details)


        else
           Print Path using dad array! (refer to code for actual details)
    }
```

3) For modified **Kruskal's Algorithm**, I used Heap-Sort to sort edges by weight in decreasing order and used MakeSet, Find with Compression, and Union operations exactly as taught in class to construct a Max Spanning Tree (MST) in form of Adjacency List. Also, I used Breadth First Search (BFS), as taught in class, on generated Max Spanning Tree to find Max BW Path.

   For storing the heap of edges, I used an array of the size of number of unique edges produced in graph generation.

   For HeapSort, I call the functions `Build_Heap_Kruskal()` to build the Max Heap first and then subsequently call `Max_Heap_Kruskal()` function to return the top-most max value and delete it from the structure by calling delete function. [explanation of these functions is given in Heap Structure - Page# 5,6]

   The Heap-Sort takes time O(mlogm) and MakeSet-Find_Compression-Union take O(mlog*n). The BFS on Max Spanning Tree takes O(n) time.

   Therefore, Time Complexity remains bounded by O(mlogm).

__Implementation overview [major portions only, some portions of code here are shortened to avoid verbosity] (Please refer to code for details)__

```
// Implemented Kruskal's Algo taught in Class with Heap-Sort
    static void Kruskal(Heap& K, int no_of_edges, int node_s, int node_t)
    {
        int dad[V];
        int rank[V];
```

```
        int u, v;
        int r1, r2;
        Kruskal_Struct* Sorted_Array = new Kruskal_Struct[no_of_edges]; // Array Reference after Heap-Sort
        Vertices Max_Spanning_Tree[V]; // Adjacency List to store Max Spanning Tree after Makeset, Find
   Compression and Union Operations


        Sorted_Array = Heap_Sort(K, no_of_edges); // Edges Sorted by Weight in decreasing order


        for (int v = 0; v < V; v++)
        {
            MakeSet(v, dad, rank); //calling MakeSet
        }

        for (int k = 0; k < no_of_edges; k++)
        {
            u = Sorted_Array[k].src_vertex;
            v = Sorted_Array[k].dest_vertex;


            r1 = Find(u, dad); r2 = Find(v, dad); //calling Find with Compression

            if (r1 != r2)
            {
                Max_Spanning_Tree[u].add_mst_edge(u, v, Max_Spanning_Tree, Sorted_Array[k].edge_weight); //
   Adding to MST Adjacency List
                Union(r1, r2, dad, rank); //calling Union
            }
        }


        // Run BFS on Max Spanning Tree to print Path
        Vertices::BFS_path(node_s, node_t, Max_Spanning_Tree);
    }



// Heap_Sort Function
    static Kruskal_Struct* Heap_Sort(Heap& K, int no_of_edges)
    {
        int sorted_index = 0;
        Kruskal_Struct* Sorted = new Kruskal_Struct[no_of_edges];

        K.Build_Heap_Kruskal(no_of_edges); // Build Heap for the First Time only

        while (K.Curr_Heap_Size() >= 0)
        {
            // Extract Max Element from Heap
            Sorted[sorted_index] = K.Max_Heap_Kruskal();
            sorted_index++;
        }

        return Sorted;
    }
```

How did I verify that my Algorithms run correctly?

So, after writing the algorithms, I tested them on examples (with less number of vertices) whose solution I knew. I made sure that I test all corner cases. Once, my algos passed that test, I introduced them to random sparse graphs produced by Step-1 with lower number of vertices. In a few scenarios, I found corner cases that I didn't address. Once I addressed those cases I checked and verified the algorithms on that sparse graph with pen and paper for correctness. Afterwards, I proceeded to run my algorithms on graphs with 5000 vertices generated in Step-1. Hence, this is how I'm confident that all of my 3 algorithms work correctly.

4. Testing

I obtained the following results for these routing Algorithms on Graph G1 (Sparse) and Graph G2 (Dense)…

(**Note:** Weight of Edges are randomly assigned between [1, 1000000], ref Page#4 1st line)
(**Also Please Note:** Source (s) and Destination (t) vertices were randomly generated for tests)

⇨ For recording time, I used the library `std::chrono::high_resolution_clock` which represents the clock with the smallest tick period.

Note: Since for Sparse Graphs, the time taken by BFS in Max Spanning Tree for finding path in Kruskal's Algorithm becomes more critical and significant relative to time taken by Kruskal's Algorithm itself. Therefore, for fair comparison and my own analysis in Sparse Graphs, I have taken both times with and without BFS on Max Spanning Tree. The results are quite interesting as you'll see.

Sparse Graph G1 (1st Run)

```
** Pseudo Random Graph with Average Degree ~6 **

Total Number of Edges: 15059
Average Degree: 6.0236
Max Degree: 15
Min Degree: 2
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	375	4881	669547	0.144741
2	4610	4024	770334	0.153559
3	3570	161	623084	0.175438
4	1181	1512	615655	0.152308
5	3229	1	747023	0.159815

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	375	4881	669547	0.109933
2	4610	4024	770334	0.0980147
3	3570	161	623084	0.0806836
4	1181	1512	615655	0.0400136
5	3229	1	747023	0.0743207

3. Kruskal's Algo (with HeapSort)

Sr#	Source	Destination	Max BW	Time with BFS (sec)	BFS on MST (sec)	Time w/o BFS (sec)
1	375	4881	669547	0.0741135	0.0295869	0.0445266
2	4610	4024	770334	0.0551037	0.0229602	0.0321435
3	3570	161	623084	0.093865	0.0553934	0.0384716
4	1181	1512	615655	0.0361198	0.0062297	0.0298901
5	3229	1	747023	0.0639999	0.0357543	0.0282456

Sparse Graph G1 (2nd Run)

```
** Pseudo Random Graph with Average Degree ~6 **

Total Number of Edges: 15048
Average Degree: 6.0192
Max Degree: 14
Min Degree: 2
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	1913	627	487647	0.227164
2	2707	2242	761338	0.297713
3	1841	3899	789991	0.253011
4	1889	2930	810736	1.30499
5	2891	96	671125	0.204161

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	1913	627	487647	0.154297
2	2707	2242	761338	0.147433
3	1841	3899	789991	0.0724469
4	1889	2930	810736	0.0484623
5	2891	96	671125	0.0640294

3. Kruskal's Algo (with HeapSort)

Sr#	Source	Destination	Max BW	Time with BFS (sec)	BFS on MST (sec)	Time w/o BFS (sec)
1	1913	627	487647	0.133818	0.0625458	0.071272
2	2707	2242	761338	0.133303	0.0689851	0.064318
3	1841	3899	789991	0.104966	0.0452349	0.059731
4	1889	2930	810736	0.0520479	0.0203616	0.031686
5	2891	96	671125	0.0657046	0.0268625	0.038842

Sparse Graph G1 (3rd Run)

```
** Pseudo Random Graph with Average Degree ~6 **

Total Number of Edges: 15052
Average Degree: 6.0208
Max Degree: 17
Min Degree: 2
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	2902	4530	668944	0.35217
2	4545	715	630153	0.231113
3	1484	1756	790378	0.335199
4	1957	3649	472067	0.141768
5	4428	2093	775526	0.219923

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	2902	4530	668944	0.142018
2	4545	715	630153	0.0991509
3	1484	1756	790378	0.0856951
4	1957	3649	472067	0.0612219
5	4428	2093	775526	0.081249

3. Kruskal's Algo (with HeapSort)

Sr#	Source	Destination	Max BW	Time with BFS (sec)	BFS on MST (sec)	Time w/o BFS (sec)
1	2902	4530	668944	0.112567	0.0386734	0.0738936
2	4545	715	630153	0.151011	0.0932251	0.0577859
3	1484	1756	790378	0.106213	0.0515919	0.0546211
4	1957	3649	472067	0.0654651	0.0242545	0.0412106
5	4428	2093	775526	0.0915769	0.0507892	0.0407877

Sparse Graph G1 (4th Run)

```
** Pseudo Random Graph with Average Degree ~6 **

Total Number of Edges: 15053
Average Degree: 6.0212
Max Degree: 17
Min Degree: 2
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	615	1804	603472	0.121441
2	4929	2722	547399	0.136979
3	4033	4622	621598	0.211483
4	394	2323	438807	0.148875
5	80	2225	723564	0.148166

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	615	1804	603472	0.0449674
2	4929	2722	547399	0.064466
3	4033	4622	621598	0.123534
4	394	2323	438807	0.0517771
5	80	2225	723564	0.057416

3. Kruskal's Algo (with HeapSort)

Sr#	Source	Destination	Max BW	Time with BFS (sec)	BFS on MST (sec)	Time w/o BFS (sec)
1	615	1804	603472	0.0875446	0.0515557	0.0359889
2	4929	2722	547399	0.0433398	0.0105721	0.0327677
3	4033	4622	621598	0.0908759	0.0207485	0.0701274
4	394	2323	438807	0.059739	0.0300652	0.0296738
5	80	2225	723564	0.0749975	0.0339551	0.0410424

Sparse Graph G1 (5th Run)

```
** Pseudo Random Graph with Average Degree ~6 **

Total Number of Edges: 15053
Average Degree: 6.0212
Max Degree: 14
Min Degree: 2
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	134	3324	720016	0.314288
2	513	4710	786961	0.475061
3	359	1100	646323	0.331657
4	1840	397	686175	0.216104
5	4623	3049	672504	0.178822

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	134	3324	720016	0.202213
2	513	4710	786961	0.160248
3	359	1100	646323	0.0665612
4	1840	397	686175	0.0533673
5	4623	3049	672504	0.0560178

3. Kruskal's Algo (with HeapSort)

Sr#	Source	Destination	Max BW	Time with BFS (sec)	BFS on MST (sec)	Time w/o BFS (sec)
1	134	3324	720016	0.200211	0.0497133	0.1504977
2	513	4710	786961	0.200117	0.128593	0.071524
3	359	1100	646323	0.153216	0.0980583	0.0551577
4	1840	397	686175	0.0927265	0.0499101	0.0428164
5	4623	3049	672504	0.0604884	0.0208289	0.0396595

Dense Graph G2 (1st Run)

```
** Pseudo Random Graph with Average Degree ~20% **

Total Number of Edges: 2507905
Average Degree: 1003.16
Max Degree: 1101
Min Degree: 903
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	2240	2310	997188	1.04478
2	805	3635	998864	1.09708
3	1034	2699	998628	1.04676
4	3005	655	997220	1.01912
5	714	2695	995888	1.02523

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	2240	2310	997188	0.950766
2	805	3635	998864	1.0046
3	1034	2699	998628	0.951431
4	3005	655	997220	0.883845
5	714	2695	995888	0.891629

Note: *(Computer Architecture level explanation)* For 5 different Source and Destination vertices, after initial run i.e. for one source/destination pair, Visual Studio optimizes time of `Max_Heap_Kruskal()` function (i.e. Delete and Heapify for Kruskal) for dense graphs because the recently generated graph is in cache of processor rather than main memory and most probably branch prediction is now being used, so for fair comparison, first observation is ignored and 5 are produced afterwards.

3. Kruskal's Algo (with HeapSort and BFS)

Sr#	Source	Destination	Max BW	Time (sec)
1	2240	2310	997188	7.62686
2	805	3635	998864	7.75602
3	1034	2699	998628	7.63357
4	3005	655	997220	7.7604
5	714	2695	995888	8.235

<u>Dense Graph G2 (2nd Run)</u>

```
** Pseudo Random Graph with Average Degree ~20% **

Total Number of Edges: 2507786
Average Degree: 1003.11
Max Degree: 1125
Min Degree: 887
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	2999	4474	998259	0.811452
2	3003	4926	998695	0.802576
3	235	1425	998249	0.791223
4	1659	266	997186	0.808049
5	440	3027	998534	0.779577

2. Dijkstra’s Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	2999	4474	998259	0.700485
2	3003	4926	998695	0.688671
3	235	1425	998249	0.750071
4	1659	266	997186	0.700583
5	440	3027	998534	0.694868

3. Kruskal’s Algo (with HeapSort and BFS)

Sr#	Source	Destination	Max BW	Time (sec)
1	2999	4474	998259	6.27127
2	3003	4926	998695	6.2348
3	235	1425	998249	6.16195
4	1659	266	997186	6.22179
5	440	3027	998534	6.12716

Dense Graph G2 (3^{rd} Run)

```
** Pseudo Random Graph with Average Degree ~20% **

Total Number of Edges: 2508197
Average Degree: 1003.28
Max Degree: 1108
Min Degree: 896
```

1. Dijkstra’s Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	3267	4482	997874	1.03442
2	3662	752	996831	0.777755
3	2546	1439	998003	0.759152
4	2686	3213	997346	0.736538
5	3504	3584	998540	0.781658

2. Dijkstra’s Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	3267	4482	997874	0.791913
2	3662	752	996831	0.692809
3	2546	1439	998003	0.678489
4	2686	3213	997346	0.65108
5	3504	3584	998540	0.703398

3. Kruskal’s Algo (with HeapSort and BFS)

Sr#	Source	Destination	Max BW	Time (sec)
1	3267	4482	997874	6.57221
2	3662	752	996831	6.22258
3	2546	1439	998003	6.1206
4	2686	3213	997346	6.15561
5	3504	3584	998540	6.15209

Dense Graph G2 (4th Run)

```
** Pseudo Random Graph with Average Degree ~20% **

Total Number of Edges: 2507987
Average Degree: 1003.19
Max Degree: 1105
Min Degree: 896
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	1162	994	997634	0.788312
2	4732	4346	997632	0.781481
3	803	2449	998375	0.898776
4	368	2030	998610	0.816905
5	1381	1715	995849	0.785844

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	1162	994	997634	0.677225
2	4732	4346	997632	0.704689
3	803	2449	998375	0.696413
4	368	2030	998610	0.7027
5	1381	1715	995849	0.731043

3. Kruskal's Algo (with HeapSort and BFS)

Sr#	Source	Destination	Max BW	Time (sec)
1	1162	994	997634	6.23354
2	4732	4346	997632	6.14808
3	803	2449	998375	6.21117
4	368	2030	998610	6.392
5	1381	1715	995849	6.2159

Dense Graph G2 (5th Run)

```
** Pseudo Random Graph with Average Degree ~20% **

Total Number of Edges: 2508185
Average Degree: 1003.27
Max Degree: 1110
Min Degree: 905
```

1. Dijkstra's Algo (Normal)

Sr#	Source	Destination	Max BW	Time (sec)
1	1703	4522	998133	0.792178
2	4341	2268	997269	0.767489
3	212	965	998465	0.763708
4	961	952	998785	0.804644
5	3068	689	998102	0.78676

2. Dijkstra's Algo (with Heap)

Sr#	Source	Destination	Max BW	Time (sec)
1	1703	4522	998133	0.681363
2	4341	2268	997269	0.71763
3	212	965	998465	0.665812
4	961	952	998785	0.717091
5	3068	689	998102	0.709941

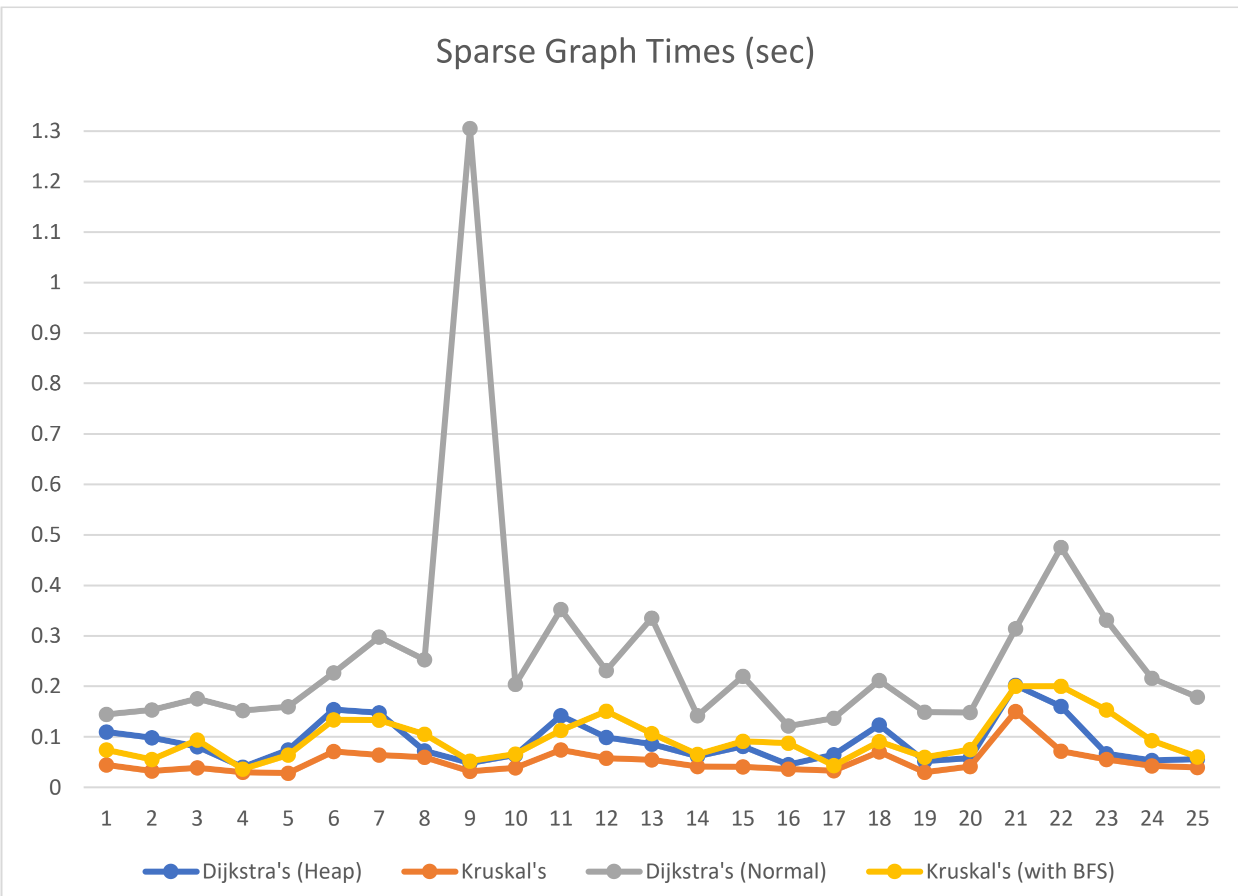
3. Kruskal's Algo (with HeapSort and BFS)

Sr#	Source	Destination	Max BW	Time (sec)
1	1703	4522	998133	6.19402
2	4341	2268	997269	6.29791
3	212	965	998465	6.15481
4	961	952	998785	6.38875
5	3068	689	998102	6.38039

Average Times

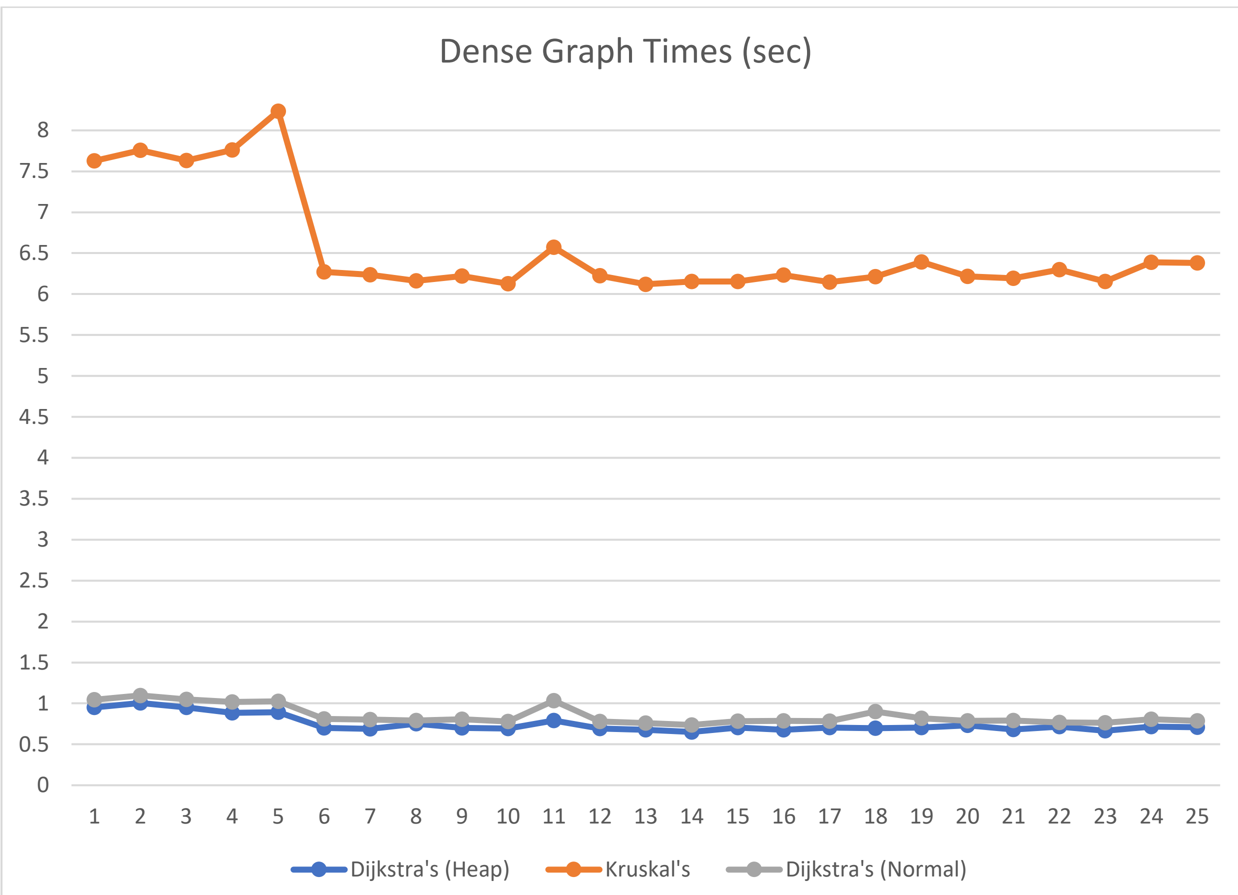
For Sparse Graphs G1 (with average degree 6)

Sr#	Algorithm	Type of Graph	Average Time (sec)
1	Dijkstra's (Normal)	Sparse	0.26543796
2	Dijkstra's (with Heap)	Sparse	0.089581476
3.1	Kruskal's (with HeapSort)	Sparse	0.051067232
3.2	Kruskal's (with HeapSort + BFS)	Sparse	0.096125204



For Dense Graphs G2 (with average degree 20%)

Sr#	Algorithm	Type of Graph	Average Time (sec)
1	Dijkstra's (Normal)	Dense	0.85205868
2	Dijkstra's (with Heap)	Dense	0.7495418
3	Kruskal's (with HeapSort + BFS)	Dense	6.5547392



Analyses

As you can see from above times for **Sparse Graphs,** the following is the order of time taken by Algorithms I have implemented:

Kruskal’s Algo (HeapSort) [Fastest] **<** Dijkstra’s Algo (Heap) **<** Dijkstra’s Algo (Normal) [Slowest]

I have observed that in cases where the Max BW Path (s-t) is in general longer for Kruskal’s Algorithm, the BFS on Max Spanning Tree eats up a lot of time and is a significant contributor to the time for Kruskal’s Algorithm. In such case, if I consider BFS to extract path, the Dijkstra’s Algo (with Heap) is the fastest, followed by Kruskal’s Algo and then Normal Dijkstra’s Algo. I have found the Kruskal’s Algorithm (without considering BFS on MST to find Path) to be always the fastest in case of Sparse Graphs. In the alternate case where we consider BFS on MST time…

Dijkstra’s Algo (Heap) [Fastest] **<** Kruskal’s Algo (HeapSort+BFS) **<** Dijkstra’s Algo (Normal) [Slowest]

For the **Dense Graphs,** the following is the order of time taken by Algorithms I have implemented:

Dijkstra’s Algo (Heap) [Fastest] **<** Dijkstra’s Algo (Normal) **<** Kruskal’s Algo (HeapSort) [Slowest]

Discussion

It is evident that for both types of Graphs (Dense and Sparse), the Dijkstra’s Algorithm (with Heap) is performing better than Normal Dijkstra’s Algorithm. This is because the Insert and Delete operations in Heap take O(logn) time and Finding Maximum element takes O(1) time. Whereas for my Normal Dijkstra’s Algorithm, it has to search/traverse and compare through whole of the Status Array every time to find the Maximum BW Fringe which takes V i.e. O(n) time. Hence the results make sense.

The Kruskal’s Algorithm (with Heapsort) is however at the opposite spectrum for the two types of Graphs. For Sparse Graphs, it is the fastest, however, for the Dense Graphs it is the slowest. For Dense

Graphs, it makes common sense that there are a lot of edges to be sorted (by BW) for Heap-Sort and thus a lot of Heapifying operations are involved every time and hence Kruskal's Algo has to take a lot of time.

Theoretically speaking, since my implementation of Kruskal's Algo is bounded by O(mlogm) which is better than Normal Dijkstra's Algo which takes $O(n^2)$, it was kind of a shock for me to discover that complexity not necessarily governs running times in practical applications because of such a large gap. 'm' can of course be equal to n^2 in worst case.

So, I believe this is the case because while we discuss worst case analysis in class, the Algorithms are not necessarily pushed to their worst case in all the implementations and analyses.

For Sparse Graphs, however, I believe that since there are much smaller number of edges 'm' to sort (which I have stored in an Array), so the time required for Kruskal's Algo significantly decreases, since heapify's 'logm' time is also a much smaller number now. Hence it becomes the fastest in all cases if I don't consider BFS on Max Spanning Tree (I have explained the reason above in Analyses, Page#28). Also, since you can add all elements at once in the array rather than waiting for the specific source node 's' input to decide on where to begin, it could also be a factor (obviously in case neglecting BFS). Normal Dijkstra's Algo without heap is regardless the slowest (even if I consider BFS on MST), due to adjacency list usage and for traversing whole Status Array for Max BW Fringe finding.

Dijkstra's Algo (with Heap) may be slower than Kruskal's because even to form the Heap of Fringes through the `Insert()` operation for finding Max BW Fringe eventually, it has to access the Adjacency List of that particular in-tree node to get the connected nodes which practically obviously takes time (can be of the order O(n)).

In short, if I consider the BFS on MST for finding path, it on average makes Kruskal's Algo slightly slower than Dijkstra's (with Heap) but still faster than Normal Dijkstra's Algo without heap. If the s-t path of Kruskal's Algo is comparatively shorter, the Kruskal's Algo still remains the fastest regardless of BFS on MST time consideration, according to my implementation.

Also, it is obvious that corresponding Algorithms on Dense Graphs will generally take longer time than in case of Sparse Graphs and it is evident from the results as well.

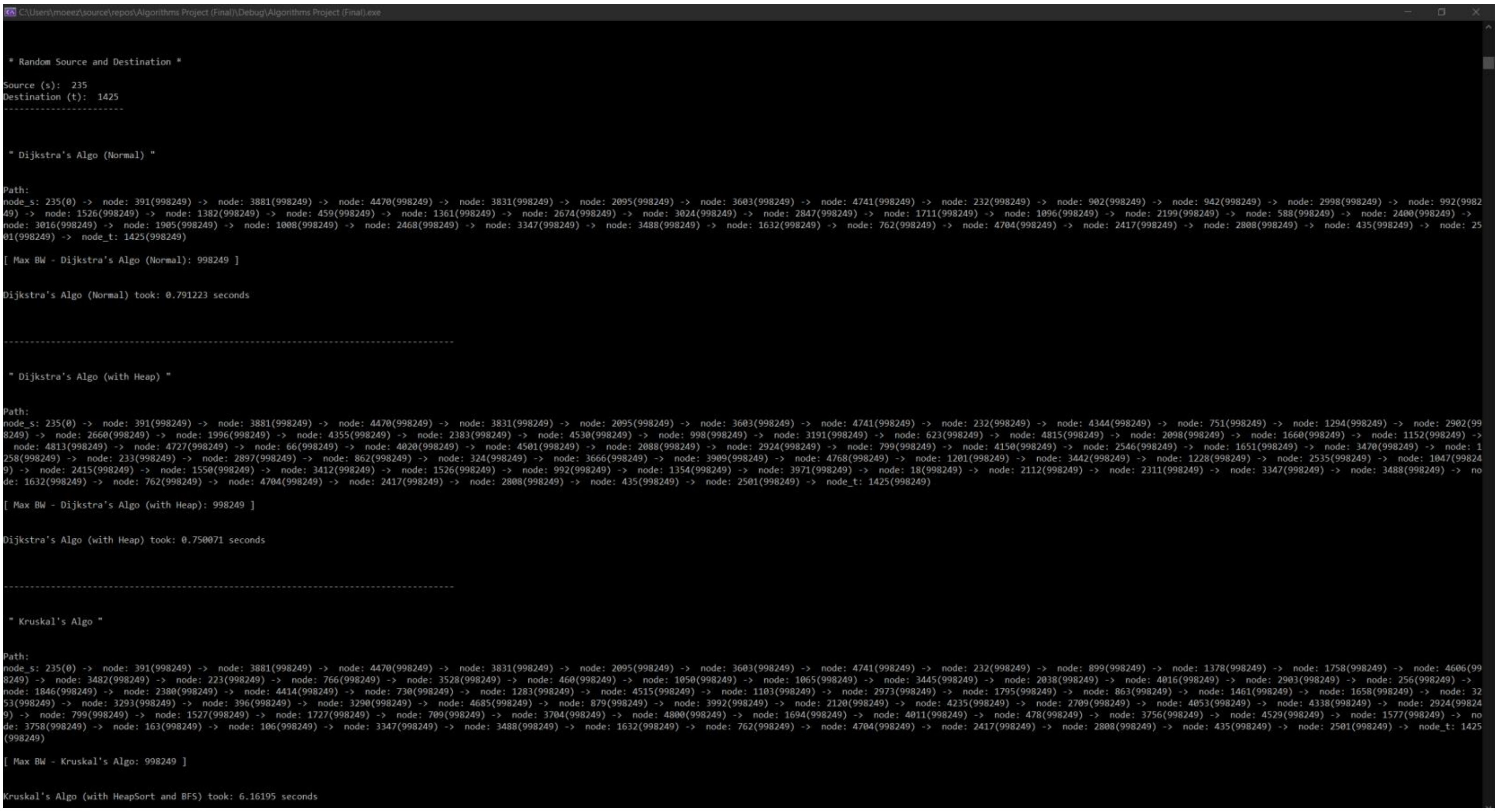
It is, therefore, safe to assume based on my test cases that Dijkstra's Algo (with Heap) offers win-win situation for both Sparse and Dense Graphs' Max BW Path generation.

Possible Further Improvements

- I have used a lot of arrays and the Heap Array of Kruskal's Algo becomes very large giving heap overflow error in my compiler unless I increase allocated heap size. I should manage the space complexity better as well next time.
- Also, I have implemented BFS exactly as Prof. Chen taught in class, but it still takes a lot of time, I could consider using DFS or other method for my Kruskal's Algo Path in the future.
- I can use better sorting Algorithms than Heap Sort (e.g. Merge Sort) for Dense Graphs in Kruskal's Algorithm because Heap Sort is taking the most time in case of Dense Graphs.
- I have realized that while generating graphs, if I keep track of max element in my adjacency list for each vertex, I can reduce the time for finding Max Fringe to O(1) in my Dijkstra's Algo without heap.

How Output is Displayed in Console Window

Figure 2: Sparse Graph



```
C:\Users\moeez\source\repos\Algorithms Project (Final)\Debug\Algorithms Project (Final).exe

 * Random Source and Destination *

Source (s):  235
Destination (t):  1425
-----------------------


 " Dijkstra's Algo (Normal) "


Path:
node_s: 235(0) ->  node: 391(998249) ->  node: 3881(998249) ->  node: 4470(998249) ->  node: 3831(998249) ->  node: 2095(998249) ->  node: 3603(998249) ->  node: 4741(998249) ->  node: 232(998249) ->  node: 902(998249) ->  node: 942(998249) ->  node: 2998(998249) ->  node: 992(9982
49) ->  node: 1526(998249) ->  node: 1382(998249) ->  node: 459(998249) ->  node: 1361(998249) ->  node: 2674(998249) ->  node: 3024(998249) ->  node: 2847(998249) ->  node: 1711(998249) ->  node: 1096(998249) ->  node: 2199(998249) ->  node: 588(998249) ->  node: 2400(998249) ->
node: 3016(998249) ->  node: 1905(998249) ->  node: 1008(998249) ->  node: 2468(998249) ->  node: 3347(998249) ->  node: 3488(998249) ->  node: 1632(998249) ->  node: 762(998249) ->  node: 4704(998249) ->  node: 2417(998249) ->  node: 2808(998249) ->  node: 435(998249) ->  node: 25
01(998249) ->  node_t: 1425(998249)

[ Max BW - Dijkstra's Algo (Normal): 998249 ]


Dijkstra's Algo (Normal) took: 0.791223 seconds



---------------------------------------------------------------------------------


 " Dijkstra's Algo (with Heap) "


Path:
node_s: 235(0) ->  node: 391(998249) ->  node: 3881(998249) ->  node: 4470(998249) ->  node: 3831(998249) ->  node: 2095(998249) ->  node: 3603(998249) ->  node: 4741(998249) ->  node: 232(998249) ->  node: 4344(998249) ->  node: 751(998249) ->  node: 1294(998249) ->  node: 2902(99
8249) ->  node: 2660(998249) ->  node: 1996(998249) ->  node: 4355(998249) ->  node: 2383(998249) ->  node: 4530(998249) ->  node: 998(998249) ->  node: 3191(998249) ->  node: 623(998249) ->  node: 4815(998249) ->  node: 2098(998249) ->  node: 1660(998249) ->  node: 1152(998249) ->
 node: 4813(998249) ->  node: 4727(998249) ->  node: 66(998249) ->  node: 4020(998249) ->  node: 4501(998249) ->  node: 2088(998249) ->  node: 2924(998249) ->  node: 799(998249) ->  node: 4150(998249) ->  node: 2546(998249) ->  node: 1651(998249) ->  node: 3470(998249) ->  node: 1
258(998249) ->  node: 233(998249) ->  node: 2897(998249) ->  node: 862(998249) ->  node: 324(998249) ->  node: 3666(998249) ->  node: 3909(998249) ->  node: 4768(998249) ->  node: 1201(998249) ->  node: 3442(998249) ->  node: 1228(998249) ->  node: 2535(998249) ->  node: 1047(99824
9) ->  node: 2415(998249) ->  node: 1550(998249) ->  node: 3412(998249) ->  node: 1526(998249) ->  node: 992(998249) ->  node: 1354(998249) ->  node: 3971(998249) ->  node: 18(998249) ->  node: 2112(998249) ->  node: 2311(998249) ->  node: 3347(998249) ->  node: 3488(998249) ->  no
de: 1632(998249) ->  node: 762(998249) ->  node: 4704(998249) ->  node: 2417(998249) ->  node: 2808(998249) ->  node: 435(998249) ->  node: 2501(998249) ->  node_t: 1425(998249)

[ Max BW - Dijkstra's Algo (with Heap): 998249 ]


Dijkstra's Algo (with Heap) took: 0.750071 seconds



---------------------------------------------------------------------------------


 " Kruskal's Algo "


Path:
node_s: 235(0) ->  node: 391(998249) ->  node: 3881(998249) ->  node: 4470(998249) ->  node: 3831(998249) ->  node: 2095(998249) ->  node: 3603(998249) ->  node: 4741(998249) ->  node: 232(998249) ->  node: 899(998249) ->  node: 1378(998249) ->  node: 1758(998249) ->  node: 4606(99
8249) ->  node: 3482(998249) ->  node: 223(998249) ->  node: 766(998249) ->  node: 3528(998249) ->  node: 460(998249) ->  node: 1050(998249) ->  node: 1065(998249) ->  node: 3445(998249) ->  node: 2038(998249) ->  node: 4016(998249) ->  node: 2903(998249) ->  node: 256(998249) ->
node: 1846(998249) ->  node: 2380(998249) ->  node: 4414(998249) ->  node: 730(998249) ->  node: 1283(998249) ->  node: 4515(998249) ->  node: 1103(998249) ->  node: 2973(998249) ->  node: 1795(998249) ->  node: 863(998249) ->  node: 1461(998249) ->  node: 1658(998249) ->  node: 32
53(998249) ->  node: 3293(998249) ->  node: 396(998249) ->  node: 3290(998249) ->  node: 4685(998249) ->  node: 879(998249) ->  node: 3992(998249) ->  node: 2120(998249) ->  node: 4235(998249) ->  node: 2709(998249) ->  node: 4053(998249) ->  node: 4338(998249) ->  node: 2924(99824
9) ->  node: 799(998249) ->  node: 1527(998249) ->  node: 1727(998249) ->  node: 709(998249) ->  node: 3704(998249) ->  node: 4800(998249) ->  node: 1694(998249) ->  node: 4011(998249) ->  node: 478(998249) ->  node: 3756(998249) ->  node: 4529(998249) ->  node: 1577(998249) ->  no
de: 3758(998249) ->  node: 163(998249) ->  node: 106(998249) ->  node: 3347(998249) ->  node: 3488(998249) ->  node: 1632(998249) ->  node: 762(998249) ->  node: 4704(998249) ->  node: 2417(998249) ->  node: 2808(998249) ->  node: 435(998249) ->  node: 2501(998249) ->  node_t: 1425
(998249)

[ Max BW - Kruskal's Algo: 998249 ]


Kruskal's Algo (with HeapSort and BFS) took: 6.16195 seconds
```

Figure 3: Dense Graph

Note: While dealing with dense graphs, I got stack and heap overflow errors. So, in my Visual Studio, I increased Stack and Heap Size to 512 MB.

If you have any questions regarding this project, please feel free to contact at <u>moeez.akmal@tamu.edu</u>.