CPSE-629 Analysis of Algorithms

Implementation of Network Routing Protocol using Data Structures and Algorithms

(Medium of implementation : JAVA)

Submitted By

NIKHILESH PANDEY
UIN : 224002412

This project is involves simulation of network topologies and study of corresponding improvements in efficiency of network routing protocols using Dijkstra's and Kruskal's algorithms over sparse and dense topologies. Network topologies have been simulated using graph data structures.

1. Random Graph Generation

The graph is implemented using an adjacency list (using ArrayList in java) of nodes. Every node consists of an attribute name which is an integer between 0 to 5000 and a list of edges. The edges have information about the edge weights and destination nodes. The graph also maintains a separate list of all the edges in the graph for quick access (when we need to access edges only e.g Kruskal's algorithm).

```
public class Graph {

   ArrayList<Node> nodes;
   ArrayList<Edge> graphEdges;

   public Graph(int V, int deg) {
      graphEdges = new ArrayList<Edge>();
      this.nodes = new ArrayList<Node>(V);
      for(int i = 0; i < V; i++){
         nodes.add(i, new Node(i,deg));
      }
   }

   public void addEdge(int u, int v, int w){
      nodes.get(u).edges.add(new Edge(u, v, w));
      nodes.get(v).edges.add(new Edge(v, u, w));
      graphEdges.add(new Edge(u, v, w));
   }

}
```

```
public class Node {
   int name;
   ArrayList<Edge> edges;

   public Node(int n, int deg){
      name = n;
      edges = new ArrayList<Edge>(deg);
   }

}
```

```
public class Edge {

    int source;
    int dest;
    int weight;

    public Edge(int u, int v, int w){
        this.source = u;
        this.dest = v;
        this.weight = w;
    }
}
```

1.1 Generation of Graph

The graph is getting generated using GraphGenerator. The neighboring nodes of a node and the weights of corresponding edges is generated randomly. We also put a cap on maximum permitted neighbors for a particular node. This value is 6 for sparse graphs and 1000 (20 % of number of vertices : 5000) for dense graphs.

```
public class GraphGenerator {

    private static final int PER_STEP_LIMIT = 500;

    public static Graph generateGraph(int vertCount, int maxDegree) {
        Graph G = new Graph(vertCount, maxDegree + 1);
        int[] nodeDegree = new int[vertCount];
        int limit;
        for (Node n : G.nodes){
            limit = 0;
            while(nodeDegree[n.name] < maxDegree && limit < PER_STEP_LIMIT){
                Random rn = new Random();
                int tempD = rn.nextInt(vertCount);
                if(nodeDegree[tempD] < maxDegree && noEdgeExists(G, n.name, tempD) && n.name != 
tempD){
                    int wt = rn.nextInt(1000) + 1;
                    G.addEdge(n.name, tempD, wt);
                    nodeDegree[n.name]++;
                    nodeDegree[tempD]++;
                }
                limit ++;
            }
        }

        return G;
    }
}
```

Here, for every node, I randomly try to find its neighbors from the set of nodes in the graph. We continue the trials till either the node gets the maximum permitted degree or we reach a fixed number of trials (500). After completion of this process, we are able to get a graph where nearly all nodes get the required number of edges.

2. Heap Structure

Here, we create

- Min-heap structure and implement subroutines for MINIMUM, INSERT and DELETE operations.
- Max-heap structure to store fringes (fringe nodes) in a Dijkstra-style algorithm.
- Max-heap structure to store edges in a Kruskal-style algorithm.

2.1 Min-heap

a) The vertices of graph are named by integers 0, 1, 2 ... , 4999
b) The heap is given by an array H[0 ... 4999], where H[i] gives the name of a vertex in the graph.
c) The vertex values are stored in a separate array D[0 ... 4999]. The value of vertex H[i] is given by D[H[i]]

MINIMUM :

```
if( H[0] is not empty )
  return H[0]
else                                        \\ H[0] is empty
  return "Heap Empty"
```

INSERT :

1. Insert the value at the end of array and keep a pointer at the end : H[ptr]   \\ ptr is end of array

```
\\  iParent = (i + 1)/2 -1 = location of parent of element at 'i'
2. while ( ptrParent >= 0 ) and D[H[ptr]] < D[H[ptrParent]]
  \\ ptrParent exists and the value of node at parent position is more than value of node

    swap(D[H[ptr]], H[ptrParent])
    ptr =  ptrParent
```

DELETE :

1. Swap the element at H[0] with the last element in heap.
   Swap(H[0], H[last])

2. Remove the last element and store the value in a variable : deletedVal
   deletedVal = H[last]
   last = last - 1

3. Swap the element at H[0] with the child with **smaller** value, provided this smaller child's value is also **lesser** than the value of node. Repeat till the node has no child with smaller value.
\\ i * 2 + 1   = position of left child of H[i]   = lChildPtr
\\ i * 2 + 2   = position of right child of H[i] = rChildPtr

ptr = 0

\\ D[H[smallerChildPos]] = Min (D[H[lChildPtr]] , D[H[rChildPtr]])
<u>while</u>(D[H[ptr]] > D[H[smallerChildPos]])
        Swap (H[ptr], H[smallerChildPos])
        ptr = smallerChildPos

4. return deletedVal

2.2 Max-heap (for fringe nodes : Dijkstra)

a) The vertices of graph are named by integers 0, 1, 2 ... , 4999
b) The heap is given by an array H[0 ... 4999], where H[i] gives the name of a vertex in the graph.
c) The vertex values are stored in a separate array D[0 ... 4999]. The value of vertex H[i] is given by D[H[i]]

MAXIMUM :

<u>if</u>(H[0] is not empty)
  return H[0]
<u>else</u>   \\ H[0] is empty
  return “Heap Empty”

INSERT :

1. Insert the value at the end of array and keep a pointer at the end : H[ptr]   \\ ptr is end of array

\\ iParent = (i + 1)/2 -1 = location of parent of element at 'i'
2. <u>while</u> (ptrParent >= 0) and D[H[ptr]] > D[H[ptrParent]]
  \\ ptrParent exists and the value of node at parent position is **less** than value of node

    swap(D[H[ptr]], H[ptrParent])
    ptr = ptrParent

DELETE :

1. Swap the element at H[0] with the last element in heap.
   Swap(H[0], H[last])

2. Remove the last element and store the value in a variable : deletedVal
   deletedVal = H[last]

last = last - 1

3. Swap the element at H[0] with the child with **larger** value, provided this larger child's value is also **larger** than the value of node. Repeat till the node has no child with larger value.
\\ i * 2 + 1 = position of left child of H[i] = lChildPtr
\\ i * 2 + 2 = position of right child of H[i] = rChildPtr

ptr = 0

\\ D[H[largerChildPos]] = Max (D[H[lChildPtr]] , D[H[rChildPtr]])
<u>while</u>(D[H[ptr]] < D[H[largerChildPos]])
    Swap (H[ptr], H[largerChildPos])
    ptr = largerChildPos

4. return deletedVal

REMOVE (val) :

1. Traverse H to find the location of 'val' – say 'k'
2. Swap the element at H[k] with the last element in heap.
   Swap(H[k], H[last])

3. Remove the last element and store the value in a variable : deletedVal
   deletedVal = H[last]
   last = last - 1

4. Swap the element at H[k] with the child with **larger** value, provided this larger child's value is also **larger** than the value of node. Repeat till the node has no child with larger value.
\\ i * 2 + 1 = position of left child of H[i] = lChildPtr
\\ i * 2 + 2 = position of right child of H[i] = rChildPtr

ptr = k

\\ D[H[largerChildPos]] = Max (D[H[lChildPtr]] , D[H[rChildPtr]])
<u>while</u>(D[H[ptr]] < D[H[largerChildPos]])
    Swap (H[ptr], H[largerChildPos])
    ptr = largerChildPos

5. return deletedVal

Analysis Of Algorithm

Finding the maximum (or minimum in case of min – heap) takes O(1) time as it involves reading of root element only. Insertion, deletion and remove all take O(log n) time where 'n' is number of elements.

3. MAX – CAPACITY – PATH

3.1 Algorithm based on modification of Dijkstra's Algorithm : without using heap

Input : An undirected graph G, source node 's' and destination node 't'

We have 3 arrays to be used in our algorithm
- status[0 .. 5000] : keeps the current state of a node – unseen 'u', intree 'i' and fringe 'f'.
- dad[0 .. 5000] : keeps the parent node for a node as per the best possible path possible to that node so far.
- D[0 .. 5000] : keeps the max – capacity value from source to that node as per current state.

```
1. For all vertices v in G, do status[v] = unseen
2. status[s] = intree
   D[s] = 10000000        \\ initialize max – capacity of source to infinity; any legal
                          \\ max – capacity value will be 1 to 1000
   dad[s] = -1            \\  initialize dad of source to null; any legal
                          \\ dad value will be 0 to 4999 i.e. name of a vertex
3. for each edge [s, w] do
   status[w] = fringe
   D[w] = weight[s, w]
   dad[w] = s
   fringeList.add(w)      \\ In this case we are maintaining the set of fringe nodes in a linked list -
                          \\ fringeList. 'w' is added at the end of list.

4. while there are fringes do

   i)   pick a fringe node v with maximum D[v] from fringeList   \\ getMaxFringeList subroutine below
   ii)  status[v] = intree
   iii) for each edge [v, w] do
          if status[w] == unseen then
               status[w] = fringe,
               D[w] = min {D[v], weight[v, w]}
               dad[w] = v
               fringeList.add(w)
          else if status[w]  == fringe and D[w] < min {D[v], weight[v, w]}
               D[w] = min {D[v], weight[v, w]}
               dad[w] = v

5. use D[0 .. 5000] and dad[0 .. 5000] to get the the max – capacity – path .
```

getMaxFringeList (fringeList) :

- traverse the fringeList to find the node with largest value. Copy this value to 'largest'.
- fringeList.remove(largest)
- return largest

```
public static void invokeMCPDijWithoutHeap(Graph myGraph, int source, int dest){
    char[] status = new char[5000];  // u = unseen; i = intree; f = fringe
    int[] D = new int[5000];
    int[] dad = new int[5000];
    LinkedList<Integer> fringe = new LinkedList<>();

    Arrays.fill(status, 'u'); // initialise all nodes to unseen
    D[source] = 10000000;     // initialise max capacity valueof source to infinity
    status[source] = 'i';     // make source intree
    Arrays.fill(dad, -1);     // initialize dad for all nodes to none


    for(Edge edge : myGraph.nodes.get(source).edges){
        status[edge.dest] = 'f';
        D[edge.dest] = edge.weight;
        dad[edge.dest] = source;
        fringe.add(edge.dest);
    }

    while(!fringe.isEmpty()){
        Integer maxFringe = fringe.element();
        for(Integer i : fringe){
            if(D[i] > D[maxFringe]){
                maxFringe = i;
            }
        }

        fringe.remove(maxFringe);

        status[maxFringe] = 'i';

        for(Edge edge : myGraph.nodes.get(maxFringe).edges){
            if(status[edge.dest] == 'u'){
                status[edge.dest] = 'f';
                D[edge.dest] = Math.min(D[edge.source], edge.weight);
                dad[edge.dest] = edge.source;
                fringe.add(edge.dest);
            }else if(status[edge.dest] == 'f' && D[edge.dest] < Math.min(D[edge.source], edge.weight))
{
                D[edge.dest] = Math.min(D[edge.source], edge.weight);
                dad[edge.dest] = edge.source;
            }
        }
    }

    printMaxPath(D,dad,source,dest);

}
```

Analysis Of Algorithm

Here as fringes are stored in a list, getting maximum value fringe node takes O(n) time and this is repeated for 'n' nodes. So, time spent in search of max fringe throughout running of algorithm is $O(n^2)$. Also, Dijkstra runs in O(m log(n)) time where 'n' is number of vertices and 'm' is number of edges. Hence, overall time complexity of the algorithm = $O(n^2)$.

3.2 Algorithm based on modification of Dijkstra's Algorithm : using heap

\\ The only difference with 3.1 is here we will use fringe heap instead of fringe list

Input : An undirected graph G, source node 's' and destination node 't'

We have 3 arrays to be used in our algorithm
- status[0 .. 5000] : keeps the current state of a node – unseen 'u', intree 'i' and fringe 'f'.
- dad[0 .. 5000] : keeps the parent node for a node as per the best possible path possible to that node so far.
- D[0 .. 5000] : keeps the max – capacity value from source to that node as per current state.

```
1. For all vertices v in G, do status[v] = unseen
2. status[s] = intree
   D[s] = 10000000        \\ initialize max – capacity of source to infinity; any legal
                          \\ max – capacity value will be 1 to 1000
   dad[s] = -1            \\  initialize dad of source to null; any legal
                          \\ dad value will be 0 to 4999 i.e. name of a vertex
3. for each edge [s, w] do
   status[w] = fringe
   D[w] = weight[s, w]
   dad[w] = s
   fringeHeap.add(w)      \\ In this case we are maintaining the set of fringe nodes in a max – heap
                          \\ fringeHeap. 'w' is added to the heap on the basis of D[w].

4. while there are fringes do

   i)   pick a fringe node v with maximum D[v] from fringeHeap  \\ fringeHeap.maximum()
   ii)  status[v] = intree
   iii) for each edge [v, w] do
          if status[w] == unseen then
               status[w] = fringe,
               D[w] = min {D[v], weight[v, w]}
               dad[w] = v
               fringeHeap.add(w)
          else if status[w]  == fringe and D[w] < min {D[v], weight[v, w]}
               fringeHeap.add(w)
               D[w] = min {D[v], weight[v, w]}
```

```
        fringeHeap.remove(w)
        dad[w] = v
```

5. use D[0 .. 5000] and dad[0 .. 5000] to get the the max – capacity – path .

Note : Refer to **Section 2.2 Max-heap (for fringe nodes : Dijkstra)** for the heap operations

```
add         => INSERT
maximum   =>  MAXIMUM
remove      => REMOVE
```

```
   public static void invokeMCPDijWithHeap(Graph myGraph, int source, int dest) throws
InterruptedException{
        char[] status = new char[5000];      // u = unseen; i = intree; f = fringe
        int[] D = new int[5000];
        int[] dad = new int[5000];

        MaxHeap fringe = new MaxHeap();

        Arrays.fill(status, 'u');       // initialise all nodes to unseen
        D[source] = 10000000;     // initialise max capacity valueof source to infinity
        status[source] = 'i';            // make source intree
        Arrays.fill(dad, -1);           // initialize dad for all nodes to none

        for(Edge edge : myGraph.nodes.get(source).edges){
            status[edge.dest] = 'f';
            D[edge.dest] = edge.weight;
            dad[edge.dest] = source;
            fringe.insert(edge.dest, D);
        }

        while(!fringe.isEmpty()){

            int maxFringe = fringe.delete(D);

            status[maxFringe] = 'i';

            for(Edge edge : myGraph.nodes.get(maxFringe).edges){
                if(status[edge.dest] == 'u'){
                    status[edge.dest] = 'f';
                    D[edge.dest] = Math.min(D[edge.source], edge.weight);
                    dad[edge.dest] = edge.source;
                    fringe.insert(edge.dest, D);
                }else if(status[edge.dest] == 'f' && D[edge.dest] < Math.min(D[edge.source], edge.weight))
{
                    fringe.remove(edge.dest, D);
                    D[edge.dest] = Math.min(D[edge.source], edge.weight);
                    dad[edge.dest] = edge.source;
```

```
                fringe.insert(edge.dest, D);

            }
        }
    }

    printMaxPath(D,dad,source,dest);
}
```

Analysis of Algorithm

The insert, delete and remove operations in heap take an overall time of O(m log(n)). Dijkstra's algorithm takes overall time of O((m + n) log(n)) which is equivalent to O(m log(n)) when 'm' is significantly larger than 'n'. Here, 'n' is number of vertices and 'm' is number of edges.

3.3 Algorithm based on modification of Kruskal's Algorithm : (using heap for edges)

This method is comprised of three parts :
- Get the set of edges which comprise of maximum spanning tree using modified Kruskal's algorithm.
- Create a (maximum spanning) tree, MST, using above set of edges : the tree created using these edges will be a maximum spanning tree.
- Search a path in MST between source (s) and destination(t) using Depth First Search.

3.3.1 Maximum spanning tree using modified Kruskal's algorithm.

We have 2 arrays to be used in our algorithm
- dad[0 .. 5000] : keeps the parent node for a node for 'find' operation so far.
- rank[0 .. 5000] : keeps the rank information for each node.

1. Sort all edges in decreasing order using Heap Sort. Use a Max – Heap for edges compared on edge weights.
$E_1, E_2, E_3, ... , E_m$

2. MST = Φ \\ an empty set

3. <u>for</u> i = 1 to m <u>do</u>

    let E_i = [u, w]
    <u>if</u> MST + E_i does not make a cycle
    \\ r1 = Find (u)
    \\ r2 = Find (w)
    \\ r1 != r2 => no cycle

    <u>then</u> MST = MST + E_i
    \\ Union (r1, r2)

4. return MST

Implementation of Kruskal's Algorithm

```
  public static void invokeMCPKruskal(Graph myGraph, int mySrc, int myDest){

      int[] dad = new int[5000];
      int[] rank = new int[5000];

      Arrays.fill(dad, -1);

      EdgeHeap edgeHeap = new EdgeHeap(myGraph.graphEdges);

      ArrayList<Edge> maxSpanningTree = new ArrayList<>();

      for(Edge edge : myGraph.graphEdges){
          int source = edge.source;
          int dest = edge.dest;

          // Find
          int r1 = find(source, dad);
          int r2 = find(dest, dad);

          // Union
          if(r1 != r2){
              maxSpanningTree.add(edge);
              if(rank[r1] > rank[r2]){
                  dad[r2] = r1;
              }else if(rank[r1] < rank[r2]){
                  dad[r1] = r2;
              }else{
                  dad[r2] = r1;
                  rank[r1]++;
              }
          }
      }
      createMCPfromMST(maxSpanningTree, mySrc, myDest);            // see 3.3.2
  }
```

3.3.2 Create a tree from set of edges (MST) returned in 3.3.1

```
 private static void createMCPfromMST(ArrayList<Edge> maxSpanningTree, int source, int dest) {
      Graph mstGraph = GraphGenerator.generateGraph(VERT_COUNT, MAX_DEGREE,
maxSpanningTree);
      int[] dad = new int[5000];
      int[] D = new int[5000];
      char[] status = new char[5000];
      Arrays.fill(status, 'w');
```

```
        Arrays.fill(dad, -1);

        D[source] = 10000000;

        dfsOnTree(source, dest, mstGraph, dad, D, status);                    // see 3.3.3
        printMaxPath(D,dad,source,dest);

    }
```

3.3.3 Run Depth First Search on tree created in 3.3.2

```
    private static void dfsOnTree(int source, int dest, Graph mstGraph, int[] dad, int[] D, char[] status) {

        status[source] = 'b';
        if(source == dest){ return;}

        for(Edge edge : mstGraph.nodes.get(source).edges){
            if(status[edge.dest] == 'b') continue;
            dad[edge.dest] = source;
            if(edge.weight < D[source]){
                D[edge.dest] = edge.weight;
            }else{
                D[edge.dest] = D[source];
            }

            dfsOnTree(edge.dest, dest, mstGraph, dad, D, status);
        }
    }
```

Analysis of Algorithm

The Make-set and union operations used in Kruskal's algorithm take O(1) time. Find operations take O(m log(n)) time which can be further improvised to get O(m log*(n)) by storing ranks and using path compression. The sorting of edges using heap sort takes O(m log(m)) time. Creation of tree from the set of edges (in MST) takes O(m) time. DFS used to get the final path takes O(m+n) time. So overall the time complexity is bounded by the operation of sorting of edges : time O(m log(n)).
Here, 'n' is number of vertices and 'm' is number of edges.
Note – sorting takes O(m log(m)) which is same as O(m log(n^2)) which is same as O(m log(n))

4. Testing and Results

Tested with the below CPU configuration :
Architecture: x86_64
Model name: Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz

SPARSE GRAPH (REGULAR-6 GRAPH) Time value in milliseconds

GRAPH	Source & Dest	Dijkstra(No Heap)	Dijkstra(Heap)	Kruskal
GRAPH 1	4324 -> 2490	144	34	29
	4295 -> 4629	145	31	29
	2792 -> 1417	143	34	28
	2731 -> 2482	145	36	38
	1138 -> 3100	145	33	27
GRAPH 2	1347 -> 4614	134	34	27
	135 -> 2000	147	34	32
	379 -> 883	143	35	30
	354 -> 1599	145	34	29
	985 -> 1699	152	37	32
GRAPH 3	4334 -> 4576	143	33	29
	373 -> 2750	146	35	30
	249 -> 2543	147	39	32
	1049 -> 1959	146	41	29
	4948 -> 4863	136	36	29
GRAPH 4	4764 -> 1712	142	33	27
	4194 -> 2227	138	35	27
	2872 -> 2698	147	39	41
	2844 -> 1663	150	38	32
	1641 -> 2864	147	37	33
GRAPH 5	2008 -> 2893	141	31	32
	3738 -> 1370	132	35	27
	17 -> 756	152	38	33
	4262 -> 1788	157	40	32
	771 -> 2915	154	38	33

DENSE GRAPH (Density : 20%) Time value in milliseconds

GRAPH	Source & Dest	Dijkstra(No Heap)	Dijkstra(Heap)	Kruskal
GRAPH 1	1959 -> 2286	223	106	2376
	1827 -> 4658	223	114	2378
	842 -> 3434	236	119	2399
	4034 -> 2295	220	117	2422
	3722 -> 4139	218	106	2350
GRAPH 2	1593 -> 3627	220	103	2365
	2455 -> 827	220	106	2416
	3159 -> 3809	226	107	2374
	3398 -> 4494	231	160	2489
	4133 -> 4417	228	119	2419
GRAPH 3	3012 -> 3189	225	112	2357
	4399 -> 4256	228	114	2367
	1254 -> 2312	212	109	2406
	4278 -> 3531	232	119	2406
	1565 -> 2250	231	121	2431
GRAPH 4	4028 -> 1026	228	132	2412
	434 -> 4485	233	114	2416
	30 -> 1685	229	117	2404
	1354 -> 4203	234	115	2440
	745 -> 1754	231	112	2451
GRAPH 5	601 -> 2354	240	114	2430
	608 -> 695	210	109	2436
	2435 -> 3407	225	113	2399
	4080 -> 601	213	115	2435
	4252 -> 1187	217	106	2413

A sample output is included below :

SPARSE GRAPH

Source and Destination : 1729 -> 371

MCP without using Heap starts :
Below is the reversed path with max path values :
371 -> 711; 1458 -> 815; 3997 -> 815; 2788 -> 815; 2787 -> 815; 2786 -> 815; 2770 -> 815; 708 -> 815; 3252 -> 815; 3558 -> 815; 577 -> 815; 578 -> 815; 4340 -> 815; 978 -> 815; 977 -> 815; 3178 -> 815; 686 -> 815; 687 -> 815; 348 -> 815; 1352 -> 815; 1351 -> 815; 1350 -> 815; 682 -> 815; 3219 -> 815; 550 -> 901; 4933 -> 901; 1730 -> 901; 1729 -> 10000000;
End !!MCP without using Heap ends. Time taken : 131

MCP with using Heap starts :
Below is the reversed path with max path values :
371 -> 711; 1458 -> 815; 3997 -> 815; 2788 -> 815; 2787 -> 815; 2786 -> 815; 2770 -> 815; 708 -> 815; 3252 -> 815; 3558 -> 815; 577 -> 815; 578 -> 815; 4340 -> 815; 978 -> 815; 977 -> 815; 3178 -> 815; 686 -> 815; 687 -> 815; 348 -> 815; 1352 -> 815; 1351 -> 815; 1350 -> 815; 682 -> 815; 3219 -> 815; 550 -> 901; 4933 -> 901; 1730 -> 901; 1729 -> 10000000;
End !!MCP with using Heap ends. Time taken : 36

MCP by using Kruskal Max Spanning tree starts :
Below is the reversed path with max path values :
371 -> 711; 1458 -> 815; 3997 -> 815; 2788 -> 815; 2787 -> 815; 2786 -> 815; 2770 -> 815; 708 -> 815; 707 -> 815; 1867 -> 815; 3041 -> 815; 4061 -> 815; 494 -> 815; 4455 -> 815; 2512 -> 815; 2732 -> 815; 3022 -> 815; 646 -> 815; 916 -> 815; 915 -> 815; 3937 -> 815; 692 -> 815; 693 -> 815; 1395 -> 815; 3053 -> 815; 1519 -> 815; 609 -> 815; 3496 -> 815; 2336 -> 815; 3524 -> 815; 3523 -> 815; 3522 -> 815; 3002 -> 815; 3421 -> 815; 1780 -> 815; 1779 -> 815; 1938 -> 815; 1939 -> 815; 1671 -> 815; 4677 -> 815; 4366 -> 815; 4365 -> 815; 4659 -> 815; 4805 -> 815; 4804 -> 815; 413 -> 815; 3980 -> 815; 4686 -> 815; 3395 -> 815; 2186 -> 815; 3881 -> 815; 2370 -> 815; 1292 -> 815; 1291 -> 815; 3790 -> 815; 238 -> 815; 239 -> 815; 240 -> 815; 591 -> 815; 3373 -> 815; 1408 -> 815; 686 -> 815; 687 -> 815; 348 -> 815; 1352 -> 815; 1351 -> 815; 1350 -> 815; 682 -> 815; 3219 -> 815; 550 -> 901; 4933 -> 901; 1730 -> 901; 1729 -> 10000000;
End !!MCP by using Kruskal Max Spanning tree ends. Time taken in milli secs: 29

DENSE GRAPH

Source and Destination : 3912 -> 3721

MCP without using Heap starts :
Below is the reversed path with max path values :
3721 -> 998; 2880 -> 998; 22 -> 998; 1104 -> 998; 3490 -> 998; 2973 -> 998; 4277 -> 998; 3232 -> 998; 1896 -> 998; 1773 -> 998; 4617 -> 998; 3364 -> 998; 1169 -> 998; 1467 -> 998; 4443 -> 998; 2241 -> 998; 2552 -> 998; 4601 -> 998; 340 -> 998; 357 -> 998; 218 -> 998; 4860 -> 998; 3376 ->

998; 416 -> 998; 2735 -> 998; 4094 -> 998; 1840 -> 998; 2608 -> 998; 3445 -> 998; 2620 -> 998; 2159 -> 998; 3278 -> 998; 3687 -> 998; 4118 -> 998; 3893 -> 998; 4670 -> 998; 268 -> 998; 452 -> 998; 3912 -> 10000000;
End !!MCP without using Heap ends. Time taken : 239

MCP with using Heap starts :
Below is the reversed path with max path values :
3721 -> 998; 2880 -> 998; 3833 -> 998; 1032 -> 998; 1538 -> 998; 1279 -> 998; 2676 -> 998; 3836 -> 998; 87 -> 998; 3788 -> 998; 1106 -> 998; 1872 -> 998; 2337 -> 998; 4612 -> 998; 452 -> 998; 3912 -> 10000000;
End !!MCP with using Heap ends. Time taken : 106

MCP by using Kruskal Max Spanning tree starts :
Below is the reversed path with max path values :
3721 -> 998; 2880 -> 998; 22 -> 998; 115 -> 998; 8 -> 998; 4890 -> 998; 3355 -> 998; 4526 -> 998; 3411 -> 998; 413 -> 998; 4202 -> 998; 2500 -> 998; 1890 -> 998; 1195 -> 998; 3997 -> 998; 3338 -> 998; 845 -> 998; 28 -> 998; 1993 -> 998; 4862 -> 998; 486 -> 998; 270 -> 998; 4121 -> 998; 4912 -> 998; 812 -> 998; 252 -> 998; 2755 -> 998; 150 -> 998; 4690 -> 998; 3683 -> 998; 1630 -> 998; 859 -> 998; 2996 -> 998; 3182 -> 998; 4207 -> 998; 1425 -> 998; 4602 -> 998; 1634 -> 998; 2763 -> 998; 46 -> 998; 2202 -> 998; 4341 -> 998; 1725 -> 998; 315 -> 998; 4612 -> 998; 452 -> 998; 3912 -> 10000000;
End !!MCP by using Kruskal Max Spanning tree ends. Time taken in milli secs: 2514

5. Analysis of results

According to the data from the **sparse** graph below is the ranking of the algorithms based on their performance –
1. Krushkal
2. Dijkstra with heap.
3. Dijkstra without heap.

Kruskal executes marginally (10% - 20%) faster than Dijkstra with heap and Dijkstra with heap executes 3-4 times faster than Dijkstra without heap. The excessive time taken in case of Dijkstra without heap is obvious as the time complexity to get maximum value fringe may take O(n^2) time. But, in cases of Dijkstra with heap and Kruskal's the time complexity is same - O(m log(n)). Clearly, the results show that the Big-O in case of Dijkstra with heap is larger than Big-O in case of Kruskal's. This can be explained as

(a) Kruskal's does sorting only once (though on a much larger set). It can get benefit of some patterns in data e.g data (edge weights nearly sorted)

(b) In case of Kruskal's algorithm, every data element(edge) undergoes "insert" and "remove" operations exactly once. This may not be the case with Dijkstra's (with heap). In case we update the value of a node which is already a fringe, we need to "remove" it and "insert" it again.

(c) The memory involved (and hence referred to) in case of Dijkstra's(with heap) is more when compared to Kruskal's.
Kruskal uses "dad" array and "rank" array
Dijkstra uses "status", "dad" and "D (node value)", and a heap for fringe nodes.

According to the data from the **dense** graph below is the ranking of the algorithms based on their performance –
1. Dijkstra with heap.
2. Dijkstra without heap.
3. Kruskal

Dijkstra with heap executes twice faster than Dijkstra without heap and Dijkstra without heap executes 10 times faster than Kruskal.
The improvement in Dijkstra without heap over Dijkstra with heap is obvious as their corresponding time complexities (O(m log(n)) and O(n^2)) .
But, Kruskal's algorithm is taking a lot more time here because as 'm' increases (m >> n), the sorting takes a lot more time. We should note that in case of Dijkstra's (with or without heap), although theoretically the the set of fringes can be of 'n' size at any given point. But practically, if we check on average, the number of elements in fringe set at any given point will be much lower than 'n'.
Hence, insertion/removal will take comparatively much lower time. On the other hand, in case of Kruskal's algorithm we need to do sorting on the complete set of edges (size 'm') without exception.

6. Improvement suggestions :

1. As sorting of edges is a one time operation in case of Kruskal's algorithm (no insertion/removal on a later stage), we can go for any sorting algorithm which performs better for our use case and not limit ourselves to heap sort. Practically, quick - sort or merge - sort give more optimal results for most scenarios. I have tested that if we replace heap sort by merge sort, in the implementation of Kruskal's algorithm, the time taken reduces to half.

2. We can be more careful in using path compression in case of "Find" operations. Path compressions aim at reducing a lot of "read" operations on the cost of introducing some "write" operations.
For most systems, "write" operations are much costly compared to "read" operations. We need to be careful because this trade off may not always be beneficial and may depend on our use case(data set) and the systems where code is supposed to run.