CSCE 629

Analysis of Algorithms

Network Routing Protocols

PROJECT REPORT

NAME: SAI BALAGI THALANAYAR SWAMINATHAN

UIN: 723006059

Introduction:

The main objective of this project is to implement a network routing protocol using data structures and algorithms. We solve the Maximum Bandwidth path problem using modifications of Dijkstra , Kruskal algorithms and evaluate the performance of these algorithms on dense and sparse graphs.

Maximum Capacity Problem:

Given an unweighted graph G=(V,E), It is the problem of finding a path between two designated vertices, maximizing the weight of the minimum weight edge in the path. The weight of the minimum weight edge is known as the capacity or band width.

This problem is used to find the maximum capacity bandwidth between routers in the internet. It has many other applications in digital compositing, metabolic analysis and the computation of maximum flows.

The following are the various polynomial time algorithms used to solve the maximum capacity problem.

1. **Modified Dijkstra Algorithm without using heaps.**
2. **Modified Dijkstra Algorithm using heaps.**
3. **Kruskal Algorithm with Heap sort and Breadth First Search.**

Pseudocode:

Modified Dikstra Algorithm without using Heaps

1. For v=1 to n do
2.   status[v]=unseen;
3. For each edge [s,w] do
4.   Status[w]=fringe;
5.   Dad[w]=S
6.   Cap[w]=weight[s,w]
7. While there are fringes do
8.   pick a fringe with the largest capacity

```
9.      status[v]=in-tree
10.     for each edge [v,w] do
11.             if(status[w]=unseen) then
12.                     status[w]=fringe;
13.                     dad[w]=v;
14.                     cap[w]=min{cap[v],weight[v,w]}
15.             elseif (status[w]=fringe&&cap[w]< min{cap[v],weight[v,w]})
16.                     dad[w]=v;
17.                     cap[w]= min{cap[v],weight[v,w]}
18.     return Dad[.]
```

If the above algorithm is implemented without a heap, then it takes **O(n^2)** time.

Modified Dijkstra Algorithm using Heaps

```
1. For v=1 to n do
2.      status[v]=unseen;
3. For each edge [s,w] do
4.      Status[w]=fringe;
5.      Dad[w]=S
6.      Cap[w]=weight[s,w]
7.While there are fringes do
8.      pick a fringe with the largest capacity
9.      status[v]=in-tree
10.     for each edge [v,w] do
11.             if(status[w]=unseen) then
12.                     status[w]=fringe;
13.                     dad[w]=v;
14.                     cap[w]=min{cap[v],weight[v,w]}
15.             elseif (status[w]=fringe&&cap[w]< min{cap[v],weight[v,w]})
```

16. dad[w]=v;

17. cap[w]= min{cap[v],weight[v,w]}

18. return Dad[.]

If the above algorithm is implemented using heaps, then it takes time **O(mlogn).**

Modified Kruskal Algorithm using Heap Sort and BFS

1. Sort the edges in non increasing order

2. For vertex v=1 to n do

3. Makeset(v)=0;

4.For i=1 to m do

5. Let ei=[vi,wi];

6. r1=find(vi)

7. r2=find(wi)

8. if(r1≠r2)

9. ei is added to T

10. Union(r1,r2)

11.return the s-t path in T

Makeset(v)

1. dad[v]=0

2. rank[v=0

Find(v)

1. let w=v

2.while dad[w]≠0 do

3. w=dad[w]

4.return[w];

Union(r1,r2)

1. if(rank[r1]>rank[r2])

2. dad[r2]=r1

3.elseif(rank[r1]<rank[r2])

3. dad[r1]=r2;

4.elseif(rank[r1]==rank[r2])

5. dad[r2]=r1

6. rank[r1]++

This algorithms theoretically takes **O(mlogn)** time.

Theoretically, dijkstra without heap takes more time compared to Dijkstra with heap and Modified kruskal Algorithm.

Practical Implementation

Generation of Random Graphs:

1. Generating a random graph of 5000 vertices with 6 degree of freedom

Since each vertex must have 6 neighbours, there are 30,000 edges in the graph. As this is an undirected graph, 15000 unique edges are present. For each vertex, we have a counter which notes down the number of neighbors. We generate 2 random numbers between 1 to 5000 and if there is no edge already between them and if the degree of each number is less than 6, we add it to our graph.

Function code:

```
void generategraphwith6degree()
 {
	 for(int i=1;i<=vertex;i++)
		vertexcounter[i]=0;

	int random1,random2;
	int counter=0;

	while(counter<(vertex*deg)/2)
	{
		random1=rand()%vertex+1;
		random2=rand()%vertex+1;

	if((vertexcounter[random1]<6)&&(vertexcounter[random2]<6)&&(random1!=random2)
&&(graph[random1][random2]!=1))
		{
			graph[random1][random2]=1;
			graph[random2][random1]=1;
			counter++;
```

```
                        int w= rand()%100+1;
                        addedge(random1,random2,w);

                }

        }
        for(int i=1;i<=4999;i++)
 {
          int w2=rand()%100+1;
          addedge(i,i+1,w2);
 }

}
```

2. Generating a Random graph of 5000 vertices where each vertex has edges going to about 20% of the other vertices.

We cannot use the previous method as it will definitely take a lot of time as this graph is dense. So, we try to generate approximately 1000 edges for each vertex. The graph matrix is initialized to zero. We generate a random number between 1 to 5000. If the number is less than 500, then we add that cell to the graph by putting the value of that cell as 1.

The following is the code:

```
void generategraphwith1000degree()
 {

 for(int i=1;i<=vertex;i++)
                for(int j=1;j<=vertex;j++)
                        graph[i][j]=0;
  for(int i=1;i<=vertex;i++)
        {
                for(int j=1;j<=vertex;j++)
                {

                if(i!=j)
                {
                        int r=rand()%5000+1;
                        if(r<500)
```

```
                {

                        seed();
                        int w1= rand()%100+1;
                        addedge(i,j,w1);

                }
            }
            }
        }

    for(int i=1;i<=4999;i++)
    {
        int w2=rand()%100+1;
        addedge(i,i+1,w2);
    }

        }
```

2. Representing Heaps

An array named heap1 was used to store the vertices and another array named heap2 was used to store the values for these vertices. Functions to maxheapify, build the max heap, extract max and heap sort have been implemented.

The following are the functions to implement heap:

```
void maxheapify(int* heap1, int i, int size)
{

        int largest;
        int left=2*i;
        int right=(2*i)+1;
        if(left<=size&&heap2[heap1[left]]>heap2[heap1[i]])
                largest=left;
        else
                largest=i;
        if(right<=size&&heap2[heap1[right]]>heap2[heap1[largest]])
                largest=right;

        if(largest!=i)
        {
                int temp=heap1[i];
                heap1[i]=heap1[largest];
                heap1[largest]=temp;
                maxheapify(heap1,largest,size);
```

```
		}
}

void buildmaxheap(int* heap1,int size)
{
	for(int i=size/2;i>=1;i--)
		maxheapify(heap1,i,size);
}


int extractmax(int* heap1)
{
	//int tempsize=size;
	if(size<1)
	{
		//cout<<size;
	   cout<<"error";
	}
	else
	{
		int max=heap1[1];
	    heap1[1]=heap1[size];
	    size=size-1;
	    maxheapify(heap1,1,size);
	    return max;
	}
}

void heapsort()
{
int tempsize=size;
 buildmaxheap(heap1,tempsize);
 for(int i=size;i>=2;i--)
 {
 int temp=heap1[1];
 heap1[1]=heap1[i];
 heap1[i]=temp;
 tempsize--;
 maxheapify(heap1,1,tempsize);
 }
}
```

Modified Dijkstra Algorithm without using a heap

The graph was implemented as an adjacency list for faster processing. An array called set was used to see which vertices where added and which vertices where not. A function called maxdistance was used to find the vertex with largest capacity among the vertices which are not visited. Once the vertex u is found, then the adjacency list of u contains the fringes. The fringes are checked using the following condition:

```
if(set[p->dest]==false&&dist[p->dest]<min(dist[u],p->weight))
```

Based on this, the capacity for each vertex is calculated. The array dist is used to store the capacity of the vertices from the source. Dist[final] gives us the max band width from the source to the final vertex. The following is the code to implement it.

```
int maxDistance(int dist[], bool sptSet[])
{
   // Initialize min value
   int max = -1000, max_index;

   for (int v = 1; v <= vertex; v++)
         {
      if (sptSet[v] == false && dist[v] >= max)
                  {
         max = dist[v];
                        max_index = v;
                  }
         }
   return max_index;
}

int min(int x,int y)
{
         if(x>y)
                  return y;
         else
                  return x;
}

void dikstra(int src1,int final)
{
         int dist[6000];
         bool set[6000];
         for(int i=1;i<=vertex;i++)
         {
                  dist[i]=-1000;
                  set[i]=false;
```

```
	}
	dist[src1]=1000;
	for(int i=1;i<=vertex;i++)
	{

		int u = maxDistance(dist,set);
		set[u]=true;
		node* p=array1[u].head;
		while(p)
		{

			if(set[p->dest]==false&&dist[p->dest]<min(dist[u],p->weight))
			{
				dist[p->dest]=min(dist[u],p->weight);

			}
			p=p->next;
		}
	}

		cout<<"max	bandth	with	between"<<src1<<"and"<<final<<"is
:"<<dist[final]<<endl;
}
```

Modified Dijkstra Algorithm using Heaps:

Here instead of storing the capacity in an array, we store it in a heap. Heap1 contains the vertices names and heap2[heap1[i]] will give the weight of the vertex present at heap1[i]. The vertex with the max capacity at every iteration is selected using extract max function of the heap. Once the vertex with highest capacity is found, the adjacency list of that vertex is checked. The condition to update the capacity for each vertex is as follows:

```
if(heap2[p->dest]<min(heap2[u],p->weight))
			{
				heap2[p->dest]=min(heap2[u],p->weight);
			}
```

The max band width is given by the array heap2 which stores the max capacity.

The code is as follows:

```
void dikstra(int src1,int final)
{
```

```
for(int i=1;i<=vertex;i++)
{
    heap1[i]=i;
    heap2[heap1[i]]=-1000;
}
heap2[heap1[src1]]=1000;
buildmaxheap(heap1,size);


for(int i=1;i<=vertex;i++)
{
    //cout<<2;
    int u = extractmax(heap1);


    node* p=array1[u].head;

    while(p)
    {

        if(heap2[p->dest]<min(heap2[u],p->weight))
        {
            heap2[p->dest]=min(heap2[u],p->weight);
        }
        p=p->next;
    }
}

//for(int i=1;i<=vertex;i++)
    cout<<"The max bandwidth between"<<src1<<"and"<<final<<"is:"<<heap2[final]<<endl;

}
```

Modified Kruskal algorithm using heap sort

A modification of kruskal algorithm is used to find the max capacity. The union, find and makeset functions are implemented. The changes that we do to the kruskal is that we arrange the edges in the decreasing order. We use an array to a structure called edge. Edge contains the following entities: from, to, weight and edgeindex. For each edge, we give these values. Heap sort is used to arrange the edges in the decreasing order. Once we get the Max spanning tree. Breadth First search is used to find the max band width between two vertices in the tree. The array mstpath is used to store the final tree structure and BFS is applied on this tree structure.The following is the code pertaining to implementing the modified kruskal.

```
 void makeset()
{
	for(int i=1;i<=vertex;i++)
	{
		dad[vertex]=0;
		rank1[vertex]=0;
	}
}

int find(int v)
{
	int w=v;
	while(dad[w]!=0)
	{
		w=dad[w];
	}
	return w;
}

void union1(int p1,int p2)
{
	if(rank1[p1]>rank1[p2])
		dad[p2]=p1;
	else if(rank1[p1]<rank1[p2])
		dad[p1]=p2;
	else if(rank1[p1]==rank1[p2])
	{
		dad[p2]=p1;
		rank1[p1]++;
	}
}



void initmstpath()
{
	for(int i=1;i<=vertex;i++)
		for(int j=1;j<=vertex;j++)
			mstpath[i][j]=0;
}

void kruskal()
{



	for(int i=size;i>=1;i--)
	{
```

```
            int k=edge1[heap1[i]].edgeindex;
            int r1= find(edge1[k].from);
            int r2= find(edge1[k].to);
            if(r1!=r2)
            {
        mstpath[r1][r2]=edge1[k].weight;
              mstpath[r2][r1]=edge1[k].weight;
              union1(r1,r2);
            }
        }
}
```

Results:

The programs where run in Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz with 8 GB ram.

Final Results:

Average time values	Dense	Sparse
Dijkstra without heap	0.5459	0.1079
Dijkstra with heap	0.3947	0.0074
Modified kruskal	0.208	0.065

Analysis:

The experimental result exactly matches the theoretical results. Theoretically modified Dijkstra without heap runs in $0(n^2)$. Dijkstra with Heap and modified kruskal runs in O(mlogn) time. The constant value in the complexity expression of modified kruskal is lesser in value as compared to modified dijkstra with heap. Theoretically Kruskal should be the fstest followed by modified dijkstra with heap and then modified dijkstra without heap. Practically, from the above table we can see that the modified kruskal is the fastest followed by the modified dijkstra with heap implementation for the maximum bandwidth path. Modified dijkstra without using a heap is the slowest among all three. Hence the theoretical results matches the practical implementation.

Dijkstra without heap runs in $O(n^2)$. The step where we pick the fringe with the largest capacity runs in O(n) time. Going through the neighbors of the vertex and then calculating the max capacity takes O(n) when we do not put it as a heap but it takes O(logn) if we put it as a heap.

Theoretically, these algorithms should run on faster on sparse graph than in dense graph due to the lesser number of edges.

Practically,we see from the table that the sparse graph implementation is faster than dense graph implementation as the m value is lesser for the sparse graphs. Hence these algorithms run faster on a sparse graph than in a dense graph. Hence we can see that the theoretical results matches the experimental results

Discussion about the implementation of various data structure and its implications:

In this project, I have implemented the graph as an adjacency list rather than adjacency matrix as it will take a longer time to go to the neighbors of the vertex in case of adjacency matrix as opposed to adjacency list. From this project, we can conclude that implementing the right data structure for an algorithm is crucial as there is difference in time of running for large vertices.

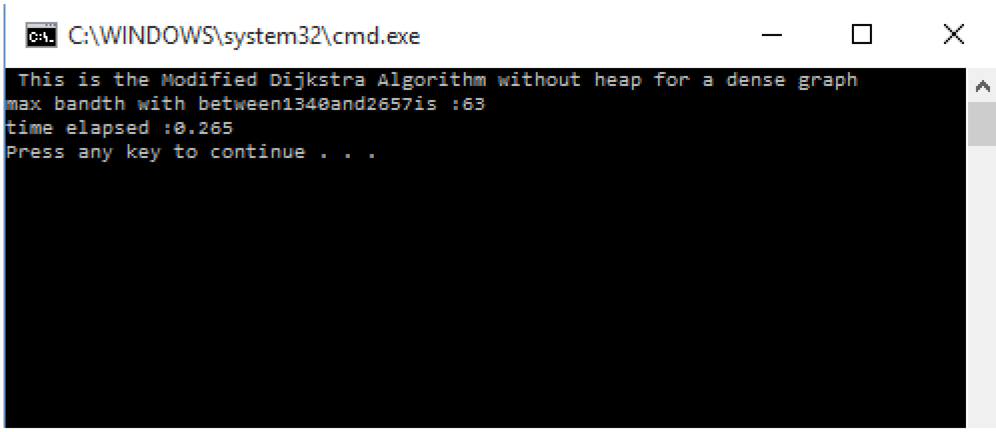
Detailed Results:

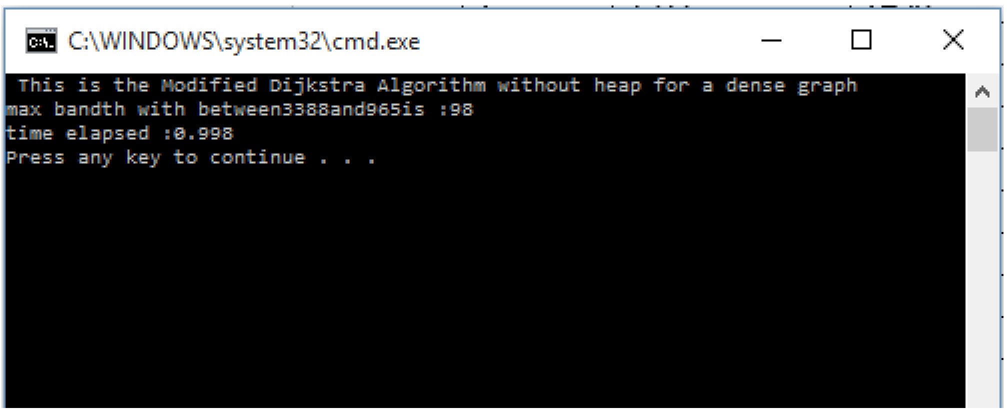
Algorithm Name: Modified Dijkstra without using Heaps

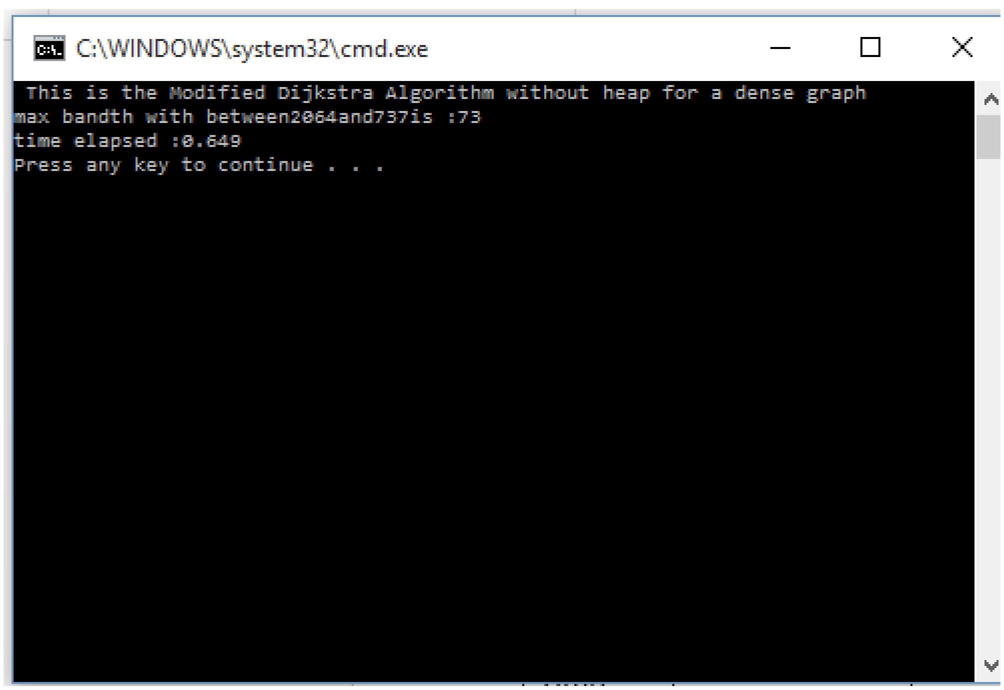
Graph Type: Dense Graph

SNO	Source	destination	Weight	Time
1	3187	3725	63	0.289
2	1517	2458	18	0.378
3	541	2447	29	0.484
4	1340	2657	63	0.265
5	3388	965	98	0.998
6	3641	2793	38	0.472
7	2064	737	73	0.649
8	3746	3478	83	0.456
9	4205	4755	20	0.976
10	1697	2201	35	0.492
Total Time				5.459
Average				0.5459

Some Snap Shots:





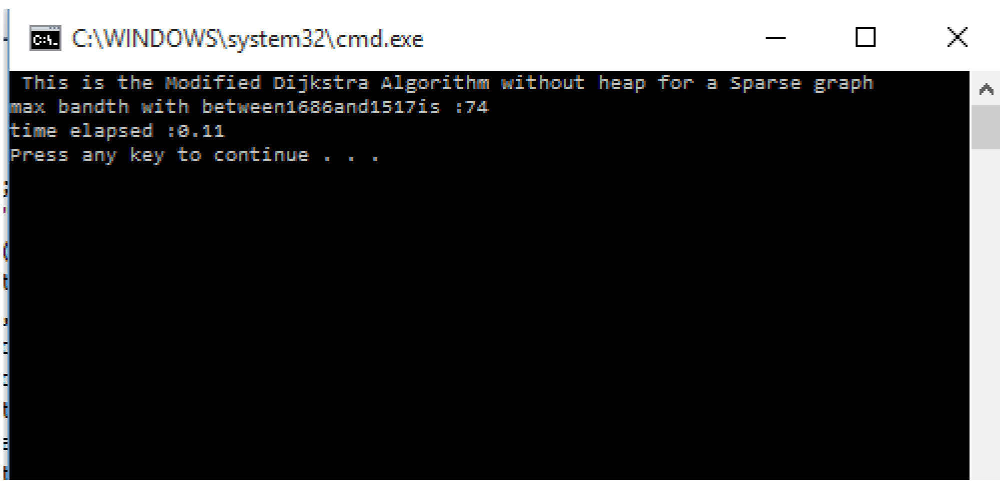


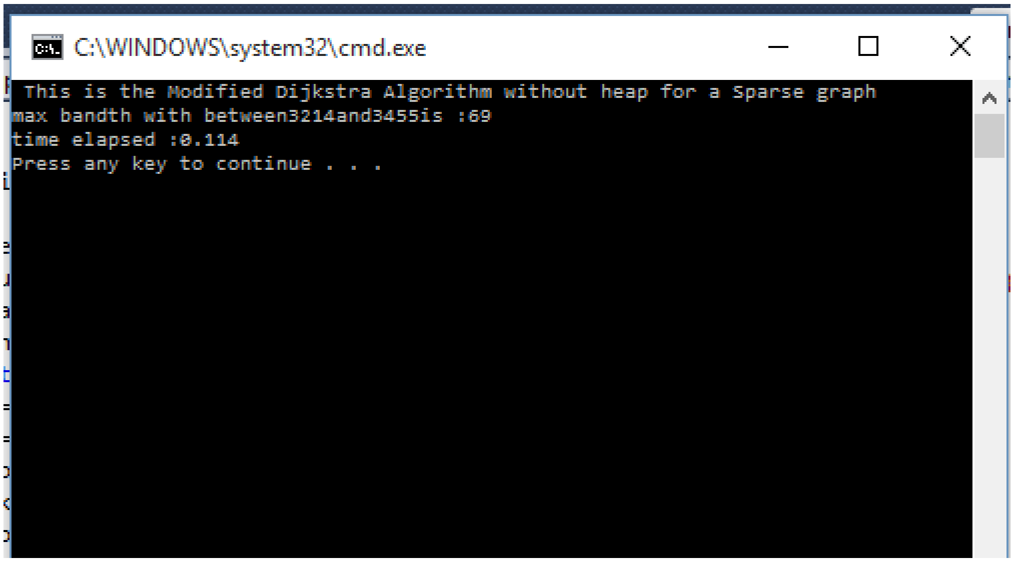
Algorithm Name: Modified Dijkstra without using Heaps

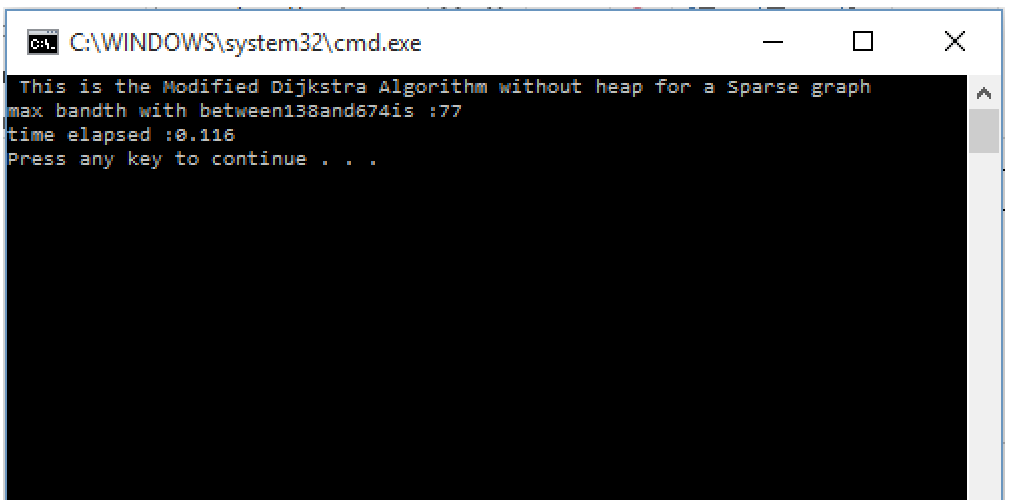
Graph Type: Sparse Graph

SNO	Source	destination	Weight	Time
1	2288	146	60	0.109
2	4939	4043	78	0.109
3	138	674	77	0.116
4	1873	4935	79	0.111
5	3214	3455	69	0.114
6	985	2665	67	0.094
7	1246	1911	72	0.1
8	369	2749	72	0.105
9	1686	1517	74	0.11
10	2357	2663	84	0.111
Total Time				1.079
Average				0.1079

Snap Shots:







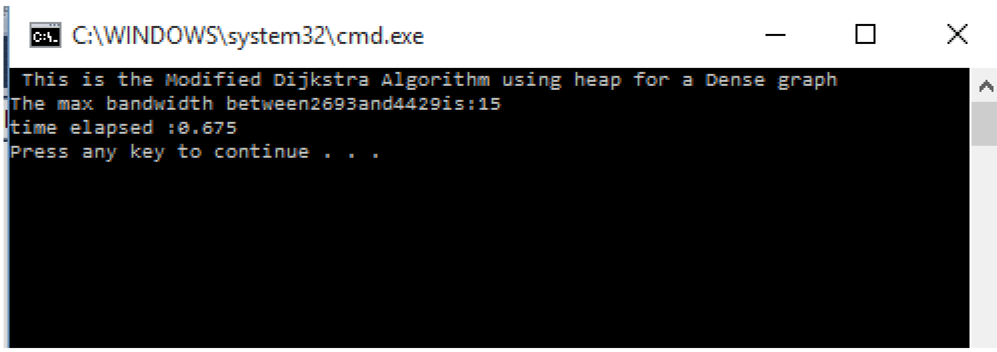
Algorithm Name: Modified Dijkstra using Heaps

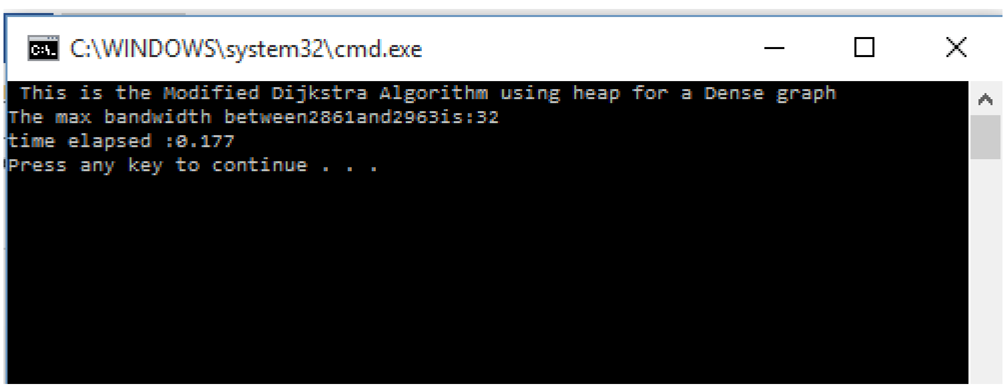
Graph Type: Dense Graph

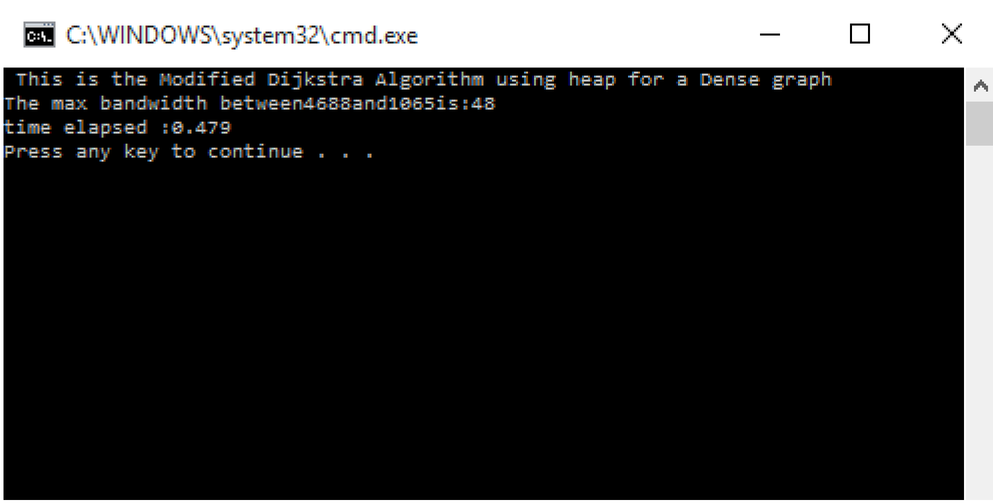
SNO	source	destination	Weight	Time
1	408	4365	99	0.442
2	4087	3977	70	0.371
3	2211	2429	78	0.34
4	3048	659	15	0.467

5	4688	1065	48	0.479
6	663	1434	30	0.457
7	2693	4429	15	0.675
8	588	1511	8	0.492
9	4206	3980	82	0.313
10	2861	2963	32	0.177
Total Time				3.947
Average				0.3947

Snap Shots:





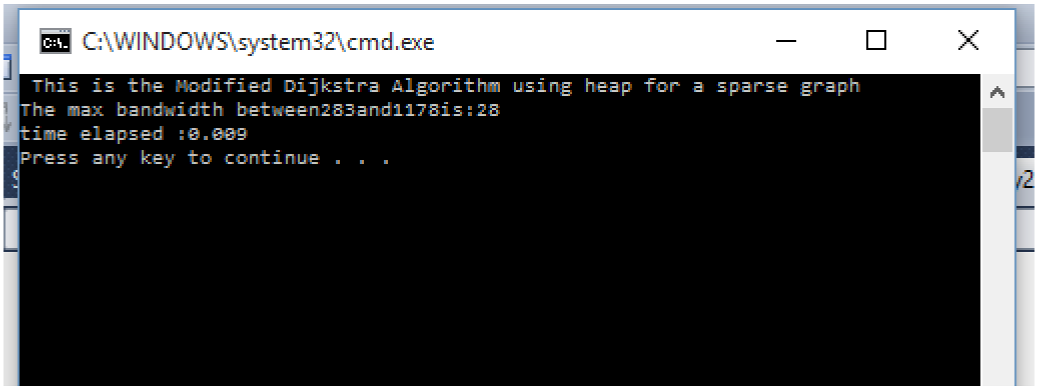


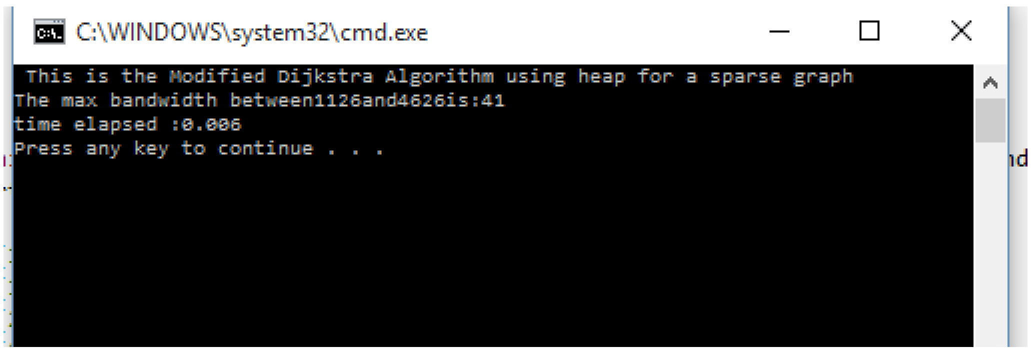
Algorithm Name: Modified Dijkstra using Heaps

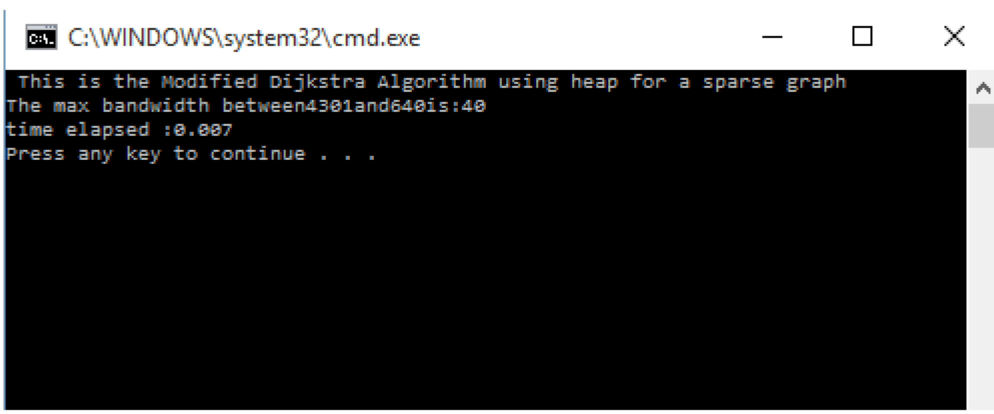
Graph Type: Sparse Graph

SNO	Source	destination	Weight	Time
1	4301	640	40	0.007
2	1126	4626	41	0.006
3	283	1178	28	0.009
4	2518	88	38	0.009
5	2870	2054	51	0.006
6	683	1965	48	0.008
7	2185	3518	46	0.006
8	765	1098	50	0.007
9	2065	97	48	0.009
10	1882	4388	41	0.007
Total Time				0.074
Average				0.0074

Snap Shots







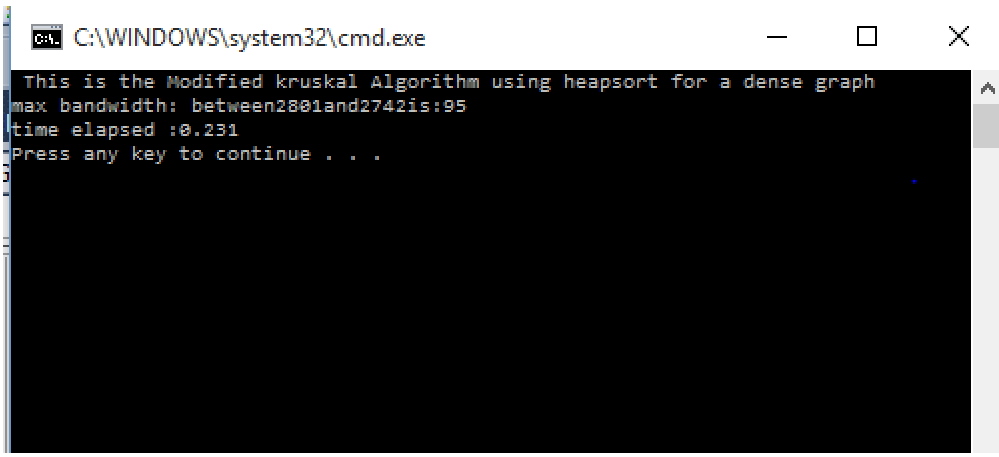
Algorithm Name: Modified Kruskal with heapsort

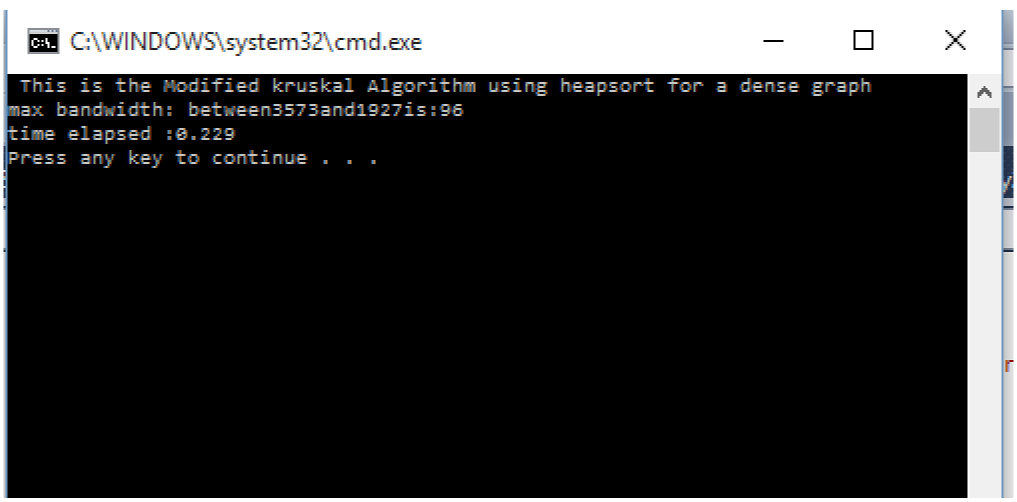
Graph Type: Dense Graph

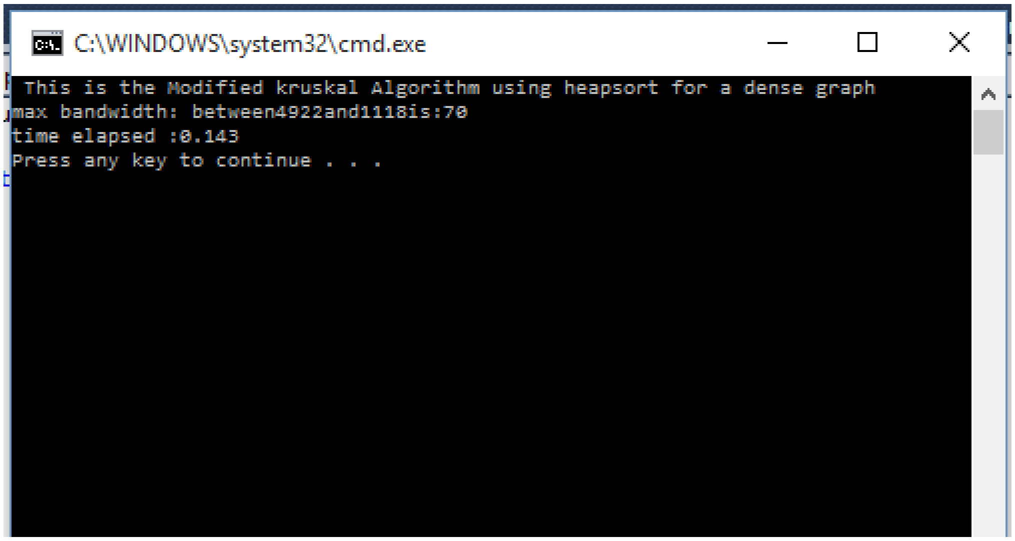
SNO	source	destination	weight	Time
1	4922	1118	70	0.143
2	3573	1927	96	0.229

3	2801	2742	95	0.231
4	1891	4146	34	0.208
5	1428	853	46	0.249
6	228	2714	82	0.185
7	4282	4503	58	0.246
8	4678	2707	42	0.142
9	196	2477	98	0.231
10	3975	3295	94	0.216
Total Time				2.08
Average				0.208

Snap Shots





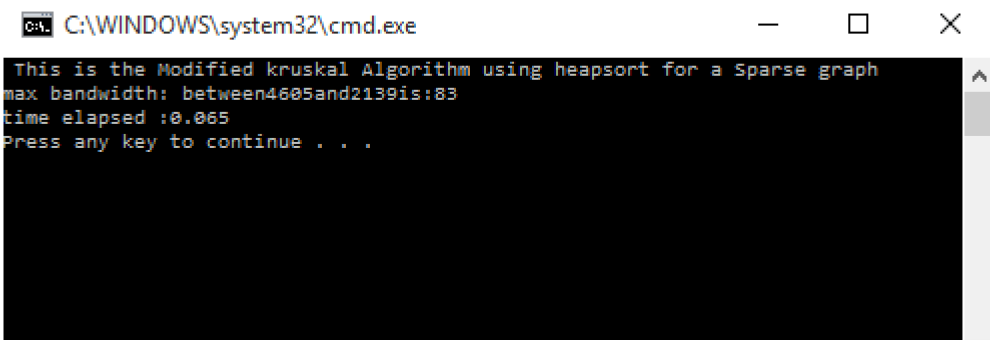


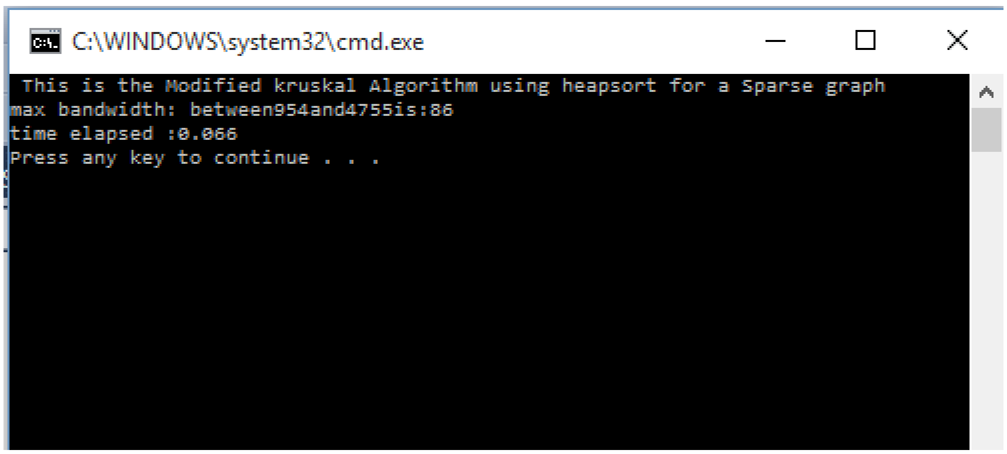
Algorithm Name: Modified Kruskal with Heap sort

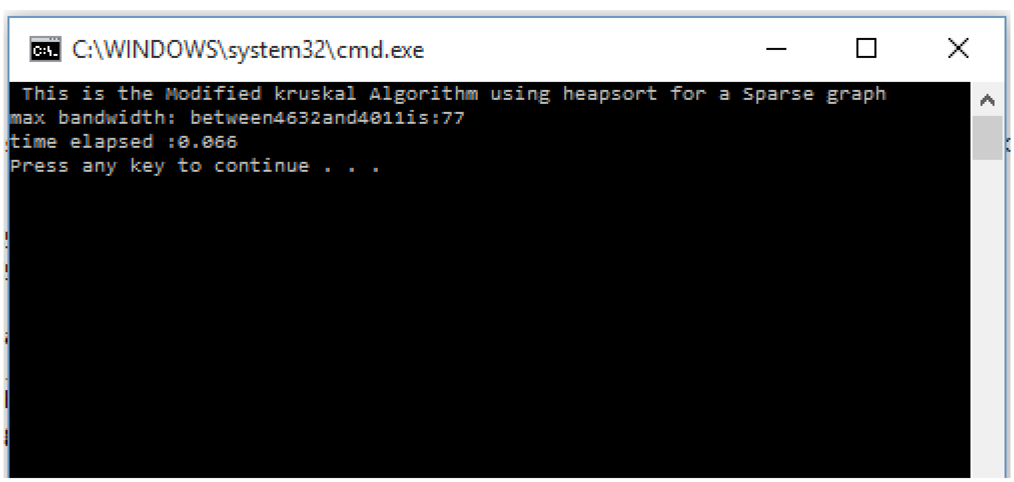
Graph Type: Sparse Graph

SNO	source	destination	weight	Time
1	4632	4011	77	0.066
2	954	4755	86	0.066
3	4605	2139	83	0.065
4	3403	2954	60	0.066
5	4658	1752	80	0.065
6	1362	1750	80	0.065
7	4546	2568	82	0.066
8	4016	3611	82	0.066
9	4111	177	59	0.065
10	4523	1744	97	0.065
Total Time				0.655
Average				0.065

Some Snap Shots


C:\WINDOWS\system32\cmd.exe
This is the Modified kruskal Algorithm using heapsort for a Sparse graph
max bandwidth: between4605and2139is:83
time elapsed :0.065
Press any key to continue . . .


C:\WINDOWS\system32\cmd.exe
This is the Modified kruskal Algorithm using heapsort for a Sparse graph
max bandwidth: between954and4755is:86
time elapsed :0.066
Press any key to continue . . .


C:\WINDOWS\system32\cmd.exe
This is the Modified kruskal Algorithm using heapsort for a Sparse graph
max bandwidth: between4632and4011is:77
time elapsed :0.066
Press any key to continue . . .