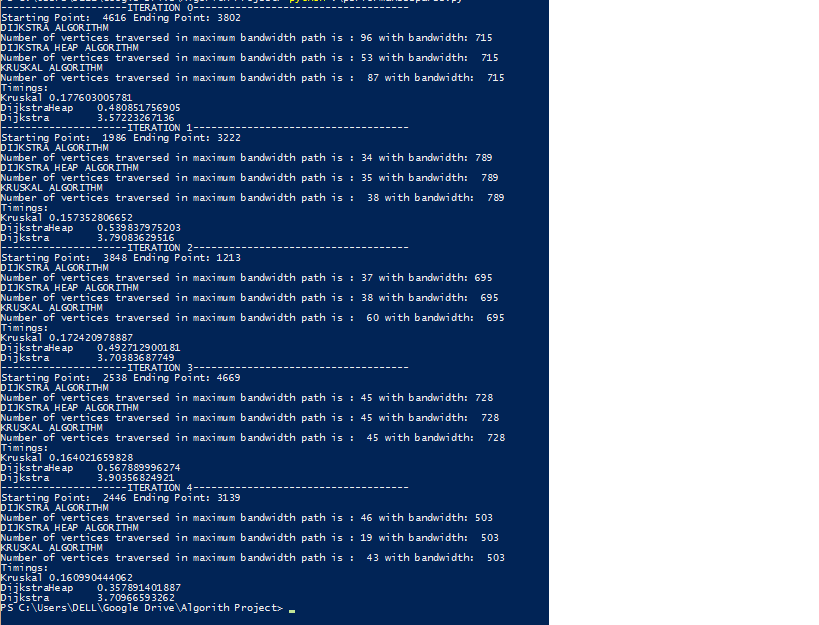
**SPARSE GRAPHS:**

Output Console:

****

1. Starting Point: 4616 Ending Point: 3802 Bandwidth: 715

Kruskal: 0.1776 sec 87 vertices covered

Dijkstra Heap: 0.4808 sec 53 vertices covered

Dijkstra: 3.5722 sec 96 vertices covered

1. Starting Point: 1986 Ending Point: 3222 Bandwidth: 789

Kruskal: 0.1573 sec 38 vertices covered

Dijkstra Heap: 0.5398 sec 35 vertices covered

Dijkstra: 3.7908 sec 34 vertices covered

1. Starting Point: 3848 Ending Point: 31213 Bandwidth: 695

Kruskal: 0.1724 sec 60 vertices covered

Dijkstra Heap: 0.4927 sec 38 vertices covered

Dijkstra: 3.7038 sec 37 vertices covered

1. Starting Point: 2538 Ending Point: 4669 Bandwidth: 728

Kruskal: 0.1640 sec 45 vertices covered

Dijkstra Heap: 0.5678 sec 45 vertices covered

Dijkstra: 3.9035 sec 45 vertices covered

1. Starting Point: 2446 Ending Point: 3139 Bandwidth: 503

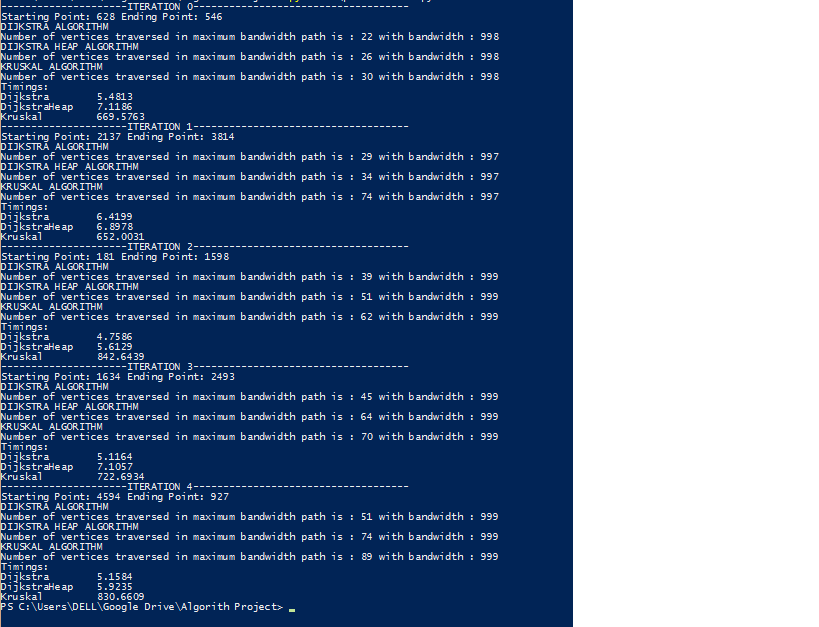
Kruskal: 0.1609 sec 43 vertices covered

Dijkstra Heap: 0.3578 sec 19 vertices covered

Dijkstra: 3.7096 sec 46 vertices covered

**DENSE GRAPH:**

Output Console:



1. Starting Point: 628 Ending Point: 546 Bandwidth: 998

Dijkstra: 5.4813 secs 22 vertices traversed

Dijkstra Heap: 7.1186 secs 26 vertices traversed

Kruskal: 669.5763 secs 30 vertices covered

1. Starting Point: 2137 Ending Point: 3814 Bandwidth: 997

Dijkstra: 6.4199 secs 29 vertices traversed

Dijkstra Heap: 6.8978 secs 34 vertices traversed

Kruskal: 652.0031 secs 74 vertices traversed

1. Starting Point: 181 Ending Point: 1598 Bandwidth: 999

Dijkstra: 4.7586 secs 39 vertices traversed

Dijkstra Heap: 5.6129 secs 51 vertices traversed

Kruskal: 842.6439 secs 62 vertices traversed

1. Starting Point: 1634 Ending Point: 2493 Bandwidth: 999

Dijkstra: 5.1164 secs 45 vertices traversed

Dijkstra Heap: 7.1057 secs 64 vertices traversed

Kruskal: 722.6934 secs 70 vertices traversed

1. Starting Point: 4594 Ending Point: 927 Bandwidth: 999

Dijkstra: 5.1584 secs 51 vertices traversed

Dijkstra Heap: 5.9235 secs 74 vertices traversed

Kruskal: 830.6609 secs 89 vertices traversed

One advantage in dijkstra is that we can stop the algorithm as soon as status of t is "in-tree" but in kruskal we cannot stop the algorithm as soon as we make t a part of MST because kruskal may have disjoint MST's at that particular point of time.

In dijkstra if we use fibonnaci heap then time will be O(V\*logV+E) therefore it will be linear time for dense graphs.

In kruskal the algorithm time is O(ElogE+E), i.e. it is dominated by time taken in sorting the edges, therefore if we have pre sorted edges then we can run kruskal algorithm in linear time.

Kruskal Algorithm is faster than dijkstra because constant c is smaller in kruskal than in dijkstra, therefore for sparse graphs kruskal is faster but for dense graphs |E| out weights and makes kruskal slower.

Dense graph with different no. of vertices and density:

200 vertices 20% connectivity-> DH>K>D

200 vertices 40%connectivity-> DH>K>D

400 vertices 20% connectivity-> DH<K<D

400 vertices 40% connectivity-> DH<D<K

600 vertices 20% connectivity-> DH>K>D

600 vertices 40% connectivity-> DH>D>K

800 vertices 20% connectivity-> DH>D>K

800 vertices 40% connectivity-> DH>D>K

5000 vertices 10% connectivity-> DH>D>K

5000 vertices 20% connectivity-> D>DH>K

Therefore, we can see that the running time of Kruskal Algorithm is inversely related to: (1) no. of vertices. (2) vertex density.

Sparse graphs with different no. of vertices:

200 vertices-> DH>K>D

400 vertices-> DH>K>D

600 vertices-> DH>K>D

800 vertices-> DH>K>D

1000 vertices-> DH>K>D

2000 vertices-> K>DH>D

3000 vertices-> K>DH>D

5000 vertices-> K>DH>D