Project Report

Introduction:

Dijkstra’s Shortest Path and Kruskal’s MST algorithms are well known and extensively studied. In this project, the algorithms are used to implement a network routing protocol which finds the maximum Bandwidth path between any 2 nodes in the network. The algorithms studied in class are implemented in C++ resulting in a deeper understanding of the concepts.

Implementation:

The graphs are implemented using C++ STL as a vector of vectors. First, a random graph is generated in the Adjacency Matrix form and then converted to the Adjacency list form to allow constant time access to neighbouring vertices. The performance of the algorithms on these graphs is analyzed.

Dijkstra’s Shortest Path Algorithm

The algorithm uses a greedy approach to find the maximum bandwidth path. At every iteration, the fringe node with the highest bandwidth is added to the tree.

a) Array Implementation

An array is used to store the best known bandwidth to any given node from s. At every iteration, selecting the fringe node with max bandwidth can be done in O(n) time where n is the number of vertices in the graph. So, each vertex changes from fringe → in_tree in O(n) time and since we have n vertices, this takes $O(n^2)$ time considering the entire execution. Every edge is considered at most twice, followed by an O(1) operation to update the status and bandwidth of the vertices. Considering the entire execution, this takes O(m) time where m is the number of edges in the graph. Since $m = O(n^2)$, the entire algorithm requires $O(n^2)$.

b) Using a Heap to store fringes

At every iteration, we need to extract the fringe node with the highest bandwidth. This can be done in O(log n) time using a heap. So, each vertex changes from fringe → in_tree in O(log n) time and considering the entire execution, this takes O(nlog n) time. Every edge is considered at most twice, followed by O(log n) insertion and deletion operations to update the status and bandwidth of the vertices. Considering the entire execution, this takes O(mlog n) time.
So, the entire algorithm requires O((m+n)log n) time.

Kruskal’s Algorithm

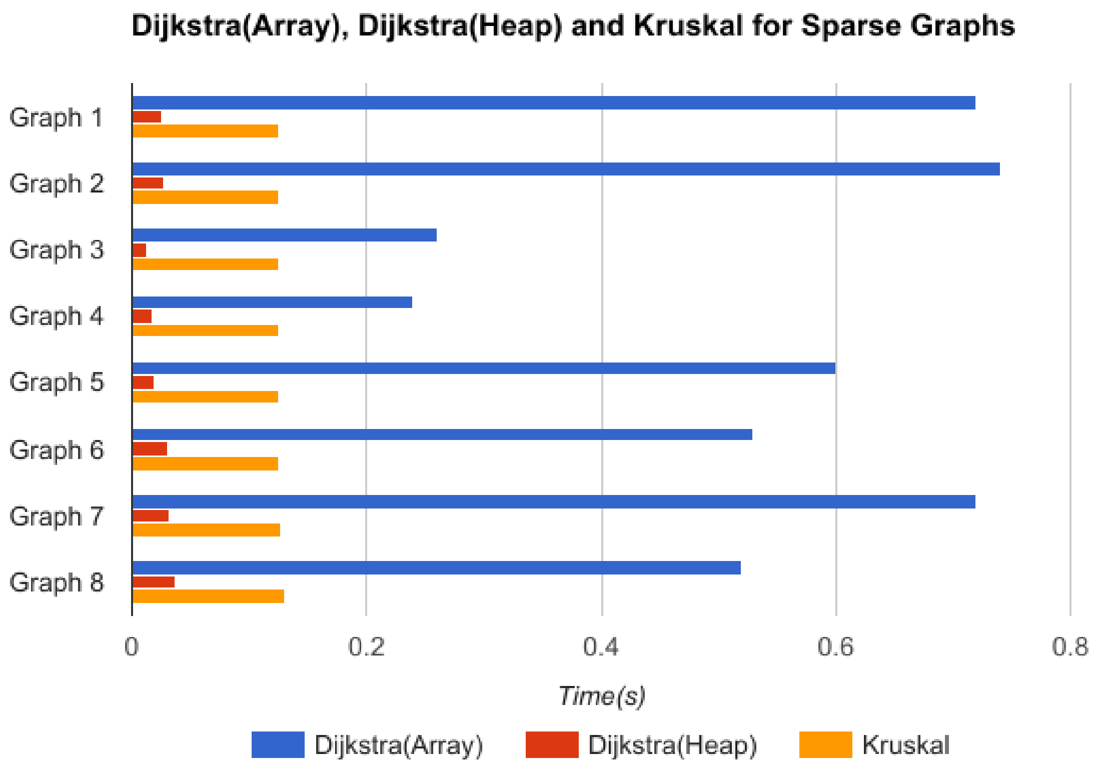
We first find the MST(Maximum Spanning Tree) for the entire graph and then use DFS to find a path from s to t in the MST. It can be proved that the path from s to t in the MST is also the maximum Bandwidth Path.
The algorithm first sorts all the edges in decreasing order using heap sort which requires O(mlogm) time. For every edge, the algorithm checks if the edge connects 2 different “pieces” and adds the edge to the MST if it does. So, for every edge, we require at most 2 find operations and 1 union operation each of which require log n time. Considering all edges, this step can be done in O(mlog n) time. So, we require O(mlog n) time to find the MST since log m = O(log n).

Then, finding the path from s to t takes O(m+n) time using DFS.

Observations and Analysis:
The random graphs considered here have n=5000 and every vertex is connected to 6 other vertices. This results in sparse graphs and the running times of the algorithms on these are compared below.

Comparison of running times for sparse graphs

	Dijkstra(Array)	Dijkstra(Heap)	Kruskal	dfs(Kruskal)
Graph 1	0.72	0.026	0.126	0.00059
Graph 2	0.74	0.028	0.125	0.00053
Graph 3	0.26	0.013	0.126	0.0015
Graph 4	0.24	0.017	0.125	0.00097
Graph 5	0.6	0.019	0.125	0.00067
Graph 6	0.53	0.031	0.126	0.0011
Graph 7	0.72	0.032	0.127	0.0027
Graph 8	0.52	0.037	0.131	0.0034



It can be seen that the performance on sparse graphs has the order as follows:

Dijkstra(Heap) > Kruskal > Dijkstra(Array)

The Dijkstra's algorithm contains two parts, the first consists of extracting the fringe vertex with the maximum bandwidth and the second part consists of updating the status and bandwidth of the vertices. These are handled differently in the Heap and Array implementations resulting in an order of magnitude difference in running times as seen in the table.

In the heap implementation of Dijkstra, the second part of the algorithm requires O(log n) time per edge. Since the graph is sparse, m = O(n) and the time required for this step is O(nlog n) resulting in an effective time bound of O(nlog n) for the algorithm.
In the array implementation, the second part requires O(1) time per edge and takes O(m) time. Since m = O(n), the first part which takes O(n^2) time dominates leaving the complexity unchanged.
For n = 5000, n^2 >> nlog n and this explains the huge difference in running times.

In the Kruskal's algorithm, sorting the edges requires O(mlog m) time and this value is small since the graph is sparse. The second part which checks if the edge connects 2 "pieces" takes O(mlogn) time, resulting in a complexity of O(mlog n) which is a small value since m is small.

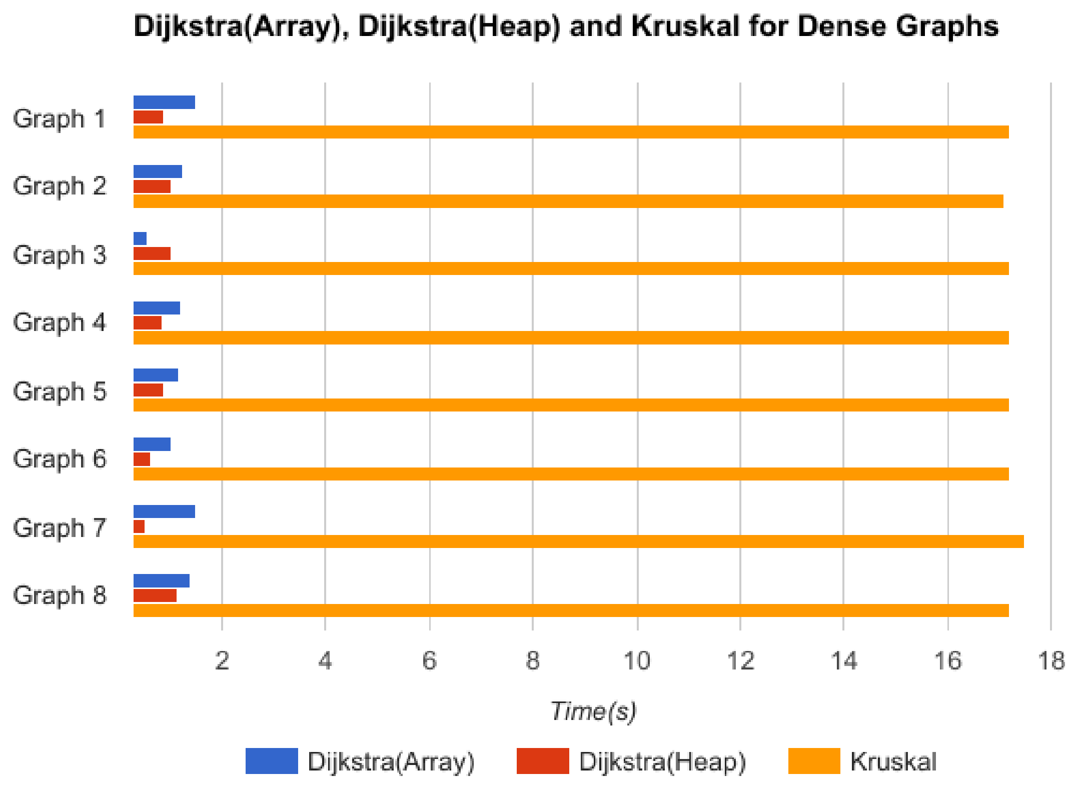
A large variation is seen in the run times of Dijkstra's algorithm since the algorithm terminates as soon as t is added to the tree. So, depending on the location of s and t in the graphs, the time taken can have a wide range of values. In contrast, the run time of Kruskal's algorithm is nearly constant. This is because the algorithm always checks all m edges to find the MST and in all the graphs above, m is roughly the same since each vertex has a degree 6. The algorithm requires 2 find operations for each of the m edges. The slight deviation is due to the nondeterministic nature of whether a union is needed or not.

The Dijkstra's algorithm must start over again for every pair of s and t vertices and roughly requires the same amount of time for every (s,t) pair. Whereas, for Kruskal's algorithm, once the MST is found, for any given (s,t) pair, finding the maximum bandwidth path only involves the run time of DFS which is negligible as seen in the table.

The random graphs considered here have n=5000 and every vertex is connected to 20% of the other vertices. This results in dense graphs and the running times of the algorithms on these are compared below.

Comparison of running times for dense graph

	Dijkstra(Array)	Dijkstra(Heap)	Kruskal	dfs(Kruskal)
Graph 1	1.5	0.86	17.2	0.0018
Graph 2	1.23	1.02	17.1	0.00018
Graph 3	0.54	1.04	17.2	0.0033
Graph 4	1.21	0.85	17.2	0.0035
Graph 5	1.17	0.87	17.2	0.0014
Graph 6	1.04	0.64	17.2	0.00099
Graph 7	1.49	0.51	17.5	0.0024
Graph 8	1.4	1.14	17.2	0.0017



It can be seen that the performance on dense graphs has the order as follows:

Dijkstra(Heap) > Dijkstra(Array) > Kruskal

In the heap implementation of Dijkstra, the second part of the algorithm which updates the heap requires O(log n) time per edge. So, this takes O(mlog n) time time and the first part which extracts the fringe with max requires O(nlog n) time. Since the graph is dense, m = O(n^2) and the first part dominates resulting in a effective complexity of O(n^2log n).
In the array implementation, the second part requires O(1) time per edge and takes O(m) time. The first part takes O(n^2) time resulting in a complexity of O(n^2) for both parts and also the entire algorithm.
For n = 5000, log n = 12 which effectively means that both implementations have a running time of O(n^2). This explains why the run times seen are comparable for dense graphs with the heap implementation performing slightly better due to the constant factors.

In the Kruskal's algorithm, sorting the edges requires O(mlog m) time and this value is significant since the graph is dense. The second part takes O(mlog n) time which is comparable to the first part. So, effectively the algorithm requires O(mlog mn) time where m is large value.

A large variation is seen in the run times of Dijkstra's algorithm while the run time of Kruskal's algorithm is constant. The reason for this is the same as seen before with sparse graphs.

For a given graph, the MST has to be found only once and as before, Kruskal's algorithm takes negligible time to find the maximum BW paths between subsequent (s,t) pairs in the same graph. Whereas, Dijkstra's algorithm takes approximately the same amount of time for every (s,t) pair.

Comparison of performance in the case of Sparse and Dense Graphs
The Heap implementation of Dijkstra's algorithm takes O(nlog n) + O(m log n) time considering the two parts separately. In the case of sparse graphs, m = O(n) resulting in a runtime of O(nlog n) while for dense graphs, m = O(n^2) and the time complexity is O(n^2log n). This explains the approximately 20 times larger run time when the graph is dense.

The Array implementation of Dijkstra's algorithm takes O(n^2) + O(m) time. In the case of sparse graphs, m = O(n) and the second component has negligible impact on the run time. For dense graphs, m = O(n^2) and the time complexity is still O(n^2) but both terms are equally significant. This why the run time for dense graphs is roughly twice that in the case of sparse graphs.

Kruskal's algorithm takes O(mlog n) time. In the case of sparse graphs, the run time is O(nlog n) since m = O(n). While for dense graphs, m = O(n^2) and run time is O(n^2log n) which explains why the run time is an order of magnitude larger in case of dense graphs.

Conclusions:
The heap implementation of Dijkstra’s algorithm has the best performance when the maximum bandwidth path between any two nodes is to be found. However, the algorithm starts all over for any new (s,t) pair even if the graph is the same. Whereas in the case of Kruskal’s algorithm, the Maximum spanning tree for the graph is only to be found once and finding the max BW paths for subsequent (s,t) pairs requires negligible time. In a practical scenario where the network remains relatively unchanged, using Kruskal’s algorithm might result in better performance than the heap implementation of Dijkstra’s algorithm.

Further improvements:
In the case of the heap implementation of Dijkstra’s algorithm, the deletion of arbitrary nodes in the heap requires that we use an array which stores the location of every element in the heap. Instead, we can avoid deleting these elements from the heap and check the status of the vertex extracted. Since, it is guaranteed that the larger value is extracted first and added to the tree, proceeding only if the vertex is not already in the tree ensures that the outdated values are not used. So, whenever a larger bandwidth path to a vertex is found, we can simply add it to the heap. The other outdated entries will be ignored due to the additional check described above.
This avoids the costly operation of keeping track of the vertices in the heap.