CSCE 629 Analysis of Algorithms
Course Project Report

Introduction:

Utilizing certain data structures and methods covered in the course, the goal of this project is to implement a network routing protocol. This project focuses on the MAX-BANDWIDTH-PATH problem, which is regarded as a significant network optimization challenge. There are several techniques to find a path that has the most bandwidth between the source and the destination, but the effectiveness of each method depends on the structures and algorithms utilized. In order to determine the best route between two vertices for this project, we employed a graph network. Additionally, we employed Dijkstra's and Kruskal's, two well-known algorithms, to find a path. To utilize a heap structure, a heap sort for edges, or not to use a heap structure at all, they were all altered and tweaked. We evaluated the performance on random weighted undirected graphs that mathematically represented a dense or sparse graph using these three alternative implementations.

Random Graph Generation:

In the given problem we are required to generate two types of graphs, sparse and dense. Each of these graphs have 5000 vertices. We use adjacency list representation of the graph for this problem. To implement adjacency list, we make
an array of linked lists. Each element of this array stores a linked list of vertices. We define a struct to define the node of this linked list, and call this "vertex". The "vertex" struct contains three fields, namely, "id of the vertex", "weight of the edge between the ith element of the array and the current vertex id" and "the pointer to the next node in the linked list". Additionally, we need to maintain an array to store all the edges of the graph. To store the edge information, we define a struct which stores a pair of integers to store the node id and a weight field to store the weight of the edge between the two nodes.

The sparse graph in our implementation must have an average degree of 6, which means that each node in the sparse graph must have 6 neighbors on average. While in case of the dense graph we ensure that any given vertex is adjacent to 20% of other vertices on average. This implies that any given vertex should have a degree of 1000 on average.

To generate the above two random graphs, we use the approach discussed in class. First of all, to ensure that the entire graph is connected, we make a cycle in the graph

by connecting the nodes from node id 0 to 4999(note that there are 5000 nodes in the graphs). After this while the average degree is less than 6 and 1000 for sparse and dense graphs respectively, we randomly pick up a node, and check while it's number of neighbors are less than 6(or 1000) and then randomly pick another node with fewer than 6(or 1000) node and make an edge between the two nodes. We also increase the total number of degrees in this process. Note that we break out of this loop once the average degree reaches 6(or 1000).

At the end of the above process we obtain randomly generated sparse and dense graphs with average degree of 6 and 1000 respectively. Note that we randomly choose an integer between 1 and 1000 to assign the weight to each of the new edges created.

Max Heap:

The maximum heap structure that will be used for Dijkstra's algorithm was implemented in the Heap class, supporting the functions of Maximum, Insert, and Delete. The implementation was closely based on the solution found from our assignment #2, but to further improve the performance of the Delete operation, we used a list to store the index of the vertices in the heap as we described in class. This position list is updated each time there is an Insert or Delete operation, where if the index of the vertex changes in the heap, then it is updated in the position list as well. This allows us to avoid inefficiently searching for a specific vertex in the heap considering we will know the exact position with this additional list.

Routing Algorithms:

The input for each of these algorithms is graph G, a source vertex s, and a destination vertex t. Each of these algorithms will then return the maximum bandwidth path from s to t.

1. Dijkstra's Algorithm(Array - Based implementation)

To implement the dijkstra algorithm we use the same idea as discussed in the course lectures. In our implementation we make three arrays namely, status, daddy and bw. The "status" array stores the status of the nodes in the graph. Unseen nodes are marked as 0, fringers as 1 and inTree nodes as 2. We use the "daddy" array to store the parent of the given node. This array stores the path from source to destination. The "bw" array stores the maximum bandwidth from source to the current node. As we are using an array to store all the fringes in the graph, we iterate through the entire array to find the fringe with the maximum bandwidth. Thus this makes the time-complexity of

Dijkstra to be O(n^2) where n is the number of nodes. We break out of the loop when we change the destination's status to inTree.

```
void Dijkstra::arrayBased(int s, int t, vertex* adj[]) {
    // Step 1: Set the status of the s node to be in-tree (2)
    status[s] = 2;

    // Step 2: For all neighbors of s, add them as fringes
    vertex* temp = adj[s];
    int nFringes = 0;
    while(temp != NULL){
        status[temp->id] = 1;
        daddy[temp->id] = s;
        bw[temp->id] = temp->weight;
        temp = temp->next;
        nFringes++;
    }

    // Step 3: for all fringes
    int v;
    while (nFringes != 0) {
        // Step 3.1:    pick the fringe v with the largest bw
        v = nodeMaxBw();
        status[v] = 2;
        nFringes--;

        // Step 3.2: For all neighbors of w
        temp = adj[v];
        while(temp != NULL){
            if (status[temp->id] == 0) {
                status[temp->id] = 1;
                bw[temp->id] = std::min(bw[v], temp->weight);
                daddy[temp->id] = v;
                nFringes++;
            }
            else if (status[temp->id] == 1 && bw[temp->id] < std::min(bw[v], temp->weight)) {
                bw[temp->id] = std::min(bw[v], temp->weight);
                daddy[temp->id] = v;
            }
            temp = temp->next;
        }

    }
    std::cout << "Maximum Bandwidth is: " << bw[t] << std::endl;
}
```

2. Dijkstra's Algorithm(Heap - Based Implementation)

We use the similar algorithm as used above but here, instead of keeping the fringes in the array and iterating over this entire array to find the maximum band-width fringer, we maintain a max-heap structure to store the fringers. As the max-heap stores the maximum element as its root, we pick the fringer present on the root of this heap in our algorithm. In addition to the three arrays described above for the array-based implementation, we also initialize a heap created from our maxheap class as described above. By using a max-heap we are able to return the maximum element in O(1) time. To achieve this, we will use the Maximum function from the heap and the Delete function on the maximum node, which has been proven to take O(log n) based on the height of the tree. Finally, our two conditions consist of an unseen status and a fringe status. If a neighbor node is still unseen, then we will insert the fringe into the heap. However, if the second condition is true, then we will delete this vertex from the heap, update the bandwidth and parent arrays and insert this new vertex back into the heap. The operations for deleting and inserting into a heap take O(log n) as mentioned previously and finding the maximum O(1), which will result in overall complexity of this version of Dijkstra to be O(m log n), where m represents the edges of the graph and n the number of vertices.

```
1  void Dijkstra::heapBased(int s, int t,  vertex *adj[]) {
2
3      // Step 1 Create heap
4      DijkstraHeap heap;
5
6      // Step 2: Set the status of the s node to be in-tree (2)
7      status[s] = 2;
8
9      // Step 3: For all neighbors of s
10     int nFringes = 0;
11     vertex* temp = adj[s];
12     while(temp != NULL) {
13         status[temp->id] = 1;
14         daddy[temp->id] = s;
15         bw[temp->id] = temp->weight;
16
17         heap.insertNode(temp->id, temp->weight);
18
19         temp = temp->next;
20         nFringes++;
21     }
22
```

```
    // Step 4: for all fringes
    int v;
    while (nFringes != 0) {

        // Step 4.1: pick the fringe v of the largest bw
        v = heap.nodeID[0];
        // Step 4.2:  delete this node from the heap
        heap.deleteNode(v);

        status[v] = 2;
        nFringes--;

        // Step 4.2: For all neighbors of w
        temp = adj[v];
        while(temp != NULL) {
            if (status[temp->id] == 0) {
                status[temp->id] = 1;
                bw[temp->id] = std::min(bw[v], temp->weight);
                daddy[temp->id] = v;
                nFringes++;
                heap.insertNode(temp->id, bw[temp->id]);
            }
            else if (status[temp->id] == 1 && bw[temp->id] < std::min(bw[v], temp->weight)) {
                bw[temp->id] = std::min(bw[v], temp->weight);
                daddy[temp->id] = v;
                heap.deleteNode(temp->id);
                heap.insertNode(temp->id, bw[temp->id]);
            }
            temp = temp->next;
        }

    }
    std::cout << "Maximum Bandwidth is: " << bw[t] << std::endl;
}
```

3. Kruskal's Algorithm(Using Heap Sort)

We implement kruskal's algorithm using the same algorithm as discussed in the class lectures. However, we do take advantage of the heap sort. We use the Kruskal algorithm to find the maximum spanning tree and then use the breadth first search to find the path between the source(s) and target(t). To achieve this we first use heapsort to sort all the edges present in the graph and implement the makeset, find and union functions as discussed in the class lectures. These functions allow us to build a new graph that contains only the edges that pertain to the maximum spanning tree. This graph is then passed to the path function which uses BFS to find the path along s to t. The returned path will be the maximum bandwidth path from s to t.

```cpp
void Kruskal::maxBandwidth(vertex* adj[], int s, int t) {
    // Step 1: Sort the edges
    HeapSort(adj);

    // Step 2: Make Set
    for (size_t i = 0; i < nNodes; i++)
        makeSet(i);

    // Step 3: For all edges
    int pos = 0;
    int e;
    int r1, r2;
    for (int i = 0; i < numEdges; i++) {
        // Step 3.1: Pick the edge with the largest weight
        e = maxHeap->sortedArray[pos].weight;

        // Step 3.2:  Find the roots of the two vertices connected using e
        r1 = find(maxHeap->sortedArray[pos].nodes.first);
        r2 = find(maxHeap->sortedArray[pos].nodes.second);

        // Step 3.3: Check if the nodes are in the same tree, if not => connect
        if (r1 != r2) {
            unionFuc(r1, r2);
            addPair(maxHeap->sortedArray[pos].nodes.first,
                    maxHeap->sortedArray[pos].nodes.second,
                    maxHeap->sortedArray[pos].weight);
        }
        pos++;
    }

    // Step 4: Find the maximum Bandwidth and the path
    findMaxAndPath(s, t);
}
```

```cpp
// === Functions for spanning Tree
void Kruskal::findMaxAndPath(int s, int t) {
    // Initialize the values to zero
    for (size_t i = 0; i < nNodes; i++) {
        color[i] = 0;
        daddy[i] = -1;
    }

    // BFS on the spanning tree

    bfs(s);
    //return;
    int min = 100000000;

    while (daddy[t] != -1) {
        if (min > capacityInTree[t])
            min = capacityInTree[t];
        t = daddy[t];

    }
    std::cout << "\nMaximum Bandwidth is: " << min;

}
```

Performance Analysis:

The results of our evaluation tests for each of these algorithms on sparse and dense graphs are shown in the tables below with their overall average running time.

Sparse Graph Results					
	Running Time				
	Source	**Destination**	**Dijkstra w/o Heap**	**Dijkstra w Heap**	**Kruskal w Heap Sort**
Sparse Graph 1	972	3389	0.095	0.021	0.027
	2347	4533	0.096	0.021	0.026
	3324	1715	0.091	0.018	0.023
	1276	2273	0.085	0.017	0.025
	3387	4498	0.078	0.019	0.028
Sparse Graph 2	3324	27	0.091	0.020	0.026
	1566	2278	0.087	0.019	0.025
	3231	485	0.088	0.020	0.027
	217	973	0.092	0.016	0.026
	4008	132	0.083	0.015	0.024
Sparse Graph 3	667	3448	0.079	0.015	0.026
	776	234	0.089	0.018	0.027
	3356	4342	0.096	0.016	0.027
	2290	2213	0.092	0.018	0.028
	2119	4987	0.084	0.020	0.029
Sparse Graph 4	2961	1765	0.081	0.018	0.026
	3908	1074	0.092	0.016	0.027
	388	1264	0.093	0.017	0.024
	4289	3293	0.089	0.016	0.022

	2187	2219	0.086	0.018	0.029
Sparse Graph 5	1649	2936	0.097	0.019	0.028
	3142	1799	0.087	0.016	0.025
	211	2769	0.083	0.018	0.028
	4765	342	0.098	0.020	0.028
	2179	32	0.079	0.016	0.024
Average Time			0.088	0.018	0.026

Dense Graph Results					
	Running Time				
	Source	Destination	Dijkstra w/o Heap	Dijkstra w Heap	Kruskal w Heap Sort
Dense Graph 1	1892	2373	0.432	0.268	3.135
	23	476	0.390	0.283	3.160
	2115	4338	0.414	0.275	3.086
	2936	3257	0.408	0.287	2.908
	2736	215	0.389	0.255	2.880
Dense Graph 2	904	3115	0.443	0.289	3.184
	4336	326	0.419	0.272	3.174
	3278	2156	0.398	0.244	3.104
	2146	2474	0.387	0.225	2.856
	12	533	0.435	0.266	3.098
Dense	3422	3783	0.431	0.286	2.756
	225	3346	0.448	0.301	3.215

Graph 3	766	2254	0.408	0.284	3.018
	3284	2763	0.387	0.234	2.985
	1758	4562	0.398	0.213	3.127
Dense Graph 4	766	2776	0.387	0.267	3.097
	2783	118	0.394	0.253	2.783
	3419	67	0.367	0.238	2.564
	2793	1563	0.349	0.245	2.740
	1936	1390	0.403	0.276	3.150
Dense Graph 5	689	459	0.390	0.275	3.085
	4753	3982	0.367	0.254	2.884
	2958	2746	0.417	0.283	2.976
	1846	1743	0.405	0.267	3.094
	4256	2745	0.387	0.259	2.786
Average Time			0.402	0.264	2.994

For the sparse graphs, we observe the following relation in the overall average running time:

(Faster) Dijkstra with Heap > Kruskal > Dijkstra without Heap (Slower)

The most time-consuming algorithm for sparse graphs is Dijkstra without the use of the heap structure, which as explained in the implementation, the overall complexity would be O(n2). This is due to having to iterate through the entire array of vertices every time it needs to find the maximum fringe. Therefore, it performs as expected. As was proven in class, Kruskal's algorithm with the use of Union-Find operations and Dijkstra's algorithm with the help of a heap structure would both take an overall complexity time of O(m log n). This is based on the notion that the Find operation would take O(log n) time and the Insert and Delete operations would also take O(log n). However, one additional modification that was added to Kruskal's algorithm was the HeapSort function, which requires to sort through all the edges of the graph. In addition, after building the maximum spanning tree, the algorithm has to run a DFS algorithm to provide the path between the two vertices. Therefore, the performance difference would be due to the

constants in the Big-O complexity. Although there could be a number of insertion and deletion operations, the amount of time it will take to sort through all edges in the graph and also perform DFS on all the edges in the new graph increase the constant value of overall time complexity, and therefore explain the reason it performs slower than Dijkstra's implementation with a heap.

For the dense graphs, we observe the following relation in the overall average running time:

(Faster) Dijkstra with Heap > Dijkstra without Heap > Kruskal (Slower)

Similar to the performance in sparse graphs, Dijkstra's algorithm with the use of the heap structure takes the least amount of time to return the maximum bandwidth path. This demonstrates the usefulness of this algorithm in both types of graphs and is able to hold the overall complexity time of O(m log n) despite the number of edges being greater than the number of vertices. On the other end of the spectrum, Kruskal's algorithm obtains the worst performance in dense graphs as it performs 16 times slower than Dijkstra, which is not a surprising observation. In our implementation, we had to use HeapSort to sort through the edges and therefore on a dense graph, this will be highly inefficient and dependent on the number of edges. In a dense graph, the number of edges will be greater than the number of vertices and the probability for two vertices to be connected to 20% of its neighbors is similar to having a vertex degree of 1000. As a result, the total number of edges would be at least over a million edges and having to sort through all these edges would be time consuming. This explains the terrible performance of Kruskal and a possible flaw in the implementation that could be improved.

Conclusion and Future Improvements

According to our performance investigation, Kruskal's method performs the poorest in dense graphs, and the majority of the time is spent on HeapSort. Using a different sorting method may help with this part of the implementation. One of the primary problems is that Kruskal's performance depends on the number of edges, therefore the more edges there are, the longer this implementation takes to complete because it must sort through all the edges in the graph. The implementation could perform better if a different sorting method could sort over the edges more quickly. Although it is debatable how much could be improved, it would be helpful even if it were reduced to a level that was close to Dijkstra's various implementations.

In conclusion, we were able to solve the network optimization problem of determining the maximum bandwidth path overall by successfully implementing three distinct methods. We then tested the effectiveness of each of these methods on sparse and dense graphs. The outcomes showed that the heap-based Dijkstra implementation performed better for both sparse and dense graphs since it required the least amount of processing time.