CSCE 629 Project Report

Nicolas Bain

1 Introduction

In the modern computing world, network optimization has poised itself as one of the challenges that computer scientists must tackle. One way to tackle this issue is by utilizing algorithms that find the maximum bandwidth path in a network. To test such algorithms, computer scientists often represent the network by using a graph structure where the edges between nodes represent the capacity of the links between nodes. In a network, communication between two selected nodes is limited by the edge that has the smallest link capacity. Maximum bandwidth algorithms, as the name suggests, seek to optimize selection of the pathway between two selected nodes in order to maximize to total bandwidth of the bath.

2 Assignment

For this project, we were tasked with implementing three maximum bandwidth algorithms: Dijkstra's algorithm, Dijkstra's algorithm using a heap structure for the fringes, and Kruskal's algorithm. To accomplish this task, I proceeded through the following steps; first I created my own graph structure and heap functions for use in the algorithms. Next I implemented the modified version of Dijkstra's algorithm for finding the maximum bandwidth path. I then made a second version of this algorithm that utilized my heap structure for the fringe list. The last algorithm I implemented was Kruskal's. After implementing each algorithm, I tested them by utilizing a sample graph [1].

I compared the results of my algorithm implementations with the known solution to the sample graph and found that they all worked correctly. From here I generated five pairs of random graphs and ran them through the algorithms, each with 5 different pairs of source/target nodes. The results of these trials are discussed later. The rest of the report details each of the steps mentioned above and proceeds to compare the results of the trials.

3 Implementation

As I mentioned above, my first step was deciding how to implement the graph and heap components. For a graph, there are two common representations: an adjacency matrix, and an adjacency list. The pros and cons of both representations are shown below in table 1.

Adjacency Matrix	Adjacency List
Pros: determine if there is an edge between vertices in O(1) time. Easier to implement and follow. Removing an edge takes O(1) time.	**Pros:** Saves space: O(|V| + |E|). Adding a vertex is easier. Good for listing adjacent vertices.
Cons: Consumes more space $O(V^2)$. Adding vertex takes $O(V^2)$ time. No efficient way to list out adjacent vertices.	**Cons:** Determining if there is an edge between two vertices takes O(V) time.

Table 1 Graph Implementation Comparison

Due to the fact that Dijkstra's algorithm relies on looking up adjacent edges, I decided to use the adjacency list data structure. I also included a vector of edges (edge list) that kept track of all edges, their vertices, and their weights for use in Kruskal's algorithm. This list was used to make a heap, which was then sorted and utilized in the algorithm. To keep track of what edges were in the maximum spanning tree, I utilized another vector of edges. At the end of the algorithm I translate this solution edge list into a map structure and perform a DFS to output the maximum bandwidth path. Detailed implementation information for each algorithm is listed below.

3.1 Dijkstra Without Heap

This algorithm is a direct implementation of the pseudo code given in class and performs as follows:

(WHITE = UNSEEN, GREY = FRINGE, BLACK = INTREE. Done this way to homogenize visited status between Dijkstra and Kruskal algorithms.)

Dijkstra(G, s, t):

1. For v=1 to n:
   visited[v] = WHITE
   cap[v] = 0
   parent[v] =v

2. visited[s] = BLACK
3. For each edge [s, w]:
   visited[w] = GREY
   cap[w] = wt[s, w]
   parent[w] = s

4. While # fringes != 0: //I utilized an counter to determine if all nodes were in tree yet
   v = max_capacity_in_fringe()// loop through all nodes and find GREY node with max weight. // O(v) time
   visited[v] = BLACK

```
For each edge [v, w]:
        if visited[w] = WHITE:
                visited[w] = GREY
                cap[w] = min(cap[v], wt[v, w])
                parent[w] = v
        else if visited[w] = GREY && cap[w] < min(cap[v], wt[v,w]) :
                cap[w] = min(cap[v], wt[v,w]
                parent[w] = v
```

Analysis shows that the outer loop runs in V times and finding the maximum capacity edge takes O(V) time. Total runtime is therefore $O(V^2)$.

3.2 Dijkstra With Heap

The bulk of the algorithm remains the same except for the use of a maximum heap for the fringe list. To represent a heap, I utilized a vector of integers to hold vertex "names" and an array to hold the vertex positions. I also implemented MAXIMUM, INSERT, and DELETE subroutines for the heap. The pseudo code for each is listed below the code for Dijkstra's algorithm.

```
DijkstraHeap(G, s, t):
1. For v=1 to n:
        visited[v] = WHITE
        cap[v] = 0
        parent[v] =v

2. visited[s] = BLACK
3. For each edge [s, w]:
        visited[w] = GREY
        cap[w] = wt[s, w]
        parent[w] = s
        fringe_flag = 1; // flag to determine if fringe is empty or not. 0 means empty.

4. HeapBuild(cap) //build heap out of capacity array
5. While # fringes != 0: //I utilized the fringe flag for this
        v = heapMax()// extract max out of heap // O(log(V))
        visited[v] = BLACK


        For each edge [v, w]:
                if visited[w] = WHITE:
                        visited[w] = GREY
                        cap[w] = min(cap[v], wt[v, w])
```

```
                    parent[w] = v
                    heapInsert(cap, v) // O(log(v)
            else if visited[w] = GREY && cap[w] < min(cap[v], wt[v,w]) :
                    cap[w] = min(cap[v], wt[v,w]
                    parent[w] = v
                    if(position of w in heap != 0)
                        heapDelete(cap, w) // O(log(v)
                        heapInsert(cap, w) // O(log(v)
```

heapBuild(cap[])

1. For v=1 to n
   Insert into heap vector
   Insert into position array
   Heapify(v)
2. Loop through heap and remove 0 values.

heapify(cap[], n)

1. l = 2*n
   r = 2*n + 1
2. determine largest value (i, l, r)
3. if(largest != i)
   swap _positions(i, largest)
   swap_values(i, largest)
   heapify(position, largest)

heapInsert(cap[], n)

1. Place n at back of heap and set position
   h = index of n in heap
2. While(h > 1 && cap[h] > cap[h/2]) :
   swap_positions(h, h/2)
   swap_values(h, h/2)
   h = h/2 // parent index

heapDelete(cap[], h)

1. Set position of h to last spot in position array
2. Set value of h in heap to last value in heap
3. if(h > 1 && cap[h] > cap[h/2]):
   while(h > 1 && cap[h] > cap[h/2])
   swap_positions(h, h/2)
   swap_values(h, h/2)
   h = h/2 // parent index
4. else:
   l = 2*n
   r = 2*n + 1
   deterimine largest value (h, l, r)
   while(largest != h) :

swap _positions(h, largest)
swap_values(h, largest)
h = largest
l = 2*largest
r = 2*largest + 1
determine largest value (h, l, r)

heapMax(cap[])
1. return H[1]

Analysis from homework 1, problem 4 shows that heaps take at most log(n) steps. Therefore all heap operations run in O(log(n)) time. Step 1 of Dijkstra's algorithm runs in O(V) time, step 2 in O(1), step 3 in O(V), step 4 in O(log(V)), step 5 runs O(E) times with inner functions taking O(log(v)) time for a grand total of O(Elog(V)) time complexity.

3.3 Kruskal's Algorithm

Due to the fact that I included an edge list in my graph structure, setting up Kruskal's algorithm was pretty straightforward. I simply built a heap out of the edge list and sorted it using a class made to compare edge weights. Solution edges were kept in a vector of kedges and translated into an integer/vector of integer mapping for a DFS, which outputs the path from a source to a target node.

Kruskal(G, s, t)
1. make_heap(g->EdgeList) // std::make_heap
   sort_heap(g->EdgeList)
2. For v = 1 to n
   parent[v] = 0
   rank[v] = 0
3. For each edge e
   u = e.u
   v = e.v
   if(r1 = kfind(u) != r2 = kfind(v))
   insert edge into solution vector
   kunion(r1, r2)
4. convert solution edge list into adj mapping
5. DFS(adj, s, t)

DFS(adj, s, t)
1. For v = 1 to adj.size()
   visited[v] = WHITE
2. Flag = 0;
3. dfsMain(adj, s, t)

dfsMain(adj, s, t, visited[])

1. visited[s] = GREY
2. if(s = t)
        flag = 1
        visited[s] = BLACK
        return
3. for each v adjacent to s
        if(visited[v] = WHITE)
            dfsMain(adj, v, t, visited[])
        if(flag = 1)
            visited[s] = BLACK
            return
4. visited[s] = BLACK

kfind and kunion functions were left out since they are extremely primitive and provided in class. To analyze the algorithm we first look at the heap functions. Both take O(log(E)) time. The union operation takes O(1) time whereas the find operation takes O(log(v)). Since the algorithm loops through all edges this gives us O(Elog(V) + log(E)) which is O(Elog(V)). DFS searches run in O(E+V) time since it iterates over all vertices and edges.

3.4 Graph Generation

To generate a graph I started by connecting each node to the next in the array. This ensures that there is a path between all vertices. After that I loop through each vertex DEGREE number of times and create a unique edge to a vertex that doesn't already have more than DEGREE edges. Each time an edge is formed, a random weight is assigned to it. I also made functions to print and read in graphs for testing purposes. This allowed me to run several s,t pairs on each generated graph during analysis and also allowed me to upload a test graph with a known solution to validate my algorithms.

4 Validation

To validate my algorithm implementations, I used a sample graph with a known solution [1]. The graph is shown below:

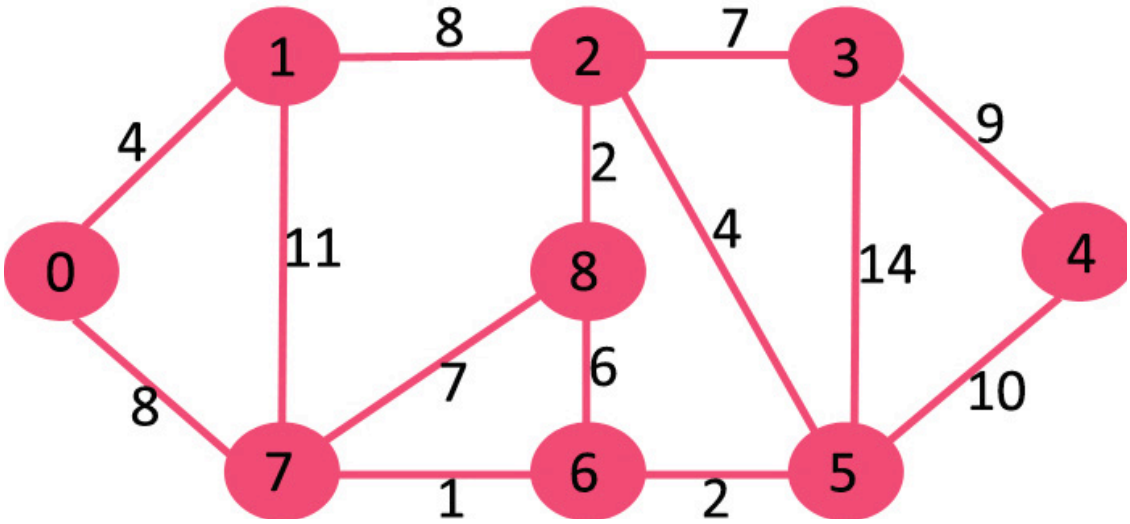

Figure 1 Sample Graph

I ran my algorithms on this graph with several s, t pairs and each time the solution was correct. The only thing of note is that sometimes Kruskal's algorithm would output a longer path. For example, if we select 0 and 4 as the s,t pair: Kruskal's algorithm outputs 0 → 7 → 1 → 2 → 3 → 5 → 4 whereas Dijkstra's (both forms) outputs 0 → 7 → 1 → 2 → 3 → 4. Regardless, both paths are correct since the path bandwidth is limited by the 2 → 3 edge.

5 Results

To analyze the algorithms I used a system timer for each algorithm and ran them on 5 pairs of graphs (one sparse, one dense) with 5 different pairs of s,t vertices. The recorded values were then averaged and graphed below. We can see from the results that my Kruskal implementation is significantly faster than the two Dijkstra implementations with the heap version being the slowest. This shouldn't be the case and I believe the problem lies somewhere in my heap implementation. Otherwise, the results make sense. All results were gathered on a Mac OS X system with a 1.7 GHz intel core i7 processor.

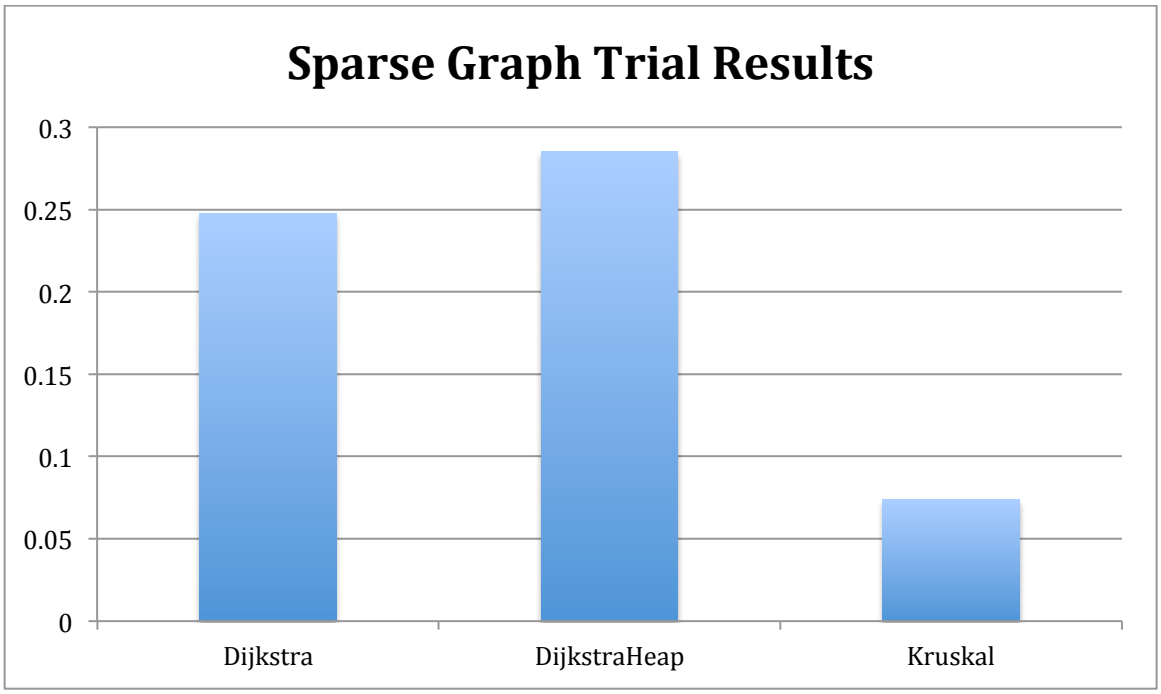

Figure 2 Sparse Graph Results

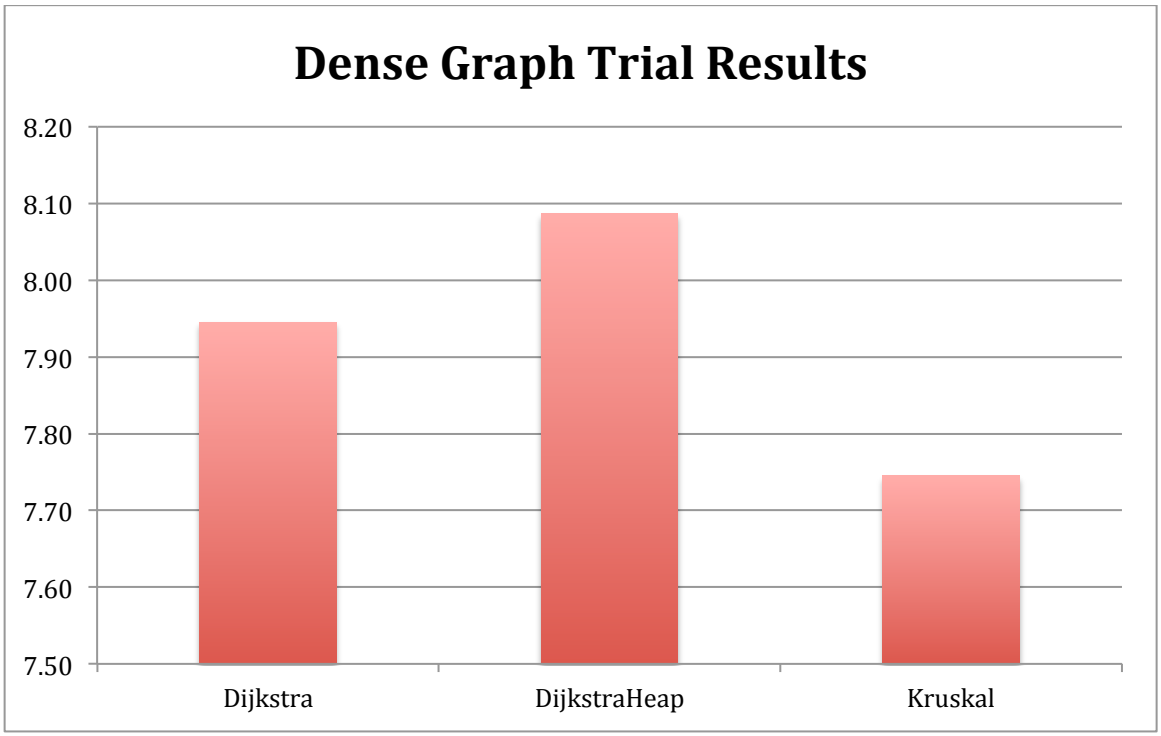

Figure 3 Dense Graph Results