Anuska Pant

CSCE 629 Course Project for Analysis of Algorithms

The GitHub repository URL is https://github.com/ANUSKAPANT/CSCE-629-Network-Optimization.

1. Random Graph Generation

We are required to implement two kinds of graphs:

1. Graph #1, where the average vertex degree is 6.
2. Graph #2, where each vertex is adjacent to about 20% of the other randomly chosen vertices

And Each of the graphs have a vertex size of 5000.

Data Structure

I have implemented 2 classes for random graph generation that are Node and Graph.
class Node stores the value vertex, weight and next pointer which allows for the implementation of adjacency list.

```
class Node {
 public:
   int vertex;
   int weight;
   Node *next;
   Node() {}
   Node(int v, int e) : vertex(v), weight(e), next(NULL) {}
};
```

The second class Graph

```
class Graph {
 public:
   int n;
   int degree;
   int percent;
   Node **edges;
   Graph(int num) : n(num), edges(new Node *[num]) {}
   Graph(int num, int deg, int prob) : n(num), degree(deg), percent(prob), edges(new
Node *[num]) {}
   void init();
   void generateGraph1();
   void generateGraph2();
   void addEdge(int u, int v, int wt);
   bool isEdge(int s, int t);
};
```

In our class graph we have member variables such as n(number of vertices), degree which is used to generate graph 1, percent which is used to define how dense the graph is. I have used an adjacency list of edges of the type node. Below is my ways to generate these two kinds of graphs:

```
void Graph::generateGraph1()
{
 int deg[5000] = {0};
 init();

 for (int i = 0; i < n; i++) {
   int j = (i + 1) % n;
   addEdge(i, j);
   deg[i]++;
   deg[j]++;
 };

 for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
     if (deg[i] == degree) break;
     int x = rand() % n;
     if (!isEdge(i, x) && deg[i] < degree && deg[x] < degree) {
       addEdge(i, x);
       deg[i]++;
       deg[x]++;
     }
   }
 }
}
```

For Graph #1,

1. The array deg is used to keep track of the degree of the vertex. We start by assigning edges to adjacent vertices and forming a circle causing each vertex to have a degree of 2. This is done to ensure that all the vertices are connected.
2. The next step is to simply add edges to all the vertices until it's equal to 6.
3. The vertices to add to the edge are selected at random using the rand() function.

For Graph #2,

Like in G1 we link all the 5000 vertices one by one to ensure that all the vertices are connected. We create a random value using rand() % 100 which generates a number between 0-99. To have 20% neighbors means around 1000 edges and if we add around 500 edges it will lead to around 1000 edges since other vertices might choose the edge as well. This causes the number of edges for each vertex to be in between 900 to 1100, i.e around 20%.

```
void Graph::generateGraph2()
{
 init();
 for (int i = 0; i < n; i++) {
   int j = (i + 1) % n;
   addEdge(i, j);
 };

 for (int i = 0; i < n; i++) {
   int j = 0;
   int num = (percent*n*0.5 - 50) + rand() % 100;
   while(j < num) {
     int x = rand() % n;
     if (!isEdge(i, x) && i != x) {
       addEdge(i, x);
       j++;
     }
     else j--;
   }
 }
}
```

2. Heap Structure

The heap structure is implemented in file heap.h where I have a class Heap containing array H, D and P to store the vertex, weight of the vertex and position of the vertex as required. It also contains member functions to aid in insert, delete and finding the maximum value maintaining the max heap property which is used in dijkstra's implementation with heap. The function heapify and heapsort are used to sort the edges in Kruskal's algorithm.
Swap, insert shiftUp and shiftDown, empty functions are also available. All of these work together to build a max heap. So each time we insert into heap it calls the shiftUp and each time we delete a node it calls the shiftDown function that maintains the max heap property. Since we need to keep track of the index of each vertex when we need to delete from the heap we make use of the array P.

3. Routing Algorithms

In this project, we implement 3 different solutions to this Max-Bandwidth-Path problem.

Dijkstra's without Heap

Here we first initialize the status, bandwidth and dad of each vertex. Then we make the start vertex s IN_TREE and assign its maxbw value to infinite. For all neighboring vertices of s stored in G.edges we make them fringes and assign dad and maxbw value to them. While the status of t isn't IN_TREE we find the maximum value fringe by looping through all the vertices that have a status fringe and proceed with the algorithm of adding the maximum bandwidth value fringe to the tree. **maxbw[t]** gives the maximum bandwidth value of the path. It's complexity is 0(n*n). It's implemented in file dijkstra.cpp.

Dijkstra's with Heap

The implementation of this is similar to the last one but the difference being we store the fringes in the heap using the insert function provided by the heap. To find the fringe with the maximum bandwidth value we use the max function from the heap. The delete is used to delete the vertex from the heap of fringes when it becomes IN_TREE. The value of maxbw is provided by maxbw[t] and we can trace the path using the dad array.
It just uses Maximum heap to speed up the max() step. For this approach, max() , insert() and delete() are all in O(log n) time. Therefore, the overall time complexity is O((m+n) * log n).
It's implemented in file dijkstra.cpp.

Kruskal's Maximum Spanning Tree

This algorithm uses Kruskal's to get the maximum spanning tree, then call BFS on this maximum spanning tree to find a path. We make use of makeset find and union to build the tree.
For the procedure of getting a maximum spanning tree, sorting step takes O(m * log m) and the time complexity of doing the sequence of Union-Find operations is O(m * log*(n)), so the time complexity is O(m * (log(m) + log*(n))).
For the procedure of calling BFS on the maximum spanning tree, it only takes O(n) time.
Therefore, the overall time complexity is O(m * (log(m) + log*(n)) + n). It's implemented in the file kruskal.cpp.

4. Testing and Performance

I built 5 pairs of graphs using the approaches in the Random Graph Generation Section, randomly selected 5 pairs of vertices and called the 3 algorithms in the Routing Algorithms Section. The result was collected in the file output.txt.
One of the examples of using the 3 routing algorithms on a pair of source and destination vertex is as follows.

```
---
Pair #2
Source=4973 ------ Destination=4106

****** DIJKSTRA ******

Maximum Bandwidth Path:
4973->4126->1539->2558->977->621->2659->2012->2273->2646->3745->671->3541->2672->3338->3097->857->2322->892->2870->3303->1087->2570->2144->2
333->2961->500->2777->3545->3117->312->2416->904->1517->837->2255->2801->58->1682->1681->4106

Max Bandwidth: 4088
Time: 0.276587

***** DIJKSTRA HEAP *****

Maximum Bandwidth Path:
4973->4126->1539->2558->977->621->2659->2012->2273->1312->4338->2660->4096->633->1407->4425->3011->1535->1780->3110->2881->170->1426->1523->
426->167->1688->2655->3460->687->58->1682->1681->4106

Max Bandwidth: 4088
Time: 0.264621

********* KRUSKAL ********

Maximum Bandwidth Path:
4973 ->4126 ->1539 ->4895 ->4430 ->1931 ->2578 ->2075 ->72 ->4690 ->3543 ->816 ->1720 ->4676 ->4227 ->928 ->1666 ->929 ->3596 ->585 ->4664 -
>4310 ->3484 ->2185 ->1640 ->3823 ->4751 ->725 ->2362 ->2255 ->837 ->1517 ->904 ->2416 ->312 ->3117 ->2951 ->2876 ->4788 ->4043 ->4156 ->106
6 ->3460 ->687 ->58 ->1682 ->1681 ->4106

Max Bandwidth: 4088
Time: 0.857868
```

It shows each algorithm's maximum bandwidth path, time taken and maximum bandwidth value.

Analysis

We can observe that Dijkstra Implementation with Heap is faster than that without Heap and this can also be seen in the output table below. This is because insertion and deletion in Heap takes O(logn) time. In non-heap implementation, it takes O(n) time to find the fringe vertex with the maximum bandwidth. Therefore, the performance of Dijkstra with heap is better than Dijkstra without heap. This result holds true for graph #1 and graph#2. Kruskal's Algorithm Maximum Spanning Tree build time is O(mlogn) as per the theory. Since building an MST involves many operations like finding the edges in decreasing order and creating a tree, we can see that the time to build a MST is larger for dense graphs. For sparse graphs Dijsktra's without heap takes the most time while in dense graphs Kruskal's takes the longest time.

Time Comparison of Graph #1 in ms

GRAPH #1	SOURCE	DESTINATION	DIJKSTRA	DIJKSTRA HEAP	KRUSKAL
1.	2626	1458	88.393	2.139	4.069
	2986	4586	87.183	2.084	4.303
	3767	2812	69.029	1.661	4.138
	3640	4783	81.239	1.751	3.95
	3989	1652	85.306	1.85	4.058
2.	4462	4176	77.459	1.72	4.158
	1457	66	89.113	2.067	4.153
	3533	3831	89.407	1.974	4.152
	585	4101	89.039	2.099	3.94
	3196	1369	70.451	1.708	4.179
3.	3666	3377	62.852	1.479	3.901
	4568	84	88.826	2.106	3.972
	2593	1747	81.051	1.758	3.908
	3862	2959	87.77	1.89	3.954
	449	652	88.492	1.902	3.995
4.	166	3000	88.949	2.032	3.969
	2969	4895	88.956	2.029	4.111
	2003	3366	88.591	1.966	3.959
	1068	4081	85.246	1.839	4.245
	1617	4718	88.668	2.031	4.105
5.	1463	2872	88.247	2.215	4.224
	1132	3062	88.048	2.242	4.16
	1706	4775	82.885	1.907	4.215
	4997	1416	77.942	1.713	4.112
	4326	3957	78.533	1.718	4.125

Time Comparison of Graph #2 in ms

GRAPH #2	SOURCE	DESTINATION	DIJKSTRA	DIJKSTRA HEAP	KRUSKAL
1.	2547	752	399.751	328.686	1310.07
	2485	540	394.97	325.175	1288.88
	1314	256	390.158	322.959	1291.18
	271	2998	366.293	324.965	1284.69
	2675	2534	390.14	325.9	1431.88
2.	3392	3192	396.232	342.36	1343.78
	4235	2717	423.682	325.428	1339.13
	1103	1428	398.517	347.416	1337.97
	2242	2186	378.854	324.447	1380.66
	2125	1524	418.511	343.124	1369.61
3.	3704	4618	395.687	327.613	1390.73
	2255	1354	400.826	324.936	1304.05
	749	415	399.048	324.882	1302.74
	4547	3930	405.411	325.538	1304.05
	2227	4015	402.542	326.595	1306.49
4.	3961	1291	386.638	326.534	1400.59
	840	2189	400.523	327.111	1320.16
	3698	2183	391.202	325.71	1365.11
	616	1096	405.318	326.107	1312.81
	1420	3298	400.052	324.526	1369.22
5.	2689	4500	395.606	323.428	1308.77
	3329	2589	396.607	322.748	1300.55
	672	4533	389.024	322.896	1299.35
	2456	3308	397.457	325.126	1298.81
	1122	4949	396.904	322.547	1319.79