**PROJECT REPORT**

**CSCE 629-602: ANALYSIS OF ALGORITHMS**

**BY:**

**Nishant Barma**

**UIN: 926000961**

**Project Report**

**Introduction:**

Network Optimization is an important area in the current research in computer science and computer engineering. This is a network routing protocol which I have implemented in Java. The implementation analyses the performance of several algorithms when run on different types of networks.

The algorithms analyzed are based on Dijkstra’s algorithm and Kruskal’s algorithm which run on data structures such as graphs and heaps.

**Implementation:**

As mentioned in the project document provided by the professor, there is an algorithm to randomly generate two kinds of graphs, each of which will have 5000 vertices. In one kind of the graphs generated, the degree of each vertex is 8 approximately. This type of graph generated can be classified as dense graph. The other kind which is generated can be considered as a dense graph. This is because the probability of an edge between any two vertices is set to be 0.2, so there being 5000 vertices in total, the expected degree of each vertex is 1000.

Also, a heap is generated using a different algorithm. This is to be used to run Dijkstra’s algorithm using the heap structure.

The following algorithms are the important ones in the program:

* To find maximum bandwidth path using Kruskal’s algorithm.
* To find maximum bandwidth path using Dijkstra’s algorithm without heap.
* To find maximum bandwidth path using Dijkstra’s algorithm with heap.

Each functionality has been written in a separate Java class file which are run as required in the main file called Network\_Optimization.

The heap functionality has been implemented in the MaxHeap.java file.

The graph generation feature is in the EdgeWeightedGraph.java which is responsible for generating

Dijkstra’s algorithm has been developed in the file Dijkstra.java which contains the algorithm that runs without using max heap.

Another file called DijkstraWithHeap.java contains the algorithm using a heap. For the heap another file called HeapDijkstra.java.

Finally, Kruskal’s algorithm is implemented in the Kruskal.java file. As Kruskal is used to find Maximum Spanning Tree, a tree needs to be implemented from the edges of the graph. For this Kruskal.java is used in which the necessary implementations are done. To implement the spanning tree, the file Union\_Find .java

Below are the details of the implementation of each of the above-mentioned things:

**HEAP:**

I use an array to implement the heap. Each node of the heap contains a vertex of the graph. It is a max heap which has INSERT, DELMAX and SORT as the interface functions and some functions are used for internal operations such as the Heapify function.

The heap is initialized with a capacity of 5000 and accordingly nodes with vertices are inserted or removed using the insert and remove functions.

The efficiency of this heap data structure and algorithm called heapsort is the main highlight here. The basic idea is to insert all the unordered items into a heap using the normal insert routine and then repeatedly using remove function which will remove items in a sorted order.

Because insert and remove functions operate in O(logN) time, and each must be applied N times, the entire sort requires O(N.logN) time, which is the same as quicksort. But, it’s not quite as fast as quicksort, partly because there are more operations in the inner while loop of the Heapify function than in the inner loop of quicksort.

So, if all the items are placed in random locations in the array and then rearranged into a heap with only N/2 calls of Heapify operation which offers a small speed advantage. This is explained below.

The heapify operation will create a correct heap if, when an out-of-order item is placed at the root, both the child subheaps of the root are correct heaps. So, Heapify can be applied to the nodes on the bottom of the heap – i.e., at the end of the array – and work its way upwards to the root at index 0. The nodes on the bottom row however are already correct heaps, because they are only trees with only one node. Therefore, there is no need to apply heapify to these nodes. So, the heapify can be started at the rightmost node with children i.e., (N/2)-1 instead of N-1 which is the last node. Hence, only half as many heapify operations are performed. The Heapify operation below:

*private void heapify(int index) {*

*while (2\*index <= n) {*

*int leftChild = 2\*index;*

*int rightChild = leftChild+1;*

*if (leftChild < n && less(leftChild, rightChild)) leftChild++;*

*if (!less(index, leftChild)) break;*

*swap(index, leftChild);*

*index = leftChild;*

*}*

*}*

**KRUSKAL:**

Used max heap and Kruskal’s algorithm to find the maximum bandwidth path from the resulting maximum spanning tree. The edges are first inserted into the max heap and then heap sort is performed on

*public Kruskal(EdgeWeightedGraph G) {*

*MaxHeap<Edge> heap = new MaxHeap<Edge>();*

*for (Edge e : G.edges()) {*

*heap.insert(e);*

*}*

*heap.sort();*

*Union\_Find uf = new Union\_Find(G.V());*

*mst\_graph = new EdgeWeightedGraph();*

*while (!heap.isEmpty() && mst.size() < G.V() - 1) {*

*Edge e = heap.delMax();*

*int v = e.either();*

*int w = e.other(v);*

*if (!uf.connected(v, w)) {*

*uf.union(v, w);*

*mst.enqueue(e);*

*mst\_graph.addEdge(e);*

*weight += e.weight();*

*}*

*}*

*assert check(G);*

*}*

**FIND PATH IN MST:**

I have implemented the Depth First Search to find the maximum bandwidth path in the final Maximum Spanning Tree that is found using Kruskal’s algorithm. Depth First Search stops when it reaches t.

*private void dfs(EdgeWeightedGraph G, int s, int t) {*

*//StdOut.println("s = "+s);*

*onPath[s] = true;*

*if(s==t) {*

*//found = true;*

*return;*

*}*

*else {*

*for (Edge u: G.adj(s)) {*

*//StdOut.println(onPath[u.other(s)]);*

*if(!onPath[u.other(s)]) {*

*//StdOut.println(u.other(s));*

*path.push(u);*

*dfs(G,u.other(s),t);*

*if(onPath[t])*

*return;*

*}*

*}*

*}*

*}*

**GRAPH:**

Used the following algorithm to generate sparse and dense graph:

public EdgeWeightedGraph(int c) {

adj = (MultiSet<Edge>[]) new MultiSet[V];

for (int v = 0; v < V; v++) {

adj[v] = new MultiSet<Edge>();

}

if(c==0) {

for (int i = 0; i < V/2; i++) {

while(degree(i) < 8) {

int j = getRandomNumberInRange(0,V-1);

if(j!=i && edgeExists(i,j) == false && degree(j)<8) {

int weight = getRandomNumberInRange(1,1000);

Edge e = new Edge(i, j, weight);

addEdge(e);

}

}

}

}

else if(c==1) {

for (int i = 0; i < V/2; i++) {

while(degree(i) < 0.2\*V) {

int j = getRandomNumberInRange(0,V-1);

if(j!=i && edgeExists(i,j) == false && degree(j)<0.2\*V) {

int weight = getRandomNumberInRange(1,1001);

Edge e = new Edge(i, j, weight);

addEdge(e);

}

}

}

}

}

The constructor takes an integer c as input which helps in creating sparse and dense graphs. If c is 0 then a sparse graph is created and if c is 1 then a dense graph is created.

**DIJKSTRA’S ALGORITHM:**

I implemented this algorithm based on the steps taught in class by the professor. It takes a max vertex from the fringe of vertices. The algorithm has been modified to select the max bandwidth instead of finding the shortest path.

*public Dijkstra(EdgeWeightedGraph G, int s, int t) { //EdgeWeightedGraph g, int s, int t*

*status = new int[5000];*

*for(int i=0; i<5000; i++) {*

*status[i] = 2;*

*}*

*status[s] = 0;*

*bw = new int[5000];*

*for(int i=0; i<5000; i++) {*

*bw[i] = 0;*

*}*

*dad = new int[5000];*

*for(int i=0; i<5000; i++) {*

*dad[i] = -1;*

*}*

*bw[s] = Integer.MAX\_VALUE;*

*for (Edge u: G.adj(s)) {*

*bw[u.other(s)] = u.weight();*

*status[u.other(s)] = 1;*

*dad[u.other(s)] = s;*

*}*

*while(status[t] != 0) {*

*int v = getFringe(bw,status);*

*status[v] = 0;*

*for (Edge e: G.adj(v)) {*

*int w = e.other(v);*

*if(status[w] == 2) {*

*bw[w] = bw[v] + e.weight();*

*dad[w] = v;*

*status[w] = 1;*

*}*

*else if(status[w] == 1 && bw[w] < bw[v] + e.weight()) {*

*bw[w] = bw[v] + e.weight();*

*dad[w] = v;*

*}*

*}*

*}*

*dist = bw[t];*

*}*

**DIJKSTRA’S USING HEAP:**

This is almost the same algorithm but the fringe of vertices is stored in a max heap. The fringe of neighboring vertices is stored in the heap and every time the max fringe is found directly by removing the max of the heap.

**RUN TIMES OF THE ALGORITHMS:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Graph type with s and t** | **Dijkstra without heap (time in milliseconds)** | **Dijkstra with heap (time in milliseconds)** | **Kruskal (time in milliseconds)** |
| Sparse graph with s = 4828 and t = 60 | 68.675898 | 26.50697 | 131.005555 |
| Dense graph with s = 4828 and t = 60 | 390.13256 | 533.40885 | 9765.741035 |
| Sparse graph with s = 2494 and t = 3238 | 65.128852 | 3.702868 | 34.370839 |
| Dense graph with s = 2494 and t = 3238 | 140.607257 | 523.157267 | 1047.365165 |
| Sparse graph with s = 2715 and t = 1919 | 47.725329 | 1.268025 | 30.932962 |
| Dense graph with s = 2715 and t = 1919 | 344.766487 | 274.767504 | 1051.271907 |
| Sparse graph with s = 1801 and t = 4307 | 275.802739 | 9.992191 | 29.49231 |
| Dense graph with s = 1801 and t = 4307 | 62.020347 | 141.292594 | 1177.800617 |
| Sparse graph with s = 2780 and t = 3635 | 107.145689 | 10.409271 | 33.704165 |
| Dense graph with s = 2780 and t = 3635 | 319.316213 | 347.85913 | 1154.566839 |

**ANALYSIS OF THE ALGORITHMS:**

Kruskal’s algorithm performs much faster on sparse graphs than on dense graphs. As expected, Kruskal’s performs admirably and gives the correct path.

Dijkstra’s algorithm is not working properly. Because the path that I get after running the algorithm is not valid. As expected, the running times are not correct as can be seen from the above table.

The algorithm that uses the heap is slower in the case of dense graphs than the algorithm that does not use heap.

**IDEAL ANALYSIS:**

Normally, the Dijkstra’s algorithm that does not use max heap to store the fringes takes O(n2) time. In every case, the algorithm with the heap should run faster.