CSCE 629-601 Analysis of Algorithms
Fall 2021 Course Project

Due date: Nov 30 *Instructor: Dr. Jianer Chen* *Name: Rajesh Satpathy*

Problem Statement:

Network optimization has been an important area in the current research in computer science and computer engineering. In this course project, you will implement a network routing protocol using the data structures and algorithms we have studied in class. This provides you with an opportunity to translate your theoretical understanding into a real-world practical computer program. Translating algorithmic ideas at a "higher level" of abstraction into real implementations in a particular programming language is not at all always trivial. The implementations often force you to work on more details of the algorithms, which sometimes may lead to much better understanding.

1 Random Graph Generation

1.1 Data structure used for storing graph

The Graphs are stored in Adjacency List. The reason for choosing adjacency list over adjacency matrix in the implementation is that it is better for storage of sparse graphs. Adjacency lists give a time complexity of $\mathbf{O(m+n)}$ compared to $\mathbf{O(n^2)}$ of adjaceny matrix (where, $m \approx n$, m = No. of edges; n = No. of vertices). Another reason to list over matrix is that the algorithm implementations part of the project, Dijkstra's and Kruskal's involve edge traversals on vertices, which is better in adjacency lists.

1.2 Graph Generation

The graphs getting generated have 5000 vertices and the number of edges are added in graph 1 and graph 2 according to the conditions given. The capacities of the edges are randomized for a given maximum capacity from $max\_capacity/2$ to $max\_capacity$(max capacity 50 will range capacities from 25 to 50). The graph generations in this implementation are undirected as suggested. The graph is first connected by adding edge between each vertex and next vertex in the adjacency list and in the end between last vertex and first vertex to generate a cycle. The number of edges added to make a graph cycle is n. Hence, the time complexity to create a generated graph is $\mathbf{O(n)}$.

1.2.1 Graph 1

The first graph **G1** requires an average vertex degree of 6. Edges are added randomly until an average degree of 6 is reached. The following approach is used to achieve this. (Figure. 1)

```
public static VertexNode[] generateGraph1(VertexNode[] graph1, int n, int weightLimit, int avgDegree) {
    int randSrc; // random src vertex variable
    int randDest; // random destination variable
    int randWeight; // random edge weight generator variable

    int avgEdgeSum = n * avgDegree / 2;
    int currEdgeSum = n;

    while (currEdgeSum < avgEdgeSum) {
        randSrc = random.nextInt(n);
        randDest = random.nextInt(n);

        if (!ContainsEdge(graph1[randSrc], randDest) && randSrc != randDest) {
            randWeight = random.nextInt(weightLimit + 1 - weightLimit / 2) + weightLimit / 2;
//          System.out.println(randSrc + ", " + randDest + ", " + randWeight);
            graph1[randSrc] = new VertexNode(randDest, randWeight, graph1[randSrc].index + 1, graph1[randSrc]);
            graph1[randDest] = new VertexNode(randSrc, randWeight, graph1[randDest].index + 1, graph1[randDest]);
            currEdgeSum++;
        }
    }

    return graph1;
}
```

Figure 1: Code for graph 2 generation

The time complexity for graph 1 generation is **O(m)**, where m is the number of edges.

1.2.2 Graph 2

The second graph **G2** requires that each vertex is adjacent to about 20% of the other vertices. A vertex is chosen and a random number of edges is added to the vertex's list. The edge is added to the start of the list to improve efficiency. The following is approach is used to achieve this. (Figure. 2)

```
public static VertexNode[] generateGraph2(VertexNode[] graph2, int n, int weightLimit, float percentage) {
    int randDest; // random destination variable
    int randWeight; // random edge weight generator variable

    int currentVertexPercentage = (int) Math.round(n * percentage) / 2;
    int currVertexSum;

    for (int i = 0; i < n; i++) {
        currVertexSum = 1;
        while (currVertexSum < currentVertexPercentage) {
            randDest = random.nextInt(n);

            if (!ContainsEdge(graph2[i], randDest) && i != randDest
                    && graph2[randDest].getIndex() != currentVertexPercentage) {
                randWeight = random.nextInt(weightLimit + 1 - weightLimit / 2) + weightLimit / 2;
                graph2[i] = new VertexNode(randDest, randWeight, graph2[i].index + 1, graph2[i]);
                graph2[randDest] = new VertexNode(i, randWeight, graph2[randDest].index + 1, graph2[randDest]);
                currVertexSum++;
            }
        }
    }

    return graph2;
}
```

Figure 2: Code for graph2 generation

The time complexity for graph 2 generation is **O(m)**, where m is the number of edges. Here as the edges are added at the start of the linked list no traversal time is spent. Hence, time complexity is decided by the number of edges added.

2 Heap Generation

A Max Heap datastructure is programmed to be used in Dijkstra's algorithm and Kruskal's algorithm. The Max Heap is an array that stores the vertex (vertexName, edgeWeight) and a hashmap is used to store the positions of the vertex in the Heap. A similar Max Heap is used for edge storage in Kruskal's algorithm (does not store a map for position of Edge because we always need only the max Edge). The operations of the Max Heap - Delete, Insert and Maximum are discussed below.

2.1 Delete

The Delete operation of heap is performed by swapping the current element with the last element of the heap. The position of the vertex to be deleted is accessed from the map. The last element of the heap is removed and a heapification is performed bottom-up and top-down from the given vertex position. The vertex positions are updated during every swap. The approach to deleting a vertex is as shown in Figure 3. Bottom-Up heapify is a part of the delete code (Figure 3.) and Top-Down Heapify is as shown in Figure. 4.

```
//  Delete vertex from Heap
    public void delete(int vertex) {
        int p = pos.get(vertex);
        swap(p, maxHeap.size() - 1);
        maxHeap.remove(maxHeap.size() - 1);
        pos.remove(vertex);

//      Bottom-Up Heapify
        while (p > 0 && p < maxHeap.size() && maxHeap.get(p / 2).edgeWt - maxHeap.get(p).edgeWt < 0) {
            swap(p / 2, p);
            p = p / 2;
        }

//      Top-Down Heapify
        heapify(p);
    }
```

Figure 3: Code for deleting from Heap

```
//  Top-Down Heapify
    private void heapify(int n) {
        int leftChild = 2 * n;
        int rightChild = 2 * n + 1;
        int largeIndex = n;
        if (leftChild < maxHeap.size() && maxHeap.get(leftChild).edgeWt - maxHeap.get(n).edgeWt > 0) {
            largeIndex = leftChild;
        }
        if (rightChild < maxHeap.size() && maxHeap.get(rightChild).edgeWt - maxHeap.get(largeIndex).edgeWt > 0) {
            largeIndex = rightChild;
        }
        if (largeIndex != n) {
            swap(n, largeIndex);
            heapify(largeIndex);
        }
    }
```

Figure 4: Code for Top-Down Heapify

The time complexity of a Top-Down Heapify and Bottom-Up Heapify are **O(log n)**, where n is the size of the Heap. Hence, a delete operation takes a time of **O(log n)**.

2.2 Insert

The vertex is inserted to end of the heap and then a Bottom-Up heapify is performed. This will bring the inserted vertex into the required position in heap. The approach is shown in Figure 5.

The time complexity of an insert operation is dependent on Bottom-Up Heapify which takes a time of **O(log n)**, where n is size of Heap. Hence, an insert operation takes a time complexity of **O(log n)**

```
//  Insert vertex into Heap
    public void insert(VertexNode fringe) {
        maxHeap.add(fringe);
        int i = maxHeap.size() - 1;
        pos.put(fringe.vertex, i);

//      bottom-up heapify
        while (i > 0 && i < maxHeap.size() && maxHeap.get(i / 2).edgeWt - maxHeap.get(i).edgeWt < 0) {
            swap(i / 2, i);
            i = i / 2;
        }
    }
```

Figure 5: Code for insert into Heap

2.3 Maximum

To get the maximum of Heap, we Delete the first element in the Heap. This takes the same time as Delete i.e. **O(log n)**, where n is size of Heap.

3 Routing Algorithms

This projects deals with solving the MAX-BANDWIDTH-PATH problem using Dijkstra's and Kruskal's algorithm. Dijkstra's algorithm implementation is tested with Heap and without Heap.

3.1 Dijkstra's without Heap

This version of Dijsktra's does not use a Heap. The fringes are stored in a list. To find the Max fringe each time, takes a time of **O(n)**. The stored dad array is used to traverse the s-t path giving a time complexity of **O(n)**. The find max fringe runs within a while loop until there are fringes which runs m times. This makes the time complexity of the algorithm to be **O(mn)** i.e. $\mathbf{O(n^2)}$ when, $m \approx n$. The approach is as shown in Figure 6.

```
public static void dijskstraWithoutHeap(VertexNode[] graph, int src, int dest, int noOfNodes) {
    int unseen = 0, fringe = 1, intree = 2;
    int[] status = new int[noOfNodes];
    int[] bw = new int[noOfNodes];
    int[] dad = new int[noOfNodes];
    List<VertexNode> fringes = new ArrayList<>();

    status[src] = intree;

    // add nodes from src
    VertexNode head = graph[src];
    while (head != null) {
        status[head.vertex] = fringe;
        dad[head.vertex] = src;
        bw[head.vertex] = head.edgeWt;
        fringes.add(head);
        head = head.next;
    }

    // add nodes for unseen and update status, dad and bw for unseen and fringe
    while (!fringes.isEmpty()) {
        VertexNode maxFringe = getMaxFringe(fringes);
        status[maxFringe.vertex] = intree;
        VertexNode node = graph[maxFringe.vertex];
        while (node != null) {
            if (status[node.vertex] == unseen) {
                status[node.vertex] = fringe;
                dad[node.vertex] = maxFringe.vertex;
                bw[node.vertex] = Math.min(bw[maxFringe.vertex], node.edgeWt);
                fringes.add(new VertexNode(node.vertex, bw[node.vertex], 0, null) {
                });
            } else if (status[node.vertex] == fringe
                    && bw[node.vertex] < Math.min(bw[maxFringe.vertex], node.edgeWt)) {
                dad[node.vertex] = maxFringe.vertex;
                bw[node.vertex] = Math.min(bw[maxFringe.vertex], node.edgeWt);
                updateFringe(fringes, node.vertex, bw[node.vertex]);
            }
            node = node.next;
        }
    }
```

Figure 6: Code for running Dijkstra without heap

3.2 Dijkstra's with Heap

This implementation of Dijkstra's algorithm uses a Heap, instead of using a linked list traversal each time to get a Max. The stored dad array is used to traverse the s-t path giving a time complexity of **O(n)**. A Max Heap returns the max vertex in **O(log n)** using Maximum. Similarly the Insert and Delete operations also take same time complexity as Maximum - discussed in **Section 2**. The loop runs for m times, hence Dijkstra's algorithm is performed in **O(mlog n)**. This is better than the previous approach not using a Heap. The approach is as given in Figure 7.

```
public static void dijkstraWithHeap(VertexNode[] graph, int src, int dest, int noOfNodes) {
    int unseen = 0, fringe = 1, intree = 2;
    int[] status = new int[noOfNodes];
    int[] bw = new int[noOfNodes];
    int[] dad = new int[noOfNodes];
    List<VertexNode> fringes = new ArrayList<>();

    status[src] = intree;

    // add nodes from src
    VertexNode head = graph[src];
    while (head != null) {
        status[head.vertex] = fringe;
        dad[head.vertex] = src;
        bw[head.vertex] = head.edgeWt;
        fringes.add(head);
        head = head.next;
    }
    MaxHeap maxHeap = new MaxHeap(fringes);

    // add nodes for unseen in heap and update status, dad and bw for unseen and
    // fringe
    while (!maxHeap.maxHeap.isEmpty()) {
        VertexNode maxFringe = maxHeap.popMax();
        status[maxFringe.vertex] = intree;
        VertexNode node = graph[maxFringe.vertex];
        while (node != null) {
            if (status[node.vertex] == unseen) {
                status[node.vertex] = fringe;
                dad[node.vertex] = maxFringe.vertex;
                bw[node.vertex] = Math.min(bw[maxFringe.vertex], node.edgeWt);
                maxHeap.insert(new VertexNode(node.vertex, bw[node.vertex], 0, null));
            } else if (status[node.vertex] == fringe
                    && bw[node.vertex] < Math.min(bw[maxFringe.vertex], node.edgeWt)) {
                maxHeap.delete(node.vertex);
                dad[node.vertex] = maxFringe.vertex;
                bw[node.vertex] = Math.min(bw[maxFringe.vertex], node.edgeWt);
                maxHeap.insert(new VertexNode(node.vertex, bw[node.vertex], 0, null));
            }
            node = node.next;
        }
    }
```

Figure 7: Code for running Dijkstra with heap

3.3 Kruskal's

We solve the MAX-BANDWIDTH-PATH problem here by generating the Max Spanning Tree for the graph. The minimum edge capacity of the path from s-t will be the max bandwidth of the path. After sorting the edges in **O(m logn)**, the maximum edges are picked in **O(log n)** from Heap and Max Spanning Tree is generated. The generation of Max Spanning Tree takes $\mathbf{O(n\log^* n)} \approx \mathbf{O(n)}$. The DFS traversal takes **O(m+n)** to get s-t path. The time complexity of Kruskal's algorithm is bounded by the sorting algorithm used which takes the time complexity of **O(mlog n)**. Hence, the time complexity of **O(mlog n)**.

It is interesting to note that in a max spanning tree **All s-t Paths** generated will be MAX-BANDWIDTH-PATHS. So for a given graph, the s-t path generation will only take **O(n)**. This is better in cases we have to find the s-t paths on the same graph. The approach is as given in Figure 8.

```java
//  Kruskal's algorithm
    public static void kruskal(VertexNode[] graph, int s, int t, int n) {
        UnionFind unionFind = new UnionFind(n);

        List<Edge> edgeList = getEdgesGraph(graph);
        MaxHeapEdge maxHeapEdge = new MaxHeapEdge(edgeList);


//      Collections.sort(edgeList, (e1, e2) -> (e2.edgeWt - e1.edgeWt));

        VertexNode[] maxSpanningTree = new VertexNode[n];

        while (!maxHeapEdge.maxHeap.isEmpty()) {
            Edge edge = maxHeapEdge.popMax();
            int v1 = edge.v1;
            int v2 = edge.v2;

            int r1 = unionFind.find(v1);
            int r2 = unionFind.find(v2);

            if (r1 != r2) {
                unionFind.union(r1, r2);

                if (maxSpanningTree[v1] == null) {
                    maxSpanningTree[v1] = new VertexNode((v2) % n, edge.edgeWt, 1, null);
                } else {
                    maxSpanningTree[v1] = new VertexNode((v2) % n, edge.edgeWt, maxSpanningTree[v1].index + 1,
                            maxSpanningTree[v1]);
                }

                if (maxSpanningTree[v2] == null) {
                    maxSpanningTree[v2] = new VertexNode((v1) % n, edge.edgeWt, 1, null);
                } else {
                    maxSpanningTree[v2] = new VertexNode((v1) % n, edge.edgeWt, maxSpanningTree[v2].index + 1,
                            maxSpanningTree[v2]);
                }
            }
        }

        boolean[] isVisited = new boolean[n];

        dfs(maxSpanningTree, s, isVisited);
```

Figure 8: Code for running Dijkstra with heap

4 Testing

The test results involve generating 5 graphs of each type and test with 5 s-t pairs for each graph generated. Figure 9 shows the output for 1 pair of s-t in milliseconds.

```
For s: 1336;t: 1084
1====================================
Graph  Dwithout     Dwith  K
G1-1   |42          |22    |33
G2-1   |156         |93    |2909
+====================================+
G1-2   |20          |6     |9
G2-2   |117         |59    |2971
+====================================+
G1-3   |22          |6     |8
G2-3   |161         |50    |2859
+====================================+
G1-4   |22          |5     |8
G2-4   |295         |203   |2900
+====================================+
G1-5   |17          |3     |7
G2-5   |309         |216   |2883
+====================================+

2====================================
G1-1   |45          |22    |20
G2-1   |157         |71    |41
+====================================+
G1-2   |20          |6     |4
G2-2   |122         |65    |38
+====================================+
G1-3   |22          |5     |0
G2-3   |185         |100   |12
+====================================+
G1-4   |24          |5     |1
G2-4   |155         |64    |20
+====================================+
G1-5   |21          |5     |0
G2-5   |123         |42    |10
+====================================+

3====================================
G1-1   |43          |21    |3
G2-1   |155         |97    |29
+====================================+
G1-2   |21          |6     |0
G2-2   |151         |94    |28
```

Figure 9: Output in ms for each graph for given s-t pair

From the results we can observe that Dijkstra with Heap performs better than Dijk-

stra without Heap. It is an expected result given the time complexities of the algorithms. We can observe that Kruskal performs better at sparse graphs(Graph 1) but performs poorly on dense graphs(Graph 2). However it can be observed that for the same graph with different s-t pairs Kruskal performs way better. This is because the paths are already generated and can be found by just running a DFS on the spanning tree, $\mathbf{O(m+n)}$ which is a huge improvement over the other 2 Dijkstra algorithms.