Comp-2540: Lab Assignment 5: Dijkstra Algorithm

Your Marks (For TA Use Only)

3.1	3.2	4.1	4.2	Total	Marked by

1 Due Date and Submission

The due time is your lab section during the week starting November 30. To be submitted 15 minutes before the end of your lab section. You can also submit in the labs one week earlier or during office hours. Make sure that you add your signature below to reaffirm that you followed Senate Bylaws 31 (click here for the document). Some PDF readers may not support digital signature well. We recommend you to use Acrobat Reader that is free for downloading.

Fill and Sign The Form

I, your name Ethan Kunce **, verify that the submitted work is my own work.**

Date 09/12/202 Student Number 110043546 EMail ID kunce

Submission site (click on this) .

2 Objectives

Through this assignment, we will

- Learn graph representation. How to read graph from text representation, represent them using a set of edges, and run algorithms on the graph.
- Understand Dijkstra's shortest path algorithm. Understand the time complexity of the algorithm.
- Revisit the pros and cons of HashMap
- Revisit the pros and cons of Heap data structure
- Improve the traditional Priority Queue by adding updating operation

3 Shortest Path Implemented Using HashMap

The starter code for the Shortest Path algorithm is listed in Fig. 3. It has three lines missing in the relaxation method. You need to fix it so that it can run in the submission site. It uses Graph and Edge classes. You need to download those two classes by clicking on them. They are abridged from the book Algorithms, 4th Edition, By Sedgewick et al.. Our code removes clutters that are not needed for the Dijkstra's algorithm. The explanation of the code and the missing parts are in the lecture.

We will run the following client code to test your shortest path algorithm. It first reads a graph from a text file named graph1.txt. Note that that graph is the same as the first example in in our slides for tracing the Dijkstra's algorithm. Then it runs the `Graph_SP`, to find the shortest path starting from source node s. You can print out the shortest paths to verify the results. Or you can change the code to print out the trace of the algorithm.

```
import java.io.*;
import java.util.*;
public class Graph_SP {
  public double[] distTo;
  public Edge[] edgeTo;
  public Map distTable = new HashMap<Integer, Double>();

  public Graph_SP(Graph G, int s) {
        distTo = new double[G.V];
        edgeTo = new Edge[G.V];
        for (int v = 0; v < G.V; v++)
                distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;
        distTable.put(s, distTo[s]);
        while (!distTable.isEmpty()) {
                int v = removeMin(distTable);
                for (Edge e : G.edgesOf[v])
                        relax(e);
        }
  }

  private void relax(Edge e) {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
          //add the relaxation part here.
          //three lines of code.
        }
  }

        public static int removeMin(Map<Integer, Double> distTable) {
                Iterator it = distTable.entrySet().iterator();
                double minValue = Integer.MAX_VALUE;
                int minKey = -1;
                while (it.hasNext()) {
                        Map.Entry pair = (Map.Entry) it.next();
                        if (minValue > (Double) pair.getValue()) {
                                minKey = (Integer) pair.getKey();
                                minValue = (Double) pair.getValue();
                        }
                }
                distTable.remove(minKey);
                return minKey;
        }
}
```

Figure 1: The shortest path algorithm using HashMap.

```
import java.io.*;
import java.util.*;

public class Graph_client {
public static void main(String[] args) throws Exception {
        Graph G = new Graph(new Scanner(new File("graph1.txt")));
        int s = 0;
        long startTime = System.currentTimeMillis();
        Graph_SP sp = new Graph_SP(G, s);
        System.out.println(System.currentTimeMillis()-startTime);

        for (int t = 0; t < 5; t++) { // print the first a few shortest paths
                if (sp.distTo[t] < Double.POSITIVE_INFINITY) {
                        System.out.printf("%d => %d (%.2f)  ", s, t, sp.distTo[t]);
                        String path = t + "";
                        for (Edge e = sp.edgeTo[t]; e != null; e =sp. edgeTo[e.from()]) {
                                path = e.from() + "->(" + e.weight() + ")->" + path;
                        }
                        System.out.println(path);
                } else {
                        System.out.printf("%d to %d         no path\n", s, t);
                }
        }
}
}
```

3.1 Fill in the missing code in Graph_SP(3 mark)

Write below the missing code in Graph_SP. You will get the marks only if your code runs successfully in the submission site.

```
distTo[w] = distTo[v] + e.weight();
edgeTo[w] = e;
distTable.put(w, distTo[w]);
```

3.2 Complexity of Graph_SP(1 mark)

Write below the time complexity of Graph_SP in terms of the node size V and edge size E. Give your justification briefly.

First we initialize all the unknown distances, in V time. We also call removeMin() which has a O (n^2)

4 Improve Graph_SP using Heap2540_SP

The algorithm needs to update the distance of a node often. Hence, we used the HashMap in our previous implementation. However, we also need to find the minimum, which is not well supported in HahMap. Finding minimum is well dealt with by Heap. So we use $Heap2540\_SP$. It is similar to Heap2540, but adapted to deal with the need of shortest path algorithm. The differences are

- Each element is a pair containing an int (node ID) and a double (distance). We use array double [] keys to save the pairs
- Comparison is based on the distance.
- It is MinHeap, not a maxHeap.
- Need to check and update the distance of nodes. To access the node quickly, we track where the node is in the heap using an `index` array.

```
public class Heap2540_SP {
        int[] heap;
        int[] index;
        Double[] distances;
        int n = 0; // actuall heap size.
        int CAPACITY = 1000001; // array size

        public Heap2540_SP() {
                heap = new int[CAPACITY];
                index = new int[CAPACITY];
                distances = new Double[CAPACITY];
                for (int i = 0; i < CAPACITY; i++)
                        index[i] = -1;
        }

        public boolean isEmpty() {
                return n == 0;
        }

        public int removeMin() {
                int min = heap[1];
                swap(1, n);
                n--;
                sink(1);
                index[min] = -1;
                distances[min] = null;
                return min;
        }

        public void insert(Integer node, Double dist) {
                n++;
                heap[n] = node;
                index[node] = n;
                distances[node] = dist;
                swim(n);
        }
        private void swim(int k) {
                while (k > 1 && greater(k / 2, k)) {
                        swap(k, k / 2);
                        k = k / 2;
                }
        }

        public void put(int node, double dist) {
                if (index[node] != -1) {
                        distances[node] = dist;
                        swim(index[node]);
                } else
                        insert(node, dist);
        }

        private void swap(int i, int j) {
                Integer temp = heap[i];
                heap[i] = heap[j];
                heap[j] = temp;
                // add code to keep track of the index of nodes
        }

        private void sink(int k) {
                while (2 * k <= n) {
                        int j = 2 * k;
                        if (j < n && greater(j, j + 1))
                                j++;
                        if (!greater(k, j))
                                break;
                        swap(k, j);
                        k = j;
                }
        }

        private boolean greater(int i, int j) {
            //add code here. return true if the dist  to i is greater
        }
}
```

Figure 2: The HEAP2540_SP class with two parts missing: One is the definition for greater. Another is in the swap method where you should update the index positions of the nodes.

4.1 Complete the code (3 points)

We provide the following starter code. You need to write the following two missing parts. Note that you can only receive marks if your code can run successfully on our submission site, and your running time is within 500ms. You can test your code on large graphs that are randomly generated by us:

- Graph with 1000 nodes
- Graph with 10000 nodes
- Write below the missing parts for the swap method:

```
temp = index[heap[i]]; //temp becomes the value heap[i]
points to, and temp points to
index[heap[i]] = index[heap[j]];
index[heap[j]] = temp;
```

- Write below the missing part for the greater method:

```
return distances[heap[i]] > distances[heap[j]];
```

4.2 Time complexity for the Dijkstra's Algorithm

When `Heap2540_SP` is used, you can see that it is faster. Write below the time complexity for the algorithm, and give a brief justification.

First we initialize all the unknown distances, in V time. We also call removeMin() which has a O(log(n)) V times, therefore O (nlog(V)). we also have E number of calls to to relax() method for checking & updating the distance, This has a O(mlog(n)). therefore our total complexity is O((V+E)Log(V)). but because