Assignment Five

Tyler Hernandez

Tyler.Hernandez1@marist.edu

December 9, 2022

This page left partially blank intentionally.

1 Main Method

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Main {

    // Driver for Assignment 5.
    public static void main(String[] args) throws Exception {

        Reader reader = new Reader("./graphs2.txt");

        String line = reader.getNextLine();
        String[] words;
        ArrayList<Graph> graphs = new ArrayList<>();

        Graph g = new Graph(); // Initialize so our compiler will stop complaining.

        // Flag to make sure while loop runs one time after reader flags end of file
        boolean readingLastLine = false;

        while (!readingLastLine) { // Until we've reached the end of the file.

            if (reader.endOfFile) {
                readingLastLine = true; // We've reached end of file, we can stop looping.
            }
            // System.out.println(line);

            // Take our next line of text and put each word into an array.
            line = line.replaceAll("\n", "");
            line = line.replaceAll("  ", " "); // Nice try, professor.
            words = line.split(" ");

            // Now, we can determine which action to take strictly based on sentence length!

            if (words.length == 0) {
                // Blank line.
            } else if (words[0].equals("--")) {
                // Do nothing because this is a comment.
            } else if (words.length == 2) {
                // Save previous graph to our list, if it was not the default graph.
                if (!g.isEmpty()) {
                    graphs.add(g);
                }

                // NEW GRAPH command
                g = new Graph();
            } else if (words.length == 3) {
                // ADD VERTEX (INT) command
                g.addVertex(Integer.parseInt(words[2]));
            } else if (words.length == 6) {
                // ADD EDGE (INT) - (INT) (WEIGHT) command
                g.addEdge(Integer.parseInt(words[2]), Integer.parseInt(words[4]), Integer.parseInt(words[5]
            }

            // Grab the next line so we can keep going!
            line = reader.getNextLine();
        } // end of while.

        // Since the final graph never reaches our 'save' in while loop.
        graphs.add(g);

        // For each graph,
        for (Graph graph : graphs) {
```

```
64              System.out.println("-------------------------------------------");
65              System.out.println("GRAPH: ");
66
67              // display it's adjacency list!
68              System.out.println("Adjacency List\n");
69              graph.printList();
70              System.out.println("-------------------------------------------");
71              int[] distances = SSSP(graph, 1);
72
73              for (int i = 0; i < distances.length; i++) {
74                  System.out.println("1 -> " + (i + 1) + " cost is " + distances[i]);
75              }
76
77              System.out.println("\n");
78
79          }
80          System.out.println("\n\n\n");
81
82          // Part 2: Spice.txt
83          ArrayList<Spice> spices = new ArrayList<>(); // will hold all spices.
84          ArrayList<Integer> commands = new ArrayList<>();
85
86          // First, read in all spices into spices arraylist and gather all 'knapsack
87          // capacity' commands.
88          reader = new Reader("./spice.txt");
89          line = reader.getNextLine();
90          readingLastLine = false; // reset flag from previous use.
91          while (!readingLastLine) { // Until we've reached the end of the file.
92
93              if (reader.endOfFile) {
94                  readingLastLine = true; // We've reached end of file, we can stop looping.
95              }
96
97              line = line.replaceAll("\n", "");
98              line = line.replaceAll(";", ""); // gets rid of those semicolons.
99              line = line.replaceAll("  ", " "); // There's a lot of double
100             line = line.replaceAll("  ", " "); // spaces we need to get
101             line = line.replaceAll("  ", " "); // rid of :)
102             line = line.replaceAll("  ", " "); // ... and this ones for good luck.
103             words = line.split(" ");
104
105             if (words.length == 0) {
106                 // Blank line.
107             } else if (words[0].equals("--")) {
108                 // Do nothing because this is a comment.
109             } else if (words.length == 4) {
110                 // Declaring commands here.
111                 // Knapsack capacity = (int) command.
112                 commands.add(Integer.parseInt(words[3]));
113             } else if (words.length == 10) {
114                 // Declaring spices here.
115                 // 3 : (name), 6 : (price), 9 : (quantity).
116                 spices.add(new Spice(words[3], words[6], words[9]));
117             }
118
119             line = reader.getNextLine();
120         }
121
122         // For each command (quantity), run our fractionalKnapsack algorithm.
123         for (int command : commands) {
124             Spice[] bag = fractionalKnapsack(spices, command);
125             printSpices(bag);
126         }
127
128     }
```

```
129
130      // This is not my algorithm, I am using it for this assignment's purpose and
131     // have modified it for my better understanding and to match my data set.
132     // Here is the original source:
133     // https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
134     public static int[] SSSP(Graph graph, int src) {
135         // Grab the length of our vertices and links to know how many options we have at
136         // a given moment.
137         int lengthOfVerticesList = graph.vertices.size();
138         int lengthOfLinksList = graph.links.size();
139         int dist[] = new int[lengthOfVerticesList];
140
141         // First, set the distance from source to every other vertices as
142         // Integer.MAX_VALUE so we don't blindly choose an unexplored path.
143         for (int i = 0; i < lengthOfVerticesList; i++) {
144             dist[i] = Integer.MAX_VALUE;
145         }
146
147         dist[src - 1] = 0;
148
149         // Next, check each edge's weight to find the single shortest path.
150         for (int i = 1; i < lengthOfVerticesList; i++) {
151             for (int j = 0; j < lengthOfLinksList; j++) {
152                 int u = graph.links.get(j).sourceId - 1;
153                 int v = graph.links.get(j).destinationId - 1;
154                 int weight = graph.links.get(j).weight;
155                 if (dist[u] != Integer.MAX_VALUE
156                         && dist[u] + weight < dist[v])
157                     dist[v] = dist[u] + weight;
158             }
159         }
160
161         // Lastly, we need to ensure a negative weight cycle is not found or else we
162         // will infinitely find "shorter" paths every time.
163         for (int j = 0; j < lengthOfLinksList; j++) {
164             int u = graph.links.get(j).sourceId - 1;
165             int v = graph.links.get(j).destinationId - 1;
166             int weight = graph.links.get(j).weight;
167             if (dist[u] != Integer.MAX_VALUE
168                     && dist[u] + weight < dist[v]) {
169                 System.out.println(
170                         "Error: Graph contains negative weight cycle");
171                 return dist;
172             }
173         }
174         return dist;
175     }
176
177     // Sort by the highest value per unit and take them in that order.
178     public static Spice[] fractionalKnapsack(ArrayList<Spice> spices, int quantity) {
179
180         // Comparator for comparing spices by unit price.
181         Comparator<Spice> compareByUnitPrice = new Comparator<Spice>() {
182             @Override
183             public int compare(Spice spice1, Spice spice2) {
184                 return spice2.compareTo(spice1);
185             }
186         };
187
188         // Sort spices by unit price.
189         Collections.sort(spices, compareByUnitPrice);
190
191         // Initialize bag to be quantity length
192         Spice[] myBag = new Spice[quantity];
193         int counter = 0; // Counter to place spices in order of value and prevent out of bounds
```

```
                        // exception.

        // Loop through spices and grab the first 'quantity' many.
        for (Spice spice : spices) {
            if (counter == quantity) {
                break;
            } else {

                // Keep appending this spice until it either runs out or we don't have anymore
                // space.
                double spiceQuantity = spice.quantity;
                while ((spiceQuantity > 0) && (counter < quantity)) {

                    spiceQuantity -= 1;
                    myBag[counter] = spice;
                    counter++;
                }

            }
        }

        return myBag;

    }
```

1.1 Main Method

For this assignment there were two milestones. First, I was given a graph text file which I needed to parse, store, and run Bellman-Ford's Single Source Shortest Path algorithm on given the first vertex. Then, I was given a spices text file which I also needed to parse, store, and run the fractional knapsack algorithm in order to produce maximum results.

Bellman-Ford's SSSP algorithm's asymptotic running time complexity is big O of V * E, where V is how many vertices we have in the graph, and E is how many edges are in the graph. This makes sense because the algorithm works by not having to traverse the same path more than once, since every time it visits a path it will overwrite it's initial "max distance" to the real distance in the weighted graph.

Next, for the Fractional Knapsack algorithm, the asymptotic running time complexity is big O of n * log n. This is because once the algorithm sorts the spices in order of what will give the highest value per capacity, all it needs to do is retrieve the first 'bag capacity' elements.

Some of the challenges I faced with this assignment included parsing the files which could contain extra spaces at any given moment. I tackled this problem by getting rid of double triple and quadruple spaces before I split the words into an array for processing.

1.2 Link

```
// Wrapper for sourceId, weight, and destination link.
public class Link {

    public int sourceId;
    public int weight;
    public int destinationId;

    public Link(int sourceId, int weight, int destinationId) {
        this.sourceId = sourceId;
        this.weight = weight;
        this.destinationId = destinationId;
    }

    public String toString() {
        return ("--(" + this.weight + ")-->" + this.destinationId);
    }

}
```

In order to run bellman ford's algorithm, I needed to find a way to hold each edge's information. This meant recording the source of the directional edge, the destination of the edge, and the weight. I accomplished this with my 'Link' class which is a mere wrapper for the information. Lastly, it's toString was helpful for visualizing the graph in the adjacency list.

1.3 Spice

```
public class Spice implements Comparable<Spice> {

    public String name;
    public double price;
    public double quantity;
    public double unitPrice;

    public Spice(String name, String price, String quantity) {

        // Parse our strings into proper data types.
        this.name = name;
        this.price = Double.parseDouble(price);
        this.quantity = Double.parseDouble(quantity);
        this.unitPrice = this.price / this.quantity;
    }

    @Override
    public int compareTo(Spice spice) {
        if (this.unitPrice > spice.unitPrice) {
            return 1;
        } else if (this.unitPrice == spice.unitPrice) {
            return 0;
        } else {
            return -1;
        }
    }

    public String toString() {
        return this.name + ": $" + this.unitPrice;
    }

}
```

This spice class was also used as a wrapper but for spices read in by our parser. Each spice contains a name, bulk cost, and quantity. Rather than having to retrieve the spice's unit price during our knapsack algorithm, I decided to calculate unit price in the constructor: the second we could. While this resulted in storing extra information, this allowed for ease when retrieving the unit price in our algorithm. Looking back at it, it might have made more sense to just store the unit price rather than the available cost summation.

1.4 Graph

```
import java.util.ArrayList;

public class Graph {

    public ArrayList<Vertex> vertices;
    public ArrayList<Link> links;
    private boolean isEmpty;
    private int highestIdFound;

    public Graph() {
        this.vertices = new ArrayList<>();
        this.links = new ArrayList<>();
        this.isEmpty = true;
    }

    public Vertex findVertexById(int id) {

        // Loop through vertices list to find a vertex with our desired id.
        Vertex v;
```

```
        for (int i = 0; i < vertices.size(); i++) {
            v = vertices.get(i);
            if (v.getId() == id) {
                return v;
            }
        }

        // Vertex does not exist.
        return new Vertex(-1);
    }

    // Given an id, creates a vertex and adds it to the graph.
    public void addVertex(int id) {
        this.isEmpty = false;

        if (id > this.highestIdFound) {
            this.highestIdFound = id;
        }

        if (findVertexById(id).id == id) { // if we can find it's id (as non negative 1)
            System.out.println("The vertex ID " + id + " already exists.");
            return;
        }

        Vertex vertex = new Vertex(id);

        vertices.add(vertex);
    }

    // Takes in two vertex id's and adds themselves to each others neighbor lists.
    public void addEdge(int vertex1, int vertex2, int weight) {
        // Create a link from vertex 1 to vertex 2.
        Link link = this.findVertexById(vertex1).addLink(vertex2, weight);
        this.links.add(link);
    }

    // Prints out adjacency list representation for each vertex in graph.
    // Initially, I was going to return it similar to createMatrix, however this was
    // a lot easier and more efficient as I do not have to loop once again for
    // display.
    public void printList() {

        // Grab each vertex associated with each other and set their coordinates to 1.
        for (Vertex v : this.vertices) {
            int vertexId = v.getId();

            System.out.println("[" + vertexId + "]" + " ->" + v.getLinks().toString());
        }

    }

    public boolean isEmpty() {
        return this.isEmpty;
    }

    public String toString() {
        String retStr = "";
        for (Vertex v : vertices) {
            retStr += "   Vertex: " + v.id + "\n";
            retStr += v.toString();
            retStr += "\n\n";
        }
        return retStr;
    }

```

```
85 }
```

Lastly, this is my graph class! In order to implement direction and weights, I added a list of links and changed my addEdge function to only add links in one direction. Because of my previous code's modularity, there were minimal changes needed to implement direction in the graph itself.

All in all, this assignment allowed me to better understand both Bellman-Ford's SSSP algorithm and the fractional knapsack algorithm by having experimented running it on custom data sets!