CMPT 435 - Fall 2022 - Dr. Labouseur

Assignment Five – Dynamic Programming and Greedy Algorithms

Genevieve Anderson
Genevieve.anderson1@marist.edu

December 10, 2022

1 Edge Class

My Edge class is where I have a constructor and a series of "getters" that pertain to the edges in the dynamic graph. First, I declare the variables for v1, v2, and the weight of the edge [2-4]. Then, I have a constructor [6] and a parameterized constructor [13] where I initialize the variables for whenever a new edge is created. Then, I have "getters" for v1 [20], v2 [24], and the weight of the edge [28] for whenever I need to obtain these values.

```
1  public class Edge {
2      Vertex v1;
3      Vertex v2;
4      int weight;
5
6      // Constructor
7      public Edge() {
8          v1 = null;
9          v2 = null;
10         weight = 0;
11     }
12
13     // Parameterized constructor
14     public Edge(Vertex v1, Vertex v2, int weight) {
15         this.v1 = v1;
16         this.v2 = v2;
17         this.weight = weight;
18     }
19
20     public Vertex getv1() {
```

```
21          return v1;
22      }
23
24      public Vertex getv2() {
25          return v2;
26      }
27
28      public int getWeight() {
29          return weight;
30      }
31  }
```

2 Vertex Class

My Vertex class is mostly recycled from assignment 4! First, I declare the variables for processed, id, neighbors, predecessor, and value [4-8]. Then inside my constructor [12] and parameterized constructor [21] I initialize all of my variables. Note that I initialize the value variable at infinity [17, 30] because each value in the dynamic graph starts at infinity, then the true weights are figured out in the algorithm.

First I have my "add()" function [35] which is for adding edges to the graph. Then, I have my "getPredecessor()" function [39] which is used when finding the path of the graph. Then I have my "getValue()" function [43] is for returning the value. My "setValue()" function [47] is used to set the value. My "setPredecessor()" function [51] is used to set the predecessor variable. My "getVertexByID()" function [56] is utilized whenever a vertex id is needed. My "neighbors()" function [61] returns all of the neighbors in the form of an ArrayList< *Edge* >. Lastly, I have a "toString()" function [66] which simply converts the object identifier of a vertex to the id we are interested in looking at.

```
1  import java.util.ArrayList;
2
3  public class Vertex {
4      boolean processed;
5      int id;
6      ArrayList<Edge> neighbors;
7      Vertex predecessor;
8      double value; // Starts at infinity
9
10
11     // Constructor
12     public Vertex() {
13         id = 0;
14         processed = false;
15         neighbors = new ArrayList<Edge>();
16         predecessor = null;
17         value = Double.POSITIVE_INFINITY;
18     }
19
20     // Parameterized Constructor, if we need to make a vertex with
       the provided id
21     public Vertex(int id) {
22         this.id = id;
23         processed = false;
24         neighbors = new ArrayList<Edge>();
25         predecessor = null;
26         if (id == 1) {
27             value = 0;
28         }
29         else {
30             value = Double.POSITIVE_INFINITY;
31         }
32     }
33
34     // Function for adding edge to vertex
```

```
35      public void add(Edge e) {
36          neighbors.add(e);
37      }
38
39      public Vertex getPredecessor() {
40          return predecessor;
41      }
42
43      public double getValue() {
44          return value;
45      }
46
47      public void setValue(double n) {
48          value = n;
49      }
50
51      public void setPredecessor(Vertex m) { // "m" is the node that
        we are going to set next equal to, for the pointer
52          predecessor = m; // I am using m so that the pointer is not
         null. the pointer will not be null until the end of the linked
         list.
53      }
54
55      // Function for getting the id
56      public int getVertexByID() {
57          return id;
58      }
59
60      // Function to return all neighbors of a vertex
61      public ArrayList<Edge> neighbors() {
62          return neighbors;
63      }
64
65      // toString for printing results
66      public String toString() {
67          return id + ""; // Converting the value being stored to a
        string so that the adjList can be printed and formatted
        correctly
68      }
69  }
```

3 Graph class

Now, I have my graph class. First, I declare array list or vertices and array list of edges [5, 6]. Then, I initialize these array lists in both a constructor [9] and a parameterized constructor [15]. I also have my "setGraph()" function [21] which sets up the graphs vertices and edges accordingly into their respective array lists.

Now, I have my "shortest()" function [27] which is the bellman ford algorithm for finding the shortest path. The algorithm works by overestimating the value of the path, in this case infinity, and "relaxes" the values until the shortest path is found. The algorithms loops through all of the edges updating the shortest path until the shortest path is confirmed. I also have my "relax()" function [42] which is called within the Bellman Ford algorithm to "relax" the value from infinity to eventually the shortest path. Lastly, I have my "path()" function [50] which provides the shortest path discovered from the Bellman Ford algorithm.

```
1  import java.util.ArrayList;
2  import java.util.Arrays;
3
4  public class Graph {
5      ArrayList<Vertex> vertexList;
6      ArrayList<Edge> edgeList;
7
8      // Constructor
9      public Graph() {
10         vertexList = null;
11         edgeList = null;
12     }
13
14     // Parameterized Constructor
15     public Graph(ArrayList<Vertex> vList, ArrayList<Edge> eList) {
16         vertexList = vList;
17         edgeList = eList;
18     }
19
20     // Setting the Graph
21     public void setGraph(ArrayList<Vertex> vertices, ArrayList<Edge
       > edges) {
22         vertexList = vertices;
23         edgeList = edges;
24     }
25
26     // Function for finding the shortest path - bellman ford
       algorithm
27     public boolean shortest() {
28         for (int i =0; i < vertexList.size() ; i++) {
29             for (Edge e: edgeList) {
30                 relax(e); // e contains first and second vertex
       along with weight
31             }
32         }
33         for (Edge e: edgeList) {
```

```
34              if (e.getv2().getValue() > e.getv1().getValue() + e.
    getWeight()) {
35                  return false;
36              }
37          }
38          return true;
39      }
40
41      // Function for relax - because it starts at infinity it must
    be "relaxed"
42      public void relax(Edge e) {
43          if (e.getv2().getValue() > e.getv1().getValue() + e.
    getWeight()) {
44              e.getv2().setValue(e.getv1().getValue() + e.getWeight()
    );
45              e.getv2().setPredecessor(e.getv1());
46          }
47      }
48
49      // Function for finding the path
50      public String path(Vertex v) {
51          String path = "";
52          if (v.getVertexByID() == 1) {
53              path += 1;
54              return path;
55          }
56          else {
57              path += path(v.getPredecessor()) + " -> " + v.
    getVertexByID();
58          }
59          return path;
60      }
61  }
```

4 Spice Class

In my spice class I create the framework for the instance of a spice. First, I declare variables for color, total price, quantity, and unit price [2-5]. I initialize these variables in both a constructor [8] and a parameterized constructor [16].

```
1  public class Spice {
2      String color;
3      double totalPrice;
4      int quantity;
5      double unitPrice; // Calculate myself
6
7      // Constructor
8      public Spice() {
9          color =  "";
10         totalPrice = 0;
11         quantity = 0;
12         unitPrice = 0;
13     }
14
15     // Parameterized constructor
16     public Spice(String color, double totalPrice, int quantity) {
17         this.color =  color;
18         this.totalPrice = totalPrice;
19         this.quantity = quantity;
20         unitPrice = totalPrice/quantity; // Calculating unit price
21     }
22 }
```

5 Main Program

In my main program I first attack the dynamic programming portion of the assignment. I use a try and catch statement for uploading the file. Now, I parse the text file. If a vertex is being added from the text file, then I create a new instance of a vertex [23] and add it to the array list of all vertices [24]. If an edge is added, I first figure out the weight of the edge [30-35]. Then, I create a new instance of an Edge by connecting it to its first and second vertices [37,42], and add it to the array list of all edges [38,43]. If a new graph is indicated from the text file, I create an instance of a graph [48, 58], Then I set the graph [49, 59], implement Bellman Ford [50, 60], and print my results [52, 62]. The running time for the Bellman Ford algorithm is $O(VE)$ where V is vertices in the graph and E is edges in the graph. This is because of the nested for loops that go through the vertices and the edges.

Next, I move on to the spices portion of the assignment. I use a try and catch statement for uploading the file. I parse the file to obtain values for color, total price, and quantity of the spices [84-87]. I also set these values to appropriate variables [90-92]. Then, I sort and add each spice to the array list of all spices using a series of if else statements [96-105]. Then, the knapsack lines of the text file are dealt with. This is where I call the greedy algorithm [110]. The running time for the fractional knapsack algorithm is typically $O(nlogn)$ accounting for the quick sort. However, I manually sorted my spices using if else statements so the running time without quick/merge sort would be $O(n)$ accounting for the loop in the fractional knapsack algorithm. At the bottom of my main program I have my "printResults()" function [119]. This is where I format how the output will be printed. Lastly, I have my "greedy()" algorithm [125]. I initialize current, total, and scoops variables [127-129]. Then inside a for loop I fill the knapsack til it reaches its capacity [131]. Inside a series of if statements I fill up the knapsack with spices til it is full. Inside the first if statement [133], the current spice of the highest unit price is taken while the total is updated and the scoops are incremented accordingly. Inside the inner if statement [137], the current spice that is being taken is incremented and the scoops is reset to 0.

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4  import java.util.*;
5
6  public class main {
7      public static void main(String[] args) {
8
9          // Dynamic programming
10
11         System.out.println("
       ------------------------------------------------------------
       Dynamic Programming
       ------------------------------------------------------------");
12
13         ArrayList<Vertex> vertices = new ArrayList<Vertex>(); //
```

```
    vertices that will be provided to the graph
        ArrayList<Edge> edges = new ArrayList<Edge>();

        try { //Trying to find the file
            File file = new File("graphs2.txt");
            Scanner sc = new Scanner(file);
            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                String[] parse = item.split(" ");
                if (parse[0].equals("add") && parse[1].equals("
    vertex")) {
                    Vertex v = new Vertex(Integer.parseInt(parse
    [2])); // Getting the vertex number and creating a new vertex
    based off of it
                    vertices.add(v); // Adding the vertex to the
    array list of all the vertices
                }
                else if (parse[0].equals("add") && parse[1].equals(
    "edge")) {
                    int firstVertex = Integer.parseInt(parse[2]);
    // First vertex the edge will be connected to
                    int secondVertex = Integer.parseInt(parse[4]);
    // Second vertex the edge will be connected to
                    int weight = 0; // Initialize weight
                    if (parse[5].equals("")) {
                        weight = Integer.parseInt(parse[6]); //
    Weight of the edge
                    }
                    else {
                        weight = Integer.parseInt(parse[5]); //
    Weight of the edge
                    }
                    if (firstVertex == 0) {
                        Edge e = new Edge(vertices.get(firstVertex)
    , vertices.get(secondVertex), weight); // Making an edge
                        edges.add(e); // Adding the new edge to the
     array list of edges
                        vertices.get(firstVertex).add(e); //
    Getting the edges that are correlated with each vertex;
                    }
                    else {
                        Edge e = new Edge(vertices.get(firstVertex
    - 1), vertices.get(secondVertex - 1), weight);
                        edges.add(e);
                        vertices.get(firstVertex - 1).add(e);
                    }
                }
                else if (parse[0].equals("new") && parse[1].equals(
    "graph") && vertices.size() > 0) {
                    Graph graph = new Graph();
                    graph.setGraph(vertices, edges);
                    graph.shortest();
                    System.out.println("------------new graph
    --------------");
                    printResults(vertices, graph); // Results
                    vertices.clear(); // So that the graphs aren't
    added on top of each other
```

```
                    edges.clear();
                }
            }
            // For the last graph
            Graph graph = new Graph();
            graph.setGraph(vertices, edges);
            graph.shortest();
            System.out.println("------------new graph--------------
");
            printResults(vertices, graph); // Results
            vertices.clear(); // So that the graphs aren't added on
 top of each other
            edges.clear();
        }
        catch (FileNotFoundException e) { // If we cant find the
file
            e.printStackTrace();
        }

        // Spices

        System.out.println("
------------------------------------------------------
Greedy Algorithm
------------------------------------------------------");

        ArrayList<Spice> spices = new ArrayList<Spice>(); // Hold
all the spices

        try { // Trying to find the file
            File file = new File("spice.txt");
            Scanner sc = new Scanner(file);

            while (sc.hasNextLine()) {
                String item = sc.nextLine();
                if (item.startsWith("s")) {
                    // Parsing the spice file
                    String[] parse = item.split(";");
                    String[] color = parse[0].split(" ");
                    String[] totalPrice = parse[1].split(" ");
                    String[] quantity = parse[2].split(" ");

                    // Setting the variables to their values
                    String c = color[color.length-1];
                    double tP = Double.parseDouble(totalPrice[
totalPrice.length-1]);
                    int q = Integer.parseInt(quantity[quantity.
length -1]);

                    // Adding spices to the array list of spices
                    Spice s = new Spice(c, tP, q);
                    if (spices.size() == 0) {
                        spices.add(s);
                    }
                    else if (spices.get(0).unitPrice > s.unitPrice)
 {
                        spices.add(s); // Since it is greater than
```

```
    the greatest spice in the list, it should be added in the
    beginning
                    }
                    else {
                        spices.add(0,s);
                    }
                }
                else if (item.startsWith("k")) {
                    String[] parse = item.split(" ");
                    int cap = Integer.parseInt(parse[parse.length
    -1].substring(0,parse[parse.length-1].length()-1)); // Parsing
    to get the knapsack capacity

                    System.out.println(greedy(cap, spices)); //
    Algorithm
                }
            }
        }
        catch (FileNotFoundException e) { // If we cant find the
    file
            e.printStackTrace();
        }
    }

    public static void printResults(ArrayList<Vertex> vertices,
    Graph g) {
        for (int i = 1; i < vertices.size(); i ++) {
            System.out.println("1 -> " + vertices.get(i) + " cost
    is " + vertices.get(i).getValue() + "; path is " + g.path(
    vertices.get(i)));
        }
    }

    public static String greedy(int cap, ArrayList<Spice> spices) {
        String result = "knapsack of capacity " + cap + " is worth
    ";
        int current = 0; // For tracking the current spice
        int total = 0;
        int scoops = 0;
        String scoopString = "";
        for (int i = 0; i < cap; i++) { // Fill the knapsack til it
     reaches it's capacity
            // Scooping the spices
            if (current < spices.size()) {
                Spice currentSpice = spices.get(current); //
    Current spice
                total += currentSpice.unitPrice;
                scoops++;
                if (currentSpice.quantity == scoops || i + 1 == cap
    ) {
                    current++;
                    scoopString += scoops + " scoop of " +
    currentSpice.color + ", ";
                    scoops = 0; // Re setting the scoops
                }
            }
            else {
```

```
                break; // For the last case because max quantity is 20
            }
        }
        result += total + " quatloos and contains " + scoopString.substring(0, scoopString.length() - 2) + ".";
        return result;
    }
}
```