Assignment Five

Shannon Cordoni

Shannon.Cordoni@Marist.edu

December 11, 2021

1 Problem One: Graphs and Greedy Algorithms

1.1 The Data Structure

Given multiple text files we were to create different directed graph representations and a fractional knapsack representation using Greedy Algorithms. First taking a list of graph instructions we were to create vertices and edges to form a directed graph in the form of linked objects to run the Bellman-Ford Algorithm to determine the single source shortest path. Then taking a list of spices to run the fractional knapsack representation using Greedy Algorithms.

1.2 Main Class

1.2.1 Description

This class is where most of our work is done, it contains multiple scanners to read in our multiple files. These files include the *graphs2.txt* file, and *spice.txt* file. Using these files we input each line of the *graphs2.txt* into an instruction array to be split up into a 2 dimensional array to access each index. We then use this split array to create the linked object representation for each graph. The *spice.txt* file was also read into an array to be passed to the *spiceItUp* method to create spice and knapsack objects to run the Greedy Algorithm. The methods contained in this class preform all these operations and more.

```
1
2  /*
3   *
4   * Assignment 5
5   * Due Date and Time: 12/10/21 before 12:00am
6   * Purpose: to implement directed graphs and greedy algorithm structures.
7   * Input: The user will be inputting a file containing a list of edges and vertices.
8   * Output: The program will output direct graph shortest paths and greedy algorithms.
9   * @author Shannon Cordoni
10  *
11  */
12
13 import java.io.File;
14 import java.io.FileNotFoundException;
```

```
15 import java.util.ArrayList;
16 import java.util.Arrays;
17 import java.util.*;
18 import java.util.Scanner;
19 import java.util.Collections;
20
21
22
23 public class Assignment5Cordoni {
24
25     //Declare keyboard
26     static Scanner keyboard = new Scanner(System.in);
27
28     public static void main(String[] args) {
29
30         //Declare and initialize variables
31         String line;
32
33
34         String[] spicearray = new String[14];
35         String[] instructionarray = new String[88];
36         String[][] splitinstructionarray = new String[88][17];
37
38         //Reads in the spice items
39         //create new file object
40         File myFile = new File("spice.txt");
41
42         try
43         {
44             //create scanner
45             Scanner input = new Scanner(myFile);
46             line = null;
47
48             int i = 0;
49
50             //while there are more lines in the file it inputs them into a spice array
51             while(input.hasNext())
52             {
53                 spicearray[i] = input.nextLine();
54                 i++;
55             }//while
56
57             input.close();
58
59         }//try
60
61         //General error message
62         catch(Exception ex)
63         {
64             System.out.println("Something went wrong");
65             ex.printStackTrace();
66         }//catch
67
68         //Reads in the graph file to create graphs
69         //create new file object
70         File myFile1 = new File("graphs2.txt");
71
72         try
73         {
74             //create scanner
75             Scanner input = new Scanner(myFile1);
76             line = null;
77
78             int i = 0;
79
```

```
80              //while there are more lines in the file it inputs them into an instruction array
81              while(input.hasNext()){
82
83                  instructionarray[i] = input.nextLine();
84                  i++;
85              }//while
86
87              input.close();
88
89          }//try
90
91          //General error message
92          catch(Exception ex)
93          {
94              System.out.println("Something went wrong");
95              ex.printStackTrace();
96          }//catch
97
98          //Spices!
99
100         //Print to check array
101         for (int i = 0; i < spicearray.length; i++){
102             //System.out.println(spicearray[i]);
103         }//for
104
105         //pass the spice array to the spiceItUp method to create the spice objects
106         spiceItUp(spicearray);
107
108
109         //Graphs!!
110
111         //Print to check array
112         for (int i = 0; i < instructionarray.length; i++){
113            //System.out.println(instructionarray[i]);
114         }//for
115
116         //split up into 2D array
117         for (int i = 0; i < splitinstructionarray.length; i++){
118             for( int j = 0; j < splitinstructionarray[i].length; j++){
119             splitinstructionarray[i] = instructionarray[i].split(" ");
120             }//for j
121         }//for
122
123         //pass the instruction array to the graphItUp method to create the vertex and edge
124         //objects for each graph
125         graphItUp(splitinstructionarray);
126
127     }//main
128
129     //This method creates the linked objects of the directed graph
130     //This also includes making a new graph object for each graph in the file
131     //along with creating the edges and vertexes for each of these graphs
132     public static void graphItUp(String[][] instructions) {
133
134         GraphCordoni graphCordoni  = null; // new GraphCordoni();
135         int k = 1;
136
137         //create
138         for (int i = 0; i < instructions.length; i++){
139
140             //if the line reads new we create a new graph
141             if(instructions[i][0].compareToIgnoreCase("new")==0){
142
143                 //if the graph is null we know its the first graph in the file
144                 if (graphCordoni == null){
```

```
                    graphCordoni = new GraphCordoni();
                    System.out.println("  ");
                    System.out.println("  ");
                    System.out.println("Graph " + k);
                    k++;
                }//if

                //else we know that we need to clear the graph object so we run the bellman
                //ford method and reset graphCordoni
                //to hold the new graph
                else{

                    bellmanFord(graphCordoni, graphCordoni.edges.get(0).getWeight(),
                    graphCordoni.vertexes.get(0));

                    graphCordoni = new GraphCordoni();
                    System.out.println("  ");
                    System.out.println("  ");
                    System.out.println("Graph " + k);
                    k++;

                }//else

            }//if

            //here we add the vertexes and edges
            else if (instructions[i][0].compareToIgnoreCase("add")==0){



                //create new vertex and set id to add to vertex array
                if(instructions[i][1].compareToIgnoreCase("vertex")==0){

                    VertexCordoni vertex = new VertexCordoni();

                    vertex.setId(instructions[i][2]);

                    //add the vertex to the vertex array list
                    graphCordoni.vertexes.add(vertex);


                }//if

                //create new edge and set id to add to edge array
                else if(instructions[i][1].compareToIgnoreCase("edge")==0){

                    EdgeCordoni edge = new EdgeCordoni();

                    for (int j = 0; j < graphCordoni.vertexes.size(); j++){

                        //set the "from" of the edge
                        if (graphCordoni.vertexes.get(j).getId()
                        .compareToIgnoreCase(instructions[i][2])==0){
                            edge.setFrom(graphCordoni.vertexes.get(j));
                        }//if

                        //set the "to" of the edge
                        if (graphCordoni.vertexes.get(j).getId()
                        .compareToIgnoreCase(instructions[i][4])==0){
                            edge.setTo(graphCordoni.vertexes.get(j));
                        }//if

                    }//for


```

```
                    //set the weight of the edge
                    if ((instructions[i][5].compareToIgnoreCase("") == 0)){
                        edge.setWeight(Integer.parseInt(instructions[i][6]));
                    }

                    else{
                        edge.setWeight(Integer.parseInt(instructions[i][5]));
                    }//else

                    //add the edge to the edge array list
                    graphCordoni.edges.add(edge);

                }//else if
            }//else if
        }//for i

        /*
        for(int i = 0; i < graphCordoni.edges.size(); i++){

            System.out.println(" ");
            System.out.println("From: "  + graphCordoni.edges.get(i).getFrom().getId());
            System.out.println("To: "  + graphCordoni.edges.get(i).getTo().getId());
            System.out.println("Weight: "  + graphCordoni.edges.get(i).getWeight());
            System.out.println(" ");

        }//for
        // */

        //call the bellman ford algorithm
        bellmanFord(graphCordoni, graphCordoni.edges.get(0).getWeight(),
        graphCordoni.vertexes.get(0));
    }//graph it up


    //This method preforms the bellman ford algorithm to find the shortest path
    public static boolean bellmanFord(GraphCordoni graph, int weight, VertexCordoni source) {

        //System.out.println("bellman ford");

        //call the single source method to set the distances to positive infinity
        singlesource(graph, source);

        boolean value = false;

        //for each vertex we loop through the edges to relax them
        for (int i = 0; i < graph.vertexes.size() - 1; i++){

            for (int j = 0; j < graph.edges.size(); j++){

                //System.out.println("relax");

                //here we call relax to "relax" each of the vertex distances down to their
                //lowest possible distance
                relax(graph.edges.get(j).getFrom(), graph.edges.get(j).getTo(),
                graph.edges.get(j).getWeight());

                /*
                System.out.println(" ");
                System.out.println("From: "  + graph.edges.get(j).getFrom().getId());
                System.out.println("To: "  + graph.edges.get(j).getTo().getId());
                System.out.println("Weight: "  + graph.edges.get(j).getWeight());
                System.out.println(" ");

                */

```

```
275             }//for
276 
277         }//for
278 
279         //here we loop through each edge to see if the from distance is greated than the to
280         //distances plus the weight.
281         for (int i = 0; i < graph.edges.size(); i++){
282 
283             //System.out.println(" ");
284             //System.out.println("From Distance:"+graph.edges.get(i).getFrom().getDistance());
285             //System.out.println("To Distance: "  + graph.edges.get(i).getTo().getDistance());
286             //System.out.println("Weight: "  + graph.edges.get(i).getWeight());
287             //System.out.println(" ");
288 
289 
290             if(graph.edges.get(i).getFrom().getDistance() >
291             graph.edges.get(i).getTo().getDistance() + graph.edges.get(i).getWeight() ){
292                 value = false;
293                 shortestPath(value, graph);
294                 return value;
295 
296             }//if
297 
298         }//for
299 
300         value = true;
301 
302         shortestPath(value, graph);
303 
304         return value;
305 
306     }//BellmanFord
307 
308     //This method sets the initial single source
309     public static void singlesource(GraphCordoni graph, VertexCordoni source) {
310 
311         //System.out.println("single source ");
312 
313         //here we loop through each of the vertexes setting their distance to positive
314         //infinity
315         for (int i = 0; i < graph.vertexes.size(); i++){
316 
317             graph.vertexes.get(i).setDistance(Double.POSITIVE_INFINITY);
318 
319             graph.vertexes.get(i).setPrevious(null);
320 
321         }//for
322 
323         //set the distance of the source vertex to 0
324         source.setDistance(0.0);
325 
326     }//singlesource
327 
328     //This method "relaxes" the vertex distance to determine the shortest path
329     public static void relax(VertexCordoni vertexEdgeFrom, VertexCordoni vertexEdgeTo,
330                              Integer weight) {
331 
332         //System.out.println("relax");
333 
334         //System.out.println(" ");
335         //System.out.println("Vertex"+vertexEdgeFrom.getId()+" distance:"
336         +vertexEdgeFrom.getDistance());
337         //System.out.println("Vertex"+vertexEdgeTo.getId()+" distance:"
338         +vertexEdgeTo.getDistance());
339         //System.out.println("Weight: "  + weight);
```

```
340         //System.out.println(" ");
341
342         //if the to vertex distance is greater than the from vertex distance then we set the
343         //to distance to
344         //be the from vertex distance plus the weight
345         if(vertexEdgeTo.getDistance() > vertexEdgeFrom.getDistance() + weight){
346
347             vertexEdgeTo.setDistance(vertexEdgeFrom.getDistance() + weight);
348
349             vertexEdgeTo.setPrevious(vertexEdgeFrom);
350
351             //System.out.println("Relax vertex " + vertexEdgeTo.getId() + " to "
352             + vertexEdgeTo.getDistance());
353
354         }//if
355
356     }//relax
357
358     //This method prints out the shortest path
359     public static void shortestPath(boolean value, GraphCordoni graph) {
360
361         //here we loop through the vertexes and print out the shortest path from the source
362         //vertex to the
363         //desired vertex
364         for(int i = 0; i < graph.vertexes.size(); i++){
365
366             VertexCordoni current = null;
367
368             ArrayList <String> path = new ArrayList <String>();
369
370             current = graph.vertexes.get(i);
371
372             System.out.println(" ");
373             System.out.println(" ");
374             System.out.println("From " + graph.vertexes.get(0).getId() + " to " +
375             graph.vertexes.get(i).getId() + " the cost is "
376             + graph.vertexes.get(i).getDistance());
377
378             System.out.println("The path is ");
379
380
381             //if the previous is set to null then we are at the source vertex
382             if(graph.vertexes.get(i).getPrevious() == null){
383
384                 System.out.print(graph.vertexes.get(i).getId());
385
386
387             }//if
388
389             //else we start at the desired vertex and work our way backwards to determine our
390             //shortest path. basing this off of the answers for graph 1 I was able to get
391             //this working for all the paths
392             //except for the last vertex in each graph
393             else{
394
395                 int j  = i;
396
397                 path.add(graph.vertexes.get(j).getId());
398                 //System.out.println(graph.vertexes.get(j).getId());
399
400                 while(graph.vertexes.get(j).getPrevious().getId() !=
401                     graph.vertexes.get(0).getId()){
402
403
404                     //System.out.println(graph.vertexes.get(j).getPrevious().getId());
```

```
                    path.add(graph.vertexes.get(j).getPrevious().getId());
                    j++;

                    //paths are correct besides last path for each graph
                    if(j == graph.vertexes.size()){
                        break;
                    }//if


                }//while

                //System.out.println(graph.vertexes.get(0).getId());
                path.add(graph.vertexes.get(0).getId());

                Collections.reverse(path);

                for(int k = 0; k < path.size(); k++){

                    System.out.print(path.get(k));
                }//for k

            }//else

        }//for

    }//shortestPath


    //Spices!

    //This method creates the spice object
    public static void spiceItUp(String[] spices) {

        System.out.println("Spice Hesit!");
        System.out.println(" ");

        String[][] splitspicearray = new String[9][9];

        int j = 0;

        for (int i = 0; i < spices.length; i++){

            //if the line starts with spice then we split by ; and then by space
            if (spices[i].startsWith("spice")){

                //System.out.println(Arrays.toString(spices[i].split(";| ")));

                splitspicearray[j] = spices[i].split(";| ");
                j++;

            }//if

            //if the line starts with knapsack then we split by ; and then by space
            else if (spices[i].startsWith("knapsack")){

                //System.out.println(Arrays.toString(spices[i].split(";| ")));

                splitspicearray[j] = spices[i].split(";| ");
                j++;

            }//if


        }//for

```

```
        //print to check array
        for(int i  = 0; i < splitspicearray.length; i++){
            for(int k = 0; k < splitspicearray[i].length; k++){

                //System.out.println(splitspicearray[i][k]);

            }//for j
        }//for i

        //pass the created array to the create spice method to create the spice objects
        createSpice(splitspicearray);
    }//spiceitup

    //this method creates the spice objects
    public static void createSpice(String[][] spices) {

        ArrayList <SpiceCordoni> spicelist = new ArrayList <SpiceCordoni>();

        //increment index to create spice array
        for (int i = 0; i < spices.length; i++){

            //if the line starts with spice then we create a new spice object
            if (spices[i][0].compareToIgnoreCase("spice")==0){
                //System.out.println(" new spice ");

                SpiceCordoni spice = new SpiceCordoni();

                //System.out.println("color: " + spices[i][3]);

                spice.setColor(spices[i][3]);

                //if the line starts with total price then we set the total price of the
                //spice object along with the qty of the spice object
                if(spices[i][8].trim().startsWith("total_price")){

                    //System.out.println("price 1: " + spices[i][11].trim());

                    spice.setPrice(Double.parseDouble(spices[i][11].trim()));

                    //System.out.println("qty 1: " + spices[i][16]);

                    spice.setQty(Integer.parseInt(spices[i][16]));

                }//if

                else if(spices[i][6].trim().startsWith("total_price")){
                    //System.out.println("price 2: " + spices[i][8].trim());

                    spice.setPrice(Double.parseDouble(spices[i][8].trim()));

                    //System.out.println("qty 2: " + spices[i][13]);
                    spice.setQty(Integer.parseInt(spices[i][13]));

                }//else

                else if(spices[i][7].trim().startsWith("total_price")){
                    //System.out.println("price 3: " + spices[i][9].trim());

                    spice.setPrice(Double.parseDouble(spices[i][9].trim()));

                    //System.out.println("qty 3: " + spices[i][14]);
                    spice.setQty(Integer.parseInt(spices[i][14]));
                }//else

                else if(spices[i][5].trim().startsWith("total_price")){
```

```
                    //System.out.println("price 4: " + spices[i][7].trim());

                    spice.setPrice(Double.parseDouble(spices[i][7].trim()));

                    //System.out.println("qty 4: " + spices[i][12]);
                    spice.setQty(Integer.parseInt(spices[i][12]));
                }//else

                //here we dd the spice object to the spice list array
                spicelist.add(spice);
            }// if

        }//for i


        //print spice to check
        //System.out.println(spicelist.toString());

        //pass the spicelist and spices array to the spice unit price method
        spiceUnitPrice(spicelist, spices);

    }//createSpice

    //This method creates each spice's unit price
    public static void spiceUnitPrice(ArrayList<SpiceCordoni> spicelist, String[][] spices) {

        //Create unit price for each spice
        for(int i = 0; i < spicelist.size(); i++){

            spicelist.get(i).setUnitPrice(spicelist.get(i).getPrice()/spicelist.get(i)
            .getQty());

        }//for

        //pass the spice list and spices array to the sort method
        sort(spicelist, spices);
    }//spiceUnitPrice

    //This method sorts spices from high to low unit price
    public static void sort(ArrayList<SpiceCordoni> spicelist, String[][] spices)
    {

        //reverse the spicelist to put them in order from highest to lowest unit price
        Collections.reverse(spicelist);

        //Check unit price for each spice
        for(int i = 0; i < spicelist.size(); i++){

            //System.out.println("Spice " + spicelist.get(i).getColor());
            //System.out.println("Price " + spicelist.get(i).getUnitPrice());

        }//for

        //pass the spicelist and spices array to the create knapsack method
        createKnapsack(spicelist, spices);

    }//sort

    //This method creates the knapsacks
    public static void createKnapsack(ArrayList<SpiceCordoni> spicelist, String[][] spices) {

        //create a knapsack arraylist
        ArrayList <KnapsackCordoni> knapsacklist = new ArrayList <KnapsackCordoni>();

        //increment index to create spice array
```

```
600         for (int i = 0; i < spices.length; i++){
601 
602             //if the line starts with knapsack then we create a new knapsack object
603             if (spices[i][0].compareToIgnoreCase("knapsack")==0){
604                 //System.out.println(" new knapsack ");
605 
606                 KnapsackCordoni knapsack = new KnapsackCordoni();
607 
608 
609                 //set the capcity of the knapsack
610                 if (spices[i][3].trim().compareToIgnoreCase("")==0){
611 
612                     knapsack.setCapacity(Integer.parseInt(spices[i][4].trim()));
613 
614                     knapsacklist.add(knapsack);
615 
616                 }//if
617 
618                 else{
619 
620                     knapsack.setCapacity(Integer.parseInt(spices[i][3].trim()));
621 
622                     knapsacklist.add(knapsack);
623 
624                 }//else
625             }//if
626 
627         }//for i
628 
629         //Create unit price for each spice
630         for(int i = 0; i < knapsacklist.size(); i++){
631 
632             //System.out.println("Capacity " + knapsacklist.get(i).getCapacity());
633 
634         }//for
635 
636         //pass the spicelist and knapsack list to the fill knapsack method
637         fillKnapsack(spicelist, knapsacklist);
638 
639     }//createKnapsack
640 
641     //This method fills the knapsacks
642     public static void fillKnapsack( ArrayList<SpiceCordoni> spicelist,
643                                      ArrayList<KnapsackCordoni> knapsacklist) {
644 
645         int knapsackcapacity = 0 ;
646 
647         //for each knapsack in the list we fill it according to the greedy algorithm method
648         for(int i = 0; i < knapsacklist.size(); i++){
649 
650             //get the knapsack capacity
651             knapsackcapacity = knapsacklist.get(i).getCapacity();
652 
653             //initialize variables so that they reset for each knapsack
654             double worth = 0;
655             int quantity = 0;
656             int scoop = 0;
657             String[] color = new String[20];
658             int orangescoops = 0;
659             int bluescoops = 0;
660             int greenscoops = 0;
661             int redscoops = 0;
662 
663             int k = 0;
664 
```

```
            //for each spice in the spice list we get the qty and while the quantity is not
            //0 and the scoop count
            //if less than the knapsack capacity then we add the spice and it's specific
            //quantity to the knapsack
            for(int j = 0; j < spicelist.size(); j++){

                quantity = spicelist.get(j).getQty();

                //int k = 0;
                while((quantity != 0)&&(scoop < knapsackcapacity)){

                    quantity = quantity - 1;
                    worth = worth + spicelist.get(j).getUnitPrice();
                    color[k] = spicelist.get(j).getColor();
                    //System.out.println(color[k]);
                    scoop++;

                    k++;

                    //System.out.println("k " + k);

                }//while

            }//for j

            //print out each knapsack and what its worth
            System.out.println("Knapsack of capacity " + knapsackcapacity + " is worth "
                            + worth + " quatloos and contains ");

            //loop through the color array to see which colors (spices) are in each knapsack
            for(int l = 0; l < color.length; l++){
                //System.out.println(color[l]);
            }

            //if a color appears in the knapsack then we up the scoopcount for that color to
            //determine how many scoops of each spice are in the knapsack
            for(int l = 0; l < color.length; l++){

                if((color[l] !=null)&&(color[l].compareToIgnoreCase("orange")==0)){
                    orangescoops++;
                }
                else if((color[l] !=null)&&(color[l].startsWith("blue"))){
                    bluescoops++;
                }
                if((color[l] !=null)&&(color[l].startsWith("green"))){
                    greenscoops++;
                }
                if((color[l] !=null)&&(color[l].startsWith("red"))){
                    redscoops++;
                }

            }//for

            //here we print out each color and the amount of scoops they have in each knapsack.
            for(int l = 0; l < color.length; l++){

                //System.out.println("orange scoops: " + orangescoops);

                //System.out.println("blue scoops: " + bluescoops);

                //System.out.println("green scoops: " + greenscoops);

                //System.out.println("red scoops: " + redscoops);

                //we go through each color like this because this way we get a hold of every
```

```
                //color that appears
                //in each knapsack
                if(orangescoops != 0){
                    System.out.println(orangescoops + " scoop(s) of orange");

                    if(bluescoops != 0){
                        System.out.println(bluescoops + " scoop(s) of blue");

                        if(greenscoops != 0){
                            System.out.println(greenscoops + " scoop(s) of green");

                            if(redscoops != 0){
                                System.out.println(redscoops + " scoop(s) of red");

                            }//if
                        }//if
                    }//if

                    break;
                }//if
                else if(bluescoops != 0){
                    System.out.println(bluescoops + " scoop(s) of blue");

                    if(greenscoops != 0){
                        System.out.println(greenscoops + " scoop(s) of green");

                        if(redscoops != 0){
                            System.out.println(redscoops + " scoop(s) of red");

                        }//if
                    }//if

                    break;
                }
                if(greenscoops != 0){
                    System.out.println(greenscoops + " scoop(s) of green");

                    if(redscoops != 0){
                        System.out.println(redscoops + " scoop(s) of red");

                    }//if

                    break;
                }
                if(redscoops != 0){
                    System.out.println(redscoops + " scoop(s) of red");
                    break;
                }

            }//for

        }//for i

    }//fillKnapsack

}//Assignment5Cordoni
```

1.2.2 Description of Main Code

The main class above consists of different methods to help create directed graphs and their bellman-ford representations. Along with running a spice heist using Greedy Algorithms. The good parts of the code first include the file sections. While reading the different *txt* files we input each line into arrays for each file. For

the Graph representations we create an instruction array to pass to our graph methods to more easily create the graph representations. Along with passing each line of the spice array from the *spice.txt* file into our spice methods to perform the spice heist. Then to keep everything out of the main method, different methods were used to help organize the code better. These methods include the *graphItUp*, *bellmanFord*, *singlesource*, *relax*, *shortestPath*, *spiceItUp*, *createSpice*, *spiceUnitPrice*, *sort*, *createKnapsack*, and *fillKnapsack* method.

The *graphItUp* method takes in a 2 dimensional set of instructions from the main method and reads it line by line. If the line starts with "new" we create a new graph object to create vertexes and edges for. If the line starts with "vertex" then we create a new vertex for the graph and add it to the vertex array list of the graph. If the line starts with "edge" we get the "to" and "from" vertex and create a new edge for the graph and add it to the edge array list of the graph. We then call the bellman ford algorithm, and pass it the graph, the source vertexes weight, and the source vertex.

The *bellmanFord* method takes in a graph, a source vertex weight and a source vertex. First we call the *singlesource* method to set the distance of each vertex to positive infinity. Then we go through each edge for each vertex and call the *relax* method to in a sense "relax" the vertex distances down from positive infinity to determine the shortest path from the source vertex to each of the other vertexes in the graph. We then loop through the edges to determine if the "from" vertex distance is greater then the "to" vertex distance plus the weight of the edge. We cycle through until this returns false or we make it all the way through the edge array list. Either way we then call the The *shortestPath* method to then print out the shortest path from the source vertex to the desired vertex and the cost of this path.

The *singlesource* method takes in a graph and a source vertex and loops through each vertex in the graph setting its distance to positive infinity and its previous vertex to null. We also set the source vertex distance to be 0.

The *relax* method looks to see if the "to" vertex distance is greater than the "from" vertex distance plus the weight of the edge. If it is we set the "to" vertex distance to be equal to the "from" vertex distance plus the weight of the edge, and the previous vertex to be equal to the "from" vertex. If not then we go back to the bellman ford algorithm and call relax for the next set of vertices.

The *shortestPath* takes in a graph and a boolean value, the method goes through each vertex in the graphs array list and prints out the shortest path from the source vertex to the next vertex in the array list. If the previous of the current vertex is null then we are at the source vertex and we print the source vertex. Else we start at the current index and go through the previous vertexes until we end up back at the source vertex to print out the shortest path.

The *spiceItUp* method takes in an array of spice and knapsack information and we loop through this array, if the line starts with "spice" or "knapsack" then we add it to a separate array to be split up to create spice and knapsack objects. We then pass this split array to the *createSpice* method.

The *createSpice* method goes through the spice information and sets the color, price, and qty of each new spice object. We then add each of the new spice objects to an array list to be passed along to the next methods, such as the *spiceUnitPrice* method.

The *spiceUnitPrice* method takes in the spice array list and the 2 dimensional array of spice information and we create and set the unit price for each spice object.

The *sort* method takes in the spice array list and the 2 dimensional array of spice information and sorts the unit price from high to low. We then passed this sorted list and the 2 dimensional array of spice information to the *createKnapsack* method.

The *createKnapsack* method takes in the spice array list and the 2 dimensional array of spice information and creates the knapsack objects based off of the knapsack information in the spice array. After creating the knapsack objects we pass the spice array list and the knapsack array list to the $fillKnapsack$ method.

The $fillKnapsack$ method takes in the spice array list and the knapsack array list. First we start by looping through the knapsack array list and for each knapsack we go through the spice array list and while the quantity of the spice is not equal to zero and the scoop count is less than the knapsack capacity we calculate the worth, color, and scoop count of each spice needed to fill the knapsack. We then print out the capacity, worth, and scoop count of each spice needed to fill each knapsack in the knapsack array list.

1.3 Edge Class

1.3.1 Description

For the creation of each edge in the directed graph we passed the edge data to the edge class to create each edge object.

```
/*
 *
 * Assignment 5
 * Due Date and Time: 12/10/21 before 12:00am
 * Purpose: to implement directed graphs and greedy algorithm structures.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output direct graph shortest paths and greedy algorithms.
 * @author Shannon Cordoni
 *
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Arrays;
import java.util.ArrayList;

public class EdgeCordoni
{
   /**
    * Declare Variables
    */
    private VertexCordoni myTo;
    private VertexCordoni myFrom;
    private int myWeight;


    /**
    * The default Constructor for EdgeCordoni
    */
   public EdgeCordoni()
   {
   myTo = null;
   myFrom = null;
```

```
34      myWeight = 0;
35      }//Edge Cordoni
36
37       /**
38       * The full constructor for NodeCordoni
39       * @param newData the incoming data of the item
40       */
41       public EdgeCordoni(VertexCordoni newTo, VertexCordoni newFrom, Integer newWeight)
42       {
43           myTo = newTo;
44           myFrom = newFrom;
45           myWeight = newWeight;
46
47       }//EdgeCordoni
48
49       /**
50       * the setter for the edge data
51       * @param newTo the incoming data of the item
52       */
53       public void setTo(VertexCordoni newTo)
54           {myTo = newTo;} //set data
55
56       /**
57        * The getter for the edge data
58        * @return the incoming data of the item
59        */
60       public VertexCordoni getTo()
61           {return myTo;}//get data
62
63       /**
64       * the setter for the edge data
65       * @param newFrom the incoming data of the item
66       */
67       public void setFrom(VertexCordoni newFrom)
68           {myFrom = newFrom;} //set data
69
70       /**
71        * The getter for the edge data
72        * @return the incoming data of the item
73        */
74       public VertexCordoni getFrom()
75           {return myFrom;}//get data
76
77       /**
78       * the setter for the edge data
79       * @param newWeight the incoming data of the item
80       */
81       public void setWeight(Integer newWeight)
82           {myWeight = newWeight;} //set data
83
84       /**
85        * The getter for the edge data
86        * @return the incoming data of the item
87        */
88       public Integer getWeight()
89           {return myWeight;}//get data
90
91
92   }//edge Cordoni
```

1.3.2 Description of Edge Code

This code for the Edge Class was created by previous knowledge working with the Node Class. Using the same set up each Edge was given a to, from, and weight. Getters and setters were created for each to make creating the edges run more smoothly, along with adding them to the edge array list for each graph.

1.4 Graph Class

1.4.1 Description

For each graph in the text file, a graph object was created to organize all the vertices and edges of each graph.

```
/*
 *
 * Assignment 5
 * Due Date and Time: 12/10/21 before 12:00am
 * Purpose: to implement directed graphs and greedy algorithm structures.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output direct graph shortest paths and greedy algorithms.
 * @author Shannon Cordoni
 *
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Arrays;
import java.util.ArrayList;

public class GraphCordoni
{
    /**
    * Declare Variables
    */

    public  ArrayList <VertexCordoni> vertexes = new ArrayList <VertexCordoni>();
    public  ArrayList <EdgeCordoni> edges = new ArrayList <EdgeCordoni>();

}//Graph Cordoni
```

1.4.2 Description of Graph Code

This code for the Graph Class was created by in class lessons but also previous knowledge from Software Development 1. Using this set up an array-list was created to hold the edges and vertices of each graph.

1.5 Knapsack Class

1.5.1 Description

For each knapsack in the spice file a knapsack object was created to house the data for better organization.

```
/*
 *
 * Assignment 5
 * Due Date and Time: 12/10/21 before 12:00am
 * Purpose: to implement directed graphs and greedy algorithm structures.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output direct graph shortest paths and greedy algorithms.
 * @author Shannon Cordoni
```

```
 *
 */

public class KnapsackCordoni{

    /**
    * Instance Variable
    */
    private Integer myCapacity;

   /**
    * The default Constructor for KnapsackCordoni
    */
    public KnapsackCordoni()
    {
       myCapacity = 0;
    }//Knapsack Cordoni

     /**
    * The full constructor for KnapsackCordoni
    */
    public KnapsackCordoni(Integer newCapacity)
    {
        myCapacity = newCapacity;

    }//KnapsackCordoni


    /**
    * The setter for the knapsack
    * @param newCapacity the incoming knapsack data
    */
    public void setCapacity(Integer newCapacity)
    {myCapacity = newCapacity;}//set Qty

    /**
    * the getter for the knapsack data
    * @return the incoming knapsack data
    */
    public Integer getCapacity()
    { return myCapacity;}//get node

}//knapsackCordoni
```

1.5.2 Description of Knapsack Code

This code for the Knapsack Class was created by in class lessons but also previous knowledge from Software Development 1. Using this set up a capacity was set for each knapsack so that we could create knapsack objects.

1.6 Vertex Class

1.6.1 Description

For each vertex in the graph an actual vertex object had to be created to hold it. This class creates the vertex object to be represented in the graph. This was so that the creation of the graph representations could run more smoothly and so that the edges could be added to help represent the graphs.

```

/*
 *
```

```
 * Assignment 5
 * Due Date and Time: 12/10/21 before 12:00am
 * Purpose: to implement directed graphs and greedy algorithm structures.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output direct graph shortest paths and greedy algorithms.
 * @author Shannon Cordoni
 *
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Arrays;
import java.util.ArrayList;

public class VertexCordoni
{
     /**
      * Declare Variables
     */
     private String myId;
     private boolean myIsProcessed;
     public ArrayList <VertexCordoni> neighbors= new ArrayList <VertexCordoni>();
     private VertexCordoni myNext;
     private VertexCordoni myPrevious;
     private Double myDistance;


     /**
      * The default Constructor for VertexCordoni
     */
     public VertexCordoni()
     {
     myId = new String();
     myIsProcessed = false;
     myNext = null;
     myPrevious = null;
     myDistance = 0.0;
     }//vertex Cordoni

     /**
     * The full constructor for VertexCordoni
     * @param newData the incoming data
     */
     public VertexCordoni(String newData)
     {
          myId = newData;
          myIsProcessed = false;
          myNext = null;
          myPrevious = null;
          myDistance = 0.0;

     }//NodeCordoni

    /**
     *
    * the setter for the vertex id
    * @param newId the incoming data of the vertex
    */
    public void setId(String newId)
    {myId = newId;} //set data

    /**
    * The getter for the vertex id
    * @return the incoming data of the vertex
    */
```

```
69      public String getId()
70      {return myId;}//get data
71
72
73      /**
74      *
75      * the setter for the next vertex
76      * @param newNext the incoming data of the vertex
77      */
78      public void setNext(VertexCordoni newNext)
79      {myNext = newNext;} //set data
80
81      /**
82      * The getter for the vertex
83      * @return the incoming data of the vertex
84      */
85      public VertexCordoni getNext()
86      {return myNext;}//get data
87
88      /**
89      *
90      * the setter for the Previous vertex
91      * @param newPrevious the incoming data of the vertex
92      */
93      public void setPrevious(VertexCordoni newPrevious)
94      {myPrevious = newPrevious;} //set data
95
96      /**
97      * The getter for the vertex
98      * @return the incoming data of the vertex
99      */
100     public VertexCordoni getPrevious()
101     {return myPrevious;}//get data
102
103     /**
104     * The setter for the process status
105     * @param newIsProcessed the incoming process status
106     */
107     public void setProcessStatus(boolean newIsProcessed)
108     {myIsProcessed = newIsProcessed;}//set data
109
110     /**
111     * the getter for the process status
112     * @return the incoming process status
113     */
114     public boolean getProcessStatus()
115     { return myIsProcessed;}//get data
116
117     /**
118     * The setter for the Distance
119     * @param newDistance the incoming Distance
120     */
121     public void setDistance(Double newDistance)
122     {myDistance = newDistance;}//set data
123
124     /**
125     * the getter for the process status
126     * @return the incoming process status
127     */
128     public Double getDistance()
129     { return myDistance;}//get data
130
131
132 }//Vertex Cordoni
```

1.6.2 Description of Vertex Code

This code for the Vertex Class was created by previous knowledge working with the Node Class. Using the same set up each Vertex was given an Id, process status, neighbor arraylist, next, previous, and distance. Getters and setters were created for each to make the creation of the vertexes and edges run more smoohtly.

1.7 Spice Class

1.7.1 Description

For each spice in the spice file a spice object was created so that we could create spice objects and run the Greedy Algorithm more smoothly.

```
/*
 *
 * Assignment 5
 * Due Date and Time: 12/10/21 before 12:00am
 * Purpose: to implement directed graphs and greedy algorithm structures.
 * Input: The user will be inputting a file containing a list of edges and vertices.
 * Output: The program will output direct graph shortest paths and greedy algorithms.
 * @author Shannon Cordoni
 *
 */

public class SpiceCordoni{

    /**
    * Instance Variable
    */
    private String myColor;
    private Double myPrice;
    private Integer myQty;
    private Double myUnitPrice;

   /**
    * The default Constructor for SpiceCordoni
    */
    public SpiceCordoni()
    {
       myColor = new String();
       myPrice = 0.0;
       myQty = 0;
       myUnitPrice = 0.0;
    }//Spice Cordoni


    /**
    * The full constructor for SpiceCordoni
    */
    public SpiceCordoni(String newColor, Double newPrice, Integer newQty, Double newUnitPrice)
    {
        myColor = newColor;
        myPrice = newPrice;
        myQty = newQty;
        myUnitPrice = newUnitPrice;

    }//SpiceCordoni

    /**
    * the setter for the spice data
    * @param newmyColor the incoming data of the spice
    */
    public void setColor(String newColor)
```

```
51      {myColor = newColor;} //set Name
52
53      /**
54      * The getter for the spice data
55      * @return the incoming data of the item
56      */
57      public String getColor()
58      {return myColor;}//get name
59
60      /**
61      * The setter for the spice
62      * @param newPrice the incoming spice data
63      */
64      public void setPrice(Double newPrice)
65      {myPrice = newPrice;}//set Price
66
67      /**
68      * the getter for the spice data
69      * @return the incoming spice data
70      */
71      public Double getPrice()
72      { return myPrice;}//get node
73
74      /**
75      * The setter for the spice
76      * @param newQty the incoming spice data
77      */
78      public void setQty(Integer newQty)
79      {myQty = newQty;}//set Qty
80
81      /**
82      * the getter for the spice data
83      * @return the incoming spice data
84      */
85      public Integer getQty()
86      { return myQty;}//get node
87
88      /**
89      * The setter for the spice
90      * @param newUnitPrice the incoming spice data
91      */
92      public void setUnitPrice(Double newUnitPrice)
93      {myUnitPrice = newUnitPrice;}//set unit price
94
95      /**
96      * the getter for the spice data
97      * @return the incoming spice data
98      */
99      public Double getUnitPrice()
100     { return myUnitPrice;}//get unit price
101
102 }//Spice Cordoni Class
```

1.7.2 Description of Spice Code

This code for the Spice Class was created by previous knowledge working with the Node Class. Using the same set up each Spice was given a color, price, qty, and unit price. Getters and setters were created for each to make the creation of the spice objects, and the greedy algorithm run more smoothly.

1.8 Overall:

Overall, these directed graph and Greedy Algorithm representations were successful in implementation. To go through each we can create a table for better data understanding, this table will show each and their asymptotic running time:

Bellman-Ford	Fractional Knapsack
O(VE)	O($nlog(n)$)

The table above shows a quick understanding of the methods used here. To go into more detail the Bellman Ford Algorithm as an asymptotic running time of O(VE), where V is the number of vertices and E is the number of edges. This is because we have to go through each edge and relax each vertex to determine the shortest path. Greedy Algorithms has an asymptotic running time of O($nlog(n)$) with *log* being of base 2, and n being equal to the size of the array. This is because we have to sort our spice objects from highest to lowest unit price and go through the knapsacks and spices to fill the knapsacks themselves. Overall, we were mostly successful in our implementation of this assignment.