CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Five

Nicholas Petrilli

December 11, 2021

1 DynamicProgramming Class - Main Method

Listing 1: Dynamic Programming - Main Method

```

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

public class DynamicProgramming {

    public static int numVerticies = 0;
    public static int numEdges = 0;
    public static void main(String[] args) {


        //Bellman ford function seems correct because it works for this examp

        int verticies = 5;
        int edges = 8;
        DirectedGraph graph = new DirectedGraph(verticies, edges);

        graph.edgeArray[0].source = 0;
        graph.edgeArray[0].destination = 1;
        graph.edgeArray[0].weight = -1;

        graph.edgeArray[1].source = 0;
        graph.edgeArray[1].destination = 2;
        graph.edgeArray[1].weight = 4;

        graph.edgeArray[2].source = 1;
        graph.edgeArray[2].destination = 2;
        graph.edgeArray[2].weight = 3;

        graph.edgeArray[3].source = 1;
```

```java
        graph.edgeArray[3].destination = 3;
        graph.edgeArray[3].weight = 2;

        graph.edgeArray[4].source = 1;
        graph.edgeArray[4].destination = 4;
        graph.edgeArray[4].weight = 2;

        graph.edgeArray[5].source = 3;
        graph.edgeArray[5].destination = 2;
        graph.edgeArray[5].weight = 5;

        graph.edgeArray[6].source = 3;
        graph.edgeArray[6].destination = 1;
        graph.edgeArray[6].weight = 1;

        graph.edgeArray[7].source = 4;
        graph.edgeArray[7].destination = 3;
        graph.edgeArray[7].weight = -3;


        graph.bellmanFord(graph, 0);




        readAndParseGraphFile();
        System.out.println("\n");
        readAndParseSpiceFile();

    }
    public static void readAndParseGraphFile() {

        File myFile = new File("graphs2.txt");


        try {
            Scanner fileScan = new Scanner(myFile);

        //Populates the array with the lines from text file
        while (fileScan.hasNext()) {
            String line = fileScan.nextLine();
            String[] words = line.split(" ");


            for (int i = 0; i < words.length; i++) {
                if (line.contains("new") && i == 0) {
```

```
                    System.out.println("Generating New Graph");
                    numVerticies = 0;
                    numEdges = 0;
                }//if
                else if (line.contains("add") && isNumber(words[i])) {
                    if (line.contains("vertex")) {
                        numVerticies++;
                    }//if
                    else if (line.contains("edge")) {
                        numEdges++;
                    }//else if
                }//else if
                //empty line means the graph is finished adding its verticies
                else if (line.isEmpty()) {
                    //System.out.println("Blankline");
                    int verticies = numVerticies;
                    int edges = numEdges / 3; //divide by three because the u

                    System.out.println("Number of verticies in graph is  " +
                    DirectedGraph graph = new DirectedGraph(verticies, edges)
                    graph.bellmanFord(graph, 0);
                }//else if
            }//for

        }
        fileScan.close();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();

        }

    }
    public static boolean isNumber(String str) {
        if (str == null || str.length() == 0) {
            return false;
        }
        for (char c: str.toCharArray()) {
            if (!(Character.isDigit(c))) {
                return false;
            }
        }
        return true;
    }

    public static void readAndParseSpiceFile() {
```

```java

        File myFile = new File("spice.txt");

        List<KnapsackItem> items = new ArrayList<>();
        Knapsack knapsack = new Knapsack(items, 0);

        try {
            Scanner fileScan = new Scanner(myFile);

        //Populates the array with the lines from text file
        while (fileScan.hasNext()) {
            String line = fileScan.nextLine();

            String spiceName = "";
            double totalPrice = 0;
            int quantity = 0;
            int knapsackCapacity = 0;

            if (!line.isEmpty() && !line.contains("—")) {
                String[] words = line.split(";");
                for (int i = 0; i < words.length; i++) {
                    //each line is split by semi-colon, so finding the last i
                    //which is the information that we want
                    String target = words[i].substring(words[i].lastIndexOf("
                    if (words[i].contains("spice_name")) {
                        spiceName = target;
                        //System.out.println(spiceName);
                    }//if
                    else if (words[i].contains("total_price")) {
                        totalPrice = Double.parseDouble(target);
                        //System.out.println(totalPrice);
                    }//else if
                    else if (words[i].contains("qty")) {
                        quantity = Integer.parseInt(target);
                        //System.out.println(quantity);
                    }//else if
                    else if (words[i].contains("capacity")) {
                        knapsackCapacity = Integer.parseInt(target);
                        //System.out.println(knapsackCapacity);
                    }
                }//for


                if (knapsackCapacity == 0) {
                double unitPrice = totalPrice / quantity;
                knapsack.addItem(spiceName, totalPrice, quantity, unitPrice);
```

```
                    System.out.println("Added the following item to the knapsack:
                    totalPrice + " quantity = " + quantity + " unitPrice = " + un
                    }//if
                    else {
                        System.out.println("Running with capacity: " + knapsackCa
                        knapsack.sortKnapsack();
                        Knapsack knapsackSolution = knapsack.solveHeist(knapsackC
                        System.out.println("Knapsack of capacity " + knapsackCapa
                    }//else

                }

            }
            fileScan.close();
            }
            catch (FileNotFoundException e) {
                e.printStackTrace();

            }
    }

}
```

Listed above is the class I created for my main method. This is where the reading and parsing of both the graph and spice files are done. For the graph file, I split the line by spaces in order to get each indivdual word, where I then looped through to check the contents of each line. When reading through the edge lines, the number of edges got incremented for the weights of each edge, so in the end the number of edges had to be divided by three to get rid of the edges. As for reading and parsing the spice file, I created an original knapsack where the contents of the file were added to. Then using that information, the solveHeist is called for each solution knapsack that was created that held the spices that were taken for the heist. In the main method I added a hard coded example showing that the bellman ford function works, but for some reason there is a problem when sending the graphs created from the file.

2 DirectedGraph Class

Listing 2: DirectedGraph Class

```

public class DirectedGraph {

    int verticies;
    int edges;

```

```java
    Edge[] edgeArray;

    public DirectedGraph(int verticies, int edges) {
        this.verticies = verticies;
        this.edges = edges;
        edgeArray = new Edge[edges];

        //create edges at each index
        for (int i = 0; i < edges; i++) {
            edgeArray[i] = new Edge();
        }
    }

    public void bellmanFord(DirectedGraph graph, int source) {
        int verticies = graph.verticies;
        int edges = graph.edges;
        int[] distance = new int[verticies];

        //initialze single source
        for (int i = 0; i < verticies; i++) {
            distance[i] = Integer.MAX_VALUE; //estimate of shortest path dist
        }
        distance[source] = 0;

        for (int i = 1; i < verticies - 1; i++) {
            for (int j = 0; j < edges; j++) {
                //get the source, destination and weight for each edge in gra
                int src = graph.edgeArray[j].source;
                int dest = graph.edgeArray[j].destination;
                int weight = graph.edgeArray[j].weight;

                if(distance[src] != Integer.MAX_VALUE && distance[dest] > dis
                    distance[dest] = distance[src] + weight;
                }//if
            }//for
        }//for

        //after the nested for loops, check edges again
        for (int j = 0; j < edges; j++) {
            int src = graph.edgeArray[j].source;
            int dest = graph.edgeArray[j].destination;
            int weight = graph.edgeArray[j].weight;

            if (distance[src] != Integer.MAX_VALUE && distance[dest] > distan
                System.out.println("There is a negative weight cycle in the g
                return;
```

```
            }
        }
        for (int i = 0; i < verticies; i++) {
            System.out.println("| 0 ——> " + i + " Cost is " + distance[i]);
        }
    }



}
```

This is my DirectedGraph class which contains my bellman ford function. The bellman ford function first initializes each element of the distance array to max value, which is to be changed when finding a shorter path. Next the distance[source] is set to 0 because the source's distance to itself is 0. Then, each edge of the graph has to be relaxed which uses a nested for loop, looping through the verticies and edges of the graph. If there is a shorter path from the source to the destination by calculating the weights, then distance[dest] is updated, so the shortest path gets shorter and shorter. After the nested loop, the edges are iterated through one more time in order to check for negative cycles in the graph because that means the single shortest path can't be calculated. The asymptotic running time for the bellman ford algorithm is O(V * E) where V is the number of verticies in the graph and E is the number of edges in the graph. This is so because of the inner nested for loop that is used to relax the edges in the graph. The outer loop loops until all of the verticies are traversed, and the inner loop runs until the edges are traversed for each vertex.

3 Edge Class

Listing 3: Edge Class

```

public class Edge {

    int source;
    int destination;
    int weight;

    public Edge() {
        source = 0;
        destination = 0;
        weight = 0;
    }

}
```

4

The DirectedGraph class uses the Edge class, as each graph has an array of edges.

5 KnapsackItem Class

Listing 4: KnapsackItem Class

```

public class KnapsackItem {

    private String spiceName;
    private double totalPrice;
    private int quantity;
    private double unitPrice;

    public KnapsackItem(String name, double price, int quantity, double unitP
        spiceName = name;
        totalPrice = price;
        this.quantity = quantity;
        this.unitPrice = unitPrice;

    }
    //Create getters for each member to be accessed from outside of class

    public String getSpiceName() {
        return spiceName;
    }
    public double getTotalPrice() {
        return totalPrice;
    }
    public int getQuantity() {
        return quantity;
    }
    public double getUnitPrice() {
        return unitPrice;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }


}
```

Above is my KnapsackItem class. My Knapsack class consists of an arraylist of

KnapsackItems.

6 Knapsack Class

Listing 5: Knapsack Class

```java

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

public class Knapsack {

    private List<KnapsackItem> knapsackItems;
    private int knapsackCapacity;

    private final int totalSpices = 20;

    private int spiceAdded = 0;

    public Knapsack(List<KnapsackItem> items, int capacity) {
        knapsackItems = items;
        knapsackCapacity = capacity;
    }

    public void addItem(String name, double price, int quantity, double unitP
        KnapsackItem item = new KnapsackItem(name, price, quantity, unitPrice
        knapsackItems.add(item);
    }

    public Knapsack solveHeist(int capacity) {
        ArrayList<KnapsackItem> solution = new ArrayList<>();
        Knapsack knapsackSolution = new Knapsack(solution, 0);

        knapsackCapacity = capacity;
        boolean remainingSpace = true;
        spiceAdded = 0;

        int counter = 0;
        while (remainingSpace) {
            if (knapsackCapacity == 0) {
                remainingSpace = false; //base case, capacity is decremented
            }//if
            else if (spiceAdded == totalSpices) {
```

```
                remainingSpace = false;
            }//else if
            else {
                int tempQuantity = knapsackItems.get(counter).getQuantity();
                while (tempQuantity > 0 && knapsackCapacity > 0) {
                    //System.out.println(knapsackCapacity);
                    if(existsInSolution(knapsackItems.get(counter).getSpiceNam
                        int tempQuantity2 = solution.get(counter).getQuantity
                        tempQuantity2++;
                        solution.get(counter).setQuantity(tempQuantity2);
                        System.out.println("Adding another scoop of " + knaps
                        spiceAdded++;
                    }//if
                    else {
                        String spiceName = knapsackItems.get(counter).getSpice
                        double price = knapsackItems.get(counter).getTotalPri
                        double unitPrice = knapsackItems.get(counter).getUnit
                        System.out.println("Adding to the solution Knapsack th
                        spiceAdded++;
                        knapsackSolution.addItem(spiceName, price, 1, unitPri
                    }//else
                    //item gets added to the knapsack so the quantity of the
                    tempQuantity--;
                    knapsackCapacity--;

                }//while
                counter++;
            }//else

        }//while

        return knapsackSolution;


    }

    //tests if a scoop of spice was already added in order to update the quan
    public boolean existsInSolution(String name, ArrayList<KnapsackItem> solu
        for (int i = 0; i < solution.size(); i++) {
            if (solution.get(i).getSpiceName().compareToIgnoreCase(name) == 0
                return true;
            }
        }
        return false;
    }

```

```java

    /* https://www.techiedelight.com/sort-list-of-objects-using-comparator-ja
    Referenced this website on how to sort arraylists using comparators
    */
    public void sortKnapsack() {
        //need to sort by unit price in order to take the higher value spices
        Collections.sort(knapsackItems, Comparator.comparing(KnapsackItem::ge
        //by default the arraylist gets sorted in increasing unit price order
        Collections.reverse(knapsackItems);
    }

    public double totalWorth() {
        double totalWorth = 0.0;
        for (int i = 0; i < knapsackItems.size(); i++) {
            totalWorth += knapsackItems.get(i).getUnitPrice() * knapsackItems
        }
        return totalWorth;
    }


}
```

Lastly, above is my Knapsack class which contains the function solveHeist, which is the function that is responsible for taking the most valuable spices and stealing them, adding to a knapsack. The function uses a while loop that terminates when either the capacity of the knapsack has been reached, or there are no more spices left to take. The sort function uses the java collections in order to sort the arraylist of knapsack items using a comparator and comparing by unitPrice of the spices. The sort method sorts in increasing unitPrice, but we want unitPrice to start at the highest value and be decreasing in order to add the most valuable spices to the knapsack first. The function utilizes the existsInSolution function which just checks if a scoop of spice has already been added to the knapsack. In that case, the quantity of the spice has to be updated, which is the use of tempQuantity2. After each iteration of a spice being added to the knapsack, the quantity of the spice is decremented as well as the knapsack capacity. The asymptotic running time for fractional knapsack is O(nlogn). This is so because the array list has to be sorted, so the while loop takes O(n) time and the sorting takes O(logn).