CMPT-435-Assignment-5

Ahmed Handulle

December 12, 2022

In this document, I will be explaining my code from Assignment 5 in detail.

This first part of the project will show the implementation details of the Bellman-Ford dynamic programming algorithm for Single Source Shortest Path (SSSP) on a few weighted, directed graphs. The second part will then show the implementation of a greedy solution to an intergalactic instance of the fractional knapsack problem; and to analyze these algorithms' performance in asymptotic terms.

1 External Java Packages

Below are the external java packages I have used to build these solve these two problems.

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

Below is the Main Class and it contains most of the logic of the program

2 Main Class

```java

public class Main {
    public static void main(String[] args) {
        // Creating an ArrayList to Available spices of class Spice
        ArrayList<Spice> spices = new ArrayList<>();
        // Creating another ArrayList of Integers to store the
    available knapsack capacity
        ArrayList<Integer> knapsack = new ArrayList<>();
        // This array will store the initial quantities of the
    spices
        ArrayList<Integer> initial_Quantities = new ArrayList<>();
        try {
            File f = new File("graphs2.txt"); // getting the
    graphs2.txt file
            Scanner myReader = new Scanner(f); // Scanner reader
    will give access to the graphs2.txt file
            // Start reading the file line by line
            while (myReader.hasNextLine()) {
                //store the data of the line
```

```java
            String data = myReader.nextLine();
            // Check if the line starts with "new graph" to create a new matrix and adjacencylist for that specific graph
            if (data.startsWith("new graph")) {
                System.out.println("new graph");
                System.out.println();

                // Creating an ArrayList to store the verticies of the linked objects graph representation
                ArrayList<LinkedObjects> linkObjGraph = new ArrayList<>();
                // Iterate over the nex few lines to count the vertices
                String tempString = myReader.nextLine();
                //count verticies variable
                int count = 0;
                // Create verticies
                while (myReader.hasNextLine() & tempString.startsWith("add vertex")) {
                    linkObjGraph.add(new LinkedObjects(getVertex(tempString)));
                    tempString = myReader.nextLine();
                    count = count + 1;
                }
                // Creating an ArrayList of Arraylist to store the weights of the edges of the neighboring vertices of each vertex
                // Each index of the Arraylist will crospond to its respective vertex
                ArrayList<ArrayList<Integer>> weights = new ArrayList<ArrayList<Integer>>();
                initializeWeightsArrlist(count, weights);
                // iterate over the edges and add edges to the linked Objects
                while (myReader.hasNextLine() & tempString.startsWith("add edge")) {
                    // this toFilterString will return an array of three number
                    int[] edgeVerticies = toFilterString(tempString);
                    //Add edge between the two verticies and their weight
                    addEdge(edgeVerticies, linkObjGraph, weights);
                    tempString = myReader.nextLine();
                    //Edge Case => making sure the last line of the file gets exuted
                    if (!myReader.hasNextLine() & tempString.startsWith("add edge")) {
                        int[] lastLineofFile = toFilterString(tempString);
                        addEdge(lastLineofFile, linkObjGraph, weights);
                    }
                }
                bellmanFord(linkObjGraph, weights, linkObjGraph.get(0), count);
```

```java
                }
            }
            System.out.println();
            System.out.println("---------------Knapsack Problem here-------------------------");
            System.out.println();

            // Access the spices.txt file and crating the spices objects and knapsacks
            createSpicesAndKnapsacks(spices, knapsack);
            // Store the initial quantities here
            initialQty(spices, initial_Quantities);
            // // Calculating the price per quantity of each of the spices
            calculatePricePerQty(spices);
            // filling up each knapsack with the most valuable spices and getting the results
            for (int i = 0; i < knapsack.size(); i++) {
                // Creating a HashMap object to store each of the spices and their quantity each knapstack has
                HashMap<String, Integer> result = new HashMap<String, Integer>();
                // the knapstack capacity that we are using
                int capacity = knapsack.get(i);
                //This function will do all the calculations needed and will also update the hashman with the results
                calculate(capacity, result, spices, initial_Quantities);
                // calculating total worth
                double worth = 0.0;
                for (String j : result.keySet()) {
                    worth = worth + (getPricePerQty(j, spices) * result.get(j));
                }
                // printing the results
                System.out.print(
                        "knapstack of capacity " + capacity + " is worth " + worth + " quatloos and contains ");
                for (String j : result.keySet()) {
                    if (result.size() == 1) {
                        System.out.print(result.get(j) + " scoop of " + j + ", ");
                    } else {
                        System.out.print(result.get(j) + " scoops of " + j + ", ");
                    }
                }
                System.out.println();
            }
            myReader.close();
        } catch (Exception e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }

    // This function creates edges between two vertices and their
```

```java
weighted value
    public static void addEdge(int[] edgeVerticies, ArrayList<LinkedObjects> linkObjGraph,
            ArrayList<ArrayList<Integer>> weights) {
        // edgeverticies & weight = [v,u,w]
        //store the neighbour object into a temprorary object variable
        LinkedObjects temObj = linkObjGraph.get(edgeVerticies[1] - 1);
        //Add vertex object 2 to vertex object 1 (edgeVerticies[0] - 1 => because Arraylist start storing at index 0 when graph starts at vertex 1)
        linkObjGraph.get(edgeVerticies[0] - 1).neighbors.add(temObj);
        // get the index of the array list crossponding to the vertex and then add weight
        //edgeVerticies[2] = weight
        weights.get(edgeVerticies[0] - 1).add(edgeVerticies[2]);

    }

    // This function will filter a string and return the two vertices where an edge will be created
    public static int[] toFilterString(String string) {
        //split the string at spaces and get the vertices from the string
        int[] edge = new int[3];
        String[] splitLine = string.split(" ");
        edge[0] = Integer.parseInt(splitLine[2]);
        edge[1] = Integer.parseInt(splitLine[4]);
        edge[2] = Integer.parseInt(splitLine[splitLine.length -1]);
        return edge;
    }

    // This funtion return the vertex number as an integer
    public static int getVertex(String tempString) {
        //Filter String
        String[] stringParts = tempString.split(" ");
        int vertex = Integer.parseInt(stringParts[stringParts.length - 1]);
        return vertex;
    }

    // This function initializes the arraylist of arrays which will store the weights
    public static void initializeWeightsArrlist(int count, ArrayList<ArrayList<Integer>> weights) {
        int i = 0;
        while (i < count) {
            weights.add(new ArrayList<Integer>());
            i++;
        }
    }

    // Bellman Ford Algorthim (SSSP) implimentation
    public static void bellmanFord(ArrayList<LinkedObjects> linkObjGraph, ArrayList<ArrayList<Integer>> weights,
```

```java
            LinkedObjects source, int count) {
                // Initialize the distance values of the verticies
                initSingleSource(linkObjGraph, source);
                // iterating through all the verticies and relaxing each vertex in (n-1) times where n is the number of vertices
                int c = 0; //count = number of verticeis
                while (c < count - 1) {
                    int n = linkObjGraph.size();
                    for (int i = 0; i < n; i++) {
                        for (int j = 0; j < linkObjGraph.get(i).neighbors.size(); j++) {
                            //Relax (u,v,w)
                            relax(linkObjGraph.get(i), linkObjGraph.get(i).neighbors.get(j), weights);
                        }
                    }
                    c++;
                }
                // Check if there are any negative cycles in the graph
                for (int i = 0; i < linkObjGraph.size(); i++) {
                    for (int j = 0; j < linkObjGraph.get(i).neighbors.size(); j++) {
                        LinkedObjects u = linkObjGraph.get(i);
                        LinkedObjects v = u.neighbors.get(j);
                        if (v.distance > u.distance + weightOf(weights, u, v)) {
                            System.out.println("Negative cycle detected");;
                            break;
                        }
                    }
                }
                // Printing the shortest path of each vertex from the source vertex
                for (int i = 1; i < linkObjGraph.size(); i++) {
                    ArrayList<Integer> path = new ArrayList<Integer>();
                    path = getPath(linkObjGraph, source, linkObjGraph.get(i), path);
                    System.out.println(source.id + " --> " + linkObjGraph.get(i).id + " cost is "
                            + linkObjGraph.get(i).distance + "; path: "+path);
                }
    }

    // This function initializes the distance value of each of the verticies
    public static void initSingleSource(ArrayList<LinkedObjects> linkObjGraph, LinkedObjects source) {
        // initialize the source vertex distance to zero and the rest is infinity
        for (LinkedObjects s : linkObjGraph) {
            // all vertex.distance will be equal to infinity except 1
            double infinity = Double.POSITIVE_INFINITY;
```

```
            s.distance = (int) infinity;
        }
        source.distance = 0;
    }

    // this Function will relax on all the verticies in (n-1) times where n is the number of verticies
    public static void relax(LinkedObjects u, LinkedObjects v, ArrayList<ArrayList<Integer>> weights) {
        // Start the relaxaztion
        if (v.distance > u.distance + weightOf(weights, u, v)) {
            v.distance = u.distance + weightOf(weights, u, v);
            v.predecessor = u;
            v.predecessor = u;   // setting predecessor vertex to u
        }
    }

    // This function will return the weight or two verticies example w(u,v)
    public static int weightOf(ArrayList<ArrayList<Integer>> weights, LinkedObjects u, LinkedObjects v) {
        int k = 0;
        for (int i = 0; i < u.neighbors.size(); i++) {
            k = i;
            if (u.neighbors.get(i) == v) {
                break;
            }
        }
        return weights.get(u.id - 1).get(k);
    }

    // This function find take the source vertex and another vertex and will desplay the path
    public static ArrayList<Integer> getPath(ArrayList<LinkedObjects> linkedObjects, LinkedObjects source,
            LinkedObjects destination, ArrayList<Integer> path) {
        while (destination.id != source.id) {
            path.add(destination.id);
            destination = destination.predecessor;
        }
        path.add(source.id);
        Collections.reverse(path);
        return path;
    }

    // This function will filter through the spice.txt file and create all the spices objects as wells as the knapsacks
    public static void createSpicesAndKnapsacks(ArrayList<Spice> spices, ArrayList<Integer> knapsack) {
        try {
            File f2 = new File("spice.txt");
            Scanner f2Reader = new Scanner(f2);
            // Start reading the file line by line
            while (f2Reader.hasNextLine()) {
                //store the data of the line
                String data = f2Reader.nextLine();

```

```java
            // Check if the line starts with spacie to create an object of class spice to store the spice details
            if (data.startsWith("spice")) {

                // Store the returned spice details in the temporary variable holder
                Object[] temp = getSpice(data);
                // converting objects to string and integeres
                String name = (String) temp[0];
                Integer qty = (Integer) temp[2];
                Double totalPrice = (Double) temp[1];

                // create a new spice and it to the spices arraylist of spices
                spices.add(new Spice(name, totalPrice, qty));
            }

            // Creating the knapstack capacity containers
            if (data.startsWith("knapsack")) {
                // create a knapsack container object
                knapsack.add(getCapacity(data));
            }
        }
        f2Reader.close();
    } catch (Exception e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}

    // This function will filter a string and return the two vertices where an edge will be created
public static Object[] getSpice(String spice_info) {
    //split the string at semi-colons and get the name, total_price and qty
    Object[] spice = new Object[3];
    String[] splitLine = spice_info.split(";");
    for (String s : splitLine) {
        String[] second_split = s.split(" ");
        if (s.contains("spice")) {
            // add name to the first index of spice object
            spice[0] = (String) second_split[second_split.length - 1];
        } else if (s.contains("total_price")) {
            // Add total price to the array of objects
            spice[1] = Double.valueOf(second_split[second_split.length - 1]);
        } else if (s.contains("qty")) {
            // Add total price to the array of objects
            spice[2] = Integer.parseInt(second_split[second_split.length - 1]);
        }
    }
    return spice;
}

// This function will filter a sting and return an integer
```

```
    which represents the knapsack capacity
    public static int getCapacity(String knapCapacity) {
        int capacity = 0;
        // split the line at semi-colons
        String[] splitLine = knapCapacity.split(";");
        for (String s : splitLine) {
            //split again by space
            String[] second_split = s.split(" ");
            capacity = Integer.parseInt(second_split[second_split.length - 1]);
        }
        return capacity;
    }

    // This function will calculate the price per quantity of the spices and update that in each of the spice object
    private static void calculatePricePerQty(ArrayList<Spice> spices) {
        for (Spice s : spices) {
            s.price_per_qty = s.total_price / (double) s.qty;
        }
    }

    // this function retruns the item with the highest price/quantity
    private static Spice getMaxItem(ArrayList<Spice> spices) {
        Spice maxItem = spices.get(0);
        for (int i = 1; i < spices.size(); i++) {
            if (spices.get(i).price_per_qty > maxItem.price_per_qty) {
                maxItem = spices.get(i);
            }
        }
        return maxItem;
    }

    // This function stores the initial quantities of the spices
    public static void initialQty(ArrayList<Spice> spices, ArrayList<Integer> initial_Quantities) {
        for (Spice s : spices) {
            initial_Quantities.add(s.qty);
        }
    }

    // This Function will calculate the result
    private static void calculate(int capacity, HashMap<String, Integer> result, ArrayList<Spice> spices, ArrayList<Integer> initial_Quantities) {
        // get the item with the highest price/quantity
        Spice maxItem = getMaxItem(spices);
        if (maxItem.price_per_qty != 0.0) {
            while (maxItem.remaining_quantity > 0 & capacity > 0) {
                // decriment the quantity remaining
                maxItem.remaining_quantity = maxItem.remaining_quantity - 1;
                // add the spice and quantity into the hashmap
                result.put(maxItem.name, result.getOrDefault(
```

```
maxItem.name, 0) + 1);
                capacity--;
            }
            // Check if you have any space left in the container
            if (capacity > 0) {
                // we now konw that this capacity has used all the spices of the maximum quantity and we still have more space left
                // Making sure we don't use the same object twice for the second iteration
                maxItem.price_per_qty = 0.0;
                // now it is time to get another spice
                calculate(capacity, result, spices, initial_Quantities);
            }
        }
        reset(spices, initial_Quantities);
    }

    // This function resets the quantities of the objects back to default
    public static void reset(ArrayList<Spice> spices, ArrayList<Integer> initial_Quantities) {
        int i = 0;
        for (Spice s : spices) {
            s.remaining_quantity = initial_Quantities.get(i);
            i++;
        }
        // resetting the prices per quantity
        calculatePricePerQty(spices);
    }

    // This function will take a string (the name of the spice) and will return the price per quantity of that spice
    public static double getPricePerQty(String spice_name, ArrayList<Spice> spices) {
        // initialize variable
        double price_per_quantity = 0.0;
        for (Spice s : spices) {
            if (s.name.compareTo(spice_name) == 0) {
                price_per_quantity = s.total_price / s.qty;
            }
        }
        return price_per_quantity;
    }
}

----------------------End of the Main Class------------------------
```

3 LinkedObjects Class

Below is the LinkedObjects class and it will be used to create linked objects for representation of waited, undirected graphs.

```java
import java.util.ArrayList;

public class LinkedObjects {
    // Object Attributes => Each Vertex is represented as an object
     with attributes as class variables
    int id;
    Boolean processed;
    int distance;
    LinkedObjects predecessor;
    ArrayList<LinkedObjects> neighbors;

    // Initialize the class/Vertex object
    LinkedObjects(int id) {
        this.id = id;
        this.distance = 0;
        this.predecessor = null;
        this.neighbors = new ArrayList<>();
    }
}
```

4 Spice Class

Below is the Spice which I used to create the spices objects.

```java

public class Spice {

    // This class will hold the available spices to take
    String name;
    double total_price;
    int qty;
    double price_per_qty;
    boolean isUsedAll;
    int remaining_quantity;

    Spice(String name, double total_price, int qty) {
        this.name = name;
        this.total_price = total_price;
        this.qty = qty;
        price_per_qty = 0.0;
        remaining_quantity = qty;
    }
}
```

//——————-Go to the Next Page for Results—————

5 Results

The Time Complexity of the Bellman-ford's Algorithm for the single shortest path is dependant on the number of vertices and edges in the graph. However, the worst case happens when we encounter a negative cycle in the graph as well as encountering disconnected graphs. So the time complexity will be (V to the power of 2 time number of edges = N to the power of 3) as we have to relax over all the vertices in v-1 times through the number number of edges.

In the best case scenario where we have a complete graph and and no negative cycles, the time complexity gets down to V x E.

The greedy technique for solving the knapsack problem takes O(N x W) where is N is the number of weight elements and W is the capacity of the knapsack.