Assignment Five

Nicholas Fiore

Nicholas.Fiore@Marist.edu

November 30, 2022

1 Directed Graphs and Bellman-Ford

Directed graphs are a form of graph in which edges have a direction, represented by an arrow. Weighted directed graphs are these graphs except that the edges also have a weight to them. The weights of the edges can be likened to the cost or difficulty of traversing that edge. This creates a varied topology in which the best way to get from point A to point B may not in fact be a straight line, as it may cost less to take a seemingly more roundabout route.

The Bellman-Ford Single Source Shortest Path (referred to as SSSP from here on) is an algorithm that finds the minimum cost (shortest path) from one vertex in the graph (single source) to every other accessible vertex in the graph. It is not the only algorithm capable of doing this, however it is one of the only ones that can function with negative weights. The SSSP algorithm accomplishes this goal in three steps: distance initialization, edge relaxation, and negative loop checking.

1.1 Distance Initialization

The first step of the algorithm is simple. The distances of the vertices from the source vertex are kept track of, and all distances are initialized to a large number. In theory, this number is ∞, but in practice it is coded as using the largest possible integer value. Once initialized to infinity, the distance from the source vertex to the source vertex is then changed to 0, as it takes no movement from the source vertex to reach the source vertex.

1.2 Edge Relaxation

Relaxing the edges of the graph is a bit more complex, but not by a great margin. The basic principle is that it goes through every *Edge* object in the *Graph*, and compares the origin point to the destination in order to assign a distance-cost. The comparison is this: first, the origin is compared to ∞. If the origin is still in that initial ∞ setting, it is ignored for now. Otherwise, it moves on to the next comparison, which is comparing the current distance-cost of the origin + the weight of the edge to the current distance-cost of the destination. If the origin + weight is less than the destination, the distance-cost of the destination is then set to the distance-cost of the origin + the weight of the edge. This is basically just showing that the cost to get to the destination is the same as the cost to get to the origin and then also taking the cost of the edge itself.

After the edges are all checked or skipped over, the loop of checking all the edges repeats. It will repeat for the total number of vertices in the object - 1. This ensures that every edge will be checked at least once so that no distances are still measured at ∞.

1.3 Negative Cycle Check

The final step is a check step that is necessary when working with negative weights. Due to how negative weight cycles can occur with negative weights. A negative cycle occurs when there are a group of edges that have a negative sum cost and are reachable from the source. When this occurs, there is no shortest path because going through the negative cycle again will reduce the cost further, and then doing so again would reduce the cost even more, and the process could repeat ad infinitum. Since you could never actually determine the shortest path in that case (since it might just approach $-\infty$), there is no shortest path.

The negative cycle check discovers a negative cycle by running that same algorithm in step two one more time. In the relaxation step, the cycle is run just enough times to get all the shortest possible paths. However, if it is run another time and a shorter path is found, then a negative cycle exists.

1.4 Asymptotic Run-time

The run-time of the Bellman-Ford SSSP algorithm is defined as $O(V * E)$. This is due to the usage of a nested loop. Since all the edges of the graph are checked for every vertex in the graph (when considering both the relaxation step and the extra check for negative cycles), it is simply the total number of vertices times the total number of edges.

2 Fractional Knapsack and Greedy Algorithms

2.1 The Knapsack Problem

The knapsack problem is a theoretical problem in you have a knapsack with a certain capacity, and various items of varied size and value that you want to take. You cannot take more than your knapsack's capacity, and the goal is to get the maximal profit from what objects you take. There are two variations of the problem, the binary (0:1/all-or-nothing) knapsack problem, and the fractional knapsack problem.

A potential solution to the knapsack problem is by applying a Greedy algorithm. That is, an algorithm that looks for the most valuable item and takes as much of it as it can, before moving on to the next most valuable item, and then the next, so on and so forth until the knapsack is either full or the objects otherwise cannot be added. This approach works in some cases of the knapsack problem, but not in others.

2.1.1 Binary Knapsack

The binary knapsack version of the problem can be illustrated like this. You have a knapsack that can hold say $5 units$ of something. You are trying to steal gold idols, which are of various size and value. The first idol is $1 unit$ in size and worth \$6. The second is $2 units$ and \$10. The third is $3 units$ and \$12. In this scenario, the best possible combination is to grab the second and third idols, as they are both worth the most at \$22, and will just fit in the bag. However, this does not work with a greedy algorithm. A greedy algorithm determines value based on unit value, or value of object divided by the total amount of units. Going by this, the first idol has a unit value of \$6/unit, the second is \$5/unit, and the third \$4/unit. To the greedy algorithm, that means that the greedy algorithm will take the first idol, since it has the highest unit value. Then it will take the second idol. However, once it reaches the third idol, it only has $1 unit$ of space left in the bag, so it cannot take the third idol. The value of the bag now is \$16, far less than the optimal choice. For this reason, a greedy algorithm does not work with a binary knapsack.

2.1.2 Fractional Knapsack

The greedy algorithm does, however, work well with a fractional knapsack. Unlike binary knapsack, where you must take either the whole object or not take it at all, fractional knapsack allows you to take a fraction of the object. Taking the same problem as before but replacing the idols with piles of gold dust, we can follow the greedy algorithm. All of the first pile of gold is taken, and then all of the second pile. There is $1unit$ left in the knapsack, and $3units$ of pile three. You take $1unit$ from the third pile, worth $12/unit. The total value of the knapsack is now $24, which surpasses even the optimal binary knapsack solution.

2.2 Asymptotic Run-time

By itself, assuming that the items are already in order, is only O(n). The algorithm will go to each pile sequentially, until it can no longer fit anything in the bag. However, it is unlikely that the list of items/piles will be in order, so a sorting algorithm will have to be implemented. This increases the complexity to be that of the sorting algorithm. Best case would use merge sort or quick sort, making the complexity $O(nlogn)$. My implementation of it used insertion sort, which makes the complexity $O(n^2)$.

3 Appendix

3.1 MainFive.java

```java
//Utility imports
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;

/*
 * This program is used to explore weighted graphs and pathfinding within them using the Bellman-Ford Single-Source Shortest Path
 * algorithm, as well as explore the fractional knapsack problem using a greedy algorithm.
 */

public class MainFive {
    public static void main(String[] args) throws IOException {
        //variable to store comparisons
        int[] compCounter = new int[1];
        System.out.println("Graphs:");
        createGraph("graphs2.txt");

        System.out.println("Greedy Knapsack problem:");
        fractionalKnapsackSpiceHeist("spice.txt");
    }

    //takes a file name and a list and puts each line of the file into a String in an array
    public static String[] fileToArray(String fileName, String[] list) throws IOException {
        long totalLines = 0;
        File file = new File(fileName);
        BufferedReader input = null;
        try {
            //gets the path of the current file in order to get the # of lines
            Path path = Paths.get(file.getName());

            input = new BufferedReader(new FileReader(fileName));

            totalLines = Files.lines(path).count();

            list = new String[(int)totalLines];

            for(int i = 0; i < totalLines; i++) {
                list[i] = input.readLine();
            }

        } catch(FileNotFoundException ex) {
            System.out.println("Failed to find file: " + file.getAbsolutePath());
        } catch(IOException ex) {
            System.out.println(ex);
        } catch(Exception ex) {
            System.out.println("Something went wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        } finally {
            if (input != null) {
                input.close();
            }
        }

        return list;
    }

```

```java
    //takes in a file and performs a greedy algoritm to solve a fractional knapsack problem for all spices and knapsacks
    public static int fractionalKnapsackSpiceHeist(String fileName) throws IOException {
        long totalLines = 0;
        File file = new File(fileName);
        BufferedReader input = null;
        String line;
        String nextLine;
        String currCmd; //keeps track of the current subcommand while parsing the file

        String[] cmdLine = null; //used to keep track of the current line for spices by assigning the String.split() String array to it

        //lists to keep track of the Spices and the Knapsacks
        ArrayList<Spice> spices = new ArrayList<Spice>();
        ArrayList<Knapsack> knapsacks = new ArrayList<Knapsack>();

        try {
            //gets the path of the current file in order to get the # of lines
            Path path = Paths.get(file.getName());

            input = new BufferedReader(new FileReader(fileName));

            totalLines = Files.lines(path).count();

            //instantiate variables for graph and vertex objects. There are two Vertex objects in the case of adding edges.
            Graph graph = null;
            Vertex vert1 = null;
            Vertex vert2 = null;
            int weight;

            nextLine = input.readLine();

            //Command parsing. Goes through each line and determines what command is being used based on strings.
            for (int i = 0; i < totalLines; i++) {
                line = nextLine;
                nextLine = input.readLine();
                line = line.trim().replaceAll("␣+", "␣"); //reformats the String so that parsing is easier
                if (line.split("␣")[0].compareTo("--") == 0 || line.isEmpty()) {
                    //ignore line; it is a comment or blank space
                } else if (line.split("␣")[0].compareTo("spice") == 0) {
                    cmdLine = line.split(";"); //splits the current line based on ; (each line before a semicolon is a command)

                    String spiceName = null; //these values are used to assign to the Spice that will be added
                    double totalPrice = 0.0;
                    int qty = 0;

                    for (int j = 0; j < cmdLine.length; j++) {
                        currCmd = cmdLine[j].trim(); //ensures there are no leading spaces
                        if (currCmd.split("␣")[1].compareTo("name") == 0) {
                            spiceName = currCmd.substring(currCmd.lastIndexOf("␣")+1).split(";")[0];
                        } else if (currCmd.split("␣")[0].compareTo("total_price") == 0){
                            totalPrice = Double.parseDouble(currCmd.substring(currCmd.lastIndexOf("␣")+1).split(";")[0]);
                        } else if (currCmd.split("␣")[0].compareTo("qty") == 0){
                            qty = Integer.parseInt(currCmd.substring(currCmd.lastIndexOf("␣")+1).split(";")[0]);
                        }
                    }//end for
```

```java
                    spices.add(new Spice(spiceName, totalPrice, qty));
                } else if (line.split("␣")[0].compareTo("knapsack") == 0) {
                    int cap = Integer.parseInt(line.substring(line.lastIndexOf("␣")+1).split(";")[0]); //parses the capacity of the knapsack
                    knapsacks.add(new Knapsack(cap));
                }

                //final check to see if the end of the file was reached
                if (nextLine == null) { //file has been parsed, begin processing
                    spiceInsertionSort(spices);
                    for(int j = 0; j < knapsacks.size(); j++) {
                        greedyKnapsack(spices, knapsacks.get(j));
                    }
                }
            }

        } catch(FileNotFoundException ex) {
            System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
        } catch(NullPointerException ex) {
            System.out.println(ex.getMessage());
        } catch(Exception ex) {
            System.out.println("Something␣went␣wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        } finally {
            if (input != null) {
                input.close();
            }
        }

        return (int)totalLines;
    }

    //the greedy algorithm used by fractionalKnapsackSpiceHeist(). The algorithm assumes the list of spices is already in descending order in terms of price per scoop of spice, meaning the highest price per scoop spice is first
    public static void greedyKnapsack(ArrayList<Spice> spiceList, Knapsack sack) {
        int capacity = sack.getCapacity();
        int i = 0;
        //variables to store for printing later
        ArrayList<Spice> stolenSpices = new ArrayList<>(); //keeps track of the spice piles that were placed into the sack
        ArrayList<Integer> scoopList = new ArrayList<>(); //keeps track of how many scoops of each spice were taken. Indexes here are a 1:1 correlation to the Spice in stolenSpices

        while (i < spiceList.size() && sack.getCurrVolume() < capacity) {
            Spice currSpice = spiceList.get(i);
            stolenSpices.add(currSpice);
            int remSpice = currSpice.getQuantity();
            int scoops = 0;
            while (remSpice > 0 && sack.getCurrVolume() < capacity) {
                sack.addScoop(currSpice);
                remSpice -= 1;
                scoops += 1;
            }
            scoopList.add(scoops);
            i++;
        }

        String prntStr = "";
        System.out.print("A␣knapsack␣of␣capacity␣" + capacity + "␣is␣worth␣" + sack.getValue() + "␣quatloos␣and␣contains␣");
        for (i = 0; i < stolenSpices.size() - 1; i++) {
```

```
            prntStr += scoopList.get(i) + "␣scoop(s)␣of␣" + stolenSpices.get(i).getName() + ",␣";
        }
        prntStr += "and␣" + scoopList.get(i) + "␣scoop(s)␣of␣" + stolenSpices.get(i).getName() + "
            .";
        System.out.println(prntStr);
    }

    //sorts a list of Spices based on their price per scoop
    public static void spiceInsertionSort(ArrayList<Spice> list) {
        //the sorting algorithm
        for (int i = 1; i < list.size(); i++) {
            Spice key = list.get(i);
            int j;

            for (j = i - 1; j >= 0 && key.getPricePerScoop() > list.get(j).getPricePerScoop(); j
                --) {
                //arr[j + 1] = arr[j];
                list.set(j + 1, list.get(j));
            }
            //arr[j + 1] = key;
            list.set(j + 1, key);
        }
    }

    //Creates graphs from instructions given in a file
    public static int createGraph(String fileName) throws IOException {
        long totalLines = 0;
        File file = new File(fileName);
        BufferedReader input = null;
        String line;
        String nextLine;
        try {
            //gets the path of the current file in order to get the # of lines
            Path path = Paths.get(file.getName());

            input = new BufferedReader(new FileReader(fileName));

            totalLines = Files.lines(path).count();

            //instantiate variables for graph and vertex objects. There are two Vertex objects in
                the case of adding edges.
            Graph graph = null;
            Vertex vert1 = null;
            Vertex vert2 = null;
            int weight;

            nextLine = input.readLine();

            int graphNum = 1;

            //Command parsing. Goes through each line and determines what command is being used
                based on strings.
            for (int i = 0; i < totalLines; i++) {
                line = nextLine;
                nextLine = input.readLine();
                line = line.trim().replaceAll("␣+", "␣"); //reformats the String so that parsing
                    is easier
                if (line.split("␣")[0].compareTo("--") == 0) {
                    //do nothing, ignore this line as it is a comment
                } else if (line.split("␣")[0].compareTo("new") == 0) { //new graph
                    //create a new graph object held by the graph variable
                    graph = new Graph();

                } else if (line.split("␣")[0].compareTo("add") == 0) { //enters add vertex/add
                    edge tree
```

```
                    if (line.split("␣")[1].compareTo("vertex") == 0) { //add vertex x
                        //adds a new vertex to the graph, parsing the ID from the line given
                        vert1 = new Vertex(Integer.parseInt(line.split("␣")[2]));
                        graph.addVertex(vert1);
                    } else if (line.split("␣")[1].compareTo("edge") == 0) { //add edge x -> y with weight z
                        //adds a new edge to the graph, based on the IDs parsed from the line
                            //the verticies are found by searching the graph's verticies ArrayList for a Vertex that matches the ID given in the line at both positions
                        vert1 = Search.linearSearchReturnVertex(graph.getVerticies(), Integer.parseInt(line.split("␣")[2]));
                        vert2 = Search.linearSearchReturnVertex(graph.getVerticies(), Integer.parseInt((line).split("␣")[4]));
                        weight = Integer.parseInt(line.split("␣")[5]);

                        graph.addEdge(vert1, vert2, weight); //the verticies are now added as neighbors, forming an adjacency
                    }
                }
                //final check to see if the next line is either empty or does not exist
                if (nextLine == null || nextLine.isEmpty()) { //empty space; commands for this graph are done, begin processing
                    System.out.println("Graph␣" + graphNum + ":");
                    graphNum++;
                    graph.singleSourceShortestPath(graph.getVerticies().get(0));
                }
            }

        } catch(FileNotFoundException ex) {
            System.out.println("Failed␣to␣find␣file:␣" + file.getAbsolutePath());
        } catch(IOException ex) {
            System.out.println(ex.getMessage());
        } catch(NullPointerException ex) {
            System.out.println(ex.getMessage());
        } catch(Exception ex) {
            System.out.println("Something␣went␣wrong.");
            System.out.println(ex.getMessage());
            ex.printStackTrace();
        } finally {
            if (input != null) {
                input.close();
            }
        }

        return (int)totalLines;
    }
}
```

3.2 Vertex.java

Modified from assignment four. In assignment four, edges were implicit; if two vertex objects had each other in their *neighbors* table, then there was an edge between them. Now that Edges are an explicitly defined class object, this part of *Vertex* was removed, and now the Vertex objects only represent the points on a graph.

```java
import java.util.ArrayList;

/*
 * Vertex objects to be used with a Graph. These give weights to edges in one direction, creating a directed graph.
 */

public class Vertex {
    /* Data Fields */
    private int id;
    private boolean processed = false;

    /* Constuctors */
    public Vertex() {
        //default empty constructor
    }

    //constuctor with ID parameter
    public Vertex(int idNum) {
        this.id = idNum;
        this.processed = false;
    }

    /* Accessors/Mutators */
    public int getId() {
        return this.id;
    }

    public boolean wasProcessed() {
        return this.processed;
    }

    public void setId(int newId) {
        this.id = newId;
    }

    public void setProcessed(boolean bool) {
        this.processed = bool;
    }
}
```

3.3 Edge.java

```java
/*
 * An object to represent a weighted edge in a graph. A weighted edge can only have one direction, starting rom the origin
 * and going to the destination.
 */

public class Edge {
    Vertex origin = null;
    Vertex destination = null;
    int weight;


```

```java
    public Edge(){
        /* Empty Constructor */
    }

    //full constructor
    public Edge(Vertex org, Vertex dest, int weight) {
        origin = org;
        destination = dest;
        this.weight = weight;
    }

    /* Accessors/Mutators */
    public int getWeight() {
        return this.weight;
    }

    public Vertex getOrigin() {
        return this.origin;
    }

    public Vertex getConnection() {
        return this.destination;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    public void setOrigin(Vertex origin) {
        this.origin = origin;
    }

    public void setConnection(Vertex destination) {
        this.destination = destination;
    }
}
```

3.4 Graph.java

The new SSSP algorithm is included in this version of Graph

```java
import java.util.*;

/*
 * Graph data structure, using Vertex objects as nodes. This version is for directed graphs
 */

public class Graph {
    /* Data Fields */
    private String name = "";
    private ArrayList<Vertex> verticies = new ArrayList<Vertex>();
    private ArrayList<Edge> edges = new ArrayList<Edge>();
    private String[][] matrix;

    /* Constructors */
    //Default, empty constuctor
    public Graph() {

    }

    //constuctor with name parameter
    public Graph(String newName) {
        this.name = newName;
```

```java
    }

    /* Accesors/Mutators */
    public String getName() {
        return this.name;
    }

    public ArrayList<Vertex> getVerticies() {
        return verticies;
    }

    public ArrayList<Edge> getEdges() {
        return edges;
    }

    public void setName(String newName) {
        this.name = newName;
    }

    /* Functions */
    //Adds a vertex to the verticies ArrayList
    public void addVertex(Vertex vert) {
        this.verticies.add(vert);
    }
    //Adds an edge between two verticies by adding each vertex to the other’s list of neighbors
    public void addEdge(Vertex vert1, Vertex vert2, int weight) {
        this.edges.add(new Edge(vert1, vert2, weight));
    }

    //resets the "processed" status on all verticies in a graph.
    public void resetProcessing() {
        for (int i = 0; i < verticies.size(); i++) {
            verticies.get(i).setProcessed(false);
        }
    }

    public boolean singleSourceShortestPath(Vertex source) {
        boolean retVal = true;
        int[] dist = new int[this.verticies.size()];

        //Step 1: initializing the distances
        for (int i = 0; i < this.getVerticies().size(); i++) {
            dist[i] = Integer.MAX_VALUE;
        }
        int srcIndex = Search.linearSearchReturnIndex(verticies, source.getId()); //finds the index of the source vertex in verticies
        dist[srcIndex] = 0; //initializes the distance from the source vertex to the source vertex as 0


        //Step 2: Relax all edges
        for (int i = 0; i < this.getVerticies().size(); i++) {
            for(int j = 0; j < this.getEdges().size(); j++) {
                //relaxation step
                Edge currEdge = this.edges.get(j);
                Vertex edgeOrigin = currEdge.getOrigin();
                int originIndex = Search.linearSearchReturnIndex(verticies, edgeOrigin.getId());
                Vertex edgeDestination = currEdge.getConnection();
                int destIndex = Search.linearSearchReturnIndex(verticies, edgeDestination.getId());
                int weight = currEdge.getWeight();
                if (dist[originIndex] != Integer.MAX_VALUE && dist[originIndex] + weight < dist[destIndex]) {
                    dist[destIndex] = dist[originIndex] + weight;
                }
```

```java
            }
        }

        //Step 3: Test for negative weight cycles
        for (int i = 0; i < this.getEdges().size(); i++) {
            Edge currEdge = this.edges.get(i);
            Vertex edgeOrigin = currEdge.getOrigin();
            int originIndex = Search.linearSearchReturnIndex(verticies, edgeOrigin.getId());
            Vertex edgeDestination = currEdge.getConnection();
            int destIndex = Search.linearSearchReturnIndex(verticies, edgeDestination.getId());
            int weight = currEdge.getWeight();

            if (dist[originIndex] != Integer.MAX_VALUE && dist[originIndex] + weight < dist[destIndex])
                retVal = false;
        }

        //Step 4: Printing
        if (retVal) {
            for (int i = 0; i < this.getVerticies().size(); i++) {
                if (i != srcIndex) {
                    System.out.println(verticies.get(srcIndex).getId() + "␣->␣" + verticies.get(i).getId() + "␣cost␣is␣" + dist[i] + ";␣");
                }
            }
        } else {
            System.out.println("There␣was␣no␣path␣found␣due␣to␣a␣negative␣loop.");
        }

        System.out.println();

        return retVal;
    }
}
```

3.5 Knapsack.java

```java
/*
 * A knapsack object for use in the fractional knapsack problem. Has a capcity, and maintains its current volume and the value of
 * that volume.
 */
public class Knapsack {
    /* Data fields */
    private int capacity;
    private double currVolume;
    private double value;


    /* Constuctors */
    public Knapsack() {
        //default constructor
    }

    public Knapsack(int cap) {
        this.capacity = cap;
    }

    /* Accessors/Mutators */
    public int getCapacity() {
        return capacity;
    }

```

```java
    public double getCurrVolume() {
        return currVolume;
    }

    public double getValue() {
        return value;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    public void setCurrVolume(double currVolume) {
        this.currVolume = currVolume;
    }

    public void setValue(double value) {
        this.value = value;
    }

    /* Functions */
    //adds a scoop to the sack, including the value of that scoop
    public void addScoop(Spice spice) {
        this.currVolume += 1;
        this.value += spice.getPricePerScoop();
    }
}
```

3.6 Spice.java

```java
/*
 * Spice to be used in the fractional knapsack problem using a greedy algorithm. Spice is the
      object being added to the knapsacks.
 * Spice has a quantity measured in scoops, which is the smallest fraction that the spice can be
      separated into. The spice also has
 * a name and a total value, which is used to calculate its price per scoop.
 */

public class Spice {
    /* Data Fields */
    private String name = null;
    private double totPrice;
    private int quantity;
    private double pricePerScoop; //determined when the object is initialized

    /* Constuctors */

    public Spice() {
        //empty constuctor
    }

    public Spice(String name, double price, int qty) {
        this.name = name;
        this.totPrice = price;
        this.quantity = qty;
        recalculatePricePerScoop();
    }

    /* Accessors/Mutators */
    public String getName() {
        return name;
    }

```

```java
    public int getQuantity() {
        return quantity;
    }

    public double getTotPrice() {
        return totPrice;
    }

    public double getPricePerScoop() {
        return pricePerScoop;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
        recalculatePricePerScoop();
    }

    public void setTotPrice(double totPrice) {
        this.totPrice = totPrice;
        recalculatePricePerScoop();
    }

    /* Functions */
    //recalculates
    public void recalculatePricePerScoop() {
        this.pricePerScoop = totPrice / quantity;
    }
}
```