Assignment Five – LaTeX Directed Graph and Greedy Knapsack

Tyler Vultaggio

December 2021

1 Summary

Given a text file containing graph commands on each line, the program will assemble Adjacency Linked Objects that represent each of these graphs, respectively. The program will then display the shortest paths of all paths from a given src. These results will be produced by Bellman-Ford SSSP algorithm. All results will be output to the console.

The second part of the program will take in a given text file that contains a spice or knapsack on each line. Spices will be chained in a linked list and knapsacks will be collected in an array. The program will begin using a greedy solution to fill up the knapsacks.

2 DirectedGraph

Bellman-Ford Dynamic Programming for SSSP– In general it is an algorithm that computes shortest paths from a single source, some node or vertex, such that it spans to all of the other nodes, or vertices, in a weighted directed graph. It is important to note that it does such amazing things by using a breadth first algorithm, not directly, but modified. The highlights of the implementation here are the following: make the graph, initialize all distances to infinity, or some ridiculously large number like I did, relax all edges, then check for any negative cycles. Do note that we make most of our money on the relax function, this is what goes through our graph and makes those shortest paths visible. However, what keeps us from becoming jobless is our check for negative cycles, it saves our ass sometimes AND where memoization occurs (memoization = keep track previous values so we don't waste time, sort of like picking up where you left off, or rather caching results). For example, if a negative cycle goes unnoticed, the graph could effectively have a infinitely small path to some target vertex. That is not good, unless you're explaining the term negative infinitely in a calculus class, then maybe you can just botch the graph and show the students. The run time of this is O(E x V), where E is the total edges, and V is the total vertices. Recall that E must be at least V-1, so lets just say that E = V. Now we get O(V x V). Let V = n, we can see that this is about O((n 2).

2.1 DirectedGraph

```
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/

import java.util.ArrayList;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.InputMismatchException;
import java.util.Scanner;
import java.lang.Math;

public class DirectedGraph
{
        private static final String FILE_NAME = "graphs2.txt";
        private static final File myFile = new File(FILE_NAME);


        public static void readFileVertexAndMakeGraph() throws FileNotFoundException
        {
            String[] regVertex;
            String[] regEdges;
            Scanner input = new Scanner(myFile);
            String currentLine;
            Graphs graph = new Graphs();
            int numGraph = 0;
            while (input.hasNextLine())
            {
                currentLine = input.nextLine();
                if (currentLine.equals("new graph"))
                {
                        graph = new Graphs();
                    numGraph++;
                }
                else if (!(currentLine.equals("")) && (currentLine.charAt(0) != '-'))
                {
                    if (currentLine.charAt(4) == 'v')
                    {
                        // add vertex
                        regVertex = currentLine.split("add vertex ");
                        graph.addVertex(Integer.parseInt(regVertex[1]));
                    }
                    else if (currentLine.charAt(4) == 'e')
                    {
                        // add edge
                        regEdges = currentLine.split("\\s+");
                        graph.addEdgetoGraph(Integer.parseInt(regEdges[2]), Integer.parseInt(r
                    }
                }
                else if (currentLine.trim().isEmpty())
                {
                    System.out.println("\n============================================");
                    System.out.println("Graph # " + numGraph);
                    System.out.println("============================================");
                    System.out.println("\n--------------------------------------------");
                    System.out.println("Linked Object");
                    System.out.println("--------------------------------------------");
                    graph.GraphDetails();
                    System.out.println("\n--------------------------------------------");
                    System.out.println("Paths");
                    System.out.println("--------------------------------------------");
                    graph.BellmanFord(graph, graph.getVertices().get(0));
                }
            }
            //input.close();
    }

    // Not a fan of how I made this class, but I don't have time to make it look any cleaner
    public static void main(String[] args) throws FileNotFoundException
    {
        readFileVertexAndMakeGraph();
    }


}
```

2.2 DirectedGraph(Sub Graph Class

This is version of my graph class is set up so that when edges are added they are added directionally and have a weight on them. There are a few new methods and they are BellmanFord at line 70, printPath at line 121, and path at line 131. BellmanFord finds the path to a vertex from the source vertex at the smallest cost possible(more info about BellmanFord at top of doc). printPath is simply the format for printing out the cost and the path took from the source vertex to the desitnation vertex. path is a recursive function that finds the path that was taken to get to a point(which was saved from BellmanFord).

2.3 Graph Class

```
import java.util.ArrayList;

public class Graphs
{

        private ArrayList<Vertex> vertices = new ArrayList<Vertex>();
        private ArrayList<Edges> edges = new ArrayList<Edges>();

        //This is the graph itself
        public Graphs()
        {

        }

        //Adds a Vertex
        public void addVertex(int vertexID)
        {
                Vertex newVertex = new Vertex(vertexID);
                vertices.add(newVertex);
        }

        //Gets the vertices
        public ArrayList<Vertex> getVertices()
        {
                return vertices;
        }


        //Adds an edge to the graph
        public void addEdgetoGraph(int startId, int endId, int weight)
        {
        int startLoc = getVertexLocation(startId);
        int endLoc = getVertexLocation(endId);
        Edges edge = new Edges();
        edge.setSource(vertices.get(startLoc));
        edge.setDestination(vertices.get(endLoc));
        edge.setWeight(weight);
        if( (startId != -1) && (endId != -1))
        {
            // If exists, lets add it
                vertices.get(startLoc).setEdge(edge);
                edges.add(edge);
            }
            else
            {
                System.out.println("Vertex cannot be connected, make sure it exists");
            }
        }

        //Gets the location of the Vertices
        public int getVertexLocation(int someVertexIdToFind)
        {

                int location = 0;
                boolean found = false;
                while ( found == false && (location < vertices.size()) )
            {
                        if (someVertexIdToFind == vertices.get(location).getID())
                {
                                found = true;
                }
                        else
                {
```

```
                                location ++;
                        }
                }
                        return location;
        }

        public void BellmanFord(Graphs graph, Vertex source)
        {
                int numberOfVertices = vertices.size();
                int numberOfEdges = edges.size();
                int distance[] = new int[numberOfVertices];

                //Sets the distance from the source to all of the vertices to infinity.
                for(int i = 1; i < numberOfVertices; i++)
                {
                        distance[i] = Integer.MAX_VALUE;
                        //System.out.println(vertices.get(i).getID());
                }
                //Sets distance to the source to the source to zero.
                distance[source.getID() - 1] = 0;

                for(int i = 1; i < numberOfVertices; i++)
                {
                        //This finds the shortest path
                        for(int j = 0; j < numberOfEdges; j++)
                        {
                                int from = edges.get(j).getSource().getID() - 1;
                                int to = edges.get(j).getDestination().getID() - 1;
                                int weight = edges.get(j).getWeight();
                                if(distance[from] != Integer.MAX_VALUE && distance[to] > dista
                                {
                                        distance[to] = distance[from] + weight;
                                        //edges.get(j).getDestination().setPrev(edges.get(j).g
                                        vertices.get(to).setPrev(edges.get(j).getSource()); //
                                        //System.out.println(edges.get(j).getDestination().get
                                        //System.out.println(vertices.get(from).getID());
                                }
                        }
                }


                        //This second loop checks to see if there is an infinite negative loop
                        for(int j = 0; j < numberOfEdges; j++)
                        {
                                int from = edges.get(j).getSource().getID() - 1;
                                int to = edges.get(j).getDestination().getID() - 1;
                                int weight = edges.get(j).getWeight();
                                if(distance[from] != Integer.MAX_VALUE && distance[to] > dista
                                {
                                        System.out.println("There is an infinite loop.");
                                        break;
                                }
                        }

                printPath(distance);
        }

```

```
	public void printPath(int distance[])
	{
		for(int i = 1; i < vertices.size(); i++)
		{
			System.out.print("1 -> " + (i+1) + " cost is: " + distance[i] + "; pat
			path(vertices.get(i));
			System.out.println("");
		}
	}

	public void path(Vertex vertex)
	{
		if(vertex.getPrev() != null)
		{
			path(vertex.getPrev());
		}
		if(vertex.getID() == 1)
		{
			System.out.print(vertex.getID());
		}
		else
		{
			System.out.print(" -> " + vertex.getID());
		}
	}


	public void GraphDetails()
	{
		for(int i = 0; i < vertices.size(); i++)
		{
			vertices.get(i).printVertexInfo();
		}
	}

	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	}

}
```

2.4 Directed Graphs(Vertex

This is a cut down version of my old vertex class. This version adds in a pointer to the prev vertex used to get to it, this is needed for the path function in graph.

2.5 Vertex

```
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/

import java.util.ArrayList;

public class Vertex
{

        public int id;
        public ArrayList<Edges> edges = new ArrayList<Edges>();
        public Vertex prev;
        //public int weight;


        public Vertex()
        {
                id = 0;
                edges = null;
                prev = null;
                //weight = 0;
        }

        public Vertex(int id)
        {
                this.id = id;
        }
        //This section is for setting methods
        //______________________________________________
        public void setID(int id)
        {
                this.id = id;
        }
        public void setEdge(Edges edge)
        {
                edges.add(edge);
        }
        public void setPrev(Vertex prev)
        {
                this.prev = prev;
        }
        //______________________________________________

        //This section is for getting methods
        //______________________________________________
        public Integer getID()
        {
                return id;
        }
        public ArrayList<Edges> getEdges()
        {
                return edges;
        }
        public Vertex getPrev()
        {
                return prev;
        }
        //______________________________________________
        public void printVertexInfo()
        {
        System.out.println( "\n__________________________________" );
        System.out.println( "Vertex ID: " + getID() );
        for( int i = 0; i < edges.size(); i++ )
        {
            System.out.println("Edge: " + edges.get(i).getSource().getID() + " - " + edges.get
        }
        }

        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }

}
```

2.6 Directed Graphs(Edges

This class is made up of 2 vertex and the weight between them.

2.7 Edges

```
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/
public class Edges
{
        public Vertex source;
        public Vertex destination;
        public int weight;

        public Edges()
        {
                source = null;
                destination = null;
                weight = 0;

        }

        //This section is for setting methods
        //__________________________________________
        public void setSource(Vertex edge)
        {
                this.source = edge;
        }
        public void setDestination(Vertex edge)
        {
                this.destination = edge;
        }
```

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

	//This section is for getting methods
	//_____________________________________________
	public Vertex getSource()
	{
		return source;
	}
	public Vertex getDestination()
	{
		return destination;
	}
	public Integer getWeight()
	{
		return weight;
	}
	//_____________________________________________


	/*
	 * public void printEdgeInfo() { System.out.println("Edge:" + edge.getID() +
	 * "Weight: " + getWeight()); }
	 */

	/**
	 * @param args
	 */
	public static void main(String[] args)
	{
		// TODO Auto-generated method stub

	}

}
```

3 Greedy

Fractional Knapsack, Greedy Algorithms – In general, greedy algorithms identify a sub problem that makes the problem move to the next one. In a way it identifies subsubproblems, in a continuous manner. Once identifying the problem, it will then take the best choice in that instance of time, sort of like getting the instantaneous rate of change on a graph (that instant varies... a lot ... and could, sometimes, not be most optimal). In this implementation, as we went over in class, the algorithm will order all spices in order from least to greatest cost. Once ordered, the algorithm will determine how much space is available in a knapsack, and use this to prep for the heuristic. Making the jump to light speed, the algorithm will fill the given knapsack with spice until that spice has run out, we love spices Alan. If there is still room in the knapsack, we will continue to the next spice and repeat the process in a continuous manner until our knapsack fills up. Have no fear, if we can not take a whole stock of spice, we'll just do some chopping with fractions so we can still be greedy. Greedy, huh?   As mentioned in class, the run time of this is definitely O(nlogn). Recall that logn is the result of sorting the objects first and that n is the result of touching each of those objects. Multiplying through we clearly get nlogn

3.1 Fractional Kanpsack

```java
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/

import java.util.ArrayList;
import java.util.Collections;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FractionalKnapsack
{
        private static final String FILE_NAME = "spice.txt";
        private static final File myFile = new File(FILE_NAME);

        public static void readFileKnapsackandSpices() throws FileNotFoundException
        {
                 String[] regSpices;
         String[] regKnapsacks;
         Knapsack[] knapsacks = new Knapsack[5];
         int count = 0;
         Scanner input = new Scanner(myFile);
         String currentLine;
         ArrayList<Spices> allSpices = new ArrayList<Spices>();


         System.out.println("Greedy Spices :");
         System.out.println("=============================================");

         while(input.hasNextLine())
         {
                 currentLine = input.nextLine();
                 if(!(currentLine.contentEquals("")) && (currentLine.charAt(0) != '-'))
                 {
                         if(currentLine.charAt(0) == 's')
                         {
                                 String temp = currentLine;
                                 temp = temp.replaceAll(";","");
                                 regSpices = temp.split(("\\s+"));
                                 Spices newSpice = new Spices(regSpices[3], Double.parseDouble
                                 allSpices.add(newSpice);
                         }
                         else if(currentLine.charAt(0) == 'k')
```

```
                        {
                                String temp = currentLine;
                                temp = temp.replaceAll(";","");
                                regKnapsacks = temp.split(("\\s+"));
                                Knapsack newSack = new Knapsack(Integer.parseInt(regKnapsacks
                                knapsacks[count] = newSack;
                                count++;
                        }
                }
        }
        Collections.sort(allSpices, Spices.SpicePricePer);

        //Prints the sorted arrayList of Spices in descending order of Price_per_qty
        //This is to see that it sorted how I wanted it to

        for (Spices spice : allSpices)
        {
            spice.printSpice();
        }

        //The link below is where I learned the above from, I adapted what they did to make i
        //https://www.geeksforgeeks.org/how-to-sort-an-arraylist-of-objects-by-property-in-ja

        int knapsackCount = 0;
        int spiceCount = 0;
        double spicePortion;
        double differnceInSpace;
        System.out.println("\n=================================================");
        System.out.println("Filling the KnapSacks");
        System.out.println("=================================================");

        while(knapsackCount < knapsacks.length)
        {
                if(spiceCount < allSpices.size() && knapsacks[knapsackCount].getCurrent_Size(
                {
                        if(knapsacks[knapsackCount].getCurrent_Size() + allSpices.get(spiceCo
                        {
                                //Adds the value to the Knapsack.
                                knapsacks[knapsackCount].addValue(allSpices.get(spiceCount).g
                                //Adds the space taken up by the amount of spice.
                                knapsacks[knapsackCount].addSize(allSpices.get(spiceCount).ge
                                //Prints what was added to the Knapsack
                                System.out.println("Added all of " + allSpices.get(spiceCount
                                spiceCount++;
                        }
                        else
                        {
                                //Finds the space left in the Knapsack.
                                differnceInSpace = (knapsacks[knapsackCount].getTotal_Size()
                                //Calculates how much of the spice it can add the to Knapsack
                                spicePortion = differnceInSpace/allSpices.get(spiceCount).get
                                //Adds the value to the Knapsack.
                                knapsacks[knapsackCount].addValue(allSpices.get(spiceCount).g
                                //Adds the space taken up by the amount of spice.
                                knapsacks[knapsackCount].addSize(allSpices.get(spiceCount).ge
                                //Prints what was added to the Knapsack
                                System.out.println("Added qty: " + spicePortion + " of spice:
                                spiceCount++;
                        }
                }
                else
                {
                        //Print out the properties of the KnapSack.
                        System.out.println("Knapsack of capcity " + knapsacks[knapsackCount].
                        System.out.println("\n_______________________________________________
                        knapsackCount++;
                        spiceCount = 0;
                }
        }

       }


public static void main(String[] args) throws FileNotFoundException
{
       readFileKnapsackandSpices();

}

}
```

3.2 Greedy(Spices)

This class contains all the properties that has to do with a spice. There is also a sort by price per qty function that I learned so that I could sort an arrylist of spices by that property.

3.3 Spices

```
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/

import java.util.Comparator;

public class Spices
{
        public String color;
        public double total_price;
        public int qty;
        public static double price_per;

        Spices()
        {
                color = null;
                total_price = -1;
                qty = -1;
                price_per = -1;
        }

        Spices(String color, double total_price, int qty)
        {
                this.color = color;
                this.total_price = total_price;
                this.qty = qty;
                //price_per = total_price/qty;
        }

        //This section is for setting methods
        //_____________________________________________
        public void setColor(String color)
        {
                this.color = color;
        }
        public void setTotal_Price(double total_price)
        {
                this.total_price = total_price;
        }
        public void setQty(int qty)
        {
                this.qty = qty;
        }
        public void setPrice_Per(double total_price, int qty)
        {
                price_per = total_price/qty;
        }
        //_____________________________________________

        //This section is for getting methods
        //_____________________________________________
        public String getColor()
        {
                return color;
        }
        public double getTotal_Price()
        {
                return total_price;
        }
        public int getQty()
        {
                return qty;
        }
        public double getPrice_per()
        {
                double total_price = getTotal_Price();
                int qty = getQty();
                return total_price/qty;
        }
        //_____________________________________________

        //Prints out all of the info for a Spice
        //_____________________________________________
        public void printSpice()
        {
                System.out.println("______________________");
                System.out.println("Spice Color: " + getColor());
                System.out.println("Spice Total Price: " + getTotal_Price());
                System.out.println("Spice Qty: " + getQty());
                System.out.println("Spice Price per Qty: " + getPrice_per());

        }

        public static Comparator<Spices> SpicePricePer = new Comparator<Spices>()
        {
        public int compare(Spices spice1, Spices spice2)
        {

            double Price_Per1 = spice1.getPrice_per();
            double Price_Per2 = spice2.getPrice_per();

            // For descending order
            return (int) (Price_Per2 - Price_Per1);

        }
    };


        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }

}
```

3.4 Greedy(Knapsack)

This is a class that just holds all of the info for a knapsack.

3.5 Knapsack

```
/** @author Tyler Vultaggio
* Assignment 5
* Due Friday 12/10/2021
* Algorithms
*/

//import java.util.ArrayList;

public class Knapsack
{
        public int total_size;
        public double current_size;
        public double value;

        public Knapsack()
        {
                total_size = 0;
                current_size = 0;
                value = 0;
        }

        public Knapsack(int newSize)
        {
                total_size = newSize;
                current_size = 0;
                value = 0;
        }

        //This is for setting methods
        //_________________________________
        public void setTotal_Size(int size)
        {
                this.total_size = size;
        }
        public void addSize(double size)
        {
                current_size = current_size + size;
        }
        public void addValue(double newValue)
        {
                value = value + newValue;
        }
        //________________________________

        //This is for getting methods
        //________________________________
        public int getTotal_Size()
        {
                return total_size;
        }
        public double getCurrent_Size()
        {
                return current_size;
        }
        public double getValue()
        {
                return value;
        }
        //_________________________________
        public static void main(String[] args)
        {
                // TODO Auto-generated method stub

        }

}
```