1 Graph Node and Spice Object

The spice object is used for fractional knapsack. It contains a name, total price, quantity and unit price attribute. The GraphNode object is much like an edge in a graph. It contains a weight, from and to attribute. I decided to make the graphnodes like edges because I thought it was good practice

```
//spice object contains a name, total price, quantity and unit price.
public class spice {
    String name = "";
    int total_price;
    int qty;
    int unity_price;
}
//GraphNode class contains attribute to simulate an edge in a graph
public class graphNode{
    int weight;
    int from;
    int to;

}
```

2 Making Graphs

It looks like there is a lot going on in this class, but there really isn't. In the Bellman Function, There is 3 steps going on. The first step is the single source function. The Single source function will put all of the distances for the objects at a value of 1000. The only value that will be 0 is the single source. Since were starting at vertex 1 for all these graphs, The default makes the value for vertex 1, zero. The next step in the function is the relax function. The function goes through every edge and vertices and calls this function. The relax function will compare the current vertex distance with other known distances until it exhausts all the elements in the edges and vertices. The last function will check for negative cycles in the list.

```
import java.util.ArrayList;

public class graph {
    //Contains vertices and edges
    ArrayList<graphNode> vertices = new ArrayList<graphNode>();
    ArrayList<graphNode> edges = new ArrayList<graphNode>();

    public static void bellman(graph G) {
        int distanceAttribute[] = new int[G.vertices.size()];
        boolean newBool = true;
        singleSource(G, distanceAttribute);
        //For every edge and vertice, we will call the relax function.
        for (int i = 1; i < G.vertices.size(); i++) {
            for (int j = 0; j < G.edges.size(); j++) {
        relax(G.edges.get(j).weight, G.edges.get(j).from, G.edges.get(j).to, distanceAttribute);
            }
        }
        // Find a negative cycle in the current graph
        //Won't print out the graph if it's negative
        for (int j = 0; j < G.edges.size(); j++) {
         if (distanceAttribute[G.edges.get(j).to - 1] > (distanceAttribute[G.edges.get(j).from - 1]
                 + G.edges.get(j).weight)) {
             newBool = false;
            }

        }
        //This prints out the graphs cost from the source to all its destinations
        if (newBool) {
            System.out.println("\n" + "New Graph ");
            for (int i = 0; i < G.vertices.size(); i++) {
        System.out.println(1 + " -> " + (i + 1) + " cost is " + distanceAttribute[i]);
            }
        }
```

```
    }
    //This will relax all of the vertices which means put their distance to infinity.
    //In this case I put it to 1000 because the weight will never exceed that
    //Vertex 0's distance is 0 because we start there.
    public static void singleSource(graph newGraph, int distanceAttribute[]) {
        for (int i = 0; i < newGraph.vertices.size(); ++i) {
            distanceAttribute[i] = 1000;

        }
        distanceAttribute[0] = 0;
    }

    public static void relax(int weight, int from, int to, int[] distanceAttribute) {

if (distanceAttribute[from - 1] != 1000 && distanceAttribute[from - 1] +
weight < distanceAttribute[to - 1]) {
        distanceAttribute[to - 1] = distanceAttribute[from - 1] + weight;
        }
    }

}
```

3 Spice Algorithm

The Spice algorithm goes through the spice array and gets the capacities and spice objects from the whole array. Once the spice objects are in the array, the array is then sorted by descending order. This is used primarily for the fractional knapsack. We want to look at the highest element per unit first then the second highest and so on. This is classified as a greedy approach. To successfully demonstrate this problem, I put the capacities in an array. The for loop will go through the capacities and have an inner for loop for the spice objects.The current capacity will be compared to the current spice object. If the capacity is more than the quantity, the algorithm will just take the whole spice and subtract the amount taken. If the capacity is less than the quantity, the algorithm will take a fraction of the total spice.

```
public static void spiceAlgorithm(String[] spiceArray) {
        // contains the capacities and spice objects from the file
        ArrayList<Integer> capacities = new ArrayList<Integer>();
        ArrayList<spice> spiceObjects = new ArrayList<spice>();
        String spicesUsed = "";

        // This will go through the whole spice array
        for (int i = 0; i < spiceArray.length; i++) {

            String[] lineSplit = spiceArray[i].split(" ");
            // Splits the current line
            // checks to see if first index of split is spice

            if (lineSplit[0].equals("spice")) {

                // Splits the line again by semicolon
                String[] firstSplit = spiceArray[i].split(";");

                // Gets the name,price and quanitity lines
                String[] nameLine = firstSplit[0].split(" ");
                String[] priceLine = firstSplit[1].split(" ");
                String[] quantityLine = firstSplit[2].split(" ");

                // creates newSpice object
                spice newSpice = new spice();

                // Gets the name,price and quantity from new split lines
                int price = (int) Float.parseFloat(priceLine[priceLine.length - 1]);
                int quantity = Integer.parseInt(quantityLine[quantityLine.length - 1]);
```

```
            String name = nameLine[nameLine.length - 1];

            // This gives the current spice object different attributes.
            newSpice.total_price = price;
            newSpice.qty = quantity;
            newSpice.name = name;
            newSpice.unity_price = price / quantity;

            // adds the spice object to newSpice arrayList
            spiceObjects.add(newSpice);
        }
        if (lineSplit[0].equals("knapsack")) {
            String[] firstSplit = spiceArray[i].split(";");
            String[] getCapacity = firstSplit[0].split(" ");

            capacities.add(Integer.parseInt(getCapacity[getCapacity.length - 1]));
        }
    }

    int arrayLength = spiceObjects.size();
    // Reverses the order of the array in descending order
    for (int i = 0; i < arrayLength - 2; i++) {

        int smallPosition = i;

        for (int j = i + 1; j < arrayLength; j++) {

            if (spiceObjects.get(j).unity_price > spiceObjects.get(smallPosition).unity_price) {

                smallPosition = j;
            }
        }
        // this is the swapping work that is done. it finds the the next smallest
        // element and swaps its position with i
        spice temp = spiceObjects.get(smallPosition);
        spiceObjects.set(smallPosition, spiceObjects.get(i));
        spiceObjects.set(i, temp);
    }

    for (int i = 0; i < capacities.size(); i++) {

        // this keeps the runningTotal of the total price
        int runningTotal = 0;
        int currentCapacity = capacities.get(i);
        for (int k = 0; k < spiceObjects.size(); k++) {

            spice currentSpice = spiceObjects.get(k);
            if (currentCapacity == 0) {
                break;
            }
            else {
                if (currentCapacity >= currentSpice.qty) {

                    // This adds the total price of the current spice to the running total
                    runningTotal += currentSpice.total_price;
                    // The current capacity then gets the current qty subtracted from it
                    currentCapacity -= currentSpice.qty;
                    spicesUsed += currentSpice.qty + " scoops of " + currentSpice.name + " ";
                } else {
                    // Divides the current capacity by current spice quantity.
                    // This number is used to multiply by the total price
                    // which in return gives us how much spice to take
                    float fraction = (float) currentCapacity / currentSpice.qty;
                    float newVal = currentSpice.total_price * fraction;
```

```
                    runningTotal += (int) newVal;

                    spicesUsed += currentCapacity + " scoops of " + currentSpice.name
                            + " ";
                    currentCapacity = 0;

                }
            }
        }

        // This prints out the current capacity and the spices collected
        System.out.println("Knapsack of capacity " + capacities.get(i) + " is worth "
                + runningTotal + " and contains " + spicesUsed);
        spicesUsed = "";
    }

}
```

4 Asymptotic running times for algorithms

Spice Algorithm	SSSP Bellman
N/A	N/A
$O(n^2)$	O(—V— + —E—)

The spice algorithm runs in n^2 time because I'm using selection sort to sort the algorithm in descending order. If I wanted the algorithm to be faster, I could use quick sort or merge sort. This would lower the time to n log n.

SSSP and bellman ford algorithm has a running time of the number of vertices plus the number of edges. This is because of the for loops involved. The algorithm loops through the number of edges first and then the number of vertices. This is used at the relax part of the algorithm.