CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Four

Jake Vissicchio
Jake.Vissicchio1@Marist.edu

December 11, 2021

1 Analysis

1.1 Single Source Shortest Path:

Has a complexity of $O(V * E)$. With V = total vertices and E = total edges.
This is because the original initialization goes to each vertex giving us $O(V)$ and we must cross every edge for each vertex at least once to find the best possible distance value giving us $O(E)$.
Note: if you see a nested loop this will often imply multiplication rather than addition.

1.2 Fractional Knapsack:

Has a complexity of $O(nlog(n))$. With n being the total amount of spices in our case.
This is because all it takes is sorting the unit price by highest to lowest and then taking, starting from the most valuable, until we cannot take anymore.
As we know, sorting algorithms such as QuickSort and MergeSort are $O(nlog(n))$.

2 Code Listings

2.1 Spice Object Class:

```
public class SpiceVissicchio
{
    String color = "";
    double total_price = 0;
    int quantity = 0;
    double unit_price = 0;

    //constructor for SpiceVissicchio
    public SpiceVissicchio (String newColor, double newTotal, int newQuant)
    {
        color = newColor;
        total_price = newTotal;
        quantity = newQuant;
        //unit_price is found by dividing the total_price by the total quantity
        unit_price = total_price/quantity;
    }

    //Getters and Setters to use in main
    public String getColor ()
    {
        return color;
    }

    public double getTotal ()
    {
        return total_price;
    }

    public void setQuantity(int newQuantity)
    {
        quantity = newQuantity;
    }

    public int getQuantity ()
    {
        return quantity;
    }

    public double getUnitPrice ()
    {
        return unit_price;
    }
}
```

This class creates the Spice Object as well as calculates what the unit price for each spice would be. It also allows access to some getters and setters for us to use in main.

2.2 QuickSort:

```
//Using a reverse QuickSort to sort the spices from highest unit_price
//to lowest unit_price
public class QuickVissicchio
{
    int comparisonCount = 0;

    //swaps two given elements in an array
    void swap(SpiceVissicchio array[], int a, int b)
    {
        SpiceVissicchio temp = array[a];
        array[a] = array[b];
        array[b] = temp;
    }//swap

    //jediPartion attempts to make sure that we do not have Sith Lord
    //partitioning (not choosing worst possible pivot which are
    //is the minimum or maximum value)
    int jediPartition(SpiceVissicchio array[], int min, int max)
    {
        SpiceVissicchio pivot = array[max];

        int i = (min - 1);

        for (int j = min; j <= max - 1; j++)
        {
            comparisonCount++;
            //int result = array[j].compareToIgnoreCase(pivot);
            if (array[j].getUnitPrice() > pivot.getUnitPrice())
            {
                comparisonCount++;
                i++;
                swap(array, i, j);
            }//if
        }//for

        //makes use of swap to swap array[i+1] with array[max]
        swap(array, i + 1, max);
        return (i + 1);
```

```
    }//jediPartition

    //will sort the given array by making use of partition()
    void quickSort(SpiceVissicchio array[], int min, int max)
    {
        if (min < max)
        {
            comparisonCount++;
            //the location within the array where the partition
            //is taking place
            int partLocation = jediPartition(array, min, max);

            //sort the elements before and after the partition location
            quickSort(array, min, partLocation - 1);
            quickSort(array, partLocation + 1, max);
        }//if
    }//quickSort
}//QuickVissicchio
```

Using QuickSort to order my array of Spice Objects from highest unit price to lowest unit price.
This QuickSort is modified from the last time I used it.
First I needed to make sure that it catered to comparing by the object's unit price rather than by alphabetic ordering unlike before.
I also needed to make sure that it is reversed it sorts by highest to lowest

2.3 How was I able to print out my Spice Heist?:

```
//This is to make sure we can print correctly without actually knowing
//how many spices there are
System.out.print("After ");
for (j = 0; j < spiceList.length; j++)
{
    while ((scoops < spiceList[j].getQuantity()) && (capacity > 0))
    {
        scoops++;
        knapsackPrice = knapsackPrice + spiceList[j].getUnitPrice();
        capacity--;
    }
        if (scoops != 0)
        {
        if (scoops == 1)
                {
                    System.out.print(scoops + " scoop of " +
                    spiceList[j].getColor() + ", ");
```

```
                }
                else
                {
                    System.out.print(scoops + " scoops of " +
                        spiceList[j].getColor() + ", ");
                }
            }
            scoops = 0;
}
System.out.print("the knapsack of capacity, "
    + originalCapacity + ", is worth " + knapsackPrice + " quatloos");
```

One of the challenges I faced while doing this assignment was how can I print out the information without being aware of how many spices there are? What I first did was go through the file once just to count the spices then go through the file again to actually take in the information and the list of spices, then sorting them and getting the capacity of the knapsack.
The while loop actually performed taking the scoops and setting the total price and the for loop handled going to the next spice and printing out the current information. Once everything is complete we can finally print out the information on the total price.

2.4 Edge Class:

```
    public class EdgeVissicchio
    {
        //from represents the source vertex
        int from = 0;
        //to represents the destination vertex
        int to = 0;
        //cost represents the cost of the source vertex going to the
        //destination vertex
        int cost = 0;

    //constructor for EdgeVissicchio
        public EdgeVissicchio(int newFrom, int newTo, int newCost)
        {
                from = newFrom;
                to = newTo;
        cost = newCost;
        }//EdgeVissicchio
}
```

The Edge class is used to create new Edge objects for our graphs.
New Edge objects need a source vertex, a destination vertex, and the cost of making that movement from the source to destination.

2.5 Graph Class:

```
    import java.util.ArrayList;

public class GraphVissicchio
{
    public ArrayList<EdgeVissicchio> edgeList;
    public int vertexTotal;

    //constructor for GraphVissicchio
    public GraphVissicchio(int vertexCount)
    {
        //vertexTotal allows us to know the total amount of edges
        //within the graph
        vertexTotal = vertexCount;
        edgeList = new ArrayList<EdgeVissicchio>();
    }


    //add the edge by creating a new edge and adding it to the edgeList
    public void addEdge(GraphVissicchio graph, int from, int to, int cost)
    {
        EdgeVissicchio newEdge = new EdgeVissicchio(from, to, cost);
        graph.edgeList.add(newEdge);

    }

    public boolean bellmanFord(GraphVissicchio graph, int source)
    {
        //distance holds an array of integers that represent the
        //distance from and to each vertex
        int distance[] = new int[vertexTotal];

        //Initialize single source
        for (int l = 0; l < vertexTotal; l++)
        {
            //set each distance to infinity, or in this case the
            //maximum possible integer
            distance[l] = Integer.MAX_VALUE;
        }
        //the distance from the source to itself should be 0 since it
        //should not have to go anywhere
        //(unless of course there is a negative weight cycle, then the
        //distance to itself would be infinitely negative)
        distance[source] = 0;
```

```
        for (int i = 1; i < vertexTotal - 1; i++)
        {
            for (int j = 0; j < graph.edgeList.size(); j++)
            {
            //Relax:
            //if the source vertex is infinite skip it for now
            //if the set distance to a destination vertex is larger
            //than the current source + the cost
            //overwrite the destination vertex
            if (distance[graph.edgeList.get(j).from] != Integer.MAX_VALUE &&
                distance[graph.edgeList.get(j).to] >
                distance[graph.edgeList.get(j).from] +
                graph.edgeList.get(j).cost)
            {
                distance[graph.edgeList.get(j).to] =
                distance[graph.edgeList.get(j).from] +
                    graph.edgeList.get(j).cost;
            }//if
            }
        }
        //Checking for negative weight cycles, if we get a negative
        //weight cycle return false
        for (int k = 0; k < graph.edgeList.size(); k++)
        {
            if (distance[graph.edgeList.get(k).from] != Integer.MAX_VALUE &&
                distance[graph.edgeList.get(k).to] >
                distance[graph.edgeList.get(k).from] +
                graph.edgeList.get(k).cost)
            {
                return false;
            }
        }
        //Printing out the results
        //Lets us know the cost of going to each vertex starting
        //from the source vertex
        System.out.println("Results:");
        System.out.println("-----------------------------------------------------------------------");
        for (int i = 1; i < vertexTotal; ++i)
        {
            System.out.println(source + "------>" + i + " cost is "
                + distance[i]);
        }
        System.out.println("");
        return true;
    }
}//GraphVissicchio
```

First I created an ArrayList of Edge Objects since I am not aware of how many edges in total there are for each graph. I also have vertexTotal that keeps track of the total amount of verticies there are in each graph. which is found out in main by counting the amount of times there is a new vertex in the file.

When we create a newGraph in main it is given a vertexTotal and it will create a new ArrayList of EdgeObjects.

addEdge() creates a new Edge Object and then adds it to our edgeList.

bellmanFord() is an implementation the Bellman-Ford dynamic programming algorithm for Single Source Shortest Path (SSSP).
It uses a Graph and a source Vertex.

First, I created distance[] which holds the distance value of each movement of going from the source vertex to the destination vertex based on the cost

Next, I needed to initialize each element of that array by setting each one to the maximum possible integer value then setting the distance from the source vertex to the source vertex to be 0 since it should not have to go anywhere (unless of course there is a negative weight cycle, then the distance to itself would be infinitely negative, but don't worry we'll check for that later)

Then, I relaxed each element.
If the if the source vertex is the max value skip it
for now if the distance value of the destination vertex is larger than the current distance value of the source vertex + the cost, overwrite the distance value of the destination vertex

Next, I checked for any negative weight cycles by reusing our if statement from the nested for loops when in Relax. Returns false if it goes the if statement goes through

Finally, printing the results and returning true if it did not previously return false.