Lab 5: Bellman-Ford SSSP and Greedy Algorithms

Nicholas Suchy
Nicholas.Suchy1@Marist.edu

May 7, 2023

1 Introduction

The goal of this lab was to implement our own versions of the Bellman-Ford Single Source Shortest Path Algorithm and also create a version of the Fractional Knapsack Problem.

For the Bellman-Ford SSSP algorithm, we were to reuse our linked objects graph from the previous lab except this time, make it a directed graph that could support weighted edges. The data for the graphs was read in from the file graphs2.txt.

As for the implementation of the Fractional Knapsack Problem data was read in from the spices.txt file.

2 Code Reuse and Changes From Previous Labs

This lab required the use of many previously made classes and code snippets from previous labs including but not limited to: LinkedList, Stack, and of course LinkedObjectsGraph.

The LinkedList and Stack classes remained largely unchanged. They were implemented in order to help the functionality of new functions and methods used in the lab.

The LinkedObjectsGraph and GraphNode classes need modification in order to support directed and weighted edges.

```
 public void addEdge(int vertex1, int vertex2, int weight) {
     vertices[vertex1].addNeighbor(vertices[vertex2], weight);
 }
```

To support directed edges, when a new edge is added, it's only added in the specified direction (from Vertex1 to Vertex2).

```
int id;
int distance;
GraphNode previous;
GraphNode neighbors[];
int weights[];
```

```
boolean visited;
```

GraphNode was given to new characteristics to aid with the implementation of the Bellman-Ford SSSP algorithm. Firstly, the GraphNodes were given a pointer to a previous Vertex, which is used to keep track of the shortest path to the Node from the source. Additionally, GraphNodes were given a distance variable. This holds the distance the vertex is from the source. It is intialized to 99999, to represent that a Node is "an infinte" distance away when beginning the Bellman-Ford algorithm.

```
public void addNeighbor(GraphNode vertex, int weight){
    int newLength = (this.neighbors.length + 1);
    GraphNode newNeighbors[] = new GraphNode[newLength];
    int newWeights[] = new int[newLength];
    for (int i = 0; i < this.neighbors.length; i++) {
        newNeighbors[i] = this.neighbors[i];
        newWeights[i] = this.weights[i];
    }
    this.neighbors = newNeighbors;
    this.weights = newWeights;
    this.neighbors[this.neighbors.length - 1] = vertex;
    this.weights[this.weights.length - 1] = weight;
}
```

Finally, to account for weighted edges, GraphNodes were given an integer array that holds weights. This array aligns with the neighbors array. So the weight at index 1 of weights[] is the weight of the edge from the current Vertex to the Vertex at index 1 of neighbors. The addition of weights to the graph is parallel to adding edges and the edited addNeighbors function of GraphNode is shown above. Note how when an vertex is added to neighbors at a given index, the weight to get to that vertex is added to weights at the same index.

3 Structure

I decided to create various files for each aspect of the lab. This division helped me maintain organization, and greatly helped with debugging as I did have just one incredibly long file.

4 Bellman-Ford SSSP Algorithm

This section pertains to everything related to implementing the Bellman-Ford SSSP algorithm, from reading in the data file to presenting the results.

4.1 Reading in graph2.txt

```
int weight = Integer.parseInt(linePart[linePart.length - 1]);
...
linkObjGraph.addEdge(vertex1, vertex2, weight);
```

The code for reading in the file containing the graph information for the lab remained largely unchanged. The program reads a line and splits is based on where any space (" ") is found and creates an array of substrings. Then, it checks the sub-string at the first index of the newly created array and creates a new graph, adds a new vertex, or creates an edge depending on what is found.

The only change comes when the program is adding an edge. The program reads in the weight from the file and then when creating an edge, adds the newly read weight to the parameters.

Additionally, all graphs start with a "0 vertex". So Vertex1 is actually Vertex0 when reading in the file.

4.2 Implementing the Algorithm

This section is dedicated to the various methods and the actual function that make up the Bellman-Ford algorithm.

4.2.1 initSingleSource()

```
public void initSingleSource(GraphNode startVertex){
    for (int i = 0; i < this.vertices.length; i++) {
        this.vertices[i].distance = 99999;
        this.vertices[i].previous = null;
    }
    startVertex.distance = 0;
}
```

This is a simple method called when the Bellman-Ford algorithm is called. It sets up each vertex for the function, setting their distance from the source to 99999 to represent infinity. Additionally, the nodes pointer to a previous vertex is set to null. Finally, the source vertex's distance is set to 0 as it is the source.

4.2.2 relax()

```
public void relax(GraphNode vertex1, GraphNode vertex2, int weight) {
    if(vertex2.distance > vertex1.distance + weight) {
        vertex2.distance = vertex1.distance + weight;
        vertex2.previous = vertex1;
    }
}
```

Another simple method used by the Bellman-Ford algorithm. It compares the current distance a vertex is from the source to a potential distance, which is comprised of another vertex's distance from the source plus the weight of the edge to travel to the current vertex. If the current vertex's distance from the source is greater than the potential distance, the vertex adopts the potential vertex as it's new distance and also takes the vertex used in the potential distance calculation and sets it to be it's previous vertex.

4.2.3 bellmanFord()

```
 public boolean bellmanFord(GraphNode startVertex) {
     int j;
     int k;
     initSingleSource(startVertex);

     for(int i = 1; i < this.vertices.length - 1; i++) {
         j = 0;
         k = 0;
         while(j < this.vertices.length) {
             if(this.vertices[j].neighbors.length == 0){
                 k = 0;
                 j++;
```

```
                } else {
                    relax(this.vertices[j], this.vertices[j].neighbors[k], this.vertices[j].weigh
                    k++;
                    if(k == this.vertices[j].neighbors.length) {
                        k = 0;
                        j++;
                    }
                }
            }
        }
        j = 0;
        k = 0;
        while(j < this.vertices.length) {
            if(this.vertices[j].neighbors.length == 0){
                k = 0;
                j++;
            } else {
                if(this.vertices[j].neighbors[k].distance > this.vertices[j].distance + this.vert
                    return false;
                }

                k++;
                if(k == this.vertices[j].neighbors.length) {
                    k = 0;
                    j++;
                }
            }
        }
        return true;
    }
```

The above algorithm is my implementation of the Bellman-Ford algorithm we had discussed in class. I heavily models the pseudo-code we talked about and reviewed.

The method takes one vertex as a source for the algorithm to work off of. It then calls initSingleSource() to setup each vertex for the algorithm. Counters j and k are initialized to 0. J is used to represent a vertex while k is used to represent a vertex's neighbor and weight. The method checks to see if the current Node has any neighbors. If a neighbor is found, the relax() method is called and the algorithm moves on to check the next neighbor. If the function were to check a neighbor that doesn't exist (is out of bounds to the current vertex's nieghbors array) the function moves onto the next vertex and begins checking its neighbors and calling relax(). This effectively checks every edge in the graph. This process is repeated for the amount of vertices in the graph - 2. Once this process is completed, the function does one more check of all the edges in the graph. This time instead of calling relax() on each neighbor, it checks the comparison that relax() would make. If the comparison (the vertex's current distance is larger than the potential distance) is true, the function returns false. This means that there exists a negative loop in the graph (one vertex's distance from the source will continually decrease until it reaches negative infinity. Otherwise, if the comparison for edges to all vertices is false then the shortest path from the source to each vertex has been found.

The asymptotic run time of the Bellman-Ford algorithm describe above can best be described as linear. The algorithm loops through all edges by a factor of number of vertices in the (graph - 2). While not quite quadratic, the run time for the algorithm runs on the longer side of linear, but can still be said to have a linear run time.

4.3 Presenting the Algorithm

This section is dedicated to the presentation of the results of the Bellman-Ford algorithm

4.3.1 path()

```
public String path(GraphNode vertex1, GraphNode vertex2) {
    Stack reversePath = new Stack();
    GraphNode tempNode = vertex2;
    String path = "";

    reversePath.push(Integer.toString(vertex2.id));

    while(tempNode != vertex1) {
        reversePath.push(Integer.toString(tempNode.previous.id));
        tempNode = tempNode.previous;
    }

    while(!reversePath.isEmpty()) {
        path += reversePath.pop();
        if (!reversePath.isEmpty()) path += " -> ";
    }

    return path;
}
```

This method produces the shortest path determined by the Bellman-Ford algorithm in a visual manner. As vertex's know their previous vertex, the destination vertex's id is pushed onto a stack, then the destination vertex's previous id is pushed onto the stack. This is repeated until the source id is pushed onto the stack. The contents of the stack are then popped off one by one and stored in path to produce a readable path.

4.3.2 presentation()

```
  public static void presentation() {
      counter++;
      System.out.println("------------------------");
      System.out.println("Graph " + counter + ":");
      System.out.println("------------------------");
      System.out.println();
      if(linkObjGraph.bellmanFord(linkObjGraph.vertices[0])) {
          for(int i = 1; i < linkObjGraph.vertices.length; i++) {
              System.out.println("The cost from V0 to V" + (i) + " is: " +
linkObjGraph.vertices[i].distance);
              System.out.println("Path: " +
linkObjGraph.path(linkObjGraph.vertices[0], linkObjGraph.vertices[i]));
              System.out.println();
          }
      }
      linkObjGraph.resetVertices();
  }
```

The above method does all the actual printing of information from produced by the Bellman-Ford algorithm. It neatly organizes and prints out the distance each node in the graph is from the source. It also calls path() and prints out the resulting path. At the end, a summary of the entire graph is produced in terms of distance from the source vertex and the shortest path from the source to the destination.

5 Greedy Algorithms

This section covers the Greedy Algorithm implemented in this lab. In this case, it covers the implementation of the fractional knapsack problem.

5.1 Reading in spice.txt

```java
if (linePart[0].equals("spice")) {

    newSpice = new Spice();

    String[] spicesInfo = line.split(";");

    for (int i = 0; i < spicesInfo.length; i++){

         String[] specficInfo = spicesInfo[i].split(" ");

         if (i == 0) {
             newSpice.name = specficInfo[(specficInfo.length - 1)];

         } else if (i == 1) {
             newSpice.totalPrice = Float.parseFloat(specficInfo[(specficInfo.length - 1)]);

         } else if (i == 2) {
             newSpice.quantity = Integer.parseInt(specficInfo[(specficInfo.length - 1)]);
         }
     }

     newSpice.unitPrice = newSpice.totalPrice / newSpice.quantity;

     if (numSpices == spices.length) spices = resizeSpices(spices);
     spices[numSpices] = newSpice;
     numSpices++;

} else if (linePart[0].equals("knapsack")) {

     knapsack.spices = spices;
     availableSpace = linePart[linePart.length - 1];
     availableSpace = availableSpace.replace(";", "");
     knapsack.capacity = Integer.parseInt(availableSpace);
     knapsackPresentation();
}
```

When it comes to reading in the spice.txt file, it follows a same premise of reading a line and initially splitting it into an array of sub-strings based on where spaces are. However, as the file contains different data, it is specialized to read in data in the way spice.txt is formatted. If the first element of the array of sub-strings equal "spice", a new spice is created and the line is then split based on where semicolons (";") exist. This is because each data point needed is followed by a semicolon. By splitting it here, we can then split the sub-string split by the semicolon by spaces. The data point needed will always be the last sub-string in the newly created array of sub-strings so we can grab it and assign it to the newly created spice's name, total price, or quantity, based on which of the semicolon array of sub-strings it was found in.

Once the data is read and the spice is created, a unit price, which gets assigned to the spice, is calculated. The spice is then added to an array of spices so it can be referenced later. Additionally, it is important to note that the size of the spices array is initialized to 4 as for the assignment. However, if at any point the spice array was to fill up, the array would be resized to accommodate for the new spice.

If the first part of the initial array of sub-strings is "knapsack" then the knapsacks spices list is updated to contain all the current spices and the capacity of the knapsack is parsed from the file and given to the knapsack object.

5.2 mostValubleBag()

```
public float mostValubleBag() {
    int currentSpice = 0;
    float netWorth = 0;
    Spice[] unitPriceHighLow = sortByUnitPrice();
    int remainingCapacity = this.capacity;

    while (remainingCapacity != 0) {

        if (unitPriceHighLow[currentSpice].quantity <= remainingCapacity) {
            netWorth += unitPriceHighLow[currentSpice].totalPrice;
            remainingCapacity -= unitPriceHighLow[currentSpice].quantity;
        } else {
            netWorth += (remainingCapacity * unitPriceHighLow[currentSpice].unitPrice);
            remainingCapacity -= remainingCapacity;
        }

        currentSpice++;

        if (currentSpice >= unitPriceHighLow.length) remainingCapacity = 0;
    }

    return netWorth;
}
```

The above method gets finds the highest price a knapsack can carry given its capacity. To aid in this, it sorts the list of spices by their unit price using a selection sort algorithm (not shown to conserve space). Then from there it repeats the following steps until the remainingCapacity of the bag is 0. First, it check to see if all of the spice with the highest unit price can fit into the knapsack. If it can, add the total price of the current spice to the netWorth of the bag and decrease the remainingCapacity by the current spices quanitity. If the all of one spice can't fit, it then multiplies the current spices unitPrice by the remaining capacity and subtracts the remaining capacity from itself. After those conditionals are checked, the currentSpice counter, which holds which spice the program is currently looking at is incremented. If, however, currentSpice exceeds the actual number of spices there are, the remainingCapacity is set to 0. Once out of the while loop, the netWorth of the back is returned.

This implementation of the fractional knapsack problem runs in linear time as in the worst case, it has to check every single spice. This is denoted by the while loop. Although it is said to run in linear time. There are many occurrences where it won't have to check every spice.

5.3 Presentation of Fractional Knapsack

```
public static void knapsackPresentation() {
    float profit = knapsack.mostValubleBag();
    int currentSpice = 0;
    int remainingCapacity = knapsack.capacity;
    int amount = 0;
    System.out.println("A knapsack of size " + knapsack.capacity + " is worth $" + profit + "
    while(profit != 0) {
        if (profit >= knapsack.spices[currentSpice].totalPrice) {
            System.out.println("  " + knapsack.spices[currentSpice].quantity + " scoops of " +
            remainingCapacity -= knapsack.spices[currentSpice].quantity;
            //System.out.println("tempCapacity: " + tempCapacity);
            profit -= knapsack.spices[currentSpice].totalPrice;
            //System.out.println("profit: " + profit);
        } else {
            System.out.println("  " + remainingCapacity + " scoops of " + knapsack.spices[curr
            remainingCapacity -= remainingCapacity;
            profit -= profit;
        }
        currentSpice++;
    }
    if(remainingCapacity != 0) {
        System.out.println("  " + remainingCapacity + " unfilled spaces");
    }
    System.out.println();
}
```

The above method prints out the information produced by mostValubleBag() in a readable way. First it calls mostValubleBag and stores the netWorth returned as the variable profit. Then in order to print out the continents of the bag, a sort of reversed knapsack algorithm is used. It takes the profit of the bag and uses it to determine how many of each spice are included. If a spice's totalPrice is less than the profit, then the spice is printed out with its total quantity. The spices total price is subtracted from the profit and the next spice is then checked and the remaningCapacity is reduced by the spice's quantity. If a spice's totalPrice is greater than the profit, then the spice is printed out along with the remainingCapacity as its quantity in the bag. The profit and remainingCapacity are both reduced by values equal to themselves. In the case where all spices have been added to the bag but there is still room left in the bag (remainingCapacity != 0) then another line is printed out stating how much room is leftover in the bag.