Assignment Five

Shannon Brady

shannon.brady2@Marist.edu

December 7, 2022

1 Assignment 5

1.1 Directed and Weighted Graphs

1. Line 15-26: Count the number of graphs in the input file and initialize an array of graphs.
2. Line 29-70: Initialize all vertices and edges for each graph in the file. item Line 73-78: Complete the Bellman-Ford Algorithm to find the Single Shortest Path (SSSP) for each graph.
3. Running Time Analysis: The asymptotic running time of SSSP is O(V*E), where V is the number of vertices within the graph and E is the number of edges. This is due to the fact that all edges will be traversed and relaxed for each vertex.

1.2 Fractional Knapsack

1. Line 98-115: Count the number of spices and knapsacks in the input file. Initialize and array of spices and an array of knapsacks.
2. Line 124-161: Create spice and knapsack objects and add them to their respective arrays.
3. : Line 170-171: Sort the spices by unit price (high to low).
4. Line 173-183: For each knapsack, add scoops of spices until the knapsack reaches capacity or the spice has no more scoops remaining.
5. Running Time Analysis: Given that the spices were sorted using Insertion Sort, the running time of this implementation of fractional knapsack (for each knapsack) is roughly $O(n^2+n)$, where $O(n^2)$ represents the complexity of Insertion sort and O(n) represents the time complexity of traversing the spice array. Since the highest degree is most relevant when considering asymptotic running time, the overall time complexity is $O(n^2)$.

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
```

```java
class Assignment_5 {

    public static void main(String[] args) {

        // 1) Directed and Weighted Graphs ------------------------------------------------------------

        // Get the graph file and process graph data
        File file = new File("graphs.txt");
        int numGraphs = 0;

        try {
            // First, count the number of graphs in the file
            Scanner myReader = new Scanner(file);
            while (myReader.hasNextLine()) {
                if (myReader.nextLine().contains("new")) {
                    numGraphs++;
                }
            }
            // Initialize an array of graphs
            Graph[] graphs = new Graph[numGraphs];
            System.out.println("Number of graphs in file: " + numGraphs);
            myReader.close();

            // Next, process the graph data for each graph
            myReader = new Scanner(file);
            int i=0;
            while (i < numGraphs) {
                if (myReader.nextLine().contains("new")) {
                    String line = myReader.nextLine();
                    String verticesStr = "";
                    while (line.contains("vertex")) {
                        // Get the vertex id (vid)
                        String vid = line.substring(line.lastIndexOf(" ") + 1);
                        // Keep a string of vid data
                        verticesStr += vid + " ";
                        line = myReader.nextLine();
                    }

                    // Create the graph using an array of vid's
                    String[] verticeArr = verticesStr.split(" ");
                    graphs[i] = new Graph (verticeArr);
                    for (int j=0; j<verticeArr.length; j++) {
                        graphs[i].addVertex(verticeArr[j]);
                    }

                    // Create all the edges
                    while (line.contains("edge")) {
                        // The line substring with edge data
                        String edgeStr = line.substring(9);

                        // The edgeStr substring with vertice data
                        String verticeStr = edgeStr.substring(0, edgeStr.lastIndexOf(" "));
                        String weight = edgeStr.substring(edgeStr.lastIndexOf(" ") + 1);

```

```
                    // Get each vid connecting the edge and create the edge
                    String[] vertices = verticeStr.split(" - ");
                    String vertex1 = vertices[0].trim();
                    String vertex2 = vertices[1].trim();
                    graphs[i].createEdge(vertex1, vertex2, Integer.parseInt(weight));

                    if (myReader.hasNextLine()) {
                        line = myReader.nextLine();
                    } else {
                        break;
                    }
                }
                System.out.println("-----------------------------------------------------------------------------------");
                System.out.println("Graph " + i + ":");
                boolean isValid = graphs[i].bellmanFord();
                if (isValid) {
                    graphs[i].getResults();
                } else {
                    System.out.print("not valid :(");
                }
                i++;
                System.out.println("-----------------------------------------------------------------------------------");
            }
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
    // END Directed and Weighted Graphs ------------------------------------------------------------------------------




    // 2) Greedy Algorithms - Spice Heist ----------------------------------------------------------------------------
    file = new File("spice.txt");
    int numSpices = 0;
    int numKnapsacks = 0;

    try {
        Scanner myReader = new Scanner(file);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            if (data.startsWith("spice")) {
                numSpices++;
            } else if (data.startsWith("knapsack")) {
                numKnapsacks++;
            }
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
```

```
113
114         Spice[] spices = new Spice[numSpices];
115         Knapsack[] knapsacks = new Knapsack[numKnapsacks];
116
117         try {
118             Scanner myReader = new Scanner(file);
119             int i = 0;
120             while(myReader.hasNextLine()) {
121                 String data = myReader.nextLine();
122
123                 // initialize an array of spices
124                 if (data.startsWith("spice")) {
125                     while (i < numSpices) {
126                         // 1. get substring after 'spice'
127                         // 2. remove whitespace
128                         // 3. split at ';'
129                         String[] spiceData = data.substring(6).replaceAll("\\s", "").split(";");
130                         if (spiceData.length == 3) { // color, total price, quantity
131                             String color = spiceData[0].split("=")[1];
132                             String totalPrice = spiceData[1].split("=")[1];
133                             String quantity = spiceData[2].split("=")[1];
134                             Spice spice = new Spice(color, Float.parseFloat(totalPrice), Integer.parseInt(quantity));
135                             spices[i] = spice;
136                         }
137                         data = myReader.nextLine();
138                         i++;
139                     }
140                   // initialize an array of knapsacks
141                 } else if (data.startsWith("knapsack")) {
142                     i = 0;
143                     while (i<numKnapsacks) {
144                         // 1. get substring after 'knapsack'
145                         // 2. remove whitespace
146                         // 3. split at ';'
147                         String[] knapsackData = data.substring(9).replaceAll("\\s", "").split(";");
148                         if (knapsackData.length == 1) { // capacity
149                             String capacity = knapsackData[0].split("=")[1];
150                             Knapsack knapsack = new Knapsack(Integer.parseInt(capacity));
151                             knapsacks[i] = knapsack;
152                         }
153                         // check there is another line, otherwise break
154                         if (myReader.hasNextLine()) {
155                             data = myReader.nextLine();
156                         } else {
157                             break;
158                         }
159                         i++;
160                     }
161                 }
162             }
163             myReader.close();
164         } catch (FileNotFoundException e) {
165             System.out.println("An error occurred.");
166             e.printStackTrace();
```

```
167         }
168
169         // sort the spices by unit price, high to low
170         InsertionSort insertionSortObj = new InsertionSort();
171         insertionSortObj.sort(spices);
172
173         for (int i=0; i<knapsacks.length; i++) {
174             for (int j=0; j<spices.length; j++) {
175                 int remaining = spices[j].getQuantity();
176                 // add a given spice until the knapsack is full or a spice has no more scoops remaining
177                 while (!knapsacks[i].isFull() && remaining != 0) {
178                     knapsacks[i].fill(spices[j]);
179                     remaining--;
180                 }
181             }
182             knapsacks[i].print();
183         }
184         // END Spice Heist ------------------------------------------------
185     }
186 }
```

2 Directed and Weighted Graphs

1. The Graph class is used to implement directed and weight graphs and has the following attributes:
   a) vertices: A linked list of all vertices within the graph
   b) edges: An ArrayList of Edges used to represent all edges within the graph
2. Line 13-15: Method that adds a vertex to the vertices list
3. Line 14-26: Method that creates a graph edge and adds it to the list of edges
4. Line 29-46: Method that implements the Bellman-Ford Algorithm to find the Single Source Shortest Path (SSSP).
   a) Line 30-34: For each vertex, relax all of the edges within the graph.
   b) Line 36-44: Check that there are no negative weight cycles within the graph.
5. Line 48-59: Method that is used to update an edge distance if a shorter path exists.
   a) Line 54-57: Set the new distance and predecessor vertex. Apply these changes to all occurrences of this given vertex throughout all graph edges.
   b) Line 62-81: Methods used to apply all changes to a given vertex to all occurrences of said vertex throughout the graph edges.
   c) Line 84-127: Method used to print the final SSSP.
      i. Line 85-99: Find a single occurrence of each graph vertex and add to the results ArrayList. These are the Vertex objects that hold the final SSSP results.
      ii. Line 102-116: Loop through the resulting vertex objects. If the current vertex is not the single source, build the path from the given vertex to the single source by traversing through the previous vertices.
      iii. Line 117-126: Print the path in the correct order.

```java
import java.util.ArrayList;

class Graph {

    private VertexLinkedList vertices = null;
    private ArrayList<Edge> edges = null;

    Graph(String[] verticeArr) {
        this.vertices = new VertexLinkedList();
        this.edges = new ArrayList<Edge>();
    }

    public void addVertex(String vid) {
        this.vertices.add(vid);
    }

    // Creates an edge between two vertices
    public void createEdge(String vid1, String vid2, int weight) {
        Vertex vertex1 = this.vertices.getVertexByID(vid1);
        Vertex vertex2 = this.vertices.getVertexByID(vid2);
        vertex1.getNeighbors().add(vid2);
        vertex2.getNeighbors().add(vid1);

        Edge edge = new Edge(vertex1, vertex2, weight);
        this.edges.add(edge);
    }

    // Bellman-Ford Algorithm, returns boolean indicating if valid
    public boolean bellmanFord() {
        for (int k=0; k<this.vertices.getSize()-1; k++) {
            for (int i=0; i<this.edges.size(); i++) {
                this.relax(this.edges.get(i));
            }
        }

        for (int i=0; i<this.edges.size(); i++) {
            Vertex v1 = this.edges.get(i).getVertices().getVertexAt(0);
            Vertex v2 = this.edges.get(i).getVertices().getVertexAt(1);
            int weight = this.edges.get(i).getWeight();

            if (v2.getDistance() > v1.getDistance() + weight) {
                return false;
            }
        }
        return true;
    }

    public void relax(Edge edge) {
        Vertex v1 = edge.getVertices().getVertexAt(0);
        Vertex v2 = edge.getVertices().getVertexAt(1);
        int weight = edge.getWeight();

        if (v1.getDistance() != Integer.MAX_VALUE && v2.getDistance() > v1.getDistance() + weight) {
```

```
54              v2.setDistance(v1.getDistance() + weight);
55              this.updateDistance(v2, v1.getDistance() + weight);
56              v2.setPrev(v1);
57              this.updatePrev(v2, v1);
58          }
59      }
60
61      // Ensures that distances are consistent throughout all occurances of a given vertex
62      public void updateDistance(Vertex v, int distance) {
63          for (int i=0; i<this.edges.size(); i++) {
64              for (int j=0; j<this.edges.get(i).getVertices().getSize(); j++) {
65                  if (this.edges.get(i).getVertices().getVertexAt(j).getID().compareTo(v.getID()) == 0) {
66                      this.edges.get(i).getVertices().getVertexAt(j).setDistance(distance);
67                  }
68              }
69          }
70      }
71
72      // Ensures that prev vertices are consistent throughout all occurances of a given vertex
73      public void updatePrev(Vertex v, Vertex prev) {
74          for (int i=0; i<this.edges.size(); i++) {
75              for (int j=0; j<this.edges.get(i).getVertices().getSize(); j++) {
76                  if (this.edges.get(i).getVertices().getVertexAt(j).getID().compareTo(v.getID()) == 0) {
77                      this.edges.get(i).getVertices().getVertexAt(j).setPrev(prev);
78                  }
79              }
80          }
81      }
82
83      // Prints the final results
84      public void getResults() {
85          ArrayList<Vertex> results = new ArrayList<Vertex>();
86          for (int k=0; k<this.vertices.getSize(); k++) {
87              edgeSearch:
88              for (int i=0; i<this.edges.size(); i++) {
89                  for (int j=0; j<this.edges.get(i).getVertices().getSize(); j++) {
90                      // Find the first occurance of an edge vertex that exists in the vertices list
91                      // Add that vertex to a results list for printing later
92                      if (this.edges.get(i).getVertices().getVertexAt(j).getID().compareTo(this.vertices.getVertexAt(k).getID()) =
93                          Vertex vertex = this.edges.get(i).getVertices().getVertexAt(j);
94                          results.add(vertex);
95                          break edgeSearch;
96                      }
97                  }
98              }
99          }
100
101         for (int i=0; i<results.size(); i++) {
102             ArrayList<Vertex> path = new ArrayList<Vertex>();
103             if (!results.get(i).isSrc()) {
104                 Vertex start = results.get(i);
105                 path.add(start);
106
107                 System.out.print("1 --> " + start.getID() + "; ");
```

```
                System.out.print("cost is " + start.getDistance() + "; ");
                System.out.print("path is ");

                Vertex prev = results.get(i).getPrev();
                while (prev != null) {
                    path.add(prev);
                    prev = prev.getPrev();
                }
            }
            for (int j=path.size()-1; j>=0; j--) {
                if (j != 0) {
                    System.out.print(path.get(j).getID() + " --> ");
                } else {
                    System.out.print(path.get(j).getID());
                }
            }
            System.out.println();
            System.out.println();
        }
    }

    public void print() {
        for (int i=0; i<this.edges.size(); i++) {
            this.edges.get(i).print();
        }
    }
}
```

3 Edge

1. The Edge class is used to represent an edge within a graph. Each instance has the following attributes:
   a) vertices: A linked list of vertices
   b) weight: An int representing the weight value associated with each edge
2. Line 16-26: Initialize all distances for each non-single-source vertex to a maximum value (representing infinity). Single source vertices get initialized to 0.

```
class Edge {

    private VertexLinkedList vertices = null;
    private int weight = 0;

    public Edge(Vertex vertex1, Vertex vertex2, int _weight) {
        this.vertices = new VertexLinkedList();
        this.vertices.add(vertex1.getID());
        this.vertices.add(vertex2.getID());
        this.weight = _weight;
        this.distanceInit();
    }

    // Initalize all non-source distances to max int
```

```
15      // Source diatnace get intialized to 0
16      public void distanceInit() {
17          for (int i=0; i<this.vertices.getSize(); i++) {
18              if (this.vertices.getVertexAt(i).getID().compareTo("1") == 0) {
19                  this.vertices.getVertexAt(i).setDistance(0);
20                  this.vertices.getVertexAt(i).setSrc(true);
21
22              } else {
23                  this.vertices.getVertexAt(i).setDistance(Integer.MAX_VALUE);
24              }
25          }
26      }
27
28      public void print() {
29          for (int i=0; i<this.vertices.getSize(); i++) {
30              this.vertices.getVertexAt(i).print();
31              System.out.println();
32          }
33          System.out.println();
34          System.out.println("weight: " + weight);
35          System.out.println();
36          System.out.println("-----------------");
37          System.out.println();
38      }
39
40      public VertexLinkedList getVertices() {
41          return this.vertices;
42      }
43
44      public int getWeight() {
45          return this.weight;
46      }
47  }
```

4 Knapsack

1. A Knapsack has the following attributes:
   a) capacity: An integer value representing the capacity of the knapsack
   b) contents: A linked list a spices
   c) value: A float value representing the total value of all contents within the knapsack
2. Line 23-49: Method that counts the number of each spice within the knapsack.

```
1   import java.text.DecimalFormat;
2
3   class Knapsack {
4
5       private static final DecimalFormat df = new DecimalFormat("0.00");
6
7       private int capacity = 0;
```

```
8       private LinkedList contents = null;
9       private float value = 0;
10
11      public Knapsack(int _capacity) {
12          this.capacity = _capacity;
13          this.contents = new LinkedList();
14          this.value = 0;
15      }
16
17      public void print() {
18          System.out.println("Knapsack of capacity " + this.capacity +
19                             " is worth " + df.format(this.value) + " Simoleons" +
20                             " and contains\n" + this.getContents());
21      }
22
23      public String getContents() {
24          int orangeCnt = 0;
25          int blueCnt = 0;
26          int greeCnt = 0;
27          int redCnt = 0;
28          for (int i=0; i<this.contents.getSize(); i++) {
29              switch (this.contents.getAt(i).getName()) {
30                  case "orange":
31                  orangeCnt++;
32                      break;
33                  case "blue":
34                      blueCnt++;
35                      break;
36                  case "green":
37                      greeCnt++;
38                      break;
39                  case "red":
40                      redCnt++;
41                      break;
42              }
43          }
44          String contents = (orangeCnt != 0 ? orangeCnt + " scoops of orange\n" : "") +
45                             (blueCnt != 0 ? blueCnt + " scoops of blue\n" : "") +
46                             (greeCnt != 0 ? greeCnt + " scoops of green\n" : "") +
47                             (redCnt != 0 ? redCnt + " scoops of red\n" : "");
48          return contents;
49      }
50
51      public boolean isFull() {
52          return this.capacity == this.contents.getSize();
53      }
54
55      public void fill(Spice spice) {
56          this.contents.add(spice.getColor());
57          this.updateValue(spice);
58      }
59
60      public void updateValue(Spice spice) {
61          this.value += spice.getUnitPrice();
```

```
    }

    public float getValue() {
        return this.value;
    }
}
```

5 Spice

1. A Spice has the following attributes:
    a) color: the color of the spice
    b) totalPrice: The total price associated with the entire spice quantity
    c) quantity: The total number of scoops allocated to the spice
    d) unitPrice: The price of each scoop

```
class Spice {

    private String color = null;
    private float totalPrice = 0;
    private int quantity = 0;
    private float unitPrice = 0;

    public Spice(String _color, float _totalPrice, int _quantity) {
        this.color = _color;
        this.totalPrice = _totalPrice;
        this.quantity = _quantity;
        this.unitPrice = _totalPrice / _quantity;
    }

    public void print() {
        System.out.println("color: " + this.color +
                           "\ntotal price: " + this.totalPrice +
                           "\nquantity: " + this.quantity +
                           "\nunit price: " + this.unitPrice + "\n");
    }

    public String getColor() {
        return this.color;
    }

    public float getTotalPrice() {
        return this.totalPrice;
    }

    public int getQuantity() {
        return this.quantity;
    }

    public float getUnitPrice() {
```

```
35          return this.unitPrice;
36      }
37  }
```