CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment Five – Dynamic & Greedy Algorithms

C. Marcus DiMarco
C.DiMarco1@Marist.edu

December 10, 2021

1 Objectives

- Create a program that reads a text file in order to create directed, weighted graphs of various numbers of vertices and edges.
- Print the shortest paths possible from a single source to each vertex in the graph along with the total cost of traversal.
- Create a program that reads a text file in order to create an instance of the fractional knapsack problem using spices of varying value with weight 1.
- Print the maximum values that can be held in each knapsack.

2 Conditions

- The program must create the relevant Objects by parsing the commands and inputs of the file.
- The program must distinguish based on the format of the file when an Object has been fully constructed.

3 Overview - Graphs

Our graphing is driven by three classes: MarcusVertex, MarcusEdge and MarcusGraphs.

3.1 MarcusVertex

This custom vertex class is the building block of the graphs. Neighboring vertices are now contained within an ArrayList of edges. Additional changes from the previous iteration include support for tracking cost and a pointer to the previous MarcusVertex for calculating and tracking the shortest path.

```
1  /**
2   * A custom implementation of a vertex object for
3   * representing graphs as linked objects.
4   */
5  import java.util.ArrayList;
6
7  public class MarcusVertex {
8    private int id;
9    private boolean isProcessed;
10   private ArrayList<MarcusEdge> edges;
11   private MarcusVertex next;
12   private int cost;
13   private MarcusVertex shortestSource;
14
15
16   public MarcusVertex(int id) {
17     this.id = id;
18     this.isProcessed = false;
19     this.edges = new ArrayList<MarcusEdge>();
20     this.next = null;
21     this.cost = 0;
22     this.shortestSource = null;
23   }
24
25   public boolean hasNeighbor(MarcusVertex neighbor) {
26     for (int i = 0; i < edges.size(); i++) {
27       if (edges.get(i).getDestination().getId() == neighbor.getId()) {
28         return true;
29       }
30     }
31
32     return false;
33   }
34
35   public int weightToVertex(MarcusVertex neighbor) {
```

```
36      for (int i = 0; i < edges.size(); i++) {
37        if (edges.get(i).getDestination().getId() == neighbor.getId()) {
38          return edges.get(i).getWeight();
39        }
40      }
41
42      System.out.println("No matching edge found.");
43      return -2112;
44    }
45
46    // Setters and getters for private fields
47    public int getId() {
48      return this.id;
49    }
50
51    public void setId(int id) {
52      this.id = id;
53    }
54
55    public boolean getIsProcessed() {
56      return isProcessed;
57    }
58
59    public void setIsProcessed(boolean isProcessed) {
60      this.isProcessed = isProcessed;
61    }
62
63    public void addEdge(MarcusEdge edge) {
64      if (edge.getSource().id != this.id) {
65        System.out.println("This is not the edge you're looking for...");
66        System.out.println("(Source of edge does not match this vertex)");
67        return;
68      }
69      this.edges.add(edge);
70    }
71
72    public ArrayList<MarcusEdge> getEdges() {
73      return this.edges;
74    }
75
76    public void printNeighbors() {
77      for (MarcusEdge currentEdge : edges) {
78        System.out.print(currentEdge.getDestination().getId() + " ");
79      }
80      System.out.print("\n");
81    }
```

```
82
83    public void setNext(MarcusVertex next) {
84      this.next = next;
85    }
86
87    public MarcusVertex getNext() {
88      return this.next;
89    }
90
91    public void setCost(int cost) {
92      this.cost = cost;
93    }
94
95    public int getCost() {
96      return cost;
97    }
98
99    public void setShortestSource(MarcusVertex source) {
100     this.shortestSource = source;
101   }
102
103   public MarcusVertex getShortestSource() {
104     return shortestSource;
105   }
106 }
```

3.2 MarcusEdge

MarcusEdge is the newest addition to the MarcusGraphing suite. Each MarcusEdge tracks its source, destination and weight.

```
1  /**
2   * A custom representation of edges between vertices to allow for
3   * the adding of weights to a directed graph.
4   */
5
6  public class MarcusEdge {
7    private MarcusVertex sourceVertex;
8    private MarcusVertex destinationVertex;
9    private int weight;
10
11   // Default constructor only allows for assignment of weighted edges
12   public MarcusEdge(MarcusVertex source, MarcusVertex destination, int weight) {
13     this.sourceVertex = source;
14     this.destinationVertex = destination;
15     this.weight = weight;
```

```
16    }
17
18    // Setters and getters for private fields
19    public void setSource(MarcusVertex source) {
20      this.sourceVertex = source;
21    }
22
23    public MarcusVertex getSource() {
24      return this.sourceVertex;
25    }
26
27    public void setDestination(MarcusVertex destination) {
28      this.destinationVertex = destination;
29    }
30
31    public MarcusVertex getDestination() {
32      return this.destinationVertex;
33    }
34
35    public void setWeight(int weight) {
36      this.weight = weight;
37    }
38
39    public int getWeight() {
40      return this.weight;
41    }
42  }
```

3.3 MarcusGraphs

Using MarcusVertex, we can assemble MarcusGraphs - a class which defines the various print and traversal methods needed. For user-friendliness, MarcusGraphs includes a method getVertexById(), which is instrumental in the creation of the edges from the text file. In addition, printMatrix() has been updated for compatibility with weighted graphs for better testing. This is also the container for our singleSourceShortestPath() method, which calculates the shortest path from a source to each vertex by setting each vertex's cost to VERY_HIGH_NUMBER and reducing it as it finds a more efficient path.

```
1  /**
2   * A custom implementation of graphs as an object containing
3   * vertices and edges. Supports matrices, adjacency lists,
4   * and linked objects, as well as both depth-first traversals
5   * and breadth-first traversals.
6   */
```

```
7  import java.util.ArrayList;
8
9  public class MarcusGraphs {
10   private ArrayList<MarcusVertex> vertices;
11   private MarcusVertex initialVertex;
12   private ArrayList<MarcusEdge> edges;
13   private boolean hasBeenPrinted;
14
15   // Default constructor
16   public MarcusGraphs() {
17     this.vertices = new ArrayList<MarcusVertex>();
18     this.initialVertex = null;
19     this.edges = new ArrayList<MarcusEdge>();
20     this.hasBeenPrinted = false;
21   }
22
23   // Prints a matrix of all vertices, printing a 1 at the intersection
24   // if there is an edge present and printing a . if not
25   public void printMatrix() {
26     for (int i = -1; i < vertices.size(); i++) {
27       for (int j = -1; j < vertices.size(); j++) {
28         if (i == -1 && j == -1) {
29           // Top left corner is blank space
30           System.out.printf("%3s", "");
31         } else if (i == -1) {
32           // Top row is vertex IDs
33           System.out.printf("%3s", vertices.get(j).getId() + "␣");
34         } else if (j == -1) {
35           // First column is vertex IDs
36           System.out.printf("%3s", vertices.get(i).getId() + "␣");
37         } else if (vertices.get(i).hasNeighbor(vertices.get(j))) {
38           // If the vertices are neighbors, print weight
39           System.out.printf("%3s",
40                 vertices.get(i).weightToVertex(vertices.get(j)));
41         } else {
42           // If not neighbors, print .
43           System.out.printf("%3s", ".␣");
44         }
45       }
46       // New line
47       System.out.print("\n\n");
48     }
49   }
50
51   // Prints each vertex followed by its neighbors
52   public void printAdjacencyList() {
```

```
53      for (int i = 0; i < vertices.size(); i++) {
54        System.out.print("[" + vertices.get(i).getId() + "] ");
55        vertices.get(i).printNeighbors();
56      }
57      System.out.print("\n");
58    }
59
60    // Traverses a graph vertex-by-vertex, going as deep as possible from
61    // the source before moving on to the next vertex. Prints IDs as
62    // encountered.
63    public void depthFirstTraversal(MarcusVertex source) {
64
65      if (!source.getIsProcessed()) {
66        System.out.print(source.getId() + " ");
67        source.setIsProcessed(true);
68      }
69      for (MarcusEdge currentEdge : source.getEdges()) {
70        if (!currentEdge.getDestination().getIsProcessed()) {
71          depthFirstTraversal(currentEdge.getDestination());
72        }
73      }
74    }
75
76    // Traverses a graph using a queue. Prints IDs as dequeued.
77    public void breadthFirstTraversal(MarcusVertex source) {
78      MarcusVertex currentVertex;
79
80      // Reset booleans from depth-first traversal
81      this.resetBooleans();
82
83      // Enqueue when encountered
84      MarcusQueue queue = new MarcusQueue();
85      queue.enqueue(source);
86      source.setIsProcessed(true);
87      while (!queue.isEmpty()) {
88        currentVertex = queue.dequeue();
89        System.out.print(currentVertex.getId() + " ");
90        for (MarcusEdge each : currentVertex.getEdges()) {
91          if (!each.getDestination().getIsProcessed()) {
92            queue.enqueue(each.getDestination());
93            each.getDestination().setIsProcessed(true);
94          }
95        }
96      }
97
98      System.out.print("\n\n");
```

```
99    }
100
101   // Find the shortest path between two vertices using Bellman-Ford
102   public void singleSourceShortestPath(MarcusVertex source) {
103     initSSSP(source);
104     for (int i = 0; i < this.vertices.size(); i++) {
105       for (MarcusEdge currentEdge : this.edges) {
106         this.relax(currentEdge.getSource(), currentEdge.getDestination(),
107               currentEdge.getWeight());
108       }
109     }
110     if (noNegativeLoops()) {
111       System.out.println("SSSP complete!");
112     } else {
113       System.out.println("SSSP failed - negative loop present");
114     }
115   }
116
117   // Initializes the SSSP algorithm, setting costs to max int value,
118   // clearing paths and setting source cost to 0
119   private void initSSSP(MarcusVertex source) {
120     final int VERY_HIGH_NUMBER = (int) Integer.MAX_VALUE - 6000;
121     for (MarcusVertex vertex : this.vertices) {
122       vertex.setCost(VERY_HIGH_NUMBER);
123       vertex.setShortestSource(null);
124     }
125     source.setCost(0);
126   }
127
128   // Checks if the cost of moving from first to second is lower than
129   // the recorded cost of second
130   private void relax(MarcusVertex first, MarcusVertex second, int weight) {
131     if (second.getCost() > first.getCost() + weight) {
132       second.setCost(first.getCost() + weight);
133       second.setShortestSource(first);
134     }
135   }
136
137   // Private test for negative loops to ensure possible success for SSSP
138   private boolean noNegativeLoops() {
139     for (MarcusEdge current : this.edges) {
140       if (current.getDestination().getCost() >
141         current.getSource().getCost() + current.getWeight()) {
142           return false;
143         }
144     }
```

```
145       return true;
146     }
147
148     // Private method for printing the shortest path from the source
149     // using MarcusStack
150     private void printPathFromSource(MarcusVertex vertex) {
151       MarcusStack stack = new MarcusStack();
152
153       while (vertex != null) {
154         stack.push(vertex);
155         vertex = vertex.getShortestSource();
156       }
157
158       while(!stack.isEmpty()) {
159         System.out.print(stack.pop().getId());
160         if (!stack.isEmpty()) {
161           System.out.print(" --> ");
162         }
163       }
164
165       System.out.print("\n");
166     }
167
168     // Reset isProcessed for each vertex in the graph
169     public void resetBooleans() {
170       for (MarcusVertex currentVertex : vertices) {
171         currentVertex.setIsProcessed(false);
172       }
173     }
174
175     // Add vertex to ArrayList and set initialVertex if needed
176     public void addVertex(MarcusVertex vertex) {
177       this.vertices.add(vertex);
178       if (this.initialVertex == null) {
179         this.initialVertex = vertex;
180       }
181     }
182
183     public ArrayList<MarcusVertex> getVertices() {
184       return this.vertices;
185     }
186
187     // Add edge to ArrayList
188     public void addEdge(MarcusEdge edge) {
189       this.edges.add(edge);
190     }
```

```
191
192   public MarcusVertex getVertexById(int vertexId) {
193     MarcusVertex returnVertex = null;
194
195     for (MarcusVertex currentVertex : vertices) {
196       if (currentVertex.getId() == vertexId) {
197         returnVertex = currentVertex;
198         break;
199       }
200     }
201
202     return returnVertex;
203   }
204
205   public MarcusVertex getInitialVertex() {
206     return initialVertex;
207   }
208
209   public void printSSSP(MarcusVertex source) {
210     for (MarcusVertex current : this.vertices) {
211       if (current.equals(source)) {
212         continue;
213       } else {
214         System.out.print(source.getId() + " ---> " + current.getId() +
215                 " cost is " + current.getCost() +
216                 "; path: ");
217         this.printPathFromSource(current);
218       }
219     }
220
221     this.hasBeenPrinted = true;
222   }
223
224   public boolean hasBeenPrinted() {
225     return this.hasBeenPrinted;
226   }
227 }
```

4 Overview - Fractional Knapsack

Our fractional knapsack approach consists of two classes - MarcusSpice and MarcusKnapsack.

4.1 MarcusSpice

MarcusSpice is a simple container for the Spice objects from the text file. Its key methods are isAvailable(), resetQuantity(), and putInKnapsack(), all of which are instrumental in properly tracking availability while retaining original quantity data for the next knapsack.

```
1  /**
2   * A container for a spice involved in a spice heist. From Arrakis in
3   * origin, in my greedy knapsack for destination.
4   */
5  public class MarcusSpice {
6    private String name;
7    private double price;
8    private int quantity;
9    private int quantityLeft;
10
11   public MarcusSpice() {
12     this.name = null;
13     this.price = 0;
14     this.quantity = 0;
15     this.quantityLeft = 0;
16   }
17
18   // Constructor based on totalPrice as input
19   public MarcusSpice(String name, double totalPrice, int quantity) {
20     this.name = name;
21     this.price = totalPrice / quantity;
22     this.quantity = quantity;
23     this.quantityLeft = quantity;
24   }
25
26   // Setters and getters for private fields
27   public void setName(String name) {
28     this.name = name;
29   }
30
31   public String getName() {
32     return name;
33   }
34
35   public void setPrice(double price) {
```

```
36      this.price = price;
37    }
38
39    public double getPrice() {
40      return this.price;
41    }
42
43    public void setQuantity(int quantity) {
44      this.quantity = quantity;
45    }
46
47    public int getQuantity() {
48      return quantity;
49    }
50
51    public boolean isAvailable() {
52      if (quantityLeft != 0) {
53        return true;
54      } else {
55        return false;
56      }
57    }
58
59    public void resetQuantity() {
60      this.quantityLeft = this.quantity;
61    }
62
63    public void putInKnapsack() {
64      if (this.quantityLeft != 0){
65        this.quantityLeft--;
66      } else {
67        System.out.println("Oops! 0 remaining.");
68      }
69    }
70  }
```

4.2 MarcusKnapsack

MarcusKnapsack is a bag full of wonder. It features the fractionalKnapsack() method, which is where all of the magic happens. It resets each spice's quantity to the original level, sorts the spice array by increasing value, then descends down the spice array until it is either full or has exhausted the spices. It features a HashMap to track how many of each spice are held within the knapsack.

```java
/**
 * A custom class to determine the greediest load of spices
 * that can be contained within a fractional knapsack of
 * varying capacities.
 */
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class MarcusKnapsack {
  private int capacity;
  private int value;
  private int spicesHeld;
  private HashMap<String, Integer> spiceInventory;

  public MarcusKnapsack(int capacity) {
    this.capacity = capacity;
    this.value = 0;
    this.spicesHeld = 0;
    spiceInventory = new HashMap<String, Integer>();
  }

  public void fractionalKnapsack(ArrayList<MarcusSpice> spices) {

    // Reset spice quantities available
    for (MarcusSpice spice : spices) {
      spice.resetQuantity();
    }

    // Ensure acting on spice array sorted by price
    sortSpices(spices);

    for (int i = spices.size() - 1; i >= 0; i--) {
      int counter = 0;
      while (spices.get(i).isAvailable() && this.hasRoom()) {
        spices.get(i).putInKnapsack();
        this.addValue(spices.get(i));
        this.spicesHeld++;
        counter++;
        spiceInventory.put(spices.get(i).getName(), counter);
      }
    }

    System.out.print("Knapsack of capacity " + this.capacity + " is worth " +
              this.value + " quatloos and contains ");

```

```
47      boolean hasLooped = false;
48
49      for (Map.Entry<String, Integer> spice : spiceInventory.entrySet()) {
50        if (hasLooped) {
51          System.out.print(", ");
52        }
53        System.out.print(spice.getValue() + " scoop");
54        if (spice.getValue() != 1) {
55          System.out.print("s");
56        }
57        System.out.print(" of " + spice.getKey());
58        hasLooped = true;
59      }
60      System.out.println(".");
61  }
62
63    public void sortSpices(ArrayList<MarcusSpice> spices) {
64      for (int i = 1; i < spices.size(); i++) {
65        MarcusSpice keySpice = spices.get(i);
66        for (int j = i - 1; j >= 0; j--) {
67          // Move all items larger than key forward 1 index
68          if (keySpice.getPrice() < spices.get(j).getPrice()) {
69            spices.set(j + 1, spices.get(j));
70            // If index 0 reached, assign it currentString
71            if (j == 0) {
72              spices.set(j, keySpice);
73            }
74          } else {
75            // Insert key at index ahead of first smaller element
76            spices.set(j + 1, keySpice);
77            break;
78          }
79        }
80      }
81    }
82
83    // Setter and getter for capacity
84    public void setCapacity(int capacity) {
85      this.capacity = capacity;
86    }
87
88    public int getCapacity() {
89      return capacity;
90    }
91
92    public void setValue(int value) {
```

```
93      this.value = value;
94    }
95
96    public int getValue() {
97      return this.value;
98    }
99
100   public void addValue(MarcusSpice spice) {
101     this.value += spice.getPrice();
102   }
103
104   public boolean hasRoom() {
105     if (this.spicesHeld >= this.capacity) {
106       return false;
107     }
108
109     return true;
110   }
111 }
```

5 Assignment5

With the classes we have just defined, we are ready to create our program. We'll need some imported libraries: namely, java.io.File to import a file, java.io.FileNotFoundException to account for errors finding the input file, java.util.Scanner to read the file, and java.util.ArrayList to contain the spices in the second half.

```
1  /**
2   * A program designed to implement the Bellman-Ford dynamic
3   * programming algorithm for Single Source Shortest Path
4   * on directed graphs and to implement a greedy solution to
5   * fractional knapsack.
6   */
7  import java.io.File;
8  import java.io.FileNotFoundException;
9  import java.util.Scanner;
10 import java.util.ArrayList;
11
12 public class Assignment5 {
13   public static void main(String[] args) {
14     // Read graphs2.txt and create matrix, adjacency list, and linked objects
```

```
15      try {
16        File graphs = new File("graphs2.txt");
17        Scanner graphRead = new Scanner(graphs);
18        MarcusGraphs graph = null;
19        String command = null;
20        String item = null;
21        while (graphRead.hasNextLine()) {
22          command = graphRead.next();
23          if (command.equals("--")) {
24          // Skip comment if null or if graph has been printed
25            if (graph == null || graph.hasBeenPrinted()) {
26              graphRead.nextLine();
27            } else {
28              // Run SSSP
29              graph.singleSourceShortestPath(graph.getInitialVertex());
30              graph.printSSSP(graph.getInitialVertex());
31            }
32          } else if (command.equals("new")) {
33          // Create new graph
34            graph = new MarcusGraphs();
35            graphRead.nextLine();
36          } else if (command.equals("add")) {
37            item = graphRead.next();
38            if (item.equals("vertex")) {
39            // Add new vertex to graph
40              MarcusVertex vertex = new MarcusVertex(graphRead.nextInt());
41              graph.addVertex(vertex);
42            } else if (item.equals("edge")) {
43            // Add new edge to graph
44              int a = graphRead.nextInt();
45              graphRead.next();
46              int b = graphRead.nextInt();
47              int edgeWeight = graphRead.nextInt();
48              MarcusVertex first = graph.getVertexById(a);
49              MarcusVertex second = graph.getVertexById(b);
50              MarcusEdge edge = new MarcusEdge(first, second, edgeWeight);
51              first.addEdge(edge);
52              graph.addEdge(edge);
53            }
54          }
55        }
56        graph.singleSourceShortestPath(graph.getInitialVertex());
57        graph.printSSSP(graph.getInitialVertex());
58        graphRead.close();
59      } catch (FileNotFoundException e) {
60        System.out.println("Whoops! Couldn't find graphs2.txt");
```

```
61          e.printStackTrace();
62        }
63
64        try {
65          File spices = new File("spice.txt");
66          Scanner spiceRead = new Scanner(spices);
67          String spiceCommand = null;
68          String spiceItem = null;
69          ArrayList<MarcusSpice> spiceArray = new ArrayList<MarcusSpice>();
70          while (spiceRead.hasNextLine()) {
71            spiceCommand = spiceRead.next();
72            if (spiceCommand.equals("--")) {
73            // Skip comment if null
74              spiceRead.nextLine();
75            } else if (spiceCommand.equals("spice")) {
76            // Create new spice
77              MarcusSpice spice = new MarcusSpice();
78              spiceItem = spiceRead.next();
79              if (spiceItem.equals("name")) {
80                // Add name to spice
81                spiceRead.next();
82                String name = spiceRead.next();
83                spice.setName(name.substring(0, name.length() - 1));
84                spiceItem = spiceRead.next();
85              }
86              if (spiceItem.equals("total_price")) {
87                // Add price and quantity to spice
88                spiceRead.next();
89                String price = spiceRead.next();
90                double totalPrice = Double.parseDouble(price.substring(0,
91                                    price.length() - 1));
92                spiceRead.next();
93                spiceRead.next();
94                String quantity = spiceRead.next();
95                spice.setQuantity(Integer.parseInt(quantity.substring(0,
96                                  quantity.length() - 1)));
97                spice.setPrice(totalPrice / spice.getQuantity());
98                spiceArray.add(spice);
99              }
100           } else if (spiceCommand.equals("knapsack")) {
101             spiceRead.next();
102             spiceRead.next();
103             String capacityString = spiceRead.next();
104             int capacity = Integer.parseInt(capacityString.substring(0,
105                                capacityString.length() - 1));
106             MarcusKnapsack bag = new MarcusKnapsack(capacity);
```

```
107             bag.fractionalKnapsack(spiceArray);
108           }
109         }
110         spiceRead.close();
111       } catch (FileNotFoundException e) {
112         System.out.println("Whoops! Couldn't find spice.txt");
113         e.printStackTrace();
114       }
115     }
116 }
```

6 Results & Analysis

The asymptotic running time of Bellman-Ford's Single Source Shortest Path algorithm is similar to $O(n^2)$, but could be better described as $O(|V * E|)$, where V is the set of all vertices in the graph and E is the set of all edges in the graph. This distinction is important as we need to track the change in the running time due to two inputs instead of just one.

The asymptotic running time of our greedy approach to fractional knapsack is $O(n)$. This is due to the linear nature of iterating through the array of spices until we have filled the knapsack.