Assignment 5 - Code Evaluation

Mason Nakamura

December 2021

1 LinkedObjects Class

```
1  import java.lang.reflect.Array;
2  import java.util.ArrayList;
3
4  public class LinkedObjects {
5      ArrayList<Integer> num_vertices = new ArrayList<>();
6      public void linked_objects(String[] lines) {
7          // use these indices throughout for loops
8          int index_start;
9          int index_end = 0;
10         // for printing purposes
11         int graph_id = 0;
12         // for vertex starting index
13         boolean indexIs0 = false;
14         // Read the lines up to add edge case
15         while (index_end <= lines.length) {
16             graph_id += 1;
17             index_start = index_end;
18             int adj_list_length = 0;
19             // get indices for for loop for edges
20             for (int i = index_start; i < lines.length; i++) {
21                 String line = lines[i];
22                 index_start += 1;
23                 // case for line is blank;
24                 if (line.isBlank()) {
25                     // skip the line
26                     continue;
27                 }
28                 // case for line is a comment
29                 if (line.charAt(1) == '-' & line.charAt(0) == '-') {
30                     // skip the line
31                     continue;
32                 }
33                 // make an array of the words from the line
34                 String[] words = line.split(" ");
35                 // case for declaring new graph
36                 if (words[0].equals("new")) {
37                     // print graph_id
38                     System.out.println("Graph Number: " + graph_id);
39                     //skip the line
40                     continue;
41                 }
42                 // case for adding vertex
43                 if (words[0].equals("add") & words[1].equals("vertex")) {
44                     adj_list_length += 1;
45                     if (words[2].equals("0")) {
46                         indexIs0 = true;
47                     }
48                     continue;
49                 }
50                 //check if hit edge case
51                 if (words[0].equals("add") & words[1].equals("edge")) {
52                     // subtract an index since we previously added it but didn't need
                           to since we entered the edge case
53                     index_start -= 1;
54                     break;
55                 }
56
57             }
58
```

```
59              // create instance of array given number of vertices declared
60              index_end = index_start + 1;
61              //make array of vertex objects
62              Vertex[] array = new Vertex[adj_list_length];
63              // check starting index
64
65              // create first vertex objects in array given index
66              if (indexIs0) {
67                  for (int i = 0; i < adj_list_length; i++) {
68                      // from Vertex class
69                      Vertex vertex = new Vertex();
70                      vertex.connecting_vertex = i;
71                      vertex.label = i;
72                      vertex.next = null;
73                      vertex.neighbors = new ArrayList<Vertex>();
74                      array[i] = vertex;
75                  }
76
77              } else {// shift index by 1
78                  for (int i = 1; i < adj_list_length + 1; i++) {
79                      Vertex vertex = new Vertex();
80                      vertex.connecting_vertex = i;
81                      vertex.label = i;
82                      vertex.next = null;
83                      vertex.neighbors = new ArrayList<Vertex>();
84                      array[i - 1] = vertex;
85                  }
86              }
87
88              // use index_start to continue in the for loop
89              for (int i = index_start; i < index_end; i++) {
90                  // case if reached to end of file/(array of lines)
91                  if (lines.length < index_end) {
92                      break;
93                  }
94                  String line = lines[i];
95                  String[] words = line.split(" ");
96                  // check if line is blank; if so, move to next graph
97                  if (line.isBlank()) {
98                      break;
99                  }
100
101                 // check if there are two spaces between the weight and the
                        connecting vertex
102                 // if so, move up an index
103                 if (words[5].equals("")){
104                     words[5] = words[6];
105                 }
106
107                 // case for adding edge
108                 if (words[0].equals("add") & words[1].equals("edge")) {
109                     index_end += 1;
110                     // you need to define vertex1 and vertex2 since not doing so
                            creates a pointer infinitely pointing to itself
111                     if (indexIs0) {// if index starts at 0 don't subtract 1
112                         Vertex vertex1 = new Vertex();
113                         vertex1.origin_vertex = Integer.parseInt(words[2]);
114                         vertex1.connecting_vertex = Integer.parseInt(words[4]);
115                         vertex1.weight = Integer.parseInt(words[5]);
116                         vertex1.label = vertex1.connecting_vertex;
117                         Vertex head1 = array[vertex1.origin_vertex - 1];
118                         while (head1.next != null) {
119                             head1 = head1.next;
120                         }
121                         head1.next = vertex1;
122
123                     } else {// subtract 1 to keep indices same
124                         Vertex vertex1 = new Vertex();
125                         vertex1.origin_vertex = Integer.parseInt(words[2]);
126                         vertex1.connecting_vertex = Integer.parseInt(words[4]);
127                         vertex1.weight = Integer.parseInt(words[5]);
128                         vertex1.label = vertex1.connecting_vertex;
129                         Vertex head1 = array[vertex1.origin_vertex - 1];
130                         while (head1.next != null) {
131                             head1 = head1.next;
132                         }
```

```
133                         head1.next = vertex1;
134                     }
135                 }
136             }
137 
138 
139             Vertex[] copy_vertexes = array.clone();
140             for(Vertex i: copy_vertexes){
141                 System.out.print("[" + i.label+ "]" + " ");
142                 while (i.next != null){
143                     i = i.next;
144                     System.out.print(i.connecting_vertex + "(" + i.weight + ")" + "
                        ");
145                 }
146                 System.out.println();
147             }
148 
149         }
150     }
151 
152     // to insert row into 2-D array in
153     public int[][] insertRow(int[][] m, int r, int[] data) {
154         int[][] out = new int[m.length + 1][];
155         for (int i = 0; i < r; i++) {
156             out[i] = m[i];
157         }
158         out[r] = data;
159         for (int i = r + 1; i < out.length; i++) {
160             out[i] = m[i - 1];
161         }
162         return out;
163     }
164 
165     public ArrayList<int[][]> matrices(String[] lines) {
166         // use these indices throughout for loops
167         int index_start;
168         int index_end = 0;
169         // for vertex starting index
170         boolean indexIs0 = false;
171         //initialize an arraylist of 2-arrays (matrices) so that we can pull the
                individual graphs in the testing file
172         ArrayList<int[][]> matrices = new ArrayList<>();
173         while (index_end <= lines.length) {
174             index_start = index_end;
175             // don't need rows and columns since adjacency matrix is symmetric ==>
                    rows = columns
176             int vertices = 0;
177             // get indices for for loop for edges
178             for (int i = index_start; i < lines.length; i++) {
179                 String line = lines[i];
180                 index_start += 1;
181                 // case for line is blank;
182                 if (line.isBlank()) {
183                     // skip the line
184                     continue;
185                 }
186                 // case for line is a comment
187                 if (line.charAt(1) == '-' & line.charAt(0) == '-') {
188                     // skip the line
189                     continue;
190                 }
191                 // make an array of the words from the line
192                 String[] words = line.split(" ");
193                 // case for declaring new graph
194                 if (words[0].equals("new")) {
195                     //skip the line
196                     continue;
197                 }
198                 // case for adding vertex
199                 if (words[0].equals("add") & words[1].equals("vertex")) {
200                     vertices += 1;
201                     if (words[2].equals("0")) {
202                         indexIs0 = true;
203                     }
204                     continue;
205                 }
```

```
206                 //check if hit edge case
207                 if (words[0].equals("add") & words[1].equals("edge")) {
208                     index_start -= 1;
209                     break;
210                 }
211 
212             }
213             // add number of vertices to list
214             num_vertices.add(vertices);
215             // create instance of matrix
216             index_end = index_start + 1;
217             int[][] matrix = new int[0][3];
218             // use index_start to continue in the for loop
219             for (int i = index_start; i < index_end; i++) {
220                 // case if reached to end of file/(array of lines)
221                 if (lines.length < index_end) {
222                     break;
223                 }
224                 String line = lines[i];
225                 String[] words = line.split(" ");
226                 // check if line is blank; if so, move to next graph
227                 if (line.isBlank()) {
228                     break;
229                 }
230 
231                 // check if there are two spaces between the weight and the
                        connecting vertex
232                 // if so, move up an index
233                 if (words[5].equals("")){
234                     words[5] = words[6];
235                 }
236 
237                 // case for adding edge
238                 if (words[0].equals("add") & words[1].equals("edge")) {
239                     index_end += 1;
240                     // use Integer.parseInt to convert string to int
241                     if (indexIs0) {// if index starts at 0 don't subtract 1
242                         // make array of data for added edge
243                         int[] newData = new int[]{Integer.parseInt(words[2]) + 1,
                                Integer.parseInt(words[4]) + 1, Integer.parseInt(words
                                [5])};
244                         // append the array to the matrix
245                         matrix = insertRow(matrix, matrix.length, newData);
246 
247                     } else {// subtract 1 to keep indices same
248                         // make array of data for added edge
249                         int[] newData = new int[]{Integer.parseInt(words[2]), Integer
                                .parseInt(words[4]), Integer.parseInt(words[5])};
250                         // append the array to the matrix
251                         matrix = insertRow(matrix, matrix.length, newData);
252                     }
253                 }
254             }
255 
256 //              for (int i = 0; i < matrix.length; i++) {
257 //                  for (int j = 0; j < matrix[i].length; j++) {
258 //                      System.out.print(matrix[i][j] + " ");
259 //                  }
260 //                  System.out.println();
261 //              }
262 //              System.out.println();
263             matrices.add(matrix);
264         }
265         return matrices;
266     }
267 }
```

In this class I used what I did in assignment 4, except I adapted it to directed graphs (fairly easy). I have two methods. The first being the linked_objects method which demonstrates that I constructed a linked object representation for each graph, outputting an adjacency list for visual purposes. The next methods called matrices() on line 165 was used as input for my SSSP algorithm where it gave the algorithm matrices to find the shortest path.

2 SSSP Class

```
1  import java.util.ArrayList;
2
3  public class SSSP {
4      static void bellman_ford(int[][] graph, int V, int E, int src) {
5          // Initialize distance of all vertices as very big value.
6          int[] dis = new int[V];
7          for (int i = 0; i < V; i++)
8              dis[i] = Integer.MAX_VALUE;
9
10         // initialize distance of source as 0
11         dis[src-1] = 0;
12
13         // RELAX
14         for (int i = 0; i < V - 1; i++) {
15             for (int j = 0; j < E; j++) {
16                 // subtract 1 if !indexis0
17                 if (dis[graph[j][0]-1] != Integer.MAX_VALUE && dis[graph[j][0]-1] +
                       graph[j][2] < dis[graph[j][1]-1]) {
18                     dis[graph[j][1]-1] = dis[graph[j][0]-1] + graph[j][2];
19                 }
20             }
21         }
22         // check if there is an infinite cycle
23         for (int i = 0; i < E; i++) {
24             int x = graph[i][0];
25             int y = graph[i][1];
26             int weight = graph[i][2];
27             if (dis[x-1] != Integer.MAX_VALUE && dis[x-1] + weight < dis[y-1]) {
28                 System.out.println("Graph contains negative weight cycle");
29             }
30         }
31
32         System.out.println("Vertex \t\t Distance from Source");
33         for (int i = 0; i < V; i++)
34             System.out.println(i+1 + "\t\t\t " + dis[i]);
35     }
36 }
```

The SSSP Class implements the BellmanFord Algorithm to find the shortest path on a directed weighted graph. In this case, we used adjacency matrices to find the shortest paths. We first initialize all the distances for each node in the graph as infinity (MAX_VALUE) at line 7. Then we apply a relax function on all the nodes in the graph and every edge in the graph checking for smallest distance relative to all other combinations of nodes and edges. Then on line 23, we check if the algorithm gets stuck in an infinite cycle. The complexity is $O(|V||E|)$ since we go through every edge on every vertex (the nested for loop in line 14).

3 Heist Class

```
1  import java.lang.reflect.Array;
2  import java.util.ArrayList;
3  import java.util.Arrays;
4
5  public class Heist implements Cloneable{
6      public static void spice_heist(String[] lines) {
7          // use these indices throughout for loops
8          final int NUMBER_OF_SPICES = 4;
9          final int NUMBER_OF_KNAPSACKS = 5;
10         int index_start = 0;
11         int index_end = 0;
12         int spice_index = 0;
13         boolean onKnapsack = false;
14         Spice[] spices = new Spice[NUMBER_OF_SPICES];
15         // use unit_prices to sort the spices array
16         float[] unit_prices = new float[NUMBER_OF_SPICES];
17         ArrayList<Integer> knapsack_capacities = new ArrayList<>();
18 //        while (index_end <= lines.length) {
19 //            index_start = index_end;
20         int last_index = 0;
21         for (int i = index_start; i < lines.length; i++) {
22             String line = lines[i];
23             index_start += 1;
24             // case for line is blank;
25             if (line.isBlank()) {
26                 // skip the line
27                 continue;
28             }
29             // case for line is a comment
30             if (line.charAt(0) == '-' & line.charAt(1) == '-') {
31                 // skip the line
32                 continue;
33             }
34             // make an array of the words from the line separated by ";"
35             String[] words = line.split(";");
36             // attributes for Spice object
37             String[] spice_attributes = new String[3];
38             for (int j = 0; j < words.length; j++) {
39                 words[j] = words[j].replaceAll("\\s", "");
40                 // separate the words in each element of words by "="
41                 String[] subwords = words[j].split("=");
42                 if (subwords[0].equals("knapsackcapacity")) {
43                     onKnapsack = true;
44                     last_index = index_start;
45                     break;
46                 }
47                 if (!onKnapsack) {
48                     // populate the spice attributes
49                     spice_attributes[j] = subwords[1];
50                 }
51             }
52             if (!onKnapsack) {
53                 Spice new_spice = new Spice();
54                 new_spice.color = spice_attributes[0];
55                 new_spice.total_price = Float.parseFloat(spice_attributes[1]);
56                 new_spice.quantity = Integer.parseInt(spice_attributes[2]);
57                 //append the unit_price to array
58                 unit_prices[spice_index] = new_spice.total_price / new_spice.quantity
                       ;
59                 new_spice.unit_price = unit_prices[spice_index];
60                 //append the Spice object in an array
61                 spices[spice_index] = new_spice;
62                 spice_index += 1;
63             }else{
64                 break;
65             }
66         }
67
68         for (int i = last_index-1; i < lines.length; i++){
69             String line = lines[i];
70             String[] words = line.split(";");
71             // attributes for Spice object
72 //            String[] spice_attributes = new String[3];
```

```
73              for (int j = 0; j < words.length; j++) {
74                  words[j] = words[j].replaceAll("\\s", "");
75                  // separate the words in each element of words by "="
76                  String[] subwords = words[j].split("=");
77                  if (subwords[0].equals("knapsackcapacity")) {
78                      knapsack_capacities.add(Integer.parseInt(subwords[1]));
79                  }
80                  else{
81                      System.out.println("ERROR in Heist line 82");
82                  }
83              }
84          }
85          RelativeInsertionSort_Decreasing RIS = new RelativeInsertionSort_Decreasing()
                ;
86          // relatively sort unit_prices and spices
87          RIS.relative_insertionSort_decreasing(unit_prices, spices);
88          // start filling each knapsack and print results
89          int out_index = 0;
90          for (int i=0; i < knapsack_capacities.size(); i++){
91              onKnapsack = false;
92              index_start = 0;
93              spice_index = 0;
94              Spice[] spices2 = new Spice[NUMBER_OF_SPICES];
95              // use unit_prices to sort the spices array
96              unit_prices = new float[NUMBER_OF_SPICES];
97              for (int k = index_start; k < lines.length; k++) {
98                  String line = lines[k];
99                  index_start += 1;
100                 // case for line is blank;
101                 if (line.isBlank()) {
102                     // skip the line
103                     continue;
104                 }
105                 // case for line is a comment
106                 if (line.charAt(0) == '-' & line.charAt(1) == '-') {
107                     // skip the line
108                     continue;
109                 }
110                 // make an array of the words from the line separated by ";"
111                 String[] words = line.split(";");
112                 // attributes for Spice object
113                 String[] spice_attributes = new String[3];
114                 for (int j = 0; j < words.length; j++) {
115                     words[j] = words[j].replaceAll("\\s", "");
116                     // separate the words in each element of words by "="
117                     String[] subwords = words[j].split("=");
118                     if (subwords[0].equals("knapsackcapacity")) {
119                         onKnapsack = true;
120                         last_index = index_start;
121                         break;
122                     }
123                     if (!onKnapsack) {
124                         // populate the spice attributes
125                         spice_attributes[j] = subwords[1];
126                     }
127                 }
128                 if (!onKnapsack) {
129                     Spice new_spice = new Spice();
130                     new_spice.color = spice_attributes[0];
131                     new_spice.total_price = Float.parseFloat(spice_attributes[1]);
132                     new_spice.quantity = Integer.parseInt(spice_attributes[2]);
133                     //append the unit_price to array
134                     unit_prices[spice_index] = new_spice.total_price / new_spice.
                            quantity;
135                     new_spice.unit_price = unit_prices[spice_index];
136                     //append the Spice object in an array
137                     spices2[spice_index] = new_spice;
138                     spice_index += 1;
139                 }else{
140                     break;
141                 }
142             }
143
144
145             Spice[] copy_spices = spices2.clone();
146
```

```
147             RelativeInsertionSort_Decreasing RIS2 = new
                    RelativeInsertionSort_Decreasing();
148             // relatively sort unit_prices and spices
149             RIS2.relative_insertionSort_decreasing(unit_prices, copy_spices);
150
151
152             int sack_size = knapsack_capacities.get(i);
153             // how much the knapsack is worth
154             float worth = 0;
155             // how many differing color spices were used
156             int red = 0, green = 0, blue = 0, orange = 0;
157             int index = 0;
158             int j = 0;
159             while (sack_size != 0 && copy_spices[copy_spices.length-1].quantity != 0)
                    {
160                 while (copy_spices[j].quantity != 0){
161 //                  for (int j = 0; j < copy_spices.length; j++){
162                       index += 1;
163 //                    if(copy_spices[j].quantity != 0){
164                         sack_size -= 1;
165                         worth += copy_spices[j].unit_price;
166                         copy_spices[j].quantity -= 1;
167 //                        System.out.println(copy_spices[1].quantity);
168                         switch (copy_spices[j].color) {
169                             case "red" -> red += 1;
170                             case "green" -> green += 1;
171                             case "blue" -> blue += 1;
172                             case "orange" -> orange += 1;
173                         }
174                         if (sack_size == 0){
175                             break;
176                         }
177 //                    }
178 //                    if(index == 3){
179 //                       break;
180 //                    }
181                 }
182                 j+=1;
183
184             }
185             int index1 = 0;
186             int[] amounts = new int[]{red, green, blue, orange};
187             String[] colors = new String[]{"red", "green", "blue", "orange"};
188             String amount = "";
189             for (int p=0; p< amounts.length; p++){
190                 int times = 0;
191                 while (amounts[p] != 0){
192                     amounts[p] -= 1;
193                     times += 1;
194                 }
195                 amount += " " + times + " scoops of " + colors[index1] + ",";
196                 index1 += 1;
197             }
198
199             System.out.println("Knapsack of capacity " + knapsack_capacities.get(i) +
                        " is worth " + worth + " and " +
200                         "contains " + amount);
201             out_index += 1;
202         }
203     }
204
205
206 }
```

The Heist class has the spice_heist() method used to solve the Knapsack problem. We first parse the spice file from line 21 to line 84 where we populated several arrays to help in our outputs later on. On line 85, we implemented a relative sorting algorithm to sort two arrays the same order. In this case, we sorted unit_prices and spices in decreasing order. Next we go through each knapsack in line 90 using the same parsing method from lines 21-84 since I was unable to deep clone the array of Spice objects and there associated attributes. On line 159, I then applied a greedy algorithm. The complexity is $O(nlogn)$ since it takes $logn$ for each item and there is a total of n items. Additionally, the $nlogn$ derives from the greedy choice part of the algorithm which is in the while loop.

4 RelativeInsertionSort Class

```
1  public class RelativeInsertionSort_Decreasing {
2      public void relative_insertionSort_decreasing(float[] A, Spice[] B){
3          int n = A.length;
4          for(int i = 1; i < n; i++){
5              float key = A[i];
6              Spice keyB = B[i];
7              int j = i-1;
8  //            comparisons += 1;
9              // if A[j] does not need to be switched, skip while
10             while(j >= 0 && A[j] < key){
11                 A[j+1] = A[j];
12                 B[j+1] = B[j];
13                 j = j-1;
14             }
15             A[j+1] = key;
16             B[j+1] = keyB;
17         }
18     }
19 }
```

Used in SSSP Class

5 Spice Class

```
1  public class Spice {
2      String color = null;
3      float total_price = -1;
4      int quantity = -1;
5      float unit_price = -1;
6  }
```

Used in Heist Class

6 Testing Class

```
1
2      import java.io.*;
3  import java.util.Arrays;
4
5  public class Test {
6
7
8
9      public static void main(String[] args) {
10         String[] lines = {};
11         // Read line by line the txt file using File reader
12         String fileName = "graphs2";  //REMEMBER TO NOT HARDCODE
13         File file = new File(fileName);
14         try {
15             FileReader fr = new FileReader(file);
16             BufferedReader br = new BufferedReader(fr);
17             String line;
18             while ((line = br.readLine()) != null) {
19                 // add strings from txt file line by line into array words
20                 lines = Arrays.copyOf(lines, lines.length + 1); //extends memory
21                 lines[lines.length - 1] = line; //adds line to extra memory
22             }
23         } catch (FileNotFoundException e) {
24             System.out.println("An error occurred.");
25             e.printStackTrace();
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29
30         String[] copy_graphs0 = lines.clone();
31
32         // 1)
33         LinkedObjects out = new LinkedObjects();
34         // outputs matrices
```

```
35          System.out.println("GRAPH AS LINKED OBJECTS TESTING: ");
36          out.linked_objects(copy_graphs0);
37          int j = 0;
38          for (int[][] i: out.matrices(copy_graphs0)){
39              // i.length = |E|, out.num_vertices.get(j) = |V|, i = matrix, 1 = node
                    source
40              SSSP.bellman_ford(i, out.num_vertices.get(j), i.length, 1);
41              j+=1;
42          }
43
44          // 2) Knapsacks
45          String[] lines2 = {};
46          // Read line by line the txt file using File reader
47          String fileName2 = "spice";  //REMEMBER TO NOT HARDCODE
48          File file2 = new File(fileName2);
49          try {
50              FileReader fr = new FileReader(file2);
51              BufferedReader br = new BufferedReader(fr);
52              String line;
53              while ((line = br.readLine()) != null) {
54                  // add strings from txt file line by line into array words
55                  lines2 = Arrays.copyOf(lines2, lines2.length + 1); //extends memory
56                  lines2[lines2.length - 1] = line; //adds line to extra memory
57              }
58          } catch (FileNotFoundException e) {
59              System.out.println("An error occurred.");
60              e.printStackTrace();
61          } catch (IOException e) {
62              e.printStackTrace();
63          }
64
65          String[] spice = lines2.clone();
66          Heist.spice_heist(spice);
67      }
68  }
```

Where I read the .txt files and did the testing.