

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import java.util.HashMap;
6 import java.util.Iterator;
7 import java.util.LinkedList;
8 import java.util.NoSuchElementException;
9 import java.util.Scanner;
10
11 /**
12  *
13  * Created: CS Team
14  * Modified: Adrianna Valle
15  * Date: 12-04-16
16  *
17  * AdjMatGraph has been optimized to include and consider weighted edges.
18  * Everything else was kept
19  * constant. An AdjMatGraph was used instead of an AdjacencyList in order to allow
20  * for both
21  * expansion capabilities as well as easy accessibility to succeeding
22  * vertices/weights.
23  */
24 public class AdjMatGraph<T> implements Graph<T>, Iterable<T> {
25     public static final int NOT_FOUND = -1;
26     private static final int DEFAULT_CAPACITY = 1; // Small so that we can test
27     expand private static final boolean VERBOSE = false; // print while reading TGF?
28
29     private int n; // number of vertices in the graph
30     private Integer[][] arcs; // adjacency matrix of arcs
31     private T[] vertices; // values of vertices
32
33     /*****
34     ***** Constructor. Creates an empty graph.
35     *****
36     @SuppressWarnings("unchecked")
37     public AdjMatGraph() {
38         n = 0;
39         this.arcs = new Integer[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
40         this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
41     }
42
43     /***** NEW METHODS *****/
44     /**
45     * Construct a copy (clone) of a given graph.
46     * The new graph will have all the same vertices and arcs as the original.
47     * A *shallow* copy is performed: the graph structure is copied, but
48     * the new graph will refer to the exact same vertex objects as the original.
49     */
50     @SuppressWarnings("unchecked")
51     public AdjMatGraph(AdjMatGraph<T> g) {
52         n = g.n;
53         vertices = (T[]) new Object[g.vertices.length];
54         arcs = new Integer[g.arcs.length][g.arcs.length];
55         for (int i = 0; i < n; i++) {
56             vertices[i] = g.vertices[i];
57             for (int j = 0; j < n; j++) {
58                 arcs[i][j] = g.arcs[i][j];
59             }
60         }
61     }
62
63     /*****
64     ***** Load vertices and edges from a TGF file into a given graph.
65     *****
66     * @param tgfFile - name of the TGF file to read
67     * @param g - graph to which vertices and arcs will be added.
68     * g must be empty to start!
69     * @throws FileNotFoundException
70     *****
71     public static void loadTGF(String tgf_file_name, AdjMatGraph<String> g) throws
72     FileNotFoundException {
73         if (!g.isEmpty()) throw new RuntimeException("Refusing to load TGF data into
74         non-empty graph.");
75         Scanner fileReader = new Scanner(new File(tgf_file_name));
76         // Keep a mapping from TGF vertex ID to AdjMatGraph vertex ID.
77         // This allows vertex IDs to be written out of order in TGF.
78         // It also supports non-integer vertex IDs.
79         HashMap<String,Integer> vidMap = new HashMap<String,Integer>();
80         try {
81             // Read vertices until #
82             while (fileReader.hasNext()) {
83                 // Get TGF vertex ID
84                 String nextToken = fileReader.next();
85                 if (nextToken.equals("#")) {
86                     break;
87                 }
88                 vidMap.put(nextToken, g.n());
89                 String label = fileReader.hasNextLine() ? fileReader.nextLine().trim() :
90                 fileReader.next();
91                 if (VERBOSE) {
92                     System.out.println("Adding vertex " + g.n() + " (" + nextToken + " = \""
93                     + label + "\"");
94                 }
95                 g.n++;
96             }
97         } catch (FileNotFoundException e) {
98             e.printStackTrace();
99         }
100     }
101
102     /Users/sl60540/Desktop/FINALPROJECT_lluo_jaguiar_avalle/FinalProject/AdjMatGraph.java 1

```

```

87         g.addVertex(label);
88     }
89
90     // Read edges until EOF
91     while (fileReader.hasNext()) {
92         // Get src and dest
93         String src = fileReader.next();
94         String dest = fileReader.next();
95         // Discard label if any
96         int label = -1;
97         if (fileReader.hasNextLine()) {
98             try {
99                 label = Integer.parseInt(fileReader.nextLine().trim());
100             } catch (NumberFormatException ex) {
101                 label = 1;
102             }
103         }
104         g.addArc(vidMap.get(src), vidMap.get(dest), label);
105     }
106 } catch (RuntimeException e) {
107     System.out.println("Error reading TGF");
108     throw e;
109 } finally {
110     fileReader.close();
111 }
112 }
113 }
114
115 /**
116  * An iterator that iterates over the vertices of an AdjMatGraph.
117  */
118 private class VerticesIterator implements Iterator<T> {
119     private int cursor = 0;
120
121     /** Check if the iterator has a next vertex */
122     public boolean hasNext() {
123         return cursor < n;
124     }
125
126     /** Get the next vertex. */
127     public T next() {
128         if (cursor >= n) {
129             throw new NoSuchElementException();
130         } else {
131             return vertices[cursor++];
132         }
133     }
134
135     /** Remove is not supported in this iterator. */
136     public void remove() {
137         throw new UnsupportedOperationException();
138     }
139 }
140
141 /**
142  * Create a new iterator that will iterate over the vertices of the array when
143  * asked.
144  * @return the new iterator.
145  */
146 public Iterator<T> iterator() {
147     return new VerticesIterator();
148 }
149
150 /**
151  * Check if the graph contains the given vertex.
152  */
153 public boolean containsVertex(T vertex) {
154     return getIndex(vertex) != NOT_FOUND;
155 }
156
157
158 /**** FAMILIAR METHODS *****/
159
160
161 /****
162  Returns true if the graph is empty and false otherwise.
163  *****/
164 public boolean isEmpty() {
165     return n == 0;
166 }
167
168 /****
169  Returns the number of vertices in the graph.
170  *****/
171 public int n() {
172     return n;
173 }
174
175 /****
176  Returns the number of arcs in the graph by counting them.
177  *****/
178 public int m() {
179     int total = 0;

```

2

```

180     for (int i = 0; i < n; i++) {
181         for (int j = 0; j < n; j++) {
182             if (arcs[i][j] != null) {
183                 total++;
184             }
185         }
186     }
187     return total;
188 }
189
190 /**
191  * Returns array of all vertices.
192  */
193 public T[] getVertices() {
194     return vertices;
195 }
196
197 /**
198  * Returns true iff a directed edge exists from v1 to v2.
199  * *****/
200 public boolean isArc(T srcVertex, T destVertex) {
201     int src = getIndex(srcVertex);
202     int dest = getIndex(destVertex);
203     return src != NOT_FOUND && dest != NOT_FOUND && arcs[src][dest] != null;
204 }
205
206 /**
207  * Returns true iff an arc exists between two given indices.
208  * @throws IllegalArgumentException if either index is invalid.
209  * *****/
210 protected boolean isArc(int srcIndex, int destIndex) {
211     if (!indexIsValid(srcIndex) || !indexIsValid(destIndex)) {
212         throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
213 + ", " + destIndex);
214     }
215     return arcs[srcIndex][destIndex] != null;
216 }
217
218 /**
219  * Returns true iff an edge exists between two given vertices
220  * which means that two corresponding arcs exist in the graph.
221  * *****/
222 public boolean isEdge(T srcVertex, T destVertex) {
223     int src = getIndex(srcVertex);
224     int dest = getIndex(destVertex);
225     return src != NOT_FOUND && dest != NOT_FOUND && isArc(src, dest) && isArc(dest,
src);
226 }
227
228 /**
229  * Returns true IFF the graph is undirected, that is, for every
230  * pair of nodes i, j for which there is an arc, the opposite arc
231  * is also present in the graph.
232  * *****/
233 public boolean isUndirected() {
234     for (int i = 1; i < n(); i++) {
235         // optimize to avoid checking pairs twice.
236         for (int j = 0; j < i; j++) {
237             if (arcs[i][j] != null && arcs[j][i] == null) {
238                 return false;
239             }
240         }
241     }
242     return true;
243 }
244
245 /**
246  * Adds a vertex to the graph, expanding the capacity of the graph
247  * if necessary. If the vertex already exists, it does not add it again.
248  * *****/
249 public void addVertex (T vertex) {
250     if (getIndex(vertex) != NOT_FOUND) return;
251     if (n == vertices.length) {
252         expandCapacity();
253     }
254     vertices[n] = vertex;
255     // for (int i = 0; i <= n; i++) {
256     //     // if (arcs[n][i] || arcs[i][n]) throw new RuntimeException("Corrupted
AdjacencyMatrix");
257     //     arcs[n][i] = false;
258     //     arcs[i][n] = false;
259     // }
260     n++;
261 }
262
263 /**
264  * Helper. Creates new arrays to store the contents of the graph
265  * with twice the capacity.
266  * *****/
267 @SuppressWarnings("unchecked")
268 private void expandCapacity() {
269     T[] largerVertices = (T[]) (new Object[vertices.length*2]);
270     // Users/s160540/Desktop/FINALPROJECT_lluo_jaguilar_avalle/FinalProject/AdjMatGraph.java

```

```

271 Integer[][] largerAdjMatrix =
272     new Integer[vertices.length*2][vertices.length*2];
273
274 for (int i = 0; i < n; i++) {
275     for (int j = 0; j < n; j++) {
276         largerAdjMatrix[i][j] = arcs[i][j];
277     }
278     largerVertices[i] = vertices[i];
279 }
280
281 vertices = largerVertices;
282 arcs = largerAdjMatrix;
283 }
284
285 /*****
286  Removes a single vertex with the given value from the graph.
287  Uses equals() for testing equality.
288  *****/
289 public void removeVertex (T vertex) {
290     int index = getIndex(vertex);
291     if (index != NOT_FOUND) {
292         removeVertex(index);
293     }
294 }
295
296 /*****
297  Helper. Removes a vertex at the given index from the graph.
298  Note that this may affect the index values of other vertices.
299  @throws IllegalArgumentException if the index is invalid.
300  *****/
301 protected void removeVertex (int index) {
302     if (!indexIsValid(index)) {
303         throw new IllegalArgumentException("No such vertex index");
304     }
305     n--;
306
307     // Remove vertex.
308     for (int i = index; i < n; i++) {
309         vertices[i] = vertices[i+1];
310     }
311
312     // Move rows up.
313     for (int i = index; i < n; i++) {
314         for (int j = 0; j <= n; j++) {
315             arcs[i][j] = arcs[i+1][j];
316         }
317     }
318
319     // Move columns left
320     for (int i = index; i < n; i++) {
321         for (int j = 0; j < n; j++) {
322             arcs[j][i] = arcs[j][i+1];
323         }
324     }
325
326     // Erase last row and last column
327     for (int a = 0; a < n; a++) {
328         arcs[n][a] = null;
329         arcs[a][n] = null;
330     }
331 }
332
333 /*****
334  Inserts an edge between two vertices of the graph.
335  If one or both vertices do not exist, ignores the addition.
336  An edge cannot be pointed to itself therefore is omitted.
337  *****/
338 public void addEdge(T vertex1, T vertex2, int weight) {
339     int index1 = getIndex(vertex1);
340     int index2 = getIndex(vertex2);
341     if (index1 != NOT_FOUND && index2 != NOT_FOUND && index1 != index2) {
342         addArc(index1, index2, weight);
343         addArc(index2, index1, weight);
344     }
345 }
346
347 /*****
348  Inserts an arc from srcVertex to destVertex.
349  If the vertices exist, else does not change the graph. Method
350  doesn't allow the vertex to ave an edge to itself.
351  *****/
352 public void addArc(T srcVertex, T destVertex, int weight) {
353     int src = getIndex(srcVertex);
354     int dest = getIndex(destVertex);
355     if (src != NOT_FOUND && dest != NOT_FOUND && src != dest) {
356         addArc(src, dest, weight);
357     }
358 }
359
360 /*****
361  Helper. Inserts an edge between two vertices of the graph. Note, an arc
362  does not point to itself in the context of our game therefore the ability has
363  been omitted.
364  @throws IllegalArgumentException if either index is invalid.

```

```

365     *****/
366     protected void addArc(int srcIndex, int destIndex, int weight) {
367         if (!indexIsValid(srcIndex) || !indexIsValid(destIndex) && srcIndex == destIndex)
368         {
369             throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
370             + ", " + destIndex);
371         }
372         arcs[srcIndex][destIndex] = weight;
373     }
374     *****/
375     Removes an edge between two vertices of the graph.
376     If one or both vertices do not exist, ignores the removal.
377     *****/
378     public void removeEdge(T vertex1, T vertex2) {
379         int index1 = getIndex(vertex1);
380         int index2 = getIndex(vertex2);
381         if (index1 != NOT_FOUND && index2 != NOT_FOUND) {
382             removeArc(index1, index2);
383             removeArc(index2, index1);
384         }
385     }
386     *****/
387     Removes an arc from vertex src to vertex dest,
388     if the vertices exist, else does not change the graph.
389     *****/
390     public void removeArc(T srcVertex, T destVertex) {
391         int src = getIndex(srcVertex);
392         int dest = getIndex(destVertex);
393         if (src != NOT_FOUND && dest != NOT_FOUND) {
394             removeArc(src, dest);
395         }
396     }
397     *****/
398     Helper. Removes an arc from index v1 to index v2.
399     @throws IllegalArgumentException if either index is invalid.
400     *****/
401     protected void removeArc(int srcIndex, int destIndex) {
402         if (!indexIsValid(srcIndex) || !indexIsValid(destIndex)) {
403             throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
404             + ", " + destIndex);
405         }
406         arcs[srcIndex][destIndex] = null;
407     }
408     *****/
409     Returns the index value of the first occurrence of the vertex.
410     Returns NOT_FOUND if the key is not found.
411     *****/
412     protected int getIndex(T vertex) {
413         for (int i = 0; i < n; i++) {
414             if (vertices[i].equals(vertex)) {
415                 return i;
416             }
417         }
418         return NOT_FOUND;
419     }
420     *****/
421     * Returns the weight of the edge from the one vertex to another.
422     If no edge is present, returns -1.
423     @param: T vertex1, T vertex2 @return int
424     *****/
425     public int getWeight(T vertex1, T vertex2) {
426         int x = getIndex(vertex1);
427         int y = getIndex(vertex2);
428         if (x < 0 || y < 0)
429             return -1;
430         return arcs[x][y];
431     }
432     *****/
433     Returns the vertex object that is at a certain index
434     *****/
435     protected T getVertex(int v) {
436         if (!indexIsValid(v)) {
437             throw new IllegalArgumentException("No such vertex index: " + v);
438         }
439         return vertices[v];
440     }
441     *****/
442     Returns true if the given index is valid.
443     *****/
444     protected boolean indexIsValid(int index) {
445         return index < n && index >= 0;
446     }
447     *****/
448     Retrieve from a graph the vertices x pointing to vertex v (x->v)
449     *****/

```

```

456     and returns them onto a linked list
457     *****/
458     public LinkedList<T> getPredecessors(T vertex) {
459         LinkedList<T> neighbors = new LinkedList<T>();
460
461         int v = getIndex(vertex);
462
463         if (v == NOT_FOUND) return neighbors;
464         for (int i = 0; i < n; i++) {
465             if (arcs[i][v] != null) {
466                 neighbors.add(getVertex(i)); // if T then add i to linked list
467             }
468         }
469         return neighbors;
470     }
471
472     /*****
473     * Retrieve from a graph the vertices x following vertex v (v->x)
474     and returns them onto a linked list
475     *****/
476     public LinkedList<T> getSuccessors(T vertex) {
477         LinkedList<T> neighbors = new LinkedList<T>();
478
479         int v = getIndex(vertex);
480         if (v == NOT_FOUND) return neighbors;
481         for (int i = 0; i < n; i++) {
482             if (arcs[v][i] != null) {
483                 neighbors.add(getVertex(i)); // if T then add i to linked list
484             }
485         }
486         return neighbors;
487     }
488
489     /*****
490     * Returns a string representation of the graph.
491     *****/
492     public String toString() {
493         if (n == 0) {
494             return "Graph is empty";
495         }
496
497         String result = "";
498
499         //result += "\nArcs\n";
500         //result += "-----\n";
501         result += "\ni ";
502
503         for (int i = 0; i < n; i++) {
504             result += " " + getVertex(i);
505             if (i < 10) { " ";
506             }
507         }
508         result += "\n";
509
510         for (int i = 0; i < n; i++) {
511             result += " " + getVertex(i) + " ";
512
513             for (int j = 0; j < n; j++) {
514                 if (arcs[i][j] != null) {
515                     result += arcs[i][j] + " ";
516                 } else {
517                     result += "- "; //just empty space
518                 }
519             }
520             result += "\n";
521         }
522         return result;
523     }
524
525     /*****
526     * Saves the current graph into a .tgf file.
527     * If it cannot save the file, a message is printed.
528     *****/
529     public void saveTGF(String tgf_file_name) {
530         try {
531             PrintWriter writer = new PrintWriter(new File(tgf_file_name));
532
533             //prints vertices by iterating through array "vertices"
534             for (int i = 0; i < n(); i++) {
535                 if (vertices[i] == null) {
536                     break;
537                 } else {
538                     writer.print((i+1) + " " + vertices[i]);
539                     writer.println("");
540                 }
541             }
542             writer.print("#"); // Prepare to print the edges
543             writer.println("");
544
545             //prints arcs by iterating through 2D array
546             for (int i = 0; i < n(); i++) {
547                 for (int j = 0; j < n(); j++) {
548                     if (arcs[i][j] != null) {
549

```

/Users/s160540/Desktop/FINALPROJECT\_lluo\_jaguar\_avalle/FinalProject/AdjMatGraph.java 6

```

550         writer.print((i+1) + " " + (j+1) + " " + arcs[i][j]);
551         writer.println("");
552     }
553 }
554 }
555 writer.close();
556 } catch (IOException ex) {
557     System.out.println("*** (T)ERROR*** The file could not be written: " + ex);
558 }
559 }
560
561 //looping to itself is prohibited.
562 /** Testing Driver for AdjMatGraph. This will not help you test AdjMatGraphPlus.
*/
563 public static void main (String args[]) throws FileNotFoundException {
564     System.out.println("NORMAL OPERATIONS");
565     System.out.println("");
566     AdjMatGraph<String> G = new AdjMatGraph<String>();
567     System.out.println("New graph is empty (true): \t" + G);
568     System.out.println("Empty=> undirected (true): \t" + G.isUndirected());
569     System.out.println("Empty graph no vertices(0): \t" + G.n());
570     System.out.println("Empty graph no arcs (0): \t" + G.m());
571     G.addVertex("A"); G.addVertex("B"); G.addVertex("C");
572     G.addVertex("D"); G.addVertex("E"); G.addVertex("F");
573     System.out.println("After adding 6 vert. (6): \t" + G.n());
574     System.out.println("After adding no arcs (0): \t" + G.m());
575     System.out.println("Still is undirected (true): \t" + G.isUndirected());
576     G.addEdge("A", "B", 2); G.addEdge("B", "C", 1); G.addEdge("C", "D", 3);
577     G.addEdge("F", "A", 2); G.addEdge("A", "D", 5);
578     System.out.println("After adding edges AB, BC, CD, AF, AD arcs");
579     System.out.println("After adding 5 edges/a.k.a. 5 pairs of arcs = 10 arcs
(10): \t" + G.m());
580     System.out.println("Still is undirected (true): \t" + G.isUndirected());
581     G.addEdge("A", "A", 6); // adding a loop
582     System.out.println("A->A loop=>directed(false): \t" + G.isUndirected());
583     System.out.println(G);
584     System.out.println(G.m());
585     G.removeArc("C", "A"); // removing an arc that does not exist is okay
586     G.removeEdge("A", "A"); // removing a loop
587     System.out.println(G.m());
588     System.out.println("removing the loop makes it undirected (true): \t" + G.
isUndirected());
589     G.addArc("A", "C", 3); // adding an arc
590     System.out.println("adding an arc makes it directed (=>false): \t" + G.
isUndirected()); //-->
591     System.out.println("Graph now has vertices (6): \t" + G.n());
592     System.out.println("Graph now has arcs (11): \t" + G.m());
593     System.out.println(G);
594     System.out.println("Successors to C (B,D): " + G.getSuccessors("C"));
595     System.out.println("Predecessors to C (A,B,D): " + G.getPredecessors("C"));
596
597
598
599
600     G.removeArc("A", "C"); // removing an arc
601     System.out.println("remov A-C => undirected (true): \t" + G.isUndirected());
602     //System.out.println(G);
603     System.out.println("FILE SAVED IN withA");
604     G.saveTGF("withA.tgf");
605
606     System.out.println("Predeces A (B, D, F) : \t" + G.getPredecessors("A"));
607     System.out.println("Success A (B, D, F) : \t" + G.getSuccessors("A"));
608
609     G.removeVertex("A");
610     System.out.println("A removed; graph has now: " + G.n() + " (5) vertices and "
+ G.m() + " (4) arcs");
611     //System.out.println(G);
612     System.out.println("Preceeding C: (B, D) " + G.getPredecessors("C"));
613     //System.out.println(G);
614     System.out.println("FILE SAVED IN withoutA");
615     G.saveTGF("withoutA.tgf");
616
617     System.out.println("removing some more vertices");
618     G.removeVertex("E"); G.removeVertex("F");
619     System.out.println(G);
620     G.removeVertex("D");
621     System.out.println("removing some more vertices");
622     int m = G.m();
623     System.out.println(G);
624     G.addVertex("Z");
625     System.out.println("adding vertex should not 'resurreect' any old edges (m = "
+ m + ") [" + G.m() + "]");
626     System.out.println(G);
627     System.out.println("Returns the weight of the edge[B, C]-->Expected[1]: " + G.
getWeight("B", "C"));
628     System.out.println("Returns the weight of the edge[B, D]-->Expected[-1]: " + G.
getWeight("B", "D"));
629     // AdjMatGraph<String> test1 = new AdjMatGraph<String>();
630     // System.out.println(test1);
631     // loadTGF("gameMap.tgf", test1);
632     // System.out.println(test1);
633 }
634

```

635 }