```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Scanner;

/**
 *
 * Created: CS Team
 * Modified: Adrianna Valle
 * Date: 12-04-16
 *
 * AdjMatGraph has been optimized to include and consider weighted edges. Everything else was kept
 * constant. An AdjMatGraph was used instead of an AdjacencyList in order to allow for both
 * expansion capabiltiies as well as easy accessiblity to succeeding vertices/weights.
 */
public class AdjMatGraph<T> implements Graph<T>, Iterable<T> {
  public static final int NOT_FOUND = -1;
  private static final int DEFAULT_CAPACITY = 1; // Small so that we can test expand
  private static final boolean VERBOSE = false;  // print while reading TGF?

  private int n;   // number of vertices in the graph
  private Integer[][] arcs;   // adjacency matrix of arcs
  private T[] vertices;   // values of vertices

  /*****************************************************************
    Constructor. Creates an empty graph.
    *****************************************************************/
  @SuppressWarnings("unchecked")
  public AdjMatGraph() {
    n = 0;
    this.arcs = new Integer[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
    this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
  }

  /********** NEW METHODS ******************************************/
  /**
   * Construct a copy (clone) of a given graph.
   * The new graph will have all the same vertices and arcs as the original.
   * A *shallow* copy is performed: the graph structure is copied, but
   * the new graph will refer to the exact same vertex objects as the original.
   */
  @SuppressWarnings("unchecked")
  public AdjMatGraph(AdjMatGraph<T> g) {
    n = g.n;
    vertices = (T[]) new Object[g.vertices.length];
    arcs = new Integer[g.arcs.length][g.arcs.length];
    for (int i = 0; i < n; i++) {
      vertices[i] = g.vertices[i];
      for (int j = 0; j < n; j++) {
        arcs[i][j] = g.arcs[i][j];
      }
    }
  }

  /*****************************************************************
    * Load vertices and edges from a TGF file into a given graph.
    * @param tgfFile - name of the TGF file to read
    * @param g - graph to which vertices and arcs will be added.
    *          g must be empty to start!
    * @throws FileNotFoundException
    *****************************************************************/
  public static void loadTGF(String tgf_file_name, AdjMatGraph<String> g) throws FileNotFoundException {
    if (!g.isEmpty()) throw new RuntimeException("Refusing to load TGF data into non-empty graph.");
    Scanner fileReader = new Scanner(new File(tgf_file_name));
    // Keep a mapping from TGF vertex ID to AdjMatGraph vertex ID.
    // This allows vertex IDs to be written out of order in TGF.
    // It also supports non-integer vertex IDs.
    HashMap<String,Integer> vidMap = new HashMap<String,Integer>();
    try {
      // Read vertices until #
      while (fileReader.hasNext()) {
        // Get TGF vertex ID
        String nextToken = fileReader.next();
        if (nextToken.equals("#")) {
          break;
        }
        vidMap.put(nextToken, g.n());
        String label = fileReader.hasNextLine() ? fileReader.nextLine().trim() : fileReader.next();
        if (VERBOSE) {
          System.out.println("Adding vertex " + g.n() + " (" + nextToken + " = \"" + label + "\")");
        }
        g.addVertex(label);
      }

      // Read edges until EOF
      while (fileReader.hasNext()) {
        // Get src and dest
        String src = fileReader.next();
        String dest = fileReader.next();
        // Discard label if any
        int label= -1;
        if (fileReader.hasNextLine()) {
          try{
            label = Integer.parseInt(fileReader.nextLine().trim());
          }
          catch(NumberFormatException ex){
            label = 1;
          }
        }
        g.addArc(vidMap.get(src), vidMap.get(dest),label);
      }
    } catch (RuntimeException e) {
      System.out.println("Error reading TGF");
      throw e;
    } finally {
      fileReader.close();
    }
  }


  /**
   * An iterator that iterates over the vertices of an AdjMatGraph.
   */
  private class VerticesIterator implements Iterator<T> {
    private int cursor = 0;

    /** Check if the iterator has a next vertex */
    public boolean hasNext() {
      return cursor < n;
    }

    /** Get the next vertex. */
    public T next() {
      if (cursor >= n) {
        throw new NoSuchElementException();
      } else {
        return vertices[cursor++];
      }
    }

    /** Remove is not supported in this iterator. */
    public void remove() {
      throw new UnsupportedOperationException();
    }
  }

  /**
   * Create a new iterator that will iterate over the vertices of the array when
asked.
   * @return the new iterator.
   */
  public Iterator<T> iterator() {
    return new VerticesIterator();
  }

  /**
   * Check if the graph contains the given vertex.
   */
  public boolean containsVertex(T vertex) {
    return getIndex(vertex) != NOT_FOUND;
  }


  /**** FAMILIAR METHODS ******************************************/


  /******************************************************************
    Returns true if the graph is empty and false otherwise.
    ******************************************************************/
  public boolean isEmpty() {
    return n == 0;
  }

  /******************************************************************
    Returns the number of vertices in the graph.
    ******************************************************************/
  public int n() {
    return n;
  }

  /******************************************************************
    Returns the number of arcs in the graph by counting them.
    ******************************************************************/
  public int m() {
    int total = 0;
```

```java

    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (arcs[i][j]!=null) {
          total++;
        }
      }
    }
    return total;
  }

  /**
   * Returns array of all vertices.
   */
  public T[] getVertices() {
    return vertices;
  }

  /*****************************************************************
    Returns true iff a directed edge exists from v1 to v2.
    *****************************************************************/
  public boolean isArc(T srcVertex, T destVertex) {
    int src = getIndex(srcVertex);
    int dest = getIndex(destVertex);
    return src != NOT_FOUND && dest != NOT_FOUND && arcs[src][dest]!=null;
  }

  /*****************************************************************
    Returns true iff an arc exists between two given indices.
    @throws IllegalArgumentException if either index is invalid.
    *****************************************************************/
  protected boolean isArc(int srcIndex, int destIndex) {
    if (!indexIsValid(srcIndex) || !indexIsValid(destIndex)) {
      throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
+ ", " + destIndex);
    }
    return arcs[srcIndex][destIndex]!=null;
  }

  /*****************************************************************
    Returns true iff an edge exists between two given vertices
    which means that two corresponding arcs exist in the graph.
    *****************************************************************/
  public boolean isEdge(T srcVertex, T destVertex) {
    int src = getIndex(srcVertex);
    int dest = getIndex(destVertex);
    return src != NOT_FOUND && dest != NOT_FOUND && isArc(src, dest) && isArc(dest,
src);
  }

  /*****************************************************************
    Returns true IFF the graph is undirected, that is, for every
    pair of nodes i,j for which there is an arc, the opposite arc
    is also present in the graph.
    *****************************************************************/
  public boolean isUndirected() {
    for (int i = 1; i < n(); i++) {
      // optimize to avoid checking pairs twice.
      for (int j = 0; j < i; j++) {
        if (arcs[i][j] != arcs[j][i]) {
          return false;
        }
      }
    }
    return true;
  }

  /*****************************************************************
    Adds a vertex to the graph, expanding the capacity of the graph
    if necessary.  If the vertex already exists, it does not add it again.
    *****************************************************************/
  public void addVertex (T vertex) {
    if (getIndex(vertex) != NOT_FOUND) return;
    if (n == vertices.length) {
      expandCapacity();
    }

    vertices[n] = vertex;
    //  for (int i = 0; i <= n; i++) {
    //  // if (arcs[n][i] || arcs[i][n]) throw new RuntimeException("Corrupted
AdjacencyMatrix");
    //    arcs[n][i] = false;
    //    arcs[i][n] = false;
    //  }
    n++;
  }

  /*****************************************************************
    Helper. Creates new arrays to store the contents of the graph
    with twice the capacity.
    *****************************************************************/
  @SuppressWarnings("unchecked")
  private void expandCapacity() {
    T[] largerVertices = (T[])(new Object[vertices.length*2]);
    Integer[][] largerAdjMatrix =
      new Integer[vertices.length*2][vertices.length*2];

    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        largerAdjMatrix[i][j] = arcs[i][j];
      }
      largerVertices[i] = vertices[i];
    }

    vertices = largerVertices;
    arcs = largerAdjMatrix;
  }

  /*****************************************************************
    Removes a single vertex with the given value from the graph.
    Uses equals() for testing equality.
    *****************************************************************/
  public void removeVertex (T vertex) {
    int index = getIndex(vertex);
    if (index != NOT_FOUND) {
      removeVertex(index);
    }
  }

  /*****************************************************************
    Helper. Removes a vertex at the given index from the graph.
    Note that this may affect the index values of other vertices.
    @throws IllegalArgumentException if the index is invalid.
    *****************************************************************/
  protected void removeVertex (int index) {
    if (!indexIsValid(index)) {
      throw new IllegalArgumentException("No such vertex index");
    }
    n--;

    // Remove vertex.
    for (int i = index; i < n; i++) {
      vertices[i] = vertices[i+1];
    }

    // Move rows up.
    for (int i = index; i < n; i++) {
      for (int j = 0; j <= n; j++) {
        arcs[i][j] = arcs[i+1][j];
      }
    }

    // Move columns left
    for (int i = index; i < n; i++) {
      for (int j = 0; j < n; j++) {
        arcs[j][i] = arcs[j][i+1];
      }
    }

    // Erase last row and last column
    for (int a = 0; a < n; a++) {
      arcs[n][a] = null;
      arcs[a][n] = null;
    }
  }

  /*****************************************************************
    Inserts an edge between two vertices of the graph.
    If one or both vertices do not exist, ignores the addition.
    An edge cannot be pointed to itself therefore is omitted.
    *****************************************************************/
  public void addEdge(T vertex1, T vertex2, int weight) {
    int index1 = getIndex(vertex1);
    int index2 = getIndex(vertex2);
    if (index1 != NOT_FOUND && index2 != NOT_FOUND && index1!=index2) {
      addArc(index1, index2, weight);
      addArc(index2, index1, weight);
    }
  }

  /*****************************************************************
    Inserts an arc from srcVertex to destVertex.
    If the vertices exist, else does not change the graph. Method
    doesn't allow the vertex to ave an edge to itself.
    *****************************************************************/
  public void addArc(T srcVertex, T destVertex, int weight) {
    int src = getIndex(srcVertex);
    int dest = getIndex(destVertex);
    if (src != NOT_FOUND && dest != NOT_FOUND && src!=dest) {
      addArc(src, dest, weight);
    }
  }

  /*****************************************************************
    Helper. Inserts an edge between two vertices of the graph. Note, an arc
    does not point to itself in the context of our game therefore the ability has
    been omitted.
    @throws IllegalArgumentException if either index is invalid.
    ************************************************************/
  protected void addArc(int srcIndex, int destIndex,int  weight) {
    if (!indexIsValid(srcIndex) || !indexIsValid(destIndex)&& srcIndex== destIndex)
{
      throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
+ ", " + destIndex);
    }
    arcs[srcIndex][destIndex] = weight;
  }

  /************************************************************
    Removes an edge between two vertices of the graph.
    If one or both vertices do not exist, ignores the removal.
    ************************************************************/
  public void removeEdge(T vertex1, T vertex2) {
    int index1 = getIndex(vertex1);
    int index2 = getIndex(vertex2);
    if (index1 != NOT_FOUND && index2 != NOT_FOUND) {
      removeArc(index1, index2);
      removeArc(index2, index1);
    }
  }

  /************************************************************
    Removes an arc from vertex src to vertex dest,
    if the vertices exist, else does not change the graph.
    ************************************************************/
  public void removeArc(T srcVertex, T destVertex) {
    int src = getIndex(srcVertex);
    int dest = getIndex(destVertex);
    if (src != NOT_FOUND && dest != NOT_FOUND) {
      removeArc(src, dest);
    }
  }

  /************************************************************
    Helper. Removes an arc from index v1 to index v2.
    @throws IllegalArgumentException if either index is invalid.
    ************************************************************/
  protected void removeArc(int srcIndex, int destIndex) {
    if (!indexIsValid(srcIndex) || !indexIsValid(destIndex)) {
      throw new IllegalArgumentException("One or more invalid indices: " + srcIndex
+ ", " + destIndex);
    }
    arcs[srcIndex][destIndex] = null;
  }


  /************************************************************
    Returns the index value of the first occurrence of the vertex.
    Returns NOT_FOUND if the key is not found.
    ************************************************************/
  protected int getIndex(T vertex) {
    for (int i = 0; i < n; i++) {
      if (vertices[i].equals(vertex)) {
        return i;
      }
    }
    return NOT_FOUND;
  }

  /***********************************************************
    * Returns the weight of the edge from the one vertex to another.
    If no edge is present, returns -1.
    * @param: T vertex1, T vertex2 @return int
    **********************************************************/
  public int getWeight(T vertex1, T vertex2){
    int x = getIndex(vertex1);
    int y = getIndex(vertex2);
    if(x<0 || y<0)
      return -1;
    return arcs[x][y];

  }

  /************************************************************
    Returns the vertex object that is at a certain index
    ************************************************************/
  protected T getVertex(int v) {
    if (!indexIsValid(v)) {
      throw new IllegalArgumentException("No such vertex index: " + v);
    }
    return vertices[v];
  }

  /************************************************************
    Returns true if the given index is valid.
    ************************************************************/
  protected boolean indexIsValid(int index) {
    return index < n && index >= 0;
  }

  /************************************************************
    Retrieve from a graph the vertices x pointing to vertex v (x->v)
    and returns them onto a linked list
    ****************************************************************/
  public LinkedList<T> getPredecessors(T vertex) {
    LinkedList<T> neighbors = new LinkedList<T>();

    int v = getIndex(vertex);

    if (v == NOT_FOUND) return neighbors;
    for (int i = 0; i < n; i++) {
      if (arcs[i][v] != null) {
        neighbors.add(getVertex(i)); // if T then add i to linked list
      }
    }
    return neighbors;
  }

  /****************************************************************
    * Retrieve from a graph the vertices x following vertex v (v->x)
    and returns them onto a linked list
    ****************************************************************/
  public LinkedList<T> getSuccessors(T vertex){
    LinkedList<T> neighbors = new LinkedList<T>();

    int v = getIndex(vertex);
    if (v == NOT_FOUND) return neighbors;
    for (int i = 0; i < n; i++) {
      if (arcs[v][i]!= null) {
        neighbors.add(getVertex(i)); // if T then add i to linked list
      }
    }
    return neighbors;
  }

  /****************************************************************
    Returns a string representation of the graph.
    ****************************************************************/
  public String toString() {
    if (n == 0) {
      return "Graph is empty";
    }

    String result = "";

    //result += "\nArcs\n";
    //result += "---------\n";
    result += "\ni ";

    for (int i = 0; i < n; i++) {
      result += "" + getVertex(i);
      if (i < 10) {
        result += " ";
      }
    }
    result += "\n";

    for (int i = 0; i < n; i++) {
      result += "" + getVertex(i) + " ";

      for (int j = 0; j < n; j++) {
        if (arcs[i][j]!=null) {
          result += arcs[i][j] + " ";
        } else {
          result += "- "; //just empty space
        }
      }
      result += "\n";
    }
    return result;
  }

  /****************************************************************
    * Saves the current graph into a .tgf file.
    * If it cannot save the file, a message is printed.
    ***************************************************************/
  public void saveTGF(String tgf_file_name) {
    try {
      PrintWriter writer = new PrintWriter(new File(tgf_file_name));

      //prints vertices by iterating through array "vertices"
      for (int i = 0; i < n(); i++) {
        if (vertices[i] == null){
          break;
        } else {
          writer.print((i+1) + " " + vertices[i]);
          writer.println("");
        }
      }
      writer.print("#"); // Prepare to print the edges
      writer.println("");

      //prints arcs by iterating through 2D array
      for (int i = 0; i < n(); i++) {
        for (int j = 0; j < n(); j++) {
          if (arcs[i][j]!=null) {
```

```
      writer.print((i+1) + " " + (j+1) + " " + arcs[i][j]);
      writer.println("");
     }
    }
   }
   writer.close();
  } catch (IOException ex) {
   System.out.println("***(T)ERROR*** The file could nt be written: " + ex);
  }
 }

 //looping to itself is prohibited.
 /** Testing Driver for AdjMatGraph.  This will not help you test AdjMatGraphPlus.
*/
 public static void main (String args[]) throws FileNotFoundException {
  System.out.println("NORMAL OPERATIONS");
  System.out.println("_________________");
  AdjMatGraph<String> G = new AdjMatGraph<String>();
  System.out.println("New graph is empty  (true): \t" + G);
  System.out.println("Empty=> undirected  (true): \t" + G.isUndirected());
  System.out.println("Empty graph no vertices(0): \t" + G.n());    System.out.
println("Empty graph no arcs    (0): \t" + G.m());
  G.addVertex("A"); G.addVertex("B"); G.addVertex("C");
  G.addVertex("D"); G.addVertex("E"); G.addVertex("F");
  System.out.println("After adding 6 vert.   (6): \t " + G.n());
  System.out.println("After adding no arcs   (0): \t" + G.m());
  System.out.println("Still is undirected (true): \t" + G.isUndirected());

  G.addEdge("A", "B",2); G.addEdge("B", "C",1); G.addEdge("C", "D",3);
  G.addEdge("F", "A",2); G.addEdge("A", "D",5);
  System.out.println("After adding edges AB, BC, CD, AF, AD arcs");
  System.out.println("After adding 5  edges/a.k.a. 5 pairs of arcs = 10 arcs
(10): \t" + G.m());
  System.out.println("Still is undirected (true): \t" + G.isUndirected());
  G.addEdge("A", "A",6); // adding a loop
  System.out.println("A->A loop=>directed(false): \t" + G.isUndirected());
  System.out.println(G);
  System.out.println(G.m());
  G.removeArc("C", "A"); // removing an arc that does not exist is okay
  G.removeEdge("A", "A"); // removing a loop
  System.out.println(G.m());
  System.out.println("removing the loop makes it undirected (true): \t" + G.
isUndirected());

  G.addArc("A", "C",3); // adding an arc
  System.out.println("adding an arc makes it directed (=>false): \t" + G.
isUndirected()); //-->
  System.out.println("Graph now has vertices   (6): \t " + G.n());
  System.out.println("Graph now has arcs      (11): \t" + G.m());
  System.out.println(G);
  System.out.println("Successors to  C (B,D): " + G.getSuccessors("C"));
  System.out.println("Predecess to C (A,B,D): " + G.getPredecessors("C"));



  G.removeArc("A", "C"); // removing an arc
  System.out.println("remov A-C => undirected (true): \t" + G.isUndirected());
  //System.out.println(G);
  System.out.println("FILE SAVED IN withA");
  G.saveTGF("withA.tgf");

  System.out.println("Predeces A (B, D, F) : \t" + G.getPredecessors("A"));
  System.out.println("Success  A (B, D, F) : \t" + G.getSuccessors("A"));

  G.removeVertex("A");
  System.out.println("A removed; graph has now: " + G.n() + " (5) vertices and "
+ G.m() + " (4) arcs");
  //System.out.println(G);
  System.out.println("Preceeding C: (B, D) " + G.getPredecessors("C"));
  //System.out.println(G);
  System.out.println("FILE SAVED IN withoutA");
  G.saveTGF("withoutA.tgf");

  System.out.println("removing some more vertices");
  G.removeVertex("E");  G.removeVertex("F");
  System.out.println(G);
  G.removeVertex("D");
  System.out.println("removing some more vertices");
  int m = G.m();
  System.out.println(G);
  G.addVertex("Z");
  System.out.println("adding vertex should not 'resurreect' any old edges  (m = "
+ m + ") [" + G.m() + "]");
  System.out.println(G);
  System.out.println("Returns the weight of the edge[B, C]-->Expected[1]:  "+ G.
getWeight("B","C"));
  System.out.println("Returns the weight of the edge[B, D]-->Expected[-1]: "+ G.
getWeight("B","D"));
//   AdjMatGraph<String> test1 = new AdjMatGraph<String>();
//   System.out.println(test1);
//   loadTGF("gameMap.tgf",test1);
//   System.out.println(test1);
 }

```

```
}
```