1 Overview

In this assignment, you will implement a “GraphFun” game based on a graph. We provide a GUI for the game, you have to implement a class for the graph, the class for path finding, and finish game implementation.

The objective of the game is as follows. First a graph is constructed and displayed, see the figure above. There are $n$ graph vertices, numbered from 0 to $n - 1$. The graph has 2 special vertices, marked in red (vertices 0 and 5 in the picture above). These special vertices are called the startVertex and the endVertex. The user plays against the computer. The user goes first and selects an edge to delete. The computer chooses an edge to freeze. The user can delete only non-frozen edges. The frozen edges are marked in red in the output window. The computer wins if it can “freeze” a path between the startVertex and the endVertex. The user wins if he/she deletes enough edges so that there is no path between the startVertex and the endVertex.

The GUI has a “new” button to start a new game, 2 windows for entering the next edge to delete. An edge to delete is specified by entering the names of its 2 endpoints in the 2 windows and pressing the “enter” button.

You must write all the code yourself. You cannot use code from the textbook, the Internet, other students, or any other sources. You can use Java built-in classes LinkedList, Stack, Vector, and built-in interface Iterator, but no other built-in class.

2 Modified Adjacency List Representation for Graph data structure

You will need to implement a simple (no parallel edges and no self-loops) and undirected (i.e. edge (u,v) is the same edge as (v,u)) graph for this assignment. You have to use the adjacency list implementation. You can implement it exactly as discussed in class, or you can implement a simpler version described in this section. Here is the suggested simpler version. Keep the vertices of the graph in a Java Vector class or Java LinkedList class. I suggest that you do not store edge objects in any container structure, although you will still need edge objects. Deletion of edges is much
simpler if you do not have the container to store edge objects. The only disadvantage is that the iterator over edge objects is a little harder to implement, but I’ll explain how to implement it later in section 5.4. Just like discussed in class, each vertex object has a reference to a linked list of the edges adjacent to it (the adjacency list). Let \( e \) be an edge with endpoints \( u \) and \( v \). To simplify things further, at the edge object \( e \) you don’t have to store the position of \( e \) in the adjacency list of \( u \) and the adjacency list of \( v \) (the thick brown-yellowish references on the slide 9 in lecture notes 16). You can simply search through the adjacency lists of \( u \) and \( v \) when you need to delete edge \( e \). Keep in mind that this is not very efficient for dense graphs (that is if a vertex can have many adjacent edges).

3 Classes Provided Fully Implemented

3.1 Coordinate

This is the class for storing \( x \) and \( y \) coordinates, as well as the name, for each graph vertex of the game. I use the Coordinate objects for displaying the vertices in GUI. I give you the code for displaying the game graph, so all you have to do is to store the Coordinate objects at each vertex of your graph. That is for your graph, the vertices are of type \( \text{Vertex} \ < \ \text{Coordinate} \ ). \) You do not need to use any methods of the class Coordinate nor create any objects of this class.

3.2 Pair

This class represents a pair of integers. If \( p \) is of class Pair, you will only need to use methods \( p \).getFirst() and \( p \).getSecond() which get you the 2 components of the Pair respectively. You will need this class to figure out which edges to insert into your game class.

3.3 TestGraph

Tests your graph implementation. Compile and run with your code by typing ”javac TestGraph.java” and then ”java TestGraph”. To start running from say, test 5, type ”java TestGraph 5”

3.4 TestPath

Tests your FindPath implementation. Compile and run with your code by typing ”javac TestPath.java” and then ”java TestPath”. To start running from test 2 type ”java TestPath 2”

4 Classes Provided Partly Implemented

4.1 GraphFun

GraphFun is the main program, which implements GUI and has the main() method. It is partially implemented. In this class, you are to implement 2 methods which handle pressing of the “enter” and the “new” buttons. When the “new” button is pressed, method \( \text{private void startNewGame()} \) is called automatically. When the “enter” button is pressed, method \( \text{private void nextMove(int i, int j)} \) is called automatically. These are the 2 methods that you need to implement. But first I explain which constants, variables, and methods already implemented by me will be useful for you.

Useful Constants

Here is a list of constants (final variables) that I already put in the code and you may wish to change to modify the size of the game (the number of nodes). Note that you don’t have to change the game size.

- \( \text{private static final int \ NUM_HORIZONTAL} = \ 3; : \) roughly the number of columns of vertices in the graph.
- \( \text{private static final int \ NUM_VERTICAL} = \ 3; : \) roughly the number of rows of vertices in the graph.
The final number of vertices in the game graph will be \( \text{NUM}_\text{VERTICAL} \times \text{NUM}_\text{HORIZONTAL} \). DO NOT change the private static final int \( \text{NUM}_\text{VERTICES} \), it will be automatically changed to the correct number of nodes if you change \( \text{NUM}_\text{VERTICAL} \) and \( \text{NUM}_\text{HORIZONTAL} \). I recommend that for debugging, you start with \( \text{NUM}_\text{HORIZONTAL} = 3 \) and \( \text{NUM}_\text{VERTICAL} = 3 \). When you are done debugging, you can increase these numbers for a larger game. Do not set \( \text{NUM}_\text{HORIZONTAL} \) and \( \text{NUM}_\text{VERTICAL} \) to less than 2.

**Useful Variables**

Here is a list of private variables that I already put in the code and you have to use in your implementation. Note that you should not delete these variables.

- **private Graph graph**: use this variable for the graph representation of the game. That is this is the graph which stores all the vertices and current edges of the game. It also stores the information about which edges are frozen.

- **private Vector<\text{Vertex<Coordinate>>} vertexLookup**: lookup from integer vertex names to \( \text{Vertex<Coordinate>} \) object storing the corresponding vertex objects. All vertices in the game graph have names 0, 1, ..., \( n-1 \), where \( n \) is the total number of nodes. After you insert a vertex with name, say 3, in the graph, the insertion method will return a reference to \( \text{Vertex<Coordinate>} \), that is the object which it actually created for the new vertex with name 3. You should store the returned reference at index 3 of Vector vertexLookup. Similarly, for any vertex with name \( i \), the vertex object corresponding to it should be stored at index \( i \) of Vector vertexLookup. Then, if you need to find the \( \text{Vertex} \) whose name is, say 7, the vertexLookup.elementAt(7) will give you the correct \( \text{Vertex} \). Just to make sure you get this right, I have already implemented inserting all vertices in the graph and storing the returned objects at the correct locations in method startGame() of class GraphFun, see Section 4.1.

- **private int startVertex**: an integer in the range from 0 to \( n-1 \) which is the special start vertex. Note that I have implemented the code which generates the start vertex, so you don’t need to worry about setting it, you can just use this variable.

- **private int endVertex**: an integer in the range from 0 to \( n-1 \) which is the special end vertex. Note that I have implemented the code which generates the end vertex, so you don’t need to worry about setting it, you can just use this variable.

**Useful Methods**

You will need to use the following methods which are completely implemented by me:

- **private void drawBoard(Iterator<\text{Vertex<Coordinate>>} nodeIt, Iterator<\text{Edge<Coordinate>>} edgeIt)** takes iterators over vertices and edges (these you can get from your graph class, see section 5.4) and redraws the game board.

- **private void stopGame()**: stops current game by making the text windows for user windows uneditable.

- **private void displayMessage(String message)**: displays the input message in the game window.

**Methods to Implement in GraphFun**

You will need to add the body to the following methods

- **private void nextMove(int i, int j)** This method will be called automatically after the user presses the “enter” button. The method takes the the user chosen edge as an input, where the edge is between vertices with names \( i \) and \( j \). You first have to check that there is an edge between nodes \( i \) and \( j \), and that this edge is not frozen by the computer. If an edge is not in the graph or it is frozen, let the user know by displaying a message using method displayMessage. If this edge is in the graph, then delete this edge from the graph and check if the user has won (that is if there is no path now between the start and end vertices). If the user won, stop the game by calling the provided method stopGame(), and display the winning message using provided method displayMessage.
If the user has not won yet, then generate a valid move by the computer, that is freeze an edge which is not already frozen. You should freeze the edge by setting its marker to true (which means that in the very beginning of the game you should set markers to be false for all the edges). The easiest thing to do here is to find any path from start to end vertices and freeze any unfrozen edge on this path. The computer will be playing really “dumb” though in this case (but no marks deducted from the assignment). If you want to implement a smarter simple strategy for the computer, then check if there is any edge such that deletion of that edge will result in the user winning. If yes, freeze that edge. Also check if there is an edge such that if you freeze that edge the computer will win. If yes, freeze that edge. If you are feeling a little more brave, you can explore what happens 2 moves down, that is check if there are 2 edges that the computer can freeze and win the game. The above simple strategy is not optimal. The optimal strategy is a bit more complicated but interesting. Talk to me if you want to know the optimal strategy. After you choose and freeze an edge, display the new graph by calling method freeze so that the user sees the new graph.

The last thing to do after generating a computer move is to check if the computer has won, by checking if there is a path of “frozen” edges between the start and end vertices. If the computer has won, stop the game by calling the provided method stopGame(), and display the winning message using provided method displayMessage.

- **private void startNewGame()** This method is called automatically when the user presses “new” button, and it initializes a new game. I put a few first lines in this method, and you have to finish the implementation. I put in the lines:

  1. Coordinate [] vertices = getVertices();
  2. Pair[] edges = getEdges(vertices);
  3. setEndVertices();
  4. graph = new Graph();
  5. vertexLookup = new Vector < Vertex < Coordinate >>(NUM_VERTICES);
  6. for ( int i = 0; i < NUM_VERTICES; i++)
      vertexLookup.add(i,graph.insertVertex(vertices[i]));

Lines (1) and (2) generate the set of nodes and edges for the game graph. You have to use these arrays to insert the vertices and edges into the graph. The array vertices holds objects of type Coordinate, and this is exactly the object class that your graph vertices should store. So, for example, to insert the first vertex (which is stored at index 0 of the array vertices), you use:

graph.insertVertex(vertices[0]);

I actually insert all the vertices for you (but not the edges). The size of array vertices is NUM_VERTICES. The array edges holds the set of possible edges for the graph, edges[i].getFirst() returns the name of the first endpoint of the edge, and edges[i].getSecond() returns the name of the second endpoint of the edge. Notice that the array edges may contain multiple copies of the same edges (because I generate edges randomly and do not check for duplicates), but you are not allowed to insert parallel edges (i.e. more than 1 edge between the same pair of vertices) into the graph. Thus before inserting any edge into the graph from array edges, you have to make sure that you didn’t insert that edge already. The size of array edges is NUM_EDGES, but the actual number of edges in the graph after you ignore any duplicates may be smaller.

Line (3) initializes the new startVertex and endVertex. Line (4) creates a new graph. Line (5) creates a new lookup table from vertex names to graph vertex objects, and it is to be used as was explained above.

The last two lines insert all the vertices and store the objects in the vertexLookup for later use.

After these 7 lines, you should write code to insert edges from the array edges[] into the graph and call method drawBoard() to draw the graph in the game window.

You can add any private variables and any private methods that you like to class GraphFun, but you have to explain (put the explanation in comments) why you need them.

## 5 Classes To Implement

You are to fully implement at 5 Java classes: GraphException, Vertex, Edge, Graph, and FindPath.
5.1 GraphException

This is the class to handle graph exceptions.

5.2 Vertex\(<V>\)

This class represents a vertex of the graph. \(V\) is a generic name for an object to be stored at the vertex. In this assignment, you will want to use the vertices of class: \(\text{Vertex} < \text{Coordinate} >\), since each vertex will correspond to a vertex on a game graph, and a vertex on a game graph has x and y position (in the game window), as well as a name, given by a number from 0 to \(n-1\). I suggest that you use any of the allowed container classes (vector, linkedList) for the adjacency list. Vertices should also have a private boolean marker variable which you should use for your path finding algorithm.

- `public Vertex(V objectIn):` constructor for the vertex. Takes an object to store at the vertex.
- `public V getObject():` returns the object stored at the vertex.
- `public void addAdjacent(Edge<V> e) throws GraphException:` adds an edge to the adjacency list of the vertex. Throws graph exception if this edge is already in the adjacency list of the vertex. Note that to check if an edge \(e\) is already in the adjacency list, you will have to get the 2 endpoints of the edge \(e\), say the endpoints are vertices \(u\) and \(v\), and you will have to check whether edge with endpoints \((u,v)\) is already in the adjacency list, or whether edge with endpoints \((v,u)\) is already in the adjacency list, since these two cases correspond to the same edge.
- `public Iterator<Edge<V>> incidentEdges():` returns iterator over the edge objects incident to the vertex.
- `public boolean isAdjacent(Vertex<V> v):` returns true if the vertex is adjacent to vertex \(v\), false otherwise.
- `public void removeAdjacent(Edge<V> e) throws GraphException:` removes an adjacent edge. Throws an exception if edge \(e\) is not actually adjacent to the vertex.
- `public void setMarker(boolean mark):` Sets the marker for this vertex.
- `public boolean getMarker():` returns the marker for this vertex.

In addition to the methods I have listed, you can implement any other public/private methods that you like. For each public method, you must indicate (in comments), why you need it to be public.

5.3 Edge\(<V>\)

This is the class implementing edge objects. Edges should also have a private boolean marker variable, which you should use for setting an edge as "frozen". A marker set to \(true\) indicates that an edge is frozen. A marker set to \(false\) indicates that the edge is not frozen. You have to stick to this convention because that’s what I assume in my `drawBoard()` method. I display all the edges with markers set to true in red color, and "unfrozen" edges with the green color. This class should have the following methods:

- `Edge(Vertex<V> u, Vertex<V> v):` constructor for the edge, takes as inputs the two end point vertices.
- `public Vertex<V> getEndPoint1():` returns first endpoint.
- `public Vertex<V> getEndPoint2():` returns second endpoint.
- `public void setMarker(boolean mark):` Sets the marker for this edge.
- `public boolean getMarker():` returns the marker for this edge.

You can implement any other methods that you like, but for each public method you should explain why you need it to be public.
5.4 Graph < V >

This class represents the graph for the game. Must implement the provided GraphInterface< V >. You cannot use the edge list or the adjacency matrix representation for the graph. You can use the adjacency list representation, or the modified adjacency list representation, as explained in the section 2. You are to implement the following public methods, and only the following methods can be public:

- **public Graph():** This is a constructor for the graph.
- **public Vertex< V > insertVertex(V o):** Adds a new vertex and stores object o at the vertex. Returns the newly created vertex.
- **int getNumVertices():** returns number of vertices
- **int getNumEdges():** returns number of edges
- **public Edge< V > findEdge(Vertex< V > u, Vertex< V > v) throws GraphException:** returns the edge between vertices u and v, or null if the edge between u and v does not exist. Throws exception if either u or v are null pointers.
- **public boolean areAdjacent(Vertex <V> v, Vertex< V > u) throws GraphException:** returns true if u and v are adjacent, false otherwise, throws exception if either u or v are null pointers.
- **public Edge< V > insertEdge(Vertex< V > v, Vertex< V > u) throws GraphException:** inserts an edge between u and v in the graph and returns the newly inserted edge. Throws exception if this edge already exists in the graph, or if u and v are the same vertex (no self-loops are allowed).
- **public void deleteEdge(Edge< V > e) throws GraphException:** Deletes edge e from the graph. Throws exception if edge e is null.
- **public Vertex< V > giveOpposite(Vertex< V > v, Edge< V > e):** Returns the endpoint opposite of v in the edge e.
- **public Iterator< Vertex< V > > vertices():** Returns iterator over the vertex objects. Simplest way to implement this method (with one line) is to store all vertex objects in an allowed Java built-in container class (LinkedList, Vector), and return a built-in iterator of that container (usually listIterator()).
- **public Iterator< Edge< V > > edges():** Returns iterator over the edge objects. Notice that I suggest you do not have a container that stores the edge objects. Instead, when you need an iterator over the edge objects, simply go over the adjacency lists for all the vertices, making sure that you don’t include the same edge twice. Notice that you will visit each edge exactly twice if you go through the adjacency lists for all vertices. Avoid duplicates in the edge iterator as follows. First declare an empty container L, say of Java’s LinkedList class. Before you do anything, set the marker to “false” for all the nodes. Then, visit all the nodes one by one, using the method vertices(). When you visit a vertex v, look at all its edges in the adjacency list. For any edge e on the adjacency list of v, get the endpoints of e. If both of the endpoints are unmarked, you have not put edge e in the container L yet, so put e in L. If at least one of the endpoints of e is marked, then you have already put edge e in L when you were visiting the marked endpoint, so do not put e in L. After you looked at all the edges incident on v, mark vertex v. With this implementation, the method edges() is still O(m), where m is the number of edges, so it is as efficient as the implementation we have studied in class. Warning: even though you can mark the edges themselves, do not use the edge marker in this method. You will need the marker for the edges to set them frozen during the game, so calling the method edges() will ruin your frozen edges markings. This is not the most elegant solution (that is markers for vertices are OK to change, markers for edges are used for a very specific purpose in the game and not OK to change), but it is the easiest implementation.

These are the only public methods that your implementation can have. You can write as many private methods as you wish.
5.5 **FindPath <V>**

This is the class for path finding in a graph.

- **public boolean markedPathExists(Graph<V> g, Vertex<V> v, Vertex<V> u):** Returns true if there is a "frozen" path between v an u in g. Or in other words, there is a path consisting of "marked" edges, where an edge e is marked if e.getMarker() == true.

- **public Iterator<Vertex<V>> givePath(Graph<V> g, Vertex<V> v, Vertex<V> u):** If there is no path between u and v in the graph g, returns null. If there is a path, returns it as an iterator of vertices on this path. Notice that the first vertex in the iterator should be v an the last vertex should be u. For this method, we do not care if the edges are frozen (that is marked) or not.

You can implement any other methods that you want. For any public method, you must indicate why it has to be public.

### 6 Grading

Your grade will be computed as follows.

- Program compiles, produces a meaningful output: 15 marks
- **GraphTest pass:** 40 marks. Ten tests, 4 marks each.
- **PathTest pass:** 10 marks. Two tests, 5 marks each.
- Coding Style: 10 marks
- Graph implementation: 10 marks
- **GraphFun program implementation:** 15 marks.

### 7 Handing In Your Program

Zip all the .java files into a single zip file and submit them via the OWL system.