**CS 220 – Data Abstraction**

**PEX 4 – Shannon Switching Game**

**PEX 4 Preliminary Submission Due: 2200, Lesson 35, Friday, 20 April**

**PEX 4 Final Submission Due: 2200, Lesson 39, Wednesday, 2 May**

|  |
| --- |
| Help Policy: **AUTHORIZED RESOURCES:** Any, except another cadet’s program.  **NOTE:**   * Never copy another person’s work and submit it as your own. * Do not jointly create a program unless explicitly allowed. * You must document all help received from sources other than your instructor. * **DFCS will recommend a course grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.**  Documentation Policy:  * You must document all help received from any source other than your instructor. * The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. * If no help was received on this assignment, the documentation statement must state “NONE.” * If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why.  Vague documentation statements must be corrected before the assignment will be graded, and will result in a 5% deduction on the assignment.Turn-in Policies:  * On-time turn-in is at 2200 on the due date, same day for both M and T day sections. * Late turn-ins will receive a 25% penalty per 24 hours late unless prior arrangements have been made with your instructor. * There is no early turn-in bonus or extra credit for this assignment. |

1. Objectives

* Be able to use a generic graph ADT
* Be able to use graph algorithms such as shortest path
* Be able to use Javadoc documentation
* Be able to create and use a GUI
* Be able to document code using Javadoc

1. Pair Programming

For this assignment you may choose to use the [Pair Programming](http://en.wikipedia.org/wiki/Pair_programming) approach or you may work independently. If you choose to work with a partner, both students will receive the same grade and must work together ***on at least half*** of the solution. You must inform your instructor of your partner selection by Lesson 33. *You may work with any student currently enrolled in CS 220, regardless of section*.

1. Background

Your task for this exercise is to implement the [Shannon Switching Game](http://en.wikipedia.org/wiki/Shannon_switching_game). An applet that plays this game can be found [here](http://dfcs2/faculty/bower/shannon/graphg.html).

1. Preliminary Exercise

For the preliminary exercise you will create and draw a random [planar graph](http://en.wikipedia.org/wiki/Planar_graph) on which the game will be played. To do this you will use a [**SimpleWeightedGraph**](http://www.jgrapht.org/javadoc/org/jgrapht/graph/SimpleWeightedGraph.html) from the [JGraphT](http://www.jgrapht.org/) library. Your graph can use the [**Point**](http://docs.oracle.com/javase/7/docs/api/java/awt/Point.html) class for the vertices and the [**DefaultWeightedEdge**](http://www.jgrapht.org/javadoc/org/jgrapht/graph/DefaultWeightedEdge.html) class for the edges:

**private SimpleWeightedGraph<Point, DefaultWeightedEdge> graph;**

You will need to put the above declaration in a class to represent the Shannon Switching Graph as it will need more functionality than the **SimpleWeightedGraph** class (see provided code). In particular, the Shannon Switching Graph needs to know the width and height of the area on which it will be drawn. You are free to choose between making your own GUI in which you will most likely draw the graph on a JPanel or you may use the DrawingPanel class that was provided to you in CS 210.

The Shannon Switching Graph constructor should do the following:

* Create a start vertex near the upper-left corner such that there is room for other vertices to be randomly placed between this vertex and the corner of the graph area.
* Create an end vertex near the lower-right corner such that there is room for other vertices to be randomly placed between this vertex and the corner of the graph area.
* Randomly generate coordinates for a sufficiently large number of vertices to “fill” the graph area.
  + Only add a new random vertex if it is not too close to another vertex. Exactly how close is too close will depend on the size of the graph area.
  + Look at the demo applet linked above and the solution demonstrated in class to gain an understanding of what it means to “fill” the graph area.
  + A graph with too few vertices will result in edges being crowded together making the game difficult to play.
  + Leave a margin around the outer edge of the graph area so no vertices are exactly at the edge of the screen!
* Add edges to the graph until the degree of each vertex is between 3 and 5 (determined randomly). When adding edges to a vertex:
  + Sort all other vertices by distance so nearest vertices are connected first.
  + While there are more possible vertices to connect to, if the degree of the other vertex is less than or equal to 5 and adding this edge will not cause two edges to overlap, add it.
    - Some vertices may end up with fewer than 3 edges if all possibilities cause overlap.  
      (This is most likely to happen in the corners which is why the start and end vertices should not be placed all the way into the corner.)
    - You will need to explicitly compare this potential new edge to every other edge.  
      You may find methods in the [**java.awt.geom.Line2D**](http://docs.oracle.com/javase/7/docs/api/java/awt/geom/Line2D.html) class useful here.
    - Set the weight of all edges to 1.0.

Be sure to draw all edges and vertices, highlighting the start and end vertices.

1. Preliminary Submission Requirements

* Fill in your name at the top of all provided source files!
* Fill in your documentation statement at the top of ShannonSwitchingGraph.java!
* Fill in your pair programming log in the PairProgrammingLog.txt file!
* Your NetBeans project name must be PEX4\_Your\_Name\_Here. (Right-click on the project in NetBeans and change the name!) Zip this entire folder to PEX4\_Your\_Name\_Here.zip.
* Using the Webpost link on the left side of the course web page, submit only one file for the preliminary exercise, a zip file containing your entire NetBeans project.

1. Programming Exercise

For the programming exercise you will implement the remainder of the game, including GUI functionality to select human or computer players, start a new game, and convey current game information to the players.

Start by implementing the game for two human players. Recall that the weight for all edges is set to 1.0 when the graph is created. Edges selected by the Short player should be assigned a weight of 0.0 and cannot be selected again by either player for the remainder of the game. The Short player wins the game when the edges she/he has selected connect the start and end vertices. This can be determined from the graph when a shortest path algorithm returns a weight of zero for the path between these two vertices. Edges selected by the Cut player should be removed from the graph. The cut player wins when no shortest path between the start and end vertices exists. (The [**DijkstraShortestPath**](http://www.jgrapht.org/javadoc/org/jgrapht/alg/DijkstraShortestPath.html) class in [JGraphT](http://www.jgrapht.org/) will be helpful here.)

Once you have two human players working, add an algorithm for the Short player. One possible strategy is to determine the shortest path between the start and end vertices and select the first edge along this path whose weight is not already zero. A possible improvement to this may be to select an edge where one of the vertices has the lowest degree of all vertices along the path (i.e., this vertex could be easily disconnected from the rest of the graph).

Once you have the human players and the Short player working, add an algorithm for the Cut player. One possible strategy is to consider all edges in the graph, in turn, and remove one that either causes a shortest path to not exist or increases the length of the shortest path. If such an edge doesn’t exist, then remove a non-selected edge anywhere along the shortest path.

1. Programming Exercise Submission Requirements

* Fill in your name at the top of all provided source files!
* Fill in your documentation statement at the top of ShannonSwitchingGraph.java!
* Fill in your pair programming log in the PairProgrammingLog.txt file!
* Your NetBeans project name must be PEX4\_Your\_Name\_Here. (Right-click on the project in NetBeans and change the name!) Zip this entire folder to PEX4\_Your\_Name\_Here.zip.
* Using the Webpost link on the left side of the course web page, submit only one file for the preliminary exercise, a zip file containing your entire NetBeans project.

1. Helpful hints

When generating the random vertices, you may wish to add **Point** objects to an **ArrayList** as they are randomly created and add the vertices to the graph when finished generating the random points.

As edges are added to the graph, a **Comparator** may be useful to sort the vertices. An **ArrayList** of **Point** objects can be created from the set of vertices in the graph:  
 **ArrayList<Point> sortedVertices = new ArrayList( this.graph.vertexSet() );**  
and this ArrayList can be easily sorted using a **Comparator** and the sort method in the **Collections** class.

Instances of the **DefaultWeightedEdge** class do not exist independently of an instance of a graph, so your code will not directly call a constructor of the **DefaultWeightedEdge** class or directly call a method in the edge object to set its weight. Instead, your code will ask the graph object to add an edge and also ask the graph object to set the weight of an edge:

**DefaultWeightedEdge edge = this.graph.addEdge( v1, v2 );  
this.graph.setEdgeWeight( edge, 1.0 );**

The [**java.awt.geom.Line2D**](http://docs.oracle.com/javase/7/docs/api/java/awt/geom/Line2D.html) class contains methods that may be useful both when creating the graph and when determining which edges were clicked when playing the game.

When drawing the graph, it will look best if all edges are drawn first, then all vertices, and finally the start and end vertices are highlighted.

**Get started early!**

**CS 220 – PEX 4 Prelim – Grade Sheet Name:**

Points

Criteria Earned Available

|  |  |  |  |
| --- | --- | --- | --- |
| Appropriately creates start and end vertices | |  | **6** |
| Appropriately creates remaining vertices | |  | **10** |
| Appropriately creates random edges | |  | **14** |
| Appropriately draws graph in GUI | |  | **10** |
| **Subtotal:** | |  | **40** |
| **Adjustments** | **All Java code meets specified standards:** |  | **− 4** |
| **Vague/Missing Documentation:** |  | **− 2** |
| **Submission Requirements Not Followed:** |  | **− 2** |
| **Late Penalties:** |  | **25/50/75%** |
| **Total w/adjustments:** |  |  |

Comments from Instructor:

**CS 220 – PEX 4 – Grade Sheet Name:**

Points

Criteria Earned Available

|  |  |  |  |
| --- | --- | --- | --- |
| Appropriately determines when Short player wins | |  | **8** |
| Appropriately determines when Cut player wins | |  | **8** |
| Appropriately implements human Short player | |  | **10** |
| Appropriately implements human Cut player | |  | **10** |
| Appropriately implements a reasonable strategy for the Short player | |  | **12** |
| Appropriately implements a reasonable strategy for the Cut player | |  | **12** |
| Appropriately implements remaining GUI functionality | |  | **10** |
| Overall quality of software design | |  | **10** |
| **Subtotal:** | |  | **80** |
| **Adjustments** | **All Java code meets specified standards:** |  | **− 8** |
| **Vague/Missing Documentation:** |  | **− 4** |
| **Submission Requirements Not Followed:** |  | **− 4** |
| **Late Penalties:** |  | **25/50/75%** |
| **Total w/adjustments:** |  |  |

Comments from Instructor: