HW8: Programming with graphs

Due: 1:59AM, Thursday, November 13 through turnin
Total: 20pts

You may work on this assignment with one other student. A team of two members must practice pair programming. Pair programming "is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test." Both members in a team must read the article All I really need to know about pair programming I learned in kindergarten. This article can be downloaded to any computer that is in the Marquette network.

Preparation

Get familiar with GridWorld. The Advanced Placement GridWorld Case Study is a useful starting point for this assignment. Begin by downloading GridWorld code, as well as the Student Manual for your reference. Online resources (such as this YouTube video) can help you get started with setting up the GridWorld JAR file under Eclipse, in order to beginning running the sample projects. GridWorld already contains a variety of `Actor`s that can inhabit the grid: `Bug`s, `Flower`s, `Rock`s and `Critter`s. This assignment will only be concerned with `Rock`s, `Flower`s, and `Critter`s.

Read the graph related sections 9.3-9.6 and source codes in the textbook. Understand that a graph contains a set of vertices and a set of edges. Know how to insert vertices and edges to a graph, and how Dijkstra's single source shortest path algorithm. To correctly and efficiently complete this assignment, you will need to read the source codes `WeightedGraph.java` and `UseGraph.java`. Both files are included in the zip file that can be downloaded from the course home at D2L.

Understand how to use `PriorityQueue`. You can either use the `PriorityQueue` implementation provided by Java or use the implementation `Heap.java` provided by the textbook. If you use the former, use `add(E e)` to add to the priority queue and `poll( )` to retrieve and remove the minimum item from the priority queue.

Task overview

For this assignment, you will implement your own extension of an existing GridWorld classes to simulate the `GeekGecko`.

The bounded grid for this assignment will always begin with precisely two `GeekGecko`s on the board, an arbitrary number of `Rock`s, and no other `Actor`s.

- The `Rock`s are obstacles that occupy the squares in the bounded grid. `Rock`s do not move.
- The `GeekGecko`s are able to move. The two `GeekGecko`s try to meet and occupy two adjacent squares (in the same row, column, or diagonal). However, a `GeekGecko` does not move until it has already figured out the shortest path that leads to its counterpart. The shortest path to its counterpart should not pass through any `Rock`s.
    - The `GeekGecko`s will use Dijkstra's single source shortest path to find the shortest path to the counterpart.

Task specifics

Class `GeekGecko`:

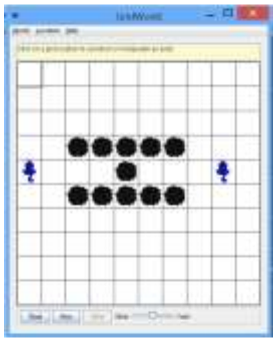
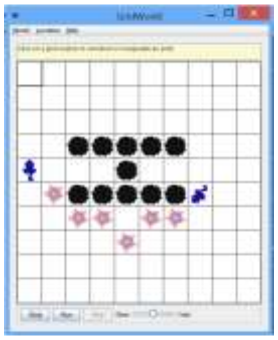
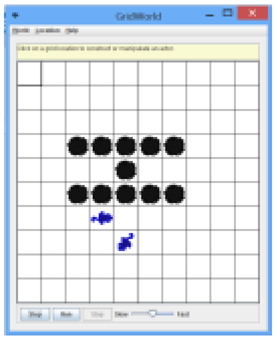
Create a new `Critter` type `GeekGecko` and place it in `GeekGecko.java`. A `GeekGecko` object detects its counterpart on the grid, figures out the shortest path to its counterpart, and moves toward it on each step following the path. Details include:

(1) The `GeekGecko` can move one square at a time, similarly as the existing `ChameleonCritter` example class (`GridWorldCode/projects/critters/ChameleonCritter.java`). Unlike `ChameleonCritter`, `GeekGecko` doesn't just randomly moves about the board, nor change its color to match adjacent `Actor`s.

(2) The `GeekGecko` will use Dijkstra's single source shortest path algorithm to find the shortest path to all other locations in the grid, and plants `Flower`s (a `Actor` subclass) along the shortest path to the counterpart.

(3) The `GeekGecko`s will eat on their way to meet each other.

The following three figures show some status of the board when `GeekGecko`s act.

 Initial board.	 A shortest path has been found, and one `GeekGecko` has moved one step.	 `GeekGecko`s have eaten the flowers and reached each other.

The `GeekGecko` class has only one public method `act()`, which is already implemented by the instructor. This method invokes some private methods and uses Graph. You will implement some of the private methods invoked by method `act()` indicated below. These methods have empty bodies in `GeekGecko.java`. Write the codes to complete these methods.

1. `private void populateVertices(Graph theGraph)`. This method uses each location in the grid to create a `Vertex` object and adds the resulting objects to the initially empty `Graph theGraph`.
2. `private void populateEdges(Graph theGraph)`. This method considers that the adjacent locations in the grid are connected by edges, and thus adds all edges in the grid to `Graph theGraph`. Such edges have a weight of 1. If a location is occupied by a `Rock` object, this location doesn't have edges to/from its adjacent locations.

The `act( )` method also uses `Graph` objects. You will implement class `Graph` and class `Vertex` to support `act( )`.

`Class Vertex:`

Design `class Vertex<T>` and place the implementation in `Vertex.java`. `Vertex` uses a type parameter `T` for type argument `Location`. `Vertex` implements `Comparable` interface. `Vertex` objects are comparable by instance variable `dist`. A `Vertex` object has three instance variables:

1. `private int dist`. `dist` represents the distance to this `Vertex` object from the single source `Vertex` object. `dist` is initially set to a maximum value of 200 (not possible on a 10 by 10 grid if there is a path between the source object to this object) for all newly created `Vertex` objects.
2. `private Vertex<T> parent`. `parent` represents the `Vertex` object that directly leads to this `Vertex` object along the shortest path from the single source `Vertex` object.
3. `private T loc`. `loc` represents the location in the grid. It is a pair of (row, column). The valid ranges of row and column are [0, 9] inclusive in the grid.

You will also implement constructor, accessor and modifier methods for all three instance variables, and `compareTo(Object o)` methods.

`Class Graph:`

Design `class Graph<T>` and place the implementation in `Graph.java`. `Graph` uses a type parameter `T` for type argument `Location`. A `Graph` object has four instance variables:

1. `private int numVertices`. The current number of vertices in the graph. A newly created `Graph` object has no vertex. `numVertices` is incremented by 1 when a `Vertex` object is added to the graph.
2. `private int maxVertices`. The possible maximum number of vertices a `Graph` object can have.
3. `private Vertex<T>[] vertices`. The array of vertices in the graph. This array is empty for a newly created `Graph` object and populated with `Vertex` objects that are added to the graph.
4. `private int[][] edges`. The adjacency matrix for the edges in the graph. `edges[row][col]` has a value of 1 for an edge between `vertices[row]` and `vertices[col]`, and a value of 200 if no edge exists.

A `Graph` object implements several public methods. A template `Graph.java` is provided. You will implement the methods with empty bodies.

Test your design and implementation.

A driver program our GeekGeckoTester.java is provided. Do not change this driver and GridWorld source codes. Fresh copies will be used to test the submissions. Points will be awarded solely based on the shortest paths found by submissions and how the results match the solutions.

A battery of testcases will be used to test submissions. As you might guess, test scenarios will consist of placing two `GeekGecko`s at various locations on the board, with diverse patterns of `Rock`s between them.

You should create many of your own testcases, but we will only evaluate your `GeekGecko.java` source file.

Here are four typical testcases:
02-OneSpace.test
01-Barrier.test
06-BlindAlley.test

Here are GeekGecko.gif image files. Placing these in the same directory with your .class files will allow the `GridWorld` system to use the images in your display.

Submission.

Submit `GeekGecko.java`, `Vertex.java`, and `Graph.java` source files only to turnin system. You can submit multiple times but only the last submission will be saved by turnin. If you work in a team of two, always let the same user submit the file. The lower grade from two submissions will be used if each team member submits.

Pay attention to the following details for successful submissions and tests.

(1) Name your class precisely -- spelling and capitalization must match this assignment specification for you to pass any of our testcases.
(2) Do not modify any of the other `GridWorld` classes -- or, at least, do not make your solution depend upon any modifications you make to other `GridWorld` classes. Your `GeekGecko.java` source file will be graded in a fresh copy of `GridWorld`, with our own test framework, by the TA-bot.
(3) Make certain that your code compiles correctly -- no points can be awarded if your submission will not run to pass any of the testcases.
(4) If you put any debugging `System.out.println()` statements in your code during development, please comment these out before submission. Stray output from your program will confuse TA-bot.