CS211 Spring 2007
Assignment 5 — Pacman
Due Thursday, 3 May 2007, 11:59:59pm

1 General Instructions

In this assignment, you will develop a Java implementation of the famous interactive game Pacman. This assignment will involve GUI programming, threads and concurrency, and graphs.

This is the most extensive assignment yet. There are two common required exercises, namely the implementation of Dijkstra's shortest-path algorithm and its application in the Pacman game to supply artificial intelligence for the ghost behavior. In addition, there are several optional exercises that you may choose from. Finally, we will have a contest for the best game as judged by the course staff, with a prize to be awarded at the final exam. This assignment should be a lot of fun, but it will also be a lot of work, so don't wait until the last minute to get started.

As before, you may work in pairs. Please follow the instructions on the website for selecting a partner. You must create a group in CMS for yourself and your partner.

2 Overview

2.1 Background

Pacman is a traditional arcade game that was first produced by a Japanese company Namco and first appeared on the scene in 1979. If you have never played Pacman before (this is a bit like saying if you have never heard the Beatles before ...), there is a good online version available at
http://www.thepcmanwebsite.com/media/pacman_flash/.
Our version is modeled on the original.

2.2 Getting Started

Download an unpack the supplied archive `A5Release.zip`, which contains the assignment code, and the file `A5Solution.jar`, which contains our solution. Play with the online version and/or our version a bit to get a feel for the game. You can run from the command line using `javaw -jar A5Solution.jar`, or in Windows, just double-click on the `.jar` file. Use the command line option `-s` to run silently if the sound effects are not to your liking.

2.3 Structure of the Application

When you unzip `A5Release.zip`, the first thing you will notice is the division of the code into several folders. Here we have used the Java package structure to organize the numerous classes that are needed in a logical way. The `game` package contains the code for the game engine that handles all the actor, map and player functionality. The `gui` package creates the pacman application and handles drawing and user input. Of particular interest for you is the `graph` package. In here you will find the `Graph` interface that you must implement.

From a code standpoint, the game runs very much like a real game would. Within the `GUI` class, there is an `Animator` class that runs on a separate thread. It contains an infinite while loop that repeatedly calculates the time passed since the last loop and then updates the unit locations, checks for collisions, and repaints the background and actors. Because we never really know how fast this game loop will run, we use the time difference since the last phase to move the units based on the time that has passed. If one iteration takes longer, we move the units farther.

The map used by the game is created based on the text file in `bin/resources/maps`. The text file specifies what type of cell should be placed in each location, including walls and non-navigable areas. The classes `PacmanMap` and `Background` create the necessary images and draw them when called. The `PacmanMap`

class knows where actors should be allowed to go and where there is a wall, and it stores this information in a `Graph` (which you will be writing).

Each iteration of the game loop requires a check to see where the units should be moving. There are two classes that handle this, namely `LocalPlayer` and `AIPlayer`. The `LocalPlayer` class is event-driven and implements `KeyListener`. It is alerted each time the user pushes an arrow button. When this occurs, it uses the `addDirection` function in the `Pacman` class to enqueue the next direction that the pacman should travel. The `AIPlayer` class is where you will use your graph code to find the shortest path to the pacman, and then give each ghost directions to the pacman based on where they are.

The `Game` class is the glue that holds everything together. It interacts closely with the `GUI` class, handling key and timer events generated by the user and the program. It translates these events into commands for all the other classes in the game.

3 Graphs, Shortest Paths, and Dijkstra's Algorithm

Purpose: Graphs as a data structure, implementation of graph algorithms, Dijkstra's shortest path algorithm
Estimated time: 25 hours
Points: 75

In this exercise, you will build a graph representing the navigable portion of the board and implement Dijkstra's shortest path algorithm.

3.1 Building The Graph

The navigable areas of the game board can be represented as an undirected graph whose nodes are the navigable cells and whose edges connect adjacent pairs of cells.

The classes `Node`, `Edge`, and `Path` are provided. You need to implement the methods in the file `MapGraph.java` using these classes.

3.1.1 The `Node` Class

All nodes will have a unique number `id`, along with their two position coordinates `x`, `y`. Each node will also have a `Vector` of its neighboring nodes.

3.1.2 The `Edge` Class

Similarly, all edges have a unique number `id` (which is not related to any node's `id`), two `Node`s which are its endpoints, and a `weight`, which can be interpreted as the distance along that edge.

3.1.3 The `Path` Class

The `Path` class consists of a `Vector` of nodes. Whenever you create a new `Path` object, it is initially empty. You can then add nodes to this path (in order) using the `appendNode(Node v)` method. The path always starts at the beginning (`currentProgress` = 0) and will advance `currentProgress` by 1 every time you call the `advance()` method.

The class `Path` implements the `Iterable` interface. This means it has a method `iterator()` that will return an iterator of `Node`s starting from wherever `currentProgress` is all the way to the end of the path. The method `reverse(Path p)` will return the path in reverse order. This newly returned `Path` will have its `currentProgress` field set to 0.

3.1.4 The `MapGraph` Class

The `MapGraph` class will implement the interface `Graph`, which defines all the methods you need, along with the asymptotic complexity we expect to see. You must implement these methods. If your algorithms exceed the given runtimes, they will have bad effects on the speed of the game.

You may implement utility classes and helper methods to assist in your methods. You are also allowed to import and use anything from the Java API.

3.1.5 Access Methods

`public abstract int addNode(int x, int y)` This method will take a position `x`, `y` and add a `Node` at that position to the data structure(s) containing your nodes, as long as one does not already exist. If there is already a node at that position, it will not add the new node, and will just return –1. Otherwise it will add the node and return this node's `id`. Note that this function should take at most $O(\log n)$ time, where n is the number of nodes in the graph. This means just keeping all your nodes in an `ArrayList` is not going to be enough, because searching through an `ArrayList` to check if the element is already there would take $O(n)$ time in the worst case.

`public abstract Node getNodeById(int id)` It should be possible to retrieve a `Node` given its `id` in constant ($O(1)$) time. This method returns `null` if the `Node` does not exist.

`public abstract Node getNodeByCoord(int x, int y)` This method is very similar to `getNodeByID`. This method is expected to run in at most $O(\log n)$ time. Again, this means that you do not have enough time to go through a list of nodes and check each coordinate. This method should return `null` if the `Node` does not exist at that point.

`public abstract int nodeCount()` This method returns the number of nodes currently in your graph. It should run in constant time.

`public abstract int addEdge(int nid1, int nid2, double weight)` This is very similar to `addNode`. It will take in two node `id`s and a `weight`. It will then check to see if this `Edge` already exists. If it already exists, or if adding this edge would form a self-loop (an edge whose two endpoints are the same `Node`), this method will return –1, otherwise it will add the `Edge` and return the `id` of the newly created edge.

`public abstract Edge getEdgeById(int id)` Similar to the corresponding method for `Node`s.

`public abstract Edge getEdgeByEndpoints(Node v1, Node v2)` This method will return an `Edge` with endpoints `v1` and `v2`. This is expected to run in at most $O(\log n)$ time, so again, linear search is out of the question. The method should return `null` if there is no such `Edge` in the graph.

`public abstract int edgeCount()` Returns the number of edges currently in the graph. It is expected to run in constant time.

3.1.6 Graph Algorithms

`public abstract double pathWeight(Path p)` This method will take a `Path p` and determine the weight of the path as a whole. This is the sum of the weights of all the edges along the path. This should run in time $O(n \log n)$. This will return `INFINITY` if the path is empty. The value `INFINITY` is a predefined constant defined in the `MapGraph` class.

`public abstract Path shortestPath(Node src, Node dst)` This is where all the hard work is. Given a source and a destination, this method should use Dijkstra's algorithm to find the shortest path from `src` to `dst`. It should return `null` if there is no such path. This should run in time $O(n \log n)$.

3.2 The AI

3.2.1 Moving the Ghosts

Now you will use your implementation of Dijkstra's algorithm to help the ghosts capture Pacman. In `AIPlayer`, call your implementation of Dijkstra's algorithm from the `run()` method of the `MoveGhostTask` inner class to determine the shortest path from each ghost to Pacman. Each ghost will take the shortest path to Pacman unless it is vulnerable or recovering. In the case that the ghost is vulnerable or recovering, the ghost should try to get away from Pacman by taking the opposite direction from the shortest path direction. Otherwise, it should move in the direction of the shortest path to Pacman.

To move the ghost in some direction, use the method `addDirection(MovementDirection nextDirection)` from the `Ghost` class. To get the next direction from the path, start with the location of the first node in the path and the ghost's current location, use the method `getDirection(int x, int y)` from the `PacmanMap` class, where `x` and `y` are the differences between the `x`,`y` coordinates of the first node in the path and the ghost's current location.

You will want to change the timing of the calculation of these shortest paths. Right now the ghosts are programmed to move in random directions, and to choose a new random direction every 2 seconds (2000 milliseconds, as specified by the value of `calculationInterval`). Once you get your AI code working, you will want to shorten the time between successive shortest-path calculations. In our solution, we have `calculationInterval` set to 50 milliseconds.

3.2.2 PacmanMap

In `PacmanMap`, there is some code that needs to be uncommented in the constructor after you finish implementing `MapGraph`. The code constructs the graph using the `MapGraph`.

```
/*
 * Creates the graph representation of this map.
 */

/*REMOVE THE FOLLOWING COMMENT and make the code active after you write the graph class */
```

4 Enhancements

Purpose: To have fun and exercise your creativity
Estimated time: The sky's the limit
Points: 25+

In addition to implementing Dijkstra's algorithm and the AI for ghosts, we ask that you enhance the game by adding at least one extra feature. This part is designed for you to have fun with the project while experimenting with the different features of the Java language.

Dijkstra's algorithm and the AI are worth 75 points. This section allows you to make up the remaining 25. Feel free to do any of the enhancements we list and as many as you like, but keep in mind that the max grade for this assignment is 100 points. Points in excess of 100 will be counted as bonus points. Those might be important at the end of the semester if you are on or slightly below a certain letter grade cutoff.

The following list gives enhancements that we have thought of and the points that they are worth. You may also come up with your own ideas. If you come up with something that you would like to do that is not on the list, please contact one of the instructors or TAs assigned to this project to negotiate the number of points you will get for it. The number of points should be in line with those listed in the list below.

We will award a prize at the final exam for the best game as judged by the course staff.

4.1 Keep Score - 10 points

Assign point values to actions in the game: eating a ghost, eating dots, eating cherries (if you implement them), etc. Keep track and display the score somewhere in the game itself, but do so gracefully. The score should not take over an important part of the game screen, nor should it be in a place where it is difficult to find.

4.2 Lives, game over screen - 10 points

Right now, Pacman has infinitely many lives and the game ends when the user quits the application. Implement Pacman to have a fixed number of lives (say 3). Everytime Pacman dies, update the number of remaining lives (you should keep this number visible somewhere on the game screen). If Pacman is out of lives, display a game over screen and allow the user to start a new game.

4.3 2 Pacmen - 20 points

Add another Pacman to the map so that a single user controls both at the same time. Bind another set of keys (say W,A,S,D) to move the second Pacman. The second Pacman should interact in exactly the same way as the original one and follow the same game rules (i.e., it eats dots, is vulnerable to ghosts, can get powerup, etc).

4.4 Levels - 20 points

Add extra levels to the game. When Pacman completes a level (by eating all the dots), go on to the next level. The AI for the ghosts should also get better with every new level. One way to do this is to increase the attention span of the ghosts. If you do something more involved or more interesting, we may be inclined to award more points. Implement a minimum of 3 levels to get the full points.

4.5 Scaling AI of ghosts - 10 points

If you do not wish to create extra levels, you may still implement a harder AI for the ghosts (you can ask for a "difficulty level" before the start of the game in a dialog box). If you create an AI that is more involved than the improvement we described in Section 4.4, then we will be inclined to award more points.

4.6 High Scores - 10 points

Save scores to a file. Keep track of the top 10. Display them after the end of the game (so you need a notion of "end game").

4.7 Splash Screen - 10 points

If you implement High Scores or Scaling (Sections 4.6 or 4.5), you may add a splash screen at the beginning of the game. It should allow players to start a new game, see high scores or set the AI of ghosts and possible set other options that you may have implemented. Feel free to get rid of the bottom bar with the start button and have the game cover the whole window.

4.8 Color Schemes - 10 points

Allow a user to select different color schemes. The color schemes should change the color of the background, walls, items in the game and ghosts and pacman themselves. Perhaps place a photo in the back instead of a background.

4.9 New Items - variable

We have subdivided items into two categories: ones that take effect immediately and ones that you can save for later use.

The immediate items are those that if Pacman (or Ghost) touches, immediately perform their effect (example is the powerup). You will get 5 points for the first item you implement and then 2 more for any subsequent ones.

Immediate items:

- Ghost Freeze - all ghosts stop in place if Pacman walks over this item

- Turbo Speed - Pacman moves very quickly around the board
- Slow Speed - Pacman moves very slowly around the board
- Slow Ghosts - all Ghosts move very slowly around the board
- Turbo Ghosts - all Ghosts move very quickly around the board
- Warp - Pacman warps to a random location on the board
- All tokens reappear - all eaten dots reappear and Pacman has to eat them again
- Cherry - just like in the original Pacman

Saveable items are those that a Pacman "picks up" and can use at a later point. You need to display those items to the user in some way and bind a key (such as the spacebar) so that Pacman uses the next item (you may also want to bind another key to "destroy" or get rid of an item already stored). These will gain you 7 points for the first and 2 more for each additional one.

Saveable items:

- summon a friendly ghost - use your imagination
- mines - drop a mine. If a ghost goes over it - BOOM!
- turn into a Ghost - turn yourself into a Ghost. Other ghosts stop chasing you and can't hurt you if they touch you. On the other hand you can't eat any dots. This effect should be temporary (and could be implemented as an immediate item).

You may add any other items here that you think of for the same amount of points as described above.

4.10 Pacman AI - 50 points

Implement an AI for Pacman (so essentially the computer plays itself). The Pacman should try to eat all the dots while avoiding Ghosts. For this one, simply picking a random direction to walk in will not work. You must really put some thought into this and think about how to make Pacman try to avoid the Ghosts. You must also describe the algorithm you used in detail so that it convinces us you put significant effort into this part. Partial credit will be awarded, but it will be based on subjective judging.

4.11 3D Pacman - 50 points

Use your imagination.

4.12 Network Pacman - 100 points

Let a user control Pacman and networked users control Ghosts. Try to balance out the game so that it is not too easy to capture the Pacman. Borrow the code from Assignment 4.

4.13 "All your base are belong to us!" - 1 point

Test out your humor skills by throwing in an "All your base are belong to us" reference somewhere in the game.

5 What to Submit

Please read this section carefully!

We will first test just your Dijkstra's algorithm code and the ghost AI plugged into our solution code. Then we will run your whole code to grade the enhancements. Please submit the following files in CMS:

- Exercise 3: `Exercise3.zip`
- Exercise 4: `Exercise4.zip`

Do not submit any other files.

The file `Exercise3.zip` should be a `.zip` archive containing at least `MapGraph.java`, `AIPlayer.java`, `PacmanMap.java`, and any other `.java` file you create or modify. If you include any other files besides the three mentioned, please also include a plain text file `readme.txt` describing how and why you modified them.

The file `Exercise4.zip` should be a `.zip` archive containing *all* of your source code (including any files already included in `Exercise3.zip`, so you essentially submit them twice), along with a file `readme.txt`.

The reason we are asking for *all* your files in `Exercise4.zip` is that you may want to submit basic versions for the Dijkstra problem and highly modified versions for your enhancements.

The file `readme.txt` in `Exercise4.zip` should be in plain text format and should have the following information:

1. how to compile and run the game (write "as provided" if you did not modify this);
2. features that you added, along with a description of how you implemented them.

In your description, please describe what modifications you made to the original code to get the features to work and the algorithms that you used to get those features to work (you will not get credit if you do not explain the algorithms). Here, we are *not* looking for lines of code that you modified, but rather which classes you modified and what those modifications were. For example:

"We modified the game class to have a global variable that keeps track of the current score. We also modified each of the items to have a score variable for that item. Whenever an action is performed on one of those items, we update the game score to reflect this. Finally, we modified the GUI to display the score in the bottom right hand corner."

We hope you enjoy doing this assignment as much as we enjoyed putting it together!