Overview

I have given you much of the code you need for this assignment. Below you will find a higher level explaining of the pieces and how they work together, as well as what you need to do.

Path Planning

Given a terrain map, a starting position, and a goal position it is the path planning algorithm’s job to find a good path from start to goal. The terrain map consists of individual tiles different land formations (roads, grass, swamp, mountains). Each type of land formation has a particular cost. That is, roads are quicker to travel over than grass, grass is quicker than swamp, etc. It is these weights that will determine the solution found. In terms of coding you should have a class `TerrainMap` which holds onto a 2D array of TerrainTiles. `TerrainTile` is an interface that sits above the concrete classes of `Road`, `Grass`, `Swamp`, and `Mountain` in the hierarchy. In terms of UML class diagrams the structure is:

The TerrainMap needs to get the actual terrain information from someplace. I have provided a simple text file with the terrain information. The contents of the file look like:
Where the first line is the number of rows and columns in the map. Each successive row contains a rows worth of tile information (r – road, g – grass, s – swamp, m – mountain).

Once you have a TerrainMap, you need a way to display it on the screen. This will be handled by the TerrainPanel class. This class extends JPanel (so paintComponent can be overriden) and also displays itself in a JFrame. Obviously this class will be in charge of the display on the screen. It is also in charge of handing the user input via event handlers.

There is also a class called Avatar which holds the information about the object we are going to be moving over the terrain. It basically keeps track of its current position and can draw itself on the screen. But it also has a method to calculate a path to a new position and then move over that path. The path planning is all done in the PathFinder class which you will be writing. And most of the method is done it a thread so it can run from within the event handler.

Now we are finally to the point where we can implement our path planning algorithm. Note that for a TerrainMap of any size that there are tons of potential paths. We certainly don’t want to enumerate all potential paths and then decide which one is the optimal one – that would take exponential time. Thus, we need a way to prune off as many paths as possible so that the code will run is a reasonable amount of time. To do this we will use the A* algorithm. A* is nothing more than a best-first search with a bound that consists of the cost so far plus an underestimate of the remaining cost which stops as soon as it finds the first solution. Thus, we will need a class to handle the general A* algorithm and another to handle the specifics of our particular terrain problem. These classes should be called: PathFinder and PathState.
PathFinder should have a constructor that takes in the terrain map as well as the terrain panel. The map will be used in the process of path planning. The panel, however, is simply there to enable you to draw the explored paths on the screen. Each time you get a new potential state to explore you can add it to the A* state of the panel, tell the panel to repaint, and then sleep for a little bit so you can see the processing take place.

Most of the work will be done in a method called `findPath` that is in charge of actually running the algorithm to find the List of Points along the optimal path from the starting point to the ending point. This is effectively a best-first search framework. You do, however, need to make sure you don’t expand locations you have already looked at. That is, unlike some of the examples we have seen before, you can end up with cycles in your “tree” if you are not careful.

PathState is in charge of producing the children and computing the bounds, etc. For this assignment we restrict the units to only move sideways and up/down (i.e. not diagonal). Thus, there are up to 4 children for each state. As for the bounds (that will control the next node expanded in the A* algorithm), we want to use the cost so far plus an underestimate of the remaining cost. Use a Manhattan distance times the least weight tile cost as a simple under estimator. You may hardcode the lowest weight tile cost. I have provided the drawing method as well as method signatures for the other methods you will need.

**Submission**

You are to turn in:

- A printed single page (max) document describing any problems that your code might still have if it is not complete. Or if it is complete, possible places for improvement in your code.
- Submit a jar file of your entire package so that I can run your program on my machine.

You may work with a partner.