Declan Urbaniak-Dornstauder
3716081
COMP 452
Assignment 2

Game 1 – A* Simulation

Overview

The application provides a simple interface to edit and search a 16 x 16 grid. To edit the grid, select one of the tile type icons in the toolbar to the right of the grid, then click a tile on the grid to change it to the selected type:

 Default terrain – cost 1

 Grassland terrain – cost 3

 Swamp terrain – cost 4

 Obstacle

 Start/character

 Goal

Obstacle tiles can be removed individually by clicking an obstacle on the grid while the obstacle type selector is selected. Because only a single start tile and a single goal tile can (and must) exist on the grid, selecting a new position for these tiles will simply change the position of the tile that already exists. Lastly, the *Clear* button sets all terrain on the grid to default terrain and clears all obstacles.

To start the path finding animation, press the *Start* button. You may wish to select the speed of the animation beforehand by selecting one of four radio buttons under *Speed*. As the animation plays, the character sprite will move across the grid as it determines the shortest path to the goal tile. Orange tiles represent visited tiles. If and once the character reaches the goal tile, a shortest path is represented by the sequence of tiles coloured blue, and the length of the shortest path can be seen in the bottom right corner under *Length*. Note that the path length includes the cost of the terrain at both the start and goal tiles.

To reconfigure the grid after the animation has finished, press the *Reset* button. The grid can only be edited after resetting. The grid can also be reset at any time during the animation.

Logic and Design

The game is implemented as a JavaFX application that uses an MVC-like design pattern to structure its various classes. In this case, there is a single model class containing the game logic and data that communicates with various view classes through a publish-subscribe mechanism. A simple controller class is used to handle the events generated in the view classes and update the model. I have organized the project directory to reflect the components of MVC, where most of the classes are organized into separate *model/* and *view/* sub-directories. The rest of the classes, including the controller class, are not contained in a sub-directory:

- *model/*
  - *AstarNode.java* – helper class to represent grid tiles as nodes with helpful properties
  - *Model.java* – main model class that contains the majority of the application's logic and data

    - *PublishSubscribe.java* – interface used by view components to receive data from the model
- *view/*
    - *ControlMenu.java* – parent component of all components that make up the bottom-right toolbar
    - *GridTile.java* – component representing a single tile of the GridView
    - *GridView.java* – grid of GridTile components, making up the entire visual grid
    - *StartResetButton.java* – component to start and reset the path finding simulation
    - *TileMenu.java* – parent component of TileSelector components, representing the tile selection menu
    - *TileSelector.java* – component to select a given tile type to modify the grid
- *App.java* – class to initialize components related to the application itself (stage, scene, root, etc.)
- *Controller.java* – event handling class used by view components to perform actions in the model
- *Main.java* – entry point class required to build JavaFX .jar files

The most important logic is contained in *Model.java.* This class contains all the logic to setup, update, and perform a search on the grid. The other classes, while managing their own states, only do so in a way to visually reflect the state of the model, and are therefore dependent on it.

The path finding A* algorithm is implemented in the model in the function *AstarSearch*() with the addition of a few helper functions. This function performs a breadth-first search on the grid to determine the shortest path from a starting node to a goal node using both the heuristic cost (hCost) and traversal costs (gCost) of the visited nodes. In this case, the heuristic cost is simply the minimum number of nodes a node is away from the goal ignoring the types of terrain on the grid. This is calculated in the *calculateHCost*() helper function used in the path finding function.

Each node of the graph is of type AStarNode, which is a helper class used to keep track of a node's fCost (hCost + gCost) and previous node throughout the traversal. These properties simplify the process of polling the next best node in the breadth-first search and retracing the shortest path if and once the goal node is reached.

Game 2 – Finite State Machine Stimulation

Overview

This application simulates the behaviour of AI ant characters in a grid. This behaviour is based on the state of an ant according to simple state machine and the type of terrain an ant is positioned on as it navigates grid:

 Default terrain

 Poison terrain

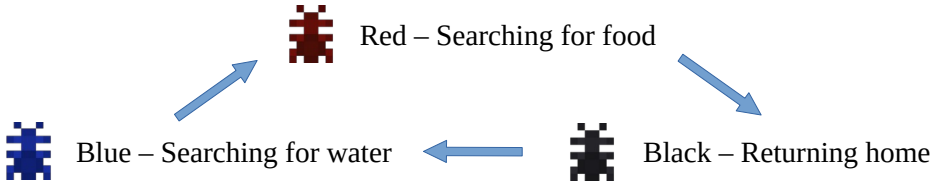
 Food terrain

 Water terrain

 Home terrain

The position of each tile in the grid is randomly determined when the simulation begins, but the tile types generate at predictable ratios of the total number of tiles: 2.5% water tiles, 2.5% poison, 0.5% home, 1.5 x # starting ants food tiles (max. 50%), and the remainder default.

Ants can be in one of three states at a time. When an ant spawns, it will be in the searching for food state, where it will move randomly to adjacent tiles until it traverses to a food tile. Upon reaching a food tile, the ant will transition to the returning home state, where it will path directly to the nearest home tile. Once an ant reaches the home tile, it will transition to the searching for water state. This state functions identically to the searching for food state, but an ant must reach a water tile instead of a food tile. Upon traversing to a water tile in this state, the ant will return to the searching for food state. During the simulation, the colour of an ant will change to reflect the state it is in:



When an ant reaches a food tile in the searching for food state, this tile will turn into default terrain and a new food tile will appear at random tile in the grid to replace the one that was consumed. When an ant reaches a home tile in the returning home state, both a new ant and a food tile will spawn on/replace random default tiles in the grid. In any state, an ant will die and be permanently removed from the grid if it traverses to a poison tile. Be aware that ants are not very patient – if a poison tile happens to be part of the shortest path to a home tile when an ant is returning home, an ant will still die when it travels to the poison tile.

To begin the simulation, enter the starting number of ants in the text field under *Initial number of ants*, then click the *Start* button under *Simulation control*. You may wish to change the dimensions of the grid to one of the three presets beforehand using the radio buttons under *Grid dimensions*. Note that the number of ants on the grid at once cannot exceed ~33.3% of the total number of tiles. This was done in an attempt to balance the ratio of ants and tiles when the grid is at low capacity.

After the simulation has started and is running, you can pause the simulation in its current state and resume it using the *Pause/Play* button under *Simulation control*. If you wish to reconfigure the simulation at any time during its runtime, you must reset it using the *Reset* button.

Logic and Design

Like the previous game, this game is also a JavaFX application and uses the exact same design pattern. Likewise, the project directory is also structured the same way:

- *model/*
  - *Ant.java* – data class to represent each ant on the grid
  - *GridPosition.java* – helper class to represent grid positions
  - *Model.java* – main model class that contains the majority of the application's logic and data
  - *PublishSubscribe.java* – interface used by view components to receive data from the model
- *view/*
  - *AntNumberEntry.java* – component to input the starting number of ants
  - *ControlMenu.java* – parent component of all components that make up the right toolbar
  - *EnvironmentGrid.java* – grid of EnvironmentTile components, making up the entire visual grid

    - *EnvironmentTile.java* – component representing a single tile of the EnvironmentGrid
    - *GridDimensionToggle.java* – component of radio buttons to select the dimensions of the grid
    - *SimulationControlButtons.java* – component of buttons to start and generally control the simulation
- *App.java* – class to initialize components related to the application itself (stage, scene, root, etc.)
- *Controller.java* – event handling class used by view components to perform actions in the model
- *Main.java* – entry point class required to build JavaFX .jar files

The *Model.java* class contains all the logic to run the simulation, including the ability to generate the environment, start the animation, and retrieve the next move from each ant. Specifically, the model only tells an ant to move, then retrieves its movement – 'how' an ant should move is handled internally by the ant in the *Ant.java* class. Based on it's current state defined by the enum, *AntState*, an ant will either execute *roam*() or *path*(). The former chooses a random adjacent tile from an ant's current tile to move to. The latter uses a breadth-first search to determine a shortest path towards the nearest home tile from an ant's current position and adds the tiles of this path to an ant's queue of next moves. The model then coordinates the movement of the ants to determine if a move to a tile is possible. If it is, the grid and Ant object will update. If not, an ant will simply stay in place for a frame, then attempt to move again on the next one. Note that only a single ant can occupy a tile at a time. For this reason it is best to use a smaller number of starting ants when running the simulation.

The state machine is implemented in *Model.java* in the *moveAnts*() function. This function uses a switch case on the current type of terrain an ant is positioned on after a frame update, then determines what should happen based on the current state of an ant. There is not much to be said about this function, as it follows the same logic describing how an ant should behave. In code, this means updating its *AntState* value and executing various functions. The exception is the addition of the *isPaused* flag for an ant. This is used as an intermediary state to pause an ant in place whenever some action is to occur, and exists solely for visual clarity. For example, without this flag, you would not see an ant move to a poison tile and die after a frame update – it would just disappear from the adjacent tile it was previously positioned at.

Issues/bugs

- There is currently an issue that occasionally causes ants to skip a tile when they are moving or not move at all after a frame update.
- The ants (I'm not sorry).

Instructions to run the games

- I have provided two zip files, *a2_1.zip* and *a2_2.zip*, both containing the project directories for both games respectively. I have also included the executable jar files for both games.
- The jar files can be ran via the command line with *java -jar jarname.jar* or by simply double clicking the executable if your OS allows it.
- Otherwise, the project can be compiled and ran using *./run* in the project's root directory. You can also rebuild the provided jars using *./build* in their root directories, and the newly built jars will be in their *out/* directory.