Running the Game

The game was built on NetBeans 8.2 and compiled through this IDE. I have built and included the .jar file for direct execution.

Overall Theme

The two games have the same core engine in that the grid is initialized with a 2d array, where each item corresponds to a node that contains features for that grid. These features include the cost to move to that node and whether there is an item on it. The first game allows the user to build obstacles, map tiles with various travel costs, and the start to end that the main character, the ant, must travel to. The second game allows the user to specify the starting number of ants within a colony. The map generates a random number of food, water, and poison. Each ant may pick up the food to transport back home, drink water, or be poisoned. Both games use the core pathfinding mechanic via the A* algorithm.

A* Pathfinding Algorithm

The A* algorithm is similar to Dijkstra's Algorithm in that it will keep a set of visited node starting with the first node. For the current node, it looks at each unvisited neighbor and finds the cost to visit. When all neighbors have been visited, the current node is removed from the unvisited set. The node with the smallest tentative cost that is unvisited is set to the current node. The algorithm repeats to find the unvisited neighbors until the ultimate shortest path is found.

A* algorithm takes it a step further in that it uses a heuristic where it also considers the distance from the current node to the target node. The idea is that it is generally a better idea to go towards the target node first to test out the tentative cost of each branch in the search. There are several ways to find this heuristic. In this case, the distance is found by finding the absolute distance of the x and y values between the node and goal node. This value is multiplied by its movement cost. This is shown in the 'Node' class by calculating heuristic 'h':

```
public void setH(Node goal) {

        h = (Math.abs(getX() - goal.getX()) + Math.abs(getY() - goal.getY())) * MOVEMENT_COST;

}
```

The G value is the cost from parent node to the current node. It is the cost of the parent plus the movement cost to the node. This is simply set in 'Node' class by calculating 'g':

```
public int calculateG(Node parent) {
        return (parent.getG() + MOVEMENT_COST);

}
```

Finally, the F value is simply the H value plus the G value to obtain an overall cost. Neighbor nodes on the open list with lower F values have priority in the search algorithm before being placed in the closed list. This is described in the 'Node' class as

```
public int getF() {

        return g + h;

}
```

Illustrating the Path Evolution in Game 1

The first game lets the user see how the search evolves before moving the ant to the goal. This is illustrated in the 'Map' class using an ArrayList of a List of Nodes:

```
        ArrayList<List<Node>> listOfLists = map.getListOfLists();
```

Each item in the ArrayList contains a path within each step in the path evolution. Each List contains all the Nodes within that path. To illustrate each path, each item in the ArrayList is iterated through and each Node in the List is iterated so that the relevant Nodes are highlighted. This is the core method used in the 'Map' class:

```
        for(Node curr : evo){

                ...

                ...

                ...

                  for(int i=0; i<listOfLists.get(counter).size(); i++){

                          ...

                    map.getNode(currX, currY).setShowPathEvo(true);

                          ...

                  }
        counter++;

    }
```

In the 'Node' class, each path that is to be highlighted to show the current path evolution is checked via 'showPathEvo' within the drapMap() method in 'Map' that renders the map:

```
if(nodes[x][y].isShowPathEvo()){

    int alpha = 60; // 25% transparent

    Color myColour = new Color(0, 0, 0, alpha);

    g.setColor(myColour);

    g.fillRect(x * 32, y * 32, 32, 32);

  }
```

Movement Mechanic

As stated, each node corresponds to a cell in the grid of the map. With a 16x16 map, each cell has 'x' and 'y' values between 1 and 16. When the map is rendered, each cell has its width multiplied by 32 to fill the grid. The 'Player' class, which corresponds to the ant, also has an 'sx' and 'sy' in addition to the 'x' and 'y' values. The 'sx' and 'sy' values correspond to the values in between the exact center of each cell. Overall, these values are taken by finding the remainder of the map position (i.e. x*32). This is kept track of in both games by using the walk() and fix() methods within the update() method for the game loop:

```
public void update() {

        if (fixing) {

                fix();

        }

        if (walking) {

                walk();

        }

}
```

Here, 'fixing' and 'walking' are booleans. These booleans are kept track of when a path has been found for the ant to follow:

```
public void followPath(List<Node> path){
        this.path = path;
        if (walking){
                fixing = true;
                walking = false;
        }
        else{
                walking = true;
        }
}
```

The fix() method aligns the ant to the exact center of the grid. If it is already aligned, it is not used. This is used more in the second game when the ant finds the food and must return it to its home.

The walk() method finds the next Node in the path, which is stored as a linked list. The 'sx' and 'sy' and adjusted by finding the next Node's coordinates. Once that Node is reached, the next Node in the linked list is found and the walking continues until the goal is reached.

The mouse listener is used within the 'Game' class. Booleans are used to see if the player clicks on the buttons on the right of the screen. These check when the user clicks on the tile to change its type and its subsequent cost. The same goes for the second game where the player determines how many starting ants should be used before starting the game.

Finite State Machine

The second game makes the ant start off moving randomly until they find the food. Once found, the ant picks up the food and returns it home, where another ant is spawn. The ant then goes on to move randomly until it finds water after which it may move randomly until it finds food again to repeat the process. If at any time the ant runs into poison, the ant is removed from the population.

The ant moves randomly by randomly picking a value -1, 0, or 1 for each x and y and adding them to the corresponding tile it is currently on. The indices are used to find the Node in which the ant should travel to next. The same A* algorithm used in the first game is used to find this next Node with the same methods for the Ant to follow that "path". When the food is found, the ant finds its path home using the same A* algorithm from the node its currently on to that with the same indices of the home. To avoid poison, the tiles with poison are given the high movement cost of 99. When food is returned, the linked list which keeps track of each ant in the colony has a new ant or 'Player' added to the linked list. When it runs into poison, that ant is removed (by keeping track of its index in the linked list). It may be important to note that ants that are added or removed are done so before looping through each element of the linked list. Doing so within the loop will cause errors. So instead, booleans are used to keep track when and where to add/remove them.