AI ReadMe

Gary Connelly - G00336837

April 2019

Contents

1 Introduction

This is the ReadMe document for the fourth year project completed as part of the Artificial Intelligence module. This document will go through the various features in the game and the design of the game and explain how they were created or modified, as well as give a run-through of how to play the game.

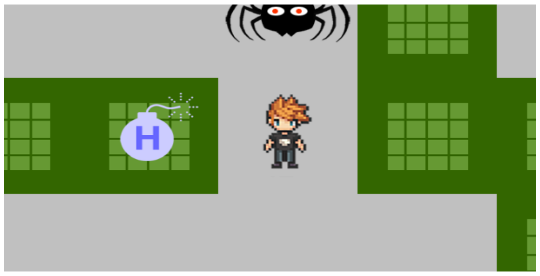
2 Installation Guide

To run this program, simply navigate to the folder with the src and resources folders, and run this command - java –cp ./game.jar ie.gmit.sw.ai.GameRunner. This will automatically start up the application by providing the user with a command line menu.

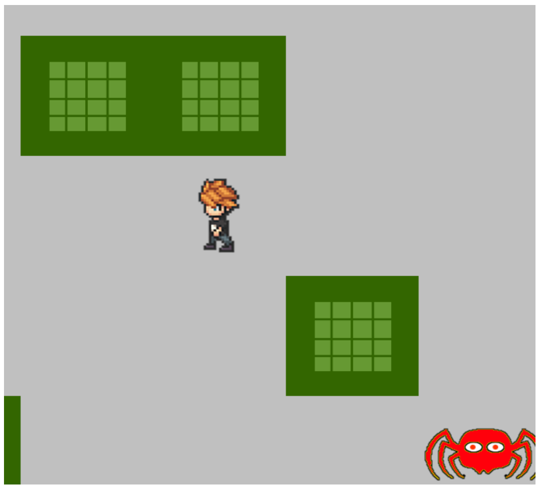
Alternatively, you can import the contents of the source folder into an eclipse project. For this to work however, you will need a JFuzzy logic jar file added to the eclipse project and add this jar file to the project class path.

3 Description

This application automatically generates a maze that the player can traverse through. The character can pickup weapons and health by walking into the parts of the maze that contain those components.



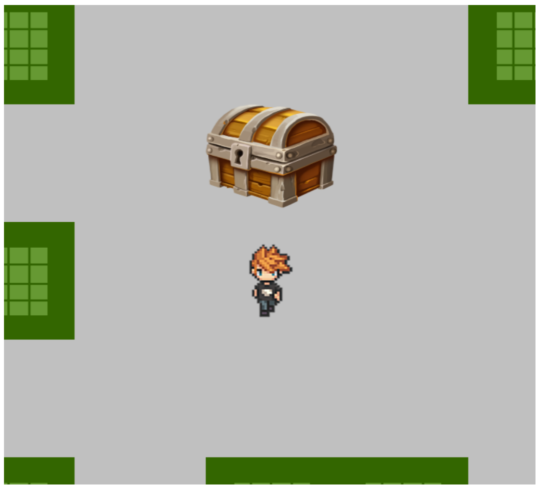
The player will also have to avoid spiders throughout the game. Some of these spiders(black spiders) will be actively hunting the player, while others(red spiders) will be using a neural network to decide whether to attack the player or to avoid it.



If the player ends up in a fight with a spider, he will lose health, strength and defence. He will need to find more of these components throughout the maze in order to recover.

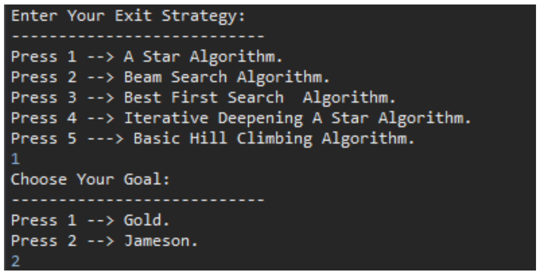
The aim of the game is to reach the maze exit before running out of health and dying.

The maze exit is represented by a treasure chest.



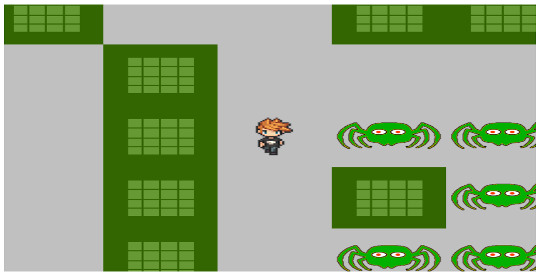
4 How to Play

When the game starts up, the user will be faced with a simple menu.

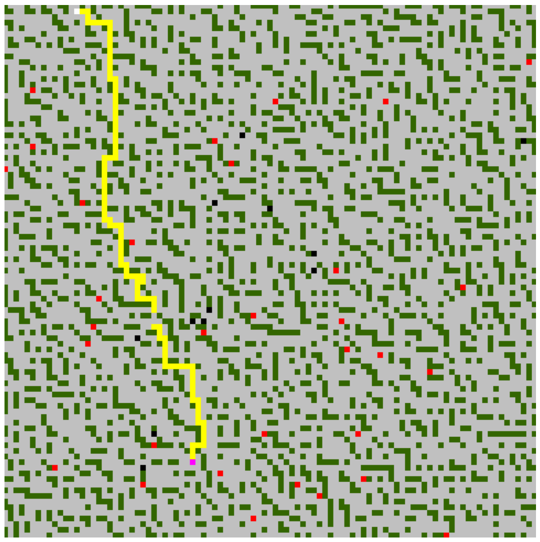


```
Enter Your Exit Strategy:
---------------------------
Press 1 --> A Star Algorithm.
Press 2 --> Beam Search Algorithm.
Press 3 --> Best First Search  Algorithm.
Press 4 --> Iterative Deepening A Star Algorithm.
Press 5 ---> Basic Hill Climbing Algorithm.
1
Choose Your Goal:
---------------------------
Press 1 --> Gold.
Press 2 --> Jameson.
2
```

This menu will simply ask the user what exit algorithm they wish to use. This algorithm can be used when the user collects a 'help' icon during the game. Suppose the user wants to use the A* algorithm to find the exit. When they collect a help icon and press 'S' on their keyboard, green spiders will outline the path to the exit using that algorithm.



If the user presses the 'Z' key on the keyboard, they can zoom out and see the quickest path to the maze exit coloured in yellow.



They can get health/strength/defence pickups along the way by running into them in the maze. Whenever the player engages in a fight with a spider, or whenever the player gets a pickup, the player's stats will be displayed on the console.

4.1 Controls

The player is controlled using the keyboard. The keys that can be used are as follows:

- Right Arrow - Moves the Player to the right.
- Left Arrow - Moves the player to the left.
- Up Arrow - Moves the player up.
- Down Arrow - Moves the player down.
- Z Key - Toggles the zoom on the game.
- S Key - Searches for the exit node.(Only if the player has collected a help icon.)

5 Spiders

There are 4 types of spiders used in this implementation.

- Dead Spider - Represented by the blue spider sprite.
- Help spider - Represented by the green spider sprite.

- Regular Spider - Represented by the red spider sprite. These spiders can move around the maze randomly. If they are about to move into where the player is located, a neural network will quickly classify the attributes of the player to decide whether the spider should attack, panic, run or hide.
- A* Spider - Represented by the black spider sprite. These spiders can use the A* heuristic search algorithm to search for the player throughout the maze.

6 Fuzzy Logic Implementation

This game uses fuzzy logic to infer the outcome of a fight between the player and a spider.

6.1 Fuzzy Sets

There are three input variables used in this fuzzy logic implementation; health, weapon, defence. Each of these values can range from 0 to 100.

6.1.1 health

The membership breakdown for the player's health is as follows;

- TERM low := (0, 1) (25, 0); - Function where a health of 0 has membership 1 of set low and a health of 25 has a membership 0 of set low.
- TERM medium := (20, 0) (45,1) (60,1) (75,0); - Function where health 20 has membership 0 of set medium, health of 45-60 has membership 1 of set medium, and health of 75+ has membership 0 of medium.
- TERM high := (70, 0) (90, 1); - Function where health of 70 has membership 0 of set high, and health 90+ has membership 1 of set high.

The rational behind these fuzzy sets came from the fact that health pickups in the game are worth 25 to the players health, therefore breaking down the players health into these fuzzy sets seemed quite intuitive.

6.1.2 weapon

The membership of the player's weapon strength is as follows;

- TERM sword := (0, 1) (25, 0); - Function where a strength of 0 has membership 1 of set sword and a strength of 25 has a membership 0 of set sword.
- TERM bomb := (40.0,1) (60.0,1) (80.0,0); - Function where strength 40 has membership 1 of set bomb, strength of 60 has membership 1 of set bomb, and strength of 80+ has membership 0 of bomb.

- TERM hbomb := (75, 0) (100, 1); - Function where strength of 75 has membership 0 of set hbomb, and strength 100 has membership 1 of set hbomb.

The names for these fuzzy sets came from the idea that if the player has the strength of 45, they have a strength equivalent of a sword. If they have a strength of 60, the have the strength equivalent of a bomb etc. The players strength is increased accordingly in the game when they pick up one of these weapons.

6.1.3 defence

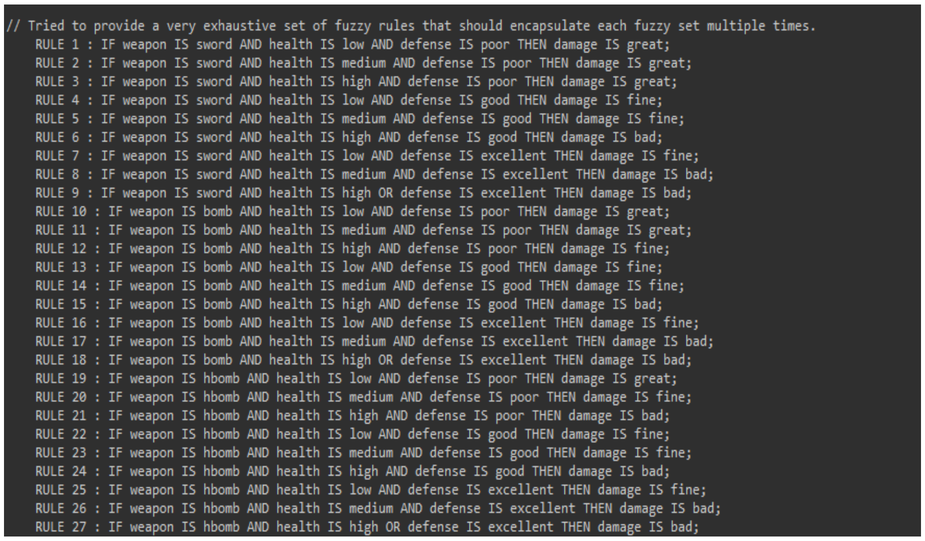
The membership of the player's defence is as follows;

- TERM poor := (0, 1) (25, 0); - Function where a defence of 0 has membership 1 of set poor and a defence of 25 has a membership 0 of set poor.
- TERM good := (20, 0) (45,1) (60,1) (75,0); - Function where defence 20 has membership 0 of set good, defence 45-60 has membership 1 of set good, and defence 75+ has membership 0 of set good.
- TERM excellent := (70, 0) (100, 1); - Function where defence of 70 has membership 0 of set excellent, and strength 100 has membership 1 of set excellent.

The rational behind these fuzzy sets came from the fact that defence(shield) pickups in the game are worth 25 to the players defence, therefore breaking down the players defence into these fuzzy sets seemed quite intuitive.

6.2 Fuzzy Rules

The rules governing the inference of this fuzzy logic are quite extensive. There are 27 rules, which cover each fuzzy set multiple times.



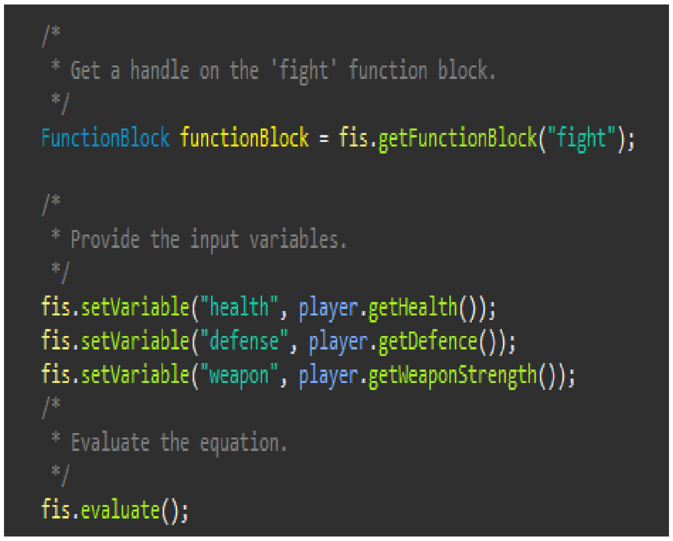
```
// Tried to provide a very exhaustive set of fuzzy rules that should encapsulate each fuzzy set multiple times.
    RULE 1 : IF weapon IS sword AND health IS low AND defense IS poor THEN damage IS great;
    RULE 2 : IF weapon IS sword AND health IS medium AND defense IS poor THEN damage IS great;
    RULE 3 : IF weapon IS sword AND health IS high AND defense IS poor THEN damage IS great;
    RULE 4 : IF weapon IS sword AND health IS low AND defense IS good THEN damage IS fine;
    RULE 5 : IF weapon IS sword AND health IS medium AND defense IS good THEN damage IS fine;
    RULE 6 : IF weapon IS sword AND health IS high AND defense IS good THEN damage IS bad;
    RULE 7 : IF weapon IS sword AND health IS low AND defense IS excellent THEN damage IS fine;
    RULE 8 : IF weapon IS sword AND health IS medium AND defense IS excellent THEN damage IS bad;
    RULE 9 : IF weapon IS sword AND health IS high OR defense IS excellent THEN damage IS bad;
    RULE 10 : IF weapon IS bomb AND health IS low AND defense IS poor THEN damage IS great;
    RULE 11 : IF weapon IS bomb AND health IS medium AND defense IS poor THEN damage IS great;
    RULE 12 : IF weapon IS bomb AND health IS high AND defense IS poor THEN damage IS fine;
    RULE 13 : IF weapon IS bomb AND health IS low AND defense IS good THEN damage IS fine;
    RULE 14 : IF weapon IS bomb AND health IS medium AND defense IS good THEN damage IS fine;
    RULE 15 : IF weapon IS bomb AND health IS high AND defense IS good THEN damage IS bad;
    RULE 16 : IF weapon IS bomb AND health IS low AND defense IS excellent THEN damage IS fine;
    RULE 17 : IF weapon IS bomb AND health IS medium AND defense IS excellent THEN damage IS bad;
    RULE 18 : IF weapon IS bomb AND health IS high OR defense IS excellent THEN damage IS bad;
    RULE 19 : IF weapon IS hbomb AND health IS low AND defense IS poor THEN damage IS great;
    RULE 20 : IF weapon IS hbomb AND health IS medium AND defense IS poor THEN damage IS fine;
    RULE 21 : IF weapon IS hbomb AND health IS high AND defense IS poor THEN damage IS bad;
    RULE 22 : IF weapon IS hbomb AND health IS low AND defense IS good THEN damage IS fine;
    RULE 23 : IF weapon IS hbomb AND health IS medium AND defense IS good THEN damage IS fine;
    RULE 24 : IF weapon IS hbomb AND health IS high AND defense IS good THEN damage IS bad;
    RULE 25 : IF weapon IS hbomb AND health IS low AND defense IS excellent THEN damage IS fine;
    RULE 26 : IF weapon IS hbomb AND health IS medium AND defense IS excellent THEN damage IS bad;
    RULE 27 : IF weapon IS hbomb AND health IS high OR defense IS excellent THEN damage IS bad;
```

Each rule in the rule block covers one set of each of the input variables. Given that there are three input variables, each with three sets, there would need to be 27 rules to achieve this(3x3x3).

6.3 Game Integration

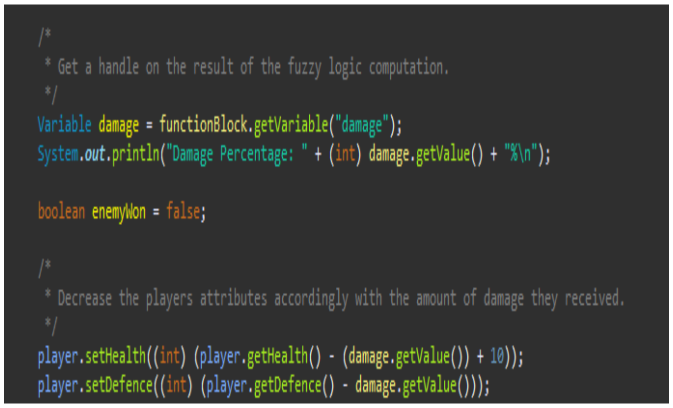
This fuzzy logic is integrated into the game in the form of a fight between the player and spider. The output variable 'damage', is calculated using center of gravity(COG). This damages value is then factored into the players attributes after each fight. This allows for very realistic fight outcomes. For example, if the players health was 80 and the players defence was 80, then the damage inflicted by a fight would be minimal. But if the players health was 80 and the defence was 20, the players health would take a bigger hit. This is logically very intuitive because naturally if the player fights an enemy with poor defence, their health would be severely damaged.

The input variables are supplied to the fuzzy logic by passing the players attributes.



```java
/*
 * Get a handle on the 'fight' function block.
 */
FunctionBlock functionBlock = fis.getFunctionBlock("fight");

/*
 * Provide the input variables.
 */
fis.setVariable("health", player.getHealth());
fis.setVariable("defense", player.getDefence());
fis.setVariable("weapon", player.getWeaponStrength());
/*
 * Evaluate the equation.
 */
fis.evaluate();
```

The damage is then factored into the players attributes after the evaluation. If the players health is 0, then the game is over.



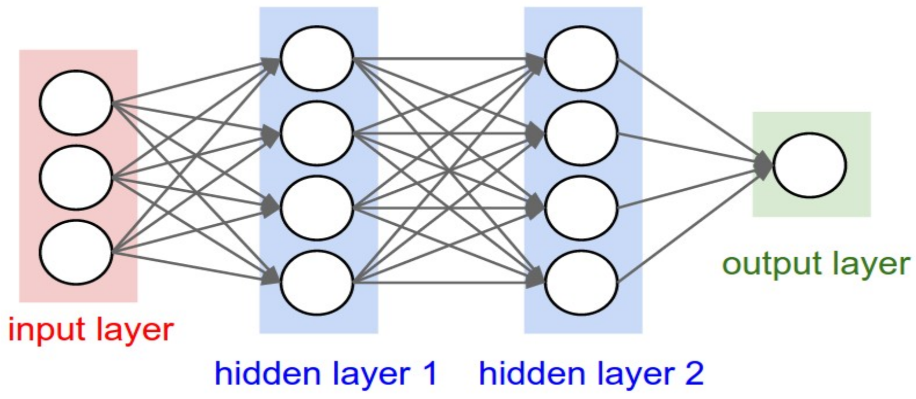
```java
/*
 * Get a handle on the result of the fuzzy logic computation.
 */
Variable damage = functionBlock.getVariable("damage");
System.out.println("Damage Percentage: " + (int) damage.getValue() + "%\n");

boolean enemyWon = false;

/*
 * Decrease the players attributes accordingly with the amount of damage they received.
 */
player.setHealth((int) (player.getHealth() - (damage.getValue()) + 10));
player.setDefence((int) (player.getDefence() - damage.getValue()));
```

If the player's health is not at 0 after the fight, this means that the player has won the fight and a boolean is returned from this method to indicate that. In the case that the player won, a blue spider is painted where the original spider was to signify that the spider is dead.

7 Neural Network Implementation

This game uses a neural network to decide what a regular(red) spider should do when they are about to run into the player.

7.1 Network Topology



A 3-layer back propagation neural network was used in this implementation. This neural network was provided on moodle.

- First layer - The input layer contains three neurons. This is because this neural network is going to be processing player attributes, of which there are three; health, strength and defence.
- Second/Hidden layer - The hidden layer contains three neurons. The rational for this came down to trial and error to see which hidden layer worked best.
- Output layer - The output layer for this network contains four neurons. This is because I wanted the spider to be able to do one of four things each time it classifies a player. The output neurons represent the actions panic, attack, hide and run.

The sigmoid activation function was used to activate the neurons. Any data that is entered to the network must be between 0 and 2. To do this, the input data had to be normalized prior to being entered.

How this input data was normalized, was loosely based off the fuzzy sets. If the player's health was less than 25, the value entered to the network was 0. If the player's health was between 25 and 75, the value entered to the network was 1. If the players health was more than 75, the value entered to the network was 2.

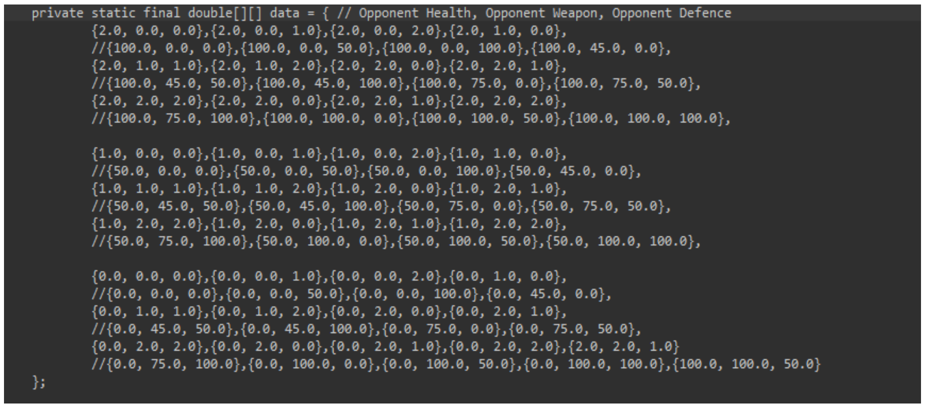
If the player's strength was less than 40, the value entered to the network was 0. If the player's strength was between 40 and 75, the value entered to the network was 1. If the player's strength was greater than 75, the value entered to the network was 2.

If the player's defence was less than 25, the value entered to the network was 0. If the player's defence was between 25 and 75, the value entered to the network was 1. If the player's defence was greater than 75, the value entered to the network was 2.

7.2 Training and Validation

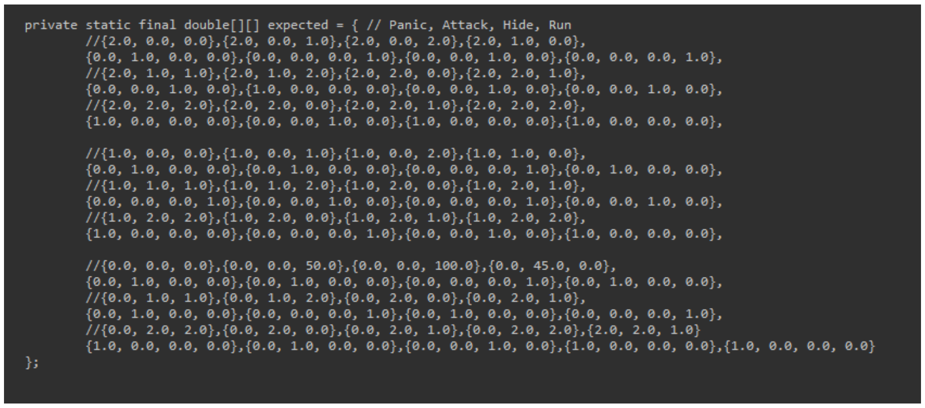
The training and validation data can be found inside the resources/neural folder. This folder contains three text files. Data.txt, DataMapping.txt, and Expected.txt. The Data.txt file contains all of the input data necessary to train the neural network. This file contains 37 inputs to train the network with. The Expected.txt file contains all of the corresponding expected outputs to the inputs. Therefore, this file also contains 37 outputs. The data used to train this network was intended to adequately exercise the neural network so that it would know how to classify any set of inputs.

The DataMappings.txt file contains the data and expected data in array format. This file is not used by the program, it is simply there to see clearly how the inputs are matched to the outputs.



```
private static final double[][] data = { // Opponent Health, Opponent Weapon, Opponent Defence
        {2.0, 0.0, 0.0},{2.0, 0.0, 1.0},{2.0, 0.0, 2.0},{2.0, 1.0, 0.0},
        //{100.0, 0.0, 0.0},{100.0, 0.0, 50.0},{100.0, 0.0, 100.0},{100.0, 45.0, 0.0},
        {2.0, 1.0, 1.0},{2.0, 1.0, 2.0},{2.0, 2.0, 0.0},{2.0, 2.0, 1.0},
        //{100.0, 45.0, 50.0},{100.0, 45.0, 100.0},{100.0, 75.0, 0.0},{100.0, 75.0, 50.0},
        {2.0, 2.0, 2.0},{2.0, 2.0, 0.0},{2.0, 2.0, 1.0},{2.0, 2.0, 2.0},
        //{100.0, 75.0, 100.0},{100.0, 100.0, 0.0},{100.0, 100.0, 50.0},{100.0, 100.0, 100.0},

        {1.0, 0.0, 0.0},{1.0, 0.0, 1.0},{1.0, 0.0, 2.0},{1.0, 1.0, 0.0},
        //{50.0, 0.0, 0.0},{50.0, 0.0, 50.0},{50.0, 0.0, 100.0},{50.0, 45.0, 0.0},
        {1.0, 1.0, 1.0},{1.0, 1.0, 2.0},{1.0, 2.0, 0.0},{1.0, 2.0, 1.0},
        //{50.0, 45.0, 50.0},{50.0, 45.0, 100.0},{50.0, 75.0, 0.0},{50.0, 75.0, 50.0},
        {1.0, 2.0, 2.0},{1.0, 2.0, 0.0},{1.0, 2.0, 1.0},{1.0, 2.0, 2.0},
        //{50.0, 75.0, 100.0},{50.0, 100.0, 0.0},{50.0, 100.0, 50.0},{50.0, 100.0, 100.0},

        {0.0, 0.0, 0.0},{0.0, 0.0, 1.0},{0.0, 0.0, 2.0},{0.0, 1.0, 0.0},
        //{0.0, 0.0, 0.0},{0.0, 0.0, 50.0},{0.0, 0.0, 100.0},{0.0, 45.0, 0.0},
        {0.0, 1.0, 1.0},{0.0, 1.0, 2.0},{0.0, 2.0, 0.0},{0.0, 2.0, 1.0},
        //{0.0, 45.0, 50.0},{0.0, 45.0, 100.0},{0.0, 75.0, 0.0},{0.0, 75.0, 50.0},
        {0.0, 2.0, 2.0},{0.0, 2.0, 0.0},{0.0, 2.0, 1.0},{0.0, 2.0, 2.0},{2.0, 2.0, 1.0}
        //{0.0, 75.0, 100.0},{0.0, 100.0, 0.0},{0.0, 100.0, 50.0},{0.0, 100.0, 100.0},{100.0, 100.0, 50.0}
};
```

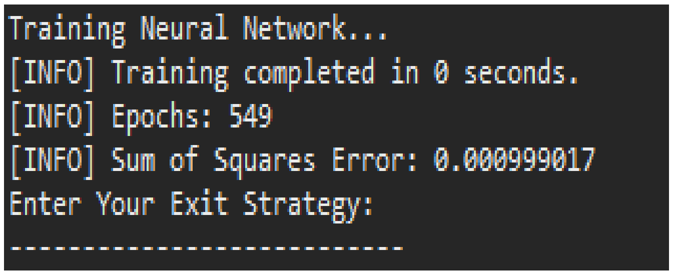
In the data array, it can be seen how the players attributes are mapped to an input. If the players attribute is 75 of higher, it is represented by a 2. If the players health is between 25 and 75, it is represented by a 1 and if a players attribute is less than 25 it is represented by a 0.



```
private static final double[][] expected = { // Panic, Attack, Hide, Run
        //{2.0, 0.0, 0.0},{2.0, 0.0, 1.0},{2.0, 0.0, 2.0},{2.0, 1.0, 0.0},
        {0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 0.0, 1.0, 0.0},{0.0, 0.0, 0.0, 1.0},
        //{2.0, 1.0, 1.0},{2.0, 1.0, 2.0},{2.0, 2.0, 0.0},{2.0, 2.0, 1.0},
        {0.0, 0.0, 1.0, 0.0},{1.0, 0.0, 0.0, 0.0},{0.0, 0.0, 1.0, 0.0},{0.0, 0.0, 1.0, 0.0},
        //{2.0, 2.0, 2.0},{2.0, 2.0, 0.0},{2.0, 2.0, 1.0},{2.0, 2.0, 2.0},
        {1.0, 0.0, 0.0, 0.0},{0.0, 0.0, 1.0, 0.0},{1.0, 0.0, 0.0, 0.0},{1.0, 0.0, 0.0, 0.0},

        //{1.0, 0.0, 0.0},{1.0, 0.0, 1.0},{1.0, 0.0, 2.0},{1.0, 1.0, 0.0},
        {0.0, 1.0, 0.0, 0.0},{0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 1.0, 0.0, 0.0},
        //{1.0, 1.0, 1.0},{1.0, 1.0, 2.0},{1.0, 2.0, 0.0},{1.0, 2.0, 1.0},
        {0.0, 0.0, 0.0, 1.0},{0.0, 0.0, 1.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 0.0, 1.0, 0.0},
        //{1.0, 2.0, 2.0},{1.0, 2.0, 0.0},{1.0, 2.0, 1.0},{1.0, 2.0, 2.0},
        {1.0, 0.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 0.0, 1.0, 0.0},{1.0, 0.0, 0.0, 0.0},

        //{0.0, 0.0, 0.0},{0.0, 0.0, 50.0},{0.0, 0.0, 100.0},{0.0, 45.0, 0.0},
        {0.0, 1.0, 0.0, 0.0},{0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 1.0, 0.0, 0.0},
        //{0.0, 1.0, 1.0},{0.0, 1.0, 2.0},{0.0, 2.0, 0.0},{0.0, 2.0, 1.0},
        {0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},{0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 0.0, 1.0},
        //{0.0, 2.0, 2.0},{0.0, 2.0, 0.0},{0.0, 2.0, 1.0},{0.0, 2.0, 2.0},{2.0, 2.0, 1.0}
        {1.0, 0.0, 0.0, 0.0},{0.0, 1.0, 0.0, 0.0},{0.0, 0.0, 1.0, 0.0},{1.0, 0.0, 0.0, 0.0},{1.0, 0.0, 0.0, 0.0}
};
```

In the expected array, the input that corresponds with each of the outputs is written over them, preceded by //. This way of formatting the data made it very easy to spot if an output was incorrect for it's corresponding input.

The neural network is trained with 30000 epochs and a 0.6 error. The network is automatically trained when the program starts up.



```
Training Neural Network...
[INFO] Training completed in 0 seconds.
[INFO] Epochs: 549
[INFO] Sum of Squares Error: 0.000999017
Enter Your Exit Strategy:
---------------------------
```

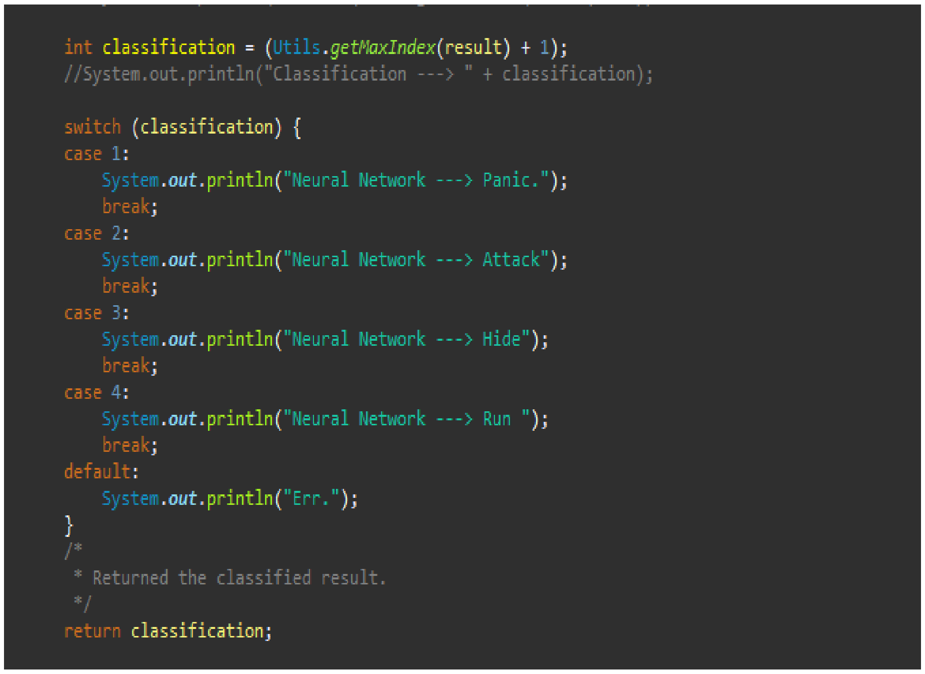
The reason for giving the network 30,000 epochs when it is clearly trained with less, is because of the fact that there seems to be a weird bug in the program some where. This was realised because of the fact that sometimes(rarely) the same network would take longer to train and have a much smaller sum of squares error. When this bug occurs, it takes all 30,000 epochs for it to train. I never figured out the reason for this, so to make sure it was getting trained fully, I upped the amount of epochs from 10000 to 30000.

7.3 Game Integration

This neural network was integrated into the game in the form of a regular spider's interaction with a player. If a spider is moving randomly, and happens to move into a space where the player is located, the players attributes are entered to the neural network and one of the following outputs occur.

- Panic - This is the worst case scenario. When the player has particularly strong attributes, the spider can just go into a state of panic. This is manifested by the spider just pausing on the spot and not moving for ten seconds. This was implemented by just freezing the spider thread that triggered the classification.
- Hide - This scenario happens when the player has quite strong attributes across the board. This is manifested by the spider seemingly disappearing from view. This is implemented by taking the spider that triggered the classification, and moving it 10 spaces away in a random direction.
- Run - This is when the player has quite poor attributes that spider doesn't need to panic or hide, but can just run away. This is manifested by the spider just moving to a different square as opposed to moving into the player's square. This is implemented by just re-running the method that causes the spider to move in a random direction.
- Attack - When the player has very poor attributes, the spider can attack him. This is manifested by the spider fighting the player. This is implemented by calling the method that triggers the fuzzy logic implementation of a fight between the spider and player.

This neural network is implemented into the game by returning a value between one and four representing the output node that was activated.



```java
int classification = (Utils.getMaxIndex(result) + 1);
//System.out.println("Classification ---> " + classification);

switch (classification) {
case 1:
    System.out.println("Neural Network ---> Panic.");
    break;
case 2:
    System.out.println("Neural Network ---> Attack");
    break;
case 3:
    System.out.println("Neural Network ---> Hide");
    break;
case 4:
    System.out.println("Neural Network ---> Run ");
    break;
default:
    System.out.println("Err.");
}
/*
 * Returned the classified result.
 */
return classification;
```

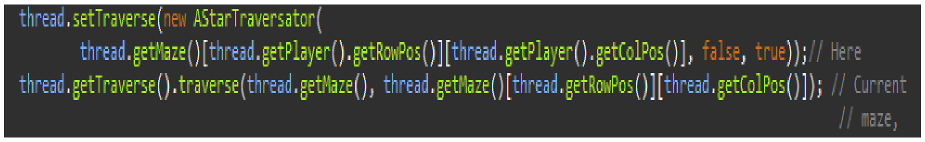
The consuming class can then perform an action based on what value between one and four is returned.

8 Heuristic Search Implementation

Heuristic search is used in two areas of the application. One is used to allow the black spiders to use the A* algorithm to follow the player and attempt to catch or surround him. The other is used to that the user can select from a variety of algorithms to search for the maze exit node.

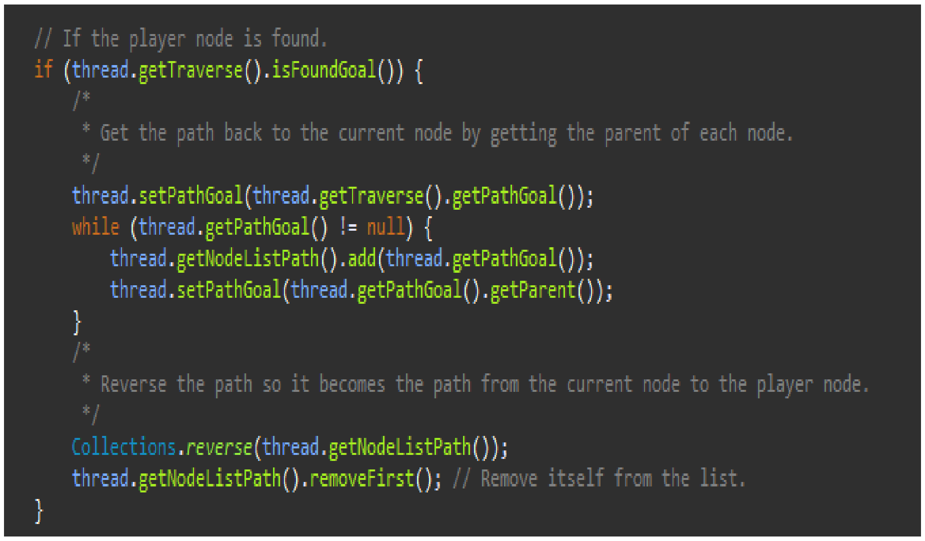
8.1 Search for Player

As mentioned, the A* algorithm is used to allow the black spiders to search for the player in the game. The reason this algorithm was used for the spiders, was the fact that is is optimal and complete. This was implemented by passing int he players position to the traverser as a goal node. The traverse method is then called on the A* traversator by passing in the current position spider.



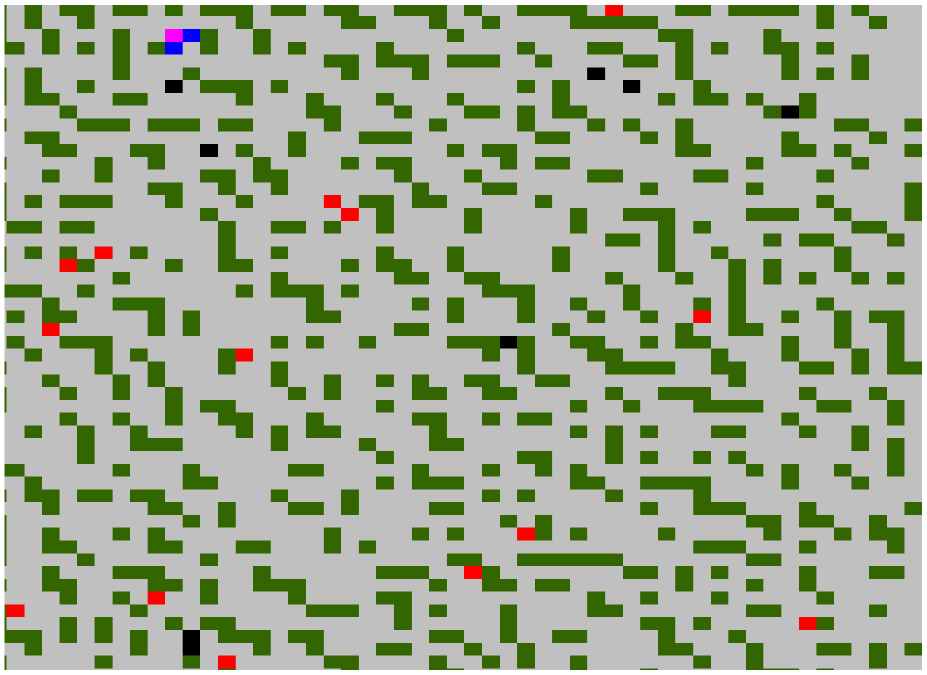
```
thread.setTraverse(new AStarTraversator(
        thread.getMaze()[thread.getPlayer().getRowPos()][thread.getPlayer().getColPos()], false, true));// Here
thread.getTraverse().traverse(thread.getMaze(), thread.getMaze()[thread.getRowPos()][thread.getColPos()]); // Current
                                                                                                            // maze,
```

This traversal continues until the players node is found. When the player's node is found, the parent of each node working backwards is added to a list. This list is then reversed so that the spider has a list of nodes to traverse through that lead straight to the the player.



```
// If the player node is found.
if (thread.getTraverse().isFoundGoal()) {
    /*
     * Get the path back to the current node by getting the parent of each node.
     */
    thread.setPathGoal(thread.getTraverse().getPathGoal());
    while (thread.getPathGoal() != null) {
        thread.getNodeListPath().add(thread.getPathGoal());
        thread.setPathGoal(thread.getPathGoal().getParent());
    }
    /*
     * Reverse the path so it becomes the path from the current node to the player node.
     */
    Collections.reverse(thread.getNodeListPath());
    thread.getNodeListPath().removeFirst(); // Remove itself from the list.
}
```

Whenever the player moves, this path is re-computed. This algorithm is integrated into the spiders behaviour by simply moving the spider to the next node on the list, giving it the appearance that is it is looking for the player.



Here, you can see how all the black spiders are gravitating towards the player(pink).

8.2 Search for Exit

Heuristic search is also used to allow the player to see a path out of the maze using an heuristic search algorithm. The algorithm used for this is selected by the user at the start of the game. The different search algorithms available are:

- A Star.
- Beam Search.
- Best First Search.
- Iterative Deepening A Star.
- Basic Hill climbing.

These algorithms were very easily implemented using the traversal algorithms provided in the labs. All that had to be done was to call traverse on whatever algorithm was picked, pass it the exit node and allow the TraversatorStats class to paint the nodes leading back to the player a certain color.

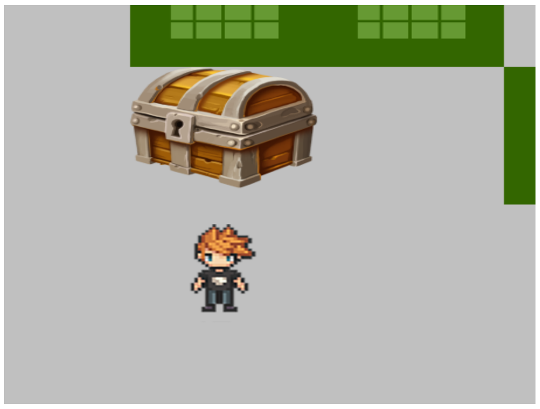
This was integrated to the game by only allowing the player to use this algorithm after they have collected the 'help' collectable. They can perform the search by pressing 's' on their keyboard.

9 Additional Features

Any extra images or sprites used in this program were edited and re-sized using photopea.com.

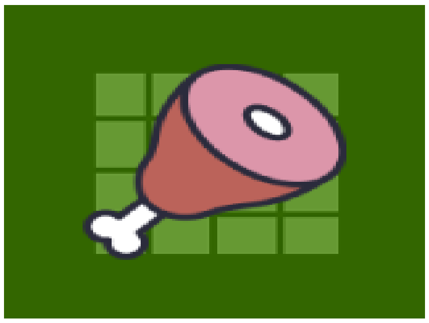
9.1 Chest

The chest sprite is used to represent the goal node/maze exit.



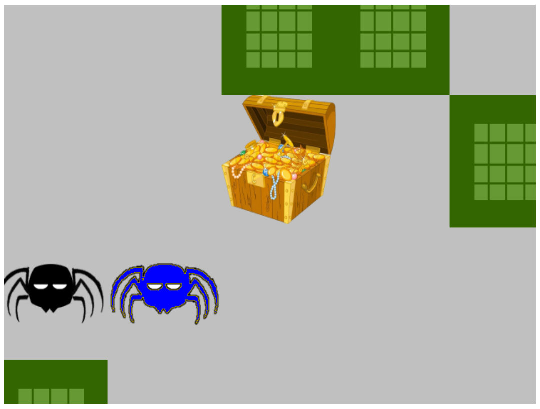
9.2 Food

The player can collect food throughout the game which will add 25 to the players health.



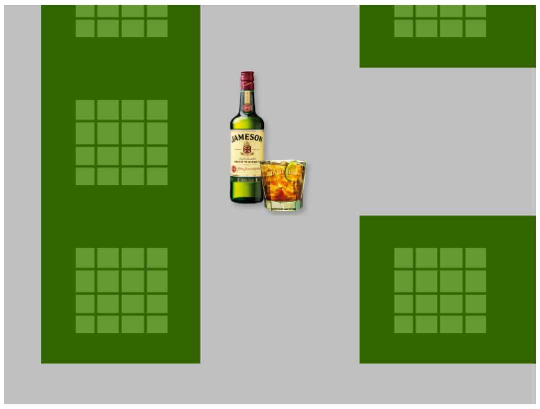
9.3 Gold

If the user selects gold over whiskey at the start menu, when they open the chest they will see gold.



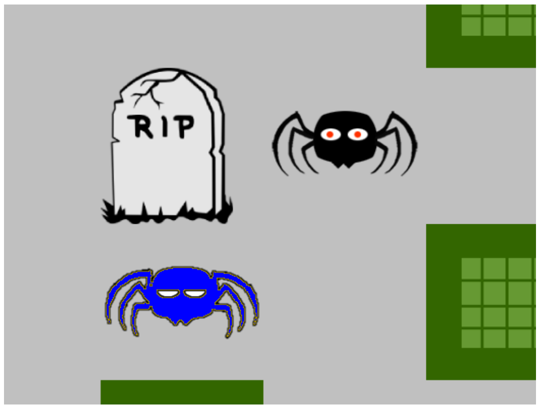
9.4 Whiskey

If the user selected whiskey at the beginning of the game, when they open the chest they will see a lovely bottle of Jameson.



9.5 RIP

When the player gets killed by the spiders, a headstone will be put in it's place.



9.6 Shield

The player can collect shields throughout the game, each shield will add 25 to the player's defence.

10 Bugs and Limitations

There are a few minor issues with this project that I didn't get around to solving prior to submission. One of these issues is that is the exit node is placed in a corner of two hedge nodes, the search algorithms seem to be unable to find it. Also, sometimes when using beam search to find the exit node, the players node just gets painted yellow, meaning you are unable to see the player.

Also, when using the IDA Start search algorithm to find the exit node, sometimes when testing this when the player was at the opposite end of the maze it could cause the program to crash. If the player is closer to the exit node however IDA Star works quite well.