IT3105 Project 2

Solving the 2048 game using Expectimax

Justification of programming language and frameworks

We have decided to implement the 2048 solver in Java mainly because of performance. We first implemented a faulty version in Python which was too slow, but quickly decided to scrap all our work in the Python project and switch over to Java. For visualization we initially utilized a GitHub repository called 2048fx [1], but when we had to debug the program later, we felt it would be better to do all the game logic ourselves, so that we would be familiar with all the code. Therefore, we started using the the lightweight Swing graphical user interface [2] provided by the course instructor on ItsLearning.

Classes, methods and data structures

Board

This class contains all the logic for the game. It represents the actual board as a 2d array of integers. Some of the most significant methods follow;

placeRandomTile()

```
public void placeRandomTile() {
    ArrayList<Coordinate> freeSlots = getFreeTiles();
    if (freeSlots.size() > 0) {
        int randomCoordinateIndex = new Random().nextInt(freeSlots.size());
        Coordinate coordinate = freeSlots.get(randomCoordinateIndex);
        placeTile(coordinate, new Random().nextDouble() < 0.9 ? 2 : 4);
    }
}
```

This method uses the utility methods *getFreeTiles()* and *placeTile()*, in addition to the Coordinate class which simply contains the X and Y coordinates of a given tile.

The movement methods are too comprehensive to be put in this document, but we implemented logic for moving left, right, up and down. *moveLeft()* and *moveUp()* scan the board lists from left to right and up to down, respectively. For each element, it checks if there is an equal tile next to it (or separated by zeroes), and if so, it sets the match to zero and doubles its own value. It then checks to move all tiles as far as possible in the corresponding direction. *moveRight()* and *moveDown()* simply reverse the board arrays and call *moveLeft()* and *moveUp()* and then reverse the arrays again.

The Board class also contains a Boolean *move* method that takes a Direction enumeration as an argument and calls the corresponding movement method. It returns true if the call resulted in tiles moving. *isAvailableMoves()* checks if there are any available moves from the current state, and *copy()* returns a copy of the current state of the Board object.

Coordinate & Direction

These two classes are very simple, and is mainly used for readability. Direction is an enum and Coordinate is a simple object that only holds an X and a Y value.

Expectimax

This is the most interesting class for this assignment. The first method of significance here is *recommendMove()*

```
private Direction recommendMove(Board board, int depth) {
    double maxScore = Double.NEGATIVE_INFINITY;
    Direction bestDirection = Direction.DOWN;

    for (Direction direction : Direction.values()) {
        Board copy = board.copy();
        if (copy.move(direction)) {
            double currentScore = expectimax(copy, depth );
            if (currentScore > maxScore) {
                maxScore = currentScore;
                bestDirection = direction;
            }
        }
    }
    return bestDirection;
}
```

For each value of the Direction enumeration, it creates a copy of the board, and if the *copy.move()* method returns true, it calculates the expectimax score of that direction. Our *expectimax()* method is too comprehensive to be included in this document, but it is well documented in our source code. It takes two parameters; *board* and *depth*. If the depth is zero, it returns the calculated heuristic evaluation of the board. If the depth is even-numbered, it creates a copy of the board for each of the values of Direction, and calls *expectimax()* recursively with the copy and depth-1 as parameters. It then returns the max of these values. If the depth is odd-numbered, that means we are at a chance node. We first set the score to zero, and then create two copies, *alpha* and *beta*, of the current board for each of the empty tiles. For each empty tile, we add a 2-tile to *alpha* and a 4-tile to *beta*, before we add *0.9*expectimax(alpha, depth-1)* and *0.1*expectimax(beta, depth-1)* to the score. After all scores have been added, we divided the score by the number of free tiles on the board and return the score. The last method of significance is *run(),* which calls *recommendMove()* with a dynamic depth of either 6 or 8, depending on whether there are less than four empty tiles or not. It then moves the specified direction, and spawns a random tile.

GUI

The *main* method in our GUI-class first creates a new Board object, and then an Expectimax instance with the newly created board as a parameter. As long as there are available moves, it runs the *Expectimax* instance's *run()* method and then converts our 2d integer array into a 1d integer array suitable for the GUI class.

Heuristic function

Our heuristic function has evolved through multiple iterations of trial and error, and we received ideas from multiple blog posts and forum threads on which heuristic function is the most advantageous. The heuristic we ended up using utilizes a snake patterned weight matrix which we multiply by the double of the current board state and sum up all the values to calculate an overall state score.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 6 & 5 & 5 & 4 \\ 7 & 9 & 12 & 15 \\ 55 & 35 & 25 & 20 \end{pmatrix}$$

Snake-patterned weight matrix

So, given some board state, the evaluation of that state would be the following;

$$\text{State} * \text{Weight Matrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 8 & 4 & 2 & 16 \\ 512 & 256 & 128 & 32 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 2 & 3 \\ 6 & 5 & 5 & 4 \\ 7 & 9 & 12 & 15 \\ 55 & 35 & 25 & 20 \end{pmatrix} =$$

$$\begin{pmatrix} 0*2*0 & 0*2*1 & 0*2*2 & 0*2*3 \\ 2*2*6 & 4*2*5 & 0*2*5 & 0*2*4 \\ 8*2*7 & 4*2*9 & 2*2*12 & 16*2*15 \\ 512*2*55 & 256*2*35 & 128*2*25 & 32*2*20 \end{pmatrix}$$

$$\rightarrow \sum \begin{pmatrix} 0 & 0 & 0 & 0 \\ 24 & 40 & 0 & 0 \\ 112 & 72 & 48 & 480 \\ 56320 & 17920 & 6400 & 1280 \end{pmatrix} = 82696$$

Using this heuristic, we achieve multiple properties; the AI tends to move towards a monotonic board, that is that the values of all tiles are all either increasing or decreasing along the horizontal or vertical axis. We also achieve smoothness, because the result of having a monotonic board tends to be that the difference in value of adjacent tiles are as small as possible. To compare how our heuristic performed in comparison to the more classical gradient weight matrix, we ran the AI ten times and calculated the statistics.

$$\begin{pmatrix} 15 & 13 & 11 & 9 \\ 13 & 9 & 7 & 4 \\ 11 & 7 & 3 & 1 \\ 9 & 4 & 1 & 0 \end{pmatrix}$$

Gradient-patterned weight matrix

	1024 tile	2048 tile	4096 tile	Average score	Best score
Snake pattern	100%	100%	70%	55158	79472
Gradient	100%	90%	50%	48206	76756

[1] https://github.com/brunoborges/fx2048
[2] https://github.com/jorgenkg/IT3105