AI Programing, Module 4

Fredrik C. Berg and Tale Prestmo

October 2015

1 Introduction

The task is to implement either the expectimax algorithm or the minimax algorithm with alpha-beta pruning for solving the 2048 game. The task was solved with Java 1.8, using the JavaFX framework for visualization.

The program is started by running the main.java file. From here it is possible to play the game by using the arrows, or make the AI make the moves. By pressing enter the AI makes one move, and by pressing 'S' the AI solves the game. When pressing 'S' the GUI won't work because the algorithm does not allow the GUI to be updated between every calculation. This could be fixed by making the GUI and the AI to work in different threads, but this was not prioritized during this task.

2 Expectimax

Expectimax is a variant of the more traditional minimax algorithm typically used to solve games. Expectimax builds a search tree considering of all of the possible board states to a predefined depth. In a two player board game, one would normally want to minimize the best moves that the opponent can perform. However, as there 2048 is a single player game with chance involved, the states generated are reflected by either the player's possible moves(left, right, up and down) or the random spawn of a new tile. Note that a new tile can spawn as 2 or 4 in any of the free squares on the board, and all possible spawns are considered.

2.1 Implementation

The algorithm is initialized with three variables, the state of the board, the search depth and a boolean maximizing_player.

```
private float expectimax(int[][] board, int depth, boolean maximizing_player) {
```

Board is the current state of the board we want to explore, depth is the maximum search depth and maximizing player describes whether or not the player is to make a move, or a random tile should appear. The algorithm is only supposed to check the heuristic at the bottom of the search tree. Thus, the first part of the algorithm checks that. If this is the case, it returns the heuristic of the board state. The implementation uses a default depth of 4.

```
if (depth == 0) {
    float h = heuristic(board);
```

```
    return h;
}
```

The second part of the algorithm describes the behaviour of the algorithm if it is called with the maximizing_player flag. It calculates a value *alpha*, with each of the four possible moves. *alpha* is set to be the best of these moves, when the recursive calls eventually returns. *maximizing_player* is set to false, as we at that stage want to explore the result of the random tile that appears after a move is made.

```
if (maximizing_player) {
    this.alpha = Integer.MIN_VALUE;
    for (int[][] neighbour : getNeighbours(board)) {
        alpha = Math.max(alpha, expectimax(neighbour, depth - 1, false));
    }
}
```

The third part of the algorithm describes the behaviour of the algorithm if it is called with a chance state. That is, the maximizing_player flag set to false. The algorithm will then recursively call itself with all possible permutations of random tiles. For each iteration, *alpha* is set to be itself plus the sum of the recursive call and the probability of the current random tile appearing.

```
else {
    this.alpha = 0;
    int numberOfZeroes = Board.countZeroes(board);
    for (int x = 0; x < 4; x++) {
        for (int y = 0; y < 4; y++) {
            if (board[x][y] == 0) {
                int[][] copy = Board.getCopy(board);
                copy[x][y] = 2;
                alpha += ((0.9 / numberOfZeroes) * expectimax(copy, depth - 1, true));
                copy = Board.getCopy(board);
                copy[x][y] = 4;
                alpha += ((0.1) / numberOfZeroes) * expectimax(copy, depth - 1, true);
            }
        }
    }
}
```

After all of the nodes in the search tree has been explored and calculated, the algorithm returns a float, indicating the degree of successfullnes for a potential move. Thus, expectimax will need to be called with all four possible moves each time.

3 Heuristic

The heuristic for this task is a combination of three different sub-heuristics. Each of them are used with the values presented, and are then added together. The range between the different heuristics is quite large, and represents how important each of them are. The mathematical formula can be represented as $Heuristic = Adjacent\ cells + Gradient + Average\ sum$, with the different parts described below.

3.1 Gradient

The first and most valued heurestic is a gradient scoringmatrix that is multiplied with the current board, and then the totalsum of the board is calculated. The scoringmatrix prefers to keep the largest number in the upper left corner, and tries to get other large values next to it. Under is an example of the calculation of the heuristic value, the right matrix is the scoring matrix that is multiplied with the values of the board.

$$\sum \left(\begin{bmatrix} 10 & 9 & 8 & 7 \\ 9 & 5 & 4 & 3 \\ 8 & 4 & 2 & 2 \\ 7 & 3 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 512 & 256 & 128 & 32 \\ 16 & 8 & 4 & 8 \\ 2 & 4 & 2 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} \right) = \sum \begin{bmatrix} 5120 & 2304 & 1024 & 224 \\ 144 & 40 & 16 & 24 \\ 16 & 16 & 4 & 0 \\ 0 & 0 & 4 & 0 \end{bmatrix} = 8936$$

3.2 Average sum

The second heurestic calculates the average sum of each tile on the board that has a higher value than zero. This will make the AI prefer states with fewer tiles, since this raises the average value per tile. This heuristic will affect the AI more the larger the value of the tiles are. An alternative we attempted was a heuristic that counts the number of free spaces, and prefer those with more white spaces. However, it was replaced, as the current solution tends to get higher values.

$$\sum \begin{bmatrix} 512 & 256 & 128 & 32 \\ 16 & 8 & 4 & 8 \\ 2 & 4 & 2 & 0 \\ 0 & 0 & 2 & 0 \end{bmatrix} = 972 \implies 972/12 = 81$$

3.3 Adjacent cells

The third heurestic counts the number of adjacent tiles that have the same value. This is done to make it possible to merge tiles with the next move. This will effect the AI in the beginning of the game, but as the value of the tiles grows, this will almost not affect the heuristic at all.

$$\begin{bmatrix} 512 & 256 & 128 & 32 \\ 16 & 8 & 4 & 8 \\ 2 & 4 & 4 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix} \implies 6$$