AI programming: Module 4

Using Expectimax to play 2048

Bjørnar Walle Alvestad

2015

Norwegian University of Science and Technology

NTNU

Introduction

This report will describe the key elements of the expectimax algorithm, implemented in project module 4 of the AI-programming course. In this module, the task is to implement either the minimax algorithm with alpha beta pruning, or expectimax, and use it to play the game “2048”.

This implementation uses the expectimax algorithm, as this is more suited for the 2048 game. This is because of the random choice of spawn location and value. There is no “intelligence” in the random spawn generator.

The module is implemented using the Java programming language.

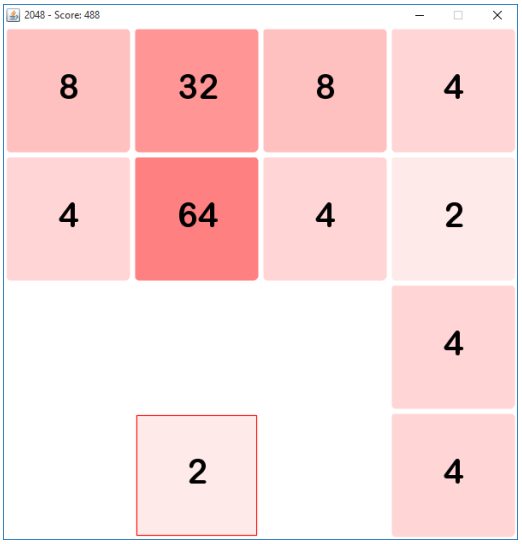

Figure 1: Screenshot of a board in the 2048 game.

The Expectimax algorithm

As mentioned in the introduction, this implementation is using the expectimax algorithm to search for the best move. It creates a search tree with a fixed depth, containing two different types of nodes:

- Max nodes. These nodes branches out to all of the legal moves on a given board. In most cases, the number of legal moves is 4, meaning the max-node will have 4 child nodes; one for each move. The max-nodes represents situations where the player must choose an action to perform, thus maximizing the score of the outcome.
- Chance nodes. These branches to all possible board configurations after a new, random tile has been spawned. The number of children for a chance node is equal to the number of free squares on the board times two, because there are two possible spawns for each free square (90% chance for the value 2, 10% chance for the value 4).

The generated tree consists of alternating layers of max-, and chance-nodes. This reflects the alternating turns of the player making a move, and the spawn of a random, new tile.

The search starts with a board state and places it in the root node, which is of type max. The algorithm is then iterating over each legal move for that board state. For each move, a subtree is created by recursively calling the `exp(Node n, int depth)` method. This method have different behavior for max nodes and chance nodes, but the high level functionality is the same; generate child nodes, and assign a value to the node being processed. Max nodes is assigned the highest value of its child nodes, while chance nodes have the expected value of its child nodes. This is calculated as follows:

1. The chance node in question will have all of its children generated, by assuming either the value 2 or 4 will spawn in any of the free squares. Therefore, the number of children is equal to the number of free squares times two.
2. Because of the recursive generation, all of the child nodes will already have their values assigned, either by the heuristic function, or by a max node.

3. Iterating over all children, multiplying the child value with the odds of that child being spawned, and then dividing by the sum of all odds. This value is then assigned to the chance node.

The `depth` argument in the `exp` method is incremented each time the method is recursively called. When the recursion reached the defined maximum depth, no more children is created. The value of the node n, passed as argument, is calculated using the heuristic function. The method then returns. This ensures that all nodes at the bottom layer will have their values assigned by the heuristic function.

After the search algorithm is completed, the root node will have its value assigned. The move associated with the child node with the same value as the root is then chosen and performed on the board. The search tree is then discarded, and a new search tree is initialized to find the next move. This loops until there is no legal moves, and the game ends.

Board representation

The 2048 game board consists of a 4 x 4 grid of cells (sometimes referred to as "tiles"). In this implementation, the highest value of a cell is $2^{15} = 32{,}768$. The lowest value is $2^1 = 2$. Additionally, any cell can be empty. This is internally represented with the value 0. A 64-bit long datatype is used to store the values of a board. Each cell will be given $\frac{64}{16} = 4$ bits to contain the tile value. The actual cell value is not stored, but rather $log_2(v)$, where v is the cell value; the exponent in which raise 2 to get the cell value.

The table below shows the relation between cell position, bit positions in the long datatype, binary and decimal value, and the cell value.

Cell position	*(0, 0)*	*(0, 1)*	*(0, 2)*	…	*(3, 1)*	*(3, 2)*	*(3, 3)*
Bit position	0-3	4-7	8-11	…	52-55	56-59	60-63
Binary value	0010	0011	0001	…	0100	0000	0110
Decimal value	2	3	1	…	4	0	6
Cell value	$2^2 = \mathbf{4}$	$2^3 = \mathbf{8}$	$2^1 = \mathbf{2}$	…	$2^4 = \mathbf{16}$	**Empty**	$2^6 = \mathbf{64}$

Heuristic function

The heuristic function used in this implementation is based on a pattern where high values are favourably placed along two sides, in an "L" formation. Each cell have a predefined weight, which is used to guide high-valued tiles to the bottom right corner. The weight of different board cells is shown in figure 2. A high heuristic value is favourable, as it represents a better board configuration. The key is to define a heuristic function that will reflect the actual state of a board. A better board state should always produce a higher heuristic value.

(0,3) 16^0	(1,3) 0	(2,3) 0	(3,3) 0
(0,2) 16^1	(1,2) 0	(2,2) 0	(3,2) 0
(0,1) 16^2	(1,1) 0	(2,1) 0	(3,1) 0
(0,0) 16^3	(1,0) 16^4	(2,0) 16^5	(3,0) 16^6

Figure 2: Weights and coordinates for different cells of the board. Tile values will be multiplied with their respective weight to produce the heuristic value.

The heuristic value in this implementation is calculated using the following formula:

$$h(b) = \sum_{(x,y)=(0,0)}^{(3,3)} \left(v(b)_{(x,y)} * w_{(x,y)} \right)$$

where:
$h(b)$ is the produced heuristic value of board b,
$v(b)_{(x,y)}$ is the cell value in position (x, y) of board b,[1]
$w_{(x,y)}$ is the predefined weight of cell (x, y), as defined in figure 2.

The heuristic function iterates over all board cells, multiplying the cell value with the corresponding cell weight. The sum of these products is considered the heuristic value for this board. Typical heuristic values is in the range of 10^7 - 10^8.

The weights are selected so that even if all weighted cells are filled with value x, merging the values into the highest weighted cell will produce a greater heuristic. This ensures that merging tiles is more favourable than just filling the weighted cells with values. Cells with weight 0 will not affect the heuristic. The 3x3 sub-grid with weight 0 is not taken into consideration when calculating heuristic. Most tiles will spawn here and is used to combine small values before they are merged into the main pattern on the left or bottom.

The heuristic function is defined in an abstract class `Heuristic`, and can be implemented in several other subclasses. This enables easy switching and development of new heuristic functions. Heuristic functions can also be assigned to the expectimax algorithm at runtime.

This heuristic is capable of producing the 8192 tile as shown in figure 3. Although at search depth 7, the 4096 tile is produced in around 90% of the runs. It took 447 tries at depth 6 to produce the 8192 tile.

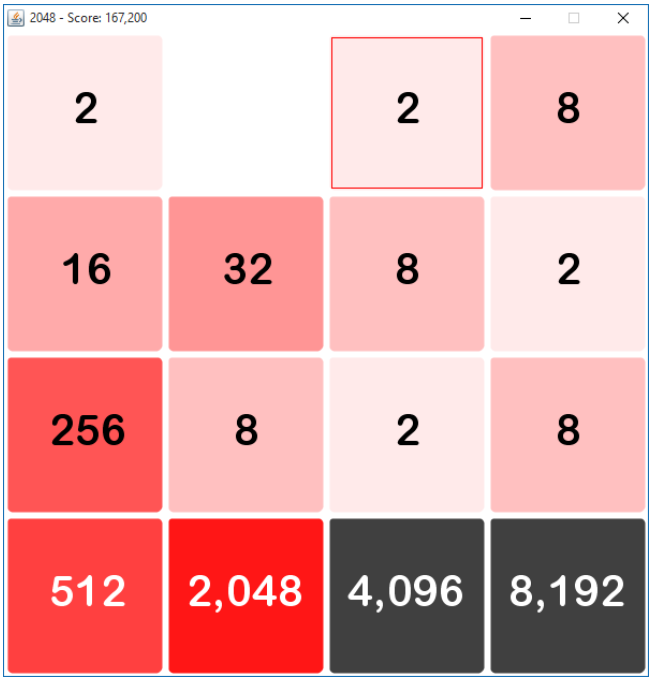

Figure 3: A board containing the 8192 tile. The screenshot is taken at move 6902. 29 moves later, the game ends. Notice how the highest valued tiles are placed according to the designed heuristic pattern.

References

Downing, Keith L., *AI Programming (IT-3105) Project Module #4: Using Minimax with Alpha-Beta Pruning to play 2048.* [Online]

[1] Note that this is not the cell value visible to the player, but rather the exponent used to represent 2^v.