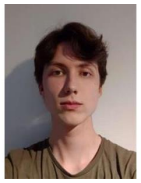
Technical University of Denmark

Board Game Assignment

Authors

Group 53:



Tobias Nicolai Frederiksen, s235086

Benjamin Philip Bakkelev, s235097

Marcus Rémi Lemser Eychenne, s230985

Maximillian Bjørn Mortensen, s236167

March 24, 2025

Contents

1 Game Rules

We have chosen the game 2048. The objective is to reach a block with the value 2048 or as high as possible. The game is played on a 4x4 grid, where each cell is either empty or contains a block with a value. At the start of the game, two blocks appear at random locations on the grid.

The player has four possible actions: up, down, left, and right. When an action is performed, all blocks move in that direction as far as possible, leaving no empty spaces between a block and the wall in the chosen direction. If two blocks are aligned along the chosen axis with no other blocks between them, they will merge into one block during the move—provided they have the same value. The value of the new block will be the sum of the two.

After each move, a new block is randomly generated in one of the empty cells. This block has a ninety percent chance of having the value two and a ten percent chance of having the value four. Each action can only be performed if it results in a change to the blocks' positions. If no moves are available, the game is over.

2 Kind Of Game

The game is a single-player high-score game based on turns, categorizing it as single-agent and static. It is not deterministic, as the block generated on each turn is unknown, making the game state after an action a stochastic set of possibilities.

Whether the game has full or partial observability depends on its implementation. Each game state is fully observable, but the transition between states introduces an unknown element. These factors led us to focus on an expectimax solution, where instead of playing against an opponent that always tries to make the least desirable move, the "opponent" is the randomness of the generated block.

This splits an action into two parts: the known part, where blocks move in a predictable manner, and the unknown part, where the randomly generated block creates an unpredictable but stochastic game state. With this implementation, it can be argued that after moving the blocks, the game state includes a hidden block that is not revealed until the next state. This suggests that the game is only partially observable.

3 Game States

Each cell can have 18 different states: it can be empty or contain a block with one of 17 possible values:

$$2^n, \quad n \in \mathbb{Z}, \quad n \in [1, 17]$$

This results in a simple upper bound of:

$$18^{16} \approx 10^{20}$$

However, not all game states are reachable. For example, no state except the initial one can have all outer cells empty, and no state can contain more than one block with the value 2^{17}.

Although determining a tighter upper bound is an interesting mathematical problem, it seems outside the scope of this work, as the result would likely remain within the same order of magnitude. The upper bound could be somewhat reduced if mirrored and rotated states were not counted. However, due to the heuristics described later, the grid will have a fixed orientation.

It is also important to note that this upper bound does not represent the total number of unique game states in the full game tree, as the same game state can appear multiple times.

4 Elements

Initial state (s0): The game starts in 4x4 grid, where 2 of the spots randomly has been chosen with a values of 2 or 4, where the rest of spots are empty.
Actions: In a given state, a player can perform 1 of 4 actions, depending on tile placement. If an empty tile exists an action may be taken to slide over this. Same applies to if tiles can be merged (same value)
Result: From a given action, a new state will occur in a deterministic way. This does not take into account the random occurrence of a new tile. This is taken into account in the expectimax algorithm. **Terminal-Test**: The game ends when the board is full and there is no way to merge any of tiles onto another. The function is implicit in our expectimax implementation, when possible moves are calculated. **Utility** : The utility of a state is calculated by the position of the tiles in the grid, based on the chosen heuristics. Each tile is looped through to calculate the total utility of a given state.

5 States and Moves

The games state is represented as a 2D array, where each element is a tile.
The move represented is done by having Enum, where the action is up,down,left and right, where the AI checks for if the move is valid to make the move and if 2 tiles have the same values so it can merge together. for make the AI it was enough to represent the game state because the AI always know the board state, because there is not any hidden cards or hidden information, where it would have to have a belief state, the only where there something hidden is the randomness from tiles that appear. however there is a probability for if the tiles gonna be a 2 or 4 therefore we are handle the probabilist from the Expectimax search.
The algorithm search for the AI was implementing with Expectimax search, choosing this algorithm was done because there is the probability with the tiles generation randomly, then having it to chose the best move by having it to check the values of every move

6 Methods

Below are some algorithms we considered implementing before making a decision:

- *Greedy algorithm*: This is not any specific algorithm, but would be a simple one, which would calculate the value of a state from a given move. The evaluation would use a simple heuristic (number placement,

number of empty tiles, and/or smoothness of the grid). It would search one move ahead and not take into account the random placement of a new tile.

- *Minimax with prunning*: We could pretend here that we play against an opponent who will choose the **worst** placement of a new tile, thus choosing the lowest value from the state of a randomly placed tile, while the AI agent would choose the highest value from any move played. Prunning would allow to search at higher depths. Alpha-beta pruning allows deeper search by cutting off branches that cannot influence the final decision.
- *Expectimax*: Similar to *Minimax* (without prunnnig), except the value of a move is calculated from the probability of randomly placed tiles, instead of the worst move being chosen. Pruning is generally not applicable due to averaging.
- *Monte Carlo*: Like Expectimax, this handles randomness, but instead of evaluating all possible outcomes, it uses **random simulations** (rollouts) to estimate the value of a move. It is a more stochastic and adaptive approach that incrementally builds a search tree based on outcomes of simulated games.

For our solution we have chosen to implement the **Expectimax solution**, as described in *Artificial Intelligence: a modern approach*[a]. Below is an illustration of the algorithm calculating one step ahead[b]. When calculating the value of all sub-sequent steps ($s1.1, s1.2, ..., s1.6$), *Expectimax* will be called again, and the process illustrated will be repeated for a total of $\frac{n}{2}$ steps, when called first. This will then be done for each branch, and each sub-sequent branch. Since we are interested in calculating the utility of the probable outcomes (and not just moves without random placed numbers), the depth must be a an even number. The reason for using this algorithm is that we are able to take into account the random nature of the game, and calculate the **probable value** of a given action, in contrast to for example, the implementation of*Minimax* in 2048, where the **worst case scenario** will be chosen by the "*opponent*", but might have very low probability of happening, thus possibly excluding a scenario that has a high probability of being a desired outcome.

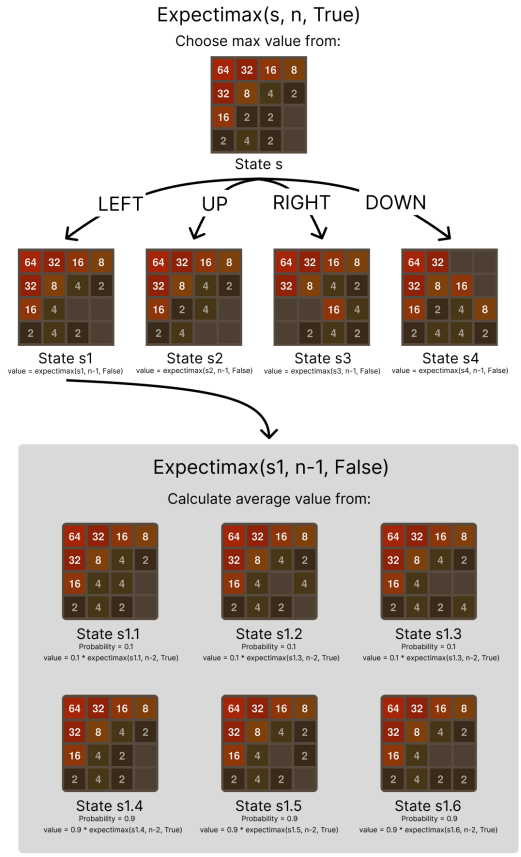


Fig. 1: Expectimax calls

[a] Global Edition - Stuart Russell and Peter Norvig, 2022. Section 16.2.4

[b] For obvious reasons only one branch of possible actions has been unfolded

7 Heuristics

We decided on applying three heuristics:

- Snake-pattern
- Empty cells
- Smoothness

The *snake-pattern* heuristic favors higher numbers from top row to bottom, alternating from left to right, and vice versa in each row, while the *empty cells* heuristic favors a higher number of empty cells, and favors their position opposite of the snake pattern. This results in the following **weight matrices**, with parameter a and b to adjust how strong each heuristic influences the utility function:

$$\text{snake} = \begin{pmatrix} a^{15} & a^{14} & a^{13} & a^{12} \\ a^{8} & a^{9} & a^{10} & a^{11} \\ a^{7} & a^{6} & a^{5} & a^{4} \\ a^{0} & a^{1} & a^{2} & a^{3} \end{pmatrix} \quad \text{empty} = \begin{pmatrix} b^{0} & b^{1} & b^{2} & b^{3} \\ b^{7} & b^{6} & b^{5} & b^{4} \\ b^{8} & b^{9} & b^{10} & b^{11} \\ b^{15} & b^{14} & b^{13} & b^{12} \end{pmatrix}$$

Below is shown how utility is calculated using the snake heuristic and empty cells, where *grid* is our board, represented as a 4x4 matrix:

$$utility_{snake} = \sum_{i=0}^{3} \sum_{j=0}^{3} grid_{i,j} \cdot snake_{i,j}$$

$$utility_{empty} = \sum_{i=0}^{3} \sum_{j=0}^{3} \text{if}(grid_{i,j} = 0)\ empty_{i,j}\ \text{ else } 0$$

For the smoothness heuristic, we calculate how similar neighboring numbers are in value, i.e. the absolute difference between them - big differences will result in lower utility:

$$utility_{smooth} = \sum_{i=0}^{3} \sum_{j=0}^{3} -\text{abs}(\text{neighborsDiff}(grid_{i,j}))$$

These 3 heuristics can then be added together or used separately, depending on which heuristics are wished to be implemented.

8 Parameters

There are several adjustable parameters in the AI player. Probably the most import being the depth, which decides the amounts of turns the AI takes and the randomly inserted tiles (nature). The depth should in theory result in better moves the larger it is, however the speed is severely limited by processing power, which is why all the benchmarks in Table 1 have been calculated with a depth of 4. The parameters a and b used in the snake heuristic and empty cell heuristic are also adjustable. Increasing their values means increasing

their importance when calculating the final utility.

a and b value	1	2	3	4	5	6	7	8	9	10
Snake heuristic	1767.2	33208	31008.8	33470.4	34832	25361.6	22509.6	31196.8	34556	27075.2
Empty cell heuristic	8166.4	4513.6	6011.2	5020	5070.4	3260	3972.8	3102.4	4060.8	2684.8
Smoothness heuristic	5097.6	-	-	-	-	-	-	-	-	-
Random moves	942.64	-	-	-	-	-	-	-	-	-

Table 1: Highscore averages for different heuristics

Table 1 contains the average highscore of five games played with the 3 heuristic functions being used individually with different values of a and b, as well as the average highscore of 10 games played with totally random moves.
We can see how the snake heuristic clearly seems to yield the best results whenever a does not equal 1, which makes sense since when a and b are 1, the values of the matrices all become 1. On the other hand, the empty cell heuristic seems to be the opposite. Whenever the values of the empty cell matrix are not the same (1 in this case), it seems to get worse with larger values of b. The smoothness heuristic does not have any adjustable variables, like the snake heuristic and empty cell heuristic, besides being able to be used or not. Its average highscore is less than the highscores of the other heuristics but is still far better than a game being played with purely random moves.

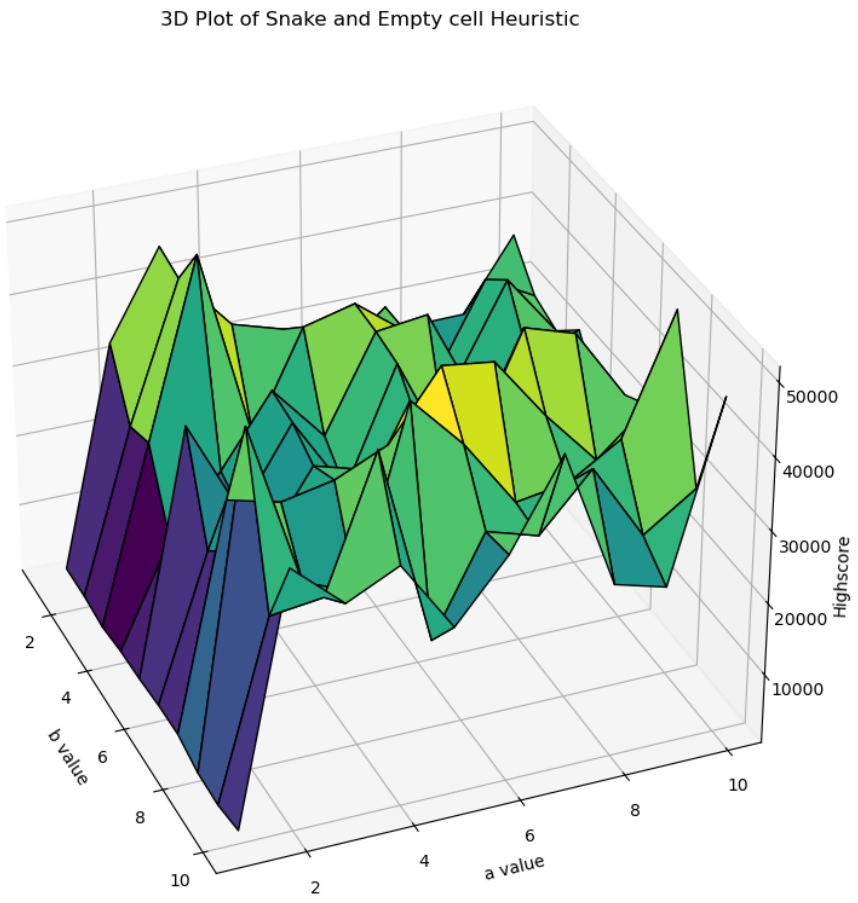


Fig. 2: 3D Plot of Snake Heuristic and Empty cell Heuristic

Figure 2 shows the average highscores of 5 different games each with different a and b values. It seems to show that as long as the a value is larger than 1, different a and b values does not have much difference that could not be explained by randomness.

9 Future Works

The biggest factor for how big our depth parameter can be and still run at a reasonable speed is the amount of states it has to consider. The speed could be significantly improved if we implemented Monte Carlo tree search to remove the unnecessary branches of the equimax tree search that we used.

By using a profiler, we could also identify the calculations being done to make moves on the board as a huge bottleneck. When running the game with a depth of 5, 44% of time was spent on compress, merge, reverse, transpose, and can_move functions. This could be improved by using a lookup table, which already contains all the solutions for every move done on a row or column, so no calculation would have to be done.

A better data structure than our current 2-dimensional array of integers to represent a board could be using one integer to represent several cells at once by using only a part of the integer bits to represent a cell.

Making the AI in Python might not have been the best idea either, considering Python is a very slow programming language to build algorithms from scratch in. A better language would have been C, although it would have taken us longer to implement everything.