Danmarks Tekniske Universitet

02180 Introduction to Artificial Intelligence

Board Game Assignment

March 29th 2021

Group 50

Enny Tao	s174491
Lasse Bredmose	s175395
Sofie Bach	s174500
Filippo Bentivoglio	s210299

Contents

1 The Game of 2048

1.1 Introduction

For this assignment, we have chosen to work with the game 2048. 2048 is an offline computer board game, usually played on a standard 4×4 grid (referred to as the board). The goal of the game is to reach a tile with the value of 2048, done by fusing two tiles of the same value resulting in a new tile with twice the value. A ongoing game can be seen in Figure 1 where the starting board is presented to the right.

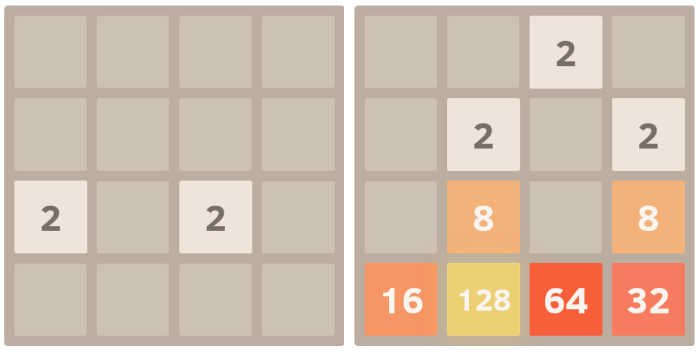

Figure 1: Left: Starting board of 2048. Right: Ongoing game of 2048.

1.2 Rules of the game

The game is played using the up/down/right/left arrows on the player's keyboard. Each key indicates which direction the player wants the tiles to move.

- An empty board will always start with two tiles randomly positioned where the probability of a tile being in each field is uniformly distributed. These two tiles can each have the value of 2 or 4, with probabilities 0.9 and 0.1 of appearing, respectively.
- A move is the chosen direction. Once applied, all tiles will move in that direction.
- A move is only valid if at least one tile can be moved in the desired direction.
- After each move, a new tile will appear in a random empty field. This tile will either have the value of 2 or 4 (with probabilities 0.1 and 0.9, respectively).
- If two tiles have the same value and are shifted in the same direction, the two tiles will merge resulting in a tile with twice the value.
- If two tiles have different values, it is impossible for the tiles to merge. Thus a tile cannot pass another tile and its movement will be stopped once it encounters another tile.
- A game is won if the player reaches a score of 2048

1.3 What kind of game?

2048 is a single player game, which in theory can be expanded as a multiplayer game if handing out the random tiles is regarded as a player. The game is stochastic in the sense that the player does not know where the next tile will be inserted and what its value will be. Therefore, the algorithm to apply to the game has to incorporate the stochastic element. The game is turn-based, as a new tile will be inserted after one has moved the tiles in a direction. The game has full observability as there is no hidden information about the state that the player is in.

1.4 State space

We start by making an upper bound for the state space of the game. To reach 2048, the player has to merge a tile 11 times ($2048 = 2^{11}$). However, once 2048 is reached, the game is over. Therefore, it is not possible for multiple tiles with the value of 2048. As previously stated, a normal board has 16 fields meaning each field can have 11 different values (including no tile on a field). Thus we have $11^{16} \cdot 16 \approx 7.35 \cdot 10^{17}$ different states in theory. This is just short of checkers (10^{18}) and a little above kalaha (10^{13}) [1].

2 Implementation

2.1 Problem description

Before implementing, the different elements of our game have to be modelled. Using the standard notation presented in the course, we can describe our game as follows:

s_0 : Initial state. The initial state for our game is an empty board with two tiles randomly positioned. These two tiles can each either have the value of 2 or 4.

PLAYER(s): The player whose turn it is in a state s. 2048 is a single-player game, and therefore it is the same player, MAX, who makes all the moves. One can however regard the one giving the random tiles as player, namely a player denoted by CHANCE.

ACTIONS(s): Legal moves in state s, denoted by North, East, West and South (N, E, W and S), corresponding to the keyboard arrow directions. A move is legal if it is able to change the current state.

RESULT(s,a): The resulting state of applying the action a to the state s. It gives us a new state where the action has been executed.

TERMINAL-TEST(s): The terminal test checks if a player has reached a tile with the value of 2048. If that is the case then the game is over, otherwise the game will go on.

UTILITY(s,p): At a certain depth, the utility function examines the board and provides an estimate of how desirable the state is. The heuristics used for this estimate is described in a later section.

2.2 States and moves

2048 is a game played on a simple 4×4 grid. Thus in our implementation, the states are represented as a 4 by 4 matrix, i.e. a 2D array. In the beginning, apart from the matrix elements containing the two randomly generated numbers, each element in the matrix has a value of 0. The value will then increase if two tiles merge, if a tile is randomly generated, or if a tile is moved to that field.

The four different kinds of moves are represented as N, E, W, and S, each corresponding to a direction where the tiles can move. Once a move is executed, i.e. an action is applied, the 2D array is updated accordingly. The way the 2D array is updated is through checking whether there are two tiles next to each other that are being merged through an action due to those having the same value. If so, their values are added and the tiles are replaced by a new tile. Once we have implemented the method such that it is applicable for a direction, i.e. W, it is possible to transpose and flip the 2D matrix such that it is applicable for the remaining directions.

2.3 Algorithms

For the game, we have chosen to implement the ExpectiMax algorithm [2]. There are other suitable algorithms that can be used such as the MiniMax algorithm and Monte Carlo Tree Search (MCTS). However, as probabilistic decisions are made, the ExpectiMax algorithm is better suited than MiniMax. A simplified example of the ExpectiMax algorithm is seen in Figure 2 for a 3×3 grid, where for a certain move, one random tile i.e. 4 or 2 is being inserted on the empty fields of the board. The probability of each random tile appearing is denoted by the directed edges. A utility/heuristic function is then used to evaluate how desirable that move is.

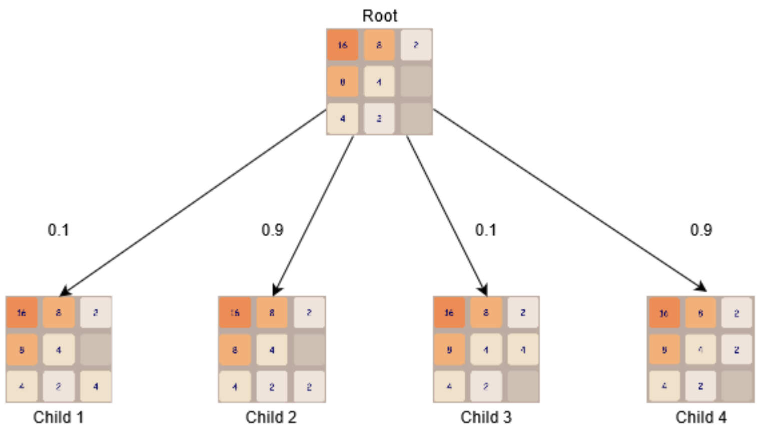

Figure 2: Example of the ExpectiMax algorithm for 3×3 grid.

Mathematically, the algorithm is described as presented in the lecture slides [3]:

$$\textsc{ExpectiMax}(s) =$$
$$\begin{cases} \textsc{Utility}(s, \textsc{max}) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in \textsc{Actions}(s)} \textsc{ExpectiMax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \sum_{a \in \textsc{Actions}(s)} P(a,s) \cdot \textsc{ExpectiMax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

2.4 Heuristics and Evaluation

In order to implement the ExpectiMax algorithm, we need a heuristic/evaluation function, which is denoted by Utility in our previous description. This function is used once the algorithm reaches a certain depth or is at the end of the search tree. At this step, we need a way to evaluate the score of the current state. In order to design and develop the heuristic, we have sought guidance in the strategies that we use ourselves when playing the game. The following heuristics are the core elements we see as necessary for completion of the game.

2.4.1 Snake evaluation

One of the main strategies that are known for the game is to put the tile with the highest value in a corner, follow it with the second-highest next to it and so on. We consider two ways of evaluation. The first way is denoted by the *snake evaluation.* The upper left field has the highest value when evaluating the board, the field right to it has the second highest value and so on as illustrated on the left of Figure 3. This makes it more desirable to have the highest-valued tiles in the fields with the maximum values. In this way, it is easier to merge the tiles by choosing the actions in the opposite direction of the snake.

2.4.2 Diagonal evaluation

The second way of evaluating the board is denoted by the *diagonal evaluation.* Here, the board is evaluated by letting the upper left field have the highest value and the two fields next to it have the second highest value and so on as illustrated on the right of Figure 3. In this way, the highest-valued tiles will be in the upper left corner and the lower-valued tiles in the lower right corner resulting in making it easier to merge the tiles with the same value.

Both the snake and the diagonal evaluations are implemented using a table of importance, which is a 4 by 4 matrix, where the fields of higher importance have higher values. Mathematically, the evaluation of the state can be expressed as:

$$\textsc{Eval}(s) = \sum_{i=1}^{4} \sum_{j=1}^{4} B_s(i,j) \cdot \textsc{Importance}(i,j), \quad \text{where } B_s \text{ is the board at state } s$$

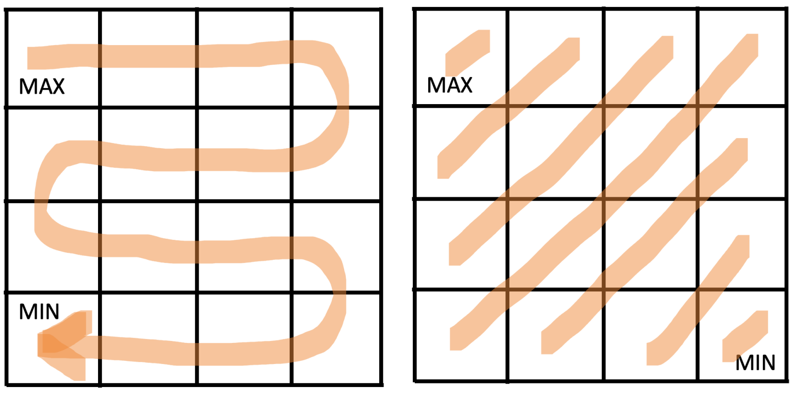

Figure 3: Left: Snake evaluation. Right: Diagonal evaluation

2.4.3 Empty fields

The criteria for losing the game, is when there are no applicable moves. When there is an empty field, there is always an applicable move in the corresponding row and column of the empty field. Therefore, it is desirable to have as many empty fields as possible. A downside of having many empty fields is that a new tile can be inserted in many different fields, making it harder to predict where the next inserted tile will be. In addition, the search tree will increase, since every empty field spawns a new child in the search tree - when a tile is stochastically inserted. For implementation, we have implemented a simple double for-loop which counts each space where there is an empty field.

2.4.4 Monotonically increasing rows and columns

The rows have to be monotonically increasing. If we assume that the largest value will be in the field of the top left corner, then for the snake evaluation, the value of the fields in the top row has to be in a monotonically increasing order towards the left. In the second row, the value has to be monotonically increasing to the right and so forth. For the diagonal evaluation, each value in a row has to be in monotonically increasing order towards the left. The analogous holds for columns, where each column has to be monotonically increasing towards the top.

2.5 Parameter adjustments and tests

We tested the different evaluation methods i.e. the snake evaluation and the diagonal evaluation. Both methods were tested with different search depths. Due to the algorithm running slowly at higher search depths, the number of games played is significantly lower. Furthermore, trials were completed where an adaptive search depth depending on the number of empty cells was deployed. This is denoted by AD in our results seen in Table 1 and Table 2 for the diagonal and the snake evaluations, respectively. We count it as a win each time 2048 has reached, but we let the algorithm run up to the highest reachable number.

2.5.1 Diagonal

Depth	Games played	Max score	Avg. moves/sec	Win rate	4096	8192
2	1500	71988	35.003	39.4 %	20/1500	0
3	600	57384	7.597	40.0 %	4/600	0/600
4	10	123864	0.993	100.0 %	3/10	3/10
AD	10	132940	0.88	100.0 %	3/10	2/10

Table 1: Results obtained with the diagonal evaluation

2.5.2 Snake

Depth	Games played	Max score	Avg. moves/sec	Win rate	4096	8192
2	1500	108248	26.53	72.34%	201/1500	1/1500
3	448	80108	4.11	69.64%	73/280	0/280
4	10	79968	0.60	90%	6/10	0/10
AD	10	81332	1.195	100.0 %	4/10	0/10

Table 2: Results obtained with the snake evaluation

We see that we reach the highest scores with the Diagonal heuristic. We also see that the diagonal seems to work better with high depths, while with depth 2 and depth 3, the snake method is favourable.

2.6 Discussion and future work

Our solution can be improved both in terms of velocity (moves per second) and the heuristics used. When running the tests, we saw that at high search depths, the algorithm was very slow. One way of solving this is through making our code more efficient, as in our implementation, the board array is traversed both in terms of rows and columns. This could be related to the way we have represented the board, which could instead be represented as (r, c, v), where r denotes the row index, c the column index, and v the value.

For the evaluation, the we have completed trials of different values in the table of importance. This can be a tedious task to do manually, and it would thus be an advantage if hyperparameterization was used to find the optimal weight of each field. This could also be used for finding the optimal importance to give for other evaluations e.g. empty fields.

A limit of our evaluation function is that it will always keep the tiles in one corner. It might sometimes be an advantage going into another corner if one is very unlucky with the spawn of a new tile. This would be an adaptive heuristic, which expands our current heuristic, but also considers the remaining 3 corners as advantageous.

Furthermore, we could expand with trials of other algorithms such as MCTS, which was explained in the lectures. This would allow us to make an estimate of the optimal algorithm to use for the game of 2048.

References

[1] https://en.wikipedia.org/wiki/Game_complexity, Last visited on 15/03-2020.

[2] Russell & Norvig: Artificial Intelligence - A Modern Approach. 3rd edition, Global Edition. Prentice Hall, 2010.

[3] Gierasimczuk, Nina: Introduction to Artificial Intelligence - Lecture 5: Adversarial Search. Technical University of Denmark.