Techincal University of Denmark

02180 Introduction to Artificial Intelligence

Assignment 1: A.I. Solution To Solve The Game Of 2048

Authors

Edrin Molla - s242628
Gisle Garen - s242715
Nacho Ripoll Gonzalez - s242875
Paulo Ricardo Beckhauser de Araujo - s242779

March 24, 2025

1 Game Rules

2048 is a single-player sliding tile puzzle game played on a 4×4 grid. The objective is to combine tiles with numbers on them to reach a tile with value 2048. The rules are as follows:

- The game starts with two tiles randomly placed on the board. Each tile is either a 2 or a 4, with 2 being more frequent.
- The player can slide all tiles in one of four directions: up, down, left, or right. Tiles move as far as possible in the selected direction.
- When two tiles of the same number collide during a move, they merge into a single tile with twice the value.
- After each move, a new tile (2 or more rarely, 4) is added to a random empty spot on the board.
- The game ends when the board is full and no merges are possible.

2 Game Characteristics and Implications for AI

- **Single-player**: AI development focuses on optimizing move sequences rather than countering an opponent.
- **Non-zero-sum**: With no direct opponent, the goal is not to minimize another player's score but to maximize the player's own performance.
- **Turn-based**: The game proceeds in turns, making it well-suited for search-based decision-making algorithms.
- **Full Observability**: The entire game state (grid and tile values) is always visible to the AI, removing uncertainty about current conditions.
- **Non-deterministic**: The game introduces randomness through tile spawning after each move.

Given these characteristics, AI strategies must account for uncertainty in future board states while optimizing moves to preserve merging opportunities and space for future tiles.

3 State Space Estimation and Implications for AI

The state space of 2048 is large due to the combination of tile placements and values. We estimate an upper bound on the number of states as follows:

- A 4×4 board has 16 positions where tiles can appear.
- Each tile can take on values from the set {0, 2, 4, 8, ..., 2048, ...} in practical play.
- According to [1], the highest theoretical tile achievable is 131,072, which is approximately 2^{17}. This results in a total of $\{0, 2^1, 2^2, 2^3, \ldots, 2^{17}\}$, leading to 18 choices per tile.
- Since each of the 16 tiles on the board can independently take any of these 18 values, the theoretical upper bound is at most 18^{16}.
- Converted to base 10, this results in an upper bound of approximately 10^{20} number of

possible state spaces, a very large number indeed.

Given the immense state space, a full search method is infeasible. Instead, AI approaches for 2048 typically uses heuristics to evaluate board configurations, which we mention in chapter 7.

4 Formal Game Representation

To describe the game formally, we define the following elements:

- **Initial state** (s_0): The game starts with a 4×4 grid containing two randomly placed tiles, either a 2 or a 4.
- **Players** (P): Since 2048 is a single-player game, there is only one player.
- **Actions** ($A(s)$): The player can take one of four possible actions at any state: `UP`, `DOWN`, `LEFT`, or `RIGHT`, which slide the board in the respective direction.
- **Result function** ($Result(s, a)$): The result of an action is a new state where tiles move in the chosen direction, merging when possible, and a new tile (either 2 or 4) is added to a random empty position.
- **Terminal test** ($Terminal - Test(s)$): The game ends when no valid moves remain, meaning the board is full and no adjacent tiles can merge.
- **Utility / Evaluation function** ($Utility(s, p)$): The utility of a state can be defined as the highest tile reached or a score based on the sum of all merged tiles, favoring strategies that enable higher-value tile creation.

5 Representation of States and Moves in 2048 AI

In the 2048 game, the game state is represented as a 4×4 grid containing numerical values corresponding to tile values. The AI interprets this grid as a two-dimensional array where each element represents a tile's value, and empty tiles are denoted by zeros. The current score is stored separately to track the game's progress.

Moves are represented as discrete actions: extttleft, extttright, extttup, and extttdown. Each move modifies the grid according to the game's merging and sliding rules. The AI evaluates these moves by simulating the resulting game state after each action, determining the optimal move based on heuristic evaluation functions or search algorithms.

For AI decision-making, deterministic algorithms such as Minimax and Expectimax are employed. These approaches model the game as a decision tree where player moves correspond to decision nodes, and the random appearance of new tiles corresponds to chance nodes. Since all tile values and positions are known at each step, there is no need to maintain belief states. The evaluation function considers heuristics such as tile monotonicity, smoothness, empty cell count, and the highest tile position to guide decision-making. The AI relies purely on deterministic state transitions and probabilistic modeling of tile generation to optimize move selection.

Thus, representing the game state as a fixed grid structure, rather than belief states, is sufficient for AI decision-making in 2048.

6 AI Methods and Algorithm Choices

For solving 2048, we explored several AI methods, where Expectimax, Minimax and Monte Carlo Tree Search were all considered, but only Minmax and Expectimax were implemented:

Implemented Algorithms

Expectimax Algorithm: Since 2048 involves randomness due to tile spawning, Expectimax is well-suited as it extends Minimax to stochastic environments. In our implementation, the AI selects moves by evaluating board configurations based on heuristic values, considering empty cells, tile merging potential, and the highest tile present, all with dynamic depth limits.

Minimax Algorithm: We implemented a basic Minimax algorithm also with a depth-limited search. However, Minimax is less effective in 2048 due to its deterministic adversarial assumption, which does not align well with the stochastic tile placements in the game. As MiniMax in 2048 hasn't been covered in the lectures, we'll briefly go through its structure in our case. Appendix A shows a sketch of how the minmax algorithm works where the min node, which places the random 2s and 4s, will pick the one that leads to the lowest score. Max picks the highest and the algorithm evaluates all possible states as long as it stays in the confines of the depth limit. Inspired by the course, we have also decided to implement alpha beta pruning to speed up computation with minmax.

7 Notation of Heuristics and Evaluation Functions

Since searching all future states in 2048 is computationally infeasible due to its large branching factor and stochastic nature, we use heuristic evaluation functions to estimate the utility of non-terminal states in both Minimax and Expectimax.

$$H(s) = \text{score}(s) + w_1 E(s) + w_2 M(s) + w_3 S(s) + w_4 C(s) \quad (1)$$

Our evaluation function [2] is a weighted sum of board features along with the previous state score:

where $score(s)$ is the score from previous states, $E(s)$ is the number of empty tiles, $M(s)$ is the monotonicity score, $S(s)$ is the smoothness score, and $C(s)$ is the corner bonus.

Empty Tiles ($E(s)$)

This component counts the number of zero-valued cells in the grid. It is calculated as:

$$E(s) = \sum_{i=1}^{4}\sum_{j=1}^{4}(s[i][j] = 0) \tag{2}$$

Monotonicity ($M(s)$)

Heuristic to see if we have clusters in the top right corner where each element from left to right gets smaller and smaller. We do the same only for the columns from top to bottom.

$$\begin{aligned} M(s) &= \sum_{i=1}^{4}\sum_{j=1}^{3}(s[i][j] \\ &\geq s[i][j+1])s[i][j] + \sum_{i=1}^{3}\sum_{j=1}^{4}(s[i][j] \\ &\geq s[i+1][j])s[i][j] \end{aligned} \tag{3}$$

Smoothness ($S(s)$)

Since adjacent tiles with the same value are required to merge, we penalize large differences between neighboring tiles. The closer it is to 0, the more uniform the grid values and the more negative it is, the different the values across the grid cells. We do this row and column wise.

$$\begin{aligned} S(s) = -\sum_{i=1}^{4}\sum_{j=1}^{3}|s[i][j] - s[i][j \\ +1]| - \sum_{i=1}^{3}\sum_{j=1}^{4}|s[i][j] - s[i+1][j]| \end{aligned} \tag{4}$$

Corner Bonus ($C(s)$)

This heuristic rewards placing the largest tile in a corner:

$$C(s) = \begin{cases} T(s), & \text{if } T(s) \in \{s[0,0], s[0,3], s[3,0], s[3,3]\} \\ -T(s), & \text{otherwise} \end{cases} \tag{5}$$

where:

$$T(s) = \max_{1\leq i\leq 4}\max_{1\leq j\leq 4} s[i,j] \tag{6}$$

8 Finetuning of Algorithms and Benchmarks

This section presents a summary of the benchmark results, showing the average number of moves, the average time for completed runs, and the number of successful runs out of five for each algorithm.

Table 1: Benchmark results summary depths

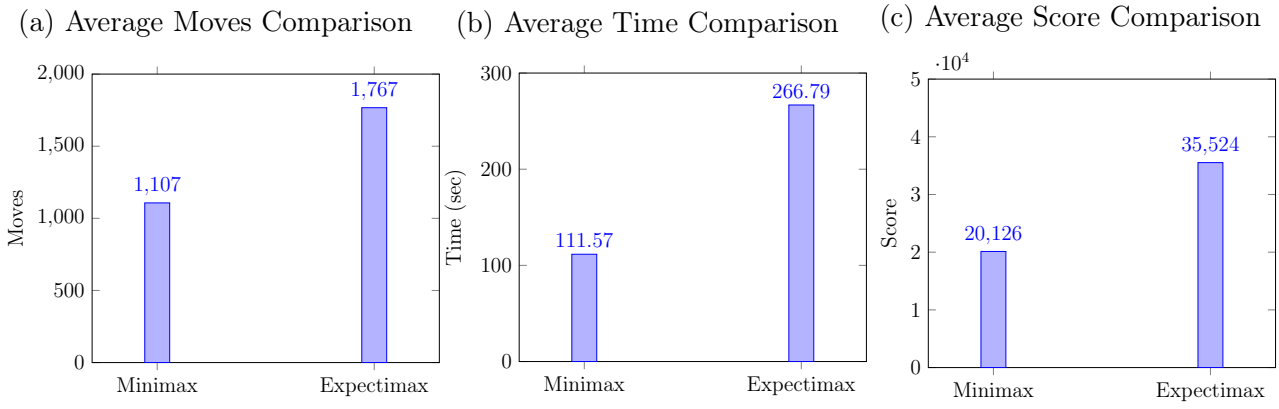
Algorithm	Avg. Moves (Passed)	Avg. Time (Passed)	Pass Count
Dynamic Depth Expectimax	969	224.4 s	5/5
Dynamic Depth Minimax	924	147.5 s	3/5
MiniMax Static Depth 3	N/A	N/A	0/5
MiniMax Static Depth 4	1092	128 s	1/5
MiniMax Static Depth 5	1010.5	229.5 s	2/5
Expectimax Static Depth 3	991.5	118.5 s	3/5
Expectimax Static Depth 4	974.25	129.5 s	4/5
Expectimax Static Depth 5	957.6	647.2 s	5/5

Table 2: Benchmark results summary heuristic weights

Algorithm	**Avg. Moves (Passed)**	**Avg. Time (Passed)**	**Pass Count**
Dynamic Depth Minimax ($w_1 = 10$, $w_2 = 1$, $w_3 = 1$, $w_4 = 1$)	1193	218 s	1/5
Dynamic Depth Expectimax ($w_1 = 10$, $w_2 = 1$, $w_3 = 1$, $w_4 = 1$)	1233	331 s	2/5
Dynamic Depth Expectimax ($w_1 = 50$, $w_2 = 10$, $w_3 = 5$, $w_4 = 10$)	969	224.4 s	5/5
Dynamic Depth Minimax ($w_1 = 50$, $w_2 = 10$, $w_3 = 5$, $w_4 = 10$)	924	147.5 s	3/5
Dynamic Depth Expectimax ($w_1 = 100$, $w_2 = 20$, $w_3 = 10$, $w_4 = 20$)	977	227.2 s	5/5
Dynamic Depth Minimax ($w_1 = 100$, $w_2 = 20$, $w_3 = 10$, $w_4 = 20$)	940	149.1 s	3/5

Due to the long duration of running each algorithm and poor computational power, we decided to select our parameters to benchmark the final algorithms by testing the different depths and heuristics until they reached 2048, in which the program would terminate. We measured the computational time and average number of moves. We ended up going for dynamic depths because it balances computational speed and accuracy. As for the heuristics, the lower weights made the algorithm "learn" a lot less and thus took more moves to pass and would fail more than the other weights. Prior to benchmarking, we played around the parameters and landed on ($w_1 = 50$, $w_2 = 10$, $w_3 = 5$, $w_4 = 10$) and thus tried something lower and higher. We didn't see too much of a difference between the higher value and the middle value heuristics so we ended up testing the depths as well as the final benchmark with the middle value heuristics.

Figure 1: Comparison of Minimax and Expectimax Performance



With the dynamic depths and the moderate heuristics, we can observe that there is a tradeoff between minimax and expectimax. Minimax' main strength comes from it's computational speed, thanks to the alpha beta pruning ability. Expectimax, although it uses longer time per move, consistently achieves larger scores than minimax. After running each algorithm five times and averaging its scores, we can see that the expectimax algorithm was

on average around 75 percent higher than minimax. Minimax' min node will always pick the worst case whilst expectimax takes the average of all the generated child nodes, thereby balancing risk more, which was reflective of its performance. Overall, the two algorithms managed to frequently hit the 2048 tile mark, which is notable given that none of us in the group could reach it ourselves.

9 Future Work

Although we managed to implement minimax and expectimax algorithms with the goal of reaching 2048 on the highest tile, the capacity for improvement is still available.

Monte Carlo Tree Search (MCTS)

Earlier in the project preparation, we had plans to implement Monte Carlo Tree Search, which has proven effective in stochastic environments like 2048. MCTS balances exploration and exploitation by simulating many random playouts from a given state to guide the search. However, due to primarily time contrains, we chose not to include MCTS in the current work, focusing instead on properly implementing minimax and expectimax.

Performance Optimizations

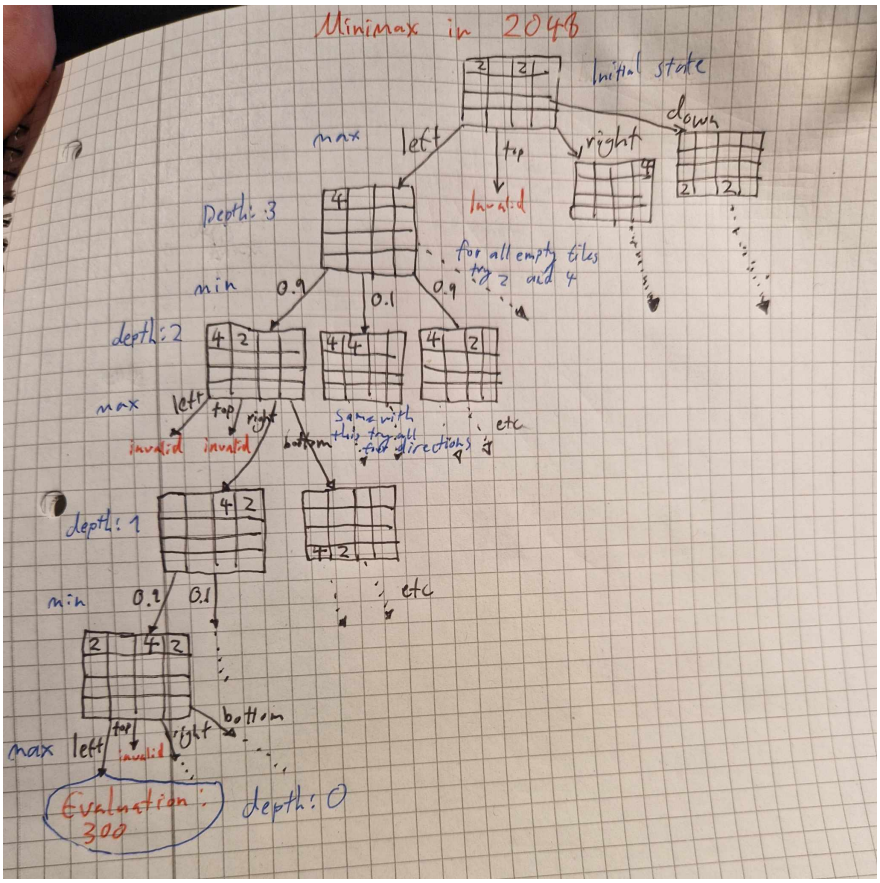
Our current implementation uses plain Python and could benefit from performance optimizations. This includes rewriting critical parts in a faster language (e.g., Cython or C++), parallelizing the search process, or implementing pruning strategies to reduce the number of evaluated nodes. These enhancements could enable deeper search depths or faster real-time decisions.

Conclusion

Our current solution provides a solid baseline and a clear framework for further experimentation. Incorporating the above techniques could significantly improve both the efficiency and effectiveness of the AI player for 2048.

A Drawing of Minimax in 2048

Figure 2: Minimax Tree Representation in 2048



References

[1] Gettysburg College. The mathematics behind 2048. https://cupola.gettysburg.edu/cgi/viewcontent.cgi?article=1025&context=csfac, 2023.

[2] Stack Overflow Community. What is the optimal algorithm for the game 2048? https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048, 2014.