
IMPLEMENTATION OF 2048

Fujiura Taishu
Kawsar Nabawi
Mads R. Harder
Maria T. Hagedorn

24. March 2025



1 Introduction of Chosen Game

The implemented game is the puzzle game **2048**, where the objective is to slide numbered tiles on a 4×4 grid to combine them into a tile showing 2048. Players may continue beyond this goal to increase their score. 2048 is a *single-player, turn-based* game with *perfect information* and *stochastic dynamics*. While the full board is always visible, a new tile (2 with 90% probability, 4 with 10% probability) spawns at a random empty position after each move, introducing uncertainty for AI decision-making.

To estimate the game's complexity, we approximate the state space. A naive upper bound is $18^{16} \approx 4.7 \cdot 10^{20}$, assuming each of the 16 cells can take 18 values (empty, 2, 4, ..., 2^{17}). In practice, tiles rarely exceed 2048 (2^{11}), making $12^{16} \approx 2.8 \cdot 10^{17}$ a more realistic upper bound. This highlights the large but tractable search space an AI must manage.

2 Algorithm Considerations

Due to the large state space, exact methods such as value iteration are impractical. Storing values for all possible board states would require over 10^{17} entries, exceeding 100 petabytes even at 1 byte per entry. We therefore adopt two approximate search algorithms suited for 2048's single-player, stochastic nature:

- **Expectimax**

The algorithm models decision making as alternating Max nodes (player moves) and Chance nodes (random tile spawns: 2 with 90% probability, 4 with 10% probability). Expected values are computed by averaging over all possible outcomes given a heuristic function. To manage the state space, the tree is searched to a fixed depth.

- **Monte Carlo Tree Search (MCTS)**

MCTS explores the space through *selection*, *expansion*, *simulation*, and *backpropagation*, using the UCT formula to balance exploration and exploitation. We also implemented a simpler variant, **Flat Monte Carlo**, which performs several random playouts per move and selects the direction with the highest average score. Unlike full MCTS, it does not build or reuse a search tree.

2.1 A Run through the Algorithms

To better illustrate how the algorithms work in the context of 2048, we provide a small walk through of Expectimax, Monte Carlo Tree Search (MCTS), and Flat Monte Carlo applied to a simplified board state.

2.1.1 Expectimax

The graph in figure 1 shows an example of the expectimax algorithm. The values shown at each node is based on the cumulative game score and the weighted probabilities for placement of tiles.

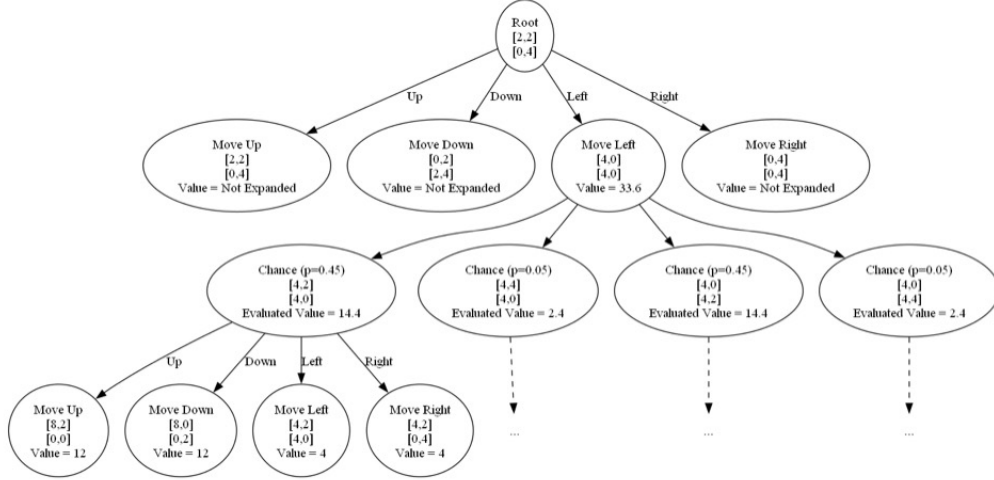


Figure 1: Simple partial example of the Expectimax algorithm with depth 3 for a smaller version of 2048.

The algorithm alternates between Max nodes (player moves) and Chance nodes (random tile spawns). At Max nodes, it selects the direction that yields the highest expected value. At Chance nodes, it models all possible tile insertions and computes the expected value as a weighted sum over outcomes. For example, in the “left” move branch, two outcomes with $p = 0.45$ and value 14.4, and two with $p = 0.05$ and value 2.4, result in a combined value of 33.6. While only one expanded branch is shown for illustration, the algorithm recursively evaluates all possible branches to select the optimal move.

2.1.2 Flat Monte Carlo and Monte Carlo Tree Search

We implemented two Monte Carlo-based strategies for 2048: Flat Monte Carlo (MC) and Monte Carlo Tree Search (MCTS with UCT). Both estimate the value of each move through repeated random playouts, but differ in how they store and reuse information.

2.2 Expectimax Heuristics

We evaluated five heuristics for Expectimax: *score*, *empty-cell count*, *smoothness*, *monotonicity*, and *merge potential*, each applied to a board state

$$B \in \mathbb{Z}_{\geq 0}^{4 \times 4},$$

where $B_{i,j} = 0$ denotes an empty cell. The *score* heuristic selects the move yielding the highest immediate game score. *Smoothness* augments this by favoring evenly distributed tiles.

2.2.1 Empty-Cell Count

This heuristic encourages flexible board states with lower risk of getting stuck. It returns the number of empty cells:

$$H_{\text{empty}}(B) = |\{(i, j) \mid B_{i,j} = 0\}|. \quad (1)$$

2.2.2 Smoothness

Smoothness measures how similar adjacent tiles are, promoting merge-friendly arrangements. It penalizes large differences between neighbors:

$$P_{\text{smooth}}(B) = \sum_{i=0}^3 \sum_{j=0}^3 \sum_{(di,dj) \in \{(0,1),(1,0)\}} \mathbf{1}_{B_{i,j}, B_{i+di, j+dj} > 0} \cdot |v_{i,j} - v_{i+di, j+dj}|, \quad (2)$$

where

$$v_{i,j} = \begin{cases} \log_2(B_{i,j}) & B_{i,j} > 0 \\ \text{undefined} & B_{i,j} = 0 \end{cases}.$$

The heuristic is then:

$$H_{\text{smooth}}(B) = -P_{\text{smooth}}(B).$$

Higher (less negative) values indicate smoother boards. We used a weighted combination of this penalty and the game score (weight = 1).

2.2.3 Monotonicity (Snake Pattern)

The idea is to keep the largest tiles in one corner and having the board values increase steadily along rows. We have based our heuristic on a snake pattern where we have a weight matrix:

$$W = \begin{bmatrix} 2^4 & 2^3 & 2^2 & 2 \\ 2^5 & 2^6 & 2^7 & 2^8 \\ 2^{12} & 2^{11} & 2^{10} & 2^9 \\ 2^{13} & 2^{14} & 2^{15} & 2^{16} \end{bmatrix},$$

Then the heuristic function is the Frobenius inner product of the weight matrix and the board

$$H_{\text{snake}}(B) = \langle B, W \rangle_F = \sum_{i=0}^3 \sum_{j=0}^3 B_{i,j} W_{i,j}. \quad (3)$$

This rewards placing large tiles in the bottom right corner and otherwise following a snake pattern along the board.

2.2.4 Merge Potential

The merge heuristic looks for potential merges by looking at adjacent tiles that can be merged. It will then reward a board configuration with more potential merges available. The function counts every adjacent pair of identical, non-zero tiles:

$$H_{\text{merge}}(B) = \sum_{i=0}^3 \sum_{j=0}^2 1 \{B_{i,j} > 0, B_{i,j} = B_{i,j+1}\} + \sum_{i=0}^2 \sum_{j=0}^3 1 \{B_{i,j} > 0, B_{i,j} = B_{i+1,j}\}. \quad (4)$$

So a larger value means more immediate merge opportunities.

2.3 MCTS Utility Estimation and Simulation Strategy

During MCTS implementation, utility is estimated through repeated random simulations than explicit heuristics at leaf nodes. From each state, the algorithm performs random valid moves up to a fixed depth (which increases as the board fills), using the resulting score as a proxy for move value.

Simulation rewards are backpropagated to update visit counts and cumulative scores, allowing MCTS to prioritize consistently high-reward paths while maintaining exploration.

While some implementations use heuristics during rollouts (e.g., favoring empty tiles or monotonicity), we opted for purely random simulations to reduce complexity and avoid bias. Nonetheless, the combination of UCT and playouts reliably produced strong moves.

To ensure responsiveness in the GUI, we used 500 iterations and a simulation depth of 50. For slower or batch evaluations, higher values (e.g., 500–1000 iterations and depth 100–1000) typically yield stronger performance.

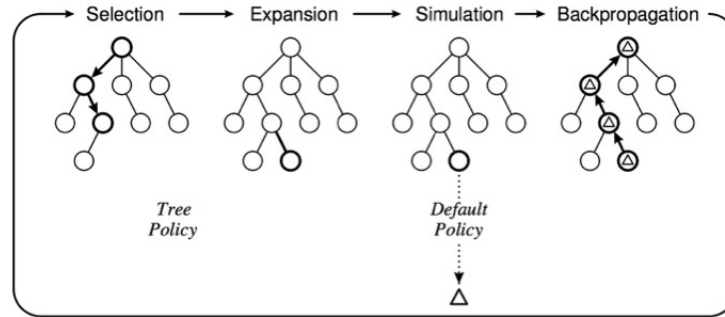


Figure 2: The four phases of Monte Carlo Tree Search: selection, expansion, simulation, and backpropagation. Figure source [1].

3 Implementation of Expectimax

3.1 Game Elements and State Representation

The initial state is a 4×4 grid of zeros, with two random tiles, 2 (90%) or 4 (10%), placed in empty cells. The two players are: the **Max player**, who selects a direction (UP, DOWN, LEFT, RIGHT) to slide and merge tiles, and the **Chance player**, who inserts a random tile after each valid move.

Each move involves:

1. Sliding tiles in the chosen direction.
2. Merging equal tiles and updating the score.
3. Adding a new tile if the board changes.

Expectimax evaluates all possible placements of 2 or 4 in empty cells, weighted by probability, to compute the expected utility. Terminal states occur when no moves or merges are possible. Utility is defined as the current score, while heuristics are used to evaluate non-terminal states. The search is bounded by a configurable depth to balance accuracy and runtime.

3.1.1 State and Move Representation

Each state comprises a 4×4 integer matrix (the board) and the current score. Moves are represented by an enumeration of the four directions.

4 Implementation of MCTS

4.1 Game Elements

MCTS shares most game elements with Expectimax (see Section 3.1). The utility function is the accumulated game score at the end of each simulation.

4.2 Algorithm Overview

Each iteration of `mcts_search` proceeds as follows:

1. **Selection:** Traverse the tree using `best_child()` and the UCT formula until the best (best value or less explored) leaf is reached.
2. **Expansion:** If the leaf is has been visited then it call `expand()` to generate valid children and select one.
3. **Simulation:** From the selected child it plays random moves until a terminal state or maximum rollout depth is reached.

4. **Backpropagation:** Propagate the final score upward, updating visit counts and rewards along the path.

After all iterations, the child of the root with the highest visit count is selected as the best move.

4.3 Implementation Considerations

Lazy Expansion: 4 child nodes are added per expansion (with a new random tile). We don't add every possible new random tile to control tree growth.

State Copying: Deep copies of game states ensure node independence.

Rollout Policy: Simulations use purely random moves; heuristics could replace this.

Memory Usage: Each node stores a game object (game score and a 4x4 board), its parent node, its children, the move to get to the node, amount of times visited, and the total reward for that node.

Search Limits: `max_iterations` and `max_simulation_depth` control runtime.

5 Results

Simulations were run on DTU's HPC supercomputer¹ with a 24-hour job limit. Some configurations have fewer entries due to timeouts. The setup ensured consistent and comparable conditions for evaluating heuristics and algorithms.

5.1 Expectimax

In appendix A a summary table of the Expectimax results for different heuristic functions can be found. Given the result in appendix A a ranking table of the heuristics can be seen below for the expectimax algorithm.

Rank	Heuristic	Depth	Mean Score	Mean CPU Time (s)	Score/CPU Time
1	game_score	3	8321.10	3.75	2219
2	smoothness	3	9115.73	4.16	2192
3	empty_cells	3	7678.65	3.76	2043
4	merge_potential	3	797.08	0.44	1812
5	snake	3	2918.58	1.62	1801

Table 1: Heuristic ranking by highest mean score per CPU-second (best depth shown).

It is visible *game score* heuristic consistently performs best, achieving the highest mean scores across most depths while maintaining reasonable CPU efficiency. In contrast, *merge potential* performs worst, yielding significantly lower scores. This is likely because it doesn't account for non-adjacent possible merges. Depth-dependent trade-offs are also apparent, as shown in Figure 3.

¹Jobs ran with 2 cores on identical Xeon 8462Y+ CPUs; logging was offloaded when possible (see `HPC.sh`).

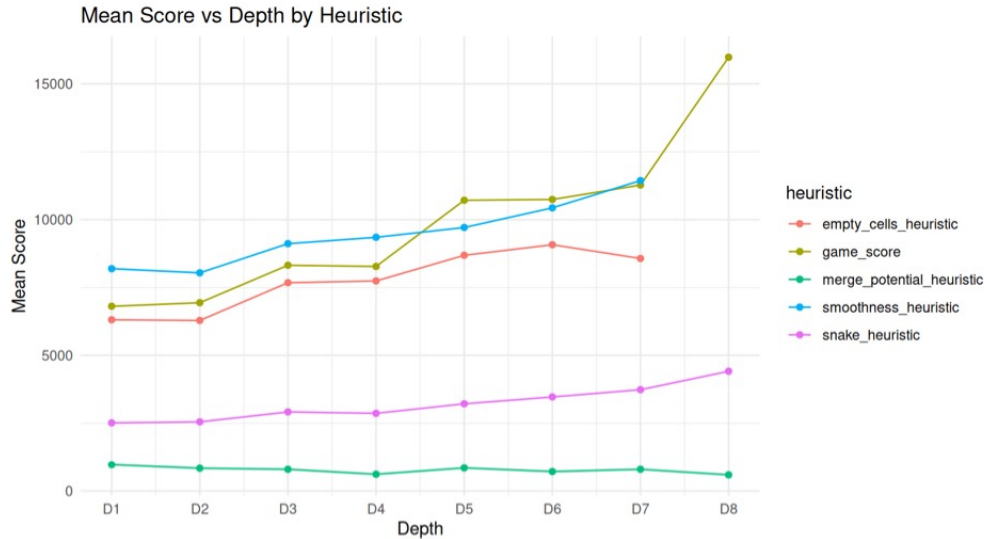


Figure 3: Mean score vs. depth for each Expectimax heuristic. Some heuristics did not complete all 100 runs at depths 6–8.

For shallow depths the differences in performance are very modest between game score, empty cells, and smoothness, with smoothness edging out the others with a bit higher score. For deep depths we see the highest absolute performance from the game score heuristic, however, as mentioned this comparison is not the fairest as we don't have enough data points from the deep depths. A diminishing return also arises between depth 5 to 6, where the score gain flattens while the CPU time increases exponentially as seen from figure 4. We also see that while smoothness can perform well it is also consistently the slowest of the heuristics.

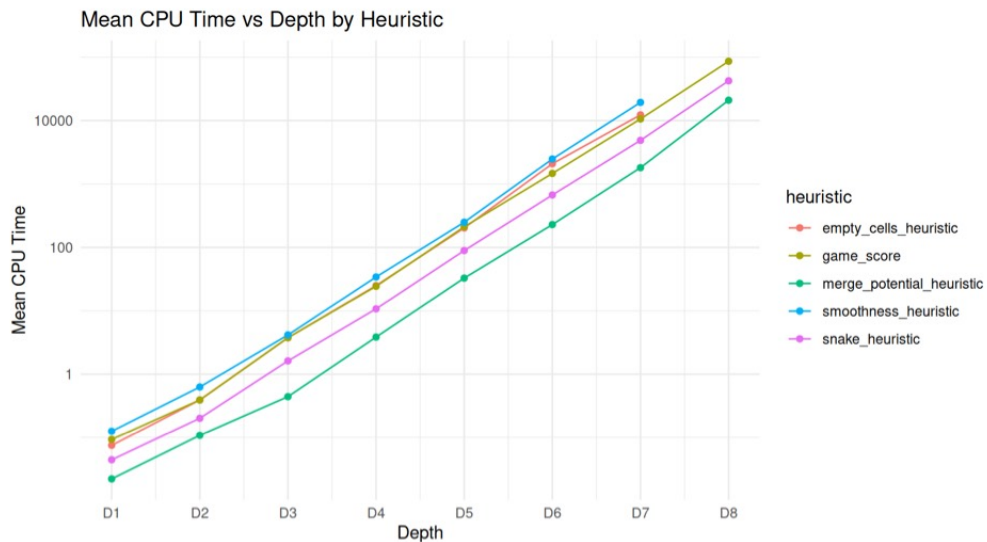


Figure 4: Mean CPU time vs. search depth for each Expectimax heuristic (log scale). The plot highlights exponential cost growth and relative efficiency. Some heuristics did not complete all 100 runs at depths 6–8.

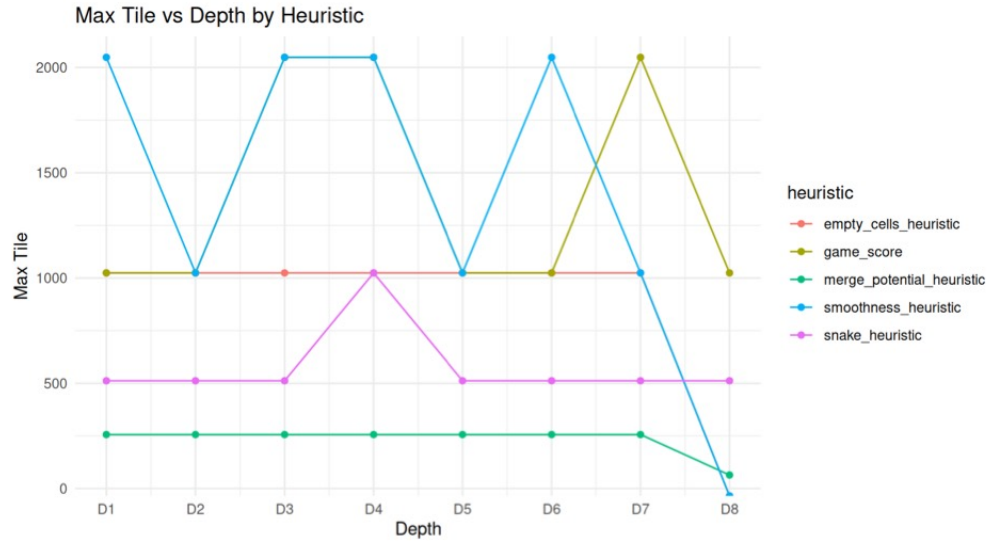


Figure 5: The absolute max tile is the biggest tile that it got over every run at that depth. Note that not every heuristic finished 100 times from depth 6 to 8.

5.2 MCTS

From the results in appendix B we see that the performance of the MCTS algorithm can vary considerably depending on the combination of parameters. Some configurations, such as when using very high numbers of MCTS iterations (e.g., 10,000) or very deep simulations (e.g., simu=500 or 5,000), result in extremely high CPU times (e.g., 4796.22 or 3144.81 s). In contrast, settings with lower mcts iter or shallower simu (e.g., mcts iter=10, simu=10) produce almost negligible CPU times (as low as 0.53). We note that the high value configurations do not have as many data points, which can skew the comparison. The scores and maximum tiles achieved do not always improve with increased computational effort. As figure 6 shows there is a very clear correlation between both the number iterations and simulations with the CPU time, increasing either by a big amount will also result in a huge increase in computational time.

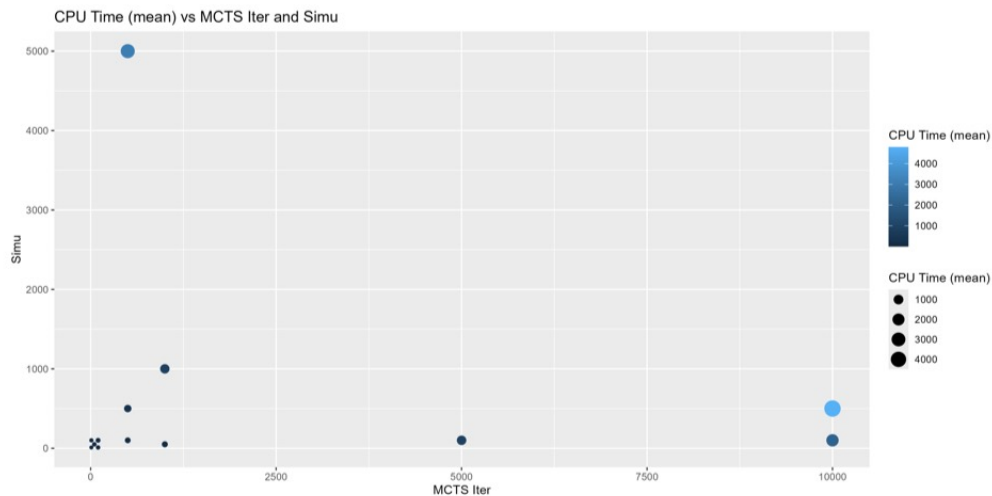


Figure 6: Scatter plot illustrating the connection between CPU time, MCTS iterations and number of random simulations.

The results hint that moderate settings, like a mcts iter around 100 with either low or moderate simu values

(rows 7, 8, 9, and 10) might provide a good balance between computational cost and performance.

6 Discussion

6.1 Comparison of The Algorithms

The MCTS algorithm can be tuned for faster runtime while maintaining competitive scores with well-chosen parameters. However, performance is sensitive to configuration, and poorly chosen settings can be computationally expensive. In contrast, Expectimax can achieve high scores, but runtime grows rapidly with search depth and heuristic complexity. Lower depths often perform worse, while deeper searches improve performance at a high computational cost. Thus, finding the right balance between depth and heuristic is crucial.

Algorithm	Configuration / Heuristic	CPU Time (s)	Score (μ)	Max Tile (μ)	Notes
MCTS	100 iterations, 100 mcts iter, 10 simu	4.11	8782.60	678.40	Efficient, balanced score/time trade-off
Expectimax	Depth 3, <i>smoothness</i> heuristic	4.16	9115.73	697.09	Comparable CPU time with a slightly higher score
Expectimax	Depth 5, <i>game_score</i> heuristic	211.36	10713.60	788.48	High computational cost but improved performance

Table 2: Comparison of representative configurations from MCTS and Expectimax for the game 2048.

In our experiments we have had the best results with the Expectimax algorithm. However, with more compute time and the inclusion of heuristics the MCTS could likely yield even better results.

6.2 Future Work and Improvements

While both Expectimax and MCTS performed competitively, there are several opportunities for improvement. General enhancements include parallelizing the search via CPU multithreading or GPU acceleration, using learned value functions trained via supervised learning, exploring hybrid Expectimax–MCTS approaches, and adopting efficient board encodings (e.g., bitboards or log-scaled tiles) to speed up simulations. Benchmarking against human players and open-source agents using metrics such as average score, max tile, win rate, and score variance could further validate performance.

For Expectimax, future work could focus on combining multiple heuristics—score, empty-cell count, smoothness, monotonicity, and merge potential—into a single weighted evaluation function. These weights could be tuned to balance short-term score gains against long-term board control, with special attention to the smoothness coefficient.

For MCTS, tuning simulation depth and iteration budgets could help balance runtime and performance. Additionally, integrating lightweight domain heuristics into rollouts—such as preferring moves that keep high tiles in corners—may reduce variance and improve stability.

Bibliography

- [1] Cameron B. Browne et al. *Monte Carlo Tree Search steps [Figure]*. https://www.researchgate.net/figure/Monte-Carlo-Tree-Search-steps-5_fig2_316451377. Accessed: 2025-03-23. 2012.

A Expectimax Result Summary

depth	iterations	metric	CPU time μ (s)	CPU time sd (s)	Score μ	Score sd	Max tile μ	Max tile sd
1	500	empty_cells	0.07	0.04	6314.39	2715.19	508.42	230.03
1	500	game_score	0.09	0.04	6812.35	3016.70	544.51	248.23
1	500	merge_potential	0.02	0.02	969.88	502.73	95.15	54.50
1	500	smoothness	0.13	0.06	8194.71	3644.11	635.14	279.68
1	500	snake	0.04	0.02	2515.33	1108.66	208.38	101.91
2	500	empty_cells	0.39	0.12	6289.81	2689.98	505.09	228.16
2	500	game_score	0.39	0.12	6943.17	3009.44	556.80	252.24
2	500	merge_potential	0.11	0.03	837.86	474.12	85.20	52.10
2	500	smoothness	0.63	0.20	8042.18	3598.67	630.53	283.06
2	500	snake	0.20	0.06	2553.94	1182.46	215.30	108.62
3	500	empty_cells	3.76	1.08	7678.65	3332.12	611.84	266.79
3	500	game_score	3.75	1.09	8321.10	3569.11	653.06	283.73
3	500	merge_potential	0.44	0.15	797.08	563.90	81.52	58.78
3	500	smoothness	4.16	1.20	9115.73	3773.53	697.09	279.75
3	500	snake	1.62	0.48	2918.58	1407.51	241.66	125.37
4	500	empty_cells	24.99	5.98	7743.96	3265.17	611.33	257.47
4	500	game_score	24.39	6.41	8276.42	3651.44	650.75	290.85
4	500	merge_potential	3.84	0.83	614.52	386.70	64.43	43.19
4	500	smoothness	34.19	8.33	9349.87	3747.48	721.41	279.76
4	500	snake	10.72	2.83	2866.70	1435.28	236.16	129.29
5	100	empty_cells	203.17	46.80	8687.84	3557.54	675.84	271.12
5	100	game_score	211.36	54.67	10713.60	4145.48	788.48	271.51
5	100	merge_potential	32.83	6.68	849.56	587.40	83.76	61.25
5	100	smoothness	249.49	64.48	9711.44	3954.98	739.84	288.57
5	100	snake	89.30	19.70	3217.92	1553.33	271.36	126.09
6	34	smoothness	2480.31	540.47	10434.24	4370.28	798.12	338.01
6	41	empty_cells	2090.21	410.14	9076.20	3787.24	686.83	282.99
6	58	game_score	1467.28	257.53	10743.72	3633.23	803.31	255.77
6	100	merge_potential	229.10	35.79	713.92	552.51	76.96	57.60
6	100	snake	670.66	125.02	3470.04	1471.86	300.80	137.81
7	4	smoothness	19377.18	2611.05	11439.00	3760.32	896.00	256.00
7	6	empty_cells	12317.24	2707.14	8570.67	5314.72	640.00	313.53
7	8	game_score	10613.31	2932.12	11275.50	7176.78	864.00	563.16
7	17	snake	4884.84	1039.08	3736.71	1591.31	331.29	143.53
7	47	merge_potential	1809.41	380.94	796.94	513.62	86.30	62.06
8	1	game_score	86193.67	NA	15980.00	NA	1024.00	NA
8	2	snake	42423.52	7902.60	4416.00	3365.83	320.00	271.53
8	4	merge_potential	20962.33	4027.04	593.00	331.45	56.00	16.00

Table 3: Summary of performance metrics for Expectimax algorithm with different heuristic functions.

B MCTS Result Summary

iterations	mcts iter	simu	CPU time μ (s)	CPU time sd (s)	Score μ	Score sd	Max tile μ	Max tile sd
2	10000	500	4796.22	302.01	6324.00	1108.74	512.00	0.00
4	500	5000	3144.81	205.83	7061.00	455.23	512.00	0.00
7	10000	100	2151.82	344.70	8574.86	2851.61	658.29	249.83
16	1000	1000	893.71	302.27	6283.00	2858.58	496.00	237.75
16	5000	100	917.93	336.83	6654.75	3649.44	568.00	299.16
43	500	500	341.05	130.58	7685.86	3922.40	595.35	292.80
100	10	10	0.53	0.29	7949.04	3475.82	619.52	282.27
100	10	100	1.57	0.58	6468.04	3195.38	522.24	248.57
100	50	50	5.88	2.37	7897.84	4075.92	618.24	291.12
100	100	10	4.11	1.42	8782.60	3603.55	678.40	268.31
100	100	100	17.67	6.91	7501.60	4053.82	613.12	346.29
100	500	100	90.05	34.78	7645.48	3884.69	609.92	293.28
100	1000	50	118.00	43.80	7426.40	3607.87	593.28	286.20

Table 4: Summary of performance metrics for MCTS with different algorithm metrics. mcts iter is number of runs the algorithm makes before making a choice, simu is how deep the random test simulations can run, and iterations are the number of games played with the algorithm.