CS 57100: Artificial Intelligence
Final Project Report
2048^3: A Twist on 2048

Siddharth Prabakar, Anish Kambhampati, Avi Khandelwal, Ryan Rittner

November 30, 2024

1 Project Overview/Background

Our idea for the CS571 project was to build an agent for extensions of the popular puzzle game 2048. We thought it was relevant to the topics we had covered in the course, and possible extensions would introduce new and interesting challenges.

The rules of 2048 are pretty simple. The game is played on a 4x4 grid, with 2 of the tiles filled with a 2 or 4 initially. The player has 4 possible moves- up, down, left or right, with each move sliding all tiles in that direction as far as possible. Two tiles of the same value combine to create a tile with a larger value, with the goal of the game being to get the largest value tile possible. At the end of each move, a new tile is randomly placed in one of the empty tiles, with value 2 (probability 90%) or 4 (probability 10%).

Our idea for the final project was to develop an AI agent to beat a variation of 2048, with a bigger board size, powers of 3 combining, and more random tiles after each move.

We believed this was an interesting idea for our final project because 2048 in and of itself is a game that has been studied using AI algorithms in the past, but the potential extensions of it that we listed above had not been explored too much. It's related to games and search techniques, which are topics we have extensively covered in class, giving us a good starting point of knowledge. But there is also an element of randomness in the game that isn't

in the games that we covered in class, which could make the game harder for deterministic algorithms like the ones we covered in class.

Today, 2048 has been solved used many different AI methods, such as reinforcement learning, Monte Carlo methods, and Expectimax algorithm agents. While 2048 has been extensively studied, we were unable to find many instances of AI agents for powers of n variants of 2048, or where multiple new tiles spawn every turn. Our goal for the project was to compare and explore algorithms we haven't covered in class such as these Monte Carlo methods or Expectimax algorithm.

2 Formal model of problem

Our initial model for our problem was the following. We imagine a variant of 2048 with the following modifications:

1. The board is of size 9x9 instead of 4x4
2. The tiles are powers of 3 instead of powers of 2
3. Three tiles (rather than 2) must combine to form 1 tile containing the next power of three.
4. After each move, the game generates 2 tiles. Each tile is either of value 3 (90% probability) or value 9 (10% probability). Tiles are randomly placed in some unoccupied space on the board.
5. Possible actions at each turn remain the same : Up, Down, Left, Right
6. The game ends when the board is full and no tiles can merge. There is no win condition as this version is also score-based.

The scoring remains the same in this variation (i.e. when tiles combine, the player earns the number of points equal to the value of the new tile created).

2.1 Updates we made to our formal model

During the course of our work on this project, we found that a 9x9 board was too large for the algorithms we were implementing combined with the computing resources we had at our disposal. A 9x9 board means there could be nearly 80 spots for the random tiles to be placed. Combined with the fact that we had two random tiles, this meant there were close to $4\binom{81}{2} = 12960$

potential successor states from any state in the worst case. Because of this, we updated the formal model of our problem to the following:

1. The size of the game board is now 6x6 instead of 9x9

3 Algorithms

For our project, we implemented 3 different algorithms to tackle our problem: random moves, Monte Carlo Tree Search (MCTS), and Expectimax. Based on our initial research, MCTS and Expectimax were common algorithms used for 2048 games, so we thought they could potentially perform well on our 2048 variant. Random moves served as a baseline to compare the performance of more sophisticated algorithms, allowing us to measure the improvements in gameplay strategy and decision-making efficiency.

3.1 Random Moves (Baseline)

We needed a baseline to test our algorithms against, so we created one strategy that simply made random moves until the game ended. At every opportunity to make a move this strategy selects one of the four possible moves (up, down, left, right), each with equal probability.

3.2 Monte Carlo Tree Search (MCTS)

The Monte Carlo Tree Search algorithm works by first simulating each possible move. Then for each resulting state, the search simulates 30 games using the random moves strategy. Once all these simulations are over, the algorithm chooses the move which led to the highest average score of the 30 games.

3.3 Expectimax

3.3.1 Algorithm

The Expectimax algorithm is a game theory algorithm that uses expected utility to determine the best move to take at a given state[1].The algorithm alternates between maximizing and chance nodes, which respectively simulate player moves and random tile spawns in the context of our problem. Maximizing nodes find the successor state with the highest expected utility, while chance nodes calculate the expected utility by taking the average of all successor nodes.

Ideally, we would explore every possible state (i.e. all tile spawns) to a certain depth. However, there are simply too many possibilities on a 6x6 board (branching factor up to $4\binom{36}{2} = 2520$) to expand the search further than one move. To counteract this, we only consider four possible tile spawns and increase the depth to 2.

3.3.2 Heuristic

Heuristics are methods/functions that allow us to evaluate a game state, giving it a score that informs us of how "good" that state is. Good heuristic functions can increase the performance of AI algorithms dramatically. Since we were using a depth-limited Expectimax algorithm, good heuristic functions were very important to maximizing performance.

The heuristic scoring function we used evaluates game states using five metrics.

1. Game Score: represents immediate progress, rewarding states with higher cumulative points
2. Number of available merges: gauges future opportunities for tile combination, promoting actions that maintain flexibility
3. Number of Empty tiles: ensure maneuverability, favoring states with more open spaces to avoid being trapped
4. Max tile value: reflects long-term success by incentivizing states with higher value tiles
5. Smoothness: measures alignment of adjacent tile values, encouraging boards that are easier to merge

In our final heuristic function, each metric was weighted differently for two main reasons: to reflect the importance/value we placed on a metric, or in order to make sure the values of a metric were on the same scale/range as other metrics.

4 Results and Evaluation

To evaluate our algorithms, we ran each algorithm for 200 trials. For each trial, we obtained two metrics- the final score of the game, and the highest

value tile achieved in that game. We used these two metrics in our final evaluation to compare the algorithms to one another.

4.1 Score

In each of the 200 trial games per algorithm, we recorded the final score of each game once it was over. Table 1 shows the average scores for each algorithm across the 200 trials. We found that while both Expectimax and Monte Carlo Tree Search outperformed random moves, Monte Carlo Tree search was better than Expectimax.

Figure 1 displays the distribution of the scores for each algorithm. We observe that MCTS has a higher mean score and max score than Expectimax, however it also has a larger variance. We believe this is potentially because of the algorithm's incorporation of random moves in it's strategy.

	Random Moves	**Expectimax**	**MCTS**
Mean Score	975.87	14274.09	19498.365

Table 1: Average Scores of Algorithms Across 200 trials

To confirm that the Monte Carlo Tree Search strategy achieves higher scores on average than the Expectimax strategy, we performed a 2 sample T-test on the scores from the games played. The T-statistic was 6.3176 with a p-value of less than 10^{-10}. Thus we can say with a very high degree of confidence that Monte Carlo Tree Search does in fact achieve generally higher scores than Expectimax. Similarly, to confirm that Expectimax does in fact achieve higher scores on average than random moves, we performed another 2 sample T-test. The T-statistic for this test was 33.8360, with a p-value of less than 10^{-40}. This also confirms with a very high degree of confidence that Expectimax is a better strategy than making random moves.

4.2 Highest Tile Value

Similar to how we recorded the final score of each trial game, another metric we recorded for evaluation purposes was the highest value tile achieved in the game. While getting a high score is something that players aim for when playing 2048, the core goal of the game was to reach the 2048 tile. In other words, the initial goal of the game is to reach higher and higher tile

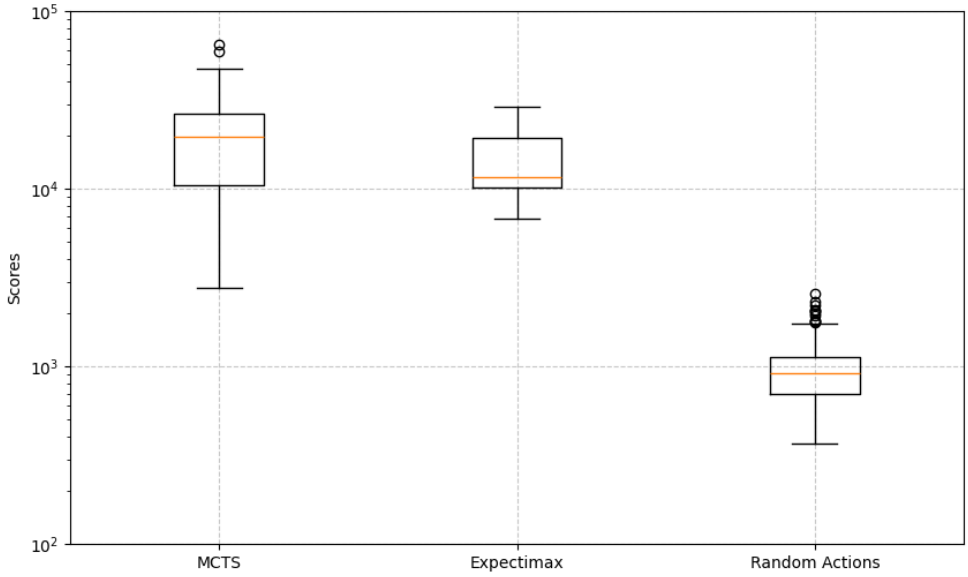

Figure 1: Comparison of distribution of scores across algorithms

values. Hence, we believed that the highest tile value was a key metric that we needed to consider in our evaluation.

Table 2 shows the highest tile value achieved by the algorithms. From the results, it's very clear that both the Expectimax and Monte Carlo Tree Search algorithms outperform the Random Moves algorithm. Expectimax and MCTS both reach the 243 tile in 100% of the games, with Expectimax even reaching the 729 tile in all 200 trials. On the other hand, Random Moves only reaches the 243 tile in 10% of the games, and never gets higher than that. This shows that both Expectimax and Monte Carlo Tree Search are much better than the baseline algorithm for our 2048 variant. Figure 2 shows the distribution of highest tile value between the 3 algorithms, and further supports this claim.

When comparing Expectimax to MCTS, we believe that similar to the mean scores, MCTS performs better than Expectimax. Expectimax does offer a higher floor, as it reached at least 729 in all 200 trials, while MCTS's

	Random Moves	Expectimax	MCTS
3	0%	0%	0%
9	0%	0%	0%
27	7%	0%	0%
81	83%	0%	0%
243	10%	0%	4.5%
729	0%	61.5%	32.5%
2187	0%	38.5%	61.5%
6561	0%	0%	1.5%

Table 2: Highest Value Tile Achieved by Algorithms (% over 200 trials)

lower bound was 243. However, when looking at the percentages in Table 2, we observe that MCTS reaches at least the 2187 tile in 63% of the trials, which is significantly higher than Expectimax's 38.5%. MCTS also manages to reach the 6561 tile in a few of the trials. Thus, we conclude that MCTS is better than Expectimax when evaluating on highest tile value achieved as well.

4.3 State of the Art Strategies

As we used a modified formulation of the 2048 game, there has not been any extensive research done on strategies, and no state-of-the-art strategy exists to compare ours against. When researching SOTA strategies for the base 2048 game we found that Expectimax and MCTS were common strategies. Reinforcement learning and Expectiminimax algorithms were also commonly used, and we saw a rise in use of deep learning models in recent years. We believe these strategies could also work for our 2048 variant, but they haven't been implemented.

5 Insights

5.1 Random Move Generation is a Poor Strategy

Unsurprisingly, randomly selecting a move to perform is a poor strategy. It was handily beaten by both of our AI algorithms, and we strongly suspect that any algorithm can outperform. This is not to discredit its usefulness however. Random move generation serves as a useful baseline with which to compare stronger AI strategies against. It also plays a crucial role in MCTS,

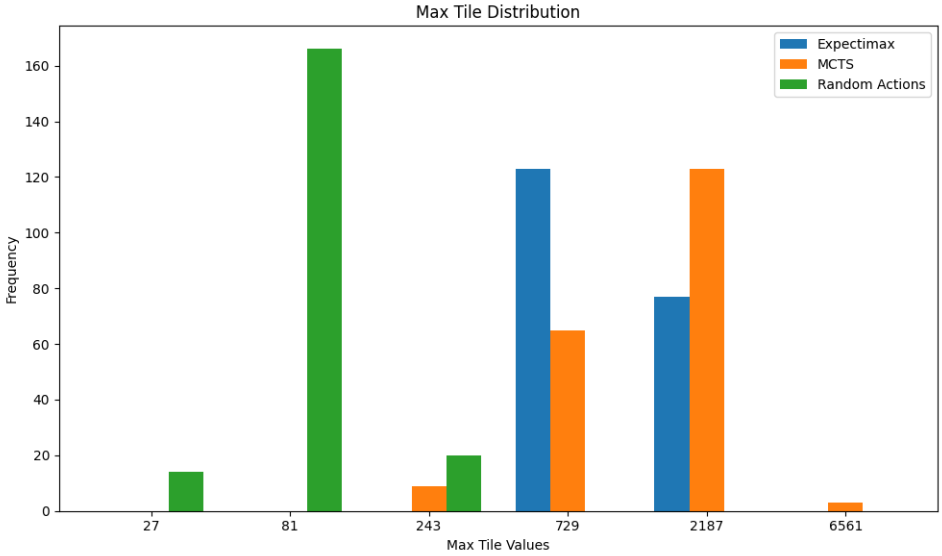

Figure 2: Comparison of distribution of Highest Tile Value across algorithms

where dozens of games are simulated with this strategy to inform the best possible move.

5.2 Our Strategies Do Not Scale

Unfortunately, both our expectimax and MCTS strategies do not effectively scale to larger grids, despite the implementation of the game itself supporting much larger grid sizes. In the former case, the issue has to do with branching and depth size. In the latter case, it has to do with the length of time it takes to simulate a game to the end. For larger grids, taking hundreds (or even thousands) of moves to simulate one of hundreds of games we need to inform even one move means that over the course of a game, we may need to simulate millions of moves. This takes an amount of time that is simply unacceptable. It may be possible to optimize the game logic to allow for faster moves (by, for example, storing the game state as a bitset), but these optimizations were not fully explored.

Our implementation of Expectimax was limited by the high branching factor present after each action. Assuming s is the number of open tiles, after each action, there are $O(\binom{s}{2})$ different subsequent game states, and thus that many chance nodes. Because of this high branching factor, we had to

limit the depth of our Expectimax algorithm to a max search depth of 2, along not considering every possible successor state. Instead we considered a few randomly selected successor states. We believe that increasing the max search depth of the Expectimax will lead to better results, potentially even outperforming the Monte Carlo Tree Search algorithm, but we simply did not have the compute power to test our theory.

5.3 MCTS is Easier to Implement than Expectimax

MCTS, as an AI strategy, is trivial to implement. Once you have the game implemented, all it takes to create a working model is simulating a bunch of games with random chance. In contrast, the expectimax algorithm is much harder to implement. Due to the state space of our version (or really any version) of 2048, enumerating all possible end states is not possible. We must therefore evaluate states with a heuristic function at a certain depth. The choice of heuristic function can radically change the choices the AI strategy makes, which in turn has a massive impact on the final result. Specifically, we initially considered the sum of score and max tile as the heuristic; however, the AI would merge whenever possible and potentially disrupt the future game state. To account for this, we added metrics testing for smoothness, available merges, and empty tiles. These extended the duration of the game and thus achieved higher scores. Finally, we ensured the metrics' importance was reflected in the heuristic by multiplying each one by a weight before summing them.

6 Conclusion

In this paper, we explored three different AI strategies for playing our variant of 2048: random choice, MCTS, and expectimax. We compared how these strategies performed using two different metrics – score and maximum tile reached. Ultimately, we found that, due to the strong correlation between the two metrics, MCTS performed the best on both. Expectimax did slightly worse, but the key takeaway is that both strategies significantly outperformed random chance. Our work provides a foundation for future AI researchers to explore other AI strategies in 2048-adjacent games. Our source code is available at `https://github.com/kambhani/2048-AI`.

References

[1] "Expectimax Algorithm in Game Theory." GeeksforGeeks, 25 Oct. 2021, https://www.geeksforgeeks.org/expectimax-algorithm-in-game-theory/. Accessed 1 Dec. 2024.