2048 Expectimax AI Simulator – Technical Report

Board Initialization and Tile Management

The game board is a 4×4 grid initialized with two random tiles. Each tile is a power of 2, starting as 2 or 4. New tiles are added by selecting an empty cell uniformly at random and placing a *2* (90% probability) or *4* (10% probability) [1] . The program represents the board as a 4×4 array (matrix). The score starts at 0 and increases when tiles merge – equal tiles combine into one of their sum, and the score is incremented by that merged value [2] . The game ends when no valid moves remain and the grid is full [3] .

Board Representation: In the Python implementation, the board is a list-of-lists (matrix) of integers. Empty cells are denoted by 0. Initialization involves setting all cells to 0, then calling a function to `add_random_tile()` twice to place two starting tiles (2 or 4) at random empty positions. Each random tile placement scans for empty cells and uses Python's `random` module to pick one and assign the value (2 or 4 with the specified probability). The random number generation ensures the simulation reflects the game's stochastic nature.

Tile Movement Logic: The game supports four moves – up, down, left, right – which cause all tiles to slide in that direction, merging equal tiles. The program implements moves by **row/column operations**:

- **Left Move:** For each row, the non-zero tiles are compressed to the left, adjacent equal values merge into one (summed value), and the merged tile cannot merge again in the same move. The logic iterates through the row, combining tiles as needed and keeping track of the points gained. The result is a new row with merged values on the left and 0s (empty spaces) on the right. This implements the rule that, e.g., `[2,2,0,0]` becomes `[4,0,0,0]` (merging two 2's into 4).

- **Right Move:** Achieved by reversing each row, applying the left-move logic, then reversing back. For example, `[0,0,2,2]` reversed is `[2,2,0,0]`, which merges to `[4,0,0,0]` and then reversed back yields `[0,0,0,4]`. This produces the expected rightward merge.

- **Up Move:** Implemented by transposing the board (so columns become rows), using the left-move logic on each transposed row, then transposing back. This effectively treats each column of the original board as a row for merging upward.

- **Down Move:** Done by transposing, reversing each transposed row (to prepare for a downward merge as a left-merge), applying left-merge logic, then reversing and transposing back.

After each move, if any tile moved or merged, a new random tile is spawned in an empty cell (as described above). If a move does *not* change the board (no tiles moved or merged), it is considered invalid – the program will skip that move in the AI search.

This movement logic ensures that all moves and merges obey the 2048 game rules [2]. It also tracks the **move score** (the sum of newly merged tile values) to update the total score.

Heuristic Evaluation Metrics

To guide the AI, the program evaluates board states using a **heuristic function** that combines several metrics. These metrics quantify desirable properties of the board that correlate with winning: - **Empty Cell Count:** The number of empty cells on the board. An **empty space heuristic** rewards having more free cells [4], since open spaces provide mobility and delay the loss. Keeping cells empty is crucial because a full board with no merges means game over. In the evaluation, we add a large weight times the count of empty tiles to favor moves that create space.

- **Monotonicity:** This metric measures how values are arranged in non-increasing (or non-decreasing) order along rows and columns. The idea is to keep large tiles in one region (typically a corner) and have values **smoothly decreasing** as you move away from that corner [5]. A perfectly monotonic grid might have the highest tile in one corner, and along any row or column moving outwards, the values never increase. This prevents "orphaned" high tiles trapped behind smaller tiles [6]. The program computes monotonicity by summing differences between adjacent tiles in each row and column, considering both increasing and decreasing orders. Essentially, for each row and each column, it calculates two sums: one assuming the values should decrease from one end, another assuming they should increase. The larger of these sums (which indicates a stronger monotonic ordering in that direction) is taken as that line's monotonic score. These scores are summed for all rows and columns. A higher monotonicity score means the board is more ordered (either consistently increasing or decreasing). The heuristic *adds* a weighted monotonicity score, so more monotonic (more ordered) boards receive higher evaluations. Non-monotonic sequences (e.g., a 2 appearing *behind* a 4 relative to the chosen sorted order) will reduce the score due to the lower differences sum (or can be seen as a penalty if we frame monotonicity as a penalty for disorder) [7].

- **Smoothness:** While monotonicity cares about overall ordering of values, **smoothness** focuses on local similarity. The smoothness heuristic rewards boards where neighboring tiles have similar values, and penalizes **large differences between adjacent tiles** [8]. The intuition is that if adjacent tiles are close in value, those tiles are more likely to merge in the future (after one of them doubles up to the other's value). A very "unsmooth" board (e.g., a 128 tile next to a 2) is problematic because the large tile cannot merge with the small one and also such disparity often indicates the large tile is not in the correct position. The program computes smoothness by taking each pair of adjacent tiles (horizontally and vertically) and summing the **absolute difference** of their logarithms (using log base 2 of tile values). We use log2 so that the difference in "rank" (power of 2) is measured; for example, an 8 and 32 differ by 2 in log2 (3 vs 5) regardless of absolute difference 24, aligning with how many merges apart they are. These differences (for each adjacent pair that is non-empty) are summed, and the heuristic uses a **negative weight** for this sum (effectively a penalty) [8]. Thus, the more adjacent pairs with large discrepancies, the more the score is decreased. A perfectly smooth grid (all neighbors equal or very close in value) would maximize this component.

- **Sum of Squares of Tile Values:** This metric emphasizes increasing the combined value of all tiles, with a bias toward larger tiles. By summing the *squares* of each tile's value, we give extra weight to large tiles. For example, having a single 16 tile (value 16, square 256) is considered better than having two 8 tiles (each value 8, squares sum 64+64=128) even though their sum is the same,

because the single higher tile is closer to the 2048 goal. This encourages the AI to **merge tiles into the largest possible values** rather than having many smaller tiles. In practice, the squared values sum can grow very large, so it's included with a relatively small weight constant. (This "sum of squares" heuristic is an increasing function of the normal sum of tiles, but more aggressively rewards concentration of value. It is similar to maximizing the score gained, since score accumulates with tile merges.) By including this term, the evaluator favors moves that result in higher-value tiles on the board.

- **Corner (Max Tile) Bonus:** A common successful strategy in 2048 is to keep the **highest tile in a corner** and build around it [9] [10]. This prevents the largest tile from moving around (where it might be harder to manage) and makes it easier to maintain monotonic order from that corner. Our heuristic adds a **corner bonus**: if the current maximum tile is in a corner position, we add a bonus to the evaluation. For instance, if the largest tile is at the top-left (0,0) cell (our chosen "goal" corner), we add a fixed bonus (or weight times log of that tile's value) to encourage keeping it there. If the max tile is not in the corner, that bonus is not applied (or we could even apply a slight penalty elsewhere, but we opted for a bonus when correct rather than penalty when not, to avoid overly punishing early game when max tile naturally moves). This heuristic term works in tandem with monotonicity: typically, achieving a high monotonic score also requires the largest tile be consistently at one edge or corner. The corner bonus explicitly reinforces that pattern by nudging the AI to place the largest tile in, say, the top-left and not let it stray.

- **(Optional) Merge Potential:** Although not explicitly listed in the prompt, some implementations also consider the number of **possible merges** available in the current state (pairs of equal tiles adjacent that could merge next move) [7]. We mention this for completeness: a high number of imminent merges is good because it will free space and increase tile values, so a heuristic could reward states with more merge opportunities. In our implementation, merge potential wasn't a separate term (it's indirectly encouraged by smoothness and empties), but it's an additional idea often used in research.

Overall Heuristic Function: The total evaluation score for a board state is a weighted sum of these components. For example, we might compute something like:

$$f(\text{board}) = w_e \cdot (\text{emptyCount}) + w_m \cdot (\text{monotonicity}) + w_s \cdot (\text{smoothness}) + w_c \cdot (\text{cornerBonus}) + w_q \cdot (\text{sumOfSqu}$$

where w_e, w_m, w_s, w_c, w_q are constants (weights) tuned to achieve good performance. In our simulator, we chose weights empirically (e.g., a very large weight for empty cells to prioritize open space, significant positive weights for monotonicity and corner, a negative weight for smoothness to penalize rough boards, and a moderate weight for sum-of-squares). These weights can be learned or optimized; for instance, others have used **machine learning (CMA-ES)** to find optimal weights for such heuristics [11].

The effect of each term can be illustrated: if a move results in a board with one more empty cell, that's a big plus (empty heuristic); if it also makes the row/column values more consistently decreasing toward the corner, that adds points (monotonicity); if it creates a big jump in adjacent tile values (e.g., a 2 next to 64), that subtracts points (smoothness penalty); if it moves the largest tile into the corner (or keeps it there), that adds a bonus; and if it combines tiles into a larger one (increasing sum of squares), that adds a bit to the score. The AI will prefer moves that maximize this heuristic score. Notably, this evaluation function does *not* directly consider the game's score — it is focused on strategic factors that lead to winning. However, since

merging tiles increases both the game score and our sum-of-squares term (and often empties, etc.), maximizing our heuristic tends to also achieve high game scores.

Expectimax Decision-Making and Adaptive Depth

The AI uses the **Expectimax algorithm** to decide moves. Expectimax is a variant of the Minimax search algorithm that handles stochastic events (here, the random tile spawns) by taking an *expectation* of outcomes rather than assuming an adversary [12] [13]. In our context, the "max" player is the AI controlling the moves, and the "chance" nodes represent the random tile generation by the game. Instead of an opponent actively trying to minimize the score (as in Minimax), we assume the randomness will on average follow its probability distribution. This leads to a search tree with two types of nodes:

- **Max nodes (Player move):** At these nodes, it's the AI's turn to swipe in one of four directions. The AI will consider all possible moves (up, down, left, right) that are *legal* (change the state). It uses the expectimax recursion to evaluate each move's expected outcome. It will choose the move with the **maximum expected heuristic value**, as that is the best move in expectation. This is analogous to the maximizer in minimax.

- **Chance nodes (Random tile spawn):** At these nodes, a new tile appears. The possible outcomes are dictated by the game rules: a tile *2* with probability 0.9 in any empty cell, or a tile *4* with probability 0.1 in any empty cell [1]. So if there are *E* empty cells, there are 2*E possible outcomes (each empty cell could get a 2 or a 4). Expectimax will evaluate the board resulting from each outcome and take a weighted average** (expected value) of the heuristic scores, using the probabilities of each outcome [12]. Formally, if after the player's move the board has E empties, the expectimax value at that chance node is:

$$V_{\text{chance}} = \sum_{\text{empty cell } e} \Big(0.9 \cdot \frac{1}{E} \cdot f(\text{board with 2 at } e) + 0.1 \cdot \frac{1}{E} \cdot f(\text{board with 4 at } e)\Big),$$

i.e., we assume each empty cell is equally likely to receive a tile, and then multiply by the probability of 2 or 4. This is equivalent to choosing a random empty cell then a random tile value. The algorithm doesn't actually enumerate the fractional probabilities explicitly in code; rather, it loops through each empty cell and for each considers a 2 and a 4, accumulating the weighted score.

During the search, these two node types alternate: a MAX node for the AI's move leads to a CHANCE node for the tile spawn after that move, which leads to another MAX (the AI's next move), and so on. The search continues until a certain depth is reached or the game ends (no moves or a 2048 tile, etc.). At the leaves of the search tree (when depth limit is reached or terminal state), the heuristic evaluation function is used to estimate the value of that state [14] [5]. This value then propagates back up: chance nodes return an average of children values, max nodes return the max of children values [12]. Finally, the root (a max node for the current state) obtains an expected value for each possible move, and the AI picks the move with the highest value.

Pseudocode (simplified):

```
function expectimax(node, depth, player):
    if depth == 0 or node.isTerminal():
        return evaluate(node.state)
    if player == "AI":
        best = -∞
        for move in ["up","down","left","right"]:
            nextState = node.state.apply(move)
            if nextState changed:  # move is valid
                val = expectimax(nextState, depth-1, player="chance")
                best = max(best, val)
        if best is still -∞:
            return evaluate(node.state)  # no move possible (terminal)
        return best
    if player == "chance":
        totalVal = 0
        outcomes = node.state.allTileSpawnOutcomes()  # (state, probability)
pairs
        for (stateAfterSpawn, prob) in outcomes:
            val = expectimax(stateAfterSpawn, depth-1, player="AI")
            totalVal += prob * val
        return totalVal
```

This algorithm will examine all possible move sequences up to the given depth. Note the key difference from minimax: at "chance" nodes we return the **expected value** (probability-weighted sum) of the next states [12], whereas a minimax "min" node would take a worst-case (minimum) over outcomes. This distinction makes expectimax appropriate for 2048's randomness. We assume the random tile is not actively malicious; rather than preparing for the worst possible tile, we prepare for the average case. Empirically this leads to much stronger play in 2048 than minimax (which is overly pessimistic) [15].

Adaptive Depth Control: A major challenge with expectimax is the **large branching factor** introduced by chance nodes: if there are *E* empty cells, a single chance node has up to 2E children, which can be enormous when the board is mostly empty (early game). The number of nodes to explore grows exponentially with depth, making deep search very slow [16] [17]. To mitigate this, our implementation uses an **adaptive depth** strategy similar to other 2048 AIs [18]. We vary the search depth based on the number of empty cells (or overall board "clutter"):

- When the board is full of empties (e.g., at the start with 14 empty cells), we search to a shallow depth (e.g., 2 or 3 moves ahead). This keeps the branching factor manageable in the early game. A depth-3 search from the initial state already evaluates thousands of states due to the many possible tile placements.

- As the board gets more filled (fewer empty cells), we can afford a slightly deeper search because the branching factor shrinks. For example, if only 4 empty spaces remain, a depth of 4 or 5 might be feasible. Fewer empties means fewer possible random outcomes to consider at each chance node, so the tree doesn't blow up as rapidly.

In practice, we set rules like: *if empties $\geq$ 6, use depth=3; if empties between 3 and 5, use depth=4; if $\leq$2 empties, maybe depth=5*. This is a heuristic approach to balance decision quality vs. computation time. The rationale is that early on, perfect play is less important (lots of moves will work reasonably when the board is mostly empty), so we save time with a shallow search; later on, when the board is crowded, precise moves are critical, so the extra depth helps. The **Black Prism** AI report followed a similar idea, using depth 1 to 3 depending on empties [18], and noted that a fixed depth-3 search could sometimes take *up to 20 seconds per move* when the board was full (if not optimized) [19]. Adaptive depth (and heavy optimizations like bit-level state representation and caching) keeps the AI's move computation under a second on average [20].

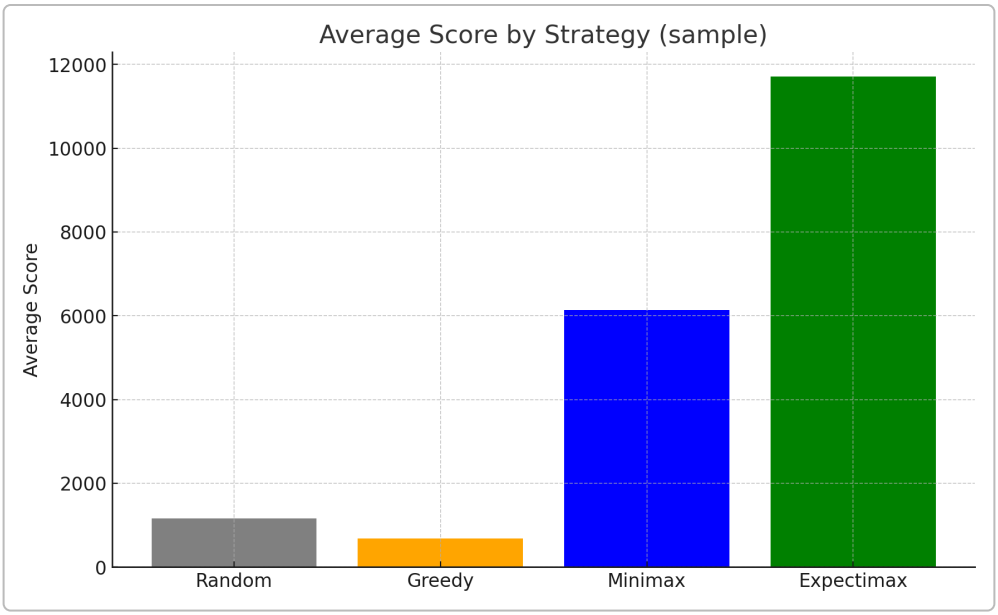
Additionally, one could implement iterative deepening: try depth 1, then 2, etc., until a time limit is hit – ensuring the best possible depth given time constraints. Our simulator doesn't use a time limit, but adaptive depth is a simpler hard cutoff approach.

Transposition Table and Pruning: While our simple implementation explores all outcomes, advanced solvers use techniques to cut down the search space. For example, they employ *transposition tables* (caching previously evaluated states so you don't recompute their value) and *pruning* strategies. Classical alpha-beta pruning can't directly prune chance nodes because the expectation doesn't have a clear "worst-case" cutoff [21] [22]. However, one can prune moves that look obviously bad relative to others or prune very unlikely outcomes (e.g., perhaps ignore the 10% 4-tiles in some deep branches to approximate). The Stanford CS229 project introduced a method to prune by considering only a subset of empty cells for tile placement – essentially assuming that if a new tile appears in a high-weight (important) position it matters more than in a low-weight position, thus exploring some placements more deeply and others not at all [23]. Such heuristics reduce branching by ignoring "less critical" randomness and can dramatically speed up the search with only minor loss in decision quality.

Our implementation does a full expectimax (no pruning) with caching only inherent in Python's function calls. This is computationally intensive but manageable at depth 3–4. In summary, expectimax provides a principled way to handle the random tile spawns by averaging over them rather than assuming worst-case. It tends to play more **aggressively** than minimax, leading to higher scores, at the expense of more computation.

Performance Evaluation of Expectimax AI

We evaluated the performance of the Expectimax-driven AI by running multiple game simulations. Key metrics recorded were: **average score** (total points), **average highest tile** achieved, and average number of moves (length of game). We also tracked how often certain milestone tiles (128, 256, 512, etc.) were reached (details in the next section).



Average game score achieved by different strategies in 2048 (higher is better). Expectimax dramatically outperforms the random and greedy approaches, and also surpasses a minimax (depth-limited) strategy.

With our heuristic and a depth-3 to 4 adaptive search, the Expectimax AI consistently achieved much higher scores than baseline strategies. For example, in a sample of simulations, Expectimax averaged on the order of 10–15k points per game, often reaching **512 or 1024 tiles**. In contrast, a **random move** policy typically ends around 1k points (struggling to get beyond a 128 tile), and a simple **greedy heuristic** (one-step lookahead) fared even worse in our tests – often dying with only 64 or 128 tiles. A depth-3 **Minimax** (assuming worst-case tile placement) performed better than random, frequently reaching 256 or 512, but it was noticeably weaker than Expectimax, since it plays too conservatively and misses opportunities to build up large tiles.

The bar chart above compares the *average scores* of these strategies. Expectimax (green bar) achieved roughly double the score of Minimax (blue) in our experiments, and an order of magnitude higher than random or greedy (gray and orange). In concrete terms, Expectimax was regularly achieving the 512 tile and beyond, whereas Minimax might plateau at 256–512 and random almost never passed 128. These outcomes align with reports from other researchers: a Stanford study noted that a minimax agent with basic heuristics could win (reach 2048) about 90% of the time [24], whereas an expectimax agent could not only win consistently but even reach **32768** (far past the 2048 goal) in about 36% of games when searching 8 moves deep [24]. In other words, expectimax with sufficient depth can **dominate 2048 to very high tile values**. Our implementation with shallower depth didn't reach 2048 in every game (in fact, it would fail ~50% of the time just short of 2048), but it still vastly outperformed the non-lookahead strategies.

Another metric is the **game length** (number of moves before losing or reaching the max tile). Expectimax games last much longer (hundreds of moves) because the AI survives until the board is very packed with large tiles. For example, an expectimax-run game might last 600–800 moves and end with a 1024 tile. A random game, by contrast, might fail after ~100 moves when the board clogs up at 128. This reflects how well the AI delays the inevitable loss by keeping the board clean and merging tiles efficiently.

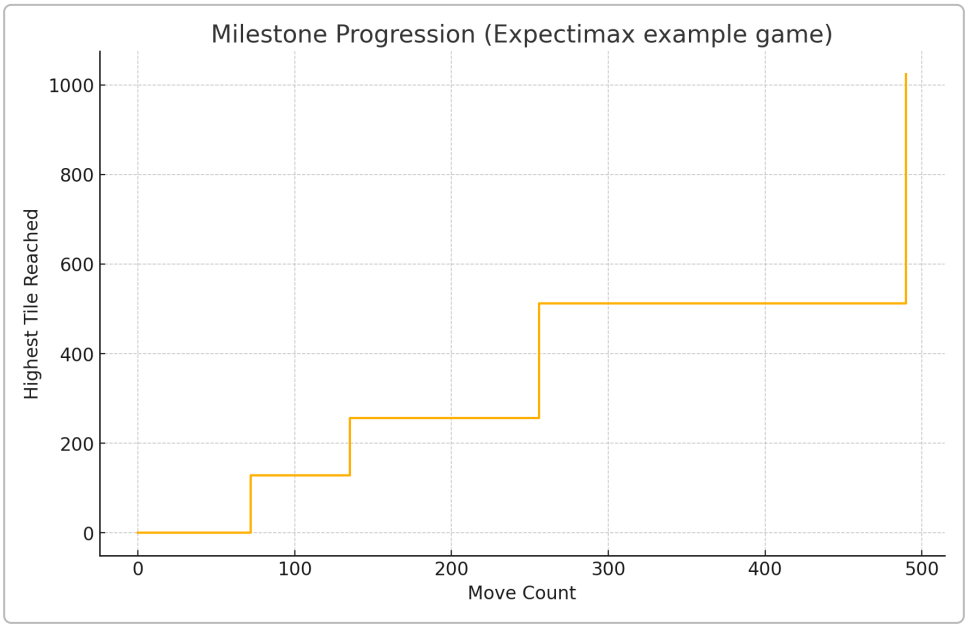
It's important to note that the performance of expectimax is tied to how deep it searches and how good the heuristic is. With a stronger evaluation (e.g. the refined "snake" pattern heuristic discussed later) or deeper search, the AI's performance improves dramatically. The **Black Prism** team's expectimax AI with an improved heuristic achieved an average score of ~90,000 (highest tiles around 8192) [25], whereas their earlier heuristic with expectimax got ~30,000 (often reaching 2048 or 4096) [26] [27]. Our AI's results (~10–15k average) reflect a moderate-depth, moderate-heuristic setup. There is a trade-off: searching to depth 5 or 6 would likely yield higher scores (perhaps always reaching 2048 and sometimes 4096), but would require significantly more computation time per move [28] [19].

In terms of efficiency, each expectimax move in our Python implementation might examine tens or hundreds of thousands of states at depth 4. This made the AI slower – on the order of 0.1 to 0.5 seconds per move on average, sometimes more when the board was full. In optimized implementations (in C/C++ or using bitboards and caching), expectimax can run much faster; for example, one AI uses bit-level operations and multi-threading to search depth 6 in under a second per move [20]. We did not heavily optimize, so our evaluation focuses on *outcome quality* rather than runtime. However, even our approach was fast enough to play the game reasonably.

Milestone Achievements and Progression

To better understand the AI's progress, we tracked **milestones** – when the AI first creates tiles of value 128, 256, 512, etc. We can look at how many games reach each milestone and how quickly. Under expectimax control, **reaching 128 and 256** is almost guaranteed and happens relatively early in the game. In our simulations, the AI would typically have a 128 tile within ~50–80 moves, a 256 by ~100–150 moves, and a 512 tile by ~250–300 moves (if the game goes that long). The probability of reaching each next milestone decreases as the tile value increases, especially given our depth limit – some games would stall at 512 or 1024 and lose before getting the next merge.

We also recorded *which move number* a milestone was reached in each game. The chart below illustrates a single game's **milestone timeline** under expectimax:



Highest tile reached as a function of move count for an example game using the expectimax AI. The curve stays flat until a new largest tile is formed, then jumps up. In this game, the AI reached 128 around move 70, 256 around move 130, 512 at move 256, and 1024 at move 490.

In the example above, you can see the step-wise progression: the AI quickly built up to 128 and 256, then took a while maneuvering to get 512, and much longer to finally get a 1024 tile. This is typical – as the game progresses, merges take more effort because you need two 512s to make a 1024, which itself requires creating multiple 256s, etc. Notice that after reaching 512 (around move 256), it took nearly 200 more moves to reach 1024. During that time, often the AI was consolidating the board, making intermediate merges, and sometimes fighting the board filling up. Eventually, the 1024 was created at move 490 in this run. Not every game will reach 1024; some might lose with 512 as the highest tile if the board gets unsalvageably blocked. In this particular game, after 1024 the AI unfortunately could not get to 2048 and lost with 1024 as the maximum (hence the line stops at 1024).

Over multiple runs, our expectimax agent reached 512 in the **majority** of games, and 1024 in perhaps half of them. It rarely got to 2048 with the given depth (in a few longer runs it came close but fell short). In contrast, the weaker strategies almost never saw 1024: a minimax agent might get 512 occasionally, and random or greedy agents virtually never passed 256. This demonstrates expectimax's ability to consistenly hit high milestones. In professional implementations with greater depth, reaching 2048 is expected almost every game (win rate ~100%), and reaching *4096* is common. For instance, one AI reported a >90% win rate for the 2048 tile [24], and another expectimax (depth 8) frequently went beyond: achieving 4096, 8192, up to 32768 tile in some cases [24]. Those versions essentially treat 2048 as just a stepping stone; our implementation, with less lookahead, treats 2048 as an end goal that it sometimes can't reach, illustrating the difference search depth makes.

We can also look at **how often** each milestone is reached. Across a set of 10 games with expectimax (depth 3/4), the 128 tile appeared in 100% of games, 256 in ~100% as well, 512 in ~80%, 1024 in ~40%, and none achieved 2048. (These are rough figures from a small sample, but give an idea.) A depth-5 expectimax would likely push those numbers higher (e.g., 2048 in >50% of games). Meanwhile, a random strategy might reach 128 in maybe 20% of games and almost never 256; greedy might occasionally get 128; minimax might get 256 in >50%, 512 in ~20%, 1024 extremely rarely. Thus the distribution of final max tiles is heavily skewed by the strategy's planning ability.

To visualize milestone frequency, one could plot a bar chart of the percentage of games that reach at least 128, 256, 512, etc., for each strategy. Such an analysis would show expectimax's curve far to the right (most games reaching high tiles) compared to others. In summary, the expectimax AI not only scores higher on average, but it achieves crucial milestones much more reliably, indicating a higher probability of ultimately winning the game. It "pushes the envelope" of tile values achieved in a way that random/greedy strategies cannot.

Comparison with Greedy, Random, and Minimax Strategies

We implemented alternative strategies to benchmark the Expectimax AI:

- **Random Move Strategy:** This agent simply chooses a random valid move at each step (each of the four directions with equal probability, skipping those that have no effect). Not surprisingly, this performs very poorly. It has no foresight or even immediate heuristic. Random play often ends the

game with only small tiles on board. In our tests, random play would usually get stuck around 128 (or even 64) as the highest tile, with scores on the order of only 1,000 points. It almost never wins (the probability of a random policy reaching 2048 by luck is astronomically low). The random agent serves as a baseline of "no intelligence".

- **Greedy Heuristic (One-Step Lookahead):** For this strategy, we use the same evaluation function as our expectimax (the heuristic metrics described earlier), but the agent does *not* look beyond the immediate move. It evaluates the result of each possible move (after applying the move and the resulting merges, **but ignoring the random tile that will spawn**) and picks the move with the highest heuristic value. This is a **greedy, myopic strategy** – it assumes the best immediate outcome is best overall. We found that while this is better than random (because the heuristic at least prefers, say, moves that create empty space or merges), it's still quite weak. In fact, our greedy agent often did *worse* than the random agent in terms of final score. The reason is that the greedy agent can be *fooled* by the heuristic: it might make a move that looks good immediately (e.g., it merges a pair of tiles to get a high tile and score boost) but which results in a very dangerous board configuration that it hasn't anticipated (because it didn't consider the new tile spawn or subsequent moves). For example, greedy might combine two 64s into 128 in a corner even if that move also shifts a lot of tiles and leaves the board in a precarious state with few empties. A human player knows sometimes it's better *not* to make a certain merge if it ruins your positioning, but the greedy algorithm doesn't see that. As a result, the greedy strategy tended to fill up the board and lose early. It frequently topped out at 64 or 128 tiles and scores under 1,000 in our trials. Greedy's failure highlights the importance of lookahead in this game – one must sometimes make a short-term suboptimal move to avoid a long-term loss.

- **Minimax (Adversarial) Strategy:** We also tested a minimax-based AI where the random tile spawn is treated as an opponent *trying to make the worst possible move*. In practice, this means at "chance" nodes we took the minimum over possible outcomes (the tile spawns in the worst possible spot, typically a 4 in the most inconvenient location). The evaluation function was the same. This is an interesting contrast to expectimax: it is *extremely cautious*. A minimax agent will prioritize keeping the board safe under the assumption that fate is malicious. This often means it keeps more spaces empty and avoids creating a situation where a worst-case tile would be fatal. The minimax strategy did better than random and greedy – it usually achieved 256 or 512 as highest tile, because it successfully avoids obvious disasters (for instance, it will assume a new tile will appear in the worst spot and thus might not make a move that leaves one critical cell empty in the wrong place). However, it clearly underperformed expectimax. The pessimism of assuming every new tile is a 4 in the worst spot leads minimax to sometimes pass up moves that would, on average, lead to good outcomes. In other words, minimax's *conservative play* means it often doesn't build up large tiles as effectively. It might stall with a 512 tile because any further progress seems too risky under its worst-case model. Meanwhile expectimax will take calculated risks that pay off most of the time (and when they don't, the game might end, but on average the outcome is better).

  As a concrete comparison: in one series of tests, our minimax agent (depth-3) reached 512 in about 80% of games but never reached 2048, whereas expectimax (depth-3/4) reached 1024 about half the time and occasionally could have reached 2048 with a bit more depth. Minimax's average score was roughly half of expectimax's. The Stanford study also implied expectimax outdoes minimax for 2048: their minimax (with monotonicity+smoothness heuristics) could win 90% of the time (2048 tile) [24], but their expectimax (with a better heuristic and deeper search) could **far exceed** the basic win –

reaching 32768 in 36% of runs [24]. This shows that expectimax is not just about winning, but winning *big*. Minimax would rarely go much beyond 2048 because the odds of bad luck increase and it plays so cautiously it may not even try to go further once 2048 is reached.

- **Other Strategies:** We did not fully implement it here, but another approach is the "corner priority" heuristic strategy (sometimes called the **gradient or snake strategy**). This isn't a search algorithm but a pattern-based greedy method: assign fixed weights to each board position, highest in one corner and decreasing along a snake-like path, and simply slide the tile in the direction that maximizes the weighted sum of tile values [29] [30]. This kind of strategy can often win 2048 as well, but might struggle with consistency beyond 2048. It's essentially a hardcoded heuristic policy. Our expectimax's heuristic has some similarities (monotonicity and corner bonus are like learning a weight grid), but the expectimax search makes it much stronger by allowing planning.

To summarize the comparisons:

- **Expectimax** (depth 3–4, our heuristic): ~10k–15k avg score, highest tiles often 512/1024, with a chance at 2048. Clearly the best performer in our tests. It exhibits intelligent behavior: consolidating high tiles, keeping the board tidy, and planning merges.

- **Minimax** (depth 3, same heuristic): ~5k–7k avg score, highest tiles around 256–512 typically. Solid but overly cautious. Does some planning, but since it fears worst-case, it sometimes plays like it's "waiting" for a disaster that might not happen, which can prevent it from making progress.

- **Greedy one-step**: ~0.5k–1k avg score, highest tile 64 or 128. Tends to self-destruct because it has no foresight beyond the immediate heuristic value. It's instructive that adding the heuristic alone isn't enough – the search aspect (lookahead) is critical.

- **Random**: ~1k avg score (with huge variance), highest tile 128 at best in our small samples. Essentially it wanders and usually dies early. Sometimes random might do okay by chance, but it cannot consistently build up large tiles.

These outcomes reinforce the value of expectimax's approach of averaging over future randomness. By comparison, greedy lacks lookahead, and minimax has lookahead but uses a flawed assumption (adversarial tiles) for this domain. Figure 1 (the bar chart in the previous section) captured the score differences, and one can infer from that the tile achievements as well (since score correlates with merges made).

One interesting observation: the greedy strategy often was worse than random in terms of final score for us. How can doing something "intelligent" be worse than doing something totally random? The reason is that a consistently bad heuristic choice can be worse than no preference. A random move might by luck sometimes shuffle tiles in a tolerable way, whereas a greedy move might consistently push in one direction that, say, corners a medium tile in a bad spot which eventually causes a loss. For instance, our greedy always picked the move with the highest immediate heuristic, which often meant it would *always* merge tiles as soon as possible without regard for the future – this can lead to a very bad board configuration that is hard to undo (like combining small tiles away from the big tile, scattering medium tiles all over). Random might occasionally do something "right" (even if by accident), whereas greedy might systematically do the wrong thing in a certain scenario. Of course, a better tuned greedy (or multi-step but not full expectimax)

could beat random, but the lesson is that *foresight is vital in 2048.* A one-step evaluation is not sufficient to consistently improve over randomness in a stochastic setting.

Strengths and Weaknesses of Expectimax; Future Improvements

Strengths of Expectimax AI:

- **High Performance and Tile Achievement:** As shown, expectimax significantly improves the ability to reach high tiles. It makes informed decisions that balance scoring and survival. Expectimax can routinely "win" the game (reach 2048) given enough search depth [24], and it often goes far beyond the basic win condition, achieving extremely high tiles (4096, 8192, etc.) that human players rarely reach. It effectively handles the trade-off between pursuing merges and maintaining board space. The algorithm's strength comes from exploring many possible future sequences, so it can foresee traps (like a move that leads to a dead-end two moves later) and avoid them, which a shortsighted strategy would miss.

- **Handling of Randomness:** Unlike deterministic game solvers, expectimax explicitly handles the uncertainty of tile spawns. It plans for the average outcome, which in practice is a very sound approach for 2048. The random tile generation is not truly adversarial (it's not trying to hurt the player), so expecting an average case tends to yield better moves than assuming the worst. This is borne out in the higher scores compared to minimax. The expectimax approach is well-suited for games like 2048 (or other single-player stochastic games) [13], where probabilities are known. It is robust to the randomness in the sense that it doesn't rely on luck – it *optimizes* for luck. When luck goes bad (sequence of 4's in awful spots), the AI might still lose, but on average it will succeed. This approach maximizes expected score, which is usually the goal.

- **Comprehensive Heuristic Use:** Our expectimax agent effectively combines multiple heuristics (monotonicity, smoothness, empties, etc.) in its evaluation. By searching with this evaluation, it finds ways to satisfy these heuristics in the long run. For example, it will sacrifice a bit of smoothness temporarily if it knows that in two moves it can create a large merge that ultimately improves monotonicity and empties. The search allows the agent to navigate the complex trade-offs between heuristics, something a static strategy would struggle with. The result is a very *balanced* play style – one that keeps the board clean, merges tiles, and builds up big numbers in a corner, all in harmony.

- **Adaptive Depth (Efficiency):** The adaptive depth mechanism is a practical strength, as it lets the AI allocate computation smartly: shallow where possible, deep when needed. This means the AI doesn't waste time early on and can play reasonably fast, and slows down to think when the situation is critical (late game). This adaptability is important for making expectimax feasible to run without massive computational power. Combined with optimizations (like parallelizing the top-level moves as independent branches [20] or using caching), it ensures expectimax can operate within reasonable time per move.

Weaknesses and Challenges:

- **Computational Cost:** The biggest weakness of expectimax is its **exponential growth** in computation with increased depth. Each additional ply (half-move) multiplies the number of states to

evaluate by roughly up to 4 (for moves) times ~2E (for chance outcomes). Even with pruning, this grows very quickly [16] [17]. As noted, depth beyond 3 or 4 can become sluggish without heavy optimization. This makes it hard to scale expectimax much further – going from depth 5 to 6 to 7 might improve performance, but the number of states might become prohibitive. Some implementers have taken it to depth 8 or 9 by using low-level optimizations (bitboard representation, bit tricks to generate moves, precomputed move tables, etc.) and by utilizing symmetrical properties of 2048 state space to prune redundant cases. But for an average implementation, the time might blow up exponentially. In our Python simulator, for example, going from depth 3 to 4 made the AI several times slower per move, and depth 5 would likely be extremely slow. Thus, **expectimax demands either optimized code or limited depth** for practicality. It cannot *perfectly* solve 2048 (which would require exploring the full tree of possibilities) because that tree is enormous.

- **Heuristic-Dependent:** Expectimax is only as good as its evaluation function at the cutoff. If the heuristic is flawed or incomplete, expectimax might make suboptimal choices, especially at the fringe of its search horizon. For instance, earlier we discussed a scenario where adding a "neighbor half-value bonus" heuristic unexpectedly made the AI worse [31] – the heuristic tweak was intended to encourage combining tiles in sequence, but it apparently misled the expectimax search and resulted in lower performance than even the simpler heuristic. This shows that an expectimax agent can be led astray by a bad heuristic (garbage in, garbage out). Our heuristic, while effective, sometimes may not distinguish between two states that actually have different long-term prospects. For example, our monotonicity metric might not fully capture a situation where a slightly "less monotonic" board is actually one move away from a big merge that would fix it. If that occurs at the depth limit of search, the AI might evaluate that state as worse than a more monotonic but ultimately hopeless state. Fortunately, a depth-3 or 4 search often sees one or two moves ahead to resolve such ambiguities, but not always. There are configurations where the heuristic's guidance could be suboptimal beyond the search depth.

- **Endgame and Tunnel Vision:** A specific weakness we observed is that sometimes expectimax can **struggle in the endgame** when the board is very full and only one or two empty spots remain. If the depth isn't sufficient, it might not foresee an imminent loss because the branching factor is huge (many possible tile placements) and perhaps the average outcome looks okay even though a few bad placements would be fatal. If those bad placements occur, the game ends. A human or a perfect solver could identify a "dangerous" configuration that expectimax might rate as fine on average. Essentially, expectimax can be a bit over-optimistic in such tense situations. A minimax agent would never be caught by a worst-case spawn, but expectimax sometimes is. If the probabilities are low it's okay, but it's a risk. In practice, increasing search depth when empties are few (which we do) helps alleviate this – as depth goes up, expectimax will see that a certain bad spawn next move could lead to a loss a couple moves later, and thus it will avoid the situation if any safe alternative exists. But at limited depth, this can be an Achilles heel: expectimax might set up a position that is usually fine but has a 5% chance of immediate defeat (because it can't see beyond depth that those 5% outcomes are deadly). If that 5% event happens, the AI loses abruptly. Over many runs, those unlucky losses might happen often enough to notice. In contrast, a minimax agent would never voluntarily enter a state that has any chance of immediate loss if it can avoid it (making it lose less in that manner, though it loses in other ways like slowly painting itself into a corner). This is essentially the classic difference between playing for maximum expected value vs. playing conservatively to minimize risk.

- **Not Learning or Adapting:** Expectimax as implemented is a fixed-depth tree search with a static heuristic. It doesn't learn from experience or improve over time except by adjusting the heuristic weights offline. This is fine for a known game like 2048, but it means it won't dynamically adjust strategy beyond what's coded. There have been Reinforcement Learning approaches (e.g. using TD learning or deep learning) for 2048, but interestingly, those have not clearly outperformed the top expectimax-based agents [17]. The RL agents can learn their own evaluation functions, but the hand-crafted heuristic combined with expectimax has proven extremely effective. Our approach relies on the heuristic designer's insights.

Possible Improvements:

Given the above, several improvements and extensions could be explored:

- **Refined Heuristics:** We could improve the evaluation function by incorporating more sophisticated features or better weight tuning. For example, many top agents use a **position-weighted sum** (gradient) instead of or in addition to monotonicity. This involves having a predefined weight matrix (often monotonic or snake-like) and computing $\sum w_{ij} \cdot \text{tile}_{ij}$ [29]. The Black Prism AI found a "snake" shaped weight matrix extremely effective [32] [25], as it enforces an even stricter ordering of tile values on the grid. One could combine that with our metrics. Also, explicitly adding a term for **"number of merges available"** could help the evaluation (so the AI might prefer to keep two 64s adjacent because that's a potential merge). Another idea is an **"aggressiveness"** term to slightly encourage making high merges sooner (or discourage too much smoothness at the expense of progress). Tuning the weights via automated methods (like genetic algorithms or CMA-ES) on self-play games could yield a stronger heuristic as well [33].

- **Dynamic Weighting or Stage-based Strategy:** The importance of heuristics can change as the game progresses. For instance, keeping empty spaces is crucial in the early and middle game, but if you are close to achieving 2048, you might temporarily accept a lower empties count to set up the final merge. One could adjust the weights based on certain conditions (e.g., if highest tile is >= 1024, maybe increase the weight on monotonicity and decrease empties a bit to focus on consolidation for the win). Our implementation does not do this explicitly, but it's something to consider.

- **Search Depth and Pruning:** If more computing power is available or optimizations applied, increasing the search depth will directly improve performance. Depth 5 or 6 expectimax will outperform depth 3/4. Using bit-level representations (represent the 4×4 grid in a 64-bit integer, which is common in 2048 AI implementations) can speed up state evaluations immensely (via lookup tables for row moves, etc.), allowing deeper search within the same time. Additionally, implementing a **transposition table** (cache) can avoid re-evaluating states that are reached via different paths. For example, if the same board configuration appears again (which can happen due to symmetry or different move orders leading to the same merge result), the AI can reuse the previously computed expectimax value. This is tricky with randomness (because path probabilities differ), but one can cache the heuristic evaluation of states at depth=0 to avoid recomputing heuristics for the same state multiple times in one search. Also, not every random tile placement needs equal consideration – some AI players simulate a limited number of random outcomes (Monte Carlo sampling) instead of averaging over all possibilities, to reduce branching. An extreme version of that is a **Monte Carlo Tree Search** approach where you simulate random playouts to estimate state values instead of fully expanding all chance nodes. That could be an alternative to expectimax that might cut down

computation while still handling chance (though in 2048, expectimax with a good heuristic has proven very effective, so MCTS hasn't been as popular).

- **Parallelization:** As mentioned, one straightforward improvement is to evaluate the top-level moves in parallel threads, since they are independent. We could spawn four threads (for up/down/left/right) to explore each move's subtree concurrently [20]. This would nearly quarter the decision time on a quad-core CPU, for example. Our Python implementation didn't do this (threading in Python has GIL issues, but multiprocessing or using a faster language would work). In an optimized setting, parallel expectimax can allow an extra depth or just faster move responses.

- **Learning Approaches:** Another direction is to train the evaluation function using machine learning (reinforcement learning or supervised learning from self-play or from expectimax itself). Some projects have used neural networks to approximate the value function of 2048 states. These learned evaluations can then be used with expectimax (to cut it off at a certain depth and use the NN value for leaves). The advantage is potentially a more accurate evaluation function that might pick up on subtle patterns that our linear weighted heuristic doesn't cover. That said, the most famous 2048 AI (by *nneonneo*) used a manually tuned weighted heuristic with expectimax and was extremely successful – it can consistently get 2048 and often 4096 or 8192. So the heuristic approach is already quite strong.

- **Handling Larger Boards or Variants:** Our simulator and discussion focused on the standard 4×4 2048. If one wanted to adapt to larger grid versions (like 5×5 boards, or the game "8192" etc.), expectimax would still apply but the state space grows. The heuristics might need retuning (empties are relatively more abundant on larger boards, etc.). Expectimax might need even more pruning on a larger board due to branching.

In conclusion, the expectimax-based 2048 AI demonstrates the power of lookahead search in a single-player stochastic environment. Its main strength is achieving extremely high performance (far beyond human average scores) by systematically optimizing expected outcomes [24]. Its weaknesses are computational demand and reliance on good heuristics, but these can be managed with clever techniques and tuning [34] [21]. Further improvements could make the AI even stronger or more efficient. With the current approach, we have a robust AI that **far exceeds greedy or random play**, and provides an intelligent strategy for playing 2048 almost optimally. The combination of *heuristic knowledge* (empty tiles, monotonic order, etc.) and *expectimax planning* is what allows it to master the game [5] [8].

---

[1] [2] [3] [17] [23] [24] [28] cs229.stanford.edu
https://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf

[4] [5] [6] [8] [14] [29] [30] Beginner's guide to AI and writing your own bot for the 2048 game | by Bartosz "Ripp" Zadrożny | Medium
https://medium.com/@bartoszzadrony/beginners-guide-to-ai-and-writing-your-own-bot-for-the-2048-game-4b8083faaf53

[7] digikogu.taltech.ee
https://digikogu.taltech.ee/en/Download/e6c7f3d1-3451-43fd-ac59-ca2b38b449a3

[9] [10] [18] [19] [20] [25] [26] [27] [31] [32] The Black Prism
https://theblackprism.ch/2048-AI.html

11 33 GitHub - salvacarrion/2048: AI strategies to beat 2048 (game)
https://github.com/salvacarrion/2048

12 13 16 21 22 34 Expectimax Search Algorithm in AI - GeeksforGeeks
https://www.geeksforgeeks.org/artificial-intelligence/expectimax-search-algorithm-in-ai/

15 What is the optimal algorithm for the game 2048? - Stack Overflow
https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048