Utilizing Probability to Solve 2048 Game

Aloisius Adrian Stevan Gunawan - 13523054[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]kremix6767@gmail.com, 13523054@std.stei.itb.ac.id

***Abstract*—This paper explores the utilization of probability to play the popular puzzle game, 2048, optimally. Through analyzing the tile generation and game mechanics, we aim to develop algorithm that increases the chance to obtain higher scores. Through simulations and probability-driven calculations, we investigate how probability could increase the chance of getting a higher score.**

***Keywords*—2048, probability game, solving 2048**

I. Introduction

In this world, probability governs over many things, ranging from the simple 2048 game, to complex thing like quantum computing. Probability offers mathematical computation to manage uncertainty. This ability enables individuals to make choice when faced with unknown possibilities.

The 2048 game that has been made by Gabrielle Cirulli in 2014, challenges its player to combine the tiles of number into larger number, up to 2048 where the player wins the game. To play the game, the player must slide the tiles, combining tiles that have the same number in it. This game has an element of uncertainty, where each time the player slides, a random new tile that have—either a "2" or a "4" —in it. This makes it more unpredictable for the players, and thus introducing strategical elements in the game.

This paper investigates the use of probability to improve decision-making in playing the game 2048. Specifically, we aim to maximize the chances of attaining the highest score.

II. Theory

A. *Probability*

Probability is mathematical method to measure the chance of an event happening in number. The value 0 indicates impossibility, and the value 1 indicates certainty.

Event is one possible situation in probability. Sample space is the set of all possible events. In other words, sample space is all possible outcomes of probability. This concept is used everywhere, including the game 2048.

B. *2048 Game*

The game uses 4x4 grid board, with each grid holding a value of either 0 (none), or 2^n, with $n > 1$. Each turn, the player would be able to move all tiles to up, down, left, or right. The mechanics of moving is by pushing all tiles to the chosen direction, and if there were tiles that contains same value, the game would merge those tiles into one tiles that contains double the value of the merged tiles. Each tile can only be merged once per move.

With each move, this game generates a random tile that contains either number "2" or "4", in a random tile. By investigating the source code, we could see that the probability of number 2 appearing is 90%, whereas the probability of number 4 appearing is 10%. This introduces several possible strategies for this game.

C. *Strategy*

Each move should consider the probability of achieving the desired result, making it more advantageous for the next event. This method involves predicting tile placement after the move, avoiding deadlock at all cost.

Risk management is also important, determining whether short-term-gain is better, or worse. There are several risk-management methods:

1. Score-based: prioritizing score over anything
2. Free space-based: prioritizing freeing up space over anything.
3. Corner strategy: prioritizing in making the tile with the largest value go to a corner of the board, to maximize the possibilities to merge faster in the middle.
4. Edge strategy: Always trying to make the largest tile on one edge, but still flexible (unlike corner strategy)
5. Snake strategy: Arranging the tile with the largest score as the head, and adjacent to it is the second largest tile, and so on.

Although there are lots of other strategies, these strategies are more general and known intuitively to beginner player.

With the cost of modularity or time complexity, AI-based decision making like Markov chains and Monte Carlo Simulation are made possible.

D. *Markov Chains*

Markov Chains is a method that stores all possible board condition. This model would be used to decide where to move. From the example in figure 1, green outlines mean highest possible tile has been reached (in a 2x2 board, 8 is the highest tile one can achieve), red means deadlock (no other possible movement could be made), and black mean the game continues, not reaching an endpoint yet.

For each of the move, the probabilities of it happening is calculated manually. First, we resolve the board in the direction, for example, Left (L). This will move all tiles to the left, and

merges all mergeable tiles. After moving, there will be a random number generated in the board. This number would be either number 2 with 90% chance, or number 4 with 10% chance.

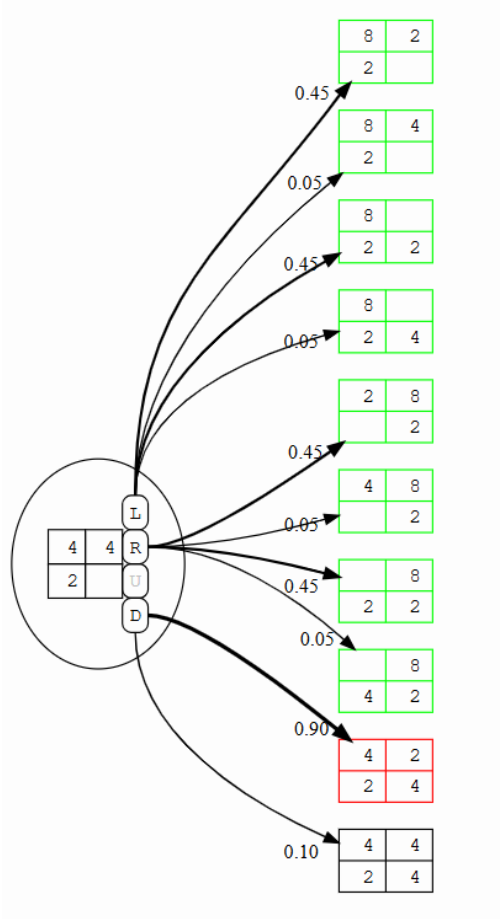


Fig 1. Example of markov chain of one board condition in a 2x2 board

Source: https://jdlm.info/articles/2018/03/18/markov-decision-process-2048.html

With markov chains method, every possible board condition is precalculated, and the highest-probability move would be chosen to win the game. When the player played effectively, the player is guaranteed to win the game.

However, markov chains consumes a lot of memory. For the original 4x4 board, it requires at least trillions of board condition. With the increasing board size, it will exponentially increase the modularity.

III. Methodology

This paper focuses on investigating the use of probability, to play optimally, given limited resources. With that in mind, we aim to analyze the correlation between probability and optimizing gameplay. This includes understanding the game mechanics, doing expectimax simulations, optimizing heuristic strategy, and doing implementation.

A. Game mechanics

Before delving deeper into the solution, understanding the mechanics of the game is a must. This portion will analyze more into the rules of the game, movement, merging tiles, scoring, and generating new tiles.

B. Simulation

The simulation presented will use expectimax method. Expectimax is a method that explores several moves into the future, then returns the highest average heuristics score. This means that although expectimax is not as capable as doing markov, it still holds as the most efficient move within a few steps into the future.

C. Heuristic strategy

Heuristic strategy is a form of probabilistic modelling that will be integrated into the decision-making. This is the cog that guides the decision-making to decide the most optimal move. The heuristics include empty tiles maximization and highest score maximization.

The heuristic prioritizes scoring. This will have the most impact on the decision-making processes. After the scoring, comes the other heuristic factor.

Empty tiles give more flexibility towards the next turn. Having no empty tiles and no valid moves would count as game over.

D. Implementation

This entire methodology will be implemented in python, using numPy library. The game simulation using expectimax method are designed to be effective and efficient.

IV. 2048 Mechanics

The original 2048 game always starts with 4x4 board that have 2 tiles with number “2” or “4”. Each turn, the player can choose to move the tiles to one direction—up, down, left, or right—and then end the turn.

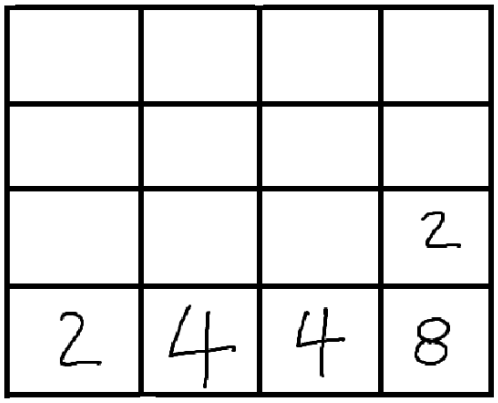


Fig 2. Board state example

Moving the tiles starts by shifting all tiles to the corresponding direction until there is another tile in that direction that have the value of more than 0, or it hits the wall. The analogy would be making a strong gravity force into that direction. For example, in figure 2, moving up would make number 2 in (4,1) go to (1,1), number 4 in (4,2) go to (1,2), number 4 in (4,3) go to (1,3), number 2 in (3,4) go to (1,4), and number 8 in (4,4) go to (2,4).

After shifting the tiles, the game will also check for mergeable tiles. It will try to combine tiles with the same value, starting from the direction to the opposite direction. The combining process is by moving the value of the corresponding tiles to 1 tile to the direction. Take figure 2 row 4 for example. Assume the player chooses right. The code will check (4,8) which is unmergeable, then (4,4) which is mergeable, making it (0,8), then it checks (2,0), unmergeable. This process will leave row 4 with [2,0,8,8]. After that, the game will move it to the direction once more, eliminating 0s in between.

Following moving to a direction, the game will generate a random tile, with number 2 or 4. By inspecting the source code, the probability of number 2 is 90%, whereas the probability of number 4 is 10%. The tile chosen as the generated tile is randomly chosen. This makes the game unpredictable as the score also relies heavily on tile generation.

Scores is directly represented by the tiles number. It is the sum of all the tiles in the board. This makes scoring one of the main considerations in the heuristic strategy.

V. IMPLEMENTATION

The implementation of this uses python as the programming code and also uses the numpy library. The implementation would be available on https://github.com/mimiCrai/2048Solver.

This implementation will ask the state of the board and return the most optimal move to make. The input of the board should be given space and/or newline for each change of tile. The board scanning starts from (1,1), then (1,2), (1,3), (1,4), then (2,1), and so on until (4,4). For example, figure 2 will be inputted as in figure 3. The program would then prints the board state as a form of confirmation.

```
Input the current board state!

Value of row-1 col-1: 0
Value of row-1 col-2: 0
Value of row-1 col-3: 0
Value of row-1 col-4: 0
Value of row-2 col-1: 0
Value of row-2 col-2: 0
Value of row-2 col-3: 0
Value of row-2 col-4: 0
Value of row-3 col-1: 0
Value of row-3 col-2: 0
Value of row-3 col-3: 0
Value of row-3 col-4: 2
Value of row-4 col-1: 2
Value of row-4 col-2: 4
Value of row-4 col-3: 4
Value of row-4 col-4: 8
0       0       0       0
0       0       0       0
0       0       0       2
2       4       4       8
```

Fig 3. Example of input in the program

After the input, solution to the board-state is next. The main strategy I use is by using expectimax heuristic strategy. This method finds the most optimal move, by choosing the highest average heuristic score of each direction.

```python
def find_best_move(board, depth=2):
    best_move = None
    best_score = float('-inf')
    for direction in ['up', 'down', 'left', 'right']:
        new_board = move(board, direction)
        if not np.array_equal(board, new_board):
            score = expectimax(new_board, depth - 1, False)
            if score > best_score:
                best_score = score
                best_move = direction
    return best_move
```

Fig 4. The code to find the best move
Source: https://github.com/mimiCrai/2048Solver

As seen in figure 4, This code snippet will simulate the next *depth* move in the future. *Depth* directly increases the accuracy, but also exponentially increases time complexity, and linearly increases modularity.

Find_best_move() requires helping functions, such as move() and expectimax().

```python
def expectimax(board, depth, is_maximizing):

    if depth == 0 or game_over(board):
        return evaluate_board(board)

    if is_maximizing:
        max_eval = float('-inf')
        for direction in ['up', 'down', 'left', 'right']:
            new_board = move(board, direction)
            if not np.array_equal(board, new_board):  # Jika langkah valid
                eval = expectimax(new_board, depth - 1, False)
                max_eval = max(max_eval, eval)
        return max_eval
    else:
        # Tile baru muncul (2 atau 4)
        empty_positions = [(i, j) for i in range(4) for j in range(4) if board[i][j] == 0]
        if not empty_positions:
            return evaluate_board(board)

        total_score = 0
        for (i, j) in empty_positions:
            for value, prob in [(2, 0.9), (4, 0.1)]:
                new_board = np.copy(board)
                new_board[i][j] = value
                total_score += prob * expectimax(new_board, depth - 1, True)
        return total_score
```

Fig 5. Function expectimax
Source: https://github.com/mimiCrai/2048Solver

Expectimax here (figure 5) have 3 parameters, that is, board, depth, and is_maximizing. Board holds the value of the current state of the board, depth holds an integer, representing the number of steps in the future to check, and is_maximizing holds boolean value.

When the is_maximizing is false, the function will try to

simulate the condition after moving the board. The function would first check for endgame condition. If it is endgame, it will simply return the current score of the board. The endgame condition triggers when the board is full. This is only true in this function, as it separates move and tile generation. When the endgame condition is not triggered, the program would generate a situation with both number 2 and number 4 in every empty tile. In each situation, it will call another expectimax, doing a recursive function. With each recursive, the function would return numbers that would then be multiplied by the probability of that board state happening, to be added into the total score. Finally, the program would return the total score.

When the is_maximizing is true, the function would simulate moving the board into four directions. For each of the direction, if the move is valid (for example, in figure 2, moving down is invalid), the function would call another expectimax function that will return total average heuristic score. The function would find the largest one, and then returns it.

```
def game_over(board):
    """return endame condition"""
    endgame = True
    win = False
    for i in range(4):
        for j in range(4):
            if board[i][j] == 2048:
                win = True
            if i >0:
                if board[i][j] == board[i-1][j]:
                    endgame = False
            if j > 0:
                if board[i][j] == board[i][j-1]:
                    endgame = False
    return endgame | win
```

Fig 6. Function game_over
Source: https://github.com/mimiCrai/2048Solver

The game over condition would trigger when there are no more possible move condition, or when there is number 2048 in the board, triggerring the endgame win condition.

```
def move(board, direction):
    """moving board"""

    newboard = np.copy(board)
    newboard = slide(newboard, direction)
    newboard = merge_tiles(newboard, direction)
    return slide(newboard, direction)
```

Fig 7. Function move

Source: https://github.com/mimiCrai/2048Solver

This function holds as the frame of the move mechanic. It calls for two other function helper, accordint to the order. It calls for sliding tiles once, then merge all mergeable tiles corresponding to the direction, then it calls sliding tiles once more. This will beautifully create a move mechanic like the original 2048 game.

```
def merge_tiles(board, direction):
    if direction == 'up':
        for i in range(3):
            for j in range(4):
                if board[i][j] == board[i+1][j] and board[i][j] != 0:
                    board[i][j] *= 2
                    board[i+1][j] = 0
    if direction == 'down':
        for i in range(3, 0, -1):
            for j in range(4):
                if board[i][j] == board[i-1][j] and board[i][j] != 0:
                    board[i][j] *= 2
                    board[i-1][j] = 0
    if direction == 'left':
        for i in range(4):
            for j in range(3):
                if board[i][j] == board[i][j+1] and board[i][j] != 0:
                    board[i][j] *= 2
                    board[i][j+1] = 0
    if direction == 'right':
        for i in range(4):
            for j in range(3, 0, -1):
                if board[i][j] == board[i][j-1] and board[i][j] != 0:
                    board[i][j] *= 2
                    board[i][j-1] = 0
    return board
```

Fig 8. function merge_tiles
Source: https://github.com/mimiCrai/2048Solver

Merging the tiles is a little trickier. It checks for the tiles closest to the direction of the move, then check 1 tile closest to it, that is at the opposite direction. This move would be repeated thrice in total, with each move gets one shift to the opposite direction of the move.

```
def slide(board, direction):
    if direction == 'up':
        for i in range(4):
            for j in range(4):
                if i > 0:
                    if board[i][j] > 0:
                        tempi = i
                        while tempi > 0 and board[tempi-1][j] == 0:
                            board[tempi][j], board[tempi-1][j] = board[tempi-1][j], board[tempi][j]
                            tempi -= 1
    if direction == 'down':
        for i in range(3, -1, -1):
            for j in range(4):
                if i < 3:
                    if board[i][j] > 0:
                        tempi = i
                        while tempi < 3 and board[tempi+1][j] == 0:
                            board[tempi][j], board[tempi+1][j] = board[tempi+1][j], board[tempi][j]
                            tempi += 1
    if direction == 'left':
        for i in range(4):
            for j in range(4):
                if j > 0:
                    if board[i][j] > 0:
                        tempj = j
                        while tempj > 0 and board[i][tempj-1] == 0:
                            board[i][tempj], board[i][tempj-1] = board[i][tempj-1], board[i][tempj]
                            tempj -= 1
    if direction == 'right':
        for i in range(4):
            for j in range(3, -1, -1):
                if j < 3:
                    if board[i][j] > 0:
                        tempj = j
                        while tempj < 3 and board[i][tempj+1] == 0:
                            board[i][tempj], board[i][tempj+1] = board[i][tempj+1], board[i][tempj]
                            tempj += 1
    return board
```

Fig 9. Function slide
Source: https://github.com/mimiCrai/2048Solver

This code moves the value in the board to the corresponding direction, starting from the tile to the most opposite direction. This will prevent 0s from getting in between the move.

VI. Time Complexity

2048 game is played on a nxn board. The size of the board directly affect the time complexity of the program. This section will calculate the time complexity of the program.

The function move contains function slide and function merge_tiles that each have $O(n^2)$ complexity. This means that the function move itself have time complexity of $O(n^2)$. Game_over and evaluate_board also have $O(n^2)$.

Expectimax is a recursive function that have increasingly large time complexity, depending on the depth. Expectimax have the time complexity of $O(n^2 * 4^{depth})$, with the worst case of $O(128^{depth})$. This makes expectimax gets exponentially longer the larger the depth is.

The overall time complexity is $O(2^{8*depth})$, meaning that the larger the depth is, not only the answer would be more accurate, but the time complexity would also grow, exponentially.

In ideal condition, most modern processors could handle up to 10^9 operations per second. This means that using this information, we could approximate maximum depth for maximum.

$$128^n \leq 10^9$$

$$n \leq \frac{9}{{}^{10}log(128)}$$

$$n \approx 4$$

The ideal depth for modern processor is around 4. More than 4 would require at worst $128^{depth-4}$ seconds to process.

VII. Testing

Testing the code could be seen as subjective, as we can't have definitive proof, unless we do markov modelling. However, we can see the difference of accuracy when we change the depth.

```
0       0       0       0
0       0       0       0
0       0       0       2
2       4       4       8
Best move:  left
```

Fig. 10. Result using depth of 1
Source: https://github.com/mimiCrai/2048Solver

```
0       0       0       0
0       0       0       0
0       0       0       2
2       4       4       8
Best move:  left
```

Fig. 11. Result using depth of 2
Source: https://github.com/mimiCrai/2048Solver

```
0       0       0       0
0       0       0       0
0       0       0       2
2       4       4       8
Best move:  right
```

Fig 12. Result of using depth 3
Source: https://github.com/mimiCrai/2048Solver

```
0       0       0       0
0       0       0       0
0       0       0       2
2       4       4       8
Best move:  right
```

Fig 13. Result of using depth 4
Source: https://github.com/mimiCrai/2048Solver

As shown in figure 10, figure 11, figure 12, and figure 13, the result may vary depending on the depth of the program. Intuitively, moving right is better than moving left, because the largest tile, 8, is kept in the corner of the board. This shows that higher depth increases the accuracy.

VIII. Conclusion

This paper has successfully proven that with the use of probabilistic algorithm, it increases the chance to obtain higher scores in 2048 game. Although it is limited by the time complexity and modularity, this method is still accurate up to several moves ahead. In the future, optimization and upgrades, especially regarding the heuristic strategy that includes score and empty tiles, are possible. There is always room for improvement.

References

[1] Jdesmileer, "The Mathemathics of 2048: Optimal Play with Markov Decision Process," Available: https://jdlm.info/articles/2018/03/18/markov-decision-process-2048.html. Accessed: Jan 7, 2025.

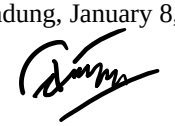
[2] G., Barghavi, "Mathematical Analysis of 2048, The Game," Available: https://www.ripublication.com/aama17/aamav12n1_01.pdf. Accessed: Jan 7, 2025.

[3] S., Alexsey, "Computational bounds for the 2048 game," Available: https://arxiv.org/pdf/2303.07266. Accessed: Jan 7, 2025.

Declaration

With this, I declare that I this paper I wrote is of my own. I do not copy, translate, nor plagiarize the work of others.

Bandung, January 8, 2025



Aloisius Adrian Stevan Gunawan, 13523054