Minimax and Expectimax Algorithm to Solve 2048

Ahmad Zaky | 13512076[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13512076@std.stei.itb.ac.id*

***Abstract*—2048 is a puzzle game created by Gabriele Cirulli a few months ago. It was booming recently and played by millions of people over the internet. People keep searching for the optimal algorithm for solving the game. Here are few approaches: minimax and expectimax algorithm. The idea is to calculate all possible moves and then select the best move by some functions. Alpha-beta pruning is also used to speed up search time. The result depends on the limit of the depth of the search tree. The greater the limit, the better the result. At some point, expectimax algorithm reaches 80% winning rate.**

***Index Terms*—2048, Depth Limited Search, Expectimax, Heuristic Function, Minimax, Search Tree**

I. Overview of the Game

Gabriele Cirulli is an Italian user interface designer and web developer. In March, he built the game 2048, which is actually a modification of some other game. Unexpectedly, in a few weeks it became a worldwide hit and played by more than 23 million people. [1]

The game itself is simple. You are given a 4×4 board, where each tiles may contain a number inside it. The numbers will always be a power of two. Initially, there are only two numbered tiles with number 2 or 4. You may alter the board by pressing arrow keys, and the tiles in the board will move according to your moves. For example, if you press left button, all of the tiles will go to the left. If the numbers on the adjacent cells match, they will merge. Consider the following board. If you move to the left, the "2" at the top of the board will move to the top-left corner, and the pair of "2" at the bottom will combine creating a "4". Additionally, after each moves, a new number will appear on one of the empty cells uniformly. The new number will always be "2" or "4".

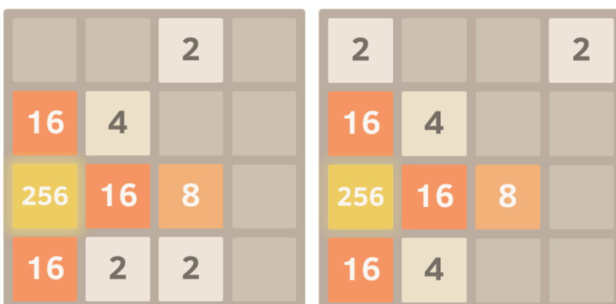

Image 1 Example of a move in 2048. After a "left" move, the board on the left will become the right.

The objective of the game is to reach 2048 tile. This can be extended to reaching maximum tile as possible. Actually there is a scoring system applied to each moves, but that will not be our concern in this paper.

II. The Algorithms: Minimax and Expectimax

2048 can be viewed as a two player game, a human versus computer game. The human's turn is moving the board to one of the four directions, while the computer's turn is placing 2 or 4 in one of the empty cells. Thus, we will use *minimax* and *expectimax* algorithm.

Minimax and expectimax are the algorithm to determine which move is the best in some two-player game. Because of that, both of them usually called *decision rule*. Actually there are other algorithms and rules that may be better than two of them, but we will only use those because the implementation is straightforward.

A. Minimax Algorithm

Minimax is a decision rule for minimizing the possible loss for a worst case scenario, or in the other word for the maximum possible loss. That is why it is called *minimax*. The rule originally used for a two-player zero-sum game (a game which a player's gain is exactly the same with the loss of the other player's, and vice versa).

The minimax theorem states that [2]

> For every two-person, zero-sum game with finitely many strategies, there exists a value V and a mixed strategy for each player, such that
> - Given player 2's strategy, the best payoff possible for player 1 is V, and
> - Given player 1's strategy, the best payoff possible for player 2 is −V.

That means player 1's strategy guarantees him a payoff of V regardless of player 2's strategy. This theorem was first published in 1928 by John von Neumann in his paper called "Zur Theorie der Gesellschaftsspiele Math" about the game theory.

The graph below is the example of the search tree using minimax algorithm.

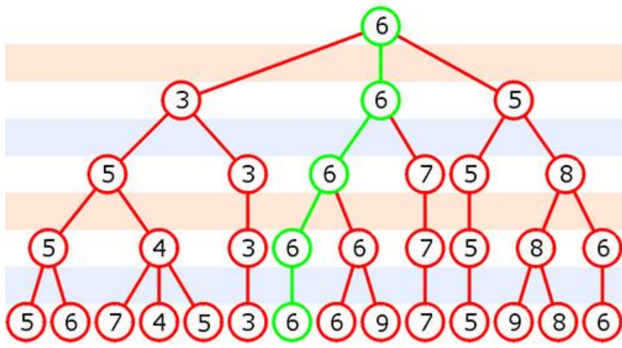

Image 2 Example of the search tree of the minimax algorithm

The nodes the depth of which is even (note that the root has depth of zero) are Player 1's turn, which we want to maximize its gain. The nodes with odd depth are the Player 2's turn. In every player 1's turn, we want to maximize the gain, and in every player 2's turn, we choose the minimum possible gain (gain here is player 1's gain, or equivalently player 2's loss), so we can guarantee to have at least such amount of gain.

The value of gain will be calculated when we reach the leaf node. In most of the cases, it is not possible to reach the "real leaf node" i.e. the situation where the game ends, because the depth is too big. So, sometimes the minimax algorithm is implemented with *depth-limited search* instead of boundless depth-first search. Because we do not know the exact value of gain at the leaf node, we will only calculate the estimation of it, using some *heuristic function*, which will be discussed in the next few chapter.

For the sake of completeness, below is the pseudocode of the minimax algorithm using depth-limited search. In the code below, the player we want to maximize the gain is player 1.

```
function minimax(node, depth, turn)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if turn == player1
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, player2)
            bestValue := max(bestValue, val);
        return bestValue
    else
        bestValue := +∞
        for each child of node
            val := minimax(child, depth - 1, player1)
            bestValue := min(bestValue, val);
        return bestValue

(* Initial call *)
minimax(root, depth, player1)
```

Pseudocode 1 Minimax algorithm

The original naive minimax algorithm requires to expand all the search tree, which can be very expensive in terms of complexity. We can improve it by *alpha-beta* pruning.

B. *Minimax Algorithm with Alpha-Beta Pruning*

Alpha-beta pruning is used to cut the number of nodes in the search tree evaluated by minimax algorithm. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move, so we do not need to evaluate it further. [3]

The pseudocode of alpha-beta pruning is showed below.

```
function alphabeta(node, depth, α, β, turn)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if turn == player1
        for each child of node
            α := max(α, alphabeta(child, depth - 1,
                     α, β, player2))
            if β ≤ α
                break (* β cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1,
                     α, β, player1))
            if β ≤ α
                break (* α cut-off *)
        return β

(* Initial call *)
Alphabeta(root, depth, -∞, +∞, player1)
```

Pseudocode 2 Alpha-Beta pruning over naive minimax

The term α is the *lower-bound* of the gain, and β is the *upper-bound*. At player 1's turn, we want to maximize the gain by evaluating all possible moves. The value of α is updated every time. When at some point we find that β is not greater than α, or in the other words the lower bound is not greater than the upper bound, then we can safely stop here, because we have proved that there are better moves. This is called β cut-off. The α cut-off is equivalent.

C. *Expectimax Algorithm*

In minimax algorithm, we will choose the move based on maximum (or minimum) gain. It means that we only need the minimum or maximum value of the child nodes. In expectimax algorithm, when we evaluate opponent's node, we will calculate all of possible moves, weighted by the probability of the occurrence. In the other words, we will calculate the *expected value* of the gain over all possible cases. [4]

For better understanding, below is the pseudocode of expectimax algorithm.

```
function expectimax(node, depth, turn)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if turn == player1
        bestValue := -∞
        for each child of node
            val:= expectimax(child, depth-1, player2)
            bestValue := max(bestValue, val);
        return bestValue
```

```
    else
        expectedValue := 0
        for each child of node
            val:= expectimax(child, depth-1, player1)
            expectedValue += Probability[child]*val;
        return expectedValue

(* Initial call *)
expectimax(root, depth, player1)
```

Pseudocode 3 Expectimax algorithm

For the most of the cases, the probability of each nodes will be the same. In our cases, 2048, the probability is already known. The choice between the empty tiles is uniform, and the probability of the occurence of 2 is 90%, while the probability of 4 is 10%.

III. Observation of the Game

For we already know the algorithm, from now on we will only focus on the heuristic function that will be used in our algorithm. Basically, a heuristic function is a value that ranks alternatives in search algorithms. This value will be greater if the condition is “better”. But which better is better? We will build our heuristic function based on some observation of the game 2048.

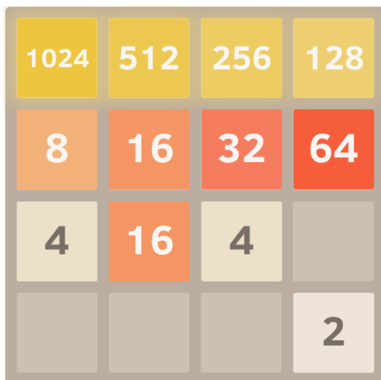

Image 3 The desired board to achieve 2048

Some observation that follows directly from playing 2048 is:

- To get a 2048 tile, we must combine two 1024 tiles
- To get two 1024 tiles, we must have a 1024 tile together with two 512 tiles, and so on

So, we may want to transform our board into something like the above diagram. However, the process is not straightforward.

A. *Empty Spaces*

The obvious thing is to leave as much empty spaces as possible. If there are many empty spaces, it is easier to “breathe”, to move the tiles and to get even bigger number.

B. *Smoothness*

The objective of the game is to get a 2048 tile, and to get a certain tile we have to combine two tiles of smaller values, and to combine them, these two tiles must be adjacent. So, when the values of adjacent tiles are close, we say that the grid is *smooth*. The term *smoothness* here is determined by the difference between the values of adjacent tiles. The smaller the differences, the smoother the grid. Here is an example of non-smooth grid.

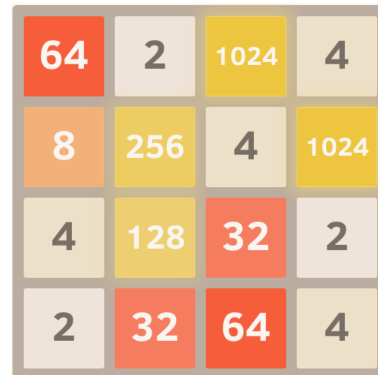

Image 4 An example of a non-smooth grid

Both 1024 tiles are adjacent with 4 and 2 tiles; the smoothness is very small. Even in the case above, the game is already over as there is no more available move.

C. *Big Tiles in Border*

The consequences of the smooth grid are we have to keep the large values together, and leave empty spaces for small pieces to transform into the bigger tiles. But smoothness is not enough; below is the example of a fairly-smooth grid, but has no available moves.

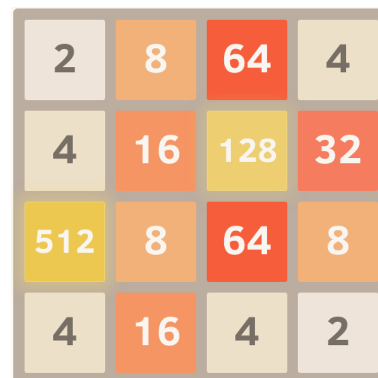

Image 5 When the big valued tiles are not in borders of the grid, the game will over immediately

We can see that in the board above the big valued tiles are placed in the middle of the grid. Any tiles located in the middle of the grid are easily driven to anywhere. However, to get our board like the neat board showed in the Image 3 earlier, we need to keep large valued tiles in the border of the grid. Moreover, we have to keep them in the same side of board, so that the tiles can be easily combined. Anyway, together with smoothness, the big tiles will meet in the same border automatically, so we do not have to check it explicitly.

Another observation shows that it is better if the largest current tile is placed in the corner. The logical explanation is the largest tile is the least possible tile to be combined

with the others. So, the corner is the best place for it.

Image 6 A situation where tiles with large values are placed in the border of the board, but not in the same side of the board

We will use those three observations above for building our heuristic function, which will be explained further in the next chapter.

IV. Heuristic Function

Based on the observations in the preceding chapter, now we can determine the heuristic function that will be used in our search algorithm.

A. *Empty Spaces*

We add this factor by simply give some large values to all empty tiles we find.

B. *Smoothness*

Similar to above, we involve this factor by decreasing the heuristic value by the total of all differences of all adjacent pair of tiles. This value can be weighted by multiplying it by some constant.

C. *Big Tiles in Border*

For every tiles in the grid, we give penalty if it is placed in the middle of the grid. The amount of the penalty is $C \times distance\ to\ the\ nearest\ border \times tile\ value$ where C is some constant. Varying this value yields different result.

Optionally, we may include largest-value-in-corner factor by giving reward to a position where tiles with largest value placed in the corner.

Let us write the heuristic function defined above in a more formal way. If E denotes the number of empty tiles, D denotes the total of all differences of adjacent pair of tiles, and P is the sum $distance\ to\ the\ nearest\ border \times tile\ value$ over all tiles, then the heuristic value is

$$H = A \times E - B \times D - C \times P$$

Where A, B, C are some constants. In the implementation, we choose $A = 4096$, and $B = 10$ and $C = 10$. The choice of A is free, as long as it large enough to dominates the heuristic value.

Additionally, if at some point we reach dead-end where no more moves are available, the heuristic function will return some very small value (-∞).

V. Implementation and Performance

Because the original 2048 was coded with javascript, minimax and expectimax will also be implemented in the same language. The integration between the AI (Artificial Intelligence) and the game itself will be easier. But, the disadvantage is that javascript is slow; the maximum possible depth for the search tree is very limited. Later we will show that the deeper the search tree, the better the AI in getting a 2048 tile.

When the depth of minimax search tree is limited to only 4, the winning rate is very low. But the performance (in terms of speed) is very high. In one second the AI can do more than 10 moves easily. The detailed statistics is given below. The data is taken from one hundred run of the same algorithm.

Highest Tile	Percentage
256	1%
512	17%
1024	45%
2048	33%
4096	4%

Table 1 The statistics of minimax algorithm when the depth is limited to 4

The maximum score achieved by the algorithm is 60484. As we can see, the most of the times the AI can reach 1024. But in 45% of the runs, it fails to get 2048 tile.

Expectimax algorithm cannot be pruned like minimax with alpha-beta pruning, because to calculate the expected value, all configurations of child nodes must be calculated. Actually there is another version of expectimax, which combines minimax and expectimax, called *expectiminimax* algorithm, that can also be pruned by something similar to alpha-beta pruning, but we will not implement it.

To achieve the same speed as minimax algorithm with depth limit 4, the depth limit of expectimax should be less than 2. When the depth limit is 3, the maximum number of moves the AI can do is only three moves. When the depth limit is increased to 4 or more, it takes more than 10 seconds to evaluate a move, so practically it is not effective.

When the depth limit is 2, the performance and winning rate is no different with minimax algorithm with depth limit 4.

Highest Tile	Percentage
256	5%

512	8%
1024	31%
2048	53%
4096	3%

Table 2 The statistics of expectimax algorithm when the depth is limited to 2

Compared to minimax, this algorithm is slightly better. The winnng rate (to achieve 2048 tile or better) of minimax is 37%, while this one have 56% of winning rate.

Now we will increase the depth limit and see if it has better winning rate. If the minimax algorithm's depth limit is increased to 8, the AI can still calculate more than 3 moves per second. From 20 consecutive runs, the result is given in the table below.

Highest Tile	Percentage
512	5%
1024	25%
2048	55%
4096	15%

Table 3 The statistics of minimax algorithm when the depth is limited to 8

The winning rate improves significantly, from only 37% to 70%. The chance of getting 4096 is even greater, and there is only one run that ends with 512 as the largest tile. The downside is that those 20 games take more than 6 hours, or equivalently one game takes about 20 minutes. This is about 5-10 times slower than the earlier minimax where the depth limit is only 4.

Expectimax algorithm is even showing better improvement after increasing its depth limit to 3. The average score is 39163, and the winning rate is 80%. The detailed result is shown in the table below.

Highest Tile	Percentage
1024	20%
2048	40%
4096	40%

Table 4 The statistics of expectimax algorithm when the depth is limited to 3

It never fails to obtain 1024, and the probability to get 4096 tile is very high. It proves that the winning rate will be increased as the depth limit increasing.

VI. Better Implementation and Performance

There are many methods and algorithms that can be proved better [5]. Two of them will be explained here, because the two of them used the same algorithm, minimax and expectimax.

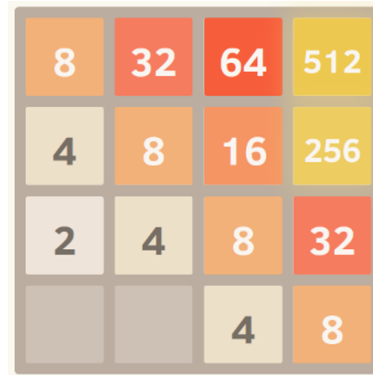

Image 7 The example of perfectly monotonic grid

The first one can be seen in http://ov3y.github.io/2048-AI/. It also used minimax algorithm with alpha-beta pruning. The difference is the heuristic function. It calculates three factors: smoothness, empty tiles, and *monotonicity*. The last heuristic ensures that the values of the tiles are all either increasing or decreasing along both the horizontal and vertical directions. The diagram above shows an example of a perfectly monotonic grid. This heuristic will keep the board very organized. The other two heuristics are similar with those mentioned in the earlier chapter.

This AI is also implemented in javascript. The default depth limit is around 6, and the AI can produce about 10 moves per second. Anyway, the winning rate is very high. It obtains 2048 tile 90% of the times. While this one is already good, there is even better AI which will be explained below.

A user of stackoverflow with username *nneonneo* developed expectimax algorithm to solve 2048. The heuristic used is even simpler: the heuristic only calculates empty tiles and add some large values if the largest value is on the edge. But it has a clean winning rate: 100%. It even reaches 16384 tile 13 times from 100 runs. The best run is showed below. The score is 377792, an astonishing result.

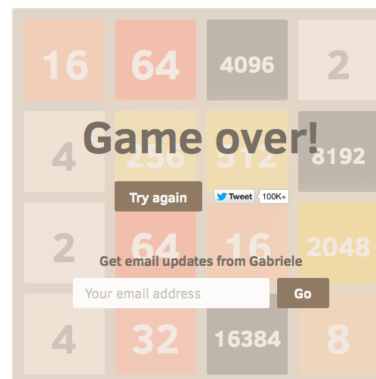

Image 8 The best run of the nneonneo's AI

What makes it has excellent performance is that the AI is implemented using C++. The language is extremely faster than javascript. It is capable to calculate over 100 millions operations per second. That is why the AI is able to evaluate expectimax with maximum search depth of 8.

The AI is connected to the original 2048 game in the browser via remote control. The average move rate is 6 to 10 moves per second. When the search depth is limited to six, the move rate is increased to more than 20. That makes this AI is the best, both in performance and the winning rate.

VII. CONCLUSION

Both minimax and expectimax search algorithm along with the heuristic function we use are fairly good to solve the 2048 game. The winning rate is higher when we increase the depth limit of the search tree. Minimax algorithm with depth limit of 8 has around 75% winning rate, while expectimax with depth limit of 3 has around 80% winning rate and has around 40% chance of getting a 4069 tile. Even better result can be achieved by improving the heuristic function or implement the algorithm better.

VIII. APPENDIX

All of the codes used for testing the algorithms are uploaded to https://github.com/azaky/2048-AI. You can also visit http://azaky.github.io/2048-AI/ to try the live performance of the AI itself.

IX. ACKNOWLEDGMENT

The author thanks to Mr. Rinaldi Munir and Mrs. Masayu Leylia Khodra for their teaching and endorsement in the Strategy of Algorithms course during this semester. The author also thanks to his friends and fellow students of Informatics/Computer Science ITB for their assistance all this time.

REFERENCES

[1] Cirulli, Gabriele. “2048, success and me”. http://gabrielecirulli.com/blog. Accessed on May 13th 2014, 14.16 UTC +07.00.
[2] Osborne, Martin J., and Ariel Rubinstein. A Course in Game Theory. Cambridge, MA: MIT, 1994. Print.
[3] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc.
[4] Weld, Dan. “Adversarial Search, Artificial Inteligence Autumn 2012”.http://courses.cs.washington.edu/courses/cse573/12au/slides/04-minmax.pdf. Accessed on May 17th 2014, 12.38 UTC +07.00.
[5] “What is the optimal algorithm for the game, 2048?”. http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048/22389702#22389702. Accessed on May 17th2014, 19.23 UTC +07.00.

STATEMENT

I hereby declare that the paper I wrote is my own work. It is not a copy nor a translation of someone else’s paper, and not a plagiarism.

Bandung, May 18-th 2014

Ahmad Zaky | 13512076