2048 Project Report (Full Version: https://docs.google.com/document/d/11EH0mT85OHNKcX8UvDgzRU2pYr03af9TtfLNWQ37NcA/edit?usp=sharing)

Please also visit: https://youtu.be/OjUdP4OTX5E and https://github.com/Normanwqn/2048 or https://github.com/Normanwqn/2048/blob/master/out/artifacts/2048_jar/2048.jar for the jar file.

Abstract

Different searching algorithms are attempted to play the game 2048. A snake-shaped weight matrix board evaluation function is introduced to guide the otherwise impossible searching process. Greedy best-first search algorithm performed poorly but with high efficiency. Informed Depth First Search algorithm performed slightly better reaching above 128 at a probability of around 50%. Traditional minimax search which regards the game system as the opponent is able to reach a value of 512 more than half of the times but it is still not able to find a valid set of moves to reach 2048. A modified version of Expectiminimax search, which introduces the notion of chance nodes to take into consideration of the randomness, wins the game at a decent chance of 85%.

I. Introduction:

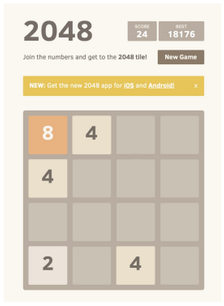


Fig. 1. the Original Game Interface

2048 is a popular single-player sliding block puzzle game designed by an Italian web developer Gabriele Cirulli [1]. Cirulli claimed that the total time of game play since its release has exceeded 3,000 years. The ultimate goal of the game is to slide the numbered tiles in a 4*4 grid [Fig. 1] to combine the tiles into a tile whose value is 2048. The player is able to slide the numbered tiles in four ways by pressing the left, right, up or down key, which slides the numbered tiles in the respective direction if there is enough room for the tiles to slide into. The system will randomly generate a tile of 2 or 4 with a probability of 80% and 20% respectively to fill a random empty tile on the board after each move. Two tiles of the same value can be combined into a tile of a larger value which is the sum of the value of two same tile. The player loses the game once the 4*4 game grid is fully filled with 16 tiles and it is not possible to combine any two tiles to continue the game. Even though the player wins once the tile of the largest value reaches 2048 before the game ends, he or she is allowed to continue the game play to combine the tiles into higher value such as 4096, 8192. A scoreboard on the top of the board interface records the current score which initializes at 0 at the start of the game. The score is incremented by the value of two combined tiles or the newly generated tile. The total score serves as an indication of the sum of all the tiles which have appeared on the board. Similar to many arcade games, the best score ever achieved by the user is also displayed in the user interface.

Even though the game seems simple, it is intriguing and even addictive to human player because of its surprising unexpected complexity. The player has to perform a significant number of merges in order to generate a tile of 2048. To generate a tile of 4, one merge between a tile of 2 and another tile of 2 needs to take place. Likewise, a tile of 8 requires merging two tiles of 4. However, one tile of 4 itself needs one operation of merging between two tiles of 2. Thus, there has be two seperate merges from tiles of 2 to form two tiles of 4 and it takes an additional merge to for these two tiles of 4 to merge into a tile of 8. Hence, it requires 3 merges together to form a tile of 8. Likewise, since $16 = 8 + 8 = (4 + 4) + (4 + 4) = [(2 + 2) + (2 + 2)] + [(2 + 2) + (2 + 2)]$, a tile of 16 requires 4+2+1 = 7 merge operations. It can be deduced that for a tile value of 2^n, there needs to be $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$ operations. Therefore to reach a tile of 2048 which is 2^{11}, the player or the agent has to at least perform $2^{11-1} - 1 = 1023$ merging operations. Even though the player may be able to merge several tiles in one move, the number of operations needed is still presumably high. This is a crucial reason which will be discussed why some algorithms may perform suboptimally.

II. Approaches

Circuili originally programmed the game 2048 in JavaScript which is intended to run in a browser. For experimental purposes and convenience, the GUI and the game logic is reprogrammed into Java inspired by the following series of tutorial [3]. Various changes in Java may affect the performance of the programme comparing with the original Javascript one.

The goal of this project is to utilise and analyse different searching techniques which aims to play the game 2048 without any human intervention. Like many other board games, the agent is aware of all the information which are encompassed by the board such as the position of each tile, the value of each tile, the position of the empty tiles, etc. However, as mentioned, the game will automatically generate a tile of value 4 or 2 after each move and the agent is not aware of exactly where the next tile will appear. The agent is only aware of the probability whether the randomly generated tile would be 2 or 4 (80% or 20% respectively).

A. Uninformed Search Approaches

The baseline of the project was originally set to utilise several basic uninformed search algorithms to attempt playing the game. Uninformed search are also referred to as blind search meaning that the algorithms utilises no additional information beyond what is provided on the board. (P81 [2]). I initially attempted to work with Depth-First Search. Depth-First Search aims to expand the deepest node in the frontier of the current game search tree. It is as if the agent keeps going deep into the search tree until it loses or wins the games as illustrated in Fig 2. An analogy in this specific game would be that the agent would continuously press a certain key, such as LEFT, in the initial stage until the ending condition is met. Afterwards, it will back track one level up and press the next preferred key in sequence such as RIGHT. Theoretically, the agent will traverse all the possible moves.

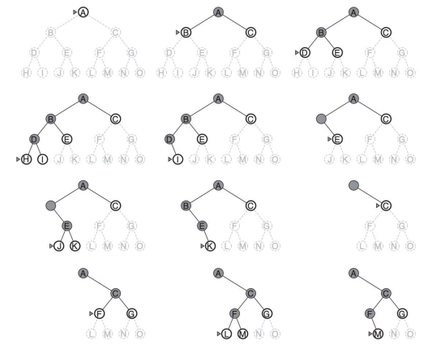


Fig. 2 Depth-First Search (As illustrated, the deepest branches of the game on the left are explored first)

However, a fatal flaw of this algorithm soon reveals itself once I completed the code. I did not realise the depth a normal computer is able to explore. Depth-First search has a time complexity of $O(b^m)$, b being the branching factor which is usually 4 in this case and m being the maximum depth of the tree. As discussed in the Introduction section, it takes at least 1023 merging operations to make a tile whose value is 2048. It is reasonable to suggest that it will take at least a few hundred moves to reach 2048. Thus b^m would be amount to complexity that is unlikely to be computed in a reasonable amount of time. Subsequently, an attempt is made to implement Depth-Limited search which turns out to be futile as there simply isn't a viable solution to reach 2048 within a depth around 12, the depth limit which enables computation within a reasonable response time. Therefore, it can be concluded that depth-limited search is incomplete in this case and no solutions were found by the algorithm. Therefore, it now seemed that uninformed search methods are not good approaches to this problem.

B. Informed Search Approaches

Informed search strategies uses problem-specific knowledge beyond the definition of the problem itself (P92 [2]) In this case, these search strategies must use information beyond just what is shown on the board including the value of each tile, the position and number of the empty tiles, etc. Thus, there needs to be a board evaluation function to serve as an indication of the favorability of the board. It is desirable for the search to glide towards the more favourable situation in general. In this case, the goal of the heuristic is to keep as few tiles as possible while making the tiles as large as possible. Several heuristics are attempted, such as scoring the board based on "monotonicity" and "smoothness" based on "ovolve's" published JavaScript implementation. Monotonicity is defined to be the degree to which "the values of the tiles are all either increasing or decreasing along both the left/right and up/down directions" [4]. The concept of monotonicity captures the intuition which many experienced players have, that the highest value should be clustered in the corner while the values of the others tiles should decrease diagonally as illustrated by Figure 3 in which the highest value is situated at the top right hand corner. It is thought that such configuration is preferable as it prevents higher valued tiles from getting isolated which reduces their opportunities to combine into larger tiles. Meanwhile, such configuration is believe to benefit the overall organisation of the tiles on the board [4]. "Oovle" also considers smoothness to be an integral part of the board evaluation function. Smoothness

is described to "measure the value difference between two tiles" [4]. A board that resembles Figure 4 is said to be very smooth as all the tiles are of the exact same value, 1024. Smoothness is an indication of how many potential merges there are currently. Therefore, it is beneficial to reduce smoothness as much as possible to leave more empty tiles which is generally regarded beneficial to prolong the gameplay. The algorithms also includes a separate penalty for free space which is not resulted due to the board's smoothness [5]. Lastly, the maximum value of all the tiles on the board is also taken into consideration. Due to the complexity of the "ovolve's" overall board evaluation, please refer to the source-code for detailed description. The following is an abstract overview of the board evaluation function:

$$Eval(b) = 0.1 \times S(b) + 1.0 \times M(b) + log(E(b)) \times 2.7 + 1.0 \times Max(b).$$

(b refers to the 4*4 matrix which consists of all the tile value. $S(b)$ denotes the smoothness sub-heuristic. The smoothness heuristics outputs a negative value. Hence, it is preferable for the Evaluation function to lower the magnitude of the smoothness function. $M(b)$ denotes the monotonicity function. $E(b)$ refers to the number of empty tiles on the board. $Max(b)$ outputs the largest value among the value of all the tiles on the board) The weights which associates with the compositional function is set by the author of the programme which is originally intended to be run on a Minimax algorithm.

Fig. 3. Monotonicity (Source: [4])

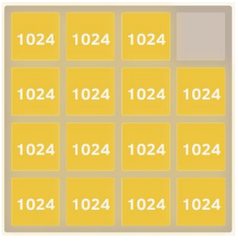


Fig. 4. Smoothness (Source: [4])

The ultimate choice of heuristic is, however, based on a relatively simplified version proposed by Yun et al [6]. A 4*4 weight matrix is used to simulate the monotonicity heuristics as each tile is associated with a different weights which affects the ultimate evaluation score differently. A position associated with higher weights will gravitate towards having tiles of larger value because a large tile value multiplied by a large weight will increment the overall heuristic more significantly. A snake-shaped weight matrix (Fig. 5) is utilised for the final implementation of the various search algorithms. A snake-shaped matrix seems to enable the agent to just "swipe along" to merge two tiles together while adhering to the "keeping the largest value in the corner" heuristic. The original evaluation function sets to directly multiply the weight matrix by the value matrix. Hence:

$Eval(b) = \sum_{i=0}^{3}\sum_{j=0}^{3} weight[i][j] \times board[i][j]$ (Source: [6]). A modification is made to use the log_2 value of the board instead to reduce the calculation complexity and space complexity. Thus,

$$Eval(b) = \sum_{i=0}^{3}\sum_{j=0}^{3} weight[i][j] \times log_2(board[i][j])$$

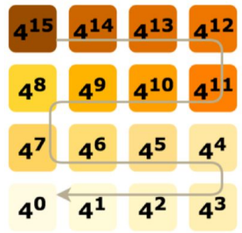


Fig. 5. Snaked-Shaped [4]

The following will discuss some searching method that is implemented.

1. Greedy best-first search

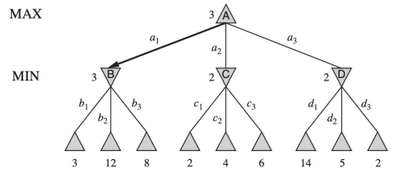
Greedy best-first search tries to expand the node that is closest to end goal [2]. In this case, the algorithms directly looks into the node in the frontier which has the largest board evaluation score while overlooking some other possibilities. Thus, it is conceivable that the greedy approach is neither complete or optimal even though the algorithm runs considerably fast.

2. Depth Limited Search Guided by the Board Evaluation Function

Unlike the previously described depth limited search, the modified version is guided by the board evaluation function.

3. Mini-max

Even though minimax is used for multi-agent games, it can be adapted to suit the gameplay of 2048. Minimal simulates a zero-sum game in which the player tries to maximise the final score on every level in which he or she is in control while the opponent tries to minimise the score of the game. Figure 6 illustrates this idea by showing that the second layer which is controlled by the "MIN" player or the opponent is actively trying to minimise the whatever it obtains from the leaves of the tree. Whereas, the "MAX" player "strives" to find the maximum value from its leave nodes.



Fig, 6. Minimax Game Tree [2]

Regarding the game 2048, the MAX player is considered to the agent itself while the MIN player is conceived to be the game system that is randomly spawning tiles after each valid move. The MIN nodes represent all the possible location or value the tile can be spawned in. The MIN player tries to minimise the favorability of the board after the new tile is spawn, meaning that the board is actively attempting to disrupt the snake-shaped configuration of the tiles whenever possible. Thus, it forces the MAX player to choice the moves which are least likely to disrupt the relatively more desirable board configuration. The following describes Minimax algorithm:

$$MINIMAX(board) = \begin{cases} Eval(b) & \text{if Terminal-Test(b)} \\ \max_{a \in Actions(b)} MINIMAX(RESULT(b,a)) & \text{if Player} = \text{MAX} \\ min_{a \in Actions(b)} MINIMAX(RESULT(b,a)) & \text{if Player} = \text{MIN} \end{cases}$$

$a \in Actions(b)$ symbolises all the possible actions (e.g. moving LEFT, RIGHT, UP, DOWN) that can be taken by the agent.

4. Expectiminimax

However, as mentioned previously, there is a probability of 80% that the newly spawned tile has a value 2, and 20% that the value is 4. Hence, it is more optimal to further incorporate such information into the algorithm. However, the algorithm is different from traditional expectiminimax in a sense that only the MIN player, the game system which randomly places the tiles, introduces randomness and unpredictability into the system not the MAX player. Each MIN players has two Chance node children which represents the cases where a tile of 2 is spawn or a tile of 4 is spawned. When "backtracking" into their parent node, the chance nodes together return the sum weighted by their probabilities.

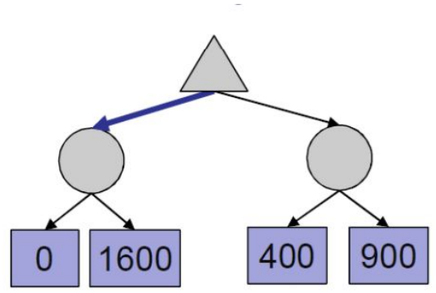


Fig. 7. An Example of the Implemented Expectiminimax Tree. The triangle denotes the MAX player. The circle denotes the MIN player. The rectangles denotes the Chance nodes which has its associated probabilities and evaluation score.

The following is an abstract description of the implemented Expectiminimax algorithm.

Algorithm 1 Expectiminimax

```
if state s is a Max node then
    return the highest Expectiminimax-value of succ(s)
end if
if state s is a Min node then
    return the lowest Expectiminimax-value of succ(s)
end if
if state s is a Chance node then
    return the average Expectiminimax-value of succ(s)
end if
```

(Source: [6])

III. Experiments

All the experiment data can be accessed from: https://docs.google.com/spreadsheets/d/1NaIrL_l3dIxVqo-fhSTf4zdN8L_3j9UCM8YaSJvwEAA/edit?usp=sharing

Source code: https://github.com/Normanwqn/2048/tree/master