2048 Solver

Sayantani Ghosh
ghosh@iastate.edu
Ph.D. Student
Department of Computer Science
Iowa State University

Piyush Mantri
mantri@iastate.edu
Masters Student
Department of Computer Science
Iowa State University

Dinesh Papineni
dinesh9@iastate.edu
Masters Student
Department of Computer Science
Iowa State University

Abstract

The report describes the 2048 game solver. The solver is built on MinMax algorithm and Alpha-Beta pruning. It discusses the approach used to build the solver and the different heuristic techniques that are used. It achieves an accuracy rate of 89% and an average run time of 70 seconds per game.

Introduction

Creating robust artificial intelligence is one of the greatest and interesting challenges for the game developers. Yet, the quality of the AI in the game is extremely vital for the success of the game. Thus, artificial intelligence and games have always complemented each other well. It is interesting how some of the basic algorithms of Artificial intelligence can be applied to solve complex games. In this paper, we would be presenting how we made use of the minimax and alpha-beta pruning algorithms for solving 2048, a puzzle game that has gained a lot of popularity over the past year.

2048: What it's about

2048 is a single player puzzle game, developed by Gabriele Cirulli. The objective of the game is to slide numbered tiles on a grid to combine them and create a tile with the number 2048. The game is over when the board is full but there are no more moves possible. The goal is to achieve the 2048 tile, although, the game continues even after the 2048 tile is achieved.

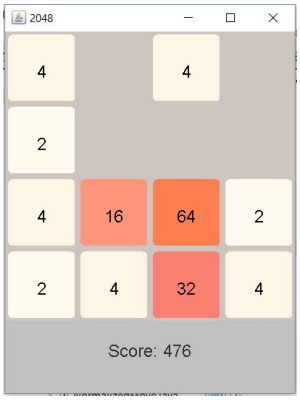

Figure 1: 2048 Game Board

Game setup: The board consists of a $4x4$ grid which can accommodate 16 tiles. The game starts with two tiles of 2's on the board. You can move the tiles left, right, up and down, which slides all the tiles on the board in that direction. After any movement, adjacent tiles with same value is merged to form a new tile with the value of the sum of the 2 merged tiles and in the position relative to the direction of movement. Moreover, after every movement, a new tile of value 2 or 4 is generated at a random position on the board. The goal is to merge tiles so as to achieve a 2048 tile. The game is over if all tiles gets filled up. The game-play continues even after the 2048 tile is reached, but a game is considered a win if the 2048 tile is achieved, and a loss if the board is full before the 2048 tile is achieved.
Scoring: Every time a merge happens, the score is updated by the value of the tile formed by merging. Although maximizing the score is not an objective of the game.

Developing the game

Programming language: Java.
Platform: Eclipse.

System Model

For the sake of convenience, we will divide the implementation tasks into 3 parts:

Board design: The board is represented by a $4 * 4$ 2-dimensional matrix, where each cell represents a tile in the game.
The game starts with two cells in the matrix of value 2, and the rest of the cells are of value 0 (Refer to figure 2) . **Movements:** There are four possible movements in the

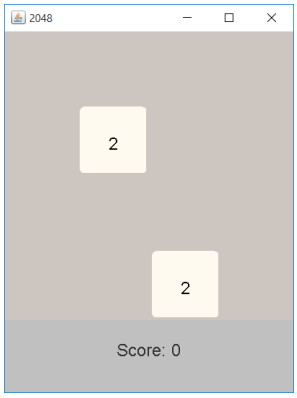

Figure 2: Game start

game - up, down, right and left. Each of these movements move all tiles in the direction of the movement. For example, the tiles will be moved to the rightmost empty spot when the movement chosen is right, and so on.
If two tiles of the same value are adjacent to each other, they will merge together (discussed in the next task section).
If a movement is not possible in any direction, there is a message printed to the player, and he/she can try to move the tiles in the other directions, if possible. If any move is not possible in any direction, i.e., the board is full, the game ends.

Merging: If two tiles of the same value are adjacent to each other, they will merge together to form a new tile whose value is the sum of the two tiles merged. For example, if after a movement to the left, after all the tiles are on the leftmost available position in that row, there are two tiles of 4's beside each other (Refer to figure 3), then they will merge to form an 8 tile in the leftmost available position (Refer to figure 4).
After every movement in any direction, a new tile of value 2 or 4 is generated at any random empty position in the matrix. The probability of the 2 tile is 90% and the probability of the 4 tile is 10%.
The score is updated by the value of the merged tile after every merging operation occurs. For example, if after a movement, two 4 tiles are merged to an 8 tile, the score will increase by 8 points.

User Interface

The board design for our game is similar to the look of the original game. We have implemented the board using JFrames and JPanels. The tile colors are different based on what numbered tile it is. A score panel displays the score, which is updated after every move. We have implemented

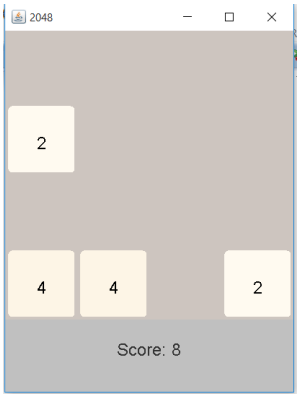

Figure 3: Pre-movement game board

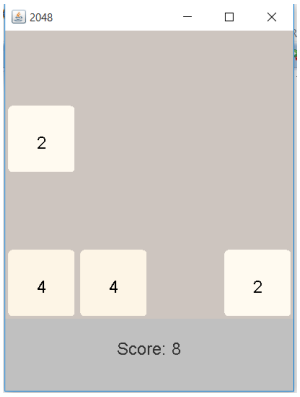

Figure 4: Game board after one move to the left

multi-threading to make the tile movements smoother.
As far as the movements are concerned, we have key press listeners that apply the moves in the four directions based on the up, down, left and right keys. Pressing the key 'A' will calculate the next best possible move from the current state based on certain algorithms and heuristics and implement one move.
When there are no moves possible, a popup displays a game over notification (Refer to figure 5).

Developing the solver

Algorithms

For developing our solver, we have implemented two algorithms: *Minimax* and *Alpha-Beta Pruning*. We will discuss them below:

Minimax The minimax (a.k.a. minmax algorithm) algorithm is a recursive algorithm which can be used for solving two ply games. One can think of the algorithm as similar to the human thought process of saying, "OK, if I make this move, then my opponent can only make three moves, and each of those would let me win. So this is the right move to make." In each state of the game a value is associated. This algorithm searches through the space of possible game states creating a tree like structure which is expanded until it

Figure 5: Game over popup

reaches a particular predefined depth. Once those leaf states are reached, their values are used to estimate the ones of the intermediate nodes. Each level represents the turn of one of the two players (MAX and MIN). In order to win each player must select the move that minimizes the opponents maximum payoff.

2048 as a minimax search problem: Since the original 2048 game is a single player game, it can be hard to imagine how the minimax algorithm can be applied in such a game. We consider the first player as the person who plays the game (MAX) and the second player as the computer which randomly inserts values in the cells of the board (MIN). The first player tries to maximize his/her score and ultimately reach to the 2048 tile. The computer in the original game was not designed to block the user by selecting the worst possible move for him. Instead, it randomly inserts values on the empty cells. If the computer was programmed to play optimally in the original game, reaching the 512 tile too would be very daunting. However, when we are modelling 2048 as a minimax search problem, we assume that the computer would select the most optimal move always. That way, our agent (MAX or the first player) is ready for making a move even against the best move of a perfect MIN (computer). By ensuring this, we would be sure that MAX will chose the best move against the computer choosing random moves. Below is a representation of how the tree would look like, for a 2x2 grid. The players choices for the left, right, up and down directions are shown from the root. To each of these choices, the possible insertion of a 2 or a 4 for a computer are shown, and the tree can be expanded so on. Similarly it can be extended to the 4x4 grid in 2048.

Alpha-Beta Pruning Since our state space is huge, going to greater depths could make the agent work very slow. One way of going to further depths while maintaining comparable speed is by using Alpha-Beta pruning. This is just an Expansion of the Minimax algorithm. It heavily decreases (prunes) the number of nodes that we must evaluate/expand. To achieve this, the algorithm estimates two values the alpha and the beta. Alpha is the best outcome MAX can force at previous decision on this path, and Beta is the best outcome MIN can force at previous decision on this path. If in a given

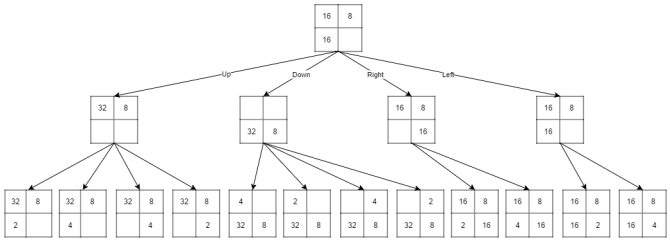

Figure 6: 2048 as a Minimax search problem:

node the beta is less than alpha then the rest of the subtrees can be pruned.

Heuristics

We have used four different heuristics out many other heuristics we have tried. We tried assigning different weights for each of the heuristic and had run a performance check. We finally achieved best performance by assigning equal weight to each of them. Following are the four heuristics:

Monotonicity: We have used monotonicity to ensure that the highest values on the board are aligned in increasing or decreasing order in one of the directions (left to right, right to left, top to bottom, bottom to top). It also ensures that higher values tiles are always at one corner of the board. Finally it keeps the board organized by merging the smaller valued tiles with the higher valued tiles. For example, in figure 7, the 32, 64, 128 256 and 512 are lined up such that the merges will happen sequentially to finally obtain a 1024 tile.

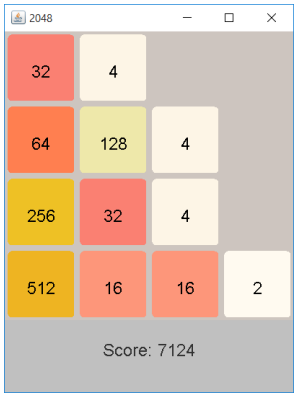

Figure 7: Monotonicity

Merge score: Calculates the number of possible merges on the board for a move. It will help us in selecting a move that will result in maximum number of merges. Maximum merges ensure us in reaching the goal state faster.
For example, consider two different possible moves at a particular state. Say one of them results in two merges and the other results in six merges. The heuristic add more

weight to the move that results in six merges.

Number of empty tiles: Calculates the total number of empty tiles resulting for a particular move. It will help us in selecting a move that will in maximum number of empty tiles on the board. Maximum number of empty tiles on the board ensures more number of possible moves per game and better performance of the agent.
For example, consider two different possible moves at a particular state. Say one of them results in four empty cells and the other results in five empty cells. The heuristic add more weight to the move that results in five empty cells.

Max value: Calculates the maximum value possible on the board for a move. Ensure that a move resulting in higher value is selected to reach the goal state faster.
For example, a move to the right that merges two 32 tiles to form a 64 tile will be chosen over a move to the left that merges two 8 tiles to form a 316 tile.

Evaluation

We have tested the code on 100 runs and made the following observations:

- The solver achieved an accuracy of 89% in hitting the 2048 tile.
- The average run time for each game is 70 seconds. Note that we define the runtime as the amount of time required to get a 2048 tile (or the amount of time the game runs for if it ends before getting a 2048 tile.
- The solver achieved an accuracy of 20% in hitting the 4096 tile (Refer to figure 8).
- The highest valued tile that we could achieve is 8192 with a very low accuracy.

Figure 8: 4096 achieved

Future Work

Overall, we are very happy with what we have achieved in the project. However, if we had more time in hand, we would have liked to deal with a few more issues.

Improving the speed: Improving the speed of the algorithm will allow us to use larger depth and thus, get better accuracy.

Tuning the heuristics: We can experiment with the way that the scores are calculated, the weights and the board characteristics that are taken into account. Our approach still has room for improvement.

Tuning the depth: We had tried to tune the depth of the search depending on the game state, but the program worked drastically slow. So, we can try to fix that.

Trying more algorithms: We can use the Iterative deepening depth-first search algorithm which is known to improve the alpha-beta pruning algorithm.

References

[1] $en.wikipedia.org/wiki/Minimax$

[2] $en.wikipedia.org/wiki/Alpha\%E2\%80\%93beta\_pruning$

[3] $gabrielecirulli.github.io/2048/$

[4] Russel Norvig, *Artificial Intelligence: A modern Approach*, Third Edition.

Acknowledgments

We would like to thank Dr. Jin Tian for all his help, support and encouragement on this project, and for making the subject interesting for us to be able to give in our best on the course.
We would also like to thank Hoda Gholami, our teaching assistant for the course, for bearing with our numerous questions and helping us out whenever we needed it.