2048: An AI Agent

CS221 Fall 2014: Final Project Report [p-final]

Team Members: [Jason Lewis, Alec Powell, Jai Sajnani]
SUNetIDs: [jlewis8, atpowell, jsajnani]

December 12, 2014

1 Introduction

Game playing is a very interesting topic area in Artificial Intelligence today. Specifically, the game 2048 is a great strategy game to which AI principles can be applied. It is straightforward to emulate gameplay and quantify results. Further, as it is a very difficult game for humans to win, we are interested in noting how much better a computer can perform against humans. All utility is received at the point when a 2048 tile is achieved, and there are no intermediary rewards. Further, as the computer only adds a random element, it is not adversarial. As this is not a classic two-player zero sum game, it is again more novel.

2 Background

This popular game is set on a 4x4 grid of tiles where each tile has a value that is a power of 2. The premise is to move the tiles in any of the cardinal directions towards combining tiles of equal value (2 + 2) to create a tile of double the value (4). The objective is to have create a tile of value 2048, though the game can continue beyond this point.

In 2048, the input is a sequence of moves where the moveset is {UP, DOWN, LEFT, RIGHT}. A move will attempt to move all of the tiles on the board in that direction. A tile will stop moving when blocked by another tile of a different value, and will combine with a tile of the same value. After a move, a 2 or 4 tile is placed randomly by the computer on an empty space on the board. 2s appear with 90% probability, and 4s with 10% probability. The output of a move is a new board state, where the tiles have been moved and/or combined appropriately and a new tile has been randomly placed on the board. A move is invalid if it does not cause a change in the state of the board before the random tile is placed. The game is over if the board is full and the player has no valid moves available. The player does not win unless a tile with value 2048 was created.

3 Task Definition

For our project, we have built a 2048 engine that employs the expectimax algorithm to work towards achieving a 2048 tile. In addition to the computational solver, a visual solution has

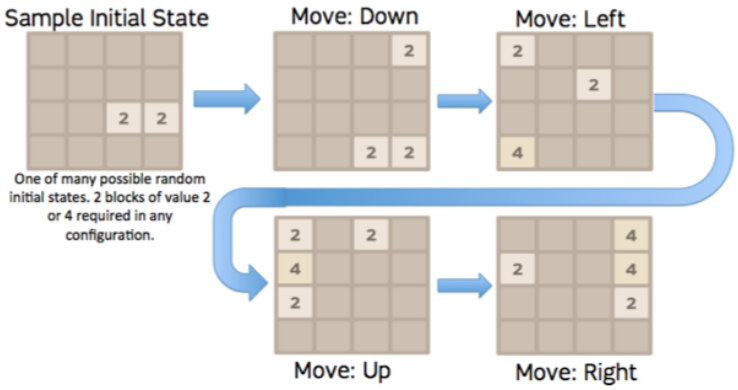

Figure 1: Sample of initial state and four move transitions

Figure 2: Game over state that didnt win (Left) and Ongoing game that did win (Right)

been created that allows the 2048 engine to communicate with a graphical user interface so that users can watch the puzzle being solved in real time at a reasonable speed. Our metric for success is an agent that follows a policy that can with a high degree of reliability, reach a board configuration that contains a 2048 block and complete this in a reasonable amount of time.

4 Related Work

Given the popularity of the puzzle and the problem, there have been multiple attempts to develop an artificially intelligent agent to solve 2048. These range from agents that take completely random moves, which even after 1 million games did not achieve 2048 once to highly optimized agents that claim to achieve 2048 100% of the time, and can even reach tiles of greater value, like 4096 and 8192. While these solvers have not been verified and published in peer-reviewed journals, there are published articles discussing various methods for the game. A study that implemented TD learning saw a 97% success rate, and a study comparing various algorithms deemed that a Depth-Limited-Search was more effective than Monte-Carlo or Minimax, but did not published numerical evidence on win rates. Given the

general hint to look towards a Depth-Limited-Search, we implement an expectimax variant of this algorithm here.

Further, with an abundance of accurate web solvers now available, any one can be used as a point of comparison. The sample oracle used, with a win rate of 90% can be visualized here: http://www.crazygames.com/game/2048-artificial-intelligence.

5 Approach

5.1 Goals

Our goals for this project were twofold: we wanted an agent that could play the game 2048 fast enough (speed) without diluting its winning ability (accuracy). What we found when beginning the implementation of our agent was that speed was the problem to focus most of our efforts - optimizations would have to made in order to get its speed up to par. At large depths, the 2048 game tree is simply too big to handle calculations quickly enough for each game state. Thus, we researched ways to improve the efficiency of our python code as well as a variable depth function to limit the game tree search depth.

5.2 Game Model

In defining the model for our project, we formalize a game. As we accept that games are a type of state space model, we realize that our model is very similar - there is a start state, a set of actions to proceed from each state, a set of successor states for each state-action pair, and a test to see if a current state is an end state. The end state has an associated value based on the final score given the tile values on the board.

We are adapting a game-playing model covered in class (player vs. nature). In our program, the game is modeled as a set of states, where each state is the current position of tiles on a 4x4 grid board. Tiles are added incrementally, and the number of given tiles on the board can increase or decrease, so the state must capture the number of tiles currently present and their exact location within the grid. The state class (defined as State2048) has two attributes that influences the algorithm: state.board which contains the tiles for a board and state.transitions which contains a list of successor states.

The *board* attribute is an array of 16 bytes that represents the tiles on two-dimensional 4x4 matrix. At each index of the array is a byte n, where 2^n is the actual value of the corresponding tile in the gameplay of 2048.

The *transitions* attribute is a list of possible states that the board can transition to (before the random tile is placed) by choosing any valid direction from the moveset. To move from state to state, there are four possible actions that the user can take: Moveset = {UP, DOWN, LEFT, RIGHT}. Depending on the position of tiles on the board at a given state, it may not be possible to move in one or more of these directions, so our model accounts for this and thus the transitions set will be smaller. When the transitions list is empty, there are no valid moves left and the game is deemed over.

Now, to formalize our definition of the model:

State s: A 4x4 grid of tiles where s is a capture of the current tile arrangement. The state has the properties listed above.
Actions(s): legal moves from UP, DOWN, LEFT, RIGHT that Player can make to shift all tiles on the board in the respective direction.
IsEnd(s): state.transitions list is empty (no moves can be made), and there are 16 tiles already on the board. This means that there are no adjacent tiles of equal number (2, 4, 8, etc) which can be combined through a move, and that there are no blank spaces on the board so a new random tile cannot be added.
Eval(s): $(sum(valuesoftilesins) * numBlankTiles(s)^2)/numFilledTiles(s)^2$

The function $numFilledTiles$ gets the number of tiles that have a nonzero value, and $numBlankTiles$ gets the number of tiles that are blank for State s. To clarify, the 'values of tiles in s' refers to the exponential value of each tile (ex. 2048 instead of 11).

This game structure can be represented as a tree with the nodes representing the states and the edges representing actions. In our search problem for 2048, we are using a cost of 0 for all state transitions and are only evaluating the value of states at a limited depth to calculate the expected value for each move. The move with the highest expected value is picked.

5.3 Infrastructure Improvements

5.3.1 Cython

The language we used for our AI agent was Python, so there were a few performance hurdles that we had to get through due to the fact that Python is an interpreted language and less efficient when compared to compiled languages. With one of our goals being speed, we used Cython to optimize the performance of our Python code. Cython is a C-based Python add-on that essentially compiles Python-like code into a C module that can be imported in Python. The use of Cython allowed the agent to more quickly process state transitions and expectimax calculations in its decision-making.

5.3.2 Precomputation

In our implementation of the expectimax algorithm, we were able to shorten computation time by precomputing row transitions. A row of 4 tiles can be compressed into a 16 bit short, where each tile is represented by 4 bits that can represent any power of 2 from 2 to 32768. For each possible row in this representation, we compute the result of transitioning this row in a specific direction; if the transition changes the row, we store the compressed value of the new row in an array of size 65536 where the index is the compressed value of the old row.

5.3.3 Ignoring Rare States

While it is important to make our expectimax algorithm as accurate as possible, we leveraged the fact that a 4 tile only spawns 10% of the time to speed up the algorithm dramatically without sacrificing much accuracy. At depth 1, we branch out from all possible successor

states where either a 2 or a 4 tile has spawned. After the first branch at depth 1, only successors where a 2 tile has spawned are considered, cutting the branching factor in half after depth 1.

5.3.4 Selenium

The other large infrastructure improvement that we made for our project was the use of Selenium for the 2048 game demo. Selenium is useful because it allowed us to pull the values of the current 2048 board on the webpage directly from the Javascript embedded with the values - once this was done we could easily run our agent on the current board and select the best move. However, the use of Selenium introduces a small amount of lag between each move, causing a game to take up to 3 times longer to play than emulating a game in memory. Our final implementation gives the option to play a game in memory or in Selenium.

Now that the algorithm was fast enough in its decision-making process, we have a fully functioning, visually stunning demo that can run our AI agent on the 2048 website.

5.4 Search

5.4.1 Expectimax Search

We are using an expectimax search tree to calculate the best move for the current state of the game. Right now, expectimax takes the expected value of each move from the current state by calling the evaluation function of a branch of the search tree down to a variable depth between 3 and 6. The algorithm then lets the agent choose the move with the highest expected value.

The evaluation function we are currently using for the expectimax algorithm is as defined above in our model definition. It is called on each possible state (grid of tiles) that is possible for a given move from our current state. This is where additional heuristics can be implemented to place greater weight on potential moves that result in higher-scoring states (based on strategies adopted from playing 2048 a number of times and from what the Oracle seems to be doing).

cexpectimax.pyx in our codebase contains the Cython implementation of our expectimax algorithm, which is compiled to cexpectimax.pyd (Windows) or cexpectimax.so (Mac/Linux).

5.4.2 Search Depth

Initially, our agent was searching the game tree only two moves ahead of the current state (depth 2). After making our agent more efficient at choosing a move (Sec. 5.3) and not having much success getting to the 2048 tile, we tried working with higher depths. We realized that with search depth there exists a tradeoff between speed (smaller depth) and accuracy (larger depth). Ultimately, we implemented a variable-depth determination function to infer the search depth when running expectimax on a given state. The goal of this variable-depth function was to allow for higher leeway when there are exponentially many moves in the game tree and less effect on the outcome (i.e. many blank spaces on the board) while also bearing down in sticky situations where the agents move has a high impact on the outcome

(i.e. few blank tiles).

$$Depth(s)=\begin{cases} numBlankTiles(s)\geq 7 & 3 \\ 7>numBlankTiles(s)\geq 3 & 4 \\ 3>numBlankTiles(s)\geq 1 & 5 \\ numBlankTiles(s)==0 & 6 \end{cases}$$

Figure 3: Formula for Variable Depth

5.5 Agent Gameplay

To play the game, we initialize a random board with two tiles - 2s are chosen with probability p(2) = 0.9, and 4s are chosen with probability p(4) = 0.1. The placement is determined by a random variable from 0-15 corresponding to a grid space. Our algorithm then begins by finding the next best optimal move up to the specified depth (depth specified in command line, default 3, or variable-depth). In the figure below, we see an example of a depth 1 search on a board with two possible transition states. On a board with more possible transition states, we can clearly see how the number of states to consider can quickly increase with increasing depth.

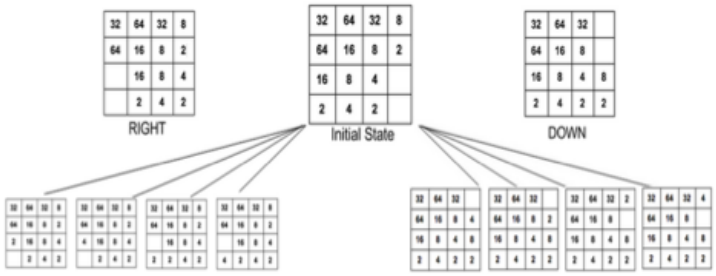

Figure 4: Depiction of successor states. When only two transitions are possible (Up, Down), there are already 8 successor states at depth 1. This number rises exponentially with greater depth.

The algorithm recursively checks states and transitions from the current root state. When the recursion reaches its depth limit, the evaluation function is used to compute the value of all leaf states, except for game over states that result in a value of zero. Every non-leaf state computes a weighted average of the values of successor states for each action and is assigned the value associated with its best action. The root state returns its best action.

6 Experiments and Results

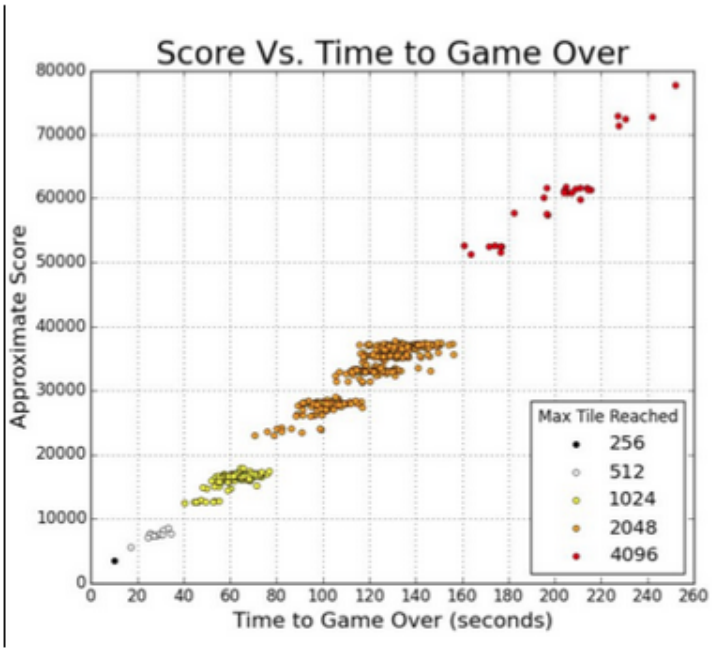

Figure 5: Scatter plot of results of 500 trials with variable depth

For robustness, we ran five-hundred trials on our agent using variable depth to test its efficiency and efficacy. In these trials, a success rate of 73.8% was achieved in an average time of 68.82 seconds. Again, success is determined by the presence of a 2048 tile at any point in the game, and can mean that tiles beyond 2048 are achieved. This was indeed true, and the percentage of games ending with a max 2048 tile were 68%, and 6% of trials (30 trials) saw a 4096 tile present on the board. Failed games reached a max value of 1024 in 23% of cases, and 512 in the remaining 3%. No game ended with a maximum tile value lower than 512. Equally, no game saw an 8192 tile or above.

For control, an equal number of trials were run under two constant depth settings. At a constant depth of three, 11% of trials reached 2048 successfully, and those that were successful reached the end state in an average 6.99 seconds. At a constant depth of four, 44% of trials were successful and it took an average 214 seconds to reach 2048 on successful trials. As clearly shown here, there is a very apparent tradeoff in speed and success rate, with trials that can quickly solve the puzzle having a low success rate and trials that take a long time to solve the problem seeing a higher win rate. In this same pattern, we hypothesized that trials run at higher depth - say depth 8 would have an even higher winning rate, though take much longer to complete. Accordingly, we can also surmount that the variable depth solution is indeed quite reasonable and surprisingly efficient given the depths it searches. A summary of the various trials is in the table below (Figure 9).

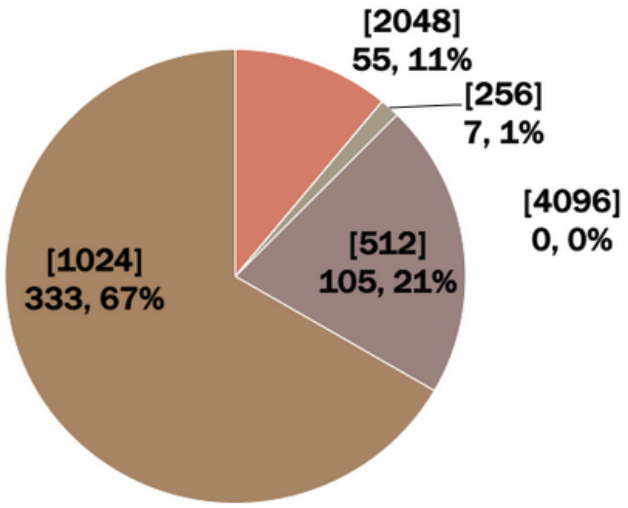

Figure 6: Max tile reached using depth 3 search, 500 trials

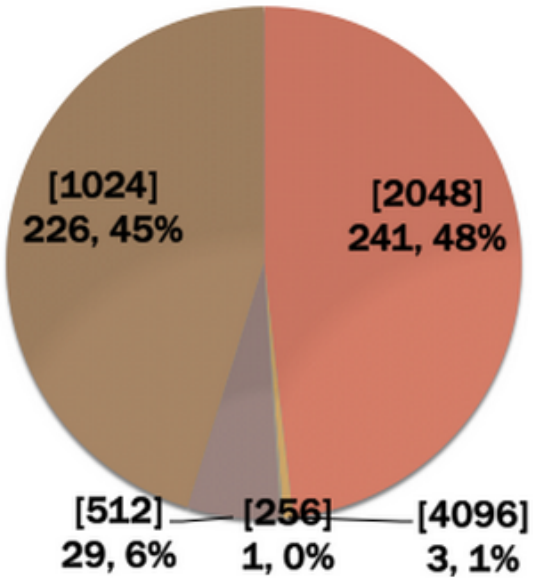

Figure 7: Max tile reached using depth 4 search, 500 trials

7 Analysis

With this project, our group was able to design and create a fully functional 2048 AI agent and a web player to play the game that achieved our two original goals of speed and accuracy. With variable depth, our agent was able to reach nearly a 75% success rate of getting the 2048 tile in a reasonable speed to completion of 69 seconds (see above).

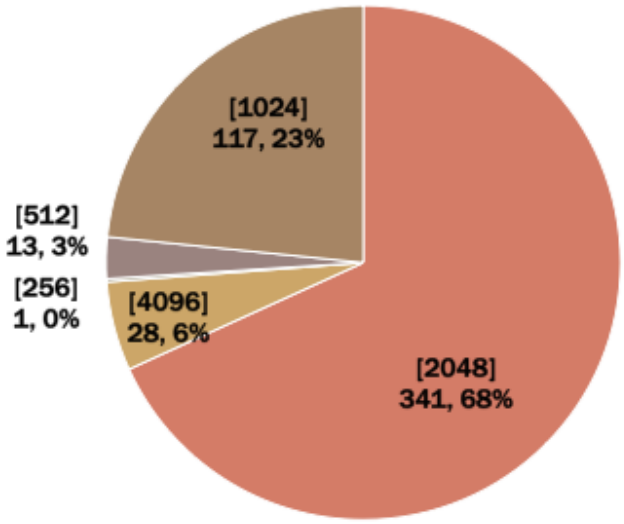

Figure 8: Max tile reached using variable depth (3-6) search, 500 trials

Depth	Success Rate	Avg. Time to 2048 (sec)
3	11.0%	6.99
4	44.0%	214.00
Variable (3-6)	73.8%	68.82

Figure 9: Summarized results of 500 trials for each of three depth settings

As we can see, the success rate highly correlates with higher depth, as well as time to completion as the gametree is much larger. Ultimately, we are satisfied with our results because we significantly cut down on completion time with variable depth implemented to just over a minute while increasing the success rate of a run by 167% compared to a constant depth of four.

8 Conclusion and Further Work

Our 2048 engine performs fairly well with the optimizations performed - not only is there a relatively high success rate of achieving the 2048 tile, but the solver is able to beat the game in just over a minute at a variable depth of three to six.

There are many improvements which can be implemented. First, we have improvements in regard to timing. Here, we solved initial problems by using cython instead of python, preprocessing state transitions, and ignoring some rare successor states. To make further improvements, effective multithreading can be examined to parallelize computation of successor states, a less memory intensive and thus more efficient board representation can be achieved using bitmasks to further reduce time required, and faster depth search can be achieved by storing the previously computed successor states.

The evaluation function is quite rudimentary. Upon analysis of playing strategies of peers and those published across the web, it may be beneficial to include small positional considerations. Here, the evaluation would reward specific strategies such as keeping the largest block in a corner, or never using a certain move(s) from the moveset. As a further implementational idea, it could be possible to derive a more ideal strategy or evaluation function using reinforcement learning techniques such as TD learning.

The code for our 2048 agent is available on our GitHub page for others interested in this application of AI.

https://github.com/Mathsvlog/Agent2048