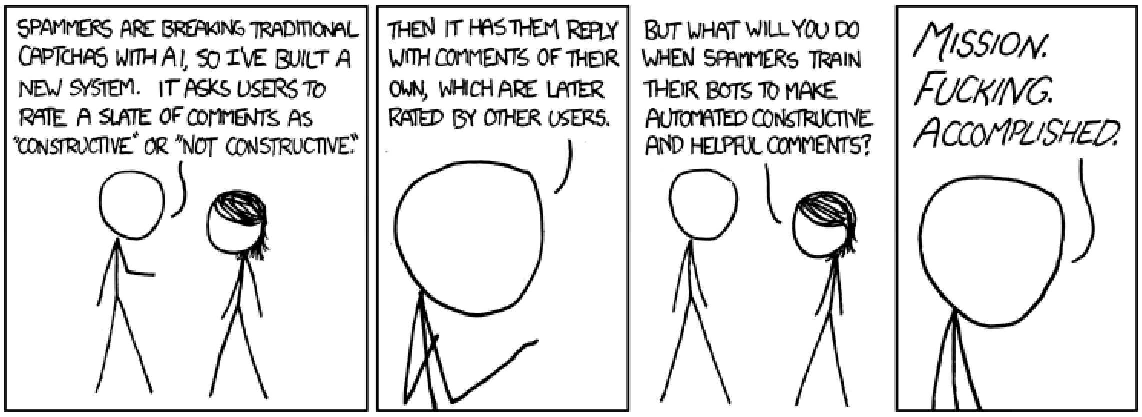
CITS3001 Algorithms and Artificial Intelligence

Threes Bot - Artificial Intelligence Project

Mitchell Pomery (21130887)
Kieran Hannigan (21151118)

May 30, 2014


SPAMMERS ARE BREAKING TRADITIONAL CAPTCHAS WITH A.I, SO I'VE BUILT A NEW SYSTEM. IT ASKS USERS TO RATE A SLATE OF COMMENTS AS "CONSTRUCTIVE" OR "NOT CONSTRUCTIVE."
THEN IT HAS THEM REPLY WITH COMMENTS OF THEIR OWN, WHICH ARE LATER RATED BY OTHER USERS.
BUT WHAT WILL YOU DO WHEN SPAMMERS TRAIN THEIR BOTS TO MAKE AUTOMATED CONSTRUCTIVE AND HELPFUL COMMENTS?
MISSION. FUCKING. ACCOMPLISHED.

Introduction

Threes[1] is a simple mobile game where you are given a four by four grid occupied by tiles. The aim is to merge tiles until the board is full and no more merges are possible, or until there are no tiles left to be added to the board. Each move (left, right, up or down) moves each tile one space on the grid if it is free, or merges it with the tile it moves into if conditions are met. You are only able to merge a one tile with a two tile, and any other tile can be merged with another of the same value. We were tasked with the creation of an artificial intelligence to play the game Threes, the goal being to get the highest score possible given a board and a list of upcoming tiles.

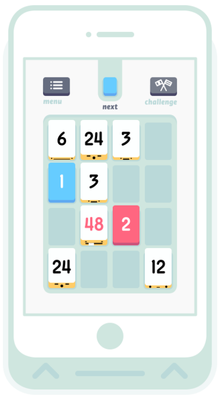


Figure 1: Threes Mobile Game Promotional Image

We were required to investigate algorithms and implementations to create an artificial intelligence, or bot, to play Threes with the intent of getting as high a score as possible. The specifications for the bot stated three main points that were considered during the design and implementation.

1. All upcoming tiles are known
2. Tile placement is deterministic
3. Inputs will be large

Knowing all upcoming tiles, and by being able to determine the location they will be placed means that at each node, there are only up to 4 different moves that can occur. Large input data sets remove the ability to search all possible combinations of moves, meaning that either a search depth or a maximum search time needs to be implemented.

Design Choices

Programming Language

Python was chosen as the initial implementation language due to the ability to rapidly prototype different algorithms and data structures. Its automatic handling of memory is convenient, and its syntax makes it easy to read and intuit - valuable attributes when co-developing an artificial intelligence. Subsequently, we began a rewrite of the agent to C. This allowed us to evaluate the strength and weaknesses of each language - an analysis which would have to be conducted if the code was to be hard optimized for production. Whilst python is conducive to comprehensible code, C outperforms any high level language when it comes to space and speed requirements. We reprogrammed the board mechanics, priority queue, and greedy algorithm in C for evaluation. The result was verbose, but faster by almost 30%.

AI Considerations

To make our artificial intelligence achieve the highest score, we need to score each board layout against several criteria. These criteria are essentially questions that can be asked to determine how good the board is, and therefore can be used to rank several boards. The boards score was the initial measurement used to rank different board combinations, due to the fact we are aiming for the maximum board score. The number of empty locations on the board is a useful measurement that we considered. Less tiles on the board means that we can add more tiles, and hence more points, without merging tiles to achieve a higher score.

The location of the high number tiles is important in a boards configuration. Having the high number tiles in the central four squares on a board means that they are stopping the lower tiles from residing next to each other. Smoothness of the board was considered after reading a Stack Overflow thread on 2048 artificial intelligence algorithms[3]. Smooth board layouts mean that high numbered tiles are clustered in one corner. These two criteria are similar as both relate to the layout of the board, so they are be converted into a single measure. The highest performing heuristic that we could device incorporated a factor of the score, the emptiness of the board, the gradient of lines (rows or columns that increase at every step are rewarded), and the edge-heaviness of the board (higher tiles on the outside ring give better results).

Algorithms

Making Moves

There are two ways the moves can be implemented in the bot, either each possible move separately, or by rotating the board and treating all moves as the one direction. In our python implementation we employ the latter of the two algorithms, as it allows us to reduce the amount of code written. However rotating a sixteen by sixteen board can take several operations, so to avoid this, instead of rotating the board, we rotate the co-ordinates. In our C experiment, we move the board using loops and array operations. We also experimented with large integer bitboards. A 64 bit bitboard allows for each tile to be encoded in a 4 bit nybble. Unfortunately this means a maximum tile size of 12288 which we predicted would be readily achievable by a fully optimized algorithm. Bitboards are optimal because they provide the CPU with a memory unit that it can make extremely efficient shift and addition operations on (holding it in a single register). Unfortunately this advantage is largely lost when multiple bitboards or 128 bit bitboards are used – as would be required for the tile size – especially on machines with 32 bit registers. We opted to use more portable code here.

Naive

Initially a naive algorithm was implemented to play Threes one move at a time. It would look at the four possible moves for the current board configuration (left, right, up and down), and execute the one that gives it the highest score. The naive algorithm is useful as an initial heuristic later in the A* implementation. In the C port, we provided a simple “count the zeros” heuristic to the naive algorithm. We found that by minimizing the tiles on the board instead of chasing high tiles, the algorithm produces a better final result.

Minimax

Both minimax and expectiminimax implementations were investigated due to the similarities between Threes and 2048, and the already existing 2048 AI by Matt Overlan[2]. A major difference between 2048 and Threes is that in 2048 the next tile is placed randomly, while Threes as specified by the project outline, the location of the next tile will be placed in the lexicographical lowest location. This difference means that the Threes bot does not end up playing against anything – it becomes a deterministic problem as opposed to a stochastic one – and so minimax is not a relevant algorithm.

A Star

A*, by definition, finds the least-cost path from the initial state to a goal state providing that its algorithms are correct and admissible. It is easy to devise monotonic heuristics for Threes, making A* particularly promising. The goal state in a game of Threes is the highest scoring full board, meaning that the algorithm needs to avoid filling the board for as long as possible. Multiple iterations of A* algorithms were implimented to glean a measure of the effectiveness of each heuristic.

A naive A* algorithm was implimented as a base algorithm. It showed immediate score increases over the straight naive algorithms. The only heuristic used was the score of the board and the only advantage of the naive algorithms was the fact that it looked forward multiple moves.

The final A* implimentation went through multiple iterations before its current state. Different iterations showed better performance on different boards as we wored towards maximising the score for every board. The heuristics used in this algorithm were the score, the weight of the edges of the board, and how lined up the rows were. By adjusting these heuristics, it is possible to tune the performance, and potentially increase the final score that the bot achieves.

Test Data

Test data was generated using a modified version of `createTestData.py` by Lyndon While[4]. It was modified to create multiple ouput files with 1000 upcoming tiles in one run of the program. `test.py` would then be used to load the input files and run them through the written algorithms, outputting the data in CSV form for analysis.

Testing was done over four algorithms, the naive python implimentation, the naive C implimentation, a naive A* implimentation and our final A* implimentation. We found that the naive algorithms would quickly fill up all the available space on the board whereas the naive A* would avoid filling the board for as long as possible. The final A* implimentation we created as depth limited and had similar final scores as the naive A* implimentation but took half the time.

Algorithm	**Average Time(ms)**	**Average Score**
Naive - C	13	309
Naive - Python	18	305
Naive A*	31951	7665
Final A*	17670	6731

Bibliography

[1] THREES - A tiny puzzle that grows on you. 2014. [ONLINE] Available at: http://asherv.com/threes/. [Accessed 28 May 2014].

[2] 2048. 2014. [ONLINE] Available at: http://ov3y.github.io/2048-AI/. [Accessed 28 May 2014].

[3] logic - What is the optimal algorithm for the game, 2048? - Stack Overflow. 2014. [ONLINE] Available at: http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048. [Accessed 29 May 2014].

[4] createThreesFiles.py. 2014. [ONLINE] Available at: http://undergraduate.csse.uwa.edu.au/units/CITS3001/project/createThreesFiles.py. [Accessed 30 May 2014].