Peter Brede
peterbrede@ucsb.edu
Perm #: 7052640
CS165A

MP2 Report

Architecture

The basic architecture of my code was simple: I planned to layer my search functionality by first searching all possible boards and their probabilities, then finding the move that maximizes their score function and note the resulting boards. After I have this new list of new boards, I could recursively iterate the above layering to deepen my search. After much testing with this plan, I eventually reduced my depth all the way to just one of these layers, and instead replaced the complex logic and overall bulk of my algorithm to my actual scoring function. In my code, this is called getScore(). This is basically my heuristic scoring function, which grades a board based off of its highest tile, whether it is located in a corner, whether the rest of the board decreases smoothly the further away from the highest, etc. I added more and more utility to this function the more I tested and tweaked with the variables and their respective value to the score. Eventually, my NextMove() function adds up the products of these scores of possible game boards and their probabilities for each possible action that can happen: swipeUp, swipeDown, swipeLeft, swipeRight, or doNothing. Finally the function returns an integer based on the maximum value for each of these actions.

Search

My search algorithm is an Expectimax algorithm. The probability of a game board is multiplied by its score to produce another expected score of this board. This expected score is added to all other expected scores of all other possible boards (of depth 1, tested on depth 2 and 3, but those eventually took too long) and all together the resulting expected score represents the expected value of a certain move. The expected value that is highest will result in NextMove() returning the move to be executed. As I explained above, the scoring function contains most of the complexity relating to monotonicity and corner strategies, which I found by playing the game myself.

Challenges

At first, I had planned to run my algorithm to look into tons of possible boards at a depth of 2,3, or even 4 iterations in the future. This took a long time, and when I was able to make it run fast enough to fit into the 40 minute testing period, I think I messed up the organization of something, because it was resulting in nonoptimal play. I spent countless hours trying to get this to work, then eventually worked on my code starting

with a depth of just one iteration. Building upon my code for this method, I saw steadily increasing test results and finally found a somewhat optimal function.

Weaknesses

As a part of my challenges portion, I expressed some of my weaknesses. I think that my main weakness is that my current algorithm does not look very far ahead and does not plan moves in the future. Because of this, it doesn't seem very adaptive or consistent. I struggled in my testing at the end, because I would score 2048 for most test cases (I think up to 9/10), but only scored a 128 on the last. This probably showcases my code's inability to get out of a tough board.