Joey Long
Lazar Zamurovic

A Genetic 2048 Player- Final Final Project Proposal

Problem definition:

The problem is to create a 2048 player with the highest sum of tiles on the board. The input of the problem is a board with 2 randomly generated tiles that either have the value 2 or 4. The input strategies are randomly generated. The output used for evaluation is the board after some number of moves. The final output of the algorithm is the strategy to play the game. We evaluate solutions by the sum of all the tiles on the board after the board is filled and there are no possible moves combined with the largest ti le on the board as well. We feel that this project is a reasonable scope. We use a prebuilt gameplay simulation and hand built the genome of the genetic program and tested multiple different representations of a strategy. Our goals include building a genome that can properly play the game as well as surpassing the score of our base case.

Infrastructure:

We are using a simulation of the game from an outside source through an Apache License (open source software) found here: https://github.com/bulenkov/2048. We adapted this for our purposes by removing the visual components to increase the speed of our tests and adding several methods that carry out the running of generations. There is a 4x4 grid with 2 initial tiles (either with a value of 2 or 4) and a randomly generated new tile (either valued at 2 or 4) is placed in an open position after each move. A move is one of four directions: up, down, left, or right and tiles will move until they encounter a wall or another tile with a different value. Equal valued tiles are combined into a new tile when pushed against a wall of the grid or a tile of a different value.

Aside from this simulation we have written several classes for representing and manipulating our genome: Tree2048, Breeder2048, & TreeGenerator. Tree2048 contains all the information necessary for representing our tree. This consists of the value of the node, the left & right subtrees, the score of the tree, the max tile, and the depth. This class also contains functions used to evaluate the the output (move) of the tree, as well as functions for mutation, crossover, & visual representation of trees. We could not get crossover to work due to problems with Java passing object references in methods in a way that we did not expect, however our algorithm is there and could be fixed given more time. TreeGenerator is a class that is solely responsible for creating random trees with a specified minimum and maximum depth. It contains two instance variables of string arrays that contain the functions and terminals in our representation. Breeder2048 is responsible for the evolutionary portion of our project and as such its functions deal with populations (arrays of trees). Breeder2048 contains all of the evolutionary parameters such as mutation probability, crossover probability, tournament size, population size, minimum and maximum depth. It also contains our two implementations of tournament selection; one which selects based on score and another that selects based on max tile. These classes are used in the class which simulates the game to run the each generation of our algorithm.

Approach:

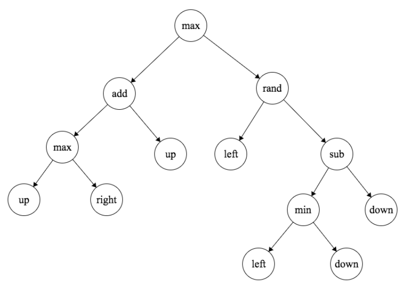
We are using a genetic program as the genome of each individual. The output of the tree determines each move while playing the game. We represent each move with an integer: 0 for up, 1 for right, 2 for down, and 3 for left. There are 4 possible sensing terminals: up, down, right, and left. Each terminal returns the integer representation of its move if there are tiles that can be combined if moved in that direction (e.g. if we can combine tiles upward, the up terminal returns 0). If there are no tiles that can be combined in a certain direction, the terminal returns a random integer between 0 and 3. We have the following functions with arity 2: add (binary, mod 3), subtract (binary, mod 3), random (binary, choose a random child), max (binary, larger of two children), min (binary, smaller of two children). If a tree outputs an illegal move, it will be reevaluated until a valid move is returned for 200 moves. If the board state does not change for these 200 moves the game is ended.

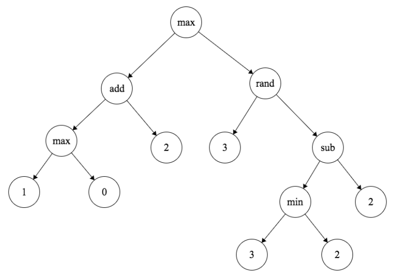
We evaluate each genome after the game has ended i.e. there are no more legal moves. We evaluate half of the population using score as the fitness and the other half of the population using max tile as the fitness. This method is inspired by Schaffer's VEGA technique for multiobjective genetic algorithms (Schaffer 1985). These subpopulations are then merged together to create the selected population for each generation. We are using tournament selection as our selection method.

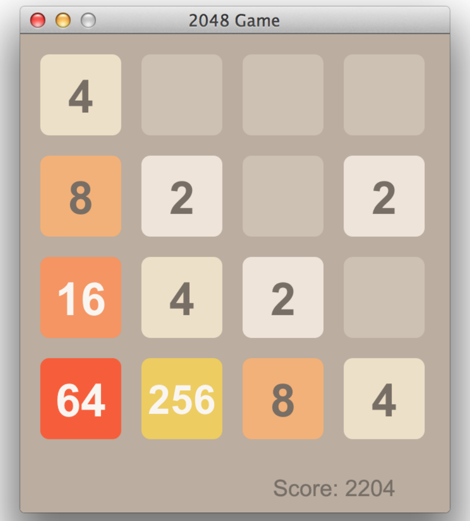
Mutation occurs after selection. In order to make sure that we do not encounter bloating, we set a max depth for all genomes. We also set a min depth for the trees in order to ensure there are not many individuals within a population containing a single terminal. We will include random mutation to happen in a small percentage of individuals. Mutation can randomly switch terminals.

An example genotype is the first of the two trees below. Our phenotype is the move that the tree outputs given a certain board state. The second tree shows what each terminal returns given the board state on the right. The given board only combine tiles if the direction of the move is left. This means that terminals up,down, and right will return a random integer 0-3 and left will return 3. The functions in the tree take the terminals as input and following the functions with our given terminals, the next move would be a 3 which would result in a left move. A mutation looks like exchanging the random function to a max function (only functions can be mutated in our algorithm).

The fitness of an individual is based on either the score or the max tile. The figure below is not the final fitness of the individual because the game has not ended (there are valid moves on the board). Once the game is finished the fitness of the individual is calculated.







Measuring Success:

It is very easy for us to measure success in this problem. The only two metrics that we care about are the sum of the tiles on the board and the max tile when the game is over. We find a solution that maximizes these by delaying the end of the game as many moves as possible by configuring the board so that tiles can be consolidated.

Our base case is randomly generated moves (randomly generating an integer 0-3). We test the base case by running through 1000 games and taking the average score of all the games. This is the lower bound of expected success for our algorithm. The upper bound of our experiment is comparing our best scores to that of a stack overflow artificial intelligence algorithm which performs better than anything we can do by hand or have ever seen. The algorithm was created a user named nneonneo and the thread can be found here: http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048.

Goal Setting:

A minimal version of our project is to implement a GP to evolve solutions using the representation detailed above. We will experiment to find optimal crossover, mutation, and selection parameters and test with at least 25 runs. We wish to be able to beat the fitness of our base case.

A good version of our project would include experimenting with different nodes and testing the results of the changes. It would beat the base case more significantly than the minimal version and get as close as we can to our upper bound (although this is a very hard task). This version would also include adding our second proposed modification to the evaluation method by creating a mathematical function to incorporate the two different objectives for evaluating fitness.

A great version of this project would include a visual representation of the strategy being used. Basically show the moves happening while running. Another aspect that would make this project great would be to include ADF's in the genome of the individual. We could have 4 ADF's, one for each direction, and create a value from each of these trees and then choose the move based on the evaluation of these trees. Each tree would be able to be mutated independently.

Baseline Testing:

Our lower bound for this project was playing the game using a randomly generated move each time. There was no use of a genetic program here, just a random number generator to choose the next move during gameplay. Over 1,000,000 games played using this strategy, we averaged a score of 1095 and a max tile of 107.

The upper bound of our experiment will be comparing our best scores to that of a stack overflow artificial intelligence algorithm which performs better than anything we can do by hand or have ever seen. The algorithm was created a user named nneonneo and the thread can be found here:
http://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048
This individual's algorithm achieved a score of 794076.

Results and Analysis:

Test 1
Mutation Probability: 100 %
Population Size: 65
Tournament Size: 5
Tree depth: 3-7

We found that our best results occurred with these settings. Over 10 different initial populations, the genetic programs averaged a score of 3302 and a max tile of 319. The mutation probability is perhaps the most surprising result, but in the context of our program it makes sense. This is because our mutation method can only mutate a maximum of one function in each genome and so even at 100%, the structure of the tree does not change significantly from generation to generation. A population size of 65 allows a nice degree of variability in our initial population and the generation of the initial population itself can find relatively good solutions without any mutation or crossover needed. Tournament size of five will most likely keep the fittest individuals from

Test 2
Mutation Probability: 35 %
Population Size: 65
Tournament Size: 5
Tree depth: 3-7

While testing for the best parameters, we found that changing the mutation rate does not affect the average score of 10 different initial populations greatly. If we change the mutation rate to 35% the average score was 2965 and a max tile of 247. This is only slightly lower compared to the average score of our best parameters so the very large decrease in mutation did not contribute significantly to a lower fitness.

Test 3
Mutation Probability: 0 %
Population Size: 65
Tournament Size: 5
Tree depth: 3-7

The same can be seen if we change the mutation rate to 0 given these parameters. The average score here was 3293 and the max tile was 287. This is also slightly lower than that of our best results but surprisingly higher than the test with 35% mutation rate. This can be explained by noting that randomly generating strong solutions in the initial populations can be more effective than generating weaker solutions and searching the solution space. In other words, this test got a lucky set of 10 initial populations and this caused it to seem to perform better than a program with mutation enabled. Our mutation method, the altering of one terminal randomly, clearly does not affect score much, but tree depth had a very noticeable effect on the quality of solutions that we found.

Test 4
Mutation Probability: 100 %
Population Size: 65
Tournament Size: 5
Tree depth: 1-4

The results here are much lower than the other tests with the smaller trees obtaining an average score of 2476 and max tile of 220. This is most likely due to the smaller trees not being able to handle as many board states effectively due to their limited number of functions and terminals.

We did not expect our genetic program to perform as well as our upper bound, which was clearly shown. We did expect our genomes to perform slightly better than they did. We attribute this result to two factors. The first being our problems with our crossover function. Successfully implementing our crossover method could have allowed for our program to search the solution space much more effectively, but we simply ran out of time. The other factor that we believe affected the scores was the return of a random move from a sensing terminal given the sensing terminal could not move in its own direction. This means that if a specific board state could not be moved to the right, the right sensing terminal could not have returned the move right. Instead, our implementation allows the terminal to choose a random move from the four possibilities. We think that this implementation causes values that correspond to non-optimal moves to move through the tree.

Recursive Reflection:

We are pleased with the progress of our project. Having found a gameplay implementation online helped greatly so that we could focus more on the creating the genetic programming aspects. We were overall able to build a population of individuals that performs significantly better than either of our baseline tests. We were also able to combine our two evaluation functions into one fitness function to better evaluate the performance of the individuals in the population. The binary tree representation of the genetic program worked very well for us. Our tree creation method did cause us some problems as we noticed that we were creating many random trees that were either depth 1 or the max depth. The algorithm used for our tree generation function could only end one subtree at a time causing there to be a high chance that trees would grow to the max depth. We added a minimum depth in order to ensure larger trees. We ran into some trouble while trying to build a crossover method. Finding a random node within a tree and then swapping it with another random node in another tree was not working. We believe it stems from the way Java clones objects. We could not get crossover to properly work. Implementing gameplay presented its own troubles as sometimes a genome can output a move given a board layout which is not legal. If the genome does not contain any random nodes, this move will continually be outputted leaving the game in an infinite loop. We had to decide on how to deal with this situation and whether it was a good idea to allow the genome to play a legal move or to punish the genome for its inability to deal with the situation. After building the genome, we proceeded to combine the genome and the gameplay. We then built mutation, selection, and population-creation functions.