Nico Judge

CS 440

October 16

A Walk Through Genetic Algorithms: One X at a Time

When looking at various algorithms for ultimate tic-tac-toe there are many options one could go with, however, one that stands apart from the rest is genetic algorithms. Modeled similar to Darwin's evolutionary theory, this algorithm evolves each new generation until a solution is reached. Although it is a relatively simple algorithm to understand, the power of this approach is in mimicking biological evolution so that we can evolve the AI to achieve the optimal solution.

One of the reasons this algorithm is so strong is that it can improve itself by learning rather than analyzing the statistical probability of a move's success. To simplify the process, each generation of the AI has its own encoding of what moves it should attempt to use. Each generation of the AI will have a given number of variations – or species - of that generations encoding which will each play the game and upon finishing the game will be given a fitness score. That score is the evaluation of success of the encoding using a given numeric score.

For the purpose of ultimate tic tac toe, there will be two main parts to the encoding. Each board (both macro and mini boards) receives an evaluation score for how likely a win, loss, or tie is. This is different from the fitness score which judges the overall performance of the strategy. The way this encoding would work is each box on the board will receive one of 3 things. If there is already a move for the current position being looked at the encoding for that particular box would be either an 'X' or 'O'. If the box is open, it will receive an evaluation score based off the chart below. These scores are based off looking at 3 positions in a row for horizontal, vertical, and diagonal.

<u>Encoding Table for X</u>

Condition	Encoding
There are no Xs or Os	1
There is 1 X and no Os	2
There are 2 Xs and no Os	3
There are no Xs and 2 Os	4
There are no Xs and 1 O	5
There is 1 X and 1 O	6

<u>Encoding Table for O</u>

Condition	Encoding
There are no Xs or Os	7
There is 1 0 and no Xs	8
There are 2 Os and no Xs	9
There are no Os and 2 Xs	A
There is no Os and 1 X	B
There is 1 X and 1 O	C

Because the encoding will be all one string, we cannot have a double character so we switch to upper case letters similar to a hex value.

Since each specie of the generation has a slightly different strategy each playthrough will get a different fitness score, so that once we have tested all the variations we can sort the encodings by fitness score and then breed the top percentage of the encodings in order to create the new generation. Although we cannot breed them in the more traditional sense, we are able to create the new generations by randomly changing bits of the encoding and crossing bits of encoding from the different top encodings.

Although it would work by randomly combining the top encodings, this is not the most efficient way to create new offspring. Instead of random combinations we can use any of the many types of ranking algorithms. A good example of this is using a Roulette wheel selection.

Since each encoding will have a fitness value associated with it, we would naturally want to choose the species with a higher fitness level. If we visualize a pie chart representing the total fitness score, we can assign each specie its own slice of the chart. Once we have that set up, we can use the pie chart similar to a roulette wheel and "spin the wheel" and choose whichever specie it lands on to use as the other parent. Since the species with higher fitness levels are a larger percentage of the wheel, they have a greater chance at getting chosen to be a parent for the crossbreeding.

After we have chosen the two parents to breed to create a new specie, we can use a crossover rate to choose how variations are created. This rate is the chance that the two parents will switch parts of their encodings, if we want a larger variability in our offspring – e.g. we want a larger amount of differences within the future generation - the rate will be much higher to promote more crossings, and likewise if we want a smaller variability, we would use a smaller rate. After using the crossover rate to determine if we want to switch bits of the encoding around, we choose a point in the gene where we want to switch so that if the algorithm chooses to initiate the crossover, the encoding will swap after the given point. To visualize this, imagine there are two simple binary encodings for a strategy for a given problem.

1) 001101101001
2) 001100010101

If we set the crossover point to be the halfway point of each of the two encodings and we choose to do the crossover, we switch everything after that chosen halfway point. So that our result would look like this.

Separate the bits	->	Swap the bits
001101 101001		001101 010101

001100 010101 001100 101001

Finally, after we have finished this step, we can move on to the last step in finalizing our offspring for the next generation. We can use a mutation rate to create a chance that we will mutate part of the encoding. For example, if we use one of the new binary encodings from the previous step we can go through each bit and randomly decide if we want to mutate it based on our rate.

Previous encoding	->	**mutating bits**
001100101001		001100101001

By mutating random bits in our encoding, we can ensure that we continue having a large enough genetic variability to create the most efficient solution to our problem.

With those steps finished, you have created a new generation and you can now run the new population through playing games and repeat the process until you feel that your AI has achieved the most optimal solution to the game. The benefit of this method of AI is that the future generation will always be better than the previous generation.

In trying to modify this algorithm to work for a game of tic-tac-toe we have to first think of the possible overhead that will occur during the computation of each possible move the AI should make. For each miniature board there are 9! (362,880) (schaefer) unique number of possible tic-tac-toe games for a fully filled in grid (3x3), giving us an upper bound on the how many different combinations there could be.

The total number of possible mid-game assignments is 3^9 (19,683) (schaefer), or to put it simply, each of the 9 squares has 3 possible configurations; X, O, or Empty. However, we can narrow the number of assignments down because of the symmetrical properties of the board.

Using this property, we can eliminate many of the states we need to search due to the fact that we can combine states that mirror each other. Using that and eliminating illegal moves we can reduce the total number of possible games to 23,129 (schaefer), however since many of these can be eliminated due to it being either a mirrored game, or the board has been flipped (all Xs are now Os or the reverse is true). After those eliminations the total number of unique situations is 827 (schaefer), a number found empirically by exhaustively constructing possible solutions (3^9) and reducing the valid ones to their base cases (they are not a mirrored board and the board configuration is legal). These 827 unique states are only for the inside boards and do not account for the outside board configuration.

The outside board would be very simple having the first 9 characters of the encoding would be the status of the board – 'X' if there is an x, 'O' for an O, and '_' if there is no current winner of that mini board. The inside board would be similar to the outside board where the encoding for the board is 9 characters long with 'X', 'O', and '_' to save the board configuration. Separate from these two board encodings is the rule set and strategy that we want the AI to learn. The strategy going to be used is similar to a hash table lookup. Once we get the configuration of the board, we would look up the corresponding move for that turn taking into account both the inside and outside board. So in practice we would have three separate global variables, the state of the outside board, the state of the inside board, and the lookup table for each possible tic tac toe configuration.

In order to judge the success and create a fitness score for each of the different species within the current generation. Using a point system, we can assign a fitness based on the following.

AI goes first	**AI goes second**

Win: +3	Win: +5
Tie: +1	Tie: +2
Loss: -1	Loss: 0

After all of the specie within a generation has been tested and a fitness score has been designated to it, we can crossbreed the top 50%, randomly create mutations, and then continue rerunning the algorithm until it is believed that the AI has reached an optimal solution and further runs of the evolutionary process will have either no effect or will not be worth the overhead of creating and running the next generation.

The difficult part about implementing this algorithm is that fine tuning the population size and the rates of mutation and crossbreeding can drastically alter the outcomes and evolutionary success of the AI. If you don't have enough variability in your generations then you will get a generation of species who do almost the exact same things. On the other hand, if you have too much genetic variability, it may be difficult for the AI to make significant headway in evolving if many of the strategies are wildly different.

Bibliography

Schaefer, Steve. 2002. *How many games of Tic-Tac-Toe are there?* Available from http://www.mathrec.org/old/2002jan/solutions.html.

Hochmuth, Gregor. “Http://Www.genetic-Programming.org/sp2003/Hochmuth.pdf.” www.genetic-programming.org/sp2003/Hochmuth.pdf.