Matt Litzsinger

Emma Dedmond

Complex Systems - D

The Interactions between Genetic Algorithms

-----Introduction----

The world we live in has many problems and nearly unlimited possible technological advances. To solve these problems, it’s becoming increasingly clear that interconnectedness among different disciplines is a necessary part of this. Biomimicry, for example, is the coalescence of studying biological phenomena with engineering. In many cases, nature has found the most elegant solutions to complex problems. This “problem solving force” is called evolution; taking imperfect organisms, and adapting them to their environment. Because this method is so efficient, scientists are beginning to look to evolution for answers to other problems, creating mathematical models which follow the principles of evolution to solve complex problems. These models can be used to solve anything from complex functions to strategies for solving mazes. By combining the concepts of natural selection and computing, genetic algorithms emerge as a useful tool for optimization of many various things. We hope to harness this power and understand interactions between two developing genetic algorithms by creating and analyzing a genetic algorithm to determine the optimal strategy for playing the game Dominion.

-----Dominion-----

Dominion is turn-based deck-building card game. There are three types of cards in Dominion: Victory Cards, Treasure Cards, and Kingdom Cards. The goal of the game is to get as many Victory Points as possible, which are gained through buying Victory Cards. If you have more Victory Points than any of your competitors, you are declared the victor. The difficult part about strategy with this game is that Victory Cards are essentially useless cards in the deck during the game. So the game is a balancing act between having too few Victory Cards (and the game ending too early) and too many. Treasure cards, however, provide a certain amount of money (depending on the card) for one turn, and action cards chained in various ways provide benefits to the player through cards, extra money, etc.

Each turn consists of two main parts: the action phase and the buy phase. First, the player draws 5 cards. If they cannot, they shuffle their discard, and then that becomes their new deck. During the action phase, the player can choose to play one Kingdom card, which can increase the amount of money, the number of buys, allow him/her to draw cards, allow him/her to play more Kingdom Cards, or do something more complex. Each Kingdom card is different and affects gameplay differently, but their usefulness is approximately proportional to their cost. Because of this, in order to limit the variables in our game, we will automatically play the highest costing action card when available.

During the buy phase, the player sums the coin value of treasure in his hand, then can choose to buy a card from the 16 available in the supply (3 Victory Cards , 3 Treasure Cards , 10 Kingdom Cards) with a cost less than or equal to the sum of money that player has in his/her hand. When this card is bought, it is transported to the player’s discard pile, and coalesces with the deck once reshuffled.

-----Creating the Game Engine-----

Creating the game engine was a long and tedious process, but fairly intuitive the whole way through. Essentially, the fundamental principle we used was that a turtle equals a player, and that they have both a chromosome, deck, discard, etc. such that a chromosome can be related to a certain fitness. The deck, discard, hand, playarea, and other main elements of the game engine were created with lists. The engine is a big while loop with the game-ending conditions constraining the loop. First, a user-interface was created to make sure that all the action cards worked properly, cards were bought properly, and to ensure all of the bugs were ironed out before we took out the user-interface, making it harder to detect those bugs. Then we stripped out all the user-interface elements, and related the chromosome to each buy decision. This was done by looking at each of the conditions (money, provinces left, supply piles left), and then altering the called index of the chromosome to match the according datum.

Automating the action cards was tough, but was done through dividing actions into 2 groups – those that have +1 Action (chaining action cards) and those that don’t (terminal action cards). On any turn, you can play as many chaining action cards as you want because as you use an action you gain another, so the computer plays all chaining action cards. For the terminal action cards, it may play one, so it plays the one with the highest cost, and if there are two tied, it plays one at random. This method is used to play games of dominion, and that reports that number of points received and relates it to a fitness, which is used in the genetic algorithm, described next.

-----Creating the Genetic Algorithm-----

Code was created with Netlogo 2.0. We started creating our genetic algorithm by creating 10 turtles of the breed “chromosome”. The breed chromosome has two variables; genome1 and fitness. Genome1 is the turtle’s genome, while fitness is determined by playing the game. For each of the turtles, we then set genome1 as a random list with a length of 144, each having a value from 0 – 16, with 6 values on the end from 0 - 50. The reason for this will be explained when we explain the functions of the game. We then had each turtle play the game Dominion.

Our genome is based on the amount of money (values 0 – 15, values above 15 equated to same decision as 15), the amount of Victory Cards left (values 0 – 8), and how many supply piles are empty (with values 0 – 2). This creates 144 different situations. The last 6 sections of the genome are set to control what happens when the genome has multiple buys due to action cards (how to split up the money among buys). The number determines the percentage of available money that should be put into each buy then converted to integer coin value.

The number of victory points was set as the variable fitness1. We had the chromosomes run through the game again, matching the top 2 together, and set this number of victory points as fitness2. After doing this three times, we added up each of these to determine the overall fitness. Once the fitness of each chromosome had been determined, we created a list of the chromosomes, in order of decreasing fitness. Initially we set the genome of the first two turtles with highest fitness as gen1 and gen2, which were global variables for the next generation, but realized later that with the top 4 advancing, this better elitism allowed for more growth in terms of fitness. This allowed us to keep our best genomes and prevent losing any progress made from this point. The rest of the new generation was determined by the fitness of the first generation. We set the global variable parent11 to be the genome of the chromosome with the most fitness 80% of the time. 80% of this 20%, or 16% of the time, parent 11 was the genome of the chromosome with the second highest fitness. This was repeated for each fitness and parent.

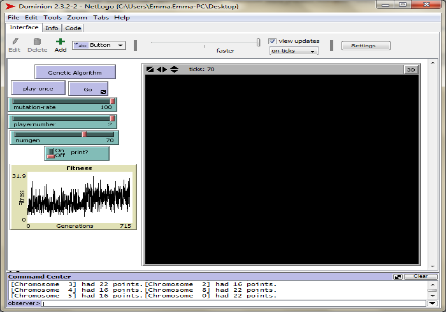
To crossover parent11 with parent12, the first value in the list was deleted a random number of times using the butfirst function (with the maximum being the length of the genome). For parent21, we repeated the butlast function as many times as the remaining length of parent11. These two lists were combined into one using the sentence function, and set as gen3. This was repeated for parent21 and 22, and so on.

The next thing we did was create genetic variation through mutation; otherwise the only possibilities would be those that were in the original 10 chromosomes. We created a function which mutated each genome with a probability determined by our variable mutation rate. We used 100% mutation rate to speed the entire process. We then picked a random number from 0 to 150. If it was under 144, the function would replace a random place in the genome between 0 and 144 with a number 0-16. If the random number was over 144, the function would replace a random place in the genome between 144 and 150 with a random number from 0-50. This kept the structure of the genome the same for each generation, which otherwise would make the structure arbitrary. The probability of a mutation occurring at any point in the genome is equal.

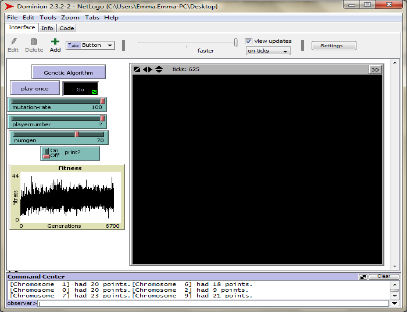
To create the next generation, chromosome 0 sets its genome as the variable gen1, and chromosome 1 sets its genome as the variable gen2. These turtles then determine their fitness by playing the game and the process repeats.

-----Conclusions-----

When running the genetic algorithm, there was rapid increase in the first couple generations, and then the line smoothed out.

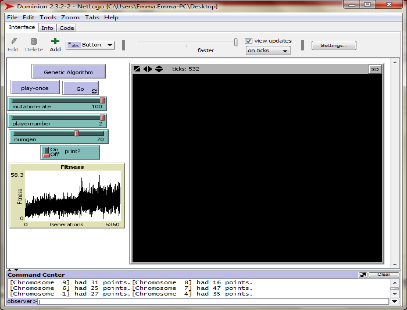


The nature of the game makes it difficult to evolve because each game is played against a different player. The fitness shown is the amount of victory points used for one round, and we used three rounds to determine the fitness. One strategy might be very successful against another specific strategy, but we were interested in a strategy which would be generally useful. After many generations, significant progress is yet to be made.



This is probably because a high fitness could actually be due to chance, and so progress is sometimes false, while actual progress is sometimes lost, even though we included elitism. Increasing the number of chromosomes kept each time could help this problem, but it would also decrease the variation, making the program run even slower than it does already.

Another run was more successful, showing two increases in the genome.



This suggests that there are at least two significant strategies that the algorithm must learn in order to improve its skill level in the game.

While the maximum number of victory points possible is 86, our algorithm appears to be averaging about 30. This is because it is practically impossible to achieve that high while competing against another player.