Spring 3-1-2018

Selection Methods of Genetic Algorithms

Ryan Champlin
rjchamplin@olivet.edu

Recommended Citation

Selection Methods of Genetic Algorithms

Ryan Champlin

Olivet Nazarene University

Contents

Introduction

From the beginning of time people have been interested in intelligence. Where does it come from? How can humans become more intelligent? Can we as a species create intelligence? There have been hundreds of thousands of attempts in creating an artificially intelligent machine, and one that is deeply entwined in the study of computer science. In the mid-1900s computer scientists envisioned a new system of artificial intelligence which they named Genetic Algorithms. The leader of these being John Holland from the University of Michigan (Goldberg 1). Taking ideas from the world around them these visionaries created a self-teaching algorithm that is born, reproduces, and dies thousands of times over in the attempt to solve a problem. In 2018 genetic algorithms are not the AI powerhouse that some thought that they could be, however they have changed the face of artificial intelligence, as well as progressed and become useful in their own niche. They are often used where there are adapting parameters, if the search space is very broad, the task doesn't require the best answer; just a good one quickly, or if the parameters are not well known (Mitchell 156 and Langdon, McPhee, Poli 111-113).

Genetic algorithms are essentially search algorithms based on the mechanics of evolution and natural genetics. Jason Brownlee says "The strategy for the Genetic Algorithm is to repeatedly employ surrogates for the recombination and mutation genetic mechanisms on the population of candidate solutions, where the cost function (also known as objective or fitness function) applied to a decoded representation of a candidate governs the probabilistic contributions a given candidate solution can make to the subsequent generation of candidate solutions." (Clever Algorithms, Genetic Algorithm). In essence a genetic algorithm gives birth to answers and breeds them until the best possible answer is found. They use randomized information exchange to search a problem set by exploiting historical information to search for new points. A genetic algorithm is comprised of five distinct parts; initialization, fitness assignment, selection, crossover, and mutation. In my research I explored the differences between four different types of selection in genetic algorithms. In this research I compared the runtime of the different selection types known as fitness proportionate selection, stochastic selection, tournament selection, and truncation selection. In order to do this, I created three different problems for a genetic algorithm to solve. The first problem was to have a user enter a sentence and starting from random evolve a string until it matches the entered string, this problem will hereby be known as the genetic sentence problem. The second problem that I created was to use genetic algorithms to find an optimal solution to the prisoner's dilemma. The final task that I used genetic algorithms to solve was to find the best solution to the traveling salesman problem. I chose these problem spaces because they encompass a wide variety of issues. The goal of this research is to determine which of these four selection types was best at solving the different kinds of problems.

Genetic Algorithms explanation

In order to understand the problem, a clearer explanation of what a genetic algorithm is and how one works is needed. In essence, a genetic algorithm is a self-learning algorithm that remembers previous attempts at solving the problem, and uses those past attempts to generate new, better attempts. As previously stated, a genetic algorithm is broken up into five separate sections; initialization, fitness assignment, selection, crossover, and mutation. These different

sections perform their name. In the initialization period, a set of random possible solutions is created. This set is then passed to have their fitness assessed. Once assessed selection is performed for which possible solutions make it to the next set of solutions. Once this selection is completed a new set is created by crossing over parent solutions into a new solution, and giving the child solution a chance to mutate. Once a whole new set of possible solutions is created, this set is checked to see if there is a correct answer. If there is, the genetic algorithm is completed and the answer is given. If there is not, it goes back to have its fitness evaluated and the cycle starts from there. This is illustrated in the graphic below.

Because genetic algorithms are based upon nature and evolution, this is mirrored in the real world. Take race horses for example. A set of horses are taken. Their fitness is checked, by comparing their speed. Once the horses are ranked, several horses are selected to be bred. Nature runs its course and two parents are crossed over. their child has the chance to mutate some of their characteristics, and a new horse is born. This horse is then put into a set of new horses and the cycle goes on until the Kentucky Derby is won. A genetic algorithm must be able to use the best of given DNA and still be able to explore the problem set. Khalid Jebari, and Mohammed Madiafi write on this saying "The balance between exploitation and exploration is essential for the behavior of genetic algorithms" (2). The pseudocode for a generic genetic algorithm is as follows

> *Initialize a population of N elements with random DNA*
> *while incomplete*
> > *evaluate Fitness*
> > > *if fitness meets criteria break loop*
> >
> > *perform Selection for mating pool*
> >
> > *for N times*
> > > *select two parents from mating pool*
> > > *crossover two selected parents*
> > > *mutate child*
> > > *add child to the new population*
> > *replace old population with new population*

Each part of the genetic algorithm has several different ways that it can be executed. Initialization can be done randomly, or with seeded values, reproduction is traditionally done with two parent solutions, but can be done with more or less. Mutation is done bit by bit, but the mutation rate can change. It can go from 0% to 100% chance. However, the mutation is typically around 1%. Anything much higher, will introduce too much randomness, and anything less, you don't get enough and the sample tends to stagnate.

Implementation

My project focused only on the selection portion of a genetic algorithm. Here I took four of the most common selection types and compared and contrasted them in their problem-solving ability. The four selection methods were fitness proportionate, stochastic, tournament, and truncation. One important point of selection is that there must be a good spread of candidates selected. Without a wide variety of DNA to choose from, the solution has a chance to get stuck on one solution that isn't the best solution. If this selection is not well done, genetic algorithms will not flourish the way that they have the ability to.

Fitness Proportionate selection

Fitness proportionate is the first type of selection that was introduced when genetic algorithms were first being developed. Because of this it has historical background. It is also known as roulette wheel selection due to the similarity of selection it has with a physical roulette wheel. How it works is that for a set of N elements, each with a fitness $F_0 \ldots F_n$, it finds the sum of the fitness for each element in the set and gives each element a chance to be selected with the individual fitness over the sum of the fitness. In mathematical notation, the chance, C, that any element X with fitness F_x would have to be chosen is $C = \frac{F_x}{\sum_{i=1}^{n} i}$. The pseudo code for this function is as follows

Fitness_proportionate(population)
    For the total population
        sum += fitness of current element
    End For

    For 0 to length of the set
        Map the fitness of the population to a number between 0 and 1
        Multiply the mapped fitness by X

        For 0 to the (mapped fitness * X)
            Add the current population to the mating pool
        End for
    End For

The mapping and subsequent multiplying of the fitness normalizes the data; this is needed in order to ensure that the number of times a specific element gets added into the pot is consistent with the others. The time complexity of this algorithm is $O(n^2)$. For my relatively small data sets, this did not cause an issue. Using a selection method such as this one allows a proportionate chance that any selection will be used. All elements are put into the mating pool at least one time, and thus have a chance to be selected. Due to the amount of time that an element with better fitness is entered into the mating pool, the elements with the best fitness have far better chances of being selected, but it is not impossible for them to miss being selected.

Stochastic Selection

The second type of selection that that I used is called stochastic selection. Stochastic is the most complex of the four studied selection algorithms. It is based upon the fitness proportional selection type; however, it is made to be fairer. It uses a single random value to get

a sampling of all the solutions by choosing them at evenly spaced intervals. Here is the pseudo code for the stochastic search.

Stochastic ()
        pointersArray = findPointers()
        for each pointer in pointersArray
            i = 0
            while total fitness of population[0 to i] < pointer
                i++
            end while
            add population[i] to mating pool
        end for
    end stochastic

    findPointers()
        fit = total fitness of population
        num = number desired to keep
        dist = distance between pointers (Fit / Num)
        start = random number between 0 and dist
        for i to number to keep
            *pointers[i] = start + (i * distance)*
        end for
        return pointers[]
    end findPointers

The way that this works is a bit like putting every fitness end to end while in order, and then adding the solutions that fall in every X^{th} order. This allows less randomness and more fairness than even fitness proportionate selection. It forces the most fit candidates to not overflow the mating pool. Here is a picture created by a forum editor with the username "Simon.Hatthon" demonstrating this visually.
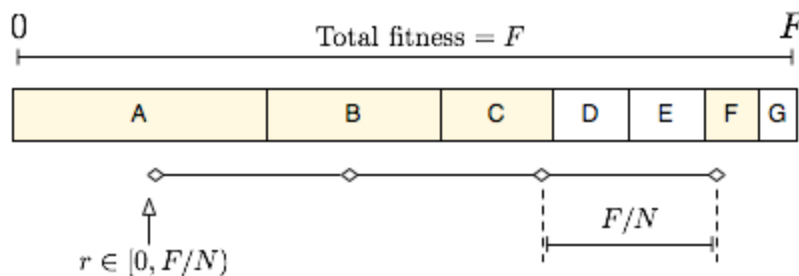
Tournament Selection

      The next selection that I used is called tournament selection. It is one of the simpler methods of selection, and intuitive to look at. This type of selection works by selecting a random set of individuals from the total population, and determining which of these has the best fitness. This one is entered into the mating pool. It completes when the mating pool is full, or at a selected number of individuals is entered depending upon the programmers' choice of development. Here is the pseudocode

    Tournament (population, number of comparison desired)
        for 0 to population length

> *Set best to 0*
>
> *for 0 to number of comparisons desired*
> > *Get current random element from population*
> > *If current element's fitness > best fitness*
> > > *Current = best*
> > *End if*
> *End for*
> *Add best to mating pool*
> *End for*

This type of selection is simple to understand and easy to implement. Depending on the number of comparisons still allows for some "poor" elements to make it into the mating pool to allow for some genetic differences. But it does lean to having only the best make it, given the number of comparisons desired. The time complexity of the tournament selection as described is between $O(n)$ and $O(n^2)$. The most common type of tournament selection has a comparison of two, and this would make the complexity $O(n)$. However, if for some reason the number of comparisons was the number of elements in the population, the complexity would be $O(n^2)$.

Truncation Selection

The final and most simple sort of selection is called truncation selection. In this sort of selection, the population is sorted by fitness, and then drop the lower percentage. The pseudo code is as follows,

> *Truncation (population, truncation percent)*
> > *Sort population by fitness*
> > *Discard bottom percent of population*
> > *Add top percent to mating pool*

The time complexity of the truncation selection is dependent on the sorting of the population. Using a sort such as the merge sort ensures that the time complexity is $O(n \log n)$. While the t truncation sort is the fastest of the discussed selections it has the downside of disallowing the most variation of information in a give evolutionary set. Because only the best opportunities are ever taken, the proposed final solutions could get stuck on a local maximum of being a good answer, but not the best answer.

Information Gathering

In order to measure the four different types of selections for algorithms I created three different programs, all solving three different types of problems. As previously stated I created program to evolve a sentence, a program to solve the traveling salesman problem, and a program to play the prisoner's dilemma. Each of these programs showcase a different strength of the genetic algorithm. The genetic sentence displays the power of genetic algorithms over stochastic guesses, the traveling salesman problem displays strength in finding solutions to NP-hard problems, and the prisoner's dilemma shows how a genetic algorithm can adapt to an outside force. Each program ran 10 times per selection type. I then measured the average number of generations for each selection type that the genetic algorithm took to find an optimal solution This will show the strengths and weaknesses of each selection type on different problem categories.

Genetic Sentences

Genetic sentence is a program that takes an input string, or a sentence, and attempts to use a genetic algorithm to evolve from a random string of equal length into the entered sentence. For example, given a string such as "Hello World! This is my first Genetic Algorithm!" has 48 total characters, including white space. If a program were to try to simply use brute force in guessing the string, using spaces, upper case letters, lower case letters, and symbols such as "**.**", "**,**", "**!**", "**?**", "**''**", there would be 58^{48} total combinations of letters + spaces + symbols. An average computer can not solve this in a reasonable amount of time. However, a genetic algorithm can. In my implementation the fitness was determined by the how close the attempt was to the given input string. For each of the 40 runs I used the same input string, a well-known line from Hamlet; "To be, or not to be, that is the question.". Each selection ran 10 times for a total of 40 times. The program will end when the genetic algorithm successfully evolves the sentence, starting from random to "To be, or not to be, that is the question.". This was inspired by Daniel Shiffman in his textbook *The Nature of Code* on page 394.
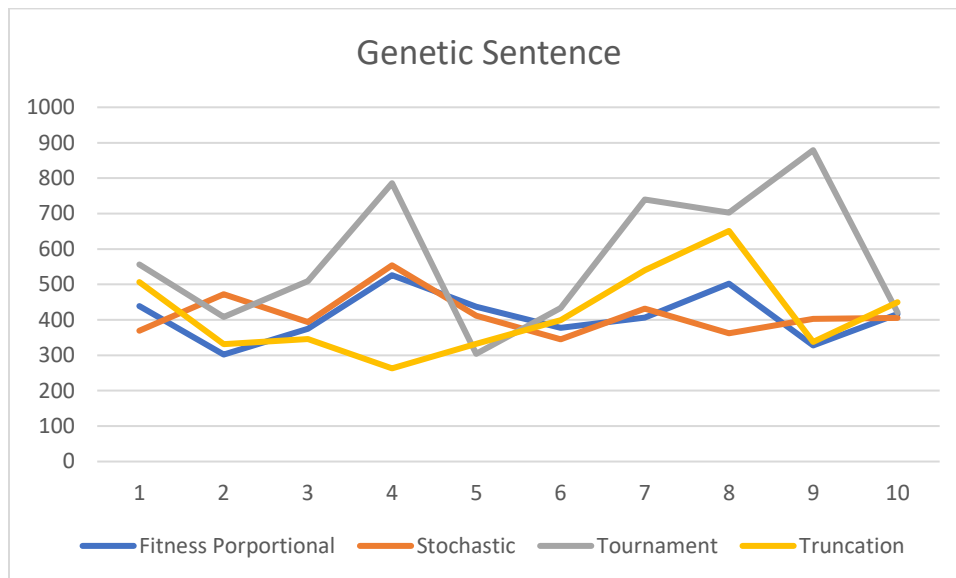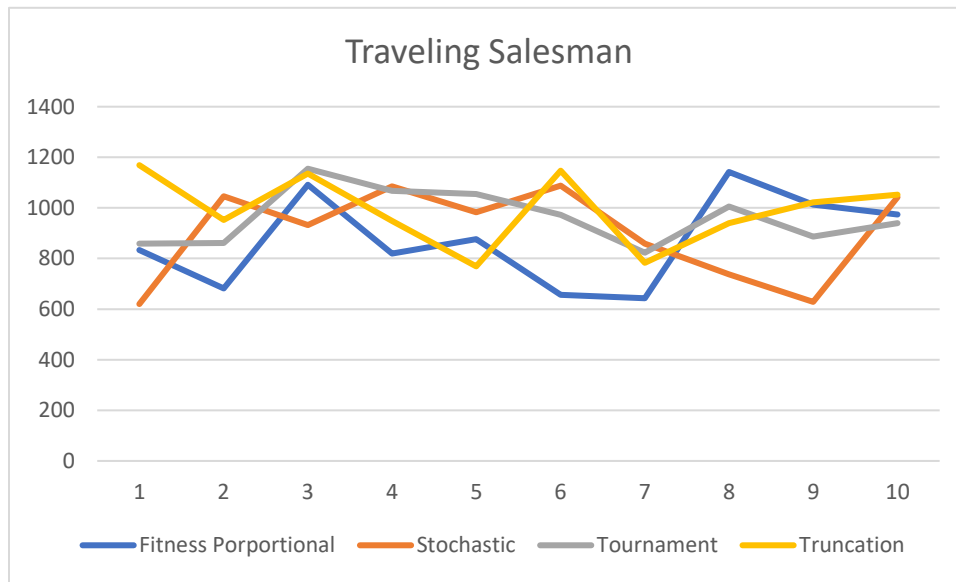

Average number of runs until sentence is evolved			
Fitness Proportional	Stochastic	Tournament	Truncation
410.8	414.8	573.9	415.7

Traveling Salesman

The traveling salesman problem is a classic NP-hard problem where a computer tries to map the best route between a list of cities in order to visit each city once in the shortest amount of time. Due to the nature of the problem being NP-hard, using a genetic algorithm it is not feasible to find a "perfect" answer, only one that is very likely correct. This program created 150 random points representing cites between (0,0) and (200,200) and attempted to map the shortest path to visit all of them. The fitness that this algorithm checked was the length of the passage, thus the fitness was inversely proportional to the total length traveled. The program will end when the genetic algorithm decides on a shortest after starting with random paths.

| Average number of runs until shortest path is found			
Fitness Proportional	Stochastic	Tournament	Truncation
873.1	902.1	962.4	991.9

Prisoner's Dilemma

The last created program was one that attempted to solve the prisoner's dilemma. This is a common game theory theoretical problem where two prisoners, A and B, each attempt to get the shortest amount of prison time. The length of the sentence is determined by if they cooperate with the each other, or if they betray each other. The length of the sentences can be seen here as described in *Genetic Algorithms in search, Optimization and Machine Learning*

		Prisoner B	
		Cooperate	Betray
Prisoner A	Cooperate	1 year, 1 year	3 years, 0 years
	Betray	0 years, 3 years	2 years, 2 years

(Goldberg 141). The fitness here was determined by the length of sentence for A + the length of sentence for B. The shortness of the sentence is inversely proportional to the fitness of the attempt. The implementation of the program assumed rationality – i.e., one prisoner did not want to hurt the other prisoner for some random reason and sacrifice his prison time to do so. The rational answer here is to cooperate, and so the program will end when the genetic algorithm "prisoners" both only decide to cooperate.

Prisoner's Dilemma
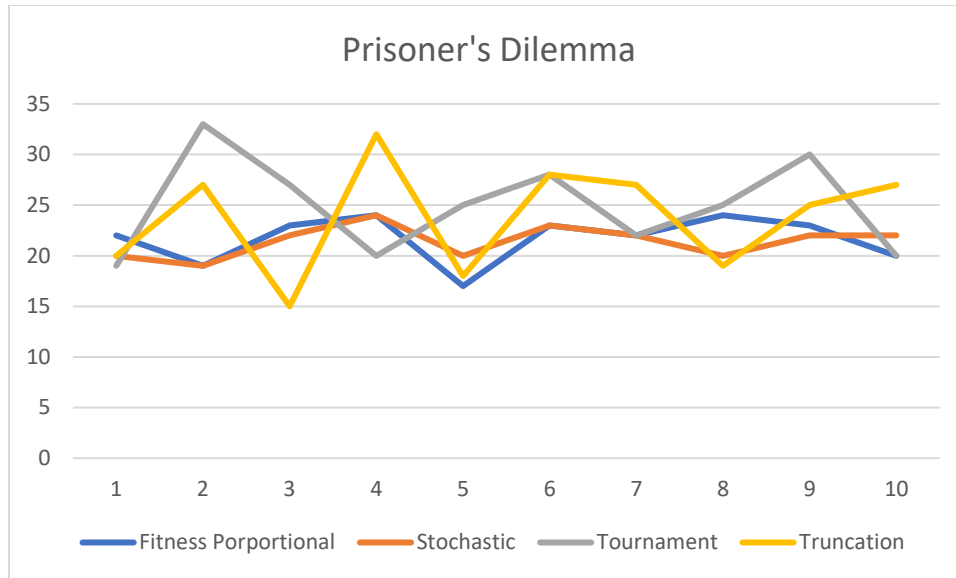
Average number of runs until total cooperation			
Fitness Proportional	Stochastic	Tournament	Truncation
21.7	21.4	24.9	23.8

Analysis

The data suggests that on the whole, there is not too much of a difference between the four different selection types, and that any of the implementations can be used to solve a variety of problem sets. Stochastic and Fitness proportional were the best and the second best on all three of the different programs that were used to test, however they were not much different between the two. This could be because they are of a similar family of selection types, with stochastic being built on top of fitness proportional. Truncation selection being better on average than tournament selection was something that was not suspected. While designing the different selections, truncation seemed to add too little randomness and be too simple to out perform any of the other three selections. This is obviously not the case. Looking at it, the reason that this may have out performed tournament is the fact that tournament had too little of a chance to get the best DNA, while truncation was guaranteed to get the top 50% of it. This gives the genetic algorithm a wide variety of DNA to work with, however it will not have to deal with the very poor DNA dragging the performance down. There was a clear distinction, however, between the performance of fitness proportional / stochastic and tournament / truncation. This is probably due to the spread of DNA that is selected in the first two, while the second is more random.

Conclusion and further testing

While selecting a selection method for any given genetic algorithm, it would be wise to use the fitness proportional method. This method combines ease of understanding and coding, with run time correctness. However, if space is an issue, truncation is the best, as it has the ability to run in the shortest amount of time. There are several ways that this can be further tested. The most obvious is to instead of changing the program to test the genetic algorithm in

different ways, change the other parts of the genetic algorithms. Changing the rate of mutation or crossover would be good examples of this. Other ways that this could be changed is running several different implementations of the same selection method on different languages to see how the performance differs.

Works Cited

Brownlee, J. (2012). *Clever algorithms: nature-inspired programming recipes*. United Kingdom: LuLu.com.

Goldberg, D. E. (2012). *Genetic algorithms in search, optimization, and machine learning*. Boston: Addison-Wesley.

Jebari, K., & Madiafi, M. (2013). Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, *3*(4), 333-344.

Mitchell, M. (1998). *An introduction to genetic algorithms*. Cambridge, MA: MIT.

Poli, R., Langdon, W. B., McPhee, N. F., & Koza, J. R. (2008). *A field guide to genetic programming*. S.l.: Lulu Press.

Shiffman, D., Fry, S., & Marsh, Z. (2012). *The nature of code*. United States: D. Shiffman.