Evolutionary Robotics: Philosophy, Design, and Implementation

Travis Hite

Abstract

Evolution is one of the leading theories by the scientific community explaining how life has reached its current state. Time and time again, scientists have looked to nature to find new ideas on how to perfect their inventions. Genetic algorithms are an attempt by scientists to work principles inherent in this theory to design functions that effectively search a space of potential results effectively.

In this paper, we are going to examine several aspects of evolution, and look at their counterpart ideas in genetics. As well, we will look at how these functions work. Finally, we will apply the concepts in this paper to our own Khepera robot. We will look at the functions present, ideas specific to our robot, problems in evolutionary robotics, and potential ideas for changes to be made in the future.

1.Why Evolution?

The first question people usually ask when I present them with my robot is "Why?" We already know that the optimum result for our particular robot is the set {-30,-30,-30,512} (I will explain what these numbers mean in great detail later), so why make such a complicated process in order to discover this when we are perfectly capable of making this robot perform this function from scratch.

Genetic algorithms, when used properly, effectively search out a result space by continuously changing its method of search in a way defined by the programming. This sort of program is very useful in tasks where the conditions continuously must be changed and adjusted for, giving GAs the flexibility that a straight coded robot will not have.

The original approach to genetic design was "perception, planning, action" known as the divide and conquer design. While this was modestly effective for algorithms that performed one basic task, as the tasks attempted became more and more dynamic this simple scheme was mostly abandoned.

Now consider a tower of child's building blocks. Without the blocks at the base, the tower would not stand. This is the method used by current GAs. The blocks at the bottom represent base responses, that is simple routines that are easy to get reactions based on (say, avoiding walls, or being light sensitive). As you build together multiple

stimuli to respond to, you get an increasingly dynamic result.

It is this power to create dynamic results that are flexible in nature and robust in search that make GAs so useful. The more we explore our ability to make base responses, the better our ability to make more complex responses, and the further we push computing.

2.How Evolution?

In GAs, several concepts of life generation is present in the functions they perform. I will briefly go over several of these concepts.

2.1. Creating a Population

An "individual" is one unique parameter or set of parameters that is being tested on. An individual can either be binary coded or real coded. A binary coded individual will look something like 10001110. A real coded individual will generally look something like {5, 18, 24, 6}. The numbers can represent actual numbers, or potentially an array of objects or responses. For instance, if the response depended upon which ball was present, you may have an array with baseball, basketball, football, soccer ball, and rubber ball. This could be represented with an array between 0 and 4.

2.2. Schema

A simple example of a binary coded schema would be 1*1. The * used in this scenario is used to represent any potential result possible. In this situation the potential results would be 101 and 111. As you increase the space between the definite bits, the amount of search space increases exponentially. Just adding one extra star gives you a search space of 1**1 with results of 1001, 1011, 1101, and 1111.

Generally, a real coded schema will represent a range of numbers. For instance, in the experiment we will speak of later, the results are anything between {-30,-30,-30,1} and {30,30,30,512}. Any potential set of numbers between these groups of numbers, and including the numbers themselves, are potential results in this schema.

2.3. Types of Population

Whereas having one individual representing the population is a possible, it is not very dynamic. Having multiple individuals increases the dynamic of the search space. With just one individual, if there are multiple potential outcomes where not every outcome is optimal, the individual may come up with a sub-par outcome. Using multiple individuals allows for you to continue searching while still holding on to at least one potential outcome.

There are several types of population schemes. In these schemes, mu will represent the population, and lambda represents the offspring (creating offspring will be

discussed later). With this in mind, your population can be represented in several ways. One is with (mu+1) which is a population withjust one slot saved for potential offspring. Another representation is (mu+lambda), where the population and its offspring are treated as one population. There is also (mu,lambda) where the population and its offspring act seperately.

2.4. Reproduction

In nature, an individual is selected for reproduction based upon some characteristic that makes it stand out from the pack. This could be anything from some sort of physical trait, pheremones, even the way something looks. What it comes down to is that reproduction in nature comes down to survival of the fittest. It is the same way in GAs.

An individual is assigned a fitness based on some common factor. This factor may be how close the result is to an optimal result, the actual value of the result, or in interactive evolution it can be a number assigned by the user in some way.

Selection of the individual to reproduce may or may not be a function of their fitness. Consider a roulette wheel. Now, place 15 individuals on this wheel. Now, give each individual a slice where the higher their fitness as compared to the rest of the population the larger the slice. In other words, to find the potential for this individual to be selected, divide the fitness of the individual by the fitness of the entire population. In this particular instance, reproduction continues until one individual clearly has the dominance of the wheel.

Selection can also be completely random. An example of this is tournament selection, where two individuals are randomly selected from the population. Fitness is then used to find the winner of the tournament, and the loser is generally overwritten by the offspring of the winner.

2.5. Asexual Recombination (Mutation)

One potential form of reproduction is mutation. This is done with just one individual, to emulate asexual reproduction. We will briefly discuss the following forms of mutation: single point, multi-point, and sign mutation.

In single point mutation, just one point in the individual is randomly mutated. Take the binary-coded individual 10001110for instance. We could potentially end up with 10101110 for an example where the third place was mutated. In a real coded example, if you had {5,12,24}, a mutation might be {5,28,24}. The second number was randomly mutated from 12 to 24 in this example.

Multi-point is just like single-point mutation, except multiple points in the individual are mutated instead of just one. For instance, say we wanted to mutate the third, fifth, and eighth digit of the previous binary-coded individual. We would end up with 10100111. Or, in the multi-point mutation, we could mutate the first two numbers,

ending up with {16,28,24}. Any potential number within the boundaries of the population is generally a potential result, though a user may choose to keep mutation within a certain limit.

The final example is only on real-coded individuals where the numbers can be between a negative number and a positive number with the boundary numbers equal to the absolute value on either side. Take for instance a trait of the individual that can go between -30 and 30. In a particular individual, this trait could be 16. With sign mutation, this becomes -16.

2.6. Sexual Recombination (Crossover)

If two or more individuals are present for recombination, crossover can be used to create the new individual. Three strategies used for crossover are single-point, two-point, and multi-point.

In single-point crossover, two individuals are crossed at a specific point. For instance, take the individuals {5,12,28,14} and {6,27,8,19}, and make the crossover point between the second and third traits. You could possibly end up with either {5,12,8,19} or {6,27,28,14}.

In multi-point crossover you use the same concept, but at multiple points instead of one single point. Using the same above individuals, and with crossover points between the first and second and between the third and fourth, you could potentially end up with {5,12,8,14} or {6,12,28,14}. The points of combination are determined by the user.

In multi-point crossover, each trait of the two individuals has a 50/50 chance of being represented in the offspring. For instance, again taking from our previous two parents, you may end up with {5,27,28,14} or {6,26,28,19}. Any combination of the two parents is possible in multi-point, and gives you the most dynamic range possible in crossover.

Often, crossover and mutation are used in conjunction in an algorithm to maximize search space.

2.7. Neural Networks

A neural network is a blueprint for a function. A neural network generally has three layers: input, hidden/internal, and output. In more complex networks, there may be multiple levels to the hidden layer. In a neural network, information comes in through the input, travels through the hidden layer, and exits through the output layer, often resulting in some form of reaction or result. The design and layout of the network is generally specific to each network. It is generally represented by a group of nodes connected together through channels.

In a feed-forward neural network, information travels directly from input to output. However, in a recurrent network, the signal may loop back into the network, or even loop through the same node in a network.

The output of a channel depends upon the way the channel is weighted. Consider a graph with x and y axis. Each node will ouput some threshold, which will be represented by the x axis. This threshold will produce the weight of the channel, represented by the y axis. There are three types of representations to this weight.

With a step function, before the threshold is hit the weight is set to either -1 or 0. After the threshold, it outputs 1. This can be considered like a switch, either it is on or off. In a linear function, the input is directly proportionate to the output, along a slope of k. Often this weight is between -1 and 1. Finally, with a sigmoid function, the weight is squashed between 0 and 1 with a defined slope of k.

Genetic algorithms are often used to evolve neural networks. There are many justifications for this:

Smoother search space
Varying evolutionary granularity
Straightforward mapping from sensor to motor
Robust to noise
Biologically plausible
GAs explore population of networks, not a single networks
No constraints on type of architecture
Detailed specification of network not needed.

2.8. Evolutionary Concepts

There are some good things to keep in mind when designing your network. For one, there is the concept of genotype vs phenotype. The genotype description of your GA is the code, the robot, and any physical part of your network. There is also the phenotype of your algorithm. This is the actual result of your GA. In interactive evolution, the person is responding to the phenotype of the individual, not the genotype. Keep this in mind, as having an obvious result of mapping between genotype and phenotype makes for a far better algorithm.

There is also the distal vs proximal view of the robot. In a distal look, you are only seeing the code and the actions taken. However, a proximal approach is also beneficial. This is sort of like thinking of the world through your robot's eyes. Remember that the view of the world through its eyes will differ significantly from your own. A good programmer will keep both views in mind during programming stage.

3. The Robot

In our experiment, we are trying to evolve a robot to avoid walls. We are not the first people to design a robot based on GA design over a neural network to do such a

thing. This is a proof of concept. While your average evolutionary robot will take up to several days to properly program, and use somewhere in the range of 40 different weights, we are designing our robot to function in a matter of minutes, based on user input.

3.1. Khepera

The robot we are using is a Khepera robot. It is 55 mm in diameter and 30 mm tall. It has a Motorola 68331 processor, eight infra-red proximity and light sensors, and two DC motors.

Khepera robots are built in such a way to provide expansions to the robot by placing new components on the top. In the future we are going to get a gripper to add to the top so it may pick up small balls or various other objects. We are also considering adding a wireless connection, though there is a delay in this connection that would be detrimental to our design.

3.2. The GRNN

Our Khepera uses a general regression neural network. A physical concept of the function of the network is to picture a round disc, which would represent the robot. Divide this robot straight down the center. One wheel is on either side of the robot. On the top hemisphere of the robot, 6 sensors are equally distant from each other, as well as 2 on the bottom hemisphere. The top sensors are set to wall detection. Light bounces back to the sensors which take a reading. When a reading of 1023 is received (chosen because it is the highest possible reading), the sensors activate with a value of 1. The sensors then feed through a connection to the wheel, with the GRNN being the focal point. This is where our algorithm comes in to play.

3.3. The Weights

We have defined a set of 4 weights. The first three weights (each corresponding to the front sensor, the mid-front sensor, and the mid-back sensor) have a range of -30 to 30. The fourth value (the sigma value) is between 1 and 512. An example value would be {18,-26,-4,354}.

The sigma value controls the overall wheel speed of the robot. The higher the value, the slower the robot goes. The other three weights either speed up or slow down the wheel on the opposite side of the sensor it is responding to. An optimal result is to have the robot go as slow as possible and turn as fast as possible to avoid walls and decrease potential damage. This result would have the value of {-30,-30,-30,512}.

The use of only four weights is another part of speeding up the process. Many algorithms will use 40 or more weights in their experiment. Using less weights decreases the amount ofprocessing time required for each individual set of weights.

3.4. The Stage

For our experiment we are using a simple white poster board stage with size 30x30 cm. The small size of the robot and the area, as well as the small size of the stage and the cushioned poster board are designed to keep the robot safer. A smaller robot will have a smaller overall velocity for the same comparative results, which leads to better safety for the robot.

3.5. The IEC

For a user-defined number of cycles, Interactive Evolutionary Computation is used (From here on referred to as the IEC). The beginning of IEC is to create a population of individuals are randomly generated, weighted at 0 to keep the population from being biased.

After a population is created, two candidates are taken at random to be parents. If one of the candidates has been aborted in the past, one of the first three weights has sign mutation applied to it. Each parent is allowed to control the robot for four seconds. During this time, the user interacts with the robot through a webpage connected to a server to the computer hosting the experiment. For each button click, or when the button is held down, hits are assigned to the individual. Multiple users can use this interface at the same time to apply fitness. The winner of each tournament is the individual with the least number of hits. If the robot uses a behavior that is extremely dangerous (erratic, hits walls often at a high velocity, lunges back and forth, etc.), the candidate is aborted and assigned the maximum number of hits.

The difference between the two individuals is then memorized for later use. For instance, if we had an individual p1{-14, -25, -17, 136} and another individual p2{-20,-25,-22,388} the distance between the two would be <1,0,1,-1,1>. A value of 1 means the first number was higher. A value of 0 means the values were equal. A value of -1 means the value was less. The last value, which can only be -1 or 1, represents the winner (either the first value or the second). Since the second value is both slower and will turn faster, the second value won.

After both individuals are tested and a difference is set, one of our evolutionary operators come in to play. If both individuals were aborted, a random individual from the population is selected and mutated twice to replace both parents. If there is a case where the winning behavior is feasible and the losing behavior is infeasible, uniform-bounded mutation is used with a range of -12 to 12 for the first three traits and -50 to 50 for the final trait. The mutation then overwrites the loser, the winner is put back into the population, and a new individual is selected to compete against the loser.

Another scenario occurs when a winner is selected that has not been aborted but has not won at least two tournaments. When this occurs, uniform-crossover is used. What happens with uniform-crossover is a new individual is created from the parents. The offspring then replaces the losing parent. The winner is placed back in the population, and a new candidate is selected to go against the newly created individual.

The third scenario occurs when both parents have been deemed to be feasible (not aborted, two wins). In this case, uniform-bounded mutation is used again (in the same way as previously discussed).

3.6. The MEC

When the IEC is finished, the MEC begins. At this point, the robot becomes still, and "meditates" (thusly the M) on its previous results. Two individuals are taken from the population and generate a distance vector as before. A search is then conducted for the difference vector that most closely represents the difference vector it came up with. A winner is then selected based on which individual was selected before. Thusly, previous user preference is applied throughout the meditative process. Evolution continues as it occurred in the IEC. Using the above scenario, the algorithm produces what it considers to be its best example.

4. Interacting With The Robot

One might say it's potentially bias, others will say its entirely necessary, however in order to properly train the robot one must learn how to interact with it. For the first couple of interactions, the robot behaves in a clunky fashion and often will run straight in to walls and bang against them repeatedly. If you just assign a negative feedback all the time at this level, the robot will not have learned.

Training the robot is comparable to training a four year old after it has had too many pixie sticks. It is erratic, full of energy, and will only partially listen to and understand what it is told. It is only by slowly building up that it will eventually learn. Start off by only assigning hits as it hits the wall. After it has learned to avoid the wall, assign hits when it comes close to the wall. Following that, assign hits when the robot does not turn smoothly. Lastly, assign hits when the robot moves too fast. This process is similar to staged evolution in practice, but create good results.

5.Problems

We have come across several problems with running this robot. For one, since user interface time is condensed, any human error can create exponential errors through the MEC. The only real way to alleviate this problem with the current design is to be sure to pay close attention to the robot during the IEC time. This is not a major problem, since the IEC generally only takes a few minutes to go through all of its iterations.

However, since the reaction is based on the phenotype, and since the fitness applies to the ENTIRE individual, ratings will never be entirely consistent. However, results will generally lead towards a desired outcome. Upon occasion, the result is entirely different from what is wanted on one or two variables. This is not the norm, so it can be overlooked.

Another problem has to do with the sigma value. Since the difference between

256 and 512 is hard to see with the human eye, often a bad sigma value can occur. The problem is also greater because hitting the robot just because it is moving faster can throw off the other weights. An alternative to this, though admittedly biased in nature, is to hit the robot every time it comes close to a wall regardless of turning or not. The concept here is that a robot that moves faster will come close to the wall more often.

6.Ideas

The largest part of my time spent on this project was research. The last two to three weeks however were spent brainstorming on ideas for way to make the robot more receptive to commands and generally run better.

The first idea we had on improving the robot was improving the interface itself. While Joe and Lacy are still working on their improvements, the idea we came up with for my improvement was found to be infeasible.

The idea was to build a steaming webcam into the webpage used to interact with the robot so the robot could be left on and interaction could take place from anywhere. I did some research on past experiments on this area and information about found out about the java media library. However, even the best results saw delays up to eight seconds. The problem is that how java steams video is that it writes the information received from the webcam to a buffer file. This file is then steamed to the receiving individual where it is decoded and displayed. Though we are sure there must be a way to bypass this delay on a large network such as Auburn's that would ensure maximum speed, it would have been too much work to attempt to implement.

Another idea we had was to use the model of user preference created by the distance vector to make mutation biased towards user preference. We did this by counting the number of times each sensor in the vector had a 1 or a -1 (flipping this if the first parent had won instead of the second). The percentage of this preference was added to the range of the mutation.

Tests so far on this new operator are inconclusive. We have had several mechanical problems, as well as many scheduling problems. In the end, we did not have nearly enough time to perform the 10 trial runs that we needed. Lacy and Joe will continue their work on the robot with Dr. Dozier this fall. A paper will be produced from these results, as well.

7. References

Dozier, G. (2001) *Evolving Robot Behavior via Interactive Evolutionary Computation: From Real-World to Simulation* Auburn, AL

Takagi, H. (2001) *Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optomization and Human Evaluation* Fukuoka, Japan

Nolfi, S. and Floreano, D. (2000) *Evolutionary Robotics* pp. 1-68

Fogel, B. and Chellapilla, K. *Revisiting Evolutionary Programming* San Diego,

California
Nolfi, S (1998) *Evolutionary Robotics: Exploiting the full power of self-organization* Roma, Italy
Dozier, G (July 27, 2005) COMP-6600: Artificial Life http://www.eng.auburn.edu/users/gvdozier/syllabusComp6600.htm
Spears, W. Jong, K. Back, T. Fogel, D.B. and Garis, H. (1993) *An Overview of Evolutionary Computation* Washington, D.C.
Walker, J. and Oliver, J. (1997) *A Survey of Artificial Life in Robotics* Ames, Iowa