Evolutionary Programming

Searching Problem Spaces

William Power

April 24, 2016

1 Evolutionary Programming

Can we solve problems by mi:micing the evolutionary process? Evolutionary programming is a methodology that believes this is possible. If a problem can have its possible solutions expressed in some manner of gene, then a solution could be found by simply trying possible genes until an optimal one is found. If a population of these possible solutions is maintained, then genetic operations like reproduction could be used to build new populations. Evolutionary programming is a method that can define these genetic operations, and use them to quickly explore the space of possible solutions.

Evolutionary programming has its roots in the original generation of artificial intelligence research. The first major text was by Holland, and described the first form of EP, genetic Algorithms. This simple model treats individuals as bit strings, with each bit representing the presence of some allele. Holland introduced the core concepts of EP, genetic operations, fitness functions, and a framework for modeling search efficiency.

Hollands work formed the basis for EP, and was expanded upon by Koza. Koza took the idea of the genetic search and applied it to a more complicated context. Instead of representing possible solutions as bit strings, they were composed of trees of atomic operations. This allowed for the modeling of more complicated, program like, problems. Koza named this framework genetic programming.

This paper will give an overview of both Genetic Algorithms and Genetic Programming. It will discuss the efficacy of the genetic search by examining Hollands schema theory. Finally, it will discuss an example implementation of a genetic programming experiment.

2 Genetic Algorithms

Genetic algorithms are one of the earliest forms of genetic search. They solve problems by tailoring a fitness function to express the relative 'goodness' of a given individual in a population. This fitness is then used as an input parameter for genetic operations that recombine past members to form new ones, all in proportion to their past fitness.

2.1 Representation

To start, each individual needs some consistent method of representation. The most common is to consider each individual as some bit string of a set length. It would then be up to the fitness function to take this bit-string and interpret each bit in the problem context, and use them to produce the fitness value. This interpretation can vary greatly from problem to problem.

On the simplest end, the bit string could be interpreted just as a value. This is useful for situations where the genetic search is trying to minimize some given function. More complicated interpretations take each bit as an indication of the presence of some allele. The presence of different alleles would then have some impact on the over all fitness.

2.2 Fitness

The interpretation of the genetic representation is tightly coupled with the fitness function. The fitness function is the basis for the genetic operations. It must be able to map a given individual to some real value, with the assumption that a good individual will have a higher fitness than a worse individual. As we saw in the representation section, these can vary greatly. One of the simplest fitness functions occurs in the minimization of a mathematical expression.

For example, consider maximizing the function f(). Each individual would be interpreted as an integer, than passed to the function. The resulting value could used directly as the fitness function.

Larger values are closer to the maximum than smaller ones, so they return a larger value. This satisfies our requirements for a fitness function.

These fitness functions can also be tailored to fit more complex contexts. So long as care is taken to ensure that the overall function is monotonic, and increases with 'goodness', it can be used in the genetic operations. Consider a situation where each bit represents the presence of some item in a bag, and these have some independently associated weights and values. The fitness function could be an attempt to maximize the value while minimizing the weight. To do so, a weighted combination of the two values could be used, and summed over all present bits.

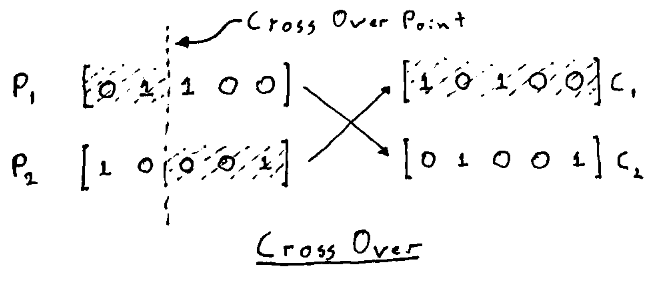
2.3 Genetic Operations

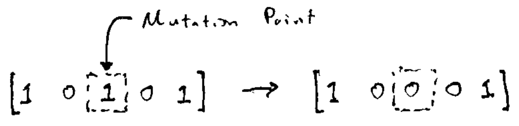
Genetic operations are the means by which a new generation is created from an old one. Hollands genetic algorithm makes use of three genetic operations; selection, crossover, and mutation.

Selection is simply adding high performing members of the old population to the new one. Generally, this is not always the best performer, but instead based on the relative fitness of each individual.

Genetic crossover produces two new individuals from two parent individuals. A location within the gene is selected, and used a crossover position. The 'chunks' on either side are swapped between the two individuals. The resulting bit strings represent the new members of the population.

Mutation occurs based on some small probability. Occasionally, a random bit within an individual will be flipped. This chance generally starts high, and steadily decreases as a genetic run processes. This ensures an increase in exploration in the start of a run, while favoring exploitation towards the end.





Mutation

The choice of which parents to use for selection and crossover is based on the fitness score of the individuals. Initially, one would think to simply select the best performing individuals directly for reproduction, however this is not beneficial. Selection must be proportional to the fitness of an individual. This further ensures a balance between exploration and exploitation of the search space. Without the small chance for a non optimal individual to reproduce, runs can quickly converge on a sub optimal local minimum of the fitness function.

2.4 Drawbacks

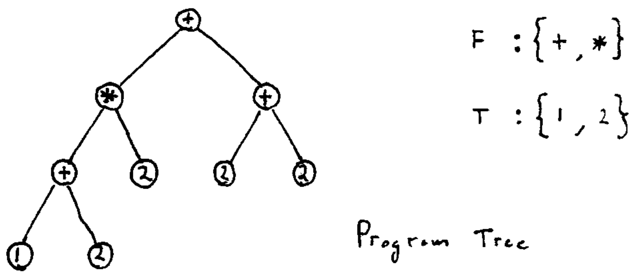
In its simple form, GA provides a means to solve many problems, however it does have its drawbacks. GA requires the formulation of a problem in a simple bit string, which is not always practical. In addition, the fitness function must be well tailored, and its creation may be a problem of larger scope than the initial one. The combination of these factors makes it difficult to use GA for complex problems. However, it does provide a basis for a framework that can model the efficacy of genetic search. This will be covered in a later section.

3 Genetic Programming

Genetic programming expands on the ideas of GA, allowing it to express more complex problem solutions. Instead of considering discrete groups of alleles, genetic programming instead considers programs as the representation of an individual. That is, given some set of atomic, or basic functions, a possible problem solution is a composition of the functions and some input values.

3.1 Representation

It is natural to represent a computer program as a tree. In fact, languages like LISP are based on the idea that a computation can be reduced to an abstract syntax tree, which outlines its composition from atomic functions. In a tree representation, programs proceed from a root node, which represents some function with some inputs. These inputs themselves would be nodes representing further functions, or eventually terminal nodes, containing just values. Consider the example below.



The possible trees are limited only by the set of functions and terminals made available to the genetic program. One can imagine the depth of problems that can be expressed using such a system. The gene of a given individual is simply the tree representing its program.

Trees are not the only possible representation. Other computational structures could be used. Consider a state machine. If a problem can be expressed as some state machine acting on a constant set of nodes, then we could apply the genetic search methods to it. Individuals would then be the set of output states for each possible transition.

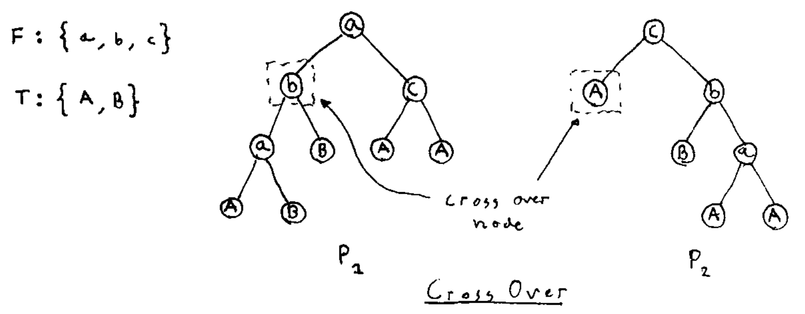
3.2 Fitness

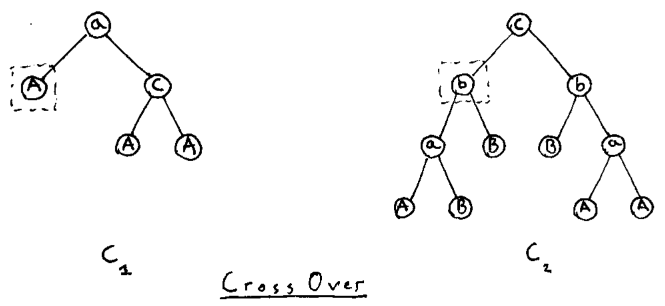
The fitness function for a GP behaves very much the same as it does in GA. It must take some individual, and provide some real value fitness score. In GP, this usually involves running the candidate program and evaluating its response to the problem context. This will be shown in more detail in the exampl implementation.

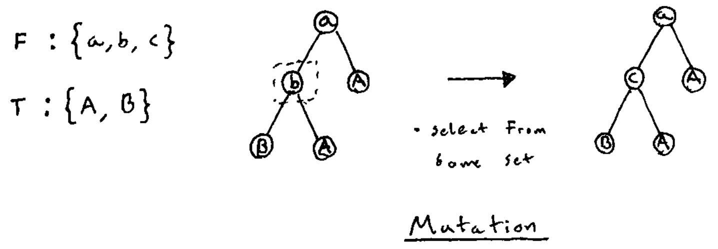
3.3 Genetic Operations

The genetic operations are similar to those used in GA. In fact, the behavior is exactly the same, the difference lies in how to select a crossover point, and how to apply mutation. These depend on the chosen representation of the program.

In a tree representation, crossover is interpreted as the selection of a node as a crossover point in each parent. These nodes, and all their children, are then swapped to create the new individuals. Mutation is the selection of a node, and the replacement of it with another function from the function set.







3.4 Drawbacks

While genetic programming improves upon genetic algorithms, it still has its drawbacks. A problem must be constructed in a way that lends itself to a programatic solution, while also having a well defined fitness function. Both of these are difficult problems in their own right. In addition, great care must be taken to ensure that local sub-optimums do not trap a simulation in its early stages. This is done through careful initialization of the population, and appropriate tuning of hyper parameters like mutation rate and maximum tree size.

4 Search Efficiency

These evolutionary methods are a search of some space, so it is worthwhile to see that they outperform a naive one. The schemata framework is a mathematical interpretation of the genetic algorithm. It provides an argument for the existence of many, well performing groupings of alleles. These building blocks will tend to be small, and the population of individuals will be able to 'process' a great many of them at a time. This is the supposed advantage of a genetic search, the parallel processing of a large number of possible 'genes'.

4.1 Schema Theory

A schema is a template of possible values for a genetic string. They are formed from an alphabet of 1, 0, and *, where * is a 'dont care' symbol. For example, given a gene of length 5, here are some possible schemas.

$A : \{0, 1\}$	$H_1 = [* \; * \; 0 \; 1 \; *]$	$[1 \; 0 \; 0 \; 1 \; 0]$ - H_1, H_3
$S : \{0, 1, *\}$	$H_2 = [0 \; * \; * \; * \; 1]$	$[0 \; 1 \; 1 \; 1 \; 1]$ - H_2
$l : 5$	$H_3 = [1 \; * \; 0 \; * \; 0]$	$[0 \; 1 \; 0 \; 1 \; 1]$ - H_1, H_2
$*$ = 'don't care'	Example Schemas	Matches

Schemas have two important characteristics. Their defining length, and their size. The defining length of a schema is the distance between the most extreme non-'dont care' symbols. The size of a schema is simply the number of non-'don't care' symbols.

$$H_3 = [1 \; * \; 0 \; * \; 0] \qquad l(H_3) = 5 \qquad s(H_3) = 3$$

$$H_1 = [* \; * \; 0 \; 1 \; *] \qquad l(H_1) = 2 \qquad s(H_1) = 2$$

Length and Size

At any time, a population of individuals will contain some number of schemata. That is, it will have strings matching some number of the possible schematas. We can attempt to model the number of individuals in the population that hold a schema at a given time. Consider the function $m(H,t)$.

$$m(H,t+1) = m(H,t)\tfrac{f(H)}{f}$$
$$\bar{f} = \text{Average Population Fitness}$$
$$f(H) = \text{Average Fitness of Schema}$$

The behavior of m() is the core of the schema theory. If we could somehow model the relative behavior of m for 'good' and 'bad' schemas, we could obtain a measure of the overall number of good schemas as generations reproduce. In general, we can imagine that a good schema will have a m() that increases over time. If the presence of schema H always yields some constant positive amount of fitness, it will have an exponential m(). Likewise, if a schema always yields some constant negative decrease in fitness, its m() will decay exponentially.

$$m(H,t+1) = (1+c) * m(H,t) \text{ Assuming Some Constant Benefit/Cost}$$
$$m(H,t) = m(H,0) * (1+c)^t \text{ Exponential Growth}$$

4.2 Implicit Parallelism

Holland uses the multi-armed bandit model to show that the number of schema present in a population can be bounded to the following range. By making assumptions about the relative frequencies of good schemas of a given length, he was able to further constrain this to an estimate for the number of good schemas being processed by a population at any time. Interestingly, this is some constant times the cube of the size of the population. That is at any time, a population of n individuals contains n cubed many good (relativly) schemas. Holland called this the implicit parallelism of the genetic algorithm.

$$n_s = \frac{(l - l_s + 1)n^3}{4}$$

l_s = length of a schema

l = length of gene

• Assumes $n = 2^{l_s/2}$

$$\therefore \quad n_s \in O(n^3)$$

<u>Number of Schemas</u>

4.3 Building Block Hypothesis

The combination of implicit parallelism and the behavior of the m functions leads to the building block hypothesis. If small, well performing schemas will be increasingly well represented in a population, then they will provided the large share of components of good individuals. The fact that there will be on the order of n cubed many of these in a population at any time suggests good individuals will quickly be found. This is the core concept behind the efficacy of genetic search. Building blocks will be found, and they will be composed in a highly parallelized manner within the population.

5 Example Simulation

To explore the techniques of EP, a simple problem context was conceived to solve with Genetic Programming. The problem is developing a succesful control function for a simulation of an insect.

5.1 Description

The simulation is a grid, representing the world of the insect. A value of 0 indicates an empty cell, a value of 1 represents food. A single unit, the insect, moves about the grid, one square at a time. In effect, it has 4 options each turn, corresponding to the 4 directions it could move. Every time step the insect looses a unit of health, and upon entering a cell with food in it, gains some amount of health. To help make the decision, the insect is able to 'see' the four cells next to it. The goal of the simulation is to find a control function that maps the view to one of the movement directions.

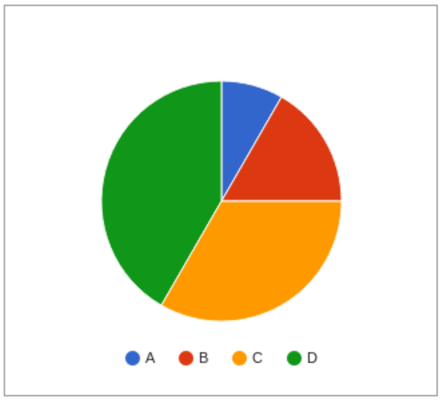
As discussed in the GP section, the program found by a genetic search will be composed from some set of functions and terminals. In this simulation, the function set is compose of four functions that each check one of the cells near the unit and branch. An additional function, random, picks a random branch between its two children. The terminal set is composed of the four directions the unit can move. This means that ultimately, a movement will be 'returned' by the program.

Functions	Terminals
`IF_UP`	`MOVE_UP`
`IF_DOWN`	`MOVE_DOWN`
`IF_RIGHT`	`MOVE_RIGHT`
`IF_LEFT`	`MOVE_LEFT`
`RAND`	

Evaluating the fitness of a given program is based on its performance within the simulation. The metric chosen for this project was to track the total number of food cells that a given individual consumed before losing all of its health. This fits the requirements discussed previously, it is monotonic, and a larger value correlates with a 'better' individual.

The tree methods described in the GP section can be used to implement the genetic operations of selection, crossover, and mutation. In addition, the simulation makes use of fitness-proportional selection. In this, the candidates for selection and crossover are selected based on their share of the total fitness. That is, each individual has a chance proportional to its fitness for being selected. This can be easily thought of as a 'roulette wheel' where each individual has a wedge sized in proportion to its fitness. Every time a new individual is needed for either selection or crossover, the wheel is spun. An entire run of a genetic simulation is outlined in algorithm one.

Individual	Fitness
A	1
B	2
C	4
D	5
Total	12



The full source code for the implementation can be found on github at www.github.com/wpower12/UnitGP. The project requires a recent java installation and maven to build. The main page of the github repository contains a more in-depth description of the entire code base, and the implementation of the specific genetic programming methods.

Algorithm 1 Genetic Programming

```
n = Size of Population
n = Number_Selected + Number_CrossOver
population[n];
newPopulation[n];
fitnesses[n];
population = randomPopulation();
for 0 to Generations do
    fitnesses = evaluate(population);
    for 0 to Number_Selected do
        newPopulation.add( select(population, fitnesses) )
    end for
    for 0 to Number_CrossOver/2 do
        parent1 = select(population, fitnesses)
        parent2 = select(population, fitnesses)
        child1, child2 = crossover( parent1, parent2 )
        newPopulation.add( child1 )
        newPopulation.add( child1 )
    end for
end for
```

6 Conclusion

In all, evolutionary programming is a very interesting topic to explore. The biomimicry aspect is appealing, and the overall concepts overlap with some of the main themes of General Artificial Intelligence. EP is a search for meaningful behavior in an efficient, and almost self organized manner. While there are many drawbacks to the naive approach, I believe it is a very useful basis for the discussion of the efficacy of EP. The building block hypothesis, if it extends to genetic programming in general, would lend itself to an exploration of the problem space for behavior. That is, I hope that EP can be used as a means to more efficiently search the space of possible 'programs' that could resemble an intelligence.

Specifically, this seems to be an exciting concept to combine with statistical methods like neural networks. Many statistical methods rely heavily on hyper-parameters. Elements of the system that can not be directly explained or modeled, and that vary greatly from problem to problem. These parameters, such as the layout of a neural network, the size of a layer, or the choice of activation function, could be considered genetic elements of an individual. Small runs, and the use o a test set for a fitness function, could lead to an efficient search of the hyper-parameter space. Perhaps speeding up the development of specifically designed networks.