A More Stable Approach To LISP Tree GP

Joseph Doliner

August 15, 2008

Abstract

In this paper we begin by familiarising ourselves with the basic concepts of Evolutionary Computing and how it can be used to hasten searches in large combinatorial optimization problems. We continue by looking at Holland's Theorem which address the problem of locality that makes scaling of genetic solutions difficult. We then look at how these concepts apply to Genetic Programming where the goal is evolution of computer programs to solve specific tasks. Finally I will present a novel solution that hopes to overcome the problem of locality.

1 Background

Evolutionary Computing (EC) is a promising subfield of Artificial Intelligence which has only recently become an active topic of research. EC aims to apply the concepts of biological evolution and natural selection to combinatorial optimization problems. Since EC is a new field, exactly how one should apply the concepts of evolution is still debated. However most EC implementations do agree on a few things: the process begins with a random selection of solutions from the solution space $\mathcal{S}$, often referred to as the "primordial soup". This random selection forms the first generation, $\mathcal{G}_0$. Each solution of the generation (called an **organism**) is then evaluated by a **fitness function**, $f : \mathcal{S} \to \mathbb{R}$, and assigned a **fitness**. The generation then procreates. Organisms are randomly selected (naturally selected) from the population with respect to a strictly increasing function of fitness to participate in simulated sexual reproduction in which offspring are created which are genetic similar to both of the parents. After reproduction the genetic information of the offspring is randomized slightly to simulate mutation. The

computation ends when an organism is found to have a certain high level of fitness (possibly maximal). Many implementations also stop after a certain number of generations. There are very little consensus on the specifics of how this should be handled, however EC has proved very good at solving certain problems which elude other methods such as exhaustive search and so interest in the field has remained high.

2 Encodings

Generally the aspects of EC that are open to interpretation are the encoding of the solutions, the algorithm used as the reproduction operation and the fitness function. In most implementations the reproduction algorithm is very simple, almost completely evident from the data structures. Thus of particular interest to this paper is the encoding of the solution, the data structure which is used to represent elements of the solution space.

2.1 Holland's Schema Theory

One theory to aid in understanding *how* EC works is the building block theory. This theory holds that over time organism discover building blocks which convey above average fitness. This building blocks then spread throughout the population and are combined with other above average blocks until a solution is found. To investigate building blocks we need to introduce the concept of a **schema**. Consider a simple encoding of strings of length N from the alphabet $\{1, 0\}$ in this situation a **schema** is a string from the alphabet $\{1, 0, \star\}$ where $\star$ represents "do not care" for example

$$\star 10 \star \qquad \text{represents} \qquad \{1100, 1101, 0100, 0101\}$$

Holland's Schema Theory predicts how the abundance of schema should change over time [2].

The **abundance** of the schema H in the generation t is denoted $m(H, t)$ and is calculated simply as the number of fraction of organisms in the population who are representatives of the schema. The number of non-$\star$ symbols is called the **order**, denoted $\mathcal{O}(H)$. The distance between the farthest two non-$\star$s is called the **defining length**, denoted $\mathcal{L}(H)$

Suppose that we use a crossover reproduction. That is to reproduce we take two strings pick an integer $k \in [1, N-1]$ and create two child strings

in which the first k symbols in the first child are the same as those in the first parent and the remaining $N-k$ are the same as those from second parent. Note we don't allow reproduction at the very ends of our string as this would propagate the parents into the next generation, something we may choose to do separately The second child is the same method but with the parents roles reversed. Note that this operation (which we denote $\heartsuit_k$ depending on our choice of k) is closed under our solution space; this is a requirement of reproduction operations. For example, if we have the two strings 1111111111 and 0000000000 ($N = 10$) then:

$$1111111111 \heartsuit_5 0000000000 = \{1111100000, 0000011111\}$$

Holland's Theory states that for a string encoding:

Theorem 2.1.

$$E\left[m(H, t+1)\right] = Mp(H,t) \cdot (1-p_m)^{\mathcal{O}(H)} \cdot \left[\frac{\mathcal{L}(H)}{N-1}(1-p(H,t))\right]$$

Here M is the size of our population; p_m is the probability of mutating a bit (recall that our model involves random mutations). $p(H,t)$ is the probability of selecting a representative of the schema calculated as $p(H,t) = \frac{m(H,t)f(H,t)}{Mf(t)}$ where $f(H,t)$ is the average fitness of the representatives of H in generation t and $f(t)$ is the average fitness of an organism in generation t. Furthermore $E\,[]$ represents the expected value of a random variable. What this is saying is that as expected fit schema will increase in abundance will unfit schema will decrease in abundance. However as our schema grow in size the chance that crossover will occur somewhere within the schema thus destroying it increases. This theorem holds for any string encoding that uses crossover reproduction and for the more advanced encodings we will merely need to adjust our concept of the length of a schema.

2.2 Locality

Holland's Theory considers EC as a search through the possible schema. As we sort through the schema randomly we come upon certain ones that are better than others. We select such that these better ones become more abundant in the next generation than their less fit counterparts and thus are explored more. As the process continues the schema are perfected and combined with one another making more complicated (longer) schema. If

we take this to be how EC works then we can expect that it will work best when the equation above is largely dependant on the fitness of the schema. Toward this goal we will completely ignore the $(1 - p_m)^{\mathcal{O}(H)}$ term; since we have complete control over it; it's normally close to 1. However the big problem here is that the equation becomes overly dependant on the term $\left[\frac{\mathcal{L}(H)-1}{N-1}(1 - p(H,t))\right]$ as $\mathcal{O}(H) \to N$ since:

$$\text{As: } \mathcal{O}(H) \to N \quad \frac{\mathcal{L}(H) - 1}{N - 1}(1 - p(H,t)) \to 1$$

For an EC to perform according the Building Block theory it is important that it increase the abundance of fit schema will decreasing the abundance of bad ones. As the theorem shows implementations have trouble doing this when the schema grow to large. This property of an EC, how good it is at proliferating fit schema regardless of length is called the locality. Encodings that exhibit poor locality have a tendency to proliferate superschema of the fit schema by crossing over within the schema. Encodings the exhibit poor locality have trouble scaling to find specific solutions in large spaces.

2.3 An Illustrative Example

Consider the following example:

Straight	
Solution Set	$[0,9]^{10}$
Fitness Function	$f(S) = \text{Sup}\{L(s) \mid s \text{ substring } S \text{ and } s \text{ is a straight}\}$
Reproduction Mechanism	Crossover (denoted $\heartsuit_k \quad k \in [1,9]$)

Here a straight is just a string $s_1 s_2 \ldots s_n$ s.t. $\forall i \ s_{i+1} = s_i + 1$. Here we use $L(s)$ as the length for a string, not for a schema. Our intuition about this example is that it should exhibit poor locality. If we consider the schema 012345678$\star$, then despite the fact that this is an exceptionally fit schema the probability that an offspring will be a member of the same schema is around $\frac{1}{9}$ since crossover occurring anywhere but the last position ($k = 9$) will result in offspring that aren't members of the schema.

3 Other Encodings

3.1 LISP Trees

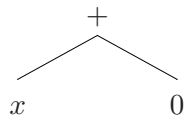
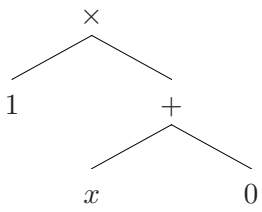
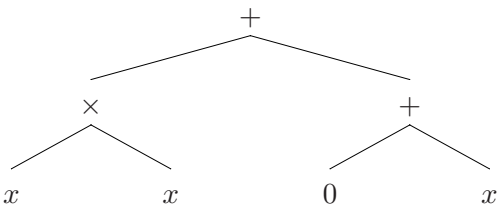
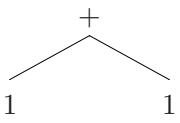
Here we turn to a subset of EC called, Genetic Programming (GP). Genetic Programming is evolutionary computing in which our solution set consists of computer programs which are evolved to solve a specific problem. GP also sometimes refers to the evolution of physical aspects of machines. In particular it has been used with great success to evolve electronic controllers and circuits. To solve this type of problem our encodings, and reproduction operations, have to get more complicated and along with them our reproduction operators. To begin with, we select a set of of **terminals** which are independent variables (inputs to our functions), constants, and zero-argument functions (these might be used to return a global state of the machine). Next we come up with a primitive set of functions to be used in the program. Again, we need a fitness function and a reproduction operator. The organisms in our population are LISP trees in which every node is a primitive function with as many children as the function has arguments. We further specify that the leaves must all be members of the terminal set. Reproduction occurs by selecting subtrees of the parents and swapping them at the root of the subtree.

3.2 An Example

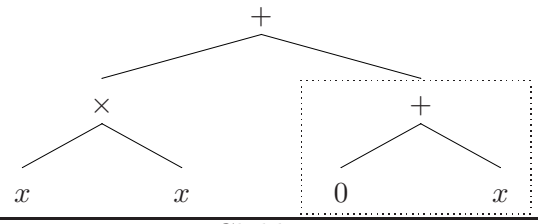
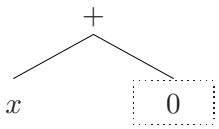
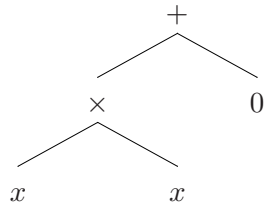
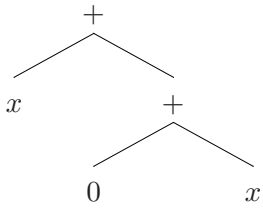
This is best understood by an example. Suppose we wish to evolve a program to calculate $g(x) = x^2+1$. We might implement the following solution:

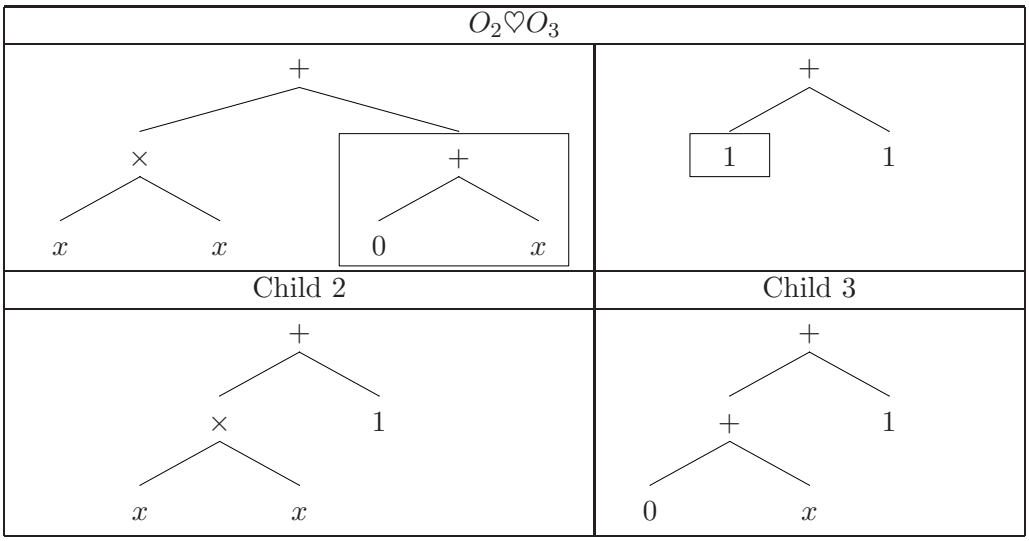
$x^2 + 1$	
Terminal Set	$\{0, 1, x\}$
Primitive Function Set	$\{+, \times\}$
Fitness Function	$f(O) = \int_{-10}^{10} \left\lvert O(x) - (x^2 + 1) \right\rvert dx$
Reproduction Operator	Subtree Splicing (denoted $\heartsuit$)

Notice that here a perfect solution would have a fitness of 0 with worse solutions diverging toward ∞. Continuing with the example suppose that we select the following initial population:

O_0	O_1
	
$f(O_0) = \frac{2060}{3}$	$f(O_1) = \frac{2060}{3}$
O_2	O_3
	
$f(O_2) = 101$	$f(O_3) = \frac{1948}{3}$

Next we go through and make new members for our population by breeding the four above. Because O_2 is substantially more fit than the others it is more likely to be chosen so our random choices might result in the combinations: $O_2 \heartsuit O_0$ $O_2 \heartsuit O_3$ since each reproduction produces two children this will be enough to completely repopulate.

$O_2 \heartsuit O_0$	
	
Child 0	Child 1
	



Now we would iterate this process again except that this time we find that Child 2 has maximum fitness so we know that we've found as good an answer as we can expect this algorithm to find (and it happens to be the best answer anyone's going to find).

3.3 LISP Tree Locality

Again in LISP trees we have the problem of locality. In our example O_2, had a particularly high fitness and we can see that this is because it figured out that the answer was the sum of x^2 and something. However, our algorithm lacks a concept of what makes the organism good. It was just lucky that the algorithm choose to swap the left daughter with 1 and not to make a more destructive swap. Ultimately the problem boils down to the fact that with this encoding we have no way of separating the useless or detrimental parts of an organism's genetic code from the helpful parts.

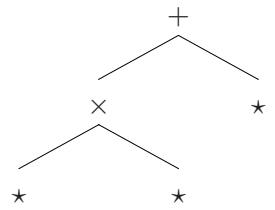
4 A Slightly Different Approach

Our problems stem from the fact that our algorithm attempts to build monolithic solutions; it lacks a concept of building blocks. In the previous example our chances of destroying the building block that made O_2 good were higher than those of making a meaningful improvement. Furthermore, the previous

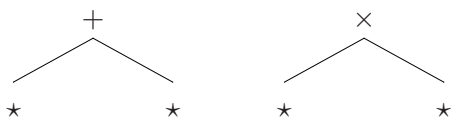
solution was a trivial test case. For bigger problems in which the correct trees are bigger we can expect the probability of making a wrong selection to grow rapidly. The number of possible decisions grow exponentially with the depth of the tree. And so we can see that this approach has trouble with scalability.

4.1 The Encoding

Instead of using the trees of the prior example, which are the equivalent of expressions, we make a slight modification to allow trees with inputs marked as $\star$s, making them equivalent to LISP functions. We'll call these functional-trees. For example:

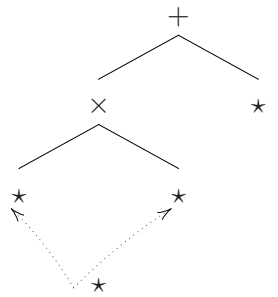


This would be equivalent to the function $f(x_1, x_2, x_3) = (x_1 \times x_2) + x_3$. In this encoding we also allow numeric constants or 0 argument functions as terminals, however this is not the method by which the independent variables are placed. Notice that this notation can also be considered as a schema, if we consider the stars as don't cares, meaning that they could be replaced with any tree. To complete these as schema there is an implied $\star$ at the top of the tree, ie not only can a member of the schema contain arbitrary trees in place of the stars, but it also need only contain the functional-tree as a subtree. Above tree is an example of the schema represented by both of the following trees:



4.2 Reproduction

With this encoding we lose a natural way to evaluate fitness, since we have members of the population that accept the wrong number of arguments. However, reproduction becomes a very graceful procedure. We have the very natural procedure of piping one function's output in as another's input to compose them. This operation is versatile in that it allows us to create a great range of functions. However its most elegant feature is that, if we think of our agents as schema then combining two schema under this operation yields a subset of their intersection, so we can expect to learn more about each. As elegant as this operation is, it has a very noticeable deficiency. When we removed the independent variables as terminals, we made the process of specifying where the variables are inputted harder. Suppose the above example were being used in a population to derive x^2+1; it would be impossible, since there's simply no way to specify that the two branches of the $\times$ node should use the same variable. So along with our pipe operation we add in a pinch operation which combines two input slots into one:



With this the tree is very close to finding a solution.

4.3 Evaluating Fitness

Notice that our trouble with evaluating these organisms is not universal. Some organisms happen to accept the correct number of arguments and can be evaluated in a straightforward manner. In searching for a global way to evaluate organisms of this encodings, we should recall our treatment of these as schema. As such, the question we want to answer about each element is: how well would members of this schema that do accept the correct num-

ber of arguments fare. Now we could begin looking at the members of the schema; of course we can't do this exhaustively since the trees grow without bound, but we always knew that we wouldn't be able to exhaustively search an infinite set so placing an upper bound on tree size is acceptable. But doing exhaustive searches is expensive inelegant and precisely what genetic solutions are supposed to replace. The whole point of Evolutionary Computing is that we have something better than randomly sampling. We have an entire population of better than average subtrees. We can use these as smarter building blocks to do a smarter evaluation.

4.4 A More Elegant Version

sh This process of evaluation, wherein we combine selections of our population in order to evaluate them, can be made just a touch more elegant. Our population itself is composed entirely of combinations of elements each generation building upon the last by combining old to make new. As mentioned some members of our population can be evaluated easily. However these elements are composed of other elements from the population. So we can consider fitness not as an assigned value but as a **currency**, the functions that happen to be evaluable are assigned a fitness but must share a certain amount with their constituent functions. This is much more elegant as we need only evaluate a subset of our population and need not deal with sampling the schema to evaluate. Note that an organism that was born in the nth generation. Is composed of organisms born in the generations prior to generation n so for this system to work organisms need to serve for multiple generations to give them a chance be used in compositions. This property of organisms surviving into later generations is commonly known as **incest**.

5 Computation

Genetic Programming is a very computationally intensive task. Furthermore Genetic Programming is more than theoretically interesting it's an extremely useful technique which can be used as an "invention machine" to rival human invention. As such we're interested in being able to do this fast and on a large scale. The main key to speed is how well this algorithm parallelizes. Parallelizing GP is really a trivial task because it's composed of a number of discrete computations that can be done independently.

6 Conclusion

This modification of LISP tree based GP may seem contrived at first. However it is really a very natural response to the problem of locality. A running theme in EC is that anytime we the implementer relinquish control over an aspect of the encoding and evolution process the evolution takes control of this and thus naturally selects the best way to handle it. This is just a logical next step wherein we stop imposing contrived locations where our organism may crossover their genetic information and instead let them specify these locations for themselves.

References

[1] D E Goldberg, B Korb, K Deb, Messy Genetic Algorithms: Motivation, Analysis and First Results 1989.

[2] R. Poli, Hyperschema Theory for GP with One-Point Crossover, Building Blocks, and Some New Results in GA Theory, 2001.