Tree-based Symbolic Regression Problem

Seth Sorensen and Todd Malone

April 4, 2013

1 Genetic Program

We used a tree representation for our genetic program. We only used one node class, which did not discriminate between being a terminal or a function until asked to evaluate. Children were stored in a list, so we could have functions of any arity.

We included recursive methods to find the size of the tree and the depth. These were included as part of our bloat control, but we did not end up using the tree size method. The depth finder was called in both our mutate and crossover methods, both as a way of limiting tree size and as a way to uniformly distribute where the alterations are performed. We included two mutate methods, mutate and pointMutate. Mutate navigated to a random depth and replaced the subtree at that point with a new random subtree. The semantics of tree generation ensured that random mutations after the selected maximum depth would only generate terminals, so that mutate would not be a source of bloat. Additionally, if any tree grew beyond the maximum depth, mutate could potentially act as bloat control, removing the parts of the tree that exceeded maxDepth.

Point mutate selects nodes in the same manner as mutate, but replaces only the node selected with a function or terminal of the same arity (where terminals are considered arity zero).

Our crossover method takes two trees, then copies them and deals exclusively with the copies. We chose to do this because our crossover is destructive. Once it has made copies, the crossover method selects a node from each in the same manner as mutate and exchanges them. The two copied and altered trees are then returned as a list.

2 Problem

The problem we decided to implement was a simple symbolic regression. We used $10 * \sin((x/5)^2)$ as our test function, in part to have an interesting function. Our test problem has fields for functions and terminals, which are used to generate the tree. A probability variable was used to influence a randomized decision for which would be used when creating a new node. This probability

would be ignored beyond a maximum depth, where the generator would create a terminal node.

In addition to the terminal/function probability, we included a tweak probability. This was used to determine whether mutate or pointMutate would be used for the mutation function. Since all the searchers call "tweak", this seemed an appropriate place.

For our function nodes, we included basic arithmetic operators as well as sine, cosine, and logarithm. Logarithm and division were protected, ensuring that division by zero would return one instead of failing and log of a negative would return one instead of failing.

The test points ran from one to ten, which was part of the reason for $x/5$ in the test problem itself. The terminals we used consisted of a single variable (x), three arbitrary constants, and a random integer generator which generated an integer between zero and fifteen. This constraint was added to avoid random integers appearing in the billions.

We determined the quality of trees by summing the absolute values of the error for each test point, where the error is the difference between the value of test function and the value of the tree. We then inverted the sum to make the problem a maximization, such as all our searchers require.

3 Exploration

We decided to compare the three population-based searchers: the Genetic Algorithm, MuCommaLambda, and MuPlusLamda. We left the Genetic Algorithm as it was, except for using our crossover, but altered the MuLambda searchers to include crossover operations. To do this, we added a step between when parents were selected and when children were generated. The crossovers were performed on pairs of the existing parents, and parents for mutation were selected from the resulting combined list. An implication of the placement of this step is that no pure crossover was ever considered as a candidate solution in the MuCommaLambda searcher.

We ran each of these searchers thirty times on the symbolic regression problem for 4000 iterations each.

4 Summary

As Figure 3 indicates, the Genetic Algorithm was much worse at estimating the test function than either of the MuLambda algorithms. It is possible that it burnt through its iterations faster, as it also appeared to produce smaller trees. The MuLambda algorithms were typically better than the GA, but were fairly inconsistent, as evidenced by the large quality spread seen on Figure 3.

We discovered that our crossover method contains no inherent bloat control. Our mutate method, called more frequently and usually on the trees that had been subjected to crossover, were our only form of control. This caused some

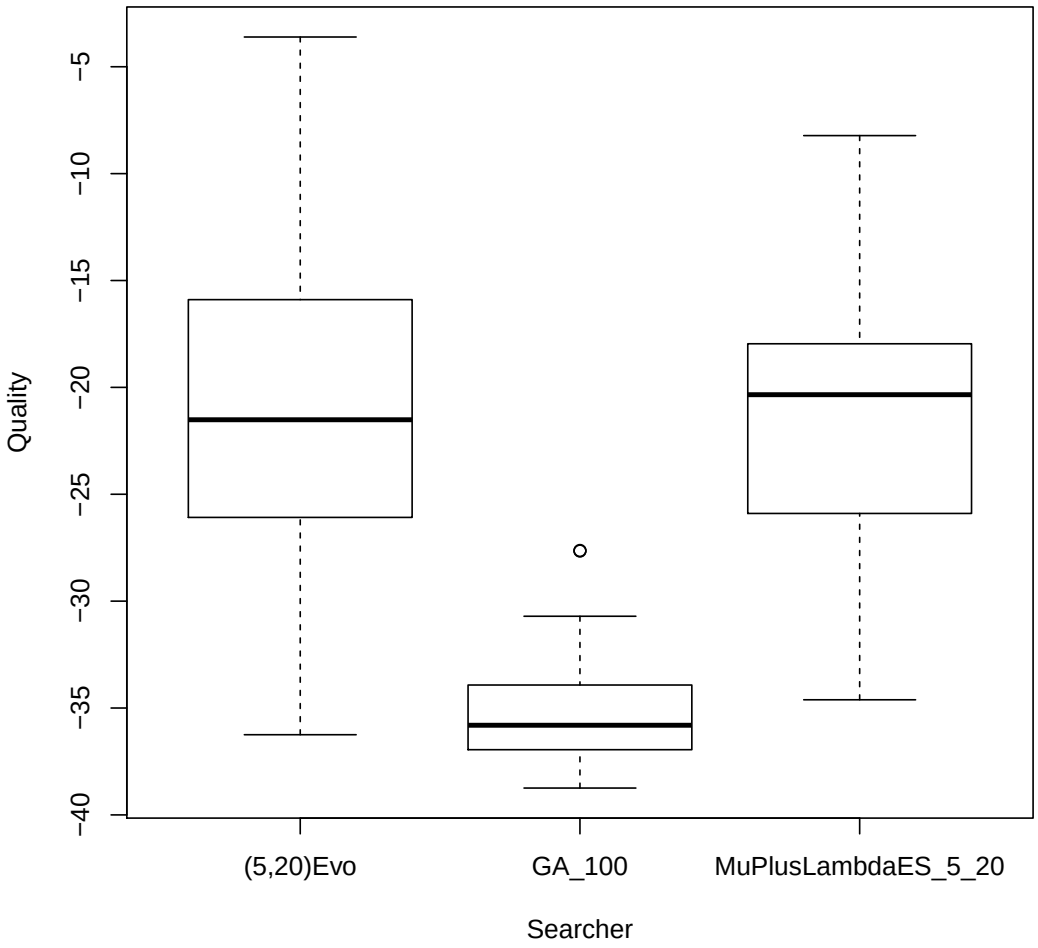

Figure 1: Plot of comparison between MuCommaLambda, MuPlusLambda, and GeneticAlgorithm.

fairly substantial trees to develop in the two MuLambda algorithms, and is something we could fix.

Another route we could have taken that may have been fairly interesting would be to test different values for our terminal and tweak probabilities, and compare the results. What we ended up doing is leaving them both at 0.5,

meaning that during the creation of trees, there was an equal chance of a node being a terminal or a function, and when a tree was tweaked, there was an equal probability that mutate or pointMutate would be performed.