**Abstract:**

*The abstract should be between four and seven sentences long. Introduce the problem you are studying. Describe what you did. Summarize your results --- what did you discover, what is the main take-away message?) Basically, you're trying to sell your paper to the reader, so be brief and to the point.* ***Do not*** *include any citations in the abstract.*

Often described as “the second-best way to solve any problem,” Genetic Programming (GP) relies on Darwinian principles of natural selection to produce models of a system. In this experiment, we applied GP to the problem of Symbolic Regression – that is, we attempted to find the function used to generate given datasets. To accomplish this, we randomly generated a population of symbol trees, each representing a different equation. We then evaluated the “fitness” of each tree and selected the fittest individuals to generate the next generation of trees. In doing this, we saw our algorithm produce a function that approximated data from Generator 1 to within 5x10-15. The resulting equation was 10/(x2 – 6x + 14), leading us to believe that this was, in fact, the equation used in Generator 1. *This is noted later too, but as of the writing of this, our run of the three variable data hasn’t finished running.* From these results we have demonstrated that GP is a viable method of performing Symbolic Regression.

**Intro:**

*In this section, you should introduce the reader to the problem you are attempting to solve. For example, for the first project: describe the 15-puzzle, and why it's interesting as an A.I. problem. You should also cite and briefly describe other related papers that have tackled this problem in the past --- things that came up during the course of your research. In the AAAI style, citations look like\cite{aima} (see the comments in the source file \texttt{intro.tex} to see how this citation was produced). Conclude by summarizing how the remainder of the paper is organized.*

Genetic programming, inspired by Darwinian evolution, uses the principles of natural selection, mutation, and reproduction, to explore a problem space. In this case, we used GP to perform Symbolic Regression and find the function used to generate a dataset. The genetic programming approach is often thought of as a double-edged sword. On one hand, it can be adapted to solve almost any type of problem. On the other hand, it is a fairly uninformed solution method. It uses relatively little information about the problem to solve it, relying instead on guided randomness to deliver a suitable result. This randomness means the path taken to arrive at the solution is hidden to the user. GP may give a supremely accurate result, but yields no insight into why that result works, raising an important question: If you have the answer without knowing why it’s right, have you really learned anything?

In the case of Symbolic Regression, how we arrived at a solution or why we chose certain steps is not as important as the function we produced. According to Koza, GP lends itself well to Symbolic Regression because it is error-driven learning (SOURCE 1). It also finds unorthodox ways of discovering mathematical identities, which allows the use of a very limited set of operations to generate a wide range of functions.

SUMMARIZE THE REMAINDER OF THE PAPER, GODDAMMIT

**Background:**

*Describe any background information that the reader would need to know to understand your work. You do not have to explain algorithms or ideas that we have seen in class. Rather, use this section to describe techniques that you found elsewhere in the course of your research, that you have decided to bring to bear on the problem at hand. Don't go overboard here --- if what you're doing is quite detailed, it's often more helpful to give a sketch of the big ideas of the approaches that you will be using. You can then say something like ``the reader is referred to X for a more in-depth description of...'', and include a citation.*

*Alternately, you may have designed a novel approach for the problem --- your own algorithm or heuristic, say. A description of these would also be placed in this section (use subsections to better organize the content in this case).*

Genetic programming is simply a method for exploring a search space without using any a priori knowledge about the search space. The inherent randomness of this search method in conjunction with the many different parameters that go into the algorithm results in many potential problems that can prevent the algorithm from finding a reasonable solution. We will discuss each of the problems and their potential solutions in the following sections.

Diversity

Maintaining a diverse population is essential to generating a reasonable solution with genetic programming. Without diversity, the algorithm runs the risk losing essential elements of a reasonable solution and converging on a suboptimal local maximum.

Diversity begins with the initial population. Both the size of the initial population and the method for generating an individual are extremely important to maintaining diversity. The size of the initial population must be large enough to be adequately dispersed across the problem space. Standard practice in the field of genetic programming is to have a population size of at least 500, although a larger population is always better (Poli, Langdon, McPhee). We use a population of 1000 in our experiments to ensure good results but also generate results in a reasonable amount of time.

Our individuals in our initial population are generated by random growth. Each node (except for the root node, which is always an operator node) has a 50% chance of becoming either an operator node or a terminal node. Once a node has become either an operator or a terminal node, it will be randomly assigned a value from the appropriate set of operators or terminals. This method is based off the ramped half and half method of generating a population, which populates a population with half full trees and half randomly grown trees (Poli, Langdon, McPhee).

Selection of individuals for a succeeding generation also plays a large role in maintaining diversity. We implemented a very popular method known as tournament selection. This selection scheme randomly selects a subset of the current population and from that subset, picks the best individual to be a member of the next generation (Gupta, Ghafir). We specifically chose this method of selection to prevent populations becoming composed of a very few individuals.

Mutation aids in maintaining diversity by introducing motifs into the population. Often, initial populations will not have all the parts necessary to arrive at an optimal solution, or a generation will lose a necessary part through random selection. Mutation offers a way for populations to recover lost or missing elements of optimal solutions. We used a point-mutation scheme, which could change operator nodes to different operations or alter the value of a terminal node. Point mutations help maintain diversity without introduce problems like overly complex tree structures which can lead to over-fitting of the data.

Overfitting

When implementing a machine learning algorithm such as GP, the experimenter must be wary of overfitting the training data. A model is said to overfit a dataset if it is specific only to those data. This results in a model that performs very well on the given set of training data, but does not generalize to data outside the training set. In the case of symbolic regression, any set of (x, y) pairs can be fit exactly by an arbitrarily complex polynomial. This does not mean, however, that you have found the actual function generating these points, and will give large errors when exposed to (x, y) pairs generated by that same function, but not included in the training set. To avoid this, we took several preventative measures.

Firstly, we implemented the most common technique to avoid overfitting, and split our data into a training set and a test set. We randomly selected 80% of the total data to be our training set, leaving 20% to be the test set. This way, when we ran ten iterations of GP, we had data that each best tree had not seen before. We determined the overall winner by evaluating each best tree on the test set.

Secondly, we implemented a dynamic depth-limiting strategy to prevent overly complex hypotheses.[[1]](#footnote-1) Since the underlying function was human-generated, we decided it was exceedingly unlikely that it be an extremely complex expression. Thus we limited the depth of trees in the population, weeding out overly complex hypotheses. This provided equations that were, on the whole, more generalizable and had similar test error and training error. In addition to limiting the depth, we also reduced the number of operators. This is discussed at length in the Experiment section, and greatly simplified trees.

Crossing over

Crossing over is the driving force behind improvement in genetic programming. Subsets from two individuals that are deemed fit are combined together to generate a new individual. When crossing over two individuals, we randomly select subtrees from each individual and swap them between the two individuals.

**Experiments:**

*In this section, you should describe your experimental setup. What were the questions you were trying to answer? What was the experimental setup (number of trials, parameter settings, etc.)? What were you measuring? You should justify these choices when necessary. The accepted wisdom is that there should be enough detail in this section that I could reproduce your work* ***exactly*** *if I were so motivated.*

To perform Symbolic Regression with genetic programming on the data from Generator 1, we began with a population of 1,000 randomly generated expression trees. The interior nodes of the trees were randomly selected from the set of valid operators, {+, \*, /}. The leaf nodes of the trees were randomly selected from the set of valid terminals, consisting of the variable x and integer constants from -5 to 5. In order to reduce the number of constant trees (that is, trees that evaluated to a constant function), we chose that the variable x would be chosen with 50% probability and a random choice from the set of integer constants would be chosen the remainder of the time. The problem at hand stated that the underlying function in Generator 1 used the operations +, -, \*, /, and integer powers of x, but as a design consideration we chose to omit the operations – and exponentiation. We were able to do so because integer powers of x can be expressed through repeated multiplication, and subtraction is the addition of negative numbers, which are valid as terminals. Omitting these two operations greatly reduced the complexity of the trees produced by the algorithm, making it far more likely to produce an equation that would generalize well.

To make a random tree, we began with a random operator node at the root and a predetermined depth limit of 10. We then built the rest of the tree recursively. Each node generated had a 50% chance of being a random terminal and ending that branch of the tree, or being a random operator. If the tree ever reached the depth limit, all new nodes were made to be terminals.

Once the initial population was generated, we applied the genetic programming algorithm. We allowed the algorithm to run for 35 generations, with the possibility of early termination if any tree’s error was less than 0.2. To calculate error, we accumulated the absolute value of the difference between the tree’s evaluation of a given x and the actual value, f(x). We chose this method over the least squares method because, like least squares, it never allows for negative error, but it also limits the possibility of the total error getting too large and causing an overflow.

We use three different methods to generate a new generation. Firstly, the fittest 10% of the population are duplicated into the next generation. This technique, called reproduction, ensures that every generation is at least as good as the preceding generation. **SPENCER CITE THIS.** Secondly, we mutated 10% of the population. When a tree was selected for mutation, we traversed the tree, changing the current node 40% of the time. Terminals were only ever changed to other terminals, and operators to other operators. We initially mutated a node with 20% frequency, but in order to increase the diversity in our population, we increased the mutation rate to 40%. **CITE THIS ONE TOO.** Mutation occurred in place, so once a tree was mutated, it was still available to be picked for crossover. In another effort to keep diversity high, we used a tournament selection algorithm to select individuals for crossover. In tournament selection, we randomly select 10% of the population and choose the fittest individual of those 10% (the winner of the tournament). This probabilistically chose fitter individuals more frequently, but left open the possibility of a relatively unfit individual being chosen. When crossing over, we randomly selected a node in each tree and swapped the subtrees rooted at the nodes. To prevent overfitting, if the result of a crossover was too deep, we removed that individual and injected a new, random individual into the population. This limits the effects of overfitting by removing overly complex hypotheses and also contributed to diversity by adding random individuals periodically. We implemented a dynamic depth limit, where we defined “too deep” as deeper than the fittest individual in the generation. We then continued crossing two trees until the new generation had the same number of individuals as the previous.

When evaluating the trees, we split the initial dataset into a training set, containing roughly 80% of the data, and a test set containing roughly 20%. To divide the data, we generated a random number for each x value and assigned it and its corresponding f(x) value to either test or training depending on the value of the random number.

After 35 generations, or a tree’s error dropped below 0.2, we selected the fittest tree to be saved for comparison against the test set. We ran this process ten times to produce ten different winners. The final answer was the winner who performed best over the test data, giving us the underlying equation behind the dataset.

**Results:**

*Present the results of your experiments. Simply presenting the data is insufficient! You need to analyze your results. What did you discover? What is interesting about your results? Were the results what you expected? Use appropriate visualizations. Prefer graphs and charts to tables as they are easier to read (though tables are often more compact, and can be a better choice if you're squeezed for space). Always include information that conveys the uncertainty in your measurements: mean statistics should be plotted with error bars, or reported in tables with a range. The 95%-confidence interval is a commonly reported statistic.*

NOTE: At the time of this writing, we are still waiting on a few trials to run, including all data for the function of three variables. This will be updated over the weekend, thanks Dr. R for making this a first draft.

To evaluate our data, we took a subset of the Generator 1 data and split it into training and test sets. We noticed that the data varied very little outside the range [-50, 50], so we considered x-values from [-50,50] in increments of 0.1, and their corresponding y-values.

Our algorithm produced spectacular results, finding an equation for the Generator 1 data with less than 5 x 10-15 total error on the test set. When evolving, this tree took 19 generations to be formed. After 19 generations, the equation 10/(x2 – 6x + 14) **SPENCER HELP ME OUT HERE WHAT WAS IT** had a total error of less than 2x10-14. When evaluated on the test set, as stated above, the error was similarly low. This indicates that the equation generalizes well and was not overfit to the training data.

After ten runs of GP, we had ten trees that were each the best in that iteration of the algorithm. On average, one of these winning trees had an error of 31.27 on the test set with a standard deviation of 22.34. When considered in the scope of the size of the test set, these results are quite impressive. An error of 31 means that over the 200 datapoints in the test set, the derived function generated values that were on average 0.15 off from the actual values.

**Conclusion:**

*In this section, briefly summarize your paper --- what problem did you start out to study, and what did you find? What is the key result / take-away message? It's also traditional to suggest one or two avenues for further work, but this is optional.*

We began with the goal of developing a program that could estimate a function using Symbolic Regression analysis for two different datasets. We used standard genetic programming practices to evolve random functions into reasonable estimates of a function for each dataset. The most notable aspects of our program are: the initial population size of [fill in], reproduction rate of 10%, a point mutation rate of 40% in 10% of each population, and random crossing over between individuals selected by tournament selection.

We inconsistently discovered a function for the first data set with a nearly non-existent margin of error on both the training set and the test set, however, we were unable to generate any reasonable estimates at a function for the second set of data. Given our results, we believe our program is only capable of estimating single variable functions with any reliable accuracy.

In future experiments, a larger generation size and a greater number of generations should be used to evolve a solution. Additionally, a secondary selection criterion similar to the order of nonlinearity selection proposed by Vladislavleva et al would be beneficial for more accurate estimations, especially for multivariable functions (Vladislavleva).

**References:**

**Genetic Programming: On the Programming of Computers by Means of Natural ...**

 By John R. Koza

1. http://digitalcommons.wpi.edu/cgi/viewcontent.cgi?article=1134&context=computerscience-pubs [↑](#footnote-ref-1)