Genetic Programming for Julia: fast performance and parallel island model implementation

Morgan R. Frank

November 30, 2015

Abstract

I introduce a Julia implementation for genetic programming (GP), which is an evolutionary algorithm that evolves models as syntax trees. While some abstract high-level genetic algorithm packages, such as *GeneticAlgorithms.jl*, already exist for Julia, this package is not optimized for genetic programming, and I provide a relatively fast implementation here by utilizing the low-level *Expr* Julia type. The resulting GP implementation has a simple programmatic interface that provides ample access to the parameters controlling the evolution. Finally, I provide the option for the GP to run in parallel using the highly scalable "island model" for genetic algorithms, which has been shown to improve search results in a variety of genetic algorithms by maintaining solution diversity and explorative dynamics across the global population of solutions.

1 Introduction

Evolving human beings and other complex organisms from single-cell bacteria is indeed a daunting task, yet biological evolution has been able to provide a variety of solutions to the problem. Evolutionary algorithms [1–3] refers to a field of algorithms that solve problems by mimicking the structure and dynamics of genetic evolution. In particular, evolutionary algorithms typically model solutions to a given problem using some data structure, which can be treated as a genotype or gene. These algorithms begin with a collection, or "population", of these data structures and iteratively generate new populations, or "generations", by implementing genetic mutation and crossover on good solutions in the existing population. Good solutions are often referred to as being "fit" if they reproduce a target signal; therefore, we typically measure fitness by measuring how well a solution minimizes error. The fitness of a given solution can be thought of as a phenotype.

Here, I focus on a particular type of evolutionary algorithm called "genetic programming" (GP). In general, genetic programming evolves syntax trees that represent models, which aim to reproduce a target signal given input signals. In theory, these syntax trees can represent any program, but we will restrict ourselves to the symbolic regression problem here. Therefore, our syntax trees will represent equations, which will act on numeric input variables and reproduce a numeric output variable. Some powerful third-party software packages exist, such as Eureqa [4], but these packages are not open-source and can be inflexible when attempting to handle diverse problems. Julia's easy vectorization, access to low-level data types, and user-friendly parallel implementation make it an ideal language for developing a GP for symbolic regression.

Finally, I will discuss a powerful and widely used method for utilizing multiple computing cores to improve solution discovery using evolutionary algorithms. Since many languages implement multi-processing, rather than multi-threading, naive methods seeking to improve the runtime of evolutionary algorithms (e.g. evaluate the fitness of population members in parallel), are often relatively fruitless due to the communication overhead required to bring the population back to a single process for essential steps in the algorithm iteration, such as genetic crossover. Furthermore, evolutionary algorithms that iterate with a single population are susceptible to convergence on a "local optima" solution, rather than seeking a more desirable "global optima" through increased exploration. The island model [5, 6] is a useful paradigm for parallel computation in evolutionary algorithms that largely overcomes both concerns by iterating an independent population of solutions on each available core. Since each population is initialized randomly, it is possible that individual populations will independently explore solution space and possibly converge on different local optima. The last piece is to implement occasional "migration" between the populations, where good solutions from one population are removed and sent to a different population running on a different parallel process. Migration

seeks to encourage the overall GP to continue exploring around the best solutions, rather than having populations converging too quickly to locally optimal solutions. Furthermore, migration is the only step in the island model that requires communication between processes during runtime, but migration is infrequently performed and only a few solutions are communicated at each migration step. In summary, we see that the island model provides a highly scalable means to improve overall performance of evolutionary algorithms, rather than simply improving algorithmic runtime.

2 GP implementation in Julia

I employ an object-oriented implementation of GP in Julia by defining four key types that are essential to running the GP. The **Tree** type will be used to represent the solution models as syntax trees. At the root of each Tree is a Julia *Expr*, which is the literal representation of the equation which we hope reproduces the target variable from input variables. The *Expr* type is a powerful low-level Julia type that is perfect for this task because of fast evaluation and easy manipulation. In fact, the Julia metaprogramming page highlights that Julia expressions are first parsed into Expr types before being possibly compiled and then evaluated. In addition to the Expr root, Trees have property fields for the age (ie. the number of generations the tree has survived unchanged), the depth (ie. the maximum tree depth of the syntax tree), the number of nodes in the syntax tree, and the fitness value of the syntax tree. Furthermore, Tree types have methods for copying themselves, listing their nodes in evaluation order, a toString method, an equals method, and a method for determining if the tree is better than another tree (if so, then we would say the tree "dominates" the other tree, but more on that below).

I define a **gpLibrary** type as a place to store the functions and terminals with which to construct Trees. Terminal values are numbers, called "constants", or variables, and they represent entities that can be used as leaves in the syntax trees. Functions are added to the library along with the number of input variables they require. These functions are available to be used as non-leaf nodes in the syntax trees, and so the number of inputs for each function determines the number of children a node will have based on the function the node represents. Beyond storing these entities, gpLibrary types also contain some nice functionality for randomly accessing stored Terminals and Functions. Deciding which functions and which constants to include in the library is problem dependent. For example, if you know your target signal is periodic, then perhaps you should include trigonometric functions in your library; otherwise, maybe you should omit trig functions if your signal is not periodic but is instead quite noisy as the trig functions might be used to overfit the noise in the target signal. My implementation of GP is restricted to using only the terminals stored in the library, but future iterations of this project would instead use only parameters and variables as terminals for the syntax trees, along with occasional parametric optimization for each syntax tree through least-squares regression or another evolutionary algorithm.

The **Population** type represents an independent GP population, which iterates over time to form new generations of solutions. Populations store a population as a list of Trees. The number of Trees to maintain in the Population is given as the *popSize* property when the Population is initialized. The subset of these Trees representing "good" solutions are stored in a separate list as well. We will see that "good" Trees define a Pareto front. In this context, "good" trees are trees that are young, simple, and reproduce the output signal. For measuring the latter, a fitness function must be provided to the Population which defines how to measure the error between the syntax tree and the target signal; for example, you might use absolute error or root-mean-squared error as a fitness function. The GP will attempt to find Trees that, in part, minimize the fitness function.

There are several ways one might define the simplicity of a syntax tree, but for our purpose we can assume that Trees with fewer nodes will be easier to interpret in the context of the problem being solved by the GP. The type of crossover typically employed in genetic programming (discussed below) tends to produce new syntax trees that are deeper than their parent trees; this tendency towards larger and more complicated trees is known as "bloat". Allowing bloat to go unchecked would result in only complicated solutions and eventually even effect the runtime of the GP algorithm. Note that bloat is not always a bad thing, as the genetic material contained therein may lead to useful combinations of library variables for a future Tree. In any case, implementing selection pressure for simplicity should help combat bloat. Future iterations of this project might allow users to assign complexity values to each function in the gpLibrary which represents the cost of executing that function or interpreting that function in the context of the larger problem.

Finally, we want "young" Trees, or specifically Trees that have not been around for many generations, to allow new solutions a chance to develop into potentially good solutions before being dominated by existing Trees. This consideration helps to maintain some exploration dynam-

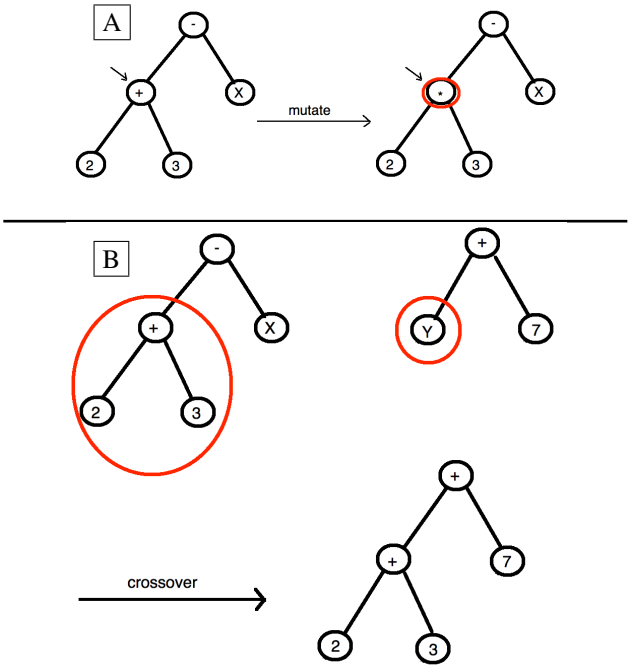

Figure 1: **(A)** An example of syntax tree mutation. **(B)** An example of syntax tree crossover.

ics in the Population, rather than quickly converging to the first decent solution it finds. Considering these three dimensions for measuring the "goodness" of a syntax tree, we will say that a tree is "non-dominated" if no other tree in the Population has lower fitness and lower complexity and is younger. Non-dominated trees define a "Pareto Front", which the Population stores. Note that in reporting the Pareto front, I will collapse the age dimension, as this dimension exists purely for maintaining exploration. All of the current fittest and simplest solutions are usually of most interest at any given time during the Population iteration. I assume that all trees in the Pareto front at a given time are of equal "goodness" as solutions for the purpose of the evolution.

The next generation of a Population is given initially by the members of the Pareto front of the previous generation. Next, *babyPopSize* new Trees are generated at random and added to the new generation. *babyPopSize* should be relatively small compared to *popSize*. This measure aims to maintain some solution exploration. Additional Trees are created in the new generation by randomly selecting individual Trees from the Pareto front with probability *mutationRate* for mutation, or, with probability 1-*mutationRate*, selecting pairs of Trees from the Parteo front at random which are used to produce a new Tree using crossover. To mutate a Tree, we first randomly select a node. If that node is a leaf, then the variable stored in that node is changed to a randomly selected terminal from the library. Otherwise, the variable stored in the node is changed to a randomly selected function from the library. In general, mutation provides an exploration dynamic. Crossover is implemented using two Trees by randomly selecting subtrees in each syntax tree and swapping the subtrees. In general, crossover provides an exploitative dynamic and tends to drive the population to converge on a single solution.

Finally, a **GP** type is used to control the global properties and dynamics of the algorithm, and allows exploration of the resulting solutions. If the GP algorithm is run serially, then the GP type is really a wrapper for the Population type. Otherwise, the GP type is responsible for initializing the independent Populations on each available process, and for organizing the communication between them. In particular, the GP type will implement migration when running in parallel. Migration should occur infrequently and serves to maintain some exploration among all of the Populations by sharing good solutions between Populations in an effort to dissuade premature convergence to local optima. Each Population has a target Population, which it sends a small number of randomly selected Trees from its local Pareto front, and a source Population, from which good Trees are received and added to the local Pareto front for at least the next generation, such that the Populations form a cyclic network. The *migrate* property of the GP type determines how many independent Population iterations should occur between migrations.

3 Sample Runtime Results:

We define our input variable X as $[0:50:2\pi]$, and our target variable $Y = (X-1)\sin(2X)$. Let's assume X and Y represent two variables for which we have observational data. Our task is to produce a model that expresses Y as a function of X. Naturally, we turn to genetic programming in Julia.

We use the @everywhere macro to make sure the GeneticProgramming.jl file is included on all processes, and to make sure that X and Y are defined on all processes. Next, we will define a gpLibrary, which contains the ingredients with which to construct syntax trees. We only need to define the library locally; the GP type will handle the distribution of the library if running in parallel.

```
# create a library of nodes that we can use to build
# syntax trees. Deciding which functions and terminals
# to include is problem dependent.
library = gpLibrary()
for i in -10:10
    library.addTerminal(i);
end
# add input variable as a terminal.
library.addTerminal(X);
library.addFunction(+,2);
library.addFunction(-,2);
library.addFunction(*,2);
library.addFunction(/,2);
library.addTerminal(pi);
library.addFunction(sin,1);
library.addFunction(cos,1);
```

Next, we must decide on a fitness function to minimize. The fitness function only needs to measure the error between a solution and the target signal. We will opt to use the root-mean-squared error here.

```
# define a fitness function. In this case, we will
# use RMSE.
fitFunct = function(treeOutput)
    sqrt(mean((treeOutput-Y).^2))
end
```

Now we initialize a GP instance to run in serial with a 50% mutation rate and a population of size 200. We then run the GP for 1000 generations and look at the resulting Pareto front (note: the age dimension is collapsed).

```
serial_G = GP(library,fitFunct,
                vars=Dict("X"=>X,"pi"=>pi),islandCount=1,
                mutate=.5,popSize=200);
```

```
@time serial_G.run(1000);
```

```
 18.821979 seconds (33.90 M allocations: 1.089 GB, 1.5
5% gc time)
```

Alternatively, we can make use of the parallel island model GP by initializing a separate GP instance with multiple islands.

```
addprocs(3)
```

```
3-element Array{Int64,1}:
 2
 3
 4
```

```
parallel_G = GP(library,fitFunct,
                vars=Dict("X"=>X,"pi"=>pi),
                islandCount=nprocs(),mutate=.5,
                popSize=200);
```

```
@time parallel_G.run(1000);
```

```
100
200
300
400
500
600
700
800
900
1000
 37.242025 seconds (37.69 M allocations: 1.273 GB, 1.1
4% gc time)
```

Notice that the runtime is actually slower than the serial run despite iterating the same number of generations and using populations of the same size. However, the slight runtime penalty is to be expected because of migration among the populations. Recall also that the goal of the parallel island model is not to simply improve runtime, but to improve the overall performance of the solution search by utilizing parallel processing without incurring too great of a runtime penalty.

Figure 2 compares the resulting populations and Pareto fronts from the serial GP and the parallel island model GP. We see that the island model GP produces a denser Pareto front, indicating that more exploration was performed by the island model GP. Furthermore, the island model GP found solutions with lower fitness (ie. better fit to the target signal). We can see this again when we compare the fittest solutions from the serial GP to the fittest solution from the island model GP. However, we should note that usually the most useful solutions are located at the apex of the Pareto front, rather than at the ends, since this solution should have a good blend of fitness and simplicity.

4 Future Work

Of course there are several ways to improve the work I have done. The research field around genetic programming, and evolutionary algorithms on a broader scale, is very active, and new ideas and implementation details are emerging regularly. Additionally, it should be unsurprising that problem specific optimizations arise as well. Here are some features I would add to my GP implementation in Julia:

- modern GP implementations for symbolic regression typically only allow parameters and variables as terminals for the syntax trees. A parameter optimization algorithm is run on each syntax tree every few generations. This additional step slows down the GP algorithm a little bit, but it allows the GP to focus on optimizing functional form instead of parameter fitting.

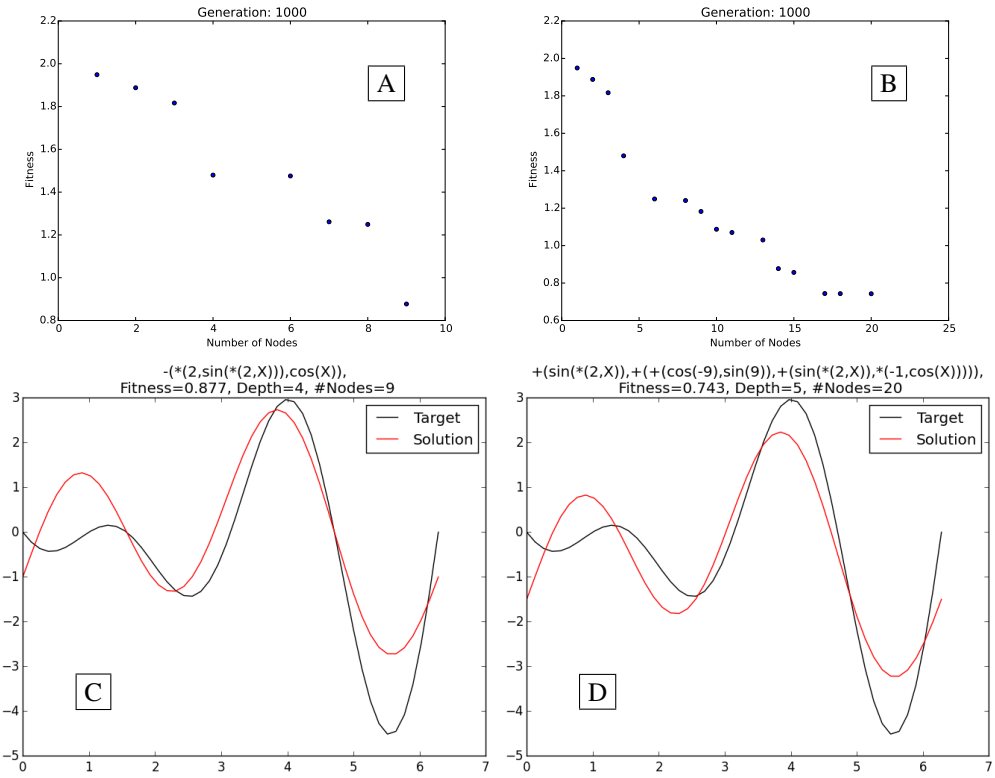

Figure 2: A comparison of serial GP to parallel island model GP results. **(A)** The final Pareto front from the serial GP. **(B)** The final global Pareto front from the island model GP. **(C)** The solution with the lowest fitness from the serial GP. **(D)** The solution with the lowest fitness from the island model GP.

- a tactic to combat tree bloat is to occasionally condense the syntax trees. This means taking a syntax tree and replacing subtrees with the value that the subtree evaluates to. Typically, one only does this for subtrees not containing a variable as a leaf node.

- GPs can theoretically be used for a wide variety of problems beyond symbolic regression. For example, GP have been used in research for automatic program debugging [7, 8]. Generally speaking, we could imagine using GPs to address problems with data structure as terminals and any function, such as for-loops or if-else statements, as the variables stored in non-leaf nodes. It would be interesting to add this capability to the genetic programming Julia library for accomplishing tasks like evolving a sorting algorithm or code debugging.

- another way to measure the complexity of a syntax tree is to assess how expensive it is to evaluate each function it contains, or perhaps measure the difficulty of interpreting a function in the context of the problem. Therefore, one nice addition would be the option for the user to define a cost for each function in the library which would be used to calculate the complexity of the syntax trees.

References

[1] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[2] Guohua Wu, Witold Pedrycz, PN Suganthan, and Rammohan Mallipeddi. A variable reduction strategy for evolutionary algorithms handling equality constraints. *Applied Soft Computing*, 37:774–786, 2015.

[3] Catherine A Bliss, Morgan R Frank, Christopher M Danforth, and Peter Sheridan Dodds. An evolutionary algorithm approach to link prediction in dynamic

social networks. *Journal of Computational Science*, 5(5):750–764, 2014.

[4] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

[5] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–48, 1999.

[6] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, and Qingfu Zhang. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 2015.

[7] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[8] John R Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In *ICGA*, pages 37–44. Citeseer, 1991.