Genetic Programming using Haskell

Design of Bioinspired Algorithms

Daniel Jurjo

January 2, 2021

1 Basic elements and operations

In this project I implemented the genetic algorithm developed by Alonso et al. [1]. I also added some small modifications in the algorithm that I feel that can improve it or, at least, made the coding easier.

The basic objects and functions we are going to use are defined in the file *Expressions.hs* as the code that I am going to comment.

First we defined the following data structures and types:

```
type Literal = Float

newtype Variable = V String
  deriving (Eq, Show)

instance Ord(Variable) where
  V var1 <= V var2 = (read (tail var1)::Int) <= (
      read (tail var2)::Int)

data Terminal = Lit Literal | Var Variable
              deriving (Eq, Ord, Show)

data Expression =
               Add Terminal Terminal
              | Sub Terminal Terminal
              | Prod Terminal Terminal
              | Div Terminal Terminal
              deriving (Eq)
```

Notice that we are restrictin variables to be strings as `u15` or `x9` in order to be able to define a order relation between them. This order is needed because we are going to use the `Data.Map` module to define slps, states and compute the semantic function. We are going to use only $\{+, -, *, //\}$.

We can add another operations just adding new constructors to `Expression` and adding the needed pattern-matchs to some functions. This way of working allow us to easily add new operations and modify their behaviour fast and avoiding adding faults in our code.

Given this basic definitions, we define the slps as a type:

```
type SLP = M.Map Variable Expression
```

Given a slp, we can compute its *effective code*. We can only compute the effective variables or the effective slp related to it. In the examples of [1][p.762. Section 2.2.] They rename the variables after the computation, I decide to let the variables without changes because it will make easier the computation of the semantic function and also the recombinations.

We can check the computation (...)

The semantic function is evaluated following the given algorithm in the paper. Instead of considering a vector of values, we use a state.

2 Fitness, Mutation and recombination

Fitness function is defined computing the mean square error between the semantic of a given slp and the expected values. This function is defined in the file `fitness.hs`.
Following the procedures given in the paper we defined the mutation in `mutation.hs` and the *slp-recombination* in `recombination.hs`. We can compute examples :

3 The main algorithm

The genetic algorithm presented in the paper is defined as follows

```
generate a random initial population
evaluate the individuals
while (not termination condition) do
    for i = 1 to Population_size do
        Op := random value in [0, 1]
        if (Op < Prob_cross)
        then do crossover
        else
        if (Op < Prob_cross +  Prob_mut)
        then do mutation
        else
        if (Op < Prob_cross + Prob_mut + Prob_repr)
        then do reproduction
        evaluate new individuals
        insert New_Pop
    update population with New_pop
```

However I decided to slightly modify the algorithm. My proposal is the next:

```
generate initail population #Evaluation done at the same time
define max_it
while (current_it < max_it) do
    for i = 1 to Population_size do
        Op := random value in [0, 1]
        if (Op < Prob_cross)
        then do crossover
        else
        if (Op < Prob_cross +  Prob_mut)
        then do mutation
```

With this approach the population is being modified during each iteration. In the case of recombination as we need two individuals one is taken at random and no other operation is applied over it. So the number of operations performed in one iteration over a population of size n is smaller than n. If the random individual where used again then the number of operations would be n. In both cases considering that `Prob_cross` + `Prob_mut` = 1.

4 Some experiments

4.1 A naïve first experiment

First we run a basic experiment. Our target is $f(x, y) = x + y + 1$ we set 50 random points in the $[-25, 25]$ interval. We will consider also (Prob_cross, Prob_mut) = (90, 10), slps of maximum size 10, 20 iteratiions and 50 individuals in the population.

I ran this experiment twice and obtained:

- The first one. In 117.445633 seconds and with an error of 0.9999998. It obtained $x + y$
- The second time. In 221.32046 seconds and with an error of $5.2452087e{-}8$. It obtained $(1 - x//(1 - x) - (1 - x)) + y + x$.

Notice that in the first case the error should be 1 and in the second one 0. This is because the input file are generated with python and the numbers are "different" in python and haskell.

4.2 A small polynom

With the same set up as before we are going to run some experiments with target $f(x) = x^3 + x^2 + x + 1$

I ran this experiment twice and obtained:

- The first one. In 248.605806 seconds and with an error of 14.11515. It obtained $x * (x * x) - (1 - (x * x))$.
- The second time. In 232.965591 seconds we obtain a really bad solution, $x * (x//(1//x))$. This has an error of 232.965591.

- A third execution give us took 192.540698 seconds and we obtain $(x+1)*x*x$ with an error of 13.1151.

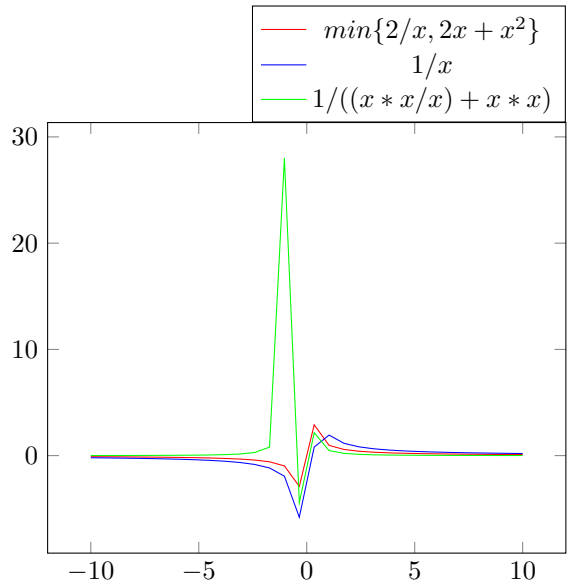
Notice that the first ran give us a quite good aproximation x^3+x^2+1. It is not the best but it only make 20 iterations. In the other hand the second run give us only x^3 . The third experiment gave us x^3+x^2 and it is worse than the solution we obtain in the first try but the error obtained is smaller! This is probably because we get a small set of sample points.

4.3 More difficult one

We are going to use the same setup and this time the target will be $min\{2/x\,, 2x+x^2\}$

In this case we obtain the following results:

- In the first run we obtain $1/x$ in 81.561111 seconds. At first sight seems to be a bad solution but the error is just $5.9257936e-2$
- In the second run we obtain $1*(1/x)$ in 151.831951 seconds with an error of $5.9257936e-2$
- In the third one we obtain $1/((x*x/x)+x*x)$ with an error of $2.4695702e{-}2$ in 271.252447 seconds

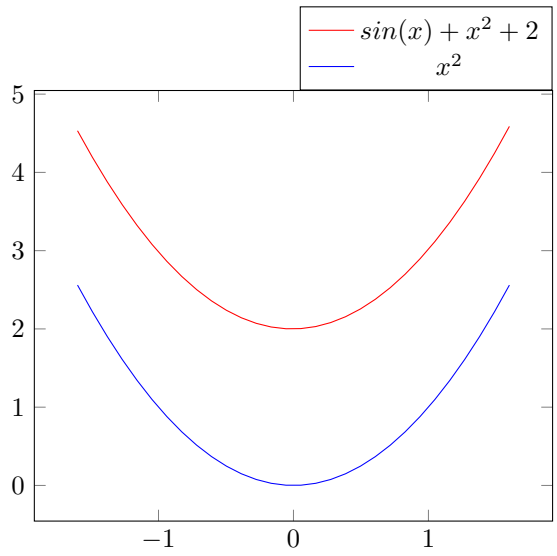


Notice that the third aproximation is quite good everywhere but in one point ($x=-1$) where it explodes.

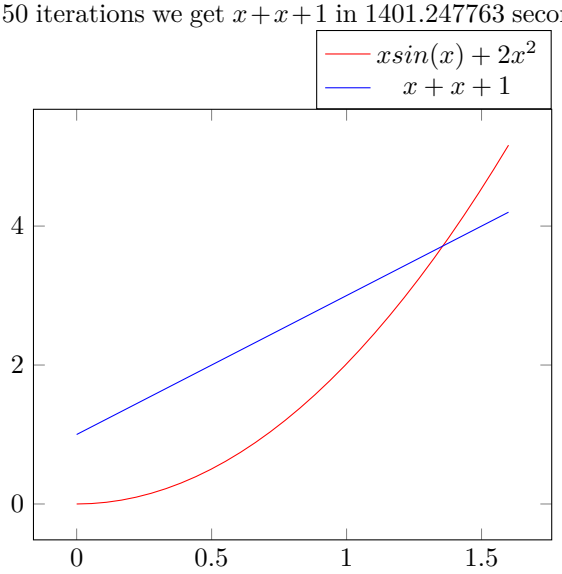
4.4 The hardest one

The last function we are going to consider is $x*sin(x)+2*x^2$.The set up will be the same but the points will be taken in the interval $[-\pi/2, \pi/2]$. Since we are going to use slps of size 10 we will not obtain a good aproximation (as a Taylor's serie for example). Anyway we can hope to have a more or less good aproximation.

- In the first execution we obtained $(1 * x) * (1 * x//1)//(1 * 1 * 1 * 1) = x^2$ in 316.126703 seconds with an error of 2.027752



Anyway if we execute with 50 iterations we get $x+x+1$ in 1401.247763 seconds and an error of 5.642695e-2



5 Conclusions and future work

We were able to present a first implementation of a genetic algorithm based on Straigh Line Programming. In this first approach we found out that the algorithm behaves more or less well and it is so easy to extend with new functions when it is needed. Anyway the algorithm cost seems to be too high (since we do not know how much time took the experiments presented in the paper). The cost of the algorithm seems to be O(???).
It could be possible to reduce the time cost of the algorithm trying a slightly

different approach and choosing more suitable data structures. Functional programs are very suitables when parallelizing and adding parallelism could boost the speed of the algorithm. In the paper the reproduction appear as a asexual behaviour, could be interesting considering it as a sexual one mixing it with the recombination. The paper also showed that slp-recombintions works better than the typicall ones, we can also consider to mix some types of recombinations in order to add more variability at our algorithm.
Could be also interesting to allow the slps to have different sizes to allow the algorithm to reach "alone" better aproximations. Anyway all of this approaches require the algorithm to reduce its costness since bigger slps and more iterations will be needed to test more complex functions.

References

[1] César Alonso, José Montaña, Jorge Puente Peinador, and Cruz Borges. A new linear genetic programming approach based on straight line programs: Some theoretical and experimental aspects. *International Journal on Artificial Intelligence Tools*, 18:757–781, 10 2009.