Genetic Programming with Clojure

Jonatan Ferm
Lunds Tekniska Hogskola
EDAN50 Intelligent System
jonatan.ferm@gmail.com

Ludwig Jacobsson
Lunds Tekniska Hogskola
EDAN50 Intelligent Systems
ludjac@gmail.com

Abstract

This document describes the basics of genetic programming in LISP. First the theoretic foundations for the algorithm is described then we move on to an implementation for approximating a mathematical function. Further on is a description of how to breed heuristic functions for the board game Reversi using the same algorithm with some alterations.

Keywords: Genetic programming, generating random code, natural selection, crossover, super heuristics, reversi, Clojure, LISP.

1 Credits

This paper is the result of a project in the course *Intelligent Systems EDAN50* at the Faculty of Engineering, Lund University.

2 Introduction

On the 24th of November 1859 Charles Darwin published *On the Origin of Species*, a pioneering view on how we and our surroundings came to be what we now are. He introduced the term **natural selection** (Darwin, 1859) in order to explain and describe how the fittest individuals shaped the generations to come. Genetic crossover, sharing of genome between two individuals, between the fittest individuals gave the next generation a better adaptation to the environment. Over thousands of generations this led to species well adapted to their environment. Species excelling in solving the problems apparent in the setting they live. Giraffes evolving long necks to reach food where other animals cant, the Garter Snake's resistance to a toxin in their pray, Primates adapting to the savanna began standing up to lookout for pray and enemies. This theory was later widely accepted as the theory of our evolution.

In the mid to late 1900's when computing started to gain major ground, research in new ways to solve optimization problems led to the introduction of Darwin's ideas on natural selection in the computer science field. Between the 1960's and 1990's multiple ways of incorporating natural selection and evolution to produce solutions to mostly optimization problems were presented, for example: **Evolution strategies** (Schwefel, 1974) (Rechenberg, 1973), **Genetic Algorithms** (Holland, 1975). John Koza presented in (1992) a way of evolving computer programs by using and altering abstract syntax trees. By using this method the hope was to widen the application area of the evolutionary algorithms. Increasing computing power led to new problems being solvable by these methods. Evolutionary algorithms is the common name for all these methods and algorithms.

With a population of random programs with functions that work on the problem domain and the possibility of measuring the fitness of each individual one can apply natural selection to the population and thereby evolve the population to fit the domain better for each generation. Genetic programming is specially useful when one wants to find the solution of a problem in an area that has a small theoretic foundation. By using natural selection the programs will adapt themselves to the environment without any need for us to have specific knowledge about the process or setting.

Human-competitive results have been produced in a multitude of fields (Koza, 2010) including quantum computing circuits, analog electrical circuits, antennas, mechanical systems, controllers,

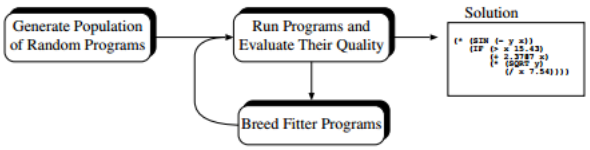

Figure 1: The basic flow of genetic programs.

game playing, nite algebras, photonic systems, image recognition, optical lens systems, mathematical algorithms, cellular automata rules, bioinformatics, sorting networks, robotics, assembly code generation, software repair, scheduling, communication protocols, symbolic regression, reverse engineering, and empirical model discovery.

2.1 Clojure

The choice of LISP is as you will see an obvious one when dealing with genetic programming, the choice of the LISP dialect Clojure is on the other hand not self explanatory. We considered the fact that Clojure compiles to Java Byte Code an important aspect since the Java platform is one of the most common in today's computer science. Clojure is also a relative new dialect and has gained a lot of ground recently and has a vibrant and active community. These factors makes us believe that Clojure will be a LISP dialect for the future.

3 Theory

We will start by going through the theory for genetic programming and then move on to an analysis of our actual results. In order to prevent any confusion regarding *programs* and *individuals* we hereby declare that they will be used as synonyms throughout the paper.

3.1 Genetic Programming

The basic routine for genetic program follows the natural selection of Darwin (1859). Initialize population with random individuals. Measure their fitness and let the fittest individuals breed offsprings to the next generation. Iterate this procedure until any of the individuals produce a satisfactory result.

3.2 Initialization

When generating a population of random programs the first problem is to define the problem domain and to build a set of functions to work on that domain in order for the program to have a toolbox of functions to apply. When defining this we introduce a restriction on what kind of solutions the algorithm will be able to produce. Only combinations of these functions will be available to the individuals. With these created we randomly pick a combination of these and put them in an abstract syntax tree. The abstract syntax tree is crucial in genetic programming since it enables random creation of programs and later on breeding by crossover. An abstract syntax tree is a tree were each node is a function with its arguments in the sub nodes. The tree is executed from the leaves up making it possible for functions to have functions as its arguments before execution. The programming language LISP is built on abstract syntax trees, i.e., the code in LISP is written in abstract syntax trees. This makes LISP languages perfect for genetic programming. Figure 2 shows the syntax tree for a mathematical function, its LISP counterpart is:

```
(max (+ x x) (+ x (* 3 y)))
```

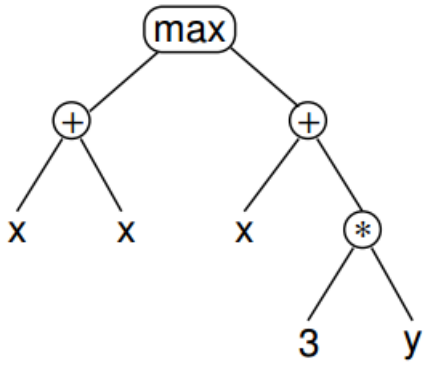

Figure 2: Abstract syntax tree representing *max((x+x), (x + (3 * y)))*

As we can se the LISP code is in an abstract syntax tree already, the function is the first value in the parenthesis and the arguments follow after. Executing the abstract thee is now just a matter of running the LISP code.

3.3 Selection

In order to breed fitter individuals we need to measure their fitness, i.e., how well they solve the problem at hand. In general if a program is good at solving the problem they will be allowed to reproduce. It is important to have a problem where we are able to measure some kind of fitness, otherwise we wont know anything about our programs. This is another restriction on which problems we

can solve with genetic programming. If the fitness measure is time consuming we get slower evolution as each generation will take longer time. This is a factor in how successful the algorithm will be since more generations often means a longer time for the programs to develop successful traits. If we are able to measure absolute fitness for the problem at hand we can rank the entire population and the choice of top individuals is trivial. This is not always the case since a lot of problems only gives us the ability to measure relative fitness, i.e., how good the individuals are in relation to each other. This means we have to let the programs compete in tournaments in order to choose individuals for reproduction.

3.3.1 Elitism

When the problem presents the ability to measure absolute fitness it is preferable to incorporate some elitism. This means we let for example the top 10 percent of the individuals move on to the next generation without altering their code. This prevents the fitness of the next generation from degrading since we make sure the top individuals are present. This is harder to achieve when we only have a relative fitness since its hard to rank the entire population. One would need to implement a comprehensive tournament.

3.4 Breeding

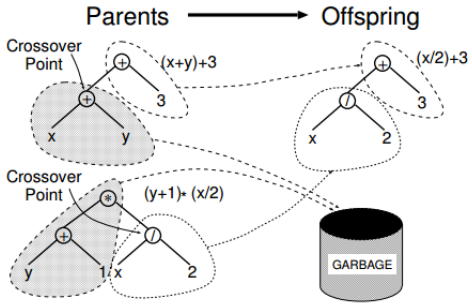

Figure 3: An example of crossover between two programs creating a new program.

We use two different methods for breeding new programs for other programs. The first is crossover, we select a node in the first tree remove a substree (leafnodes are underrepresented to prevent bloating.) and a node in the second tree to find a subtree to replace the removed subtree from the first tree. This is shown in figure 3. The second method is mutation, here we select a node in a tree and insert a random subtree instead of one found in another program.

4 Mathematical functions

Our first goal after delving into the world of genetic programming was to reproduce the results of John Koza (1992). In his paper he approximates a given function using the techniques described above. We created random programs from the functions addition, multiplication and division with arguments ranging from the integers from -10 to 10 and x (a variable). The fitness was measured as the sum of the squared distances at 10 points. Depending on the given goal function a reasonable approximation (fitness lower than 0.01) could be found after a couple of hundred generations. We experimented with sinusoids of different frequencies as our goal function and lower frequencies was no problem where as when the frequency was in the range of 5 there the algorithm showed no sign of finding a reasonable approximation.

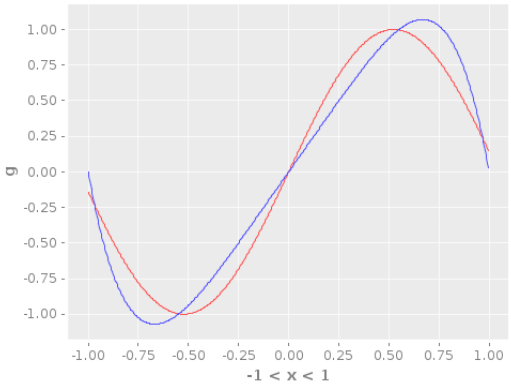

Figure 4: Sinus curve and approximation, red - sinus, blue - approximation

```
(*
 (d (* -1 (d -2 1)) (d (- 0 -1) (- x 0)))
 (*
  (- -1 0)
  (+
   -2
   (*
    1
    (+ (d 0 (- 0 (- (* x x) -1))) (- (- 1 (d -2 -1) (* -2 2) 1) 1)))
   (* (* 1 x) (* x x) x))))
```

Figure 5: Sinus curve approximation

For additional figures see *Appendix A. Plots of population for mathematical approximations.*

5 Back to Reversi

This section describes our attempt to evolve an AI-player heuristic for reversi using our genetic pro-

gramming algorithm. Our goal was to create a player that would prove hard for us to beat.

5.1 Heuristics

As described in the initialization section the ability to solve the problem depends on the definition of the problem domain and the toolbox of functions available. During our research we tried different approaches to toolbox functions.

- Low level functions
- High level functions
- Mix of high and low level

Our hypothesis was that different toolbox functions would result in different AI's, and that a set of lower functions would give the program the possibility of evolving into something we, humans, could not think of but that were successful.

5.1.1 Low level

Going as low as possible in describing the reversi board we wrote a function for each of the squares. The current state of the board and the current player are the arguments for the functions and they return 1 or 0 if there is a dot or not for the current player respectively.

```
(got11 board player)
(got12 board player)
...
(gotxy board player)
```

5.1.2 High level

```
(newif a b A B)
IF a > b do A ELSE do B
(roundnr board)
Number of plays until this moment
(countv board player)
Number of dots for each player
```

5.1.3 Mix of high and low

Using the high level functions as a structure for the low level functions we were hoping the AI would evolve the ability to change strategy at different stages of the game.

5.2 Fitness

As the Reversi fitness for a certain heuristic function only can be expressed in relation to the other heuristics a tournament had to be implemented. To select parent programs for breeding we randomly selected four programs that pairwise played against eachother in a single match. The loser were discarded and the winners breed two new programs as described section 3.4.

5.3 Results

The first results from our algorithm was not very satisfying. The AI showed very little ability to make wise choices. We altered the toolbox of functions and included the higher level heuristic functions. Despite this change we did not see any large improvement. Our next attempt was to include breakpoints for when to change tactics for the AI and limit the number of low level functions. This way we could limit the growth and size of the program and make sure it had different tactics for different stages of the game. The result did not significantly improve after these alterations.

6 Discussion

There are several things we feel we could have done differently which may have yielded better results. In our implementation if a program loses a single match it is discarded, which may not be the best approach. A tournament structure which tested the programs more than once may have been preferable, even though simulating the matches are what needs most computation power. We also believe that the choice of low level heuristics are the key to find a program that performs well in a reasonable amount of time. We think our choices where either way to low level or too high level, we never really succeeded in finding the right balance.

References

Charles Darwin 1859. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, 1st edition. John Murray, London

Ingo Rechenberg 1973. *Evolutionsstrategie* Holzmann-Froboog, Stuttgart

Hans-Paul Schwefel 1974. *Numerische Optimierung von Computer-Modellen* Phd-Thesis

John H Holland 1975. *Adaption in natural and artificial systems.* MIT

John R Koza 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA

John R Koza 2010. *Human-competitive results produced by genetic programming.* Genetic Programming and Evolvable Machines 11.3-4 (2010): 251-284.

Appendix A. Plots of population for mathematical approximations.

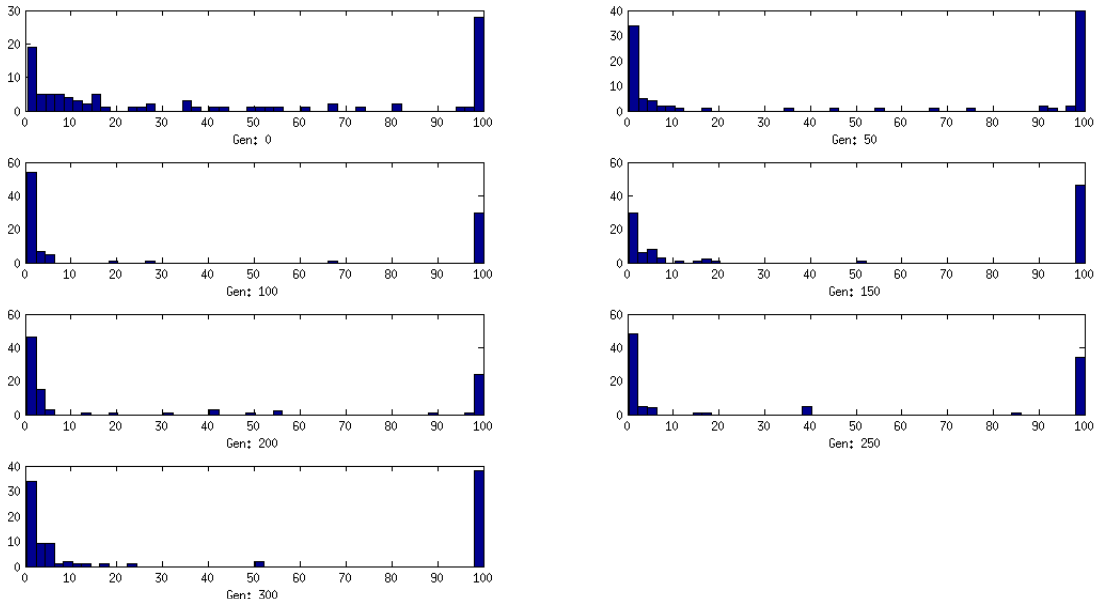

Figure 6: Histogram of the fitness for individuals in different generations, all individuals with fitness over 100 have been put in last bin.

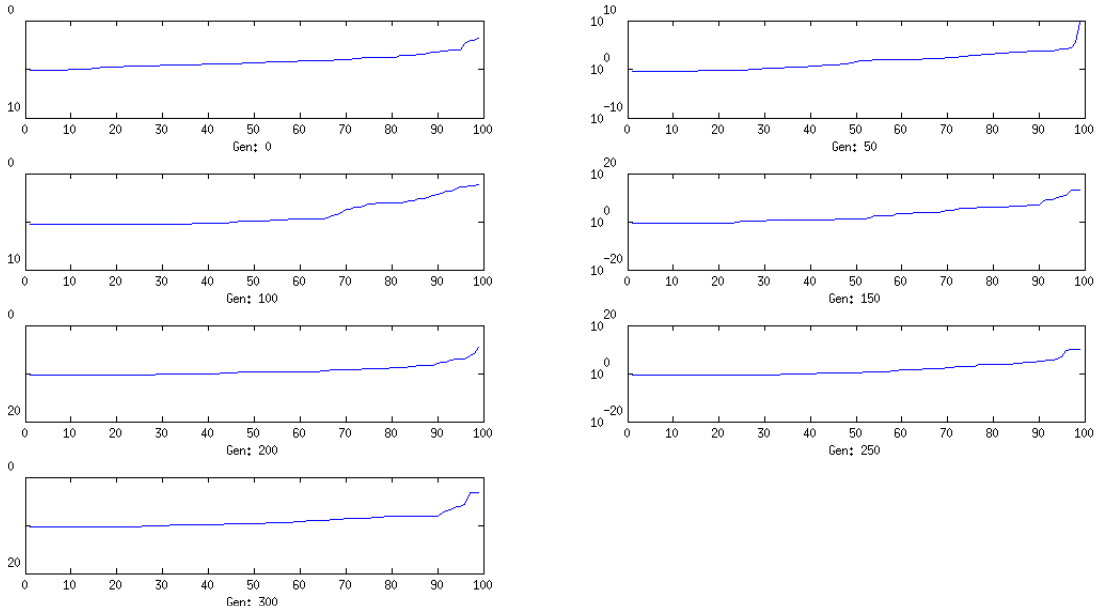

Figure 7: Logarithm of the fitness of the individuals, sorted on fitness from right to left

Appendix B. Graphic interface for Reversi in Clojure.

In order to test the heuristics from our genetic program we constructed a GUI for reversi. This was done using the intercall feature in Clojure, i.e., we used Javas Swing library to contruct the GUI, but all function calls were made from Clojure.

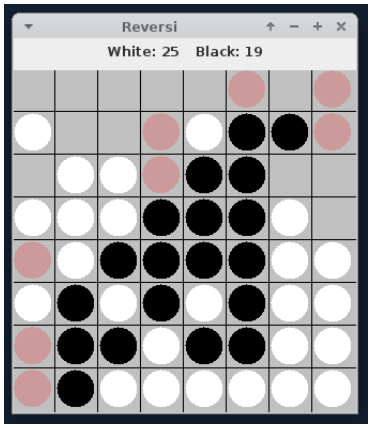

Figure 8: Graphic interface for Reversi in Clojure

```
(defn game-panel
  []
  (let [panel (proxy [JPanel MouseListener]
    []
    (paintComponent [g]
      (proxy-super paintComponent g)
      (do
        (doto g
         ...
         ))
      )]
    (doto panel
      ...
))))
```

Example of using *proxy* and *super-proxy* for intercalling Java functions.