Clojure Robby
A Simple Genetic Algorithm

Adam Krupička

May 27, 2015

Abstract

This short report aims to describe a simple genetic algorithm and make a few observations of the emergent behaviors found in the model. We will evolve a robot, which successfully picks up gold pieces on a two-dimensional map. This algorithm is based on a similar model found in the NetLogo model library.

1 Model Description

The purpose of the model is to create a simple robot through an evolution process. It is based on a model called Robby the Robot, found in the NetLogo model library [1], [2].

The robot's DNA is made up of six genes, as seen in Table 1. Each gene describes a specific action that the robot can take. The robot's surroundings serve as an enzyme, which always activates exactly one gene. The robot then executes the action the gene describes.

Gene	Description
u	Move up
d	Move down
l	Move left
r	Move right
p	Pick up
x	Move random

Table 1: The robot's action genes.

This is encoded in a pseudo-DNA sequence, which functions as a look-up table—each possible situation is projected to a specific action to take.

We simulate the robot on a two-dimensional playground and only let it see the tile it is standing on, and the four tiles adjacent to its own position. The tiles come in three types: (1) a path tile, (2) a wall tile, and (3) a path tile with a gold piece. This makes for 3^5 possible situations the robot can find itself in, if we ignore a few practically impossible situations. Hence, the pseudo DNA consists of 3^5 action descriptors, where each one encodes the action to take in a given situation.

1.1 Artificial Selection

The evolution algorithm starts with a pool of individuals made up of random pseudo-DNA sequences. Then it iterates through many generations of artificial selection and consequent breeding, attempting to produce a more fit generation in each step.

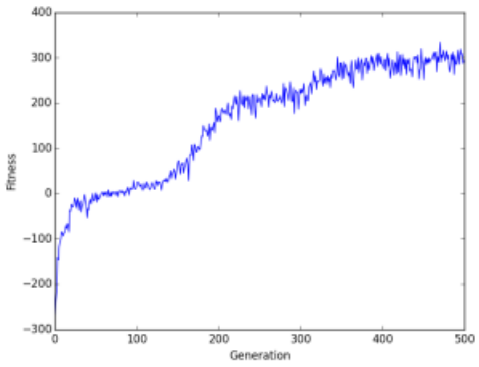

Figure 1: The average fitness per population during the evolution of 500 generations in the default configuration.

The selection is performed thusly: (1) order the individuals by fitness they achieved in multiple runs on different maps, (2) throw away the worse half. The breeding process then doubles the remaining population by crossover. Crossing two individuals means taking the genetic information from a random parent for each element in the pseudo DNA. A plot of this process can be seen in Figure 1. The fitness of a single individual in a run on a map is calculated by simulating the actions of the robot and assigning each action a score value.

Picking up a gold piece is worth ten points, walking into a wall deducts five points, and attempting to pick up a gold piece when there is none deducts one point. However, the robot receives no feedback during the simulation. A single step of the simulation is presented in Figure 2. The frame was extracted from a GIF animation, which was generated from data created by the simulation software. The GIF animation is available online for your viewing[1]. Default parameters were used when evolving the population. The most fit individual of the 500th generation was selected and then simulated on a randomly generated map.

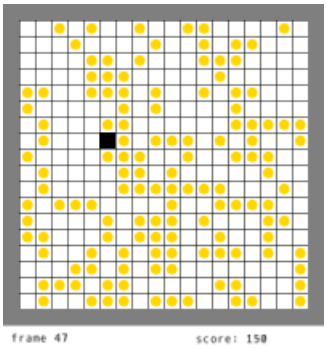

Figure 2: A single frame of the simulation. The gray tiles represent walls, the white tiles are path tiles, the yellow circles represent path tiles with gold present, and the black-filled square is the robot.

[1] `https://www.fi.muni.cz/~xkrupick/clojure-robby/500.gif`

2 Emergent Behaviors

We will now take a look at some of the behaviors that appear during the evolution process, and their dependence on configuration parameters.

2.1 Gold Occurrence Probability

The robot develops a different tactic, depending on how abundant is the gold in the environment. When the gold is scarce, it favors random movement to distinct kinds of movement. This is apparent from Table 2.

Gold Probability	Generation	u	d	l	r	p	x
50%	400	17.3%	20.6%	14%	15.2%	**21.4%**	11.5%
	1000	17.7%	14.8%	13.2%	16.1%	**26.3%**	11.9%
10%	400	9.9%	16.5%	18.9%	16.1%	**23.1%**	15.6%
	1000	11.9%	20.2%	11.1%	16.9%	18.5%	**21.4%**

Table 2: Action distribution in environments with different gold probability. The prevailing actions are highlighted in bold.

However, the robot performs rather poorly in these environments short in gold. The robot can only depend on random movement to get to a different area of map it has not been to yet. In other words, it performs a two-dimensional random walk to attempt to get to tiles it has not visited yet.

Plots of the average fitness are pictured in Figure 3. We can see that it also takes a larger amount of populations to evolve a strategy that attains a positive fitness in environments with scarce gold.

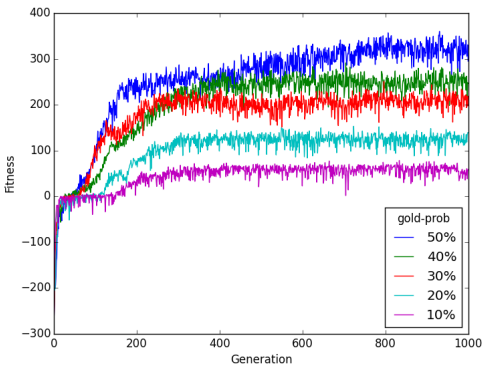

Figure 3: Average score per generation plotted for multiple values of gold occurrence probability.

Nevertheless, the evolution process still manages to create a valid strategy in all of these configurations. This seems to suggest that evolution is an effective tool even in extreme conditions, and that evolution is not easily subverted; although it takes a longer time for a valid strategy to emerge in harsh environments. Luckily, with computers at our disposal, we do not have to wait for so long!

2.2 Starting Position

During the fitness evaluation, the robot starts each simulation at the upper left corner of the map. At first I thought it would be better for the robot to start at the center of the map, however, starting in the corner proved to be more interesting. Table 2 from the previous subsection suggests that the robot evolves to prefer moving down and right; that is, towards the areas of the map that it has not visited yet. This is especially obvious in the harsh environment with scarce gold.

2.3 Gene Mutation Probability

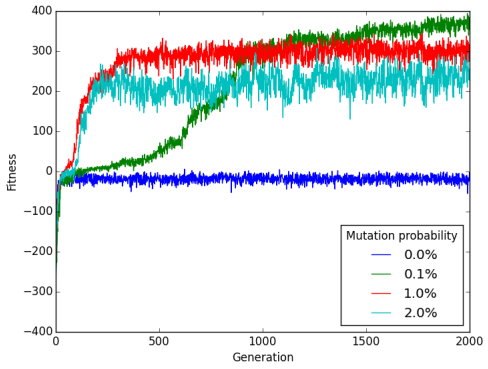

Figure 4: Average score per generation plotted for multiple values of mutation probability.

In the default configuration, the probability of any given gene mutating during the crossover is 1%. This is fine for a quick simulation, however, this probability makes too abrupt mutations after reaching a certain fitness. On the other hand, the mutation probability of zero makes evolving a valid strategy improbable—a valid strategy would have to be present in the variations of the starting population. Theoretically, the optimal value would be an infinitesimal number larger than zero, however, such an evolution would take an infinite time to reach a valid strategy.

Plots of multiple evolutions with different mutation rates are pictured in Figure 4. In our case, selecting the gene mutation probability of 0.1% yields finer results, than the default 1%. A good solution might be to lower the mutation rate as we approach the maximum fitness. Of course, this might become problematic, as it is possible we would reach a local maximum more often than we would reach the global one. In this simple model, this did not prove to be a problem.

3 Implementation

I have decided to implement the genetic algorithm form scratch. I used a functional language called Clojure [3], [4] for this task. Clojure is a dialect of Lisp with an emphasis on functional programming. I enjoyed using the language very much[2]. The algorithm is partially concurrent; the fitness evaluation of many individuals on many maps is performed as a parallel map-reduce operation.

In addition, a simple Python script was created for the purposes of generating the plots for this report. Another Python script was used for the generation of the animated GIF images, which visualize a selected individual.

The complete source can be found on GitHub [5], as can be the executable JVM (Java Virtual Machine) archive [6]. The Java archive needs to be launched from the command line, because it only offers a simple REPL (read-eval-print loop) interface.

4 Concluding Remarks

I would like to mention a few differences between this model and the NetLogo model. Firstly, the DNA crossover is implemented in a different way. To the best of my understanding, the NetLogo model splits the DNA sequences of parents at a random point and glues the two fitting bits together. As described before, my model chooses each action descriptor randomly between the two parents. This seems to have no discernible effect.

Secondly, my model uses a larger map (20 by 20 tiles, walls included) by default, whereas the NetLogo model uses a smaller, 10 by 10 map. This proved to be unnecessarily large, as it takes many steps for the robot to traverse the map (recall that it can mostly rely only on the two-dimensional random walk).

[2]For some time now, I have been meaning to try out a Lisp, and this small project seemed like the perfect opportunity.

References

[1] M. Mitchell, S. Tisue, and U. Wilensky. (2012). Netlogo robby the robot model, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, [Online]. Available: `http://ccl.northwestern.edu/netlogo/models/RobbytheRobot` (visited on 05/25/2015).

[2] U. Wilensky. (1999). Netlogo, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, [Online]. Available: `http://ccl.northwestern.edu/netlogo/` (visited on 05/25/2015).

[3] R. Hickey, "The Clojure programming language", in *Proceedings of the 2008 Symposium on Dynamic Languages*, ser. DLS '08, Paphos, Cyprus: ACM, 2008, 1:1–1:1, ISBN: 978-1-60558-270-2. DOI: `10.1145/1408681.1408682`.

[4] (2007). Clojure, [Online]. Available: `http://clojure.org` (visited on 05/25/2015).

[5] A. Krupička. (2015). Clojure-robby, [Online]. Available: `https://github.com/osense/clojure-robby` (visited on 05/25/2015).

[6] ——, (2015). Clojure-robby executable archive, [Online]. Available: `https://github.com/osense/clojure-robby/releases/download/1.1/clojure-robby-1.1-standalone.jar` (visited on 05/25/2015).