

Genetic Algorithms in the Real World

Simon D. Levy¹
December 20, 2006

1. Introduction

Reports on the industrial application of genetic algorithms (GAs) and related "biologically inspired" search methods appear frequently in the popular science and engineering press. At a superficial level, anyone familiar with the theory of natural selection can imagine how a difficult search problem might be attacked by combining the elements of reasonably good solutions and randomly mutating the resulting "offspring" to preserve variety. At a deeper level, there is often a disconnect between these real-world problems and the kind of problems that students can be expected to attack in an undergraduate AI or machine-learning course. Once students are ready to move beyond the toy problems and easy-to-code algorithms from the introductory phase of the course, they are likely to be faced with an unappealing set of options: wade through and re-factor a poorly-documented source code written for a different problem, purchase a commercial package, or start a lengthy and complicated project from scratch.

2. Project overview

The primary aim of this project is to develop a set of tools to make state-of-the-art genetic algorithms available to undergraduate students with little or no AI background. The target audience includes both computer science majors and students in the natural sciences (biology, geology) and engineering interested in applying GAs to their own research areas. Because of its popularity in undergraduate teaching and professional applications, the Python programming language will be used. The project will focus on the following two themes:

1. Current multi-objective optimization (MOO) approaches like the Nondominated Sorting GA II (NSGA II; Deb *et al.* 2000), which will be provided to students as a simple application programming interface (API). Students will focus on

¹ Corresponding author: levys@wlu.edu, Department of Computer Science, Washington & Lee University, 407 Parmly Hall, Lexington, VA 24450.

- implementing the Simple Genetic Algorithm (SGA; Mitchell 1996) in Python, and then comparing it to an implementation of NSGA-II written by the faculty member, using benchmark problems from the GA literature.
2. Current solutions to benchmark GA problems like the evolution of controllers for pole-balancing (Geva & Sitte 1993). Students will learn the basics of the NEAT (NeuroEvolution of Augmenting Topologies) algorithm (Stanley & Miikkulainen 2001), and will then have the opportunity to see the algorithm at work by training teams of agents in the newly-released version 2.0 of the NERO (Neuro-Evolving Robotic Operatives) video game (Stanley et al. 2005).

3. Project description

The project is split into two parts: (1) learning the NSGA-II algorithm and comparing it to the Simple Genetic Algorithm; (2) learning the NEAT algorithm and seeing how it can be used to train a team of robotic agents in a combat game. Hereafter I describe the project phases in detail along with the deliverables that the students need to submit on completion of each stage.

3.1 -- SGA vs. NSGA-II

A few years ago I wrote a little genetic algorithms package called SuEAP (Suite of Evolutionary Algorithms in Parallel), which enabled Matlab users to use genetic algorithms with fitness computation done in parallel on a Beowulf cluster or other multi-processor Unix platform. Mainly this was a way to run NSGA-II with parallel fitness computation, but I factored out the fitness computation into a parent class, to facilitate addition of future algorithms to the package. I have translated SuEAP into Python, minus the parallelism, and implemented NSGA-II for you. (Which means that the P really stands for Python, not Parallel!) In this part of the project you will implement the Simple Genetic Algorithm (SGA) in Python, and compare the two algorithms on two or more benchmark problems.

First, add the following line to your `.bashrc` file:

```
export PYTHONPATH=/home/courses/cs315/shared/python
```

which will give you access to a nice open-source plotting package used by SuEAP. Next, download and unzip the `ps7.zip` file into a new directory. Then, in a file called `sgap.py`, implement the SGAP class. Specifically, you will have to implement the `__init__` constructor and the update method as specified here. Your constructor should pass the optional argument `seed` to `SuEAP.__init__` (the superclass constructor) and then store the required arguments (`pc`, `mu`, `elit`) in `self` for later. Your update method will be where you implement the Simple Genetic Algorithm shown in the lecture (slide #18). Crossover should take place with probability `pc`: you toss a biased coin, and if it comes up heads (`True`), you cross the parents using the `cross` method, and put the resulting child into the new population. If you get tails (`False`), you randomly pick one of the parents (toss an unbiased coin) and put it into the new population. Like crossover, mutation is already implemented in the test classes; you just call the child's `mutate` method with arguments `mu`, `gen`, and `ngen`.

Note that SuEAP sets up the initial population for you and runs the generations loop (including fitness computation), so you only have to implement the part of SGA highlighted in yellow in the lecture slide. In order to deal with multi-objective fitnesses, SuEAP represents each member's fitness as an array of numbers, so you will have to convert each fitness into a single scalar number before doing fitness-proportionate selection. As we discussed in class, you can just replace each such array by its product or sum. To help you answer the questions below, your update method should report the maximum and mean fitnesses before returning the new population. You may find the utility functions described in `numutils.html` useful in implementing SGA. Note that the "pure" SGA in the lecture does not implement elitism (keeping some fraction of the fittest individuals), so you can ignore the `elit` parameter in your implementation, or implement elitism for extra credit.

I have provided two test problems: simple bit strings, with one-dimensional fitness, and the two-dimensional multi-objective problem from Fonseca & Fleming (1993), cited by Deb et al. (2000) as an example of the superiority of NSGA-II. Once you've written your SGA implementation, you can test it by running the following command at from your Linux terminal:

```
python sueap_test.py
```

The default settings in this test code will run your SGA on the bit-string problem, with a high crossover probability and low mutation rate.

Once you have got your SGA working with these default values, experiment with them to answer the following questions:

- 1. What is the effect on mean fitness of having little or no crossover?*
- 2. What is the effect of a high mutation rate?*

Now you are ready to explore the difference between SGA and NSGA-II. Change your crossover and mutation back to their original values (high crossover, low mutation), and switch from the Bits problem to the Fon problem. Running `sueap_test` with this problem will launch a two-dimensional plot of fitnesses over each generation (close the plot window to see the next generation.) With NSGA-II, each front is plotted using its rank, with the size of the number indicating the crowding distance of that member.

Observe what happens and answer the following question:

- 3. What is happening to the population fitnesses when you collapse them into one dimension for SGA? Do you get a nice, spread-out Pareto front?*

Now change `sueap_test` again so that you're running the NSGA-II algorithm, and answer these questions:

4. How does the Pareto front obtained with NSGA-II on the Fon problem compare with what you get from SGA?

5. With NSGA-II, what happens to the Pareto fronts as the solutions evolve?

If you implemented elitism in your SGA, answer the following question for extra credit:

6 (Extra credit). Does a non-zero elitism value improve the performance of your SGA on either or both test problems?

Here's another extra-credit problem you may want to try:

7 (Extra credit). Copy *fon.py* into another file and modify it to implement one of the other test problems from Table I on p. 187 of the Deb article. How do NSGA-II and SGA compare on this problem?

3.1.1 Part 1 Deliverable

A document answering questions 1-5, with 6 and/or 7 as extra credit

3.2 -- The NEAT algorithm and the NERO game

In this part of the project, you will train a team in NEROGAME, the videogame developed around the rNEAT (real-time) version of the NEAT algorithm. This should be fun and will not require you to write any code. The easiest way to get started is to download the game onto your laptop or home computer (Windows, Mac, or Linux). On all three platforms, you can run it by double-clicking on the NERO icon in the folder where you installed it. As a last resort, you can browse or `cd` to the shared directory where I installed it, and run it from there:

```
cd /home/courses/cs315/shared/nero2_linux_i386/  
./nero.bin
```

Unfortunately, it seems to run faster on Mac and Windows than on Linux. So you can also try installing it on your H: drive if you have enough disk, and run it on one of the Windows machines in P405. For some reason, the tutorial text is munged on the right-hand side on those machines, so I've copied and posted the images for the simple and advanced tutorials for you to follow along with. Let me know ASAP if you can't get it to run at a usable speed on the machines available to you.

To get started, choose Single Player / Simple Tutorial, then move on to the Advanced Tutorial. When you understand the game, select Single Player / Start Training to launch a training mission. For your arena, choose Virtual Sandbox, which is what you were using in the tutorial (it shows as the default for training, but you may have to select it explicitly, or you'll get something else). For your starting team, choose the default <Create New Team>, and for your training team size choose 30. After clicking BEGIN, spawn an enemy team for your team to play against; then spawn your team and start training them. Once you're happy with the results, hit ESC and save your team to a file via SAVE ARMY. Your file name should be your username plus anything else you wish to call it; e.g., levy-team.rtf. Make a note of the team you trained against.

Now you can test your team in battle: go back to the top menu, click Sandbox Battle, load your saved team for Blue Team, and the one you trained against for Red. This is how I am going to test your team, too. You can keep training and testing till you're happy. For maximum fun, share your team with others in the class (you can send each other your .rtf file as an attachment and save it in the nero/data/saves/brains directory), so you can play against each other offline. If we have enough time, we'll try out a multi-player tournament too.

3.2.1 Part 2 Deliverable

A .rtf file containing the specification for the team you trained.

4. Possibilities for future work

If you would like to implement parallel fitness computation as a project (perhaps an R.E. Lee summer scholarship), please let me know.

References and Readings

[Deb *et al.*, 2000] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, *IEEE Trans. Evolutionary Comput.* 6 (2000), 182-197.

[Fonseca & Fleming 1993] C.M Fonseca and P.J. Fleming, Genetic algorithms for multi-objective optimization; Formulation, discussion, and generalization. *Proceedings of the Fifth International Conference on Genetic Algorithms*, Morgan Kauffman (1993), 416-423.

[Geva & Sitte 1993] S. Geva and J. Sitte, A Cartpole Experiment benchmark for trainable controllers, *IEEE Control Systems Magazine*, October 1993, 40—51.

[Mitchell 1998] M. Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, Massachusetts: MIT Press (1996)

[Stanley & Miikkulainen 2001] K. O. Stanley and R. Miikkulainen, Efficient evolution of neural network topologies. *Proceedings of the 2000 Congress on Evolutionary Computation (CEC '02)*, IEEE Press.

[Stanley *et al.* 2005] K. O. Stanley, R. Cornelius, and R. Miikkulainen: Real-time learning in the NERO video game. *AIIDE 2005*: 159-160.