Extracted from:

Genetic Algorithms in Elixir

Solve Problems Using Evolution

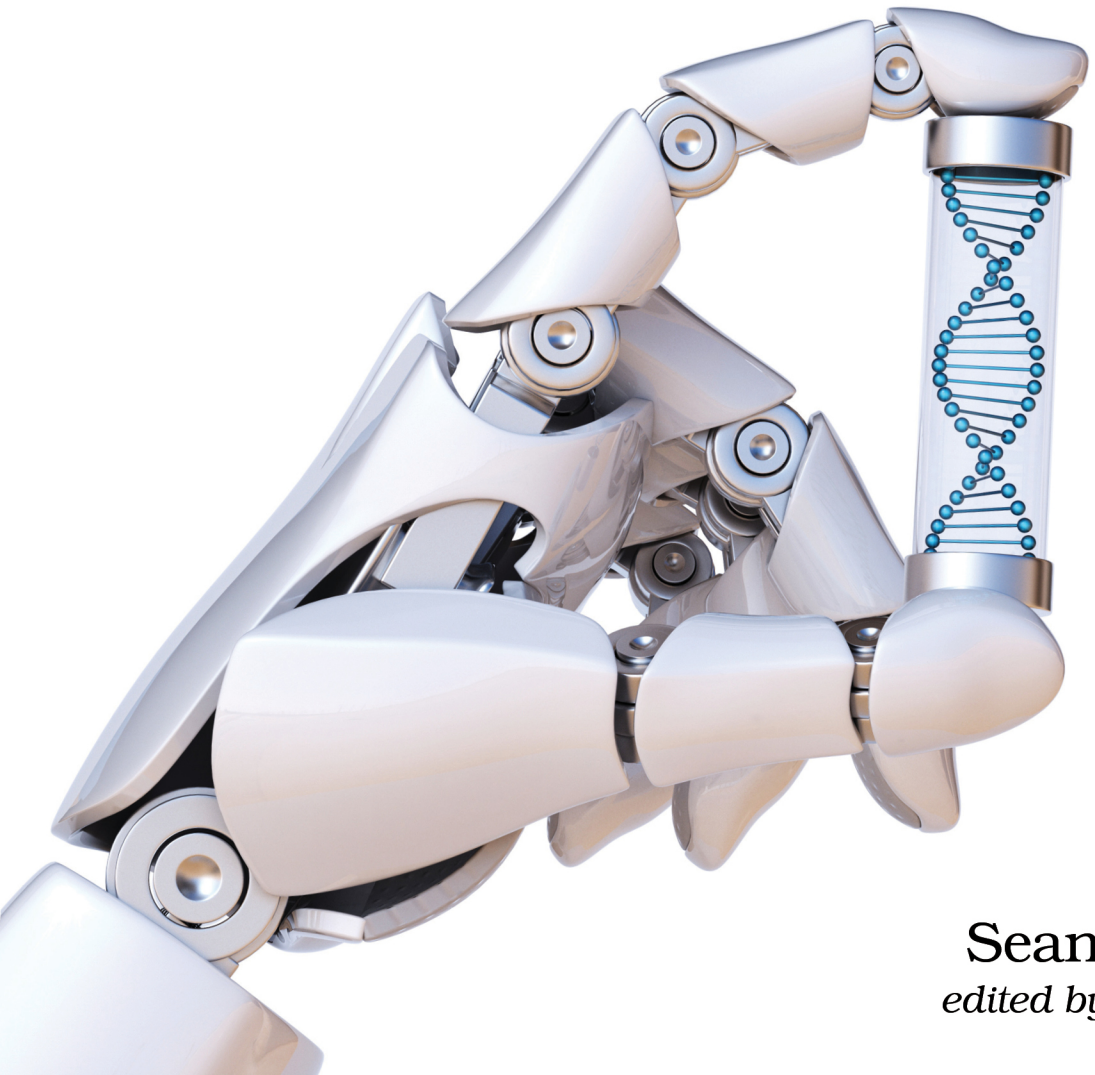
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Genetic Algorithms in Elixir

Solve Problems Using Evolution



Sean Moriarity

edited by Tammy Coron

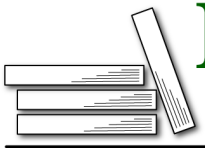
Genetic Algorithms in Elixir

Solve Problems Using Evolution

Sean Moriarity

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Tammy Coron
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-794-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2021

Logging Statistics Using ETS

During an evolution, you may want to track statistics about fitness, age, or variation in your population over the course of the evolution. For example, perhaps you want to determine the distribution of a particular gene at different generations during the evolution.

In this section, you'll create a statistics server using a GenServer and an ETS table. Remember, a GenServer is an abstraction around state that models client-server behavior. It allows you to spin up a long-running process and alter its state through message passing. ETS stands for Erlang Term Storage and offers a built-in storage API through Erlang interpolation. The GenServer will allow you to supervise the ETS table. The ETS table will allow you to quickly and easily insert and look up statistics across generations. Additionally, it'll be easy for you to expand this approach to all kinds of statistics and metrics.

Creating the Statistics Server

Start by creating a new utilities directory inside lib, and inside that new directory, add a new file named statistics.ex. This file will contain the implementation for your statistics server. Start by defining a bare-bones GenServer implementation:

```
defmodule Utilities.Statistics do
  use GenServer

  def init(_opts) do
    :ok
  end

  def start_link(opts) do
    GenServer.start_link(__MODULE__, opts, name: __MODULE__)
  end
end
```

Your GenServer is a wrapper around the ETS table to ensure it's created when your application is started. You only need to implement callbacks for init/1 and start_link/1.

You'll also want to add the Statistics module to your supervision tree. You'll need to create a new file application.ex that implements a supervision tree. The file should look like this:

```
defmodule Genetic.Application do
  use Application
```

```
  def start(_type, _args) do
    children = [
      {Utilities.Statistics, []},
    ]

    opts = [strategy: :one_for_one, name: Genetic.Supervisor]
    Supervisor.start_link(children, opts)
  end
end
```

You also need to update application in mix.exs to look like this:

```
def application do
  [
    extra_applications: [:logger],
    mod: {Genetic.Application, []}
  ]
end
```

This code ensures your GenServer starts on application start.

Now you'll need functionality for accessing the statistics for a generation and for inserting the statistics of a generation. ETS allows you to store any Elixir term in a key-value pair. That means you can use generations as keys and maps of statistics as values. Each field in the map will represent a different statistic you want to track, such as minimum fitness, maximum fitness, average fitness, and so on.

GenServers typically use a client-server paradigm, but for this example, you just need the GenServer to encapsulate your ETS table and initialize it on Application startup. The only GenServer function you need to implement is init/1, like this:

```
def init(opts) do
  :ets.new(:statistics, [:set, :public, :named_table])
  {:ok, opts}
end
```

This function will run when your application is started and ensures you have a new ETS table that you can access with the name :statistics.

You now need to implement insert and lookup functions. insert takes a generation and a map of statistics. You can implement it like this:

```
def insert(generation, statistics) do
  :ets.insert(:statistics, {generation, statistics})
end
```

ETS lookup works much the same way as insertion. You can implement lookup like this:

```
def lookup(generation) do
  hd(:ets.lookup(:statistics, generation))
end
```

:ets.lookup/2 returns a list, so you return the head of the list to extract the statistics entry. There should only be one entry for every generation, so this implementation is fine.

In these functions, you use the ETS API to implement basic insertion and lookup functionality. Your statistics will be logged as a map of statistics every generation. For example, if you wanted to track mean fitness and mean age for an evolution of 1000 generations, your ETS table would contain 1000 entries each with a map containing mean_fitness and mean_age entries.

With your Statistics server set up, you just need to ensure your algorithm tracks statistics during your evolution.

Tracking Statistics in Your Framework

Before you can access the different statistics of an evolution, you need to ensure your algorithm tracks them after every generation. To do this, you'll implement a new function statistics that runs immediately after your population is evaluated and updates the statistics server appropriately.

Start by updating evolve/4 to call statistics/2, like this:

```
def evolve(population, problem, generation, opts \\ []) do
  population = evaluate(population, &problem.fitness_function/1, opts)
➤ statistics(population, generation, opts)
  best = hd(population)
  # ...
end
```

Next, you need to implement statistics/3. To customize the statistics you take between generations, you can accept a :statistics option in opts, which is a keyword list of functions that implement different calculations on your population. You'll want to define a default suite of statistics so you don't have to define these every time. Implement statistics/3 like this:

```
def statistics(population, generation, opts \\ []) do
  default_stats = [
    min_fitness: &Enum.min_by(&1, fn c -> c.fitness end).fitness,
    max_fitness: &Enum.max_by(&1, fn c -> c.fitness end).fitness,
    mean_fitness: &Enum.sum(Enum.map(&1, fn c -> c.fitness end))
  ]
  stats = Keyword.get(opts, :statistics, default_stats)
  stats_map =
    stats
    |> Enum.reduce(%{},
```

```
        fn {key, func}, acc ->
          Map.put(acc, key, func.(population))
        end
      )
  Utilities.Statistics.insert(generation, stats_map)
end
```

First, you define a suite of default statistics, in this case min and max fitness. You then use Keyword.get/3 to obtain the statistics passed to opts. Next, you create a statistics map that applies every function in stats to your population. Finally, you insert this map into your statistics table.

Accessing the Statistics

To access the statistics of an evolution, you can either use the basic API you implemented previously or you can access the :statistics table using ETS. For example, if you wanted to look up the minimum fitness during the third generation of your simulation, you'd do this:

```
# After Algorithm runs

{_, third_gen_stats} = Utilities.Statistics.lookup(3)
IO.write("Min fitness after 3rd Generation: #{third_gen_stats.min_fitness}")
```

The ETS entry is a tuple of {generation, map}. In this example you use pattern matching to extract just the map of statistics.

You can use the basic statistics you've implemented to see how the population tends toward the best fitness over time. Try taking a look at the mean fitness of the 0th, 500th, and 1000th generations:

```
{_, zero_gen_stats} = Utilities.Statistics.lookup(0)
{_, fivehundred_gen_stats} = Utilities.Statistics.lookup(500)
{_, onethousand_gen_stats} = Utilities.Statistics.lookup(1000)

IO.write("""
0th: #{zero_gen_stats.mean_fitness}
500th: #{fivehundred_gen_stats.mean_fitness}
1000th: #{onethousand_gen_stats.mean_fitness}
""")
```

When you run your algorithm, you'll see:

```
$ mix run scripts/tiger_simulation.exs
...
0th: 2.43
500th: 7.09
1000th: 7.0
```

Notice your mean fitness doesn't change much at all between the 500th and 1000th generation. It actually goes slightly down. Your population probably

converges well before your algorithm terminates. In Chapter 10, Visualizing the Results, on page ?, you'll see how you can turn these statistics into graphs and identify about when your algorithm begins to converge.

That's all it takes. You can extend the statistics utility using libraries like elixir-statistics or your own custom statistics functions.

Finding the Average Tiger

Now that you have extensible statistics tracking in place, you can use it to monitor more insightful statistics for your evolution—such as the average tiger for each climate.

You've already identified the fittest tiger for each climate; however, what matters more is how the entire population changes in a given climate. To identify this, you can implement an average_tiger statistic that tells you the average tiger for any given generation.

Start by creating the following average_tiger/1 function in your TigerSimulation module:

```
def average_tiger(population) do
  genes = Enum.map(population, & &1.genes)
  fitnesses = Enum.map(population, & &1.fitness)
  ages = Enum.map(population, & &1.age)
  num_tigers = length(population)

  avg_fitness = Enum.sum(fitnesses) / num_tigers
  avg_age = Enum.sum(ages) / num_tigers
  avg_genes =
    genes
    |> Enum.zip()
    |> Enum.map(& Enum.sum(&1) / num_tigers)

  %Chromosome{genes: avg_genes, age: avg_age, fitness: avg_fitness}
end
```

If you recall from how you implemented statistics/3, each statistic reflects a measure of some value over the entire population. Your average_tiger/1 function takes in the entire population and calculates averages for age, fitness, and genes. Average genes are the average value of each trait.

Now you need to adjust your run to account for this statistic on every generation:

```
tiger = Genetic.run(TigerSimulation,
                    population_size: 20,
                    selection_rate: 0.9,
                    mutation_rate: 0.1,
                    statistics:
                      %{average_tiger: &TigerSimulation.average_tiger/1})
```

Next, rather than inspecting the mean fitness at the 0th, 500th, and 1000th generation, inspect the average tiger:

```
{_, zero_gen_stats} = Utilities.Statistics.lookup(0)
{_, fivehundred_gen_stats} = Utilities.Statistics.lookup(500)
{_, onethousand_gen_stats} = Utilities.Statistics.lookup(1000)

IO.inspect(zero_gen_stats.average_tiger)
IO.inspect(fivehundred_gen_stats.average_tiger)
IO.inspect(onethousand_gen_stats.average_tiger)
```

When you run the simulation in a tropic climate, this is what you should see:

```
%Types.Chromosome{
  age: 1.0,
  fitness: 3.19,
  genes: [0.46, 0.51, 0.38, 0.55, 0.48, 0.6, 0.49, 0.54],
}
%Types.Chromosome{
  age: 1.0,
  fitness: 7.245,
  genes: [0.58, 0.98, 0.99, 0.97, 0.99, 0.96, 0.1, 0.66],
}
%Types.Chromosome{
  age: 1.0,
  fitness: 7.165,
  genes: [0.6, 0.98, 0.94, 0.96, 0.99, 0.97, 0.08, 0.67],
}
```

And you'll see this in a tundra climate:

```
%Types.Chromosome{
  age: 1.0,
  fitness: 2.3,
  genes: [0.49, 0.51, 0.58, 0.39, 0.6, 0.49, 0.55, 0.49],
}
%Types.Chromosome{
  age: 1.0,
  fitness: 6.98,
  genes: [0.96, 0.98, 0.1, 0.09, 0.94, 0.98, 0.94, 0.64],
}
%Types.Chromosome{
  age: 1.0,
  fitness: 7.055,
  genes: [0.98, 0.96, 0.06, 0.08, 0.97, 0.97, 0.97, 0.66],
}
```

You can notice some distinct differences here. In a tropical climate, fat stores and fur thickness are detrimental to a tiger's ability to survive, so over time tigers with those traits become less prevalent. Similiarly, in a tundra climate, these traits are important, so tigers with these traits become more prevalent.

You can also notice the traits that didn't have any meaning in a climate, such as tail length, tend to be present in around 50%–60% of tigers. The evolution doesn't place much emphasis on shorter or longer tails in either environment, so there isn't much of a trend in either direction.

If you're wondering what type of tigers are developed in each environment, it's the Bengal tiger and Siberian tiger.