Extracted from:

Genetic Algorithms in Elixir

Solve Problems Using Evolution

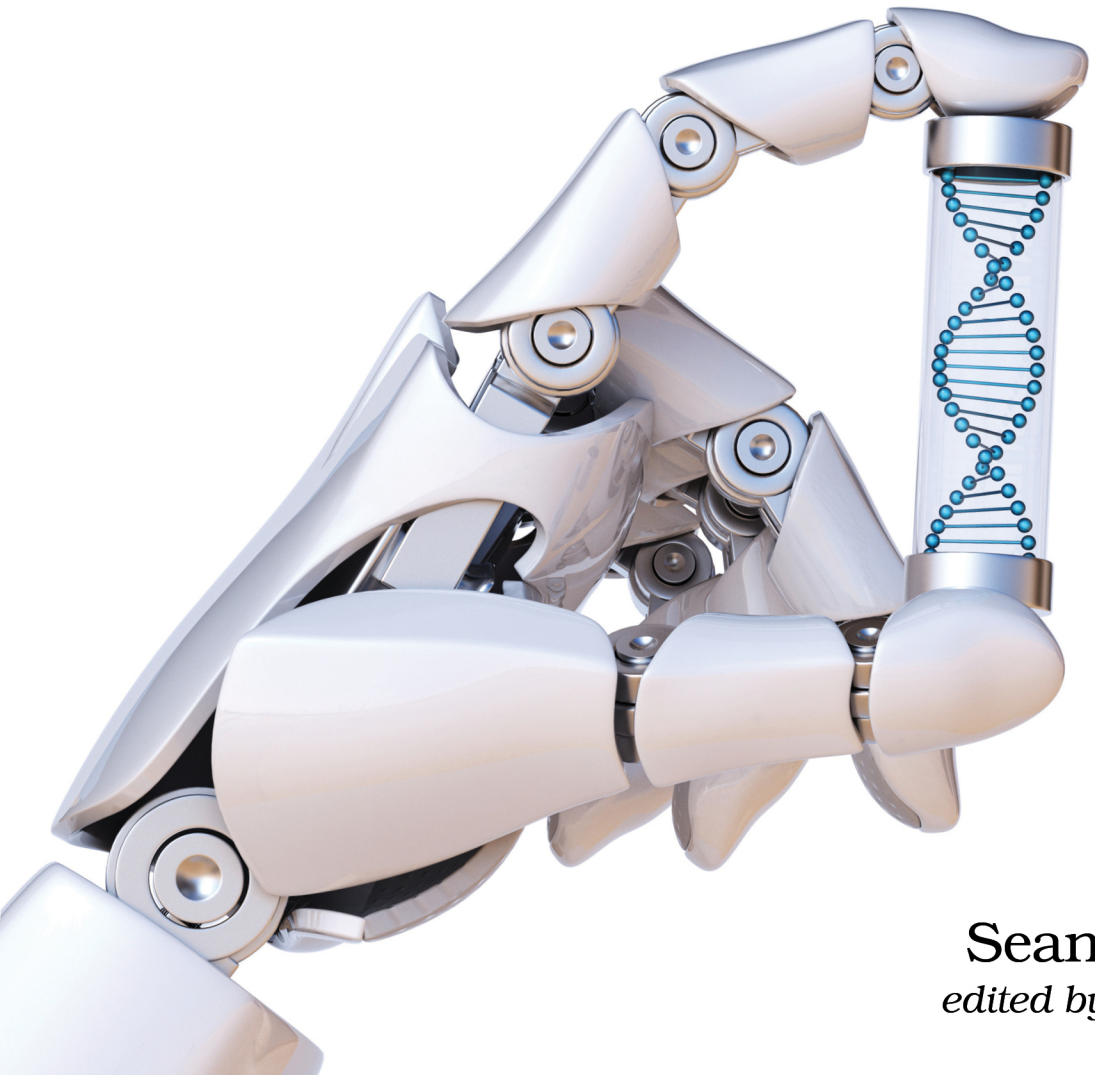
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Genetic Algorithms in Elixir

Solve Problems Using Evolution



Sean Moriarity

edited by Tammy Coron

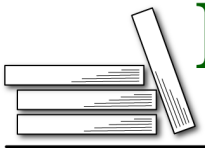
Genetic Algorithms in Elixir

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

Not all problems are created equal. Every optimization problem you face will inevitably have a unique set of challenges. However, this doesn't mean that you should approach every problem differently, as patterns will often arise from one problem to the next.

A key aspect of problem solving is to model problems in a way that makes them easier to understand and thus easier to solve. This might mean translating data into formats that are easier to work with, choosing or creating data structures that simplify solutions, or transforming the problem itself into a form you already know. The steps you take at the beginning when planning your approach to solving a specific problem are vital to finding its solution.

In the previous chapter, you designed a framework for writing genetic algorithms. The framework you designed generalized the steps common to all genetic algorithms. The purpose of this exercise was both to better understand the structure of genetic algorithms and optimization problems and to make it easier for you to write genetic algorithms in the future.

In the process of designing this framework, you separated problem-specific aspects from more general aspects of genetic algorithms. In this chapter, you'll take a closer look at these problem-specific aspects and how to handle them.

Using Structs to Represent Chromosomes

The chromosomes you created in the previous chapters are enumerable objects that represent solutions to a problem. At the most fundamental level, this is correct; however, in practice, this isn't a viable implementation.

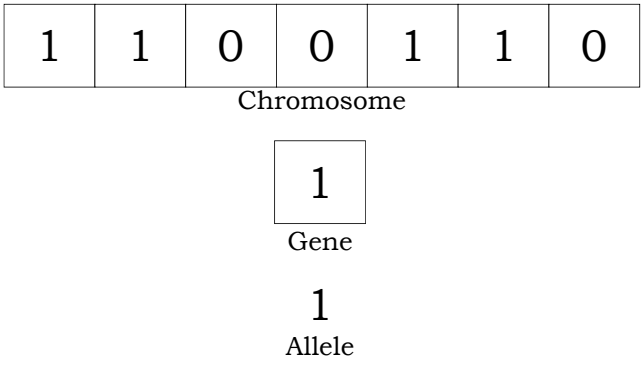
Consider this: you're attempting to solve a problem in which the age of the chromosome determines its fitness. One reason you'd do this is to ensure enough variance between generations. Ideally, you'd persist older chromosomes between generations to a certain point, before killing them off once they've reached a certain age. In this respect, you ensure an equal distribution of both old and young chromosomes and, thus, naturally occurring variance in the population. Solving a problem like this using only an `Enum` type to represent a chromosome creates unnecessary complexity. It's often the case that you need a more robust data structure to keep track of a number of metrics at a time. In Elixir, you can accomplish this task using a *struct*.

A struct is a map with a few additional features. Structs allow you to define default values and required fields. They also cannot take on additional fields after their creation.

The guarantees that structs provide make them a perfect fit for defining custom types—without the fear of breaking your programs. With structs, you can ensure that a predefined chromosome type is initialized with a predefined set of genes—one that won't break your genetic algorithms.

Creating a chromosome struct offers a number of conveniences that you wouldn't have if you simply used an `Enum` or some other data type to represent a chromosome. For example, if you wanted to calculate the average fitness of a population, you would need to recalculate the fitness of each individual chromosome first, which can be a computationally expensive task. Using a struct, however, you can save time by only calculating the fitness once, and then storing it as a key-value pair within the struct itself.

Understanding Chromosomes



Before you're ready to create a struct that models a chromosome, you first need to understand what a chromosome is and what characteristics it has.

At the most basic level, a chromosome is a single solution to your problem. It's a series of genes consisting of values known as *alleles*. Genes can represent any number of things. For example, in the shipping problem introduced in Chapter 1, Writing Your First Genetic Algorithm, on page ?, each gene represents a successive stop in a city. The entire chromosome, then, represents a complete path to every city defined in the problem.

Genes are typically represented using list types or other enumerable data types, like trees, sets, and arrays. In Elixir, the Enum library provides a number of useful functions for manipulating any data type that implements the Enumerable protocol. In fact, the framework you wrote in the previous chapter exclusively uses Enum library functions. Therefore, you can represent genes using any data type that implements the Enumerable protocol—even ones that are not part of the Elixir standard library.

While genes are the most fundamental piece of a chromosome, there are several characteristics you can track for both convenience and functionality. A basic chromosome struct could include fitness, size, and age on top of genes. For simplicity, the chromosome struct in this book will consist of these exact features. In later chapters, you'll see the convenience that tracking these characteristics can provide.

The characteristics you may choose to add to your chromosome struct have no limits. Some problems may require additional features not described in this chapter—the beauty of structs is in their flexibility. Choosing to represent a chromosome in this manner gives you the ability to rapidly adjust what you need for each problem.

Creating a Chromosome Struct

Open a terminal and navigate to the genetic/lib directory. From there, create a new directory named types, as well as a new file named chromosome.ex, like this:

```
$ mkdir types
$ touch types/chromosome.ex
```

You'll create the chromosome struct within types/chromosome.ex. Open the types/chromosome.ex file and add the following code:

```
defmodule Types.Chromosome do

  defstruct [:genes, :size, :fitness, :age]

end
```

This code defines a module Types.Chromosome, which contains a struct consisting of the keys :genes, :size, :fitness, and :age. Remember, defstruct is used to define a new struct. The atoms that follow are the fields the struct contains. You can create a new chromosome struct using the %Types.Chromosome syntax.

This version of the chromosome struct will work, but it lacks a bit of functionality. Currently, none of the fields have default values and there are no required keys. This means that a newly created chromosome struct could technically contain fields with all nil values.

Change the chromosome struct by adding defaults for :size, :fitness, and :age, like this:

```
defstruct [:genes, size: 0, fitness: 0, age: 0]
```

Now, any newly created chromosome will have a default size, fitness, and age of 0. You could technically make the default values whatever you want—it all depends on what you're trying to accomplish.

Finally, all chromosomes must contain genes. If a chromosome doesn't have any genes, it's not really a chromosome. To make this chromosome require genes, add the following code above defstruct:

```
@enforce_keys :genes
```

Your final module will look like this:

```
defmodule Types.Chromosome do
  @enforce_keys :genes
  defstruct [:genes, size: 0, fitness: 0, age: 0]
end
```

You can now create chromosome structs that track the genes, size, fitness, and age of a chromosome in your populations.

Creating a Chromosome Type

Elixir is a dynamically typed language; however, it's often useful to create *typespecs* for custom data types. Typespecs are useful for documentation and static code analysis using tools like dialyzer. This book won't cover the use of dialyzer, but you'll use the types you create in this section later in the chapter.

A typespec is defined using the @type attribute followed by the name and definition of the type. Elixir supports compound types as well as the creation of custom types using structs.

You'll create your chromosome type in the types/chromosome.ex file. Open the types/chromosome.ex file and add the following code above the struct you defined in the previous section:

```
@type t :: %__MODULE__{
  genes: Enum.t,
  size: integer(),
  fitness: number(),
  age: integer()
}
```

This code creates a custom type t, which is an instance of a Types.Chromosome struct. The __MODULE__ keyword is a macro that gets replaced with the name of the module in which it's defined. t is a standard practice for defining module types in Elixir.

The chromosome type also declares specific types for the fields of the chromosome. As mentioned previously, genes must be an Enum type. Size and age are both integers. Fitness is a number, which is a built-in Elixir type representing a float or integer.

The final Chromosome module will look like this:

```
defmodule Types.Chromosome do

  @type t :: %__MODULE__{
    genes: Enum.t,
    size: integer(),
    fitness: number(),
    age: integer()
  }

  @enforce_keys :genes
  defstruct [:genes, size: 0, fitness: 0, age: 0]
end
```

Using Behaviours to Model Problems

Recall that one technique to solving problems is to transform them into a form you already understand. While every problem seems different, and on the surface may require different techniques to solve, they almost always have patterns and similarities between them. This is especially true with the problems you'll solve with genetic algorithms.

The framework you built in Chapter 2, Breaking Down Genetic Algorithms, on page ?, separates a few problem-specific parameters from the common aspects of a genetic algorithm. These parameters are a fitness function, a genotype, and termination criteria. This means that every problem you attempt

to solve using a genetic algorithm must implement all three of these functions—the nature of the framework creates a natural *abstraction* for problems.

An abstraction is a simplification of underlying complexities and implementations. The purpose of abstraction is to force you to think of things at different levels of specificity. It gives you an idea of what to look for before you approach a problem. This is especially useful when approaching new problems with genetic algorithms. When you want to approach a new problem, you already know that you need a fitness function, or a way to measure success; a genotype, or a way to represent solutions; and some termination criteria, or a way to tell the algorithm when to stop. While the specifics are the difficult part, you're never starting from scratch with this abstraction in place.

Unfortunately, Elixir doesn't feature abstract classes, interfaces, or traits like other object-oriented languages. Instead, you can implement abstraction using *behaviours*.

Mind the "u"

Elixir uses the British spelling of "behaviour."