
UiT The Arctic University of Norway
Genetic algorithms
An optimization algorithm modelled after Darwin's evolutionary theory
Genetic Algorithm Code example
Andreas Dyrøy Jansson
C3190

Hello there

What we will cover

- In this lecture, we will take a look at the code for a simple GA
  - Making a class representing a chromosome with a set of features
  - Implement a simple mutation function with constraints
  - Implement arithmetic crossover
  - Implement a mapping function from lower-level to higher-level representation
  - Implement a fitness function based on a given problem
  - Assemble each part into a working algorithm

In this lecture, we will cover the implementation of a complete genetic algorithm. First, we will make a class to represent the chromosome and its features.

Next, we will make a mutation function and show how we can deal with constraints.

We will implement arithmetic crossover, which is the simplest crossover method

We will implement a mapping function to map from one representation level to another

We will implement a fitness function based on a given

problem

And finally, we will assemble all these parts into a complete, working genetic algorithm

The problem

- Consider the following problem:
  - *a ? b = 42*
  - Where *a* and *b* are two unknown numbers, and "?" is some operator
- We have the following constraints:
  - *a* **must** be between -1000 and 1000
  - *b* **must** be between -200 and 200
  - The operator *?* can be either +, -, *, or /
- In the next few slides, we will solve this problem using a genetic algorithm

For this example, let's consider the following problem: A with B equals 42. A and B are two unknown numbers, and the question mark is a mathematical operator, like addition, subtraction, division and multiplication.

We also have a set of constraints: The value of A MUST be larger than -1000 and smaller than 1000, while B must be between -200 and 200. The operator can be one of four possible options: plus, minus, star, or slash.

In the next few slides, we will solve this problem by implementing a genetic algorithm

The chromosome class

- We start by creating a chromosome class to hold our features
  - GAs are generally easier to implement in an object-oriented approach
  - An alternative is to use dictionaries and multidimensional vectors
- We analyze the problem, and find that we have three features: *a, b,* and the operator *?*
  - We see that the chromosome should represent an **equation**
- We will refer to them in the code as *feature_1*, *feature_2*, and *feature_3*
- We set the fitness to infinity
  - Huh?

```
8   class Chromosome:
9       def __init__(self, feature_1, feature_2, feature_3):
10          self.feature_1 = feature_1 # [-1000, 1000]
11          self.feature_2 = feature_2 # [-200, 200]
12          self.feature_3 = feature_3 # [+, -, *, /]
13          self.fitness = float('inf')
```

We start by creating a class representing a chromosome. In general, it's much easier to implement genetic algorithms through an object-oriented approach. Of course, it's possible to just use lists, dictionaries and multidimensional vectors to implement the GA, but object orientation makes it easier to see the connection with the flowchart we looked at in the previous lectures.

First, we must analyze and understand the problem we are given. We find that we have three features, A, B, and the operator. We conclude that the chromosome we are making should represent an equation.

Going forward, we will refer to these features as feature 1, feature 2, and feature 3.

The chromosome also holds its own fitness value. For the moment, we set this to infinity. This may seem counterintuitive, but if you think about it, we are trying to minimize the difference between the expression in the chromosome, and the given answer. Since we don't have values for A and B, we assume the worst, and set the initial fitness to a very large number.

Representation

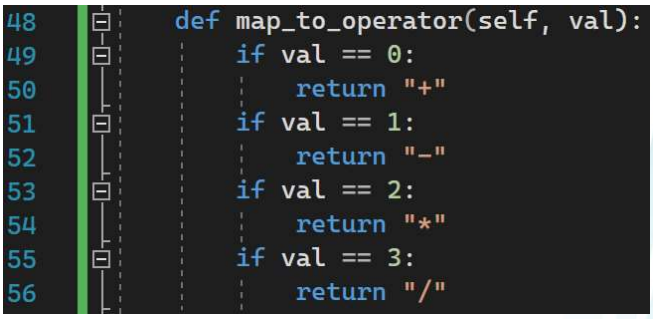
- When implementing a genetic algorithm to solve a problem, we must decide how we want to represent the features
- Numerical values can be used directly
  - We call this high-level representation
  - Can be mutated by adding or subtracting a random value
- In our problem, we also have the operator *feature_3*
  - We could save it as a string with values “+”, “-”, “*”, or “/”
- But - how do we “mutate” a string?
  - We can map the string to a numerical representation

Another important factor we must consider is how we want to represent the values in the chromosome. For example, in this case, we can represent feature 1 and feature 2 as numerical values without any mapping. We call this high-level representation. We can mutate the values by adding or subtracting a random number.

We also have feature 3, which is the operator. We could try to just save it in the chromosome as a string, but this raises the question: how can we easily mutate a string? The easiest solution is to map the string to a numerical representation.

Mapping features

- We create a helper function which takes an integer 0 – 3 and returns the corresponding operator



```
48      def map_to_operator(self, val):
49          if val == 0:
50              return "+"
51          if val == 1:
52              return "-"
53          if val == 2:
54              return "*"
55          if val == 3:
56              return "/"
```

Here is an example of how this can be done. We say that 0 represents plus, one is minus, 2 is multiply and 3 is divide.

Fitness and crossover

- We create simple getters and setters for the fitness
- We implement **arithmetic** crossover
  - For the numerical values *feature_1* and *feature_2*, we can simply take the **average**
  - We can't take the "average" of two operators. We decide to copy *feature_3* randomly from one of the parents (25)
  - The method returns a new chromosome as a mix of both "parents" (23)

```
15      def set_fitness(self, new_fitness):
16          self.fitness = new_fitness
17
18      def get_fitness(self):
19          return self.fitness
```

```
21  def crossover(self, other):
22      d = bool(rnd.randint(0, 1))
23      return Chromosome((self.feature_1 + other.feature_1) / 2, \
24          (self.feature_2 + other.feature_2) / 2, \
25          self.feature_3 if d else other.feature_3)
```

Next we create simple getters and setters for the fitness. We then implement crossover, and for this example, we use the simplest arithmetic crossover. Intuitively, you can imagine that we are simply taking the average of the two parents to create the offspring. This is easy enough when we are working with numerical values. However, things become slightly trickier when we are dealing with discrete values like the operator in feature 3. It doesn't really make sense to take the average of two operators like + and – directly. This can be solved in many ways, for example, we can take inspiration from multi-point

crossover, and copy the feature directly from one of the parents.

First we generate a random True or False value, and on line 25 in the code, we select the operator from either self or other and copy it to the offspring using the constructor.

Finally, the crossover function returns a new offspring, which is a mix of both parents “self” and “other”.

Mutation

- To avoid getting stuck in local minima, we use mutation to perturb the population
- We try a mutation rate of 5%
  - Mutation rate is a hyperparameter of GAs
- Since we are using high-level representation on *feature_1* and *feature_2*, we offset them with random values (30), (36)
  - Note that the random range corresponds to the legal values of each feature
- Since we mapped the operator to a numerical value, it is easy to mutate in the same way as *feature_1* and *feature_2* (42)
- We must also check that we do not exceed the constraints of each feature (31-34), (37-40), (43-46)

```
1   import random as rnd
2
3   mutation_rate = 0.05
```

```
27      def mutate(self):
28          chance = rnd.random()
29          if chance < mutation_rate:
30              self.feature_1 += rnd.randint(-100, 100)
31              if self.feature_1 > 1000:
32                  self.feature_1 = 1000
33              elif self.feature_1 < -1000:
34                  self.feature_1 = -1000
35
36              self.feature_2 += rnd.randint(-30, 30)
37              if self.feature_2 > 200:
38                  self.feature_2 = 200
39              elif self.feature_2 < -200:
40                  self.feature_2 = -200
41
42              self.feature_3 += rnd.randint(-2, 2)
43              if self.feature_3 > 3:
44                  self.feature_3 = 3
45              elif self.feature_3 < 0:
46                  self.feature_3 = 0
```

We also need to implement the mutation operator. This is a function to help the population to avoid getting stuck in a local minimum. Mutations introduce some randomness into the population, and if we are lucky, we get a beneficial mutation that helps certain individuals solve the problem better, resulting in increased fitness.

Usually, the mutation rate is quite low, for example from 2 to 10 percent. We try a mutation rate of 5 percent, but we can tweak this later if we see that it is too high or too low. We say that mutation rate is a hyperparameter of genetic algorithms.

Since we are using high-level representation in feature 1 and feature 2, we can mutate them by adding a random number which can be positive or negative. Note that the mutation range should be adjusted to fit with the legal values for each feature. Feature 1 has a much larger range, -1000 to 1000, so we can use a larger range. Feature 2 can only be between -200 and 200, so we use a smaller range.

Also, since we mapped the operator to numerical values, we can do the same for feature 3. Since feature 3 can only be one of four values, we use a very small range.

Finally, we must check that the mutated values do not violate our constraints. If they do, we simply clamp them to their respective legal values.

Evaluation

- In order to compute the fitness of each chromosome, we need a way to test it against our problem
  - *feature_1 "feature_3" feature_2 =* 42
- We creature the function "eval()" which computes the expression represented by the chromosome
- We want to minimize the difference between the expression and the answer
  - Low or zero difference thus equals high fitness
- Finally, we make a function to print out the chromosome in a readable format

```
58      def eval(self):
59          if self.map_to_operator(self.feature_3) == "+":
60              return self.feature_1 + self.feature_2
61          if self.map_to_operator(self.feature_3) == "-":
62              return self.feature_1 - self.feature_2
63          if self.map_to_operator(self.feature_3) == "*":
64              return self.feature_1 * self.feature_2
65          if self.map_to_operator(self.feature_3) == "/" and self.feature_2 != 0:
66              return self.feature_1 / self.feature_2
67          else:
68              return float('inf')
```

```
70      def __str__(self):
71          return str(self.feature_1) + " " + self.map_to_operator(self.feature_3) \
72              + " " + str(self.feature_2) + " = " + str(self.eval())
```

Finally, we must create a way to evaluate and test the chromosome against the problem. In this case, we are creating an expression, so we evaluate each chromosome by computing its expression, and comparing the result to the answer. As we said in the start, we want to minimize the difference between the output of each chromosome, and the actual answer.

To help us debug the code, we also create a str-function to print out the features in a more readable format.

Create the initial population

- Create an empty list (74)
- Define the population size (75)
- Populate the list with chromosomes (77-79)
  - Initial values are random within the legal ranges

```
74  population = []
75  pop_size = 100
76
77  for i in range(pop_size):
78      population.append(Chromosome(rnd.randint(-1000, 1000), \
79          rnd.randint(-200, 200), rnd.randint(0, 3)))
```

Now we can begin to implement the actual genetic algorithm. We create an empty list to hold our population, and we define a population size. We initially set it to 100, but we can change this later. This is another hyperparameter.

We then populate the list with randomly generated chromosomes.

Selection implementation

```
5    answer = 42
```

- We need a way to select the fittest individuals for crossover
- We implement **elitist** selection
- First, assign fitness to each chromosome (86-88)
  - Fitness is the absolute value of the difference. In this particular case, low “self.fitness” actually means high fitness
  - We could *invert* self.fitness by setting it to *1/difference,* but this would cause problem when the difference is exactly 0, which is actually what we want
    - In this case, it’s easier to just keep it as is
- Sort the population based on fitness (90)
- Select a number of the top fittest pairs and perform crossover (92-95)
  - We choose 20% (92)
- Mutate the new offspring (94)
- We also replace the least fit individuals (95)

```
86  for p in population:
87      f = abs(p.eval() - answer)
88      p.set_fitness(f)
89
90  population.sort(key=lambda x: x.get_fitness())
91
92  for j in range(1, 21):
93      new_offspring = population[j - 1].crossover(population[j + 1])
94      new_offspring.mutate()
95      population[len(population) - j] = new_offspring
```

Now we need a way to select the most fit individuals for crossover. We can use elitist selection, which is easy to implement. The selection algorithm works as follows: first, we compute the fitness of every single chromosome in the population. Since we are minimizing the difference, we define the fitness as the absolute difference between the output of each chromosome, and the answer. So in this case, a low “self.fitness” value is actually the same as high fitness. Alternatively, we could invert this value by taking one over difference, which would produce a large number when the

difference is small, and a small number when the difference is big. However, we want the difference to be exactly zero, and thus would not work as we would get a division by zero. So in this case, it is easier to just keep it as is. Remember that, depending on the type of problem we are solving using GA, how we interpret the fitness value is implementation dependent.

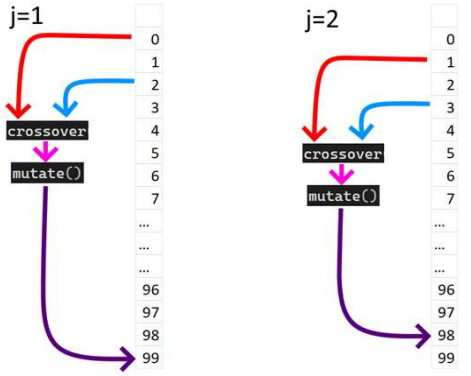
Next, we sort the population based on fitness, or difference in this case.

As we know, elitist selection works by selecting a number of individuals from the most fit percentage for crossover. In this case, we try to pick from the top 20%. This is also a hyperparameter that we can change later. We get a new offspring, apply mutation, and replace one of the least fit individuals.

Elitist selection breakdown

- We know that the most fit individuals are near the start of the list after sorting
- The for-loop works in the following way:
  - On the first iteration, *j* is 1
  - This means that we select individual *j-1* = 0 and *j + 1* = 2
  - We replace *the len()-jth* individual
- The loop runs for 20 iterations
  - ➢ We replace the 20 least fit chromosomes

```
for j in range(1, 21):
    new_offspring = population[j - 1].crossover(population[j + 1])
    new_offspring.mutate()
    population[len(population) - j] = new_offspring
```



Let's take a closer look at how the selection algorithm is implemented. We know that the most fit individuals are near the start of the list after we sorted it. We use a for-loop, and pick pairs of individuals in the following way: first, the value of j is 1. We take individual j minus 1 and j plus one from the list, and perform crossover. So for the first iteration, we pick number zero and two, next we pick number one and three, and so on, until we have 20 new chromosomes. These 20 new chromosomes replace the 20 least fit ones.

Assembling the algorithm

- We set population size to 100 (74)
  - Another hyperparameter
- We create the initial population (76-78)
- We run the GA until the average fitness (difference from the answer) is less than a certain value (82)
- For each generation:
  - Compute the fitness of each chromosome (86-89)
  - Compute the average fitness (91)
  - Sort the population based on fitness (93)
  - Perform selection (98-101)

```
73  population = []
74  pop_size = 100
75
76  for i in range(pop_size):
77      population.append(Chromosome(rnd.randint(-1000, 1000), \
78          rnd.randint(-200, 200), rnd.randint(0, 3)))
79
80  average_population_fitness = float('inf')
81
82  while average_population_fitness > 0.1:
83
84      pop_sum = 0
85
86      for p in population:
87          f = abs(p.eval() - answer)
88          p.set_fitness(f)
89          pop_sum += f
90
91      average_population_fitness = pop_sum / len(population)
92
93      population.sort(key=lambda x: x.get_fitness())
94
95      print("Average population fitness: " + str(average_population_fitness) \
96          + " best individual: " + str(population[0]))
97
98      for j in range(1, 21):
99          new_offspring = population[j - 1].crossover(population[j + 1])
100         new_offspring.mutate()
101         population[len(population) - j] = new_offspring
```

Now we have all the parts we need, and we put them together to make the full genetic algorithm.

First, we set the population size to 100, and create the initial population. We then use a while-loop to run the algorithm until the average fitness of the entire population is less than some small value. This is also a hyperparameter we can change. We could also make the while-loop so that it runs for a certain number of iterations before it stops. In GAs, the number of iterations of the while-loop is the same as the number of generations of chromosomes.

So, to summarize: for each generation, compute the fitness of every chromosome, compute the average fitness, sort the population so that the most fit individuals are grouped together, perform the selection and crossover, replace the least fit, and repeat.