Constrained Function Optimization using Evolutionary Learning

David Qorashi - Project #4b

As always, the implementation utilized the power of Ruby language.
Two primary classes in the program are **Chromosome** and **Population**.

Chromosome (chromosome.rb)

If the constructor got called with not passing any parameters, it will randomly initialize two float numbers (will be used for solving the equation). After that, it will encode these two numbers into a binary string (consisted of 0 and 1s) using a utility method named encode_float.

The encode_float implementation: (code will be found in float_utility.rb)

```
def FloatUtility.encode_float(number, num_bits_floats, a, b)
  ((number - a) / ((b - a).to_f / (2 ** num_bits_floats - 1)))
      .round.to_s(2).rjust(num_bits_floats, '0')
end
```

The method receives a number along with a range (a,b). Then it will try to encode the number into a string with length of num_bits_floats. (I end up setting num_bit_floats to 32, so I am building strings with size of 32 characters from the float numbers. The bigger the number, the more accurate float number we generate from our “float string” is.

The decode_float implementation: (code will be found in float_utility.rb)

```
def FloatUtility.decode_float(bit_string, a, b)
  bit_string.to_i(2) * ((b - a).to_f / (2 ** bit_string.length - 1)) + a
end
```

The method simply receives a float number representation in string format along with a range (a, b). Then it tries to decipher and map the string representation to a float number in the given range.

At the end, the constructor will concatenate two string representations of the float numbers and then create a ‘genes string’ with the length of NUM_BITS_GENES.

Otherwise, if we feed a string into the constructor, it sets its instance genes variable’s value to passed parameter.

Fitness: the function simply decrypts two float numbers from the genes string and then applies the decoded values to the Goldstein-Price function.

Mutate!: the function tries to introduce random bit changes to the genes string according to a MUTATION_RATE defined in the application. Current value of MUTATION_RATE is 0.01.

Crossover (&): the method performs a crossover between current chromosome and a second one. It takes the first float number from the first one and second float number from the second one and put them whole together into a new chromosome. Further, it extracts the second float number from the first chromosome and the first float number from the second chromosome and combine them into another new chromosome. The two parents will got replaced with these two children chromosomes.

Population (population.rb)

The constructor creates a vector.

Seed!: it creates POPULATION_SIZE chromosomes and stores them in the vector built already in constructor.

tournament_selection: I hold tournaments to specify which chromosomes can find their way to the next generation. I hold the tournament TOURNAMENT_NUM times. The winner will got selected to survive through the next generation.

```
def tournament_select
    best = nil
    TOURNAMENT_NUM.times do
      selected_citizen = self.chromosomes[rand(POPULATION_SIZE)]
      if (best == nil) or selected_citizen.fitness < best.fitness
        best = selected_citizen
      end
    end
    return best
end
```

Due to nature of the problem (minimization), in each tournament the winner is the chromosome which has the lower fitness. If it was a finding maxima problem we wanted to select the winner with a greater fitness value.

test.rb

The program creates a new population and fills it with random chromosomes. For NUM_GENERATIONS it will try to generate an offspring population based on current population. Selecting chromosomes from parent population and putting them into offspring is done with running tournaments between the chromosomes of the current population. It mates two selected parents with likelihood of CROSSOVER_RATE . The parents are replaced with two children; otherwise, two parents will be added directly to the offspring. It tries to run mutate! method on two children with probability of MUTATION_RATE.
At the end, the population will got replaced by the offspring.

Output

Criteria:
POPULATION_SIZE = 2000
NUM_GENERATIONS = 100
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.01

I just sampled some chromosomes from my population in the last generation.

```
Chromosome: 1000000000000000100010000110011000111111111111111100010100100111, Fitness: 3.0000004501236948
Chromosome: 1000000000000000000010000100110000111111111111111101010000000100, Fitness: 3.0000000150940336
Chromosome: 1000001000000000000000111000001000000000000000000000100100000010110, Fitness: 65557.99789843598
Chromosome: 1000000000000000100011100111110001000000000000000000000000011100, Fitness: 3.0000002906330407
Chromosome: 1000000000000000000010000110011000111111111111111110100011110111, Fitness: 3.0000000164157368
Chromosome: 1000000000000000001001111100010000111111101111111111100111010111, Fitness: 3.001659141126843
Chromosome: 1000000000000000000010100000010100111111111111111111111100101001, Fitness: 3.0000000015591386
Chromosome: 1000000000000000000010100011011001000000000000000000010011110010, Fitness: 3.000000014746315
Chromosome: 1000000000000000000010000010111101000000000000001000110010011100111, Fitness: 3.000007711428697
Chromosome: 1000000000000001001000000111010010100000000000100000010000000011110, Fitness: 3.0000984401872945
Chromosome: 1000000000000010000001101010001101000000000000000011000100111000 0, Fitness: 3.000003464126974
Chromosome: 1000000000000000001001001100100100111111111111111111110001010001, Fitness: 3.000000021381768
Chromosome: 1000000000000001000000101010011000111111111111111111001101000010, Fitness: 3.0000010027961164
Chromosome: 1000000000000000001010011100100100111111111111111110011100001000, Fitness: 3.0000000531324353
e.Fina x,y: 0.0,-1.0
Fitness: 3.0000000000020735
```

Finally, the program is sorting the fitnesses values for the last generation and returns the minimum one.

Effectiveness

I used several different values for constans, but finally I end up using the mentioned values.

I tried not to do mutation on the code just for testing purposes. It caused early convergence in the ecosystem. Most of the times my best fitness ended up around 3.03 (averagely)

Sample output with no mutation:

```
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217
Chromosome: 1000000001100110011111001010111001000000100010111110110010010101, Fitness: 3.0296390358543217

Final x,y :0.006255,-0.99146
Fitness: 3.0296390358543217
```

As we see, all the chromosomes end up to be the same. Of course no anomaly observed among the population members due to lack of mutation. I should mention that program execution decreased from 21.30 seconds to 7.40 with not applying mutation.

Another issue was choosing the right size for the population. My finding is that using a greater POPULATION_SIZE causes better results.

I set POPULATION_SIZE to 20 (instead of 2000). My best result was:
Best x, y: 1.963285,0.312509
Fitness: 89.29398027254591

As you see, it is not impressing at all.

Source Code

```
## test.rb
```

```ruby
require_relative 'chromosome'
require_relative 'population'

POPULATION_SIZE = 2000
NUM_GENERATIONS = 100
CROSSOVER_RATE = 0.7
MUTATION_RATE = 0.01


population = Population.new
population.seed!

NUM_GENERATIONS.times do |generation|
  offspring = Population.new

  population.print_fitness_values

  while offspring.size < population.size
    parent1 = population.tournament_select
    parent2 = population.tournament_select

    if rand <= CROSSOVER_RATE
      child1, child2 = parent1 & parent2
    else
      child1 = parent1
      child2 = parent2
    end

    child1.mutate!
    child2.mutate!

    if POPULATION_SIZE.even?
      offspring.chromosomes << child1 << child2
    else
      offspring.chromosomes << [child1, child2].sample
    end
  end

  population = offspring
end

population.inspect
```

chromosome.rb

```ruby
require_relative 'float_utility'

class Chromosome
```

```
  attr_accessor :genes

  MIN_RANGE = -2
  MAX_RANGE = 2
  NUM_BITS_GENES = 64
  NUM_BITS_FLOATS = 32

  def initialize(genes = "")
    if genes == ""
      initial_x = FloatUtility.random_float_in_range(MIN_RANGE, MAX_RANGE)
      initial_y = FloatUtility.random_float_in_range(MIN_RANGE, MAX_RANGE)
      self.genes = FloatUtility.encode_float(initial_x, NUM_BITS_FLOATS,
MIN_RANGE, MAX_RANGE) + FloatUtility.encode_float(initial_y, NUM_BITS_FLOATS,
MIN_RANGE, MAX_RANGE)
    else
      self.genes = genes
    end
  end

  def to_s
    genes.to_s
  end

  def fitness
    x = FloatUtility.decode_float(self.genes[0..NUM_BITS_FLOATS - 1],
MIN_RANGE, MAX_RANGE)
    y = FloatUtility.decode_float(self.genes[NUM_BITS_FLOATS, 2 *
NUM_BITS_FLOATS - 1], MIN_RANGE, MAX_RANGE)

    return (1 + ((x + y + 1) ** 2) * (19 - 14 * x + 3 * (x ** 2) - 14 * y + 6
* x * y + 3 * (y ** 2))) * (
      ((2 * x - 3 * y) ** 2) *
      (18 - 32 * x + 12 * (x ** 2) + 48 * y - 36 * x * y + 27 * (y ** 2)) +
      30)
  end

  def mutate!
    mutated = ""
    0.upto(genes.length - 1).each do |i|
      allele = genes[i]
      if rand <= MUTATION_RATE
        mutated += (allele == "0") ? "1" : "0"
      else
        mutated += allele
      end
    end
```

```
    self.genes = mutated
  end

  def &(other)
    locus = NUM_BITS_FLOATS

    child1 = genes[0, NUM_BITS_FLOATS] + other.genes[locus, NUM_BITS_FLOATS]
    child2 = other.genes[0, NUM_BITS_FLOATS] + genes[locus, NUM_BITS_FLOATS]

    return [Chromosome.new(child1), Chromosome.new(child2)]
  end
end
```

population.rb

```
require_relative 'float_utility'
```

```ruby
class Population
  attr_accessor :chromosomes

  TOURNAMENT_NUM = 2

  MIN_RANGE = -2
  MAX_RANGE = 2

  def initialize
    self.chromosomes = Array.new
  end

  def print_fitness_values
    vals = chromosomes.map(&:fitness)
    chromosomes.each_with_index do |c, i|
      puts "Chromosome: #{c}, Fitness: #{vals[i]}"
    end
  end

  def inspect
    fitnesses = {}
    chromosomes.each do |chromosome|
      fitnesses[chromosome.genes] = chromosome.fitness
    end

    key = fitnesses.min_by{|k,v| v}[0]
    max_fitness = fitnesses[key]
    x, y = FloatUtility.decode_gene(key, MIN_RANGE, MAX_RANGE)
    puts "#{x},#{y} : #{max_fitness}"
  end

  def seed!
    chromosomes = Array.new
    1.upto(POPULATION_SIZE).each do
      chromosomes << Chromosome.new
    end

    self.chromosomes = chromosomes
  end

  def size
    self.chromosomes.size
  end

  def tournament_select
      best = nil
      TOURNAMENT_NUM.times do
        selected_citizen = self.chromosomes[rand(POPULATION_SIZE)]
```

```
        if (best == nil) or selected_citizen.fitness < best.fitness
          best = selected_citizen
        end
      end
      return best
  end
end
```

float_utility.rb

```
class FloatUtility
```

```
  def FloatUtility.random_float_in_range(min, max)
    rand * (max - min) + min
  end

  def FloatUtility.encode_float(number, num_bits_floats, a, b)
    ((number - a) / ((b - a).to_f / (2 ** num_bits_floats -
1))).round.to_s(2).rjust(num_bits_floats, '0')
  end

  def FloatUtility.decode_float(bit_string, a, b)
    bit_string.to_i(2) * ((b - a).to_f / (2 ** bit_string.length - 1)) + a
  end

  def FloatUtility.halve(str)
    # str length should be even
    boundary = str.length / 2
    head = str.slice(0, boundary)
    tail = str.slice(boundary, str.length)
    return head, tail
  end

  def FloatUtility.decode_gene(str_gene, a, b)
    halves = FloatUtility.halve(str_gene)
    x = FloatUtility.decode_float(halves[0], a, b)
    y = FloatUtility.decode_float(halves[1], a, b)
    return x.round(6), y.round(6)
  end
end
```