Genetic Algorithm that evolves the population to output "Hello World"

Author: Michal Pavlíček
Student ID: i6306065

This is a Jupyter Notebook for a Computational and Cognitive Neuroscience (a Data Science & AI 1st year 2nd period course at Maastricht University) assignment.

This Jupyter Notebook aims to explain the methods I used to develop the my version of the genetic algorithm and to clarify my choices. To run the code, enter "python .\ga_hello_world.py" to the terminal. The "ga_hello_world.py" file contains all the same code as this Jupyter Notebook apart from the statistics, which is exclusive for the Jupyter Notebook and can be found at the end of this document.

The aim of this program is to develop a genetic algorithm that evolves the population to have contain chromosome "Hello World". It does so by first generating a population of random chromosomes. After that the algorithm evaluates them by assigning fitness. That is calculated by counting how many letters did each inidividual have in common with the TARGET variable. After that, the choice of the surviving individuals is made by one of three (elitist, roulette, tournament) selections available. When the best individuals are chosen, they go through crossover, which means they mate with each other and create new offspring. These then go through mutation, which gives them a chance to randomly alter their chromosome. This process is repeated until the TARGET variable is reached.

Answers to Questions:

(More information about them at the end of the Notebook)

1. The bigger the population, the faster the algorithm evolves. That is because there is a greater chance of having fitter individuals at the beginning as well as in each particular population, for example due to greater chance of lucky mutations.
2. From my testing, the mutation seems to have a kind of bell curve distribution, as the best results are obtained around 10 % mutation rate. If the mutation rate is either too big or too small, then the genetic algorithm takes longer to evolve.
3. Upon leaving out the crossover function alltogether, the genetic algorithm is way slower. That is due to the fact that it only relies on a random chance of mutation, which is not very effective. The crossover function is necessary to create new offspring from the best individuals.
4. Without mutation, the elitist selection becomes unreasonable because it loses a lot of genetic material upon each generation. But even if I choose tournament selection, the algorithm never converges to the target. With population big enough, it would be able to eventually converge in most cases.

5. For this kind of problem, I found the optimum to be *elitist selection* with *10 % mutation rate*. That will definitely not be the case for all problems, because the elitist selection loses a lot of information in each generation, which may make it way worse then the other two selection methods. Ultimately, the sweet spots will vary from problem to problem.

```
# Imports

import random
import math
```

The three cells below have been provided, I have only rewritten them to Python.

The Individual class is an encapsulation for each individual in a population, where each individual has a certain chromosome (string) and a fitness value (int).

```
# Individual class implementation

class Individual:
    def __init__(self, chromosome: str):
        self.chromosome = chromosome
        self.fitness = 0

    def __repr__(self):
        return "Individual(" + self.genoToPhenotype() + ", fitness=" + str(self.fitnes

    def getChromosome(self):
        return self.chromosome

    def setChromosome(self, chromosome):
        self.chromosome = chromosome

    def getFitness(self):
        return self.fitness

    def setFitness(self, fitness):
        self.fitness = fitness

    def genoToPhenotype(self):
        return ("".join(self.chromosome))
```

The HeapSort algorithm sorts Individuals descendingly based on the the getFitness() method.

```
"""
@(#)HeapSortAlgorithm.java   1.0 95/06/23 Jason Harrison

Copyright (c) 1995 University of British Columbia

Permission to use, copy, modify, and distribute this software
and its documentation for NON-COMMERCIAL purposes and without
fee is hereby granted provided that this copyright notice
appears in all copies. Please refer to the file "copyright.html"
for further important copyright and licensing information.

UBC MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF
THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED
TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
PARTICULAR PURPOSE, OR NON-INFRINGEMENT. UBC SHALL NOT BE LIABLE FOR
```

```
ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

A heap sort demonstration algorithm
SortAlgorithm.java, Thu Oct 27 10:32:35 1994

Modified by Steven de Jong for Genetic Algorithms.

Modified by Jo Stevens for practical session.

Rewritten by Michal Pavlíček to Python (with help from GitHub Copilot).

@author Jason Harrison@cs.ubc.ca
@version 1.0, 23 Jun 1995

@author Steven de Jong
@version 1.1, 08 Oct 2004

@author Jo Stevens
@version 1.2, 14 Nov 2008

@author Michal Pavlíček
@version 1.3, 19 Nov 2022
"""

class HeapSort:
    def sort(self, i: list[Individual]):
        N = len(i)

        k = int(N / 2)
        while k > 0:
            self.downheap(i, k, N)
            k -= 1

        while N > 1:
            T = i[0]
            i[0] = i[N - 1]
            i[N - 1] = T

            N = N - 1
            self.downheap(i, 1, N)

    def downheap(self, i, k, N):
        T = i[k - 1]

        while k <= N / 2:
            j = k + k
            if j < N and i[j - 1].getFitness() > i[j].getFitness():
                j += 1

            if T.getFitness() <= i[j - 1].getFitness():
                break
            else:
                i[k - 1] = i[j - 1]
                k = j

        i[k - 1] = T
```

The TARGET constant is the text we want the population to evolve to. POPULATION_SIZE constant determines how many individuals we have in each population.

```python
# Constant variables

TARGET = "HELLO WORLD"
POPULATION_SIZE = 100
alphabet = [chr(i) for i in range(ord('A'), ord('Z') + 1)]
alphabet.append(' ')
```

The function *create_initial_population()* creates a list of Individual objects, each having random chromosome.

Function *assign__fitness(population)* assigns each individual a fitness value. The fitness value is determined by how many letters the Individual's chromosome share at the same index with the TARGET constant.

So if TARGET = "TEST" and Individual's chromosome = "AESA", the Individual's fitness is 2.

```python
# Helper functions

def create_initial_population(population_size) -> list[Individual]:
    # we initialize the population with random characters
    population = []
    for i in range(population_size):
        tempChromosome = []
        for j in range(len(TARGET)):
            tempChromosome.append(
                alphabet[random.randint(0, len(alphabet) - 1)])
        population.append(Individual(tempChromosome))

    return population


def print_population(population: list[Individual]):
    for i in population:
        print(i.genoToPhenotype())


def assign_fitness(population: list[Individual]):
    for individual in population:
        fitness = 0
        chromosome = individual.getChromosome()

        for index in range(len(chromosome)):
            if chromosome[index] == TARGET[index]:
                fitness += 1

        individual.setFitness(fitness)

def sort(population: list[Individual]):
    heapSort = HeapSort()
    heapSort.sort(population)

    return population
```

Selection

The code below contains all the selections I've considered.

First, the **elitist selection** chooses the best *percentage* of individuals in a population.

The **roulette selection** gives each individual a chance to be selected based on it's fitness value.

Lastly, the **tournament selection** generates a random number *k*, then randomly selects *k* individuals and chooses the fittest one.

```python
# Selection functions

def elitist_selection(population: list[Individual], percentage: int):
    if percentage > 100:
        percentage = 100
    # select the best *percentage* individuals
    return population[:int(len(population) / 100 * percentage)]

def roulette_selection(population: list[Individual], percentage: int):
    if percentage > 100:
        percentage = 100
    # Power is used to increase the probability of the fittest individuals
    power = 10
    new_individuals = []

    for _ in range(int(len(population) / 100 * percentage)):
        # Calculates the total fitness in population
        total_fitness = 0

        for individual in population:
            total_fitness += individual.getFitness() ** power

        # Select a random number from 0 to total_fitness
        rand = random.randrange(0, total_fitness)

        # The code below adds the fitness of each individual to the sum until it is gr
        # Once the sum is greater than the random number, the current individual is se
        selected_individual = None
        temp_sum = 0
        for individual in population:
            if temp_sum >= rand:
                selected_individual = individual
                break
            temp_sum += individual.getFitness() ** power
        else:
            selected_individual = population[-1]

        new_individuals.append(selected_individual)
        population.remove(selected_individual)

    return new_individuals

def tournament_selection(population: list[Individual], percentage: int):
    if percentage > 100:
        percentage = 100
    # Generates tournament size (= k) between 2 and len(population) / 2
    tournament_size = random.randint(2, int(len(population) / 2))

    new_individuals = []
    for _ in range(int(len(population) / 100 * percentage)):
        # Selects k random individuals from population
        sample = random.sample(population, tournament_size)
        # Chooses the best individual from the sample
        best = max(sample, key=lambda x: x.getFitness())

        # Adds the best individual to the new population and removes it from the old c
        new_individuals.append(best)
        population.remove(best)

    return new_individuals
```

Crossover

The crossover function takes number of crossovers as an input and then mates each individual with each other individual. Out of the new offspring, it randomly selects a sample of size POPULATION_SIZE.

Example: if the TARGET is "NEUROSCIENCE" (= length 21) and num_crossovers is 2, then the new offspring of two individuals would inherit first 4 letters from 1st parent, 4 middle letters from 2nd parent and last 4 again from the 1st parent. The second new offspring would have it the other way around.

```
def middle_crossover(population: list[Individual], num_crossovers: int, population_siz
    new_population = []

    for individual1 in population:
        for individual2 in population:
            chromosome1 = []
            chromosome2 = []

            alternate = True
            step = math.ceil(len(TARGET) / (num_crossovers+1))

            for i in range(0, len(TARGET), step):
                added_chromosome_1 = individual1.getChromosome()[i:i+step]
                added_chromosome_2 = individual2.getChromosome()[i:i+step]

                if alternate:
                    chromosome1.append(added_chromosome_1)
                    chromosome2.append(added_chromosome_2)
                else:
                    chromosome1.append(added_chromosome_2)
                    chromosome2.append(added_chromosome_1)
                alternate = not alternate

            res_1 = chromosome1[0]
            for i in range(1, len(chromosome1)):
                res_1 += chromosome1[i]

            res_2 = chromosome2[0]
            for i in range(1, len(chromosome2)):
                res_2 += chromosome2[i]

            offspring1 = Individual(res_1)
            offspring2 = Individual(res_2)
            new_population.append(offspring1)
            new_population.append(offspring2)

    return random.sample(new_population, population_size)
```

Mutation

The mutation function takes *chance* of mutation as an input and then each letter has a *chance* of being mutated to a random letter.

```
# Mutation functions

def mutation(population: list[Individual], chance: int):
    if chance > 50:
        chance = 50

    for individual in population:
        for index in range(len(individual.getChromosome())):
            if random.randint(0, 100) <= chance:
                individual.getChromosome()[index] = alphabet[random.randint(
                    0, len(alphabet) - 1)]

    return population
```

The code below contains a function *run_ga(population)* (= run Genetic Algorithm), that executes all the neccessary steps one at a time.

```
# Actual code

def run_ga(population: list[Individual], selection_function, success_rate: int, num_cr
    generation = 0
    while True:
        if prnt:
            print("Generation: " + str(generation))

        assign_fitness(population)

        population = sort(population)

        if prnt:
            print(population[0])

        if population[0].getFitness() == len(TARGET):
            if prnt:
                print("Evolution finished!")
            return generation

        population = selection_function(population, success_rate)

        population = middle_crossover(population, num_crossovers, population_size)

        population = mutation(population, mutation_chance)

        generation += 1


population = create_initial_population(500)

run_ga(population, elitist_selection, success_rate=15, num_crossovers=2, mutation_char
Generation: 0
Individual(IENLGS I LA, fitness=3)
Generation: 1
Individual(HXKLO SOFJI, fitness=5)
Generation: 2
Individual(HEQXODWRILD, fitness=6)
Generation: 3
Individual(HELXO SZILD, fitness=7)
Generation: 4
Individual(HELXO SOVLD, fitness=8)
Generation: 5
Individual(HELXO SOVLD, fitness=8)
Generation: 6
Individual(HELXO WOILD, fitness=9)
Generation: 7
Individual(HELXO WORLC, fitness=9)
Generation: 8
Individual(HELXO WORLD, fitness=10)
Generation: 9
Individual(HELXO WORLD, fitness=10)
Generation: 10
Individual(HELLO WORLD, fitness=11)
Evolution finished!
```

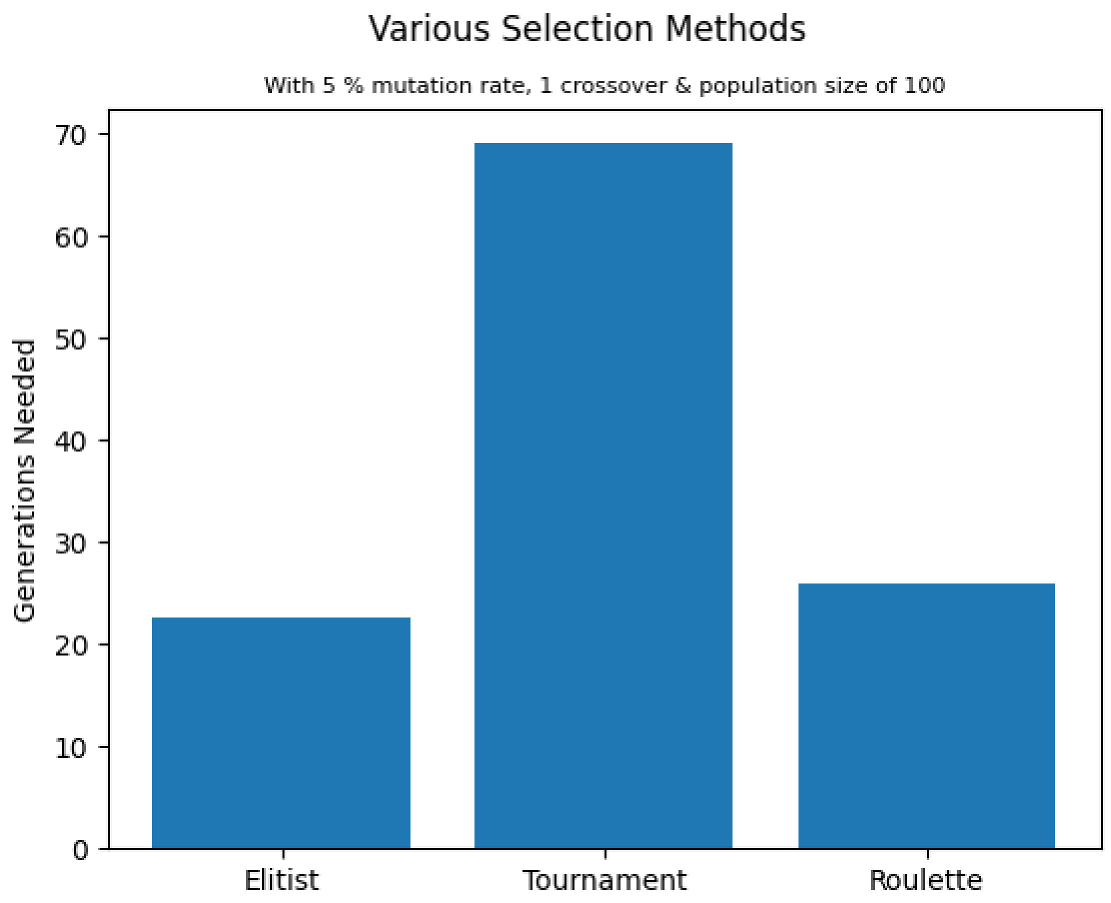
```
Out[ ]: 10
```

Statistics

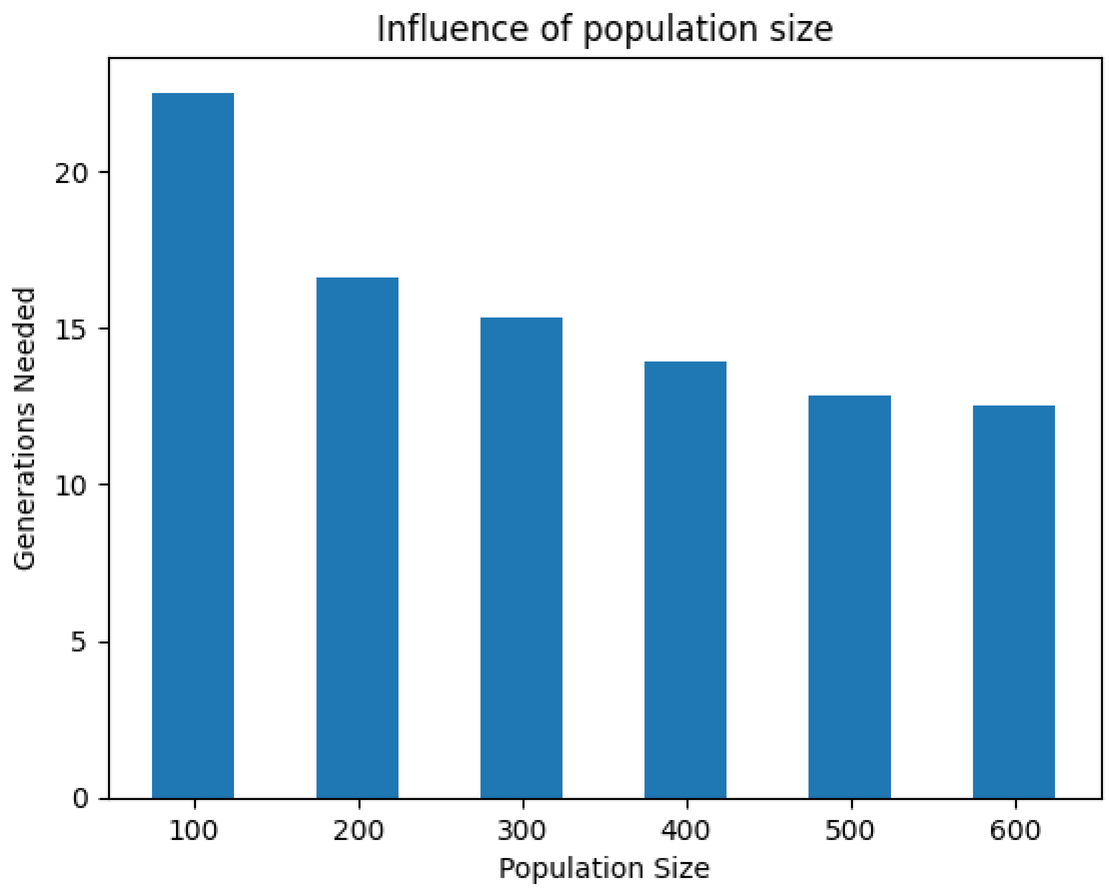
Bonus Section

The code below is used to gather some staticstics about the genetic algorithm. The aim of that is to find out the average number of generations it takes for the population to evolve to the TARGET value (in this case "HELLO WORLD").

The first and most imporatant remark is, that no matter what values are changed, **elitist selection always performs the best**. After that is the tournament selection, and the worst is by far the roulette selection. It might be some implementation detail that I missed, but it's more likely that the roulette selection is just not a good choice for this kind of problem.

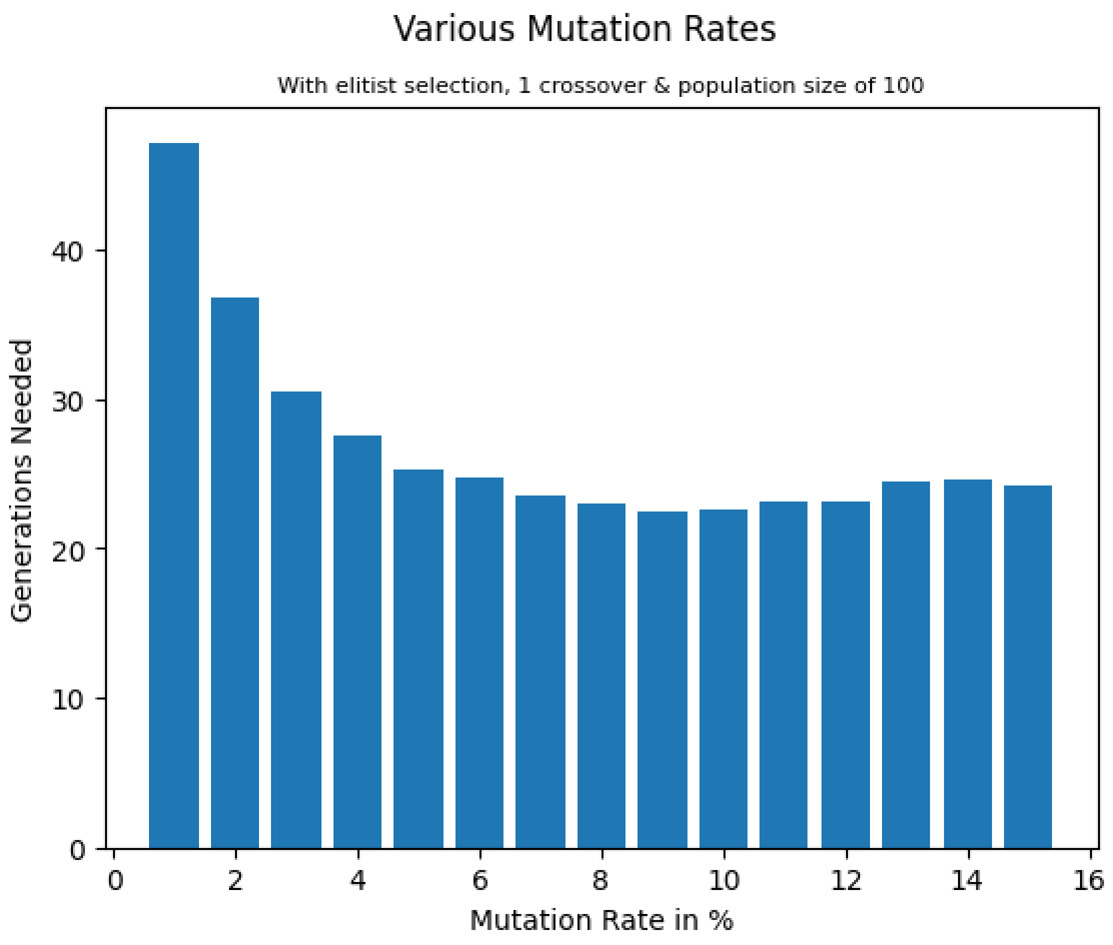


Then it is clear that **the bigger the population, the faster the population evolves**. That can be seen when comparing TEST 1 and TEST 2. When the population doubled, the results took about third of the generations. It is visible from the plot that this holds even for further increases in population size.



Another interesting fact is that if we only let a **lower number of individuals to mate**, the population evolves faster. That can be seen when comparing TEST 3 and TEST 1. In TEST 1, I used 15 % rate to mate, while in TEST 3 I used initially 10 %, which slightly improved the results. Then I tried 20 %, which in turn made the results slightly worse.

Lastly, the mutation rate seems to **peek at around 9 %**.



Note: The code is quite slow, around 2 minutes per each test. Use on your own risk.

```
run_tests = 100


# TEST 1
# Tests success rate 15 %, 1 crossover, 5 % mutation, 100 population size
if False:
    population_size = 100

    elitist_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        elitist_test += run_ga(population, elitist_selection, success_rate=15, num_cro

    print("Elitist average: " + str(elitist_test / run_tests)) # 22.5

    roulette_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        roulette_test += run_ga(population, roulette_selection, success_rate=15, num_c

    print("Roulette average: " + str(roulette_test / run_tests)) # 69

    tournament_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        tournament_test += run_ga(population, tournament_selection, success_rate=15, r

    print("Tournament average: " + str(tournament_test / run_tests)) # 26

# TEST 2
# Tests success rate 15 %, 1 crossover, 5 % mutation, 200 population size
if False:
    population_size = 200

    elitist_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        elitist_test += run_ga(population, elitist_selection, success_rate=15, num_cro

    print("Elitist average: " + str(elitist_test / run_tests)) # 16.6

    roulette_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        roulette_test += run_ga(population, roulette_selection, success_rate=15, num_c

    print("Roulette average: " + str(roulette_test / run_tests)) # 34

    tournament_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        tournament_test += run_ga(population, tournament_selection, success_rate=15, r

    print("Tournament average: " + str(tournament_test / run_tests)) # 17.4

# TEST 3
# Tests success rate 10 % & 20 % respectively, 1 crossover, 5 % mutation, 100 populati
if False:
    population_size = 100

    elitist_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        elitist_test += run_ga(population, elitist_selection, success_rate=20, num_cro

    print("Elitist average: " + str(elitist_test / run_tests)) # 21, 25

    roulette_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        roulette_test += run_ga(population, roulette_selection, success_rate=20, num_c

    print("Roulette average: " + str(roulette_test / run_tests)) # 78.5, 74
```

```python
    tournament_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        tournament_test += run_ga(population, tournament_selection, success_rate=20, r

    print("Tournament average: " + str(tournament_test / run_tests)) # 26, 27

# TEST 4
# Tests success rate 15 %, 1 crossover, 1 % mutation, 100 population size
if False:
    population_size = 100

    elitist_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        elitist_test += run_ga(population, elitist_selection, success_rate=15, num_cro

    print("Elitist average: " + str(elitist_test / run_tests)) # 32

    roulette_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        roulette_test += run_ga(population, roulette_selection, success_rate=15, num_c

    print("Roulette average: " + str(roulette_test / run_tests)) # 176

    tournament_test = 0
    for i in range(run_tests):
        if i % 10 == 0:
            print(i)
        population = create_initial_population(population_size)
        tournament_test += run_ga(population, tournament_selection, success_rate=15, r

    print("Tournament average: " + str(tournament_test / run_tests)) # 35
```

Made with ❤️ in Maastricht