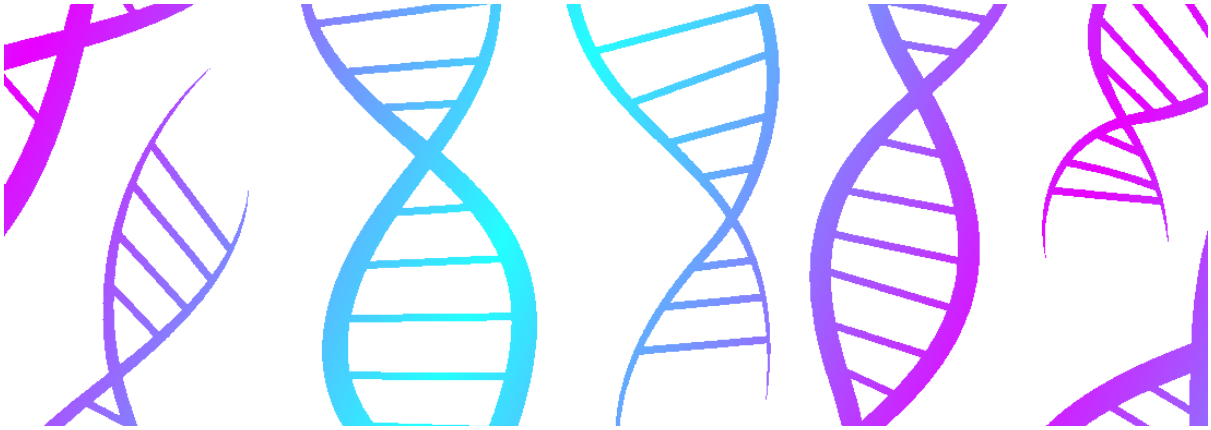

# CI Homework 4

✍️ Amin Ghasvari 97521432



## 🤔 Equation Solver

### Overview

We use a class to implement our implementation

```
class EquationSolver:

    def __init__(self, equation, population_size=80, search_range=(-2000, 2000), epoch=100):
        ...

    def give_birth(self):
        ...

    @staticmethod
    def get_sign(p1, p2):
        ...

    @staticmethod
    def produce_chromosome(p1, p2):
        ...

    def run(self):
        ...
```

This is how we want to use this class. We define the equation using a lambda expression. Search range actually defines maximum and minimum for variable values.

```
# Test equations
SAMPLE_EQUATION_1 = lambda x: 9 * x ** 5 - 194.7 * x ** 4 + 1680.1 * x ** 3 - 7227.94 * x ** 2 + 15501.2 * x - 13257.2

if __name__ == '__main__':
    root = EquationSolver(
        SAMPLE_EQUATION_1,
        search_range=(-300, 300),
```

```
        epoch=300
    ).run()
    print("Root: {}\nValue in {} is {}".format(root, root, SAMPLE_EQUATION_1(root)))
```

## Equation Solver Class

```
def __init__(self, equation, population_size=80, search_range=(-2000, 2000), epoch=100):
        """
        :param equation:
        :param population_size:
        :param search_range: at first we initially generate values and the values in this range.
        :param epoch:
        """

        # Assign initial values
        self.equation = equation
        self.population_size = population_size
        self.min_value, self.max_value = search_range
        self.epoch = epoch

        # Randomly initialize the first generation of the process
        self.population = []
        for i in range(self.population_size):
            v = np.random.randint(self.min_value, self.max_value, size=1)
            self.population.append(v)
```

After creating the object the main method to produce next generations is "run". Finally, it returns the root.

```
def run(self):
        for i in range(self.epoch):
            print("Generation's Index: {}".format(i))
            self.give_birth()

        return self.population[0][0]
```

This function is actually cross over. It randomly picks 2 elements in the population and creates their chromosome. Then, uses the half of first chromosome and the second half of the second one to create the child's chromosome.

```
def give_birth(self):
        results = [self.equation(person[0]) for person self.population]
        children = [self.population[np.argmin(np.absolute(results))]]

        while len(children) < self.population_size:
            ra, rb = [np.random.randint(0, self.population_size) for _ in range(2)]
            p1 = self.population[ra if results[ra] < results[rb] else rb]

            ra, rb = [np.random.randint(0, self.population_size) for _ in range(2)]
            p2 = self.population[ra if results[ra] < results[rb] else rb]

            s = self.get_sign(p1, p2)

            g1, g2 = self.produce_chromosome(p1, p2)
            children.append(np.asarray([s[0] * int(g1, 2), s[0] * int(g2, 2)]))

        self.population = children
```

Decoding and creating new chromosome are in this function.

```
@staticmethod
    def produce_chromosome(p1, p2):
        chromosome_1 = [format(abs(i), '010b') for i in p1]
        chromosome_2 = [format(abs(i), '010b') for i in p2]
```

```
        c = []
        for i, j in zip(chromosome_1, chromosome_2):
            for k, l in zip(i, j):
                c.append(k if k == l else str(np.random.randint(min(k, l), max(k, l))))
        string = ''.join(c)

        # divide it to the same pieces to return the result
        half = len(string) // 2

        return string[0:half], string[half:len(string)]
```

In my solution, there is no mechanism to calculate the error and ends the process. It is essential that we generate 80 generations to finish the process.

Here is the result. it usually finds the number 4  or 5 for sample equation.

```
Generation's Index: 1
....
Generation's Index: 295
Generation's Index: 296
Generation's Index: 297
Generation's Index: 298
Generation's Index: 299
Root: 5
Value in 5 is 0.2999999999992724
```

# 🏠 Salesman

## Overview

I didn't implement a mechanism to handle all kinds of graphs, Because the main purpose of this question wasn't about graphs.

So here I just created a tiny class that calculates the matrix for me.

```
class UndirectedWeightedGraph:

    def __init__(self, number_of_nodes, edges):
        self.matrix = []

        # Builds the graph matrix
        for i in range(number_of_nodes):
            self.matrix.append([0] * number_of_nodes)
        for (v1, v2, w) in edges:
            self.matrix[v1][v2] = w
            self.matrix[v2][v1] = w
```

Then I hardcoded the results in my code. There are some constants like

- "CROSS_PROB" is the possibility of mutation for the children.
- "RATIO" is the ratio for picking the parents.

```
INF = math.inf
POPULATION_SIZE = 10
CROSS_PROB = 0.02
RATIO = 0.5

if __name__ == "__main__":
    num_of_houses = 7
    houses = {0: '1', 1: '2', 2: '3', 3: '4', 4: '5', 5: '6', 6: '7'}
    distances = {
        0: [INF, 12, 10, INF, INF, INF, 12],
        1: [12, INF, 8, 12, INF, INF, INF],
```

```
        2: [10, 8, INF, 11, 3, INF, 9],
        3: [INF, 12, 11, INF, 11, 10, INF],
        4: [INF, INF, 3, 11, INF, 6, 7],
        5: [INF, INF, INF, 10, 6, 0, 9],
        6: [12, INF, 9, INF, 7, 9, INF]
    }

    problem = SalesmanSolver(list(range(num_of_houses)),
                             len(houses),
                             lambda x: decode_node(x),
                             lambda y: fit_salesman(y))
    cont = 0
    while cont <= POPULATION_SIZE:
        genetic_algorithm_t(problem, 2, min, 200, 100, RATIO, CROSS_PROB)
        cont += 1
```

## Evolution process

The main function is "genetic_algotithm"

```
def genetic_algorithm(problem, k, opt, num_gen, size, ratio_cross, prob_mutate):
    population = initial_population(problem, size)
    n_parents = round(size *    ratio_cross)
    n_parents = (n_parents if n_parents % 2 == 0 else n_parents - 1)
    n_directs = int(size - n_parents)

    for _ in range(num_gen):
        population = new_generation_t(problem, k, opt, population, n_parents, n_directs, prob_mutate)

    best_chromosome = opt(population, key=problem.fitness)
    genotype = problem.decode(best_chromosome)
    print("Chromosome: {}\nSolution: {}".format(best_chromosome, (genotype, problem.fitness(best_chromosome))))
    return genotype, problem.fitness(best_chromosome)
```

At first it delimits the parents and then it starts to produce new generations and the best chromosome is the one with less cost. So the operation is minimum function.

In the next step, to create new children, it selects the parents, calculates the cross over and performs the mutation.

```
def cross_parents(problem, parents):
    children = []
    for i in range(0, len(parents), 2):
        children.extend(problem.crossover(parents[i], parents[i + 1]))
    return children


def tournament_selection(problem, population, n, k, opt):
    winners = []

    for _ in range(int(n)):
        elements = random.sample(population, k)
        winners.append(opt(elements, key=problem.fitness))
    return winners


def mutate(problem, population, prob):
    for i in population:
        problem.mutation(i, prob)
    return population


def new_generation_t(problem, k, opt, population, n_parents, n_directs, prob_mutate):
    directs = tournament_selection(problem, population, n_directs, k, opt)
    crosses = cross_parents(problem,
                            tournament_selection(problem, population, n_parents, k, opt))
    mutations = mutate(problem, crosses, prob_mutate)
    new_generation = directs + mutations
```

```
    return new_generation
```

Fitness is defined to have less cost.

Here is the cross over function. As you see, it picks a random number of bits (they are not bits though) from each one of parents.

```
def crossover(self, parent1, parent2):
        def process_gen_repeated(copy_child1, copy_child2):
            count1 = 0
            for gen1 in copy_child1[:pos]:
                ...

            count1 = 0
            for gen1 in copy_child2[:pos]:
                ...

            return [child1, child2]

        pos = random.randrange(1, self.individuals_length - 1)
        child1 = parent1[:pos] + parent2[pos:]
        child2 = parent2[:pos] + parent1[pos:]

        return process_gen_repeated(child1, child2)
```

The result

```
...
Chromosome: [3, 1, 0, 2, 4, 6, 5]
Solution: (['4', '2', '1', '3', '5', '7', '6'], 63)
```