University of Sarajevo
Faculty of Electrical Engineering
Department of Data Science and Artificial Intelligence

Laboratory Exercise 9

Artificial Intelligence

Authors: Admir Ahmespahić, Nedim Bečić
Professor: Izudin Džafić

Contents

1 Goal of the Exercise

The goal of this laboratory exercise is for students to apply the **genetic algorithm** for solving an optimization problem. For this, students will have to implement a version of genetic algorithm described below. The genetic algorithm will then be used for finding the minimum and maximum of given functions.
By completing the task, students will understand the implementation and functionality of all the parts of the genetic algorithm, learn about different methods used to run the algorithm and how to choose the right method for a given task. By completing the bonus task, students will also understand how to visualize the solution at different steps of the code execution.

2 Theoretical Introduction

The genetic algorithm works by simulating the process of natural evolution. Every iteration is defined by it's population. Every member of the population has certain traits that are used to calculate the fitness function. For example, if the fitness function is some mathematical function $f(x)$, every member of the population will have the trait x.
After every iteration, the entire population (parents) is replaced by the new population (children) which inherit traits of their parents. In order to ensure that every new iteration gives better, parents with better fitness function value will have more chance of passing on their traits.
To better explain these functionalities, the algorithm is divided in several steps.

2.1 Initialization

The First step is to initialize the starting population for the genetic algorithm. For this, we need to define how many members the population has and a way to generate these members. This is often done by defining random or pseudo-random traits. If we know that the solution to our optimization problem is in a certain area, we can concentrate the population in that area.

2.2 Fitness

After initializing our population, we need to define the fitness function. This is the function that we want to optimize using the genetic algorithm. As explained, this function will make sure that every iteration gives better and better results.

2.3 Selection

Selection is used to choose which individuals will pass their traits to the next generation. There are many ways to do this part and it will greatly influence the progression of the algorithm. On of the ways to select the individual is to take a few population members at random and select the one with the best traits (this simulates competition). A small number of individuals is chosen for competition in order to keep the diversity of the population.

2.4 Crossover

After we select two population members, we need to determine how the traits will be passed to the next generation. For this we use crossover. Usually the traits are inherited randomly using one-point, two-point, or uniform crossover.

2.5 Mutation

Mutation is used to make small changes to child solution in order to explore new regions of the problem field which crossover can't cover. This can be done by flipping a random bit (bit mutation), adding random noise (real-valued mutation)... For finding the optimum of a mathematical function, we often use real-valued mutation).

2.6 Termination

After a some termination condition is met, we can finish the code execution. We will usually terminate the code execution if we find an acceptable solution, if new iterations don't improve the solution enough, or if we reach the maximum number of iterations.

3 Main Task

3.1 Task 1: Create genetic algorithm solver

In this task, you will have to implement the class *Solver* with the following interface:

```
class Solver {
    int _populationSize;
    std::vector<double>_minX;
    std::vector<double>_maxX;
    std::vector<std::vector<double> >_population;
    std::function<double(std::vector<double>)>_fitness;
public:
    Solver(int populationSize, std::vector<double> minX, std::vector<
   double> maxX);
    void setFitness(std::function<double(std::vector<double>)>fitness);
    void Initialize();
    std::vector<double> Select();
    std::vector<double> Crossover(std::vector<double>a, std::vector<
   double>b);
    std::vector<double> Mutation(std::vector<double>c);
    void solve(double &y, std::vector<double> &x);
};
};
```

- **_populationSize** - the size of the population
- **_minX, _maxX** - minimum and maximum values each trait can have
- **_population** - population where each member of the vector is a vector of the population traits
- **_fitness** - fitness function
- **setFitness** - sets the fitness function

- **Initialize** - run the logic for population initialization
- **Select** - return the selected member of the population
- **Crossover** - return the child solution from two parent solutions
- **Mutation** - mutate the child solution c
- **solve** - run the genetic algorithm and return the optimal value y and the solution x.

3.2 Task 2: Test the genetic algorithm

In this task you will need to test the genetic algorithm with these problems:

- Find the minimum of $x^2 + (y-2)^2$ in range $x \in (-10, 10)$ and $y \in (-10, 10)$
- Find the maximum of e^{-x^2} in range $x \in (-15, 15)$
- Find the minimum of $\sum_{i=1}^{5}(x_i - i)^2$ in range $x_i \in (-10, 10)$

4 Bonus Task

4.1 Task 1: Visualization

Implement the same algorithm using natID. For the first function in Task 2, display the location of every population member in a 2D grid for every iteration. Use this to determine if your algorithm is always moving the solution in the right direction.