Exercise on Genetic Algorithm

Download ga_studen.py before working on this exercise.

Overall workflow

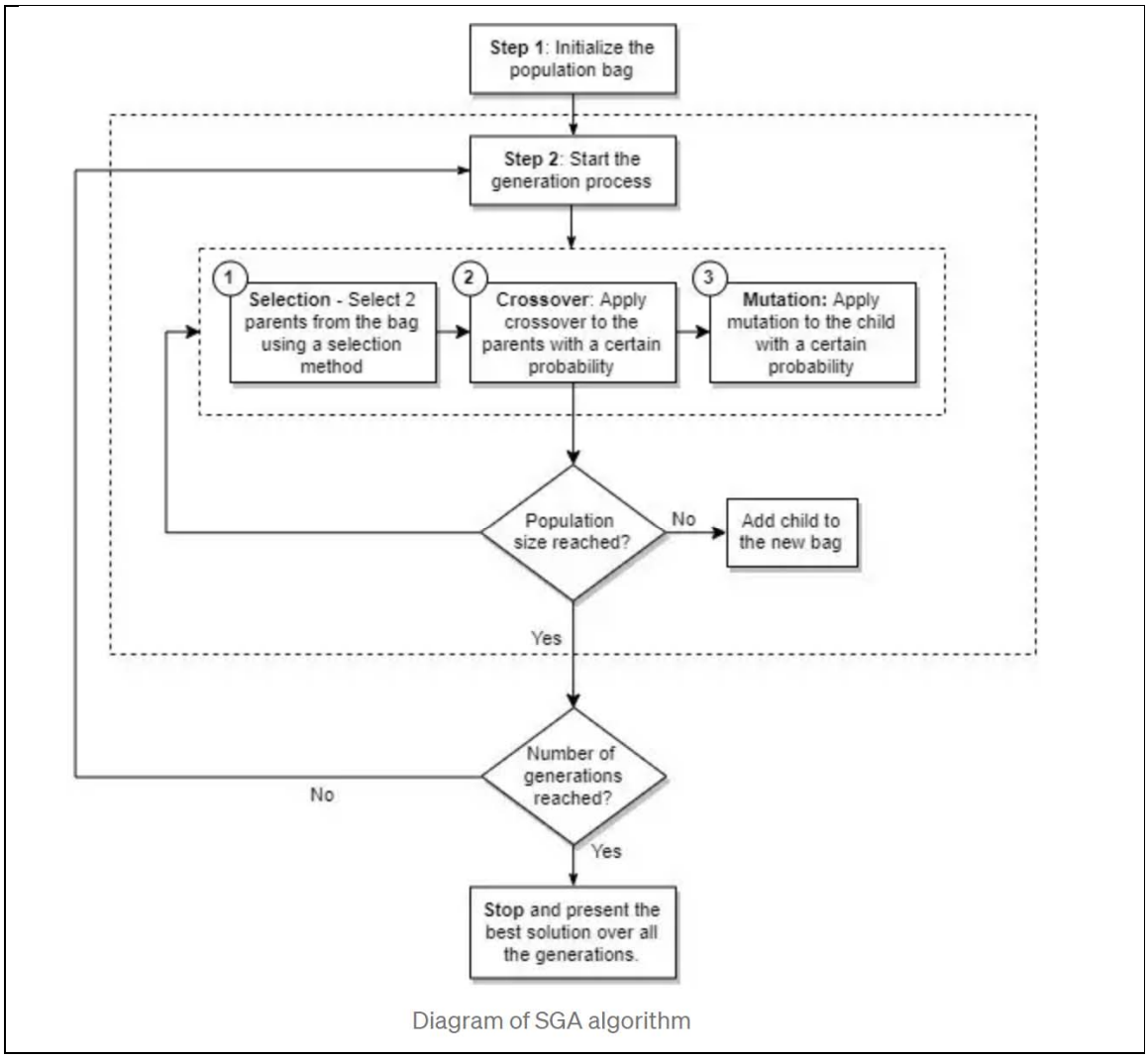


Diagram of SGA algorithm

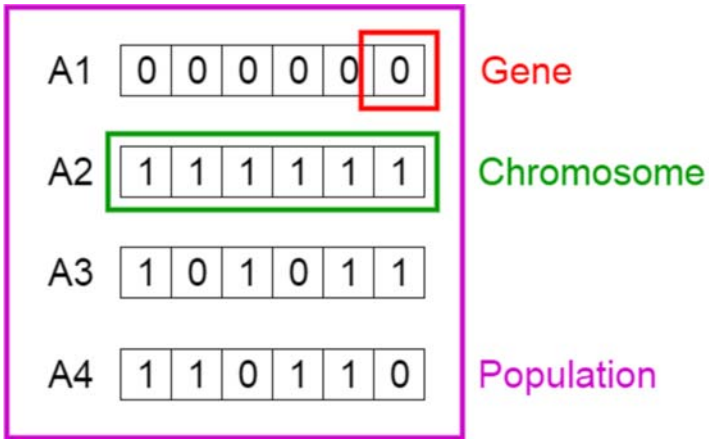
Problem's context - Simplified version of Travel Salesman Problem (TSP)

There are five cities. We want to create a route that connects all the points in the best possible way, this is done by minimizing the distance between the cities.

Initial Population:

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution).



Parameters' implementation

The encoding of the solution: In this case, the encoding is well represented by the order of the cities to visit (the route). One solution could be for instance: [3, 5, 1, 4, 2]. Where we visit first city number 3 followed by the 5 and then 1 and 4, and finally the city number 2. In this case, it is important to notice that we cannot repeat cities in the encoding since we cannot visit a city twice.

The decision variables: Since we have 5 cities to choose, our decision variables in this problem are those five cities called by their indexes.

decision_variables = [0,1,2,3,4]

we define a variable **distance** - contains all the distances (or cost of travel between cities) of every possible connection between cities; it is a squared matrix with diagonal of zeros since the cost of visit the same city is zero. Let's assume that the distance matrix is defined as follows:

```
distance =
[[ 0.00 28.02 17.12 27.46 46.07]
[28.02 0.00 34.00 25.55 25.55]
[17.12 34.00 0.00 18.03 57.38]
[27.46 25.55 18.03 0.00 51.11]
[46.07 25.55 57.38 51.11 0.00]]
```

Cost of travel from city 3 to city 4	51.11
cost of travel from 4 to 2	57.38

Cost of travel from 2 to 0	17.12
Cost of travel from 0 to 1	28.02

Download ga.py before working on the exercises.

Exercise 1:

Define a function ***initialize()***. This function will generate a population of individuals. You can create them randomly using our decision_variables variable and shuffle it in order to generate a random chromosome (individual) that is included in the population bag (represented by the variable pop_bag) until we reach the population size of 10. Repetition of chromosome is allowed.

Exercise 2:

Fitness Function

The fitness function is a function which takes a candidate solution to the problem as input and produces as output how "fit" or how "good" the solution is with respect to the problem in consideration.

In this problem context, we can define the fitness function by the sum of all distances in the encoding. And the idea is to minimize this value.

Define a function **fitness_function(solution).** This function will return the total distance.

Suppose we have the route [3 4 2 0 1]. Then, the total distance for it will be the cost of travel from city 3 to city 4 plus the cost of travel from 4 to 2, plus the cost of travel from 2 to 0, plus the cost from 0 to 1. This is 51.11 + 57.38 + 17.12 + 28.02 = 153.63.

Evolutionary Process

Now that we have the initial population established, then we can start the evolutionary process of creating the generations. Each generation is going to be represented by each step in a for loop in python. Inside each generation, we'll need to perform three basic genetic operators - Selection, crossover, and mutation.

Selection

The process of selection is the first step to create a new population for the next generation. There are three well-known methodologies to approach this: the Roulette Wheel method, the Rank method, and the Tournament Size method.

We will be using the Roulette Wheel method in this implementation.
As its name suggests, the Roulette Wheel method is basically a selection process where we pick one individual from our population bag by “spinning a Roulette Wheel”. In this case, the probability of each individual being selected depends directly on its fitness value.

Evaluation function

The evaluation function eval_fit_population has been coded for you. The evaluation function eval_fit_population will return the fitness weights for each individual, and each weight is going to represent the probability of an individual of being selected in that particular bag.

The implementation is done using the Roulette Wheel method; Fitter individual will has greater pie on the wheel (shorter distance will have a higher percentage).

The implementation is done in the following manner:

1. Determine the maximum distance among individuals in the population.
2. Calculate the difference between each individual's distance and the maximum distance. A larger difference indicates a fitter individual, as it has a smaller distance.
3. Compute the weighted sum by adding up all the values obtained in step 2.
4. Tabulate the probability of each individual based on their weighted sum relative to the total weighted sum.

```
# --- EVALUATE FITNESS IN POPULATION BAG ---
def eval_fit_population(pop_bag):
    result = {}
    fit_vals_lst = []
    solutions = []
    for solution in pop_bag:
        fit_vals_lst.append(fitness_function(solution))
        solutions.append(solution)
    result["fit_vals"] = fit_vals_lst
    min_wgh = [np.max(list(result["fit_vals"]))-i for i in
list(result["fit_vals"])]
    result["fit_wgh"]  = [i/sum(min_wgh) for i in min_wgh]
    result["solution"] = np.array(solutions)
    return result
```

Exercise 3:

Define and implement the function pickOne(pop_bag).

The function pickOne takes the population bag as input and evaluates the fitness function for each element in order to select one individual randomly but based on its fitness.

Hint: This function will make use of eval_fit_population(..) function during the evaluation process.

Crossover

The crossover operator is analogous to reproduction and biological crossover. In this, more than one parents are selected and offspring is produced using the genetic material of the parents. Only crossover can combine information from two parents.

There are many methods of parent's recombination – single crossover point, Two-point crossover, Order 1 Crossover etc

Parent 1 : [3 2 0 1 4]

Parent 2: [0 2 1 3 4]

1. Take the first parent and **select two random cutpoints** between their gens (elements). Let's say in our example that we've chosen the following cutpoints: [3 2 | 0 1 | 4].
2. The next step will be to **place those elements that are inside the cutpoints and place them into the new child.** So it will be something like [X X | 0 1 | X], where the "X" represents empty gaps that we'll fill in the next step.
3. Now, **go elements by element in Parent 2 and try to fill the child asking whether that element is already in the child or not.** In our example, we start by the first element in parent 2 which is zero, and because it is already included in the child, we don't mind it and we move to the next one that is number 2. Since number 2 is not included in the child then we put it inside the child and position it in the first empty gap ("X"). So we have [2 X | 0 1 | X]. We continue this process until we end up with the child **[2 3 0 1 4]**, that as you can see does not have any repeated number and is semantically correct.

Exercise 4:
Define a function crossover(solA, solB) using Order 1 crossover.

Mutation
Once we create the child using crossover, we can start performing the mutation operator. There are several ways to achieve this; we can use swap, scramble, insert, reverse or any other kind of mutation that fits with our design.

Exercise 5:
Define a function mutation(sol): using the "swap" mutation.

That is essentially when we take two elements at random positions of the child and we swap these positions in the arrange. For instance, if we have the child [2 0 1 4 3] and we chose randomly positions 1 and 4, then the mutated individual would be [4 0 1 2 3], where we swapped the number 2 at position 1 for the number 4 at position 4.

Merging everything
To accomplish this task, we simply need to iterate over each step using a for loop and inside each step we need to call the genetic operators that we already established in order to create the new bag, always keeping track of the best individual so far.

Optional exercises:

1. Implement a different "Selection" method.
2. Implement a different "cross-over" method.
3. Suggest an implementation for "elitism"?

Appendix:

Exercise 1: Sample output

```
[[2 3 1 0 4]
 [3 2 1 4 0]
 [1 4 3 0 2]
 [0 4 2 3 1]
 [1 0 3 2 4]
 [3 1 4 0 2]
 [3 0 1 2 4]
 [3 4 0 1 2]
 [3 0 2 4 1]
 [1 2 0 3 4]]
```

Exercise 2: Sample output

```
chromosome [4 1 2 0 3] ,fitness: 104.13
chromosome [4 0 3 2 1] ,fitness: 125.56
chromosome [1 3 4 0 2] ,fitness: 139.85
chromosome [1 3 0 2 4] ,fitness: 127.51000000000002
chromosome [3 4 2 0 1] ,fitness: 153.63000000000002
chromosome [1 0 3 2 4] ,fitness: 130.89000000000001
chromosome [3 4 2 0 1] ,fitness: 153.63000000000002
chromosome [4 2 3 1 0] ,fitness: 128.98
chromosome [2 4 1 3 0] ,fitness: 135.94
chromosome [0 2 3 4 1] ,fitness: 111.81
```