Genetic Algorithm: Report TP#1

Due on Thu, November 12, 2022

Master 2 GSI

Tarek Berkane

1 Purpose of project

Understanding how genetic algorithms **GA** works, through implementation of this algorithm step by step using python language.

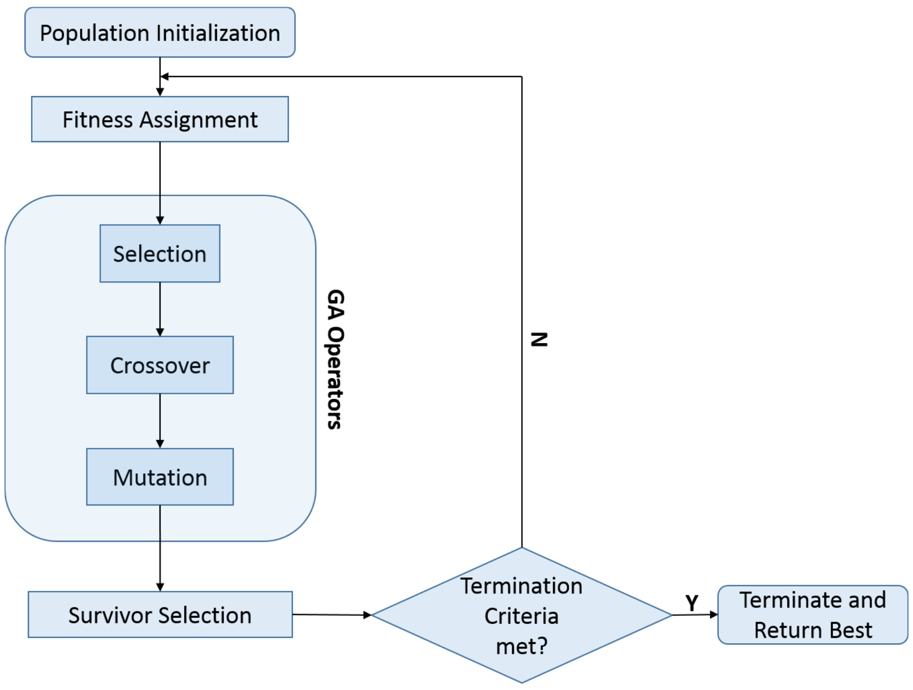
2 Expected goal

in general the main but from this homework is to understand GA and its steps, through implementing it in real problem which is trying to solve travel sales problem using python language and it's different libraries that help us to simplify the process of creating GA steps.

2.1 Understading GA

Genetic algorithms are a family of search algorithms inspired by the principles of evolution in nature. By imitating the process of natural selection and reproduction, genetic algorithms can produce high-quality solutions for various problems involving search, optimization, and learning. At the same time, their analogy to natural evolution allows genetic algorithms to overcome some of the hurdles that are encountered by traditional search and optimization algorithms, especially for problems with a large number of parameters and complex mathematical representations.

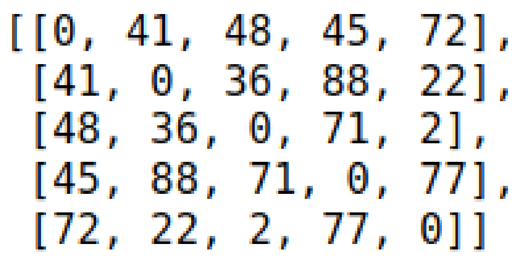
2.2 Steps of Genetic algorithme



2.3 Creating the initial population

```python
def generate_cities(number_of_cities: int):
    cities = []
    for i in range(number_of_citiies):
        city = []
        for j in range(number_of_citiies):
            if i == j:
                city.append(0)
            else:
                city.append(randint(1, 100))
        cities.append(city)

    for i in range(number_of_citiies):
        for j in range(number_of_citiies):
            cities[i][j] = cities[j][i]

    return cities
```

`generate_cities` is a function that accept **integer parameter** (`number_of_cities`) as input and return a matrix of $n \times n$ size .
Example:
`generate_cities(5)`



```
[[0, 41, 48, 45, 72],
 [41, 0, 36, 88, 22],
 [48, 36, 0, 71, 2],
 [45, 88, 71, 0, 77],
 [72, 22, 2, 77, 0]]
```

Note: index of rows and columns are considered cities and the value inside matrix as considered as distance between 2 cities.

2.4 Generate of population

2.4.1 Generate of one path

```python
def generate_individuale(cities_number: int):
    if not start_node:
        start_node = randint(0, cities_number - 1)

    cities: List = list(range(cities_number))
    shuffle(cities)
    return cities
```

`generate_individual` accept one paramter **number of citeis**, and return a path generated from cities matrix
Example:
`generate_individuale(cities)`
Will return `[1,3,4,5,0,2]`

2.4.2 Generate of population

```python
def generate_first_population(individuals_number, cities_number: int):
    population = []
    for i in range(individuals_number):
        population.append(generate_individuale(cities_number))

    return population
```

`generate_first_population` will accept two parameters number of individual we want to generate and cities number. Inside this function we use `generate_individuale` function many times to generate each time a new path and then add it to a list of population. at the end of this process we return a list of population.
Example:
`generate_first_population(4,5)`
will return

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

2.5 Evaluation

2.5.1 Calculation of distance cities

```python
def path_total_distance(cities, path):
    distance = 0
    for i in range(len(path) - 1):
        city_1 = path[i]
        city_2 = path[i + 1]
        distance += cities[city_1][city_2]

    return distance
```

`path_total_distance` calcule distance between two cities, using citeis matrix.
Example:
`path_total_distance(citeis, [0,1,2,3,4])`
this will be calculated as follow:
distance = cities[0][1]+ cities[1][2]+ cities[2][3]+ cities[3][4]

2.6 Calculation of fitness for each city

```python
def fitness_reverse_devide(cities, path) -> float:
    return 1 / path_total_distance(cities, path)
```

`fitness_reverse_devide(cities, [1,2,3,4,5])` will calculate as following $\frac{1}{distance(path_i)}$. revere mean that cities will large distance will result with a poor fitness and cities with short path will result with a high fitness.

2.7 Calculate fitness for all population

```python
def evaulate_population(cities, population):
    evaluation = []
    for i in range(len(population)):
        indivudal = population[i]
        evaluation.append(fitness_reverse_devide(cities, indivudal))

    return evaluation
```

`evaluate_pupulation` will calculate fitness for all population using `fitness_reverse_devide` function.

2.8 Selection

2.8.1 Calculation of probability for each individual

```python
def calc_relative_portion(p_evaluation: List[float]):
    p_probability = []

    sum_of_score = 0
    for individual_evaluation in p_evaluation:
        sum_of_score += individual_evaluation

    for individual_evaluation in p_evaluation:
        proba = individual_evaluation / sum_of_score
        p_probability.append(proba)

    return p_probability
```

`calc_relative_portion` accept fitness list of population as input and convert it to probability as output
Example:
`calc_relative_portion([10, 20, 50, 20])`
will return
`[0.1, 0.2, 0.5, 0.2]` sum equale to 1

2.8.2 Selection based on probability

```python
def select_by_probability(p_probability, probability):
    assert 0 <= probability <= 1

    some = 0
    for index, individual_probability in enumerate(p_probability):
        some += individual_probability
        if probability < some:
            return index

    raise Exception(f"Proba out of range some={some}, proba={probability}")
```

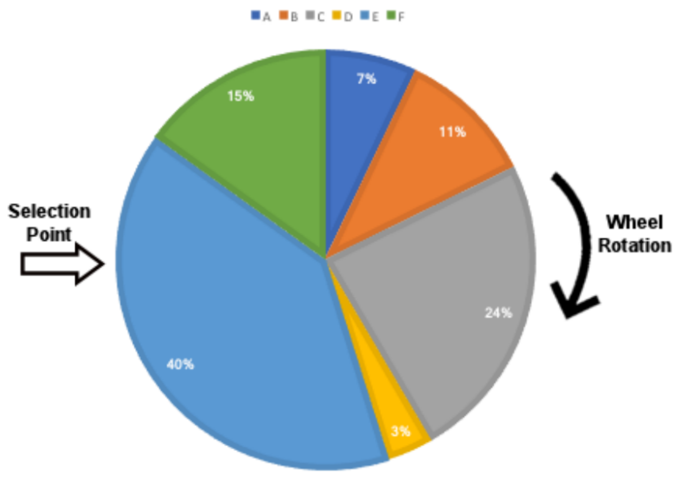
`select_by_probability` return a random index bassed on probability of each individual.
Note: individuals with high probability will have a more chance to be selected.
Example:

`select_by_probability([0.2,0.5,0.3],0.3)`
this will return 0.5

2.8.3 Roulette wheel selection

```python
def roulette_wheel_selection(
    population: List, p_evaluation: List[int], parents_number_proba: float
):
    population_copy = population.copy()
    p_evaluation_copy = p_evaluation.copy()

    number_individual = len(population)
    parent_number = int(number_individual * parents_number_proba)

    selected_parent = []
    for i in range(parent_number):
        p_probabilty = calc_relative_portion(p_evaluation_copy)
        random_value = random()
        index_selected_parent = select_by_probability(p_probabilty, random_value)

        selected_individual = population_copy[index_selected_parent]
        selected_parent.append(selected_individual)
        population_copy.pop(index_selected_parent)
        p_evaluation_copy.pop(index_selected_parent)

    return selected_parent
```

`roulette_wheel_selection` accept three parameters, `population` (list of individuales), `p_evaluation` a list contain fitness for each individual, `parents_number_proba` is the probabilty of selection individual (if proba is 0.8 that's mean 80% of population will be selected to be a parent or $\frac{8}{10}$ individuals will be seleted)



`roulette_wheel_selection` is based on `select_by_probability` that's mean each time we want to select individual using probability we need to call `select_by_probability` in order to selecte item randomly.

2.9 Crossover

```python
def random_crossover_selection(parent: List):
    if len(parent) < 2:
        return []

    parent_copy = parent.copy()

    generated_children = []

    for i in range(len(parent)):
        shuffle(parent_copy)
        selected_parent_1, selected_parent_2 = parent_copy[:2]
        children = ordered_crossover(selected_parent_1, selected_parent_2)
        generated_children.extend(children)

    return generated_children
```

`random_crossover_selection` accept list of parents, the selection of parent to implement crossover. the function start by selection 2 parents randomly and call `orderd_crossover` algorithm.

2.9.1 Orderd crossover

```python
def ordered_crossover(individual_1: List, individual_2: List) -> tuple:
    assert len(individual_1) == len(individual_2)

    cut_length = 2

    start_cut_point = 2
    end_cut_point = start_cut_point + cut_length

    new_chromosome_1 = generate_new_chromosome(
        individual_1, individual_2, start_cut_point, end_cut_point
    )

    new_chromosome_2 = generate_new_chromosome(
        individual_2, individual_1, start_cut_point, end_cut_point
    )

    return (new_chromosome_1, new_chromosome_2)
```

`ordered_crossover` is one of many algorithms that can be used to implement crossover. `orderd_crossover` will assure that all new children generated by this crossover are valide children. this algorithm require to implement crossover of two segment and then reorder the rest of child features based on a new segments. and try to keep the same order of features as possible.
Example `[1,0,3,4,2] crossover with [2,0] at index 2`
Result `[_,_,2,0,_] > [_,_,2,0,1] > [3,_,2,0,1] > [3,4,2,0,1]`

2.10 Mutaion

```python
def individual_mutation(indv: List):
    gene_1 = randint(0, len(indv) - 1)
    gene_2 = randint(0, len(indv) - 1)
    indv[gene_2], indv[gene_1] = indv[gene_1], indv[gene_2]


def population_mutation(population, mutation_proba):
    assert 1 >= mutation_proba >= 0
    population_length = len(population)
    individual_length = len(population[0])
    mutation_ratio = round((population_length * individual_length) *
    ↪  mutation_proba)
    population_copy = deepcopy(population)
    for i in range(mutation_ratio):
        selected_parent = choice(population_copy)
        individual_mutation(selected_parent)

    return population_copy
```

`individual_mutation` will implement swaping mutation between 2 features in the same individual.
Example `[1,2,3,4,5] mutaion between index 1 and 4`
Result `[1,5,3,4,2]`
`population_mutation` will try to choose randomly and based on `mutation_proba` an individual or individuals to implment a swaping mutation. this function will call inside of it `individual_mutation`

2.11 Selecting best individuals based on fitness

2.11.1 Ordering population

```python
# sort method
def second_element(elem):
    return elem[1]


def order_population(population: List[List[int]], p_evaluation: List[int]):
    population_with_evaluation = [
        [individual, evaluation]
        for individual, evaluation in zip(population, p_evaluation)
    ]

    population_with_evaluation.sort(key=second_element, reverse=True)

    ordred_p = strip_population(population_with_evaluation)
    return ordred_p
```

`order_population` will oder a list of population based on `p_evaluation` list.

2.11.2 Selecting best individuals

```python
def get_best_individual(
    population: List, p_evaluation: List[float], individuals_number: int
):
    ordred_p = order_population(population, p_evaluation)
    return ordred_p[:individuals_number]
```

`get_best_individual` call ordering function `get_best_individual` and return a list with a max size of `individuals_number`.

2.12 Helper functions

```python
def remove_duplicate_individual(population: List[List[int]]) -> List:
    new_population = []

    for individual in population:
        not_exit = True
        for selected_indv in new_population:
            if selected_indv == individual:
                not_exit = False
                break
        if not_exit:
            new_population.append(individual)

    return new_population



def validate_city_not_revisited(individual: List[int]) -> bool:
    """
    Return True if there is no duplicate city in path
    e.g [1,2,3,4,5,1] >> return False
        [1,2,3,4,5,6] >> return True
    """
    indiv_set = set(individual)
    return len(individual) == len(indiv_set)




def remove_parent(population,parent):
    p = []
    for ind in population:
        if ind not in parent:
            p.append(ind)
    return p
```

3 Example

```python
CITIES_NUMBER = 20
INDIVIDUALS_NUMBER = 50
SELECTED_PARENT = int(INDIVIDUALS_NUMBER / 2)
MUTATION_PROBA = 0.1
SELECTION_PROBA = 0.1


cities = generate_cities(CITIES_NUMBER)
init_population = generate_first_population(INDIVIDUALS_NUMBER, CITIES_NUMBER)

current_population = init_population
population_evaluated = evaulate_population(cities, current_population)
for i in range(100):
    selected_parent = roulette_wheel_selection(
        current_population, population_evaluated, SELECTION_PROBA
    )
    new_children = random_crossover_selection(selected_parent)
    current_population.extend(new_children)
    # remove_parent(population,selected_parent)

    population_mutated = population_mutation(current_population, 0.07)
    current_population.extend(population_mutated)


    valid_indvidual = remove_duplicate_individual(current_population)

    p_evaluated = evaulate_population(cities, valid_indvidual)
    best_individual = get_best_individual(current_population, p_evaluated,
    ↪  INDIVIDUALS_NUMBER)
    current_individuals = best_individual


    # print(best_individual[0], path_total_distance(cities, best_individual[0]))

    current_population = best_individual
    population_evaluated = evaulate_population(cities, current_population)

print("First generation:")
print(init_population[0],path_total_distance(cities,init_population[0]))
print("Last generation ")
print(best_individual[0],path_total_distance(cities,best_individual[0]))
```

3.1 Result of execution

```
First generation:
[2, 17, 11, 9, 19, 16, 13, 3, 1, 12, 18, 7, 15, 0, 14, 10, 4, 6, 5, 8] 1094
Last generation
[4, 8, 2, 13, 12, 17, 14, 1, 6, 10, 5, 18, 9, 15, 11, 3, 0, 7, 19, 16] 661
```

4 Conclusion

In this homework we did Introduce GA, and we explore different steps of it. We then implemented GA to solve TSP using python. As result we conclude that is using GA can help us to reduce the time to find approximately optimal value. Rather than using classic algorithms which will take a long time to finish.

References

[1] Eyal Wirsansky, *Hands-On Genetic Algorighms with Python*, packets (2020).

[2] Wikipedia, Genetic algorigthm, `https://en.wikipedia.org/wiki/Genetic_algorithm`