Maryam Ibrahim Mahmoud
March 2024

Genetic Algorithms for Optimization

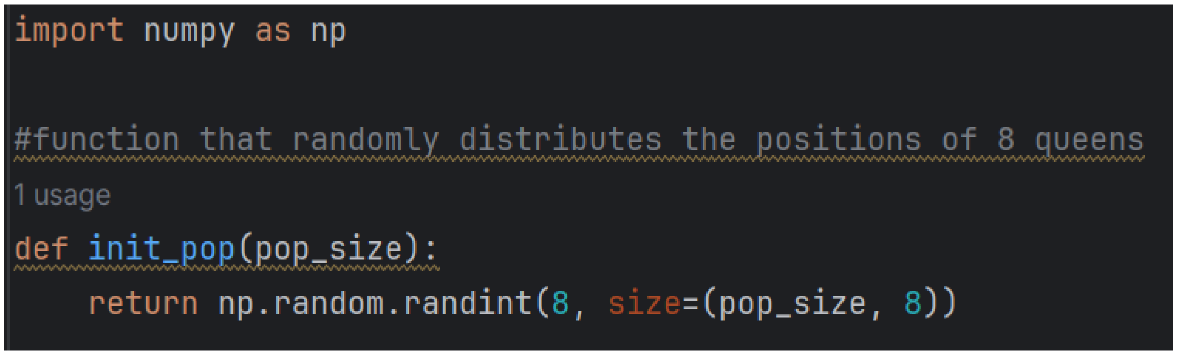
1. Introduction

Genetic algorithms are a class of optimization algorithms inspired by the process of natural selection and genetics. They are widely used to solve complex optimization problems where traditional methods struggle to find optimal solutions. In this paper, we'll be discussing the implementation of the Genetic Algorithms using Python language relating to the 8 Queens problem.

2. Genetic Algorithm Components

1. **Initialization:** A population of candidate solutions is randomly generated.

   The next line of code illustrates how to initialize the populations of a sample by choosing random placement for the 8 queens example. It creates n number of lists each list contains 8 elements(columns) which represent the placement of the queens and pop_size which represents the number of populations (rows) that'll be used in the sample.

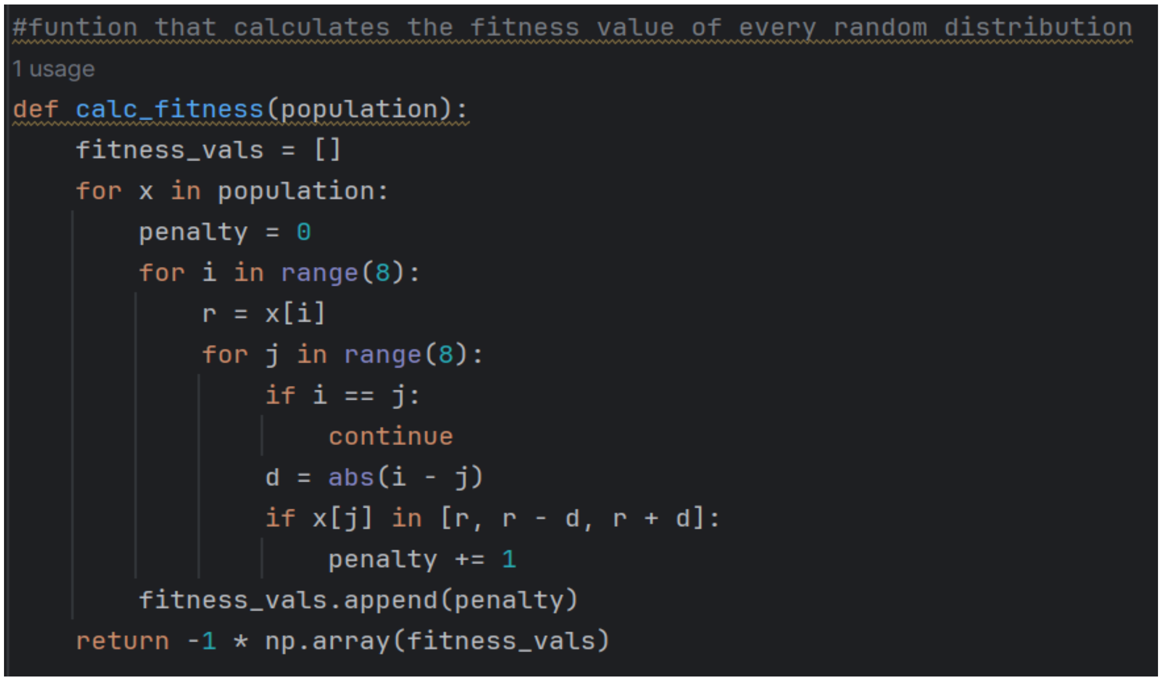


```
import numpy as np

#function that randomly distributes the positions of 8 queens
1 usage
def init_pop(pop_size):
    return np.random.randint(8, size=(pop_size, 8))
```

2. **Fitness Evaluation:** Each solution's fitness is evaluated using a fitness function that measures its quality.

The following code illustrates how to calculate the fitness. **Fitness** is simply the key component for the genetic algorithm. It measures how well a solution is (the higher the value the better the solution). In the 8 queens problem, we work with the **Penalty** which is the sum of the number of queens a single queen can threaten in a single sample and it has an inverse proportion relation with the fitness; which means that the lower value of penalty, the better solution. So to solve this, we'll assign the value of the fitness to -ve the value of the penalty. This means that later the best solution will be when both have the minimum value (point of intersection) which is zero.

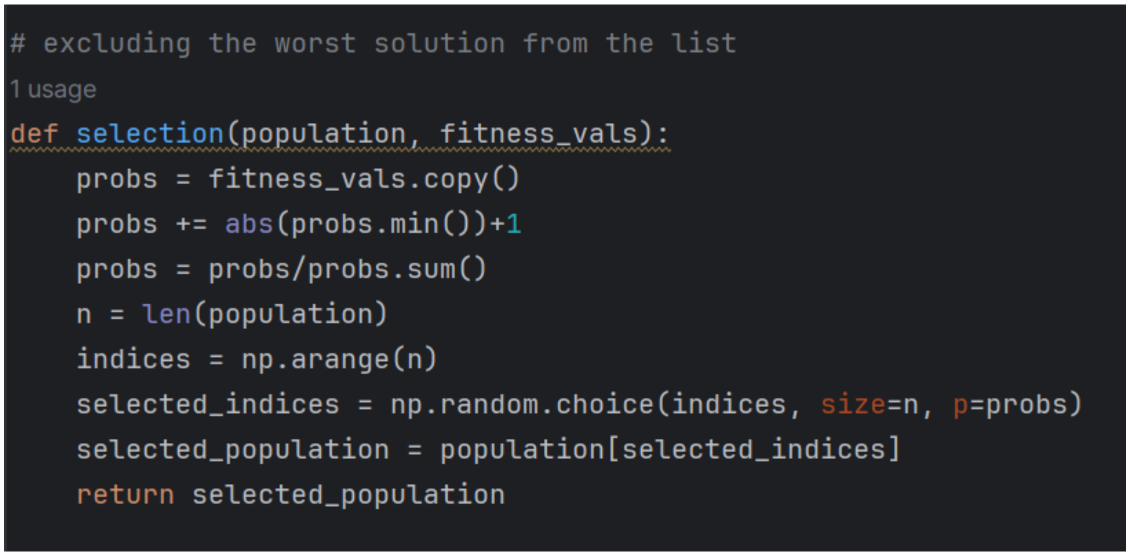
Here, we start to check whether the row or diagonal of the position of each queen contains another queen or not. If so, we'll increase the value of the penalty by one. R represents the row of the current queen, I represents the index, j represents the movement from the initial queen place, and d represents the position that'll be checked whether it has a queen or not.
This function will return a list of fitness values of each individual in a population(-1*list of fitness values because I stored the value of the penalty in the list previously so we should reverse them).



```
#funtion that calculates the fitness value of every random distribution
1 usage
def calc_fitness(population):
    fitness_vals = []
    for x in population:
        penalty = 0
        for i in range(8):
            r = x[i]
            for j in range(8):
                if i == j:
                    continue
                d = abs(i - j)
                if x[j] in [r, r - d, r + d]:
                    penalty += 1
        fitness_vals.append(penalty)
    return -1 * np.array(fitness_vals)
```

3. **Selection:** Solutions are selected for reproduction based on their fitness values, with fitter solutions having a higher chance of being selected.

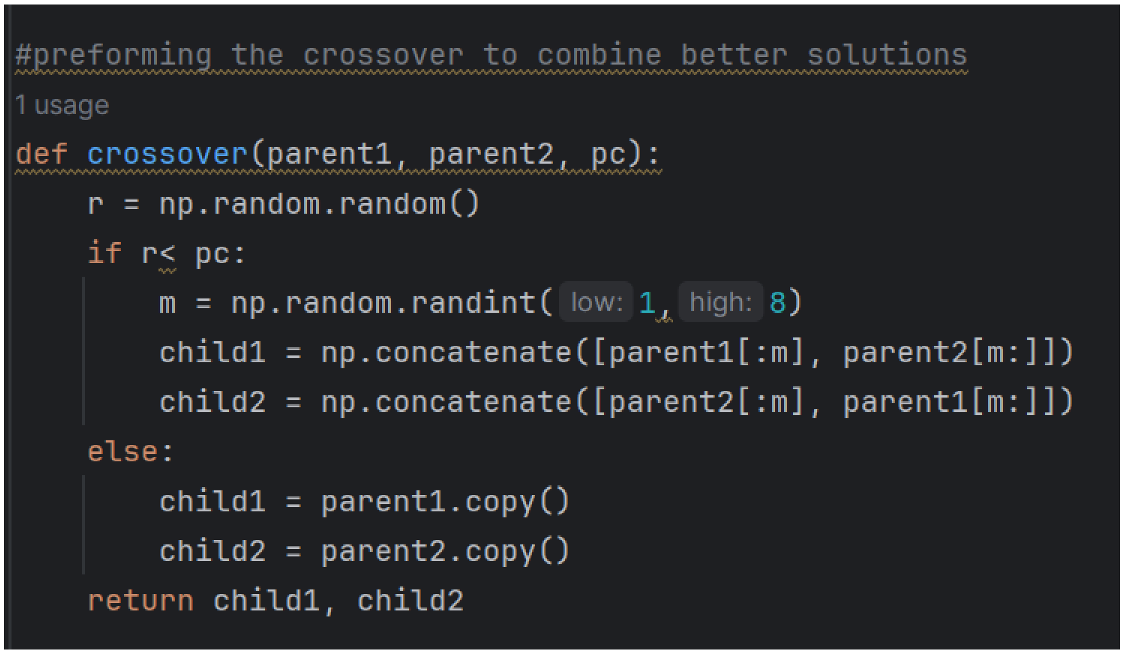
This function implements the selection process, where individuals from the population are selected to perform reproduction based on their fitness value. First, we'll take a copy of the fitness values list and then we'll add the absolute value of the minimum fitness value to all the other value plus 1, why?? Bec if we added the min. value to its absolute value, we'll get zero which will lead to having a wrong optimal solution.
Then, we'll divide every value by the sum of the total new fitness values to get the probability of each(normalization). Then, we'll select individuals randomly according to the highest probabilities. These individuals are the ones that will represent the reproduction later.



```
# excluding the worst solution from the list
1 usage
def selection(population, fitness_vals):
    probs = fitness_vals.copy()
    probs += abs(probs.min())+1
    probs = probs/probs.sum()
    n = len(population)
    indices = np.arange(n)
    selected_indices = np.random.choice(indices, size=n, p=probs)
    selected_population = population[selected_indices]
    return selected_population
```

4. **Crossover:** Genetic information is exchanged between selected solutions to create offspring solutions, this facilitates the exploration of the solution space.

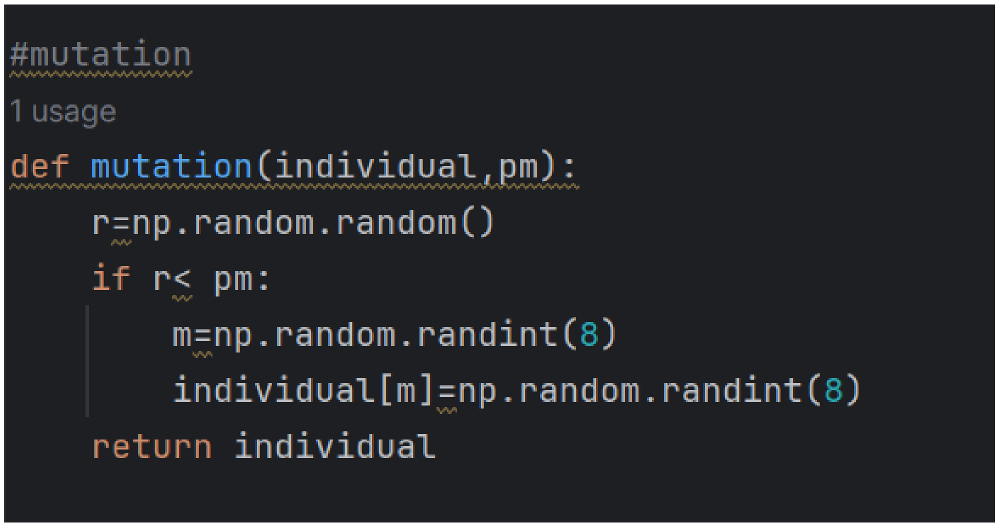
This function performs crossover or recombination between two parents selected from the previous selection method to produce another two new offspring solutions. It randomly selects a cross-over point ‘m’ from 1 to 7 (8 is not included here) and combines this genetic information before and after the crossover point. This occurs with a probability of cross-over ‘pc’ . if the random number selected is less than this ‘pc’ the cross-over will occur; otherwise, a copy from the original parents will be the values of the children.



```
#preforming the crossover to combine better solutions
1 usage
def crossover(parent1, parent2, pc):
    r = np.random.random()
    if r< pc:
        m = np.random.randint( low: 1, high: 8)
        child1 = np.concatenate([parent1[:m], parent2[m:]])
        child2 = np.concatenate([parent2[:m], parent1[m:]])
    else:
        child1 = parent1.copy()
        child2 = parent2.copy()
    return child1, child2
```

5. **Mutation:** Random changes are introduced into offspring solutions to maintain genetic diversity.

This function randomly introduces changes to the selected individuals, this occurs referring to the probability of mutation. A random number 'r' is selected and if it's less than the probability the mutation occurs. The mutation point 'm' represents the Index of the element that will be mutated/ changed. This allows for exploring new regions of the solution space.

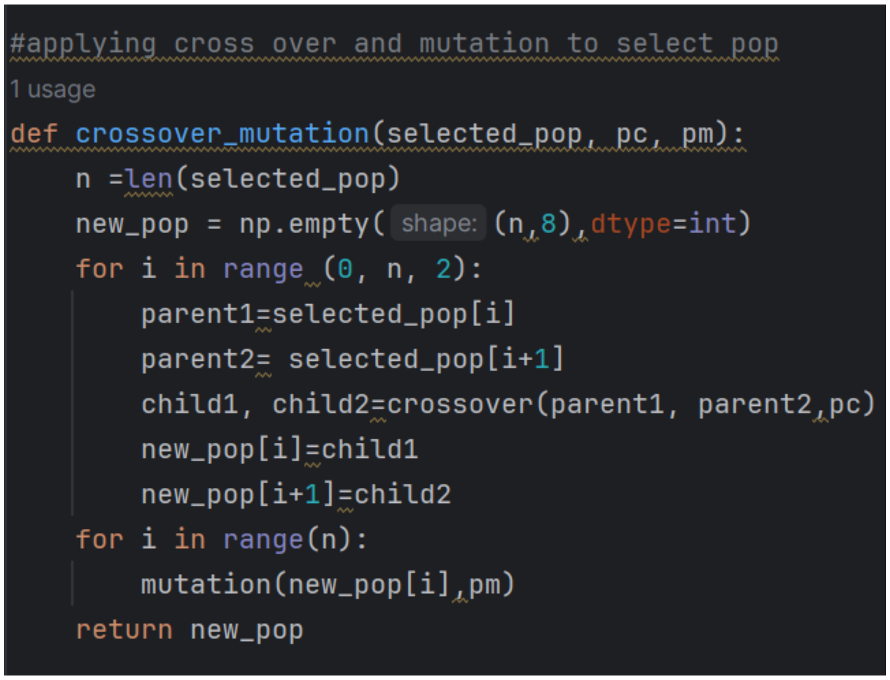


```
#mutation
1 usage
def mutation(individual,pm):
    r=np.random.random()
    if r< pm:
        m=np.random.randint(8)
        individual[m]=np.random.randint(8)
    return individual
```

6. **Crossover mutation:** this implements the combination of crossover and mutation operations in a genetic algorithm, generating offspring individuals with genetic diversity from pairs of selected parent individuals.

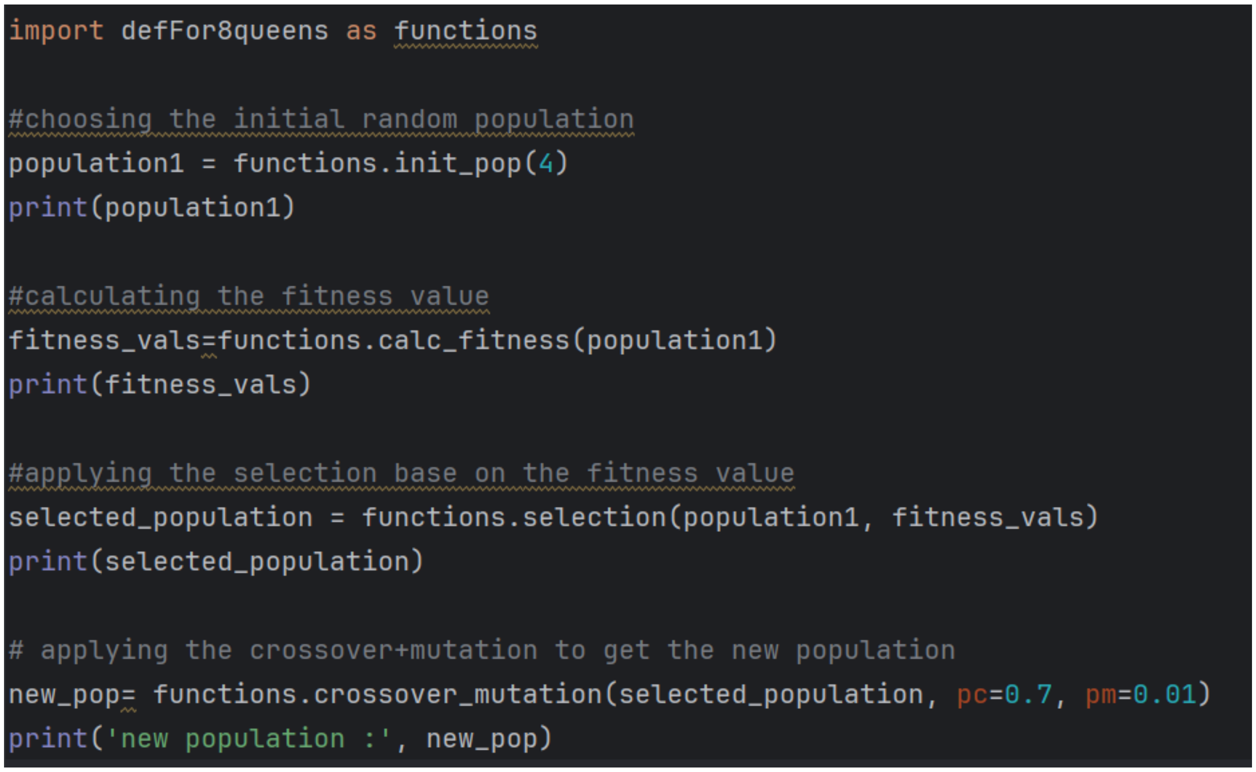
This function first takes the length of the selected population and creates a new list to store offspring individuals, with each individual represented as an array of length 8. Then, it performs a cross-over between every two pairs in the selected population to generate new offspring and then assigns the results to the corresponding rows of the new_pop list. Then it iterates through each individual in the new list and applies mutation to them determined by the probability of mutation 'pm'.

Maryam Ibrahim Mahmoud
March 2024



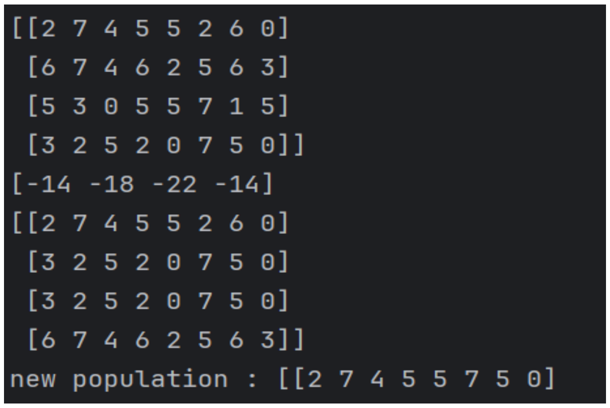
```
#applying cross over and mutation to select pop
1 usage
def crossover_mutation(selected_pop, pc, pm):
    n =len(selected_pop)
    new_pop = np.empty( shape: (n,8),dtype=int)
    for i in range (0, n, 2):
        parent1=selected_pop[i]
        parent2= selected_pop[i+1]
        child1, child2=crossover(parent1, parent2,pc)
        new_pop[i]=child1
        new_pop[i+1]=child2
    for i in range(n):
        mutation(new_pop[i],pm)
    return new_pop
```

These methods can be tested as the following :



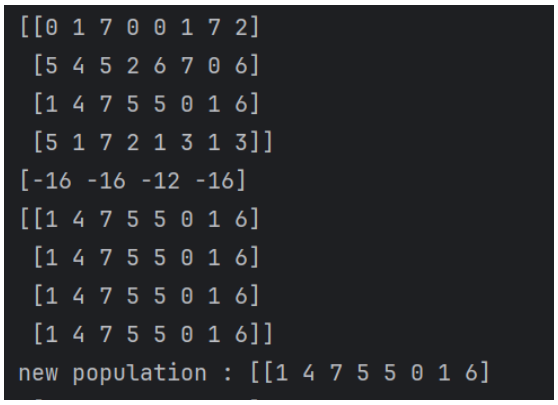
```
import defFor8queens as functions

#choosing the initial random population
population1 = functions.init_pop(4)
print(population1)

#calculating the fitness value
fitness_vals=functions.calc_fitness(population1)
print(fitness_vals)

#applying the selection base on the fitness value
selected_population = functions.selection(population1, fitness_vals)
print(selected_population)

# applying the crossover+mutation to get the new population
new_pop= functions.crossover_mutation(selected_population, pc=0.7, pm=0.01)
print('new population :', new_pop)
```

And will give us the output of (<u>note that this whole process is based on random selection and probabilities; which means that the output won't be the same every time the code is run</u>):



```
[[2 7 4 5 5 2 6 0]
 [6 7 4 6 2 5 6 3]
 [5 3 0 5 5 7 1 5]
 [3 2 5 2 0 7 5 0]]
[-14 -18 -22 -14]
[[2 7 4 5 5 2 6 0]
 [3 2 5 2 0 7 5 0]
 [3 2 5 2 0 7 5 0]
 [6 7 4 6 2 5 6 3]]
new population : [[2 7 4 5 5 7 5 0]
```

Output1



```
[[0 1 7 0 0 1 7 2]
 [5 4 5 2 6 7 0 6]
 [1 4 7 5 5 0 1 6]
 [5 1 7 2 1 3 1 3]]
[-16 -16 -12 -16]
[[1 4 7 5 5 0 1 6]
 [1 4 7 5 5 0 1 6]
 [1 4 7 5 5 0 1 6]
 [1 4 7 5 5 0 1 6]]
new population : [[1 4 7 5 5 0 1 6]
```

Output2

3. Advantages and Applications

Genetic algorithms have several advantages such :

- **Parallelism**: This means performing multiple tasks simultaneously. Genetic Algorithms can tackle several parts of a problem at once.
- **Robustness**: The ability of genetic algorithms to handle various situations and still produce good results. Even if there are changes or uncertainties in the problem, genetic algorithms can often find solutions that work well.
- **Ability to handle complex, multimodal search spaces**: Genetic Algorithms can deal with problems that have many possible solutions, and where the best solution might not be obvious; It can explore different options and find good solutions even in complicated situations.

Overall, Genetic algorithms can efficiently search solution spaces and converge toward optimal or near-optimal solutions in complex optimization problems.