Bonus task

Design a GA (chromosome, crossover, mutation, fitness function) to solve 8-queen problem

Introduction

To express this problem in the form of this algorithm, we have considered each chromosome as an array of eight, and the indices of this array represent the columns of the chess house and each value in this array is the row number. The values of this array are non-repeating and alternating numbers from one to eight. Here I have considered the initial population as 100, but any number can be chosen as desired.

The condition for stopping the algorithm is to complete 10,000 rounds or to find the solution to the problem. The way of choosing parents to produce children is that we choose 5 chromosomes randomly and after these five chromosomes, two chromosomes that have the maximum utility function are selected. The way to calculate the usefulness of a chromosome is that for each gene or queen in a chromosome, the number of threats is calculated and from their sum and then reversing it, the value of the utility function for a chromosome is calculated. The goal is to maximize the utility function. A random point in the chromosome is also selected to produce children. The first child inherits the genes to the left of this point in the first parent and the second child inherits the genes to the left of this point in the second parent. Then, to fill the rest of the genes on the right side of this point in the first child, scroll the second parent from left to right and insert any value that was not present in the first child in the same order of scrolling. To fill the right side of the second child, we repeat the same thing with the first parent.

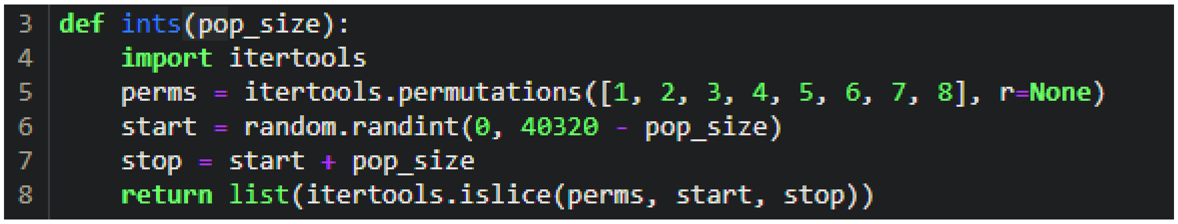
Two random indices are selected to mutate the children and the values in these indices are moved together. Then, the entire population is sorted according to the fitness function, and we replace the last two chromosomes, which are among the worst, with the children.

Implementation

In the following, according to the explanations given, the required functions are implemented and we give a brief explanation of each of them.

Functions

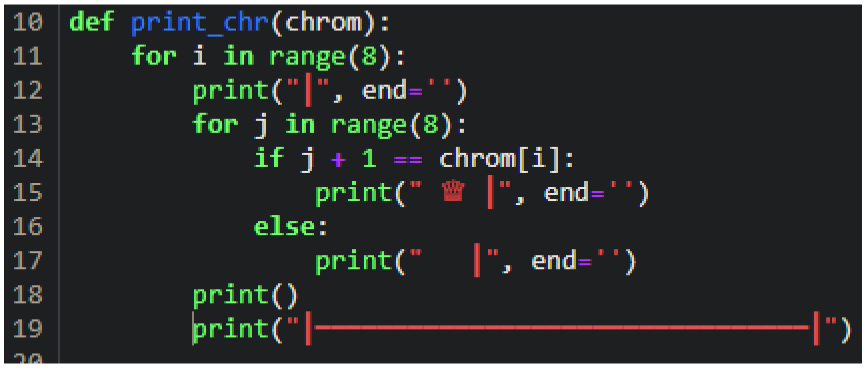
Ints :



```python
def ints(pop_size):
    import itertools
    perms = itertools.permutations([1, 2, 3, 4, 5, 6, 7, 8], r=None)
    start = random.randint(0, 40320 - pop_size)
    stop = start + pop_size
    return list(itertools.islice(perms, start, stop))
```

The first function that is called in the implementation of this algorithm is the “ints” function to generate the initial population, which receives an input of the population size. According to the number of inputs of this function, a list of permutations of numbers one to eight is randomly generated.

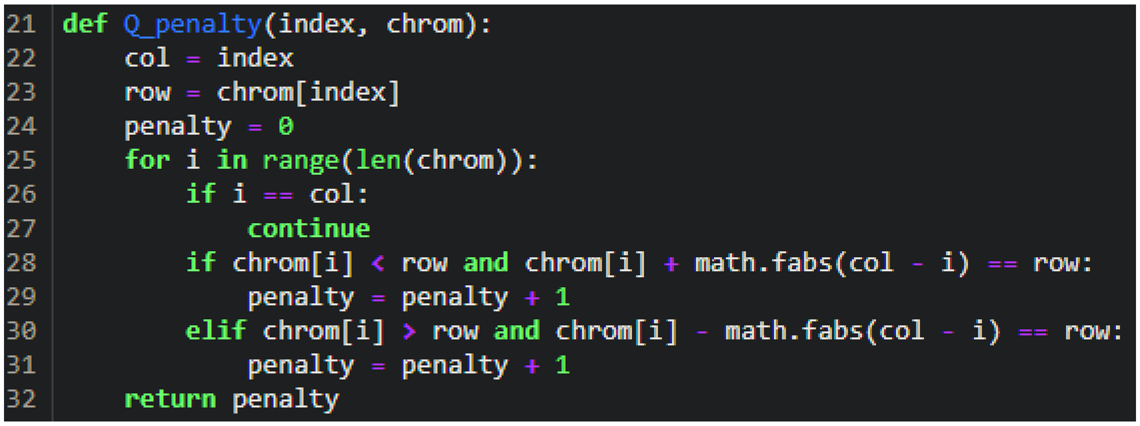
Print_Chr:



```python
def print_chr(chrom):
    for i in range(8):
        print("|", end='')
        for j in range(8):
            if j + 1 == chrom[i]:
                print(" ♛ |", end='')
            else:
                print("   |", end='')
        print()
        print("|────────────────────────────────|")
```

This function is used to print a chromosome on the chess board.

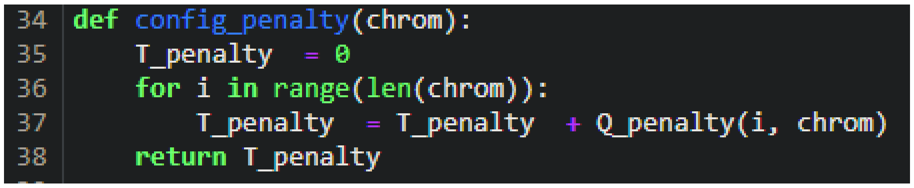
Q_penalty:



```python
def Q_penalty(index, chrom):
    col = index
    row = chrom[index]
    penalty = 0
    for i in range(len(chrom)):
        if i == col:
            continue
        if chrom[i] < row and chrom[i] + math.fabs(col - i) == row:
            penalty = penalty + 1
        elif chrom[i] > row and chrom[i] - math.fabs(col - i) == row:
            penalty = penalty + 1
    return penalty
```

This function calculates and returns the number of threats available for a minister. Its input is the column number of the desired minister and its corresponding chromosome. The way to calculate the meeting of two ministers is based on Manhattan distance. If we add or subtract from the column number of the two ministers based on whether the other minister is ahead or behind, the two ministers should face each other. If this happens, these two ministers will threaten each other.

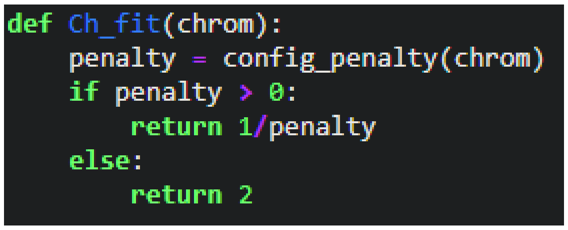
Config_penalty:



```python
def config_penalty(chrom):
    T_penalty  = 0
    for i in range(len(chrom)):
        T_penalty  = T_penalty  + Q_penalty(i, chrom)
    return T_penalty
```

This function returns the total amount of threats in a chromosome by calculating this value of each of the queens (genes) with the help of the Q_penalty function and their sum.

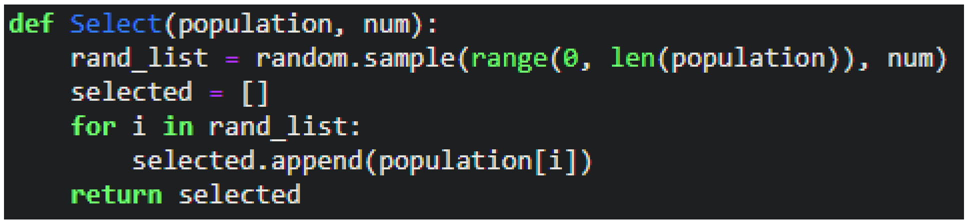
Ch_fit:



```python
def Ch_fit(chrom):
    penalty = config_penalty(chrom)
    if penalty > 0:
        return 1/penalty
    else:
        return 2
```

This function is responsible for calculating the fitness for a chromosome. By calculating the total amount of threats in a chromosome and inverting it, the utility of a chromosome is obtained. If in the chromosome, the queens do not threaten each other, or if the desired chromosome is the answer to the problem, the usefulness of that value is 2, which is the maximum.

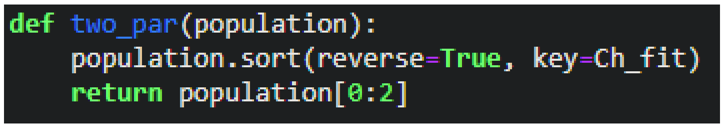
Select:



```python
def Select(population, num):
    rand_list = random.sample(range(0, len(population)), num)
    selected = []
    for i in rand_list:
        selected.append(population[i])
    return selected
```

This function randomly selects 5 chromosomes for reproduction. Its input is the list of total population and number of parents to select.

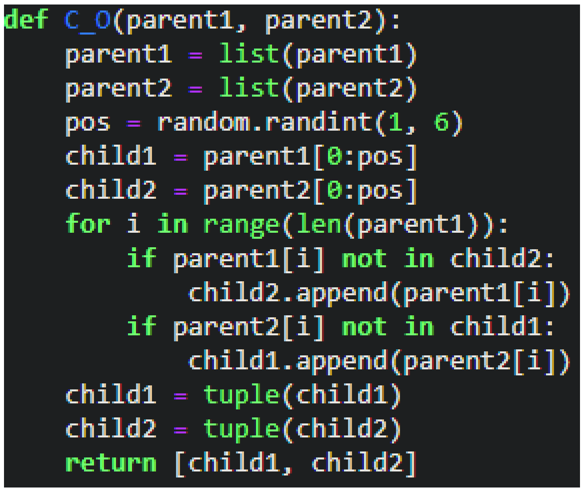
Tow_par:



```python
def two_par(population):
    population.sort(reverse=True, key=Ch_fit)
    return population[0:2]
```

This function returns the two parents of the chromosomes selected in the selection function with maximum utility.

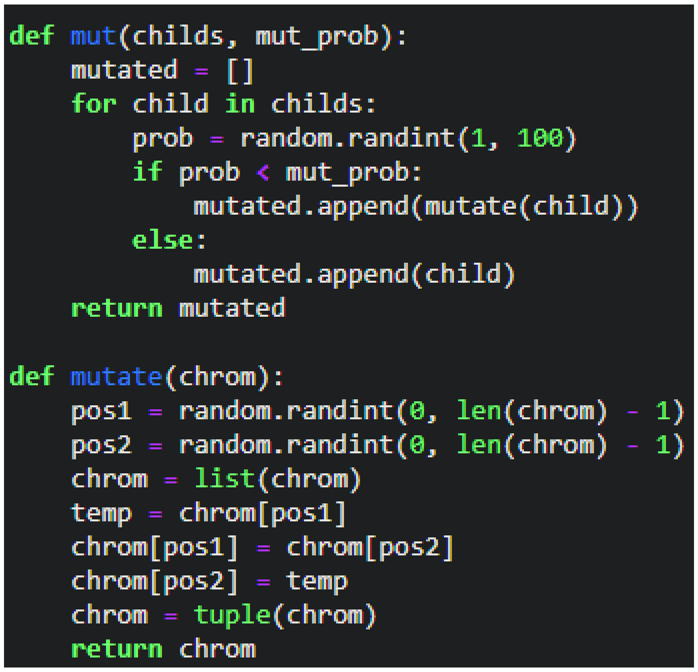
C_O:



```python
def C_O(parent1, parent2):
    parent1 = list(parent1)
    parent2 = list(parent2)
    pos = random.randint(1, 6)
    child1 = parent1[0:pos]
    child2 = parent2[0:pos]
    for i in range(len(parent1)):
        if parent1[i] not in child2:
            child2.append(parent1[i])
        if parent2[i] not in child1:
            child1.append(parent2[i])
    child1 = tuple(child1)
    child2 = tuple(child2)
    return [child1, child2]
```

This function produces 2 children with two parents selected by the two_par function with the mentioned method.

Mut and Mutate:

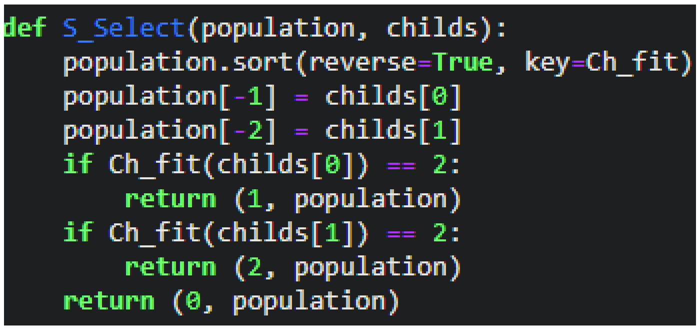


```python
def mut(childs, mut_prob):
    mutated = []
    for child in childs:
        prob = random.randint(1, 100)
        if prob < mut_prob:
            mutated.append(mutate(child))
        else:
            mutated.append(child)
    return mutated

def mutate(chrom):
    pos1 = random.randint(0, len(chrom) - 1)
    pos2 = random.randint(0, len(chrom) - 1)
    chrom = list(chrom)
    temp = chrom[pos1]
    chrom[pos1] = chrom[pos2]
    chrom[pos2] = temp
    chrom = tuple(chrom)
    return chrom
```

These two functions are responsible for performing the mutation operation with the mentioned method in the produced children. The mut function takes two children and generates a probability for each. If it is below 80%, it creates a mutation in the child with the mutate function, otherwise no mutation is created.

S_Select:



```python
def S_Select(population, childs):
    population.sort(reverse=True, key=Ch_fit)
    population[-1] = childs[0]
    population[-2] = childs[1]
    if Ch_fit(childs[0]) == 2:
        return (1, population)
    if Ch_fit(childs[1]) == 2:
        return (2, population)
    return (0, population)
```

This function also replaces the generated children with the method mentioned in the total population list.

Parameters

pop_size : initial population size.

select_random: the number of chromosomes that are randomly selected to select parents.

mut_prob: probability of mutation.

rounds: the number of iterations of the algorithm.

Final result

According to the selected population, the algorithm can reach the answer after more than 200 repetitions. In one of the executions of this algorithm, I got the answer after 273 repetitions.

Top ten chromosomes:

(4, 1, 5, 8, 2, 7, 3, 6),

(1, 5, 2, 6, 3, 7, 4, 8),

(5, 4, 2, 7, 3, 6, 8, 1),

(4, 2, 5, 8, 1, 7, 3, 6),

(1, 6, 4, 2, 8, 5, 3, 7),

(1, 5, 2, 8, 4, 7, 3, 6),

(4, 5, 1, 8, 6, 3, 7, 2),

(1, 5, 2, 6, 3, 7, 8, 4),

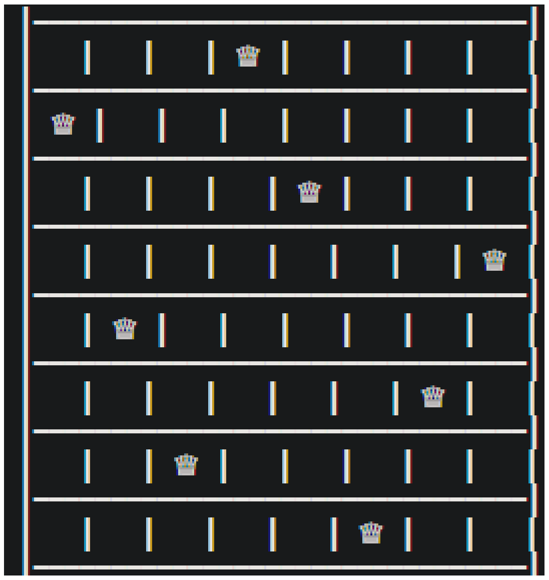
(5, 1, 2, 6, 3, 7, 4, 8),

(1, 5, 2, 8, 4, 7, 3, 6)

The superior chromosome and the answer to the problem:

(4, 1, 5, 8, 2, 7, 3, 6)

Graphical output of the algorithm:



“This code was written for the optimization class assignment”

Iman Kiani Nezhad