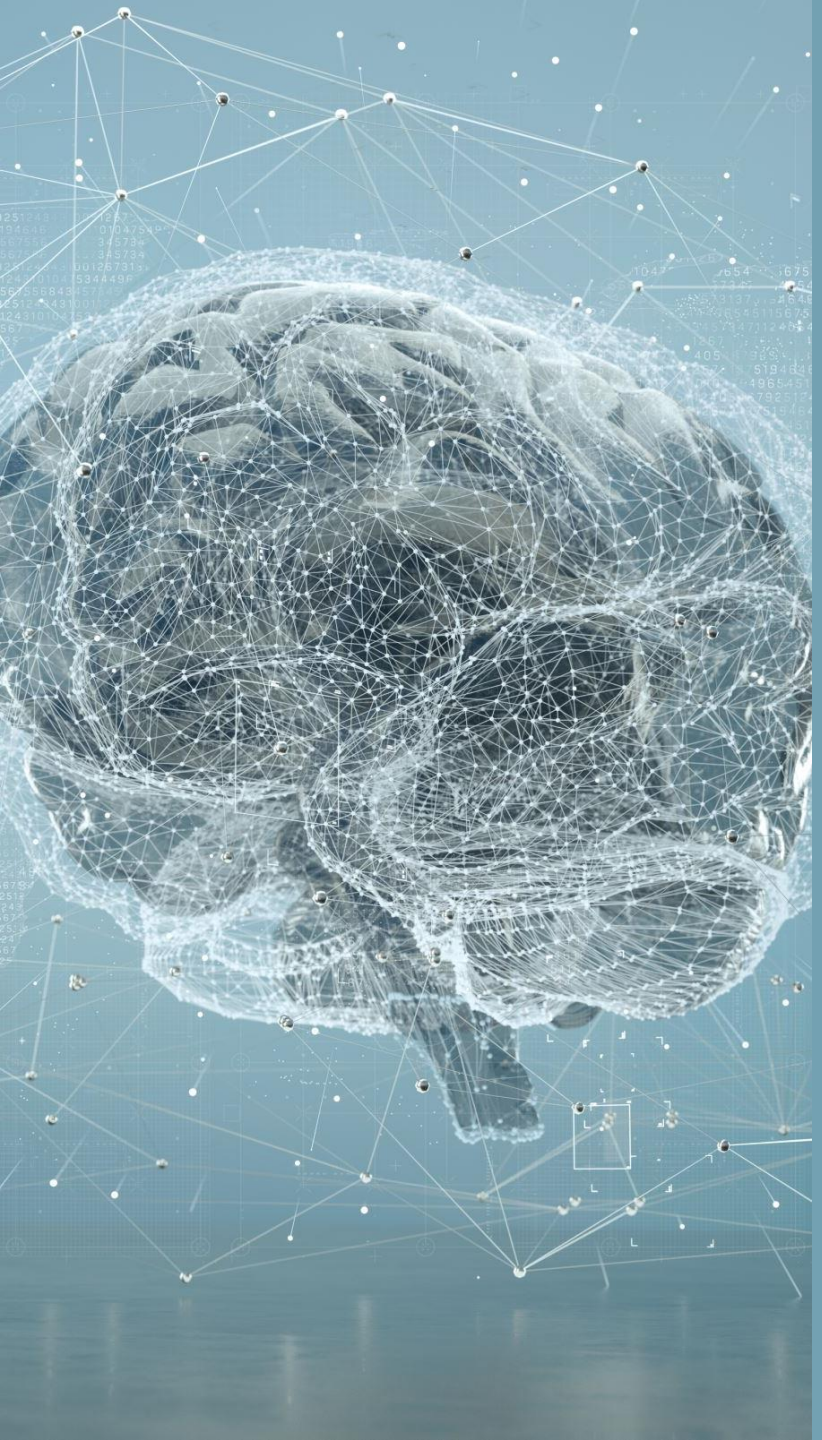


كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE
LAB 10 – GENETIC ALGORITHM
PART (2) –PRACTICAL-

AI Course Team

Eng. Yousef Elbaroudy – Eng Sahar Mohammed

Eng. Reham Ibrahim – Eng. Mawada Ashraf

Eng. Eman Nasser

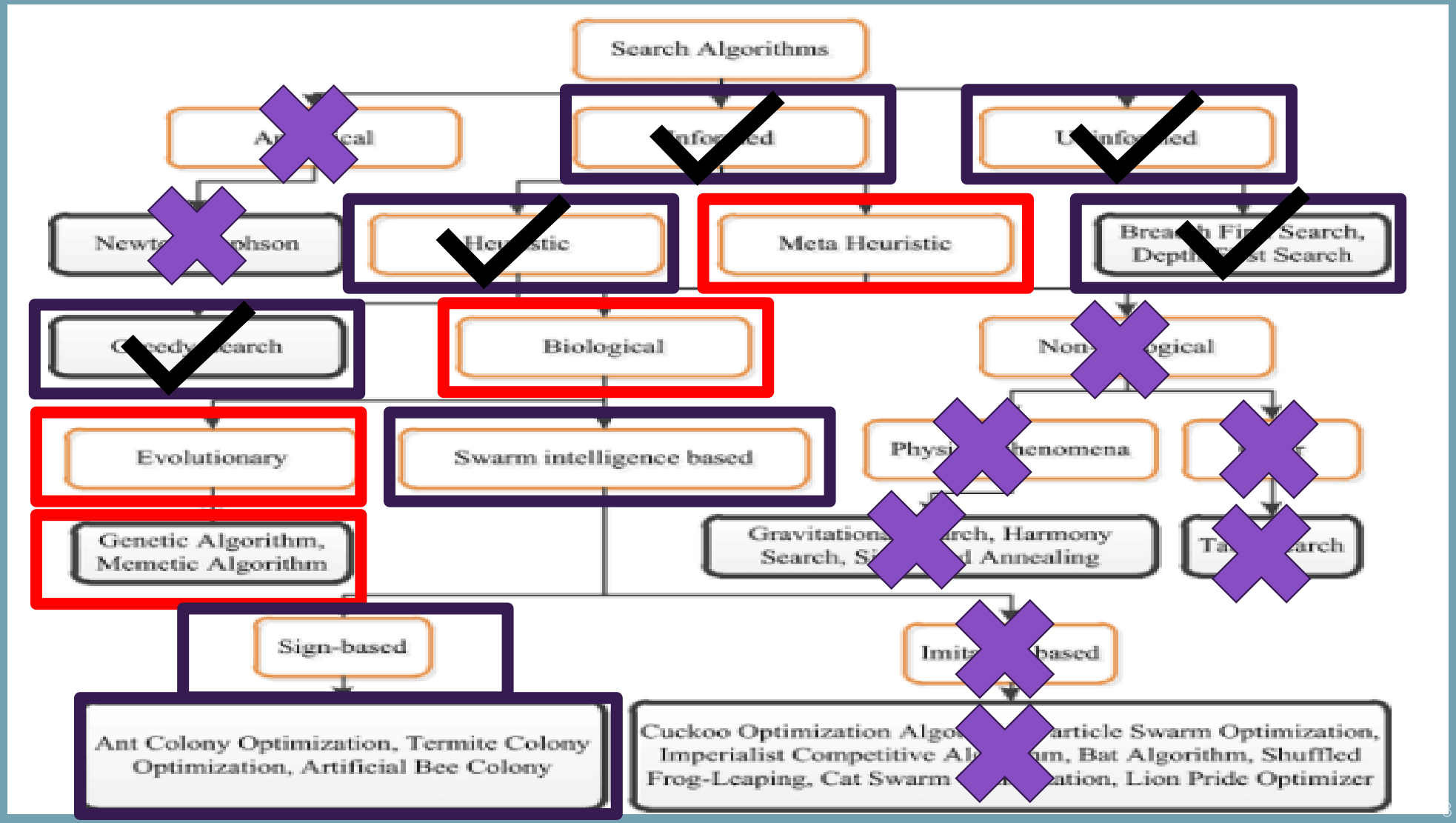
2025/2026

GUIDELINES

You don’t need to memorize what will be mentioned. Instead, try to understand

Each PowerPoint Presentation will be available as PDF file on Google Drive Folder of the Course

THE CLASSIFICATION OF SEARCHING ALGORITHMS



KEY CONCEPTS OF META-HEURISTIC ALGORITHMS (RECAP)


Inspired from Artificial Intelligence rather than pure math, which means they are not based on algebraic models


There is no theoretical, which means lack of proofs


Capability of addressing both discrete and continues optimization problems


Efficiently explore the search space in order to find near-optimal feasible solutions


Provide no guarantee of global or local optima (BEST BENEFIT) which can lead to computational complexity


They balance exploration and exploitation, and incorporate randomness to avoid getting stuck in local solutions

- Meta-Heuristic: high-level, problem-independent strategies or frameworks designed to solve complex optimization problems.

EXPLORATION AND EXPLOITATION (RECAP)

Exploration

- Searching new, unexplored areas of the solution space.
- Discover diverse solutions and avoid local optima.
- **Randomness**, **diversification**, and wide search space coverage.
- May explore unnecessary regions, increasing computational cost.

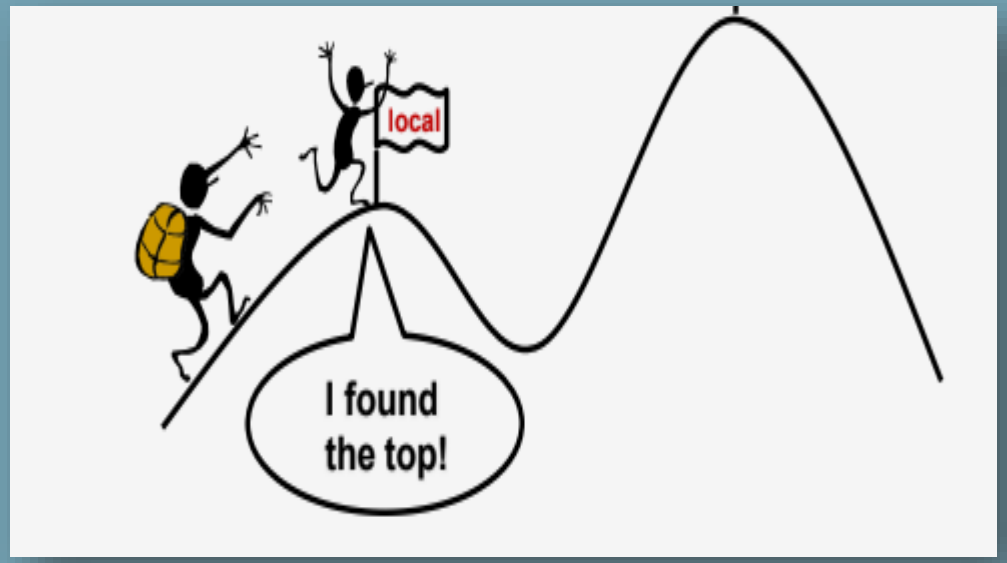
Exploitation

- Refining known good solutions to improve performance.
- Optimize the best-known solutions for higher quality results.
- Focused, intensification, and local optimization.
- May get trapped in local optima and miss the global best solution.

GENETIC ALGORITHM

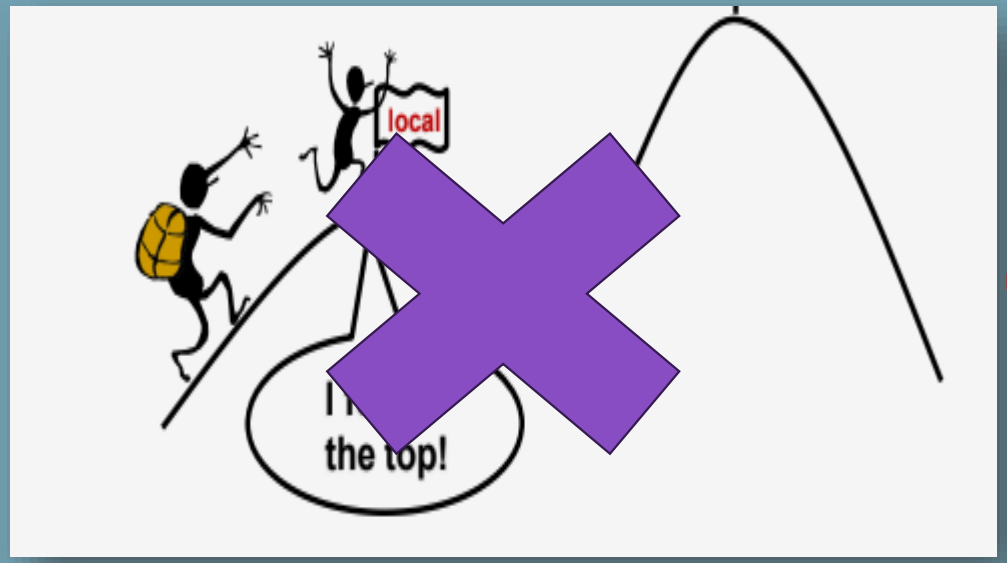
GENETIC ALGORITHM

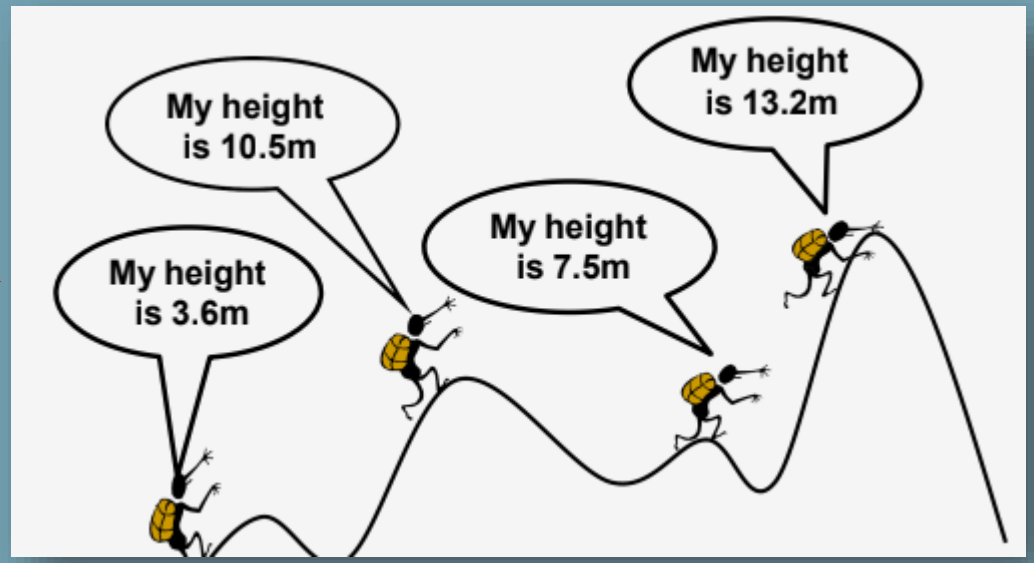
Genetic Algorithm (GA) is a method for solving optimization and search problems by mimicking the process of natural evolution. It's especially useful when the search space is huge, complex, or poorly understood.



GENETIC ALGORITHM

Genetic Algorithm (GA) is a method for solving optimization and search problems by mimicking the process of natural evolution. It's especially useful when the search space is huge, complex, or poorly understood.





GENETIC ALGORITHM

Genetic Algorithm (GA) is a method for solving optimization and search problems by mimicking the process of natural evolution. It's especially useful when the search space is huge, complex, or poorly understood.

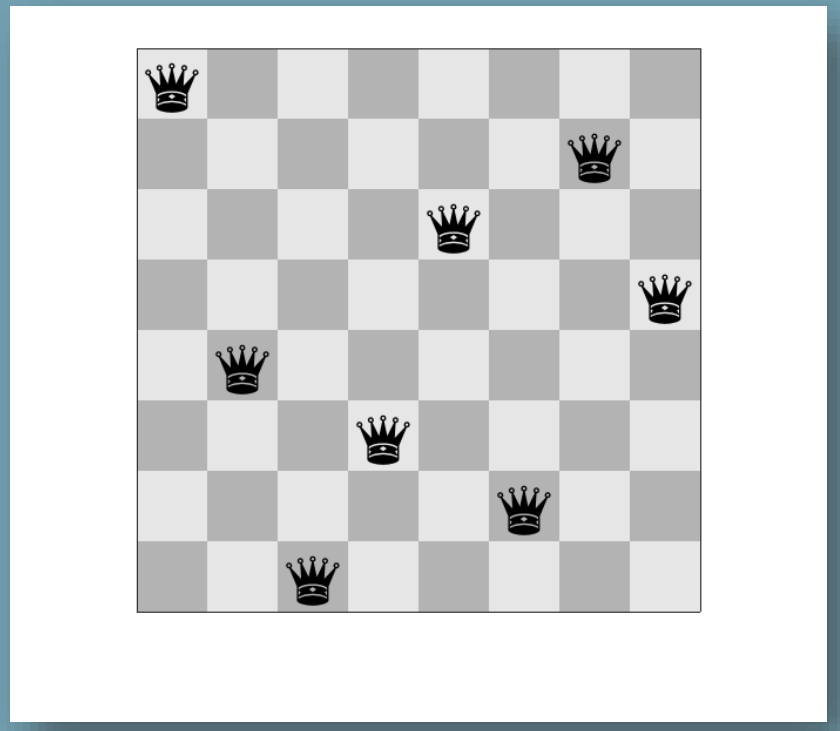
The Genetic Algorithm (Mainly all meta-heuristic algorithms) uses multiple climbers in parallel to find the global optimum **(a population-based algorithm).**

GA's are useful for solving multidimensional problems containing many local maxima (or minima) in the solution space.

COMMON PROBLEMS TO SOLVE

8-Queen Problem

placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other



IMPLEMENTATION

8-Queen Problem

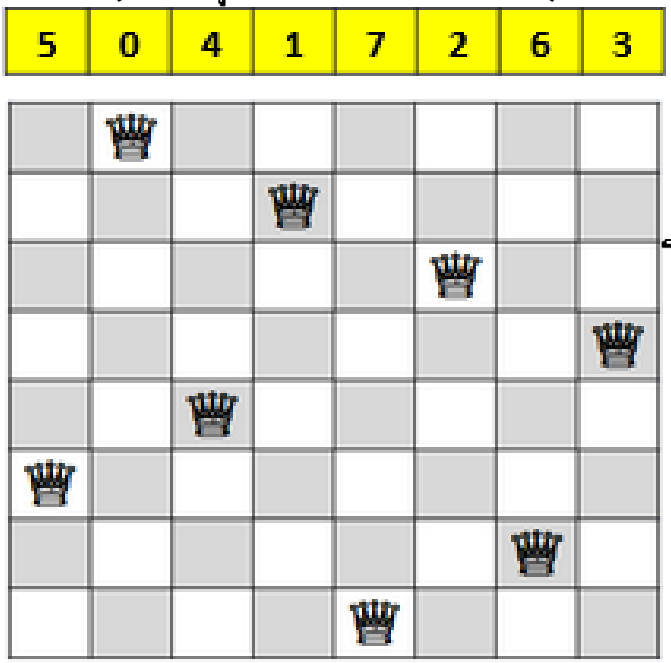
EIGHT QUEENS CLASS (INITIALIZE POPULATION)

```python
# Represent the problem
class eight_queens:
    def __init__(self, population_size, crossover_probability, mutation_probability, num_of_generations):
        self._population_size = population_size
        self._crossover_probability = crossover_probability
        self._mutation_probability = mutation_probability
        self._num_of_generations = num_of_generations

        self._population = []
        one_individual = [0] * 8
        for i in range(self._population_size):
            copy = one_individual.copy()
            for j in range(8):
                copy[j] = random.randint(0,7)

            self._population.append(copy)
```

Columns as indices
(i.e. queen 1 at col 0)



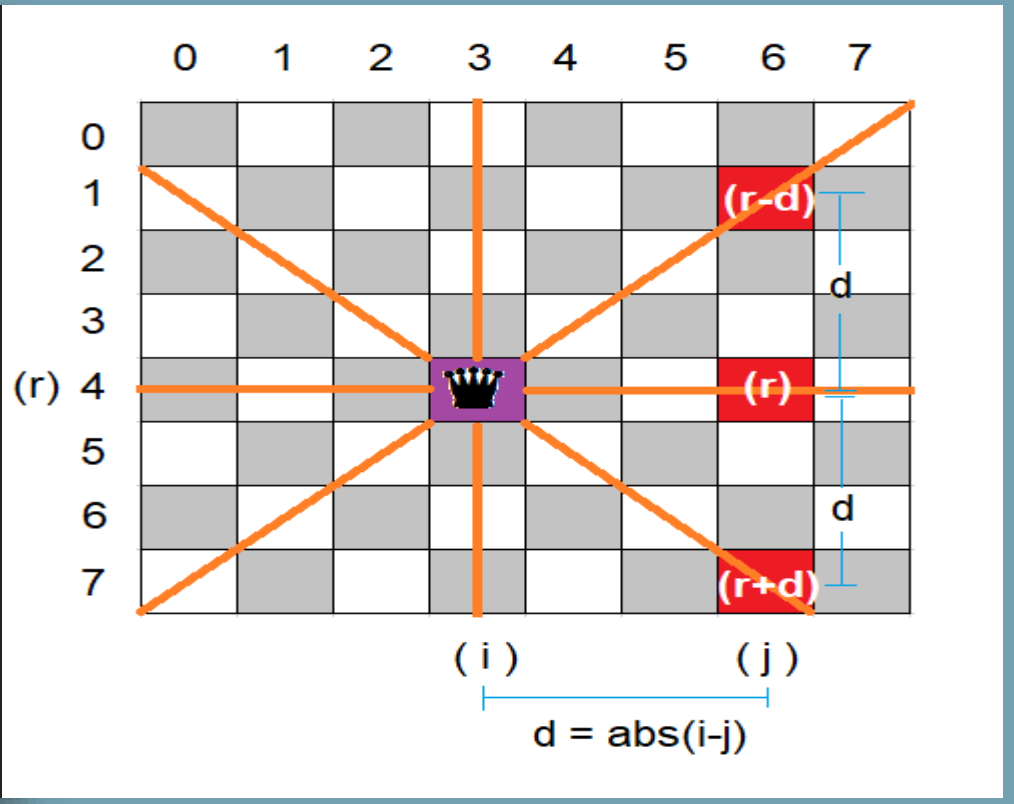
Rows as elements where the queen exists

I will represent (encode) each individual as 8-indices as columns, each index will be the number of row which the queen exists

EIGHT QUEENS CLASS (CALCULATE FITNESS METHOD)

```python
def __calculate_fitness(self):
    fitness_values = []
    for individual in self.__population:
        penalty = 0
        for queen in range(len(individual)):
            for threat in range(len(individual)):
                if queen == threat:
                    continue
                i = queen
                j = threat
                r = individual[queen]
                d = abs(i - j)
                if individual[threat] in [r, r-d, r+d]:
                    penalty += 1
        fitness_values.append(-penalty)

    return fitness_values
```

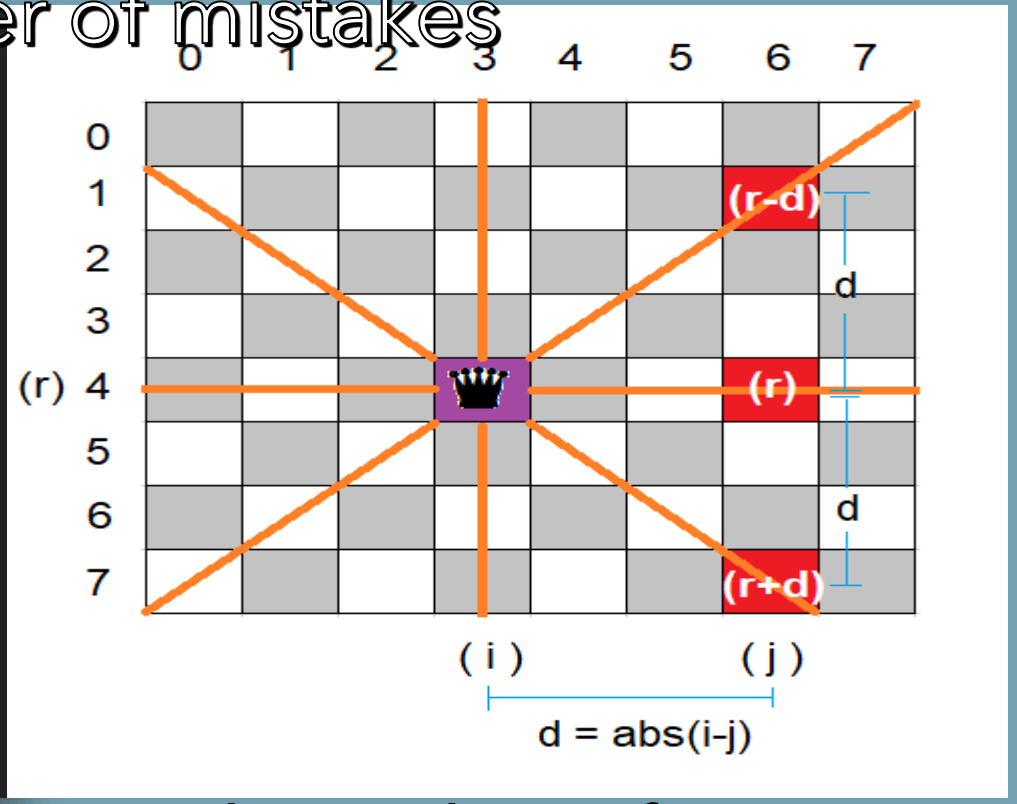


EIGHT QUEENS CLASS (CALCULATE FITNESS METHOD)

Each threat will be considered as penalty which is the number of mistakes

```python
def __calculate_fitness(self):
    fitness_values = []
    for individual in self.__population:
        penalty = 0
        for queen in range(len(individual)):
            for threat in range(len(individual)):
                if queen == threat:
                    continue
                i = queen
                j = threat
                r = individual[queen]
                d = abs(i - j)
                if individual[threat] in [r, r-d, r+d]:
                    penalty += 1
        fitness_values.append(-penalty)

    return fitness_values
```



We will multiply each penalty by -1 to indicate that refers to mistakes

EIGHT QUEENS CLASS (SELECTION METHOD)

To change the interval of the penalty → fitness = penalty + abs(minimum) + 1

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

We needed to do that to keep the best individual with highest fitness → Maximize

EIGHT QUEENS CLASS (SELECTION METHOD)

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

Assume that we have three fitness
[12, 7, 3]

EIGHT QUEENS CLASS (SELECTION METHOD)

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

Assume that we have three fitness
[12, 7, 3] → [-12, -7, -3]
After changing its interval it should be
[1, 6, 10]

EIGHT QUEENS CLASS (SELECTION METHOD)

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

Assume that we have three fitness
[12, 7, 3]
After changing its interval it should be
[1, 6, 10]
To calculate their probabilities, we should obtain the sum = 17

EIGHT QUEENS CLASS (SELECTION METHOD)

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

Assume that we have three fitness
[12, 7, 3]
After changing its interval it should be
[1, 6, 10]
To calculate their probabilities, we should obtain the sum = 17
Now divide each value on the sum
[1/17, 6/17, 10/17] = [0.05, 0.35, 0.58]

EIGHT QUEENS CLASS (SELECTION METHOD)

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

Assume that we have three fitness
[12, 7, 3]
After changing its interval it should be
[1, 6, 10]
To calculate their probabilities, we should obtain the sum = 17
Now divide each value on the sum
[1/17, 6/17, 10/17] = [0.05, 0.35, 0.58]

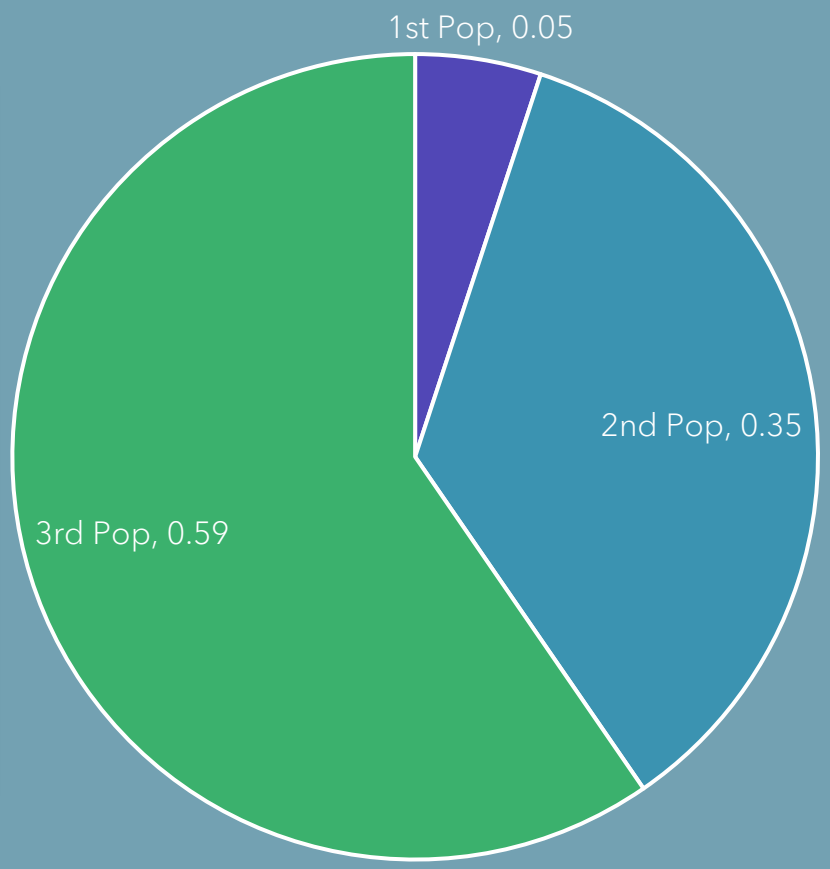
That's why we changed the interval, to keep the best fitness with highest probability

EIGHT QUEENS CLASS (SELECTION METHOD)

Roulette Wheel

```
def _selection(self, fitness_vals):
    fitness_values = fitness_vals.copy()
    minimum_fitness = abs(min(fitness_values)) + 1
    scaled_fitness = [value + minimum_fitness for value in fitness_values]
    sum_of_fitness = sum(scaled_fitness)
    probabilities = [value/sum_of_fitness for value in scaled_fitness]

    selected_population = random.choices(self._population, k = self._population_size, weights=probabilities)
    return selected_population
```

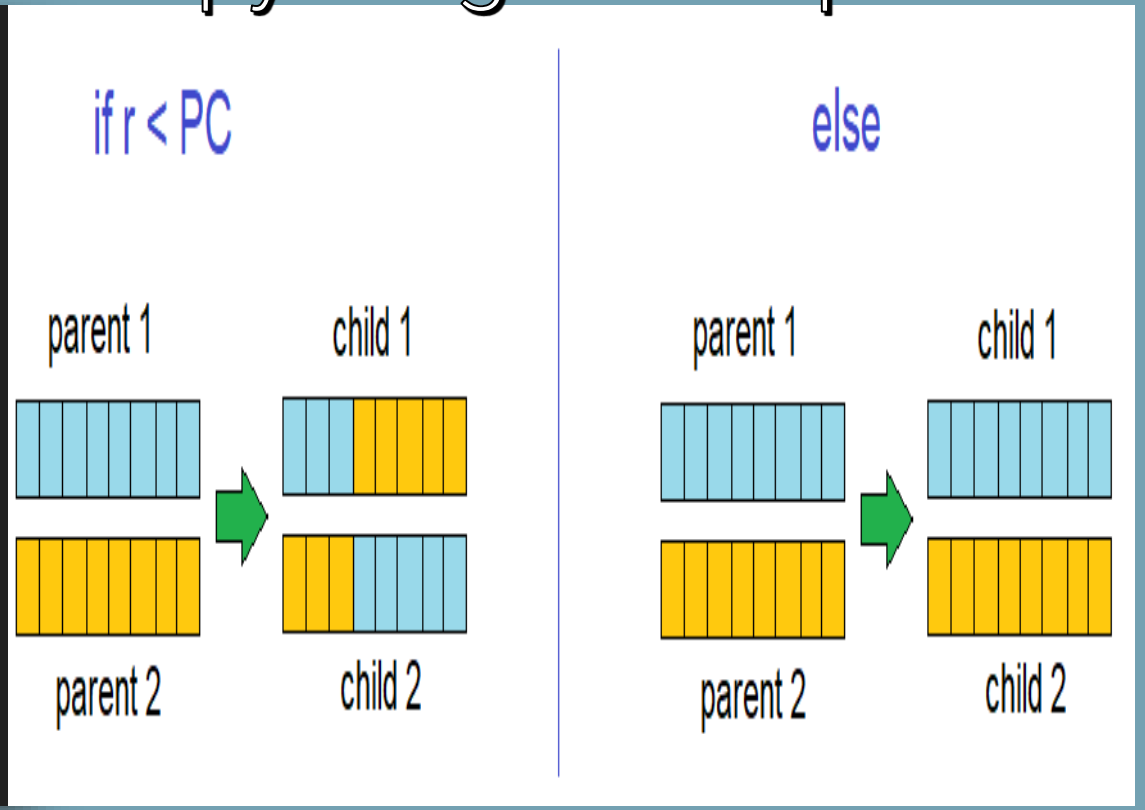


EIGHT QUEENS CLASS (CROSSOVER METHOD)

Simulating reality, there is a chance that chromosomes could crossover, and it may not then copy the genes of parents

```
def __crossover(self, parent1, parent2, probability):
    r = random.random()
    if r < probability:
        middle_point = random.randint(0,7)
        offspring1 = parent1[:middle_point] + parent2[middle_point:]
        offspring2 = parent2[:middle_point] + parent1[middle_point:]
    else:
        offspring1 = parent1.copy()
        offspring2 = parent2.copy()

    return offspring1, offspring2
```



EIGHT QUEENS CLASS (MUTATION METHOD)

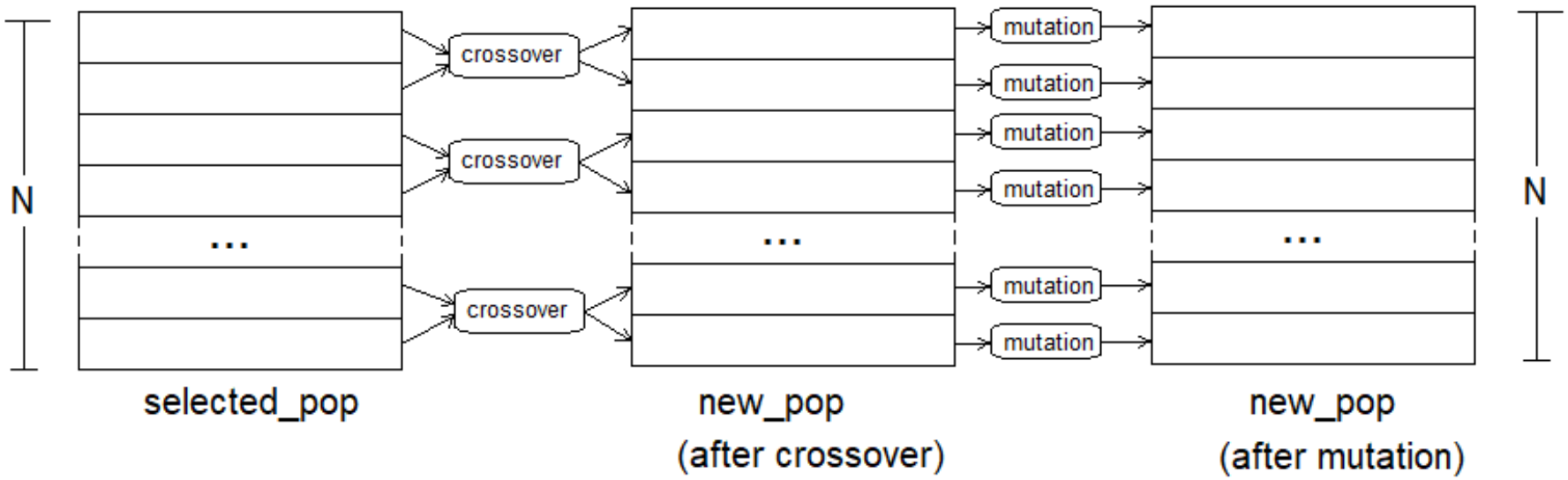
Remember that mutation happens RARELY

It means that the genes could be modified

```
def __mutation(self, individual, probability):
    ind = individual.copy()
    r = random.random()
    if r < probability:
        random_neuclutide = random.randint(0,7)
        ind[random_neuclutide] = random.randint(0,7)
    return ind
```

EIGHT QUEENS CLASS (CROSSOVER-MUTATION METHOD)

Every two parents should produce two childs
Each child can be mutated or not



EIGHT QUEENS CLASS (CROSSOVER-MUTATION METHOD)

```
def __crossover_mutation(self, crossover_probability, mutation_probability):
    new_population = []
    for index in range(0, self.__population_size, 2):
        child1, child2 = self.__crossover(self.__population[index], self.__population[index+1], crossover_probability)
        new_population.append(child1)
        new_population.append(child2)
    for index in range(self.__population_size):
        new_population[index] = self.__mutation(new_population[index], mutation_probability)

    self.__population = new_population.copy()
```

EIGHT QUEENS CLASS (SOLUTION –PUTTING ALL TOGETHER-METHOD)

```
def solution(self):
    best_fitness_overall = None
    best_individual = None
    for generation in range(self.__num_of_generations):
        fitness_values = self.__calculate_fitness()
        index_of_best_fitness = fitness_values.index(max(fitness_values))
        best_fitness = fitness_values[index_of_best_fitness]
        if best_fitness_overall is None or best_fitness > best_fitness_overall:
            print(f"\rNo. of generation: {generation:06}   Best fitness (negative)={-best_fitness:03}", end="")
            best_individual = self.__population[index_of_best_fitness]

        if best_fitness == 0:
            print("\nFound optimal Solution")
            return best_individual
        self.__population = self.__selection(fitness_values)
        self.__crossover_mutation(self.__crossover_probability, self.__mutation_probability)

    return best_individual
```

EIGHT QUEENS CLASS (DISPLAY GRID METHOD)

This method only to simulate the final solution

```
def display_grid(self, sol):
    lst = ["    "] * 8
    representation = []
    for i in range(8):
        representation.append(lst.copy())

    for col in range(len(sol)):
        representation[sol[col]][col] = " Q "

    row_line = "+" + ('---+' * 8)
    for row in representation:
        print(row_line)
        print("|" + "|".join(cell for cell in row) + "|")
    print(row_line)
```

TESTING

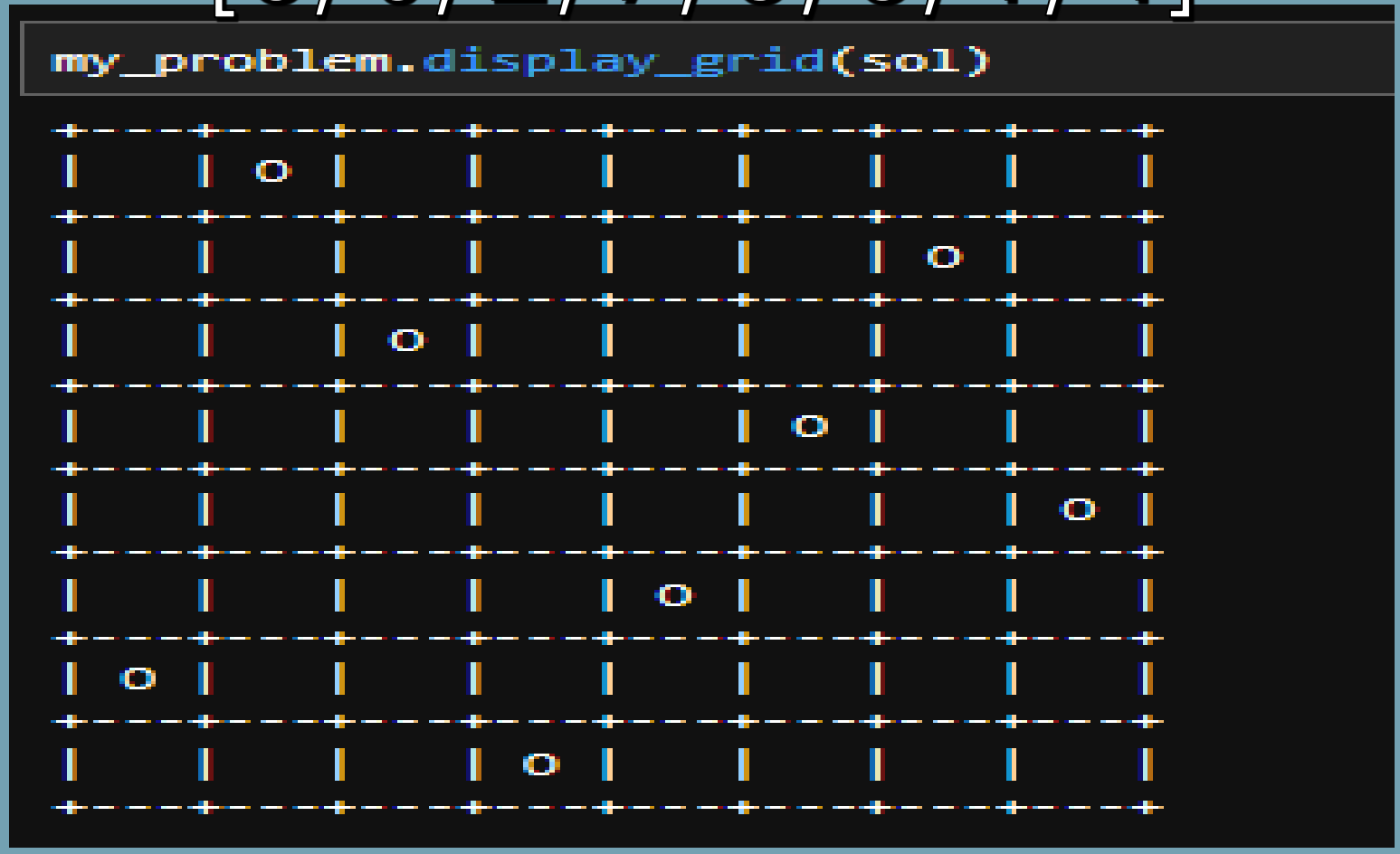
```
my_problem = eight_queens(population_size=6, crossover_probability=0.68, mutation_probability=0.1, num_of_generations=10000)
sol = my_problem.solution()
print(sol)
```

```
No. of generation: 000120   Best fitness (negative)=000
Found optimal Solution
[6, 0, 2, 7, 5, 3, 1, 4]
```

LET'S SEE IF IT IS TRUE ANSWER ?

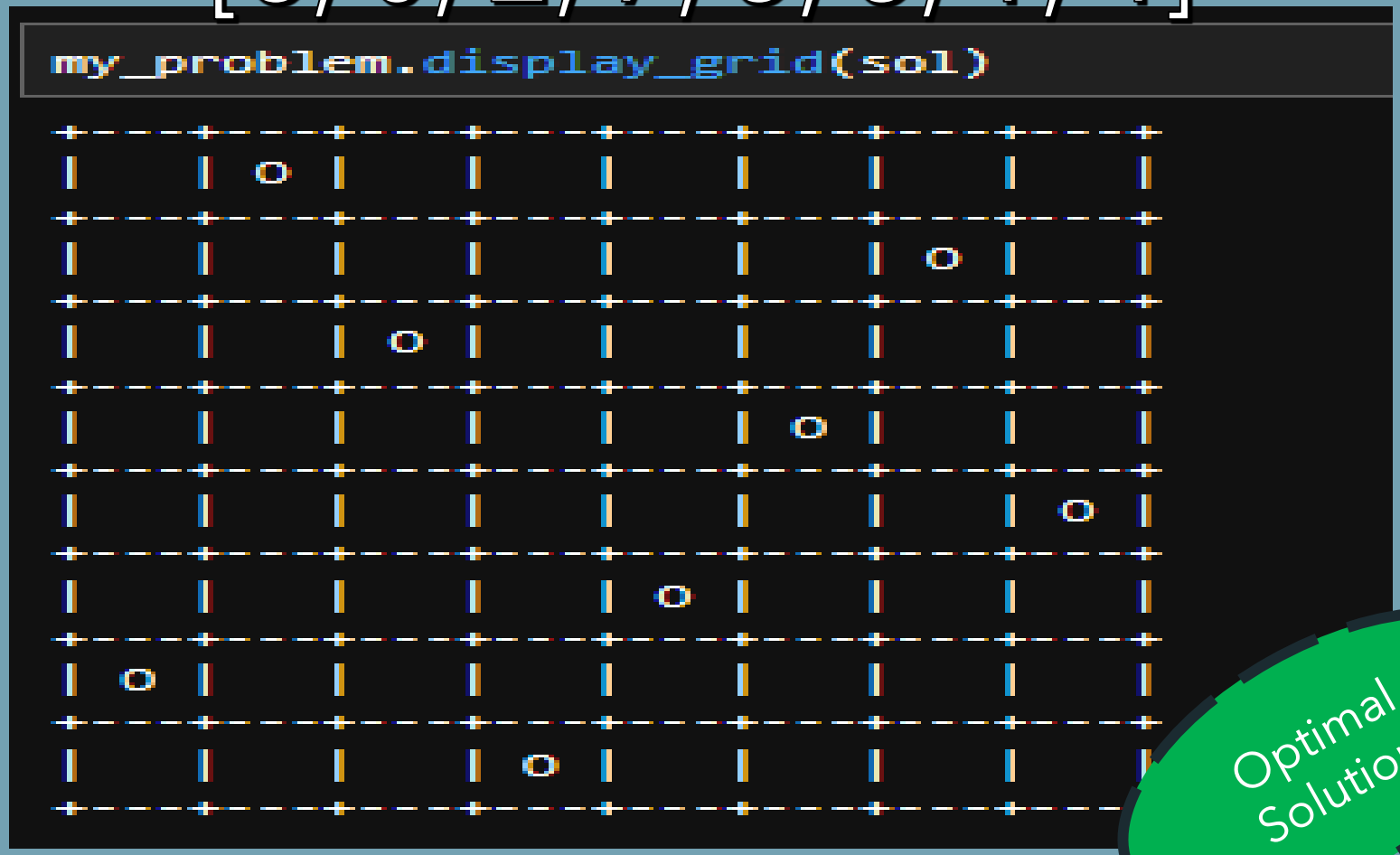
[6, 0, 2, 7, 5, 3, 1, 4]

```
my_problem.display_grid(sol)
```



LET'S SEE IF IT IS TRUE ANSWER ?

[6, 0, 2, 7, 5, 3, 1, 4]



COMPARE BETWEEN FIRST AND LAST GENERATION

First Generation

```
[[2, 2, 3, 0, 0, 3, 7, 0], [7, 2, 0, 1, 4, 4, 7, 2], [6, 0, 5, 6, 5, 2, 4, 3], [7, 3, 6, 1, 0, 3, 0, 4], [1, 0, 7, 1, 2, 3, 2, 4], [4, 7, 2, 2, 0, 0, 5, 0]]
```

Last Generation (Convergent)

```
[[6, 1, 7, 2, 0, 3, 7, 4], [6, 1, 7, 2, 0, 3, 7, 4], [6, 1, 5, 2, 0, 3, 7, 4], [6, 1, 7, 2, 0, 3, 7, 4], [6, 1, 7, 2, 0, 3, 7, 4], [6, 1, 7, 2, 0, 3, 7, 4]]
```


END OF COURSE
☺
Thank you