8-Queens Problem with Genetic Algorithm

Overview

- **Purpose**: Solving the 8-queens puzzle, where the goal is to place eight queens on a chessboard without them attacking each other.
- **Method**: Implementation of a Genetic Algorithm (GA) using the DEAP library in Python.
- **Key Concepts**: 8-Queens Problem, Genetic Algorithms, DEAP library.

Genetic Algorithm Approach

- **Population Initialization**: Begins with a randomly generated population of chessboard arrangements.
- **Fitness Evaluation**: Each arrangement is evaluated based on how many queens are in non-attacking positions.
- **Selection**: Selects the best-performing arrangements for the next generation.
- **Genetic Operations**: Applies crossover and mutation to generate new solutions.

DEAP Library Utilization

- Used for creating and managing the GA components like individuals, populations, and genetic operations.

Python Implementation

- **Code Structure**: Includes Python code for setting up and executing the GA.

This code is an implementation of a genetic algorithm to solve the 8-queens problem. It uses DEAP to create the necessary data structures and functions for the genetic algorithm. We find solutions to the 8-queens problem by evolving populations of queen placements.

In [9]:
```
# imports necessary modules like random, numpy, and various component
# such as algorithms, base, creator, and tools.
import random
import numpy

from deap import algorithms
from deap import base
from deap import creator
from deap import tools
```

- NB_QUEENS: This is set to 8, indicating the size of the chessboard and the number of queens.

In [10]:
```
#Problem parameter
#NB_QUEENS = 10
#NB_QUEENS = 15
#NB_QUEENS = 20
NB_QUEENS = 8
```

- The evaluation/**fitness** function counts the number of conflicts along the diagonals. The lower the conflict count, the better the fitness. So it's a minimization problem.

In [11]:
```
def evalNQueens(individual):
    size = len(individual)
    #Count the number of conflicts with other queens.
    #The conflicts can only be diagonal, count on each diagonal line
    left_diagonal = [0] * (2*size-1)
    right_diagonal = [0] * (2*size-1)

    #Sum the number of queens on each diagonal:
    for i in range(size):
        left_diagonal[i+individual[i]] += 1
        right_diagonal[size-1-i+individual[i]] += 1

    #Count the number of conflicts on each diagonal
    sum_ = 0
    for i in range(2*size-1):
        if left_diagonal[i] > 1:
            sum_ += left_diagonal[i] - 1
        if right_diagonal[i] > 1:
            sum_ += right_diagonal[i] - 1
    return sum_,
```

- Now we use **DEAP's creator** module - as imported - to create a fitness function (FitnessMin) and an individual class (Individual). The fitness function is designed to minimize the evaluation function (i.e., the number of conflicts).

In [12]:
```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,)) # +1.0 ma
creator.create("Individual", list, fitness=creator.FitnessMin)
```

```
/home/a10/anaconda3/lib/python3.11/site-packages/deap/creator.py:18
5: RuntimeWarning: A class named 'FitnessMin' has already been crea
ted and it will be overwritten. Consider deleting previous creation
of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and i
t "
/home/a10/anaconda3/lib/python3.11/site-packages/deap/creator.py:18
5: RuntimeWarning: A class named 'Individual' has already been crea
ted and it will be overwritten. Consider deleting previous creation
of that class or rename it.
  warnings.warn("A class named '{0}' has already been created and i
t "
```

- The **Toolbox class** from DEAP is used to register various GA operations:
  - Permutation: Generates a random permutation of numbers, representing the positions of queens on the board.
  - Individual and Population Initialization: Initializes individuals and the population

using the permutation.

- Genetic Operators: Registers functions for evaluation, crossover, mutation, and selection.

In [13]:
```
#Since there is only one queen per line,
#individual are represented by a permutation
toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(NB_QUEENS), NB_(
#This set up to generate permutations of numbers from 0 to 7 (8-queer
```

In [14]:
```
#Structure initializers
#An individual is a list that represents the position of each queen.
#Only the line is stored, the column is the index of the number in th
toolbox.register("individual", tools.initIterate, creator.Individual,
toolbox.register("population", tools.initRepeat, list, toolbox.indivi

toolbox.register("evaluate", evalNQueens)
toolbox.register("mate", tools.cxPartialyMatched)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=2.0/NB_QUEE
toolbox.register("select", tools.selTournament, tournsize=3)
```

- **Main Function**: This function sets up and runs the GA using algorithms.eaSimple.
- It initializes a population, defines a hall of fame (to store the best individual), and sets up statistics to track the evolution process. The GA parameters like ´crossover probability´ (cxpb), mutation probability (mutpb), and number of generations (ngen) are specified here.

In [15]:
```
#main function where the GA algorithm is executed.
def main(seed=0):
    random.seed(seed)
    # initializes a population of 300 individuals with random queen p
    pop = toolbox.population(n=300)
    # Hall of Fame (hof) to keep track of the best individuals found
    hof = tools.HallOfFame(1)

    #Statistics about the population
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("Avg", numpy.mean) #average
    stats.register("Std", numpy.std) #standard deviation
    stats.register("Min", numpy.min) #min
    stats.register("Max", numpy.max) #max fitness

    #perform the evolutionary process
    #It runs for 100 generations with a crossover probability of 0.5,
    algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=100,
                        halloffame=hof, verbose=True)

    return pop, stats, hof
```

In [16]:
```
if __name__ == "__main__":
    main()
```

```
gen     nevals  Avg     Std     Min     Max
0       300     3.94333 1.25437 1       9
1       176     3.4     1.23288 0       7
2       149     3.09667 1.20581 0       7
3       181     2.98333 1.32025 0       7
4       175     2.84    1.35931 0       7
5       163     2.74    1.48965 0       7
6       204     2.82    1.51468 0       7
7       183     2.57667 1.48687 0       7
8       170     2.29333 1.46764 0       6
9       186     2.35333 1.51278 0       7
10      176     2.21667 1.63597 0       8
11      170     2       1.7282  0       8
12      198     1.94    1.69009 0       6
13      184     1.68667 1.72099 0       6
14      174     1.3     1.65429 0       6
15      173     0.953333        1.45756 0       6
16      167     0.996667        1.61555 0       6
17      177     0.82            1.54087 0       6
```

- It shows the progress of the genetic algorithm over 100 generations. Each row in the output represents a generation, with the columns showing the number of evaluations, average fitness, standard deviation of fitness, minimum fitness, and maximum fitness in that generation.

1. Generation (gen): The current generation number in the evolutionary process.
2. Number of Evaluations (nevals): The number of individuals evaluated in each generation.
3. Average (Avg): The average fitness of the population in each generation.
4. Standard Deviation (Std): The standard deviation of the fitness in the population, indicating the diversity.
5. Minimum (Min): The best (lowest) fitness score in the population.
6. Maximum (Max): The worst (highest) fitness score in the population.

Conclusion

This is an evolutionary process over 100 generations.

- **Progressive Improvement**:So in generation 0, the average fitness is around 3.94, from generation 0 to 100 there's a decrease in the average fitness (Avg) score. This indicates The genetic algorithm is effectively evolving better solutions over time.
- **Optimal Solutions**: The **minimum fitness(Min)** reaching 0 early in the process (by generation 1) and consistently appearing in subsequent generations suggests that the algorithm quickly finds an optimal solution (a configuration with no conflicts) and retains it. This retention is likely due to the selection mechanism favoring individuals with the best fitness scores.
- The **maximum fitness(Max)** are relatively high in some generations. This indicates a considerable variation among individuals in the population, with some far from optimal. This diversity is essential as it prevents premature convergence to local solutions.
- **Standard Deviation (Std)**: The standard deviation values, which measure the spread

of the fitness values in the population, do not consistently decrease. This non-decreasing, indicates the maintenance of genetic diversity in the population, which is crucial for exploring different parts of the solution space.
- **Convergence**: Towards the later generations, the rate of improvement in average fitness decreases, which might suggest the algorithm is converging. This could mean that most individuals in the population are becoming increasingly similar to the best solution found, reducing the effectiveness of genetic diversity.

In [ ]:

```

```