Assignment 4

- Find the solution to 8-Queens Problem using local search method - genetic algorithm.

CODE:

```
# Calculate the fitness percentage of given arrangements

def calculate_fitness(board):
    n = len(board)
    max_conflicts = (n * (n - 1)) // 2

    def count_conflicts(board):
        conflicts = 0
        for i in range(n):
            for j in range(i + 1, n):
                if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                    conflicts += 1
        return conflicts

    conflicts = count_conflicts(board)
    fitness_percentage = (max_conflicts - conflicts) / max_conflicts * 100
    return fitness_percentage

figure_a = [3, 2, 7, 5, 2, 4, 1, 1]
figure_b = [2, 4, 7, 4, 8, 5, 5, 2]
figure_c = [3, 2, 5, 4, 3, 2, 1, 3]
figure_d = [2, 4, 4, 1, 5, 1, 2, 4]

# Calculate fitness percentages for each figure
fitness_a = calculate_fitness(figure_a)
fitness_b = calculate_fitness(figure_b)
fitness_c = calculate_fitness(figure_c)
fitness_d = calculate_fitness(figure_d)

# Print the fitness percentages
print("Fitness% of Figure A:", fitness_a)
print("Fitness% of Figure B:", fitness_b)
print("Fitness% of Figure C:", fitness_c)
print("Fitness% of Figure D:", fitness_d)
```

```
Fitness% of Figure A: 82.14285714285714
Fitness% of Figure B: 85.71428571428571
Fitness% of Figure C: 39.285714285714285
Fitness% of Figure D: 71.42857142857143
```

```python
import random

def fitness(board):
    # Calculate the number of conflicts between queens
    conflicts = 0
    n = len(board)

    for i in range(n):
        for j in range(i+1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def select_op(population, fitness_func):
    # Perform tournament selection to choose parents
    tournament_size = 3
    tournament = random.sample(population, tournament_size)
    tournament.sort(key=lambda x: fitness_func(x))
    return tournament[0]

def crossover(p1, p2):
    # Perform crossover to create a child
    n = len(p1)
    cp = random.randint(1, n-1)
    child = p1[:cp] + p2[cp:]
    return child

def mutation(child):
    # Perform mutation by changing a queen's position
    n = len(child)
    mutate_point = random.randint(0, n-1)
    new_value = random.randint(0, n-1)
    child[mutate_point] = new_value
    return child

def genetic_algorithm(population_size, n_queens):
    # Initialize the population with random queen placements
    population = [[random.randint(0, n_queens-1) for _ in range(n_queens)] for
_ in range(population_size)]
    gen_limit = 1000
    mut_prob = 0.5

    for generation in range(gen_limit):
        new_population = []
```

```
        for _ in range(population_size):
            p1 = select_op(population, fitness)
            p2 = select_op(population, fitness)
            child = crossover(p1, p2)
            if random.random() < mut_prob:
                child = mutation(child)
            new_population.append(child)

        population = new_population
        best_individual = min(population, key=fitness)
        if fitness(best_individual) == 0:
            print("Generation:", generation)
            # If a solution is found, print the generation number and exit
            return best_individual

    return min(population, key=fitness)

if __name__ == "__main__":
    population_size = 50
    n_queens = 8
    solution = genetic_algorithm(population_size, n_queens)
    print("Solution:", solution)
```

```
Generation: 34
Solution: [2, 4, 1, 7, 5, 3, 6, 0]
```