Aaron Bernard
CS441
Anthony Rhodes

Programming Assignment 2 Write-up

Summary of Findings, Technique, and Conclusions

In this project, I implemented a genetic algorithm to solve the 8-Queens problem. The goal was to find an arrangement of eight queens on a chessboard such that no two queens threaten each other. This involves ensuring that no two queens share the same row, column, or diagonal.

I experimented with various parameters such as population size, mutation probability, and the frequency of random restarts. The parameters used for the final version of the algorithm were:

- Population Size: 400

- Generations: 1000

- Mutation Probability: 0.3

- Random Restart Interval: 30 generations

- Elite Size: 2 (top 2 individuals carried over to the next generation)

Fitness Function and Mutation

- Fitness Function: The fitness function calculated the number of non-attacking pairs of queens. The maximum fitness for the 8-Queens problem is 28, representing all 8 queens being in non-attacking positions.

- Mutation: Mutation was performed with a probability of 0.3. During mutation, a random position in the individual (queen's position) was changed to a new random position.

Results and Observations

Despite various adjustments, the algorithm occasionally got stuck at a fitness level of 27, indicating one pair of queens still threatening each other. The introduction of random restarts helped maintain diversity in the population and occasionally helped escape local optima.

Plots

The plot below shows the average fitness and best fitness over generations:

Examples of Individuals

Below are examples of individuals sampled from different generations:

- Initial Population State: [7, 1, 5, 4, 0, 2, 4, 1]

- First Solution Found: [5, 2, 4, 6, 0, 3, 1, 7]

- First Perfect Solution Found in Generation 76: [5, 2, 4, 6, 0, 3, 1, 7]

- Best Solution Found: [5, 2, 4, 6, 0, 3, 1, 7]

- Fitness of Best Solution: 28

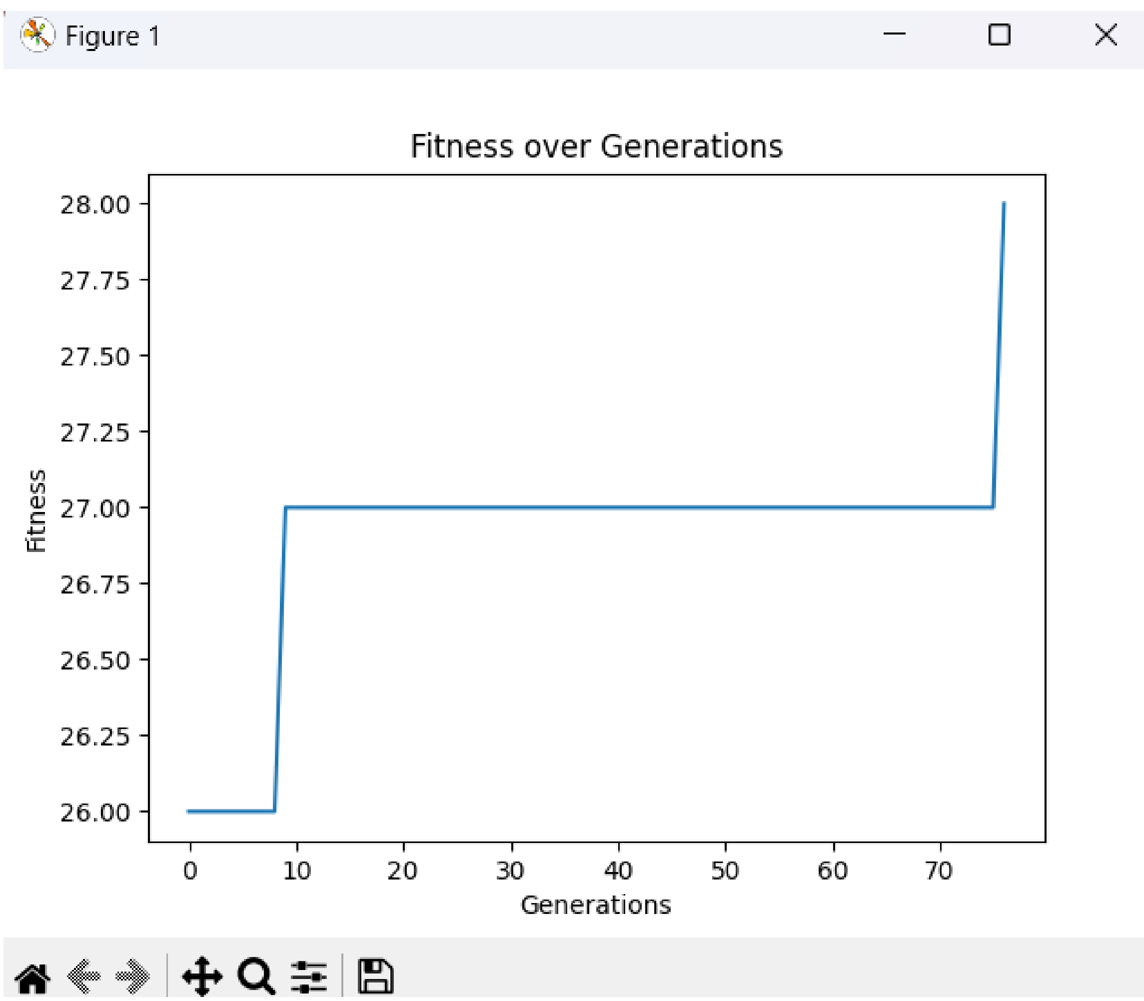
These examples illustrate the positions of queens on the chessboard. The genetic algorithm successfully converged to a solution where no two queens threaten each other.

Conclusion

The genetic algorithm demonstrated the ability to find solutions to the 8-Queens problem, though it occasionally required parameter tuning to avoid local optima. The visualization helped in understanding the progression of the algorithm over generations.

Code and Instructions

Fitness Plot



Code

```
import tkinter as tk

import random

import time

import matplotlib.pyplot as plt

# Genetic Algorithm Functions

def create_individual():

    return [random.randint(0, 7) for _ in range(8)]

def fitness(individual):

    n = len(individual)

    maxFitness = 28  # 8 queens can form 28 non-attacking pairs

    horizontal_collisions = sum([individual.count(queen) - 1 for queen in individual]) // 2

    main_diagonal_collisions = 0

    anti_diagonal_collisions = 0

    main_diagonals = [0] * (2 * n - 1)

    anti_diagonals = [0] * (2 * n - 1)

    for i in range(n):

        main_diagonals[i - individual[i] + (n - 1)] += 1

        anti_diagonals[i + individual[i]] += 1

    for count in main diagonals:

        if count > 1:

            main diagonal collisions += (count - 1)

    for count in anti diagonals:

        if count > 1:

            anti diagonal collisions += (count - 1)
```

```
    total collisions = horizontal collisions + main diagonal collisions + anti diagonal
collisions

    return int(maxFitness - total collisions)

def probability(individual, fitness):

    return fitness(individual) / 28  # Max fitness is 28

def random_pick(population, probabilities):

    total = sum(probabilities)

    r = random.uniform(0, total)

    upto = 0

    for i, probability in enumerate(probabilities):

        if upto + probability >= r:

            return population[i]

        upto += probability

    assert False, "Shouldn't get here"

def reproduce(x, y):

    n = len(x)

    c = random.randint(0, n - 1)

    return x[:c] + y[c:]

def mutate(individual):

    n = len(individual)

    c = random.randint(0, n - 1)

    m = random.randint(0, n - 1)

    individual[c] = m

    return individual

# Visualization Functions
```

```
def draw_board(canvas, individual):

    canvas.delete("all")

    for row in range(8):

        for col in range(8):

            color = "white" if (row + col) % 2 == 0 else "black"

            canvas.create_rectangle(col*50, row*50, (col+1)*50, (row+1)*50, fill=color)

    for col, row in enumerate(individual):

        canvas.create_oval(col*50 + 10, row*50 + 10, col*50 + 40, row*50 + 40, fill="red")

def genetic_algorithm_visualized(canvas, population, fitness, mutation_probability=0.3, 
generations=1000, elite_size=2, random_restart_interval=30):

    maxFitness = 28

    fitness_history = []

    for generation in range(generations):

        population sort key=fitness, reverse=True)

        new_population = population[:elite_size]  # Carry over the best individuals

        probabilities = [probability(n, fitness) for n in population]

        for _ in range((len(population) - elite_size) // 2):

            parent1 = random_pick(population, probabilities)

            parent2 = random_pick(population, probabilities)

            child1 = reproduce(parent1, parent2)

            child2 = reproduce(parent2, parent1)

            if random.random() < mutation probability:

                child1 = mutate(child1)

            if random.random() < mutation probability:

                child2 = mutate(child2)
```

```
            new_population.append(child1)

            new_population.append(child2)

        population = new_population

        best_individual = max(population, key=fitness)

        current_fitness = fitness(best_individual)

        fitness history.append(current fitness)

        draw_board(canvas, best_individual)

        canvas update()

        time.sleep(0.1)  # Slow down the visualization for observation

        print(f"Generation {generation}: Best Fitness = {current_fitness}")

        if current fitness == maxFitness:

            print(f"Perfect solution found in generation {generation}: {best_individual}")

            break  # This ensures the loop stops once the perfect solution is found

        if generation % random restart interval == 0:

            # Introduce new random individuals to the population

            new_individuals = [create_individual() for _ in range(len(population) // 2)]

            population[-len(new individuals):] = new individuals

    return population, fitness history

# Main Function

def main():

    root = tk.Tk()

    root.title("8-Queens Genetic Algorithm Visualization")

    canvas = tk.Canvas(root, width=400, height=400)

    canvas.pack()

    population size = 400  # Further increased population size for more diversity
```

```
    generations = 1000
    mutation probability = 0.3  # Further increased mutation probability for more exploration
    random restart interval = 30  # More frequent random restarts
    population = [create individual() for _ in range(population size)]
    solution, fitness history = genetic_algorithm_visualized(canvas, population, fitness, mutation probability, generations, elite size=2, random restart interval=random restart interval)
    best_individual = max(solution, key=fitness)
    print("Best solution found:", best_individual)
    print("Fitness of best solution:", fitness(best_individual))
    plt.plot(fitness history)
    plt.title("Fitness over Generations")
    plt.xlabel("Generations")
    plt.ylabel("Fitness")
    plt.show()
    root.mainloop()
if __name__ == "__main__":
    main()
```

Instructions on How to Compile and Run the Code

1. Requirements: will need the tkinter and matplotlib libraries. You can install them using pip:

```
pip install matplotlib
```

2. Running the Code: Save the provided code into a file named 8-Queens.py. Run:

```
python 8-Queens.py
```