Complex Engineering Problem Report

ARTIFICIAL INTELLIGENCE

Third Year-Computer and Information Systems Engineering

Batch: 2022

Group Members:	
Azka Sohail	(CS-22002)
Rubas Sadeem Ansari	(CS -22004)
Sidra Tariq	(CS -22010)

Submitted to: Miss Hameezah Ahmed

Submission Date: 26/11/2024

ABSTRACT:

This project introduces a robust Sudoku-solving system utilizing the ***Genetic Algorithm (GA)****, an optimization technique inspired by the principles of natural selection and evolution. Sudoku, being an NP-complete problem, poses significant challenges to traditional deterministic methods, especially when dealing with large and complex grids. To address this, the implemented system leverages GA's adaptability and stochastic nature to efficiently explore the solution space and evolve grids toward a valid Sudoku solution.*

The algorithm begins by generating an initial population of random Sudoku grids, ensuring diversity and maintaining constraints from the input puzzle. It then iteratively refines this population through three primary genetic operations: ***selection****,* ***crossover****, and* ***mutation****. Selection identifies the fittest individuals based on a fitness function that penalizes rule violations in rows, columns, and 3x3 subgrids. Crossover allows the exchange of genetic material between individuals to create offspring with mixed traits, while mutation introduces variability by randomly altering genes, further enhancing diversity and avoiding premature convergence.*

Over successive generations, the algorithm fine-tunes the population, progressively reducing fitness penalties and converging toward grids with minimal or no violations. The final output is a grid with the highest fitness score, representing an optimized Sudoku solution. The use of GA in this project is particularly advantageous due to its scalability and ability to handle the combinatorial complexity of Sudoku, providing an efficient and elegant approach to solving such problems.

TABLE OF CONTENTS:

INTRODUCTION:

Sudoku is a popular logic-based puzzle that requires filling a 9x9 grid with numbers from 1 to 9, ensuring no repetition in rows, columns, or 3x3 subgrids. Due to its NP-complete nature, finding solutions for complex Sudoku puzzles is computationally challenging. This project employs a **Genetic Algorithm (GA)** to solve Sudoku puzzles efficiently. GA, inspired by the process of natural selection, iteratively optimizes solutions by mimicking biological evolution through selection, crossover, and mutation. This approach is well-suited for exploring large search spaces, making it a practical choice for solving Sudoku puzzles.

PROBLEM STATEMENT:

Traditional methods for solving Sudoku puzzles often rely on brute force or heuristic techniques, which can be computationally expensive and inefficient for complex puzzles. The challenge lies in finding a scalable and adaptive approach to solve puzzles while adhering to Sudoku's constraints. This project addresses the problem by implementing a GA-based system that optimizes solutions over successive generations, ensuring efficient exploration of the solution space and convergence to valid Sudoku grids.

WHY GENETIC ALGORITHM?

Genetic algorithms are a type of optimization technique inspired by the process of natural selection. They are particularly useful for solving complex problems like Sudoku, where the search space is large and the optimal solution is not easily identifiable. Genetic algorithms are optimal for this problem because they can efficiently explore the search space and converge to a good solution. They are also robust and can handle noisy or incomplete data. However, genetic algorithms are not guaranteed to find the global optimum, and the quality of the solution depends on the choice of parameters and the fitness function. Additionally, genetic algorithms can be computationally expensive, especially for large problems.

GENETIC ALGORITHM:

COMPONENTS OF GENETIC ALGORITHM:

Inspired by natural selection and genetics genetic algorithm uses biological terminology:

POPULATION: Population is a set of candidate solutions (chromosomes) that evolve over generations. It's initialized randomly or using heuristics. Population size affects convergence speed and diversity.

CHROMOSOME (SOLUTION): A chromosome is a blueprint carrying genetic information that determines characteristics. It's made of DNA and proteins, and comes in pairs. In genetics, chromosomes pass traits from parents to offspring. In Genetic Algorithms, chromosomes represent solutions that evolve through mutation, crossover, and selection

GENE (VARIABLE): A gene is a segment of DNA that carries information to create proteins. Genes determine traits, characteristics, and functions in living organisms. They are passed from parents to offspring and influence growth, development, and behavior. In Genetic Algorithms, genes represent variables or features that combine to form solutions.

DNA (SOLUTION REPRESENTATION): In Genetic Algorithms, DNA represents a solution or chromosome. It's typically encoded as a binary string, real-valued vector, or permutation. DNA is composed of genes (variables) that combine to form a solution. Operations like mutation, crossover, and selection modify the DNA to evolve better solutions.

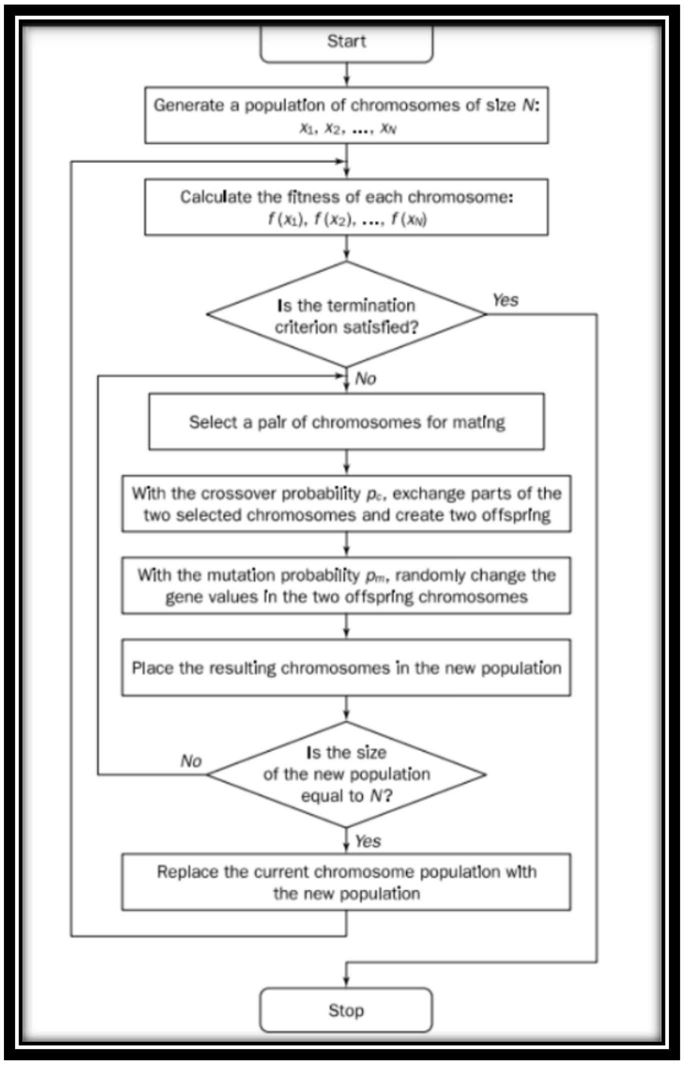
OPERATORS OF GENETIC ALGORITHM:

MUTATION (RANDOM CHANGE): Mutation is a genetic operator that introduces random changes to a DNA (solution) in Genetic Algorithms. It alters one or more genes (variables) to create a new solution. Mutation helps maintain diversity, avoids local optima, and explores new search spaces.

CROSSOVER (COMBINATION): Crossover is a genetic operator that combines two parent DNA (solutions) to create new offspring solutions in Genetic Algorithms. It exchanges genetic information, mixing variables (genes) to produce diverse solutions

SELECTION (SURVIVAL OF THE FITTEST): Selection, or "survival of the fittest," is a genetic operator that chooses the best solutions based on fitness to reproduce. It guides the algorithm toward optimal solutions by ensuring that the most fit individuals pass on their genetic traits, improving the population with each generation. Various selection methods, like tournament or roulette wheel selection, help balance diversity and convergence in the population.

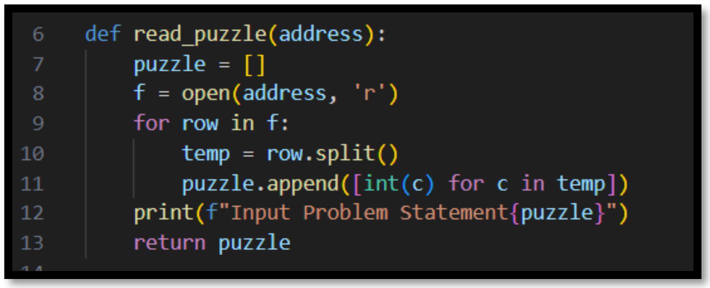
WORKING OF GENETIC ALGORITHM:


Start
Generate a population of chromosomes of size N: $x_1, x_2, \ldots, x_N$
Calculate the fitness of each chromosome: $f(x_1), f(x_2), \ldots, f(x_N)$
Is the termination criterion satisfied?
Yes
No
Select a pair of chromosomes for mating
With the crossover probability p_c, exchange parts of the two selected chromosomes and create two offspring
With the mutation probability p_m, randomly change the gene values in the two offspring chromosomes
Place the resulting chromosomes in the new population
Is the size of the new population equal to N?
No
Yes
Replace the current chromosome population with the new population
Stop

WORKING OF THE CODE:

1. read_puzzle(address)

- **Purpose:** Reads the Sudoku puzzle from a file.
- **How it works:**

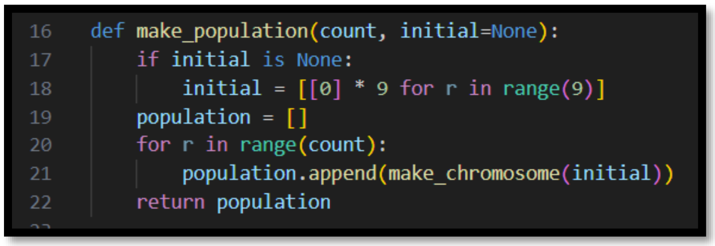


```python
def read_puzzle(address):
    puzzle = []
    f = open(address, 'r')
    for row in f:
        temp = row.split()
        puzzle.append([int(c) for c in temp])
    print(f"Input Problem Statement{puzzle}")
    return puzzle
```

✓ It opens the file at the given address and reads each row of the puzzle.
✓ The row is split into individual string elements and converted into integers.
✓ Returns the puzzle as a 9x9 grid (list of lists).

2. make_population(count, initial=None)

- **Purpose:** Creates an initial population of possible solutions (chromosomes).
- **How it works:**

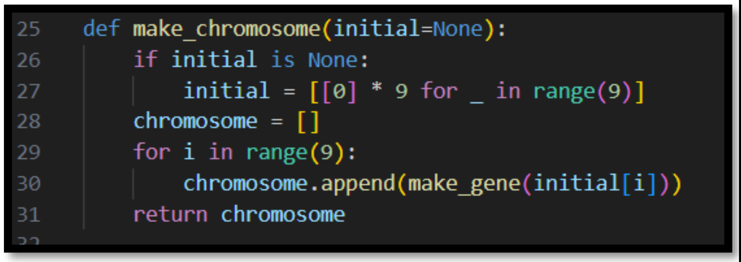


```python
def make_population(count, initial=None):
    if initial is None:
        initial = [[0] * 9 for r in range(9)]
    population = []
    for r in range(count):
        population.append(make_chromosome(initial))
    return population
```

✓ If initial is not provided, a grid of zeros (9x9) is created.
✓ Creates count number of chromosomes (population) by calling make_chromosome for each.
✓ Returns the population.

3. make_chromosome(initial=None)

- **Purpose:** Generates a single chromosome (possible solution).
- **How it works:**

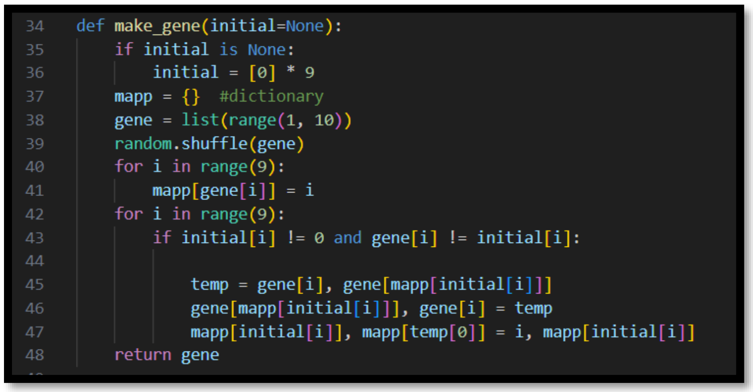


```python
def make_chromosome(initial=None):
    if initial is None:
        initial = [[0] * 9 for _ in range(9)]
    chromosome = []
    for i in range(9):
        chromosome.append(make_gene(initial[i]))
    return chromosome
```

✓ For each row in the 9x9 grid, it generates a list of numbers (genes) using make_gene.
✓ Returns the chromosome (a 9x9 grid of numbers).

4. make_gene(initial=None)

- **Purpose:** Creates a gene (row of the puzzle) by shuffling the values from 1 to 9.
- **How it works:**



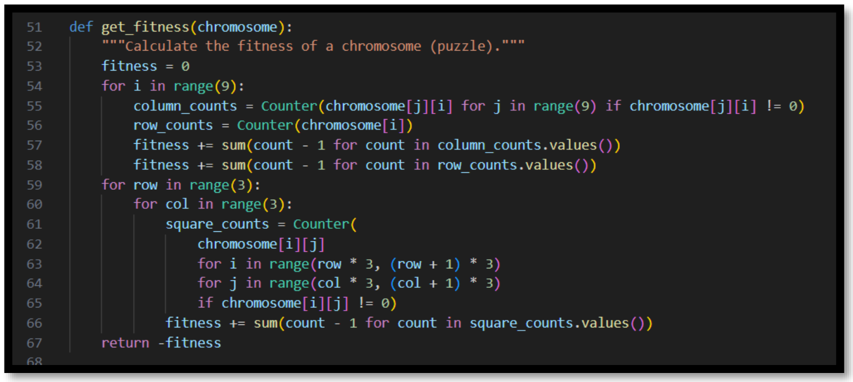
```python
def make_gene(initial=None):
    if initial is None:
        initial = [0] * 9
    mapp = {}  #dictionary
    gene = list(range(1, 10))
    random.shuffle(gene)
    for i in range(9):
        mapp[gene[i]] = i
    for i in range(9):
        if initial[i] != 0 and gene[i] != initial[i]:

            temp = gene[i], gene[mapp[initial[i]]]
            gene[mapp[initial[i]]], gene[i] = temp
            mapp[initial[i]], mapp[temp[0]] = i, mapp[initial[i]]
    return gene
```

✓ Randomly shuffles the numbers from 1 to 9.
✓ If a number in the row is not zero, it ensures that the shuffled value corresponds to the initial number by swapping positions.
✓ Returns a valid gene (row).

5. get_fitness(chromosome)

- **Purpose:** Evaluates the fitness of a given chromosome.
- **How it works:**

- ✓ The fitness is calculated by checking for duplicate values in rows, columns, and 3x3 subgrids.
- ✓ Each duplicate in rows, columns, or subgrids increases the fitness penalty.
- ✓ Returns the negative fitness value (higher penalties mean lower fitness).

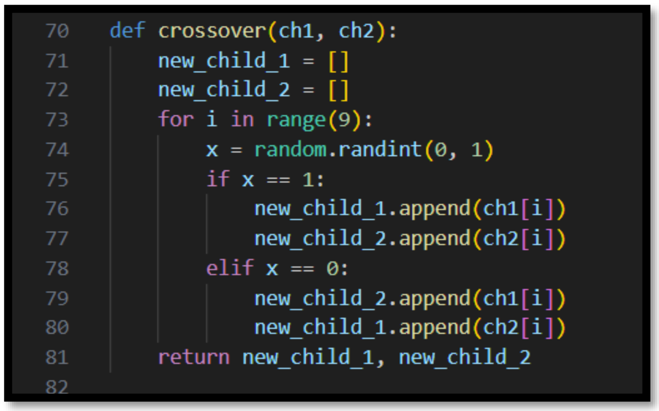


```python
def get_fitness(chromosome):
    """Calculate the fitness of a chromosome (puzzle)."""
    fitness = 0
    for i in range(9):
        column_counts = Counter(chromosome[j][i] for j in range(9) if chromosome[j][i] != 0)
        row_counts = Counter(chromosome[i])
        fitness += sum(count - 1 for count in column_counts.values())
        fitness += sum(count - 1 for count in row_counts.values())
    for row in range(3):
        for col in range(3):
            square_counts = Counter(
                chromosome[i][j]
                for i in range(row * 3, (row + 1) * 3)
                for j in range(col * 3, (col + 1) * 3)
                if chromosome[i][j] != 0)
            fitness += sum(count - 1 for count in square_counts.values())
    return -fitness
```

6. crossover(ch1, ch2)

- **Purpose:** Performs crossover (mating) between two chromosomes to create offspring.
- **How it works:**

- ✓ Iterates through the rows, randomly choosing whether to swap rows between two chromosomes.
- ✓ Returns two new offspring chromosomes.

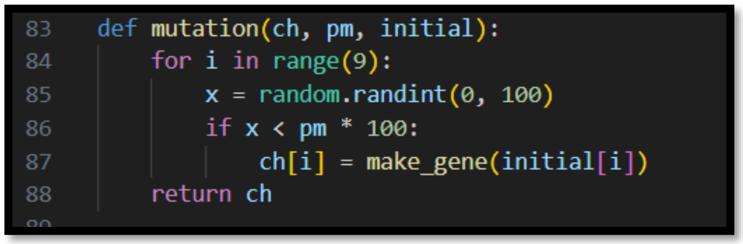


```python
def crossover(ch1, ch2):
    new_child_1 = []
    new_child_2 = []
    for i in range(9):
        x = random.randint(0, 1)
        if x == 1:
            new_child_1.append(ch1[i])
            new_child_2.append(ch2[i])
        elif x == 0:
            new_child_2.append(ch1[i])
            new_child_1.append(ch2[i])
    return new_child_1, new_child_2
```

7. mutation(ch, pm, initial)

- **Purpose:** Performs mutation on a chromosome.
- **How it works:**

- ✓ For each row, a mutation occurs with a probability pm. If mutation occurs, it generates a new gene (row) for that position using make_gene.
- ✓ Returns the mutated chromosome.

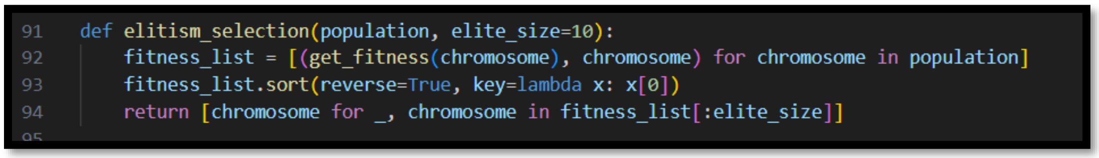


```python
def mutation(ch, pm, initial):
    for i in range(9):
        x = random.randint(0, 100)
        if x < pm * 100:
            ch[i] = make_gene(initial[i])
    return ch
```

8. elitism_selection(population, elite_size=10)

- **Purpose:** Selects the best chromosomes (elite) based on fitness.
- **How it works:**

- ✓ Calculates the fitness for each chromosome.
- ✓ Sorts the population by fitness (highest first).
- ✓ Returns the top elite_size chromosomes.

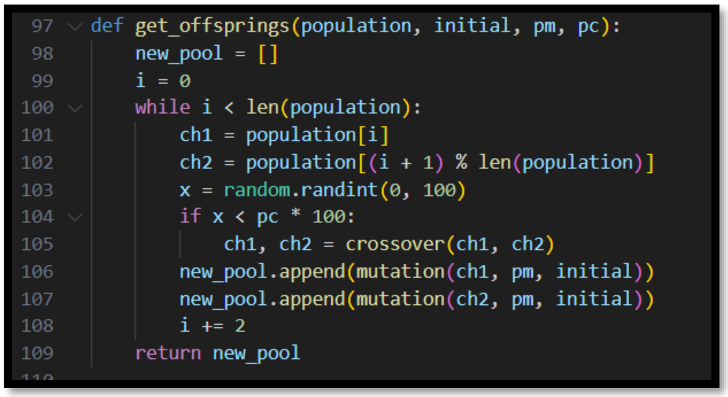


```python
def elitism_selection(population, elite_size=10):
    fitness_list = [(get_fitness(chromosome), chromosome) for chromosome in population]
    fitness_list.sort(reverse=True, key=lambda x: x[0])
    return [chromosome for _, chromosome in fitness_list[:elite_size]]
```

9. get_offsprings(population, initial, pm, pc)

- **Purpose:** Generates offspring for the next generation.
- **How it works:**

✓ Iterates through pairs of chromosomes, performing crossover with probability pc and mutation with probability pm.
✓ Returns the new population of offspring.

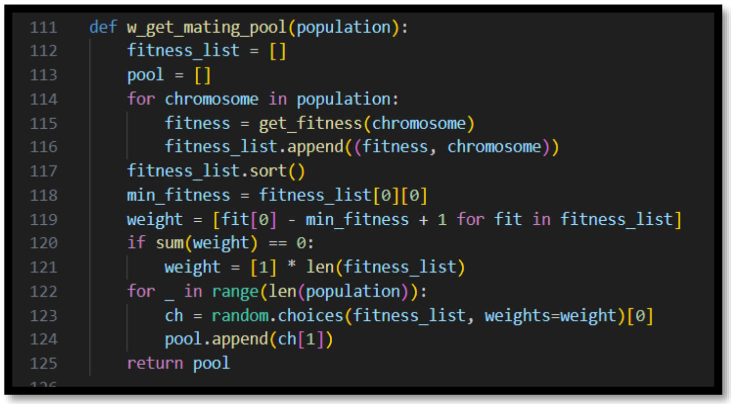


```python
def get_offsprings(population, initial, pm, pc):
    new_pool = []
    i = 0
    while i < len(population):
        ch1 = population[i]
        ch2 = population[(i + 1) % len(population)]
        x = random.randint(0, 100)
        if x < pc * 100:
            ch1, ch2 = crossover(ch1, ch2)
        new_pool.append(mutation(ch1, pm, initial))
        new_pool.append(mutation(ch2, pm, initial))
        i += 2
    return new_pool
```

10. w_get_mating_pool(population)

- **Purpose:** Selects a mating pool based on fitness, where chromosomes with better fitness are more likely to be selected.
- **How it works:**

✓ Calculates the fitness of each chromosome and normalizes it by subtracting the minimum fitness.
✓ Uses the fitness values as weights to randomly select chromosomes for the mating pool.
✓ Returns the mating pool for the next generation.

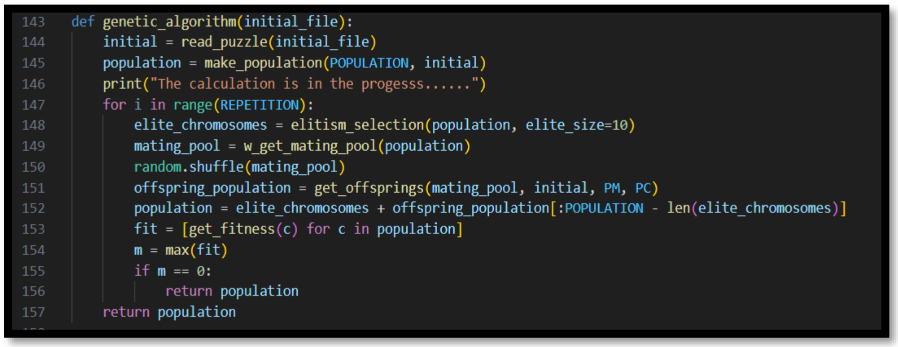


```python
def w_get_mating_pool(population):
    fitness_list = []
    pool = []
    for chromosome in population:
        fitness = get_fitness(chromosome)
        fitness_list.append((fitness, chromosome))
    fitness_list.sort()
    min_fitness = fitness_list[0][0]
    weight = [fit[0] - min_fitness + 1 for fit in fitness_list]
    if sum(weight) == 0:
        weight = [1] * len(fitness_list)
    for _ in range(len(population)):
        ch = random.choices(fitness_list, weights=weight)[0]
        pool.append(ch[1])
    return pool
```

11. genetic_algorithm(initial_file)

- **Purpose:** Main genetic algorithm function to solve the Sudoku puzzle.
- **How it works:**

✓ Reads the initial puzzle from the file.
✓ Initializes the population.
✓ Iterates for a set number of generations (REPETITION).
✓ In each generation, it performs elitism selection, generates offspring, and combines them with the elite chromosomes.
✓ The loop continues until a perfect solution is found (fitness = 0).
✓ Returns the final population.

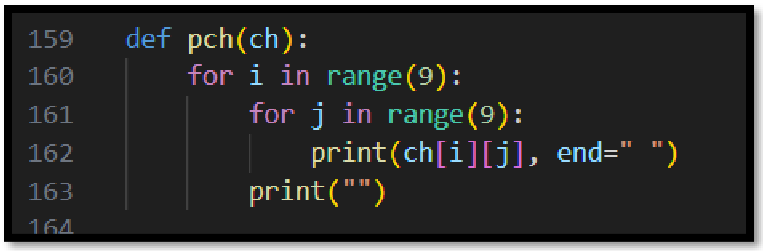


```python
def genetic_algorithm(initial_file):
    initial = read_puzzle(initial_file)
    population = make_population(POPULATION, initial)
    print("The calculation is in the progesss......")
    for i in range(REPETITION):
        elite_chromosomes = elitism_selection(population, elite_size=10)
        mating_pool = w_get_mating_pool(population)
        random.shuffle(mating_pool)
        offspring_population = get_offsprings(mating_pool, initial, PM, PC)
        population = elite_chromosomes + offspring_population[:POPULATION - len(elite_chromosomes)]
        fit = [get_fitness(c) for c in population]
        m = max(fit)
        if m == 0:
            return population
    return population
```

12. pch(ch)

- **Purpose:** Prints a 9x9 grid (Sudoku solution).
- **How it works:**

✓ Prints each row of the chromosome (Sudoku solution).

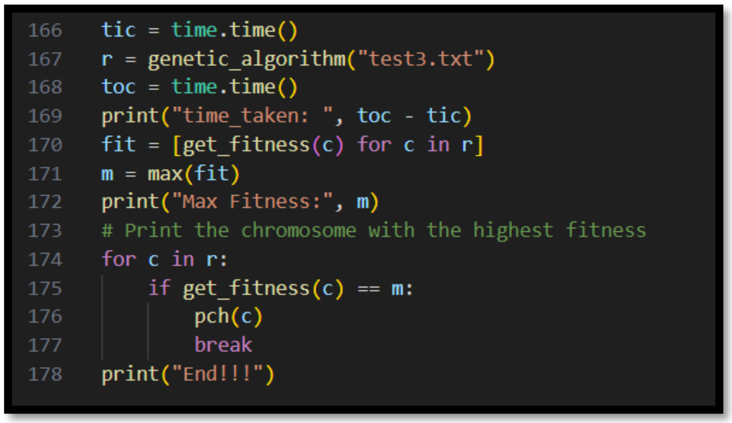


```python
def pch(ch):
    for i in range(9):
        for j in range(9):
            print(ch[i][j], end=" ")
        print("")
```

13. Main Execution:

- **Purpose:** Runs the genetic algorithm and measures the time taken.
- **How it works:**

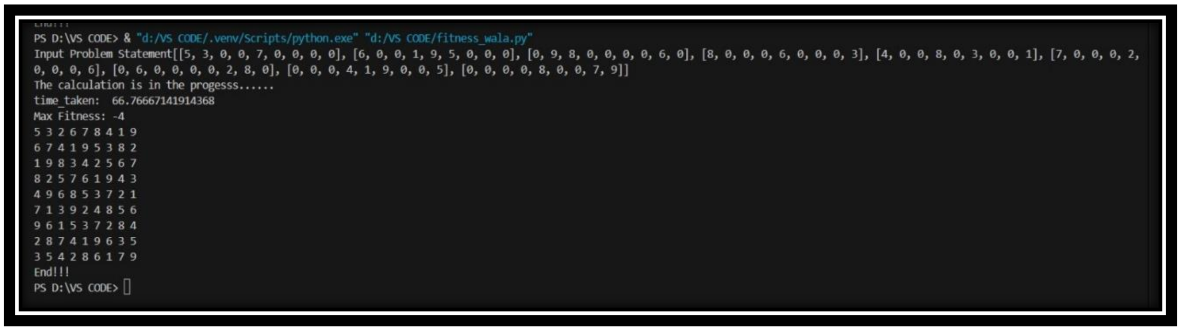
✓ Calls genetic_algorithm with the initial puzzle file.
✓ Prints the time taken for the algorithm to run.
✓ Finds and prints the solution with the highest fitness (or the optimal solution).



```python
tic = time.time()
r = genetic_algorithm("test3.txt")
toc = time.time()
print("time_taken: ", toc - tic)
fit = [get_fitness(c) for c in r]
m = max(fit)
print("Max Fitness:", m)
# Print the chromosome with the highest fitness
for c in r:
    if get_fitness(c) == m:
        pch(c)
        break
print("End!!!")
```

RESULT:

The genetic algorithm successfully solved the Sudoku puzzle, converging to a valid solution after a set number of generations. The solution satisfied all Sudoku rules, with no repetition in rows, columns, or sub-grids, demonstrating the algorithm's effectiveness in solving complex optimization problems.



```
PS D:\VS CODE> & "d:/VS CODE/.venv/Scripts/python.exe" "d:/VS CODE/fitness_wala.py"
Input Problem Statement[[5, 3, 0, 0, 7, 0, 0, 0, 0], [6, 0, 0, 1, 9, 5, 0, 0, 0], [0, 9, 8, 0, 0, 0, 0, 6, 0], [8, 0, 0, 0, 6, 0, 0, 0, 3], [4, 0, 0, 8, 0, 3, 0, 0, 1], [7, 0, 0, 0, 2,
0, 0, 0, 6], [0, 6, 0, 0, 0, 0, 2, 8, 0], [0, 0, 0, 4, 1, 9, 0, 0, 5], [0, 0, 0, 0, 8, 0, 0, 7, 9]]
The calculation is in the progesss......
time_taken:  66.76667141914368
Max Fitness: -4
5 3 2 6 7 8 4 1 9
6 7 4 1 9 5 3 8 2
1 9 8 3 4 2 5 6 7
8 2 5 7 6 1 9 4 3
4 9 6 8 5 3 7 2 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 5 4 2 8 6 1 7 9
End!!!
PS D:\VS CODE> 
```

CONCLUSION:

This code implements a genetic algorithm to solve a Sudoku puzzle. It starts by reading the initial puzzle from a file and creates a population of random Sudoku grids (chromosomes). Through crossover and mutation, new generations of solutions are evolved. The algorithm evaluates the fitness of each solution based on the number of duplicates in rows, columns, and 3x3 grids, aiming to minimize these duplicates. Elite chromosomes (best solutions) are preserved, while others are modified through genetic operations. The process continues until an optimal solution (a completed Sudoku puzzle) is found or the maximum number of generations is reached. The code effectively uses genetic algorithm principles to approach problem-solving with randomness and fitness-based selection.