National University of Computer & Emerging Sciences, Karachi

Spring -2023
AL2002-Artificial Intelligence - Lab
Lab 05

Roll No#20K-0190 **Name: Mohamamd Usama**

Task 01:

```python
import random

# Define the genes to be used in the population
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

# Define the target string to be generated
TARGET = "Artificial Intelligence Lab"

# Define the size of the population
POPULATION_SIZE = 70

# Define the maximum number of generations to run
MAX_GENERATIONS = 1000

# Define the mutation rate
MUTATION_RATE = 0.1

# Define a function to generate a random chromosome
def generate_chromosome():
    return [random.choice(GENES) for _ in range(len(TARGET))]

# Define a function to calculate the fitness score of a chromosome
def calculate_fitness(chromosome):
    return sum([1 for i in range(len(TARGET)) if chromosome[i] != TARGET[i]])

# Define a function to select parents for crossover
def selection(population):
    return random.choices(population, weights=[1/fitness for fitness in [calculate_fitness(chromosome) for chromosome in population]], k=2)

# Define a function to perform crossover
def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(TARGET) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2
```

```python
    return mutated_chromosome

# Define the main function to run the genetic algorithm
def run_genetic_algorithm():
    # Generate the initial population
    population = [generate_chromosome() for _ in range(POPULATION_SIZE)]

    # Iterate over generations
    for generation in range(MAX_GENERATIONS):
        # Evaluate the fitness of each chromosome in the population
        fitness_scores = [calculate_fitness(chromosome) for chromosome in population]

        # Check if we've reached the target
        if 0 in fitness_scores:
            index = fitness_scores.index(0)
            return ''.join(population[index]), generation

        # Select parents for crossover
        parent1, parent2 = selection(population)

        # Perform crossover to generate two children
        child1, child2 = crossover(parent1, parent2)

        # Perform mutation on the children
        mutated_child1 = mutation(child1)
        mutated_child2 = mutation(child2)

        # Add the children to the population
        population.append(mutated_child1)
        population.append(mutated_child2)

        # Remove the two least fit chromosomes from the population
        least_fit_index = fitness_scores.index(max(fitness_scores))
        del population[least_fit_index]
        del fitness_scores[least_fit_index]
        second_least_fit_index = fitness_scores.index(max(fitness_scores))
        del population[second_least_fit_index]

    # Return the best chromosome and the number of generations it took to reach it
    index = fitness_scores.index(min(fitness_scores))
    return ''.join(population[index]), MAX_GENERATIONS
```

```python
# Run the genetic algorithm and print the results
best_chromosome, generations = run_genetic_algorithm()
print(f"Target: {TARGET}")
print(f"Best Chromosome: {best_chromosome}")
print(f"Generations: {generations}")
```

```
Target: Artificial Intelligence Lab
Best Chromosome: H9&Wfi1iaJ Cn"@#ligln1x yPb
Generations: 1000
```

Task 02

```python
[ ] import numpy as np
import random

# Define the cities and distances
cities = ["A", "B", "C", "D"]
distances = np.array([
  [0, 3, 6, 2],
  [3, 0, 4, 7],
  [6, 4, 0, 4],
  [2, 7, 4, 0],
])

# Define the genetic algorithm parameters
POPULATION_SIZE = 20
MUTATION_RATE = 0.1
NUM_GENERATIONS = 50

# Create the initial population
def create_population(size):
    population = []
    for i in range(size):
        chromosome = list(range(1, len(cities)))
        random.shuffle(chromosome)
        chromosome.insert(0, 0)
        chromosome.append(0)
        population.append(chromosome)
    return population

# Calculate the fitness of a chromosome
def calculate_fitness(chromosome):
    distance = 0
    for i in range(len(chromosome)-1):
        distance += distances[chromosome[i], chromosome[i+1]]
    fitness = 1/distance
    return fitness

# Perform mutation on a chromosome
def mutate(chromosome):
    idx1, idx2 = random.sample(range(1,4), 2)
    chromosome[idx1], chromosome[idx2] = chromosome[idx2], chromosome[idx1]
    return chromosome

# Select parents for crossover
def select_parents(population):
    fitnesses = [calculate_fitness(chromosome) for chromosome in population]
    total_fitness = sum(fitnesses)
    probabilities = [fitness/total_fitness for fitness in fitnesses]
    parents = random.choices(population, weights=probabilities, k=2)
    return parents
```

```python
    return parents

# Perform crossover on parents to create offspring
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1)-2)
    child1 = parent1[:crossover_point] + [i for i in parent2 if i not in parent1[:crossover_point]] + parent1[crossover_point:]
    child2 = parent2[:crossover_point] + [i for i in parent1 if i not in parent2[:crossover_point]] + parent2[crossover_point:]
    return child1, child2

# Perform the genetic algorithm
def genetic_algorithm():
    population = create_population(POPULATION_SIZE)
    for generation in range(NUM_GENERATIONS):
        new_population = []
        for i in range(POPULATION_SIZE):
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            if random.uniform(0, 1) < MUTATION_RATE:
                child1 = mutate(child1)
            if random.uniform(0, 1) < MUTATION_RATE:
                child2 = mutate(child2)
            new_population.append(child1)
            new_population.append(child2)
        population = new_population
        best_chromosome = max(population, key=calculate_fitness)
        print("Generation {}: Best chromosome = {}, Fitness = {:.4f}".format(generation+1, best_chromosome, calculate_fitness(best_chromosome)))
    return best_chromosome

# Run the genetic algorithm and print the best solution
best_solution = genetic_algorithm()
best_distance = 1/calculate_fitness(best_solution)
print("Best solution: {}, Distance: {:.4f}".format(best_solution, best_distance))
```

```
Generation 1: Best chromosome = [0, 1, 2, 3, 3, 0], Fitness = 0.0769
Generation 2: Best chromosome = [0, 3, 2, 1, 1, 0], Fitness = 0.0769
Generation 3: Best chromosome = [0, 3, 2, 1, 1, 0], Fitness = 0.0769
Generation 4: Best chromosome = [0, 3, 2, 1, 1, 1, 0], Fitness = 0.0769
Generation 5: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 6: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 7: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 8: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 9: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 10: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 11: Best chromosome = [0, 3, 2, 1, 1, 1, 0], Fitness = 0.0769
Generation 12: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 13: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 14: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 15: Best chromosome = [0, 2, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0526
Generation 16: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 17: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 18: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 19: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 20: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
```


Generation 16: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 17: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 18: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 19: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 20: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769


```
Generation 21: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 22: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 23: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 24: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 25: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 26: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 27: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 28: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 29: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 30: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 31: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 32: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 33: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 34: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 35: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 36: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 37: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 38: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 39: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 40: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 41: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 42: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 43: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 44: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 45: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 46: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 47: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 48: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 49: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Generation 50: Best chromosome = [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Fitness = 0.0769
Best solution: [0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 0], Distance: 13.0000
```

Task 03

```
[ ] import heapq

class Board:
    def __init__(self, state, g, h):
        self.state = state
        self.g = g
        self.h = h

    def __lt__(self, other):
        return self.g + self.h < other.g + other.h

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(str(self.state))

    def is_goal(self):
        return self.h == 0

    def get_successors(self):
        successors = []
        zero_index = self.state.index(0)
        row, col = zero_index // 3, zero_index % 3

        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_index = new_row * 3 + new_col
                new_state = self.state[:]
                new_state[zero_index], new_state[new_index] = new_state[new_index], new_state[zero_index]
                successors.append(Board(new_state, self.g + 1, manhattan_distance(new_state)))

        return successors

def manhattan_distance(state):
    distance = 0
    for i in range(9):
        row, col = i // 3, i % 3
        value = state[i]
        if value != 0:
            goal_row, goal_col = (value - 1) // 3, (value - 1) % 3
            distance += abs(row - goal_row) + abs(col - goal_col)
    return distance
```

```
[ ] def solve_8_puzzle(initial_state):
    initial_board = Board(initial_state, 0, manhattan_distance(initial_state))
    open_list = [initial_board]
    closed_set = set()

    while open_list:
        current_board = heapq.heappop(open_list)
        if current_board.is_goal():
            return current_board.g, current_board.state

        closed_set.add(current_board)

        for successor in current_board.get_successors():
            if successor in closed_set:
                continue

            if successor not in open_list:
                heapq.heappush(open_list, successor)
            else:
                existing_board = open_list[open_list.index(successor)]
                if existing_board.g > successor.g:
                    existing_board.g = successor.g
                    existing_board.h = successor.h

    return None

initial_state = [1, 2, 3, 0, 4, 6, 7, 5, 8]
print("Initial state: ")
print(initial_state[0:3])
print(initial_state[3:6])
print(initial_state[6:9])

result = solve_8_puzzle(initial_state)

if result is not None:
    num_moves, final_state = result
    print("Solution found in {} moves: ".format(num_moves))
    print(final_state[0:3])
    print(final_state[3:6])
    print(final_state[6:9])
else:
    print("No solution found")

print()
```

```python
result = solve_8_puzzle(initial_state)

if result is not None:
    num_moves, final_state = result
    print("Solution found in {} moves: ".format(num_moves))
    print(final_state[0:3])
    print(final_state[3:6])
    print(final_state[6:9])
else:
    print("No solution found")

print()
```

```
Initial state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Solution found in 3 moves:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

Task 04

```python
import heapq

def get_blank_pos(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                target_row = (value - 1) // 3
                target_col = (value - 1) % 3
                distance += abs(i - target_row) + abs(j - target_col)
    return distance

def get_neighbors(state):
    neighbors = []
    i, j = get_blank_pos(state)
    if i > 0:
        new_state = [row[:] for row in state]
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
        neighbors.append(new_state)
    if i < 2:
        new_state = [row[:] for row in state]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Task 05

```
[ ] import numpy as np
    import random

    # Define the cities and distances
    cities = ["A", "B", "C", "D"]
    distances = np.array([
      [0, 3, 6, 2],
      [3, 0, 4, 7],
      [6, 4, 0, 4],
      [2, 7, 4, 0],
    ])

    # Define the genetic algorithm parameters
    POPULATION_SIZE = 20
    MUTATION_RATE = 0.1
    NUM_GENERATIONS = 50

    # Create the initial population
    def create_population(size):
        population = []
        for i in range(size):
            chromosome = list(range(1, len(cities)))
            random.shuffle(chromosome)
            chromosome.insert(0, 0)
            chromosome.append(0)
            population.append(chromosome)
        return population

    # Calculate the fitness of a chromosome
    def calculate_fitness(chromosome):
        distance = 0
        for i in range(len(chromosome)-1):
            distance += distances[chromosome[i], chromosome[i+1]]
        fitness = 1/distance
        return fitness

    # Perform mutation on a chromosome
    def mutate(chromosome):
        idx1, idx2 = random.sample(range(1,4), 2)
        chromosome[idx1], chromosome[idx2] = chromosome[idx2], chromosome[idx1]
        return chromosome
```

```
# Select parents for crossover
def select_parents(population):
    fitnesses = [calculate_fitness(chromosome) for chromosome in population]
    total_fitness = sum(fitnesses)
    probabilities = [fitness/total_fitness for fitness in fitnesses]
    parents = random.choices(population, weights=probabilities, k=2)
    return parents

# Perform crossover on parents to create offspring
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1)-2)
    child1 = parent1[:crossover_point] + [i for i in parent2 if i not in parent1[:crossover_point]] + parent1[crossover_point:]
    child2 = parent2[:crossover_point] + [i for i in parent1 if i not in parent2[:crossover_point]] + parent2[crossover_point:]
    return child1, child2

# Perform the genetic algorithm
def genetic_algorithm():
    population = create_population(POPULATION_SIZE)
    for generation in range(NUM_GENERATIONS):
        new_population = []
        for i in range(POPULATION_SIZE):
            parent1, parent2 = select_parents(population)
            child1, child2 = crossover(parent1, parent2)
            if random.uniform(0, 1) < MUTATION_RATE:
                child1 = mutate(child1)
            if random.uniform(0, 1) < MUTATION_RATE:
                child2 = mutate(child2)
            new_population.append(child1)
            new_population.append(child2)
        population = new_population
        best_chromosome = max(population, key=calculate_fitness)
        print("Generation {}: Best chromosome = {}, Fitness = {:.4f}".format(generation+1, best_chromosome, calculate_fitness(best_chromosome)))
    return best_chromosome

# Run the genetic algorithm and print the best solution
best_solution = genetic_algorithm()
best_distance = 1/calculate_fitness(best_solution)
print("Best solution: {}, Distance: {:.4f}".format(best_solution, best_distance))
```

Task 06

```python
import random

# Define the genes to be used in the population
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
QRSTUVWXYZ 1234567890, .-;:_!"#%&/()=?@${[]}'''

# Define the target string to be generated
TARGET = "Artificial Intelligence Lab"

# Define the size of the population
POPULATION_SIZE = 70

# Define the maximum number of generations to run
MAX_GENERATIONS = 1000

# Define the mutation rate
MUTATION_RATE = 0.1

# Define a function to generate a random chromosome
def generate_chromosome():
    return [random.choice(GENES) for _ in range(len(TARGET))]

# Define a function to calculate the fitness score of a chromosome
def calculate_fitness(chromosome):
    return sum([1 for i in range(len(TARGET)) if chromosome[i] != TARGET[i]])

# Define a function to select parents for crossover
def selection(population):
    return random.choices(population, weights=[1/fitness for fitness in [calculate_fitness(chromosome) for chromosome in population]], k=2)

# Define a function to perform crossover
def crossover(parent1, parent2):
    crossover_point = random.randint(0, len(TARGET) - 1)
    child1 = parent1[:crossover_point] + parent2[crossover_point:]
    child2 = parent2[:crossover_point] + parent1[crossover_point:]
    return child1, child2
```

```python
    return mutated_chromosome

# Define the main function to run the genetic algorithm
def run_genetic_algorithm():
    # Generate the initial population
    population = [generate_chromosome() for _ in range(POPULATION_SIZE)]

    # Iterate over generations
    for generation in range(MAX_GENERATIONS):
        # Evaluate the fitness of each chromosome in the population
        fitness_scores = [calculate_fitness(chromosome) for chromosome in population]

        # Check if we've reached the target
        if 0 in fitness_scores:
            index = fitness_scores.index(0)
            return ''.join(population[index]), generation

        # Select parents for crossover
        parent1, parent2 = selection(population)

        # Perform crossover to generate two children
        child1, child2 = crossover(parent1, parent2)

        # Perform mutation on the children
        mutated_child1 = mutation(child1)
        mutated_child2 = mutation(child2)

        # Add the children to the population
        population.append(mutated_child1)
        population.append(mutated_child2)

        # Remove the two least fit chromosomes from the population
        least_fit_index = fitness_scores.index(max(fitness_scores))
        del population[least_fit_index]
        del fitness_scores[least_fit_index]
        second_least_fit_index = fitness_scores.index(max(fitness_scores))
        del population[second_least_fit_index]

    # Return the best chromosome and the number of generations it took to reach it
    index = fitness_scores.index(min(fitness_scores))
    return ''.join(population[index]), MAX_GENERATIONS
```