Project Work Of Artificial Intelligence
Tribhuvan University

Amrit Science Campus

Thamel, Kathmandu

Submitted By: Arjun Mijar
Faculty: CSIT
Roll No.: 18
Section: A

Submitted To: Abhimanu Yadav

Internal Examiner:
Signature: ________________

External Examiner
Signature: ________________

Name: Arjun Mijar Roll No. 160

INDEX

S.N	LAB Questions	Date	Signature
1.	Write a Program to Implement Breadth-First Search using Python.	2081/05/26	
2.	Write a Program to Implement Depth First Search using Python.	2081/05/26	
3.	Write a Program to Implement the Tic-Tac-Toe game using Python.	2081/05/26	
4.	Write a Program to Implement the Water-Jug problem using Python.	2081/05/26	
5.	Write a Program to Implement a Travelling Salesman Problem using Python.	2081/05/26	
6.	Write a Program to Implement the Tower of Hanoi using Python.	2081/05/26	
7.	Write a Program to Implement the Monkey Banana Problem using Python.	2081/05/27	
8.	Write a Program to Implement the N-Queens Problem using Python.	2081/05/27	
9.	Write a Program to Implement the Naïve Bayes algorithm using Python.	2081/05/27	
10.	Write a Program to Implement a Back-propagation Algorithm using Python	2081/05/28	
11.	Write a Program to Implement a Genetics algorithm using Python.	2081/05/28	
12.	Write a program to implement the A* Search Algorithm.	2081/05/28	
13.	Write a program to implement a greedy search algorithm.	2081/05/29	
14.	Write a program to implement the uniform cost search Algorithm.	2081/05/29	

1. Write a Program to Implement Breadth-First Search using Python.

Source Code:

```
from collections import defaultdict

class Graph:
    # Constructor
    def __init__(self):
        self.graph = defaultdict(list)

    # Function to add an edge to the graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to perform BFS
    def BFS(self, s):
        # Mark all vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:
            # Dequeue a vertex from the queue and print it
            s = queue.pop(0)
            print(s, end=" ")

            # Get all adjacent vertices of the dequeued vertex s
            # If an adjacent vertex has not been visited, mark it visited and enqueue it
            for i in self.graph[s]:
                if not visited[i]:
                    queue.append(i)
                    visited[i] = True

# Create a graph and add edges
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is Breadth-First Traversal (starting from vertex 2):")
g.BFS(2)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\breadthFirstSearch.py
Following is Breadth-First Traversal (starting from vertex 2):
2 0 3 1
```

2. Write a Program to Implement Depth First Search using Python.

Source Code:

```
from collections import defaultdict

# This class represents a directed graph using adjacency list representation
class Graph:
    # Constructor
    def __init__(self):
        # Default dictionary to store the graph
        self.graph = defaultdict(list)

    # Function to add an edge to the graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):
        # Mark the current node as visited and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses recursive DFSUtil()
    def DFS(self, v):
        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code
g = Graph()
g.addEdge(0, 1)
```

```
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2):")
g.DFS(2)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\depthFirstSearch.py
Following is DFS from (starting from vertex 2):
2 0 1 3
```

3. Write a Program to Implement Tic-Tac-Toe game using Python.

Source Code:

```
# Tic-Tac-Toe Program using random numbers in Python

# Importing all necessary libraries
import numpy as np
import random
from time import sleep

# Creates an empty board
def create_board():
    return np.array([[0, 0, 0],
                     [0, 0, 0],
                     [0, 0, 0]])

# Check for empty places on the board
def possibilities(board):
    l = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == 0:
                l.append((i, j))
    return l

# Select a random place for the player
def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
```

```
    board[current_loc] = player
    return board

# Checks whether the player has three of their marks in a horizontal row
def row_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[x, y] != player:
                win = False
                break
        if win:
            return win
    return False

# Checks whether the player has three of their marks in a vertical row
def col_win(board, player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                break
        if win:
            return win
    return False

# Checks whether the player has three of their marks in a diagonal row
def diag_win(board, player):
    # Check for first diagonal
    win = True
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
            break
    if win:
        return True

    # Check for second diagonal
    win = True
    for x in range(len(board)):
        if board[x, len(board) - 1 - x] != player:
            win = False
            break
    return win

# Evaluates whether there is a winner or a tie
def evaluate(board):
    winner = 0
    for player in [1, 2]:
        if row_win(board, player) or col_win(board, player) or diag_win(board, player):
```

```
            winner = player
            break
    if np.all(board != 0) and winner == 0:
        winner = -1  # Tie
    return winner

# Main function to start the game
def play_game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)

    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print(f"Board after {counter} move:")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return winner

# Driver Code
print("Winner is: " + str(play_game()))
```

Output:

```
PS D:\Arjun Mijar (160)> py .\ticTacToeGame.py
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move:
[[0 1 0]
 [0 0 0]
 [0 0 0]]
Board after 2 move:
[[0 1 0]
 [2 0 0]
 [0 0 0]]
Board after 3 move:
[[0 1 0]
 [2 1 0]
 [0 0 0]]
Board after 4 move:
[[0 1 0]
 [2 1 0]
 [2 0 0]]
Board after 5 move:
[[0 1 0]
 [2 1 0]
 [2 1 0]]
Winner is: 1
```

4. Write a Program to Implement the Water-Jug problem using Python.

Source Code:

```
from collections import defaultdict

# Jug capacities and the target amount
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with default value as False to track visited states
visited = defaultdict(lambda: False)

# Recursive function to solve the water jug problem
def waterJugSolver(amt1, amt2):
    # If we reach the goal, print the solution and return True
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # If this state has already been visited, skip to avoid infinite recursion
    if visited[(amt1, amt2)] == False:
        # Print the current state
        print(amt1, amt2)

        # Mark the current state as visited
        visited[(amt1, amt2)] = True

        # Explore all possible moves:
        return (waterJugSolver(0, amt2) or          # Empty jug1
                waterJugSolver(amt1, 0) or          # Empty jug2
                waterJugSolver(jug1, amt2) or       # Fill jug1
                waterJugSolver(amt1, jug2) or       # Fill jug2
                waterJugSolver(amt1 + min(amt2, jug1 - amt1), amt2 - min(amt2, jug1 - amt1)) or   #
Pour jug2 into jug1
                waterJugSolver(amt1 - min(amt1, jug2 - amt2), amt2 + min(amt1, jug2 - amt2)))     #
Pour jug1 into jug2

    # Return False if no solution is found
    else:
        return False

# Driver code
print("Steps: ")
waterJugSolver(0, 0)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\waterJugProblem.py
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

5. Write a Program to Implement a Travelling Salesman Problem using Python

Source Code:

```
from sys import maxsize
from itertools import permutations

# Number of vertices in the graph (cities)
V = 4

# Implementation of the Travelling Salesman Problem
def travellingSalesmanProblem(graph, s):
    # Store all vertices except the source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # Store the minimum weight of the Hamiltonian Cycle
    min_path = maxsize

    # Generate all permutations of the vertices (except the start vertex)
    next_permutation = permutations(vertex)

    # Check each permutation
    for i in next_permutation:
        # Store current path weight (cost)
        current_pathweight = 0

        # Compute the current path weight
        k = s
        for j in i:
```

```
            current_pathweight += graph[k][j]
            k = j

        # Add the cost to return to the starting point
        current_pathweight += graph[k][s]

        # Update the minimum path cost
        min_path = min(min_path, current_pathweight)

    return min_path

# Driver Code
if __name__ == "__main__":
    # Matrix representation of the graph
    graph = [[0, 10, 15, 20],
             [10, 0, 35, 25],
             [15, 35, 0, 30],
             [20, 25, 30, 0]]

    # Starting vertex (source city)
    s = 0

    # Output the minimum path cost
    print("Minimum path cost:", travellingSalesmanProblem(graph, s))
```

Output:

```
PS D:\Arjun Mijar (160)> py .\travelSalesManProblem.py
Minimum path cost: 80
```

6. Write a Program to Implement the Tower of Hanoi using Python.

Source Code:

```
# Recursive Python function to solve Tower of Hanoi
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 1:
        print("Move disk 1 from rod", from_rod, "to rod", to_rod)
        return

    # Move n-1 disks from the source to auxiliary rod
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)

    # Move nth disk from source to destination rod
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)

    # Move the n-1 disks from auxiliary rod to destination rod
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

# Driver code
n = 3  # Number of disks
TowerOfHanoi(n, 'A', 'C', 'B')  # A, C, B are the names of the rods
```

Output:

```
PS D:\Arjun Mijar (160)> py .\towerOfHonai.py
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

7. Write a Program to Implement the Monkey Banana Problem using Python.

Source Code:

```
from poodle import Object, schedule
from typing import Set

class Position(Object):
    def __str__(self):
        if not hasattr(self, "locname"):
            return "unknown"
        return self.locname

class HasHeight(Object):
    height: int

class HasPosition(Object):
    at: Position

class Monkey(HasHeight, HasPosition):
    pass

class PalmTree(HasHeight, HasPosition):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.height = 2

class Box(HasHeight, HasPosition):
    pass

class Banana(HasHeight, HasPosition):
    owner: Monkey
    attached: PalmTree

class World(Object):
    locations: Set[Position]

# Create locations
p1 = Position()
p1.locname = "Position A"

p2 = Position()
```

```
p2.locname = "Position B"

p3 = Position()
p3.locname = "Position C"

# Create world and add positions
w = World()
w.locations = set()  # Initialize the set
w.locations.add(p1)
w.locations.add(p2)
w.locations.add(p3)

# Create objects
m = Monkey()
m.height = 0  # Ground level
m.at = p1

box = Box()
box.height = 2
box.at = p2

p = PalmTree()
p.at = p3

b = Banana()
b.attached = p

# Define functions for monkey actions
def go(monkey: Monkey, where: Position):
    assert where in w.locations
    assert monkey.height < 1, "Monkey can only move while on the ground"
    monkey.at = where
    return f"Monkey moved to {where}"

def push(monkey: Monkey, box: Box, where: Position):
    assert monkey.at == box.at
    assert where in w.locations
    assert monkey.height < 1, "Monkey can only move the box while on the ground"
    monkey.at = where
    box.at = where
    return f"Monkey moved box to {where}"

def climb_up(monkey: Monkey, box: Box):
    assert monkey.at == box.at
```

```
    monkey.height += box.height
    return "Monkey climbs the box"

def grasp(monkey: Monkey, banana: Banana):
    assert monkey.height == banana.height
    assert monkey.at == banana.at
    banana.owner = monkey
    return "Monkey takes the banana"

def infer_owner_at(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree
    banana.at = palmtree.at
    return "Remembered that if the banana is on the palm tree, its location is where the palm tree is"

def infer_banana_height(palmtree: PalmTree, banana: Banana):
    assert banana.attached == palmtree
    banana.height = palmtree.height
    return "Remembered that if the banana is on the tree, its height equals the tree's height"

# Plan execution
print('\n'.join(x() for x in schedule(
    [go, push, climb_up, grasp, infer_banana_height, infer_owner_at],
    [w, p1, p2, p3, m, box, p, b],
    goal=lambda: b.owner == m
)))
```

Output:

```
PS D:\Arjun Mijar (160)> py .\monkeyBananaProblem.py

Monkey moved to Position B
Remembered that if banana is on the tree,
its height equals tree's height Remembered that if banana is on palm tree,
its location is where palm tree is Monkey moved box to Position C
Monkey climbs the box Monkey takes the banana
```

8. Write a Program to Implement the N-Queens Problem using Python.

Source Code:

```
# Python program to solve N Queen Problem using backtracking

# Global variable for the size of the board
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=' ')
        print()

def isSafe(board, row, col):
    # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on left side
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solveNQUtil(board, col):
    # Base case: If all queens are placed, return true
    if col >= N:
        return True

    # Consider this column and try placing this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # Recur to place rest of the queens
            if solveNQUtil(board, col + 1):
                return True

            # If placing queen in board[i][col] doesn't lead to a solution, then backtrack
            board[i][col] = 0
```

```
    # If the queen cannot be placed in any row in this column, return false
    return False

def solveNQ():
    # Initialize the board
    board = [[0 for _ in range(N)] for _ in range(N)]

    if not solveNQUtil(board, 0):
        print("Solution does not exist")
        return False

    printSolution(board)
    return True

# Driver program to test the above function
solveNQ()
```

Output:

```
PS D:\Arjun Mijar (160)> py .\nQueensProblem.py
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

9. Write a Program to Implement the Naïve Bayes algorithm using Python.

Source Code:

```
import numpy as np

class NaiveBayes:
    def __init__(self):
        self.class_probs = {}
        self.feature_probs = {}
        self.classes = []
        self.feature_count = 0

    def fit(self, X, y):
        """
        Fit the Naive Bayes model using training data.

        :param X: 2D numpy array of shape (n_samples, n_features) representing the feature matrix
        :param y: 1D numpy array of shape (n_samples,) representing the class labels
        """
        self.classes = np.unique(y)
        self.feature_count = X.shape[1]

        # Calculate prior probabilities
        self.class_probs = {c: np.mean(y == c) for c in self.classes}

        # Calculate likelihoods
        self.feature_probs = {}
        for c in self.classes:
            X_c = X[y == c]
            # Use Laplace smoothing
            class_feature_probs = (np.sum(X_c, axis=0) + 1) / (X_c.shape[0] + 2)
            self.feature_probs[c] = class_feature_probs

    def predict(self, X):
        """
        Predict the class labels for the provided data.

        :param X: 2D numpy array of shape (n_samples, n_features) representing the feature matrix
        :return: 1D numpy array of shape (n_samples,) representing the predicted class labels
        """
        predictions = []
        for x in X:
            class_probs = {}
            for c in self.classes:
                # Calculate log-probabilities to avoid numerical issues with small numbers
                log_prob = np.log(self.class_probs[c])
                log_prob += np.sum(x * np.log(self.feature_probs[c]) + (1 - x) * np.log(1 -
self.feature_probs[c]))
                class_probs[c] = log_prob
            # Predict the class with the highest probability
```

```
            predicted_class = max(class_probs, key=class_probs.get)
            predictions.append(predicted_class)
        return np.array(predictions)

# Example usage
if __name__ == "__main__":
    # Sample data: 4 samples, 2 features
    X_train = np.array([
        [1, 0],
        [1, 1],
        [0, 1],
        [0, 0]
    ])
    y_train = np.array([1, 1, 0, 0])  # Binary classification

    # Create and train the Naive Bayes classifier
    nb = NaiveBayes()
    nb.fit(X_train, y_train)

    # Test data
    X_test = np.array([
        [1, 0],
        [0, 1]
    ])

    # Predict the class labels
    predictions = nb.predict(X_test)
    print("Predictions:", predictions)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\naiveBayes.py
Predictions: [1 0]
```

Name: Arjun Mijar Roll No. 160

10. Write a Program to Implement a Back-propagation Algorithm using Python

Source Code:

```
import numpy as np

# Define activation functions and their derivatives
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define the neural network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)

        self.bias_hidden = np.random.rand(self.hidden_size)
        self.bias_output = np.random.rand(self.output_size)

    def forward(self, X):
        # Forward pass
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output) +
self.bias_output
        self.output = sigmoid(self.output_layer_input)
        return self.output

    def backward(self, X, y, learning_rate):
        # Backward pass
        output_error = y - self.output
        output_delta = output_error * sigmoid_derivative(self.output)

        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden_layer_output)

        # Update weights and biases
        self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
        self.weights_hidden_output += np.dot(self.hidden_layer_output.T, output_delta) * learning_rate

        self.bias_hidden += np.sum(hidden_delta, axis=0) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0) * learning_rate

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            output = self.forward(X)
```

```
            self.backward(X, y, learning_rate)
            if (epoch + 1) % 1000 == 0:
                loss = np.mean(np.square(y - output))
                print(f'Epoch {epoch + 1}, Loss: {loss}')

# Example usage
if __name__ == "__main__":
    # Example data: XOR problem
    X_train = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    y_train = np.array([
        [0],
        [1],
        [1],
        [0]
    ])

    # Initialize and train the neural network
    input_size = 2
    hidden_size = 4
    output_size = 1

    nn = NeuralNetwork(input_size, hidden_size, output_size)
    nn.train(X_train, y_train, epochs=10000, learning_rate=0.1)

    # Test the neural network
    print("Predictions:")
    for x in X_train:
        print(f"Input: {x}, Prediction: {nn.forward(x)}")
```

Output:

```
PS D:\Arjun Mijar (160)> py .\backPropagation.py
Epoch 1000, Loss: 0.24761079780630108
Epoch 2000, Loss: 0.23157685457336868
Epoch 3000, Loss: 0.1722873043472424
Epoch 4000, Loss: 0.0832199777563821
Epoch 5000, Loss: 0.02748595956189996
Epoch 6000, Loss: 0.012049187995551575
Epoch 7000, Loss: 0.006865991836196638
Epoch 8000, Loss: 0.004570427392398989
Epoch 9000, Loss: 0.003341437219462063
Epoch 10000, Loss: 0.0025960529135158495
Predictions:
Input: [0 0], Prediction: [0.03587493]
Input: [0 1], Prediction: [0.94301913]
Input: [1 0], Prediction: [0.95295258]
Input: [1 1], Prediction: [0.06028717]
```

Name: Arjun Mijar Roll No. 160

11. Write a Program to Implement a Genetics algorithm using Python.

Source Code:

```
import numpy as np

# Parameters
population_size = 10
genome_length = 10
mutation_rate = 0.01
n_generations = 100

# Create initial population
def create_population(size, length):
    return np.random.randint(2, size=(size, length))

# Fitness function: count the number of 1's
def fitness(genome):
    return np.sum(genome)

# Selection function: select parents based on fitness
def select_parents(population, fitness_scores):
    probabilities = fitness_scores / np.sum(fitness_scores)
    parents_indices = np.random.choice(range(population_size), size=2, p=probabilities)
    return population[parents_indices]

# Crossover function: create offspring from parents
def crossover(parent1, parent2):
    point = np.random.randint(1, genome_length-1)
    offspring1 = np.concatenate((parent1[:point], parent2[point:]))
    offspring2 = np.concatenate((parent2[:point], parent1[point:]))
    return offspring1, offspring2

# Mutation function: flip bits in genome
def mutate(genome, rate):
    for i in range(len(genome)):
        if np.random.rand() < rate:
            genome[i] = 1 - genome[i]

# Main genetic algorithm function
def genetic_algorithm():
    population = create_population(population_size, genome_length)
    for generation in range(n_generations):
        fitness_scores = np.array([fitness(genome) for genome in population])
        parents = select_parents(population, fitness_scores)
        offspring1, offspring2 = crossover(parents[0], parents[1])
        mutate(offspring1, mutation_rate)
        mutate(offspring2, mutation_rate)

        # Replace the worst two individuals with the new offspring
        worst_indices = np.argsort(fitness_scores)[:2]
```

```
        population[worst_indices[0]] = offspring1
        population[worst_indices[1]] = offspring2

        # Print summary every 10 generations
        if generation % 10 == 0:
            best_fitness = np.max(fitness_scores)
            best_individual = population[np.argmax(fitness_scores)]
            print(f"Generation {generation}: Best Fitness = {best_fitness}")

    # Print final result
    final_fitness_scores = np.array([fitness(genome) for genome in population])
    best_fitness = np.max(final_fitness_scores)
    best_individual = population[np.argmax(final_fitness_scores)]
    print(f"Final Best Individual: {best_individual}")
    print(f"Final Best Fitness: {best_fitness}")

# Run the genetic algorithm
genetic_algorithm()
```

Output:

```
PS D:\Arjun Mijar (160)> py .\geneticAlgorithm.py
Generation 0: Best Fitness = 8
Generation 10: Best Fitness = 9
Generation 20: Best Fitness = 9
Generation 30: Best Fitness = 9
Generation 40: Best Fitness = 9
Generation 50: Best Fitness = 10
Generation 60: Best Fitness = 10
Generation 70: Best Fitness = 10
Generation 80: Best Fitness = 10
Generation 90: Best Fitness = 10
Final Best Individual: [1 1 1 1 1 1 1 1 1 1]
Final Best Fitness: 10
```

12.Write a program to implement the A* Search Algorithm.

Source Code:

```
import heapq

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to current node
        self.h = 0  # Heuristic cost from current node to end
        self.f = 0  # Total cost

    def __lt__(self, other):
        return self.f < other.f

def a_star_search(start, end, grid):
    open_list = []
    closed_list = set()

    start_node = Node(start)
    end_node = Node(end)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_list.add(current_node.position)

        if current_node.position == end_node.position:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Adjacent squares: right, down, left, up
        for neighbor in neighbors:
            node_position = (current_node.position[0] + neighbor[0], current_node.position[1] +
neighbor[1])
            if (0 <= node_position[0] < len(grid) and 0 <= node_position[1] < len(grid[0]) and
                grid[node_position[0]][node_position[1]] == 0):
                if node_position in closed_list:
                    continue

                neighbor_node = Node(node_position, current_node)
                neighbor_node.g = current_node.g + 1
                neighbor_node.h = ((end_node.position[0] - neighbor_node.position[0]) ** 2 +
                                   (end_node.position[1] - neighbor_node.position[1]) ** 2) ** 0.5
                neighbor_node.f = neighbor_node.g + neighbor_node.h
```

```
            if any(node.position == neighbor_node.position and node.f <= neighbor_node.f for node in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)

    return None  # No path found

# Example grid: 0 is walkable, 1 is blocked
grid = [[0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0]]

start = (0, 0)  # Start position
end = (4, 4)    # End position

path = a_star_search(start, end, grid)
print("Path found:", path)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\aSearchAlgorithm.py
Path found: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```

Name: Arjun Mijar Roll No. 160

13.Write a program to implement a greedy search algorithm.

Source Code:

```
import heapq

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.h = 0  # Heuristic cost

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

def greedy_best_first_search(start, end, grid):
    open_list = []
    closed_list = set()

    start_node = Node(start)
    end_node = Node(end)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        closed_list.add(current_node.position)

        if current_node.position == end_node.position:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]

        neighbors = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Adjacent squares: right, down, left, up
        for neighbor in neighbors:
            node_position = (current_node.position[0] + neighbor[0], current_node.position[1] +
neighbor[1])
            if (0 <= node_position[0] < len(grid) and 0 <= node_position[1] < len(grid[0]) and
                grid[node_position[0]][node_position[1]] == 0):
                if node_position in closed_list:
                    continue

                neighbor_node = Node(node_position, current_node)
                neighbor_node.h = ((end_node.position[0] - neighbor_node.position[0]) ** 2 +
                                   (end_node.position[1] - neighbor_node.position[1]) ** 2) ** 0.5

                if any(node.position == neighbor_node.position and node.h <= neighbor_node.h for
node in open_list):
                    continue
```

```
                heappush(open_list, neighbor_node)

    return None  # No path found

# Example grid: 0 is walkable, 1 is blocked
grid = [[0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0]]

start = (0, 0)  # Start position
end = (4, 4)    # End position

path = greedy_best_first_search(start, end, grid)
print("Path found:", path)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\greedySearchAlgorithm.py
Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

14.Write a program to implement the uniform cost search Algorithm.

```
import heapq

class Node:
    def __init__(self, position, cost=0, parent=None):
        self.position = position
        self.cost = cost
        self.parent = parent

    def __lt__(self, other):
        return self.cost < other.cost

def uniform_cost_search(start, goal, graph):
    open_list = []
    closed_list = set()

    start_node = Node(start, 0)
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
        current_position = current_node.position

        if current_position in closed_list:
            continue

        closed_list.add(current_position)

        if current_position == goal:
            path = []
            total_cost = current_node.cost
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1], total_cost

        for neighbor, cost in graph.get(current_position, {}).items():
            if neighbor in closed_list:
                continue

            neighbor_node = Node(neighbor, current_node.cost + cost, current_node)
            heapq.heappush(open_list, neighbor_node)

    return None, float('inf')  # No path found

# Example graph represented as an adjacency list with costs
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
```

```
    'D': {'B': 5, 'C': 1}
}

start = 'A'
goal = 'D'

path, cost = uniform_cost_search(start, goal, graph)
print("Path found:", path)
print("Total cost:", cost)
```

Output:

```
PS D:\Arjun Mijar (160)> py .\uniformCostSearchAlgo.py
Path found: ['A', 'B', 'C', 'D']
Total cost: 4
```