**LAB 1**

**Breadth First Search**

**AIM:** To Implement Breadth First Search using Python.

**Source Code:**

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': ['G', 'H'],

'E': ['I'],

'F': [],

'G': [],

'H': [],

'I': []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

# print '->' after each node except the last one

print(m, end='->' if m != 'I' else '\n')

for neighbour in graph[m]:

if neighbour not in visited:

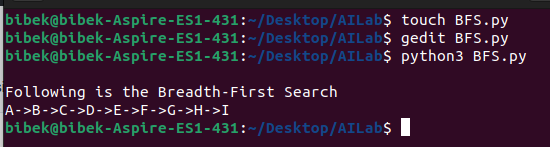
visited.append(neighbour)

queue.append(neighbour)

print("\nFollowing is the Breadth-First Search")

bfs(visited, graph, 'A')

**Output:**



**LAB 2**

**Depth First Search**

**AIM:** To Implement Depth First Search using Python.

**Source Code:**

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': ['G', 'H'],

'E': ['I'],

'F': [],

'G': [],

'H': [],

'I': []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): # function for dfs

if node not in visited:

# print '->' after each node except the last one

print(node, end='->' if node != 'F' else '\n')

visited.add(node)

for neighbour in graph[node]:

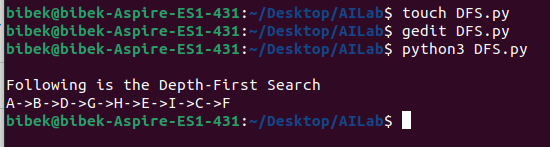
dfs(visited, graph, neighbour)

# Driver Code

print("\nFollowing is the Depth-First Search")

dfs(visited, graph, 'A')

**Output:**



**LAB 3**

**Tic-Tac-Toe**

**AIM:** To Implement Tic-Tac-Toe game using python.

**Source Code:**

board = [' ' for x in range(9)]

player = 1

''' Win Flags '''

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

# This Function Draws Game Board

def DrawBoard():

print(" %c | %c | %c " % (board[0], board[1], board[2]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[3], board[4], board[5]))

print("\_\_\_|\_\_\_|\_\_\_")

print(" %c | %c | %c " % (board[6], board[7], board[8]))

print(" | | ")

# This Function Checks position is empty or not

def CheckPosition(x):

if (board[x] == ' '):

return True

else:

return False

# This Function Checks player has won or not

def CheckWin():

global Game

# Horizontal winning condition

if (board[0] == board[1] and board[1] == board[2] and board[0] != ' '):

Game = Win

elif (board[3] == board[4] and board[4] == board[5] and board[3] != ' '):

Game = Win

elif (board[6] == board[7] and board[7] == board[8] and board[6] != ' '):

Game = Win

# Vertical Winning Condition

elif (board[0] == board[3] and board[3] == board[6] and board[0] != ' '):

Game = Win

elif (board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif (board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

# Diagonal Winning Condition

elif (board[0] == board[4] and board[4] == board[8] and board[4] != ' '):

Game = Win

elif (board[2] == board[4] and board[4] == board[6] and board[4] != ' '):

Game = Win

# Match Tie or Draw Condition

elif (board[0] != ' ' and

board[1] != ' ' and

board[2] != ' ' and

board[3] != ' ' and

board[4] != ' ' and

board[5] != ' ' and

board[6] != ' ' and

board[7] != ' ' and

board[8] != ' '):

Game = Draw

else:

Game = Running

print("---- Tic-Tac-Toe ----\n\n")

print("Player 1 [X] --- Player 2 [O]\n\n\n")

while (Game == Running):

DrawBoard()

if (player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(

input("Enter the position between [0-8] where you want to mark: "))

if (CheckPosition(choice)):

board[choice] = Mark

player += 1

CheckWin()

DrawBoard()

if (Game == Draw):

print("Game is tied!🏅🏆")

elif (Game == Win):

player -= 1

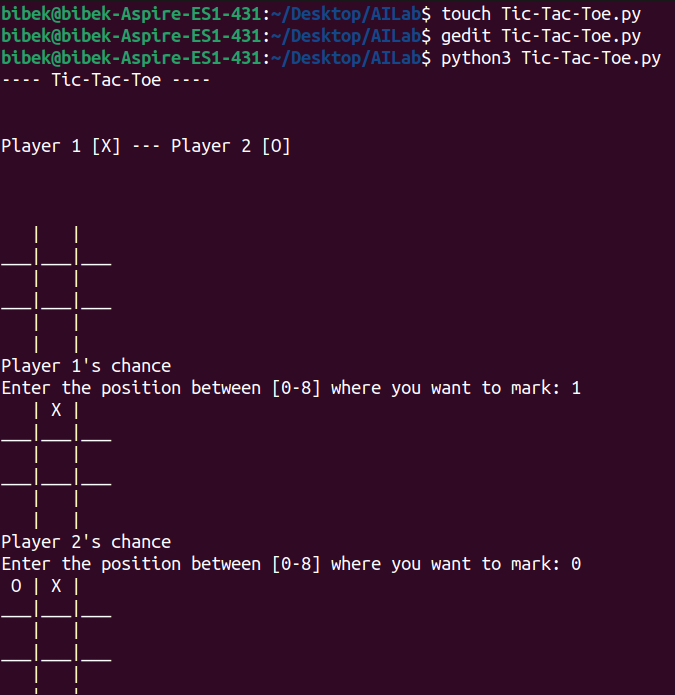
if (player % 2 != 0):

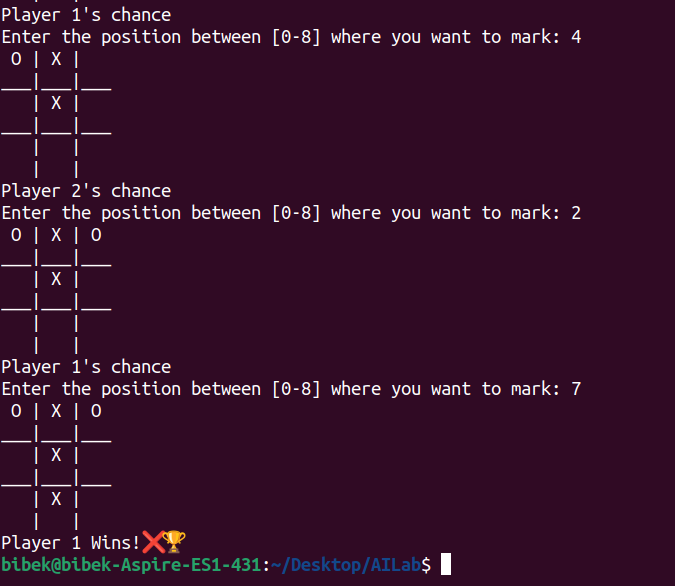
print("Player 1 Wins!❌🏆")

else:

print("Player 2 Wins!⭕🏆")

**Output:**





**LAB 4**

**Water-Jug Problem**

**AIM:** To Implement Water-Jug problem using python.

**Source Code:**

from collections import defaultdict

# Max capacities of jugs and the target amount

jug1, jug2, aim = 4, 3, 2

# Initialize dictionary to track visited states

visited = defaultdict(lambda: False)

# Recursive function to print the steps to the solution

def waterJugSolver(amt1, amt2):

if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

print(amt1, amt2)

return True

if not visited[(amt1, amt2)]:

visited[(amt1, amt2)] = True

print(amt1, amt2)

return (waterJugSolver(0, amt2) or # Empty jug 1

waterJugSolver(amt1, 0) or # Empty jug 2

waterJugSolver(jug1, amt2) or # Fill jug 1

waterJugSolver(amt1, jug2) or # Fill jug 2

waterJugSolver(amt1 + min(amt2, jug1 - amt1), amt2 - min(amt2, jug1 - amt1)) or # Pour jug 2 into jug 1

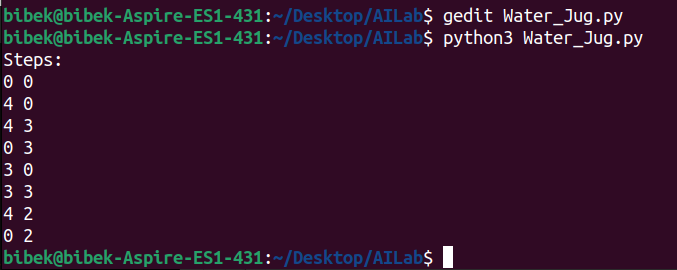
waterJugSolver(amt1 - min(amt1, jug2 - amt2), amt2 + min(amt1, jug2 - amt2))) # Pour jug 1 into jug 2

return False

print("Steps:")

waterJugSolver(0, 0)

**Output:**

****

**LAB 5**

**Travelling Salesman Problem**

**AIM:** To Implement Travelling Salesman problem using python.

**Source Code:**

from sys import maxsize

from itertools import permutations

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

V = 4

# Function to implement the Travelling Salesman Problem

def travellingSalesmanProblem(graph, start):

# Store all vertices except the start vertex

vertex = [i for i in range(V) if i != start]

# Initialize minimum path weight as infinite

min\_path = maxsize

# Generate all permutations of the vertex list

next\_permutations = permutations(vertex)

# Calculate the path weight for each permutation

for perm in next\_permutations:

current\_pathweight = 0

k = start

# Compute the total path weight for this permutation

for j in perm:

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][start] # Return to start

# Update minimum path if the current one is smaller

min\_path = min(min\_path, current\_pathweight)

return min\_path

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

# Graph represented as an adjacency matrix

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

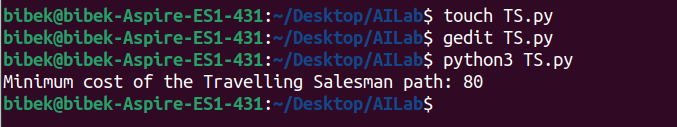
[20, 25, 30, 0]

]

start = 0

print(f"Minimum cost of the Travelling Salesman path: {travellingSalesmanProblem(graph, start)}")

**Output:**

****

**LAB 6**

**Tower of Hanoi**

**AIM:** To Implement Tower of Hanoi using python.

**Source Code:**

def toh(n, source, destination, temp):

if n == 1:

print(f"Move disk 1 from {source} to {destination}")

return

toh(n - 1, source, temp, destination)

print(f"Move disk {n} from {source} to {destination}")

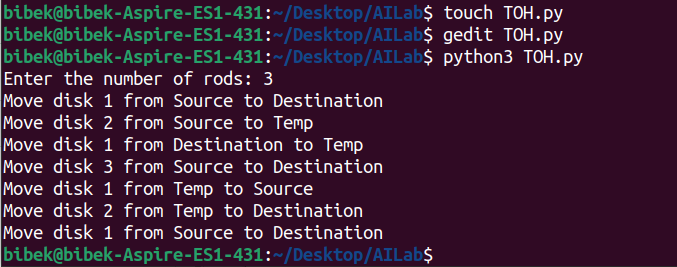
toh(n-1, temp, destination, source)

# Enter the number of rods

n = int(input("Enter the number of rods: "))

toh(n, 'Source', 'Destination', 'Temp')

**Output:**

****

**LAB 7**

**Monkey Banana Problem**

**AIM:** To Implement Monkey Banana Problem using python.

**Source Code:**

from typing import Set, Callable, List, Any

class Position:

def \_\_init\_\_(self, locname=None):

self.locname = locname

def \_\_str\_\_(self):

return self.locname if self.locname else "unknown"

class HasHeight:

def \_\_init\_\_(self, height=0):

self.height = height

class HasPosition:

def \_\_init\_\_(self, at=None):

self.at = at

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):

def \_\_init\_\_(self, owner: Monkey = None, attached: PalmTree = None):

super().\_\_init\_\_()

self.owner = owner

self.attached = attached

class World:

def \_\_init\_\_(self):

self.locations: Set[Position] = set()

# Create the world and positions

p1 = Position("Position A")

p2 = Position("Position B")

p3 = Position("Position C")

w = World()

w.locations = set([p1, p2, p3])

# Create the monkey, box, palm tree, and banana

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 the functions for actions and reasoning

def go(monkey: Monkey, where: Position):

assert where in w.locations, "Invalid location"

assert monkey.height == 0, "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, "Monkey and box must be at the same location"

assert where in w.locations, "Invalid location"

assert monkey.height == 0, "Monkey can only push 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 must be at the same location as the box"

monkey.height += box.height

return "Monkey climbs the box"

def grasp(monkey: Monkey, banana: Banana):

assert monkey.height == banana.height, "Monkey must be at the same height as the banana"

assert monkey.at == banana.at, "Monkey must be at the same location as the banana"

banana.owner = monkey

return "Monkey takes the banana"

def infer\_owner\_at(palmtree: PalmTree, banana: Banana):

assert banana.attached == palmtree, "Banana must be attached to the palm tree"

banana.at = palmtree.at

return "Banana's location is inferred to be the palm tree's location"

def infer\_banana\_height(palmtree: PalmTree, banana: Banana):

assert banana.attached == palmtree, "Banana must be attached to the palm tree"

banana.height = palmtree.height

return "Banana's height is inferred to be the tree's height"

# Define the schedule function

def schedule(actions: List[Callable], args: List[Any], goal: Callable):

results = []

for action in actions:

try:

result = action() # Call the action without args, we'll provide them later in lambdas

results.append(result)

if goal():

break

except AssertionError as e:

results.append(f"Action {action.\_\_name\_\_} failed: {str(e)}")

return results

# Define the schedule of actions and reasoning

print('\n'.join(schedule(

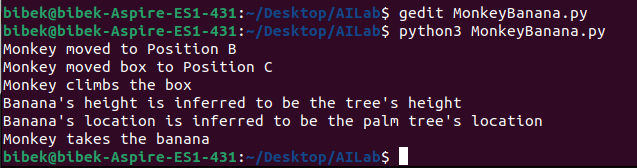
[lambda: go(m, p2), lambda: push(m, box, p3), lambda: climb\_up(m, box), lambda: infer\_banana\_height(p, b), lambda: infer\_owner\_at(p, b), lambda: grasp(m, b)],

[w, p1, p2, p3, m, box, p, b],

goal=lambda: b.owner == m

)))

**Output:**

****

**LAB 8**

**N-Queens Problem**

**AIM:** To Implement N-Queens Problem using python.

**Source Code:**

'''

Program to implement for N-Queen problem

'''

'''

logic where the queen must not be placed:

1) row, col+-

2) row--, col--

3) row++, col--

'''

def is\_safe(board, row, col, n):

for c in range(col, -1, -1): # check for the same row in left side of the board

if board[row][c] == 'Q':

return False

i = row

j = col

while i >= 0 and j >= 0: # check for the left diagonal in the upper side of the board

if board[i][j] == 'Q':

return False

i -= 1

j -= 1

i = row

j = col

while i < n and j >= 0: # check for the left diagonal in the bottom side of the board

if board[i][j] == 'Q':

return False

i += 1

j -= 1

return True

def nqueen(board, col, n):

if col >= n:

return True

for i in range(n):

if is\_safe(board, i, col, n):

board[i][col] = 'Q'

if nqueen(board, col+1, n):

return True

board[i][col] = 0

return False

n = int(input("Enter the number of queens: "))

board = [[0 for j in range(n)] for i in range(n)]

if nqueen(board, 0, n) == True:

for i in range(n):

for j in range(n):

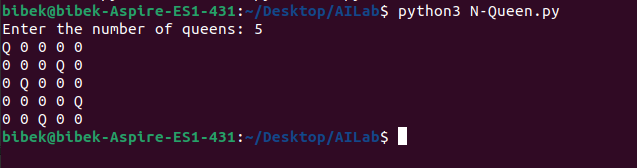
print(board[i][j], end=' ')

print()

else:

print("Not possible")

**Output:**

****

**LAB 9**

**Naive Bayes Algorithm**

**AIM:** To Implement Naive Bayes Algorithm using python.

**Source Code:**

# Import necessary libraries

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

from sklearn.datasets import load\_iris

# Load the Iris dataset

iris = load\_iris()

X = iris.data # Features

y = iris.target # Labels

# Split the dataset into training and testing sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a Gaussian Naive Bayes classifier

model = GaussianNB()

# Train the model

model.fit(X\_train, y\_train)

# Predict the labels for the test set

y\_pred = model.predict(X\_test)

# Evaluate the model

accuracy = accuracy\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

class\_report = classification\_report(y\_test, y\_pred)

# Output the results

print(f"Accuracy: {accuracy \* 100:.2f}%")

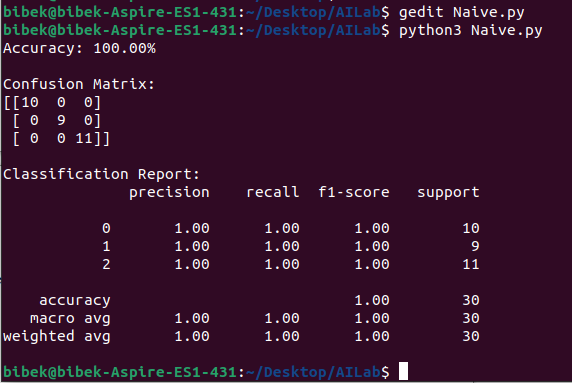
print("\nConfusion Matrix:")

print(conf\_matrix)

print("\nClassification Report:")

print(class\_report)

**Output:**

****

**LAB 10**

**Back Propagation Algorithm**

**AIM:** To Implement Back Propagation Algorithm using python.

**Source Code:**

import numpy as np

# Sigmoid activation function

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Derivative of the sigmoid function

def sigmoid\_derivative(x):

return x \* (1 - x)

# Input data (4 samples, 2 features each)

X = np.array([[0, 0],

[0, 1],

[1, 0],

[1, 1]])

# Output labels (XOR truth table)

y = np.array([[0],

[1],

[1],

[0]])

# Seed for reproducibility

np.random.seed(42)

# Initialize weights randomly with mean 0

input\_layer\_neurons = X.shape[1] # Number of input neurons

hidden\_layer\_neurons = 2 # Number of hidden neurons

output\_neurons = 1 # Number of output neurons

# Weights for the input to the hidden layer

W1 = np.random.uniform(size=(input\_layer\_neurons, hidden\_layer\_neurons))

# Weights for the hidden to the output layer

W2 = np.random.uniform(size=(hidden\_layer\_neurons, output\_neurons))

# Learning rate

learning\_rate = 0.1

# Number of iterations for training

epochs = 10000

# Training process

for epoch in range(epochs):

# Forward Propagation

hidden\_layer\_input = np.dot(X, W1) # Input to hidden layer

hidden\_layer\_output = sigmoid(hidden\_layer\_input) # Output from hidden layer

output\_layer\_input = np.dot(hidden\_layer\_output, W2) # Input to output layer

predicted\_output = sigmoid(output\_layer\_input) # Output from output layer

# Calculate the error (difference between actual and predicted)

error = y - predicted\_output

# Backpropagation

d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output) # Derivative of output

error\_hidden\_layer = d\_predicted\_output.dot(W2.T) # Error propagated to hidden layer

d\_hidden\_layer\_output = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output) # Derivative of hidden layer output

# Update the weights

W2 += hidden\_layer\_output.T.dot(d\_predicted\_output) \* learning\_rate

W1 += X.T.dot(d\_hidden\_layer\_output) \* learning\_rate

# Print the error every 1000 epochs

if epoch % 1000 == 0:

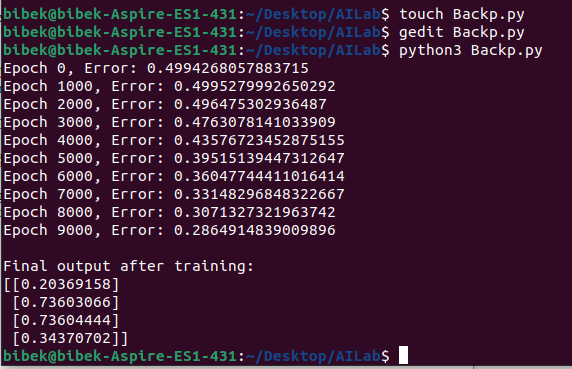
print(f"Epoch {epoch}, Error: {np.mean(np.abs(error))}")

# Final output after training

print("\nFinal output after training:")

print(predicted\_output)

**Output:**

****

**LAB 11**

**Genetics Algorithm**

**AIM:** To Implement Genetics Algorithm using python.

**Source Code:**

import random

# Define the target string and population parameters

TARGET = "1101010101" # Target binary string

POPULATION\_SIZE = 100 # Size of the population

MUTATION\_RATE = 0.01 # Mutation rate

GENERATIONS = 1000 # Maximum number of generations

# Individual class representing a binary string

class Individual:

def \_\_init\_\_(self, chromosome=None):

if chromosome:

self.chromosome = chromosome

else:

# Randomly initialize a chromosome (binary string)

self.chromosome = ''.join(random.choice('01') for \_ in range(len(TARGET)))

self.fitness = self.calculate\_fitness()

def calculate\_fitness(self):

# Fitness is the number of matching bits with the target string

return sum(1 for i, j in zip(self.chromosome, TARGET) if i == j)

# Genetic Algorithm functions

def selection(population):

# Select two individuals from the population based on fitness (roulette wheel selection)

return random.choices(population, weights=[ind.fitness for ind in population], k=2)

def crossover(parent1, parent2):

# Single-point crossover: combine the chromosomes of the two parents

crossover\_point = random.randint(0, len(TARGET) - 1)

child\_chromosome = parent1.chromosome[:crossover\_point] + parent2.chromosome[crossover\_point:]

return Individual(chromosome=child\_chromosome)

def mutate(individual):

# Randomly mutate the individual's chromosome based on the mutation rate

mutated\_chromosome = ''.join(

gene if random.random() > MUTATION\_RATE else random.choice('01')

for gene in individual.chromosome

)

return Individual(chromosome=mutated\_chromosome)

# Main Genetic Algorithm function

def genetic\_algorithm():

# Initialize the population with random individuals

population = [Individual() for \_ in range(POPULATION\_SIZE)]

for generation in range(GENERATIONS):

# Sort population by fitness

population = sorted(population, key=lambda x: x.fitness, reverse=True)

# If the fittest individual matches the target, stop

if population[0].fitness == len(TARGET):

print(f"Target reached in generation {generation}!")

print(f"Best individual: {population[0].chromosome}")

break

# Print best individual every 100 generations

if generation % 100 == 0:

print(f"Generation {generation}, Best fitness: {population[0].fitness}")

# Create a new population using selection, crossover, and mutation

new\_population = []

# Elitism: carry forward the best individual

new\_population.append(population[0])

# Generate the rest of the population

while len(new\_population) < POPULATION\_SIZE:

# Select parents based on fitness

parent1, parent2 = selection(population)

# Perform crossover

child = crossover(parent1, parent2)

# Mutate the child

child = mutate(child)

# Add child to the new population

new\_population.append(child)

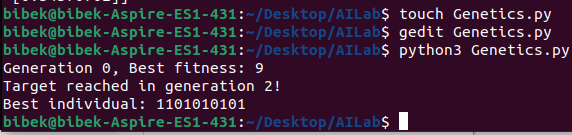
# Replace the old population with the new one

population = new\_population

# Run the Genetic Algorithm

genetic\_algorithm()

**Output:**

****

**LAB 12**

**A\* Search Algorithm**

**AIM:** To Implement A\* Search Algorithm using python.

**Source Code:**

'''

Program to implement A Star Search Algorithm

'''

# Defining the graph nodes in dict with given costs to traverse

adj\_list = {

's': [('a', 1), ('g', 10)],

'a': [('b', 2), ('c', 1)],

'b': [('d', 5)],

'c': [('d', 3), ('g', 4)],

'd': [('g', 2)],

'g': []

}

# Defining heuristic values for each nodes

heuristic = {

's': 5,

'a': 3,

'b': 4,

'c': 2,

'd': 6,

'g': 0

}

# A Star Search Algorithm

def astar\_search(adj\_list, heuristic, start\_node, goal\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

def get\_neighbors(node):

return adj\_list[node]

def h(node):

return heuristic[node]

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + h(v) < g[n] + h(n):

n = v

if n == None:

print('Path does not exist!')

return None

if n == goal\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

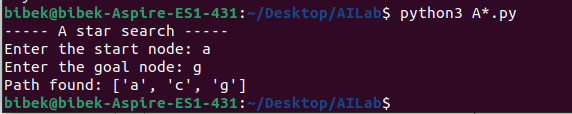
print("----- A star search -----")

start\_node = input("Enter the start node: ")

goal\_node = input("Enter the goal node: ")

astar\_search(adj\_list, heuristic, start\_node, goal\_node)

**Output:**

****

**LAB 13**

**Greedy Search Algorithm**

**AIM:** To Implement Greedy Search Algorithm using python.

**Source Code:**

import heapq

# Class to represent the graph

class Graph:

def \_\_init\_\_(self):

self.edges = {}

self.heuristics = {}

# Add an edge between two nodes

def add\_edge(self, node1, node2, cost):

if node1 not in self.edges:

self.edges[node1] = []

self.edges[node1].append((node2, cost))

# Set heuristic value for a node

def set\_heuristic(self, node, h\_value):

self.heuristics[node] = h\_value

# Greedy Best-First Search algorithm

def greedy\_search(self, start, goal):

# Priority queue to store (heuristic, node) pairs

open\_list = []

heapq.heappush(open\_list, (self.heuristics[start], start))

# To keep track of visited nodes

visited = set()

# To store the path

came\_from = {}

while open\_list:

# Get the node with the lowest heuristic value

\_, current\_node = heapq.heappop(open\_list)

# If we reach the goal, reconstruct and return the path

if current\_node == goal:

return self.reconstruct\_path(came\_from, start, goal)

visited.add(current\_node)

# Explore neighbors

for neighbor, cost in self.edges.get(current\_node, []):

if neighbor not in visited:

heapq.heappush(open\_list, (self.heuristics[neighbor], neighbor))

came\_from[neighbor] = current\_node

return None # No path found

# Function to reconstruct the path from the start to the goal

def reconstruct\_path(self, came\_from, start, goal):

path = [goal]

while goal in came\_from:

goal = came\_from[goal]

path.append(goal)

path.reverse()

return path

# Example usage:

graph = Graph()

# Add edges: graph.add\_edge('A', 'B', cost)

graph.add\_edge('A', 'B', 1)

graph.add\_edge('A', 'C', 3)

graph.add\_edge('B', 'D', 3)

graph.add\_edge('B', 'E', 1)

graph.add\_edge('C', 'F', 5)

graph.add\_edge('E', 'F', 2)

graph.add\_edge('D', 'G', 6)

graph.add\_edge('F', 'G', 2)

# Set heuristic values for each node (estimated cost to the goal)

graph.set\_heuristic('A', 6)

graph.set\_heuristic('B', 4)

graph.set\_heuristic('C', 5)

graph.set\_heuristic('D', 2)

graph.set\_heuristic('E', 2)

graph.set\_heuristic('F', 1)

graph.set\_heuristic('G', 0) # Goal node heuristic is 0

# Run Greedy Best-First Search from 'A' to 'G'

path = graph.greedy\_search('A', 'G')

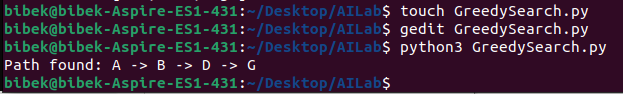
if path:

print(f"Path found: {' -> '.join(path)}")

else:

print("No path found")

**Output:**

****

**LAB 14**

**Uniform Cost Search Algorithm**

**AIM:** To Implement Uniform Cost Search Algorithm using python.

**Source Code:**

import heapq

# Class to represent the graph

class Graph:

def \_\_init\_\_(self):

self.edges = {}

# Add an edge between two nodes with a given cost

def add\_edge(self, node1, node2, cost):

if node1 not in self.edges:

self.edges[node1] = []

self.edges[node1].append((cost, node2))

# Uniform Cost Search algorithm

def uniform\_cost\_search(self, start, goal):

# Priority queue to store (cumulative\_cost, node, path)

open\_list = []

heapq.heappush(open\_list, (0, start, [start])) # (cost, node, path)

# To store the visited nodes and their costs

visited = {}

while open\_list:

# Get the node with the least cumulative cost

current\_cost, current\_node, path = heapq.heappop(open\_list)

# If we have reached the goal, return the path and the cost

if current\_node == goal:

return path, current\_cost

# If the node is not visited or found at a lower cost

if current\_node not in visited or current\_cost < visited[current\_node]:

visited[current\_node] = current\_cost

# Explore the neighbors

for cost, neighbor in self.edges.get(current\_node, []):

new\_cost = current\_cost + cost

new\_path = path + [neighbor]

heapq.heappush(open\_list, (new\_cost, neighbor, new\_path))

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

# Example usage

graph = Graph()

# Add edges: graph.add\_edge('A', 'B', cost)

graph.add\_edge('A', 'B', 1)

graph.add\_edge('A', 'C', 4)

graph.add\_edge('B', 'D', 2)

graph.add\_edge('C', 'D', 1)

graph.add\_edge('D', 'E', 5)

graph.add\_edge('B', 'E', 12)

graph.add\_edge('C', 'E', 5)

# Run Uniform Cost Search from 'A' to 'E'

path, cost = graph.uniform\_cost\_search('A', 'E')

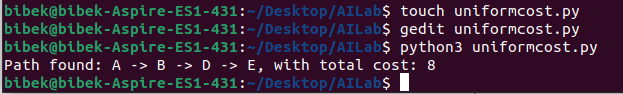
if path:

print(f"Path found: {' -> '.join(path)}, with total cost: {cost}")

else:

print("No path found")

**Output:**

****