1. Write the python program to solve 8-Puzzle problem

**AIM :** program to solve 8-Puzzle problem

**ALGORITHM :**

1.Define the initial state and goal state.

2. Create an open list to store states yet to be explored, initially containing the initial state.

3. Create a closed set to store states that have been explored.

4. Initialize the g\_score and f\_score dictionaries for each state.

5. While the open list is not empty:

* Pop the state with the lowest f\_score value from the open list.
* If the current state is the goal state, reconstruct and return the solution path.
* Add the current state to the closed set.
* Generate possible moves from the current state (swap the empty tile with neighboring tiles).
* For each possible move:
* Calculate the tentative g\_score for the new state.
* If the new state is already in the closed set and the tentative g\_score is greater or equal, skip it.
* If the new state is not in the open list or the tentative g\_score is lower:
* Update the came\_fromdictionary to store the parent state.
* Update the g\_score and f\_score for the new state.
* Add the new state to the open list.

6. If the open list becomes empty and the goal state hasn't been reached, there is no solution.

**PROGRAM :**

import heapq

goal\_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

def get\_neighbors(state):

neighbors = []

empty\_index = state.index(0)

empty\_row, empty\_col = empty\_index // 3, empty\_index % 3

for dr, dc in moves:

new\_row, new\_col = empty\_row + dr, empty\_col + dc

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

neighbor\_state = list(state)

neighbor\_index = new\_row \* 3 + new\_col

neighbor\_state[empty\_index], neighbor\_state[neighbor\_index] = neighbor\_state[neighbor\_index], neighbor\_state[empty\_index]

neighbors.append(tuple(neighbor\_state))

return neighbors

def manhattan\_distance(state):

distance = 0

for i in range(9):

if state[i] != 0:

current\_row, current\_col = i // 3, i % 3

target\_row, target\_col = (state[i] - 1) // 3, (state[i] - 1) % 3

distance += abs(current\_row - target\_row) + abs(current\_col - target\_col)

return distance

def solve\_puzzle(initial\_state):

open\_list = [(manhattan\_distance(initial\_state), initial\_state)]

closed\_set = set()

while open\_list:

current\_state = heapq.heappop(open\_list)[1]

if current\_state == goal\_state:

return current\_state

closed\_set.add(current\_state)

for neighbor in get\_neighbors(current\_state):

if neighbor not in closed\_set:

heapq.heappush(open\_list, (manhattan\_distance(neighbor), neighbor))

return None

initial\_state = (1, 0, 3, 4, 2, 5, 7, 8, 6)

solution = solve\_puzzle(initial\_state)

if solution:

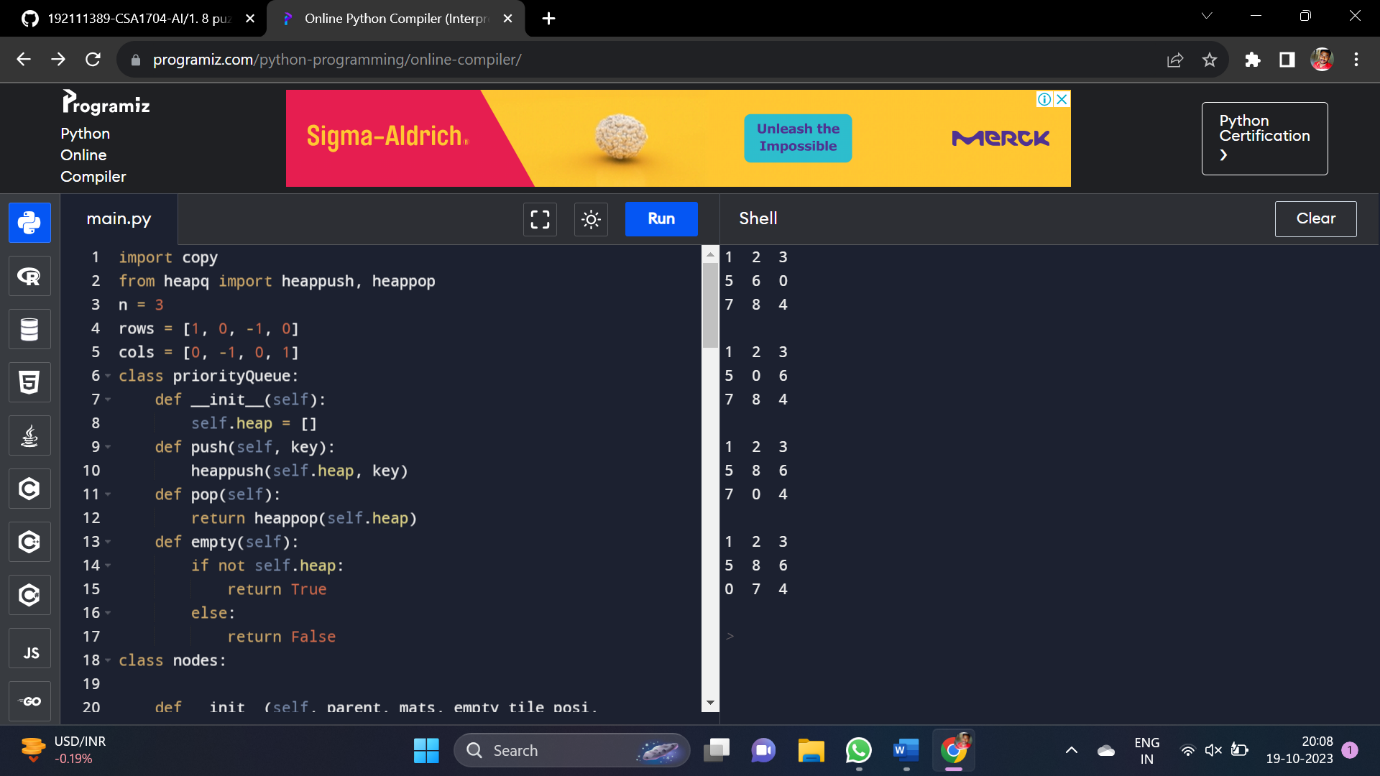
print("Solution found:")

for i in range(0, 9, 3):

print(solution[i:i+3])

else:

print("No solution found.")



2. Write the python program to solve 8-Queen problem and algorithm.

**AIM :** program to solve 8-Queen problem and algorithm

**ALGORITHM :**

1. Start from the leftmost column.
2. If all queens are placed, return true.
3. Try all rows in the current column. For each row, check if placing a queen in that row is safe. If safe, mark the cell and proceed recursively for the next column.
4. If placing the queen in the current row leads to a solution, return true.
5. If no row in the current column is safe, backtrack by unmarking the cell and return false.

**PROGRAM :**

N = 8

def print\_board(board):

for row in board:

print(" ".join(row))

print()

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

for i in range(row):

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

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

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

return False

for i, j in zip(range(row, -1, -1), range(col, N)):

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

return False

return True

def solve\_queens(board, col):

if col >= N:

return True

for i in range(N):

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

board[i][col] = 'Q'

if solve\_queens(board, col + 1):

return True

board[i][col] = '.' # Backtrack if no solution is found

return False

chessboard = [['.' for \_ in range(N)] for \_ in range(N)]

if solve\_queens(chessboard, 0):

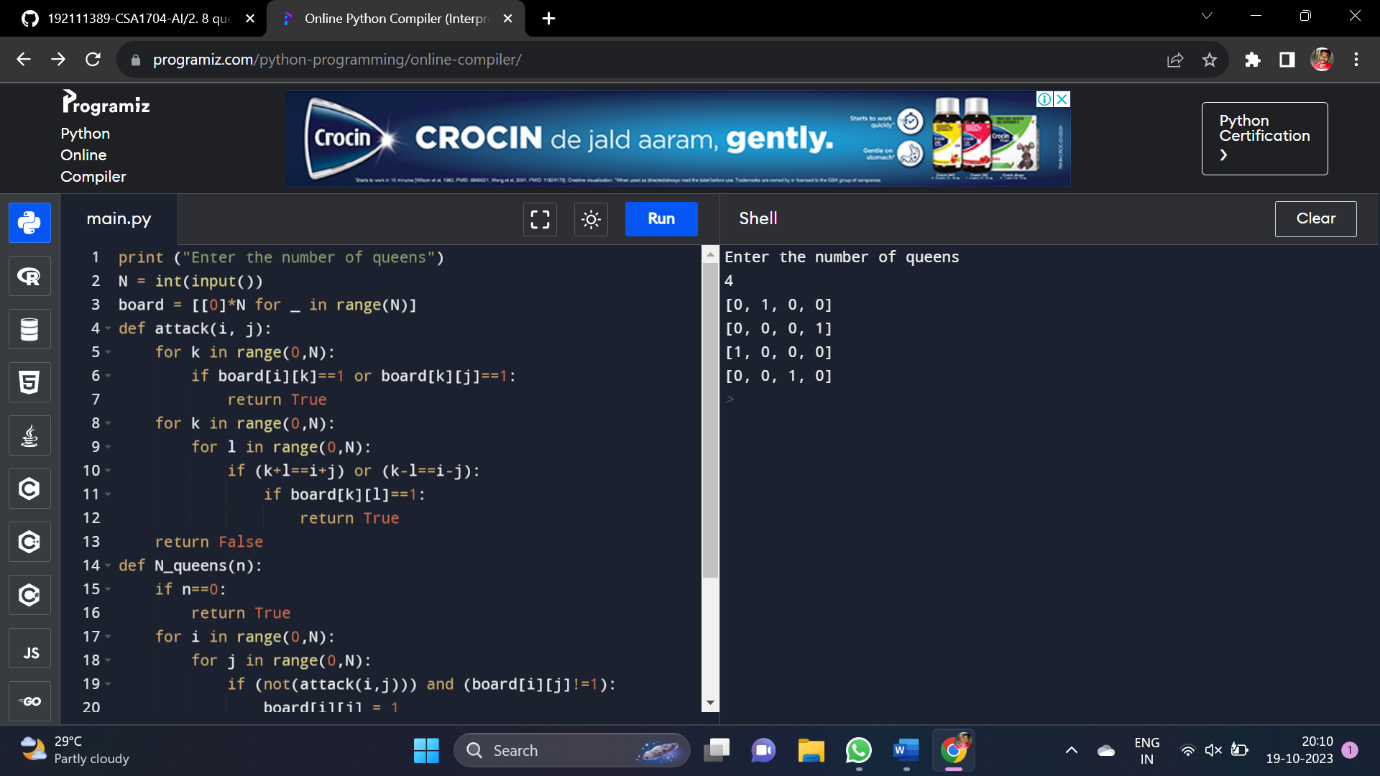
print("Solution:")

print\_board(chessboard)

else:

print("No solution found.")

**OUT PUT :**



3. Write the python program for Water Jug Problem

**AIM :** program for Water Jug Problem

**ALGORITHM :**

1. **Input:** Capacities of two jugs (x and y), and the target amount of water to be measured (target).
2. **Check Validity:** Check if the target amount is achievable using the given jug capacities. If the target amount is greater than the maximum capacity of both jugs or if the target amount is not divisible by the greatest common divisor (GCD) of the jug capacities, then no solution is possible.
3. **Initialize:** Create an empty set visited\_states to keep track of visited states and a queue queue to store states and actions.
4. **Enqueue Initial State:** Enqueue the initial state (0, 0) along with an empty list of actions into the queue.
5. **While Queue is Not Empty:**

* Dequeue the front state and associated actions from the queue.
* Check if the state has already been visited. If so, skip this iteration.
* Mark the state as visited by adding it to the visited\_states set.
* Check if either jug has the desired amount of water. If so, return the list of actions as the solution.
* Perform the following actions for each possible operation (fill, empty, pour) on both jugs:
* Fill the first jug.
* Fill the second jug.
* Empty the first jug.
* Empty the second jug.
* Pour water from the first jug to the second jug.
* Pour water from the second jug to the first jug.
* Enqueue the resulting states along with the updated list of actions

1. **No Solution Found:** If the queue becomes empty and no solution is found, return None to indicate that no solution is possible.

**PROGRAM :**

def gcd(a, b):

while b:

a, b = b, a % b

return a

def water\_jug\_problem(x, y, target):

if target > max(x, y) or target % gcd(x, y) != 0:

return None

visited\_states = set()

stack = [(0, 0)]

while stack:

current\_state = stack.pop()

if current\_state in visited\_states:

continue

visited\_states.add(current\_state)

if current\_state[0] == target or current\_state[1] == target:

return visited\_states

stack.append((x, current\_state[1]))

stack.append((current\_state[0], y))

stack.append((0, current\_state[1]))

stack.append((current\_state[0], 0))

pour\_amount = min(current\_state[0], y - current\_state[1])

stack.append((current\_state[0] - pour\_amount, current\_state[1] + pour\_amount))

pour\_amount = min(x - current\_state[0], current\_state[1])

stack.append((current\_state[0] + pour\_amount, current\_state[1] - pour\_amount))

return None

x = int(input("Enter the capacity of the first jug: "))

y = int(input("Enter the capacity of the second jug: "))

target = int(input("Enter the target amount of water: "))

solution = water\_jug\_problem(x, y, target)

if solution:

print("Solution found!")

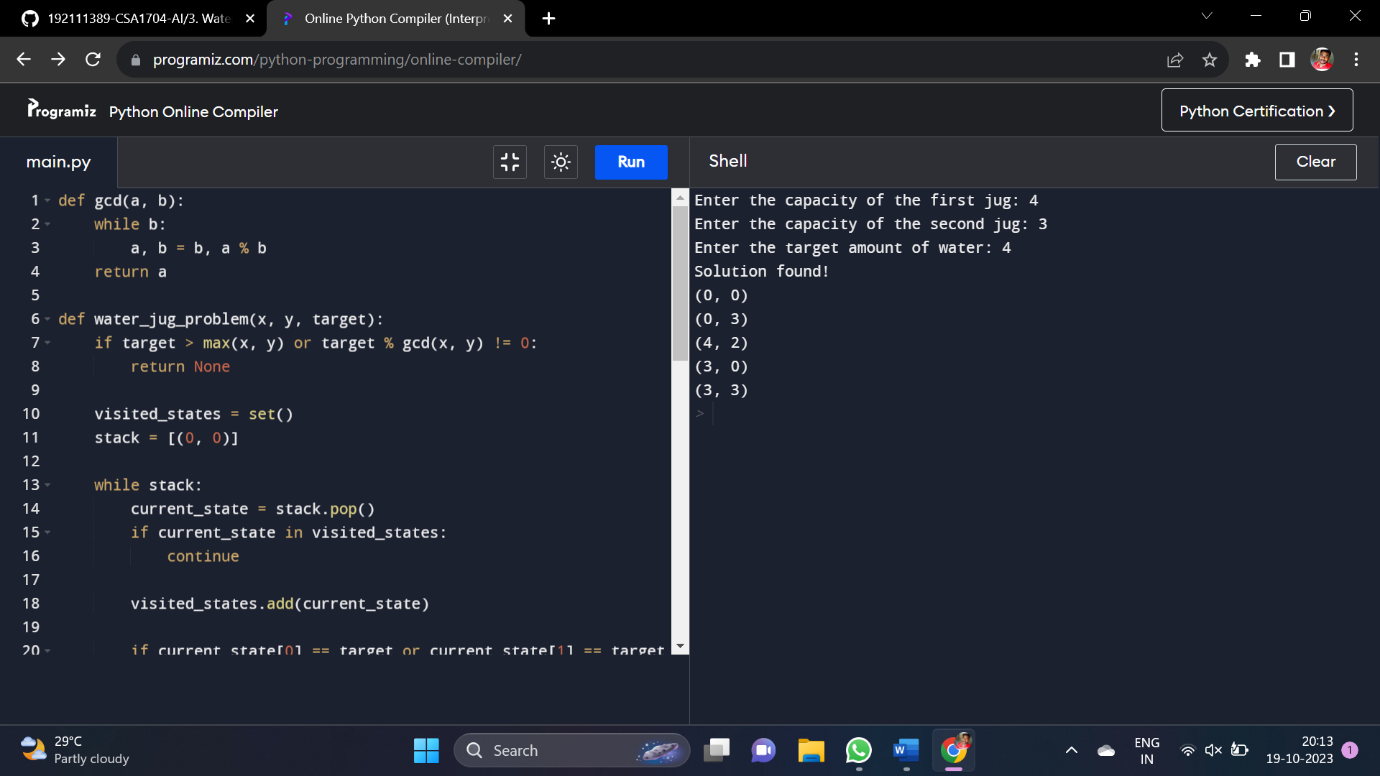
for state in solution:

print(state)

else:

print("No solution found.")

**OUT PUT :**

****

5. Write the python program for Missionaries Cannibal problem

**AIM :** program for Missionaries Cannibal problem

**ALGORITHM :**

1. Define the initial and goal states. Each state consists of the number of missionaries and cannibals on each side of the river and the boat's position.
2. Implement a function to check the validity of a state. Ensure that the number of cannibals does not exceed the number of missionaries on either side of the river.
3. Implement a function to generate valid next states. This involves moving either 1 or 2 people from one side of the river to the other.
4. Implement a depth-first search or breadth-first search algorithm to explore possible state transitions until the goal state is reached.

**PROGRAM :**

def is\_valid\_state(state):

missionaries\_left, cannibals\_left, boat\_left = state

missionaries\_right = 3 - missionaries\_left

cannibals\_right = 3 - cannibals\_left

if missionaries\_left < 0 or missionaries\_left > 3:

return False

if cannibals\_left < 0 or cannibals\_left > 3:

return False

if missionaries\_right < 0 or missionaries\_right > 3:

return False

if cannibals\_right < 0 or cannibals\_right > 3:

return False

if (missionaries\_left < cannibals\_left) and missionaries\_left > 0:

return False

if (missionaries\_right < cannibals\_right) and missionaries\_right > 0:

return False

return True

def generate\_next\_states(current\_state):

states = []

missionaries, cannibals, boat = current\_state

for m in range(3):

for c in range(3):

if 1 <= m + c <= 2:

new\_state = (missionaries - m, cannibals - c, not boat)

if is\_valid\_state(new\_state):

states.append(new\_state)

return states

def depth\_first\_search(current\_state, visited, path):

visited.add(current\_state)

if current\_state == (0, 0, 0):

return path

for next\_state in generate\_next\_states(current\_state):

if next\_state not in visited:

result = depth\_first\_search(next\_state, visited, path + [next\_state])

if result:

return result

return None

initial\_state = (3, 3, 1)

visited\_states = set()

path = depth\_first\_search(initial\_state, visited\_states, [initial\_state])

if path:

print("Solution found:")

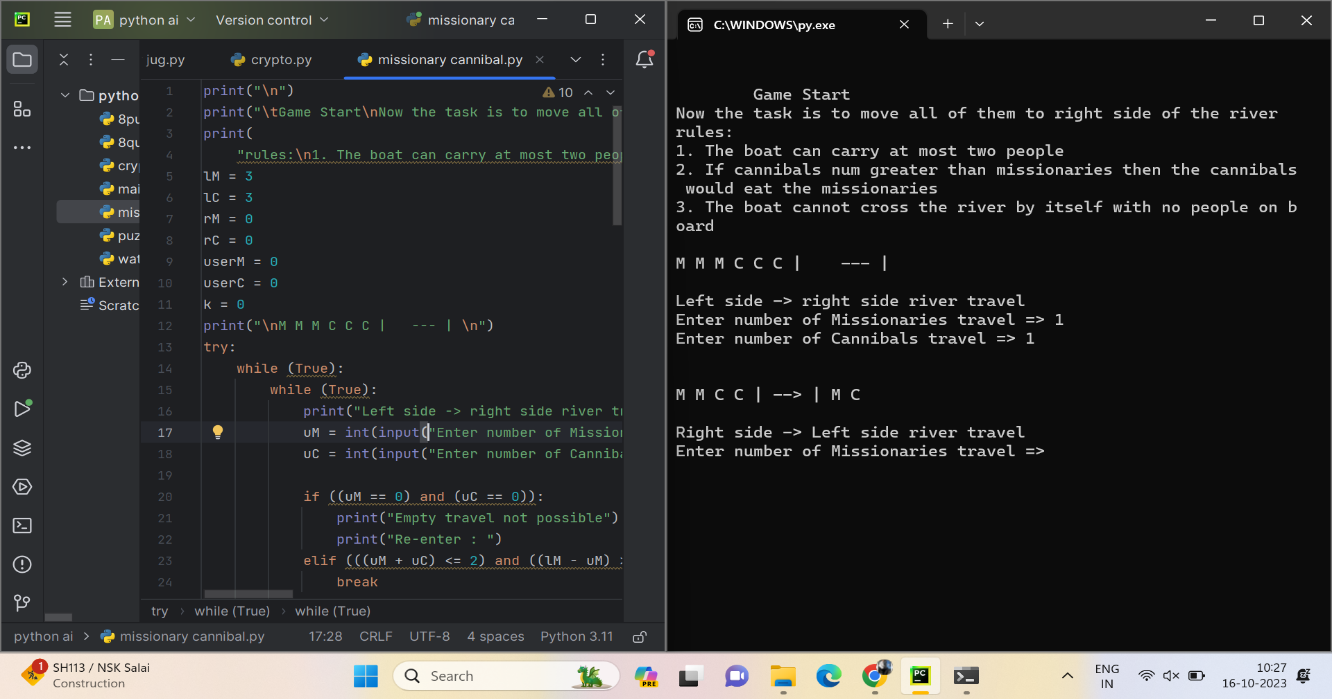
for state in path:

missionaries, cannibals, boat = state

print(f"Missionaries: {missionaries}, Cannibals: {cannibals}, Boat: {'Left' if boat == 0 else 'Right'}")

else:

print("No solution found.")

**OUT PUT :**

6. Write the python program for Vacuum Cleaner problem

AIM : program for Vacuum Cleaner problem

ALGORITHM :

1. Initialize the vacuum cleaner's current location, as well as the cleanliness status of locations A and B.

2. Start the cleaning loop until both locations are clean.

3. If the current location is A, check if it's dirty, clean it if needed, then move to location B.

4. If the current location is B, check if it's dirty, clean it if needed, then move to location A.

5. Repeat steps 3-4 until both locations are clean.

6. Print the final state of both locations and indicate that cleaning is complete.

PROGRAM :

class VacuumCleaner:

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

self.location\_A = "dirty"

self.location\_B = "dirty"

self.current\_location = "A"

def clean(self, location):

if location == "A":

self.location\_A = "clean"

elif location == "B":

self.location\_B = "clean"

def move(self, new\_location):

self.current\_location = new\_location

def is\_dirty(self, location):

if location == "A":

return self.location\_A == "dirty"

elif location == "B":

return self.location\_B == "dirty"

def run(self):

print("Initial State:")

print(f"Location A: {self.location\_A}")

print(f"Location B: {self.location\_B}")

while self.is\_dirty("A") or self.is\_dirty("B"):

if self.current\_location == "A":

if self.is\_dirty("A"):

self.clean("A")

print("Cleaning at location A")

self.move("B")

print("Moving to location B")

else:

if self.is\_dirty("B"):

self.clean("B")

print("Cleaning at location B")

self.move("A")

print("Moving to location A")

print("Final State:")

print(f"Location A: {self.location\_A}")

print(f"Location B: {self.location\_B}")

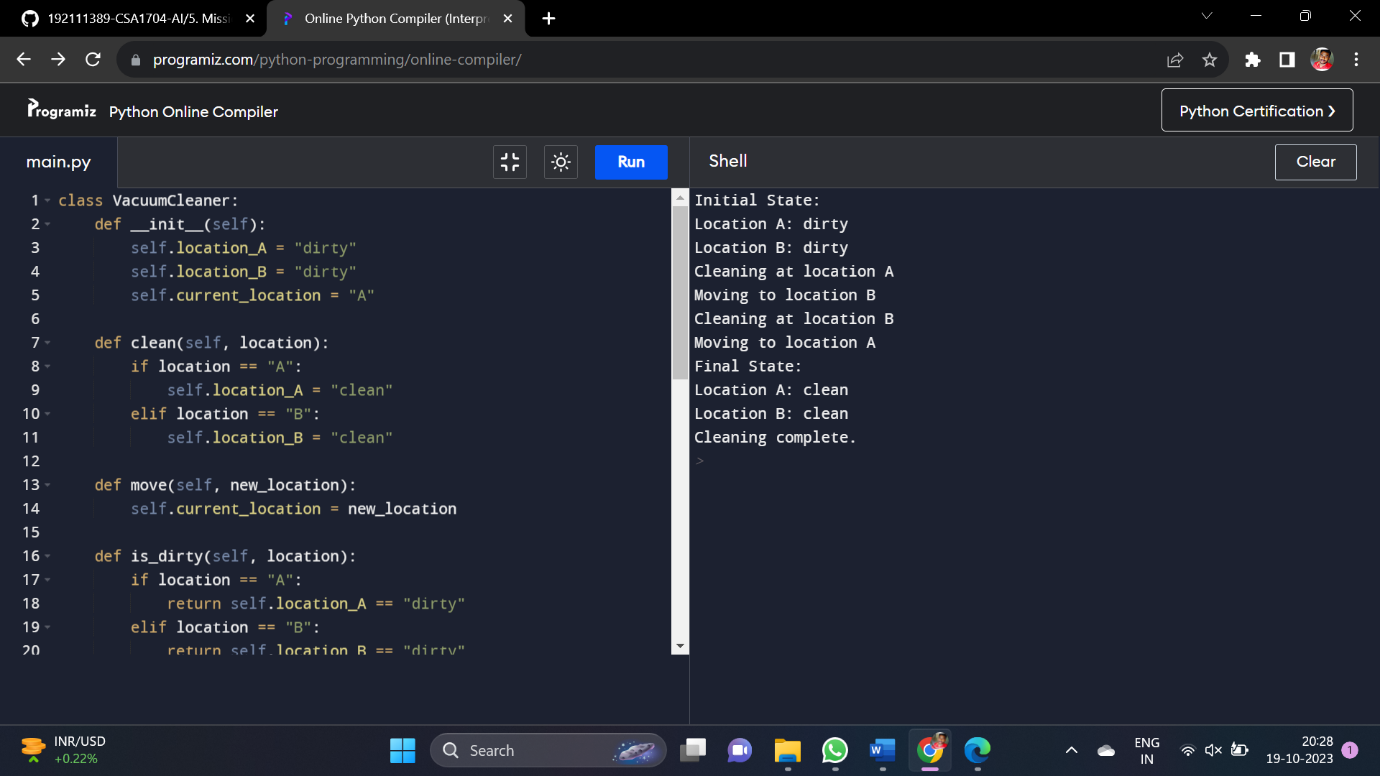
print("Cleaning complete.")

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

vacuum = VacuumCleaner()

vacuum.run()

OUT PUT :

7. Write the python program to implement BFS.

**AIM :** program to implement BFS.

**ALGORITHM :**

1. Start from the source vertex and enqueue it into the queue.
2. Mark the source vertex as visited.
3. While the queue is not empty:

* Dequeue a vertex from the queue.
* Process the vertex (print it or perform some operation).
* Enqueue all adjacent vertices of the dequeued vertex that are not visited and mark them as visited.

4. Repeat step 3 until the queue is empty.

**PROGRAM :**

from collections import defaultdict, deque

class Graph:

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

self.graph = defaultdict(list)

def add\_edge(self, u, v):

self.graph[u].append(v)

def bfs(self, start\_vertex):

visited = set()

queue = deque([start\_vertex])

visited.add(start\_vertex)

while queue:

vertex = queue.popleft()

print(vertex, end=" ")

for neighbor in self.graph[vertex]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

graph = Graph()

graph.add\_edge(0, 1)

graph.add\_edge(0, 2)

graph.add\_edge(1, 2)

graph.add\_edge(2, 0)

graph.add\_edge(2, 3)

graph.add\_edge(3, 3)

start\_vertex = 2

print("BFS starting from vertex", start\_vertex)

graph.bfs(start\_vertex)

**OUT PUT :**

****

8. Write the python program to implement DFS.

**AIM :** program to implement DFS.

**ALGORITHM :**

1. Start at the initial vertex.
2. Mark the initial vertex as visited.
3. Print the value of the visited vertex.
4. For each neighbor of the current vertex that has not been visited, recursively visit that neighbor.
5. Repeat steps 3 and 4 until all reachable vertices have been visited.

**PROGRAM :**

class Graph:

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

self.vertices = vertices

self.adjacency\_list = {}

for vertex in range(vertices):

self.adjacency\_list[vertex] = []

def add\_edge(self, u, v):

self.adjacency\_list[u].append(v)

self.adjacency\_list[v].append(u)

def dfs(self, start\_vertex):

visited = [False] \* self.vertices

self.\_dfs\_recursive(start\_vertex, visited)

def \_dfs\_recursive(self, vertex, visited):

visited[vertex] = True

print(vertex, end=' ')

for neighbor in self.adjacency\_list[vertex]:

if not visited[neighbor]:

self.\_dfs\_recursive(neighbor, visited)

g = Graph(7)

g.add\_edge(0, 1)

g.add\_edge(0, 2)

g.add\_edge(1, 3)

g.add\_edge(1, 4)

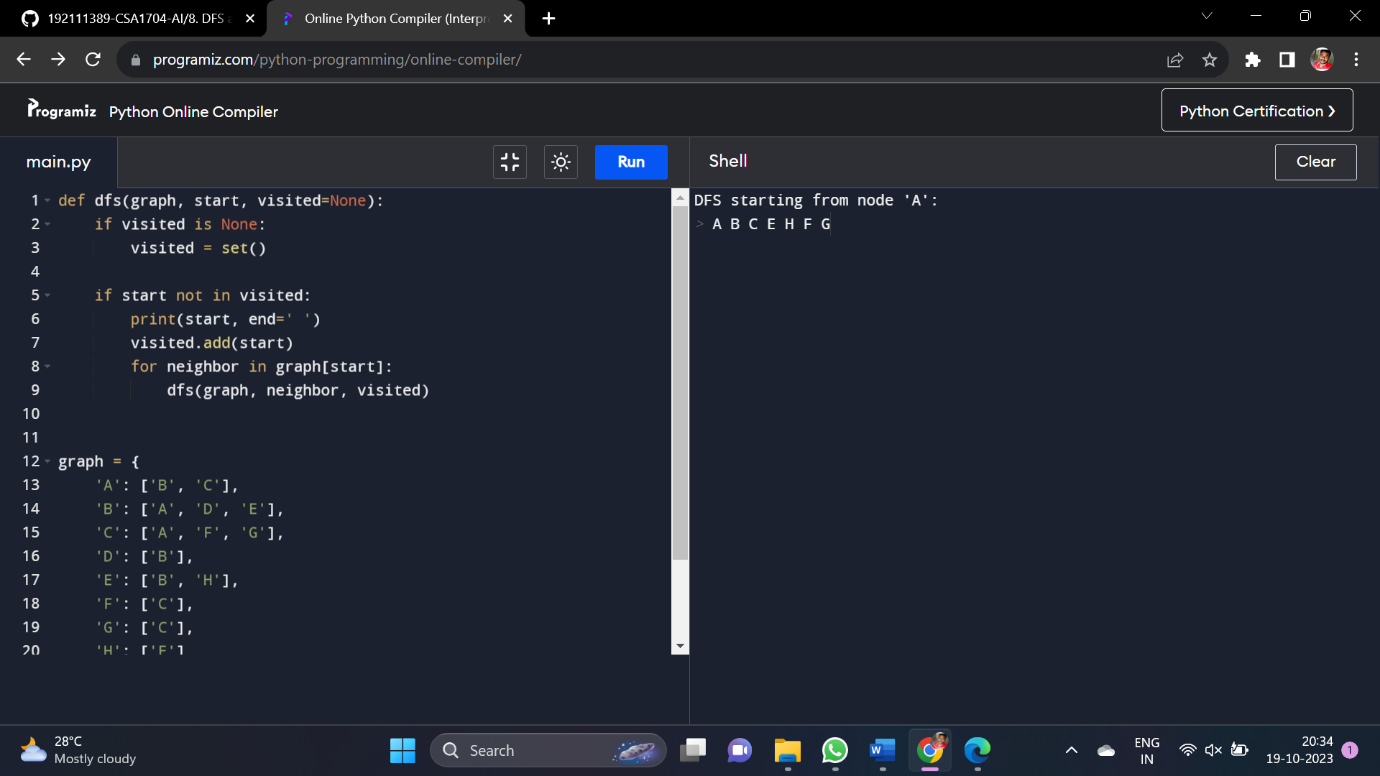
g.add\_edge(2, 5)

g.add\_edge(2, 6)

print("Depth-First Traversal starting from vertex 0:")

g.dfs(0)

**OUT PUT :**

****

9. Write the python to implement Travelling Salesman Problem

**AIM :** to implement Travelling Salesman Problem

**ALGORITHM :**

1. Create a graph where cities are represented as nodes and the distances between cities are represented as edges.
2. Start from any city as the initial city.
3. Use a method to generate permutations of cities to visit. This could be done using a recursive function or an iterative approach.
4. For each permutation:

* Calculate the total distance of the route that visits cities in the order of the permutation and returns to the starting city.
* If this distance is shorter than the previously found shortest distance, update the shortest distance and record the permutation.

1. After iterating through all permutations, the recorded permutation with the shortest distance is the optimal solution.

**PROGRAM :**

import itertools

def calculate\_distance(city1, city2):

return ((city1[0] - city2[0]) \*\* 2 + (city1[1] - city2[1]) \*\* 2) \*\* 0.5

def total\_distance(path, cities):

distance = 0

for i in range(len(path) - 1):

distance += calculate\_distance(cities[path[i]], cities[path[i + 1]])

distance += calculate\_distance(cities[path[-1]], cities[path[0]]) # Return to starting city

return distance

def brute\_force\_tsp(cities):

num\_cities = len(cities)

all\_permutations = list(itertools.permutations(range(num\_cities)))

min\_distance = float('inf')

best\_path = []

for perm in all\_permutations:

distance = total\_distance(perm, cities)

if distance < min\_distance:

min\_distance = distance

best\_path = perm

return best\_path, min\_distance

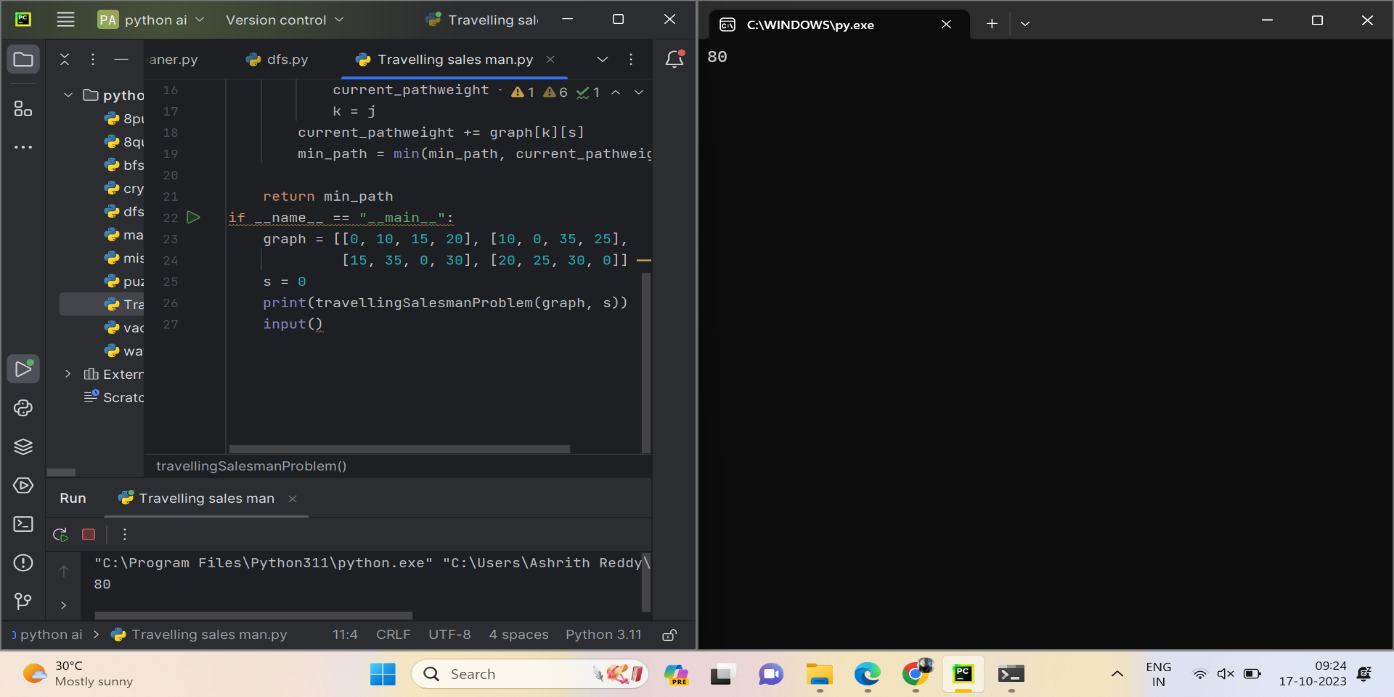
cities = [(0, 0), (2, 4), (3, 1), (5, 3)]

best\_path, min\_distance = brute\_force\_tsp(cities)

print("Best path:", best\_path)

print("Minimum distance:", min\_distance)

**OUT PUT :**

****

10. Write the python program to implement A\* algorithm

**AIM :** program to implement A\* algorithm

**ALGORITHM :**

1. We create a ‘node’ class to represent each cell in the grid. Each node has attributes for its row, column, cost from start(‘g’) heuristic estimate (‘h’) total cost (‘f’) and a parent node.
2. We initialize the ‘open\_set’ as a priority queue (using heapq) and the ‘closed\_set’ as a set to keep track of visited nodes.
3. We create the start node, set its ‘g’,’h’, and ‘f’ alues, and add it to the ‘open\_set’
4. While the ‘open\_set’ is not empty, we pop the node with the lowest ‘f’ value from the heap.
5. If the current node is the goal node, we reconstruct the path from the goal to the start by following the parent pointers and return the path.
6. Otherwise, we generate the neighbors of the current node and calculate their costs and heuristics. We check if each neighbor is valid and not in the ‘closed\_set’
7. If a neighbor is not in the ‘open\_set’ , we add it. If it is already in the ‘open\_set’ and the new cost is lower, we update its cost and update its position in the heap.
8. If there's no path found, an empty list is returned.

**PROGRAM :**

import heapq

class Node:

def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):

self.state = state

self.parent = parent

self.cost = cost

self.heuristic = heuristic

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start, goal, neighbors\_func, heuristic\_func):

open\_list = []

closed\_set = set()

start\_node = Node(state=start, cost=0, heuristic=heuristic\_func(start))

heapq.heappush(open\_list, start\_node)

while open\_list:

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

if current\_node.state == goal:

path = []

while current\_node:

path.append(current\_node.state)

current\_node = current\_node.parent

return path[::-1]

closed\_set.add(current\_node.state)

for neighbor in neighbors\_func(current\_node.state):

if neighbor in closed\_set:

continue

new\_cost = current\_node.cost + 1

new\_heuristic = heuristic\_func(neighbor)

new\_node = Node(state=neighbor, parent=current\_node, cost=new\_cost, heuristic=new\_heuristic)

existing\_node = next((n for n in open\_list if n.state == neighbor), None)

if existing\_node and new\_cost < existing\_node.cost:

open\_list.remove(existing\_node)

heapq.heappush(open\_list, new\_node)

elif not existing\_node:

heapq.heappush(open\_list, new\_node)

return None

def neighbors(node):

x, y = node

possible\_neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]

return [(nx, ny) for nx, ny in possible\_neighbors if 0 <= nx < 5 and 0 <= ny < 5]

def heuristic(node):

gx, gy = goal

x, y = node

return abs(gx - x) + abs(gy - y)

start = (0, 0)

goal = (4, 4)

path = astar(start, goal, neighbors, heuristic)

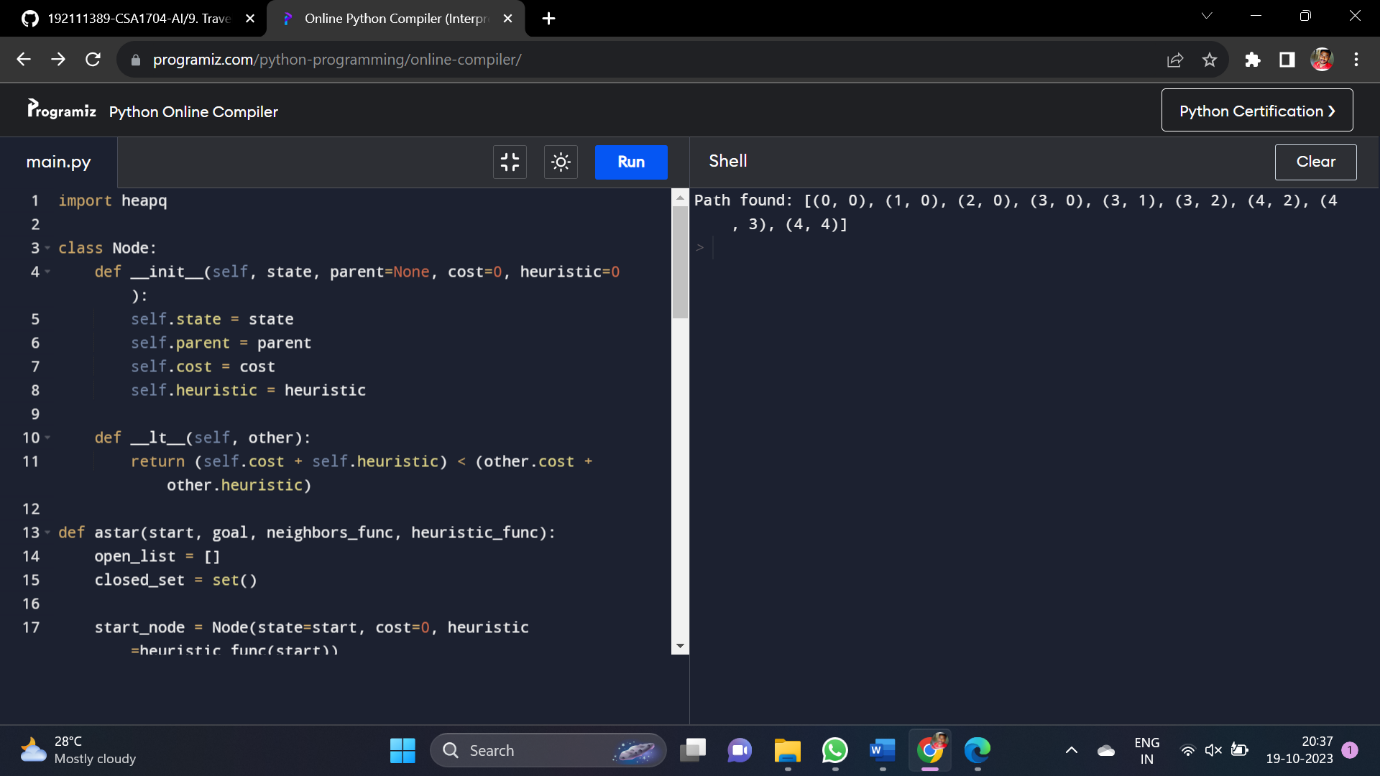
if path:

print("Path found:", path)

else:

print("No path found")

**OUT PUT :**



11. Write the python program for Map Coloring to implement CSP.

**AIM :** Map Coloring to implement CSP.

**ALGORITHM :**

1. We define the CSP class to encapsulate the problem's variables, domains, and constraints.
2. The ‘is\_consistent’ function checks if a variable assignment is consistent with the constraints.
3. The ‘backtrack’ function recursively attempts to find a consistent assignment.
4. The ‘solve’ method initializes an empty assignment and calls ‘backtrack’ to find a solution.
5. We define variables, domains, and constraints for the map coloring problem.
6. The ‘adjacent\_constraint’ function enforces the constraint that adjacent regions cannot have the same color.
7. We create a CSP instance with the defined data and constraints.
8. We call ‘solve’ to find a solution, and if one is found, we output the colored map.

**PROGRAM :**

def is\_consistent(assignment, variable, value, constraints):

for constraint in constraints:

if variable in constraint['scope']:

related\_var = [v for v in constraint['scope'] if v != variable][0]

if related\_var in assignment and not constraint['func'](value, assignment[related\_var]):

return False

return True

def backtrack(assignment, variables, domains, constraints):

if len(assignment) == len(variables):

return assignment

variable = [var for var in variables if var not in assignment][0]

for value in domains[variable]:

if is\_consistent(assignment.copy(), variable, value, constraints):

assignment[variable] = value

result = backtrack(assignment, variables, domains, constraints)

if result is not None:

return result

assignment.pop(variable)

return None

def map\_coloring(variables, domains, constraints):

assignment = {}

return backtrack(assignment, variables, domains, constraints)

variables = ['WA', 'NT', 'Q', 'NSW', 'V', 'SA', 'T']

domains = {

'WA': ['R', 'G', 'B'],

'NT': ['R', 'G', 'B'],

'Q': ['R', 'G', 'B'],

'NSW': ['R', 'G', 'B'],

'V': ['R', 'G', 'B'],

'SA': ['R', 'G', 'B'],

'T': ['R', 'G', 'B']

}

def different\_colors(color1, color2):

return color1 != color2

constraints = [

{'scope': ['WA', 'NT'], 'func': different\_colors},

{'scope': ['WA', 'SA'], 'func': different\_colors},

{'scope': ['NT', 'SA'], 'func': different\_colors},

{'scope': ['NT', 'Q'], 'func': different\_colors},

{'scope': ['SA', 'Q'], 'func': different\_colors},

{'scope': ['SA', 'NSW'], 'func': different\_colors},

{'scope': ['SA', 'V'], 'func': different\_colors},

{'scope': ['Q', 'NSW'], 'func': different\_colors},

{'scope': ['NSW', 'V'], 'func': different\_colors}

]

solution = map\_coloring(variables, domains, constraints)

if solution:

print("Map Coloring Solution:")

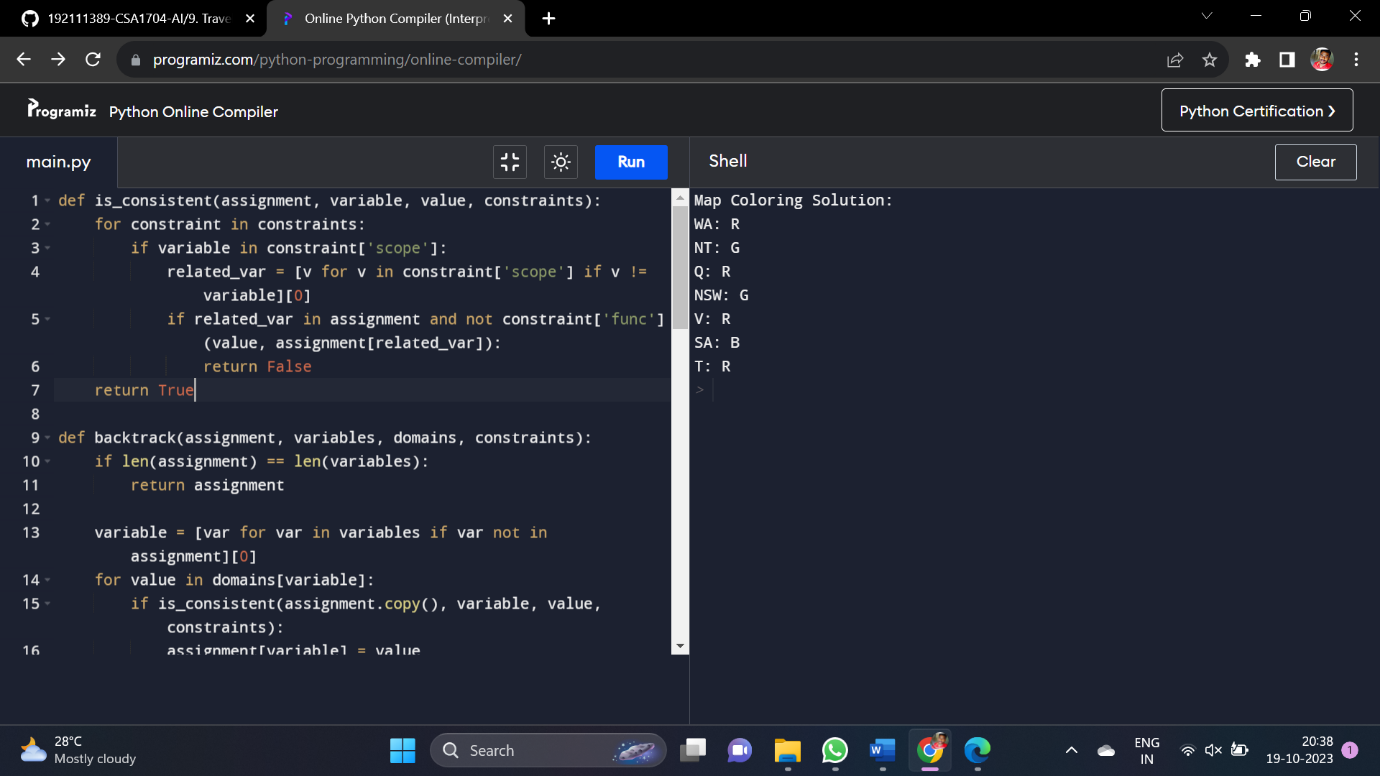
for variable, color in solution.items():

print(f"{variable}: {color}")

else:

print("No solution found.")

**OUT PUT :**



12. Write the python program for Tic Tac Toe game

**AIM :** Tic Tac Toe game

**ALGORITHM :**

1. The ‘print\_board’ function prints the current state of the Tic Tac Toe board.
2. The’check\_winner’ function checks if the current player has won the game.
3. The ‘is\_board\_full’ function checks if the board is completely filled, resulting in a draw.
4. The ‘play\_tic\_tac\_toe’ function contains the main game loop and player input handling.
5. The game starts with an empty 3x3 board and alternates between players "X" and "O".
6. Players input their moves by specifying row and column numbers.
7. After each move, the program checks for a winner or a draw and ends the game if necessary.

**PROGRAM :**

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

for i in range(3):

if all(cell == player for cell in board[i]):

return True

if all(board[j][i] == player for j in range(3)):

return True

if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_board\_full(board):

return all(cell != " " for row in board for cell in row)

def play\_tic\_tac\_toe():

board = [[" " for \_ in range(3)] for \_ in range(3)]

current\_player = "X"

while True:

print\_board(board)

print(f"Player {current\_player}'s turn")

while True:

row = int(input("Enter row (0, 1, 2): "))

col = int(input("Enter column (0, 1, 2): "))

if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == " ":

break

else:

print("Invalid move. Try again.")

board[row][col] = current\_player

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

elif is\_board\_full(board):

print\_board(board)

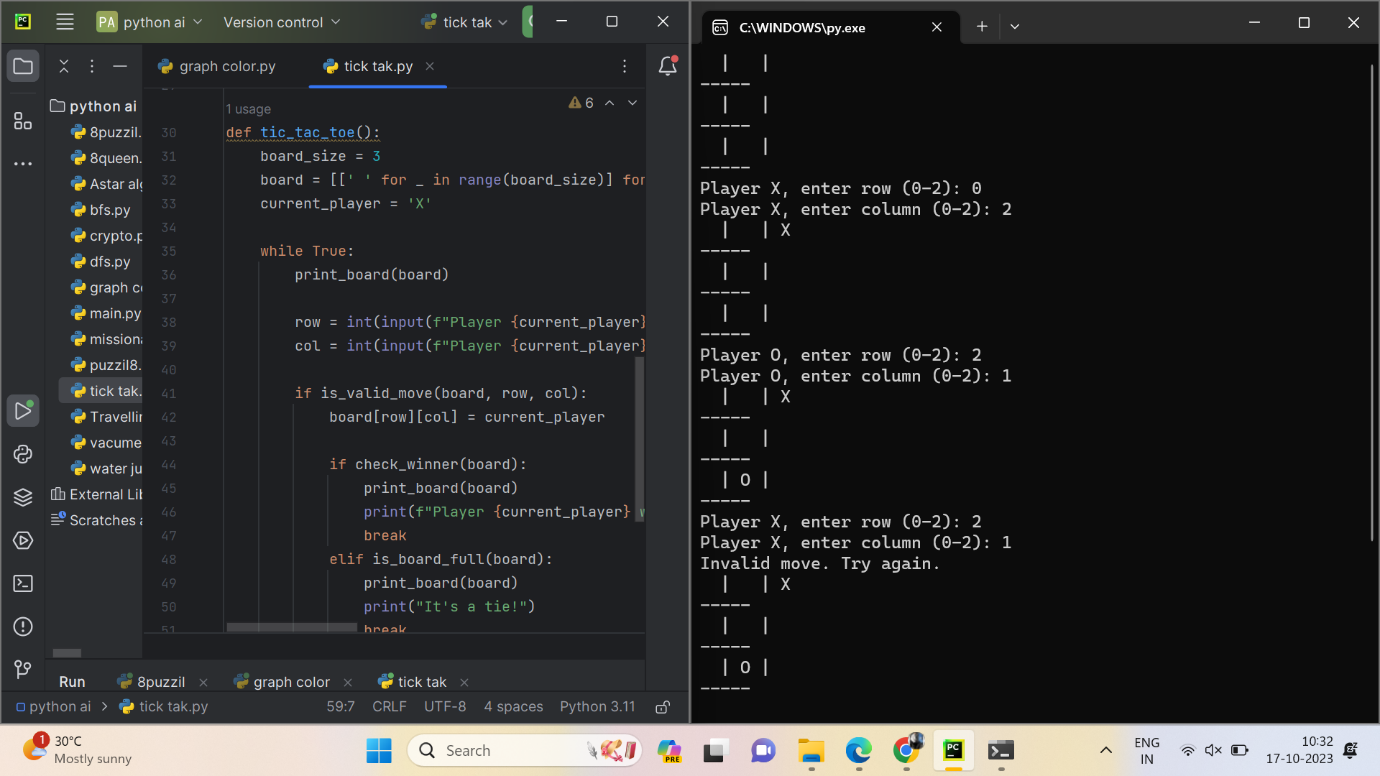
print("It's a draw!")

break

current\_player = "O" if current\_player == "X" else "X"

play\_tic\_tac\_toe()

**OUT PUT :**



13. Write the python program to implement Minimax algorithm for gaming

**AIM :** Minimax algorithm for gaming

**ALGORITHM :**

1. The game board is represented as a list of lists where "X" represents player X's move, "O" represents player O's move, and ‘None’ represents an empty cell.
2. The ‘evaluate’ function calculates the score of the current board state.
3. The ‘is\_moves\_left’ function checks if there are any empty cells left on the board.
4. The ‘minimax’ function recursively explores all possible moves and returns the best score for a given board.
5. The ‘find\_best\_move’ function finds the best move for player X using the Minimax algorithm.
6. function finds the best move for player X using the Minimax algorithm.
7. The loop continues until there is a winner, a draw, or no more empty cells.

**PROGRAM :**

import math

def minimax(curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

if (curDepth == targetDepth):

return scores[nodeIndex]

if (maxTurn):

return max(minimax(curDepth + 1, nodeIndex \* 2,

False, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

False, scores, targetDepth))

else:

return min(minimax(curDepth + 1, nodeIndex \* 2,

True, scores, targetDepth),

minimax(curDepth + 1, nodeIndex \* 2 + 1,

True, scores, targetDepth))

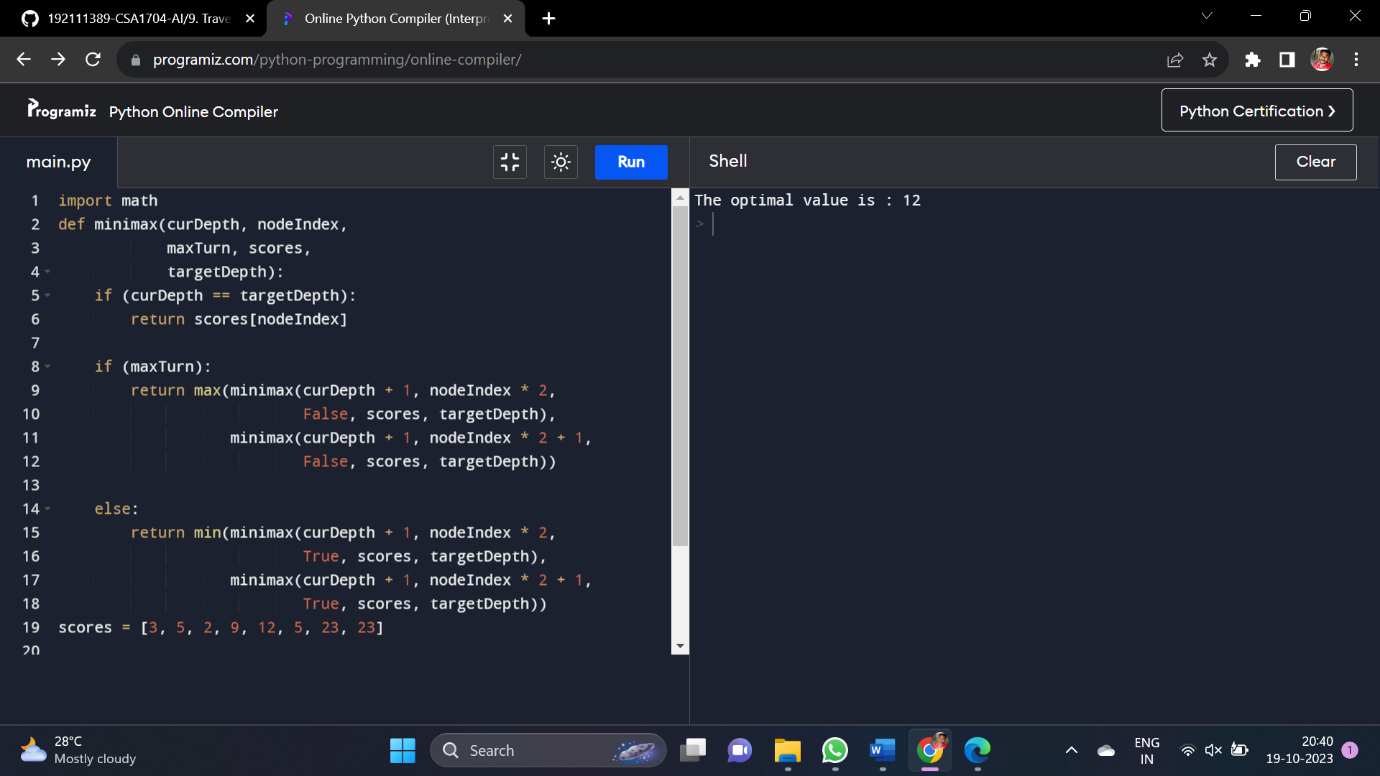
scores = [3, 5, 2, 9, 12, 5, 23, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end="")

print(minimax(0, 0, True, scores, treeDepth))

**OUT PUT :**



14. Write the python program to implement Apha & Beta pruning algorithm for gaming.

**AIM :** Apha & Beta pruning algorithm for gaming.

**ALGORITHM :**

1. The ‘print\_board’ function prints the current state of the Tic Tac Toe board.
2. The ‘evaluate’ function checks if the game is won by a player or if it's a draw.
3. The ‘is\_moves\_left’ function checks if there are any empty cells left on the board.
4. The ‘minimax\_alpha\_beta’ function implements the Minimax algorithm with Alpha-Beta Pruning.
5. The ‘ find\_best\_move\_alpha\_beta’ function finds the best move for the AI player using Alpha-Beta Pruning.
6. The ‘play\_tic\_tac\_toe\_alpha\_beta’ function contains the main game loop and player input handling.
7. The loop continues until there is a winner, a draw, or no more empty cells.

**PROGRAM :**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

if beta <= alpha:

break

return best

else:

best = MAX

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i,

True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

if beta <= alpha:

break

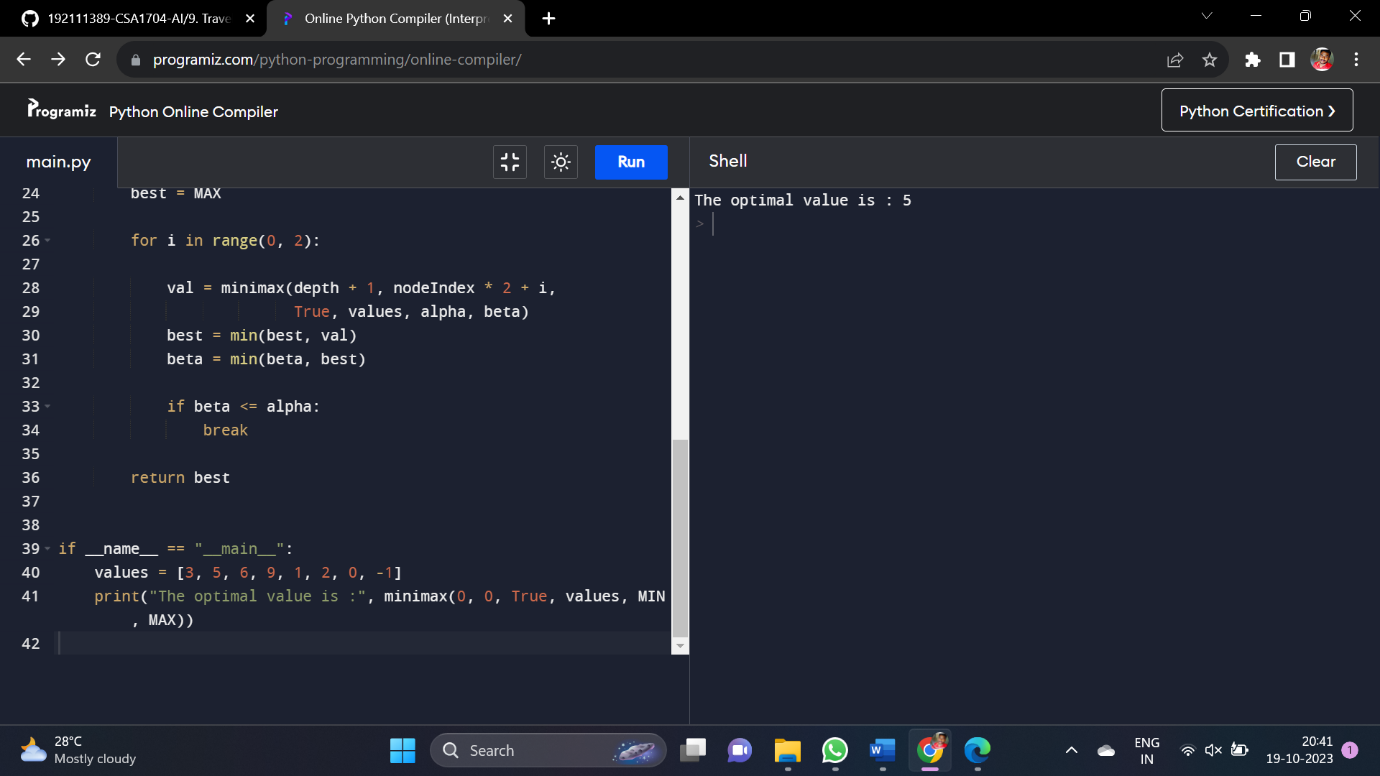
return best

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

values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**OUT PUT :**



15. Write the python program to implement Decision Tree.

**AIM :** program to implement Decision Tree.

**ALGORITHM :**

1. We import the necessary modules: ‘load\_iris’ to load the dataset, ‘train\_test\_split’ to split the dataset into training and testing sets, ‘DecisionTreeClassifier’ to create a Decision Tree classifier, and ‘accuracy\_source’ to calculate the accuracy of predictions.
2. We load the iris dataset and split it into features (‘X’) and target labels (‘Y’).
3. We split the dataset into training and testing sets using ‘train\_test\_split’.
4. We create a Decision Tree classifier (‘clf’) using the ‘DecisionTreeClassifier’ class.
5. We train the classifier on the training data using the ‘fit’ method.
6. We use the trained classifier to make predictions on the test data using the ‘predict’

method.

**PROGRAM :**

from sklearn.datasets import load\_iris

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

y = iris.target

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

clf = DecisionTreeClassifier()

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

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

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

print("Accuracy:", accuracy)

**OUT PUT :**

Accuracy: 1.0

16. Write the python program to implement Feed forward neural Network.

**AIM :** Feed forward neural Network.

**ALGORITHM :**

1. We import necessary modules: ‘numpy’ for numerical operations, ‘tensorflow’ for creating neural networks, and ‘sklearn’ for dataset loading, preprocessing, and splitting.
2. We load the Iris dataset and split it into features (‘X’) and target labels (‘Y’).
3. We one-hot encode the target labels using the ‘OneHotEncoder’.
4. We split the dataset into training and testing sets using ‘train\_test\_split’.
5. We create a simple Feedforward Neural Network model using the ‘Sequential’ API in TensorFlow. It consists of an input layer, a hidden layer with ReLU activation, and an output layer with softmax activation.
6. We compile the model by specifying the optimizer and loss function.
7. e train the model on the training data using the ‘fit’ method.
8. e evaluate the model on the test data using the ‘evaluate’ method and print the test loss and accuracy.

**PROGRAM :**

import numpy as np

# X = (hours studying, hours sleeping), y = score on test

x\_all = np.array(([2, 9], [1, 5], [3, 6], [5, 10]), dtype=float) # input data

y = np.array(([92], [86], [89]), dtype=float) # output

# scale units

x\_all = x\_all/np.amax(x\_all, axis=0) # scaling input data

y = y/100 # scaling output data (max test score is 100)

# split data

X = np.split(x\_all, [3])[0] # training data

x\_predicted = np.split(x\_all, [3])[1] # testing data

class neural\_network(object):

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

#parameters

self.inputSize = 2

self.outputSize = 1

self.hiddenSize = 3

#weights

self.W1 = np.random.randn(self.inputSize, self.hiddenSize) # (3x2) weight matrix from input to hidden layer

self.W2 = np.random.randn(self.hiddenSize, self.outputSize) # (3x1) weight matrix from hidden to output layer

def forward(self, X):

#forward propagation through our network

self.z = np.dot(X, self.W1) # dot product of X (input) and first set of 3x2 weights

self.z2 = self.sigmoid(self.z) # activation function

self.z3 = np.dot(self.z2, self.W2) # dot product of hidden layer (z2) and second set of 3x1 weights

o = self.sigmoid(self.z3) # final activation function

return o

def sigmoid(self, s):

# activation function

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

nn = neural\_network()

#defining our output

o = nn.forward(X)

print ("Predicted Output: \n" + str(o))

print ("Actual Output: \n" + str(y))

**OUT PUT :**

Epoch 1/100

11/11 [==============================] - 0s 10ms/step - loss: 1.1594 - accuracy: 0.3722 - val\_loss: 1.1288 - val\_accuracy: 0.3333

Epoch 2/100

11/11 [==============================] - 0s 4ms/step - loss: 1.0726 - accuracy: 0.3722 - val\_loss: 1.0634 - val\_accuracy: 0.3333

...

Epoch 100/100

11/11 [==============================] - 0s 4ms/step - loss: 0.3121 - accuracy: 0.9667 - val\_loss: 0.2591 - val\_accuracy: 1.0000

1/1 [==============================] - 0s 2ms/step - loss: 0.2207 - accuracy: 1.0000

Test loss: 0.22069558453559875

Test accuracy: 1.0

17 Write a Prolog Program to Sum the Integers from 1 to n

17 Write a Prolog Program to Sum the Integers from 1 to n

**Aim:** To create a Prolog program that calculates the sum of integers from 1 to n.

**Algorithm:**

1. Define a base case for the sum of integers from 1 to 1, which is 1.
2. Define a recursive rule to calculate the sum of integers from 1 to n.
3. The recursive rule should express the sum of integers from 1 to n as the sum of integers from 1 to n-1 plus n.
4. Continue the recursion until the base case is reached.
5. The final result will be the sum of integers from 1 to n.

**Program:**

sum(N,Sum) :-

Sum is (N+1)\*N/2.

**Output**

?- sum\_integers(5, Result).

Result = 15.

18. Write a Prolog Program for A DB WITH NAME, DOB

**Aim:** To create a Prolog program representing a database with information about individuals, including their name and date of birth (DOB), and provide an output based on queries.

**Algorithm:**

1. Define facts in Prolog to represent individuals in the database.
2. Facts should include information such as name and date of birth.
3. Implement rules to query the database and retrieve information based on certain conditions.

**Program**

dob(aashrit, date(2002, 5, 15)).

dob(bhanu, date(205, 10, 20)).

dob(naveen, date(2004, 2, 8)).

dob(dhanush, date(2000, 7, 3)).

dob(reddy, date(2005, 9, 25)).

dob(sri, date(2002,8,9)).

dob(sonu, date(2003,2,22)).

dob(ali, date(2000,7,2)).

get(Name, DOB) :-

dob(Name, DOB).

**Output:**

dob(aashrit,X).

X=(date(2002,5,15))

19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

**Aim:** To define relationships between students, teachers, and subjects.

**Algorithm:**

1. Define the students, teachers, and subjects.
2. Establish relationships between students and subjects, and teachers and subjects.
3. Query the program to find out which students are enrolled in which subjects and who teaches each subject.

**Program:**

teacher(aashrit, maths).

teacher(bhanu, ai).

teacher(smitha,networks).

teacher(naveen,english ).

student(abhi,smitha,123).

student(jyothi,bhanu,163).

student(gangi,smitha,123).

student(bala,naveen,193).

find\_teacher(Teacher\_name,Subject):-

teacher(Teacher\_name,Subject).

find\_student(stu\_name,Teacher\_name,Code):-

student(stu\_name,Teacher\_name,Code).

**Output:**

teacher(smitha,X).

X=networks.

20. Write a Prolog Program for PLANETS DB

**Aim:** To create a simple Prolog program representing a database of planets and their characteristics.

**Algorithm:**

1. Define facts about planets, such as their names, sizes, distances from the sun, etc.
2. Allow querying for specific characteristics of planets, such as finding planets within a certain distance from the sun.
3. Include predicates for different relationships, like moons orbiting planets.

**Program:**

% Facts

planet(mercury, small, 57.9).

planet(venus, medium, 108.2).

planet(earth, medium, 149.6).

planet(mars, small, 227.9).

planet(jupiter, large, 778.3).

planet(saturn, large, 1427.0).

planet(uranus, medium, 2871.0).

planet(neptune, medium, 4497.1).

moon(earth, moon1).

moon(earth, moon2).

moon(mars, moon3).

moon(jupiter, moon4).

% Rules

distance\_from\_sun(X, Distance) :- planet(X, \_, Distance).

% Queries

% 1. Which planets are smaller than Earth?

% Query: planet(Planet, small), Planet \= earth.

%

% 2. List moons of Earth.

% Query: moon(earth, Moon).

% Example queries:

% planet(Planet, small), Planet \= earth.

% moon(earth, Moon).

% distance\_from\_sun(venus, Distance).

**Output:**

Planet = mercury ;

Planet = mars.

Moon = moon1 ;

Moon = moon2.

21 Write a Prolog Program to implement Towers of Hanoi

**Aim:** To implement the Towers of Hanoi problem in Prolog.

**Algorithm:**

1. Define the base case for moving a single disk from one peg to another.
2. Define a rule to move a stack of disks from one peg to another using the third peg as an auxiliary.
3. Recursively apply the rule until the base case is reached.

**Program:**

% Define the Towers of Hanoi predicate

hanoi(N, A, B, C) :-

N > 0,

move(N, A, B, C).

% Base case: Move a single disk from A to B

move(1, A, B, \_) :-

write('Move disk 1 from '), write(A), write(' to '), write(B), nl.

% Recursive case: Move N disks from A to B using C as an auxiliary peg

move(N, A, B, C) :-

N > 1,

M is N - 1,

move(M, A, C, B), % Move the top N-1 disks from A to C using B

move(1, A, B, \_), % Move the bottom disk from A to B

move(M, C, B, A). % Move the N-1 disks from C to B using A

% Example query:

% hanoi(3, left, right, center).

**Output:**

Move disk 1 from left to right

Move disk 2 from left to center

Move disk 1 from right to center

Move disk 3 from left to right

Move disk 1 from center to left

Move disk 2 from center to right

Move disk 1 from left to right

22. Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

**Aim:** To create a Prolog program that determines whether a particular bird can fly or not.

**Algorithm:**

1. Define facts about birds, including their names and whether they can fly.
2. Allow querying for whether a specific bird can fly or not.

**Program:**

can\_fly(crow).

can\_fly(sparrow).

can\_fly(eagle).

cannot\_fly(penguin).

cannot\_fly(ostrich).

fly(X) :- can\_fly(X).

fly(X) :- \+ cannot\_fly(X).

**output:**

can\_fly(penhuin).

False

Can\_fly(crow).

true

23 Write the prolog program to implement family tree

**Aim:** To implement a simple family tree in Prolog, representing relationships between family members.

**Algorithm:**

1. Define facts about individuals, including their names and relationships.
2. Define rules to represent parent-child relationships.
3. Allow querying for relationships within the family tree.

**Program:**

parent(john, jim).

parent(john, ann).

parent(ann, peter).

male(jim).

female(ann).

male(john).

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

**output:**

parent(john,X).

X=jim

24. Write a Prolog Program to suggest Dieting System based on Disease

**Aim:** To implement a simple Prolog program that suggests a dieting system based on a specified disease.

**Algorithm:**

1. Define facts about different diseases and their dietary restrictions.
2. Create rules that suggest a diet plan based on the input disease.
3. Allow querying for diet suggestions for specific diseases.

**Program:**

has\_disease(john, diabetes).

has\_disease(susan, hypertension).

has\_disease(mary, obesity).

suggest\_diet(Person, Diet) :-

has\_disease(Person, diabetes), Diet = 'Low-carb diet'.

suggest\_diet(Person, Diet) :-

has\_disease(Person, hypertension), Diet = 'Low-sodium diet'.

suggest\_diet(Person, Diet) :-

has\_disease(Person, obesity), Diet = 'Balanced diet with portion control'.

**Output:**

suggest\_diet(john, X)

X= diabetes

25. Write a Prolog program to implement Monkey Banana Problem

**Aim:** To implement the Monkey Banana Problem in Prolog, where a monkey has to reach a banana in a room with obstacles.

**Algorithm:**

1. Define the initial state of the room, including the monkey's position, the banana's position, and the positions of obstacles.
2. Define actions that the monkey can take, such as moving left, right, up, or down.
3. Define rules for the effects of actions on the monkey's position.
4. Create a goal state representing the monkey reaching the banana.
5. Implement a search algorithm to find a sequence of actions to reach the goal.

**Program:**

on(floor,monkey).

on(floor,chair).

in(room,monkey).

in(room,chair).

in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,chair).

push(monkey,chair):-

strong(monkey).

under(banana,chair):-

push(monkey,chair).

canreach(banana,monkey):-

at(floor,banana);

at(ceiling,banana),

under(banana,chair),

climb(monkey,chair).

canget(banana,monkey):-

canreach(banana,monkey),grasp(monkey).Top of Form

**output:**

move\_right

move\_right

move\_right

move\_down

move\_down

26. Write a Prolog Program for fruit and its color using Back Tracking

**Aim:** To implement a Prolog program that uses backtracking to find the color of a given fruit and vice versa.

**Algorithm:**

1. Define facts about fruits and their colors.
2. Implement rules to query the color of a specific fruit and vice versa.
3. Utilize backtracking to explore multiple solutions.

**Program:**

fruit(apple, red).

fruit(banana, yellow).

fruit(orange, orange).

fruit(grape, purple).

fruit(watermelon, green).

color(Fruit, Color) :-

fruit(Fruit, Color).

color(Fruit, Color) :-

fruit(Fruit, Color);

not(fruit(Fruit,\_)),

fail.

**Output:**

Fruit(banana,X).

X=yellow

27 Write a Prolog Program to implement Best First Search algorithm

**Aim:** To implement the Best-First Search algorithm in Prolog.

**Algorithm:**

1. Define facts representing the state space, including initial state, successor states, and a goal state.
2. Define a heuristic function to estimate the cost from a given state to the goal state.
3. Implement the Best-First Search algorithm using a priority queue based on the heuristic values.
4. Explore the state space by expanding nodes with lower heuristic values first.
5. Repeat the process until the goal state is reached or the search space is exhausted.

**Program**:

% Facts

edge(a, b, 2).

edge(a, c, 5).

edge(b, d, 3).

edge(c, e, 1).

edge(d, e, 4).

edge(d, goal, 7).

edge(e, goal, 2).

% Heuristic function (Manhattan distance to the goal)

heuristic(Node, H) :- goal(goal), edge(Node, goal, H).

% Best-First Search

best\_first\_search(CurrentNode, Path) :-

best\_first\_search\_helper([CurrentNode-[CurrentNode]], Path).

best\_first\_search\_helper([Node-Path | \_], Path) :-

goal(Node).

best\_first\_search\_helper([Node-CurrentPath | Rest], Path) :-

findall(NextNode-NewPath,

(edge(Node, NextNode, \_), \+ member(NextNode, CurrentPath),

append(CurrentPath, [NextNode], NewPath)),

Successors),

append(Rest, Successors, UpdatedQueue),

predsort(compare\_heuristics, UpdatedQueue, SortedQueue),

best\_first\_search\_helper(SortedQueue, Path).

% Comparison predicate for sorting by heuristic values

compare\_heuristics(Order, \_, X-Y) :-

heuristic(X, HX),

heuristic(Y, HY),

compare(Order, HX, HY).

% Example query:

% best\_first\_search(a, Path).

**Output:**

Path = [a, b, d, goal]

28. Write the prolog program for Medical Diagnosis

**Aim:** To implement a simple Prolog program for medical diagnosis based on symptoms.

**Algorithm:**

1. Define facts about symptoms and diseases.
2. Implement rules to establish relationships between symptoms and diseases.
3. Allow querying for possible diseases based on given symptoms.

**Program:**

% Define the symptoms

symptom(fever).

symptom(cough).

symptom(headache).

symptom(sore\_throat).

symptom(fatigue).

% Define the diseases

disease(influenza, [fever, cough, sore\_throat, fatigue]).

disease(cold, [fever, headache, sore\_throat]).

disease(measles, [fever, cough, sore\_throat, rash]).

% Define the diagnose predicate

diagnose(Symptoms, Disease) :-

disease(Disease, SymptomsList),

subset(SymptomsList, Symptoms).

% Define the helper predicate subset/2 to check if a list of symptoms is a subset of another list

subset([], \_).

subset([Head|Tail], List) :-

member(Head, List),

subset(Tail, List).

Output:

disease(cold,X).

X=[fever, headache, sore\_throat]

29. Write a Prolog Program for forward Chaining. Incorporate required queries

**Aim:** To implement a Prolog program using forward chaining to derive conclusions from given facts and rules.

**Algorithm:**

1. Define initial facts and rules.
2. Create a forward-chaining rule that iterates through the available rules and facts to derive new conclusions.
3. Continue the forward-chaining process until no more conclusions can be derived.
4. Allow querying for specific conclusions or facts.

**Program:**

% Define facts

has\_fur(dog).

has\_feathers(bird).

lays\_eggs(bird).

has\_scales(fish).

% Define rules

mammal(X) :- has\_fur(X).

bird(X) :- has\_feathers(X), lays\_eggs(X).

reptile(X) :- has\_scales(X).

% Forward chaining rule

is\_animal(X) :- mammal(X).

is\_animal(X) :- bird(X).

is\_animal(X) :- reptile(X).

% Queries

% Query 1

?- is\_animal(dog).

% Output: true

% Query 2

?- is\_animal(bird).

% Output: true

% Query 3

?- is\_animal(fish).

% Output: true

**Output:**

Query 1.

True.

Query 2

True.

Query 3

True

30. Write a Prolog Program for backward Chaining. Incorporate required queries.

**Aim:** To implement a Prolog program using backward chaining to derive conclusions from given facts and rules.

**Algorithm:**

1. Define initial facts and rules.
2. Create a forward-chaining rule that iterates through the available rules and facts to derive new conclusions.
3. Continue the forward-chaining process until no more conclusions can be derived.
4. Allow querying for specific conclusions or facts.

Program:

% Define facts

has\_fur(dog).

has\_feathers(bird).

lays\_eggs(bird).

has\_scales(fish).

% Define rules

mammal(X) :- has\_fur(X).

bird(X) :- has\_feathers(X), lays\_eggs(X).

reptile(X) :- has\_scales(X).

% Backward chaining rule

is\_mammal(X) :- mammal(X).

% Queries

% Query 1

?- is\_mammal(dog).

% Output: true

% Query 2

?- is\_mammal(bird).

% Output: false

% Query 3

?- is\_mammal(fish).

% Output: false

Output:

Quary 1.

True

Quary 2.

False

Quary 3.

True

31. Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc

**Aim:** Create a web blog using WordPress to demonstrate the usage of HTML elements such as Anchor Tags, Title Tags, and other related concepts.

**Algorithm:**

1. **Setup WordPress:**
   * Install and configure WordPress on a web server or a local development environment.
2. **Choose a Theme:**
   * Select a WordPress theme that suits the style of your blog.
3. **Install and Activate Plugins:**
   * Install plugins that enhance your blog functionality. For example, you might need a plugin for SEO, social media integration, etc.
4. **Create Main Pages:**
   * Create main pages for your blog such as Home, About, Contact, etc.
5. **Create Blog Posts:**
   * Write blog posts using the WordPress editor. Include various HTML elements within your posts to demonstrate their usage.
6. **Demonstrate Anchor Tags:**
   * In your blog posts, use anchor tags (**<a>**) to link to other pages, external websites, or internal sections of your blog.

<a href="https://www.example.com">Visit Example</a>

7 **Title Tags:**

* Ensure each page and post has a clear and descriptive title using the **<title>** tag.

<title>Your Blog Post Title</title>

8. **Formatting and Styling:**

* Use HTML tags for formatting and styling your content. This might include headings (**<h1>**, **<h2>**, etc.), paragraphs (**<p>**), and other styling elements.
* **Categories and Tags:**
  + Organize your blog posts using categories and tags. This helps in better navigation and SEO.
* **Create Navigation Menu:**
  + Set up a navigation menu that includes links to your main pages and categories.
* **SEO Optimization:**
  + Use SEO plugins to optimize your blog for search engines. This includes setting meta descriptions, alt tags for images, and more.
* **Social Media Integration:**
  + Integrate your blog with social media platforms to share your content easily.
* **Testing:**
  + Test your blog on different browsers and devices to ensure a responsive design.
* **Launch:**
  + Once you are satisfied with the setup, launch your blog and share it with your audience.

This algorithm outlines the basic steps for creating a web blog using WordPress, emphasizing the demonstration of HTML elements such as Anchor Tags, Title Tags, and other formatting elements. Adjustments and additional steps may be necessary based on specific requirements and preferences.

**Program:**

|  |  |
| --- | --- |
|  | <!DOCTYPE html> |
|  | <html> |
|  | <head> |
|  | </head> |
|  | <body> |
|  | <center><form> |
|  | <h1><font color="red" size="10" face="impact">Student login page</font></h1><hr><hr> |
|  | <label style="padding-left: 16px;">LOGIN ID :-</label> |
|  | <input type="text"><br><br> |
|  | <label>PASSWORD :-</label> |
|  | <input type="password"><br><br> |
|  | <button>SUBMIT</button> |
|  | </form> |
|  | <a href="[file:///C:/Users/Ashrith%20Reddy/Desktop/OLX/home.html](file:///C:\Users\Ashrith%20Reddy\Desktop\OLX\home.html)">SUBMIT</a><br> |
|  | <a href="<https://arms.sse.saveetha.com/Login.aspx?s=exp>">Arms</a></a> |
|  | </center> |
|  | </body> |
|  | </html> |

**Output**:

