**Hindi Vidya Prachar Samiti’s**

**Ramniranjan Jhunjhunwala College of Arts, Science & Commerce**

**(Empowered Autonomous College)**

****

**Affiliated to**

**UNIVERSITY OF MUMBAI**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**2024-2025**

**M.Sc. (IT) part i SEM i**

**PAPER RJSPIT104P – Artificial Intelligence**

**Name :- Vishwakarma Shivam Suresh Sushila**

**Roll No. 6620**

**Hindi Vidya Prachar Samiti’s**

**RAMNIRANJAN JHUNJHUNWALA COLLEGE**

**Ghatkopar (W), Mumbai-400 086**

***Certificate***



**This is to certify that Mr./Ms. Vishwakarma Shivam Suresh Suhila Roll No 6620 of M.Sc.(I.T.) Part-1 class has completed the required number of experiments in the subject of Artificial Intelligence in the Department of Information Technology during the academic year 2024 - 2025 .**

**Professor In-Charge Co-ordinator of IT Department**

**Prof. Bharati Bhole**

**College Seal & Date Examiner**

#### **INDEX**

**Artificial Intelligence**

| **Sr. No** | **Practical** | **Date** |
| --- | --- | --- |
| 1 | **Implementation of following search algorithms for the given tree:**   1. Breadth First Search 2. Depth First Search | Jul 15, 2024 |
| 2 | **Implementation of Heuristic search algorithms:**   1. A\* Search 2. AO\* | Sep 16, 2024 |
| 3 | **Implementation of**   1. Tic-Tac-Toe game problem 2. Water-Jug problem | Jul 22, 2024 |
| 4 | **Implementation of**   1. Map/Region Coloring Problem in AI. 2. Construct a Maze Problem Solver | Jul 29, 2024 |
| 5 | Implementation of N-Queens Problem. | Sep 16, 2024 |
| 6 | Implementation of constraint satisfaction problems using Prolog. | Sep 23, 2024 |
| 7 | Implementation of logic programming using Prolog. | Aug 12, 2024 |
| 8 | Implementation of a fuzzy-based application using Python/R. | Sep 16, 2024 |

#### 

#### **Practical 1 - Implementation of following search algorithms**

Jul 15, 2024

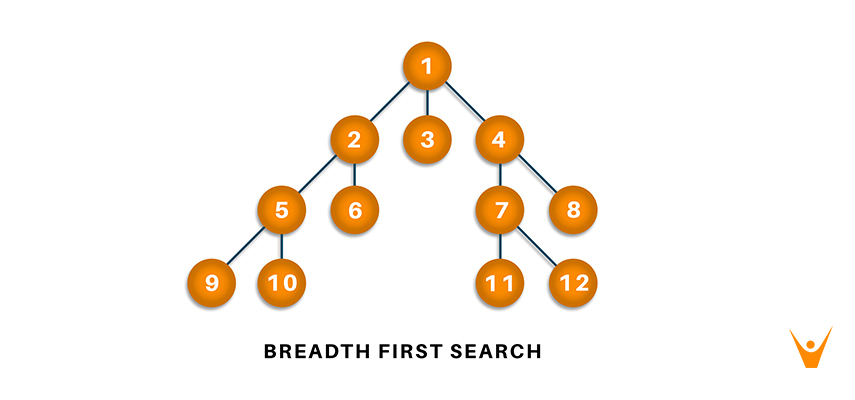
**Implementation of following search algorithms for the given tree:**

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

**Reference :**

1. <https://favtutor.com/blogs/breadth-first-search-python>
2. <https://www.youtube.com/watch?v=U5-bRX2AHNY>

Breadth-first search and Depth-first search in python are algorithms used to traverse a graph or a tree. Considering below example :



##### **A. Breadth First Search (BFS)**

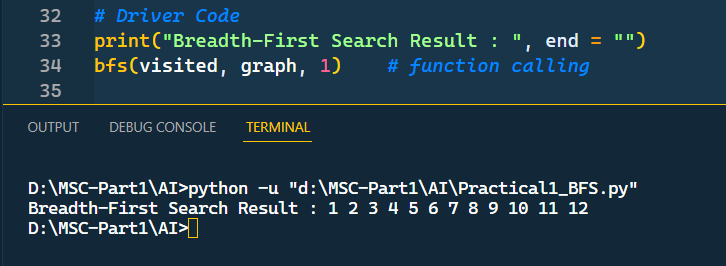
* Breadth-First Search is a recursive algorithm to search all the vertices of a graph or a tree.
* BFS in python can be implemented by using data structures like a dictionary and lists. Breadth-First Search in tree and graph is almost the same. Difference is that the graph may contain cycles, so we may traverse to the same node again.
* The steps of the **algorithm** work as follow:
  + Start by putting any one of the graph’s vertices at the back of the queue.
  + Now take the front item of the queue and add it to the visited list.
  + Create a list of that vertex's adjacent nodes. Add those which are not within the visited list to the rear of the queue.
  + Keep continuing steps two and three till the queue is empty.
* Complexity Analysis - **Time complexity is O(V+E)**, where V is the representation of the number of nodes and E is the number of edges. Also, the space complexity of the BFS algorithm is O(V).
* **Applications of BFS Algorithm**
  + In GPS navigation, it helps in finding the shortest path available from one point to another.
  + In pathfinding algorithms
  + Cycle detection in an undirected graph
  + In minimum spanning tree
  + To build index by search index
  + In Ford-Fulkerson algorithm to find maximum flow in a network.

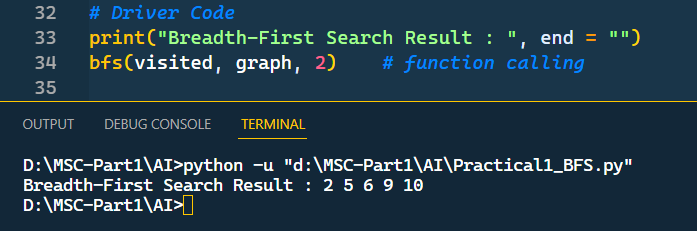
**code.py**

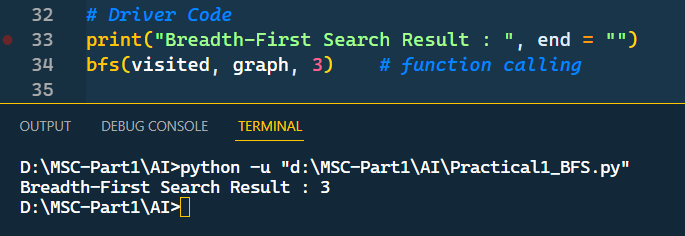
| visited = [] # List for visited nodes.  queue = [] # Initialize a queue  def bfs(visited, graph, node): # function for BFS  visited.append(node)  queue.append(node)  while queue: # Creating loop to visit each node  m = queue.pop(0)  print (m, end = " ")  for neighbour in graph[m]:  if neighbour not in visited:  visited.append(neighbour)  queue.append(neighbour)  graph = {  1: [2, 3, 4],  2: [5, 6],  3: [],  4: [7, 8],  5: [9, 10],  6: [],  7: [11, 12],  8: [],  9: [],  10: [],  11: [],  12: []  }  # Driver Code  print("Breadth-First Search Result : ", end = "")  bfs(visited, graph, 1) # function calling |
| --- |

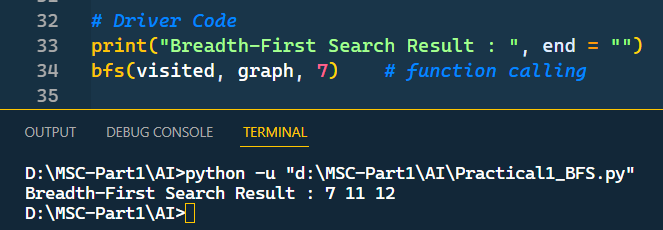


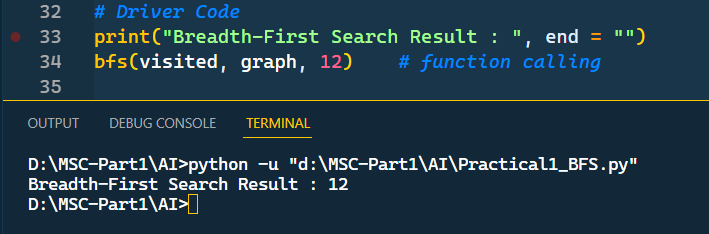
**Output**









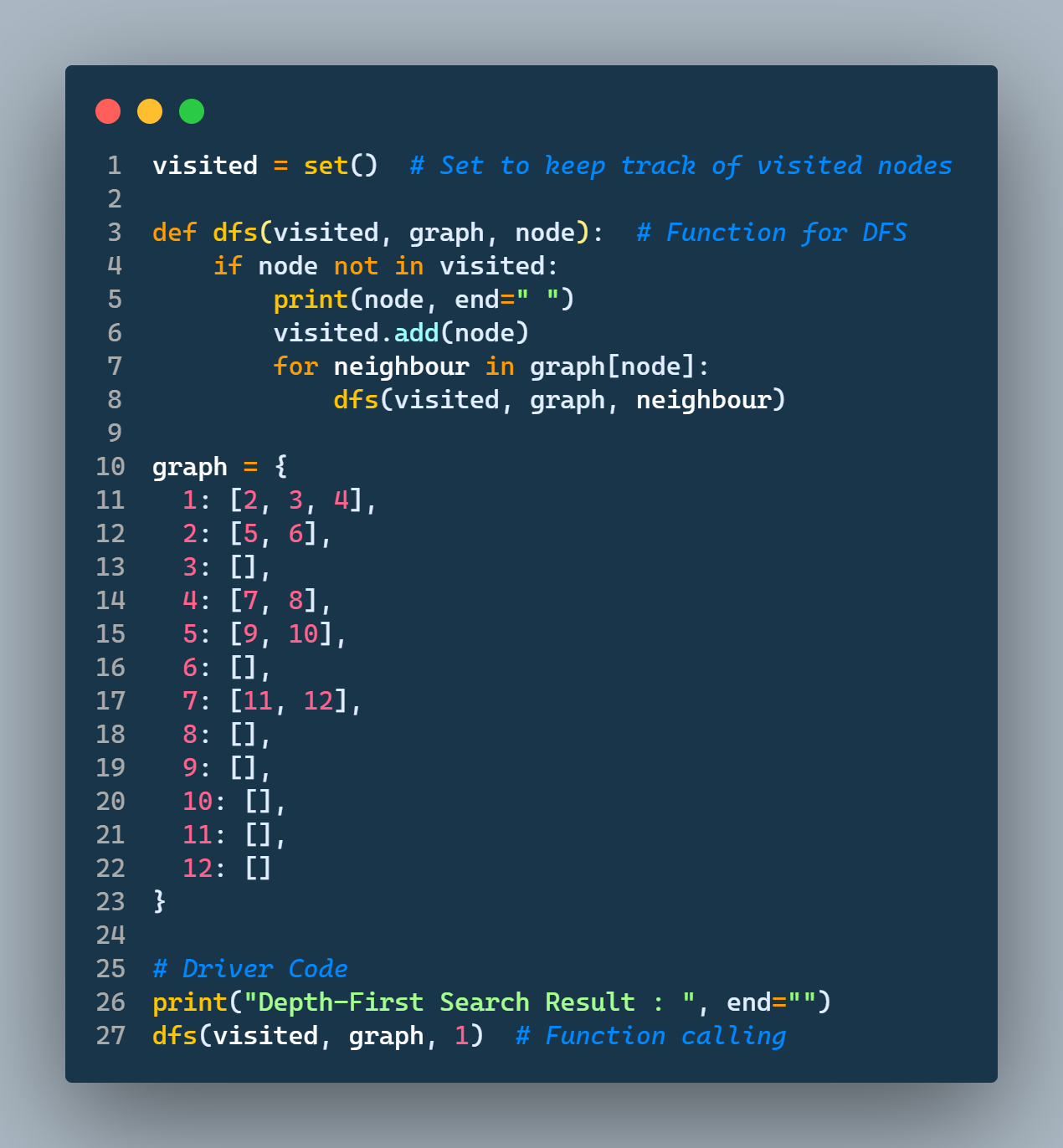


##### **B. Depth First Search (DFS)**

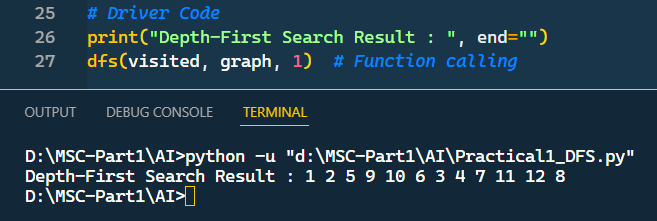
* Depth First Search (DFS) algorithm is a recursive algorithm for searching all the vertices of a graph or tree data structure.
* This algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
* Algorithm
  + Initialize an empty stack for storage of nodes, S.
  + For each vertex u, define u.visited to be false.
  + Push the root (first node to be visited) onto S.
  + While S is not empty:
  + Pop the first element in S, u.
  + If u.visited = false, then:
  + U.visited = true
  + for each unvisited neighbor w of u:
  + Push w into S.
  + End process when all nodes have been visited.

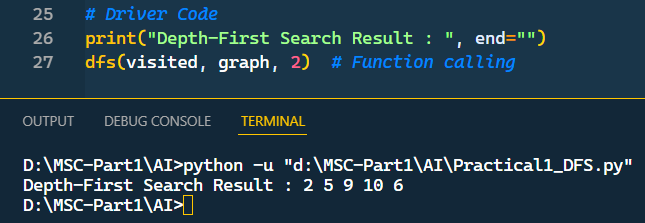
**code.py**

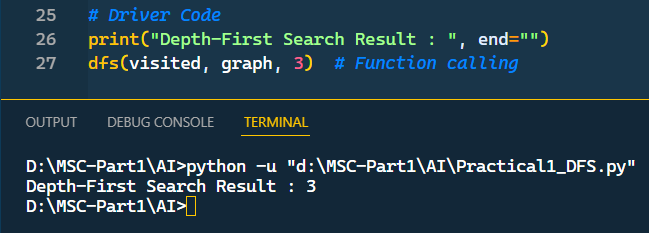
| visited = set() # Set to keep track of visited nodes  def dfs(visited, graph, node): # Function for DFS  if node not in visited:  print(node, end=" ")  visited.add(node)  for neighbour in graph[node]:  dfs(visited, graph, neighbour)  graph = {  1: [2, 3, 4],  2: [5, 6],  3: [],  4: [7, 8],  5: [9, 10],  6: [],  7: [11, 12],  8: [],  9: [],  10: [],  11: [],  12: []  }  # Driver Code  print("Following is the Depth-First Search")  dfs(visited, graph, '1') # Function calling |
| --- |

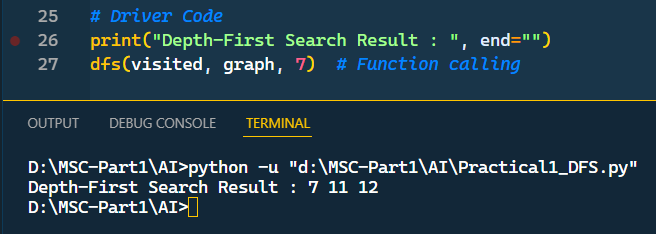


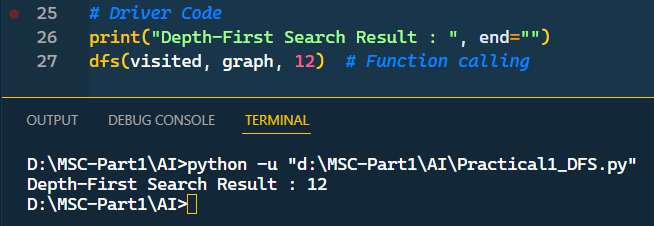
**Output**











#### 

#### **Practical 2 - Implementation of Heuristic search algorithms**

Jul 22, 2024

**Implementation of Heuristic search algorithms:**

1. A\* Search
2. AO\*

**Reference :**

1. <https://www.geeksforgeeks.org/ao-algorithm-artificial-intelligence/>
2. <https://docs.google.com/presentation/d/1H4719x9pPx2hvE1KkNSC01cu7Jom4pXYEPJtcglaTXY/edit#slide=id.p>

**What are Heuristic search algorithms ?**

* This search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution.
* Such techniques help in finding a solution within reasonable time and space (memory).

##### **A. A\* Search (A Star Search)**

* The most widely known form of best-first search is called A∗ Search
* It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

***f(n) = g(n) + h(n) .***

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

**f(n) = estimated cost of the cheapest solution through n .**

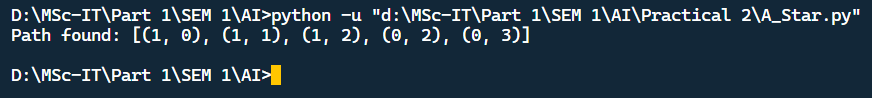
* A∗ search is both complete and optimal.

**code.py**

| import heapq  import math  def heuristic\_distance(point1, point2):  #using manhaten distance  #return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])  #using eucledian distance  return math.sqrt((point1[0] - point2[0]) \*\* 2 + (point1[1] - point2[1]) \*\* 2)  def a\_star(grid, start, goal):  open\_list = [(0, start)]  came\_from = {}  g\_score = {node: float('inf') for node in grid}  g\_score[start] = 0  while open\_list:  \_, current = heapq.heappop(open\_list)  if current == goal:  path = []  while current in came\_from:  path.insert(0, current)  current = came\_from[current]  path.insert(0, start)  return path  for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0)]:  neighbor = current[0] + dx, current[1] + dy  if neighbor in grid:  tentative\_g\_score = g\_score[current] + 1  if tentative\_g\_score < g\_score[neighbor]:  came\_from[neighbor] = current  g\_score[neighbor] = tentative\_g\_score  heapq.heappush(open\_list, (g\_score[neighbor] + heuristic\_distance(neighbor, goal), neighbor))  return None # No path found  # Example grid (dict with (x, y) as keys)  grid = {  (0, 0), (0, 1), (0, 2),(0,3),  (1, 0), (1, 1), (1, 2),(1,3),  (2, 0), (2, 1), (2, 2),(2,3)  }  start = (1, 0)  goal = (0, 3)  path = a\_star(grid, start, goal)  if path:  print("Path found:", path)  else:  print("No path found.") |
| --- |



**Output**



##### **B. AO\* (AND-OR Star Search)**

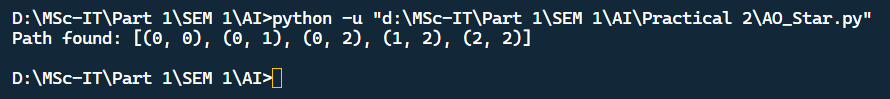
* AO Search\* is an extension of the A\* algorithm
* The AO\* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept.
* The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.

**code.py**

| import heapq  def ao\_star(start, goal, heuristic, neighbors):  open\_list = [(0, start)]  # Priority queue initialized with the start node  g\_costs = {start: 0}  # Dictionary to store the cost of the shortest path to each node  came\_from = {}  # Dictionary to reconstruct the path  while open\_list:  \_, current = heapq.heappop(open\_list)    if current == goal:  path = [current]  while current in came\_from:  current = came\_from[current]  path.append(current)  return list(reversed(path))  # Return the path from start to goal    for neighbor in neighbors(current):  tentative\_g = g\_costs[current] + 1  # Assuming uniform cost for each step  if tentative\_g < g\_costs.get(neighbor, float('inf')):  g\_costs[neighbor] = tentative\_g  f\_cost = tentative\_g + heuristic(neighbor, goal)  came\_from[neighbor] = current  heapq.heappush(open\_list, (f\_cost, neighbor))    return None  # Return None if no path is found  # Example usage:  def heuristic(point1, point2):  # Manhattan distance heuristic for grid-based pathfinding  return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])  def neighbors(node):  # Example neighbor function for a grid (4-connected)  x, y = node  return [(x + dx, y + dy) for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]]  # Example grid and start/goal points  start = (0, 0)  goal = (2, 2)  path = ao\_star(start, goal, heuristic, neighbors)  print("Path found:", path) |
| --- |



**Output**



#### **Practical 3 - Implementation the following**

Jul 22, 2024

**Implementation of**

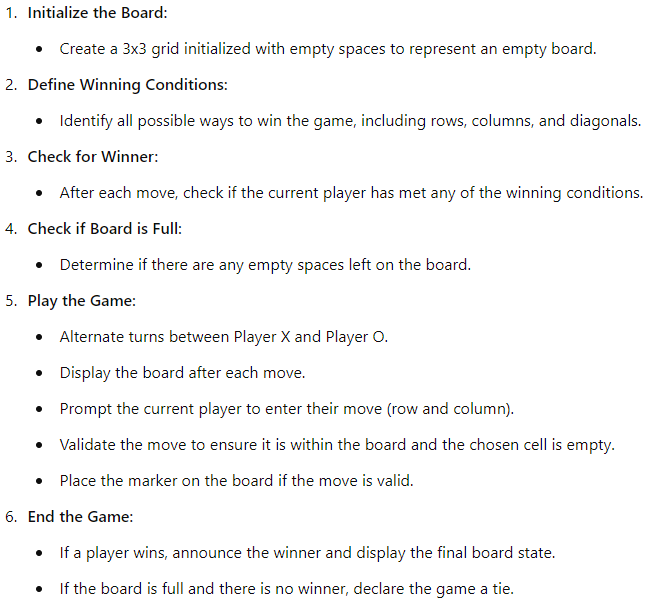
1. Tic-Tac-Toe game problem
2. Water-Jug problem

##### **A. Tic-Tac-Toe game problem**

**Objective**

* Implement a Tic-Tac-Toe game where two players take turns placing their markers on a 3x3 grid.
* Determine the winner or if the game ends in a tie.
* Handle user input and validate moves.

**Algorithm**



**code.py**

| # Function to print the Tic Tac Toe board  def print\_board(board):  for row in board:  print(" | ".join(row)) # Join elements of each row with ' | ' and print  print("-" \* 9) # Print a horizontal line as a separator  # Function to check if a player has won  def check\_winner(board, player):  # Define all possible winning combinations on the board  win\_conditions = [  [board[0][0], board[0][1], board[0][2]], # Top row  [board[1][0], board[1][1], board[1][2]], # Middle row  [board[2][0], board[2][1], board[2][2]], # Bottom row  [board[0][0], board[1][0], board[2][0]], # Left column  [board[0][1], board[1][1], board[2][1]], # Middle column  [board[0][2], board[1][2], board[2][2]], # Right column  [board[0][0], board[1][1], board[2][2]], # Diagonal from top-left to bottom-right  [board[0][2], board[1][1], board[2][0]] # Diagonal from top-right to bottom-left  ]  # Check if any of the win conditions are fulfilled by the player  return [player, player, player] in win\_conditions  # Function to check if the board is full  def is\_board\_full(board):  # Return True if there are no empty spaces (' ') left on the board  return all(cell != ' ' for row in board for cell in row)  def tic\_tac\_toe():  board = [[' ']\*3 for i in range(3)] # Initialize the board with empty spaces  players = ['X', 'O'] # Define the players ('X' goes first, 'O' goes second)  turn = 0 # Initialize turn counter  # Continue the game until there's a winner or the board is full  while not (check\_winner(board, 'X') or check\_winner(board, 'O')) and not is\_board\_full(board):  # Print the current state of the board  print\_board(board)  # Determine whose turn it is based on the turn counter  current\_player = players[turn % 2]  print(f"Player {current\_player}'s turn.")  # Prompt the current player to input their move  while True:  try:  row, col = map(int, input("Enter row and column numbers (e.g., 0 0): ").split())  # Check if the chosen position is empty (' ')  if board[row][col] == ' ':  break # Valid move, exit the loop  else:  print("That position is already taken! Try again.")  except (ValueError, IndexError):  print("Invalid input! Please enter valid row and column numbers.")  # Place the current player's marker ('X' or 'O') on the board  board[row][col] = current\_player  turn += 1 # Increment the turn counter  print\_board(board) # Game ended, print the final state of the board  # Check and print the result of the game  if check\_winner(board, 'X'):  print("Player X wins!")  elif check\_winner(board, 'O'):  print("Player O wins!")  else:  print("It's a tie!")  if \_\_name\_\_ == "\_\_main\_\_":  tic\_tac\_toe() |
| --- |





**Output 1**

D:\MSc-IT\Part 1\AI>python -u "d:\MSc-IT\Part 1\AI\Practical 3\TicTacToe.py"

| |

---------

| |

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 0 0

X | |

---------

| |

---------

| |

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 1 1

X | |

---------

| O |

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 2 2

X | |

---------

| O |

---------

| | X

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 0 1

X | O |

---------

| O |

---------

| | X

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 2 1

X | O |

---------

| O |

---------

| X | X

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 2 0

X | O |

---------

| O |

---------

O | X | X

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 0 2

X | O | X

---------

| O |

---------

O | X | X

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 1 2

X | O | X

---------

| O | O

---------

O | X | X

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 1 0

X | O | X

---------

X | O | O

---------

O | X | X

---------

It's a tie!

D:\MSc-IT\Part 1\AI>

**Output 2**

D:\MSc-IT\Part 1\AI>python -u "d:\MSc-IT\Part 1\AI\Practical 3\TicTacToe.py"

| |

---------

| |

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 2 2

| |

---------

| |

---------

| | X

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 2 1

| |

---------

| |

---------

| O | X

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 1 2

| |

---------

| | X

---------

| O | X

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 1 1

| |

---------

| O | X

---------

| O | X

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 0 2

| | X

---------

| O | X

---------

| O | X

---------

Player X wins!

D:\MSc-IT\Part 1\AI>

**Output 3**

D:\MSc-IT\Part 1\AI>python -u "d:\MSc-IT\Part 1\AI\Practical 3\TicTacToe.py"

| |

---------

| |

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 1 2

| |

---------

| | X

---------

| |

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 0 0

O | |

---------

| | X

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 1 1

O | |

---------

| X | X

---------

| |

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 1 0

O | |

---------

O | X | X

---------

| |

---------

Player X's turn.

Enter row and column numbers (e.g., 0 0): 0 1

O | X |

---------

O | X | X

---------

| |

---------

Player O's turn.

Enter row and column numbers (e.g., 0 0): 2 0

O | X |

---------

O | X | X

---------

O | |

---------

Player O wins!

D:\MSc-IT\Part 1\AI>

##### **B. Water-Jug problem**

**Given:**

Two jugs with capacities **jug1** and **jug2** (denoted as **cap1** and **cap2**).

An integer **target** that represents the amount of water you want to measure.

**Objective:**

Determine whether it is possible to measure exactly **target** units of water using these two jugs, and if possible, describe a sequence of steps to achieve this measurement.

**Jug Operations:**

* You can fill either jug completely to its capacity.
* You can empty either jug completely.
* You can pour water from one jug into the other until the first jug is empty or the second jug is full.

**Initial State:**

Both jugs are initially empty.

**Goal:**

To measure exactly **target** units of water in either of the jugs.

**Example:**

cap1 = 4 (capacity of the first jug)

cap2 = 3 (capacity of the second jug)

target = 2 (the amount of water to measure)

**defaultdict:** A defaultdict is a dictionary that returns a default value if you try to access a key that doesn’t exist.

**Example :**

visited = defaultdict(lambda: False)

visited is initialized as a defaultdict with a default value of False for any key that doesn’t exist. This is done using lambda: False.

**Algorithm for Water Jug Problem**

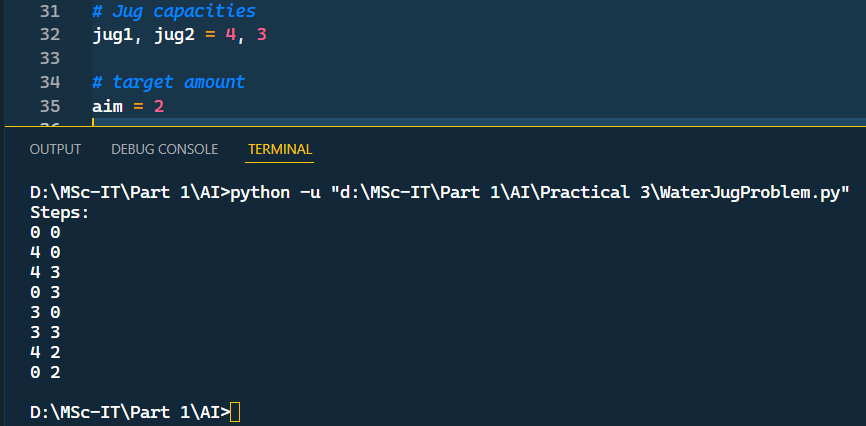


**code.py**

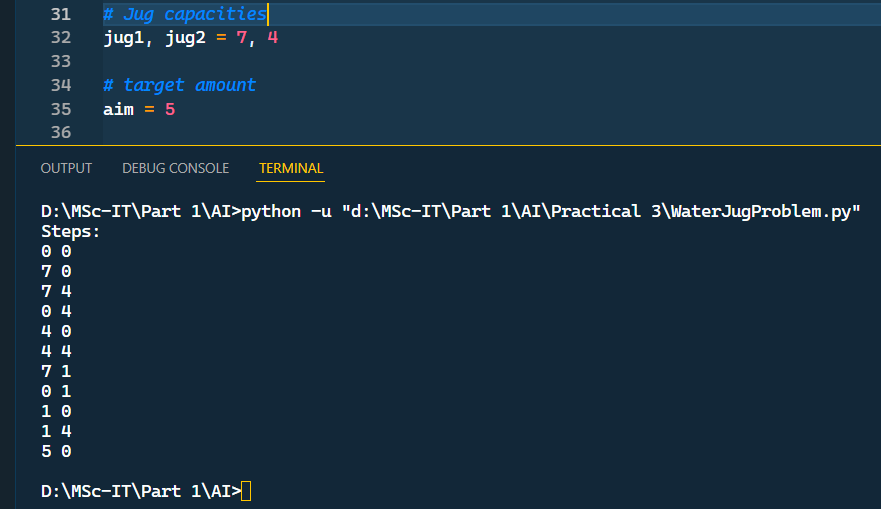
| from collections import defaultdict  # Jug capacities  jug1, jug2 = 4, 3  # target amount  aim = 2  visited = defaultdict(lambda: False)  def waterJugSolver(amt1, amt2):  # Check if we've reached the target amount in either jug with the other empty  if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):  print(amt1, amt2)  return True  # If this state has not been visited yet  if not visited[(amt1, amt2)]:  print(amt1, amt2) # Print the current state  visited[(amt1, amt2)] = True    # Recursively try all possible actions  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 from jug2 to jug1  waterJugSolver(amt1 - min(amt1, jug2 - amt2),  amt2 + min(amt1, jug2 - amt2)) # Pour from jug1 to jug2  )  else:  return False  print("Steps: ")  waterJugSolver(0, 0) |
| --- |



**Output**



For Jugs of capacity of 7 and 4 and aim is 5



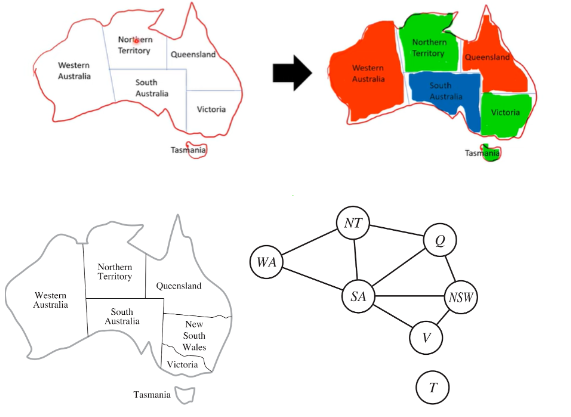
#### **Practical 4 - Implementation the following**

Jul 29, 2024

**Implementation of**

1. Map/Region Coloring Problem in AI.
2. Construct a Maze Problem Solver

##### **A. Map/Region Coloring Problem in AI.**



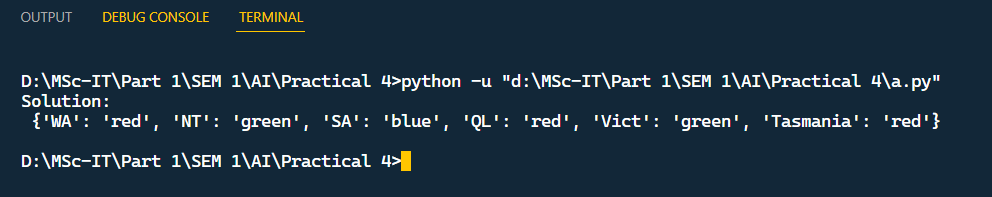
Pip install simpleai

**code.py**

| from simpleai.search import CspProblem, backtrack  # Define the constraint function  def different\_colors\_constraint(names, values):  return values[0] != values[1]  # Define the problem  def main():  # Define variables  variables = ['WA', 'NT','SA', 'QL','Vict', 'Tasmania']    # Define domains (color choices)  domains = {  'WA': ['red', 'green', 'blue'],  'NT': ['red', 'green', 'blue'],  'QL': ['red', 'green', 'blue'],  'Vict': ['red', 'green', 'blue'],  'Tasmania': ['red', 'green', 'blue'],  'SA': ['red', 'green', 'blue']  }    # Define constraints  constraints = [  (('WA', 'NT'), different\_colors\_constraint),  (('WA', 'SA'), different\_colors\_constraint),  (('SA', 'QL'), different\_colors\_constraint),  (('SA', 'NT'), different\_colors\_constraint),  (('SA', 'Vict'), different\_colors\_constraint),  (('QL', 'NT'), different\_colors\_constraint),  (('QL', 'Vict'), different\_colors\_constraint),  (('Tasmania'), different\_colors\_constraint)  ]    # Create CSP problem  problem = CspProblem(variables, domains, constraints)    # Solve the problem  solution = backtrack(problem)    print("Solution:\n", solution)  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |



**Output**



##### **B. Construct a Maze Problem Solver**

**What is a Maze Problem?**

A Maze Problem refers to an algorithm or system designed to find a path from a start point to an end point within a maze, connected in a complex pattern. The challenge lies in determining an optimal route, avoiding dead ends, and navigating through obstacles.   
  
**Goal:** To efficiently move from the starting point to the destination, solving the maze.

**Challenges:**

Exploration: differentiate between valid paths and dead ends.

Efficiency: minimize the time to reach the goal.

Optimality: finding the shortest path.

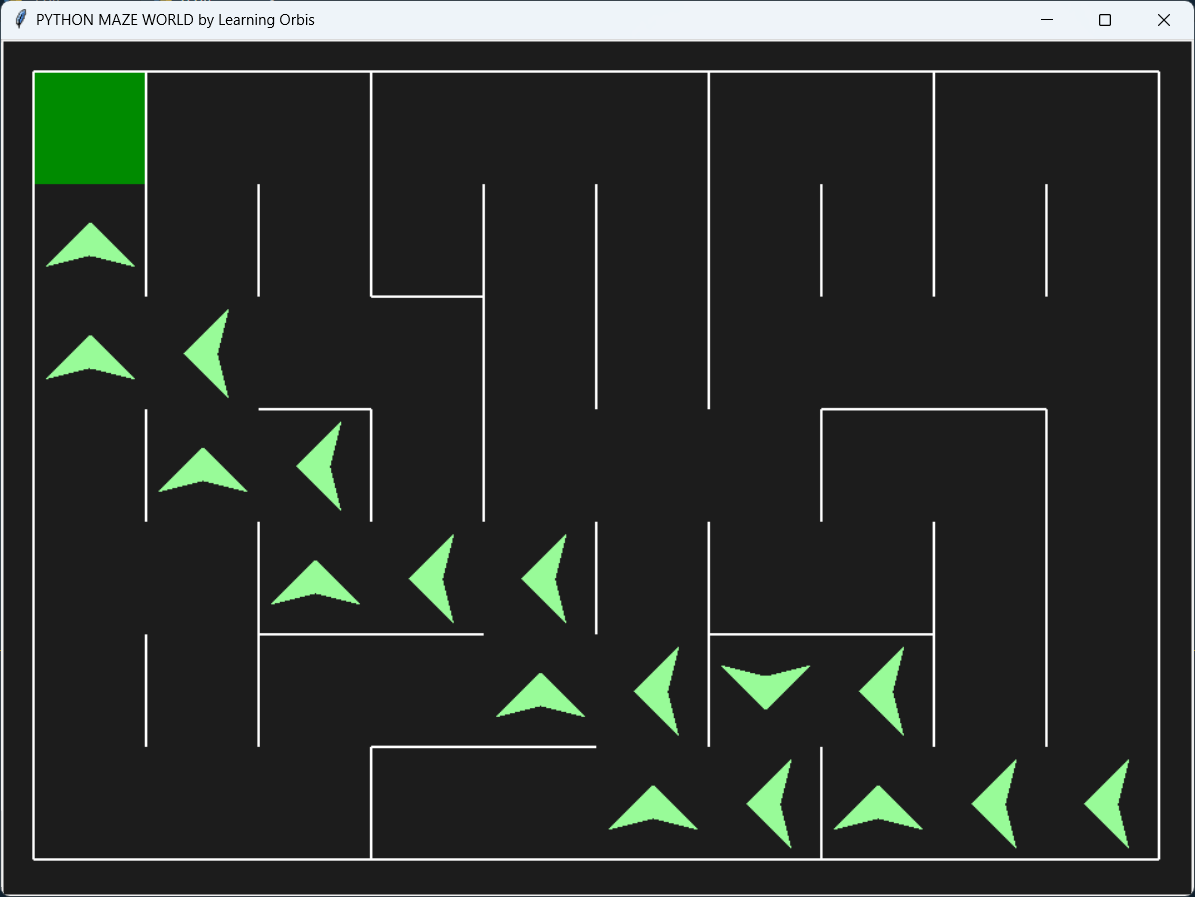
Handling Loops: avoid revisiting nodes unnecessarily

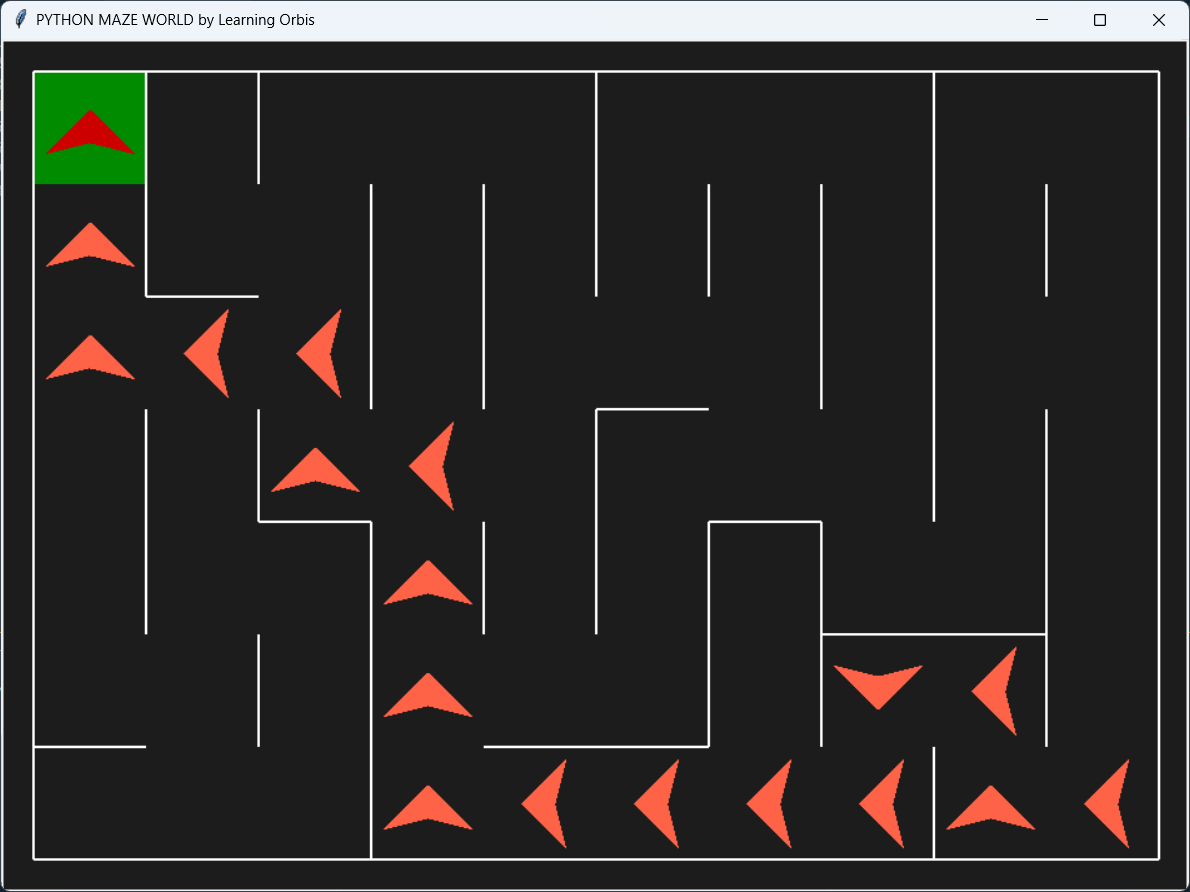
**code.py**

| from pyamaze import maze,agent,COLOR  m=maze(7,10)  m.CreateMaze( pattern='v', loopPercent=40, theme=COLOR.dark)  a=agent(m,filled=True,shape = 'arrow', footprints=True, color='green')  m.tracePath({a:m.path})  m.run() |
| --- |



**Output**

****

****

#### **Practical 5 - Implementation of N-Queens Problem.**

Sep 16, 2024

**What is the N-Queen Problem ?**

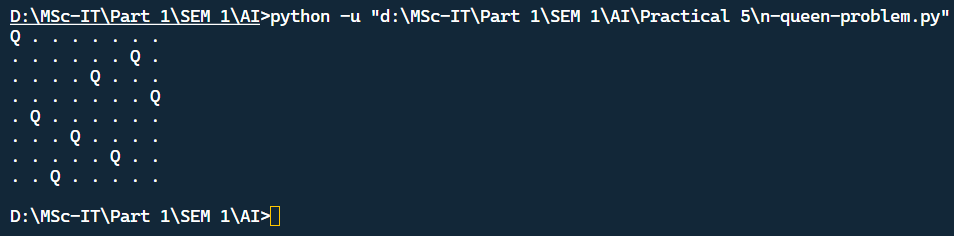
* The N-Queens problem is a combinatorial optimization problem in artificial intelligence (AI). It involves placing N queens on an N x N chessboard such that no two queens threaten each other.
* This means: No two queens can be placed in the same row, column, or diagonal.
* The challenge is to find one or more valid configurations of placing N queens on the board where this condition holds.

**code.py**

| def is\_safe(board, row, col):  # Check the left side of the current row  for i in range(col):  if board[row][i] == 1:  return False  # Check upper diagonal on the left  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  if board[i][j] == 1:  return False  # Check lower diagonal on the left  for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):  if board[i][j] == 1:  return False  return True  def solve\_n\_queens(board, col):  if col >= len(board):  return True  for i in range(len(board)):  if is\_safe(board, i, col):  board[i][col] = 1  if solve\_n\_queens(board, col + 1):  return True  board[i][col] = 0  return False  def n\_queens(n):  board = [[0] \* n for \_ in range(n)]  if not solve\_n\_queens(board, 0):  print("No solution found.")  return  for row in board:  print(" ".join(["Q" if cell == 1 else "." for cell in row]))  if \_\_name\_\_ == "\_\_main\_\_":  n = 8 # Change this to the desired board size  n\_queens(n) |
| --- |

****

**Output**

****

#### **Practical 6 - Implementation of constraint satisfaction problems using Prolog.**

Sep 23, 2024

**Download prolong →** <http://www.gprolog.org/#download>

**Tutorial** → <https://www.youtube.com/watch?v=SykxWpFwMGs&t=543s>

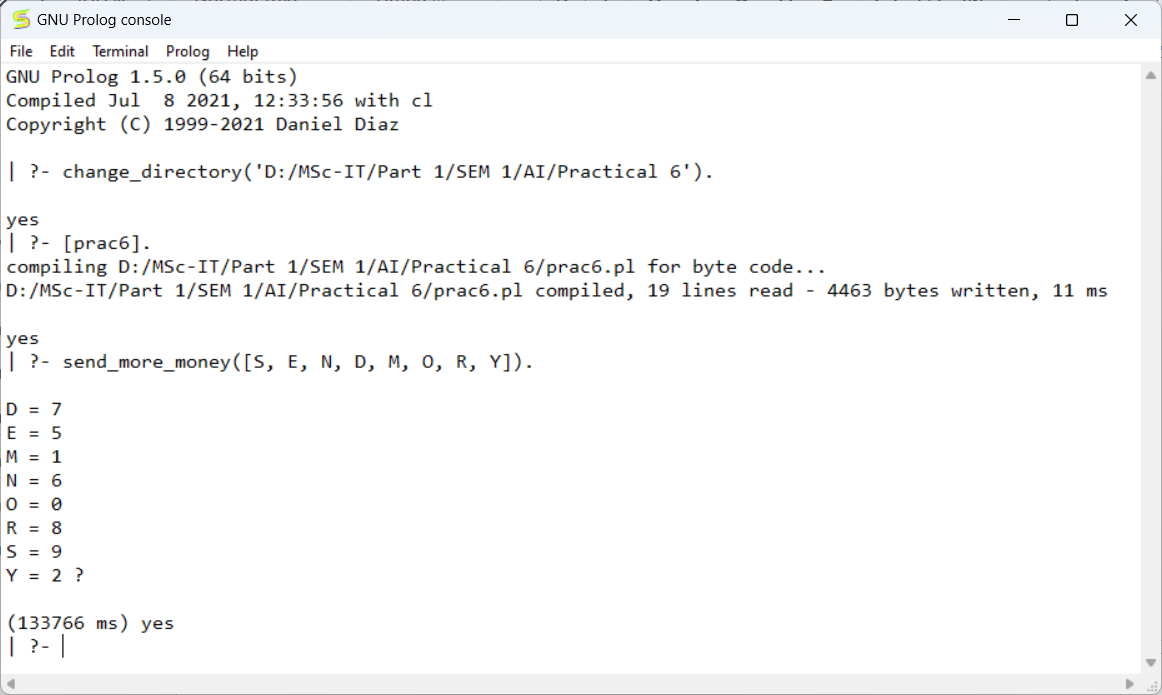
**Rules of prolog :**

* File extension should be **name.pl**
* Statement terminated using . (period operator / full stop)
* To compile a file give cmd like **[filename].**
* **:- (implies)**
* % can be used for single line comment

**code.pl**

| % Define the constraint that ensures no two variables have the same digit.  all\_different([]).  all\_different([H|T]) :- \+ member(H, T), all\_different(T).  % Define the main predicate for solving the puzzle.  send\_more\_money([S, E, N, D, M, O, R, Y]) :-  Digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  member(S, Digits), S > 0,  member(E, Digits),  member(N, Digits),  member(D, Digits),  member(M, Digits), M > 0,  member(O, Digits),  member(R, Digits),  member(Y, Digits),  all\_different([S, E, N, D, M, O, R, Y]),  1000 \* S + 100 \* E + 10 \* N + D +  1000 \* M + 100 \* O + 10 \* R + E =:=  10000 \* M + 1000 \* O + 100 \* N + 10 \* E + Y. |
| --- |

**Output**



#### **Practical 7 - Implementation of logic programming using Prolog.**

Aug 12, 2024

**Implementation of logic programming using Prolog.**

Define apple, orange, banana, grapes etc… as fruits

Eg- fruits(apple).

Define tomato, chilli, potato, capsicum etc… as veg

Eg- veg(tomato).

Define some fruits as sweet and some fruits as sour.

Eg- sweet(apple).

Eg- sour(grapes).

Write two rules for stating ‘I like sweet fruits’ and ‘i don't like sour fruits’

Eg. like(X) :- fruit(X), sweet(X)

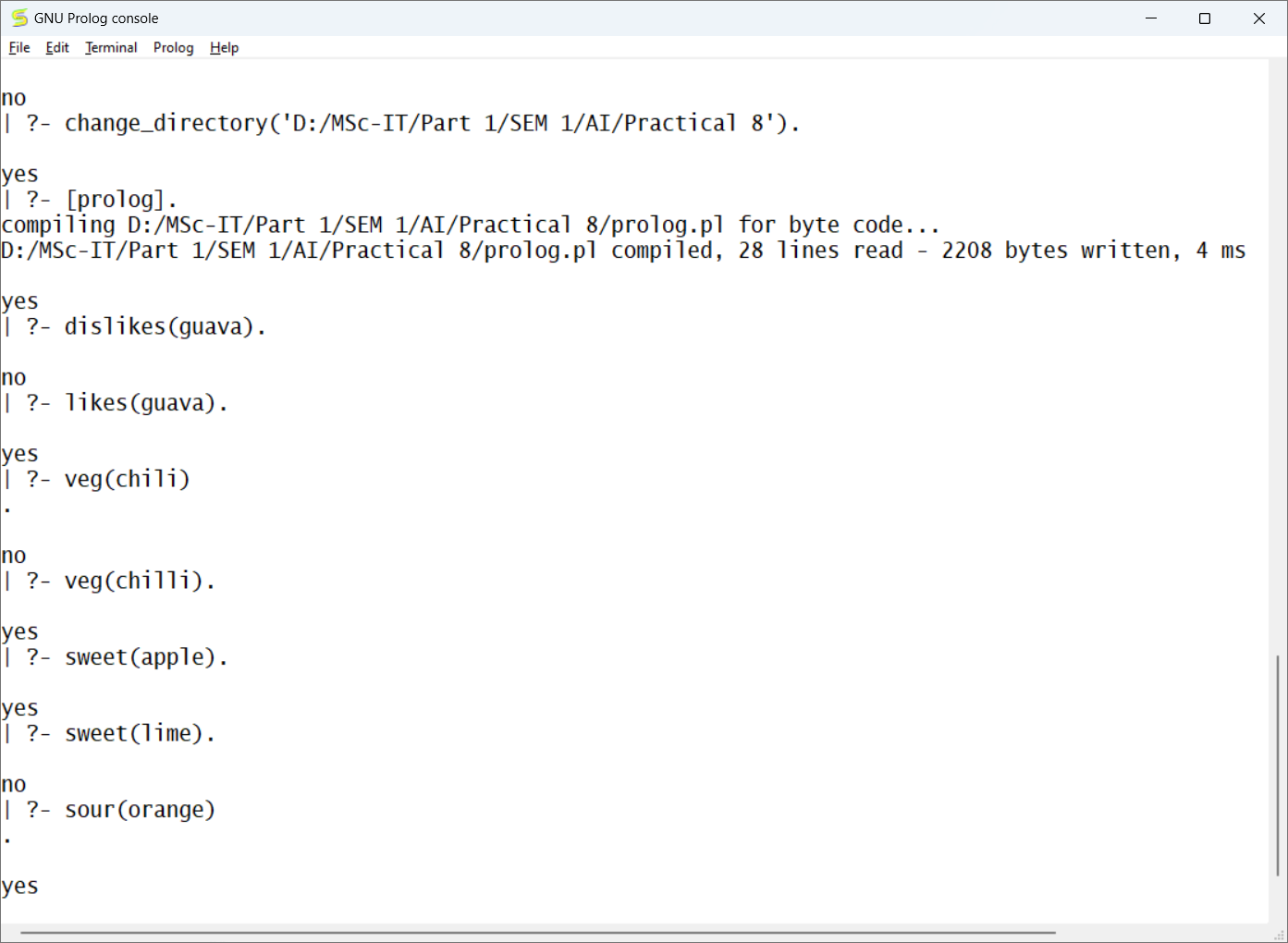
dont\_like(X):-fruit(X), sour(X)

Query- which fruit you like or don't like??

**code.pl**

| fruits(apple).  fruits(kiwi).  fruits(pomegranate).  fruits(strawberry).  fruits(orange).  fruits(grapes).  fruits(bananas).  fruits(guava).  sweet(guava).  sweet(kiwi).  sweet(pomegranate).  sweet(strawberry).  sweet(banana).  sweet(apple).  sour(grapes).  sour(orange).  veg(tomato).  veg(chilli).  veg(potato).  veg(capsicum).  likes(X) :- fruits(X),sweet(X).  dislikes(Y) :- fruits(Y),sour(Y). |
| --- |

**Output**



#### **Practical 8 - Implementation of a fuzzy-based application using Python / R.**

Sep 16, 2024

##### **Example 1 - Fan speed**

**code.py**

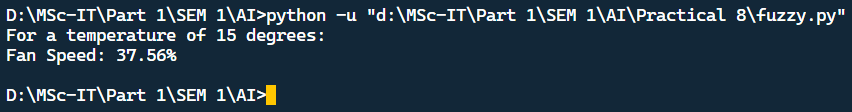
| import numpy as np  import skfuzzy as fuzz  from skfuzzy import control as ctrl  # Create input and output variables  temperature = ctrl.Antecedent(np.arange(0, 101, 1), 'temperature')  fan\_speed = ctrl.Consequent(np.arange(0, 101, 1), 'fan\_speed')  # Define fuzzy sets for temperature  temperature['cold'] = fuzz.trimf(temperature.universe, [0, 0, 50])  temperature['warm'] = fuzz.trimf(temperature.universe, [0, 50, 100])  temperature['hot'] = fuzz.trimf(temperature.universe, [50, 100, 100])  # Define fuzzy sets for fan\_speed  fan\_speed['low'] = fuzz.trimf(fan\_speed.universe, [0, 0, 50])  fan\_speed['medium'] = fuzz.trimf(fan\_speed.universe, [0, 50, 100])  fan\_speed['high'] = fuzz.trimf(fan\_speed.universe, [50, 100, 100])  # Define rules  rule1 = ctrl.Rule(temperature['cold'], fan\_speed['low'])  rule2 = ctrl.Rule(temperature['warm'], fan\_speed['medium'])  rule3 = ctrl.Rule(temperature['hot'], fan\_speed['high'])  # Create the control system  fan\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])  # Create a simulation  fan\_sim = ctrl.ControlSystemSimulation(fan\_ctrl)  # Input temperature value  temperature\_input = 15  # Set the input temperature  fan\_sim.input['temperature'] = temperature\_input  # Compute the fan speed  fan\_sim.compute()  # Get the fan speed value  fan\_speed\_output = fan\_sim.output['fan\_speed']  print(f"For a temperature of {temperature\_input} degrees:")  print(f"Fan Speed: {fan\_speed\_output:.2f}%") |
| --- |

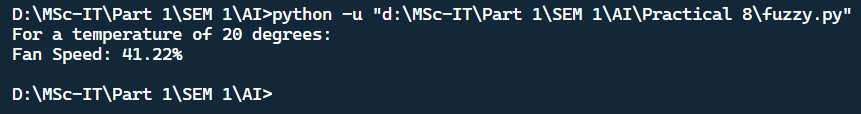


**Install the below following modules**

* pip install skfuzzy / **pip install -U scikit-fuzzy**
* Pip install scipy
* pip install networkx

**Output**





##### **Example 2 - Tip amount**

**code.py**

| import numpy as np  import skfuzzy as fuzz  from skfuzzy import control as ctrl  import matplotlib.pyplot as plt  # Define the variables  quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')  service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')  tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')  # Auto membership functions  quality.automf(3)  service.automf(3)  # Tip membership functions  tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])  tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])  tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])  # Define fuzzy rules  rules = [  ctrl.Rule(quality['good'] & service['good'], tip['high']),  ctrl.Rule(quality['good'] & service['average'], tip['medium']),  ctrl.Rule(quality['average'] & service['good'], tip['medium']),  ctrl.Rule(quality['average'] & service['average'], tip['medium']),  ctrl.Rule(quality['poor'] & service['poor'], tip['low']),  ctrl.Rule(quality['poor'] & service['average'], tip['low']),  ctrl.Rule(quality['average'] & service['poor'], tip['low']),  ]  # Create control system  tip\_ctrl = ctrl.ControlSystem(rules)  tip\_simulation = ctrl.ControlSystemSimulation(tip\_ctrl)  # Example input  tip\_simulation.input['quality'] = int(input("Rate (1-10) for Quality of food: "))  tip\_simulation.input['service'] = int(input("Rate (1-10) for Service of waiter: "))  # Compute the tip  tip\_simulation.compute()  # Output  print(f'Tip amount: {tip\_simulation.output["tip"]}')  tip.view(sim=tip\_simulation)  plt.show() |
| --- |



**Output**

