Principles of Artificial Intelligence

Course Code: MEAD-605

Lab File

Submitted to: **Dr. Kalpana Johari**

Submitted by

**Shantanu Shukla**

**Enrollment No. - 01211805424**

in partial fulfillment for the award of the degree

of

*Master of Technology - AI/DS*



Center for Development of Advanced Computing, Noida

*Affiliated to Guru Gobind Singh Indraprastha University*

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Title** | **Page No.** | **Signature** |
| 1. | Generate image frames from videos | 3 |  |
| 2. | Build a Grievance Portal | 5 |  |
| 3. | Generate all possible states for tic-tac-toe game | 11 |  |
| 4. | Implement A\* search algorithm | 13 |  |
| 5. | Implement AO\* search algorithm | 15 |  |
| 6. | Implement BFS algorithm | 17 |  |
| 7. | Implement DFS algorithm | 19 |  |
| 8. | Implement N Queens problem | 21 |  |

* **ASSIGNMENT-1**

**Objective:** Extraction of image frames from a video.

**Code:**

1. import cv2

2. import os

3.

4. # Set the path to the folder where you want to save the frames

5. output\_folder = 'saved\_frames\_3'

6.

7. # Create the folder if it doesn't exist

8. if not os.path.exists(output\_folder):

9.     os.makedirs(output\_folder)

10.

11. # Open the video file

12. video\_path = '3.mp4'  # Replace with your video path

13. cap = cv2.VideoCapture(video\_path)

14.

15. frame\_number = 0

16. saved\_frame\_count = 0

17.

18. # Loop through the video

19. while cap.isOpened():

20.     ret, frame = cap.read()

21.

22.     if not ret:

23.         break

24.

25.     # Check if the frame is the 10th one

26.     if frame\_number % 10 == 0:

27.         frame\_name = os.path.join(output\_folder, f'frame\_{saved\_frame\_count}.jpg')  # Name the saved frame

28.         cv2.imwrite(frame\_name, frame)  # Save the frame

29.         print(f"Saved : {saved\_frame\_count}")

30.         saved\_frame\_count += 1

31.

32.     frame\_number += 1

33.

34. # Release the video capture object

35. cap.release()

36. cv2.destroyAllWindows()

37.

38. print(f"Saved {saved\_frame\_count} frames to '{output\_folder}' folder.")

39.

**Output:**

A screenshot of a computer

Description automatically generated

* **ASSIGNMENT-2**

**Objective:** Build a Grievance Portal

**Project Structure:**

A screenshot of a computer

Description automatically generated

**Sample Code:**

**Backend –**

**models.py –** contains logic for our database models.

1. from flask\_sqlalchemy import SQLAlchemy

2. from flask\_login import UserMixin

3.

4. db = SQLAlchemy()

5.

6.

7. class User(db.Model, UserMixin):

8. id = db.Column(db.Integer, primary\_key=True)

9. username = db.Column(db.String(150), nullable=False, unique=True)

10. password = db.Column(db.String(150), nullable=False)

11. # Set to True for admin users

12. is\_admin = db.Column(db.Boolean, default=False)

13.

14.

15. class Grievance(db.Model):

16. id = db.Column(db.Integer, primary\_key=True)

17. title = db.Column(db.String(200), nullable=False)

18. description = db.Column(db.Text, nullable=False)

19. user\_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

20. status = db.Column(db.String(50), default='Pending')

21.

22. user = db.relationship('User', backref='grievances')

23.

24. def \_\_repr\_\_(self):

25. return f'<Grievance {self.title}>'

26.

**Frontend –**

**base.html –** our base html template that other html files inherit from.

1. <!DOCTYPE html>

2. <html lang="en">

3. <head>

4. <meta charset="UTF-8">

5. <meta name="viewport" content="width=device-width, initial-scale=1.0">

6. <title>Grievance Portal</title>

7. <!-- Bootstrap CSS -->

8. <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">

9. <!-- Custom CSS -->

10. <link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}">

11. </head>

12.

13. <body>

14. <nav>

15. <a href="{{ url\_for('home') }}">Home</a>

16. {% if current\_user.is\_authenticated %}

17. <a href="{{ url\_for('dashboard') }}">Dashboard</a>

18. <a href="{{ url\_for('logout') }}">Logout</a>

19. {% else %}

20. <a href="{{ url\_for('login') }}">Login</a>

21. <a href="{{ url\_for('register') }}">Register</a>

22. {% endif %}

23. </nav>

24. <div class="container">

25. {% with messages = get\_flashed\_messages(with\_categories=true) %}

26. {% if messages %}

27. {% for category, message in messages %}

28. <div class="alert {{ category }}">{{ message }}</div>

29. {% endfor %}

30. {% endif %}

31. {% endwith %}

32. {% block content %}{% endblock %}

33. </div>

34. <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>

35. </body>

36. </html>

37.

**style.css –** our base css stylesheet**.**

1. body {

2. font-family: Arial, sans-serif;

3. background-color: #f8f9fa;

4. margin: 0;

5. padding: 0;

6. }

7.

8. nav {

9. background-color: #343a40;

10. padding: 10px;

11. }

12.

13. nav a {

14. color: white;

15. text-decoration: none;

16. margin-right: 15px;

17. }

18.

19. nav a:hover {

20. text-decoration: underline;

21. }

22.

23. .container {

24. margin-top: 20px;

25. }

26.

27. .alert {

28. margin: 10px 0;

29. }

30.

31. .dashboard-table {

32. margin-top: 20px;

33. }

34.

**Output:**

**Homepage –**

**A screenshot of a computer

Description automatically generated**

**User Registration –**

**A screenshot of a computer

Description automatically generated**

**Login –**

**A screenshot of a computer

Description automatically generated**

**User Dashboard –**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

* **ASSIGNMENT-3**

**Objective:** Generate all the possible states of Tic Tac Toe.

**Code:**

1. from itertools import product

2.

3. # Function to check if a board configuration is valid based on Tic-Tac-Toe rules

4. def is\_valid\_board(board):

5. # Count the number of X's and O's

6. x\_count = board.count('X')

7. o\_count = board.count('O')

8.

9. # Ensure the number of X's is equal to or one more than the number of O's

10. if not (x\_count == o\_count or x\_count == o\_count + 1):

11. return False

12.

13. # Check for winning conditions

14. if check\_winner(board, 'X') and check\_winner(board, 'O'):

15. return False # Both players can't win simultaneously

16. return True

17.

18. # Function to check if a player has won

19. def check\_winner(board, player):

20. # Win conditions for rows, columns, and diagonals

21. win\_conditions = [

22. [0, 1, 2], [3, 4, 5], [6, 7, 8], # Rows

23. [0, 3, 6], [1, 4, 7], [2, 5, 8], # Columns

24. [0, 4, 8], [2, 4, 6] # Diagonals

25. ]

26. for condition in win\_conditions:

27. if all(board[i] == player for i in condition):

28. return True

29. return False

30.

31. # Function to generate all valid Tic-Tac-Toe board configurations

32. def generate\_all\_boards():

33. all\_boards = []

34. # Generate all combinations of 'X', 'O', and ' ' for the 9 positions

35. for comb in product('XO ', repeat=9):

36. board = list(comb)

37. if is\_valid\_board(board):

38. all\_boards.append(board)

39. return all\_boards

40.

41. # Get all valid boards

42. valid\_boards = generate\_all\_boards()

43.

44. # Display all valid boards

45. for idx, board in enumerate(valid\_boards):

46. print(f"Board {idx + 1}: {board}")

47.

**Output:**

**A screenshot of a computer

Description automatically generated**

* **ASSIGNMENT-4**

**Objective:** ImplementA\* search Algorithm.

**Code:**

1. import heapq

2.

3. class Node:

4. def \_\_init\_\_(self, state, parent=None, g=0, h=0):

5. self.state = state # The state represents the node (e.g., a position in a maze)

6. self.parent = parent # The parent node

7. self.g = g # Cost from start to the current node

8. self.h = h # Heuristic from current node to goal

9. self.f = g + h # Total cost (g + h)

10.

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

12. # Comparison operator for the priority queue (heapq)

13. return self.f < other.f

14.

15. def a\_star\_search(start, goal, get\_neighbors, heuristic):

16. """

17. A\* Search Algorithm to find the shortest path from start to goal.

18. :param start: The start node (state)

19. :param goal: The goal node (state)

20. :param get\_neighbors: Function to get neighbors of a node

21. :param heuristic: Heuristic function to estimate the cost to the goal

22. :return: List of states representing the path from start to goal, or None if no path found

23. """

24. # Initialize open and closed lists

25. open\_list = []

26. closed\_list = set()

27.

28. # Push the start node into the open list

29. start\_node = Node(start, None, 0, heuristic(start, goal))

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

31.

32. while open\_list:

33. # Get the node with the lowest f value

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

35.

36. # If goal is reached, reconstruct the path

37. if current\_node.state == goal:

38. path = []

39. while current\_node:

40. path.append(current\_node.state)

41. current\_node = current\_node.parent

42. return path[::-1] # Return reversed path (from start to goal)

43.

44. closed\_list.add(current\_node.state)

45.

46. # Get the neighbors of the current node

47. for neighbor, cost in get\_neighbors(current\_node.state):

48. if neighbor in closed\_list:

49. continue

50.

51. g = current\_node.g + cost

52. h = heuristic(neighbor, goal)

53. neighbor\_node = Node(neighbor, current\_node, g, h)

54.

55. # Add the neighbor to the open list if it is not already there

56. if all(neighbor\_node.f < node.f for node in open\_list):

57. heapq.heappush(open\_list, neighbor\_node)

58.

59. return None # Return None if no path found

60.

61. # Example heuristic function (Manhattan distance for grid-based pathfinding)

62. def heuristic(state, goal):

63. x1, y1 = state

64. x2, y2 = goal

65. return abs(x1 - x2) + abs(y1 - y2)

66.

67. # Example function to get neighbors (4-directional movement for a grid)

68. def get\_neighbors(state):

69. neighbors = []

70. x, y = state

71. # Move up, down, left, right (grid-based example)

72. for dx, dy, cost in [(-1, 0, 1), (1, 0, 1), (0, -1, 1), (0, 1, 1)]:

73. neighbor = (x + dx, y + dy)

74. neighbors.append((neighbor, cost))

75. return neighbors

76.

77. # Example usage

78. start = (0, 0) # Starting position (x, y)

79. goal = (4, 4) # Goal position (x, y)

80.

81. path = a\_star\_search(start, goal, get\_neighbors, heuristic)

82.

83. if path:

84. print("Path found:", path)

85. else:

86. print("No path found")

**Output:**

****

* **ASSIGNMENT-5**

**Objective:** ImplementAO\* search algorithm.

**Code:**

1. class AONode:

2. def \_\_init\_\_(self, name, heuristic\_cost):

3. self.name = name

4. self.heuristic\_cost = heuristic\_cost # Heuristic cost of the node

5. self.successors = [] # List of AND/OR successor groups

6. self.solved = False # Whether the node is solved

7. self.best\_successor = None # Best successor group (optimal path)

8.

9. def add\_successors(self, successors):

10. """

11. Add a successor group to the node.

12. Each successor group is a list of nodes (AND branch).

13. """

14. for group in successors:

15. if not all(isinstance(child, AONode) for child in group):

16. raise ValueError("All children in successors must be instances of AONode.")

17. self.successors.extend(successors)

18.

19. def ao\_star(node, trace\_path=[]):

20. """

21. The AO\* algorithm to find the optimal solution path.

22. """

23. if node.solved:

24. return node.solved

25.

26. print(f"Visiting Node: {node.name}")

27. trace\_path.append(node.name)

28.

29. # If it's a leaf node, mark it as solved

30. if not node.successors:

31. node.solved = True

32. trace\_path.pop()

33. return True

34.

35. # Evaluate all successor groups to find the best one

36. min\_cost = float('inf')

37. best\_successor = None

38.

39. for successors in node.successors:

40. # Compute the total cost for this group

41. total\_cost = sum(child.heuristic\_cost for child in successors)

42. if total\_cost < min\_cost:

43. min\_cost = total\_cost

44. best\_successor = successors

45.

46. # Set the best successor group

47. node.best\_successor = best\_successor

48.

49. # Recursively solve the best successor group

50. all\_solved = True

51. for child in best\_successor:

52. if not ao\_star(child, trace\_path):

53. all\_solved = False

54.

55. # If all successors in the best group are solved, mark the node as solved

56. node.solved = all\_solved

57. if all\_solved:

58. node.heuristic\_cost = min\_cost

59.

60. trace\_path.pop()

61. return node.solved

62.

63. def print\_solution(node):

64. """

65. Print the solution path found by the AO\* algorithm.

66. """

67. if not node or not node.best\_successor:

68. return

69. print(f"Node {node.name} -> ", [child.name for child in node.best\_successor])

70. for child in node.best\_successor:

71. print\_solution(child)

72.

73. # Example usage

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

75. # Creating an example And-Or graph

76. A = AONode("A", 10)

77. B = AONode("B", 8)

78. C = AONode("C", 7)

79. D = AONode("D", 6)

80. E = AONode("E", 5)

81.

82. # Define successors (AND/OR groups)

83. A.add\_successors([[B, C]]) # AND group: A -> {B AND C}

84. A.add\_successors([[D]]) # OR group: A -> D

85. B.add\_successors([[E]]) # OR group: B -> E

86.

87. # Run AO\* algorithm

88. trace\_path = []

89. ao\_star(A, trace\_path)

90.

91. print("\nSolution Path:")

92. print\_solution(A)

**Output:**

**A black screen with blue and red text

Description automatically generated**

* **ASSIGNMENT-6**

**Objective:** ImplementBFS algorithm.

**Code:**

1. from collections import deque

2.

3. def bfs\_maze(maze, start, goal):

4. """

5. Perform BFS to find the shortest path in a maze from start to goal.

6.

7. :param maze: 2D list representing the maze (0 = open, 1 = wall)

8. :param start: Tuple (row, col) representing the start position

9. :param goal: Tuple (row, col) representing the goal position

10. :return: List representing the path from start to goal or None if no path exists

11. """

12. # Directions (up, down, left, right)

13. directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

14.

15. # Queue for BFS (FIFO)

16. queue = deque([(start, [start])]) # (current position, path taken to reach that position)

17. visited = set() # Set to track visited positions

18.

19. while queue:

20. current\_pos, path = queue.popleft() # Dequeue the first node in the queue

21.

22. # Goal test

23. if current\_pos == goal:

24. return path # Return the path if the goal is found

25.

26. visited.add(current\_pos) # Mark the position as visited

27.

28. # Generate neighbors (adjacent cells)

29. for direction in directions:

30. new\_row, new\_col = current\_pos[0] + direction[0], current\_pos[1] + direction[1]

31.

32. # Check if the new position is within bounds and not a wall

33. if (0 <= new\_row < len(maze)) and (0 <= new\_col < len(maze[0])) and maze[new\_row][new\_col] == 0:

34. new\_pos = (new\_row, new\_col)

35. if new\_pos not in visited:

36. visited.add(new\_pos)

37. queue.append((new\_pos, path + [new\_pos])) # Enqueue the new position with updated path

38.

39. return None # Return None if no path is found

40.

41.

42. # Example usage

43. maze = [

44. [0, 0, 0, 0, 0],

45. [0, 1, 1, 1, 0],

46. [0, 1, 0, 0, 0],

47. [0, 1, 1, 1, 0],

48. [0, 0, 0, 0, 0]

49. ]

50.

51. start = (0, 0) # Starting position (row, col)

52. goal = (4, 4) # Goal position (row, col)

53.

54. path = bfs\_maze(maze, start, goal)

55.

56. if path:

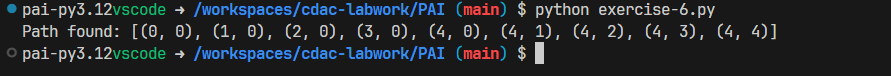
57. print("Path found:", path)

58. else:

59. print("No path found")

60.

**Output:**

****

* **ASSIGNMENT-7**

**Objective:** Implement DFS algorithm.

**Code:**

1. def dfs\_maze(maze, start, goal):

2. """

3. Perform DFS to find the shortest path in a maze from start to goal.

4.

5. :param maze: 2D list representing the maze (0 = open, 1 = wall)

6. :param start: Tuple (row, col) representing the start position

7. :param goal: Tuple (row, col) representing the goal position

8. :return: List representing the path from start to goal or None if no path exists

9. """

10. # Directions (up, down, left, right)

11. directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

12.

13. # Stack for DFS (LIFO)

14. stack = [(start, [start])] # (current position, path taken to reach that position)

15. visited = set() # Set to track visited positions

16.

17. while stack:

18. current\_pos, path = stack.pop() # Pop the last node in the stack

19.

20. # Goal test

21. if current\_pos == goal:

22. return path # Return the path if the goal is found

23.

24. visited.add(current\_pos) # Mark the position as visited

25.

26. # Generate neighbors (adjacent cells)

27. for direction in directions:

28. new\_row, new\_col = current\_pos[0] + direction[0], current\_pos[1] + direction[1]

29.

30. # Check if the new position is within bounds and not a wall

31. if (0 <= new\_row < len(maze)) and (0 <= new\_col < len(maze[0])) and maze[new\_row][new\_col] == 0:

32. new\_pos = (new\_row, new\_col)

33. if new\_pos not in visited:

34. visited.add(new\_pos)

35. stack.append((new\_pos, path + [new\_pos])) # Push the new position to the stack with updated path

36.

37. return None # Return None if no path is found

38.

39.

40. # Example usage

41. maze = [

42. [0, 0, 0, 0, 0],

43. [0, 1, 1, 1, 0],

44. [0, 1, 0, 0, 0],

45. [0, 1, 1, 1, 0],

46. [0, 0, 0, 0, 0]

47. ]

48.

49. start = (0, 0) # Starting position (row, col)

50. goal = (4, 4) # Goal position (row, col)

51.

52. path = dfs\_maze(maze, start, goal)

53.

54. if path:

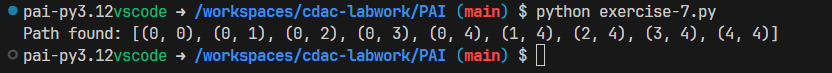
55. print("Path found:", path)

56. else:

57. print("No path found")

58.

**Output:**

****

* **ASSIGNMENT-8**

**Objective:** Implement N Queens problem.

**Code:**

1. def print\_solution(board):

2. """Print the chessboard."""

3. for row in board:

4. print(" ".join("Q" if col else "." for col in row))

5. print()

6.

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

8. """

9. Check if placing a queen at board[row][col] is safe.

10. """

11. # Check the current column

12. for i in range(row):

13. if board[i][col]:

14. return False

15.

16. # Check the upper-left diagonal

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

18. if board[i][j]:

19. return False

20.

21. # Check the upper-right diagonal

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

23. if board[i][j]:

24. return False

25.

26. return True

27.

28. def solve\_n\_queens(board, row, n):

29. """

30. Solve the N Queens problem using backtracking.

31. """

32. if row >= n:

33. print\_solution(board)

34. return True

35.

36. res = False

37. for col in range(n):

38. if is\_safe(board, row, col, n):

39. # Place the queen

40. board[row][col] = True

41.

42. # Recurse for the next row

43. res = solve\_n\_queens(board, row + 1, n) or res

44.

45. # Backtrack and remove the queen

46. board[row][col] = False

47.

48. return res

49.

50. def n\_queens(n):

51. """

52. Initialize the board and solve the N Queens problem.

53. """

54. board = [[False] \* n for \_ in range(n)]

55. if not solve\_n\_queens(board, 0, n):

56. print("No solution exists.")

57.

58. # Example Usage

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

60. N = 8 # Change this to solve for a different board size

61. n\_queens(N)

62.

**Output:**

**A screenshot of a computer

Description automatically generated**