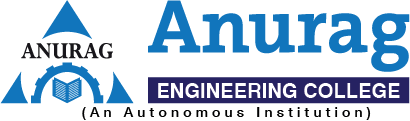
****

**Artificial Intelligence**

**Lab Manual**

**Experiment 1: Breadth-First Search (BFS)**

graph = {

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

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

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

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

'E' : ['C','D']

}

# To store visited nodes.

visitedNodes = []

# To store nodes in queue

queueNodes = []

# function

def bfs(visitedNodes, graph, snode):

visitedNodes.append(snode)

queueNodes.append(snode)

print()

print("RESULT :")

while queueNodes:

s = queueNodes.pop(0)

print (s, end = " ")

for neighbour in graph[s]:

if neighbour not in visitedNodes:

visitedNodes.append(neighbour)

queueNodes.append(neighbour)

# Main Code

snode = input("Enter Starting Node(A, B, C, D, or E) :").upper()

# calling bfs function

bfs(visitedNodes, graph, snode)

**Output:**

Enter Starting Node(A, B, C, D, or E) :c

RESULT :

C A B E D

**Experiment 2: Depth-First Search (DFS)**

### # Input Graph

### graph = {

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

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

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

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

### 'E' : ['C','D']

### }

### # Set used to store visited nodes.

### visitedNodes = list()

### # function

### def dfs(visitedNodes, graph, node):

### if node not in visitedNodes:

### print (node,end=" ")

### visitedNodes.append(node)

### for neighbour in graph[node]:

### dfs(visitedNodes, graph, neighbour)

### # Driver Code

### snode = input("Enter Starting Node(A, B, C, D, or E) :").upper()

### # calling bfs function

### print("RESULT :")

### print("-"\*20)

### dfs(visitedNodes, graph, snode)

**Output:**

#### Enter Starting Node(A, B, C, D, or E) :D

#### RESULT :

#### --------------------

#### D B A C E

#### ****Experiment 3: Minimax Algorithm for Tic-Tac-Toe game****

### How It Works:

1. **Board Representation**: The board is represented as a 3x3 list of lists.
2. **Player Move**: Players input a move (1-9), which corresponds to a position on the board.
3. **Check for Winner**: The game checks rows, columns, and diagonals to determine if there is a winner.
4. **Check for Full Board**: If the board is full without a winner, the game declares a tie.
5. **Game Loop**: The game alternates between players 'X' and 'O' until there is a winner or a tie.

# Tuple to store winning positions.

win\_positions = (

(0, 1, 2), (3, 4, 5), (6, 7, 8),

(0, 3, 6), (1, 4, 7), (2, 5, 8),

(0, 4, 8), (2, 4, 6)

)

def game(player):

# diplay current mesh

print("\n", " | ".join(mesh[:3]))

print("---+---+---")

print("", " | ".join(mesh[3:6]))

print("---+---+---")

print("", " | ".join(mesh[6:]))

# Loop until player valid input cell number.

while True:

try:

ch = int(input(f"Enter player {player}'s choice : "))

if str(ch) not in mesh:

raise ValueError

mesh[ch-1] = player

break

except ValueError:

print("Invalid position number.")

# Return winning positions if player wins, else None.

for wp in win\_positions:

if all(mesh[pos] == player for pos in wp):

return wp

return None

player1 = "X"

player2 = "O"

player = player1

mesh = list("123456789")

for i in range(9):

won = game(player)

if won:

print("\n", " | ".join(mesh[:3]))

print("---+---+---")

print("", " | ".join(mesh[3:6]))

print("---+---+---")

print("", " | ".join(mesh[6:]))

print(f"\*\*\* Player {player} won! \*\*\*")

break

player = player1 if player == player2 else player2

else:

# 9 moves without a win is a draw.

print("Game ends in a draw.")

**Output:**

**1 | 2 | 3**

**---+---+---**

**4 | 5 | 6**

**---+---+---**

**7 | 8 | 9**

**Enter player X's choice : 2**

**1 | X | 3**

**---+---+---**

**4 | 5 | 6**

**---+---+---**

**7 | 8 | 9**

**Enter player O's choice : 1**

**O | X | 3**

**---+---+---**

**4 | 5 | 6**

**---+---+---**

**7 | 8 | 9**

**Enter player X's choice : 6**

**O | X | 3**

**---+---+---**

**4 | 5 | X**

**---+---+---**

**7 | 8 | 9**

**Enter player O's choice : 5**

**O | X | 3**

**---+---+---**

**4 | O | X**

**---+---+---**

**7 | 8 | 9**

**Enter player X's choice : 8**

**O | X | 3**

**---+---+---**

**4 | O | X**

**---+---+---**

**7 | X | 9**

**Enter player O's choice : 4**

**O | X | 3**

**---+---+---**

**O | O | X**

**---+---+---**

**7 | X | 9**

**Enter player X's choice : 9**

**O | X | 3**

**---+---+---**

**O | O | X**

**---+---+---**

**7 | X | X**

**Enter player O's choice : 7**

**O | X | 3**

**---+---+---**

**O | O | X**

**---+---+---**

**O | X | X**

**\*\*\* Player O won! \*\*\***

**Experiment 4:** program to implement 8-puzzle problem using python A\* search algorithm.

from collections import deque

def bfs(start\_state):

target = [1, 2, 3, 4, 5, 6, 7, 8 , 0]

dq = deque([start\_state])

visited = {tuple(start\_state): None}

while dq:

state = dq.popleft()

if state == target:

path = []

while state:

path.append(state)

state = visited[tuple(state)]

return path[::-1]

zero = state.index(0)

row, col = divmod(zero, 3)

for move in (-3, 3, -1, 1):

new\_row, new\_col = divmod(zero + move, 3)

if 0 <= new\_row < 3 and 0 <= new\_col < 3 and abs(row - new\_row) + abs(col - new\_col) == 1:

neighbor = state[:]

neighbor[zero], neighbor[zero + move] = neighbor[zero + move], neighbor[zero]

if tuple(neighbor) not in visited:

visited[tuple(neighbor)] = state

dq.append(neighbor)

def printSolution(path):

for state in path:

print("\n".join(' '.join(map(str, state[i:i+3])) for i in range(0, 9, 3)), end="\n-----\n")

# Example Usage

startState = [1, 3, 0 , 6, 8, 4, 7, 5, 2]

solution = bfs(startState)

if solution:

printSolution(solution)

print(f"Solved in {len(solution) - 1} moves.")

else:

print("No solution found.")

**Output:**

1 3 0

6 8 4

7 5 2

-----

1 3 4

6 8 0

7 5 2

-----

1 3 4

6 8 2

7 5 0

-----

1 3 4

6 8 2

7 0 5

-----

1 3 4

6 0 2

7 8 5

-----

1 3 4

0 6 2

7 8 5

-----

1 3 4

7 6 2

0 8 5

-----

1 3 4

7 6 2

8 0 5

-----

1 3 4

7 0 2

8 6 5

-----

1 3 4

7 2 0

8 6 5

-----

1 3 0

7 2 4

8 6 5

-----

1 0 3

7 2 4

8 6 5

-----

1 2 3

7 0 4

8 6 5

-----

1 2 3

7 4 0

8 6 5

-----

1 2 3

7 4 5

8 6 0

-----

1 2 3

7 4 5

8 0 6

-----

1 2 3

7 4 5

0 8 6

-----

1 2 3

0 4 5

7 8 6

-----

1 2 3

4 0 5

7 8 6

-----

1 2 3

4 5 0

7 8 6

-----

1 2 3

4 5 6

7 8 0

-----

Solved in 20 moves.

**Experiment 5:** program to implement Water jug problem using python

# jug1 and jug2 contain the value

jug1, jug2, goal = 4, 3, 2

# Initialize a 2D list for visited states

# The list will have dimensions (jug1+1) x (jug2+1) to cover all possible states

visited = [[False for \_ in range(jug2 + 1)] for \_ in range(jug1 + 1)]

def waterJug(vol1, vol2):

# Check if we reached the goal state

if (vol1 == goal and vol2 == 0) or (vol2 == goal and vol1 == 0):

print(vol1,"\t", vol2)

print("Solution Found")

return True

# If this state has been visited, return False

if visited[vol1][vol2]:

return False

# Mark this state as visited

visited[vol1][vol2] = True

# Print the current state

print(vol1,"\t", vol2)

# Try all possible moves:

return (

waterJug(0, vol2) or # Empty jug1

waterJug(vol1, 0) or # Empty jug2

waterJug(jug1, vol2) or # Fill jug1

waterJug(vol1, jug2) or # Fill jug2

waterJug(vol1 + min(vol2, (jug1 - vol1)), vol2 - min(vol2, (jug1 - vol1))) or # Pour water from jug2 to jug1

waterJug(vol1 - min(vol1, (jug2 - vol2)), vol2 + min(vol1, (jug2 - vol2))) # Pour water from jug1 to jug2

)

print("Steps: ")

print("Jug1 \t Jug2 ")

print("----- \t ------")

waterJug(0, 0)

**Output:**

Steps:

Jug1 Jug2

----- ------

0 0

4 0

4 3

0 3

3 0

3 3

4 2

0 2

Solution Found

**Experiment 6:** program to implement Travelling Salesman problem using python

from collections import deque

def tsp\_bfs(graph):

n = len(graph) # Number of cities

startCity = 0 # Starting city

min\_cost = float('inf') # Initialize minimum cost as infinity

opt\_path = [] # To store the optimal path

# Queue for BFS: Each element is (cur\_path, cur\_cost)

dq = deque([([startCity], 0)])

print("Path Traversal:")

while dq:

cur\_path, cur\_cost = dq.popleft()

cur\_city = cur\_path[-1]

# Print the current path and cost

print(f"Current Path: {cur\_path}, Current Cost: {cur\_cost}")

# If all cities are visited and we are back at the startCity

if len(cur\_path) == n and cur\_path[0] == startCity:

total\_cost = cur\_cost + graph[cur\_city][startCity]

if total\_cost < min\_cost:

min\_cost = total\_cost

opt\_path = cur\_path + [startCity]

continue

# Explore all neighboring cities (add in BFS manner)

for next\_city in range(n):

if next\_city not in cur\_path: # Visit unvisited cities

new\_path = cur\_path + [next\_city]

new\_cost = cur\_cost + graph[cur\_city][next\_city]

dq.append((new\_path, new\_cost))

return min\_cost, opt\_path

# Example graph as a 2D adjacency matrix

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

# Solve TSP using BFS

min\_cost, opt\_path = tsp\_bfs(graph)

print("\nOptimal Solution:")

print(f"Minimum cost: {min\_cost}")

print(f"Optimal path: {opt\_path}")

**Output:**

Path Traversal:

Current Path: [0], Current Cost: 0

Current Path: [0, 1], Current Cost: 10

Current Path: [0, 2], Current Cost: 15

Current Path: [0, 3], Current Cost: 20

Current Path: [0, 1, 2], Current Cost: 45

Current Path: [0, 1, 3], Current Cost: 35

Current Path: [0, 2, 1], Current Cost: 50

Current Path: [0, 2, 3], Current Cost: 45

Current Path: [0, 3, 1], Current Cost: 45

Current Path: [0, 3, 2], Current Cost: 50

Current Path: [0, 1, 2, 3], Current Cost: 75

Current Path: [0, 1, 3, 2], Current Cost: 65

Current Path: [0, 2, 1, 3], Current Cost: 75

Current Path: [0, 2, 3, 1], Current Cost: 70

Current Path: [0, 3, 1, 2], Current Cost: 80

Current Path: [0, 3, 2, 1], Current Cost: 85

Optimal Solution:

Minimum cost: 80

Optimal path: [0, 1, 3, 2, 0]

**Experiment 7:** program to implement Towers of Hanoi using python

def towers\_of\_hanoi(n, source, target, auxiliary):

if n == 1:

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

return

towers\_of\_hanoi(n - 1, source, auxiliary, target)

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

towers\_of\_hanoi(n - 1, auxiliary, target, source)

# Example usage

n = 3 # Number of disks

towers\_of\_hanoi(n, 'A', 'C', 'B')

**Output:**

### **Move disk 1 from A to C**

### **Move disk 2 from A to B**

### **Move disk 1 from C to B**

### **Move disk 3 from A to C**

### **Move disk 1 from B to A**

### **Move disk 2 from B to C**

### **Move disk 1 from A to C**

**Experiment 8:**program to implement Monkey Banana Problem using python

class MonkeyBananaProblem:

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

self.state = {

'monkey': 'floor', # 'floor' or 'stool'

'position': 'A', # Position can be 'A', 'B', or 'C'

'stool': 'B', # Position of the stool

'banana': 'hanging', # 'hanging' or 'grabbed'

}

def move\_to(self, position):

print(f"Monkey moves to position {position}")

self.state['position'] = position

def push\_stool(self, position):

print(f"Monkey pushes stool to position {position}")

self.state['stool'] = position

self.state['position'] = position

def climb\_stool(self):

if self.state['position'] == self.state['stool']:

print("Monkey climbs the stool")

self.state['monkey'] = 'stool'

else:

print("Monkey must be at the stool's position to climb")

def grab\_banana(self):

if self.state['monkey'] == 'stool' and self.state['position'] == 'B' and self.state['banana'] == 'hanging':

print("Monkey grabs the banana")

self.state['banana'] = 'grabbed'

else:

print("Monkey cannot grab the banana in the current state")

def is\_goal\_state(self):

return self.state['banana'] == 'grabbed'

def solve(self):

# Step 1: Move to the stool

if self.state['position'] != self.state['stool']:

self.move\_to(self.state['stool'])

# Step 2: Push the stool to position B (under the banana)

if self.state['stool'] != 'B':

self.push\_stool('B')

# Step 3: Climb the stool

self.climb\_stool()

# Step 4: Grab the banana

self.grab\_banana()

# Check if goal state is reached

if self.is\_goal\_state():

print("Monkey successfully grabbed the banana!")

else:

print("Monkey failed to grab the banana.")

# Example usage

problem = MonkeyBananaProblem()

problem.solve()

### **Explanation**:

1. **State Representation**: The state dictionary represents the current state of the monkey, stool, and banana.
2. **Actions**: The monkey can perform several actions:
   * move\_to(position): Moves the monkey to a specified position.
   * push\_stool(position): Pushes the stool to a specified position.
   * climb\_stool(): Climbs the stool if the monkey is at the stool's position.
   * grab\_banana(): Grabs the banana if the monkey is on the stool under the banana.
3. **Goal State**: The goal state is when the banana is grabbed (state['banana'] == 'grabbed').
4. **Solving the Problem**: The solve method orchestrates the actions to solve the problem step-by-step.

**Output**:

Monkey moves to position B

Monkey climbs the stool

Monkey grabs the banana

Monkey successfully grabbed the banana!

**Experiment 9: program to implement Alpha-Beta pruning using python**

"""

Alpha Beta Pruning :

-------------------

depth (int): Current depth in the game tree.

node\_index (int): Index of the current node in the values array.

maximizing\_player (bool): True if the current player is maximizing, False otherwise.

values (list): List of leaf node values.

alpha (float): Best value for the maximizing player.

beta (float): Best value for the minimizing player.

Returns:

int: The optimal value for the current player.

"""

import math

def alpha\_beta\_pruning(depth, node\_index, maximizing\_player, values, alpha, beta):

# Base case: leaf node

if depth == 0 or node\_index >= len(values):

return values[node\_index]

if maximizing\_player:

max\_eval = -math.inf

for i in range(2): # Each node has two children

eval = alpha\_beta\_pruning(depth - 1, node\_index \* 2 + i, False, values, alpha, beta)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break # Beta cutoff

return max\_eval

else:

min\_eval = math.inf

for i in range(2): # Each node has two children

eval = alpha\_beta\_pruning(depth - 1, node\_index \* 2 + i, True, values, alpha, beta)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break # Alpha cutoff

return min\_eval

# Example usage

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

# Leaf node values for a complete binary tree

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

depth = 3 # Height of the tree

optimal\_value = alpha\_beta\_pruning(depth, 0, True, values, -math.inf, math.inf)

print(f"The optimal value is: {optimal\_value}")

**Output:**

The optimal value is: 5

**Experiment 10: program to implement 8-Queens problem using python**

def printSolution(board):

"""Print the chessboard configuration."""

for row in board:

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

print("\n")

def isSafe(board, row, col, n):

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

# Check column

for i in range(row):

if board[i][col]:

return False

# Check upper-left diagonal

i, j = row, col

while i >= 0 and j >= 0:

if board[i][j]:

return False

i -= 1

j -= 1

# Check upper-right diagonal

i, j = row, col

while i >= 0 and j < n:

if board[i][j]:

return False

i -= 1

j += 1

return True

def solveNQueens(board, row, n):

"""Use backtracking to solve the N-Queens problem."""

if row == n:

printSolution(board)

return True

result = False

for col in range(n):

if isSafe(board, row, col, n):

# Place the queen

board[row][col] = 1

# Recur to place the rest of the queens

result = solveNQueens(board, row + 1, n) or result

# Backtrack

board[row][col] = 0

return result

def nQueens(n):

"""Driver function to solve the N-Queens problem."""

board = [[0] \* n for \_ in range(n)]

if not solveNQueens(board, 0, n):

print("No solution exists.")

else:

print("Solutions printed above.")

# Solve the 8-Queens problem

nQueens(8)

**Output:**

Q . . . . . . .

. . . . Q . . .

. . . . . . . Q

. . . . . Q . .

. . Q . . . . .

. . . . . . Q .

. Q . . . . . .

. . . Q . . . .