ARUNAI ENGINEERING COLLEGE

(Affiliated to Anna University)

Velu Nagar,Thiruvannamalai-606 603 www.arunai.org

DEPARTMENT OF
ARTIFICIAL INTELLIGENCE & DATA SCIENCE

BACHELOR OF TECHNOLOGY

2024- 2025

THIRD SEMESTER

AD3311
ARTIFICIAL INTELLIGENCE LAB

SYLLABUS

AD3311-ARTIFICIAL INTELLIGENCE LABORATORY

OBJECTIVES:

To design and implement search strategies
To implement game playing techniques
To implement CSP techniques
To develop systems with logical reasoning
To develop systems with probabilistic reasoning

LIST OF EXPERIMENTS:

1. Implement basic search strategies – 8-Puzzle, 8 - Queens problem, Cryptarithmetic.
2. Implement A* and memory bounded A* algorithms
3. Implement Minimax algorithm for game playing (Alpha-Beta pruning)
4. Solve constraint satisfaction problems
5. Implement propositional model checking algorithms
6. Implement forward chaining, backward chaining, and resolution strategies
7. Build naïve Bayes models
8. Implement Bayesian networks and perform inferences
9. Mini-Project

OUTCOMES:

At the end of this course, the students will be able to:

CO1: Design and implement search strategies
CO2: Implement game playing and CSP techniques
CO3: Develop logical reasoning systems
CO4: Develop probabilistic reasoning systems

INDEX

Ex: 1.a **8 PUZZLE PROBLEM**

Aim: To implement the 8 Puzzle Problem using python

Algorithm

START

1. In order to maintain **the list of live nodes**, algorithm **LCSearch** employs the functions **Least()** and **Add().**
2. **Least()** identifies a live node with the **least c(y),** removes it from the list, and returns it.
3. **Add(y)** adds **y** to the list of live nodes.
4. **Add(y)** implements the list of live nodes as a **min-heap**.

END

Program

```
# Importing copy for deepcopy function
import copy

# Importing the heap functions from python
# library for Priority Queue
from heapq import heappush, heappop

# This variable can be changed to change
# the program from 8 puzzle(n=3) to 15
# puzzle(n=4) to 24 puzzle(n=5)...
n = 3

# bottom, left, top, right
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

# A class for Priority Queue
class priorityQueue:
```

```
    # Constructor to initialize a
    # Priority Queue
    def _init_(self):
        self.heap = []

    # Inserts a new key 'k'
    def push(self, k):
        heappush(self.heap, k)

    # Method to remove minimum element
    # from Priority Queue
    def pop(self):
        return heappop(self.heap)

    # Method to know if the Queue is empty
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure
class node:

    def _init_(self, parent, mat, empty_tile_pos,
                cost, level):

        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

        # Stores the matrix
        self.mat = mat
         # Stores the position at which the
```

```
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Stores the number of misplaced tiles
        self.cost = cost

        # Stores the number of moves so far
        self.level = level

    # This method is defined so that the
    # priority queue is formed based on
    # the cost variable of the objects
    def __lt__(self, nxt):
        return self.cost < nxt.cost

# Function to calculate the number of
# misplaced tiles ie. number of non-blank
# tiles not in their goal position
def calculateCost(mat, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1

    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:

    # Copy data from parent matrix to current matrix
    new_mat = copy.deepcopy(mat)
```

```
    # Move tile by 1 position
    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

    # Set number of misplaced tiles
    cost = calculateCost(new_mat, final)

    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node

# Function to print the N x N matrix
def printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")

        print()

# Function to check if (x, y) is a valid
# matrix coordinate
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node
def printPath(root):

    if root == None:
        return
```

```
    printPath(root.parent)
    printMatrix(root.mat)
    print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state.
def solve(initial, empty_tile_pos, final):

    # Create a priority queue to store live
    # nodes of search tree
    pq = priorityQueue()

    # Create the root node
    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)

    # Add root to list of live nodes
    pq.push(root)

    # Finds a live node with least cost,
    # add its children to list of live
    # nodes and finally deletes it from
    # the list.
    while not pq.empty():

        # Find a live node with least estimated
        # cost and delete it form the list of
        # live nodes
        minimum = pq.pop()

        # If minimum is the answer node
        if minimum.cost == 0:
```

```
            # Print the path from root to
            # destination;
            printPath(minimum)
            return

        # Generate all possible children
        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]

            if isSafe(new_tile_pos[0], new_tile_pos[1]):

                # Create a child node
                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)

                # Add child to list of live nodes
                pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Solvable Final configuration
# Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
          [ 5, 8, 6 ],
```

```
        [ 0, 7, 4 ] ]

# Blank tile coordinates in
# initial configuration
empty_tile_pos = [ 1, 2 ]

# Function call to solve the puzzle
solve(initial, empty_tile_pos, final)
```

Output

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 4 7
```

Result : Thus the 8 - Puzzle Problem is solved by Python code.

Ex: 1.b **8 QUEEN PROBLEM**

Aim: To implement the 8 Queen Problem using python

Algorithm

START
1. begin from the leftmost column
2. if all the queens are placed,
   return true/ print configuration
3. check for all rows in the current column
   a) if queen placed safely, mark row and column; and
      recursively check if we approach in the current
      configuration, do we obtain a solution or not
   b) if placing yields a solution, return true
   c) if placing does not yield a solution, unmark and
      try other rows
4. if all rows tried and solution not obtained, return
   false and backtrack

END

Program

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())

# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]

def attack(i, j):
        #checking vertically and horizontally
        for k in range(0,N):
                if board[i][k]==1 or board[k][j]==1:
                        return True
        #checking diagonally
for k in range(0,N):
```

```
            for l in range(0,N):
                    if (k+l==i+j) or (k-
                            l==i-j): if
                            board[k][l]=
                            =1:
                                    return True
            return False

    def  N_queens(n):
            if n==0:
                    retur
            n True for i
            in
            range(0,N):
                    for j in range(0,N):
                            if (not(attack(i,j))) and
                            (board[i][j]!=1): board[i][j] = 1
                            if N_queens(n-
                                    1)==True: return
                                    True
                            board[i][j] = 0
            return False

  N_queen
   s(N) for
        i in
     board:
    print (i)
```

Output

Enter the number of
queens 8

[1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0]

Result : Thus the 8 - Puzzle Problem is solved by Python code.