

**RAJALAKSHMI ENGINEERING COLLEGE**  
**[AUTONOMOUS]**

**RAJALAKSHMI NAGAR, THANDALAM – 602 105**



**AI23231 – PRINCIPLES OF ARTIFICIAL INTELLIGENCE**  
**LABORATORY RECORD NOTEBOOK**

**NAME:** ..... **Andryan Durai Jebin** .....

**YEAR / SEMESTER:** ..... **Ist / II** .....

**BRANCH / SECTION:** ..... **Artificial intelligence and Data Science/ FA** .....

**REGISTER NO.** ..... **2116241801020** .....

**COLLEGE ROLL NO.** ..... **241801020** .....

**ACADEMIC YEAR:** **2024 – 2025**

**RAJALAKSHMI ENGINEERING COLLEGE**  
**[AUTONOMOUS]**  
**RAJALAKSHMI NAGAR, THANDALAM – 602 105**  
**BONAFIDE CERTIFICATE**

Name : ..... Andryan Durai Jebin .....

Academic Year : .... 2024-2025. .... Semester : 02 Branch : AIML

Register No.

2116241801020

Certified that this is the bonafide record of work done by the above student in the  
**AI23231 – PRINCIPLES OF ARTIFICIAL INTELLIGENCE**  
during the year 2024 - 2025.

Signature of Faculty in-charge

Submitted for the Practical Examination held on .....

Internal Examiner

External Examiner

## INDEX

| Ex. No | Date       | Name of the Experiment                                 | Git Hub QR Code | Sign |
|--------|------------|--|-----------------|------|
| 1      | 28-02-2025 | Write a program to solve 8 Queen problem               |                 |      |
| 2      | 07-03-2025 | Solve any problem using depth first search             |                 |      |
| 3      | 21-04-2025 | Implementation of MINIMAX algorithm                    |                 |      |
| 4      | 28-05-2025 | Implementation of A* algorithm                         |                 |      |
| 5      | 09-05-2025 | Implementation of Prolog Minimax algorithm             |                 |      |
| 6      | 16-05-2025 | Implementation of Unification and Resolution Algorithm |                 |      |
| 7      | 16-05-2025 | Implementation of Backward Chaining                    |                 |      |
| 8      | 23-05-2025 | Implementation of Forward Chaining                     |                 |      |
| 9      | 23-05-2025 | Implementation of Blocks World program                 |                 |      |
| 10     | 30-05-2025 | Implementing a fuzzy inference system                  |                 |      |

# IMPLEMENT EIGHT QUEENS PROBLEM

## Aim:

To Implement Eight Queens Problem using backtracking

## PROGRAM:

N = 8

```
def printSolution(board):
```

```
    for row in board:
```

```
        for i in range(N):
```

```
            print("Q" if row[i] else ".", end=" ")
```

```
        print()
```

```
    print()
```

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

```
    # Check the column
```

```
    for i in range(row):
```

```
        if board[i][col]:
```

```
            return False
```

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

```
        if board[i][j]:
```

```
            return False
```

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

```
        if board[i][j]:
```

```
            return False
```

```
    return True
```

```
def solve(board, row):
```

```
    if row == N:
```

```

        printSolution(board)
        return True
    for col in range(N):
        if isSafe(board, row, col):
            board[row][col] = 1
            if solve(board, row + 1):
                return True
            board[row][col] = 0
    return False

def eightQueens():
    board = [[0 for _ in range(N)] for _ in range(N)]
    if solve(board, 0):
        print("One possible solution is:")
    else:
        print("Solution does not exist")
eightQueens()

```

### OUTPUT:

```

Q.....
....Q...
.....Q
....Q..
..Q.....
.....Q.
.Q.....
...Q....
One possible solution is:

```

## IMPLEMENTATION OF DEPTH FIRST SEARCH

### Aim:

To implement depth first search

### PROGRAM:

```
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = {i: [] for i in range(vertices)}

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)

    def dfs(self, v, visited):
        print(v, end=" ")
        visited[v] = True
        for neighbor in self.graph[v]:
            if not visited[neighbor]:
                self.dfs(neighbor, visited)

def depth_first_search():
    g = Graph(6)
    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)
visited = [False] * g.vertices
print("Depth First Search starting from vertex 0:")
g.dfs(0, visited)
```

```
depth_first_search()
```

OUTPUT:

```
Depth First Search starting from vertex 0:
0 1 3 4 2 5
```

## IMPLEMENTATION OF MINIMAX algorithm

### Aim:

To implement MINIMAX algorithm.

### PROGRAM:

```
PLAYER_X = 1
```

```
PLAYER_O = -1
```

```
EMPTY = 0
```

```
def evaluate(board):
```

```
    for i in range(3):
```

```
        if board[i][0] == board[i][1] == board[i][2] != EMPTY:
```

```
            return board[i][0]
```

```
        if board[0][i] == board[1][i] == board[2][i] != EMPTY:
```

```
            return board[0][i]
```

```
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
```

```
        return board[0][0]
```

```
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
```

```
        return board[0][2]
```

```
    return 0
```

```
def isMovesLeft(board):
```

```
    for row in board:
```

```
        if EMPTY in row:
```

```
            return True
```

```
    return False
```



```

def minimax(board, isMaximizing):
    winner = evaluate(board)
    if winner != 0:
        return winner
    if not isMovesLeft(board):
        return 0
    if isMaximizing:
        bestScore = -float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_X
                    score = minimax(board, False)
                    board[row][col] = EMPTY
                    bestScore = max(bestScore, score)
        return bestScore
    else:
        bestScore = float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == EMPTY:
                    board[row][col] = PLAYER_O
                    score = minimax(board, True)
                    board[row][col] = EMPTY
                    bestScore = min(bestScore, score)
        return bestScore

```

```
def findBestMove(board):  
    bestValue = -float('inf')  
    bestMove = (-1, -1)  
    for row in range(3):  
        for col in range(3):  
            if board[row][col] == EMPTY:  
                board[row][col] = PLAYER_X  
                moveValue = minimax(board, False)  
                board[row][col] = EMPTY  
                if moveValue > bestValue:  
                    bestMove = (row, col)  
                    bestValue = moveValue  
    return bestMove
```

```
def printBoard(board):  
    for row in board:  
        for cell in row:  
            if cell == PLAYER_X:  
                print("X", end=" ")  
            elif cell == PLAYER_O:  
                print("O", end=" ")  
            else:  
                print(".", end=" ")  
        print()
```

```
board = [
```

```
[PLAYER_X, PLAYER_O, PLAYER_X],  
[PLAYER_O, PLAYER_X, EMPTY],  
[EMPTY,  PLAYER_O, PLAYER_X]  
]
```

```
print("Current Board:")
```

```
printBoard(board)
```

```
move = findBestMove(board)
```

```
print(f"\nBest Move: {move}")
```

```
board[move[0]][move[1]] = PLAYER_X
```

```
print("\nBoard after best move:")
```

```
printBoard(board)
```

### **OUTPUT:**

Current Board:

X O X

O X .

. O X

Best Move: (2, 0)

Board after best move:

X O X

O X .

X O X

## IMPLEMENTATION OF A\* SEARCH ALGORITHM

### Aim:

To implement A\* Search Algorithm

### PROGRAM:

```
import heapq

class Node:

    def __init__(self, state, parent, g, h):

        self.state = state

        self.parent = parent

        self.g = g

        self.h = h

        self.f = g + h

    def __lt__(self, other):

        return self.f < other.f

def a_star_search(start, goal, heuristic, neighbors):

    open_list = []

    closed_list = set()

    start_node = Node(start, None, 0, heuristic(start, goal))

    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_list.add(current_node.state)

for neighbor, cost in neighbors(current_node.state):
    if neighbor in closed_list:
        continue
    g = current_node.g + cost
    h = heuristic(neighbor, goal)
    neighbor_node = Node(neighbor, current_node, g, h)
    if not any(open_node.state == neighbor and open_node.f <= neighbor_node.f for
open_node in open_list):
        heapq.heappush(open_list, neighbor_node)

return None

def heuristic(state, goal):
    return abs(state[0] - goal[0]) + abs(state[1] - goal[1])

def neighbors(state):

```

```
x, y = state
return [
    ((x + 1, y), 1),
    ((x - 1, y), 1),
    ((x, y + 1), 1),
    ((x, y - 1), 1)
]
```

```
start = (0, 0)
goal = (3, 3)
path = a_star_search(start, goal, heuristic, neighbors)
print("Path from start to goal:", path)
```

#### **OUTPUT:**

For start = (0, 0) and goal = (3, 3):

Path from start to goal: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3)]

# IMPLEMENTATION OF DECISION MAKING AND KNOWLEDGE REPRESENTATION

## Aim:

To implement decision making and knowledge representation using prolog tool.

## PROGRAM:

```
likes(mary, food).
```

```
likes(mary, wine).
```

```
likes(john, wine).
```

```
likes(john, mary).
```

```
% Rules based on the conditions:
```

```
likes(john, X) :- likes(mary, X). % John likes anything that Mary likes
```

```
likes(john, Y) :- likes(Y, wine). % John likes anyone who likes wine
```

```
likes(john, Y) :- likes(Y, Y). % John likes anyone who likes themselves
```

```
% Sample queries:
```

```
% Query 1: Does John like food?
```

```
% ?- likes(john, food).
```

```
% Query 2: Does John like wine?
```

```
% ?- likes(john, wine).
```

```
% Query 3: Does John like food if Mary likes food?
```

```
% ?- likes(john, food).
```

% Query 4: Who does John like?

% ?- likes(john, Y).

**Output:**

Query: ?- likes (john, food),

yes

Query: ?- likes(john, wine),

yes

Query: ?- likes (john, food),

yes

Query: ?- likes (john, Y) .

Y = mary ;

Y = john ;

Y = wine ;

Query ?- likes(john, Y).

Y = mary ;

Y=john ;

Y = wine ;



# IMPLEMENTATION OF UNIFICATION AND RESOLUTION ALGORITHM

## Aim:

To implement unification and resolution algorithm using python.

## PROGRAM:

```
def unify(var1, var2, subst):  
    if subst is None:  
        return None  
    elif var1 == var2:  
        return subst  
    elif isinstance(var1, str) and var1.islower():  
        return unify_var(var1, var2, subst)  
    elif isinstance(var2, str) and var2.islower():  
        return unify_var(var2, var1, subst)  
    elif isinstance(var1, list) and isinstance(var2, list) and len(var1) == len(var2):  
        return unify(var1[1:], var2[1:], unify(var1[0], var2[0], subst))  
    else:  
        return None  
  
def unify_var(var, x, subst):  
    if var in subst:  
        return unify(subst[var], x, subst)  
    elif x in subst:  
        return unify(var, subst[x], subst)  
    elif occurs_check(var, x, subst):  
        return None
```

else:

    subst[var] = x

    return subst

def occurs\_check(var, x, subst):

    if var == x:

        return True

    elif isinstance(x, list):

        return any(occurs\_check(var, xi, subst) for xi in x)

    elif x in subst:

        return occurs\_check(var, subst[x], subst)

    return False

def resolution(kb, query):

    clauses = kb + [negate(query)]

    while True:

        new\_clauses = []

        for i, ci in enumerate(clauses):

            for j, cj in enumerate(clauses):

                if i >= j:

                    continue

                resolvents = resolve(ci, cj)

                if [] in resolvents:

                    return True

        for res in resolvents:

            if res not in new\_clauses:

                new\_clauses.append(res)

```
    if all(new in clauses for new in new_clauses):  
        return False  
    clauses.extend(new_clauses)
```

```
def negate(literal):  
    if isinstance(literal, list):  
        return [negate(lit) for lit in literal]  
    if literal.startswith("~"):  
        return literal[1:]  
    else:  
        return f"~{literal}"
```

```
def resolve(clause1, clause2):  
    resolvents = []  
    for lit1 in clause1:  
        for lit2 in clause2:  
            if lit1 == negate(lit2):  
                new_clause = list(set(clause1 + clause2))  
                new_clause.remove(lit1)  
                new_clause.remove(lit2)  
                if new_clause not in resolvents:  
                    resolvents.append(new_clause)  
    return resolvents
```

```
if __name__ == "__main__":  
    knowledge_base = [  
        ["~P", "Q"],
```

```
["P"],  
["~Q", "R"],  
["~R"]  
]  
query = ["R"]  
print("Knowledge Base:", knowledge_base)  
print("Query:", query)  
result = resolution(knowledge_base, query)  
if result:  
    print("The query is satisfiable.")  
else:  
    print("The query is not satisfiable.")
```

#### **OUTPUT:**

Knowledge Base: [['~P', 'Q'], ['P'], ['~Q', 'R'], ['~R']]

Query: ['R']

The query is satisfiable.

# IMPLEMENTATION OF BACKWARD CHAINING

## Aim:

To implement backward chaining.

## PROGRAM:

```
facts = {
```

```
    'a': True,
```

```
    'b': True,
```

```
    'c': False
```

```
}
```

```
rules = [
```

```
    ('d', ['a', 'b']),
```

```
    ('e', ['b', 'c']),
```

```
    ('f', ['d', 'e'])
```

```
]
```

```
def backward_chaining(goal, facts, rules):
```

```
    if goal in facts:
```

```
        return facts[goal]
```

```
    for rule in rules:
```

```
        head, body = rule
```

```
        if head == goal:
```

```
            if all(backward_chaining(fact, facts, rules) for fact in body):
```

```
                return True
```

```
return False
```

```
goal = 'f'
```

```
if backward_chaining(goal, facts, rules):
```

```
    print(f"The goal '{goal}' can be achieved.")
```

```
else:
```

```
    print(f"The goal '{goal}' cannot be achieved.")
```

### **OUTPUT:**

Expected Output for the Goal 'f':

The goal 'f' cannot be achieved.

## IMPLEMENTATION OF FORWARD CHAINING

### Aim:

To implement forward Chaining.

### PROGRAM:

```
facts = {  
    'a': True,  
    'b': True,  
    'c': False  
}  
  
rules = [  
    ('d', ['a', 'b']),  
    ('e', ['b', 'c']),  
    ('f', ['d', 'e'])  
]  
  
def forward_chaining(facts, rules, goal):  
    inferred = set(facts.keys())  
    new_inferred = set(facts.keys())  
    while new_inferred:  
        current_inferred = set()  
        for rule in rules:  
            head, body = rule  
            if head not in inferred and all(fact in inferred for fact in body):  
                current_inferred.add(head)
```

```
if current_inferred:
    inferred.update(current_inferred)
    new_inferred = current_inferred
else:
    new_inferred = set()
return goal in inferred
```

```
goal = 'f'
```

```
if forward_chaining(facts, rules, goal):
    print(f"The goal '{goal}' can be achieved.")
else:
    print(f"The goal '{goal}' cannot be achieved.")
```

### **OUTPUT:**

The goal 'f' cannot be achieved.



# IMPLEMENTATION OF BLOCKS WORLD PROGRAM

## Aim:

To implement binary search tree.

## PROGRAM:

```
class BlocksWorld:
```

```
    def __init__(self, num_blocks):
```

```
        self.state = [[block] for block in range(num_blocks)]
```

```
        self.num_blocks = num_blocks
```

```
    def display_state(self):
```

```
        for stack in self.state:
```

```
            print(f"Block(s) on stack: {stack}")
```

```
    def move(self, block, destination):
```

```
        source_stack = self.find_block(block)
```

```
        destination_stack = self.find_block(destination)
```

```
        if source_stack is None or destination_stack is None:
```

```
            print(f"Invalid block {block} or destination {destination}.")
```

```
            return
```

```
        source_stack.remove(block)
```

```
        destination_stack.append(block)
```

```
        self.display_state()
```

```
    def find_block(self, block):
```

```
    for stack in self.state:
        if block in stack:
            return stack
    return None
```

```
def goal_state(self, goal):
    self.state = goal
    print("Goal state set.")
    self.display_state()
```

```
def main():
    blocks_world = BlocksWorld(3)
    print("Initial state:")
    blocks_world.display_state()
```

```
goal = [[0, 1], [2]]
blocks_world.goal_state(goal)
```

```
print("\nPerforming Moves:")
blocks_world.move(0, 2)
blocks_world.move(1, 2)
blocks_world.move(2, 0)
```

```
if __name__ == "__main__":
    main()
```

## OUTPUT:

Initial state:

Block(s) on stack: [0]

Block(s) on stack: [1]

Block(s) on stack: [2]

Goal state set.

Block(s) on stack: [0, 1]

Block(s) on stack: [2]

Performing Moves:

Block(s) on stack: [1]

Block(s) on stack: [2, 0]

Block(s) on stack: [2, 0]

Block(s) on stack: [1]

Block(s) on stack: [2, 0, 1]

Block(s) on stack: []

# IMPLEMENTATION OF A FUZZY INFERENCE SYSTEM

## Aim:

To implement Fuzzy Inference System.

## PROGRAM:

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

temperature = np.arange(0, 41, 1)
fan_speed = np.arange(0, 11, 1)

temp_low = fuzz.trimf(temperature, [0, 0, 20])
temp_medium = fuzz.trimf(temperature, [10, 20, 30])
temp_high = fuzz.trimf(temperature, [20, 30, 40])

fan_low = fuzz.trimf(fan_speed, [0, 0, 5])
fan_medium = fuzz.trimf(fan_speed, [2, 5, 8])
fan_high = fuzz.trimf(fan_speed, [5, 10, 10])

plt.figure(figsize=(10, 6))

plt.subplot(2, 1, 1)
plt.plot(temperature, temp_low, label='Low')
plt.plot(temperature, temp_medium, label='Medium')
plt.plot(temperature, temp_high, label='High')
```

```
plt.title("Temperature Membership Functions")
plt.xlabel("Temperature (°C)")
plt.ylabel("Membership Degree")
plt.legend()
```

```
plt.subplot(2, 1, 2)
plt.plot(fan_speed, fan_low, label='Low')
plt.plot(fan_speed, fan_medium, label='Medium')
plt.plot(fan_speed, fan_high, label='High')
plt.title("Fan Speed Membership Functions")
plt.xlabel("Fan Speed")
plt.ylabel("Membership Degree")
plt.legend()
```

```
plt.tight_layout()
plt.show()
```

```
temperature_input = 28
```

```
temp_low_level = fuzz.interp_membership(temperature, temp_low, temperature_input)
temp_medium_level = fuzz.interp_membership(temperature, temp_medium,
temperature_input)
temp_high_level = fuzz.interp_membership(temperature, temp_high, temperature_input)
```

```
fan_low_level = temp_low_level
fan_medium_level = temp_medium_level
fan_high_level = temp_high_level
```

```
fan_output = fuzz.defuzz(fan_speed, fan_low * fan_low_level + fan_medium *  
fan_medium_level + fan_high * fan_high_level, 'centroid')
```

```
print(f"Temperature: {temperature_input}°C")
```

```
print(f"Fuzzified fan speed: {fan_output:.2f}")
```

### **OUTPUT:**

Temperature: 28°C

Fuzzified fan speed: 7.25