B.M.S. COLLEGE OF ENGINEERING BENGALURU
Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Harsh Kumar
1BM21CS261

Department of Computer Science and Engineering
B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
Nov-Mar 2024

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Harsh Kumar (1BM21CS261) during the 5th Semester Nov-March-2024.

Signature of the Faculty Incharge:

Prof. Shravya AR
Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

1. Tic-Tac-Toe

Code:

Tic Tac Toe

```
def initialize_board():
        return [[' ' for _ in range(3)]
                for _ in range(3)]

def display_board(board):
        for row in board:
                print('|'.join(row))
                print('-' * 5)
def is_winner(board, player):
        for i in range(3):
        if all(board[i][j] == player
        for j in range(3)) or all(
        board[j][i] == player for j in
        range(3)):
                return true
        if (all(board[i][i] == player for
        i in range(3)) or all(
        board[i][2-i] == player for
        i in range(3)):
                return true
        return false
```

```
def is_board_full(board):
    return all(board[i][j]
        = ' ' for i in
    range(3) for j in range(3))

def tic_tac_toe():
    board = initialize_board()
    current_player = 'X'
    while true:
        display_board(board)
    row = int(input(f"player {
        current_player}, Enter row
        (0, 1 or 2): "))

    col = int(input("Enter column (0, 1,
                or 2): "))

    if board[row][col] == ' ':
        board[row][col]
            = currentplayer.

    if is_winner(board, current_player):
        display_board(board)
        print(f"player {current_player}
            wins!")
```

```
            break
elif is_board_full(board):
        display_board(board)
        print("its a draw!")
        break
else
        current_player = 'O'
        if current_player == 'X' else
        'X'
else
        print("invalid move. Try
            again")
if __name__ == "__main__"
    tic_tac_toe()
```

O/P

Harish kumar

welcome to tic tac toe :-

```
   |   |
   |   |
   |   |
 ___  ___  ___
```

Output:

```
Would you like to go first or second? (1/2)
1
   |   |
---+---+---
   |   |
---+---+---
   |   |

Player move: (0-8)
0
 o |   |
---+---+---
   |   |
---+---+---
   |   |

 o |   |
---+---+---
   | x |
---+---+---
   |   |

Player move: (0-8)
1
 o | o |
---+---+---
   | x |
---+---+---
   |   |
```

```
 O | O | X
---+---+---
   | X |
---+---+---
   |   |

Player move: (0-8)
6
 O | O | X
---+---+---
   | X |
---+---+---
 O |   |

 O | O | X
---+---+---
 X | X |
---+---+---
 O |   |

Player move: (0-8)
5
 O | O | X
---+---+---
 X | X | O
---+---+---
 O |   |

 O | O | X
---+---+---
 X | X | O
---+---+---
 O |   | X
```

```
Player move: (0-8)
7
 O | O | X
---+---+---
 X | X | O
---+---+---
 O | O | X
The game was a draw.
```

2. 8 Puzzle Breadth First Search Algorithm

Code:

prog-2 Breadth first search using some 8 block puzzle

Algorithm :-

1) define the puzzle :
represent a puzzle as 3x3 grid where each number represents a tile and 0 represents empty space

2) move the blank space :-
→ identify the position of blank spod (0) in the grid
→ we will then define movements left, right, up, down
→ create a new grid by moving the blank space in each moving valid direction

3) Check goal state :
→ determine that current config. matches the desired state. Solution.

4) Breadth-first search

* use a queue to explore possible moves systematically.
* start with an initial grid and an empty path.
* while there are configurations in the queue
  → take first config from the queue
  → check if it matches the goal state
  → if yes the path to this config is the solution
  → if not, generate new configuration by moving blank space and add them to queue

5) Repeat until solution
  → repeat until a solution is found or all possible configurations are explored.

6) print solution
  → if a solution is found print the path taken to reach the goal state.

Output:

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

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

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

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

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

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

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

3. 8 Puzzle Iterative Deepening Search Algorithm

Code:

code for 8 puzzle using iterative
deeping search function

Code :-

```
def iterative_depening_search(initial state,
                               goal_state)
depth_limit = 0
while True :
  result = depth_limited_search(initial_state,
              goal state, depth_limit)
  if result == "goal found" :
      return "goal found"
  elif result == "cutoff" :
      depth_limit += 1
  elif result == "failure" :
      return "goal not reachable"

def depth_limited_search(state, goal_state,
                          depth_limit):
    return recursive_dls( state,
                          goal_state,
                          depth_limit

def recursive_dls (state, goal_state,
                   depth_li
```

```
if state == goal_state :
    return "goal found"
elif depth_limit == 0:
    return "cutoff"
else
    cutoff_occurred = False
    for successor in generate_successors(state):
        result = recursive_dls(successor,
                    goal_state, depth_limit)
        if result == "goal_found":
            return "goal_found"
        elif result == "cutoff":
            cutoff_occurred = True
    if cutoff occurred:
        return "cutoff"
    else:
        return "failure"
def generate_successors(state):

    pass
```

Output:

```
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _



  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

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

4. 8 Puzzle A* Search Algorithm

Code:

A* Search Algorithm
for 8 block puzzle

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None,
                 move=None, cost=0,
                 heuristic=0):

        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost +
                self.heuristic) <
               (other.cost + other.
                heuristic)

def astar_search(initial_state, goal_state)
    open_set = [PuzzleNode(initial_state,
                None, None, 0, manhattan_
                               distance
                (initial_state,
                 goal_state))]
```

```
def find_zero_position(state):
    for i in range(3)
        for j in range(3)
            if state[i][j] == 0:
                return (i, j)

def is_valid_move(position)
    return 0 <= position[0] < 3
        and 0 <= position[1] < 3

def move_tile(state, from_position,
              to_position):
    new_state = [row[:] for row
                 in state]
    x1, y1 = from_position
    x2, y2 = to_position
    new_state[x1][y1], new_state[x2][y2]
        = new_state[x2][y2], new_state[x1][y1]

def reconstruct_path(node):
    initial_state = [[1, 2, 3],
                     [8, 0, 4],
                     [7, 6, 5]]
    goal_state = [[1, 2, 3],
                  [8, 4, 0],
                  [7, 6, 5]]
```

```
def find_goal_position (value, goal_state):
    for i in range(3):
        for j in range(3):
            if goal_state[i][j] == value:
                return (i, j)

def generate_successors(state):
    successor = [ ]
    zero_position = find_zero_position
                    (state)
    moves=[(-1,0), (1,0), (0,-1),(0,1)]

    for move in moves:
        new_position
          = (zero_position[0] +
             move[0] + zero_position[1]
             + move[1])
        if is_valid_move(new_position)
            successor = move_tiles (state,
                        zero_position, new
                        _position)
            successor.append(successor)

    return successors.
```

```
closed-set = set()
while open-set :
    current-node = heapq.heappop(open
                                 -set)
    if current_node.state == goal_state :
        return reconstruct_path(current
                                 _node)

    closed-set.add(tuple(map(tuple,
                   current_node.state)))

    for successor in generate-successor(
                        current_node.state):
        if tuple(map(tuple, successor)) not
                in closed-set :
            heap.heappush(open-set, puzzlenode)

return "goal not reachable"

def manhattan-distance(state, goal-state):
    distance = 0
    for i in range(3) :
        for j in range(3) :
            if state[i][j] != 0 :
                goal_position = find-goal-
                     position(state[i][j]
                              goal-state)

    distance += abs(i - goal_position[0])
               + abs(j - goal_position[1])
return distance
```

Output:

```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

5. Vacuum Cleaner

Code:

Vaccum Cleaner Week-6

vaccum cleaner to clean for 4 rooms

```
import random
def display(room):
    print(room)
room = [
    [1, 1],
    [1, 1]
]
print("All rooms are dirty")
display(room)
x=0
y=0
while x<2:
    while y<2:
        room[x][y] = random.choice(0,1)
        y+=1
    x+ =1
    y=0
print(" Before cleaning the room I detect
        all these random dirts")
display(room)
```

```
x = 0
y = 0

while x<2:
    while y<2:
        if room [x][y] =
            print ("vacum cleaner is
                  on this ", x, y)
            room [x][y] = 0
            print ("cleaned", x, y)

        y += 1
    x += 1
    y = 0

print ("Room is clean now")
display (room)
```

for a ~~[illegible]~~ ~~[illegible]~~
room changes were ile

~~room = [1, 1]~~
~~[illegible]~~

```
import random
def display(room):
        print(room)

room = [1, 1]
print("all rooms are dirty")
display(room)
x = 0
room[x] = random.choice(0,1)
    x += 1
print(before cleaning the room'!)
display(room)
x = 0
if room[x] == 1:
        print("vaccum cleaner is
                in this", x)
        room[x] = 0
        print("cleaned", x)

    x += 1
print('room is clean now')
display(room)
```

O/P

All rooms are dirty

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

After cleaning the rooms i detect these random dirt

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

Vaccum cleaner is at location 0 0

Cleaned 0 0

Vaccum cleaner is at location 1 0

Cleaned 1 0

Vaccum in this location 1 1

Cleaned 1 1

Room is cleaned now

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

22/12/23

Output:

```
Enter clean status for Room 1 (1 for dirty, 0 for clean): 1
Enter clean status for Room 2 (1 for dirty, 0 for clean): 1
[('Room 1', 1), ('Room 2', 1)]
Cleaning Room 1 (Room was dirty)
Room 1 is now clean.
Cleaning Room 2 (Room was dirty)
Room 2 is now clean.
Returning to Room 1 to check if it has become dirty again:
Room 1 is already clean.
Room 1 is clean after checking.
```

6. Knowledge Base Entailment

Code:

Knowledge based entailment

```
def weather_based_entailment(hypothesis,
                             premise, a, b, c):
#assume a, b and c are predetermined condition
if premise == "humid" and a > 70:
    return true
elif premise == "cloudy" and b > 50:
    return True
elif premise == "some_other_condition" and
                                     c > 30:
    return True
else:
    return false
premise_condition = "humid"
hypothesis_text = "The weather is uncomfortable."
humidity_condition = 75
cloudiness_condition = 60
some_other_condition = 40
result = weather_based_entailment(hypothesis_text,
        premise_condition, humidity_condition,
        cloudiness_condition, some_other_
        condition)
if result:
    print(f"The hypothesis is entailed by
           the {premise_condition}
           condition.")
```

```
else:
    print(f"The hypothesis is not entailed by
        the {premise_condition}
        condition.")
```

O/P

[The hypothesis is entailed by the premise condition.]

Output:

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

7. Knowledge Base Resolution

Code:

Knowledge Based Resolution

[KB = P, ¬P∨Q, P∨¬Q∨R, ¬Q∨R]

```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import P, Q, R

def main():
  try:
    expression1 = P
    Expression2 = Or(Not(P), Q)
    Expression3 = Or(P, NOT(Q), R)
    Expression4 = Or(Not(Q), R)
    knowledge_base = Or(Expression1,
    expression2, expression3, expression4)
    print("Knowledge Base:")
    print(knowledge_base)
    resolved_kb = knowledge_base.simplify()
    print("\n Resolved knowledge Base:")
    print(resolved_kb)
    negation_of_P = NOT(P)
    negation_of_Q = NOT(Q)
    negation_of_R = Not(R)
```

```
    print("\n Negation of P:")
    print(negation_of_P)
    print("Negation of Q:")
    print(negation_of_Q)
    print("Negation of R:")
    print(negation_of_R)

except Exception as e:
    print(f"An error occurred: {e}")

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

Output

knowledge base:

P & (Q | ~P) & (R | ~Q) & (P | R | ~Q)

Resolved knowledge base:

~~[illegible]~~ True

Negation of P:

~P

Negation of Q:

~Q

Negation of R:

~R

29.12.22

Output:

```
Step    |Clause |Derivation
-----------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

```python
rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
goal = 'R'
main(rules, goal)
```

8. Unification

Code:

Implemenet unification in first order logic

```
def unify-var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta

def unify(x, y, theta={}):
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x[0].islower():
        return unify-var(x, y, theta)
    elif isinstance(y, str) and y[0].is lower():
        return unify var(y, x, theta)
```

```
elif isinstance(x, list) and isinstance(y, list):
    if len(x) != len(y):
        return None
    for xi, yi in zip(x, y):
        theta = unify(xi, yi, theta)
        if theta is None:
            return None
    return theta
else:
    return None
```

O/P

```
# sample input
expression1 = ['P', 'a', 'x']
expression2 = ['P', 'y', 'z']

result = unify(expression1, expression2)

print("input:")
print("Expression1:", expression1)
print("Expression 2:", expression2)
print("\n Output")

if result is not None:
    print("unification successful")
    print("substitution theta:", result)
```

Output:

```
exp1 = "knows(A,x)"
exp2 = "knows(y,Y)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('A', 'y'), ('Y', 'x')]
```

9. FOL to CNF

Code:

program - 9

Implement a given first-order logic statement into congnitive normal form (CNF)

```
from sympy import symbols, to_cnf,
                  parse_expr
def convert_to_cnf(logic_statement)
    parsed_statement = parse_expr
                        (logic_statement)

    cnf = to_cnf(parsed_statement)
    return cnf
if __name__ == "__main__"
logic_statement = "(P|~q) +
                   (~P|r)"
cnf_result = convert_
             to_cnf(logic_statement)
print("Original statement",
              logic_statement)
print("CNF form:", cnf_result)
```

Output:

```
print(fol_to_cnf("bird(x)=>~fly(x)"))
print(fol_to_cnf("∃x[bird(x)=>~fly(x)]"))
```

```
~bird(x)|~fly(x)
[~bird(A)|~fly(A)]
```

10. Forward reasoning

Code:

program10

→ Create a knowledge base consisting of first order logic statement & prove the given query using forward reasoning

```
from sympy import symbols,
    Eq, And, Or, Implies, ask,
    satisfiable

John, Mary, Alice, Bob = symbols
        ('John Mary Alice Bob')

Parent = symbols ('parent')
Grandparent = symbols ('Grandparent')

knowledge_base = [Eq (parent (John,
  Alice), True), Eq (parent (Mary,
  Alice), True), Eq (parent (Alice, Bob),
  True), Implies (parent (x, y), Grand
  parent (x, y)), ]
query = Grandparent (John, Bob)
def forward_reasoning (knowledge_
                       base, query):
    new_facts = set()
```

```
while True:
    for fact in knowledge base:
        if ask(fact):
            continue

    if satisfiable(fact):
        new_facts.add(fact)
    if not new_facts:
        break
    knowledge_base.extend(new_facts)
    return ask(query)
result = forward_reasoning(knowledge_base, query)
print("Query", query)
print("Results", result)
```

O/P

Query: Grandparent(John, Bob)

Results: ~~f~~ True

19/1/24

Output:

```
kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono,America)')
kb.tell('owns(Nono,M1)')
kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

```
Querying criminal(x):
        1. criminal(West)
All facts:
        1. missile(M1)
        2. weapon(M1)
        3. enemy(Nono,America)
        4. owns(Nono,M1)
        5. hostile(Nono)
        6. criminal(West)
        7. american(West)
        8. sells(West,M1,Nono)
```