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:

Gauri Ramabhadran
1BM21CS066

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 Gauri Ramabhadran (1BM21CS066) during the 5th Semester Nov-March- 2024.

Signature of the Faculty Incharge:

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

Table of Contents

1. Tic-Tac-Toe

Code:

```
# Create a 3x3 tic tac toe board of "" strings for each value
board = [' '] * 9

# Create a function to display your board
def display_board(board):
    print(f" {board[0]} | {board[1]} | {board[2]} ")
    print("---+---+---")
    print(f" {board[3]} | {board[4]} | {board[5]} ")
    print("---+---+---")
    print(f" {board[6]} | {board[7]} | {board[8]} ")



#Create a function to check if anyone won, Use marks "X" or "O"
def check_win(player_mark, board):
    win = [f'{player_mark}'] * 3
    return board[:3] == win or board[3:6] == win or board[6:9] == win or \
        [board[0], board[4], board[8]] == win or [board[2], board[4], board[6]] == win or \
        [board[0], board[3], board[6]] == win or [board[1], board[4], board[7]] == win or [board[2],
board[5], board[8]] == win

def check_draw(board):
    return ' ' not in board

# Create a Function that makes a copy of the board
def board_copy(board):
    new_board = []
    for c in board:
        new_board += c
    return new_board

def test_win_move(move, player_mark, board):
```

```python
    copy = board_copy(board)

    copy[move] = player_mark
    return check_win(player_mark, copy)


def win_strategy(board):
    if board[4] == ' ':
        return 4
    for i in [0, 2, 6, 8]:
        if board[i] == ' ':
            return i
    for i in [1, 3, 5, 7]:
        if board[i] == ' ':
            return i

def get_agent_move(board):
    for i in range(9):
        if board[i] == ' ' and test_win_move(i, 'X', board):
            return i
    for i in range(9):
        if board[i] == ' ' and test_win_move(i, 'O', board):
            return i
    return win_strategy(board)

def tictactoe():
    playing = True
    while playing:
        in_game = True
        board = [' '] * 9
        print('Would you like to go first or second? (1/2)')
        choice = input()
        player_marker = 'O' if choice == '1' else 'X'
        display_board(board)
```

```
while in_game:
    print('\n')

    if player_marker == 'O':
        print('Player move: (0-8)')
        move = int(input())
        if board[move] != ' ':
            print('Invalid move')
            continue
    else:

        move = get_agent_move(board)
    board[move] = player_marker
    if check_win(player_marker,board):
        in_game = False
        display_board(board)
        if player_marker == 'O':
            print('O won')
        else:
            print('X won')
        break
    if check_draw(board):
        in_game = False
        display_board(board)
        print('The game was a draw.')
        break
    display_board(board)
    if player_marker == 'O':
        player_marker = 'X'
    else:
        player_marker = 'O'
print('Continue playing? (y/n)')
ans = input()
```

```python
        if ans not in 'yY':
            playing = False

# Play!!!
tictactoe()
```

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:

```
import numpy as np
import pandas as pd
import os

def gen(state, m, b):
    temp = state.copy()

    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    elif m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    elif m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    elif m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp  # Return the modified state

def possible_moves(state, visited_states):
    b = state.index(0)
    d = []

    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')

    pos_moves_it_can = []
```

```
    for i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in visited_states]

def bfs(src, target):
    queue = []
    queue.append(src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print(source[0],'|',source[1],'|',source[2])
        print(source[3],'|',source[4],'|', source[5])
        print(source[6],'|', source[7],'|',source[8])
        print()

        if source == target:
            print("success")
            return

        poss_moves_to_do = possible_moves(source, exp)

        for move in poss_moves_to_do:
            if move not in exp and move not in queue:
                queue.append(move)

src = [1, 2, 3, 4, 5, 6, 0, 7, 8]
target = [1, 2, 3, 4, 5, 6, 7, 8, 0]
bfs(src, target)
```

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:

```
def id_dfs(puzzle, goal, get_moves):
    import itertools
#get_moves -> possible_moves
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move], depth - 1)
                if next_route:
                    return next_route

    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route

def possible_moves(state):
    b = state.index(0)  # ) indicates White space -> so b has index of it.
    d = []  # direction

    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')

    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
```

```
    pos_moves = []
    for i in d:
        pos_moves.append(generate(state, i, b))
    return pos_moves


def generate(state, m, b):
    temp = state.copy()

    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp

# calling ID-DFS
initial = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

route = id_dfs(initial, goal, possible_moves)


if route:
    print("Success!! It is possible to solve 8 Puzzle problem")
    print("Path:", route)
else:
     print("Failed to find a solution")
```

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:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
```

```
            return temp_puz
        else:
            return None



    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """



        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j

class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
```

```
    puz = []

    for i in range(0,self.n):
        temp = input().split(" ")
        puz.append(temp)
    return puz

def f(self,start,goal):
    """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
    return self.h(start.data,goal)+start.level

def h(self,start,goal):
    """ Calculates the different between the given puzzles """
    temp = 0
    for i in range(0,self.n):
        for j in range(0,self.n):
            if start[i][j] != goal[i][j] and start[i][j] != '_':
                temp += 1
    return temp

def process(self):
    """ Accept Start and Goal Puzzle state"""
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()

    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    """ Put the start node in the open list"""
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
```

```
            print("")



            print("  | ")
            print("  | ")
            print(" \\'/ \n")

            for i in cur.data:
                for j in i:
                    print(j,end=" ")

                print("")

            """ If the difference between current and goal node is 0 we have reached the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)
puz = Puzzle(3)
puz.process()
```

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:

For 2 rooms:

```
def clean_room(room_name, is_dirty):
    if is_dirty:
        print(f"Cleaning {room_name} (Room was dirty)")
        print(f"{room_name} is now clean.")
        return 0  # Updated status after cleaning
    else:
        print(f"{room_name} is already clean.")
        return 0  # Status remains clean

def main():
    rooms = ["Room 1", "Room 2"]
    room_statuses = []

    for room in rooms:
        status = int(input(f"Enter clean status for {room} (1 for dirty, 0 for clean): "))
        room_statuses.append((room, status))
    print(room_statuses)

    for i, (room, status) in enumerate(room_statuses):
        room_statuses[i] = (room,clean_room(room, status)) # Update status after cleaning

    print(f"Returning to {rooms[0]} to check if it has become dirty again:")
    room_statuses[0]= (rooms[0],clean_room(rooms[0], room_statuses[0][1])) # Checking Room 1 after cleaning all rooms

    print(f"{rooms[0]} is {'dirty' if room_statuses[0][1] else 'clean'} after checking.")

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

For 4 rooms :

Code:

```
def clean_room(floor, room_row, room_col):
    if floor[room_row][room_col] == 1:
        print(f"Cleaning Room at ({room_row + 1}, {room_col + 1}) (Room was dirty)")
        floor[room_row][room_col] = 0
        print("Room is now clean.")
    else:
        print(f"Room at ({room_row + 1}, {room_col + 1}) is already clean.")

def main():
    rows = 2
    cols = 2
    floor = [[0, 0], [0, 0]]  # Initialize a 2x2 floor with clean rooms

    for i in range(rows):
        for j in range(cols):
            status = int(input(f"Enter clean status for Room at ({i + 1}, {j + 1}) (1 for dirty, 0 for clean): 
"))
            floor[i][j] = status

    for i in range(rows):
        for j in range(cols):
            clean_room(floor, i, j)

    print("Returning to Room at (1, 1) to check if it has become dirty again:")
    clean_room(floor, 0, 0)  # Checking Room at (1, 1) after cleaning all rooms

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

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:

```
from sympy import symbols, And, Not, Implies, satisfiable

def create_knowledge_base():
    # Define propositional symbols
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')
a
    # Define knowledge base using logical statements
    knowledge_base = And(
        Implies(p, q), # If p then q
        Implies(q, r), # If q then r
        Not(r) # Not r
    )

    return knowledge_base

def query_entails(knowledge_base, query):
    # Check if the knowledge base entails the query
    entailment = satisfiable(And(knowledge_base, Not(query)))

    # If there is no satisfying assignment, then the query is entailed
    return not entailment

if __name__ == "__main__":
    # Create the knowledge base
    kb = create_knowledge_base()

    # Define a query
    query = symbols('p')
```

```
# Check if the query entails the knowledge base
result = query_entails(kb, query)

# Display the results
print("Knowledge Base:", kb)
print("Query:", query)
print("Query entails Knowledge Base:", result)
```

Output:

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

7. Knowledge Base Resolution

Code:

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1

def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''

def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_terms('~PvR')

def contradiction(goal, clause):
    contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']
    return clause in contradictions or reverse(clause) in contradictions
```

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:


                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn
null. \
                                \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal}
is true."
                                return steps
                    elif len(gen) == 1:
```

```
                    clauses += [f'{gen[0]}']
                else:
                    if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                        temp.append(f'{terms1[0]}v{terms2[0]}')
                        steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \
                        \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal}
is true."
                        return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps


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

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:

```
def unify_var(var, x, theta):
    """
    Helper function for unifying a variable with a term.
    """
    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={}):
    """
    Unify two expressions x and y with the given substitution 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].islower():
        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

# Example usage:
x = ['P', 'a', 'x']
y = ['P', 'y', 'z']

result = unify(x, y)
print(result)

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

# Unify the expressions
result = unify(expression1, expression2)

# Display the result
print("Input:")
print("Expression 1:", expression1)
print("Expression 2:", expression2)

print("\nOutput:")
if result is not None:
    print("Unification Successful!")
    print("Substitution theta:", result)
else:
    print("Unification Failed.")
```

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:

```
from sympy import symbols, to_cnf, parse_expr

def convert_to_cnf(logic_statement):
    # Parse the logic statement
    parsed_statement = parse_expr(logic_statement)

    # Convert to CNF
    cnf = to_cnf(parsed_statement)

    return cnf

if __name__ == "__main__":
    # Example: (A & B) | (~C & D)
    logic_statement = "(A & B) | (~C & D)"

    # Convert to CNF
    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:

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

# Define individuals (family members)
John, Mary, Alice, Bob = symbols('John Mary Alice Bob')

# Define predicates
Parent = symbols('Parent')
Grandparent = symbols('Grandparent')

# Define knowledge base
knowledge_base = [
    Eq(Parent(John, Alice), True),
    Eq(Parent(Mary, Alice), True),
    Eq(Parent(Alice, Bob), True),
    Implies(Parent(x, y), Grandparent(x, y)),
]

# Define query
query = Grandparent(John, Bob)

# Perform forward reasoning
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)

# Check if the query is true based on the knowledge base
result = forward_reasoning(knowledge_base, query)

# Print the result
print("Query:", query)
print("Result:", result)
```

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)
```