KATHFORD
INTERNATIONAL COLLEGE OF ENGINEERING & MANAGEMENT
AFFILIATED TO TRIBHUVAN UNIVERSITY

Practical Report

Submitted by:

Sakar KC

BEI IV/I

Roll no.: 08

Subject: Artificial Intelligence

Submitted To:

Department of Electronics, Communication and Information Engineering

Date: July, 2079

LAB 1: VACUUM WORLD

OBJECTIVES:

1. To get familiar with problem solving.
2. To understand the concept of agent.

THEORY:

Problem Solving: It is the area of finding answers for unknown situation. Its process are:

i. Understanding

ii. Representation

iii. Formulation

iv. Solving

Its types are:

i. Simple: It can be solved using deterministic approach

ii. Complex: Lack of full information

Example:

1. **Vacuum World**

   a) Understanding:

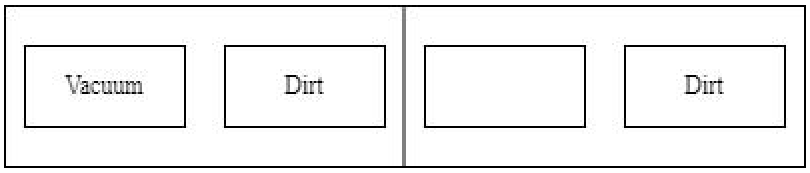
      i. n rooms and 1 vacuum cleaner.

      ii. Dirt can be in any room. It has to be cleaned.

      iii. Vacuum cleaner is present in any one room at a time.

      iv. Goal: clean all rooms

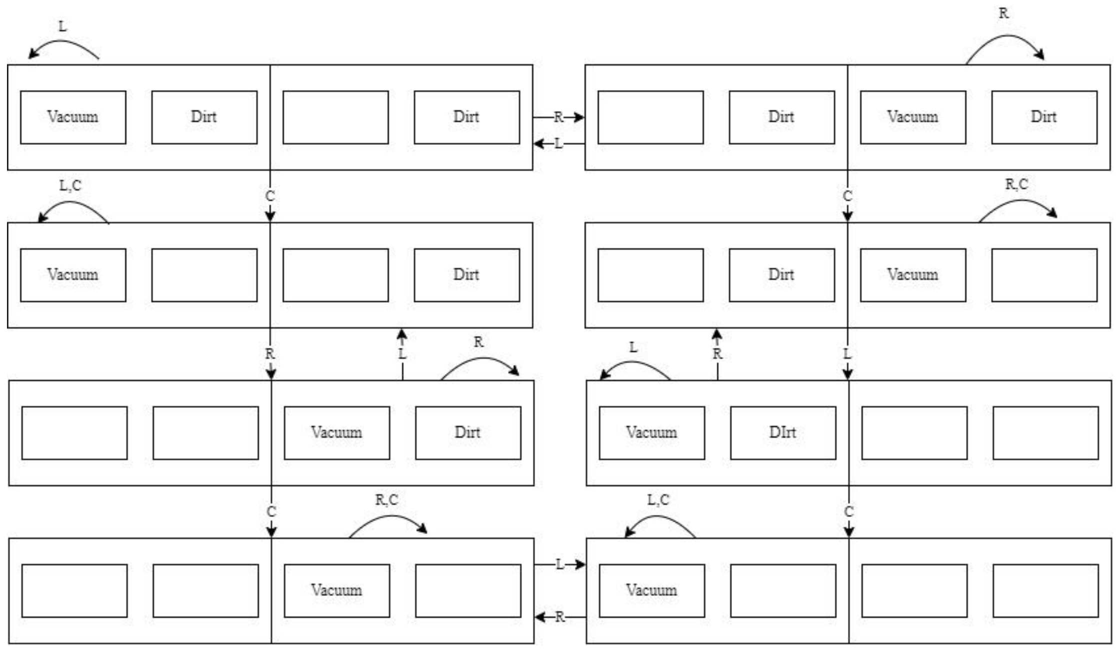
   b) Representation**:**



   c) Formulation**:**

      i. Possible action: move left, move right, and clean dirt

d) Solving:



IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python Code:

```
import copy
from glob import glob

### Variable declaration
initialState = []
initialState1 = []
states = []
goalState = []
latestState =[]
numOfState = 0
roomWithVacuum = []

### clean the room with vacuum
def clean(state, k, roomWithVacuum, i):
	global states
	global goalState
	for j in range(i,k):
		if list(state[0][j].keys())[1] == roomWithVacuum:
			if state[0][j][roomWithVacuum] == '1':
				temp = copy.deepcopy(state[0])
				states.append(temp)
				states[-1][j][roomWithVacuum] = '0'
				break
			else:
				break

### move vacuum to the room in right
```

```
def moveRight(state, k, roomWithVacuum, i):
    global states
    p = 0
    for j in range(i, k):
        if list(state[0][j].keys())[1] == roomWithVacuum:
            p = j     # we have to find the position of the room with vacuum
            break     # so that we will figure out if we can go right of left
        else:
            continue
    if len(states[0]) > p:
        temp = copy.deepcopy(states[-1])
        states.append(temp)
        states[-1][p][list(state[0][p+1].keys())[0]] = '0'
        states[-1][p+1][list(state[0][p+1].keys())[0]] = '1'
    return states

### move vacuum to the room in right
def moveLeft(state, k, roomWithVacuum, i):
    global states
    p = 0
    for j in range(i, k):
        if list(state[0][j].keys())[1] == roomWithVacuum:
            p = j     # we have to find the position of the room with vacuum
            break     # so that we will figure out if we can go right of left
        else:
            continue
    if p != 0:
        temp = copy.deepcopy(states[-1])
        states.append(temp)
        states[-1][p][list(state[0][p-1].keys())[0]] = '0'
        states[-1][p-1][list(state[0][p-1].keys())[0]] = '1'
    return states

### Recursive function to find room name with vacuum
def findVacuum(state, k, i):
    room = list(state[0][i].keys())
    if i < k:
        if state[0][i][room[0]] == '1':
            # print("Vacuum is in room " + room[1])
            return [i, room[1]]
        else:
            i = i+1
            return findVacuum(state, k, i)
    else:
        pass

def vacuumWorld():
    ### Define Initial States
    intNumOfRoom = int(input("Enter the number of room: "))
    for i in range(0, intNumOfRoom):
        roomName = input("Enter room name: ")
        # clean/dirty (0/1)
        dirtStatus = input("Enter status of dirt in " + roomName + " (clean/dirty - 0/1): ")
        # absent/present (0/1)
        vacuumStatus = input("Enter status of vacuum in", roomName, "(absent/present - 0/1): ")
        # representation: [{}, {}, ...]
        initialState.append({'V': vacuumStatus, roomName: dirtStatus})
    # Final representation: [[{}, {}, ...], [{}, {}, ...], ...]
    initialState1.append(initialState)
    ### Initial States complete

    ### define Goal States:
    global goalState
    goalState = copy.deepcopy(initialState1)
    for i in range(0, intNumOfRoom):
        for j in range(0, intNumOfRoom):
            if i == j:
                goalState[i][j][list(goalState[i][j].keys())[1]] = '0'
                goalState[i][j][list(goalState[i][j].keys())[0]] = '1'
                continue
            else:
                goalState[i][j][list(goalState[i][j].keys())[0]] = '0'
                goalState[i][j][list(goalState[i][j].keys())[1]] = '0'
        temp = copy.deepcopy(goalState[0])
        goalState.append(temp)
    goalState.pop(-1)
    ### Goal States complete

    global states
    global latestState
    global roomWithVacuum
    states = copy.deepcopy(initialState1)
    i = 0
    roomWithVacuum = findVacuum(initialState1, intNumOfRoom, i=0)
    # Case 1:
    if roomWithVacuum[0] == 0:
        while not (states[-1] in goalState):
            latestState.append(states[-1])
            roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
            clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
            if list(states[-1][-1].keys())[1] != roomWithVacuum[1]:
                if len(states) > i:
                    i += 1
                    latestState.pop(-1)
                    latestState.append(states[-1])
                    moveRight(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                i += 1
                latestState.pop(-1)
    # Case 2:
    elif roomWithVacuum[0] == len(initialState1[0])-1:
        while not (states[-1] in goalState):
            latestState.append(states[-1])
            roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
            clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
            if list(states[-1][0].keys())[1] != roomWithVacuum[1]:
                if len(states) > i:
                    i += 1
                    latestState.pop(-1)
                    latestState.append(states[-1])
                    moveLeft(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                i += 1
                latestState.pop(-1)
    # Case 3:
    elif (roomWithVacuum[0] - 0) >= (intNumOfRoom-roomWithVacuum[0]-1):
        while not (states[-1] in goalState):
            latestState.append(states[-1])
            roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
            clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
            if roomWithVacuum[0] == len(initialState1[0])-1:
                while not (states[-1] in goalState):
                    latestState.pop(-1)
                    latestState.append(states[-1])
                    roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
                    clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                    if list(states[-1][0].keys())[1] != roomWithVacuum[1]:
                        if len(states) > i:
                            i += 1
                            latestState.pop(-1)
                            latestState.append(states[-1])
                            moveLeft(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                        i += 1
                break
            if list(states[-1][-1].keys())[1] != roomWithVacuum[1]:
                if len(states) > i:
                    i += 1
                    latestState.pop(-1)
                    latestState.append(states[-1])
                    moveRight(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                i += 1
                latestState.pop(-1)
    # Case 4
    elif (roomWithVacuum[0] - 0) < (intNumOfRoom-roomWithVacuum[0]):
        while not (states[-1] in goalState):
            latestState.append(states[-1])
            roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
            clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
            if roomWithVacuum[0] == 0:
                while not (states[-1] in goalState):
                    latestState.pop(-1)
                    latestState.append(states[-1])
                    roomWithVacuum = findVacuum(latestState, intNumOfRoom, i=0)
                    clean(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                    if list(states[-1][-1].keys())[1] != roomWithVacuum[1]:
                        if len(states) > i:
                            i += 1
                            latestState.pop(-1)
                            latestState.append(states[-1])
                            moveRight(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                        i += 1
                break
            if list(states[-1][0].keys())[1] != roomWithVacuum[1]:
                if len(states) > i:
                    i += 1
                    latestState.pop(-1)
                    latestState.append(states[-1])
```

```
                        moveLeft(latestState, intNumOfRoom, roomWithVacuum[1], i=0)
                    i += 1
                    latestState.pop(-1)
    else:
        print("There shouldn't be any problem. I don't know why you get this message.")
        print("I hope you figure out the bug and send a pull request. Thank you        ")

    print("Initial State: ", end="")
    print(states[0])
    states.pop(0)
    for i in range(0, len(states)):
        print("State ", i+1, end=": ")
        print(states[i])
    print("Optimum number of state: " + str(len(states)))

vacuumWorld()
```

<u>Output:</u>

```
Enter the number of room: 4
Enter room name: A
Enter status of dirt in A (clean/dirty - 0/1): 1
Enter status of vacuum in A(absent/present - 0/1): 0
Enter room name: B
Enter status of dirt in B (clean/dirty - 0/1): 1
Enter status of vacuum in B(absent/present - 0/1): 0
Enter room name: C
Enter status of dirt in C (clean/dirty - 0/1): 1
Enter status of vacuum in C(absent/present - 0/1): 1
Enter room name: D
Enter status of dirt in D (clean/dirty - 0/1): 1
Enter status of vacuum in D(absent/present - 0/1): 0
Initial State: [{'V': '0', 'A': '1'}, {'V': '0', 'B': '1'}, {'V': '1', 'C': '1'}, {'V': '0', 'D': '1'}]
State  1: [{'V': '0', 'A': '1'}, {'V': '0', 'B': '1'}, {'V': '1', 'C': '0'}, {'V': '0', 'D': '1'}]
State  2: [{'V': '0', 'A': '1'}, {'V': '0', 'B': '1'}, {'V': '0', 'C': '0'}, {'V': '1', 'D': '1'}]
State  3: [{'V': '0', 'A': '1'}, {'V': '0', 'B': '1'}, {'V': '0', 'C': '0'}, {'V': '1', 'D': '0'}]
State  4: [{'V': '0', 'A': '1'}, {'V': '0', 'B': '1'}, {'V': '1', 'C': '0'}, {'V': '0', 'D': '0'}]
State  5: [{'V': '0', 'A': '1'}, {'V': '1', 'B': '1'}, {'V': '0', 'C': '0'}, {'V': '0', 'D': '0'}]
State  6: [{'V': '0', 'A': '1'}, {'V': '1', 'B': '0'}, {'V': '0', 'C': '0'}, {'V': '0', 'D': '0'}]
State  7: [{'V': '1', 'A': '1'}, {'V': '0', 'B': '0'}, {'V': '0', 'C': '0'}, {'V': '0', 'D': '0'}]
State  8: [{'V': '1', 'A': '0'}, {'V': '0', 'B': '0'}, {'V': '0', 'C': '0'}, {'V': '0', 'D': '0'}]
Optimum number of state: 8
```

DISCUSSION:

Firstly, we learned the concepts of intelligent agents and its type. We understand that agent function maps from percept histories to actions. Thus, we take the vacuum world problem to understand the agent more clearly. In the code: ‘01vacuumWorld’, I have used the rational agent approach, i.e, simple reflex agent. The function ‘vacuumWorld()’ is the agent function in the program. The function like ‘clean()’, ‘moveRight()’, and ‘moveLeft()’, are the actuator in this program. Finally, condition-action (if-then) rules are figured out for the problem and then implemented in the ‘vacuumWorld()’.

Algorithm:

1. At first, problem instances (initial state & goal state) are defined.
2. Find the room with a vacuum
   a) If the vacuum is in 1st room, clean the room and move right and clean until it reaches the last room.
   b) If the vacuum is in the last room, clean the room and move left and clean until it reaches the first room.
   c) If vacuum is in between first and last
      i. If (index of room with a vacuum - 0) is less than (number of room - room with vacuum), then move left and clean the room.
      ii. Else, move right and clean the room.

CONCLUSION:

After a brief study and analysis of the agents and their type, I implemented a simple reflex agent for the vacuum world problem. I have used the rational agent approach. Thus, this program gives the number of states that a vacuum takes to clean all the rooms (n rooms). The output also provides every next state after any one actuator acts. Thus it can be used for real-world vacuum cleaner robots. The performance measure of this program is the number of optimal state vacuums taken to clean all the rooms.

LAB 2: DEPTH-FIRST SEARCH & BREAD-FIRST SEARCH

OBJECTIVES:

1. To get familiar with searching.
2. To understand the concept of basic searching strategies, like depth-first search & breadth-first search.

THEORY:

Depth-First Search (DFS):

It proceeds down a single brah of the tree at a time. It expands the root node, then the leftmost child of the root node, and so on. It always expands a node at the deepest level of the tree. Only when the search hits a dead end, search backtrack is done and expands node at higher levels.

It uses stack to keep track of nodes, i.e LIFO approach.

Its time complexity is O(b^m) and

Its space complexity is O(bm)

where b is the maximum branching factor of the search tree and m is the maximum depth of the state space.

Breadth-First Search (BFS):

It proceeds level by level down the search tree. Starting from the root node, (initial state) explores all children of the root node left to right; if no solution is found, expand the root node's first (leftmost) child. Then expand the second node and its children similarly.

BFS forms a queue where nodes are expanded in FIFO manner.

Its time complexity is O(b^{d+1}) and

Its space complexity is O(b^{d+1})

where b is the maximum branching factor of the search tree and d is the level of goal node in the search table.

IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python code for BFS:

```
#Using a Python dictionary to act as an adjacency list
graph = {
   'A' : ['B','C'],
   'B' : ['D', 'E'],
   'C' : ['F'],
   'D' : [],
   'E' : [],
   'F' : [],
}

visitedBFS = [] # List to keep track of visited nodes.
visitedDFS = set()
queue = []          #Initialize a queue

def bfs(visited, graph, node):
   visited.append(node)
   queue.append(node)
   print("Breadth-First Search:", end=" ")
   while queue:
      s = queue.pop(0)
      print (s, end = " ")
      for neighbour in graph[s]:
         if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)

def dfs(visited, graph, node):

      if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbour in graph[node]:
                  dfs(visited, graph, neighbour)

# Driver Code
bfs(visitedBFS, graph, 'A')
print("\nDepth-First Search:", end=" ")
dfs(visitedDFS, graph, 'A')
```

Output:

```
Breadth-First Search: A B C D E F
Depth-First Search: A B D E C F
```

DISCUSSION:

Firstly, we learned the concepts of searching. Its different terminologies like: search space, problem instance (initial state & goal state), problem space, searching strategies, search tree, etc. Among many searching strategies, we implemented the basic DFS & BFS in code. *'02DFS.py'* and *'02BFS.py'* are the two programs for DFS and BFS respectively in the repository. The programs starts with a defined graph and its output shows in what order nodes in the graph are visited.

CONCLUSION:

After a brief study of searching techniques, DFS and BFS are the basic searching technique to get started. BFS and DFS will get the result eventually if the searched node is present in the graph. But if we choose them in the wrong application, they will take high time and memory management. BFS & DFS are used in path-finding algorithms, like BFS in the Ford-Fulkerson algorithm and DFS, in solving puzzles with only one solution, such as a maze or a sudoku puzzle.

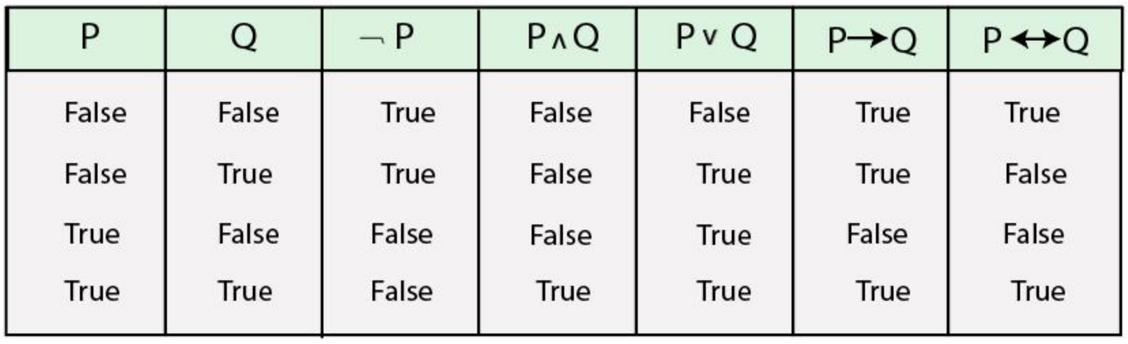
LAB 3: PROPOSITIONAL LOGIC

OBJECTIVES:

1. To get familiar with propositional logic.
2. To understand the concept of conjunction (AND), disjunction (OR), negation (NOT), Implication (conditional), bi-implication (bi-conditional) statement.

THEORY:

Propositional Logic: It is a system based on propositions. It, also known as sentential logic and statement logic, is the branch of logic that studies ways of joining and/or modifying entire propositions, statements or sentences to form more complicated propositions, statements or sentences, as well as the logical relationships and properties that are derived from these methods of combining or altering statements.



P	Q	¬ P	P∧Q	P∨Q	P→Q	P↔Q
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

Truth tables for five logical Connectives

The signs '¬', '^', 'v', '→' & '<->' correspond respectively the truth-functions negation, conjunction, disjunction, implication and bi-implication.

IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python Code

```
# AND
def AND(A, B):
   print("\nConjunction (AND)")
   print("The output of", A, "and", B, "is:", A and B)

# OR
def OR(A, B):
   print("\nDisjunction (OR)")
   print("The output of", A, "or", B, "is:", A or B)

# NOT
```

```
def NOT(A):
    print("\nNegation (NOT)")
    print("The output of not", A, "is:", not A)

# Implies | Conditional
def conditional(A, B):
    print("\nImplication (conditional)")
    if A == 'True':
        print("If A =", A, "and B =", B, ", A->B =", B)
    else:
        print("If A =", A, "and B =", B, ", A->B =", True)

# If and only if | Biconditional
def biConditional(a, b):
    print("\nBi-implication (bi-conditional) statement")
    def conditional(a, b):
        if a == True:
            return b
        return True
    A = conditional(a, b)
    B = conditional(b, a)
    if A == B:
        print("If A =", A, "and B =", B, ", A<->B =", True)
    else:
        print("If A =", A, "and B =", B, ", A<->B =", False)

A = input("A: ")
B = input("B: ")
AND(A, B)
OR(A, B)
NOT(A)
conditional(A, B)
biConditional(A, B)
```

Output:

```
A: True
B: True

Conjunction (AND)
The output of True and True is: True

Disjunction (OR)
The output of True or True is: True

Negation (NOT)
The output of not True is: False

Implication (conditional)
If A = True and B = True , A->B = True
```

Bi-implication (bi-conditional) statement
If A = True and B = True , A<->B = True

DISCUSSION:

Firstly, we learned the concepts of propositional logic. The five logical connectives: conjunction (AND), disjunction (OR), negation (NOT), Implication (conditional), and bi-implication (bi-conditional) statements were understood and implemented in the code. The program output will be the same as shown in truth table in the figure above.

CONCLUSION:

After a brief study of propositional logic, its five connectives: conjunction (AND), disjunction (OR), negation (NOT), Implication (conditional), and bi-implication (bi-conditional) statements, they help in knowledge representation.

LAB 4: TOWER OF HANOI

OBJECTIVES:

1. To get familiar with the Tower of Hanoi puzzle.
2. To understand how to implement it in recursive function.

THEORY:

The Tower of Hanoi and sometimes called pluralized as Towers, or simply pyramid puzzle is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a disk that is smaller than it.

IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python Code

```
# Recursive Python function to solve the tower of hanoi
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
        print("Move disk 1 from source",source,"to destination",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print("Move disk",n,"from source",source,"to destination",destination)
    TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code
n = 3
TowerOfHanoi(n,'A','C','B')
# A, C, B are the name of rods
```

Output:

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
```

DISCUSSION:

Firstly, we learned the concepts of the puzzle called Tower of Hanoi (TOH). Then, it is implemented using recursive function as in the program. We also saw the output of the program gives the best result.

CONCLUSION:

After a brief study of the Tower of Hanoi puzzle, we implemented it in the python code.

LAB 5: FIRST-ORDER PREDICATE LOGIC (FOPL)

OBJECTIVES:

3. To get familiar with first-order predicate logic (FOPL).
4. To understand how to implement it in recursive function.

THEORY:

The first-order predicate logic (FOPL) is the backbone of AI, a method of formal representation of Natural Language (NL) text. The Prolog language for AI programming has its foundations in FOPL.

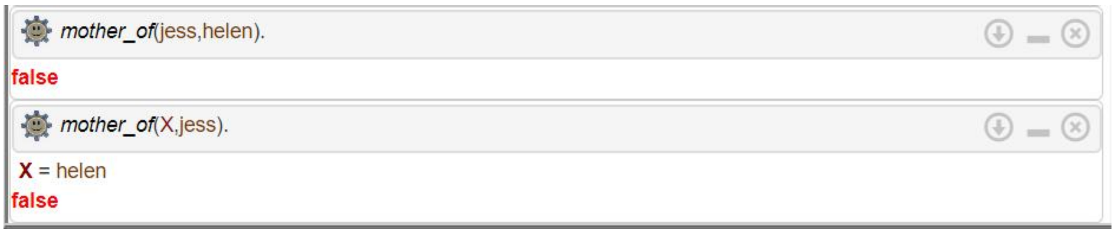
IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Prolog Code implemented in: https://swish.swi-prolog.org/

```
% type the following facts
/* Facts */
male(jack).
male(oliver).
male(ali).
male(james).
male(simon).
male(harry).
female(helen).
female(sophie).
female(jess).
female(lily).

parent_of(jack,jess).
parent_of(jack,lily).
parent_of(helen, jess).
parent_of(helen, lily).
parent_of(oliver,james).
parent_of(sophie, james).
parent_of(jess, simon).
parent_of(ali, simon).
parent_of(lily, harry).
parent_of(james, harry).

% type the following rules
/* Rules */
father_of(X,Y):- male(X),
      parent_of(X,Y).

mother_of(X,Y):- female(X),
      parent_of(X,Y).

grandfather_of(X,Y):- male(X),
        parent_of(X,Z),
        parent_of(Z,Y).

grandmother_of(X,Y):- female(X),
        parent_of(X,Z),
        parent_of(Z,Y).

sister_of(X,Y):- %(X,Y or Y,X)%
        female(X),
        father_of(F, Y), father_of(F,X),X \= Y.

sister_of(X,Y):- female(X),
        mother_of(M, Y), mother_of(M,X),X \= Y.

aunt_of(X,Y):- female(X),
        parent_of(Z,Y), sister_of(Z,X),!.

brother_of(X,Y):- %(X,Y or Y,X)%
        male(X),
        father_of(F, Y), father_of(F,X),X \= Y.

brother_of(X,Y):- male(X),
        mother_of(M, Y), mother_of(M,X),X \= Y.

uncle_of(X,Y):-
        parent_of(Z,Y), brother_of(Z,X).

ancestor_of(X,Y):- parent_of(X,Y).
ancestor_of(X,Y):- parent_of(X,Z),
        ancestor_of(Z,Y).
```

Output:



```
mother_of(jess,helen).
false
mother_of(X,jess).
X = helen
false
```

DISCUSSION:

Firstly, we learned the concepts of the first-order predicate logic (FOPL).
Initially, facts and rules were written in prolog language at https://swish.swi-prolog.org/. Then, different clauses were run as queries in the terminal. We saw the correct result from the program as well.

CONCLUSION:

After a brief study of the first-order predicate logic (FOPL) and its implementation in prolog, FOPL develops information about the objects more easily and can also express the relationship between those objects.

LAB 6: NATURAL LANGUAGE PROCESSING

OBJECTIVES:

5. To get familiar with natural language processing.
6. To implement sentence and word tokenization.

THEORY:

Natural language processing (NLP) refers to the branch of computer science, specifically, the branch of artificial intelligence (AI), concerned with giving computers the ability to understand text and spoken words in much the same way human beings can. NLP combines computational linguistics, and rule-based modeling of human language with statistics, machine learning, and deep learning models. Together, these technologies enable computers to process human language in the form of text or voice data and to 'understand' its full meaning, complete with the speaker or writer's intent and sentiment.

IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python Code

```
import nltk
nltk.download('punkt')

from nltk.tokenize import sent_tokenize
text = "My name is Sakar KC. I am awesome."
list1 = sent_tokenize(text) # Sentence Tokenization
print(list1)

from nltk.tokenize import word_tokenize

text = "My name is Sakar"
list2 = word_tokenize(text) # Word Tokenization
print(list2)
```

Output:

```
[nltk_data] Downloading package punkt to
[nltk_data]         C:\Users\stefe\AppData\Roaming\nltk_data...
[nltk_data]     Package punkt is already up-to-date!
['My name is Sakar KC.', 'I am awesome.']
['My', 'name', 'is', 'Sakar']
```

DISCUSSION:

Firstly, we learned the concepts of Natural language processing (NLP), then sentence and word tokenization were implemented.

CONCLUSION:

After a brief study of Natural language processing (NLP) and implementation of some of its features, we got a more comprehensive idea of how it works.

LAB 7: SIMPLE NEURAL NETWORK

OBJECTIVES:

7. To get familiar with Simple Neural Network.
8. To understand how to implement it in recursive function.

THEORY:

The Tower of Hanoi and sometimes called pluralized as Towers, or simply pyramid puzzle is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules:

4. Only one disk may be moved at a time.
5. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
6. No disk may be placed on top of a disk that is smaller than it.

IMPLEMENTATION:

Github link: https://github.com/sakarkc2122/practicalAI

Python Code

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K

(x_train, y_train), (x_test, y_test) = mnist.load_data()

num_of_trainImgs = x_train.shape[0] #60000 here
num_of_testImgs = x_test.shape[0] #10000 here
img_width = 28
img_height = 28

x_train = x_train.reshape(x_train.shape[0], img_height, img_width, 1)
x_test = x_test.reshape(x_test.shape[0], img_height, img_width, 1)
input_shape = (img_height, img_width, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

```
num_classes = 10
y_train = keras.utils.np_utils.to_categorical(y_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))


model.compile(loss=keras.losses.categorical_crossentropy,
              metrics=['accuracy'])


model.fit(x_train, y_train,
          batch_size=128,
          epochs=2,
          verbose=1,
          validation_data=(x_test, y_test))


score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Output:

```
Epoch 1/2
469/469 [==============================] - 69s 147ms/step - loss: 0.2238 -
accuracy: 0.9312 - val_loss: 0.0515 - val_accuracy: 0.9838
Epoch 2/2
469/469 [==============================] - 65s 138ms/step - loss: 0.0847 -
accuracy: 0.9750 - val_loss: 0.0405 - val_accuracy: 0.9868
Test loss: 0.040512848645448685
Test accuracy: 0.9868000149726868
```

DISCUSSION:

Firstly, we learned the concepts of the simple neural network. Then, it is implemented using python libraries like TensorFlow and Keras. TensorFlow focuses on the training and inference of deep neural networks whereas Keras acts as an interface for the TensorFlow library. The program calculates the test loss and test accuracy for the model.

CONCLUSION:

After a brief study of the simple neural network, we implemented it in the python code. Thus it gave a more comprehensive idea about the simple neural network.