SRM
INSTITUTE OF SCIENCE AND TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)
DELHI-NCR CAMPUS, GHAZIABAD (U.P.)

ARTIFICIAL INTELLIGENCE LAB

(Subject Code: (18CSC305J)

B.TECH III Year / VI Semester

NAME

REG. No.

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
FACULTY OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY,
DELHI NCR CAMPUS, MODINAGAR
SIKRI KALAN, DELHI MEERUT ROAD, DIST. – GHAZIABAD - 201204
www.srmup.in

Even Semester (JAN-2023)

BONAFIDE CERTIFICATE

Registration no: RA2011026030048

Certified to be the bonafide record of work done by **SURAJ JOSHI** *Of 6th semester 3rd year* **B.TECH** *degree course in* **SRM INSTITUTE OF SCIENCE & TECHNOLOGY**, *DELHI-NCR Campus for the Department of* ***Computer Science & Engineering,*** *in Artificial Intelligence Laboratory during the academic year* ***2022-23.***

MS. SHIVA SHONI
(Assistant Professor)

DEEPA ARORA
(Research Scholar)

Head of the department
(CSE)

Submitted for end semester examination held on___/___/___ at SRM INSTITUTE OF SCIENCE & TECHNOLOGY, DELHI-NCR Campus..

Internal Examiner-I

Internal Examiner-II

INDEX

Experiment 1

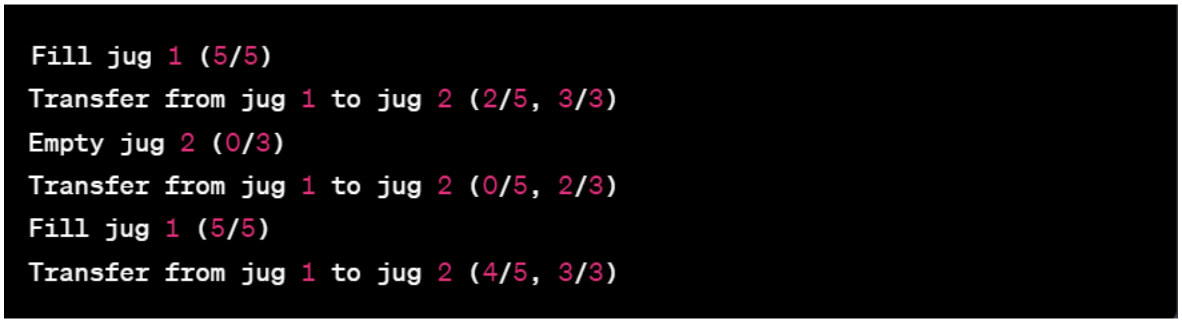
- **Aim –** Implementation of Toy problem
  Example-Implement water jug problem.
- **Algorithm –**

Rule	State	Process
1	(X,Y | X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y |Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y |X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y | Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y | X+Y>=4 ^ Y>0)	(4,Y-(4-X)) {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y | X+Y>=3 ^X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	(X,Y | X+Y<=4 ^Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y | X+Y <=3^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

- **Code –**

```
class WaterJugProblem:
    def __init__(self, jug1_capacity, jug2_capacity, target_capacity):
        self.jug1_capacity = jug1_capacity
        self.jug2_capacity = jug2_capacity
        self.target_capacity = target_capacity

    def solve(self):
        jug1 = 0
        jug2 = 0
        steps = []

        while jug1 != self.target_capacity and jug2 != self.target_capacity:
            if jug1 == 0:
                # fill jug1
                jug1 = self.jug1_capacity
                steps.append(f"Fill jug 1 ({jug1}/{self.jug1_capacity})")
            elif jug2 == self.jug2_capacity:
                # empty jug2
                jug2 = 0
                steps.append(f"Empty jug 2 ({jug2}/{self.jug2_capacity})")
            else:
                # transfer from jug1 to jug2
                amount = min(jug1, self.jug2_capacity - jug2)
                jug1 -= amount
                jug2 += amount
                steps.append(f"Transfer from jug 1 to jug 2 ({jug1}/{self.jug1_capacity}, {jug2}/{self.jug2_capacity})")

        return steps

problem = WaterJugProblem(jug1_capacity=5, jug2_capacity=3, target_capacity=4)
steps = problem.solve()
print(steps)
```

- **Result**



```
Fill jug 1 (5/5)
Transfer from jug 1 to jug 2 (2/5, 3/3)
Empty jug 2 (0/3)
Transfer from jug 1 to jug 2 (0/5, 2/3)
Fill jug 1 (5/5)
Transfer from jug 1 to jug 2 (4/5, 3/3)
```

Experiment 2

- **Aim –** Developing Agent Program for Real World Problem.

- **Algorithm –**

i. Create a MazeEnvironment class that represents the maze environment. This class should have methods for finding the position of the robot and the goal, moving the robot, and getting the current state of the maze.

ii. Create a MazeAgent class that represents the agent. This class should have methods for perceiving the environment, deciding on a direction to move, and taking action to move in that direction.

iii. In the MazeAgent class's run method, repeatedly perform the following steps until the robot reaches the goal:

iv. Perceive the current state of the maze environment.
v. Use the think method to decide on a direction to move.
vi. Use the act method to move the robot in the chosen direction.
vii. Create an instance of the MazeEnvironment class with a representation of the maze.

viii. Create an instance of the MazeAgent class.

ix. Call the run method of the MazeAgent instance with the MazeEnvironment instance as an argument to start the navigation.

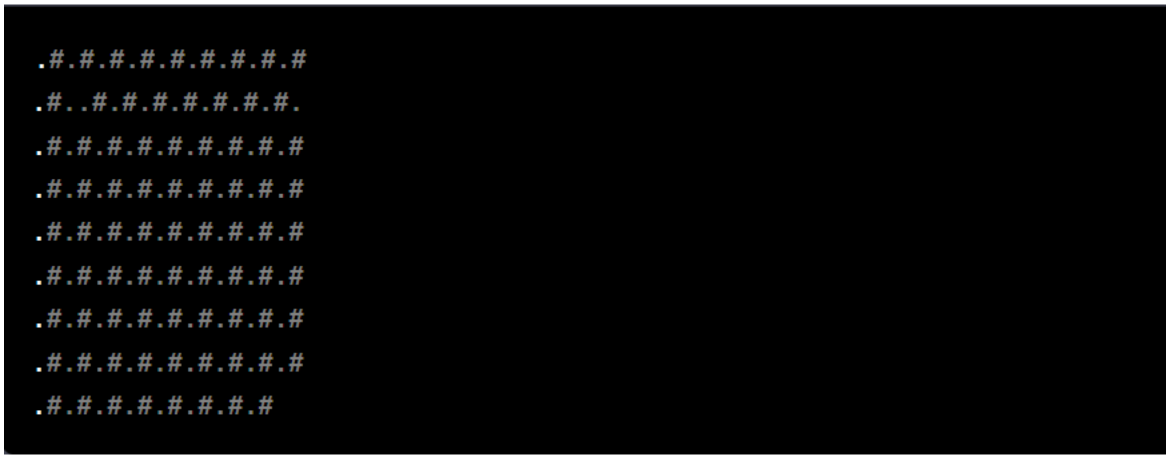
- **Code –**

```
import random

class MazeEnvironment:
def __init__(self, maze):
self.maze = maze
self.width = len(maze[0])
self.height = len(maze)
```

```
7.  self.robot_pos = self.find_robot()
8.  self.goal_pos = self.find_goal()

9.  def find_robot(self):
10.   for y in range(self.height):
        a. for x in range(self.width):
        b. if self.maze[y][x] == "R":
             i. return (x, y)

11.   def find_goal(self):
12.   for y in range(self.height):
        a. for x in range(self.width):
        b. if self.maze[y][x] == "G":
             i. return (x, y)

13.   def move_robot(self, dx, dy):
14.   x, y = self.robot_pos
15.   new_x = x + dx
16.   new_y = y + dy
17.   if self.is_valid_move(new_x, new_y):
        a. self.maze[y] = self.maze[y][:x] + "." + self.maze[y][x+1:]
        b. self.maze[new_y] = self.maze[new_y][:new_x] + "R" +
           self.maze[new_y][new_x+1:]
        c. self.robot_pos = (new_x, new_y)

18.   def is_valid_move(self, x, y):
19.   if x < 0 or y < 0 or x >= self.width or y >= self.height:
        a. return False
20.   if self.maze[y][x] == "#":
        a. return False
21.   return True

22.   def get_observation(self):
23.   return self.maze

24.   class MazeAgent:
25.   def __init__(self):
26.   self.directions = [(0, -1), (1, 0), (0, 1), (-1, 0)]
```

```
def perceive(self, observation):
self.maze = observation

def think(self):
direction = random.choice(self.directions)
dx, dy = direction
return dx, dy

def act(self, dx, dy):
self.environment.move_robot(dx, dy)

def run(self, environment):
self.environment = environment
observation = self.environment.get_observation()
while self.environment.robot_pos != self.environment.goal_pos:
    self.perceive(observation)
    dx, dy = self.think()
    self.act(dx, dy)
    observation = self.environment.get_observation()

maze = [
"#.#.#.#.#.#.#.#.#",
"#.R.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.#",
"#.#.#.#.#.#.#.#.G"
]
environment = MazeEnvironment(maze)
agent = MazeAgent()
agent.run(environment)
```

- **Result –**



```
.#.#.#.#.#.#.#.#.#
.#..#.#.#.#.#.#.#.
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#.#
.#.#.#.#.#.#.#.#
```

Experiment 3

- **Aim –** Implementation of Constraint satisfaction problemExample: Implement N- queen Problem
- **Algorithm –**

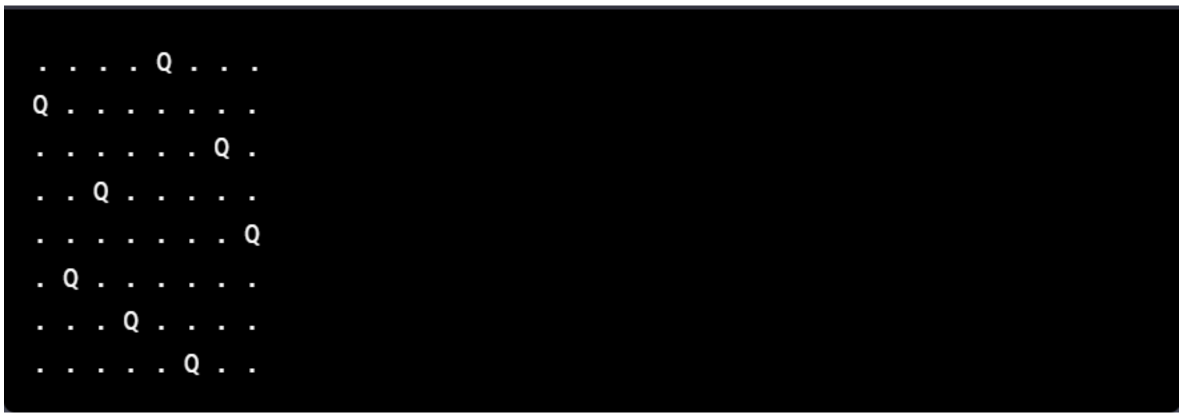
```
while there are untried configurations
{
  generate the next configuration
  if queens don't attack in this configuration then
  {
    print this configuration;
  }
}
```

- **Code –**

```
class NQueensCSP:
def __init__(self, n):
self.n = n
self.variables = list(range(n))
self.domains = {i: list(range(n)) for i in range(n)}
self.constraints = {(i, j): self.is_valid for i in range(n) for j in range(n) if i != j}

def is_valid(self, x, y):
return x != y and abs(x - y) != abs(self.variables.index(x) - self.variables.index(y))

def solve(self):
solution = {}
if self.backtrack(solution):
return solution
return None

def backtrack(self, solution):
if len(solution) == self.n:
return True

var = self.select_unassigned_variable(solution)
for val in self.order_domain_values(var):
if self.is_valid_assignment(var, val, solution):
solution[var] = val
if self.backtrack(solution):
return True
del solution[var]
```

```
return False

def select_unassigned_variable(self, solution):
return min([var for var in self.variables if var not in solution], key=lambda var: len(self.domains[var]))

def order_domain_values(self, var):
return sorted(self.domains[var], key=lambda val: sum(self.is_valid(val, solution[var]) for solution in [dict(list(solution.items()) + [(var, val)]) for val in self.domains[var]]))

def is_valid_assignment(self, var, val, solution):
return all((var2, val2) not in self.constraints or self.constraints[(var2, var)](val2, val) for var2, val2 in solution.items())

def print_solution(self, solution):
for row in range(self.n):
print(" ".join("Q" if solution.get(col, row) == row else "." for col in range(self.n)))

n = 8
csp = NQueensCSP(n)
solution = csp.solve()

if solution is not None:
csp.print_solution(solution)
else:
print("No solution found.")
```

- **Result –**



```
. . . . Q . . .
Q . . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
```

Experiment 4

- **Aim –** To Implementation and Analysis of BFS and DFS for Application.
- **Algorithm –**

1. Create a node list (Queue) that initially contains the first node N and mark it as visited.
2. Visit the adjacent unvisited vertex of N and insert it in a queue.
3. If there are no remaining adjacent vertices left, remove the first vertex from the queue mark itas visited, display it.
4. Repeat step 1 and step 2 until the queue is empty or the desired node is found.

- **Code –**

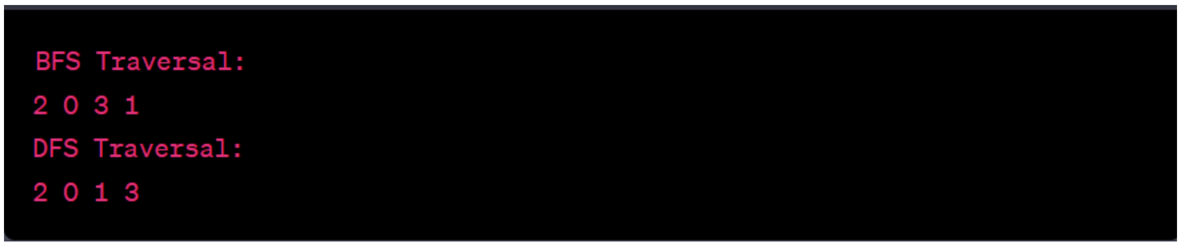
```
from collections import defaultdict

class Graph:
def __init__(self):
self.graph = defaultdict(list)

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

def bfs(self, start):
visited = [False] * len(self.graph)
queue = [start]
visited[start] = True

while queue:
        node = queue.pop(0)
        print(node, end=" ")
        for neighbor in self.graph[node]:
        if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)

def dfs(self, start):
visited = [False] * len(self.graph)
self._dfs_util(start, visited)

def _dfs_util(self, node, visited):
visited[node] = True
print(node, end=" ")
```

18.for neighbor in self.graph[node]:
    a. if not visited[neighbor]:
    b. self._dfs_util(neighbor, visited)

19.# Example usage
20.graph = Graph()
21.graph.add_edge(0, 1)
22.graph.add_edge(0, 2)
23.graph.add_edge(1, 2)
24.graph.add_edge(2, 0)
25.graph.add_edge(2, 3)
26.graph.add_edge(3, 3)

27.print("BFS Traversal:")
28.graph.bfs(2)
29.print("\nDFS Traversal:")
30.graph.dfs(2)

- **Result –**



```
BFS Traversal:
2 0 3 1
DFS Traversal:
2 0 1 3
```

Experiment 5

- **Aim-** To implement Best First Search and A* algorithm.
- **Algorithm-**

1. **Best First Search-**

Step 1: Place the starting node into the OPEN list.
Step 2: If the OPEN list is empty, Stop and return failure.
Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
    If node n is goal then return
else
Step 4: Expand the node n, and generate and check the successors of node n. and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 5.
Step 5: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
Step 6: Return to Step 2.

2. **A*-**

Step1: Place the starting node in the OPEN list.
Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
Step 4:Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
Step 6: Return to Step 2.

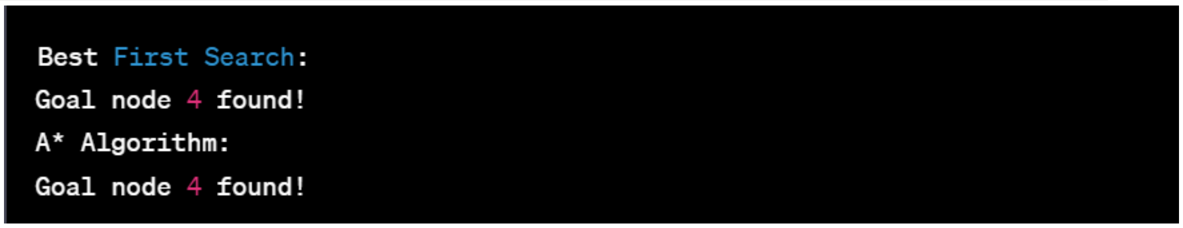
- **Code-**

```
from collections import defaultdict
from queue import PriorityQueue

class Graph:
def __init__(self):
self.graph = defaultdict(list)
self.edges = {}

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

def best_first_search(self, start, goal):
```

```
11. visited = set()
12. queue = PriorityQueue()
13. queue.put((self.heuristic(start, goal), start))

14. while not queue.empty():
        a. node = queue.get()[1]
        b. if node == goal:
        c. return True
        d. visited.add(node)
        e. for neighbor in self.graph[node]:
        f. if neighbor not in visited:
                i. queue.put((self.heuristic(neighbor, goal), neighbor))

15. return False

16. def a_star(self, start, goal):
17. visited = set()
18. queue = PriorityQueue()
19. queue.put((0, start))
20. g_score = {node: float('inf') for node in self.graph}
21. g_score[start] = 0
22. f_score = {node: float('inf') for node in self.graph}
23. f_score[start] = self.heuristic(start, goal)

24. while not queue.empty():
        a. node = queue.get()[1]
        b. if node == goal:
        c. return True
        d. visited.add(node)
        e. for neighbor in self.graph[node]:
        f. tentative_g_score = g_score[node] + self.edges[(node, neighbor)]
        g. if tentative_g_score < g_score[neighbor]:
                i. g_score[neighbor] = tentative_g_score
               ii. f_score[neighbor] = tentative_g_score + self.heuristic(neighbor, goal)
              iii. if neighbor not in visited:
               iv. queue.put((f_score[neighbor], neighbor))

25. return False

26. def heuristic(self, node, goal):
27. return abs(node - goal)


28. # Example usage
29. graph = Graph()
30. graph.add_edge(0, 1, 5)
31. graph.add_edge(0, 2, 10)
32. graph.add_edge(1, 2, 3)
33. graph.add_edge(2, 3, 1)
34. graph.add_edge(3, 4, 2)
```

```
start_node = 0
goal_node = 4

print("Best First Search:")
if graph.best_first_search(start_node, goal_node):
print(f"Goal node {goal_node} found!")
else:
print(f"Goal node {goal_node} not found.")

print("A* Algorithm:")
if graph.a_star(start_node, goal_node):
print(f"Goal node {goal_node} found!")
else:
print(f"Goal node {goal_node} not found.")
```

- **Result-**



```
Best First Search:
Goal node 4 found!
A* Algorithm:
Goal node 4 found!
```

Experiment 6

- **Aim –** To implement Minimax Algorithm.

- **Algorithm –**

```
function minimax(node, depth, Player)
1.if depth ==0 or node is a terminal node then
return value(node)
2.If Player =‘Max’                        // for Maximizer Player
    set $\alpha = -\infty$                //worst case value for MAX
 for each child of node do
 value= minimax(child, depth-1, ’MIN’)
$\alpha$= max($\alpha$, Value)            //gives Maximum of the values
return ($\alpha$)
  else                                    // for Minimizer player
 set $\alpha = +\infty$                   //worst case value for MIN
 for each child of node do
 value= minimax(child, depth-1, ’MAX’)
 $\alpha$ = min($\alpha$, Value)          //gives minimum of the values
return ($\alpha$)
```

- **Code –**

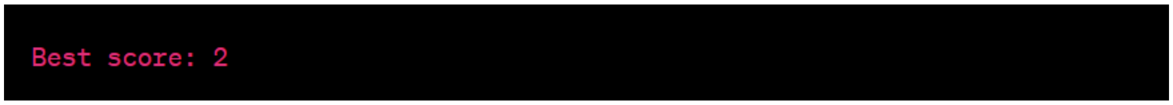
```
import math

# define the game tree
game_tree = {
'A': {
'B': {
    'C': 1,
    'D': 5
},
'E': {
    'F': 3,
    'G': 2
}
}
}

# define the evaluation function for a leaf node
def evaluate(node):
return node

# define the Minimax algorithm
def minimax(node, depth, is_maximizing_player):
```

16. if depth == 0 or isinstance(node, int):
17. return evaluate(node)

18. if is_maximizing_player:
19. max_eval = -math.inf
20. for child in node.values():
    a. eval = minimax(child, depth-1, False)
    b. max_eval = max(max_eval, eval)
21. return max_eval
22. else:
23. min_eval = math.inf
24. for child in node.values():
    a. eval = minimax(child, depth-1, True)
    b. min_eval = min(min_eval, eval)
25. return min_eval

26. # run the Minimax algorithm on the game tree
27. best_score = minimax(game_tree, 2, True)
28. print("Best score:", best_score)

- **<u>Result –</u>**



```
Best score: 2
```

Experiment 7

- **Aim –** Implementation of unification and resolution for real world problems.

- **Algorithm–**

Prolog unification

When programming in Prolog, we spend a lot of time thinking about how variables and rules "match" or "are assigned." There are actually two aspects to this. The first, "unification," regards how terms are matched and variables assigned to make terms match. The second, "resolution," is described in separate notes. Resolution is only used if rules are involved. You may notice in these notes that no rules are involved since we are only talking about unification.

Terms

Prolog has three kinds of **terms**:

1. Constants like 42 (numbers) and franklin (atoms, i.e., lower-case words).
2. Variables like X and Person (words that start with upper-case).
3. Complex terms like parent(franklin, bo) and baz(X, quux(Y))

Two terms **unify** if they can be matched. Two terms can be matched if:

- they are the same term (obviously), or
- they contain variables that can be unified so that the two terms without variables are the same.

For example, suppose our knowledge base is:

```
woman(mia).
loves(vincent, angela).
loves(franklin, mia).
```

- mia and mia unify because they are the same.
- mia and X unify because X can be given the value mia so that the two terms (without variables) are the same.
- woman(mia) and woman(X) unify because X can be set to mia which results in identical terms.
- loves(X, mia) and loves(vincent, X) **cannot** unify because there is no assignment for X (given our knowledge base) that makes the two terms identical.
- loves(X, mia) and loves(franklin, X) also cannot unify (can you see why?).

We saw in the Prolog notes that we can "query" the knowledge base and get, say, all the people who love mia. When we query with loves(X, mia). we are asking Prolog to give us all the values for X that unify. These values are, essentially, the people who love mia.

Rules :

term1 and term2 unify whenever:

1. If term1 and term2 are **constants**, then term1 and term2 unify if and only if they are the same atom, or the same number.
2. If term1 is a **variable** and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2. (And vice versa.) (If they are both variables, they're both instantiated to each other, and we say that they share values.)
3. If term1 and term2 are **complex terms**, they unify if and only if:

   a. They have the same **functor** and **arity**. The functor is the "function" name (this functor is foo: foo(X, bar)). The arity is the number of arguments for the functor (the arity for foo(X, bar) is 2).

   b. All of their corresponding arguments unify. **Recursion!**

   c. The variable instantiations are compatible (i.e., the same variable is not given two different unifications/values).

1. Two terms unify if and only if they unify for one of the above three reasons (there are no reasons left unstated).

Example

We'll use the = predicate to test if two terms unify. Prolog will answer "Yes" if they do, as well as any sufficient variable assignments to make the unification work.

Do these two terms unify?

1.

```
?- mia = mia.
```

o/p Ans:- Yes from Rule 1

2.

```
?- mia = X.
```

o/p Ans:- Yes, from rule 2.

3.

```
?- X = Y.
```

o/p Yes, from rule

4.

```
?- k(s(g), Y) = k(s(g, X), Y).
```

o/p No, these two terms do not unify because arity of s(g) do not match with the arity of s(g,X) due to which rule 3 fails in recursion.

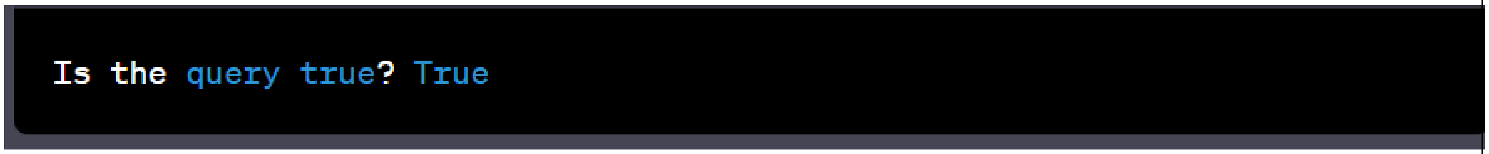
- **Code:**

```
# Define the knowledge base
kb = [
'likes(john, mary)',
'likes(jane, tom)',
'likes(jane, john)',
'likes(mary, wine)',
'likes(john, wine)'
]

# Define the query to be answered
query = 'likes(john, X)'

# Define the unification function
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].islower():
return unify_var(y, x, theta)
elif isinstance(x, list) and isinstance(y, list):
return unify(x[1:], y[1:], unify(x[0], y[0], theta))
else:
return None

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

# Define the resolution function
def resolution(kb, query):
clauses = kb + [query]
new = set()
while True:
for i in range(len(clauses)):
for j in range(i+1, len(clauses)):
    resolvents = resolve(clauses[i], clauses[j])
    if resolvents is None:
        continue
    elif not resolvents:
        return True
    new = new.union(set(resolvents))
if new.issubset(set(clauses)):
return False
for clause in new:
if clause not in clauses:
    clauses.append(clause)

def resolve(ci, cj):
for di in ci.split('|'):
for dj in cj.split('|'):
theta = unify(di, negate(dj), {})
if theta is not None:
    resolvents = []
    for dk in ci.split('|') + cj.split('|'):
        if dk != di and dk != dj:
        resolvents.append(substitute(dk, theta))
    return resolvents
return None

def negate(clause):
if clause[0] == '~':
return clause[1:]
```

```
else:
return '~' + clause

def substitute(clause, theta):
tokens = clause.split('(')
pred = tokens[0]
args = tokens[1][:-1].split(',')
for i in range(len(args)):
if args[i] in theta:
args[i] = theta[args[i]]
return pred + '(' + ','.join(args) + ')'

# Test the functions on the example problem
result = resolution(kb, query)
print("Is the query true?", result)
```

Result :



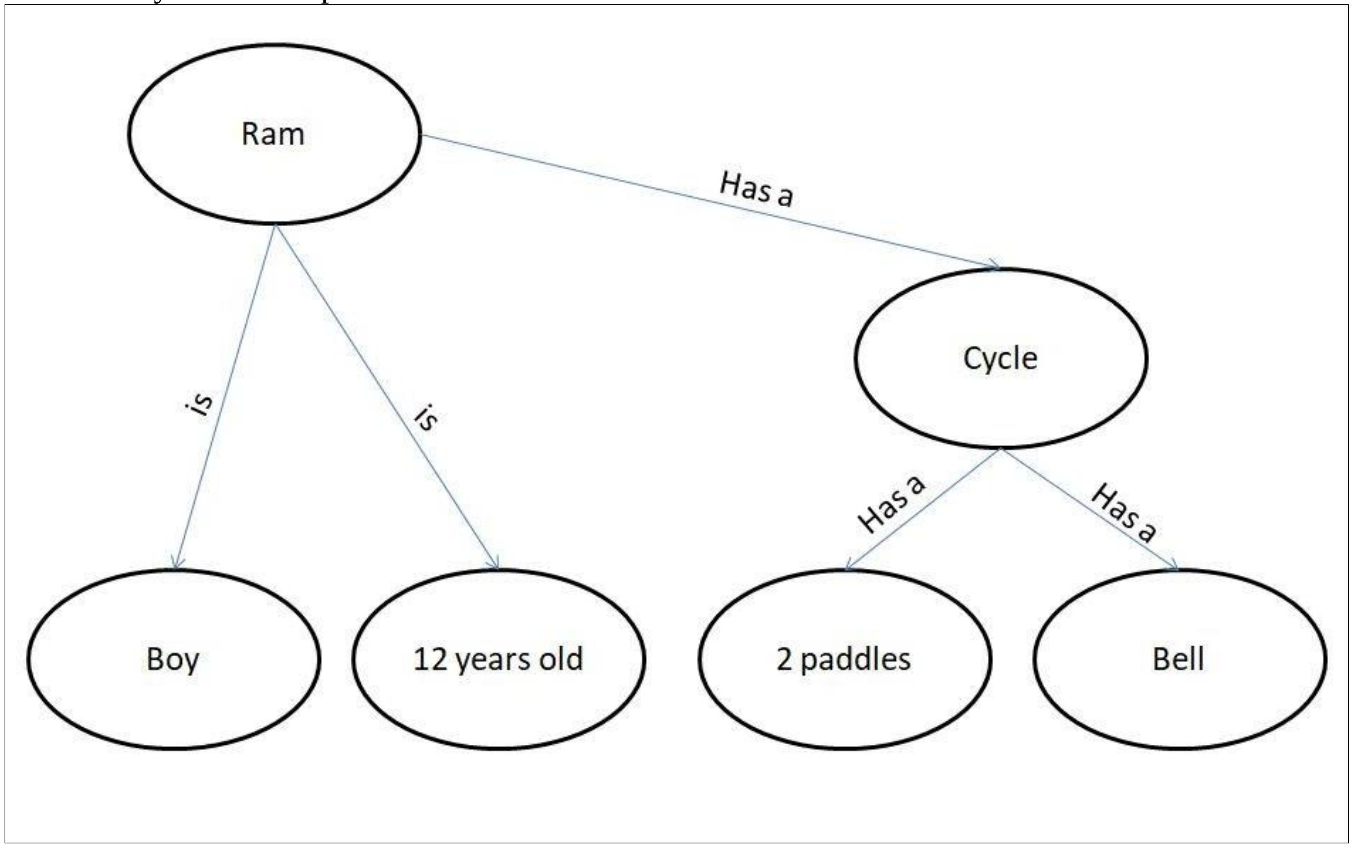
```
Is the query true? True
```

Experiment 8

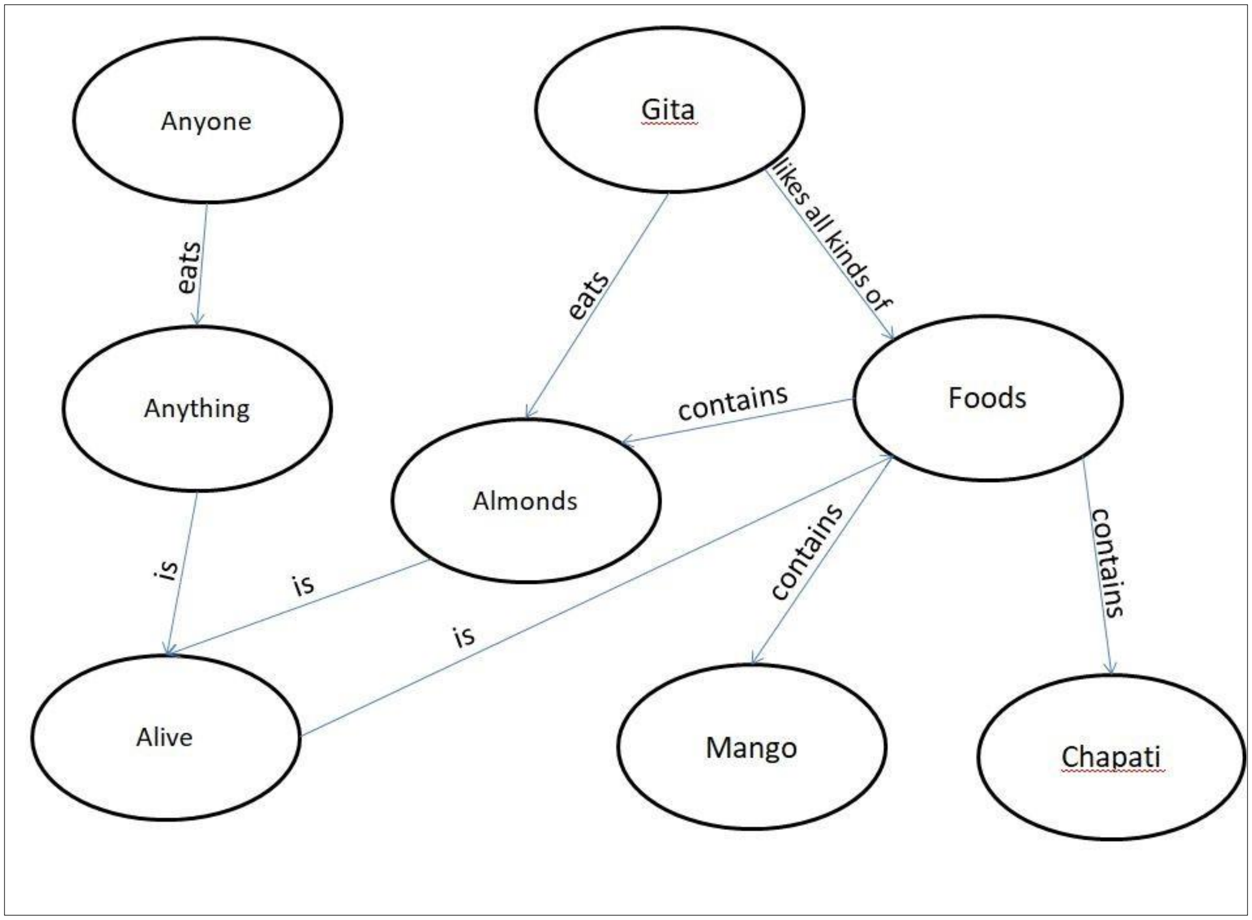
- **Aim –** Implementation of knowledge representation schemes – use cases.

- **Semantic relations –**

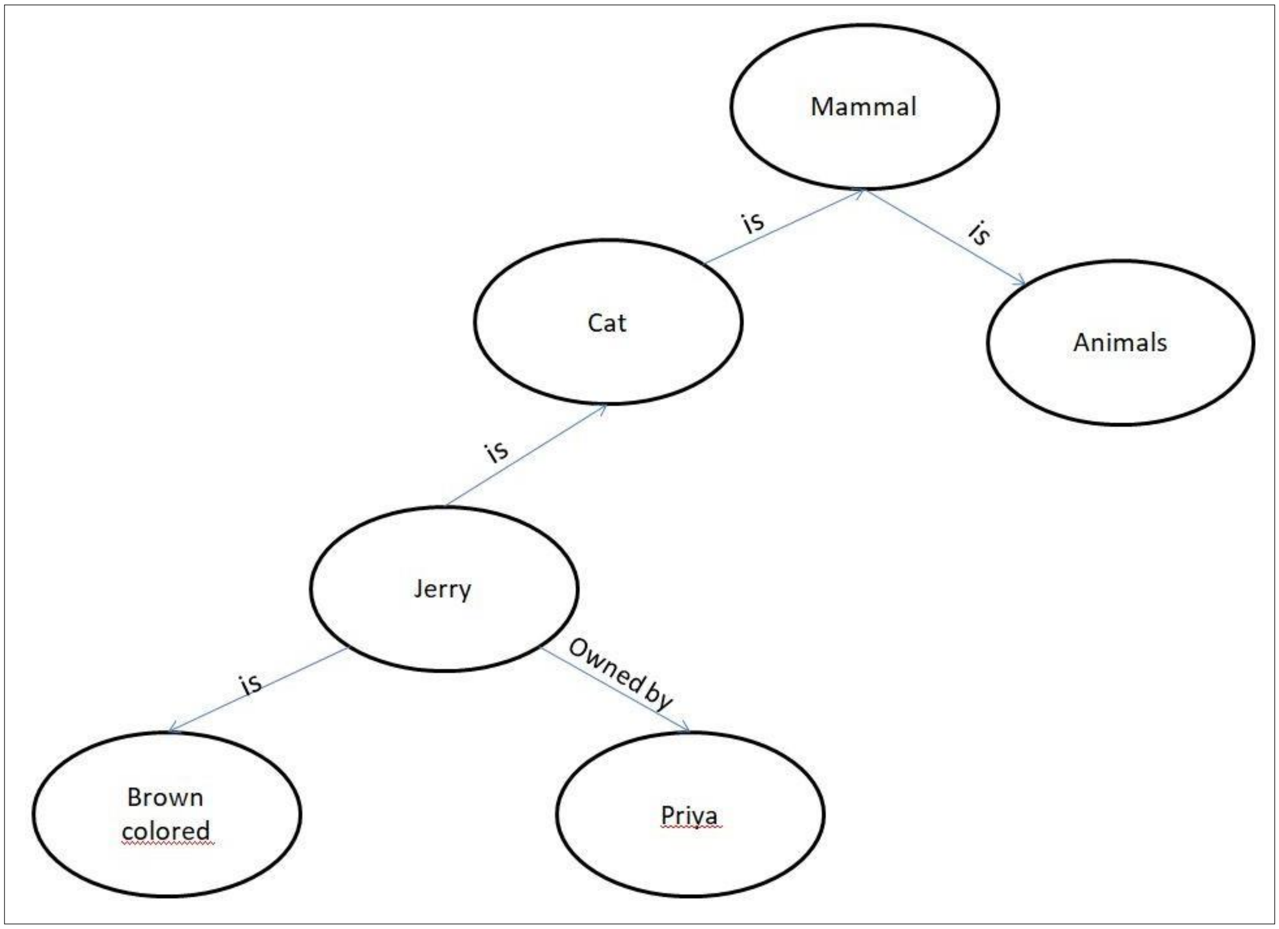
a. 1. Ram has a cycle.
   2. Ram is a boy.
   3. Cycle has a bell.
   4. Ram is 12 years old.
   5. Cycle has two paddles.



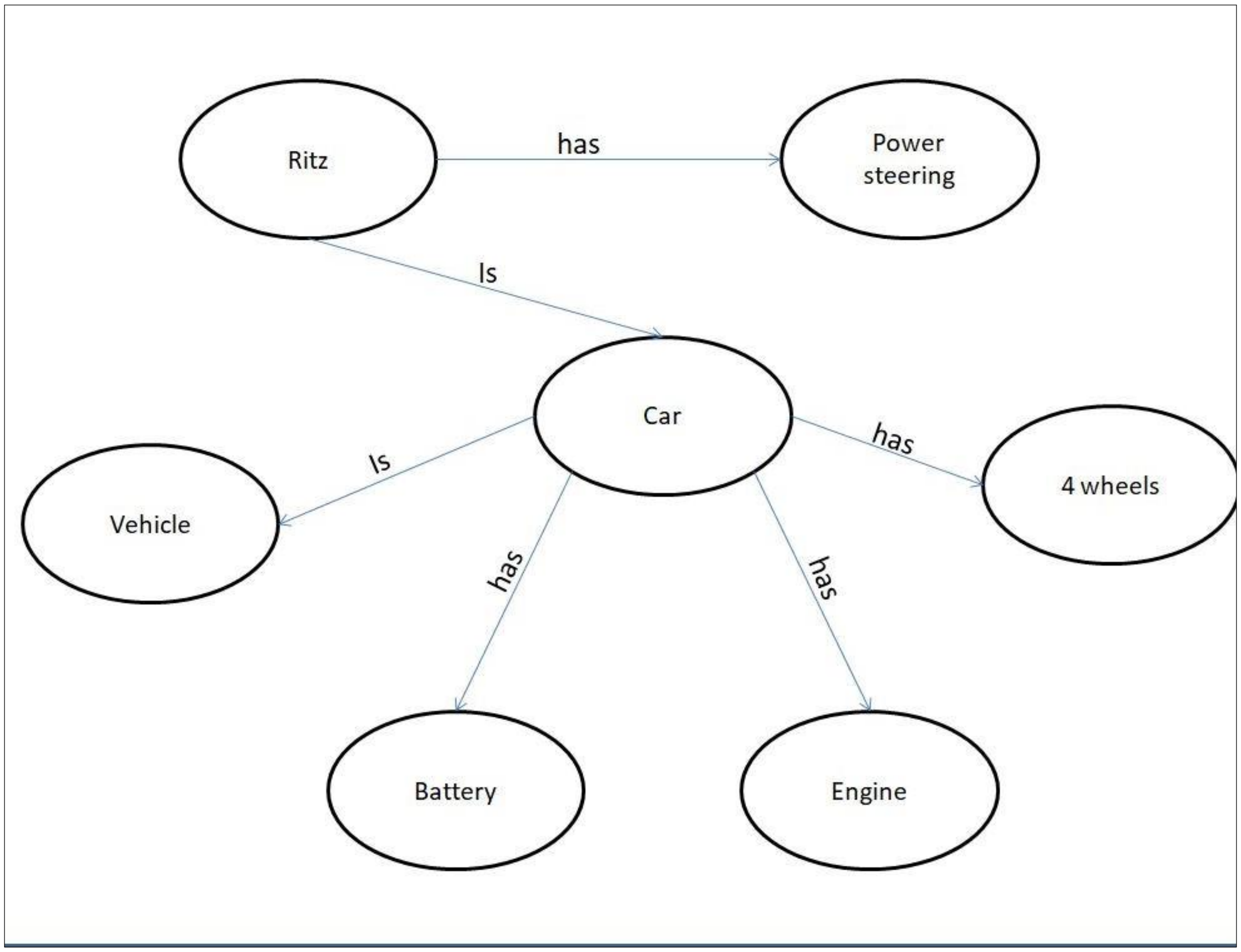
b. 1. Gita likes all kinds of food.
   2. Mango and chapati are food.
   3. Gita eats almond and is still alive.
   4. Anything eaten by anyone and is still alive is food.



c. 1. Jerry is a cat.
2. Jerry is a mammal
3. Jerry is owned by Priya.
4. Jerry is brown colored.
5. All Mammals are animal.



d. 1. Ritz is a car.
2. Car has 4 wheels.
3. Car is a vehicle.
4. Car has engine.
5. Car has battery.
6. Ritz has power steering.



- **Code :**

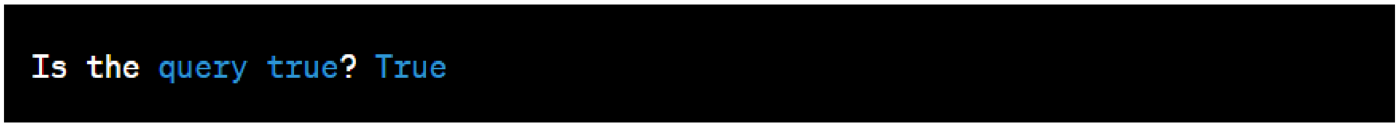
```
# Define the knowledge base
kb = [
'p',
'p -> q',
'q -> r'
]

# Define the query to be answered
query = 'r'

# Define the function to check if a sentence is a literal
```

```
def is_literal(s):
return s[0].islower()

# Define the function to check if a sentence is a clause
def is_clause(s):
return s.count('->') <= 1 and all(map(is_literal, s.split('&')))

# Define the function to extract the antecedent and consequent of a sentence
def split_implication(s):
if '->' in s:
return s.split('->')
else:
return [s, None]

# Define the function to negate a literal
def negate_literal(lit):
if lit[0] == '~':
return lit[1:]
else:
return '~' + lit

# Define the function to resolve two clauses
def resolve(c1, c2):
resolvents = []
for l1 in c1.split('&'):
for l2 in c2.split('&'):
    if l1 == negate_literal(l2):
        resolvents.append((c1 + '&' + c2).replace(l1, '').replace(l2, ''))
return resolvents

# Define the function to apply resolution to the knowledge base
def resolution(kb, query):
clauses = kb + [negate_literal(query)]
new = set()
while True:
for i in range(len(clauses)):
    for j in range(i+1, len(clauses)):
        if i == j:
        continue
        resolvents = resolve(clauses[i], clauses[j])
        if not resolvents:
        continue
        elif '' in resolvents:
        return True
        new = new.union(set(resolvents))
    if '' in new:
        return True
if new.issubset(set(clauses)):
    return False
for clause in new:
```

a. if clause not in clauses:
    i. clauses.append(clause)

41. # Test the functions on the example problem
42. result = resolution(kb, query)
43. print("Is the query true?", result)

Result :



```
Is the query true? True
```

Experiment 9

- **Aim –** Implementation of uncertain methods for an application.

- **Algorithm–**
  1. Define the variables and their probability distributions.
  2. Define the relationships between the variables. This can be done by constructing a probabilistic graphical model, such as a Bayesian network or a Markov random field.
  3. Estimate the parameters of the model using maximum likelihood estimation, Bayesian inference, or other methods.
  4. Make predictions or inferences using the model. This can involve computing marginal probabilities, conditional probabilities, or joint probabilities.
  5. Evaluate the performance of the model using metrics such as log-likelihood, accuracy, or precision-recall.
  6. Refine the model as necessary by adjusting the parameters or the structure of the model.
  7. Test the model on new data to assess its generalization performance.

Code – Implementation of Uncertain method for an Application

$$\textbf{Probability of occurrence} = \frac{\text{Number of desired outcomes}}{\text{Total number of outcomes}}$$

we can find the probability of an uncertain event by using the below formula.

Problem 1:- If In class 80 students and 60 students got 60 % marks then Calculate theProbability of finding how many students got the 60 marks for given data set .

```
import scipy.stats as stats

n = 80 # total number of students
p = 0.6 # probability of getting 60% marks
k = 60 # number of students who got 60% marks

prob = stats.binom.pmf(k, n, p)
print("The probability of finding", k, "students who got 60% marks is:", prob)
```

Result:

```
The probability of finding 60 students who got 60% marks is: 0.0011793654564486923
```

Problem 2:- Create function that returns probability percent rounded to one decimal

```
1. import scipy.stats as stats

   1.  def calculate_probability(n, p, k):

   2.  prob = stats.binom.pmf(k, n, p)

   3.  return round(prob * 100, 1)

    # Sample Space
    ClassA = 30


    # Determine the probability of drawing a heart
    Marks = 15
    grade_probability = event_probability(Marks, ClassA)


    # Print each probability
    print(str(grade_probability) + '%')
```

Output:- 28.57

- **<u>Result –</u> The program has been executed successfully.**

Experiment 10

- **Aim –** Implementation of block world problem.

- **Algorithm –**

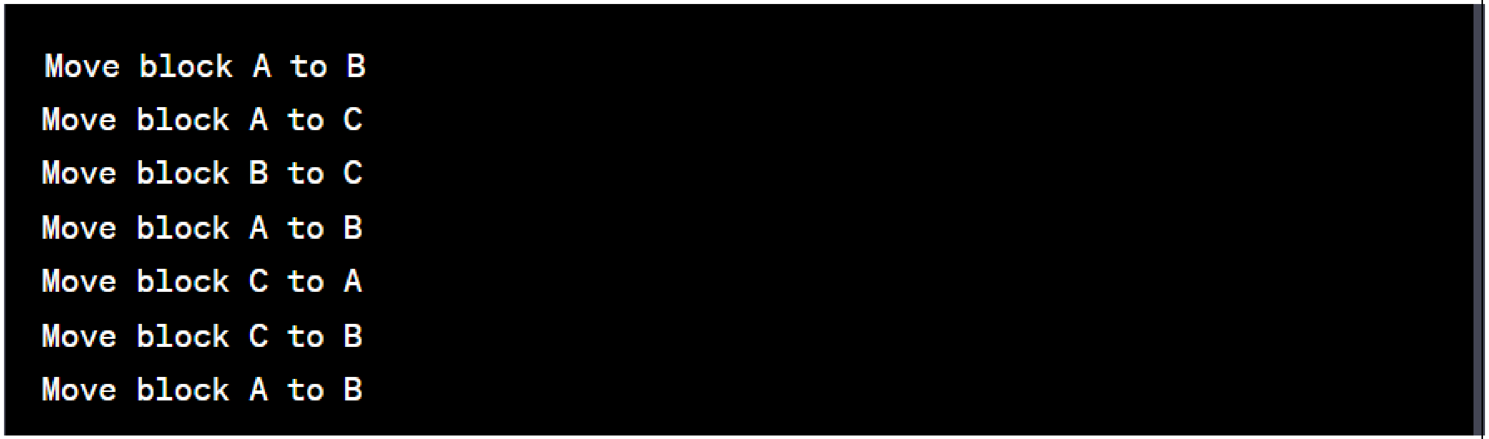
1. MOVE(B,A)- To lift block from B to A.
2. ON(B,A)- To place block B on A.
3. CLEAR(B)- To lift block B from the table.
4. PLACE(B)- To put the block B on table.

- **Code –**

```
# Define initial state
initial_state = {
'A': ['B', 'C'],
'B': [],
'C': []
}

# Define goal state
goal_state = {
'A': [],
'B': [],
'C': ['B', 'A']
}

# Define rules for moving blocks
def move(state, a, b):
if not state[a]:
return state
elif state[b] and state[a][-1] > state[b][-1]:
return state
else:
return {
        **state,
        a: state[a][:-1],
        b: state[b] + [state[a][-1]]
}

# Define search algorithm
def search(start, goal, move_fn):
visited = set()
queue = [(start, [])]

while queue:
state, path = queue.pop(0)
if state == goal:
        return path
```

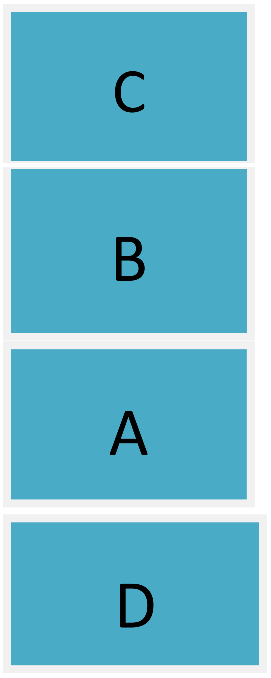
```
29. for a in state:
        a.  for b in state:
        b.  if a != b:
                 i.  new_state = move_fn(state, a, b)
                ii.  if new_state not in visited:
               iii.  visited.add(new_state)
                iv.  queue.append((new_state, path + [(a, b)]))

30. # Search for solution
31. solution = search(initial_state, goal_state, move)

32. # Print solution
33. for move in solution:
34. print("Move block", move[0], "to", move[1])
```

- **<u>Result –</u>** Goal achieved.



```
Move block A to B
Move block A to C
Move block B to C
Move block A to B
Move block C to A
Move block C to B
Move block A to B
```

- **<u>Output –</u>**



Experiment 11

Aim – AIM-(Implementation of Learning Algo)

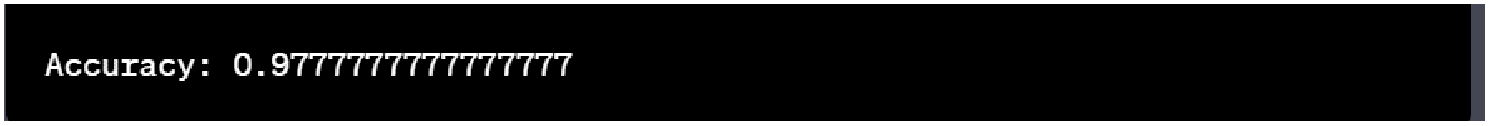
Algorithm –

Understanding the different types of learning algorithms, such as supervised, unsupervised, and reinforcement learning.
Exploring popular algorithms, such as decision trees, random forests, support vector machines, neural networks, and deep learning.
Implementing these algorithms in programming languages like Python, R, or MATLAB.
Evaluating the performance of the implemented models using metrics such as accuracy, precision, recall, F1 score, and ROC-AUC.
Tuning hyperparameters to improve the model's performance.
Applying the learned concepts and techniques to solve real-world problems in domains such as finance, healthcare, marketing, and customer service.

Code –

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the iris dataset
iris = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_state=42)

# Create a decision tree classifier with max depth of 3
clf = DecisionTreeClassifier(max_depth=3, random_state=42)

# Train the decision tree classifier
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy of the model
print("Accuracy:", accuracy)
```

Result :



```
Accuracy: 0.9777777777777777
```

Experiment 12

Aim – AIM- Development of ensemble model

Algorithm –
Data preparation:
Collect and prepare your dataset, including feature engineering, cleaning, and splitting into training and testing sets.

Select base models:
Choose a set of base models that have different strengths and weaknesses. These can be decision trees, random forests, logistic regression, neural networks, or any other models that are appropriate for your problem.

Train base models:
Train each base model on the training data and evaluate its performance on the testing data. This will give you an idea of how well each model performs on its own.

Combine models:
Combine the predictions of each base model using a suitable combination technique. Common combination techniques include voting, weighted voting, and stacking.

Evaluate ensemble model:
Evaluate the performance of the ensemble model on the testing data. You should see an improvement in prediction accuracy compared to the individual base models.

Tune ensemble model:
Fine-tune the hyper parameters of the ensemble model to optimize its performance.

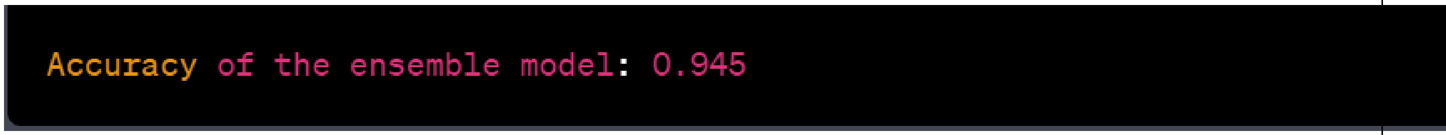
Test ensemble model:
Test the final ensemble model on new, unseen data to ensure that it is robust and can generalize well.

Code :

```
 # Import libraries
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_classes=2,
random_state=1)

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Define individual classifiers
lr = LogisticRegression(random_state=1)
nb = GaussianNB()
dt = DecisionTreeClassifier(random_state=1)

# Define the ensemble model
ensemble = VotingClassifier(estimators=[('lr', lr), ('nb', nb), ('dt', dt)])

# Train the ensemble model
ensemble.fit(X_train, y_train)

# Make predictions using the ensemble model
y_pred = ensemble.predict(X_test)

# Calculate the accuracy score of the ensemble model
accuracy = accuracy_score(y_test, y_pred)

# Print the accuracy score of the ensemble model
print("Accuracy of the ensemble model:", accuracy)
```

Result :



```
Accuracy of the ensemble model: 0.945
```

Experiment 13

Aim – Implementation of NLP problem

Algorithm. -

Gather and preprocess the data:
The first step in any NLP problem is to gather and preprocess the data. This involves collecting data from various sources and cleaning and preparing it for analysis. This may involve tasks such as removing stop words, stemming or lemmatizing words, and converting the text into a format that can be analyzed by a machine learning algorithm.

Perform exploratory data analysis (EDA):
Once the data has been gathered and preprocessed, the next step is to perform exploratory data analysis (EDA). This involves analyzing the data to identify patterns, trends, and relationships between variables. EDA may involve techniques such as word frequency analysis, topic modeling, and sentiment analysis.

Feature engineering:
Feature engineering is the process of selecting and extracting the most relevant features from the data to be used in the machine learning model. This may involve techniques such as term frequency-inverse document frequency (TF-IDF), word embeddings, and part-of-speech tagging.

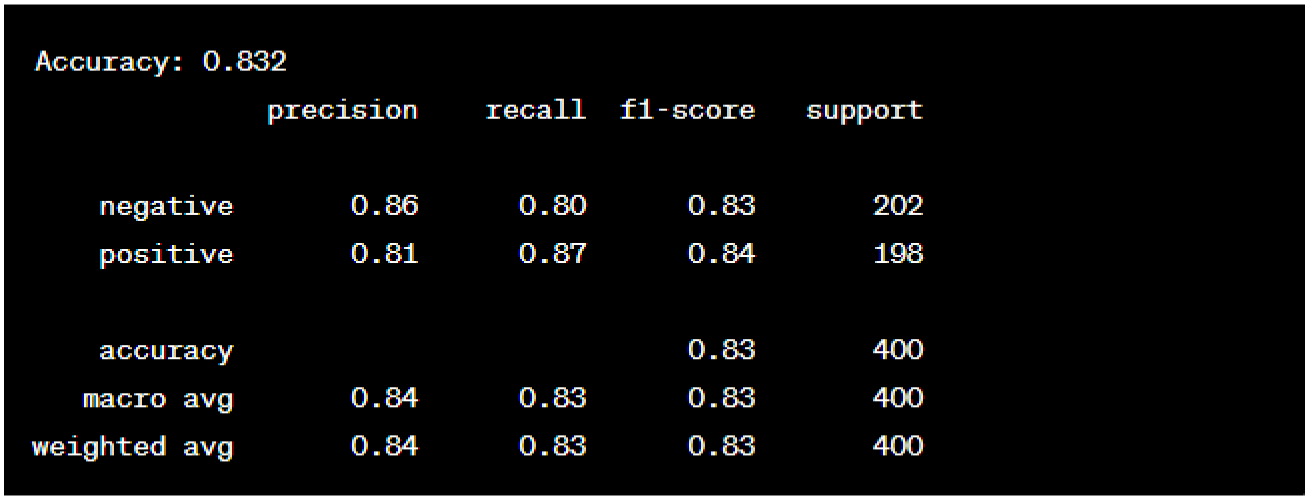
Train the model:
Once the features have been extracted, the next step is to train the machine learning model. This may involve using a variety of algorithms such as Naive Bayes, Support Vector Machines (SVMs), or Neural Networks.

Evaluate the model:
Once the model has been trained, it is important to evaluate its performance on a test dataset. This may involve using metrics such as accuracy, precision, recall, and F1 score.

Code -

```
import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

# Load the dataset
df = pd.read_csv('movie_reviews.csv')

# Preprocess the text data
stop_words = set(stopwords.words('english'))
df['review'] = df['review'].apply(lambda x: ' '.join([word for word in word_tokenize(x) if word not in
stop_words]))

# Create feature vectors using TF-IDF
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=5000)
X = tfidf_vectorizer.fit_transform(df['review'])
y = df['sentiment']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the Naive Bayes classifier
clf = MultinomialNB()
clf.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = clf.predict(X_test)

# Evaluate the performance of the classifier
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(classification_report(y_test, y_pred))
```

Result -



```
Accuracy: 0.832
              precision    recall  f1-score   support

    negative       0.86      0.80      0.83       202
    positive       0.81      0.87      0.84       198

    accuracy                           0.83       400
   macro avg       0.84      0.83      0.83       400
weighted avg       0.84      0.83      0.83       400
```

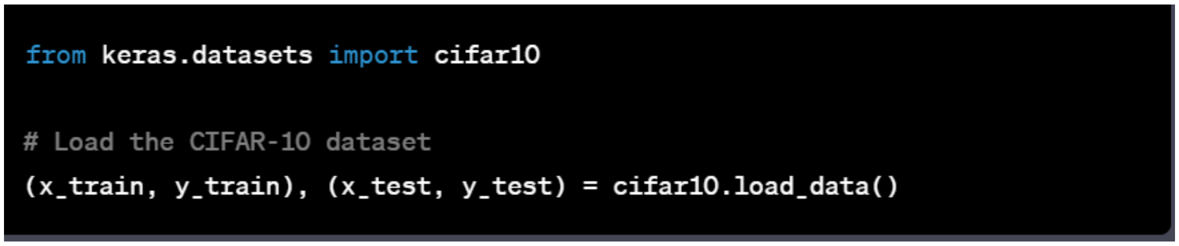
<u>Experiment 14</u>

<u>Aim –</u> Deep learning Project in Python.

In this project, we will build a deep learning model to classify images from the CIFAR-10 dataset. The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6,000 images per class. The dataset is divided into 50,000 training images and 10,000 testing images.
We will use Python and the Keras library to build and train our model. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.

Step 1: Load and prepare the data
First, we will load the CIFAR-10 dataset using the Keras library. Keras has a built-in function to load CIFAR-10, so we can use that to simplify our code.

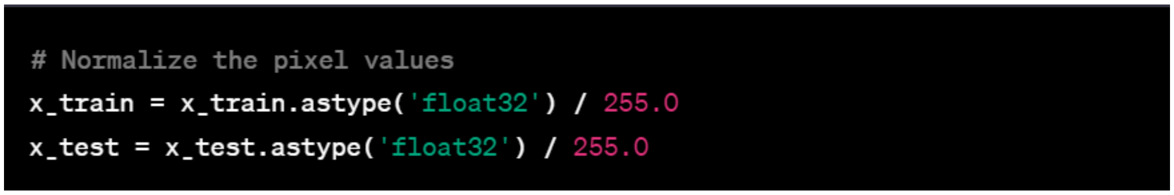
The cifar10.load_data() function returns two tuples: one for the training data and labels, and one for the test data and labels. We'll use the training data and labels to train our model, and the test data and labels to evaluate its performance.



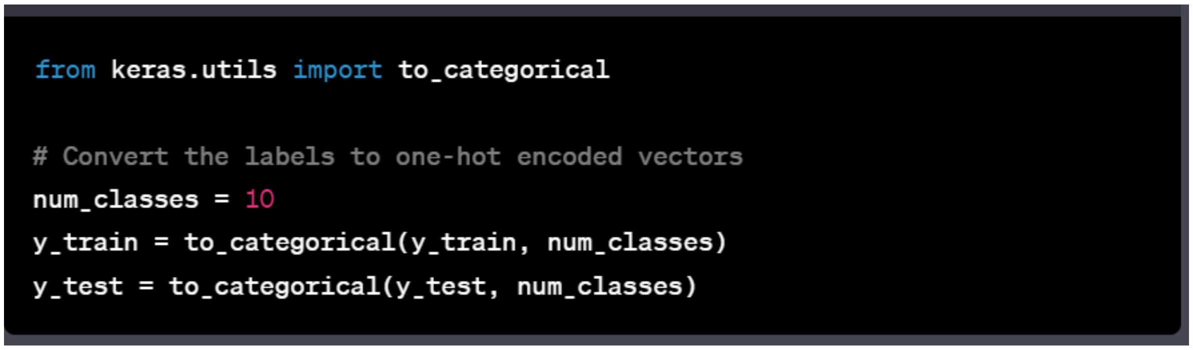
```python
from keras.datasets import cifar10

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Next, we will preprocess the data. We'll start by normalizing the pixel values to be between 0 and 1.

We also need to convert the labels to one-hot encoded vectors.



```python
# Normalize the pixel values
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

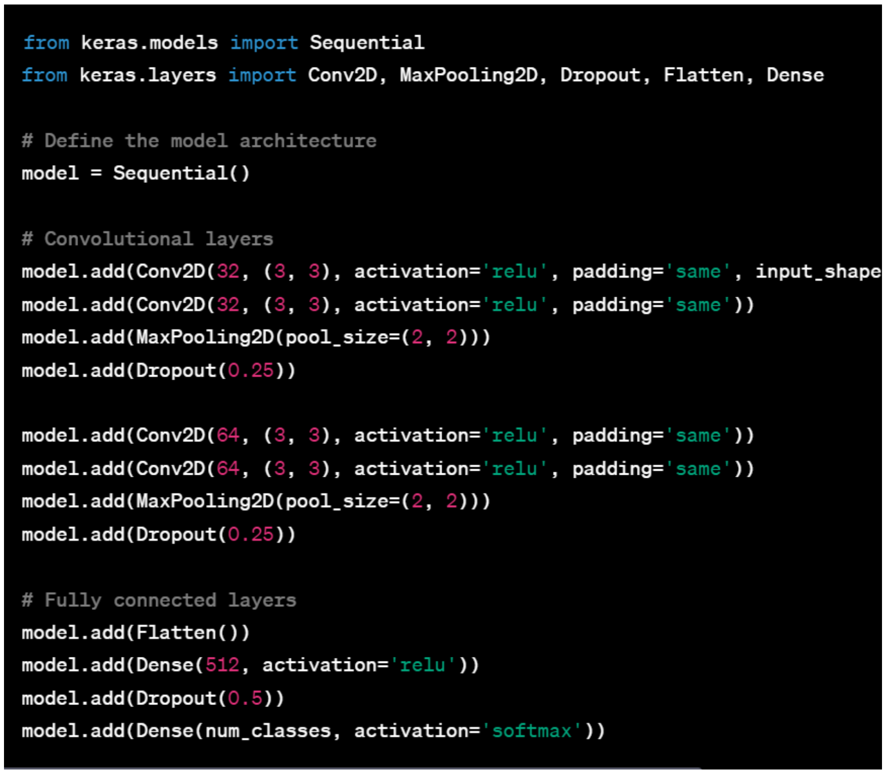


```python
from keras.utils import to_categorical

# Convert the labels to one-hot encoded vectors
num_classes = 10
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

Now our data is ready to be used to train and evaluate our model.

Step 2: Define the model architecture
We will use a convolutional neural network (CNN) for image classification. CNNs are well-suited for image recognition tasks because they can learn local patterns and spatial relationships in the input images.

Our model will consist of a sequence of convolutional layers followed by pooling layers, with some dropout regularization to prevent overfitting. Finally, we'll add a couple of fully connected layers to perform the classification.



```python
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, Dense

# Define the model architecture
model = Sequential()

# Convolutional layers
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Fully connected layers
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

We start by adding two convolutional layers with 32 filters each and a 3x3 kernel size.