Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

# Academic Year : 2021-2022

| | |
|---|---|
| Name | Junaid Girkar |
| Sap ID | 60004190057 |
| Class | TE Comps |
| Batch | A-4 |
| Sem | V |
| Course | Artificial Intelligence |
| Course Code | DJ19CEC503 |

# INDEX

# EXPERIMENT 1

**AIM**: Select a problem statement relevant to AI .

i) Identify the problem
ii) PEAS Description
iii) Problem formulation

**DESCRIPTION**:

Artificial intelligence is defined as the study of rational agents. A rational agent could be anything that makes decisions, as a person, firm, machine, or software. It carries out an action with the best outcome after considering past and current percepts(agent's perceptual inputs at a given instance). An AI system is composed of an agent and its environment. The agents act in their environment. The environment may contain other agents.

An agent is anything that can be viewed as :

- perceiving its environment through sensors and
- acting upon that environment through actuators

## TYPES OF AGENTS:

Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- Simple Reflex Agents
- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents
- Learning Agent

## TYPES OF ENVIRONMENTS:

An environment in artificial intelligence is the surrounding of the agent. The agent takes input from the environment through sensors and delivers the output to the environment through actuators. There are several types of environments:

- Fully Observable vs Partially Observable
- Deterministic vs Stochastic
- Competitive vs Collaborative
- Single-agent vs Multi-agent
- Static vs Dynamic
- Discrete vs Continuous

## PART A)

Write the problem faced in each one of them

1) **AGENT**: Chess playing with a clock
    a. **Performance Measure:**
        i. Win:Lose ratio
        ii. Speed
        iii. Total game time
    b. **Environment:**
        i. Chessboard
        ii. Clock
    c. **Environment Type:**
        i. Fully observable as the entire chessboard can be seen.
        ii. Discrete as at any stage, there are a finite number of possible moves.
        iii. Dynamic as with every move, the chessboard changes.
        iv. Competitive as it is competing against another human/agent.
    d. **Actuator:**
        i. Pausing of the clock
        ii. Movement of chess pieces
    e. **Sensors:**
        i. Movement arm
        ii. Servo Motors
        iii. Location of chess pieces using reed switches.
    f. **Problem:**
        i. Playing a timed game human vs AI. Every piece has a fixed movement pattern and there are a fixed number of possible movements.

2) **<u>AGENT</u>**: Driving a car

a. **Performance Measure:**
   i. Speed
   ii. Time Taken
   iii. Comfort
   iv. Fuel Economy

b. **Environment:**
   i. Car
   ii. Road
   iii. Traffic
   iv. Signposts
   v. Potholes

c. **Environment Type:**
   i. Partially Observable as the agent cannot see everything at the same time and has a few blind spots
   ii. Stochastic as the environment is random
   iii. Dynamic as the environment keeps on changing.
   iv. Collaborative as all the cars have to follow proper rules so as to avoid accidents.

d. **Actuator:**
   i. Steering Wheel
   ii. Brake
   iii. Accelerator
   iv. Mirror
   v. Gearstick

e. **Sensors:**
   i. GPS
   ii. Odometer
   iii. Speedometer
   iv. Fuel tank capacity meter

f. **Problem:**
   i. Driving a car from point A to B. There may or may not be passengers and traffic. Road conditions can change.

3) **<u>AGENT</u>**: Interactive English tutor

a. **Performance Measure:**
   i. Language Improvement
   ii. Increase in test score
   iii. Number of errors made per paragraph

b. **Environment:**

   i. Classroom
   ii. Table
   iii. Chair
   iv. Students
   v. Whiteboard
   vi. Books

c. **Environment Type:**
   i. Deterministic as the current state determines the next action.
   ii. Collaborative as the student and teacher have to work in accordance.
   iii. Multi-agent as there is a student as well as a teacher agent

d. **Actuator:**
   i. Writing on whiteboard
   ii. Opening and reading the books
   iii. Checking of test papers

e. **Sensors:**
   i. Eyes
   ii. Ears
   iii. Books
   iv. Test Papers

f. **Problem:**
   i. Learning the English language from a tutor in a classroom setting with all the necessary stationary.

4) **<u>AGENT</u>**: Part picking robot

a. **Performance Measure:**
   i. % Efficiency of the robot

b. **Environment:**
   i. Parts
   ii. Conveyer belt

c. **Environment Type:**
   i. Collaborative as multiple agents are working together
   ii. Dynamic as the environment keeps on changing
   iii. Discrete as at any stage, there are only finite number of possible motions.

d. **Actuator:**
   i. Picking up the parts
   ii. Sorting the parts

e. **Sensors:**
   i. Camera
   ii. Robot Arm
   iii. Distance Sensor

- iv. Servo motors

f. **Problem:**
- i. Pick up pieces from a conveyor belt, analyse it and sort it accordingly into bins.

5) **AGENT**: Satellite Image Analysis System

a. **Performance Measure:**
- i. % Correct Analysis
- ii. Time taken

b. **Environment:**
- i. Camera

c. **Environment Type:**
- i. Dynamic as the environment keeps changing continuously
- ii. Discrete as the system will have a finite number of processing steps it will perform.
- iii. Multi-agent as multiple systems can collaborate to get one picture of higher quality and data analysis.
- iv. Collaborative as there is no competition and requires multiple systems to collaborate.

d. **Actuator:**
- i. Capturing of Images
- ii. Movement of the satellite

e. **Sensors:**
- i. Camera
- ii. Color Sensor

f. **Problem:**
- i. Take pictures from a satellite, analyse the images to get useful data from them.

6) **AGENT**: Medical Diagnosis System

a. **Performance Measure:**
- i. % of correct diagnosis
- ii. Time taken
- iii. Systems correctly found
- iv. Number of lawsuits
- v. Cost

b. **Environment:**
- i. Patient
- ii. Hospital

c. **Environment Type:**
- i. Deterministic as the current state defines the next actions.

- ii. Partially Observable as not everything can be observed simultaneously.

- **d. Actuator:**
  - **i.** Asking questions
  - **ii.** Recommending further tests
  - **iii.** Printing reports
  - **iv.** Dispensing medicines
- **e. Sensors:**
  - **i.** Camera
  - **ii.** Microphone
  - **iii.** Speaker
  - **iv.** Printer
- **f. Problem:**
  - i. Create an AI machine that can diagnose patients by asking questions, can recommend tests, print out reports and dispense medicines.

## 7) <u>**AGENT**</u>: Refinery Controller

- **a. Performance Measure:**
  - **i.** % Efficiency
  - **ii.** Speed
- **b. Environment:**
  - **i.** Refinery workers
  - **ii.** Machines
- **c. Environment Type:**
  - **i.** Collaborative as multiple agents have to work in synchronization.
  - **ii.** Multi-Agents as different agents have to work together.
- **d. Actuator:**
  - **i.** Turn on/off systems
  - **ii.** Adjust temperatures
  - **iii.** Adjust pressures
- **e. Sensors:**
  - **i.** Temperature sensor
  - **ii.** Pressure sensor
  - **iii.** Proximity sensor
- **f. Problem:**
  - i. Control and regulate sections of the refinery through an AI machine. The controller machine should work in tandem with the human workers.

## 8) <u>**AGENT**</u>: Poker playing

a. **Performance Measure:**
    i. Rounds won
    ii. Number of correct moves
b. **Environment:**
    i. Cards
    ii. Humans
c. **Environment Type:**
    i. Discrete as there exist only a finite number of moves at any time.
    ii. Competitive as multiple agents compete to win.
d. **Actuator:**
    i. Dealing the cards
    ii. Playing the cards
e. **Sensors:**
    i. Camera
    ii. Color sensor
    iii. Servo motor
    iv. Movement Arm
f. **Problem:**
    i. Have an AI controlled movement arm that deals the cards. One player is AI and the others are humans. The AI must play the game properly and try to win the game.

9) **<u>AGENT</u>**: Chatbot

a. **Performance Measure:**
    i. Time taken
    ii. Grammatical accuracy
b. **Environment:**
    i. Chatbot
    ii. Human
c. **Environment Type:**
    i. Continuous as the actions cannot be numbered.
    ii. Collaborative as it works in parallel with one or more agents.
d. **Actuator:**
    i. Displaying the questions
    ii. Taking responses
e. **Sensors:**
    i. Keyboard
    ii. Screen
f. **Problem:**

i. An AI robot that can interact with humans and reply to questions with human-like answers or ask human-like questions. The input and output can be in the form of text or audio.

10) **AGENT**: Soccer playing robot

a. **Performance Measure:**
   i. Number of goals scored
   ii. Number of goals saved
   iii. Number of penalties
   iv. Number of games won

b. **Environment:**
   i. Soccer field
   ii. Goal posts
   iii. Goat net
   iv. Humans (Other players)

c. **Environment Type:**
   i. Continuous as the actions at a time cannot be numbered.
   ii. Collaborative as the robot has to play alongside other agents.

d. **Actuator:**
   i. Movement of the ball

e. **Sensors:**
   i. Servo motors
   ii. Proximity sensors

f. **Problem:**
   i. The robot will play soccer alongside humans. It has to predict the moves of the players and apply the counter moves. It has to try to win by shooting goals and saving goals (if its in the position of a goalkeeper).

11) **AGENT**: Recommender system

a. **Performance Measure:**
   i. % Efficiency

b. **Environment:**
   i. Dataset
   ii. Input variables

c. **Environment Type:**
   i. Continuous as the results cannot be numbered.
   ii. Competitive as it tries to increase its efficiency continuously.

d. **Actuator:**
   i. Creating the algorithm
   ii. Using the algorithm

e. **Sensors:**

i. Keyboard
ii. Screen

f. **Problem:**
   i. An AI bot that can make recommendations based on patterns found in past datasets.

---

## PART B)

i) Identify a suitable problem faced during the pandemic

ii) Describe its PEAS properties

iii) Formulate the AI problem

During the pandemic, sanitization was one of the main ways of preventing the spread of the virus. It was an important thing that needed to be done at frequent intervals. So I have written the PEAS for a robot that automates the process and itself requires no human contact. Multiple robots can work together if the room area is high and we want to reduce the time.

**AGENT**: Sanitization robot (Modified roomba)

- **Performance Measure:**
  - Speed
  - Battery Capacity
- **Environment:**
  - Sanitizer
  - Room
- **Environment Type:**
  - **Continuous** as the movements cannot be numbered.
  - **Partially Observable** as the robot can see/sense only specific areas at a time.
  - **Collaborative** as all agents have to work together in achieving the goal of sanitizing the room.

    - **Multi-agent** as multiple robots can work together for achieving the same task.
- **Actuator:**
    - Navigation of the robot
    - Spraying of sanitizer
- **Sensors:**
    - GPS
    - Servo motors
    - Spray pump
    - Proximity sensors
- **Problem:**
    - A problem faced during the pandemic was contactless sanitization of areas.
    - A study revealed that coming in contact with the sanitization machines itself was a covid risk
    - Create a robot that will sanitize everything, will avoid crashing into humans and would require no human contact.
    - Multiple robots should be able to work together.

# EXPERIMENT 2A

**AIM**: Identify, analyze and implement BFS/DFS search algorithms to reach goal state

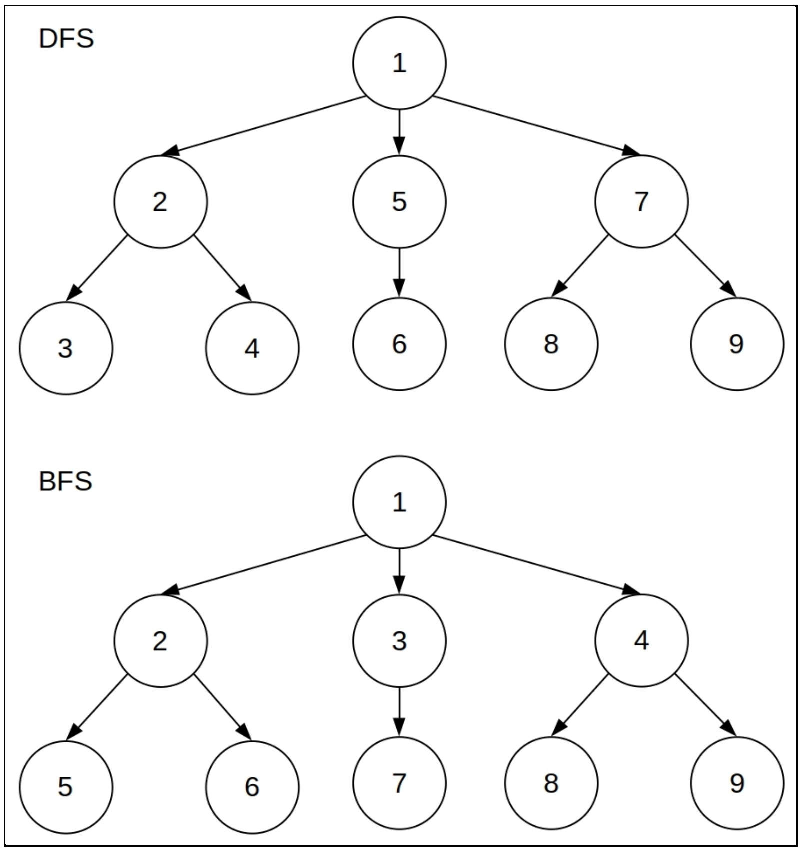
## THEORY:

## BFS:

BFS stands for Breadth First Search is a vertex based technique for finding a shortest path in a graph. It uses a Queue data structure which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked, then its adjacent vertices are visited and stored in the queue. It is slower than DFS. Here, siblings are visited before the children.

## DFS:

DFS stands for Depth First Search is a edge based technique. It uses the Stack data structure, performs two stages, first visited vertices are pushed into stack and second if there are no vertices then visited vertices are popped. Here, children are visited before the siblings.



## CODE:

## bfs.py

```
# visited = []
queue = []
def bfs(visited, graph, node, goal_state):
visited.append(node)
queue.append(node)

while queue: # Creating loop to visit each node
m = queue.pop(0)
print (m, end = " ")
if(m==goal_state):
print("GOAL", end= " ")
break
for neighbour in graph[m]:
if neighbour not in visited:
visited.append(neighbour)
queue.append(neighbour)

def dfs(graph, source,path = []):
if source not in path:
path.append(source)
if source not in graph:# leaf node, backtrack
return path
for neighbour in graph[source]:
path = dfs(graph, neighbour, path)
return path
```

Driver.py

```
from bfs import bfs
from dfs import dfs
graph = {
'5' : ['3','7'],
'3' : ['2', '4'],
'7' : ['8'],
'2' : [],
'4' : ['8'],
'8' : []
}

# 5
# / \
# 3 7
# / \ \
# 2 4 ---- 8
lecture_graph = {
'1' : ['4', '2'],
'4' : ['3'],
'3' : ['10', '9', '2'],
'10' :[],
'9' : [],
'2' : ['5', '8'],
'5' : ['2', '6','7'],
'7' : ['5', '8'],
'8' : ['2', '7'],
'6' : []
}
# 1
# / \
# 4 2
# \ / | \
# 3 5 8
# / | | \ |
# 10 9 6 7
# BFS
visited = []
print("Breadth-First Search traversal sequence")
bfs(visited, lecture_graph, '1', goal_state='8')
print("\n")
#DFS
print("Depth-First Search traversal sequence")
path = dfs(lecture_graph, "1")
goal_state = '9'
new_path=[]
for item in path:
if(item==goal_state):
break
else:
new_path.append(item)
new_path.append("GOAL")
```

```
print(" ".join(new_path))
```

## OUTPUT:

```
Breadth-First Search traversal sequence
1 4 2 3 5 8 GOAL
Depth-First Search traversal sequence
1 4 3 10 GOAL
```

## CONCLUSION:

We learnt about BFS and DFS searching techniques and implemented these algorithms in a python program. We also learnt about their time complexity which is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

# EXPERIMENT 2B

**AIM**: Implementation of DFID search algorithm.

**THEORY**:
A search algorithm which suffers neither the drawbacks of breadth-first nor depth-first search on trees is depth-first iterative-deepening (DFID). The algorithm works as follows: First, perform a depth-first search to depth one. Then, discarding the nodes generated in the first search, start over and do a depth-first search to level two. Next, start over again and do a depth-first search to depth three, etc., continuing this process until a goal state is reached. Since DFID expands all nodes at a given depth before expanding any nodes at a greater depth, it is guaranteed to find a shortest-length solution. Also, since at any given time it is performing a depth-first search, and never searches deeper than depth d, the space it uses is O(d).

The disadvantage of DFID is that it performs wasted computation prior to reaching the goal depth. In fact, at first glance it seems very inefficient. Below, however, we present an analysis of the running time of DFID that shows that this wasted computation does not affect the asymptotic growth of the run time for exponential tree searches. The intuitive reason is that almost all the work is done at the deepest level of the search. Unfortunately, DFID suffers the same drawback as depth-first search on arbitrary graphs, namely that it must explore all possible paths to a given depth.

**CODE**:

```
from collections import defaultdict

class Graph:

    def __init__(self,vertices):

        self.V = vertices # Number of vertices
        self.graph = defaultdict(list)


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

    def DLS(self,src,target,maxDepth):

        if src == target : return True
        if maxDepth <= 0 : return False

        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1)):
                    return True
        return False

    def IDDFS(self,src, target, maxDepth):
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False
```

```
# Create a graph
#       0
#     /   \
#    1     2
#   / \    | \
#  3   4   5   6

g = Graph (7)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

for i in range(2):
   target = [5, 3]; maxDepth = [3,2]; src = 0

   if g.IDDFS(src, target[i], maxDepth[i]) == True:
      print ("Target "+str(target[i]) +" is reachable from source within max depth of 
"+str(maxDepth[i]))
   else :
      print ("Target "+str(target[i]) +" is NOT reachable from source within max depth 
of "+str(maxDepth[i]))
```

**OUTPUT**:

```
Target 5 is reachable from source within max depth of 3
Target 3 is NOT reachable from source within max depth of 2
```

**CONCLUSION**:
We learnt about the DFID search algorithm and implemented it in python.

# EXPERIMENT 3

**Aim:** Identify and analyze informed search algorithms to solve the problem. Implement A* search algorithm to reach goal state.

**Theory:**
A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

**Explanation:**
Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below:
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’.

**Heuristics:**
A) Either calculate the exact value of h (which is certainly time consuming).
OR
B ) Approximate the value of h using some heuristics (less time consuming).

We will discuss both of the methods.
**A) Exact Heuristics –**

We can find exact values of h, but that is generally very time consuming.
Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

**B) Approximation Heuristics –**
There are generally three approximation heuristics to calculate h –
**1) Manhattan Distance –**

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
h = abs (current_cell.x - goal.x) + (current_cell.y - goal.y)
```

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

**2) Diagonal Distance-**

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
dx = abs(current_cell.x - goal.x)
dy = abs(current_cell.y - goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

where D is the length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2) ).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

**3) Euclidean Distance-**

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
h = sqrt ((current_cell.x - goal.x)2 + current_cell.y - goal.y)2)
```

- When to use this heuristic? – When we are allowed to move in any directions.

**Limitations**

Although being the best pathfinding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

**Time Complexity**
Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target)
So the worst case time complexity is O(E), where E is the number of edges in the graph
**Algorithm**:

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor
        iii) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
      end (for loop)

    e) push q on the closed list
    end (while loop)
```

**Code:**

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def heuristic(self, n):
        H = {
            'S': 15,
            '1': 14,
            '2': 10,
            '3':  8,
            '4': 12,
            '5': 10,
            '6': 10,
            '7': 0
        }

        return H[n]

    def a_star(self, start, stop):

        open_list = set([start])
        closed_list = set([])

        distance = {} # Distance from Start.
        distance[start] = 0

        adjacent_nodes = {} # Adjacent Mapping of all Nodes
        adjacent_nodes[start] = start

        while len(open_list) > 0:
            n = None

            for v in open_list:
                if n == None or distance[v] + self.heuristic(v) < distance[n] + self.heuristic(n):
                    n = v
```

```
if n == None:
    print('Path does not exist!')
    return None

if n == stop: # If current node is stop, we restart
    reconst_path = []

    while adjacent_nodes[n] != n:
        reconst_path.append(n)
        n = adjacent_nodes[n]

    reconst_path.append(start)

    reconst_path.reverse()

    print('\nPath found: {}\n'.format(reconst_path))
    return reconst_path

for (m, weight) in self.get_neighbors(n): # Neighbours of current node

    if m not in open_list and m not in closed_list:
        open_list.add(m)
        adjacent_nodes[m] = n
        distance[m] = distance[n] + weight

    else: # Check if its quicker to visit n then m
        if distance[m] > distance[n] + weight:
            distance[m] = distance[n] + weight
            adjacent_nodes[m] = n

            if m in closed_list:
                closed_list.remove(m)
                open_list.add(m)


open_list.remove(n) # Since all neighbours are inspected
closed_list.add(n)
print("OPEN LIST : ", end="")
print(open_list)
print("CLOSED LIST : ", end="")
print(closed_list)
print("-----------------------------------")
```

```
        print('Path does not exist!')
        return None


adjacent_list2 = {
    'S': [('1', 3), ('4', 4)],
    '1': [('S', 3), ('2', 4), ('4', 5)],
    '2': [('1', 4), ('3', 4), ('5', 5)],
    '3': [('2', 4)],
    '4': [('S', 4), ('1', 5), ('5', 2)],
    '5': [('4', 2), ('2', 5), ('6', 4)],
    '6': [('5', 4), ('7', 3)],
    '7': [('6', 3)],
}

g = Graph(adjacent_list2)
g.a_star('S', '7')
```

**Output:**

```
OPEN LIST : {'1', '4'}
CLOSED LIST : {'S'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'5', '1'}
CLOSED LIST : {'S', '4'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'2', '6', '1'}
CLOSED LIST : {'S', '5', '4'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'2', '6'}
CLOSED LIST : {'S', '1', '5', '4'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'6', '3'}
CLOSED LIST : {'2', '5', 'S', '1', '4'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'6'}
CLOSED LIST : {'2', '5', 'S', '3', '1', '4'}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
OPEN LIST : {'7'}
CLOSED LIST : {'2', '5', '6', 'S', '3', '1', '4'}
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Path found: ['S', '4', '5', '6', '7']
```

**Conclusion**: We learnt about A star algorithm, its uses, limitations and implemented it in a python code.

# EXPERIMENT 4

**Aim:** Program to implement Local Search algorithm: Hill climbing search.

**Theory:**
Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
A node of hill climbing algorithm has two components which are state and value.
Hill Climbing is mostly used when a good heuristic is available.
In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Following are some main features of Hill Climbing Algorithm:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The - Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

Different regions in the state space landscape:
- Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

- Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

- Current state: It is a state in a landscape diagram where an agent is currently present.

- Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

- Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:
- Simple hill Climbing:

- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

Algorithm

```
Step 1: Evaluate the initial state. If it is a goal state then stop and return success.
Otherwise, make initial state as current state.

Step 2: Loop until the solution state is found or there are no new operators present
which can be applied to the current state.
 a) Select a state that has not been yet applied to the current state and apply it to
produce a new state.
b) Perform these to evaluate new state:
1. If the current state is a goal state, then stop and return success.
2. If it is better than the current state, then make it current state and proceed
further. 3. If it is not better than the current state, then continue in the loop until a
solution is found.

Step 3: Exit.
```

**CODE:**

```python
import copy
visited_states = []
def gn(curr_state,prev_heu,goal_state):
    global visited_states
    state = copy.deepcopy(curr_state)
    for i in range(len(state)):
        temp = copy.deepcopy(state)
        if len(temp[i]) > 0:
            elem = temp[i].pop()
            for j in range(len(temp)):
                temp1 = copy.deepcopy(temp)
                if j != i:
                    temp1[j] = temp1[j] + [elem]
                    if (temp1 not in visited_states):
                        curr_heu=heuristic(temp1,goal_state)
                        if curr_heu>prev_heu:
                            child = copy.deepcopy(temp1)
                            return child
    return 0
def heuristic(curr_state,goal_state):
    goal_=goal_state[3]
    val=0
    for i in range(len(curr_state)):
        check_val=curr_state[i]
        if len(check_val)>0:
            for j in range(len(check_val)):
                if check_val[j]!=goal_[j]:
                    # val-=1
                    val-=j
                else:
                    # val+=1
                    val+=j
    return val

def sln(init_state,goal_state):
    global visited_states
    if (init_state == goal_state):
        print (goal_state)
        print("solution found!")
        return
    current_state = copy.deepcopy(init_state)

    while(True):
        visited_states.append(copy.deepcopy(current_state))
        print(current_state)
        prev_heu=heuristic(current_state,goal_state)
        child = gn(current_state,prev_heu,goal_state)
        if child==0:
            print("Final state - ",current_state)
            return

        current_state = copy.deepcopy(child)

def main():
    global visited_states
    initial = [[],[],[],['B','C','D','A']]
    goal = [[],[],[],['A','B','C','D']]
    sln(initial,goal)
main()
```

OUTPUT:

```
[[], [], [], ['B', 'C', 'D', 'A']]
[['A'], [], [], ['B', 'C', 'D']]
[['A', 'D'], [], [], ['B', 'C']]
[['A'], ['D'], [], ['B', 'C']]
[['A'], ['D'], ['C'], ['B']]
[['A', 'B'], ['D'], ['C'], []]
[['A', 'B', 'C'], ['D'], [], []]
[['A', 'B', 'C', 'D'], [], [], []]
Final state -  [['A', 'B', 'C', 'D'], [], [], []]
```

**Conclusion:** Hill Climbing algorithm is good in solving the optimization problem while using only limited computation power. We learnt about the Hill Climbing algorithm and implemented it in a python program.

# EXPERIMENT 5

**AIM:** Program on Genetic Algorithm to solve an optimization problem in AI.

## Genetic Algorithm

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Genetic algorithms simulate the process of natural selection which means those species who can adapt to changes in their environment are able to survive and reproduce and go to next generation. In simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. Each generation consist of a population of individuals and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

Five phases are considered in a genetic algorithm.

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

**Initial Population**

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve.
An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a Chromosome (solution).
In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.

**Fitness Function**

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

**Selection**
The idea of the selection phase is to select the fittest individuals and let them pass their genes to the next generation.
Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chances to be selected for reproduction.

**Crossover**
Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.
For example, consider the crossover point to be 3 as shown below.

**Mutation**
In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

Mutation: Before and After
Mutation occurs to maintain diversity within the population and prevent premature convergence.

**Termination**
The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

ALGORITHM:

```
Initialize a random population of individuals
Compute fitness of each individual
WHILE NOT finished DO
BEGIN /* produce new generation */
FOR population_size DO
BEGIN /* reproductive cycle */
Select two individuals from old generation, recombine the
two individuals to give two offspring
Make a mutation for selected individuals by altering a
random bit in a string
Create a new generation (new populations)
END
IF population has converged THEN
finished := TRUE
END
```

**Why use Genetic Algorithms**

- They are Robust
- Provide optimisation over large space state.
- Unlike traditional AI, they do not break on slight change in input or presence of noise
- Application of Genetic Algorithms

**Genetic algorithms have many applications, some of them are –**

- Recurrent Neural Network
- Mutation testing
- Code breaking
- Filtering and signal processing
- Learning fuzzy rule base etc

CODE:

```
from numpy.random import randint
from numpy.random import rand
import math

def binary_to_decimal(bin):
  decimal=0
  for i in range(len(bin)):
   decimal+=bin[i]*pow(2, 4-i)
  return decimal

def decimal_to_binary(dec):
	binaryVal=[]
	while(dec>0):
		binaryVal.append(dec%2)
		dec=math.floor(dec/2)
	for _ in range(5-len(binaryVal)):
		binaryVal.append(0)
	binaryVal=binaryVal[::-1]
	return binaryVal

def crossover(parent1,parent2,r_cross):
	child1,child2 = parent1.copy(), parent2.copy()
	r = rand()
	point = 0
```

```
        if r > r_cross:
                point = randint(1,len(parent1)-2)
                child1 = parent1[:point] + parent2[point:]
                child2 = parent2[:point] + parent1[point:]
        return child1,child2,point

def mutation(chromosome,r_mut):
    for i in range(len(chromosome)):
        if rand()<r_mut:
            chromosome[i] = 1 - chromosome[i]
    return chromosome

def fitness_function(x):
        return pow(x,2)

def genetic_algorithm(iterations, population_size, r_cross, r_mut):
        input = [randint(0, 32) for _ in range(population_size)]
        pop = [decimal_to_binary(i) for i in input]
        for generation in range(iterations):
                print(f'\nITERATION : {generation+1}',end='\n\n')
                decimal = [binary_to_decimal(i) for i in pop]
                fitness_score = [fitness_function(i) for i in decimal]
                f_by_sum = [fitness_score[i]/sum(fitness_score) for i in range(population_size)]
                exp_cnt = [fitness_score[i]/(sum(fitness_score)/population_size) for i in
range(population_size)]
                act_cnt = [round(exp_cnt[i]) for i in range(population_size)]
                print('SELECTION\n\nInitial \tDecimal Value\tFitness Score\t\tFi/Sum\t\tExpected
\tActual ')
                for i in range(population_size):

print(pop[i],'\t',decimal[i],'\t\t',fitness_score[i],'\t\t',round(f_by_sum[i],2),'\t\t',round(exp_cnt[i],2),'\t\t',a
ct_cnt[i])
                print('Sum : ',sum(fitness_score))
                print('Average : ',sum(fitness_score)/population_size)
                print('Maximum : ',max(fitness_score),end='\n')
                max_count = max(act_cnt)
                min_count = min(act_cnt)
                max_count_index = 0
                for i in range(population_size):
                        if max_count == act_cnt[i]:
                                maxIndex=i
                                break
                for i in range(population_size):
```

```
            if min_count == act_cnt[i]:
                pop[i] = pop[max_count_index]
        crossover_children = list()
        crossover_point = list()
        for i in range(0,population_size,2):
            child1, child2, point_of_crossover = crossover(pop[i],pop[i+1],r_cross)
            crossover_children.append(child1)
            crossover_children.append(child2)
            crossover_point.append(point_of_crossover)
            crossover_point.append(point_of_crossover)
        print("\nCROSS OVER\n\nPopulation\t\tMate\t Crossover Point\t Crossover
Population")
        for i in range(population_size):
            if (i+1)%2 == 1:
                mate = i+2
            else:
                mate = i
            print(pop[i],'\t',mate,'\t',crossover_point[i],'\t\t\t',crossover_children[i])
        mutation_children = list()
        for i in range(population_size):
            child = crossover_children[i]
            mutation_children.append(mutation(child,r_mut))
        new_population = list()
        new_fitness_score = list()
        for i in mutation_children:
            new_population.append(binary_to_decimal(i))
        for i in new_population:
            new_fitness_score.append(fitness_function(i))
        print("\nMUTATION\n\nMutation population\t New Population\t Fitness")
        for i in range(population_size):
            print(mutation_children[i],'\t',new_population[i],'\t\t',new_fitness_score[i])
        print('Sum : ',sum(new_fitness_score))
        print('Maximum : ',max(new_fitness_score))
        pop = mutation_children
        print("----------------------------------------------------------------------------------------------------")

def main():
    iterations = 3
    population_size = 4
    r_cross = 0.5
    r_mut = 0.05
    genetic_algorithm(iterations,population_size,r_cross,r_mut)
```

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

**OUTPUT:**

```
ITERATION : 1

SELECTION

Initial         Decimal Value  Fitness Score         Fi/Sum        Expected        Actual
[0, 1, 0, 0, 1]        9           81           0.11         0.43          0
[1, 0, 1, 0, 0]        20           400          0.53         2.12          2
[0, 0, 1, 1, 1]        7           49           0.06         0.26          0
[0, 1, 1, 1, 1]        15           225          0.3          1.19          1
Sum :  755
Average :  188.75
Maximum :  400

CROSS OVER

Population              Mate    Crossover Point        Crossover Population
[0, 1, 0, 0, 1]        2      0                   [0, 1, 0, 0, 1]
[1, 0, 1, 0, 0]        1      0                   [1, 0, 1, 0, 0]
[0, 1, 0, 0, 1]        4      2                   [0, 1, 1, 1, 1]
[0, 1, 1, 1, 1]        3      2                   [0, 1, 0, 0, 1]

MUTATION

Mutation population     New Population  Fitness
[0, 1, 0, 0, 1]        9           81
[1, 0, 1, 0, 0]        20           400
[0, 1, 1, 1, 1]        15           225
[0, 1, 0, 0, 1]        9           81
Sum :  787
Maximum :  400
----------------------------------------------------------------------------------------------
ITERATION : 2

SELECTION

Initial         Decimal Value  Fitness Score         Fi/Sum        Expected        Actual
```

```
[0, 1, 0, 0, 1]      9          81          0.1          0.41         0
[1, 0, 1, 0, 0]      20          400          0.51          2.03          2
[0, 1, 1, 1, 1]      15          225          0.29          1.14          1
[0, 1, 0, 0, 1]      9          81          0.1          0.41         0
Sum :  787
Average :  196.75
Maximum :  400

CROSS OVER
Population            Mate    Crossover Point       Crossover Population
[0, 1, 0, 0, 1]      2      0                  [0, 1, 0, 0, 1]
[1, 0, 1, 0, 0]      1      0                  [1, 0, 1, 0, 0]
[0, 1, 1, 1, 1]      4      2                  [0, 1, 0, 0, 1]
[0, 1, 0, 0, 1]      3      2                  [0, 1, 1, 1, 1]

MUTATION

Mutation population     New Population  Fitness
[0, 1, 0, 0, 1]      9          81
[1, 0, 1, 0, 0]      20          400
[0, 1, 1, 0, 1]      13          169
[0, 1, 1, 1, 1]      15          225
Sum :  875
Maximum :  400
------------------------------------------------------------------------------------------
ITERATION : 3

SELECTION

Initial        Decimal Value  Fitness Score         Fi/Sum         Expected        Actual
[0, 1, 0, 0, 1]      9          81          0.09         0.37          0
[1, 0, 1, 0, 0]      20          400          0.46          1.83          2
[0, 1, 1, 0, 1]      13          169          0.19          0.77          1
[0, 1, 1, 1, 1]      15          225          0.26          1.03          1
Sum :  875
Average :  218.75
Maximum :  400

CROSS OVER

Population            Mate    Crossover Point       Crossover Population
[0, 1, 0, 0, 1]      2      1                  [0, 0, 1, 0, 0]
[1, 0, 1, 0, 0]      1      1                  [1, 1, 0, 0, 1]
```

```
[0, 1, 1, 0, 1]     4    0          [0, 1, 1, 0, 1]
[0, 1, 1, 1, 1]     3    0          [0, 1, 1, 1, 1]

MUTATION

Mutation population    New Population  Fitness
[0, 0, 1, 0, 0]     4        16
[1, 1, 0, 0, 1]     25        625
[0, 1, 1, 0, 1]     13        169
[0, 1, 1, 0, 1]     13        169
Sum :  979
Maximum :  625
----------------------------------------------------------------------------------------------------
```

**CONCLUSION**:
We learnt about the Genetic Algorithm, its workings and its uses and also implemented it in a python program. We also learnt about other terms associated with genetic algorithm such as crossover, mutation, fitness score, etc.

# EXPERIMENT 6

## CASE STUDY

**AIM:** Case study of an AI Application.

**Paper Link:** https://ieeexplore.ieee.org/abstract/document/9325622

**Introduction**:
This paper is a comparative study for deep reinforcement learning with CNN, RNN, and LSTM in autonomous navigation. For the comparison, a PyGame simulator has been used with the final goal that the representative will learn to move without hitting four different fixed obstacles. Autonomous vehicle movements were simulated in the training environment and the conclusion drawn was that the LSTM model was better than the others.

**Approach**:
The research is wholly based on reinforcement learning which is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones. This method assigns positive values to the desired actions to encourage the agent and negative values to undesired behaviors. This programs the agent to seek long-term and maximum overall reward to achieve an optimal solution.

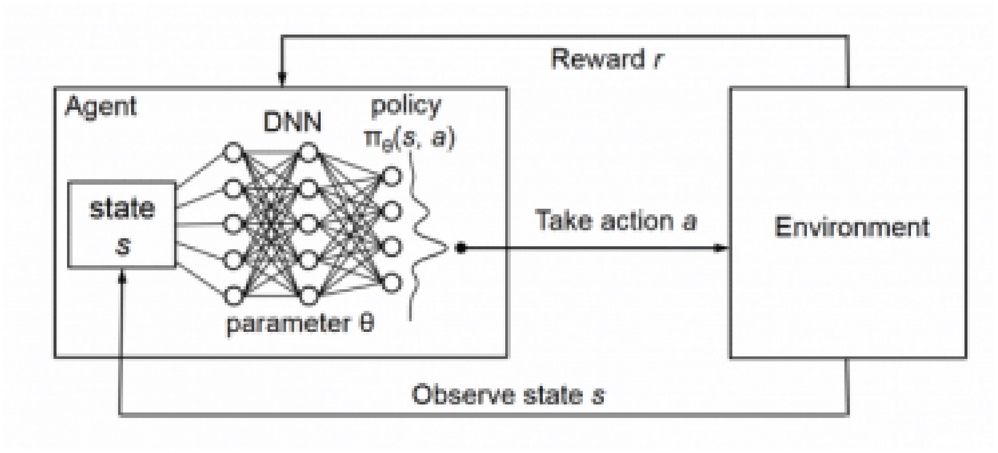
These long-term goals help prevent the agent from stalling on lesser goals. With time, the agent learns to avoid the negative and seek the positive. This learning method has been adopted in artificial intelligence (AI) as a way of directing unsupervised machine learning through rewards and penalties.

The main advantage of reinforcement learning in this scenario is that unlike deep learning (DL) algorithms, it does not require a data set during the training phase, increasing its popularity and making it more suitable.

The PyGame simulator interface consists of an agent that learns to move without hitting 4 different randomly positioned obstacles and edges limiting the area. In addition, the paper presents a model-free, off policy approach in this study.

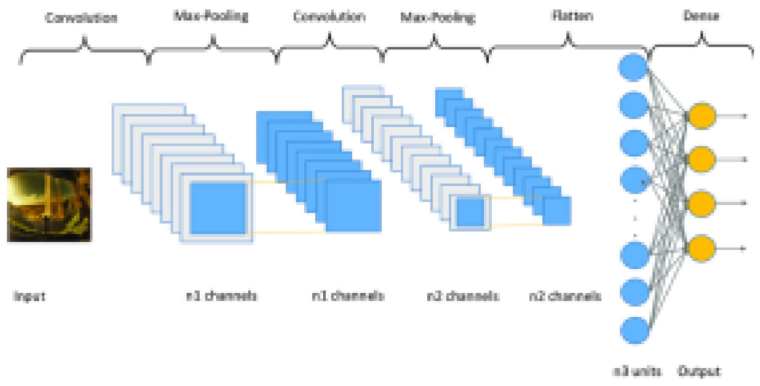
During the research, 4 algorithms were compared. They are:

1) **Deep Q-Network**:It trains on inputs that represent active players in areas or other experienced samples and learns to match those data with desired outputs. This is a powerful method in the development of artificial intelligence that can play games like chess at a high level, or carry out other high-level cognitive activities – the Atari or chess video game playing example is also a good example of how AI uses the types of interfaces that were traditionally used by human agents.

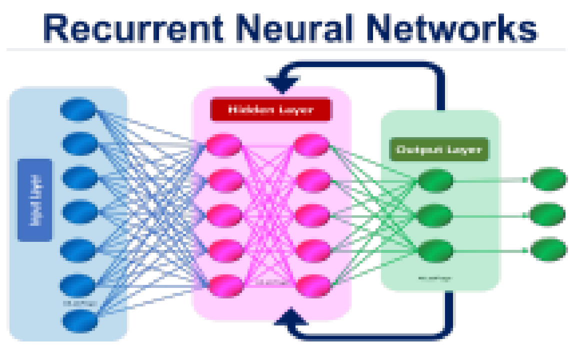


2) **CNN:** A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.
The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.
The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

The main purpose of the convolution process is to extract the feature map from the input data.

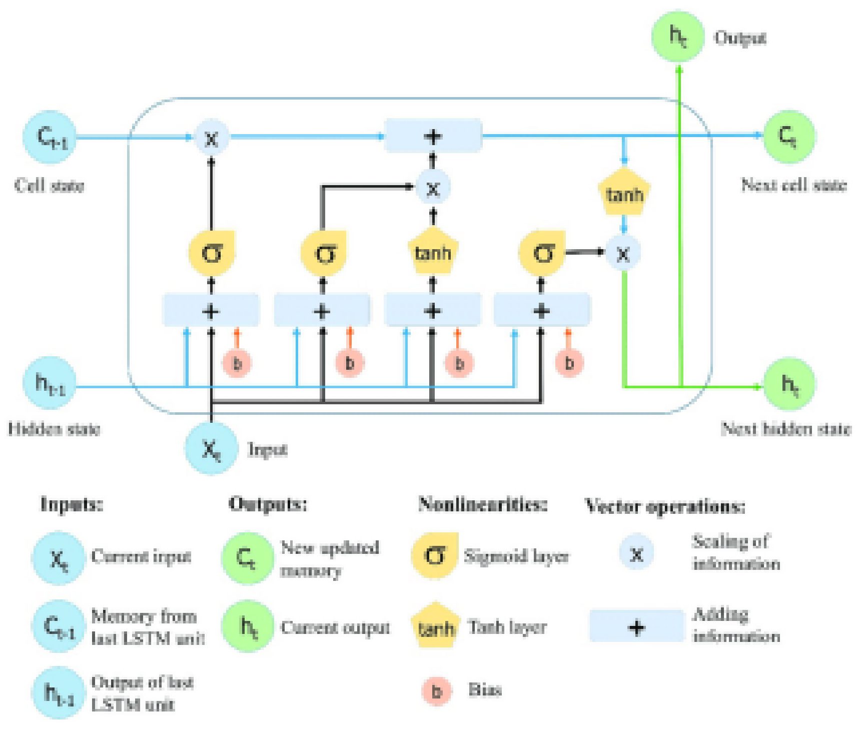


3) **RNN:** Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is the Hidden state, which stores some information about a sequence.



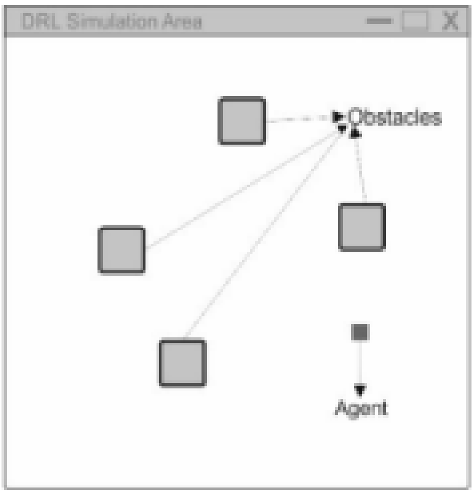
**4) LSTM:** Long Short Term Memory is a kind of recurrent neural network. In RNN output from the last step is fed as input in the current step. LSTM was designed by Hochreiter & Schmidhuber. It tackled the problem of long-term dependencies of RNN in which the RNN cannot predict the word stored in the long-term memory but can give more accurate predictions from the recent information. As the gap length increases RNN does not give an efficient performance. LSTM can by default retain the information for a long period of time. It is used for processing, predicting, and classifying on the basis of time-series data. LSTM has a chain structure that contains four neural networks and different memory blocks called cells. Information is retained by the cells and the memory manipulations are done by the gates. There are three gates – Forget gate, Input gate and the output gate.

With LSTMs, there is no need to keep a finite number of states from beforehand as required in the hidden Markov model (HMM). LSTMs provide us with a large range of parameters such as learning rates, and input and output biases. Hence, no need for fine adjustments. The complexity to update each weight is reduced to O(1) with LSTMs, similar to that of Back Propagation Through Time (BPTT), which is an advantage.



**SIMULATION**:

In this work, PyGame library was used as a robot simulation environment. 4 different obstacles were placed randomly in a 360 * 360 pixel area and the agent was allowed to float within the specified area without hitting these obstacles as shown below.



The agent loses -150 points when hitting obstacles during the learning phase and -50 points when it hits walls. It gets +2 points for every step where it does not hit walls and obstacles.

Four different actions in this simulation are shown in the table below.

| Num. | Action |
|---|---|
| 0 | go to the left |
| 1 | go to the right |
| 2 | go to the up |
| 3 | go to the bottom |

For the model training, python was used as the software language and Keras, a deep learning library was used to create the neural networks. Mean Squared Error was used as the Loss function. Linear was preferred as the activation function. Sigmoid function is used as the activation function in the output layer. The training took 5 hours for CNN, 18 hours for RNN and 35 hours for LSTM on a standard equipped (core i5 Processor and 8GB RAM) computer

**CONCLUSION**:
Multiple deep learning algorithms were separately tested on the PyGame simulation interface and the conclusions were drawn. The first conclusion was that even though deep reinforcement learning (DRL)

models provide fast and safe solutions for autonomous vehicles, their training time is very high. After training it was observed that RNN and LSTM,which are generally used to solve language processing problems, can also be successful in such autonomous navigation problems. The second conclusion was that while the LSTM model took the maximum time to train, it showed the highest success in the success-episode graphics. The paper then concludes with saying that this is a very rich field in terms of future research prospects.

# EXPERIMENT 7

**Aim:** Program to implement learning: Perceptron Learning/Backpropagation Algorithm

**Theory**:
Perceptrons are a type of artificial neuron that predates the sigmoid neuron. It appears that they were invented in 1957 by Frank Rosenblatt at the Cornell Aeronautical Laboratory.

A perceptron can have any number of inputs, and produces a binary output, which is called its activation.

First, we assign each input a weight, loosely meaning the amount of influence the input has over the output.
To determine the perceptron's activation, we take the weighted sum of each of the inputs and then determine if it is above or below a certain threshold, or *bias, *represented by b.

The formula for perceptron neurons can can be expressed like this:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

**Algorithm**:

```
def perceptron(inputs, bias)

 weighted_sum = sum {
  for each input in inputs
    input.value * input.weight
  }

  if weighted_sum <= bias
    return 0
  if weighted_sum > bias
    return 1

end
```

**Code**:

```
def sgn(net_input):
    if net_input <= 0 :
        return -1
    return 1

def pattern_classifier(n_iterations, input, weight, desired_output, learning_rate):
    for iteration in range(n_iterations):
        print(f'Iteration {iteration+1}')
        output = []
        for i,X in enumerate(input):
            net_input = 0
            for j in range(len(X)):
                net_input+=weight[j]*X[j]
            generated_output = sgn(net_input)
            output.append(generated_output)
            if generated_output != desired_output[i]:
                difference = desired_output[i] - generated_output
                for position in range(len(weight)):
                    weight[position] = float("{:.2f}".format(weight[position] +
learning_rate*difference*X[position]))
        print(f'Generated Output vector for Iteration {iteration+1} : {output}')
        print(f'Weight vector after Iteration {iteration+1} : {weight}')
        print("------"*25)
        if output == desired_output:
            break
    return output,weight

def main():
    input = [
        [1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1], #L starts here
        [1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,1,1,1,1],
        [1,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1],
        [0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,1,1,1,1],
        [1,0,0,0,0,1,0,0,0,0,1,0,0,0,1,1,1,1,1,1],
        [0,1,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,1,1,1],
        [0,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,1],
        [1,0,0,0,0,1,0,0,0,0,1,0,1,0,0,1,1,0,1,1],
        [0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,1,1,0,1,1],
        [1,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,0,1,1],
        [0,1,0,1,0,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1], #M starts here
        [1,0,0,0,1,1,1,0,1,1,1,0,1,0,1,1,0,0,0,1],
        [1,0,0,0,1,1,1,0,1,1,1,0,1,0,1,1,0,1,0,1],
        [1,1,0,1,1,1,0,1,0,1,1,0,1,0,1,1,0,0,0,1],
        [1,1,0,1,1,1,0,1,0,1,1,0,0,0,1,1,0,0,0,1],
        [1,0,0,0,1,1,1,0,1,1,1,0,0,0,1,1,0,0,0,1],
        [1,0,0,0,1,1,1,0,1,1,1,1,0,1,1,1,0,1,0,1],
        [1,1,0,1,1,1,0,1,0,1,1,0,1,0,1,1,0,1,0,1],
        [1,0,0,0,1,1,1,0,1,1,1,0,1,0,1,0,0,0,0,0],
        [1,0,0,0,1,1,1,1,1,1,1,0,1,0,1,1,0,0,0,1],
    ]
    desired_output = [1,1,1,1,1,1,1,1,1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
    initial_weight = [1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1,0,1,1]
    learning_rate = 0.05
    n_iterations = 3

    classification_output, weight_vector = pattern_classifier(n_iterations, input,
initial_weight, desired_output, learning_rate)

    count = 0
    for i, output in enumerate(classification_output):
        if output == desired_output[i]:
            count+=1

    accuracy = (count / len(input))*100

    print(f'Accuracy of Classifier : {accuracy} %')

    print('Classifying an Unknown Sample of L (Output = 1)')
    unknown_sample = [1,1,0,0,0,1,0,0,0,0,1,0,0,0,0,1,1,1,1,0]
    print('Unknown Sample : ',unknown_sample)
    net_input=0
    for i in range(len(unknown_sample)):
        net_input+=weight_vector[i]*unknown_sample[i]
    predicted_output = sgn(net_input)
    print('Predicted Output : ', predicted_output)
    print("\n")

main()
```

**Output**:

```
Iteration 1
Generated Output vector for Iteration 1 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, 1]
Weight vector after Iteration 1 : [0.2, 0.6, 0.0, 0.6, 0.2, -0.9, 0.4, 0.6, -0.6, 0.1, 0.1, -0.1, 0.4,
0.9, -0.9, 0.1, 1.0, -0.3, 1.0, 0.1]
--------------------------------------------------------------------------------------------------------
Iteration 2
Generated Output vector for Iteration 2 : [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 2 : [0.1, 0.5, 0.0, 0.5, 0.1, -1.0, 0.4, 0.5, -0.6, 0.0, 0.0, -0.1, 0.3,
0.9, -1.0, 0.0, 1.0, -0.3, 1.0, 0.0]
--------------------------------------------------------------------------------------------------------
Iteration 3
Generated Output vector for Iteration 3 : [1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, -1,
-1]
Weight vector after Iteration 3 : [0.1, 0.4, 0.0, 0.4, 0.0, -1.0, 0.4, 0.4, -0.6, -0.1, 0.0, -0.1, 0.2,
0.9, -1.0, 0.0, 1.1, -0.2, 1.1, 0.0]
--------------------------------------------------------------------------------------------------------
Accuracy of Classifier : 90.0 %
Classifying an Unknown Sample of L (Output = 1)
Unknown Sample :  [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0]
Predicted Output :  1
```

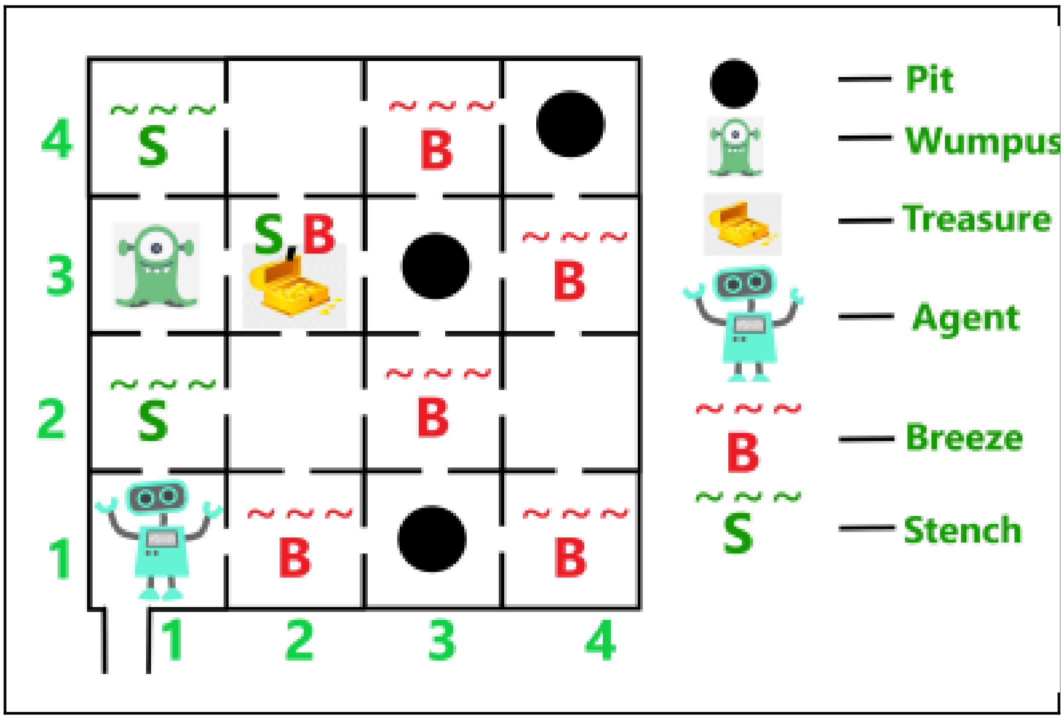
# EXPERIMENT 8

Aim: Implementation on any AI Problem : Wumpus world,

Tic-tac-toe Theory:

The Wumpus world is a cave with 16 rooms (4×4). Each room is connected to others through walkways (no rooms are connected diagonally). The knowledge-based agent starts from Room[1, 1]. The cave has – some pits, a treasure and a beast named Wumpus. The Wumpus can not move but eats the one who enters its room. If the agent enters the pit, it gets stuck there. The goal of the agent is to take the treasure and come out of the cave. The agent is rewarded, when the goal conditions are met. The agent is penalized, when it falls into a pit or is eaten by the Wumpus.

Some elements support the agent to explore the cave, like -The wumpus's adjacent rooms are stenchy. -The agent is given one arrow which it can use to kill the wumpus when facing it (Wumpus screams when it is killed). – The adjacent rooms of the room with pits are filled with breeze. -The treasure room is always glittery.



PEAS description of Wumpus world:
To explain the Wumpus world we have given PEAS description as below:

**Performance measure:**
● +1000 reward points if the agent comes out of the cave with the gold.
● -1000 points penalty for being eaten by the Wumpus or falling into the pit.
● -1 for each action, and -10 for using an arrow.
● The game ends if either agent dies or came out of the cave. Environment:
● A 4*4 grids of rooms.
● The agent initially in room square [1, 1], facing toward the right.
● Location of Wumpus and gold are chosen randomly except the first square [1,1].
● Each square of the cave can be a pit with probability 0.2 except the first square.

**Actuators:**
● Left turn,
● Right turn
● Move forward
● Grab
● Release
● Shoot.

**Sensors:**
● The agent will perceive the stench if he is in the room adjacent to the Wumpus. (Not diagonally).
● The agent will perceive a breeze if he is in the room directly adjacent to the Pit.
● The agent will perceive the glitter in the room where the gold is present. ● The agent will perceive the bump if he walks into a wall.
● When the Wumpus is shot, it emits a horrible scream which can be perceived anywhere in the cave.
● These precepts can be represented as a five element list, in which we will have different indicators for each sensor.
● Example if agent perceives stench, breeze, but no glitter, no bump, and no scream then it can be represented as:
[Stench, Breeze, None, None, None].

**The Wumpus world Properties:**
● Partially observable: The Wumpus world is partially observable because the agent can only perceive the close environment such as an adjacent room.
● Deterministic: It is deterministic, as the result and outcome of the world are already known.
● Sequential: The order is important, so it is sequential.
● Static: It is static as Wumpus and Pits are not moving.

● Discrete: The environment is discrete.
● One agent: The environment is a single agent as we have one agent only and Wumpus is not considered as an agent.

Code:

```
import java.util.Scanner;

class WumpusWorld {
    static Block[][] maze;
    static int n;

    public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);

     System.out.print("Enter the order of the maze: ");
     n = sc.nextInt();

     maze = new Block[n][n];
     for(int i=0; i<n; i++) {
         maze[i] = new Block[n];
         for(int j=0; j<n; j++)
             maze[i][j] = new Block();
     }

     System.out.print("\nEnter the number of pits: ");
     int pits = sc.nextInt();

     for(int i=0; i<pits; i++) {
         System.out.print("Enter the location of pit " + (i+1) + ": ");
         addPit(n-sc.nextInt(), sc.nextInt()-1);
     }

     System.out.print("\nEnter the location of wumpus: ");
     addWumpus(n-sc.nextInt(), sc.nextInt()-1);

     System.out.print("\nEnter the location of gold: ");
     addGold(n-sc.nextInt(), sc.nextInt()-1);

     System.out.print("\nEnter the starting location: ");
     int r = n - sc.nextInt();
     int c = sc.nextInt() - 1;
     int rPrev = -1, cPrev = -1;

    System.out.print("\nYour Position : *\nWumpus : X\nGold : $\nPit : O");

     int moves = 0;
     System.out.println("\nInitial state:");
     printMaze(r, c);

     while(!maze[r][c].hasGold) {
         maze[r][c].isVisited = true;
         maze[r][c].pitStatus = Block.NOT_PRESENT;
         maze[r][c].wumpusStatus = Block.NOT_PRESENT;

         if(!maze[r][c].hasBreeze) {
             if(r >= 1 && maze[r-1][c].pitStatus == Block.UNSURE)
                 maze[r-1][c].pitStatus = Block.NOT_PRESENT;
             if(r <= (n-2) && maze[r+1][c].pitStatus == Block.UNSURE)
                 maze[r+1][c].pitStatus = Block.NOT_PRESENT;
             if(c >= 1 && maze[r][c-1].pitStatus == Block.UNSURE)
                 maze[r][c-1].pitStatus = Block.NOT_PRESENT;
             if(c <= (n-2) && maze[r][c+1].pitStatus == Block.UNSURE)
                 maze[r][c+1].pitStatus = Block.NOT_PRESENT;
         }

         if(!maze[r][c].hasStench) {
             if(r >= 1 && maze[r-1][c].wumpusStatus == Block.UNSURE)
                 maze[r-1][c].wumpusStatus = Block.NOT_PRESENT;
             if(r <= (n-2) && maze[r+1][c].wumpusStatus == Block.UNSURE)
                 maze[r+1][c].wumpusStatus = Block.NOT_PRESENT;
             if(c >= 1 && maze[r][c-1].wumpusStatus == Block.UNSURE)
                 maze[r][c-1].wumpusStatus = Block.NOT_PRESENT;
             if(c <= (n-2) && maze[r][c+1].wumpusStatus == Block.UNSURE)
                 maze[r][c+1].wumpusStatus = Block.NOT_PRESENT;
         }

         boolean foundNewPath = false;

         if(r >= 1 && !((r-1) == rPrev && c == cPrev) && maze[r-1][c].isVisited == false &&
maze[r-1][c].pitStatus == Block.NOT_PRESENT && maze[r-1][c].wumpusStatus ==
Block.NOT_PRESENT) {
             rPrev = r;
             cPrev = c;
             r--;
             foundNewPath = true;
         }
         else if(r <= (n-2) && !((r+1) == rPrev && c == cPrev) && maze[r+1][c].isVisited == false
&& maze[r+1][c].pitStatus == Block.NOT_PRESENT && maze[r+1][c].wumpusStatus ==
Block.NOT_PRESENT) {
             rPrev = r;
             cPrev = c;

             r++;
             foundNewPath = true;
         }
         else if(c >= 1 && !(r == rPrev && (c-1) == cPrev) && maze[r][c-1].isVisited == false &&
maze[r][c-1].pitStatus == Block.NOT_PRESENT && maze[r][c-1].wumpusStatus ==
Block.NOT_PRESENT) {
             rPrev = r;
             cPrev = c;

             c--;
             foundNewPath = true;
         }
         else if(c <= (n-2) && !(r == rPrev && (c+1) == cPrev) && maze[r][c+1].isVisited == false
&& maze[r][c+1].pitStatus == Block.NOT_PRESENT && maze[r][c+1].wumpusStatus ==
Block.NOT_PRESENT) {
             rPrev = r;
             cPrev = c;

             c++;
             foundNewPath = true;
         }

         if(!foundNewPath) {
             int temp1 = rPrev;
             int temp2 = cPrev;

             rPrev = r;
             cPrev = c;

             r = temp1;
             c = temp2;
         }
         moves++;

         System.out.println("\n\nMove " + moves + ":");
         printMaze(r, c);

         if(moves > n*n) {
             System.out.println("\nNo solution found!");
             break;
         }
     }

     if(moves <= n*n)
         System.out.println("\nFound gold in " + moves + " moves.");

     sc.close();
   }

   static void addPit(int r, int c) {
     maze[r][c].hasPit = true;

     if(r >= 1)
         maze[r-1][c].hasBreeze = true;
     if(r <= (n-2))
         maze[r+1][c].hasBreeze = true;
     if(c >= 1)
         maze[r][c-1].hasBreeze = true;
     if(c <= (n-2))
         maze[r][c+1].hasBreeze = true;
   }

   static void addWumpus(int r, int c) {
     maze[r][c].hasWumpus = true;

     if(r >= 1)
         maze[r-1][c].hasStench = true;
     if(r <= (n-2))
         maze[r+1][c].hasStench = true;
     if(c >= 1)
         maze[r][c-1].hasStench = true;
     if(c <= (n-2))
         maze[r][c+1].hasStench = true;
   }
```

```
    static void addGold(int r, int c) {
     maze[r][c].hasGold = true;
    }

    static void printMaze(int r, int c) {
     for(int i=0; i<n; i++) {
        for(int j=0; j<n; j++) {
            char charToPrint = '-';
            if(r == i && c == j)
                charToPrint = '*';
            else if(maze[i][j].hasPit)
                charToPrint = 'O';
            else if(maze[i][j].hasWumpus)
                charToPrint = 'X';
            else if(maze[i][j].hasGold)
                charToPrint = '$';

            System.out.print(charToPrint + "\t");
        }
        System.out.println();
     }
    }
}
```

Output:

```
Enter the order of the maze: 4

Enter the number of pits: 2
Enter the location of pit 1: 1 4
Enter the location of pit 2: 3 4

Enter the location of wumpus: 2 4

Enter the location of gold: 2 3

Enter the starting location: 1 1

Your Position : *
```

```
Wumpus : X
Gold : $
Pit : O
Initial state:
-    -    -    -
-    -    -    O
-    -    $    X
*    -    -    O


Move 1:
-    -    -    -
-    -    -    O
*    -    $    X
-    -    -    O


Move 2:
-    -    -    -
*    -    -    O
-    -    $    X
-    -    -    O


Move 3:
*    -    -    -
-    -    -    O
-    -    $    X
-    -    -    O


Move 4:
-    *    -    -
-    -    -    O
-    -    $    X
-    -    -    O


Move 5:
-    -    -    -
-    *    -    O
-    -    $    X
-    -    -    O
```

```
Move 6:
-    -    -    -
-    -    -    O
-    *    $    X
-    -    -    O


Move 7:
-    -    -    -
-    -    -    O
-    -    $    X
-    *    -    O


Move 8:
-    -    -    -
-    -    -    O
-    -    $    X
-    -    *    O


Move 9:
-    -    -    -
-    -    -    O
-    -    *    X
-    -    -    O

Found gold in 9 moves.
```

**Conclusion**: We learnt about Knowledge based agents and about the Wumpus World game. We then explored the PEAS of an agent in the Wumpus World game and wrote a code in java to simulate the game.

# EXPERIMENT 9

**Aim**: Program to implement Family Tree in Prolog.

Theory:

- Prolog is a language built around the Logical Paradigm: a declarative approach to problem-solving.
- There are only three basic constructs in Prolog: facts, rules, and queries.
- knowledge base (or a database): A collection of facts and rules is called
- Relationship is one of the main features that we have to properly mention in Prolog.
- These relationships can be expressed as facts and rules.
- In Prolog programs, it specifies relationship between objects and properties of the objects.
- There are various kinds of relationships, of which some can be rules as well. A rule can find out about a relationship even if the relationship is not defined explicitly as a fact.

Code:

```
male(shankar).
male(ulhas).
male(satish).
male(saurabh).
male(prashant).

female(umabai).
female(mrunal).
female(sadhana).
female(swati).

parent(shankar,umabai,ulhas).
parent(shankar,umabai,satish).
parent(ulhas,mrunal,prashant).
parent(satish,sadhana,saurabh).
parent(satish,sadhana,swati).
```

```
brother(ulhas,satish).
brother(satish,ulhas).
brother(prashant,saurabh).
brother(saurabh,prashant).
sister(swati,saurabh).
sister(swati,prashant).

father(X,Y) :- parent(X,Z,Y).
mother(X,Y) :- parent(Z,X,Y).

son(X,Y,Z) :- male(X),father(Y,X),mother(Z,X).
daughter(X,Y,Z) :- female(X),father(Y,X),mother(Z,X).

wife(X,Y) :- female(X),parent(Y,X,Z).

grandfather(X,Y) :- male(X),father(X,Z),father(Z,Y).

uncle(X,Y) :- male(X),brother(X,Z),father(Z,Y).

aunt(X,Y) :- wife(X,Z),uncle(Z,Y).

cousin(X,Y) :- father(Z,X),brother(Z,W),father(W,Y).

ancestor(X,Y,Z) :- parent(X,Y,Z).
ancestor(X,Y,Z) :- parent(X,Y,W),ancestor(W,U,Z).
```

Output:

```
true.

?- father(X,Y).
X = shankar,
Y = ulhas ;
X = shankar,
Y = satish ;
X = ulhas,
Y = prashant ;
X = satish,
Y = saurabh ;
X = satish,
Y = swati.
```

```
?- mother(X,Y).
X = umabai,
Y = ulhas ;
X = umabai,
Y = satish ;
X = mrunal,
Y = prashant ;
X = sadhana,
Y = saurabh ;
X = sadhana,
Y = swati.

?- parent(X,Y,Z).
X = shankar,
Y = umabai,
Z = ulhas ;
X = shankar,
Y = umabai,
Z = satish ;
X = ulhas,
Y = mrunal,
Z = prashant ;
X = satish,
Y = sadhana,
Z = saurabh ;
X = satish,
Y = sadhana,
Z = swati.

?- grandfather(X,Y).
X = shankar,
Y = prashant ;
X = shankar,
Y = saurabh ;
X = shankar,
Y = swati ;
false.

?- aunt(X,Y).
X = mrunal,
Y = saurabh ;
X = mrunal,
Y = swati ;
```

```
X = sadhana,
Y = prashant ;
X = sadhana,
Y = prashant ;
false.
```

**Conclusion**: We learnt about a very useful logical programming language called Prolog which is associated with artificial intelligence and computational linguistics.It is well-suited for specific tasks that benefit from rule-based logical queries such as searching databases, voice control systems, and filling templates. We wrote a program that implemented a family tree and displayed the output.

# EXPERIMENT 10

**AIM:** Identify, analyze, implement a planning problem/Rule based Expert System in a real world scenario.

**THEORY**

**RULES**

Any rule consists of two parts: the IF part, called the antecedent (premise or condition) and the THEN part called the consequent (conclusion or action). The basic syntax of a rule is:

```
IF <antecedent>
THEN <consequent>
```

In general, a rule can have multiple antecedents joined by the keywords AND (conjunction), OR (disjunction) or a combination of both. However, it is a good habit to avoid mixing conjunctions and disjunctions in the same rule. The antecedent of a rule incorporates two parts: an object (linguistic object) and its value. In our road crossing example, the linguistic object 'traffic light' can take either the value green or the value red. The object and its value are linked by an operator. The operator identifies the object and assigns the value. Operators such as is, are, is not, are not are used to assign a symbolic value to a linguistic object. But expert systems can also use mathematical operators to define an object as numerical and assign it to the numerical value.

Rules can represent relations, recommendations, Rules can represent relations, recommendations, directives, strategies and heuristics: directives, strategies and heuristics.

„ **Relation**

IF the 'fuel tank' is empty
THEN the car is dead

„ **Recommendation**

IF the season is autumn AND the sky is cloudy AND the forecast is drizzle
THEN the advice is 'take an umbrella'

„ **Directive**

F the car is dead AND the 'fuel tank' is empty

THEN the action is ‘refuel the car’

„ **Strategy**
IF the car is dead
THEN the action is ‘check the fuel tank’; the action is ‘check the fuel tank’;

„ **Heuristic**
IF the spill is liquid AND the ‘spill pH’ < 6 AND the ‘spill smell’ is vinegar the ‘spill smell’ is vinegar
THEN the ‘spill material’ is ‘acetic acid’ the ‘spill material’ is ‘acetic acid’

**EXPERT SYSTEM**

In artificial intelligence, an expert system is a computer system that emulates the decision-making ability of a human expert. Expert systems are designed to solve complex problems by reasoning about knowledge, like an expert, and not by following the procedure of a developer as is the case in conventional programming. The first expert systems were created in the 1970s and then proliferated in the 1980s. Expert systems were among the first truly successful forms of AI software.

As soon as knowledge is provided by a human expert, we can input it into a computer. We expect the computer to

1. Act as an intelligent assistant in some specific domain of expertise or to solve a problem that would otherwise have to be solved by an expert.
2. Be able to integrate new knowledge and to show its knowledge in a form that is easy to read and understand, and to deal with simple sentences in a natural language rather than an artificial programming language.
3. Explain how it reaches a particular conclusion.

In other words, we have to build an expert system, a computer program capable of performing at the level of a human expert in a narrow problem area.

The most popular expert systems are rule-based systems. A great number have been built and successfully applied in such areas as business and engineering, medicine and geology, power systems and mining.

**THE APPLICATIONS OF EXPERT SYSTEMS**

The spectrum of applications of expert systems technology to industrial and commercial problems is so wide as to defy easy characterization. The applications find their way into most areas of knowledge work. They are as varied as helping salespersons sell modular factory-built homes to helping NASA plan the maintenance of a space shuttle in preparation for its next flight.

Applications tend to cluster into seven major classes.

1- **Diagnosis and Troubleshooting of Devices and Systems of All Kinds**: This class comprises systems that deduce faults and suggest corrective actions for a malfunctioning device or process. Medical diagnosis was one of the first knowledge areas to which ES technology was applied, but diagnosis of engineered systems quickly surpassed medical diagnosis. There are probably more diagnostic applications of ES than any other type. The diagnostic problem can be stated in the abstract as: given the evidence presenting itself, what is the underlying problem/reason/cause?

2- **Planning and Scheduling**: Systems that fall into this class analyze a set of one or more potentially complex and interacting goals in order to determine a set of actions to achieve those goals, and/or provide a detailed temporal ordering of those actions, taking into account personnel, materiel, and other constraints. This class has great commercial potential, which has been recognized. Examples involve airline scheduling of flights, personnel, and gates; manufacturing job-shop scheduling; and manufacturing process planning.

3- **Configuration of Manufactured Objects from Subassemblies**: Configuration, whereby a solution to a problem is synthesized from a given set of elements related by a set of constraints, is historically one of the most important of expert system applications. Configuration applications were pioneered by computer companies as a means of facilitating the manufacture of semi-custom minicomputers. The technique has found its way into use in many different industries, for example, modular home building, manufacturing, and other problems involving complex engineering design and manufacturing.

4- **Financial Decision Making**: The financial services industry has been a vigorous user of expert system techniques. Advisory programs have been created to assist bankers in determining whether to make loans to businesses and individuals. Insurance companies have used expert systems to assess the risk presented by the customer and to determine a price for the insurance. A typical application in the financial markets is in foreign exchange trading.

5- **Knowledge Publishing** This is a relatively new, but also potentially explosive area. The primary function of the expert system is to deliver knowledge that is relevant to the user's problem, in the context of the user's problem. The two most widely distributed expert systems in the world are in this category. The first is an advisor which counsels a user on appropriate grammatical usage in a text. The second is a tax advisor that accompanies a tax preparation program and advises the user on tax strategy, tactics, and individual tax policy.

6- **Process Monitoring and Control**: Systems falling in this class analyze real time data from physical devices with the goal of noticing anomalies, predicting trends, and controlling for both optimality and failure correction. Examples of real-time systems that actively monitor processes can be found in the steelmaking and oil refining industries.

7- **Design and Manufacturing**: These systems assist in the design of physical devices and processes, ranging from high-level conceptual design of abstract entities all the way to factory floor configuration of manufacturing processes.

**Roles in Expert System Development**

Three fundamental roles in building expert systems are:

1. ***Expert*** - Successful ES systems depend on the experience and application of knowledge that the people can bring to it during its development. Large systems generally require multiple experts.

2. ***Knowledge engineer -*** The knowledge engineer has a dual task. This person should be able to elicit knowledge from the expert, gradually gaining an understanding of an area of expertise. Intelligence, tact, empathy, and proficiency in specific techniques of knowledge acquisition are all required of a knowledge engineer. Knowledge-acquisition techniques include conducting interviews with varying degrees of structure, protocol analysis, observation of experts at work, and analysis of cases.

On the other hand, the knowledge engineer must also select a tool appropriate for the project and use it to represent the knowledge with the application of the ***knowledge acquisition facility***.

3. ***User*** - A system developed by an end user with a simple shell, is built rather quickly an inexpensively. Larger systems are built in an organized development effort. A prototype-oriented iterative development strategy is commonly used. ESs lends themselves particularly well to prototyping.

**ADVANTAGES**

- Natural knowledge representation. An expert usually explains the problem-solving procedure with expressions like "In such-and-such situation, in such a situation, I do so-and-so". These expressions can be so". These expressions can be represented quite naturally as IF-THEN production THEN production rules.

- Uniform structure. Production rules have the Production rules have the uniform IF-THEN structure. Each rule is a THEN structure. Each rule is an independent piece of knowledge. The very syntax of production rules enables them to be self of production rules enables them to be self documented. documented.

- Separation of knowledge from its processing Separation of knowledge from its processing. The structure of a rule-based expert system based expert system provides an effective separation of the knowledge and provides an effective separation of the knowledge base from the inference engine. This makes it base from the inference engine. This makes it possible to develop different applications using the possible to develop different applications using the same expert system shell. same expert system shell.

- Dealing with incomplete and uncertain Dealing with incomplete and uncertain knowledge. Most rule-based expert systems are based expert systems capable of representing and reasoning with capable of representing and reasoning with incomplete and uncertain knowledge.

**DISADVANTAGES**:

- Opaque relations between rules. Although the individual production rules are relatively simple and individual production rules are relatively simple and self-documented, their logical interactions within the documented, their logical interactions within the large set of rules may be opaque. Rule-based systems make it difficult to observe how individual systems make it difficult to observe how individual rules serve the overall strategy. rules serve the overall strategy.

- Ineffective search strategy. The inference engine applies an exhaustive search through all the applications and an exhaustive search through all the production rules during each production rules during each cycle. Expert systems cycle. Expert systems with a large set of rules (over 100 rules) can be slow, with a large set of rules (over 100 rules) can be slow, and thus large rule-based systems can be unsuitable based systems can be unsuitable for real for real-time applications. time applications. Disadvantages of rule-based expert systems based expert systems

- Inability to learn. In general, rule-based expert based expert systems do not have an ability to learn from the systems do not have an ability to learn from the experience. Unlike a human expert, who knows experience. Unlike a human expert, who knows when to "break the rules", an expert system cannot when to "break the rules", an expert system cannot automatically modify its knowledge base, or adjust automatically modify its knowledge base, or adjust existing rules or add new ones. The knowledge of existing rules or add new ones. The knowledge engineer is still responsible for revising and engineer is still responsible for revising and maintaining the system. maintaining the system.

**IMPLEMENTATION EXAMPLE**:

**Topic: Rule Based Expert System for a weapons collection**

Modern warfare possesses some typical characteristics such as dynamic and uncertainty with the emergence of many multiplatform weapon systems. However, it is impossible to rely on real combat entirely to verify the feasibility of battlefield operational plans. Consequently, a virtual battlefield environment is highly recommended to be built for military training in order to improve the decision-making capacity of military commanders.

As the weapon systems have become increasingly important in modern warfare, higher requirements are brought forward for military commanders of joint operations to enhance their skills of using weapon systems.

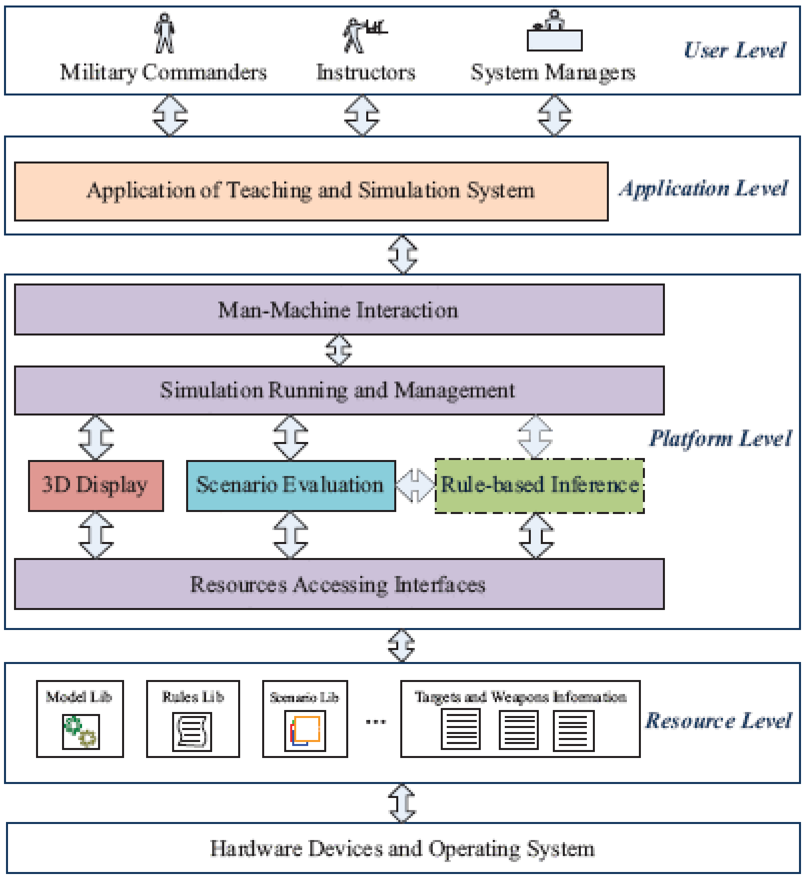
It is noted that the virtual battlefield environment could improve the users' sense of immersion and interactivity greatly, enabling military commanders to master the rules of weapon systems better.

The proposed teaching and simulation system for application of weapon systems world adopt a distributed C/S framework with integrated use of database, network, 3D-GIS and so forth. In addition to general functions such as resource browsing, fuzzy queries and decision support, it would possess the capacity to connect with other systems through the local area *network(LAN)* with the purpose of sharing resources.

**Distributed Architecture of Expert System**

The essential information about targets and weapons were stored in the server computer as well as the knowledge base and models library which were downloaded to the client computer while they were being requested

Military commanders learn how to make good use of weapon systems under instructions from these instructors while system administrators try to manage and maintain the system on the user level. Instructors may set different military backgrounds and arrange distinct courses for those military commanders on the application layer. The platform layer, as the core of the whole system, consists of simulation management module, rule-based reasoning module, 3D-display module and other components. The underlying data including basic information on targets and equipment are provided to support the expert system from the resource layer.



**Implementation of Rule-Based Expert System for Weapon Systems**

The adoption of a rule-based inference engine allows the separation of expert domain knowledge and its control strategy. As a consequence, system administrators have the ability to add, delete and edit expert rules, thereby enhancing the system's scalability and flexibility. Rule-based expert systems(RBES) usually contain three components: rule base, working memory and inference engine. The working memory provides the facts as input parameters for the inference engine, and the other input parameter is the domain knowledge stored in the form of rule components.

The inference engine is made up of three components similarly, namely, pattern matcher, execution engine and the agenda. Pattern matcher decides which rules could meet the demands of input facts, then grants different priorities to these rules and inserts them on the agenda. Execution engine is responsible for the implementation of specific operations according to the rule that has the top priority.

**Representation and Format of Production Rules**

The rules about the application of weapon systems can be extracted into the five following categories.

1. **Target-matching rules**. Military commanders should select the appropriate combat weapons in accordance with the unique characteristics of the enemy targets;
2. **Environment-matching rules**. The conditions of natural environment such as appropriate time, favourable weather, transitable terrain and other restrictions must not be neglectable;
3. **Performance-constraint rules**. The inherent capacity and performances of weapon systems are supposed to be taken into account. For instance, the accuracy of satellite reconnaissance should be considered when the satellite is going to be chosen;
4. **Equipment-combination rules**. Such rules describe static attachment relationships between equipments and their ammunition along with other affiliations;
5. **Evaluation-analysis rules**. The assessment of military commanders' scenarios will be undertaken on the basis of these principles, like the dynamic sequence constraints of the joint use of equipment, for example.

It can be found that these expert rules are mostly the standardizable description of such content like the state of targets, equipment and the environment. The general form of the production rules can be described as follows.

```
IF  x1 AND x2 AND...AND xn   THEN Y1 OR Y2
```

**Conclusion:** We learnt about rule based expert systems and implemented it in an example where we developed a weapons expert system for modern warfare weapons.