

EXPERIMENT NO. 1

Date of Performance :

Date of Submission :

AIM: Discuss suitable Agent Architecture for the problem.

THEORY:

1.1 Specifying Task Environment

In artificial intelligence, an agent is a computer program or system that is designed to perceive its environment, make decisions and take actions to achieve a specific goal or set of goals. The agent operates autonomously, meaning it is not directly controlled by a human operator.

Agents can be classified into different types based on their characteristics, such as whether they are reactive or proactive, whether they have a fixed or dynamic environment, and whether they are single or multi-agent systems.

1.2 Types of Agents

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

- Simple Reflex Agents
- Goal-Based Agents
- Learning Agent
- Multi-agent systems
- Hierarchical agents

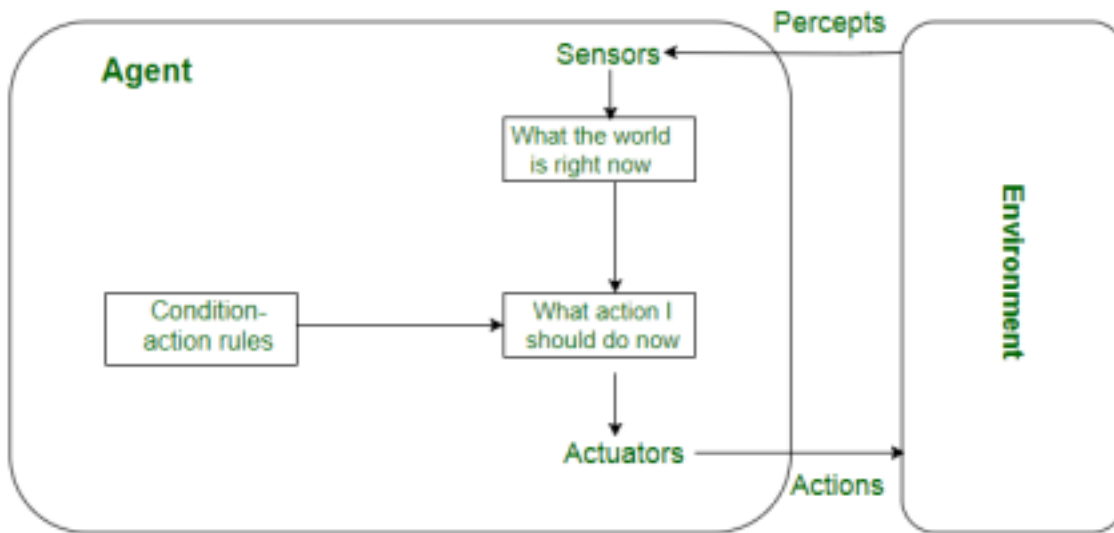
1.2.1 Simple Reflex Agents :

Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept. Percept history is the history of all that an agent has perceived to date. The agent function is based on the condition-action rule. A condition-action rule is a rule that maps a state i.e., a condition to an action. If the condition is true, then the action is taken, else not.

Problems with Simple reflex agents are :

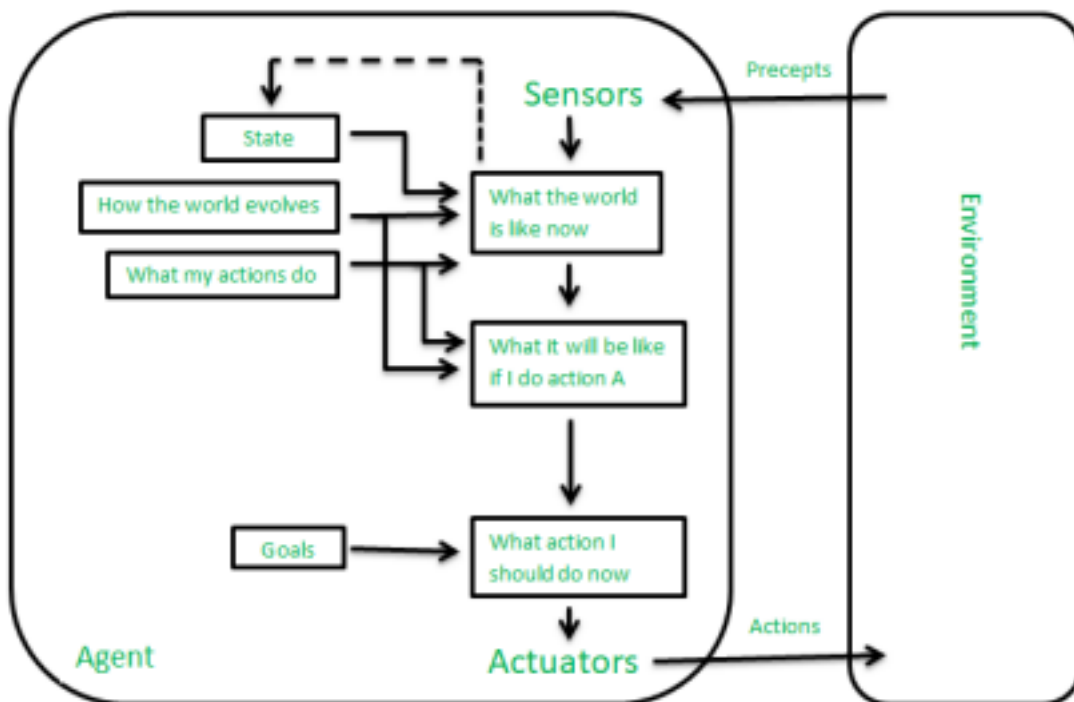
- Very limited intelligence.

- No knowledge of non-perceptual parts of the state.
- Usually too big to generate and store.



1.2.2 Goal-Based Agents :

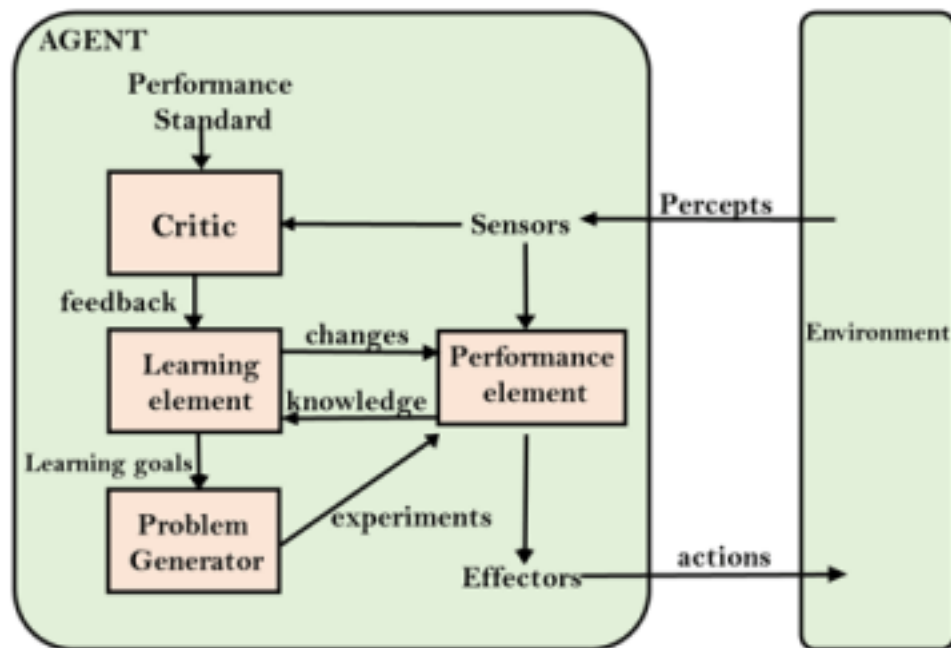
These kinds of agents take decisions based on how far they are currently from their goal (description of desirable situations). Their every action is intended to reduce their distance from the goal. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state.



1.2.3 Learning Agent :

A learning agent in AI is the type of agent that can learn from its past experiences or it has learning capabilities. It starts to act with basic knowledge and then is able to act and adapt automatically through learning. A learning agent has mainly four conceptual components, which are:

- Learning element: It is responsible for making improvements by learning from the environment.
- Critic: The learning element takes feedback from critics which describes how well the agent is doing with respect to a fixed performance standard.
- Performance element: It is responsible for selecting external action.
- Problem Generator: This component is responsible for suggesting actions that will lead to new and informative experiences.



1.2.4 Learning Agent :

These agents interact with other agents to achieve a common goal. They may have to coordinate their actions and communicate with each other to achieve their objective.

A multi-agent system (MAS) is a system composed of multiple interacting agents that are designed to work together to achieve a common goal. These agents may be autonomous or semi-autonomous and are capable of perceiving their environment, making decisions, and taking action to achieve the common objective.

1.2.5 Hierarchical Agents :

These agents are organized into a hierarchy, with high-level agents overseeing the behavior of lower-level agents. The high-level agents provide goals and constraints, while the low-level agents carry out specific tasks. Hierarchical agents are useful in complex environments with many tasks and sub-tasks.

Hierarchical agents are agents that are organized into a hierarchy, with high-level agents overseeing the behavior of lower-level agents. The high-level agents provide goals and constraints, while the low-level agents carry out specific tasks. This structure allows for more efficient and organized decision-making in complex environments.

Hierarchical agents can be implemented in a variety of applications, including robotics, manufacturing, and transportation systems. They are particularly useful in environments where there are many tasks and sub-tasks that need to be coordinated and prioritized.

1.3 Uses Of Agents :

Agents are used in a wide range of applications in artificial intelligence, including:

- Robotics: Agents can be used to control robots and automate tasks in manufacturing, transportation, and other industries.
- Smart homes and buildings: Agents can be used to control heating, lighting, and other systems in smart homes and buildings, optimizing energy use and improving comfort.
- Transportation systems: Agents can be used to manage traffic flow, optimize routes for autonomous vehicles, and improve logistics and supply chain management.
- Healthcare: Agents can be used to monitor patients, provide personalized treatment plans, and optimize healthcare resource allocation.
- Finance: Agents can be used for automated trading, fraud detection, and risk management in the financial industry.

CONCLUSION:We have discussed suitable Agent Architecture for the problem.

SIGN AND REMARK:

DATE:

R1	R2	R3	R4	Total	Signature
(3 Marks)	(4 Marks)	(4 Marks)	(4 Marks)	(15 Marks)	

EXPERIMENT NO. 2

Date of Performance:

Date of Submission:

AIM: Assignments on State space formulation and PEAS representation for various AI applications.

THEORY:

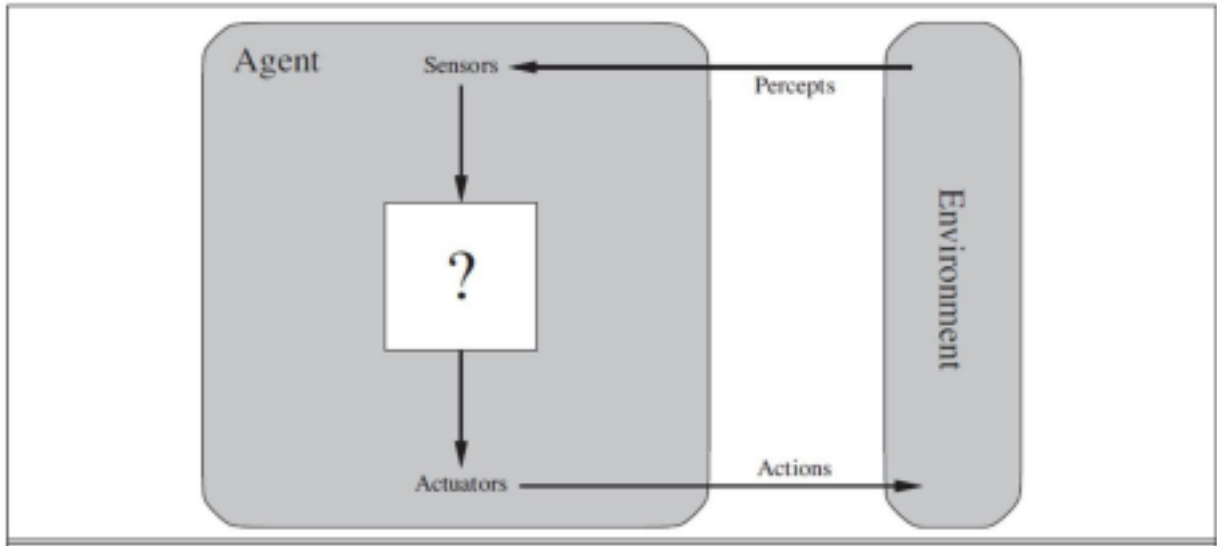
1.1 Specifying Task Environment

Task environments, which are essentially the "problems" to which rational agents are the "solutions." In designing an agent, the first step must always be to specify the task environment as fully as possible with the help of PEAS (Performance, Environment, Actuators, Sensors) description.

1.2 Example of PEAS

Problem Definition:

- PEAS(Performance, Environment, Actuators, Sensors)
- First step to design an agent is to specify task environment as fully as possible.
- While designing a rational agent we have to specify the performance measure, the environment, and agent's actuators and sensors.
- We group them under the heading task environment



PEAS Descriptors:

A goal directed agent needs to achieve certain goals. Such an agent selects its actions based on the goal it has. Many problems can be represented as a set of states and a set of rules of how one state is transformed to another. Each state is an abstract representation of the agent's environment. It is an abstraction that denotes a configuration of the agent.

1.3 Problem Formulation

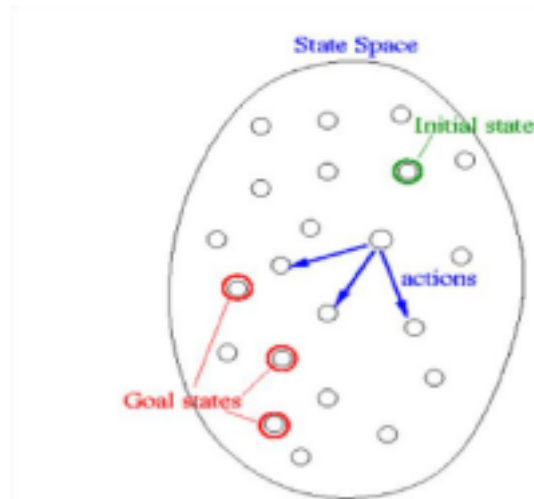
- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states .
- A state is defined by the specification of the values of all attributes of interest in the world
 - An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The initial state is where you start
- The goal state is the partial description of the solution

An initial state is the description of the starting configuration of the agent. An action or an operator takes the agent from one state to another state which is called a successor state. A state can have a number of successor states. A plan is a sequence of actions. The cost of a plan is referred to as the path cost. The path cost is a positive.

Problem formulation means choosing a relevant set of states to consider, and a feasible set of operators for moving from one state to another. Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

Search Problem is formally described as following:

- S : the full set of states
- s_0 : the initial state
- $A:S \rightarrow S$ is a set of operators
- G is the set of final states. Note that $G \subseteq S$



The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state $g \in G$. A sequence of states is called a path. The cost of a path is positive.

1.4 Example of Problem formulation

Problem Formulation:

Agent Type	Performance Measure	Environment	Actuators	Sensors
------------	---------------------	-------------	-----------	---------

Part Picking Robot	Error Recovery, Scalability, Safety, Throughput Rate, Item Sorting, Predictive Maintenance	Hazardous Material Handling, Low Light Conditions, Multi-Level Racks	Gripper Customization, Conveyor Speed Control, Arm Path Planning	3D Object Detection, Environmental Sensors, Shelf Space Utilization, Object Fragility Detection
--------------------	--	--	--	---





Partial Tabulation Chart for Part Picking Robot

Percept Sequence	Action
Camera Detection	Robot Activation
Identifying Location	Planning the optimal path
Detect a fragile item	Adjusting its grip
Identifies the items destination	Plans the optimal path to reach the destination
Detects an obstacle ahead	Robot slows down
Item placed at Destination	Robot releases the item

CONCLUSION: We have successfully implemented State space formulation and PEAS representation for various AI applications.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT NO. 3

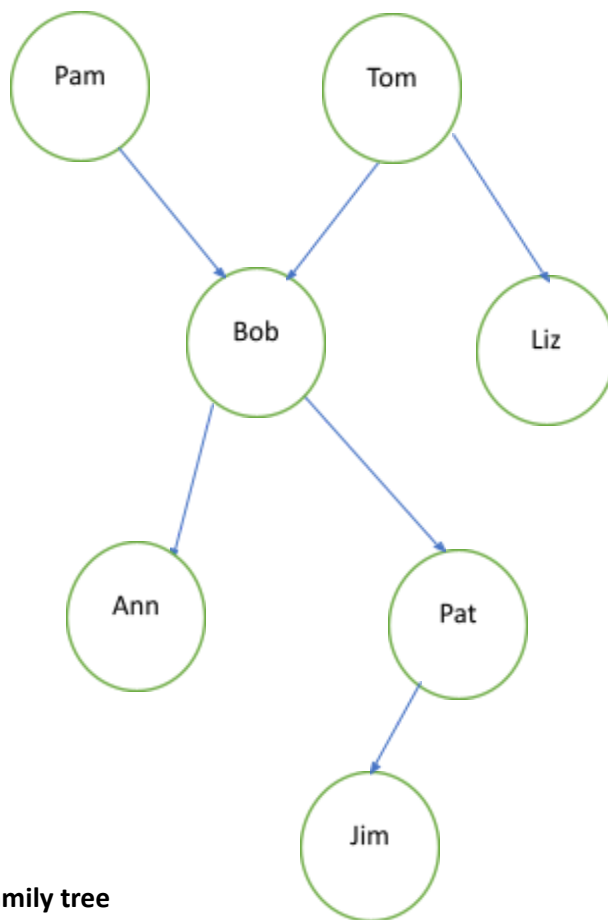
Date of Performance :

Date of Submission :

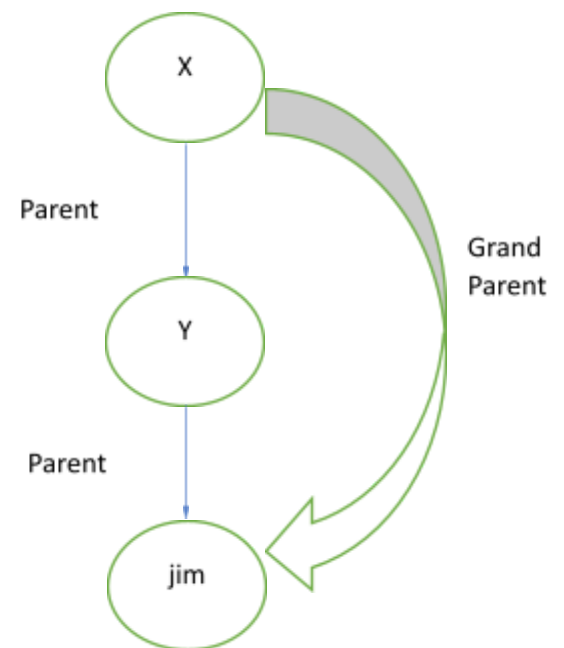
AIM: Exploring Prolog and understanding prolog programs as an AI programming language

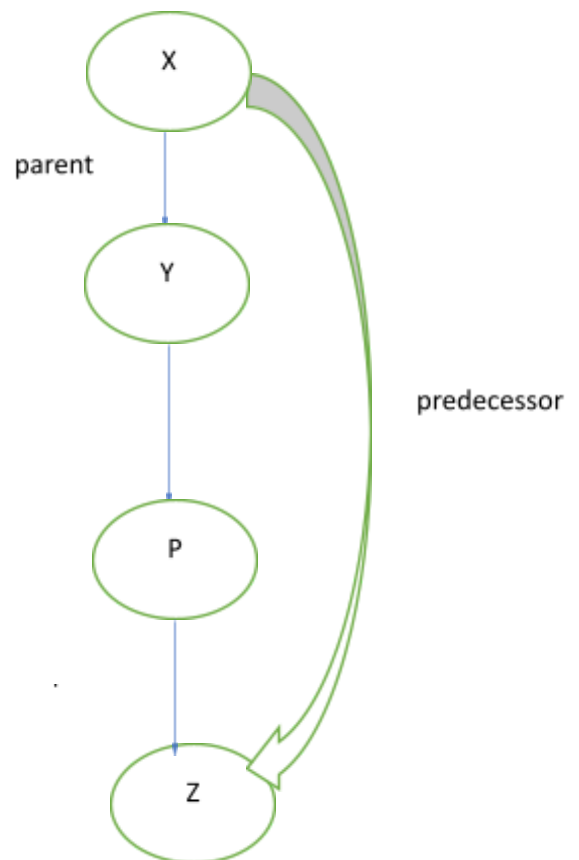
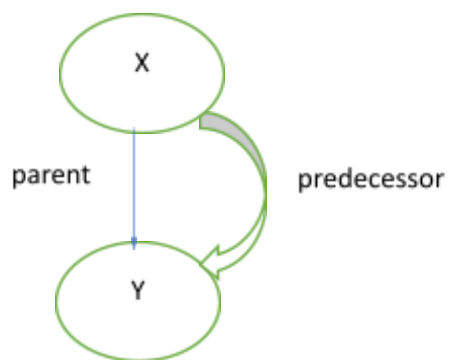
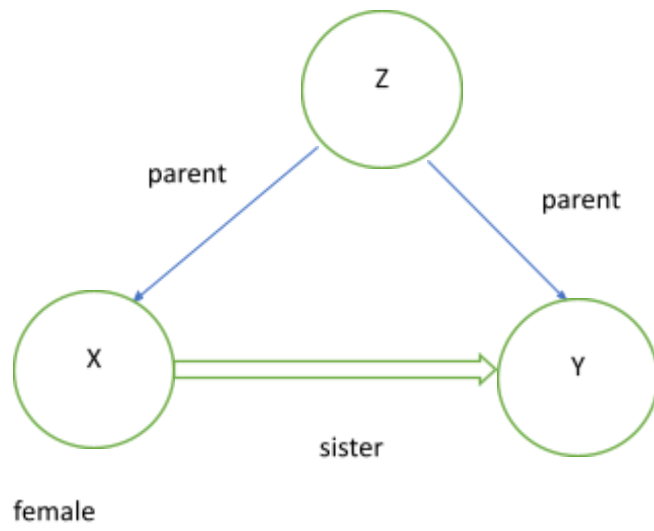
THEORY:

3.1 Understanding Prolog



Family tree





CODE 1:

```
female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
mother(X,Y):- parent(X,Y),female(X).
father(X,Y):- parent(X,Y),male(X).
haschild(X):- parent(X,_).
sister(X,Y):- parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
```

OUTPUT:

```
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021. 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/family1.pl').
compiling D:/prolog_assignments/family1.pl for byte code...
D:/prolog_assignments/family1.pl compiled, 20 lines read - 3081 bytes written, 18 ms

(15 ms) yes
| ?- parent(X,jim).
X = pat ? ;
X = peter
yes
| ?-
mother(X,Y).
X = pam
Y = bob ? ;
X = pat
Y = jim ? ;
no
| ?- haschild(X).
X = pam ? ;
X = tom ? ;
X = tom ? ;
X = bob ? ;
X = bob ? ;
X = pat ? ;
X = bob ? ;
X = peter
```

```

(31 ms) yes
| ?- sister(X,Y).

X = liz
Y = bob ? ;

X = ann
Y = pat ? ;

X = ann
Y = peter ? ;

X = pat
Y = ann ? ;

X = pat
Y = peter ? ;

no
| ?- |

```

CODE 2:

```

female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
mother(X,Y):- parent(X,Y),female(X).
father(X,Y):-parent(X,Y),male(X).
sister(X,Y):-parent(Z,X),parent(Z,Y),female(X),X\==Y.
brother(X,Y):-parent(Z,X),parent(Z,Y),male(X),X\==Y.
grandparent(X,Y):-parent(X,Z),parent(Z,Y).
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
grandfather(X,Z):-father(X,Y),parent(Y,Z).
wife(X,Y):-parent(X,Z),parent(Y,Z),female(X),male(Y).
uncle(X,Z):-brother(X,Y),parent(Y,Z).

```

OUTPUT:

```

compiling D:/prolog_assignments/family2.pl for byte code...
D:/prolog_assignments/family2.pl compiled, 28 lines read - 4632 bytes written, 10 ms

yes
| ?- uncle(X,Y).
X = peter
Y = jim ? ;

no
| ?- grandparent(X,Y).
X = pam
Y = ann ? ;
X = pam
Y = pat ? ;
X = pam
Y = peter ? ;
X = tom
Y = ann ? ;
X = tom
Y = pat ? ;
X = tom
Y = peter ? ;
X = bob
Y = jim ? ;
X = bob
Y = jim ? ;

(47 ms) no
| ?- wife(X,Y).
X = pam
Y = tom ? ;
X = pat
Y = peter ? ;

no
| ?- |

```

CODE 3:

```

female(pam).
female(liz).
female(pat).
female(ann).
male(jim).
male(bob).
male(tom).
male(peter).
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
parent(bob,peter).
parent(peter,jim).
predecessor(X, Z) :- parent(X, Z).
predecessor(X, Z) :- parent(X, Y),predecessor(Y, Z).

```

OUTPUT:

```
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/family3.pl').
compiling D:/prolog_assignments/family3.pl for byte code...
D:/prolog_assignments/family3.pl compiled, 20 lines read - 1837 bytes written, 9 ms

yes
| ?- predecessor(peter,X).

X = jim ? ;

no
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- predecessor(bob,X).
    1 1 Call: predecessor(bob,_23) ?
    2 2 Call: parent(bob,_23) ?
    2 2 Exit: parent(bob,ann) ?
    1 1 Exit: predecessor(bob,ann) ?

X = ann ? ;
    1 1 Redo: predecessor(bob,ann) ?
    2 2 Redo: parent(bob,ann) ?
    2 2 Exit: parent(bob,pat) ?
    1 1 Exit: predecessor(bob,pat) ?

X = pat ? ;
    1 1 Redo: predecessor(bob,pat) ?
    2 2 Redo: parent(bob,pat) ?
    2 2 Exit: parent(bob,peter) ?
    1 1 Exit: predecessor(bob,peter) ?
```

```
X = peter ? ;
    1 1 Redo: predecessor(bob,peter) ?
    2 2 Call: parent(bob,_92) ?
    2 2 Exit: parent(bob,ann) ?
    3 2 Call: predecessor(ann,_23) ?
    4 3 Call: parent(ann,_23) ?
    4 3 Fail: parent(ann,_23) ?
    4 3 Call: parent(ann,_141) ?
    4 3 Fail: parent(ann,_129) ?
    3 2 Fail: predecessor(ann,_23) ?
    2 2 Redo: parent(bob,ann) ?
    2 2 Exit: parent(bob,pat) ?
    3 2 Call: predecessor(pat,_23) ?
    4 3 Call: parent(pat,_23) ?
    4 3 Exit: parent(pat,jim) ?
    3 2 Exit: predecessor(pat,jim) ?
    1 1 Exit: predecessor(bob,jim) ?

X = jim ? ;
    1 1 Redo: predecessor(bob,jim) ?
    3 2 Redo: predecessor(pat,jim) ?
    4 3 Call: parent(pat,_141) ?
    4 3 Exit: parent(pat,jim) ?
    5 3 Call: predecessor(jim,_23) ?
    6 4 Call: parent(jim,_23) ?
    6 4 Fail: parent(jim,_23) ?
    6 4 Call: parent(jim,_190) ?
    6 4 Fail: parent(jim,_178) ?
    5 3 Fail: predecessor(jim,_23) ?
    3 2 Fail: predecessor(pat,_23) ?
    2 2 Redo: parent(bob,pat) ?
    2 2 Exit: parent(bob,peter) ?
    3 2 Call: predecessor(peter,_23) ?
    4 3 Call: parent(peter,_23) ?
    4 3 Exit: parent(peter,jim) ?
    3 2 Exit: predecessor(peter,jim) ?
    1 1 Exit: predecessor(bob,jim) ?
```



```

X = jim ? ;
  1   1 Redo: predecessor(bob,jim) ?
  3   2 Redo: predecessor(peter,jim) ?
  4   3 Call: parent(peter,_141) ?
  4   3 Exit: parent(peter,jim) ?
  5   3 Call: predecessor(jim,_23) ?
  6   4 Call: parent(jim,_23) ?
  6   4 Fail: parent(jim,_23) ?
  6   4 Call: parent(jim,_190) ?
  6   4 Fail: parent(jim,_178) ?
  5   3 Fail: predecessor(jim,_23) ?
  3   2 Fail: predecessor(peter,_23) ?
  1   1 Fail: predecessor(bob,_23) ?

(188 ms) no
{trace}
| ?- |

```

CONCLUSION: SUCCESSFULLY IMPLEMENTED BASIC PROGRAMS AND DEALT WITH VARIOUS OTHER PROBLEMS AS PROLOG IS USEFUL FOR SOLVING FAMILIAR PROBLEMS LIKE MONKEY BANANA PUZZLE AND FLIGHT ROUTES. USING THE RIGHT DATA STRUCTURES MADE THE SOLUTION EVEN BETTER, REVEALING HOW SMART PLANNING AND PROBLEM-SOLVING METHODS MATTER FOR DEALING WITH TOUGH SITUATIONS.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT 4

Date of Performance :

Date of Submission :

AIM: To implement Breadth First Search as Uninformed Search and Depth First Search.

S/W USED : C/C++/Java/Prolog

THEORY:

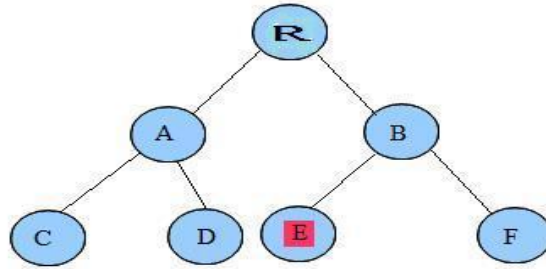
Breadth First Search

Breadth First Search (BFS) searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may be a potential candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
 - a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
 - b. For each way that each rule can match the state described in E do:
 - i) Apply the rule to generate a new state.
 - ii) If the new state is the goal state, quit and return this state.
 - iii) Otherwise add this state to the end of NODE-LIST

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, *breadth-first search* always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d . Since the number of nodes at level d is b^d , the total number of nodes generated in the worst case is $b + b^2 + b^3 + \dots + b^d$ i.e. $O(b^d)$, the asymptotic time complexity of breadth first search.



Breadth First Search

Look at the above tree with nodes starting from root node, R at the first level, A and B at the second level and C, D, E and F at the third level. If we want to search for node E then BFS will search level by level. First it will check if E exists at the root. Then it will check nodes at the second level. Finally it will find E at the third level.

Advantages Of Breadth-First Search

1. Breadth first search will never get trapped exploring the useless path forever except if the no. of successors to any are infinite
2. BFS is **complete** i.e. If there is a solution, BFS will definitely find it out.
3. BFS is **optimal** i.e. If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages Of Breadth-First Search

1. The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is $O(b^d)$. As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
2. If the solution is farther away from the root, breadth first search will consume a lot of time.

Depth First Search:-

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

Algorithm for DFS:-

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

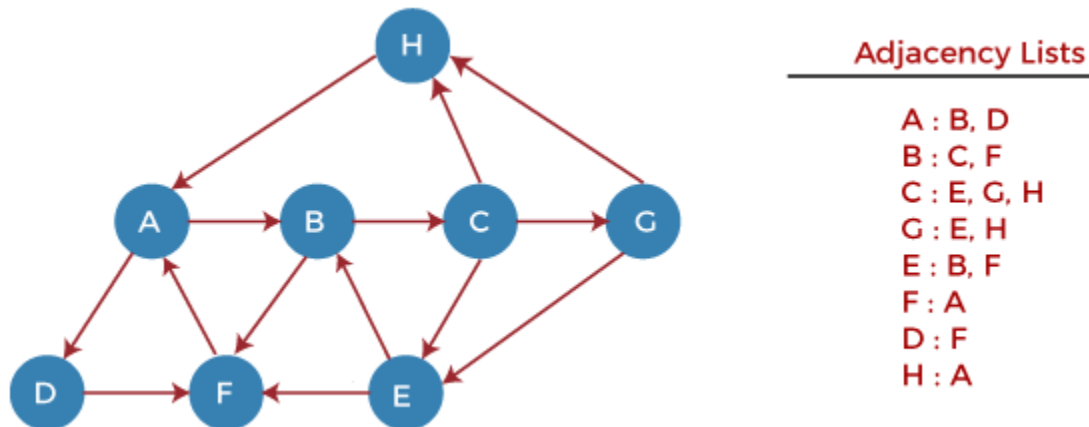
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Advantages of DFS:-

- Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth d and branching factor b (the number of children at each node, the outdegree) is $O(bd)$ since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
- If depth-first search finds a solution without exploring much in a path then the time and space it takes will be very less.
- DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

Disadvantages of Depth First Search:

- The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d , the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d , a large price is paid in execution time, and the first solution found may not be an optimal one.

Depth-First Search is not guaranteed to find the solution. And there is no guarantee to find a minimal solution, if more than one solution.

Code for Bfs:-

```
% Sample graph represented by facts.
```

```
edge(a, b).
```

```
edge(a, c).
```

```
edge(b, d).
```

```
edge(b, e).
```

```
edge(c, f).
```

```
edge(c, g).
```

```
edge(d, h).
```

```
edge(d, i).
```

```
% BFS predicate
```

```
bfs(Start, Goal) :-
```

```
    bfs_queue([[Start]], Goal).
```

```
bfs_queue([[Goal|Path]|_], Goal) :- % Goal found
```

```
    reverse([Goal|Path], Solution),
```

```
    write('Solution: '), write(Solution), nl.
```

```
bfs_queue([Path|Paths], Goal) :-
```

```
    extend(Path, NewPaths),
```

```
    append(Paths, NewPaths, UpdatedPaths),
```

```
    bfs_queue(UpdatedPaths, Goal).
```

```
extend([Node|Path], NewPaths) :-
```

```

findall([NewNode, Node|Path],
        (edge(Node, NewNode), \+ member(NewNode, [Node|Path])),
        NewPaths),
    !. % Cut to prevent backtracking
extend(_, []). % No more extensions

% Sample usage:
% ?- bfs(a, f)

```

Code for Dfs:-

```

% Define a graph represented by facts
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).

% DFS predicate
dfs(Start, End, Path) :-
    dfs_helper(Start, End, [], Path).

% Base case: Reached the destination
dfs_helper(Node, Node, Visited, [Node|Visited]).

% Recursive case: Traverse to adjacent nodes

```

```

dfs_helper(Current, End, Visited, Path) :-
    edge(Current, Next),
    \+ member(Next, Visited), % Avoid revisiting nodes
    dfs_helper(Next, End, [Current|Visited], Path).

% Predicate to start DFS and print the result
dfs_start(Start, End) :-
    dfs(Start, End, Path),
    reverse(Path, PathReversed),
    format("DFS path from ~w to ~w: ~w", [Start, End, PathReversed]).

```

INPUT and OUTPUT:

Breadth-First Search:

```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/bfs.pl').
compiling D:/prolog_assignments/bfs.pl for byte code...
D:/prolog_assignments/bfs.pl compiled, 32 lines read - 2869 bytes written, 6 ms

yes
| ?- bfs(a,f).
Solution: [a,c,f]

true ?

yes
| ?- bfs(a,g).
Solution: [a,c,g]

true ?

yes
| ?- bfs(g,b).

no
| ?-
```

Depth-First Search:

```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/dfs.pl').
compiling D:/prolog_assignments/dfs.pl for byte code...
D:/prolog_assignments/dfs.pl compiled, 24 lines read - 3377 bytes written, 7 ms

yes
| ?- dfs_start(a,f).
DFS path from a to f: [a,c,f]

true ?

yes
| ?- dfs_start(a,g).

no
| ?- dfs_start(b,g).

no
| ?- dfs_start(a,e).
DFS path from a to e: [a,b,e]

true ?

yes
| ?- |
```


CONCLUSION: Successfully Implemented Breadth First Search as Uninformed Search and Depth First Search.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R3 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT 5

Date of Performance :

Date of Submission :

AIM: Developing and Executing an Interactive Human-Computer Interaction System for Solving the 8-Puzzle Problem using Prolog

S/W USED : C/C++/Java/Prolog

THEORY:

This experiment focuses on solving the 8-Puzzle Problem using the Prolog language. By utilizing Prolog's logical reasoning capabilities, we implement Breadth-First Search (BFS) and Depth-First Search (DFS) as uninformed search strategies to navigate the puzzle's state

space. Through a comparative analysis, we investigate the performance of these strategies in terms of solution quality, computational efficiency, and memory usage. This study provides insights into the efficacy of Prolog for solving complex problems and offers a perspective on the effectiveness of different search approaches for the 8-Puzzle Problem.

Algorithm: Min- max, Alpha – Beta Pruning

Breadth-First Search (BFS) Algorithm for 8-Puzzle Problem in Prolog:

Define the initial state of the 8-Puzzle problem.

Create a queue and enqueue the initial state.

Create an empty set to track visited states.

While the queue is not empty:

- a. Dequeue the front state from the queue.
- b. If the current state is the goal state, terminate with success.
- c. Generate all possible valid successor states from the current state.
- d. For each successor state:
 - i. If it hasn't been visited:

- Mark it as visited.
- If the new state is a goal state, terminate with success.
- Enqueue the state onto the queue.

Depth-First Search (DFS) Algorithm for 8-Puzzle Problem in Prolog:

Define the initial state of the 8-Puzzle problem.

Create a stack and push the initial state.

Create an empty set to track visited states.

While the stack is not empty:

- a. Pop the top state from the stack.
- b. If the current state is the goal state, terminate with success.
- c. Generate all possible valid successor states from the current state.
- d. For each successor state:
 - i. If it hasn't been visited:
 - Mark it as visited.
 - If the new state is a goal state, terminate with success.
 - Push the state onto the stack.

In both algorithms, the initial state represents the configuration of the 8-Puzzle, the goal state represents the desired arrangement, and the successor states are generated by moving tiles to empty positions. The BFS and DFS strategies differ in how they manage the queue and stack for state exploration.

In your Prolog implementation, you would need to define predicates for generating valid successor states, checking for the goal state, tracking visited states, and managing the queue or stack accordingly. The specific syntax and structure would depend on the Prolog dialect you are using and your implementation approach.

Advantages of Best-First Search (BFS) for Solving the 8-Puzzle Problem:

1. Completeness: BFS is guaranteed to find a solution if one exists. It explores all possible states level by level, ensuring that the shortest solution is found.
2. Optimal Solution: When applied to the 8-Puzzle Problem, BFS will always find the optimal

solution, i.e., the solution with the fewest number of moves. This is because BFS explores states in order of their distance from the initial state.

3. Exploration of Multiple Solutions: If there are multiple solutions with the same length, BFS will explore and find multiple solutions.

Disadvantages of Best-First Search (BFS) for Solving the 8-Puzzle Problem:

1. Memory Intensive: BFS stores all generated states in memory, which can quickly become impractical for large state spaces like the 8-Puzzle. The memory requirement grows exponentially with the depth of the search.
2. Slower for Deeper Solutions: BFS explores all nodes at a given depth before moving to the next level. If the solution is deep in the search tree, BFS may take a long time to reach it.

Advantages of Depth-First Search (DFS) for Solving the 8-Puzzle Problem:

1. Memory Efficiency: DFS stores fewer states in memory compared to BFS, as it explores deeper into the state space before backtracking. This makes DFS suitable for large state spaces.
2. Faster for Deep Solutions: If the solution is located deep in the search tree, DFS might find it faster compared to BFS since it prioritizes going deep before exploring wider.

Disadvantages of Depth-First Search (DFS) for Solving the 8-Puzzle Problem:

1. Completeness not Guaranteed: DFS might not find a solution even if one exists. It can get stuck in infinite loops or explore non-promising branches early, missing the optimal solution.
2. Non-Optimal Solution: The solution found by DFS might not be optimal. It tends to find the first solution it encounters, which might not be the shortest one.
3. Lack of Multiple Solutions: DFS generally finds a single solution and may not explore multiple solutions of the same length.

PROGRAM:

%problem specific part

%move blank cell right

%S is current state

%Snew is next state

%the same is in left, up, down.

move(S,Snew):-

 right(S,Snew).

%move blank cell right

%first parameter is current state

%Snew is next state

right([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew):-

 R3>0,

 R6>0,

 R9>0,

 blank_right([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew).

%move blank cell right

%first parameter is current state

%Snew is next state

blank_right([R1,R2,R3,R4,R5,R6,R7,R8,R9],S):-
 nth0(N,[R1,R2,R3,R4,R5,R6,R7,R8,R9],0),
 Z is N+1,
 nth0(Z,[R1,R2,R3,R4,R5,R6,R7,R8,R9],R),
 substitute(R,[R1,R2,R3,R4,R5,R6,R7,R8,R9],10,Q),
 substitute(0,Q,R,V),
 substitute(10,V,0,S).

move(S,R4,R5,R6,R7,R8,R9],Snew):-Snew):-
 left(S,Snew).

left([R1,R2,R3,
 R1>0,
 R4>0,
 R7>0,
 blank_left([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew).

blank_left([R1,R2,R3,R4,R5,R6,R7,R8,R9],S):-
 nth0(N,[R1,R2,R3,R4,R5,R6,R7,R8,R9],0),
 Z is N-1,
 nth0(Z,[R1,R2,R3,R4,R5,R6,R7,R8,R9],R),
 substitute(R,[R1,R2,R3,R4,R5,R6,R7,R8,R9],10,Q),
 substitute(0,Q,R,V),
 substitute(10,V,0,S).

move(S,Snew):-
 down(S,Snew).

down([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew):-
 R7>0,
 R8>0,
 R9>0,
 blank_down([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew).

```

blank_down([R1,R2,R3,R4,R5,R6,R7,R8,R9],S):-
    nth0(N,[R1,R2,R3,R4,R5,R6,R7,R8,R9],0),
    Z is N+3,
    nth0(Z,[R1,R2,R3,R4,R5,R6,R7,R8,R9],R),
    substitute(R,[R1,R2,R3,R4,R5,R6,R7,R8,R9],10,Q),
    substitute(0,Q,R,V),
    substitute(10,V,0,S).

```

```

move(S,Snew):-
    up(S,Snew).

```

```

up([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew):-
    R1>0,
    R2>0,
    R3>0,
    blank_up([R1,R2,R3,R4,R5,R6,R7,R8,R9],Snew).

```

```

blank_up([R1,R2,R3,R4,R5,R6,R7,R8,R9],S):-

```

```

% get position of blank cell
    nth0(N,[R1,R2,R3,R4,R5,R6,R7,R8,R9],0),
    Z is N-3,

```

```

% get element in pos Z
    nth0(Z,[R1,R2,R3,R4,R5,R6,R7,R8,R9],R),

```

```

%substitute element of pos Z with blank cell "0"
    substitute(R,[R1,R2,R3,R4,R5,R6,R7,R8,R9],10,Q),
    substitute(0,Q,R,V),
    substitute(10,V,0,S).

```

%substitutes the first parameter with the third parameter and the third parameter with the first parameter in the second parameter (list) and produces a new list (forth parameter).

%e.g. substitute(1, [1,2,3,1,4], 4, X) will make X=[4,2,3,4,1]

%first parameter is value to be substituted by third parameter or replaces it.

%second parameter is the given list to be substituted.

%third parameter is the value that will replace the first parameter or will be substituted by the first parameter

%forth parameter is the new list after substitution.

substitute(_, [], _, []):-!.

substitute(X, [X|T], Y, [Y|T1]):-

substitute(X, T, Y, T1),!.

substitute(X, [Y|T], Y, [X|T1]):-

substitute(X, T, Y, T1),!.

substitute(X, [H|T], Y, [H|T1]):-

substitute(X, T, Y, T1).

% end of specific part

%general algorithm

%query of user and takes start state and next state

go(Start,Goal):-

getHeuristic(Start, H, Goal),

path([[Start,null, 0, H, H]],[],Goal).%open, closed, goal, path_cost, heuristic, total

cost

%main predicate that takes open list, closed list and goal state

path([], _, _):-

write('No solution'),nl,!.

path(Open, Closed, Goal):-

getBestChild(Open, [Goal, Parent, PC, H, TC], RestOfOpen),

write('A solution is found'), nl ,

printsolution([Goal,Parent, PC, H, TC], Closed),!.

path(Open, Closed, Goal):-

getBestChild(Open, [State, Parent, PC, H, TC], RestOfOpen),

getchildren(State, Open, Closed, Children, PC, Goal),

addListToOpen(Children , RestOfOpen, NewOpen),

path(NewOpen, [[State, Parent, PC, H, TC] | Closed], Goal).

%gets Children of State that aren't in Open or Close

getchildren(State, Open ,Closed , Children, PC, Goal):-

bagof(X, moves(State, Open, Closed, X, PC, Goal), Children) .

getchildren(_,_,_, [],_,_).

%adds children to open list (without best child) to form new open list

addListToOpen(Children, [], Children).

addListToOpen(Children, [H|Open], [H|NewOpen]):-

addListToOpen(Children, Open, NewOpen).

%gets the best state of the open list and another list without this best state

%first parameter is the open list

%second parameter is the best child

%third parameter is the open list without the best child

getBestChild([Child], Child, []).

getBestChild(Open, Best, RestOpen):-

 getBestChild1(Open, Best),

 removeFromList(Best, Open, RestOpen).

%gets the best state of the open list

getBestChild1([State], State).

getBestChild1([State|Rest], Best):-

 getBestChild1(Rest, Temp),

 getBest(State, Temp, Best).

%compares two states with each other (according to their Total cost) and returns the state with lower total cost TC

getBest([State, Parent, PC, H, TC], [_, _, _, TC1], [State, Parent, PC, H, TC]):-

 TC < TC1, !.

getBest([_, _, _, _], [State1, Parent1, PC1, H1, TC1], [State1, Parent1, PC1, H1, TC1]).

%removes an element (usually the best state) from a list (open list) and returns a new list

removeFromList(_, [], []).

removeFromList(H, [H|T], V):-

!, removeFromList(H, T, V).

removeFromList(H, [H1|T], [H1|T1]):-

removeFromList(H, T, T1).

%gets next state given the current state

moves(State, Open, Closed,[Next,State, NPC, H, TC], PC, Goal):-

move(State,Next),

\+ member([Next, _, _, _],Open),

\+ member([Next, _, _, _],Closed),

NPC is PC + 1,

getHeuristic(Next, H, Goal),

TC is NPC + H.

%calculate heuristic of some state

%here it is calculated as number of misplaced numbers

getHeuristic([], 0, []):-!.

getHeuristic([H|T1],V,[H|T2]):-!,

```
getHeuristic(T1,V, T2).
```

```
getHeuristic([_|T1],H,[_|T2]):-
```

```
    getHeuristic(T1,TH, T2),
```

```
    H is TH + 1.
```

```
%prints the path from start state to goal state
```

```
printsolution([State, null, PC, H, TC],_):-
```

```
    write(State), write(' PC: '), write(PC), write(' H:'), write(H), write(' TC: '), write(TC),  
    nl.
```

```
printsolution([State, Parent, PC, H, TC], Closed):-
```

```
    member([Parent, GrandParent, PC1, H1, TC1], Closed),
```

```
    printsolution([Parent, GrandParent, PC1, H1, TC1], Closed),
```

```
    write(Parent), write(State), write(' !!PC: '), write(PC), write(' H:'), write(H), write(' TC: '),  
    write(TC), nl.
```

INPUT and OUTPUT:

```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/8puzzlebfs.pl').
compiling D:/prolog_assignments/8puzzlebfs.pl for byte code...
D:/prolog_assignments/8puzzlebfs.pl:117-121: warning: singleton variables [RestOfOpen] for path/3
D:/prolog_assignments/8puzzlebfs.pl compiled, 197 lines read - 29422 bytes written, 16 ms

yes
| ?- go([4,1,3,7,2,5,0,8,6],[1,2,3,4,5,6,7,8,0]).
A solution is found
[4,1,3,7,2,5,0,8,6] PC: 0 H:7 TC: 7
[4,1,3,7,2,5,0,8,6][4,1,3,0,2,5,7,8,6] !!PC: 1 H:6 TC: 7
[4,1,3,0,2,5,7,8,6][0,1,3,4,2,5,7,8,6] !!PC: 2 H:5 TC: 7
[0,1,3,4,2,5,7,8,6][1,0,3,4,2,5,7,8,6] !!PC: 3 H:4 TC: 7
[1,0,3,4,2,5,7,8,6][1,2,3,4,0,5,7,8,6] !!PC: 4 H:3 TC: 7
[1,2,3,4,0,5,7,8,6][1,2,3,4,5,0,7,8,6] !!PC: 5 H:2 TC: 7
[1,2,3,4,5,0,7,8,6][1,2,3,4,5,6,7,8,0] !!PC: 6 H:0 TC: 6

true ?

yes
| ?- |
```

CONCLUSION: We learned to Develop and Execute an Interactive Human-Computer Interaction System for Solving the 8-Puzzle Problem using Prolog successfully.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT 6

Date of Performance:

Date of Submission :

AIM: Designing and Implementing a Tic-Tac-Toe Game using Prolog for Human-Computer Interaction

S/W USED : C/C++/Java/Prolog

THEORY:

Efficient AI Decision-Making in Tic-Tac-Toe using Minimax with Alpha-Beta Pruning

This experiment investigates the utilization of Prolog for creating and engaging in a game of Tic-Tac-Toe. Prolog's logical reasoning abilities offer an intriguing foundation for implementing the game's rules and strategic decision-making. By encoding the game's logic and providing an interactive interface, this experiment explores how Prolog can facilitate the development of AI-driven games, showcasing its potential in simplifying complex decision structures for interactive entertainment.

Algorithm: Min- max, Alpha – Beta Pruning

Create a variable called `NODE_LIST` and set it to the initial state of the Tic-Tac-Toe game. Perform the BFS loop until the goal state (winning state or draw) is found or `NODE_LIST` is empty.

- a. Remove the first element from `NODE_LIST`. If `NODE_LIST` is empty, the search process terminates.
- b. For each possible move that can be made from the current state, do the following:
 - i) Apply the move to generate a new game state.
 - ii) If the new state is a goal state (either a winning state or a draw), the search process terminates, and you've found a solution.
 - iii) Otherwise, add this new state to the end of `NODE_LIST`.

Depth-First Search (DFS) for Tic-Tac-Toe in Prolog:

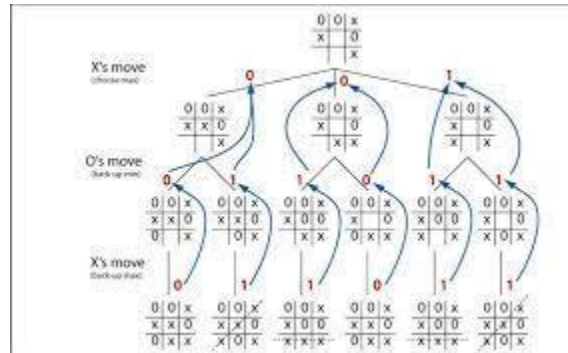
Create a variable called `NODE_LIST` and set it to the initial state of the Tic-Tac-Toe game.

Perform the DFS loop until the goal state (winning state or draw) is found or `NODE_LIST` is empty.

- a. Remove the first element from `NODE_LIST`. If `NODE_LIST` is empty, the search process terminates.
- b. For each possible move that can be made from the current state, do the following:
 - i) Apply the move to generate a new game state.
 - ii) If the new state is a goal state (either a winning state or a draw), the search process terminates, and you've found a solution.
 - iii) Otherwise, add this new state to the beginning of `NODE_LIST`.

It's important to note that in the context of a Tic-Tac-Toe game, the game state would include the current positions of X and O on the board. The "possible moves" would refer to empty cells where the current player can place their symbol. The "goal state" would be when a player wins or when the game ends in a draw.

Remember, the implementation of BFS and DFS in Prolog would involve defining predicates that represent the game state, possible moves, and checking for a goal state. The search loop would involve using recursion and backtracking, which are fundamental concepts in Prolog programming.



Advantages of Alpha-Beta Pruning in Prolog for Tic-Tac-Toe:

1. Significant Reduction in Search Space:

- Alpha-beta pruning helps in cutting off branches of the game tree that do not need to be explored.
- This reduction in search space results in a significant speedup, allowing the algorithm to explore deeper levels of the game tree within a reasonable time frame.

2. Efficiently Identifies Best Moves:

- Alpha-beta pruning allows the algorithm to determine the best possible move by minimizing the number of nodes evaluated.
- This efficiency is particularly important in games like Tic-Tac-Toe, where the search space is manageable, but the algorithm still needs to consider multiple moves ahead.

3. Improves Search Depth:

- By eliminating irrelevant branches early in the search process, alpha-beta pruning enables the algorithm to explore deeper levels of the game tree.
- This improved search depth often results in better gameplay decisions by considering more complex tactics and strategies.

Disadvantages of Alpha-Beta Pruning in Prolog for Tic-Tac-Toe:

1. Complex Implementation:

- Implementing alpha-beta pruning correctly requires careful programming and understanding of the algorithm.

- The logic for maintaining alpha and beta values, along with recursive calls, can be challenging to implement and debug.

2. Influence of Move Ordering:

- The effectiveness of alpha-beta pruning can be influenced by the order in which moves are considered.
- A suboptimal move ordering might lead to fewer pruning opportunities, reducing the overall efficiency of the algorithm.

3. Doesn't Address All Decision Factors:

- While alpha-beta pruning is excellent for minimizing unnecessary evaluations, it doesn't account for all aspects of the game state.
- Some game positions might require considering additional evaluation criteria beyond what alpha-beta pruning provides.

4. Doesn't Replace Heuristic Evaluation:

- Alpha-beta pruning optimizes the search process but still relies on an effective heuristic evaluation function to assess the desirability of game states.
- The quality of the heuristic function can greatly impact the algorithm's decision-making abilities.

In conclusion, alpha-beta pruning offers significant advantages by optimizing the search process and identifying strong moves in games like Tic-Tac-Toe. However, its implementation complexity and dependence on move ordering and heuristic evaluation should be taken into account when designing a game-playing algorithm in Prolog.

PROGRAM:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Prolog TicTacToe alpha-beta expert
%% Design to play against human (Java GUI)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% The empty Tac Tac Toe board
%% Z1 Z2 Z3
%% Z4 Z5 Z6 ~ [Z1,Z2,Z3,Z4,Z5,Z6,Z7,Z8,Z9]
%% Z7 Z8 Z9
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- dynamic board/1.
```

init:-

```
retractall(board(_),
```



```

    assert(board([_Z1,_Z2,_Z3,_Z4,_Z5,_Z6,_Z7,_Z8,_Z9])).
:- init.

%%%%%%%%
%%% Generate possible marks on a free spot on the board.
%%% Use mark(+,-X,-Y) to query/generate possible moves (X,Y).
%%%%%%%%
mark(Player, [X|_],1,1) :- var(X), X=Player.
mark(Player, [_X|_],2,1) :- var(X), X=Player.
mark(Player, [_,_X|_],3,1) :- var(X), X=Player.
mark(Player, [_,_,_X|_],1,2) :- var(X), X=Player.
mark(Player, [_,_,_X|_],2,2) :- var(X), X=Player.
mark(Player, [_,_,_,_X|_],3,2) :- var(X), X=Player.
mark(Player, [_,_,_,_X|_],1,3) :- var(X), X=Player.
mark(Player, [_,_,_,_X|_],2,3) :- var(X), X=Player.
mark(Player, [_,_,_,_X|_],3,3) :- var(X), X=Player.

%%%%%%%%
%%% Move
%%%%%%%%
move(P,(1,1),[X1|R],[P|R]) :- var(X1).
move(P,(2,1),[X1,X2|R],[X1,P|R]) :- var(X2).
move(P,(3,1),[X1,X2,X3|R],[X1,X2,P|R]) :- var(X3).
move(P,(1,2),[X1,X2,X3,X4|R],[X1,X2,X3,P|R]) :- var(X4).
move(P,(2,2),[X1,X2,X3,X4,X5|R],[X1,X2,X3,X4,P|R]) :- var(X5).
move(P,(3,2),[X1,X2,X3,X4,X5,X6|R],[X1,X2,X3,X4,X5,P|R]) :- var(X6).
move(P,(1,3),[X1,X2,X3,X4,X5,X6,X7|R],[X1,X2,X3,X4,X5,X6,P|R]) :- var(X7).
move(P,(2,3),[X1,X2,X3,X4,X5,X6,X7,X8|R],[X1,X2,X3,X4,X5,X6,X7,P|R]) :- var(X8).
move(P,(3,3),[X1,X2,X3,X4,X5,X6,X7,X8,X9|R],[X1,X2,X3,X4,X5,X6,X7,X8,P|R]) :- var(X9).

%%%%%%%%
%%% Record a move: record(+,+,+).
%%%%%%%%
record(Player,X,Y) :-
    retract(board(B)),
    mark(Player,B,X,Y),
    assert(board(B)).

%%%%%%%%
%%% A winning line is ALREADY bound to Player.
%%% win(+Board,+Player) is true or fail.
%%% e.g., win([P,P,P|_],P). is NOT correct, because could bind
%%%%%%%%
win([Z1,Z2,Z3|_],P) :- Z1==P, Z2==P, Z3==P.
win([_,_,Z1,Z2,Z3|_],P) :- Z1==P, Z2==P, Z3==P.
win([_,_,_,Z1,Z2,Z3],P) :- Z1==P, Z2==P, Z3==P.
win([Z1,_,Z2,_,Z3,_,_],P) :- Z1==P, Z2==P, Z3==P.
win([_,Z1,_,Z2,_,Z3,_,_],P) :- Z1==P, Z2==P, Z3==P.
win([_,_,Z1,_,Z2,_,Z3],P) :- Z1==P, Z2==P, Z3==P.
win([Z1,_,_,Z2,_,_,Z3],P) :- Z1==P, Z2==P, Z3==P.
win([_,_,Z1,_,Z2,_,Z3,_,_],P) :- Z1==P, Z2==P, Z3==P.

```

```

%%%%%%%%
%% A line is open if each position is either free or equals the Player
%%%%%%%%
open([Z1,Z2,Z3|_],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) | Z3 ==
Player).
open([_,_,_,Z1,Z2,Z3|_],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) | Z3
== Player).
open([_,_,_,_,Z1,Z2,Z3],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).
open([Z1,_,_,Z2,_,_,Z3,_,_],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).
open([_,Z1,_,_,Z2,_,_,Z3,_,_],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).
open([_,_,Z1,_,_,Z2,_,_,Z3],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).
open([Z1,_,_,_,Z2,_,_,_,Z3],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).
open([_,_,Z1,_,_,Z2,_,_,_,Z3,_,_],Player) :- (var(Z1) | Z1 == Player),(var(Z2) | Z2 == Player), (var(Z3) |
Z3 == Player).

```

```

%%%%%%%%
%% Calculate the value of a position, o maximizes, x minimizes.
%%%%%%%%
value(Board,100) :- win(Board,o), !.
value(Board,-100) :- win(Board,x), !.
value(Board,E) :-
    findall(o,open(Board,o),MAX),
    length(MAX,Emax),    % # lines open to o
    findall(x,open(Board,x),MIN),
    length(MIN,Emin),    % # lines open to x
    E is Emax - Emin.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% using minimax procedure with alpha-beta cutoff.
% Computer (o) searches for best tic tac toe move,
% Human player is x.
% Adapted from L. Sterling and E. Shapiro, The Art of Prolog, MIT Press, 1986.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

:- assert(lookahead(2)).
:- dynamic spy/0. % debug calls to alpha_beta
:- assert(spy). % Comment out stop spy.

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
search(Position,Depth,(Move,Value)) :-

```

```

alpha_beta(o,Depth,Position,-100,100,Move,Value).

alpha_beta(Player,0,Position,_Alpha,_Beta,_NoMove,Value) :-
    value(Position,Value),
    spy(Player,Position,Value).

alpha_beta(Player,D,Position,Alpha,Beta,Move,Value) :-
    D > 0,
    findall((X,Y),mark(Player,Position,X,Y),Moves),
    Alpha1 is -Beta, % max/min
    Beta1 is -Alpha,
    D1 is D-1,
    evaluate_and_choose(Player,Moves,Position,D1,Alpha1,Beta1,nil,(Move,Value)).

evaluate_and_choose(Player,[Move|Moves],Position,D,Alpha,Beta,Record,BestMove) :-
    move(Player,Move,Position,Position1),
    other_player(Player,OtherPlayer),
    alpha_beta(OtherPlayer,D,Position1,Alpha,Beta,_OtherMove,Value),
    Value1 is -Value,
    cutoff(Player,Move,Value1,D,Alpha,Beta,Moves,Position,Record,BestMove).
evaluate_and_choose(_Player,[],_Position,_D,Alpha,_Beta,Move,(Move,Alpha)).

cutoff(_Player,Move,Value,_D,_Alpha,Beta,_Moves,_Position,_Record,(Move,Value)) :-
    Value >= Beta, !.
cutoff(Player,Move,Value,D,Alpha,Beta,Moves,Position,_Record,BestMove) :-
    Alpha < Value, Value < Beta, !,
    evaluate_and_choose(Player,Moves,Position,D,Value,Beta,Move,BestMove).
cutoff(Player,_Move,Value,D,Alpha,Beta,Moves,Position,Record,BestMove) :-
    Value <= Alpha, !,
    evaluate_and_choose(Player,Moves,Position,D,Alpha,Beta,Record,BestMove).

other_player(o,x).
other_player(x,o).

spy(Player,Position,Value) :-
    spy, !,
    write(Player),
    write(' '),
    write(Position),
    write(' '),
    writeln(Value).
spy(_,_,_). % do nothing

%%%%%%%%%%%%%%
%%%%%%%%%% For testing, use h(+,+) to record human move,
%%%%%%%%%% supply coordinates. Then call c (computer plays).
%%%%%%%%%% Use s to show board.
%%%%%%%%%%%%%%
%%%%%%%%%%
h(X,Y) :-

```

```
record(x,X,Y),  
showBoard.
```

```
c :-  
    board(B),  
    alpha_beta(o,2,B,-200,200,(X,Y),_Value),  
    record(o,X,Y),  
    showBoard.
```

```
showBoard :-  
    board([Z1,Z2,Z3,Z4,Z5,Z6,Z7,Z8,Z9]),  
    write('  '),mark(Z1),write(' '),mark(Z2),write(' '),mark(Z3),nl,  
    write('  '),mark(Z4),write(' '),mark(Z5),write(' '),mark(Z6),nl,  
    write('  '),mark(Z7),write(' '),mark(Z8),write(' '),mark(Z9),nl.  
s :- showBoard.
```

```
mark(X) :-  
    var(X),  
    write('#').  
mark(X) :-  
    \+var(X),  
    write(X).
```

INPUT and OUTPUT:

```
SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)
File Edit Settings Run Debug Help
% d:/prolog_assignments/ttt.pl compiled 0.02 sec, 57 clauses
?-
| h(1,2).
  ###
  x ##
  ###
true .

?- c.
o [o,x,_17794,x,_17806,_17812,_17818,_17824,_17830] -1
o [o,_17788,x,x,_17806,_17812,_17818,_17824,_17830] -2
o [o,_17788,_17794,x,x,_17812,_17818,_17824,_17830] -2
o [o,_17788,_17794,x,_17806,x,_17818,_17824,_17830] 0
o [o,_17788,_17794,x,_17806,_17812,x,_17824,_17830] -1
o [o,_17788,_17794,x,_17806,_17812,_17818,x,_17830] -1
o [o,_17788,_17794,x,_17806,_17812,_17818,_17824,x] -2
o [x,o,_17794,x,_17806,_17812,_17818,_17824,_17830] -2
o [x,_17788,o,x,_17806,_17812,_17818,_17824,_17830] -1
o [_17782,x,o,x,_17806,_17812,_17818,_17824,_17830] -1
o [_17782,_17788,o,x,x,_17812,_17818,_17824,_17830] -2
o [x,_17788,_17794,x,o,_17812,_17818,_17824,_17830] 0
o [_17782,x,_17794,x,o,_17812,_17818,_17824,_17830] 0
o [_17782,_17788,x,x,o,_17812,_17818,_17824,_17830] -1
o [_17782,_17788,_17794,x,o,x,_17818,_17824,_17830] 1
o [_17782,_17788,_17794,x,o,_17812,x,_17824,_17830] 0
o [_17782,_17788,_17794,x,o,_17812,_17818,x,_17830] 0
o [_17782,_17788,_17794,x,o,_17812,_17818,_17824,x] -1
o [x,_17788,_17794,x,_17806,o,_17818,_17824,_17830] -2
o [x,_17788,_17794,x,_17806,_17812,o,_17824,_17830] -1
o [x,_17788,_17794,x,_17806,_17812,_17818,o,_17830] -2
o [x,_17788,_17794,x,_17806,_17812,_17818,_17824,o] -1
  ###
  x o #
  ###
```

```
true .

?- c.
o [x,o,x,x,o,_28116,_28122,_28128,_28134] -1
o [x,o,_28098,x,o,x,_28122,_28128,_28134] 0
o [x,o,_28098,x,o,_28116,x,_28128,_28134] -100
o [x,o,_28098,x,o,_28116,_28122,x,_28134] -1
o [x,o,_28098,x,o,_28116,_28122,_28128,x] -1
o [x,x,o,x,o,_28116,_28122,_28128,_28134] 1
o [x,_28092,o,x,o,x,_28122,_28128,_28134] 1
o [x,_28092,o,x,o,_28116,x,_28128,_28134] -100
o [x,x,_28098,x,o,o,_28122,_28128,_28134] 0
o [x,_28092,x,x,o,o,_28122,_28128,_28134] -1
o [x,_28092,_28098,x,o,o,x,_28128,_28134] -100
o [x,x,_28098,x,o,_28116,o,_28128,_28134] 1
o [x,_28092,x,x,o,_28116,o,_28128,_28134] 0
o [x,_28092,_28098,x,o,x,o,_28128,_28134] 1
o [x,_28092,_28098,x,o,_28116,o,x,_28134] 0
o [x,_28092,_28098,x,o,_28116,o,_28128,x] 0
o [x,x,_46,x,o,_64,_70,o,_82] 0
o [x,x,_46,x,o,_64,_70,_76,o] 1
o [x,_40,x,x,o,_64,_70,_76,o] 0
  x ##
  x o #
  o ##
true .

?- h(3,2).
  x ##
  x o x
  o ##
true .
```

```
?- c.
o [x,o,x,x,o,x,o,_6248,_6254] 1
o [x,o,_6218,x,o,x,o,x,_6254] 0
o [x,o,_6218,x,o,x,o,_6248,x] 1
o [x,x,o,x,o,x,o,_6248,_6254] 100
o [x,_6212,o,x,o,x,o,x,_6254] 100
o [x,_6212,o,x,o,x,o,_6248,x] 100
o [x,x,_6218,x,o,x,o,o,_6254] 0
o [x,x,_6218,x,o,x,o,_6248,o] 1
  x # o
  x o x
  o # #
true .
```

CONCLUSION: We designed and Implemented a Tic-Tac-Toe Game using Prolog for Human-Computer Interaction.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R3 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT 7

Date of Performance:

Date of Submission:

AIM: Learning Hill Climbing Algorithm in AI through Practical Implementation in Prolog.

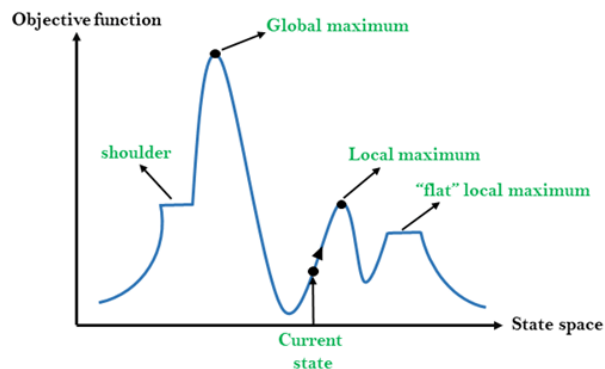
S/W used: C/C++/Java/Prolog.

THEORY:

The Hill Climbing algorithm engages in an ascent through the problem space, aiming to find an optimal solution by iteratively improving the current state. Much like scaling a peak in a landscape, each state symbolizes a potential solution configuration. Hill Climbing begins at an initial state and advances toward better states by making locally optimal choices. It explores neighbouring states, selecting the one with the highest ascent, and continues this process until it reaches a peak, a state where no immediate improvement is possible. Implementing Hill Climbing often involves comparing the evaluation of neighbouring states to identify the best direction to progress.

Algorithm for Hill Climbing

1. Define the initial state of the problem.
2. Evaluate the current state using an objective function to determine its quality.
3. Repeat until no better state can be found or a stopping criterion is met: a. Generate neighbouring states by making small changes to the current state. b. Evaluate the quality of each neighbouring state using the objective function. c. Select the neighbouring state with the highest evaluation (if it's better than the current state). d. If the selected state is better than the current state, update the current state with the selected state.
4. Return the current state as the solution or the best-found solution.



Features of Hill Climbing algorithm:

1. Hill Climbing is a simple and intuitive algorithm that is easy to understand and implement.
2. It can be used in a wide variety of optimization problems, including those with a large search space and complex constraints.
3. Hill Climbing is often very efficient in finding local optima, making it a good choice for problems where a good solution is needed quickly.
4. The algorithm can be easily modified and extended to include additional heuristics or constraints.

Disadvantages of Hill Climbing algorithm:

1. Hill Climbing can get stuck in local optima, meaning that it may not find the global optimum of the problem.
2. The algorithm is sensitive to the choice of initial solution, and a poor initial solution may result in a poor final solution.
3. Hill Climbing does not explore the search space very thoroughly, which can limit its ability to find better solutions.
4. It may be less effective than other optimization algorithms, such as genetic algorithms or simulated annealing, for certain types of problems.

PROGRAM:

% Define the initial state and its evaluation

initial_state(State) :-

 % Define your initial state here

 State = [1, 2, 3, 4].

% Define a predicate to evaluate the current state

evaluate(State, Value) :-

 % Define your evaluation function here

 sum_list(State, Value).

% Define a predicate to generate neighboring states

generate_neighbor(State, Neighbor) :-

 select(X, State, Rest), % Select an element X from the State

 member(Y, [1, 2, 3, 4]), % Generate a new element Y

 X \= Y, % Ensure the new element is different from X

 permutation([Y|Rest], Neighbor). % Generate Neighbor with Y in place of X

% Hill climbing algorithm

hill_climb(State, BestState) :-

 initial_state(CurrentState),

 hill_climb(CurrentState, 0, BestState).

% Recursive hill climbing algorithm with a stopping criterion

hill_climb(State, _, State) :-

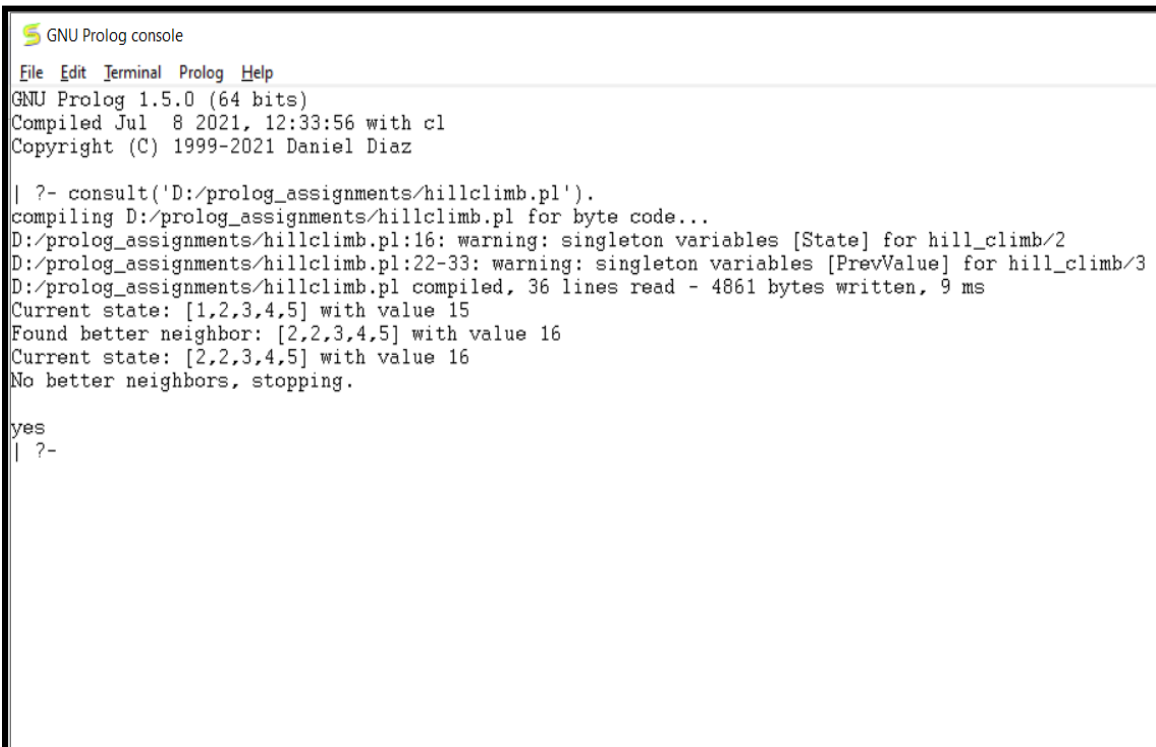
 evaluate(State, Value),


```
\+ generate_neighbor(State, _), % No better neighbors found
write('Optimal solution found: '), write(State), write(' with value '), write(Value), nl.
```

```
hill_climb(State, PrevValue, BestState) :-
    evaluate(State, Value),
    write('Current state: '), write(State), write(' with value '), write(Value), nl,
    generate_neighbor(State, Neighbor),
    evaluate(Neighbor, NeighborValue),
    (NeighborValue > Value ->
        write('Found better neighbor: '), write(Neighbor), write(' with value '), write(NeighborValue), nl,
        hill_climb(Neighbor, Value, BestState)
    );
    write('No better neighbors, stopping. '), nl,
    BestState = State
).
```

OutPut:

INPUT and OUTPUT:

A screenshot of the GNU Prolog console window. The title bar says "GNU Prolog console". The menu bar includes "File", "Edit", "Terminal", "Prolog", and "Help". The console text shows the Prolog version (1.5.0, 64 bits), compilation date (Jul 8 2021, 12:33:56), and copyright (1999-2021 Daniel Diaz). The user enters a query to consult a file 'D:/prolog_assignments/hillclimb.pl'. The system compiles the file and issues warnings about singleton variables [State] and [PrevValue]. The execution output shows the initial state [1,2,3,4,5] with value 15, finding a better neighbor [2,2,3,4,5] with value 16, and then stopping because no better neighbors are found. The user responds with 'yes' and the prompt returns to the Prolog console.

```
GNU Prolog console
File Edit Terminal Prolog Help
GNU Prolog 1.5.0 (64 bits)
Compiled Jul  8 2021, 12:33:56 with cl
Copyright (C) 1999-2021 Daniel Diaz

| ?- consult('D:/prolog_assignments/hillclimb.pl').
compiling D:/prolog_assignments/hillclimb.pl for byte code...
D:/prolog_assignments/hillclimb.pl:16: warning: singleton variables [State] for hill_climb/2
D:/prolog_assignments/hillclimb.pl:22-33: warning: singleton variables [PrevValue] for hill_climb/3
D:/prolog_assignments/hillclimb.pl compiled, 36 lines read - 4861 bytes written, 9 ms
Current state: [1,2,3,4,5] with value 15
Found better neighbor: [2,2,3,4,5] with value 16
Current state: [2,2,3,4,5] with value 16
No better neighbors, stopping.

yes
| ?-
```

CONCLUSION: Learned Hill Climbing Algorithm in AI through Practical Implementation in Prolog.

DATE:

SIGN AND REMARK:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R3 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT 8

Date of

Performance:

Date of Submission:

AIM: Prove goal sentences from following set statements in FOPL by applying forward, backward.

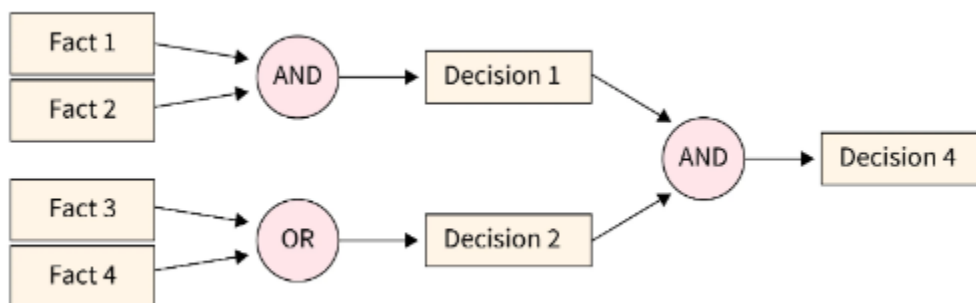
S/W used: C/C++/Java/Prolog.

THEORY:

Forward Chaining in AI using FOL Proof

When utilizing an inference engine, forward chaining is indeed referred to as forward deduction or forward reasoning. Forward chaining is a type of reasoning in which atomic clauses in a knowledge base are used to retrieve more data in the forward path using inference principles (Modus Ponens).

Before deriving new information, the inference engine evaluates current data, derivations, and conditions in this sort of chaining. The manipulation of information in the knowledge base results in the achievement of an objective (goal).



Forward chaining may be employed for application planning, monitoring, management, and interpretation.

ii. Properties of forward chaining

- It follows a down-up strategy, going from bottom to top.
- It is the process of reaching a decision based on known facts or information, beginning with the initial state and progressing to the target state.
- The forward-chaining method is also known as data-driven because we achieve our objective by employing available data.
- The forward-chaining method is widely used in expert systems such as CLIPS, business rule systems, and manufacturing rule systems.

Before delving further into the chaining illustration, let us first understand the preprocessing that must be performed on the provided sentences to implement forward chaining, and that preprocessing is as follows:

1. Change the clauses to First Order Predicate Logic.
2. Distinguish truths from generated clauses.
3. Make note of what we need to establish at the end.

iii. Examples of forward chaining

Some Fact:

1. It is a crime for Indians to give weapons to enemies of India.
2. Country XYZ is an enemy of India
3. XYZ has some Nuclear warheads
4. All Nuclear warheads were sold to XYZ by Rajiv
5. Rajiv is Indian
6. A nuclear warhead is a weapon

Goal:

Rajiv is criminal

iv. Facts Conversion into FOL

Facts Conversion into First-Order Logic

Step 1: It is a crime for Indians to give weapons to enemies of India.

FOL: $\text{Indian}(P) \wedge \text{Weapon}(Q) \wedge \text{Enemy}(R) \wedge \text{Sells}(P, Q, R) \rightarrow \text{Criminal}(P)$

Step 2: Country XYZ is an enemy of India

FOL: $\text{Enemy}(\text{XYZ}, \text{India})$

Step 3: XYZ has some Nuclear warheads

FOL: $\text{Owns}(\text{XYZ}, x) \wedge \text{Nuclear warheads}(x)$

Step 4: All Nuclear warheads were sold to XYZ by Rajiv

FOL: $\text{Nuclear warheads}(x) \wedge \text{Owns}(\text{XYZ}, x) \rightarrow \text{Sell}(\text{Rajiv}, x, \text{XYZ})$

Step 5: Rajiv is Indian

FOL: $\text{Indian}(\text{Rajiv})$

Step 6: A nuclear warhead is a weapon

FOL: $\text{Nuclear warhead}(x) \rightarrow \text{Weapon}(x)$

v. Forward chaining proof

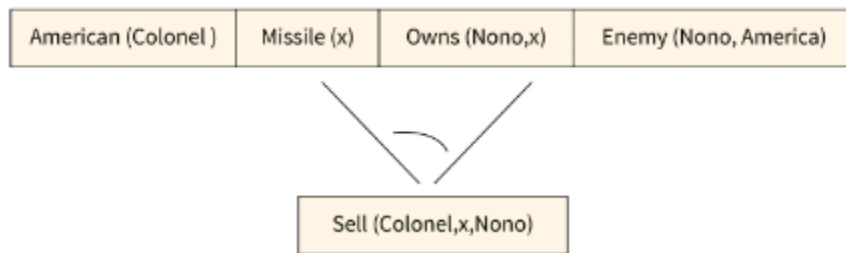
Iteration 1:

1. Begin with the knowledge base and well-known facts: The facts that cannot be inferred from any other (represented in the LHS of the clauses)

American (Colonel)	Missile (x)	Owns (Nono,x)	Enemy (Nono, America)
---------------------	-------------	---------------	-----------------------

2. To join the connections, add inferences one at a time.
 - As Rule (1) does not fulfill the, it is unlikely to be incorporated.

- Rules (2) and (3) FOL have already been incorporated.
- Rule (4) corresponds to the conclusion, so we will include it.



3.

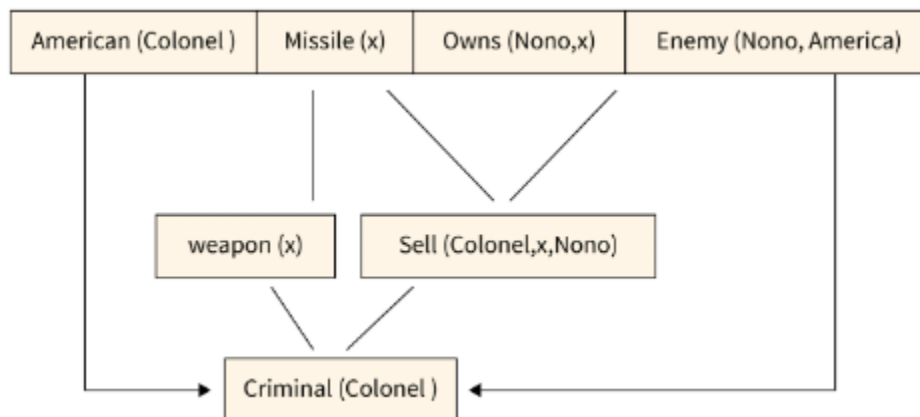
- Rule(5) lacks RHS, therefore nothing would be incorporated.
- Rule (6) has a weapon in RHS that is taken from Nuclear Warhead(x), which has already been a component, so we will include a weapon in the following state.

4.

This wraps our first iteration

Iteration 2:

As we can see from the first iteration, Rule(1) meets all of the LHS requirements. So now that we have all four FOL in LHS, we can put Criminal(x) from RHS into the following stage.



We concluded that Rajiv is a criminal.

Hence proved!

Simple Example:-

```
% Define some facts
```

```
bird(sparrow).
```

```
bird(eagle).
```

```
mammal(cat).
```

```
mammal(dog).
```

```
% Define rules
```

```
can_fly(X) :- bird(X).
```

```
has_fur(X) :- mammal(X).
```

```
% Query to perform forward chaining
```

```
?- can_fly(X).
```

In this example, we have defined facts about birds and mammals. We also defined two rules: `can_fly(X) :- bird(X)` and `has_fur(X) :- mammal(X)`. When you query `can_fly(X)`., Prolog will automatically use forward chaining to infer that sparrow and eagle can fly.

So, in Prolog, you don't explicitly execute a forward chaining command; instead, you define your knowledge base with facts and rules and let Prolog's inference engine perform forward chaining when you make queries.

vi. Advantages

- It may be employed to reach several inferences.
- It serves as a solid foundation for drawing inferences.
- It is more adaptable than backward chaining since the data generated from it is not limited.

vii. Disadvantages

- Forward chaining can be a time-consuming procedure. Eliminating and synchronizing available info may take a long period.

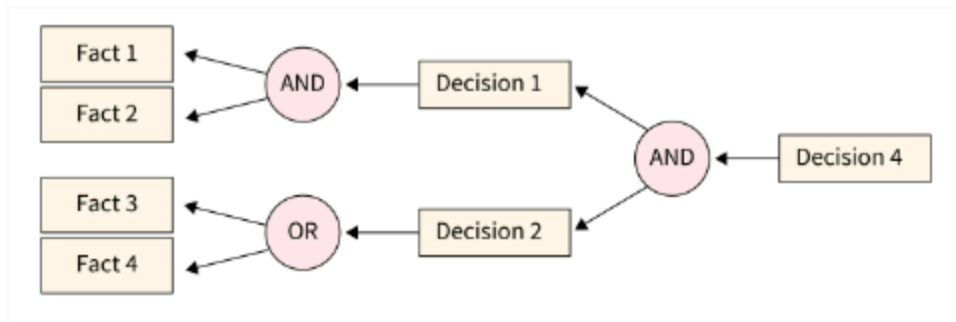
The interpretation of facts or findings for this form of chaining is not as obvious as it is for backward chaining. The former employs a goal-driven approach that yields quick results.

Backward chaining

i. What is backward chaining?

Backward Chaining, also known as backward reasoning, is an inference engine reasoning method that begins with an imagined objective. It employs backtracking to discover the most optimum method for dispute resolution or to achieve the target state, in which the search begins from the outcome and travels back to comprehend the circumstances from which it came.

Inference engines use this reasoning method to discover the circumstances and rules that led to a logical outcome or conclusion.



ii. Properties of backward chaining

The following are the main traits that distinguish backward chaining from forward chaining:

- It is referred to as a top-down strategy.
- Backward chaining is founded on the law of modus ponens reasoning.
- Backward chaining divides the objective into sub-goals to demonstrate the truth of the facts.
- A goal-driven strategy is used because a list of objectives determines which rules are chosen and used.
- The backward-chaining method is used in game theory, automatic theorem-proving techniques, inference engines, prove assistants and other artificial intelligence uses.
- For evidence, the backward-chaining technique mostly employed a depth-first search strategy.

Before going any further with the chaining example, let's first comprehend the algorithm that needs to be applied to the given phrases to apply backward chaining:

1. First, the system checks the information base to see if the objective has been stated.
2. If not, it scans the knowledge base for rules until it finds one with the objective in its THEN portion in the shape of premises.

3. Once the hypotheses are specified and recognized, the system checks to see if they are in the knowledge base; if not, a new objective is set. This new objective is known as a sub-goal to be proven.
4. The system repeats the above stages until it discovers the premise that's not given by any regulation. It is referred to as a primitive premise.
5. Finally, when the system locates the primitive, it prompts the user for additional data, which is employed to establish both the sub-goal and the initial goal.

iii. Examples of backward chaining

Some Facts:

1. Ravi enjoys a wide variety of foods.
2. Banana are food.
3. Pizza is food.
4. A food is anything that anyone consumes and isn't harmed by.
5. Sam eats Idli and is still alive.
6. Bill eats everything Sam eats.

Goal:

Ravi Like Idli

Conversion into FOL:

The first sentence can be expressed as follows: $\forall x \text{ Food } (x) \rightarrow \text{Likes } (\text{Ravi}, x)$

The second sentence can be expressed as follows:: $\text{Food } (\text{Banana})$

The third sentence can be expressed as follows:: $\text{Food } (\text{Pizza})$

The fourth sentence can be expressed as follows:: $\forall x \forall y \text{ Eats}(x, y) \wedge \neg \text{Harmed}(y) \rightarrow \text{Food}(x)$

The fifth sentence can be expressed as follows:: $\text{Eats } (\text{Idli}, \text{Sam}) \rightarrow \neg \text{Harmed } (\text{Sam})$

The sixth sentence can be expressed as follows:: $\forall x \text{ Eats } (x, \text{Sam}) \rightarrow \text{Eats } (x, \text{Bill})$

iv. Backward-Chaining proof

Iteration 1:

Separation of Facts:

- Food (Banana)
- Food (Pizza)
- Eats (Idli, Sam) $\rightarrow \neg$ Harmed (Sam)

Iteration 2:

1. We must regard the statement we must establish to be true, i.e. Likes (Ravi, Idli) are now true.
2. We can deduce from the first rule that Food (Idli) is true.
3. We can deduce from the fourth rule that Eats (Idli, Ravi) $\rightarrow \neg$ Harmed (Ravi) is true.

We demonstrated that the assertion we wished to prove is legitimate because we ended up with all of the statements being true.

Hence Proved!

Simple example:-

% Define some facts

parent(john, mary).

parent(john, lisa).

parent(mary, ann).

parent(lisa, pat).

```
% Define rules
```

```
ancestor(X, Y) :- parent(X, Y).
```

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

```
% Query using backward chaining
```

```
?- ancestor(john, pat).
```

In this example, we have a knowledge base that represents parent-child relationships. We also have two rules: `ancestor/2` that defines the relationship between ancestors and descendants. When you query `ancestor(john, pat).`, Prolog will use backward chaining to satisfy this goal. It will find that john is the parent of lisa, and lisa is the parent of pat, which satisfies the goal, so it will return true. If you query `ancestor(john, ann).`, Prolog will recursively apply the rule to find that john is the parent of mary, and mary is the parent of ann, and it will return true. In Prolog, you can use various built-in predicates like `call/1`, `assert/1`, and `retract/1` to control the flow of backward chaining and manipulate the knowledge base during querying. Backward chaining allows you to find solutions to specific goals or queries based on the available facts and rules in the knowledge base.

v. Advantages

Backward Chaining, like Forward Chaining, has several advantages, a few of which are listed below:

- The problem typically begins by developing a hypothesis and afterward determining whether or not it can be proven.
- Because it is concentrated on a specific objective, it only employs parts of the knowledge base that are pertinent to the hypothesis.
- It is appropriate for issues requiring an expert system for diagnosis, prescribing,

and troubleshooting.

- Backward chaining is a faster procedure because it only analyzes and verifies the necessary rules.

vi. Disadvantages

Backward Chaining is an immensely helpful reasoning method that conducts task analysis swiftly and effectively assists the system in reaching the target state. However, this method has a few disadvantages, including:

- It may pursue a specific line of reasoning and inquiry even if it demonstrates to be impractical in the end.
- Backward chaining can be difficult to execute at times.

Differences Between Forward and Backward Chaining

The following is the distinction between forward and backward chaining:

- As the name implies, forward chaining begins with known facts and moves forward by implementing logical rules to extract more information, and it keeps going until it reaches the objective, whereas backward chaining begins with the goal and moves backward by implementing logical deduction rules to evaluate the information that satisfies the objective.
- Backward chaining is a goal-driven inference method, whereas forward chaining is a data-driven inference technique.
- Forward chaining is referred to as the bottom-up method, whereas backward chaining is referred to as the top-down approach.
- Backward chaining employs a depth-first search strategy, whereas forward chaining employs a breadth-first search strategy.
- Both forward and reverse chaining use the Modus ponens reasoning rule.
- Forward chaining is useful for tasks like planning, design process tracking, diagnostic, and categorization, whereas backward chaining is useful for tasks like

categorization and diagnosis.

- Forward-chaining may include multiple ASK questions from the information source, whereas backward chaining may include fewer ASK questions.
- Forward chaining is slow because it tests all of the rules, whereas backward chaining is quick because it only examines the rules that are needed.
- In artificial intelligence, backward and forward chaining are essential techniques of logical thinking. The primary distinctions between these ideas are methodology, strategy, technique, speed, and tactical direction.
- Forward chaining is essential for coders who want to create effective computer-based systems using data-driven methods. Backward chaining is essential for coders who want to use goal-driven algorithms to create effective solutions in complicated database systems.

Conclusion: Goal sentences from following set statements in FOPL by applying forward, backward proved.

DATE:

SIGN AND REMARK:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT NO. 9

Date of Performance :

Date of Submission :

AIM: Create a Bayesian Network for given problem statement and draw inferences from it.(You can belief and decision networks tool for modeling Bayesian Networks)

THEORY:

Bayesian Belief Network in artificial intelligence. Bayesian belief network is key computer technology

for dealing with probabilistic events and to solve a problem which has uncertainty. We can define a

Bayesian network as:

"A Bayesian network is a probabilistic graphical model which represents a set of variables and their

conditional dependencies using a directed acyclic graph."

It is also called a Bayes network, belief network, decision network, or Bayesian model. Bayesian

networks are probabilistic, because these networks are built from a probability distribution, and also

use probability theory for prediction and anomaly detection.

Real world applications are probabilistic in nature, and to represent the relationship between multiple

events, we need a Bayesian network. It can also be used in various tasks including prediction,

anomaly detection, diagnostics, automated insight, reasoning, time series prediction, and decision making under uncertainty.

Bayesian Network can be used for building models from data and experts opinions, and it consists of

two parts: Directed Acyclic Graph, Table of conditional probabilities.

The generalized form of Bayesian network that represents and solve decision problems under uncertain

knowledge is known as an Influence diagram. A Bayesian network graph is made up of nodes and

Arcs (directed links), where:

- o Each node corresponds to the random variables, and a variable can be continuous or discrete.

- o Arc or directed arrows represent the causal relationship or conditional probabilities between

random variables. These directed links or arrows connect the pair of nodes in the graph.

These links represent that one node directly influence the other node, and if there is no directed

link that means that nodes are independent with each other

- o In the above diagram, A, B, C, and D are random variables represented by the nodes of the network graph.

- o If we are considering node B, which is connected with node A by a directed arrow, then node A is called the parent of Node B.

- o Node C is independent of node A.

The Bayesian network has mainly two components: Causal Component and Actual numbers

Each node in the Bayesian network has condition probability distribution $P(X_i | \text{Parent}(X_i))$, which

determines the effect of the parent on that node. Bayesian network is based on Joint probability

distribution and conditional probability. So let's first understand the joint probability distribution:

Joint probability distribution:

If we have variables $x_1, x_2, x_3, \dots, x_n$, then the probabilities of a different combination of x_1, x_2, x_3, \dots

x_n , are known as Joint probability distribution.

$P[x_1, x_2, x_3, \dots, x_n]$, it can be written as the following way in terms of the joint probability distribution.

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2, x_3, \dots, x_n]$$

$$= P[x_1 | x_2, x_3, \dots, x_n] P[x_2 | x_3, \dots, x_n] \dots P[x_{n-1} | x_n] P[x_n].$$

In general for each variable X_i , we can write the equation as:

$$P(X_i | X_{i-1}, \dots, X_1) = P(X_i | \text{Parents}(X_i))$$

Explanation of Bayesian network:

Let's understand the Bayesian network through an example by creating a directed acyclic graph:

Example: Harry installed a new burglar alarm at his home to detect burglary. The alarm reliably

responds at detecting a burglary but also responds for minor earthquakes. Harry has two neighbors

David and Sophia, who have taken a responsibility to inform Harry at work when they hear the alarm.

David always calls Harry when he hears the alarm, but sometimes he got confused with the phone

ringing and calls at that time too. On the other hand, Sophia likes to listen to high music, so sometimes

she misses to hear the alarm. Here we would like to compute the probability of Burglary Alarm.

Problem:

Calculate the probability that alarm has sounded, but there is neither a burglary, nor an earthquake occurred, and David and Sophia both called the Harry.

Solution:

o The Bayesian network for the above problem is given below. The network structure is showing

that burglary and earthquake is the parent node of the alarm and directly affecting the probability of alarm's going off, but David and Sophia's calls depend on alarm probability.

o The network is representing that our assumptions do not directly perceive the burglary and also

do not notice the minor earthquake, and they also not confer before calling.

o The conditional distributions for each node are given as conditional probabilities table or CPT.

o Each row in the CPT must be sum to 1 because all the entries in the table represent an exhaustive set of cases for the variable.

o In CPT, a boolean variable with k boolean parents contains 2^K probabilities. Hence, if there are

two parents, then CPT will contain 4 probability values

List of all events occurring in this network:

o Burglary (B)

o Earthquake(E)

o Alarm(A)

o David Calls(D)

o Sophia calls(S)

We can write the events of problem statement in the form of probability: $P[D, S, A, B, E]$, can rewrite

the above probability statement using joint probability distribution:

$$P[D, S, A, B, E] = P[D | S, A, B, E] \cdot P[S, A, B, E]$$

$$= P[D | S, A, B, E] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$= P[D | A] \cdot P[S | A, B, E] \cdot P[A, B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B, E]$$

$$= P[D | A] \cdot P[S | A] \cdot P[A | B, E] \cdot P[B | E] \cdot P[E]$$

Let's take the observed probability for the Burglary and earthquake component:

$P(B = \text{True}) = 0.002$, which is the probability of burglary.

$P(B = \text{False}) = 0.998$, which is the probability of no burglary.

$P(E = \text{True}) = 0.001$, which is the probability of a minor earthquake

$P(E = \text{False}) = 0.999$, Which is the probability that an earthquake not occurred.

We can provide the conditional probabilities as per the below tables:

Conditional probability table for Alarm A:

The Conditional probability of Alarm A depends on Burglar and earthquake:

B E $P(A = \text{True})$ $P(A = \text{False})$

True True 0.94 0.06

True False 0.95 0.04

False True 0.31 0.69

False False 0.001 0.999

Conditional probability table for David Calls:

The Conditional probability of David that he will call depends on the probability of Alarm.

A P(D= True) P(D= False)

True 0.91 0.09

False 0.05 0.95

Conditional probability table for Sophia Calls:

**The Conditional probability of Sophia that she calls is depending on its Parent Node
"Alarm."**

A P(S= True) P(S= False)

True 0.75 0.25

False 0.02 0.98

4

**From the formula of joint distribution, we can write the problem statement in the form of
probability**

distribution:

$$P(S, D, A, \neg B, \neg E) = P(S|A) * P(D|A) * P(A|\neg B \wedge \neg E) * P(\neg B) * P(\neg E).$$

$$= 0.75 * 0.91 * 0.001 * 0.998 * 0.999$$

$$= 0.00068045.$$

**Hence, a Bayesian network can answer any query about the domain by using Joint
distribution.**

Program:

```
!pip install networkx

!pip install pybbn

import pandas as pd # for data manipulation

import networkx as nx # for drawing graphs

import matplotlib.pyplot as plt # for drawing graphs

# for creating Bayesian Belief Networks (BBN)

from pybbn.graph.dag import Bbn

from pybbn.graph.edge import Edge, EdgeType

from pybbn.graph.jointree import EvidenceBuilder

from pybbn.graph.node import BbnNode

from pybbn.graph.variable import Variable

from pybbn.pptc.inferencecontroller import InferenceController

# Set Pandas options to display more columns

pd.options.display.max_columns=50

# Read in the weather data csv

df=pd.read_csv('weatherAUS.csv', encoding='utf-8')

# Drop records where target RainTomorrow=NaN

df=df[pd.isnull(df['RainTomorrow'])==False]

# For other columns with missing values, fill them in with column mean

df=df.fillna(df.mean())

# Create bands for variables that we want to use in the model
```

```
df['WindGustSpeedCat']=df['WindGustSpeed'].apply(lambda x: '0.<=40' if x<=40 else
'1.40-50' if 40<x<=50 else '2.>50')
```

```
df['Humidity9amCat']=df['Humidity9am'].apply(lambda x: '1.>60' if x>60 else '0.<=60')
```

```
df['Humidity3pmCat']=df['Humidity3pm'].apply(lambda x: '1.>60' if x>60 else '0.<=60')
```

```
# Show a snaphsot of data
```

```
df
```

```
<ipython-input-4-344dd601463a>:11: FutureWarning: The default value of numeric_only
```

```
df=df.fillna(df.mean())
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGus
0	2008-12-01	Albury	13.4	22.9	0.6	5.469824	7.624853	
1	2008-12-02	Albury	7.4	25.1	0.0	5.469824	7.624853	\
2	2008-12-03	Albury	12.9	25.7	0.0	5.469824	7.624853	\

```
# Create nodes by manually typing in probabilities
```

```
H9am = BbnNode(Variable(0, 'H9am', ['<=60', '>60']), [0.30658, 0.69342])
H3pm = BbnNode(Variable(1, 'H3pm', ['<=60', '>60']), [0.92827, 0.07173,
0.55760, 0.44240])
W = BbnNode(Variable(2, 'W', ['<=40', '40-50', '>50']), [0.58660, 0.24040, 0.17300])
RT = BbnNode(Variable(3, 'RT', ['No', 'Yes']), [0.92314, 0.07686,
0.89072, 0.10928,
0.76008, 0.23992,
0.64250, 0.35750,
0.49168, 0.50832,
0.32182, 0.67818])
```

```
# This function helps to calculate probability distribution, which goes into BBN (note, can handle up to 2
parents)
```

```
def probs(data, child, parent1=None, parent2=None):
```

```

if parent1==None:

    # Calculate probabilities

    prob=pd.crosstab(data[child], 'Empty', margins=False,
normalize='columns').sort_index().to_numpy().reshape(-1).tolist()

elif parent1!=None:

    # Check if child node has 1 parent or 2 parents

    if parent2==None:

        # Caclucate probabilities

        prob=pd.crosstab(data[parent1],data[child], margins=False,
normalize='index').sort_index().to_numpy().reshape(-1).tolist()

    else:

        # Calculate probabilities

        prob=pd.crosstab([data[parent1],data[parent2]],data[child], margins=False,
normalize='index').sort_index().to_numpy().reshape(-1).tolist()

    else: print("Error in Probability Frequency Calculations")

return prob

# Create nodes by using our earlier function to automatically calculate probabilities

H9am = BbnNode(Variable(0, 'H9am', ['<=60', '>60']), probs(df, child='Humidity9amCat'))

H3pm = BbnNode(Variable(1, 'H3pm', ['<=60', '>60']), probs(df, child='Humidity3pmCat',
parent1='Humidity9amCat'))

W = BbnNode(Variable(2, 'W', ['<=40', '40-50', '>50']), probs(df, child='WindGustSpeedCat'))

RT = BbnNode(Variable(3, 'RT', ['No', 'Yes']), probs(df, child='RainTomorrow',
parent1='Humidity3pmCat', parent2='WindGustSpeedCat'))

# Create Network

```

```
bbn = Bbn() \

.add_node(H9am) \

.add_node(H3pm) \

.add_node(W) \

.add_node(RT) \

.add_edge(Edge(H9am, H3pm, EdgeType.DIRECTED)) \

.add_edge(Edge(H3pm, RT, EdgeType.DIRECTED)) \

.add_edge(Edge(W, RT, EdgeType.DIRECTED))
```

```
# Convert the BBN to a join tree
```

```
join_tree = InferenceController.apply(bbn)
```

```
# Set node positions
```

```
pos = {0: (-1, 2), 1: (-1, 0.5), 2: (1, 0.5), 3: (0, -1)}
```

```
# Set options for graph looks
```

```
options = {

    "font_size": 16,

    "node_size": 4000,

    "node_color": "white",

    "edgecolors": "black",

    "edge_color": "red",

    "linewidths": 5,
```



```
"width": 5,}
```

```
# Generate graph
```

```
n, d = bbn.to_nx_graph()
```

```
nx.draw(n, with_labels=True, labels=d, pos=pos, **options)
```

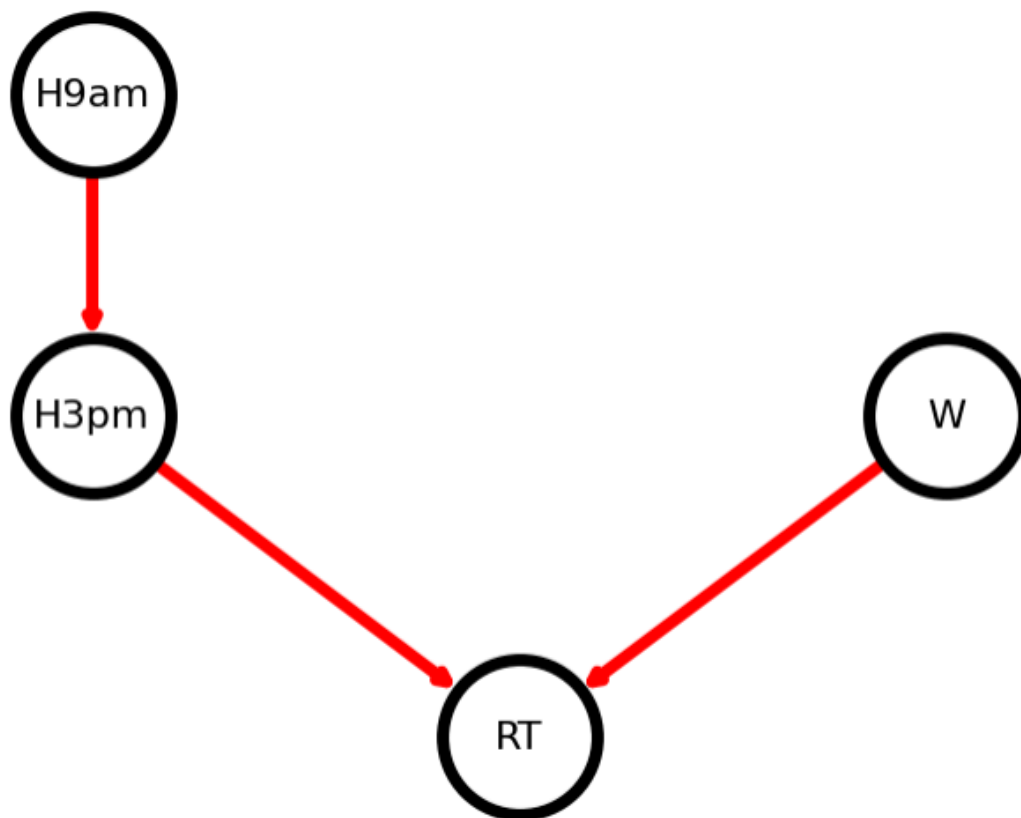
```
# Update margins and print the graph
```

```
ax = plt.gca()
```

```
ax.margins(0.10)
```

```
plt.axis("off")
```

```
plt.show()
```



```
# Define a function for printing marginal probabilities
```

```
def print_probs():
```

```
    for node in join_tree.get_bbn_nodes():
```

```
        potential = join_tree.get_bbn_potential(node)
```

```
        print("Node:", node)
```

```
        print("Values:")
```

```
        print(potential)
```

```
        print('-----')
```

Use the above function to print marginal probabilities

print_probs()

Node: 1|H3pm|<=60,>60

Values:

1=<=60|0.67124

1=>60|0.32876

Node: 0|H9am|<=60,>60

Values:

0=<=60|0.30658

0=>60|0.69342

Node: 2|W|<=40,40-50,>50

Values:

2=<=40|0.58660

2=40-50|0.24040

2=>50|0.17300

Node: 3|RT|No,Yes

Values:

3=No|0.77655

3=Yes|0.22345

To add evidence of events that happened so probability distribution can be recalculated

```
def evidence(ev, nod, cat, val):
```

```
    ev = EvidenceBuilder() \
```

```
        .with_node(join_tree.get_bbn_node_by_name(nod)) \
```

```
        .with_evidence(cat, val) \
```

```
        .build()
```

```
    join_tree.set_observation(ev)
```

```
# Use above function to add evidence
```

```
evidence('ev1', 'H9am', '>60', 1.0)
```

```
# Print marginal probabilities
```

```
print_probs()
```

```
Node: 1|H3pm|<=60,>60
```

```
Values:
```

```
1=<=60|0.55760
```

```
1=>60|0.44240
```

Node: 0|H9am|<=60,>60

```
Values:
```

```
0=<=60|0.00000
```

```
0=>60|1.00000
```

Node: 2|W|<=40,40-50,>50

Values:

2=<=40|0.58660

2=40-50|0.24040

2=>50|0.17300

Node: 3|RT|No,Yes

Values:

3=No|0.73833

3=Yes|0.26167

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT NO. 10

Date of Performance :

Date of Submission :

AIM: Implement a Planning Agent (Planning Programming)

THEORY:

What is planning in AI?

- The planning in Artificial Intelligence is about the decision making tasks performed by the robots or computer programs to achieve a specific goal.
- The execution of planning is about choosing a sequence of actions with a high likelihood to complete the specific task.

Blocks-World planning problem

- The blocks-world problem is known as **Sussman Anomaly**.
- Noninterleaved planners of the early 1970s were unable to solve this problem, hence it is considered as anomalous.
- When two subgoals G1 and G2 are given, a noninterleaved planner produces either a plan for G1 concatenated with a plan for G2, or vice-versa.
- In blocks-world problem, three blocks labeled as 'A', 'B', 'C' are allowed to rest on the flat surface. The given condition is that only one block can be moved at a time to achieve the goal.
- The start state and goal state are shown in the following diagram.

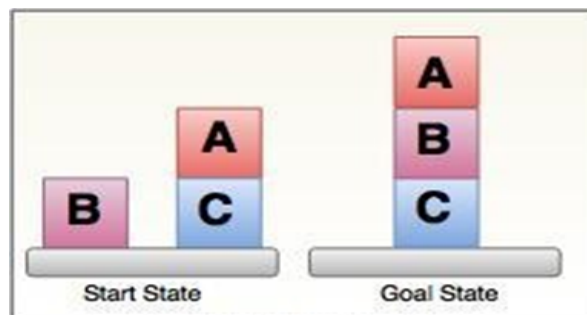


Fig: Blocks-World Planning Problem

Components of Planning System

The planning consists of following important steps:

- Choose the best rule for applying the next rule based on the best available heuristics.
- Apply the chosen rule for computing the new problem state.
- Detect when a solution has been found.
- Detect dead ends so that they can be abandoned and the system's effort is directed in more fruitful directions.
- Detect when an almost correct solution has been found.

Goal stack planning

This is one of the most important planning algorithms, which is specifically used by **STRIPS**.

- The stack is used in an algorithm to hold the action and satisfy the goal. A knowledge base is used to hold the current state, actions.
- Goal stack is similar to a node in a search tree, where the branches are created if there is a choice of an action.

The important steps of the algorithm are as stated below:

- i. Start by pushing the original goal on the stack. Repeat this until the stack becomes empty. If stack top is a compound goal, then push its unsatisfied sub goals on the stack.
- ii. If stack top is a single unsatisfied goal then, replace it by an action and push the action's precondition on the stack to satisfy the condition.
- iii. If stack top is an action, pop it from the stack, execute it and change the knowledge base by the effects of the action.
- iv. If stack top is a satisfied goal, pop it from the stack.

Program

% Initial state

on(a, table).

on(b, table).

on(c, a).

% Goal state

on(a, b).

on(b, table).

on(c, table).

```

% Define the pickup action
pickup(Block, From, State, NewState) :-
    % Preconditions
    on(Block, From),
    clear(Block, State),
    % Effects
    remove(Block, From, State, TempState),
    assert(on(Block, agent)),
    assert(holding(Block, TempState)),
    retract(State, TempState).

% Define the putdown action
putdown(Block, To, State, NewState) :-
    % Preconditions
    holding(Block, State),
    % Effects
    remove(holding(Block, State), State, TempState),
    assert(on(Block, To)),
    assert(clear(Block, TempState)),
    retract(State, TempState).

% Helper predicate to remove a fact from the state
remove(Fact, State, NewState) :-
    select(Fact, State, NewState).

% Helper predicate to check if a block is clear
clear(Block, State) :-
    \+ on(_, Block, State).

% Depth-first search
dfs(State, Goal, _, []) :- State = Goal.
dfs(State, Goal, Visited, [Action | Plan]) :-
    move(State, Action, NewState),
    \+ member(NewState, Visited), % Avoid revisiting states
    dfs(NewState, Goal, [NewState | Visited], Plan).

% Main planning predicate
plan(Plan) :-
    initial_state(InitialState),
    goal_state(GoalState),
    dfs(InitialState, GoalState, [InitialState], Plan).

```

Executable code
?- plan(Plan).

CONCLUSION:

The planning process of any organization is essential to the overall success of the company. All levels of planning must do their part to incorporate situational analysis, alternative goals and plans, goals and plan evaluation, goal and plan selection, implementation, and monitor control into their organization to achieve the desired results. In addition to these steps, managers must also consider the various impacts such as legal issues, ethics, and corporate social responsibility as well as internal and external factors when devising plans for their organization. As proven by the ongoing success of the Boeing Company, these steps and considerations in the hands of the right managers insure a promising future.

SIGN AND REMARK:**DATE:**

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT NO. 11

Date of Performance :

Date of Submission :

AIM: Design prototype for Expert System.

THEORY:

Designing a prototype for an expert system involves several steps and considerations. An expert system is a computer program that uses knowledge and inference techniques to solve specific problems or provide expert-level advice in a particular domain. Here's a high-level outline of how you can design a prototype for an expert system:

Define the Problem Domain:

- Clearly define the problem or domain your expert system will address. This could be anything from medical diagnosis to technical support.

Identify Expertise:

- Determine the source of expertise for your system. This could involve consulting domain experts or gathering data from books, research papers, or databases.

Knowledge Acquisition:

- Gather and organize knowledge from experts or existing sources. This knowledge can be represented as rules, facts, and heuristics. Common techniques for knowledge representation include:
 - Rule-based systems: If-Then rules
 - Knowledge graphs: Representing knowledge as nodes and edges
 - Frames: Structured representation of objects and their properties
 - Ontologies: Formal representation of knowledge with relationships and semantics

Knowledge Base:

- Create a knowledge base to store the acquired knowledge. This is where all the rules, facts, and data will be stored for the expert system to use during inference.

Inference Engine:

- Develop the core inference engine responsible for making decisions and providing recommendations based on the knowledge in the knowledge base. Common inference techniques include:
 - Forward chaining: Start with available facts and derive conclusions.
 - Backward chaining: Start with a goal and work backward to find the

necessary facts.

- Bayesian networks: Probability-based reasoning.
- Fuzzy logic: Dealing with uncertainty.

User Interface:

- Design a user-friendly interface for interacting with the expert system. Depending on your target audience, this could be a command-line interface, a web application, or a mobile app.

Testing and Validation:

- Test the expert system with various scenarios and use cases to ensure it provides accurate and reliable recommendations or solutions.

Knowledge Refinement:

- Continuously update and refine the knowledge base as new information becomes available or the domain evolves.

Deployment:

- Deploy the expert system in a real-world setting where it can be used by individuals or organizations to solve problems or provide expert advice.

Monitoring and Maintenance:

- Implement monitoring tools to keep track of system performance and user feedback. Regularly update and maintain the system to ensure its accuracy and relevance.

User Training:

- Provide training and documentation to users on how to effectively use the expert system.

Scaling and Optimization:

- As the system gains more users and data, consider scalability and performance optimization.

Remember that developing an expert system can be a complex task, and the choice of programming languages and technologies may vary depending on your specific requirements. Additionally, you may need to consider ethical and legal implications, especially if the system is used in critical domains like healthcare or finance.

Practical implementation of an expert system prototype involves several steps and considerations. Below, I'll outline a simplified example of how you might practically implement an expert system prototype for a medical diagnosis domain using Python. Keep in mind that real-world expert systems can be much more complex and may require a dedicated team and

resources.

Define the Problem Domain:

- In this example, let's create an expert system prototype for diagnosing common illnesses based on symptoms.

Identify Expertise:

- Consult medical experts or gather medical literature to identify symptoms, diseases, and diagnostic rules.

Knowledge Acquisition:

- Create a knowledge base with symptom-disease associations. For instance, if "fever" and "cough" are symptoms, they may suggest "common cold" as a possible disease.

Knowledge Base:

- Store the knowledge in a structured format, such as a Python dictionary or database.

```
knowledge_base = {  
  
    "fever": ["common cold", "flu"],  
  
    "cough": ["common cold", "bronchitis"],  
  
    # Add more symptom-disease associations  
  
}
```

Inference Engine:

- Develop an inference engine to make diagnoses based on user-entered symptoms. You can use a simple rule-based approach for this prototype.

```
def diagnose(symptoms):  
    possible_diseases = []  
    for symptom in symptoms:  
        if symptom in knowledge_base:  
            possible_diseases.extend(knowledge_base[symptom])  
    return list(set(possible_diseases))  
  
user_symptoms = ["fever", "cough"]  
diagnosis = diagnose(user_symptoms)
```

```
print("Possible diseases:", diagnosis)
```

User Interface:

- Create a basic command-line interface for users to input their symptoms and receive a diagnosis.

Testing and Validation:

- Test the system with various symptom combinations to ensure it provides accurate diagnoses based on the knowledge base.

Knowledge Refinement:

- Continuously update and refine the knowledge base with new symptom-disease associations or medical guidelines.

Deployment:

- Deploy the prototype locally for testing or share it within a controlled environment.

Monitoring and Maintenance:

- Monitor user interactions and feedback to identify any issues or improvements needed.

User Training:

- Provide instructions on how to use the system effectively.

Scaling and Optimization:

- If you plan to expand the system or make it more efficient, consider optimizing the code and potentially migrating to a more robust platform or framework.

Remember that this is a simplified example. Real-world expert systems in domains like medicine would require a much more extensive and accurate knowledge base, as well as compliance with medical regulations. Additionally, you may want to implement more advanced inference techniques and deploy the system in a secure and accessible manner.

EXPERIMENT NO. 12

Date of Performance :

Date of Submission :

Aim : Case study of any existing successful AI system

Case Study: Google's AlphaGo

Background: Go is a complex and ancient board game with an enormous number of possible moves, making it significantly more challenging for computers to play compared to chess. For many years, Go was considered one of the last frontiers of human strategic gameplay that AI couldn't master.

Problem Statement: Google's DeepMind set out to develop an AI system capable of playing Go at a world-class level and defeating the world champion.

AI Solution:

Deep Reinforcement Learning: AlphaGo employed a combination of deep neural networks and reinforcement learning. It used neural networks to evaluate board positions and predict the best moves based on past experience.

Monte Carlo Tree Search (MCTS): AlphaGo used MCTS, a search algorithm that simulates millions of possible future moves and outcomes, to select the best moves during gameplay.

Training Data: The AI system was trained on a massive dataset of expert-level Go games to learn patterns, strategies, and tactics. It also engaged in self-play to improve its gameplay continuously.

Learning from Human Experts: AlphaGo played thousands of games against top human Go players to learn and adapt its strategies.

Results:

- AlphaGo vs. Lee Sedol: In 2016, AlphaGo faced the world champion Go player, Lee Sedol, in a five-game match. AlphaGo won four out of five games, demonstrating its superiority in strategic gameplay.
- Advancements in AI: The success of AlphaGo marked a significant milestone in AI research, showcasing the potential of deep learning, reinforcement learning, and neural networks in solving complex, real-world problems.
- Impacts Beyond Games: The techniques developed for AlphaGo have applications in fields like healthcare, finance, and logistics, where AI can be used for complex decision-making and optimization.

- AlphaGo Zero: DeepMind later developed AlphaGo Zero, a version of the AI system that learned entirely from self-play without any human data. It achieved an even higher level of performance, further highlighting the capabilities of AI in mastering complex tasks.

Google's AlphaGo serves as an exemplary case study of how AI, particularly deep reinforcement learning and neural networks, can excel in tasks previously considered too challenging for machines. It not only revolutionized the field of AI but also demonstrated the potential for AI to tackle complex problems across various domains.

EXPERIMENT NO. 13

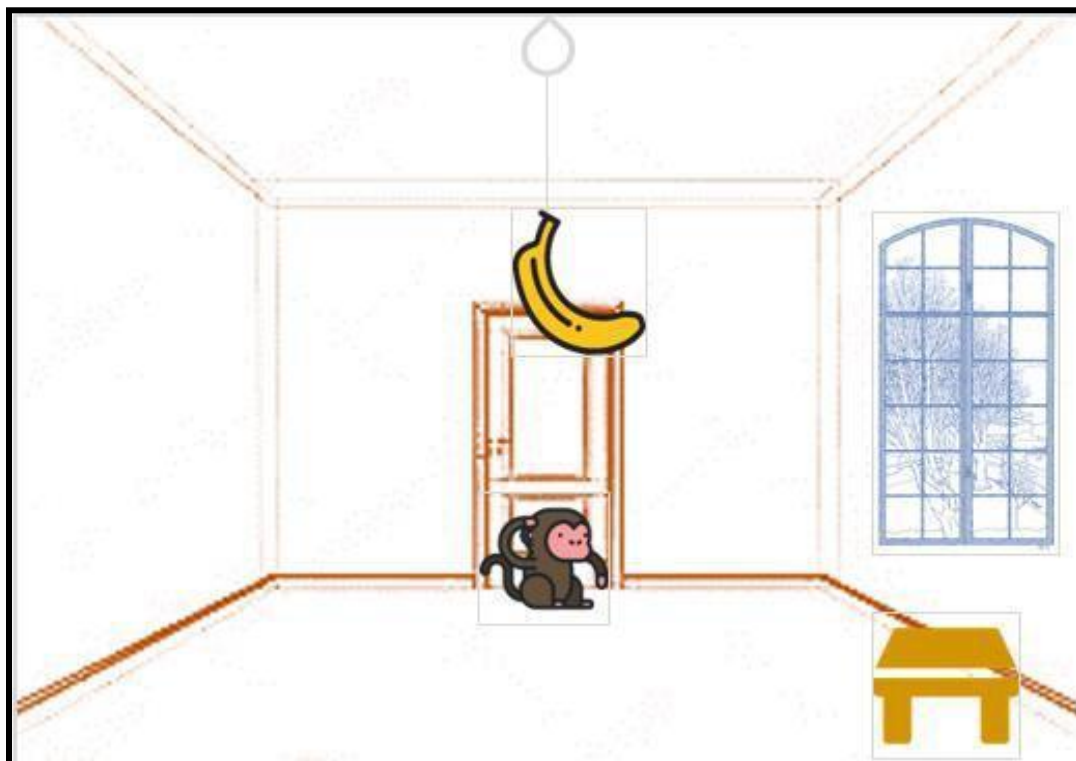
Date of Performance :

Date of Submission :

{EXPERIMENT BEYOND SYLLABUS}

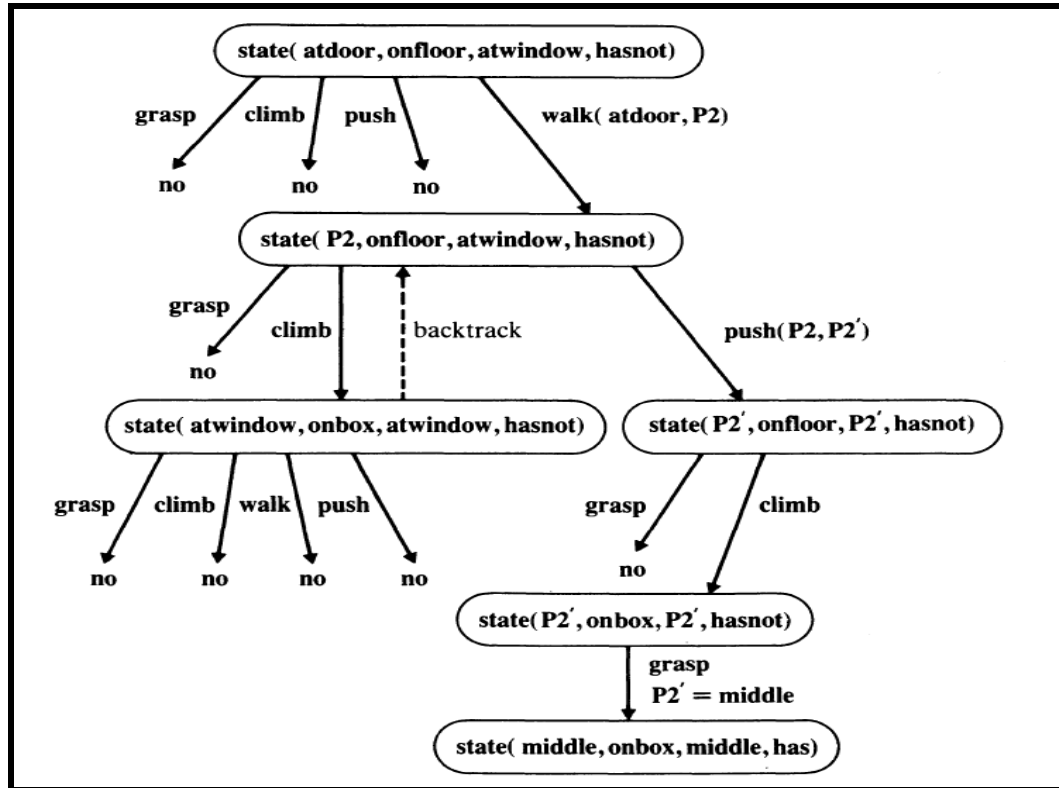
AIM: Explore and analyze the application of Prolog for solving classic problems like the monkey banana and optimizing solutions through the effective use of data structures.

13.1 Monkey Banana problem



Moves

1. Grasp banana
2. Climb box
3. Push box
4. Walk around



CODE:

```

move(state(middle,onbox,middle,hasnot),
  grasp,
  state(middle,onbox,middle,has)).
move(state(P,onfloor,P,H),
  climb,
  state(P,onbox,P,H)).
move(state(P1,onfloor,P1,H),
  drag(P1,P2),
  state(P2,onfloor,P2,H)).
move(state(P1,onfloor,B,H),
  walk(P1,P2),
  state(P2,onfloor,B,H)).
canget(state(_,_,_,has)).
canget(State1) :-
  move(State1,_,State2),
  canget(State2).

```

OUTPUT:

```
| ?- consult('D:/prolog_assignments/Monkeybanana.pl').
compiling D:/prolog_assignments/Monkeybanana.pl for byte code...
D:/prolog_assignments/Monkeybanana.pl compiled, 15 lines read - 2157 bytes written, 8 ms

yes
| ?- canget(state(atdoor,onfloor,atwindow,hasnot)).

true ?

yes
| ?- trace.
The debugger will first creep -- showing everything (trace)

yes
{trace}
| ?- canget(state(atdoor,onfloor,atwindow,hasnot)).
1 1 Call: canget(state(atdoor,onfloor,atwindow,hasnot)) ?
2 2 Call: move(state(atdoor,onfloor,atwindow,hasnot),_52,_92) ?
2 2 Exit: move(state(atdoor,onfloor,atwindow,hasnot),walk(atdoor,_80),state(_80,onfloor,atwindow,hasnot)) ?
3 2 Call: canget(state(_80,onfloor,atwindow,hasnot)) ?
4 3 Call: move(state(_80,onfloor,atwindow,hasnot),_110,_150) ?
4 3 Exit: move(state(atwindow,onfloor,atwindow,hasnot),climb,state(atwindow,onbox,atwindow,hasnot)) ?
5 3 Call: canget(state(atwindow,onbox,atwindow,hasnot)) ?
6 4 Call: move(state(atwindow,onbox,atwindow,hasnot),_165,_205) ?
6 4 Fail: move(state(atwindow,onbox,atwindow,hasnot),_165,_193) ?
5 3 Fail: canget(state(atwindow,onbox,atwindow,hasnot)) ?
4 3 Redo: move(state(atwindow,onfloor,atwindow,hasnot),climb,state(atwindow,onbox,atwindow,hasnot)) ?
4 3 Exit: move(state(atwindow,onfloor,atwindow,hasnot),drag(atwindow,_138),state(_138,onfloor,_138,hasnot)) ?
5 3 Call: canget(state(_138,onfloor,_138,hasnot)) ?
6 4 Call: move(state(_138,onfloor,_138,hasnot),_168,_208) ?
6 4 Exit: move(state(_138,onfloor,_138,hasnot),climb,state(_138,onbox,_138,hasnot)) ?
7 4 Call: canget(state(_138,onbox,_138,hasnot)) ?
8 5 Call: move(state(_138,onbox,_138,hasnot),_223,_263) ?
8 5 Exit: move(state(middle,onbox,middle,hasnot),grasp,state(middle,onbox,middle,has)) ?
9 5 Call: canget(state(middle,onbox,middle,has)) ?
9 5 Exit: canget(state(middle,onbox,middle,has)) ?
7 4 Exit: canget(state(middle,onbox,middle,hasnot)) ?
5 3 Exit: canget(state(middle,onfloor,middle,hasnot)) ?
3 2 Exit: canget(state(atwindow,onfloor,atwindow,hasnot)) ?
1 1 Exit: canget(state(atdoor,onfloor,atwindow,hasnot)) ?

true ?

(15 ms) yes
{trace}
| ?- |
```

13.2 Data Structures

13.2.1 List and Members

CODE:

member(X,[X|Tail]).

member(X,[Head|Tail]):-member(X,Tail).

conc([],L,L).

conc([X|L1],L2,[X|L3]):-conc(L1,L2,L3).

del(X,[X|Tail],Tail).

del(X,[Y|Tail],[Y|Tail1]):-del(X,Tail,Tail1).

sublist(S,L):-conc(L1,L2,L),conc(S,L3,L2).

OUTPUT:

```
| ?- consult('D:/prolog_assignments/list.pl').
compiling D:/prolog_assignments/list.pl for byte code...
D:/prolog_assignments/list.pl:1: warning: singleton variables [Tail] for member/2
D:/prolog_assignments/list.pl:2: warning: singleton variables [Head] for member/2
D:/prolog_assignments/list.pl:10: warning: singleton variables [L1,L3] for sublist/2
D:/prolog_assignments/list.pl compiled, 9 lines read - 1508 bytes written, 5 ms
error: D:/prolog_assignments/list.pl:1: native code procedure member/2 cannot be redefined (ignored)
error: D:/prolog_assignments/list.pl:10: native code procedure sublist/2 cannot be redefined (ignored)

(15 ms) yes
| ?- conc(Before,[may|After],[jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec]).

After = [jun,jul,aug,sep,oct,nov,dec]
Before = [jan,feb,mar,apr] ? ;

(31 ms) no
| ?- member(zaurez,[zaurez,shrishti,fawaz]).

true ? ;

no
| ?- member(vinayak,[zaurez,shrishti,fawaz]).

no
| ?- member([shrishti,fawaz],[zaurez,shrishti,fawaz]).

no
| ?- conc([zaurez,shrishti],[x,y,z],L).

L = [zaurez,shrishti,x,y,z]

yes
| ?- del(zaurez,[zaurez,shrishti,fawaz],L).

L = [shrishti,fawaz] ? p
```

```
| ?- permutation([a,b,c],P).

P = [a,b,c] ?

yes
| ?- sublist(S,[a,b,c]).

S = [a,b,c] ? ;
S = [b,c] ? ;
S = [c] ? ;
S = [] ? ;
S = [b] ? ;
S = [a,c] ? ;
S = [a] ? ;
S = [a,b] ? ;

no
| ?-
```

13.2.2 Segmentation

CODE:

```
vertical(seg(point(X,Y),point(X,Y1))).  
horizontal(seg(point(X,Y),point(X1,Y))).
```

OUTPUT:

```
GNU Prolog 1.5.0 (64 bits)  
Compiled Jul  8 2021, 12:33:56 with cl  
Copyright (C) 1999-2021 Daniel Diaz  
  
| ?- consult('D:/prolog_assignments/segmentation.pl').  
compiling D:/prolog_assignments/segmentation.pl for byte code...  
D:/prolog_assignments/segmentation.pl:1: warning: singleton variables [Y,Y1] for vertical/1  
D:/prolog_assignments/segmentation.pl:2: warning: singleton variables [X,X1] for horizontal/1  
D:/prolog_assignments/segmentation.pl compiled, 1 lines read - 797 bytes written, 7 ms  
  
yes  
| ?- vertical(seg(point(X,Y),point(X,Y1))).  
  
yes  
| ?- vertical(seg(point(12,5),point(12,7))).  
  
yes  
| ?- vertical(seg(point(X,Y),point(X1,Y))).  
X1 = X  
  
yes  
| ?- horizontal(seg(point(X,Y),point(X1,Y))).  
  
(16 ms) yes  
| ?- horizontal(seg(point(4,5),point(3,2))).  
  
no  
| ?- horizontal(seg(point(4,5),point(7,9))).  
  
no  
| ?- horizontal(seg(point(7,8),point(7,9))).  
  
no  
| ?- horizontal(seg(point(12,5),point(12,7))).  
  
no  
| ?- horizontal(seg(point(12,5),point(7,5))).  
  
yes  
| ?-
```

CONCLUSION: SUCCESSFULLY IMPLEMENTED AND LEARNED THAT PROLOG IS USEFUL FOR SOLVING FAMILIAR PROBLEMS LIKE MONKEY BANANA PUZZLE AND DATA STRUCTURE . USING THE RIGHT DATA STRUCTURES MADE THE SOLUTION EVEN BETTER, REVEALING HOW SMART PLANNING AND PROBLEM-SOLVING METHODS MATTER FOR DEALING WITH TOUGH SITUATIONS.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature

EXPERIMENT NO. 14

Date of Performance :

Date of Submission :

{EXPERIMENT BEYOND SYLLABUS}

AIM: Explore and analyze the application of Prolog for solving classic problems like flight planning.

14. Flight Problem

CODE:

```
% A FLIGHT ROUTE PLANNER
:- op(50,xfy,:).
flight( Place1, Place2, Day, Fnum, Deptime, Arrtime) :-
timetable( Place1 , Place2, Flightlist),
member( Deptime / Arrtime / Fnum / Daylist , Flightlist),
flyday( Day, Daylist).
member(X,[X|_]).
member(X,[_|_]):-member(X,_).
flyday( Day, Daylist) :-
member( Day, Daylist).
flyday(Day, alldays) :- member(Day,[mo,tu,we,th,fr,sa,su]).
% Direct flight
route(P1, P2, Day,[P1-P2 : Fnum : Deptime]) :-
flight(P1, P2,Day,Fnum, Deptime, _).
% Indirect flight
route(P1, P2, Day, [P1-P3 : Fnum1:Dep1 | Route]):-
route(P3, P2, Day, Route),
flight(P1, P3, Day, Fnum1, Dep1,Arr1),
deptime( Route, Dep2),
transfer( Arr1, Dep2).
deptime([P1-P2:Fnum:Dep|_],Dep).
transfer(Hours1:Mins1,Hours2:Mins2):-60*(Hours2-Hours1)+Mins2-Mins1>=40.
% A FLIGHT DATABASE
timetable( edinburgh, london,
[ 9:40/ 10:50/ba4733/ alldays,
13:40/ 14:50/ba4773/ alldays,
19:40/ 20:50/ba4833/ [mo,tu,we,th,fr,su]] ).
timetable( london, edinburgh,
```

```

[ 9:40/ 10:50/ba4732/ alldays,
11:40/ 12:50/ba4752/ alldays,
18:40/ 19:50/ba4822/ [mo,tu,we,th,fr]] ).
timetable( london, ljubljana,
[ 13:20/ 16:20/ ju201 / [fr],
13:20/ 16:20/ ju213/[su]]).
timetable( london, zurich,
[ 9:10 / 11:45/ ba6l4 / alldays,
14:45 / 17:20 / sr805 / alldays] ).
timetable( london, milan,
[8:30 / 11:20 / ba510/ alldays,
11:00/ 13:50/ az459/ alldays] ).
timetable( ljubljana, zurich,
[ 11:30/ 12:40/ ju322/ [tu,th] ] ).
timetable( ljubljana, london,
[ 11: 10/ 12:20/ yu200/ [fr],
11:25/ 12:20/ yu212/ [su]]).
timetable( milan, london,
[ 9:10 / 10:00/ az458/ alldays,
12:20/ 13:10/ ba511/ alldays] ).
timetable( milan, zurich,
[ 9:25 / 10:15 / sr621 / alldays,
12:45 / 13:35 / sr623 / alldays] ).
timetable( zurich, ljubljana,
[ 13:30/ 14:40/ yu323/ [tu,th]] ).
timetable( zurich, london,
[ 9:00 / 9:40 / ba6l3 / [mo,tu,we,th,fr,sa],
16:10/ 16:55 / sr806/ [mo,tu,we,th,fr,su]] ) .
timetable( zurich, milan,
[ 7:55 / 8:45 / sr620 | alldays] ).
add(X,L,[X|L]).
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]):-del(X,Tail,Tail1).
% permutation([],[]).
% permutation(L,[X|P]):-del(X,L,L1),permutation(L1,P).

```

OUTPUT:

```
D:/prolog_assignments/Flightproblem.pl:29: warning: singleton variables [P1,P2,Fnum] for deptime/2
D:/prolog_assignments/Flightproblem.pl compiled, 82 lines read - 17454 bytes written, 50 ms
error: D:/prolog_assignments/Flightproblem.pl:10: native code procedure member/2 cannot be redefined (ignored)
```

```
(15 ms) yes
| ?- flight(london, ljubljana, Day,_,_,_).
```

```
Day = fr ?
```

```
(15 ms) yes
| ?- flight(london, edinburgh, Day,_,_,_).
```

```
Day = mo ?
```

```
yes
| ?- flight(london, zurich, Day,_,_,_).
```

```
Day = mo ?
```

```
yes
| ?- flight(london, zurich, Day,_,_,_).
```

```
Day = mo ? ;
```

```
Day = tu ?
```

```
(15 ms) yes
| ?- flight(london, zurich, Day,_,_,_).
```

```
Day = mo ? ;
```

```
Day = tu ? ;
```

```
Day = we ? ;
```

```
Day = th ?
```

```
yes
| ?- flight(london, ljubljana, Day,_,_,_).
```

```
Day = fr ? ;
```

```
Day = su ?
```

```
yes
| ?-
```

```
| ?- route(london,milan,tu,Route1),
route(milan,ljubljana,_,Route2),
route(ljubljana,zurich,th,Route3),
route(zurich,london,fr,Route4).
```

```
Route1 = [london-milan:ba510:8:30]
```

```
Route2 = [milan-london:az458:9:10,london-ljubljana:ju201:13:20]
```

```
Route3 = [ljubljana-zurich:ju322:11:30]
```

```
Route4 = [zurich-london:ba613:9:0] ? |
```


CONCLUSION: SUCCESSFULLY IMPLEMENTED AND LEARNED THAT **PROLOG** IS USEFUL FOR SOLVING FAMILIAR PROBLEMS LIKE FINDING PERFECT FLIGHT ROUTES. **IT** ALSO REVEALS HOW SMART PLANNING AND PROBLEM-SOLVING METHODS MATTER FOR DEALING WITH TOUGH SITUATIONS.

SIGN AND REMARK:

DATE:

R1 (3 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (4 Marks)	Total (15 Marks)	Signature