GURU TEGH BAHADUR 4TH CENTENARY ENGINEERING COLLEGE

G-8 AREA, RAJOURI GARDEN, NEW DELHI-110064

INTELLIGENT AND EXPERT SYSTEM

PRACTICAL FILE

Course Code: AI-405P

Semester: 7th

Submitted to :

Ms. Babita

Submitted by :

SAIMA

03423802722

CSE 1

INDEX

S.No.	Experiment Name	Date	Signature
1	STUDY OF PROLOG	28-08=2025	
2	PROGRAM TO SOLVE EIGHT QUEENS PROBLEM	28-08-2025	
3	PROGRAM TO IMPLEMENT DEPTH FIRST SEARCH	04-09-2025	
4	PROGRAM TO IMPLEMENT BEST FIRST SEARCH	04-09-2025	
5	SOLVE 8 PUZZLE PROBLEM USING BEST FIRST SEARCH	11-09-2025	
6	PROGRAM TO SOLVE ROBOT TRAVERSAL PROBLEM USING MEANS END ANALYSIS	11-09-2025	
7	IMPLEMENTATION OF TRAVELING SALESMAN PROBLEM	18-09-2025	
8	PROGRAM TO IMPLEMENT BREADTH FIRST SEARCH	18-09-2025	
9	PROGRAM TO FIND OUT UNION AND INTERSECTION OF TWO LISTS.	25-09-2025	
10	PROGRAM TO CALCULATE FACTORIAL OF A GIVEN NUMBER	25-09-2025	
11	PROGRAM TO FLATTEN A LIST	09-10-2025	
12	PROGRAM TO SOLVE MONEY BANANA PROBLEM	09-10-2025	

GURU TEGH BAHADUR 4TH CENTENARY ENGINEERING COLLEGE

SESSION: 2022-2026

PRACTICAL FILE

Branch: CSE-1

INTELLIGENT AND EXPERT SYSTEM

Name:
Enrollment no.:

Submitted to: Ms. Babita

INDEX

S.No.	Experiment Name	Date	Teacher's Signature
1	STUDY OF PROLOG		
2	PROGRAM TO SOLVE EIGHT QUEENS PROBLEM		
3	PROGRAM TO IMPLEMENT DEPTH FIRST SEARCH		
4	PROGRAM TO IMPLEMENT BEST FIRST SEARCH		
5	SOLVE 8 PUZZLE PROBLEM USING BEST FIRST SEARCH		
6	PROGRAM TO SOLVE ROBOT TRAVERSAL PROBLEM USING MEANS END ANALYSIS		
7	IMPLEMENTATION OF TRAVELING SALESMAN PROBLEM		
8	PROGRAM TO IMPLEMENT BREADTH FIRST SEARCH		
9	PROGRAM TO FIND OUT UNION AND INTERSECTION OF TWO LISTS.		
10	PROGRAM TO CALCULATE FACTORIAL OF A GIVEN NUMBER		
11	PROGRAM TO FLATTEN A LIST		
12	PROGRAM TO SOLVE MONEY BANANA PROBLEM		

PROGRAM-1

OBJECTIVE: **STUDY OF PROLOG**

PROLOG-PROGRAMMING IN LOGIC

PROLOG stands for *Programming In Logic* – an idea that emerged in the early 1970's to use logic as programming language. The early developers of this idea included Robert Kowalski at Edinburgh (on the theoretical side),Marteen van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).
David D.H. Warren's efficient implementation at Edinburgh in the mid -1970's greatly contributed to the popularity of PROLOG.

PROLOG is a programming language centered around a small set of basic mechanisms, Including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework.
PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

SYMBOLIC LANGUAGE

PROLOG is a programming language for symbolic, non-numeric computation. It is Especially well suited for solving problems that involve objects and relations between objects.

For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y, and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for *Artificial* Language*(AI)* and non- numerical programming.

There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code. When the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES

Programming in PROLOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program. Responses to user queries are determined through a form of inferencing control known as *resolution.*

FOR EXAMPLE:

FACTS:
Some facts about family relationships could be written as :
sister(sue,bill) parent(ann,sam) male(jo) female(riya)

RULES:
To represent the general rule for grandfather, we write: grandfather(X,Z)
parent(X,Y) parent(Y,Z) male(X)

QUERIES:
Given a database of facts and rules such as that above, we may make queries by typing after a query a symbol '?_' statements such as:
?_ parent(X,sam) X=ann
?_grandfather(X,Y) X=jo, Y=sam

PROLOG IN DESIGINING EXPERT SYSTEMS

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources such as texts, journal articles, databases etc. and encoded in a form suitable for the system to use in its inference or reasoning processes. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system.

PROLOG serves as a powerful language in designing expert systems because of its following features:

Use of knowledge rather than data
Modification of the knowledge base without recompilation of the control programs.
Capable of explaining conclusion.
Symbolic computations resembling manipulations of natural language.
Reason with meta-knowledge.

META PROGRAMMING

A meta program is a program that takes other programs as data. Interpreters and compilers are examples of meta-programs. Meta-interpreter is a particular kind of meta- program: an interpreter for a language written in that language. So a PROLOG mete- interpreter is an interpreter for PROLOG, itself written in PROLOG.
Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta- programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

PROGRAM- 2

OBJECTIVE: PROGRAM TO SOLVE EIGHT QUEENS PROBLEM

ALGORITHM TO SOLVE EIGHT QUEENS PROBLEM

Step 1: Represent the board position as 8*8 vector, i.e., [1,2,3,4,5,6,7.8]. Store the set of queens in the list 'Queens'.
Step 2: Calculate the permutation of the above eight numbers stored in set P.
Step 3: Let the position where the first queen to be placed be(1,Y), for second be (2,Y1), and so on and store the position in S.
Step 4: Check for the safety of the queens through the predicate, 'no attack()'.
Step 5: Calculate Y1-y and Y-Y1. If both are not equal to Xdist, which is the X-distance between the first queen and others, then go to step 6 else go to step 7.

Step 6: Increment Xdist by 1.

Step 7: Repeat above for the rest of the queens, until the end of the list is reached. Step 8: Print S as answer.
Stop

SOURCE CODE:

```
% Solves the 8-Queens puzzle.

solution(Queens) :-
% The list of queens is a permutation of the row numbers [1..8].
permutation([1,2,3,4,5,6,7,8], Queens),
% Check if this permutation is a safe arrangement.
safe(Queens).

% --- Predicates for permutation ---

permutation([], []).
permutation([H|T], PL) :-
permutation(T, PT),
% del(Item, ListWithItem, ListWithoutItem)
% Here it's used to "insert" H into PT to get PL.
del(H, PL, PT).

del(I, [I|L], L).
del(I, [F|L1], [F|L2]) :-
del(I, L1, L2).

% --- Predicates for checking safety ---

% An empty list of queens is safe.
safe([]).
% A list is safe if the tail is safe, and the head doesn't attack the tail.
safe([Queen|Others]) :-
safe(Others),
noattack(Queen, Others, 1).

% A queen does not attack an empty list of other queens.
noattack(_, [], _).
```

```
% Check the current queen against the rest of the list.
noattack(Y, [Y1|Ylist], Xdist) :-
% Check for diagonal attacks.
Y1 - Y =\= Xdist,
Y - Y1 =\= Xdist,
% Increment the column distance and recurse.
Dist1 is Xdist + 1,
noattack(Y, Ylist, Dist1).
```

OUTPUT:

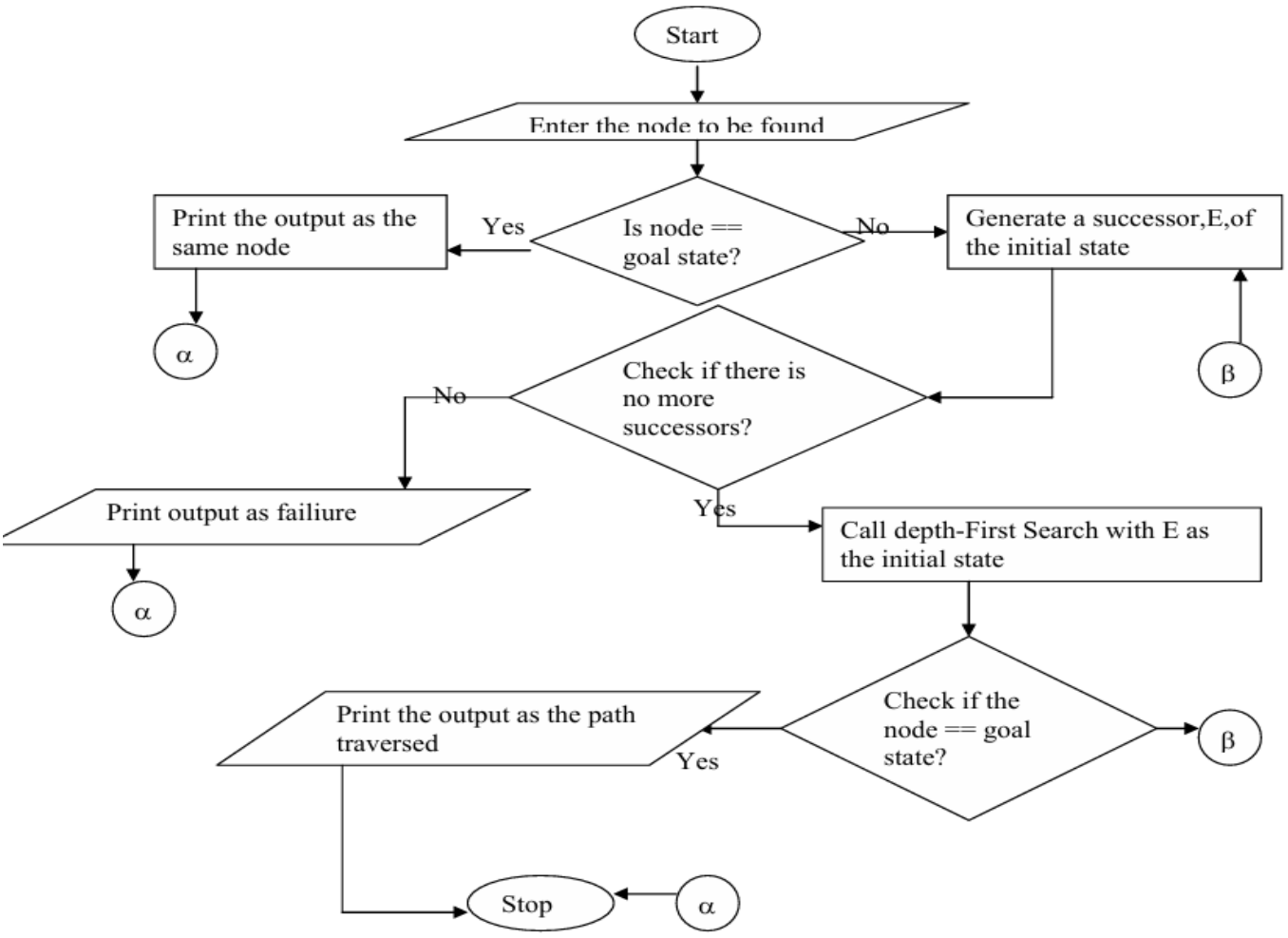
Query : solution(X).

X = [5, 2, 6, 1, 7, 4, 8, 3]

PROGRAM-3

OBJECTIVE: PROGRAM TO IMPLEMENT DEPTH FIRST SEARCH

FLOWCHART TO IMPLEMENT DEPTH FIRST SEARCH



ALGORITHM TO IMPLEMENT DEPTH-FIRST SEARCH

Start
Step 1: Enter the node to be found
Step 2: If the initial state is a goal state, quit and return success
Step 3: Otherwise, do the following until success or failure is signaled.
Generate a successor, E, of the initial state. If there are no more successors, signal failure
Call Depth-First Search with E as the initial state.
If success is retuned, signal success. Otherwise continue in this loop.
Step 4: Print the output as the path traversed Stop

SOURCE CODE:

```
% Defines the directed edges of the graph
childnode(a,b).
childnode(a,c).
childnode(c,d).
childnode(c,e).

% Base case: A path exists if there is a direct edge.
path(A, B, [A,B]) :-
childnode(A,B).

% Recursive step: A path exists if there's an edge from A to X,
% and a path from X to B.
path(A, B, [A|RestOfPath]) :-
childnode(A,X),
path(X, B, RestOfPath).
```

OUTPUT :

Query : path(a, d, P).
P = [a, c, d]

PROGRAM-4

OBJECTIVES: PROGRAM TO IMPLEMENT BEST FIRST SEARCH

ALGORITHM TO IMPLEMENT BEST FIRST SEARCH

Step 1: Enter the node to be found.
Step 2: Start with OPEN containing just the initial state.
Step 3: Until a goal is found or there are no nodes left on OPEN do:
Pick the best node on OPEN.
Generate its successors.
For each successor do:
If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.
Step 4: Print the output as the path traversed.
Step 5: Exit

SOURCE CODE:

```
% ================================================================
% GREEDY BEST-FIRST SEARCH IMPLEMENTATION
% ================================================================

% --- MAIN PREDICATE ---
% best_first_search(StartNode, GoalNode, Path).
best_first_search(Start, Goal, Path) :-
% Start the search with the Open list containing just the initial path.
% The Open list stores paths, e.g., [[c,b,a], [d,a], ...], sorted by heuristic.
solve([[Start]], Goal, RevPath),
reverse(RevPath, Path).

% --- THE SOLVER ---
% solve(OpenList, GoalNode, SolutionPath).

% Base Case: The best path on the Open list leads to the goal.
solve([[Goal | Path] | _], Goal, [Goal | Path]).

% Recursive Step: Expand the best node from the Open list.
solve([[Current | Path] | RestOpen], Goal, Solution) :-
% Find all valid new paths by extending the current path by one step.
findall([Next, Current | Path],
(s(Current, Next), \+ member(Next, [Current | Path])),
NewPaths),
% Merge the new paths into the Open list, keeping it sorted by heuristic value.
merge_open_lists(NewPaths, RestOpen, NewOpen),
solve(NewOpen, Goal, Solution).

% Fail Case: No more nodes to explore.
solve([], _, _) :-
write('No solution found.'), nl,
fail.
```

```
% --- MERGE PREDICATE (Heuristic Sorting) ---
% Merges new paths into the Open list, keeping it sorted by heuristic value.
merge_open_lists(New, Old, Merged) :-
append(New, Old, Temp),
predsort(compare_paths, Temp, Merged).
% Comparison predicate for sorting paths based on the heuristic of their head node.
compare_paths(Order, [State1 | _], [State2 | _]) :-
h(State1, H1),
h(State2, H2),
compare(Order, H1, H2).


% --- UTILITY PREDICATES ---
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

append([], L, L).
append([X|L1], L2, [X|L3]) :-
append(L1, L2, L3).
% ========================================================================
% EXAMPLE PROBLEM DEFINITION
% ========================================================================

% --- 1. Define the graph using s(Node1, Node2) ---
s(a, b). s(a, c).
s(b, d). s(b, e).
s(c, f). s(c, g).
s(e, h).
s(f, h).

% --- 2. Define the heuristic h(Node, Value) ---
% This is the estimated distance from the node to the goal 'h'.
h(a, 6). h(b, 4). h(c, 4).
h(d, 5). h(e, 2). h(f, 2).
h(g, 5). h(h, 0). % Heuristic for the goal is always 0.

% --- 3. Define the start and goal ---
start_node(a).
goal_node(h).
```

OUTPUT :

Query : start_node(S), goal_node(G), best_first_search(S, G, Path).
G = h ,
Path = [a,b,e,h] ,
S = a

PROGRAM-5

OBJECTIVE: SOLVE 8 PUZZLE PROBLEM USING BEST FIRST SEARCH

SOURCE CODE:

```
% ================================================================
% 8-PUZZLE SOLVER USING BEST-FIRST SEARCH
% ================================================================
```

1. **Represent the puzzle** as a list of 9 numbers — 0 means the empty tile.
   Example:
   [1,2,3,4,0,5,6,7,8]
2. **Goal state:** [1,2,3,4,5,6,7,8,0]
3. **Heuristic:** count how many tiles are not in their goal position.
4. **Moves:** blank (0) can move up, down, left, or right.
5. **Best-first search:** always expand the state with the fewest misplaced tiles.

```
% ---------- SIMPLE 8 PUZZLE USING BEST FIRST SEARCH ----------

% Goal state
goal([1,2,3,4,5,6,7,8,0]).

% Heuristic: number of misplaced tiles
h(State, H) :-
goal(Goal),
findall(X, (nth0(I, State, X), nth0(I, Goal, Y), X \= 0, X \= Y), L),
length(L, H).

% Possible moves of blank tile
move(State, Next) :-
nth0(Pos, State, 0),              % find position of 0
move_blank(Pos, NewPos),          % get new blank position
swap(State, Pos, NewPos, Next).   % swap positions

% Valid moves for the blank
move_blank(Pos, NewPos) :- NewPos is Pos - 3, NewPos >= 0.       % up
move_blank(Pos, NewPos) :- NewPos is Pos + 3, NewPos < 9.        % down
move_blank(Pos, NewPos) :- Pos mod 3 =\= 0, NewPos is Pos - 1.   % left
move_blank(Pos, NewPos) :- Pos mod 3 =\= 2, NewPos is Pos + 1.   % right

% Swap two positions in a list
swap(List, I, J, Result) :-
nth0(I, List, ElemI),
nth0(J, List, ElemJ),
replace(List, I, ElemJ, Temp),
replace(Temp, J, ElemI, Result).

replace([_|T], 0, X, [X|T]).
replace([H|T], I, X, [H|R]) :-
I > 0, I1 is I - 1,
replace(T, I1, X, R).

% ---------- Best First Search ----------
```

```
best_first(Start, Path) :-
h(Start, H),
search([[Start, [], H]], Path).

search([[State, Path, _]|_], FinalPath) :-
goal(State),
reverse([State|Path], FinalPath),
write('Goal reached!'), nl.

search([[State, Path, _]|Rest], FinalPath) :-
findall([Next, [State|Path], Hn],
(move(State, Next),
\+ member(Next, [State|Path]),
h(Next, Hn)),
Children),
append(Rest, Children, NewOpen),
sort(2, @=<, NewOpen, SortedOpen),
search(SortedOpen, FinalPath).

% ---------- Sample Run ----------
% ?- best_first([1,2,3,4,0,5,6,7,8], Path).
```

QUERY:?- best_first([1,2,3,4,0,5,6,7,8], Path).

OUTPUT: Goal reached!

```
Path = [
[1,2,3,4,0,5,6,7,8],
[1,2,3,4,5,0,6,7,8],
[1,2,3,4,5,6,0,7,8],
[1,2,3,4,5,6,7,0,8],
[1,2,3,4,5,6,7,8,0]
].
```

PROGRAM-6

OBJECTIVE: PROGRAM TO SOLVE ROBOT TRAVERSAL PROBLEM USING MEANS END ANALYSIS

```
% --- MAIN PLANNER ---
plan(State, Goals, [], State) :-
satisfied(State, Goals).
plan(State, Goals, Plan, FinalState) :-
append(PrePlan, [Action | PostPlan], Plan),
select_goal(State, Goals, Goal),
achieves(Action, Goal),
can(Action, Conditions),
plan(State, Conditions, PrePlan, MidState1),
apply(MidState1, Action, MidState2),
plan(MidState2, Goals, PostPlan, FinalState).

% --- HELPER PREDICATES ---
satisfied(_, []).
satisfied(State, [Goal | RestGoals]) :-
member(Goal, State),
satisfied(State, RestGoals).

select_goal(State, Goals, Goal) :-
member(Goal, Goals),
\+ member(Goal, State).

achieves(Action, Goal) :-
adds(Action, AddList),
member(Goal, AddList).

apply(State, Action, NewState) :-
deletes(Action, DelList),
delete_all(State, DelList, State1),
adds(Action, AddList),
append(AddList, State1, NewState).

% --- UTILITY PREDICATES ---
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

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

delete_all([], _, []).
delete_all([X|L1], L2, Diff) :-
member(X, L2), !,
delete_all(L1, L2, Diff).
delete_all([X|L1], L2, [X|Diff]) :-
delete_all(L1, L2, Diff).

% ============================================================
% ROBOT TRAVERSAL PROBLEM DEFINITION
% (Grouped correctly to avoid warnings)
```

```
% ============================================================

% --- Define Actions: can(Action, Preconditions) ---
:- discontiguous can/2, adds/2, deletes/2.

can(go(door, table),   [at(robot, door)]).
can(go(table, door),   [at(robot, table)]).
can(go(table, window), [at(robot, table)]).
can(go(window, table), [at(robot, window)]).
can(pickup(key),       [at(robot, table), at(key, table), hand_empty]).
can(unlock(door),      [at(robot, door), has(robot, key)]).

% --- Define Action Effects: adds(Action, AddList) ---
adds(go(door, table),   [at(robot, table)]).
adds(go(table, door),   [at(robot, door)]).
adds(go(table, window), [at(robot, window)]).
adds(go(window, table), [at(robot, table)]).
adds(pickup(key),       [has(robot, key)]).
adds(unlock(door),      [unlocked(door)]).

% --- Define Action Effects: deletes(Action, DeleteList) ---
deletes(go(door, table),   [at(robot, door)]).
deletes(go(table, door),   [at(robot, table)]).
deletes(go(table, window), [at(robot, table)]).
deletes(go(window, table), [at(robot, window)]).
deletes(pickup(key),       [at(key, table), hand_empty]).
deletes(unlock(door),      [locked(door)]).

% ============================================================
% PROBLEM INSTANCE
% ============================================================

initial_state([at(robot, window), at(key, table), locked(door), hand_empty]).
goal_state([unlocked(door)]).
```

OUTPUT

Query : initial_state(S), goal_state(G), plan(S, G, Plan, _).
G = [unlocked(door)],
Plan = [go(window,table), pickup(key), go(table,door), unlock(door)],
S = [at(robot,window), at(key,table), locked(door), hand_empty]

PROGRAM-7

OBJECTIVE: IMPLEMENTATION OF TRAVELING SALESMAN PROBLEM

```
% ============================================================
% TSP SOLVER USING THE NEAREST NEIGHBOR HEURISTIC
% ============================================================

% --- Main Predicate ---
nearest_neighbor([Start | OtherTowns], Route, Distance) :-
% Start the recursive route-finding from the first town.
nn_route(Start, OtherTowns, 0, TempDist, TempRoute),
% Complete the final leg of the journey back to the start.
last(TempRoute, LastTown),
distance(LastTown, Start, FinalLeg),
Distance is TempDist + FinalLeg,
Route = [Start | TempRoute].

% --- Recursive Route Finder ---
% nn_route(CurrentTown, Unvisited, DistSoFar, FinalDist, FinalPath)

% Base Case: No more towns to visit.
nn_route(_, [], Dist, Dist, []).

% Recursive Step: Find the nearest unvisited town and travel there.
nn_route(Current, Unvisited, DistSoFar, FinalDist, [Nearest | RestOfPath]) :-
find_nearest(Current, Unvisited, Nearest, DistToNearest),
remove(Nearest, Unvisited, Remaining),
NewDist is DistSoFar + DistToNearest,
nn_route(Nearest, Remaining, NewDist, FinalDist, RestOfPath).

% --- Helper to find the nearest town in a list ---
find_nearest(Current, [Town | Rest], Nearest, MinDist) :-
distance(Current, Town, Dist),
find_nearest_helper(Current, Rest, Town, Dist, Nearest, MinDist).

find_nearest_helper(_, [], Nearest, Dist, Nearest, Dist).
find_nearest_helper(Current, [Town | Rest], BestSoFar, DistSoFar, Nearest, MinDist) :-
distance(Current, Town, Dist),
( Dist < DistSoFar ->
find_nearest_helper(Current, Rest, Town, Dist, Nearest, MinDist)
;
find_nearest_helper(Current, Rest, BestSoFar, DistSoFar, Nearest, MinDist)
).

% --- Standard Utility Predicates ---
remove(X, [X|T], T).
remove(X, [H|T], [H|R]) :-
remove(X, T, R).

distance(X, Y, D) :- X @< Y, !, e(X, Y, D).
distance(X, Y, D) :- e(Y, X, D).

% last(List, LastElement) is a standard helper.
```

```
last([X], X).
last([_|T], X) :- last(T, X).

% ============================================================
% DATA: Graph edges defined by e(From, To, Distance)
% ============================================================
e(a,b,11). e(a,c,41). e(a,d,27). e(a,e,23). e(a,f,43). e(a,g,15). e(a,h,20).
e(b,c,32). e(b,d,16). e(b,e,21). e(b,f,33). e(b,g, 7). e(b,h,13).
e(c,d,25). e(c,e,49). e(c,f,35). e(c,g,34). e(c,h,21).
e(d,e,26). e(d,f,18). e(d,g,14). e(d,h,19).
e(e,f,31). e(e,g,15). e(e,h,34).
e(f,g,28). e(f,h,36).
e(g,h,19).
```

OUTPUT

Query : *nearest_neighbor*([a,b,c,d,e,f,g,h], Route, Distance).
Distance = 177,
Route = [a, b, g, d, f, e, h, c]

PROGRAM-8

OBJECTIVE- PROGRAM TO IMPLEMENT BREADTH FIRST SEARCH

ALGORITHM TO IMPLEMENT BREADTH FIRST SEARCH

Step 1: Enter the node to be found
Step 2: Create a variable called NODE-LIST and set it to the initial state
Step 3: Until a goal state is found or NODE-LIST is empty do:
Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.
For each way that each rule can match the state described in E do:
Apply the rule to generate a new state
If the new state is goal state, quit and return this state.
Step 3: Otherwise add the new state to the end of NODE-LIST.
Step 4: Print the output as the path traversed
Step 5: Exit

SOURCE CODE:

```
% ================================================================
% BREADTH-FIRST SEARCH (BFS) IMPLEMENTATION
% ================================================================

% --- Main Predicate ---
% solve(StartNode, GoalNode, Path).
solve(Start, Goal, Path) :-
% Start the search with a queue containing just the initial path: [Start]
bfs([[Start]], Goal, RevPath),
reverse(RevPath, Path). % Reverse the path for the correct order.

% --- The BFS Solver ---
% bfs(QueueOfPaths, GoalNode, SolutionPath).

% Base Case: The first path in the queue starts with the goal node.
bfs([[Goal | T] | _], Goal, [Goal | T]).

% Recursive Step: Expand the first path in the queue.
bfs([[Current | Path] | RestQueue], Goal, Solution) :-
% Find all new, valid paths by extending the current path by one step.
findall([Next, Current | Path],
(s(Current, Next), \+ member(Next, [Current | Path])),
NewPaths),
% Add the new paths to the END of the queue. This is the core of BFS.
append(RestQueue, NewPaths, NewQueue),
bfs(NewQueue, Goal, Solution).

% Fail Case: The queue is empty and the goal was not found.
bfs([], _, _) :-
write('No solution found.'), nl,
fail.
```

```
% --- Standard Utility Predicates ---
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

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

% ================================================================
% GRAPH DEFINITION
% s(Node1, Node2) means there is a connection from Node1 to Node2.
% ================================================================
s(a, b).
s(b, c).
s(c, e).
s(c, d).
s(e, f).
s(d, f).
s(f, g).
s(g, h).
% Add connections for a more connected graph
s(a, d).
s(b, f).
```

OUTPUT

Query : solve(a, e, Path).
Path = [a, b, c, e]

PROGRAM-9

OBJECTIVES: PROGRAM TO FIND OUT UNION AND INTERSECTION OF TWO LISTS.

ALGORITHM TO FIND OUT UNION OF TWO LISTS.

Step 1. Start
Step 2. If first list is empty return second list as the union of two lists and go to step 6.
Step 3. If head of first list is not a member of second list add this element in the union of both lists produced by calling union function with and go to step 5.
Step 4. If head of first list is a member of second list call the union function for tail of first list and second list
Step 5. Exit

ALGORITHM TO FIND OUT INTERSECTION OF TWO LISTS.

Step 1. Start
Step 2. If first list is empty return empty list as intersection of both lists.
Step 3. If head of first list is a member of second list then add the head to the list obtained by calling intersection function with tail of first list and second list and go to step 5.
Step 4. If head of first list is not a member of second list then call intersection function with tail of first list and second list.
Step 5. Exit

SOURCE CODE:

```
% ================================================================
% UNION AND INTERSECTION OF TWO LISTS
% ================================================================

% --- Union of two lists ---
% union(List1, List2, UnionResult).

% Base case: The union of an empty list and another list is the other list.
union([], L, L).

% Recursive step 1: If the head of the first list is in the second,
% it's a duplicate. Ignore it and find the union of the rest.
union([H | T], L, Result) :-
member(H, L), !,
union(T, L, Result).

% Recursive step 2: If the head is not in the second list,
% add it to the result and find the union of the rest.
union([H | T], L, [H | Result]) :-
union(T, L, Result).

% --- Intersection of two lists ---
% intersection(List1, List2, IntersectionResult).

% Base case: The intersection of an empty list and any other list is empty.
intersection([], _, []).

% Recursive step 1: If the head of the first list is in the second,
% keep it and find the intersection of the rest.
intersection([H | T], L, [H | Result]) :-
```

```
member(H, L), !,
intersection(T, L, Result).

% Recursive step 2: If the head is not in the second list,
% discard it and find the intersection of the rest.
intersection([_ | T], L, Result) :-
intersection(T, L, Result).


% --- Helper Predicate: member/2 ---
% member(Element, List).
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

OUTPUT:

Query: union([a,b,c],[c,d,e],X).
X = [a, b, c, d, e]

Query: Intersection([a,b,c],[c,d,e],X).
X = [c]

PROGRAM-10

OBJECTIVE: PROGRAM TO CALCULATE FACTORIAL OF A GIVEN NUMBER

ALGORITHM TO CALCULATE FACTORIAL OF A GIVEN NUMBER

Step 1. Start.
Step 2. If the number is 0 then return 1 and go to step 5.
Step 3. Call the function with a number one less than the original number.
Step 4. Return product of number received in step 3 and number itself as factorial.
Step 5. Exit.

SOURCE CODE:

```
% ===================================================================
% FACTORIAL OF A GIVEN NUMBER
% ===================================================================

% --- Main Predicate ---
% fact (Number, Factorial).

% Base case: The factorial of 0 is 1.
fact (0, 1).

% Recursive step: The factorial of N is N * factorial(N-1).
fact (N, F) :-
N > 0,            % Ensure the number is positive.
N1 is N - 1,      % Use 'is' for arithmetic.
fact(N1, F1),     % Use 'fact' (lowercase) for the recursive call.
F is N * F1.      % Use 'is' for arithmetic.
```

OUTPUT-

Query: fact (5, X).
X=120

PROGRAM-11

OBJECTIVES: PROGRAM TO FLATTEN A LIST

ALGORITHM TO FLATTEN A LIST

Start
STEP 1: Obtain the given list as L.
STEP 2: Let H be the Head abd T be the Tail of the list L.
STEP 3: Let list be the null list then, The resulted list is null list and cut.
STEP 4: If H is head of the list and an atom, then write it to the resulted list and cut.
STEP 5: Else divide H the list head into two parts H1 and T1
And write H1 to the resulted list and repeat from step 2 for tail T1.
STEP 6: Repeat from step 2 for the tail T of the list.
STEP 7: Print the resulted list and Exit.
Stop.

SOURCE CODE;

```
% ================================================================
% FLATTEN A NESTED LIST
% ================================================================
% --- Main Predicate ---
% flatten(NestedList, FlatList).
% Base Case: Flattening an empty list results in an empty list.
flatten([], []).

% Recursive Step 1: If the head of the list (H) is itself a list.
flatten([H|T], FlatList) :-
is_list(H), !,        % Check if H is a list.
flatten(H, FlatH),    % Flatten the head.
flatten(T, FlatT),    % Flatten the tail.
append(FlatH, FlatT, FlatList). % Append the results.

% Recursive Step 2: If the head of the list (H) is not a list (it's an element).
flatten([H|T], [H|FlatT]) :-
flatten(T, FlatT).    % Keep the element and flatten the rest of the list.
% --- Standard Utility Predicate: append/3 ---
% append(List1, List2, ResultingList).
append([], L, L).
append([X|L1], L2, [X|L3]) :-
append(L1, L2, L3).
```

OUTPUT
Query: `flatten([a, [b, c, d], m, [[n]]], Result).`
Result = [a, b, c, d, m, n]

PROGRAM-12

OBJECTIVE: PROGRAM TO SOLVE MONKEY BANANA PROBLEM

```
% ================================================================
% MONKEY AND BANANA PROBLEM SOLVER
% ================================================================

% --- Defines the valid moves in the Monkey and Banana world ---

% grasp: Monkey grabs the banana if on the box in the middle.
move(state(middle, onbox, middle, hasnot), grasp, state(middle, onbox, middle, has)).

% climb: Monkey climbs on the box.
move(state(P, onfloor, P, H), climb, state(P, onbox, P, H)).

% -- Concrete PUSH moves for our specific room --
move(state(at_window, onfloor, at_window, H), push, state(middle, onfloor, middle, H)).

% -- Concrete WALK moves for our specific room --
move(state(at_door, onfloor, B, H), walk, state(at_window, onfloor, B, H)).


% --- The Main Solver ---
% solve(StartState, Plan) finds a Plan to get the banana from the initial State.
solve(StartState, Plan) :-
solve_recursive(StartState, [StartState], Plan).

% solve_recursive(CurrentState, VisitedStates, Plan)

% Base case: If the current state is the goal, the rest of the plan is empty.
solve_recursive(state(_,_,_,has), _, []).

% Recursive step: Find a valid move, avoid visited states, and find the rest of the plan.
solve_recursive(State1, Visited, [Move | RestOfPlan]) :-
move(State1, Move, State2),
\+ member(State2, Visited),                    % Check if we've been here before
solve_recursive(State2, [State2 | Visited], RestOfPlan).   % Continue search from the new state

% --- Helper Predicate ---
% member/2: Standard list membership check.
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

OUTPUT-

Query : solve(state(at_door, onfloor, at_window, hasnot), Plan).
Plan = [walk, push, climb, grasp]

FREQUENTLY ASKED QUESTIONS

Unit I – Introduction to AI & Intelligent Agents

1. Define Artificial Intelligence and list its major application areas.
2. Explain the PEAS (Performance, Environment, Actuators, Sensors) description of task environments with one example.
3. Differentiate between Simple Reflex Agents and Model-Based Agents.
4. Compare informed and uninformed search strategies. Give one example of each.
5. Solve a 4-puzzle problem using Breadth-First Search and show the state expansion order.
6. Explain the working of A* search algorithm with an illustrative graph and cost table.

Unit II – Knowledge and Reasoning

7. Describe the syntax and semantics of Propositional Logic with examples.
8. Convert the following English statements into First-Order Logic: "All humans are mortal. Socrates is a human. Therefore, Socrates is mortal."
9. Explain the Resolution Principle for First-Order Logic and show a sample derivation.
10. What is Adversarial Search? Describe the Minimax Algorithm with a suitable game tree.
11. Demonstrate Alpha–Beta Pruning on a given Minimax tree and show which nodes are pruned.
12. Discuss the advantages and limitations of Knowledge-Based Agents over simple reflex agents.

Unit III – Uncertain Knowledge, Reasoning & Learning

13. Define Bayes' Theorem. A disease affects 1% of a population. A test is 95% sensitive and 90% specific. If a person tests positive, calculate the probability that they actually have the disease.
14. Explain different types of Probabilistic Reasoning used in AI.
15. Describe the concept of Markov Decision Process and its relevance in uncertain reasoning.
16. Explain Learning Decision Trees. Give an example using the ID3 algorithm.
17. Differentiate between Passive and Active Reinforcement Learning.
18. Write short notes on Neural Networks and their use in knowledge representation.

Unit IV – Expert Systems

19. Define an Expert System. List and explain its key features.
20. Describe the architecture/organization of an Expert System with a neat diagram.
21. What are the main knowledge representation techniques used in Expert Systems?
22. Explain the working of the MYCIN expert system and highlight its strengths and weaknesses.
23. Describe the production rules approach to knowledge representation with an example.
24. Discuss the role of the Inference Engine in an expert system and the steps it follows during reasoning.
25. Compare Expert Systems with conventional software systems, focusing on knowledge base and reasoning.