KALASALINGAM
ACADEMY OF RESEARCH AND EDUCATION
(DEEMED TO BE UNIVERSITY)
Under sec. 3 of UGC Act 1956. Accredited by NAAC with "A" Grade
Anand Nagar, Krishnankoil - 626126. Srivilliputtur (Via), Virudhunagar (Dt), Tamil Nadu | info@kalasalingam.ac.in | www.kalasalingam.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

ARTIFICIAL INTELLIGENCE

(212INT2308)

PRACTICAL LAB MANUAL

Academic Year 2024-2025

EVEN Semester

Bachelor of Technology in Information Technology

Prepared by

A.BARANIDHARAN, Assistant Professor / IT

LIST OF EXPERIMENTS

1. Study of Prolog.
2. Write simple fact for the statements using PROLOG.
3. Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.
4. WAP to implement Factorial, Fibonacci of a given number.
5. Write a program to solve 4-Queen problem.
6. Write a program to solve 8 queens problem
7. Write a program to solve traveling salesman problem.
8. Write a program to solve water jug problem using LISP
9. Solve any problem using depth first search.
10. Solve any problem using best first search.
11. Solve 8-puzzle problem using best first search
12. Solve Robot (traversal) problem using means End Analysis
13. Write a program to solve Number Guessing Game

INDEX

S.No	Practical's Name	Marks	Signature
1	Study of Prolog.		
2	Write simple fact for the statements using PROLOG.		
3	Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.		
4	WAP to implement Factorial, Fibonacci of a given number.		
5	Write a program to solve 4-Queen problem.		
6	Write a program to solve 8 queens problem		
7	Write a program to solve traveling salesman problem.		
8	Write a program to solve water jug problem using LISP		
9	Solve any problem using depth first search.		
10	Solve any problem using best first search.		
11	Solve 8-puzzle problem using best first search		
12	Solve Robot (traversal) problem using means End Analysis		
13	Write a program to solve Number Guessing Game		

EX NO. : 1 DATE:	**Study of Prolog**

Aim : To study about basics 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 Kowaiski at Edinburgh (on the theoretical side), Marrten 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 centred 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 ArtJIcia1 LanguageA1,) 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

Progmmming in PROIOG 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 inference control known as resolution.

FOR EXAIPLE:

a) FACTS:

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

b) RULES:

To represent the general rule for grandfather,
we write:grand f.gher(X2)
parent(X,
Y) parent(
Y,Z)
male(X)

c) QUERIES:

Given a database of facts and rules such as that above, we may make queries by typing after aquery a symbol'?' statements such as:
?-parent(X,sam) Xann
?grandfather(X,
Y)X=jo,
Y=sam

PROLOG IN DISGINING 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 mela-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So

a PROLOG 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.

Result : Thus the basics of prolog were studied.

EX NO. : 2 DATE:	**Write simple fact for the statements using PROLOG.**

Aim: Write simple fact for following:

a. Ram likes mango.
b. Seema is a girl.
c. Bill likes Cindy.
d. Rose is red.
e. John owns gold.

Program:

```
Clauses
likes(ram
,mango).
girl(seema).
red(rose).
likes(bill
,cindy).
owns(john
,gold).
```

Output:

```
Goal
querie
s
?-
likes(ram,What
).
What= mango
?-
likes(Who,cindy
).Who= cindy
?-
red(What).
What=
rose
?-
owns(Who,What
).
```

Who= john
What= gold.

Result: Thus the simple facts for the statements given using prolog were written successfully.

EX NO. : 3 DATE:	**Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.**

Aim: To Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

Program:

Production rules:
Arithmetic:

c_to_f ⟶ f is c * 9 / 5 +32
freezing ⟶ f < = 32

Rules:

```
c_to_f(C,F) :-
F is C * 9 / 5 +
32.freezing(F)
:-
F =< 32.
```

Output:
Queries:

```
?-
c_to_f(100,X).
X = 212
Yes
?- freezing(15)
.Yes
?-
freezing(45).
No
```

Result: Thus the predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing was written successfully.

EX NO. : 4 DATE:	**WAP to implement Factorial, Fibonacci of a given number.**

Aim: To implement factorial, Fibonacci of a given number with WAP

Program:

Factorial:

```
factorial(0,1).

factorial(N,F) :-
  N>0,
  N1 is N-1,
  factorial(N1,F
  1),F is N *
  F1.
```

Output:
Goal:
?-
factorial(4,X).
X=24

Fibonacci:

```
fib(0, 0).
fib(X, Y) :- X > 0, fib(X,
Y, _).fib(1, 1, 0).
fib(X, Y1,
 Y2) :-X > 1,
 X1 is X - 1,
 fib(X1, Y2,
 Y3),Y1 is Y2
 + Y3.
```

Output:
Goal:
?-
fib(10,X).
X=55

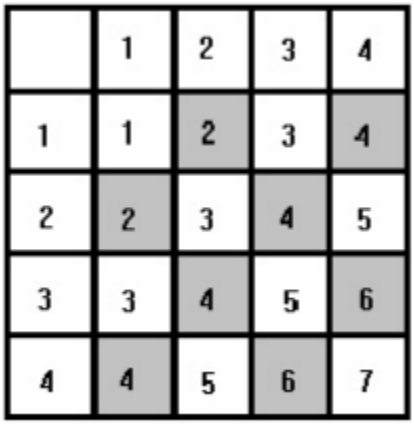
Result : Thus the factorial, Fibonacci of a given number were implemented successfully using WAP

EX NO. : 5 DATE:	**Write a program to solve 4-Queen problem.**

Aim: To Write a program to solve 4-Queen problem.

Program:

In the 4 Queens problem the object is to place 4 queens on a chessboard in such a way that noqueens can capture a piece. This means that no two queens may be



placed on the same row, column, or diagonal.

The n Queens Chessboard.

```
domains
queen = q(integer,
integer)queens =
queen*
freelist = integer*
board = board(queens, freelist, freelist, freelist,
freelist)predicates
nondeterm placeN(integer, board, board)
nondeterm place_a_queen(integer, board,
board)nondeterm nqueens(integer)
nondeterm makelist(integer, freelist)
nondeterm findandremove(integer, freelist,
freelist)nextrow(integer, freelist, freelist)
clauses
nqueens(N):-
makelist(N,L),
Diagonal=N*2
-1,
makelist(Diagonal,LL),
placeN(N,board([],L,L,LL,LL),
```

```
Final),write(Final).
placeN(_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-!.
```

```
placeN(N,Board1,Result):-
place_a_queen(N,Board1,Board2),
placeN(N,Board2,Result).
place_a_queen(N,
board(Queens,Rows,Columns,Diag1,
Diag2),
board([q(R,C)|Queens],NewR,NewC,NewD1,Ne
wD2)):-nextrow(R,Rows,NewR),
findandremove(C,Columns,NewC),
D1=N+C-
R,findandremove(D1,Diag1,NewD1),
D2=R+C-
1,findandremove(D2,Diag2,NewD2).
findandremove(X,[X|Rest],Rest).
findandremove(X,[Y|Rest],[Y|Tail]):-
findandremove(X,Rest,Tail).
makelist(1,[1]).
makelist(N,[N|Rest]) :-
N1=N-
1,makelist(N1,Rest).
nextrow(Row,[Row|Rest],

Rest).Output:

Goal:
?-nqueens(4),nl.
board([q(1,2),q(2,4),q(3,1),q(4,3),[],[],[7,4,1]
,[7,4,1])
yes
```

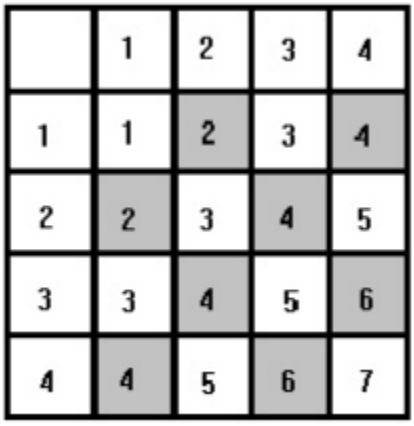
Result: Thus the program to solve 4-Queens problem was written and executed successfully.

EX NO. : 6 DATE:	**Write a program to solve 8-Queen problem.**

Aim: To Write a program to solve 8-Queen problem using prolog.

Program:

In the 8 Queens problem the object is to place 8 queens on a chessboard in such a way that noqueens can capture a piece. This means that no two queens may be



placed on the same row, column, or diagonal.

```
queen(Row, Col).
   Row = R ;  % Same row
  Col = C ;  % Same column
  Diff is abs(Row - R),
  Col = C + Diff ;  % Diagonal attack (positive slope)
  Col = C - Diff.  % Diagonal attack (negative slope).
% Ensures no queens on the board attack each other
safe(Queens) :-
  \+ attacking(Q1, Q2) for Q1 <- Queens, Q2 <- Queens, Q1 \= Q2.
% Places a queen in a specific column (Col) considering safe positions from previous columns (Queens)
place_queen(Queens, Col, NewQueens) :-
  between(1, 8, Row),  % Try all rows in the column
  safe([queen(Row, Col) | Queens]),  % Check if safe with existing queens
  append(Queens, [queen(Row, Col)], NewQueens).

% Solves the problem recursively. Tries placing queens in each column and checks for a safe configuration
```

```
solve(Queens) :-
  length(Queens, 8),  % Ensure 8 queens are placed
  safe(Queens).

solve_queens :-
  solve(Queens),
  write('Solution:'), nl,
  % Print the solution (one way to represent the board)
  maplist(write_queen, Queens), nl.

write_queen(queen(Row, _)) :-
  write('Row: '), write(Row), nl.

road("gordon","kansas_city",
130).
route(Town1,Town2,Distanc
e):-
road(Town1,Town2,Distanc
e).
route(Town1,Town2,Distanc
e):- road(Town1,X,Dist1),
route(X,Town2,Dist2),
Distance=Dist1+Dist2,!.
Output:
Goal:
route("tampa", "kansas_city", X),
write("Distance from Tampa to Kansas City is ",X),nl.

Distance from Tampa to Kansas City
is 320X=320
```

Result : Thus the travelling salesmen problem using prolog was implemented successfully.

EX NO. : 7 DATE:	**Write a program to solve traveling salesman problem.**

Aim : To Write a program to solve Travelling salesmen problem using prolog.

Program :

```
% Define facts about distances between cities
road(city1, city2, distance)  % Example: road(a, b, 3).

% Check if there's a direct road between two cities
distance(Start, End, Distance) :-
  road(Start, End, Distance).

% Recursive predicate to find a possible route
get_route(Start, End, Waypoints, DistanceAcc, Visited, TotalDistance) :-
  % Check if the path ends at the starting city
  distance(Start, End, Distance),
  reverse([End|Waypoints], Visited),
  TotalDistance is DistanceAcc + Distance.

% Explore paths with intermediate cities
get_route(Start, End, Waypoints, DistanceAcc, Visited, TotalDistance) :-
  distance(Start, NextCity, Distance),  % Go to next city
  \+ member(NextCity, Visited),         % Avoid revisiting cities
  NewVisited = [NextCity|Visited],      % Update visited list
  get_route(NextCity, End, Waypoints, DistanceAcc + Distance, NewVisited,
TotalDistance).

% Find the shortest route (incomplete - needs optimization)
shortest_route(Start, End, Path, Distance) :-
  get_route(Start, End, [], 0, [], TotalDistance),
  % This part needs improvement to find the shortest path among all possibilities
  Distance = TotalDistance,
  Path = [Start|TotalDistance].  % Replace with actual path list
```

Result: Thus a program to solve Travelling salesmen problem using prolog was implemented successfully.

EX NO. : 8 DATE:	**Write a program to solve water jug problem using LISP**

AIM : To Write a program to solve water jug problem using LISP

Program:

```
(defun water-jug (jug1-size jug2-size target)
  "Solves the Water Jug Problem to reach target amount in jug2.

  Args:
     jug1-size: Capacity of jug 1.
     jug2-size: Capacity of jug 2.
     target: Target amount of water to reach in jug 2.

  Returns:
     A list of actions (fill, empty, pour jug1 to jug2) required
     to reach the target or nil if not possible."

  ;; Define states as a list: (jug1-water jug2-water)
  (defun explore (state actions)
    "Explores possible actions from a given state.

    Args:
       state: A list representing water level in each jug (jug1-water jug2-water).
       actions: A list of actions taken so far.

    Returns:
       A list of lists, where each sublist represents a new state and actions taken."

    (let ((jug1-water (car state))
          (jug2-water (cadr state)))
      (append
       ;; Fill jug1
       (if (< jug1-water jug1-size)
         (list (cons jug1-size jug2-water) (cons (append actions '(fill jug1))))
         nil)
       ;; Empty jug1
       (if (> jug1-water 0)
```

```
      (list (cons 0 jug2-water) (cons (append actions '(empty jug1))))
      nil)
    ;; Pour jug1 to jug2 (ensure jug2 has space)
    (if (and (> jug1-water 0) (< (+ jug2-water jug1-water) jug2-size))
      (list (cons (- jug1-water (+ jug2-water jug1-water)) jug1-water)
          (cons (append actions '(pour jug1 jug2))))
      nil))))
(defun solve (jug1-size jug2-size target)
  "Solves the water jug problem iteratively.

  Args:
     jug1-size: Capacity of jug 1.
     jug2-size: Capacity of jug 2.
     target: Target amount of water to reach in jug 2.

  Returns:
     A list of actions required to reach the target or nil if not possible."

  (let ((frontier (list (cons 0 0) nil)))  ; Initial state with both jugs empty
    (loop for state in frontier
        unless (null state)
        do (let ((new-states (explore state nil)))
           (setq frontier (append (remove state frontier) new-states)))))
  (cond ((null frontier) nil)
      ((equal (car (car frontier)) target) (cdr (car frontier)))))
(solve jug1-size jug2-size target))
```

Result: Thus a program to solve water jug problem using LISP was written and executed successfully.

EX NO. : 9 DATE:	**Solve any problem using depth first search.**

Aim : To Solve a Problem using depth first search using prolog.

Program :

```
% Define facts or rules representing your problem space
% (Replace this with your specific problem representation)
problem_fact(State1, State2).  % Example: connected(city1, city2).

% Function to check if a state is the goal state
goal(State) :-
  % (Replace this with your goal condition)
  State = some_goal_state.

% Function to explore possible next states from a given state
get_next_states(State, NextStates) :-
  % (Replace this with logic to find connected/reachable states)
  problem_fact(State, NextState1),
  NextStates = [NextState1].  % Add more states if applicable

% Depth-first search predicate
depth_first_search(StartState, SolutionPath) :-
  depth_first_search(StartState, [], SolutionPath).

depth_first_search(State, Visited, SolutionPath) :-
  goal(State),
  reverse(Visited, ReversedVisited),
  append([State], ReversedVisited, SolutionPath),
  !.

depth_first_search(State, Visited, SolutionPath) :-
  \+ member(State, Visited),
  get_next_states(State, NextStates),
  NewVisited = [State | Visited],
  findall(Solution, (member(NextState, NextStates), depth_first_search(NextState,
NewVisited, Solution)), PartialSolutions),
  append(PartialSolutions, [], FlattenedSolutions),
```

```
  member(MostRecentSolution, FlattenedSolutions),
  SolutionPath = [State | MostRecentSolution].
```

Result : Thus a Problem using depth first search using prolog was executed successfully.

EX NO. : 10 DATE:	**Solve any problem using best first search.**

Aim : To Solve a Problem using best first search using prolog.

Program:

```
% Define facts or rules representing your problem space
% (Replace this with your specific problem representation)
problem_fact(State1, State2, Cost).  % Example: connected(city1, city2, Distance).

% Function to check if a state is the goal state
goal(State) :-
  % (Replace this with your goal condition)
  State = some_goal_state.

% Function to estimate the cost of reaching the goal from a state (heuristic)
estimate_cost(State, Cost) :-
  % (Replace this with your heuristic function)
  % This function should estimate the remaining cost to reach the goal from a state.

% Function to explore possible next states and their costs from a given state
get_next_states(State, NextStatesWithCosts) :-
  % (Replace this with logic to find connected/reachable states and their costs)
  problem_fact(State, NextState1, Cost1),
  NextStatesWithCosts = [(NextState1, Cost1)].  % Add more states with costs if
applicable

% Best-first search predicate with priority queue
best_first_search(StartState, SolutionPath) :-
  empty_queue(Open),
  insert_by_priority(Open, (StartState, 0, nil), estimate_cost(StartState, _)),
  best_first_search(Open, [], SolutionPath).

best_first_search(empty_queue(_), _, fail).
best_first_search(Open, Visited, SolutionPath) :-
  remove_by_priority(Open, (State, Cost, Parent), _),
  ( goal(State) ->
    reverse([State|Visited], ReversedVisited),
```

```
    append([(State, Cost)], ReversedVisited, SolutionPath)
  ;
    \+ member(State, Visited),
    get_next_states(State, NextStatesWithCosts),
    NewVisited = [State | Visited],
    forall(member((NextState, NextCost), NextStatesWithCosts),
           ( TotalCost is Cost + NextCost,
             \+ member((NextState, _, _), Visited),
             insert_by_priority(Open, (NextState, TotalCost, State),
estimate_cost(NextState, _))
           )
    ).
```

Result : Thus a Problem using depth first search using prolog was executed successfully.

EX NO. : 11 DATE:	**Solve 8-puzzle problem using best first search**

Aim: To Solve 8-Puzzle Problem using best first search using prolog.

Program :

```
% Define the 8-puzzle state representation as a list of numbers
% 0 represents the empty tile

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

% Successor function to generate reachable states by moving the empty tile
successor(State, NextState) :-
  move(State, Move, NextState),
  not(State = NextState).  % Avoid returning the same state

% Define moves to change the position of the empty tile
move([X1,X2,X3, Empty, X5, X6, X7, X8, X9], up, [X1,X2,X3, X9, X5, X6, X7, Empty, X8]) :-
  Empty \= X1.
move([X1,X2,X3, X4, Empty, X6, X7, X8, X9], down, [X1,X2,X3, Empty, X4, X6, X7, X8, X9]) :-
  Empty \= X7.
move([X1, Empty, X3, X4, X5, X6, X7, X8, X9], left, [Empty, X1, X3, X4, X5, X6, X7, X8, X9]) :-
  Empty \= X2.
move([X1, X2, X3, Empty, X5, X6, X7, X8, X9], right, [X1, X2, Empty, X4, X5, X6, X7, X8, X9]) :-
  Empty \= X4.

% Manhattan distance heuristic - calculates the number of moves each tile needs to reach its goal position
heuristic(State, Cost) :-
  findall((X, PosX, PosY, GoalX, GoalY), (member(X, State), nth(Pos, State, PosX), nth(X, [1,2,3,4,5,6,7,8], GoalY), nth(PosY, State, PosX)), DistanceList),
  maplist(sum, DistanceList, XYSums),
  sum(XYSums, Cost).

% Best-first search predicate with priority queue
```

```
best_first_search(StartState, SolutionPath) :-
  empty_queue(Open),
  insert_by_priority(Open, (StartState, 0, nil), heuristic(StartState, Cost)),
  best_first_search(Open, [], SolutionPath).

best_first_search(empty_queue(_), _, fail).
best_first_search(Open, Visited, SolutionPath) :-
  remove_by_priority(Open, (State, Cost, Parent), _),
  ( goal(State) ->
    reverse([State|Visited], ReversedVisited),
    append([(State, Cost)], ReversedVisited, SolutionPath)
  ;
    \+ member(State, Visited),
    findall(NextState, (successor(State, NextState), \+ member(NextState, Visited)),
NextStates),
    NewVisited = [State | Visited],
    forall(member(NextState, NextStates),
          ( NextCost is heuristic(NextState, _) + Cost,
            insert_by_priority(Open, (NextState, NextCost, State), heuristic(NextState, _))
          )
  ).

% Additional predicates (replace placeholders with your chosen library)
empty_queue(_).
insert_by_priority(_, _, _).
remove_by_priority(_, _, _).

% Goal state (all numbers in order with empty tile at the end)
goal([1,2,3,4,5,6,7,8,0]).
```

Result : Thus 8-Puzzle Problem using depth first search using prolog was executed successfully.

EX NO. : 12 DATE:	**Solve Robot (traversal) problem using means End Analysis**

Aim: To Solve Robot (traversal) problem using means End Analysis

Program:

```
position(X, Y).
orientation(Direction).
holding(Object).
% Sample actions (replace with your specific logic)
move_forward(state(position(X, Y), orientation(Dir), Holding),
            state(position(X + 1, Y), Dir, Holding)).
turn_left(state(Pos, orientation(Dir)), state(Pos, new_orientation(Dir))).
% ... define turn_right, pick_up, drop, etc.
goal_state(state(position(3, 2), orientation(east), holding(key))).
object_at(1, 1, key).
wall_at(2, 1).  % Example obstacle
plan(InitialState, GoalState, Plan) :-
plan(InitialState, GoalState, [], Plan).
plan(State, State, Plan, Plan).
plan(State, GoalState, Visited, Plan) :-
 \+ member(State, Visited),
 findall((Action, NextState),
       (  member(Action, [move_forward/2, turn_left/2]),  % Replace with your actions
          Action(State, NextState),
          \+ member(NextState, Visited)
       ), Actions),
 maplist(append, [(Action, [])], Actions, SubPlans),
 findall(SubPlan, (member(SubPlan, SubPlans), plan(NextState, GoalState, SubPlan)),
```

```
PartialPlans),
  append(PartialPlans, [], FlattenedPlans),
  member(MostRecentPlan, FlattenedPlans),
  append([[State, Action]], MostRecentPlan, Plan),
  !.
plan(_, _, Visited, _) :-
  write('No plan found!'), nl,
  fail.
```

Result: Thus the Robot (traversal) problem using means End Analysis was executed successfully.

EX NO. : 13 DATE:	**Write a program to solve Number Guessing Game**

Aim: To Write a program to solve Number Guessing Game

Program :

```
% Define the secret number (replace with your desired number)
secret_number(5).

% Function to check if the guess is correct, higher, or lower
guess_result(Guess, Correct) :-
  secret_number(Secret),
  Guess = Secret,
  Correct = 'correct'.

guess_result(Guess, Higher) :-
  secret_number(Secret),
  Guess < Secret,
  Higher = 'higher'.

guess_result(Guess, Lower) :-
  secret_number(Secret),
  Guess > Secret,
  Lower = 'lower'.

% Function to play the game interactively
play :-
  repeat,
    write('Enter your guess: '),
    read(Guess),
    guess_result(Guess, Result),
    ( Result = 'correct' ->
      write('You guessed it!'), nl;
      write('Your guess is '), write(Result), nl
    ),
  not(Result = 'correct').
```

Result : Thus a Program to solve Number Guessing Game was written and executed successfully.