BAJAJ INSTITUTE OF TECHNOLOGY, WARDHA

Department of Computer Engineering

Lab Manual

Artificial Intelligence Lab

(BTCOL707)

Year/Semester: 4th Yr./VIIth Sem

Prepared By

Mr. Vikas Palekar

Assistant Professor

Department of Computer Engineering

2024-25 (Odd)

EXPERIMENT LIST

Experiment 1: Study of PROLOG programming language. Write a sample program using PROLOG.

Theory: Prolog stands for programming in logic. Prolog is a declarative language, which means that a program consists of data based on the facts and rules (Logical relationship) rather than computing how to find a solution. 'Query' is a sequence of one or more goals. These goals are entered by the user at the prompt. In Prolog, the program contains a sequence of one or more clauses. The clauses can run over many lines. Using a dot character, a clause can be terminated. This dot character is followed by at least one 'white space' character. The clauses are of two types: facts and rules. Examples of facts are as follows: likes(A, prolog), dexterous(monkey). Rules are specified in the form:

head:- t1, t2, t3,….., tk. Where k>=1

A rule will be read as 'if t1, t2, t3,…., tk are all true, head is true'. :- is known as the clause neck.

Program:

```
/*
Family Relationship in Prolog (SWI-Prolog or GNU Prolog).
Usage:
?- uncle(X,Y).
?- grandparent(X,Y).
?- wife(X,Y).
?- parent(X,jim).
*/

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: Attach your screenshot

Conclusion: In this way, we understood the syntax and semantics of the Prolog language.

Experiment 2: Write a program to solve 8 queen problem.

Theory: This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board. This can be formulated as a problem as follows:

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdot \cdots \cdot 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions are easy to find.

Program:

```
use_module(library(lists)).
/*
N-queens problem solved in Prolog (SWI-Prolog or GNU Prolog).
Usage: n_queen(4,X).
*/

n_queen(N, Solution) :-
      %create a list of N dummy variabiles
      length(Solution, N),

      queen(Solution, N). %search for a configuration of N queens


%returns a list of integer from K to N included es up2N(1,3,X) X = [1,2,3]
up2N(N,N,[N]) :-!.
up2N(K,N,[K|Tail]) :- K < N, K1 is K+1, up2N(K1, N, Tail).


queen([],_). %No queens is a solution for any N queens problem. All queens
are in a safe position.

queen([Q|Qlist],N) :-

      queen(Qlist, N), %first we solve the subproblem

      %we then generate all possible positions for queen Q
      up2N(1,N,Candidate_positions_for_queenQ),

      %we pick one of such position
      member(Q, Candidate_positions_for_queenQ),

      %we check whether the queen Q is safe
      check_solution(Q,Qlist, 1).
```

```
check_solution(_,[], _).

check_solution(Q,[Q1|Qlist],Xdist) :-
      Q =\= Q1, %not on the same row
      Test is abs(Q1-Q),
      Test =\= Xdist, %diagonal distance
      Xdist1 is Xdist + 1,
      check_solution(Q,Qlist,Xdist1).
```

Output: Attach your screenshot

Conclusion: In this way, we understood the N Queen problem and implemented it using Prolog.

Experiment 3: Write a Prolog program to solve cryptarithmetic problem (SEND+MORE=MONEY).

Theory: In a cryptarithmetic problem, each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed.

SEND	DONALD	CROSS
+MORE	+GERALD	+ROADS
.....	.....	.....
MONEY	ROBERT	DANGER

Fig. *Some Cryptarithmetic Problems*

Program:

```
/*
A program for cryptoarithmetic puzzles (SWI-Prolog).
Usage: sum([0,S,E,N,D],[0,M,O,R,E],[M,O,N,E,Y]).
*/

%    Solving cryptarithmetic puzzles

sum(N1, N2, N)  :-                    % Numbers represented as lists of
digits
  sum1( N1, N2, N,
      0, 0,                          % Carries from right and to left both
0
        [0,1,2,3,4,5,6,7,8,9], _).    % All digits available

sum1( [], [], [], C, C, Digits, Digits).

sum1( [D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs)  :-
  sum1( N1, N2, N, C1, C2, Digs1, Digs2),
  digitsum( D1, D2, C2, D, C, Digs2, Digs).

digitsum( D1, D2, C1, D, C, Digs1, Digs)  :-
  del_var( D1, Digs1, Digs2),         % Select an available digit for D1
  del_var( D2, Digs2, Digs3),         % Select an available digit for D2
  del_var( D, Digs3, Digs),           % Select an available digit for D
  S  is  D1 + D2 + C1,
  D  is  S mod 10,                    % Reminder
  C  is  S // 10.                     % Integer division

del_var( A, L, L) :-
  nonvar(A), !.         .             % A already instantiated

del_var( A, [A|L], L).

del_var( A, [B|L], [B|L1])  :-
  del_var(A, L, L1).


% Some puzzles

puzzle1( [D,O,N,A,L,D],
         [G,E,R,A,L,D],
```

```
         [R,O,B,E,R,T] ).

puzzle2( [0,S,E,N,D],
         [0,M,O,R,E],
         [M,O,N,E,Y] ).
```

<u>Output:</u> Attach your screenshot

Conclusion: In this way, we have studied and implemented cryptarithmetic problem.

Experiment 4: Solve Sudoku using Prolog.

Theory: A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the prefilled squares have a domain consisting of a single value. In addition, there are 27 different Alldiff constraints: one for each row, column, and box of 9 squares.

Alldiff (A1,A2,A3,A4,A5,A6, A7, A8, A9)

Alldiff (B1,B2,B3,B4,B5,B6,B7,B8,B9)

. . .

Alldiff (A1,B1,C1,D1,E1, F1,G1,H1, I1)

Alldiff (A2,B2,C2,D2,E2, F2,G2,H2, I2)

. . .

Alldiff (A1,A2,A3,B1,B2,B3,C1,C2,C3)

Alldiff (A4,A5,A6,B4,B5,B6,C4,C5,C6)

Program:

```
% to run the code in SWI-Prolog, do
%        ?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).


:- use_module(library(clpfd)).
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).

problem(0, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).

problem(1, P) :-
```

```
        P = [[1,_,_,8,_,4,_,_,_],
             [_,2,_,_,_,_,4,5,6],
             [_,_,3,2,_,5,_,_,_],
             [_,_,_,4,_,_,8,_,5],
             [7,8,9,_,5,_,_,_,_],
             [_,_,_,_,_,6,2,_,3],
             [8,_,1,_,_,_,7,_,_],
             [_,_,_,1,2,3,_,8,_],
             [2,_,5,_,_,_,_,_,9]].


problem(2, P) :-
        P = [[_,_,2,_,3,_,1,_,_],
             [_,4,_,_,_,_,_,3,_],
             [1,_,5,_,_,_,_,8,2],
             [_,_,_,2,_,_,6,5,_],
             [9,_,_,_,8,7,_,_,3],
             [_,_,_,_,4,_,_,_,_],
             [8,_,_,_,7,_,_,_,4],
             [_,9,3,1,_,_,_,6,_],
             [_,_,7,_,6,_,5,_,_]].

problem(3, P) :-
        P = [[1,_,_,_,_,_,_,_,_],
             [_,_,2,7,4,_,_,_,_],
             [_,_,_,5,_,_,_,_,4],
             [_,3,_,_,_,_,_,_,_],
             [7,5,_,_,_,_,_,_,_],
             [_,_,_,_,_,9,6,_,_],
             [_,4,_,_,_,6,_,_,_],
             [_,_,_,_,_,_,_,7,1],
             [_,_,_,_,_,1,_,3,_]].
```

Output: Attach your screenshot

Conclusion: In this way, we have understood and solved Sudoku problem using Prolog.

Experiment 5: Write a Prolog program to solve 3 Missionaries 3 Cannibals puzzle.

Theory: The **missionaries and cannibals** problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place.

Program:

```
% to run the code in SWI-Prolog, do
%        ?- find.

% Represent a state as [CL,ML,B,CR,MR]
start([3,3,left,0,0]).
goal([0,0,right,3,3]).

legal(CL,ML,CR,MR) :-
      % is this state a legal one?
      ML>=0, CL>=0, MR>=0, CR>=0,
      (ML>=CL ; ML=0),
      (MR>=CR ; MR=0).

% Possible moves:
move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
      % Two missionaries cross left to right.
      MR2 is MR+2,
      ML2 is ML-2,
      legal(CL,ML2,CR,MR2).

move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
      % Two cannibals cross left to right.
      CR2 is CR+2,
      CL2 is CL-2,
      legal(CL2,ML,CR2,MR).

move([CL,ML,left,CR,MR],[CL2,ML2,right,CR2,MR2]):-
      %  One missionary and one cannibal cross left to right.
      CR2 is CR+1,
      CL2 is CL-1,
      MR2 is MR+1,
      ML2 is ML-1,
      legal(CL2,ML2,CR2,MR2).

move([CL,ML,left,CR,MR],[CL,ML2,right,CR,MR2]):-
      % One missionary crosses left to right.
      MR2 is MR+1,
      ML2 is ML-1,
      legal(CL,ML2,CR,MR2).

move([CL,ML,left,CR,MR],[CL2,ML,right,CR2,MR]):-
      % One cannibal crosses left to right.
      CR2 is CR+1,
      CL2 is CL-1,
      legal(CL2,ML,CR2,MR).
```

```
move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
      % Two missionaries cross right to left.
      MR2 is MR-2,
      ML2 is ML+2,
      legal(CL,ML2,CR,MR2).

move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
      % Two cannibals cross right to left.
      CR2 is CR-2,
      CL2 is CL+2,
      legal(CL2,ML,CR2,MR).

move([CL,ML,right,CR,MR],[CL2,ML2,left,CR2,MR2]):-
      %  One missionary and one cannibal cross right to left.
      CR2 is CR-1,
      CL2 is CL+1,
      MR2 is MR-1,
      ML2 is ML+1,
      legal(CL2,ML2,CR2,MR2).

move([CL,ML,right,CR,MR],[CL,ML2,left,CR,MR2]):-
      % One missionary crosses right to left.
      MR2 is MR-1,
      ML2 is ML+1,
      legal(CL,ML2,CR,MR2).

move([CL,ML,right,CR,MR],[CL2,ML,left,CR2,MR]):-
      % One cannibal crosses right to left.
      CR2 is CR-1,
      CL2 is CL+1,
      legal(CL2,ML,CR2,MR).

% Recursive call to solve the problem
path([CL1,ML1,B1,CR1,MR1],[CL2,ML2,B2,CR2,MR2],Explored,MovesList) :-
   move([CL1,ML1,B1,CR1,MR1],[CL3,ML3,B3,CR3,MR3]),
   not(member([CL3,ML3,B3,CR3,MR3],Explored)),

path([CL3,ML3,B3,CR3,MR3],[CL2,ML2,B2,CR2,MR2],[[CL3,ML3,B3,CR3,MR3]|Explo
red],[ [[CL3,ML3,B3,CR3,MR3],[CL1,ML1,B1,CR1,MR1]] | MovesList ]).

% Solution found
path([CL,ML,B,CR,MR],[CL,ML,B,CR,MR],_,MovesList):-
      output(MovesList).

% Printing
output([]) :- nl.
output([[A,B]|MovesList]) :-
      output(MovesList),
      write(B), write(' -> '), write(A), nl.

% Find the solution for the missionaries and cannibals problem find :-
   path([3,3,left,0,0],[0,0,right,3,3],[[3,3,left,0,0]],_).
```

Output: Attach your screenshot

Conclusion: In this way, we have studied and implemented 3 Missionaries and 3 Cannibals problem.

Experiment 6: Implement Monkey Banana problem using Prolog. Also, demonstrate the use of trace and notrace command of the Prolog.

Theory: The Monkey and Bananas Problem is stated as: A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the bananas. However. in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around. carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?

Program:

```
/*
Monkey-Banana problem solved in Prolog (SWI-Prolog or GNU Prolog).
Usage: can_reach(X,Y).
*/

in_room(bananas).
in_room(chair).
in_room(monkey).
dexterous(monkey).
tall(chair).
can_move(monkey,chair,bananas).
can_climb(monkey,chair).
can_reach(X,Y):-dexterous(X),is_close(X,Y).
is_close(X,Z):-can_climb(X,Y),under(Y,Z),tall(Y).
under(Y,Z):-in_room(X),in_room(Y),in_room(Z),can_move(X,Y,Z).
```

Output: Attach your screenshot

Conclusion: In this way, we have understood and solved Monkey Banana puzzle.

Experiment 7: Solve water jug problem using depth first search.

Theory: A Water Jug Problems is stated as: You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?
Another instance of the problem is:
You have three jugs, measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet. You can fill the jugs up or empty them out from one to another or onto the ground. You need to measure out exactly one gallon.

Program:

```
/*
Waterjug problem solved in Prolog (SWI-Prolog).
Usage: test_dfs(jugs, Moves).
*/

solve_dfs(State, History, []) :- final_state(State).
solve_dfs(State, History, [Move|Moves]) :-
    move(State, Move),
    update(State, Move, State1),
    legal(State1),
    not(member(State1, History)),
    solve_dfs(State1, [State1|History], Moves).

test_dfs(Problem, Moves) :-
    initial_state(Problem, State), solve_dfs(State, [State], Moves).

capacity(1, 10).
capacity(2, 7).

initial_state(jugs, jugs(0, 0)).
final_state(jugs(0, 6)).
%final_state(jugs(4, 0)).

legal(jugs(V1, V2)).

move(jugs(V1, V2), fill(1)) :- capacity(1, C1), V1 < C1, capacity(2, C2),
V2 < C2.
move(jugs(V1, V2), fill(2)) :- capacity(2, C2), V2 < C2, capacity(1, C1),
V1 < C1.

move(jugs(V1, V2), empty(1)) :- V1 > 0.
move(jugs(V1, V2), empty(2)) :- V2 > 0.
move(jugs(V1, V2), transfer(1, 2)).
move(jugs(V1, V2), transfer(2, 1)).

adjust(Liquid, Excess, Liquid, 0) :- Excess =< 0.
adjust(Liquid, Excess, V, Excess) :- Excess > 0, V is Liquid - Excess.

update(jugs(V1, V2), fill(1), jugs(C1, V2)) :- capacity(1, C1).
update(jugs(V1, V2), fill(2), jugs(V1, C2)) :- capacity(2, C2).
update(jugs(V1, V2), empty(1), jugs(0, V2)).
update(jugs(V1, V2), empty(2), jugs(V1, 0)).

update(jugs(V1, V2), transfer(1, 2),jugs(NewV1, NewV2)) :-
```

```
    capacity(2, C2),
    Liquid is  V1 + V2,
    Excess is Liquid - C2,
    adjust(Liquid, Excess, NewV2, NewV1).

update(jugs(V1, V2), transfer(2, 1),jugs(NewV1, NewV2)) :-
    capacity(1, C1),
    Liquid is  V1 + V2,
    Excess is Liquid - C1,
    adjust(Liquid, Excess, NewV1, NewV2).
```

Output: Attach your screenshot

Conclusion: In this way, we have studied and implemented water jug problem using DFS.

Experiment 8: Demonstrate recursion in Prolog using Tower of Hanoi problem.

Theory: Towers of Hanoi Problem is a famous puzzle to move N disks from the source peg/tower to the target peg/tower using the intermediate peg as an auxiliary holding peg. There are two conditions that are to be followed while solving this problem −

1. A larger disk cannot be placed on a smaller disk.
2. Only one disk can be moved at a time.

Program:

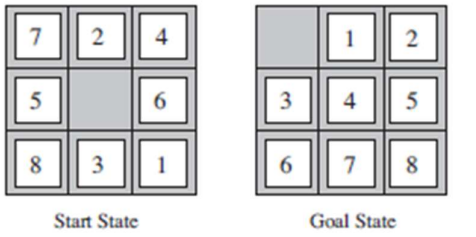
```
% to run the code in SWI-Prolog, do
%        ?- move(3,source,target,aux).

move(1,X,Y,_) :-
   write('Move top disk from '), write(X), write(' to '), write(Y), nl.
move(N,X,Y,Z) :-
   N>1,
   M is N-1,
   move(M,X,Z,Y),
   move(1,X,Y,_),
   move(M,Z,Y,X).
```

Output: Attach your screenshot

Conclusion: In this way, we have studied and understood recursion through Tower of Hanoi problem.

Experiment 9: Solve 8-puzzle problem using Prolog.

Theory: The 8-Puzzle problem is stated as: The 8-puzzle is a square tray in which are placed, eight square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.



A typical instance of the 8-puzzle.

Program:

```
% to run the code in SWI-Prolog, do
%        ?- ids.

ids :-
   start(State),
   length(Moves, N),
   dfs([State], Moves, Path), !,
   show([start|Moves], Path),
   format('~nmoves = ~w~n', [N]).

dfs([State|States], [], Path) :-
   goal(State), !,
   reverse([State|States], Path).

dfs([State|States], [Move|Moves], Path) :-
   move(State, Next, Move),
   not(memberchk(Next, [State|States])),
   dfs([Next,State|States], Moves, Path).

show([], _).
show([Move|Moves], [State|States]) :-
   State = state(A,B,C,D,E,F,G,H,I),
   format('~n~w~n~n', [Move]),
   format('~w ~w ~w~n',[A,B,C]),
   format('~w ~w ~w~n',[D,E,F]),
   format('~w ~w ~w~n',[G,H,I]),
   show(Moves, States).

% Empty position is marked with '*'

start( state(6,1,3,4,*,5,7,2,0) ).

goal( state(*,0,1,2,3,4,5,6,7) ).

move( state(*,B,C,D,E,F,G,H,J), state(B,*,C,D,E,F,G,H,J), right).
```

```
move( state(*,B,C,D,E,F,G,H,J), state(D,B,C,*,E,F,G,H,J), down ).
move( state(A,*,C,D,E,F,G,H,J), state(*,A,C,D,E,F,G,H,J), left ).
move( state(A,*,C,D,E,F,G,H,J), state(A,C,*,D,E,F,G,H,J), right).
move( state(A,*,C,D,E,F,G,H,J), state(A,E,C,D,*,F,G,H,J), down ).
move( state(A,B,*,D,E,F,G,H,J), state(A,*,B,D,E,F,G,H,J), left ).
move( state(A,B,*,D,E,F,G,H,J), state(A,B,F,D,E,*,G,H,J), down ).
move( state(A,B,C,*,E,F,G,H,J), state(*,B,C,A,E,F,G,H,J), up   ).
move( state(A,B,C,*,E,F,G,H,J), state(A,B,C,E,*,F,G,H,J), right).
move( state(A,B,C,*,E,F,G,H,J), state(A,B,C,G,E,F,*,H,J), down ).
move( state(A,B,C,D,*,F,G,H,J), state(A,*,C,D,B,F,G,H,J), up   ).
move( state(A,B,C,D,*,F,G,H,J), state(A,B,C,D,F,*,G,H,J), right).
move( state(A,B,C,D,*,F,G,H,J), state(A,B,C,D,H,F,G,*,J), down ).
move( state(A,B,C,D,*,F,G,H,J), state(A,B,C,*,D,F,G,H,J), left ).
move( state(A,B,C,D,E,*,G,H,J), state(A,B,*,D,E,C,G,H,J), up   ).
move( state(A,B,C,D,E,*,G,H,J), state(A,B,C,D,*,E,G,H,J), left ).
move( state(A,B,C,D,E,*,G,H,J), state(A,B,C,D,E,J,G,H,*), down ).
move( state(A,B,C,D,E,F,*,H,J), state(A,B,C,D,E,F,H,*,J), left ).
move( state(A,B,C,D,E,F,*,H,J), state(A,B,C,*,E,F,D,H,J), up   ).
move( state(A,B,C,D,E,F,G,*,J), state(A,B,C,D,E,F,*,G,J), left ).
move( state(A,B,C,D,E,F,G,*,J), state(A,B,C,D,*,F,G,E,J), up   ).
move( state(A,B,C,D,E,F,G,*,J), state(A,B,C,D,E,F,G,J,*), right).
move( state(A,B,C,D,E,F,G,H,*), state(A,B,C,D,E,*,G,H,F), up   ).
move( state(A,B,C,D,E,F,G,H,*), state(A,B,C,D,E,F,G,*,H), left ).
```

<u>Output:</u> Attach your screenshot

Conclusion: In this way, we have studied and implemented 8-Puzzle problem.

Experiment 10: Implement traveling salesman problem using Prolog.

Theory: The Traveling Salesman Problem is stated as: A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.

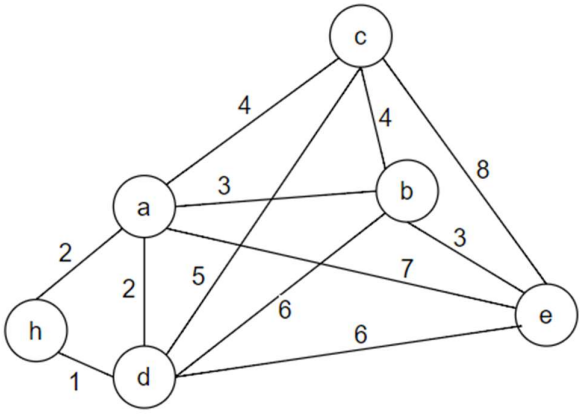


Fig. Sample route of the TSP

Program:

```
/*This is the data set.*/
edge(a, b, 3).
edge(a, c, 4).
edge(a, d, 2).
edge(a, e, 7).
edge(b, c, 4).
edge(b, d, 6).
edge(b, e, 3).
edge(c, d, 5).
edge(c, e, 8).
edge(d, e, 6).
edge(b, a, 3).
edge(c, a, 4).
edge(d, a, 2).
edge(e, a, 7).
edge(c, b, 4).
edge(d, b, 6).
edge(e, b, 3).
edge(d, c, 5).
edge(e, c, 8).
edge(e, d, 6).
edge(a, h, 2).
edge(h, d, 1).

len([], 0).
len([H|T], N):- len(T, X), N is X+1 .

best_path(Visited, Total):- path(a, a, Visited, Total).
```

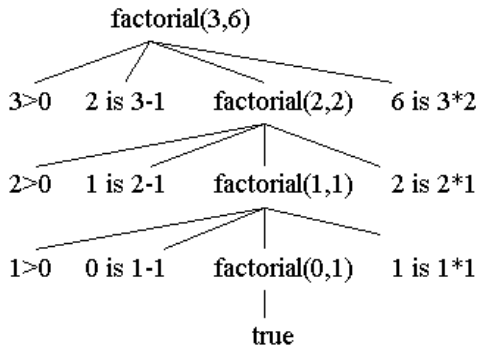
```
path(Start, Fin, Visited, Total) :- path(Start, Fin, [Start], Visited, 0,
Total).

path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, StopLoc, Distance), NewCostn is Costn + Distance, \+
member(StopLoc, CurrentLoc),
    path(StopLoc, Fin, [StopLoc|CurrentLoc], Visited, NewCostn, Total).

path(Start, Fin, CurrentLoc, Visited, Costn, Total) :-
    edge(Start, Fin, Distance), reverse([Fin|CurrentLoc], Visited),
len(Visited, Q),
    (Q\=7 -> Total is 100000; Total is Costn + Distance).

shortest_path(Path):-setof(Cost-Path, best_path(Path,Cost),
Holder),pick(Holder,Path).

best(Cost-Holder,Bcost-_,Cost-Holder):- Cost<Bcost,!.
best(_,X,X).

pick([Cost-Holder|R],X):- pick(R,Bcost-Bholder),best(Cost-Holder,Bcost-
Bholder,X),!.
pick([X],X).
```

Output: Attach your screenshot

Conclusion: In this way, we have studied and implemented travelling salesman problem.

Experiment 11: Find factorial of a given number using Prolog.

Theory: In Prolog, following program has two clauses. The first clause has no body, and it is a unit clause. The second clause of Prolog has a body. On the right-hand side of '-', the body of the second clause exists. The symbol '-' will be read as "if". The body consists of literals and that literals can be separated using the commas ','. The symbol '.' will be read as "and". If the clause is a unit clause, then the head of the class will be the whole class. Otherwise, the part of the clause which appears to the left of the colon in ':-' will be the head of the class. The first clause (unit clause) is used to show that "factorial 0 is 1". The second clause is used to show that "factorial of N is F if N>0, N1 is N-1, the factorial of N1 is F1 and F is N*F1".



Program:

```
% to run the code in SWI-Prolog, do
%        ?- factorial(5).

factorial(0,1).

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

factorial(N):-
   factorial(N,X),
   write(X).
```

Output: Attach your screenshot

Conclusion: In this way, we have studied and implemented Prolog program to find factorial of a given natural number.

****