DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

Artificial Intelligence Lab Manual

Prepared by

Sandeep Kumar Vishwakarma
ASST-PROF,CSE

ARYABHATT COLLEGE OF ENGG. & TECHNOLOGY
13th Km stone, Baghpat - Meerut Road, Vill. Daula Baghpat - 250 601 (U.P)
Contact Phone Number : 09359765640 / 80 Mail at: info@acet.in, Web-Site-
www.acet.in

LIST OF EXPERIMENTS

1. Write a program in prolog to implement simple facts and Queries.
2. Write a program in prolog to implement simple arithmetic.
3. Write a program in prolog to solve Monkey banana problem.
4. Write a program in prolog to solve Tower of Hanoi.
5. Write a program in prolog to solve 8 Puzzle problems.
6. Write a program in prolog to solve 4-Queens problem.
7. Write a program in prolog to solve Traveling salesman problem.
8. Write a program in prolog for Water jug problem.

EXPERIMENT NO. 1

AIM: Write simple fact for following

1. Ram likes mango.
2. Seema is a girl.
3. Bill likes Cindy.
4. Rose is red.
5. John owns gold.

Clauses

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

Goal

```
?- likes (ram,What).
   What = mango.
   1 solution.
```

Viva Questions:

1- First create a source file for the genealogical logicbase application. Start by adding a few members of your family tree. It is important to be accurate, since we will be exploring family relationships. Your own knowledge of who your relatives are will verify the correctness of your Prolog programs.

2- Enter a two-argument predicate that records the parent-child relationship. One argument represents the parent, and the other the child. It doesn't matter in which order you enter the arguments, as long as you are consistent. Often Prolog programmers adopt the convention that parent(A,B) is interpreted "A is the parent of B".

3- Create a source file for the customer order entry program. We will begin it with three record types (predicates). The first is customer/3 where the three arguments are

arg1	Customer name
arg2	City
arg3	Credit rating (aaa, bbb, etc)

4- Next add clauses that define the items that are for sale. It should also have three arguments

arg1
    Item identification number

arg2
    Item name
arg3
    The reorder point for inventory (when at or below this level, reorder)

5- Next add an inventory record for each item. It has two arguments.

arg1
    Item identification number (same as in the item record)
arg2
    Amount in stock

Assignment:

Aim: Write facts for following:

1. Ram likes apple.
2. Ram is taller then Mohan.
3. My name is Subodh.
4. Apple is fruit.
5. Orange is fruit.
6. Ram is male.

AIM: Write simple queries for following facts.

Simple Queries

Now that we have some facts in our Prolog program, we can consult the program in the listener and query, or call, the facts. This chapter, and the next, will assume the Prolog program contains only facts. Queries against programs with rules will be covered in a later chapter.

Prolog queries work by pattern matching. The query pattern is called a **goal**. If there is a fact that matches the goal, then the query succeeds and the listener responds with 'yes.' If there is no matching fact, then the query fails and the listener responds with 'no.'

Prolog's pattern matching is called **unification**. In the case where the logicbase contains only facts, unification succeeds if the following three conditions hold.

- The predicate named in the goal and logicbase are the same.
- Both predicates have the same arity.
- All of the arguments are the same.

Before proceeding, review figure 3.1, which has a listing of the program so far.

The first query we will look at asks if the office is a room in the game. To pose this, we would enter that goal followed by a period at the listener prompt.

```
?- room(office).
yes
```

Prolog will respond with a 'yes' if a match was found. If we wanted to know if the attic was a room, we would enter that goal.

```
?- room(attic).
no
```

Solution:-

clauses

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

queries

```
?-likes(ram,What).
  What= mango
  ?-likes(Who,cindy).
  Who= cindy

?-red(What).
  What= rose
?-owns(Who,What).
  Who= john
  What= gold
```

__**Viva Questions:**__

1- Consider the following Prolog logic base

```
easy(1).
easy(2).
easy(3).
gizmo(a,1).
gizmo(b,3).
gizmo(a,2).
gizmo(d,5).
gizmo(c,3).
gizmo(a,3).
gizmo(c,4).
```

and predict the answers to the queries below, including all alternatives when the semicolon (;) is entered after an answer.

```
?- easy(2).
?- easy(X).
?- gizmo(a,X).
?- gizmo(X,3).
?- gizmo(d,Y).
?- gizmo(X,X).
```

2- Consider this logicbase,

```
harder(a,1).
harder(c,X).
harder(b,4).
harder(d,2).
```

and predict the answers to these queries.

```
?- harder(a,X).
?- harder(c,X).
?- harder(X,1).
?- harder(X,4).
```

3- Enter the listener and reproduce some of the example queries you have seen against location/2. List or print location/2 for reference if you need it. Remember to respond with a semicolon (;) for multiple answers. Trace the query.

4- Pose queries against the genealogical logicbase that:

- Confirm a parent relationship such as parent(dennis, diana)
- Find someone's parent such as parent(X, diana)
- Find someone's children such as parent(dennis, X)
- List all parent-children such as parent(X,Y)

5- If parent/2 seems to be working, you can add additional family members to get a larger logicbase. Remember to include the corresponding male/1 or female/1 predicate for each individual added.

EXPERIMENT NO. 2

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

Arithmetic

Prolog must be able to handle arithmetic in order to be a useful general purpose programming language. However, arithmetic does not fit nicely into the logical scheme of things.

That is, the concept of evaluating an arithmetic expression is in contrast to the straight pattern matching we have seen so far. For this reason, Prolog provides the built-in predicate 'is' that evaluates arithmetic expressions. Its syntax calls for the use of operators, which will be described in more detail in chapter 12.

```
X is <arithmetic expression>
```

The variable X is set to the value of the arithmetic expression. On backtracking it is unassigned.

The arithmetic expression looks like an arithmetic expression in any other programming language.

Here is how to use Prolog as a calculator.

```
?- X is 2 + 2.
X = 4

?- X is 3 * 4 + 2.
X = 14
```

Parentheses clarify precedence.

```
?- X is 3 * (4 + 2).
X = 18

?- X is (8 / 4) / 2.     X = 1
```

In addition to 'is,' Prolog provides a number of operators that compare two numbers. These include 'greater than', 'less than', 'greater or equal than', and 'less or equal than.' They behave more logically, and succeed or fail according to whether the comparison is true or false. Notice the order of the symbols in the greater or equal than and less than or equal operators. They are specifically constructed not to look like an arrow, so that you can use arrow symbols in your programs without confusion.

```
X > Y
X < Y
```

```
X >= Y
X =< Y
```

Here are a few examples of their use.

```
?- 4 > 3.
Yes
?- 4 < 3.
No
?- X is 2 + 2, X > 3.
X = 4
?- X is 2 + 2, 3 >= X.
No
?- 3+4 > 3*2.
Yes
```

<u>Production rules:</u>

c_to_f $\longrightarrow$ f is c * 9 / 5 +32
freezing $\longrightarrow$ f < = 32

<u>Rules:</u>

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

<u>Queries :</u>

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

<u>Viva Questions:</u>

1- Write a predicate valid_order/3 that checks whether a customer order is valid. The arguments should be customer, item, and quantity. The predicate should succeed only if the customer is a valid customer with a good credit rating, the item is in stock, and the quantity ordered is less than the quantity in stock.

2- Write a reorder/1 predicate which checks inventory levels in the inventory record against the reorder quantity in the item record. It should write a message indicating whether or not it's time to reorder.

<u>Assignment:</u>

Aim: Using the following facts answer the question

1. Find car make that cost is exactly 2,00,000/-
2. Find car make that cost is less then 5 lacs.
3. List all the cars available.
4. Is there any car which cost is more then 10 lacs.

<u>EXPERIMENT NO. 3</u>

Aim:- Write a program to solve the Monkey Banana problem.

Imagine a room containing a monkey, chair and some bananas. That have been hanged from the center of ceiling. If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair .

The problem is to prove the monkey can reach the bananas.

Production Rules

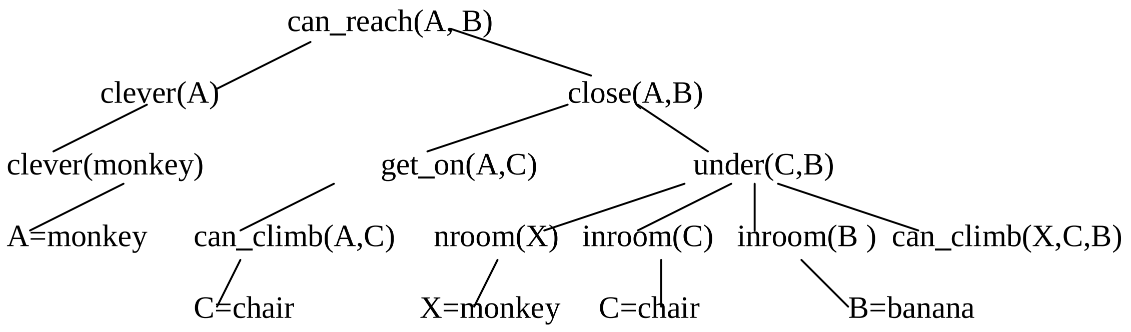
can_reach→clever,close.

get_on:→ can_climb.

under→in room,in_room, in_room,can_climb.

Close→get_on,under| tall

<u>Parse Tree</u>



So Can_climb(monkey,chair) close(monkey,banana)

A=monkey B=banana

Solution:-

Clauses:

```
in_room(bananas).
in_room(chair).
in_room(monkey).
clever(monkey).
can_climb(monkey, chair).
tall(chair).
can_move(monkey, chair, bananas).
can_reach(X, Y):-clever(X),close(X, Y).
get_on(X,Y):- can_climb(X,Y).
```

```
under(Y,Z):-in_room(X),in_room(Y),
                  in_room(Z),can_climb(X,Y,Z).
      close(X,Z):-get_on(X,Y), under(Y,Z);
                  tall(Y).
```

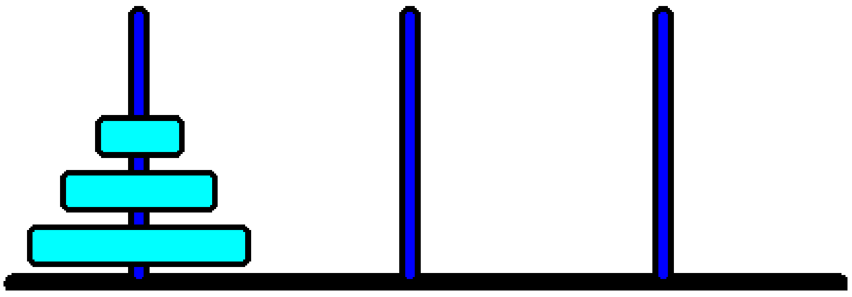
Queries:

```
?- can_reach(A, B).
   A = monkey.
   B = banana.

?- can_reach(monkey, banana).
   Yes.
```

EXPERIMENT NO. 4

Aim:- Write a program to solve Tower of Hanoi.

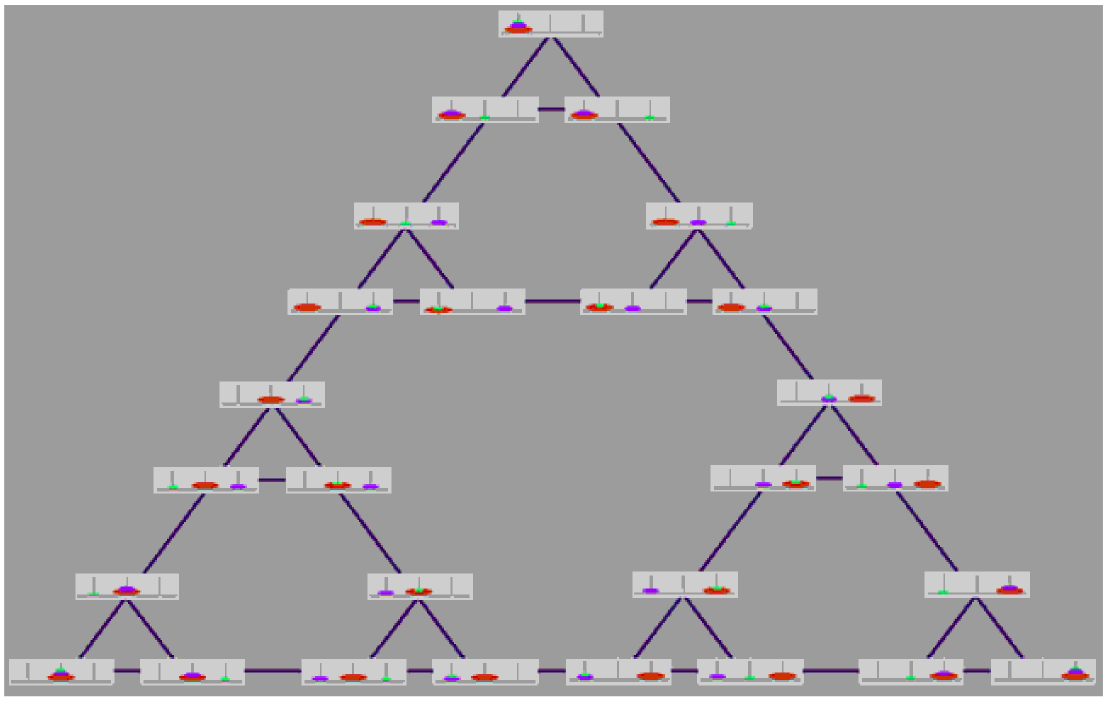
This object of this famous puzzle is to move N disks from the left peg to the right peg using the center peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk. The following diagram depicts the starting setup for N=3 disks.



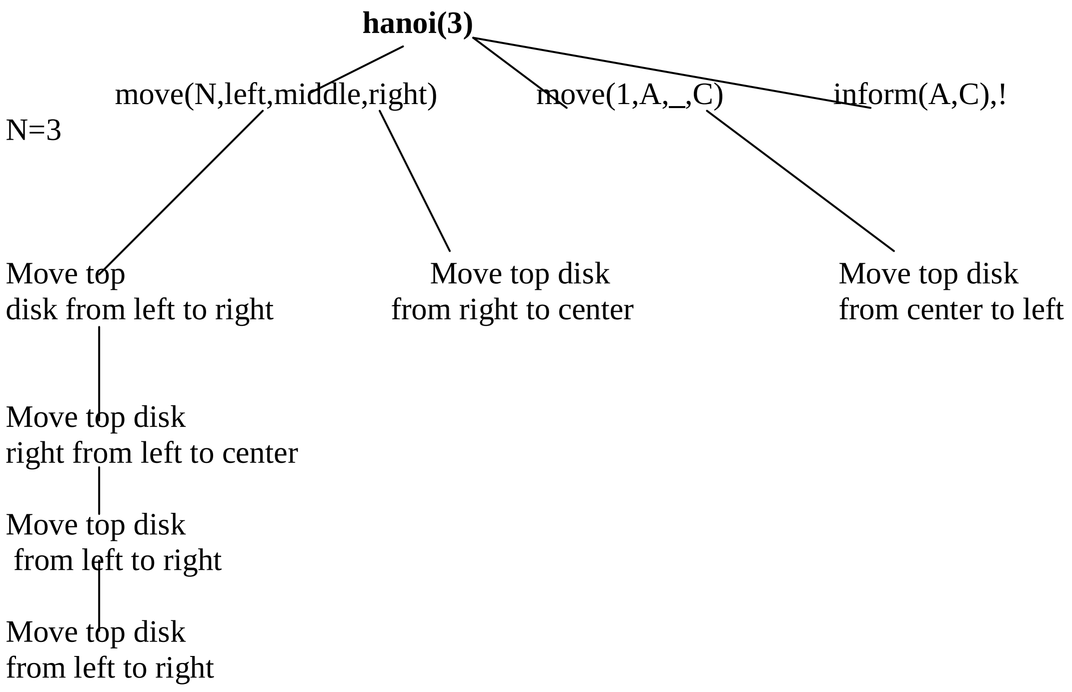
Production Rules

hanoi(N)→move(N,left,middle,right).
move(1,A,_,C)→inform(A,C),fail.
move(N,A,B,C)→N1=N-1,move(N1,A,C,B),inform(A,C),move(N1,B,A,C).

Diagram:-



Parse Tree:-


hanoi(3)
move(N,left,middle,right)
move(1,A,_,C)
inform(A,C),!
N=3
Move top disk from left to right
Move top disk from right to center
Move top disk from center to left
Move top disk right from left to center
Move top disk from left to right
Move top disk from left to right

Solution:-

domains
```
  loc =right;middle;left
```

predicates
```
  hanoi(integer)
  move(integer,loc,loc,loc)
  inform(loc,loc)
```

clauses
```
  hanoi(N):-
             move(N,left,middle,right).
             move(1,A,_,C):-
             inform(A,C),!.

  move(N,A,B,C):-
             N1=N-1,
             move(N1,A,C,B),
             inform(A,C),
             move(N1,B,A,C).

  inform(Loc1, Loc2):-
             write("\nMove a disk from ", Loc1, " to ", Loc2).
```

goal

```
  hanoi(3).
 Move(3,left,right,center).
Move top disk from left to right
Move top disk from left to center
Move top disk from right to center
Move top disk from left to right
Move top disk from center to left
Move top disk from center to right
Move top disk from left to right

Yes
```

Viva Questions:

1 What is Tower of Hanoi problem?

2 What is the role of recursion in Tower of Hanoi problem?

3 what will the effect if the number of disk is 4

Assignment:

Aim: Draw the search tree for Tower of Hanoi for N=4.

EXPERIMENT NO. 5

Aim:- Write a program to solve 8-Puzzle problem.

The title of this section refers to a familiar and popular sliding tile puzzle that has been around for at least forty years. The most frequent older versions of this puzzle have numbers or letters an the sliding tiles, and the player is supposed to slide tiles into new positions in order to realign a scrambled puzzle back into a goal alignment. For illustration, we use the 3 x 3 8-tile version, which is depicted here in goal configuration

7	2	3
4	6	5
1	8	

To represent these puzzle "states" we will use a Prolog term representation employing '/' as a separator. The positions of the tiles are listed (separated by '/') from top to bottom, and from left to right. Use "0" to represent the empty tile (space). For example, the goal is ...
goal(1/2/3/8/0/4/7/6/5).

Production Rules :-

h_function(Puzz,H) → p_fcn(Puzz,P), s_fcn(Puzz,S),H is P + 3*S.

The 'move' productions are defined as follows.
move(P,C,left) → left(P,C).
move(P,C,up) → up(P,C).
move(P,C,right) → right(P,C).
move(P,C,down) → down(P,C).

```
p_fcn(A/B/C/D/E/F/G/H/I, P) →   a(A,Pa), b(B,Pb), c(C,Pc),
                                  d(D,Pd), e(E,Pe), f(F,Pf),
                                  g(G,Pg), h(H,Ph), i(I,Pi),
                                P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.
```

```
s_fcn(A/B/C/D/E/F/G/H/I, S) →1 s_aux(A,B,S1), s_aux(B,C,S2),
                                s_aux(C,F,S3),  s_aux(F,I,S4),
                                s_aux(I,H,S5), s_aux(H,G,S6),
                                s_aux(G,D,S7), s_aux(D,A,S8),
                                s_aux(E,S9),
                            S is S1+S2+S3+S4+S5+S6+S7+S8+S9.
```

s_aux(0,0) → cut
s_aux(X,Y,0) :- Y is X+1, !.
s_aux(8,1,0) :- !.

The heuristic function we use here is a combination of two other estimators: p_fcn, the Manhattan distance function, and s_fcn, the sequence function, all as explained in Nilsson (1980), which estimates how badly out-of-sequence the tiles are (around the outside).

```
h_function(Puzz,H) :- p_fcn(Puzz,P),
                  s_fcn(Puzz,S),
                  H is P + 3*S.
```

The 'move' predicate is defined as follows.

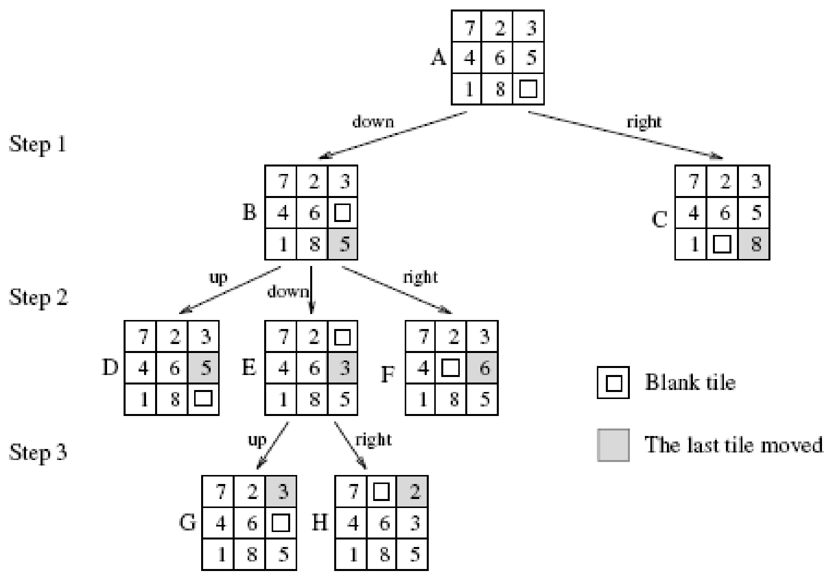
```
move(P,C,left) :-  left(P,C).
move(P,C,up) :-  up(P,C).
move(P,C,right) :-  right(P,C).
move(P,C,down) :-  down(P,C).
```

Here is the code for p and s.

```
  %%% Manhattan distance
p_fcn(A/B/C/D/E/F/G/H/I, P) :-
    a(A,Pa), b(B,Pb), c(C,Pc),
    d(D,Pd), e(E,Pe), f(F,Pf),
    g(G,Pg), h(H,Ph), i(I,Pi),
    P is Pa+Pb+Pc+Pd+Pe+Pf+Pg+Ph+Pg+Pi.

a(0,0). a(1,0). a(2,1). a(3,2). a(4,3). a(5,4). a(6,3). a(7,2). a(8,1).
b(0,0). b(1,0). b(2,0). b(3,1). b(4,2). b(5,3). b(6,2). b(7,3). b(8,2).
c(0,0). c(1,2). c(2,1). c(3,0). c(4,1). c(5,2). c(6,3). c(7,4). c(8,3).
d(0,0). d(1,1). d(2,2). d(3,3). d(4,2). d(5,3). d(6,2). d(7,2). d(8,0).
e(0,0). e(1,2). e(2,1). e(3,2). e(4,1). e(5,2). e(6,1). e(7,2). e(8,1).
f(0,0). f(1,3). f(2,2). f(3,1). f(4,0). f(5,1). f(6,2). f(7,3). f(8,2).
g(0,0). g(1,2). g(2,3). g(3,4). g(4,3). g(5,2). g(6,2). g(7,0). g(8,1).
h(0,0). h(1,3). h(2,3). h(3,3). h(4,2). h(5,1). h(6,0). h(7,1). h(8,2).
i(0,0). i(1,4). i(2,3). i(3,2). i(4,1). i(5,0). i(6,1). i(7,2). i(8,3).

  %%% the out-of-cycle function
s_fcn(A/B/C/D/E/F/G/H/I, S) :-
    s_aux(A,B,S1), s_aux(B,C,S2), s_aux(C,F,S3),
    s_aux(F,I,S4), s_aux(I,H,S5), s_aux(H,G,S6),
    s_aux(G,D,S7), s_aux(D,A,S8), s_aux(E,S9),
    S is S1+S2+S3+S4+S5+S6+S7+S8+S9.

s_aux(0,0) :- !.
s_aux(_,1).

s_aux(X,Y,0) :- Y is X+1, !.
s_aux(8,1,0) :- !.
s_aux(_,_,2).
```

The Prolog program from the previous section and the program outlined in this section can be used as an 8-puzzle solver.

```
?- solve(0/8/1/2/4/3/7/6/5, S).
```



Solution:

left(A/0/C/D/E/F/H/I/J , 0/A/C/D/E/F/H/I/J).
left(A/B/C/D/0/F/H/I/J , A/B/C/0/D/F/H/I/J).
left(A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/0/H/J).
left(A/B/0/D/E/F/H/I/J , A/0/B/D/E/F/H/I/J).
left(A/B/C/D/E/0/H/I/J , A/B/C/D/0/E/H/I/J).
left(A/B/C/D/E/F/H/I/0 , A/B/C/D/E/F/H/0/I).

up(A/B/C/0/E/F/H/I/J , 0/B/C/A/E/F/H/I/J).
up(A/B/C/D/0/F/H/I/J , A/0/C/D/B/F/H/I/J).
up(A/B/C/D/E/0/H/I/J , A/B/0/D/E/C/H/I/J).
up(A/B/C/D/E/F/0/I/J , A/B/C/0/E/F/D/I/J).
up(A/B/C/D/E/F/H/0/J , A/B/C/D/0/F/H/E/J).
up(A/B/C/D/E/F/H/I/0 , A/B/C/D/E/0/H/I/F).

right(A/0/C/D/E/F/H/I/J , A/C/0/D/E/F/H/I/J).
right(A/B/C/D/0/F/H/I/J , A/B/C/D/F/0/H/I/J).
right(A/B/C/D/E/F/H/0/J , A/B/C/D/E/F/H/J/0).
right(0/B/C/D/E/F/H/I/J , B/0/C/D/E/F/H/I/J).
right(A/B/C/0/E/F/H/I/J , A/B/C/E/0/F/H/I/J).
right(A/B/C/D/E/F/0/I/J , A/B/C/D/E/F/I/0/J).

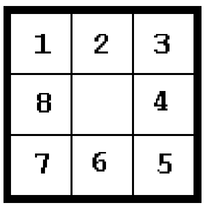
down(A/B/C/0/E/F/H/I/J , A/B/C/H/E/F/0/I/J).
down(A/B/C/D/0/F/H/I/J , A/B/C/D/I/F/H/0/J).
down(A/B/C/D/E/0/H/I/J , A/B/C/D/E/J/H/I/0).
down(0/B/C/D/E/F/H/I/J , D/B/C/0/E/F/H/I/J).
down(A/0/C/D/E/F/H/I/J , A/E/C/D/0/F/H/I/J).
down(A/B/0/D/E/F/H/I/J , A/B/F/D/E/0/H/I/J).

Viva Questions:

1. What is 8-puzze problem.

2. What can be the next states if the board is in following position?

a)



1	2	3
8		4
7	6	5

b)

2	**5**	**1**
3	**7**	**6**
4	**8**	

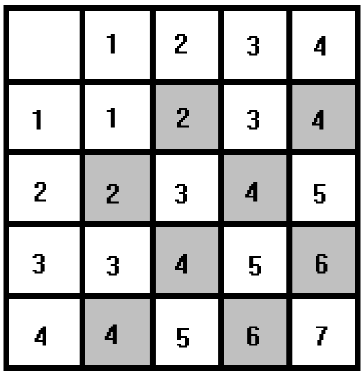
Assignment:

Aim: - Solve the program for the sequence (8,7,6,5,4,1,2,3,0).

EXPERIMENT NO. 6

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

In the 4 Queens problem the object is to place *4* queens on a chessboard in such a way that no queens can capture a piece. This means that no two queens may be placed on the same row, column, or diagonal.



	1	2	3	4
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7

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,NewD2)):-
```

```
      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).
```

goal

```
   nqueens(4),nl.

board([q(1,2),q(2,4),q(3,1),q(4,3),[],[],[7,4,1],[7,4,1])
yes
```

Viva Questions:

1. Explain N-Queen problem.
2. What do you mean by integer *?

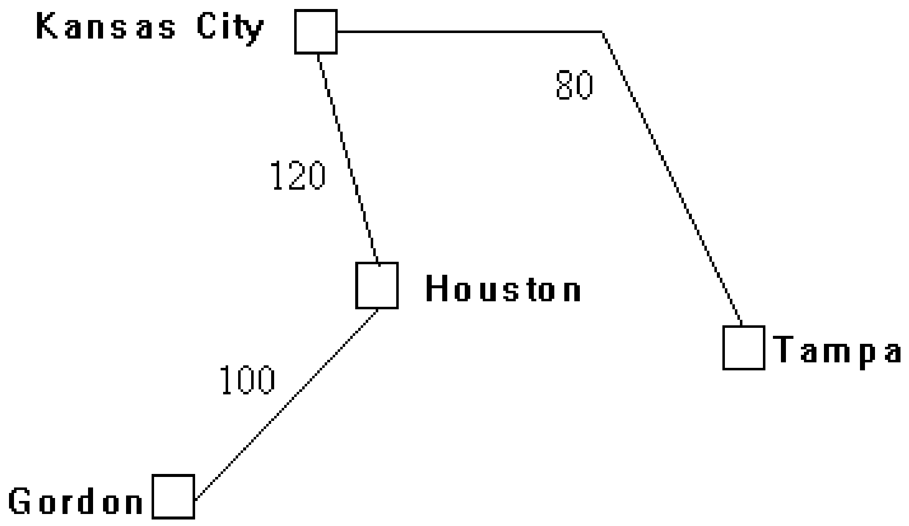
Assignment:

Aim:- Write a program to solve N-Queen problem.

EXPERIMENT NO. 7

Aim:-Write a program to solve traveling salesman problem.

The following is the simplified map used for the prototype:



Production Rules:-

```
  route(Town1,Town2,Distance)→ road(Town1,Town2,Distance).
  route(Town1,Town2,Distance)→ road(Town1,X,Dist1),
                                        route(X,Town2,Dist2),
                                        Distance=Dist1+Dist2,
```

```
domains
  town     = symbol
  distance = integer

predicates
  nondeterm road(town,town,distance)
  nondeterm route(town,town,distance)

clauses
  road("tampa","houston",200).
  road("gordon","tampa",300).
  road("houston","gordon",100).
  road("houston","kansas_city",120).
  road("gordon","kansas_city",130).
 route(Town1,Town2,Distance):-
       road(Town1,Town2,Distance).
   route(Town1,Town2,Distance):-
       road(Town1,X,Dist1),
       route(X,Town2,Dist2),
       Distance=Dist1+Dist2,
       !.
```

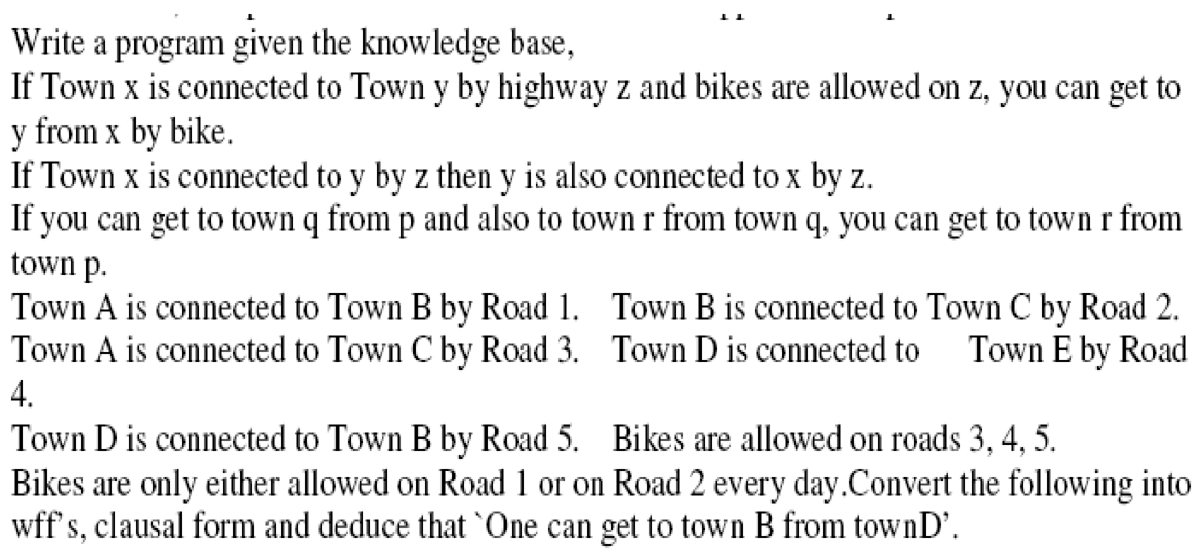
```
goal
     route("tampa", "kansas_city", X),
       write("Distance from Tampa to Kansas City  is ",X),nl.
```

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

Viva Questions:

1. What do you mean by Traveling Salesman problem?
2. Why you use domain?
3. What is the output of following:
   route(kansascity, gordon, X)).

Assignment :



Write a program given the knowledge base,
If Town x is connected to Town y by highway z and bikes are allowed on z, you can get to y from x by bike.
If Town x is connected to y by z then y is also connected to x by z.
If you can get to town q from p and also to town r from town q, you can get to town r from town p.
Town A is connected to Town B by Road 1. Town B is connected to Town C by Road 2.
Town A is connected to Town C by Road 3. Town D is connected to Town E by Road 4.
Town D is connected to Town B by Road 5. Bikes are allowed on roads 3, 4, 5.
Bikes are only either allowed on Road 1 or on Road 2 every day.Convert the following into wff's, clausal form and deduce that `One can get to town B from townD'.

EXPERIMENT NO. 8

Aim:- Write a program for water jug problem.

"You are given two jugs, a 4-gallon one and a 3-gallon one. Neither have any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?".

Production Rules:-

R1: (x,y) --> (4,y) if x < 4
R2: (x,y) --> (x,3) if y < 3
R3: (x,y) --> (x-d,y) if x > 0
R4: (x,y) --> (x,y-d) if y > 0
R5: (x,y) --> (0,y) if x > 0
R6: (x,y) --> (x,0)   if y > 0
R7: (x,y) --> (4,y-(4-x))   if x+y >= 4 and y > 0
R8: (x,y) --> (x-(3-y),y)   if x+y >= 3 and x > 0
R9: (x,y) --> (x+y,0)   if x+y =< 4 and y > 0
R10: (x,y) --> (0,x+y)   if x+y =< 3 and x > 0

Parse Tree

```
        <0,0>
         / \
       R1   R2
      <0,3>
      / |...\
     /  |... \
   R1   R4...  R9
            <3,0>
            / | \
           /  |  \
          R2  R3   R5
        <3,3>
        / |...\...
       /  |... \...
      R1  R3   R7
             <4,2>
             / |...\...
            /  |... \...
           R2  R3   R5
                  <0,2>
                  / | \
                 /  |  \
                R1  R7  R9
                     <2,0>
```

<u>Solution:-</u>

Clauses

```
water_jugs :-
   SmallCapacity = 3,
   LargeCapacity = 4,
   Reservoir is SmallCapacity + LargeCapacity + 1,
   volume( small, Capacities, SmallCapacity ),
   volume( large, Capacities, LargeCapacity ),
   volume( reservoir, Capacities, Reservoir ),
   volume( small, Start, 0 ),
   volume( large, Start, 0 ),
   volume( reservoir, Start, Reservoir ),
   volume( large, End, 2 ),
   water_jugs_solution( Start, Capacities, End, Solution ),
   phrase( narrative(Solution, Capacities, End), Chars ),
   put_chars( Chars ).

water_jugs_solution( Start, Capacities, End, Solution ) :-
   solve_jugs( [start(Start)], Capacities, [], End, Solution ).

solve_jugs( [Node|Nodes], Capacities, Visited, End, Solution ) :-
            node_state( Node, State ),
            ( State = End ->
                        Solution = Node
            ; otherwise ->
                        findall(
                                    Successor,
                                    successor(Node, Capacities, Visited, Successor),
                                    Successors
                                    ),
                        append( Nodes, Successors, NewNodes ),
                        solve_jugs( NewNodes, Capacities, [State|Visited], End, Solution )
            ).

successor( Node, Capacities, Visited, Successor ) :-
            node_state( Node, State ),
            Successor = node(Action,State1,Node),
            jug_transition( State, Capacities, Action, State1 ),
            \+ member( State1, Visited ).
jug_transition( State0, Capacities, empty_into(Source,Target), State1 ) :-
            volume( Source, State0, Content0 ),
            Content0 > 0,
            jug_permutation( Source, Target, Unused ),
            volume( Target, State0, Content1 ),
            volume( Target, Capacities, Capacity ),
            Content0 + Content1 =< Capacity,
            volume( Source, State1, 0 ),
            volume( Target, State1, Content2 ),
            Content2 is Content0 + Content1,
            volume( Unused, State0, Unchanged ),
            volume( Unused, State1, Unchanged ).
jug_transition( State0, Capacities, fill_from(Source,Target), State1 ) :-
            volume( Source, State0, Content0 ),
            Content0 > 0,
            jug_permutation( Source, Target, Unused ),
            volume( Target, State0, Content1 ),
            volume( Target, Capacities, Capacity ),
            Content1 < Capacity,
            Content0 + Content1 > Capacity,
            volume( Source, State1, Content2 ),
            volume( Target, State1, Capacity ),
            Content2 is Content0 + Content1 - Capacity,
            volume( Unused, State0, Unchanged ),
            volume( Unused, State1, Unchanged ).

volume( small, jugs(Small, _Large, _Reservoir), Small ).
volume( large, jugs(_Small, Large, _Reservoir), Large ).
volume( reservoir, jugs(_Small, _Large, Reservoir), Reservoir ).


jug_permutation( Source, Target, Unused ) :-
            select( Source, [small, large, reservoir], Residue ),
            select( Target, Residue, [Unused] ).


node_state( start(State), State ).
node_state( node(_Transition, State, _Predecessor), State ).


narrative( start(Start), Capacities, End ) -->
            "Given three jugs with capacities of:", newline,
            literal_volumes( Capacities ),
            "To obtain the result:", newline,
            literal_volumes( End ),
            "Starting with:", newline,
            literal_volumes( Start ),
            "Do the following:", newline.
narrative( node(Transition, Result, Predecessor), Capacities, End ) -->
            narrative( Predecessor, Capacities, End ),
            literal_action( Transition, Result ).

literal_volumes( Volumes ) -->
            indent, literal( Volumes ), ";", newline.

literal_action( Transition, Result ) -->
            indent, "- ", literal( Transition ), " giving:", newline,
            indent, indent, literal( Result ), newline.

literal( empty_into(From,To) ) -->
```

```
        "Empty the ", literal( From ), " into the ",
        literal( To ).
literal( fill_from(From,To) ) -->
        "Fill the ", literal( To ), " from the ",
        literal( From ).
literal( jugs(Small,Large,Reservoir) ) -->
        literal_number( Small ), " gallons in the small jug, ",
        literal_number( Large ), " gallons in the large jug and ",
        literal_number( Reservoir ), " gallons in the reservoir".
literal( small ) --> "small jug".
literal( large ) --> "large jug".
literal( reservoir ) --> "reservoir".

literal_number( Number, Plus, Minus ) :-
        number( Number ),
        number_chars( Number, Chars ),
        append( Chars, Minus, Plus ).

indent --> "  ".

newline --> "  ".
```

Goal

```
?- water_jugs.

Given three jugs with capacities of:
3 gallons in the small jug, 4 gallons in the large jug and 8 gallons in the reservoir;
To obtain the result:
  0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir;
Starting with:
  0 gallons in the small jug, 0 gallons in the large jug and 8 gallons in the reservoir;
Do the following:
  - Fill the small jug from the reservoir giving:
    3 gallons in the small jug, 0 gallons in the large jug and 5 gallons in the reservoir
  - Empty the small jug into the large jug giving:
    0 gallons in the small jug, 3 gallons in the large jug and 5 gallons in the reservoir
  - Fill the small jug from the reservoir giving:
    3 gallons in the small jug, 3 gallons in the large jug and 2 gallons in the reservoir
  - Fill the large jug from the small jug giving:
    2 gallons in the small jug, 4 gallons in the large jug and 2 gallons in the reservoir
  - Empty the large jug into the reservoir giving:
    2 gallons in the small jug, 0 gallons in the large jug and 6 gallons in the reservoir
  - Empty the small jug into the large jug giving:
    0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir

yes
```

Viva Questions:

1. Witch type of search algorithm you used?
2. Can you use any other algorithm to implement it?
3. Can I use this program to solve other water jug problem?
4. Write 2 more solution for the water jug problem.

Assignment:

Aim :- Write a program for water jug problem 5-lit and 2-lit jug, at the end 5-lit jug have

1 liter of water.