SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur-603203.

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Lab Manual

1904009 ARTIFICIAL INTELLIGENCE LABORATORY

(VI SEMESTER)

Academic Year: 2024-2025(EVEN)

Prepared By

Ms. W.V Sherlin Sherly, Asst. Prof. (O.G)

EX. No: 1 - STUDY OF PROLOG

INTRODUCTION

- Prolog or **PRO**gramming in **LOG**ics is a logical and declarative programming language.
- It is one major example of the fourth generation language that supports the declarative programming paradigm.
- This is particularly suitable for programs that involve symbolic or non-numeric computation.
- This is the main reason to use Prolog as the programming language in Artificial Intelligence, where symbol manipulation and inference manipulation are the fundamental tasks.
- In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it.
- However, in Prolog we are supposed to give clues as the solution method.

Prolog language basically has three different elements:

Facts: The fact is predicate that is true, for example, if we say, "Tom is the son of Jack", then this is a fact.

Rules: Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as:

grandfather(X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

Questions: And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

HISTORY OF PROLOG

The heritage of prolog includes the research on theorem provers and some other automated deduction system that were developed in 1960s and 1970s. The Inference mechanism of the Prolog is based on Robinson's Resolution Principle, that was proposed in 1965, and Answer extracting mechanism by Green (1968). These ideas came together forcefully with the advent of linear resolution procedures.

The explicit goal-directed linear resolution procedures, gave impetus to the development of a general purpose logic programming system. The first Prolog was the Marseille Prolog based on the work by Colmerauer in the year 1970. The manual of this Marseille Prolog interpreter (Roussel, 1975) was the first detailed description of the Prolog language.

Prolog is also considered as a fourth generation programming language supporting the declarative programming paradigm. The well-known Japanese Fifth-Generation Computer Project, that was announced in 1981, adopted Prolog as a development language, and thereby grabbed considerable attention on the language and its capabilities.

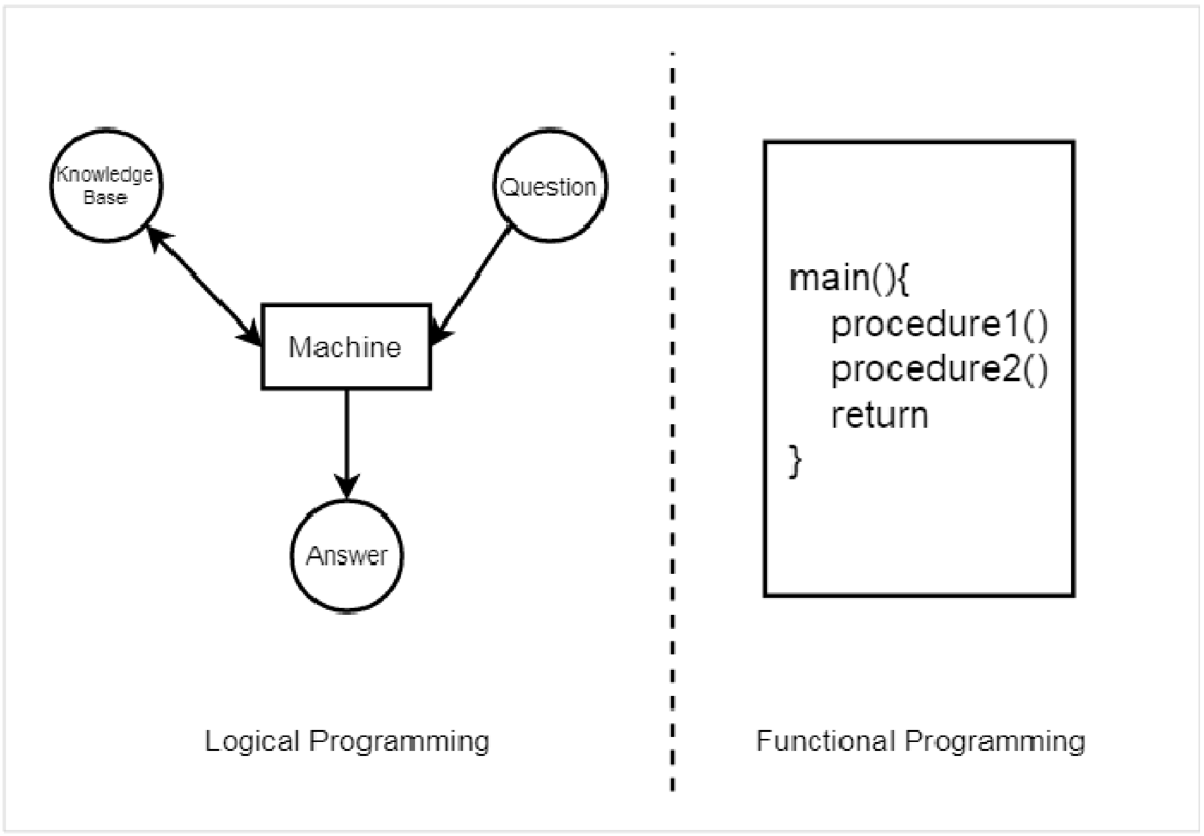
SOME APPLICATIONS OF PROLOG

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used:

- Intelligent Database Retrieval
- Natural Language Understanding
- Specification Language
- Machine Learning
- Robot Planning

- Automation System
- Problem Solving
- Expert Systems

LOGIC AND FUNCTIONAL PROGRAMMING



From this illustration, we can see that in Functional Programming, we have to define the procedures, and the rule how the procedures work. These procedures work step by step to solve one specific problem based on the algorithm. On the other hand, for the Logic Programming, we will provide knowledge base. Using this knowledge base, the machine can find answers to the given questions, which is totally different from functional programming.

In functional programming, we have to mention how one problem can be solved, but in logic programming we have to specify for which problem we actually want the solution.

Then the logic programming automatically finds a suitable solution that will help us solve that specific problem.

Now let us see some more differences below:

FUNCTIONAL PROGRAMMING	LOGIC PROGRAMMING
Functional Programming follows the Von-Neumann Architecture, or uses the sequential steps.	Logic Programming uses abstract model, or deals with objects and their relationships.

The syntax is actually the sequence of statements like (a, s, I).	The syntax is basically the logic formulae (Horn Clauses).
The computation takes part by executing the statements sequentially.	It computes by deducting the clauses.
Logic and controls are mixed together.	Logics and controls can be separated.

PROLOG SOFTWARE

PROLOG is Open Source Software Application.

SWI-Prolog can be downloaded from the https://www.swi-prolog.org

CONCLUSION: Thus the PROLOG programming language has been studied.

EX. No: 2(A) SIMPLE FACTS & QUERIES IN PROLOG

AIM: Write a program in prolog to implement simple facts and Queries

PROCEDURE:

Represent the below given statements as Clauses in Prolog

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

Clauses in Prolog

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

Queries: To Test the Validity of the Clauses

```
?- likes (ram,What).
?- girl(name).
?- red(color).
?- likes (Whom,cindy).
?- owns(john,What).
```

OUTPUT:

RESULT: The simple facts and queries has been written and implemented using Prolog.

EX. No: 2(B) **CREATING RULES IN PROLOG**

AIM: Write a program to create simple logical Rules in prolog.

PROCEDURE:

Represent the below given statements as Clauses in Prolog

“Mary likes all animals but snakes”

It would be very easy and straight forward, if the statement is “Mary likes all animals”. In that case we can write “Mary likes X if X is an animal”. And in prolog we can write this statement as, likes(mary, X) := animal(X).

Our actual statement can be expressed as:

- ❖ If X is snake, then “Mary likes X” is not true
- ❖ Otherwise if X is an animal, then Mary likes X.

In prolog we can write this as:

- ❖ likes(mary,X) :- snake(X), !, fail.
- ❖ likes(mary, X) :- animal(X).

Clauses in Prolog

```
animal(dog).

animal(cat).

animal(elephant).

animal(tiger).

animal(cobra).

animal(python).

snake(cobra).

snake(python).

likes(mary, X) :- snake(X), !, fail.

likes(mary, X) :- animal(X).
```

Queries: To Test the Validity of the Clauses

```
?- likes(mary,elephant).

?- likes(mary,tiger).

?- likes(mary,python).

?- likes(mary,cobra).
```

OUTPUT:

RESULT: The simple logical rules has been created and implemented using Prolog.

EX. No: 2(C) TO REPRESENT RELATIONSHIPS IN PROLOG

AIM: Write a program to represent complex relationships in prolog.

PROCEDURE:

Relationship is one of the main features that we have to properly mention in Prolog.

- These relationships can be expressed as facts and rules.
- Relations specifies relationship between objects and properties of the objects.
- There are various kinds of relationships, of which some can be rules as well.
- A rule can find out about a relationship even if the relationship is not defined explicitly as a fact.

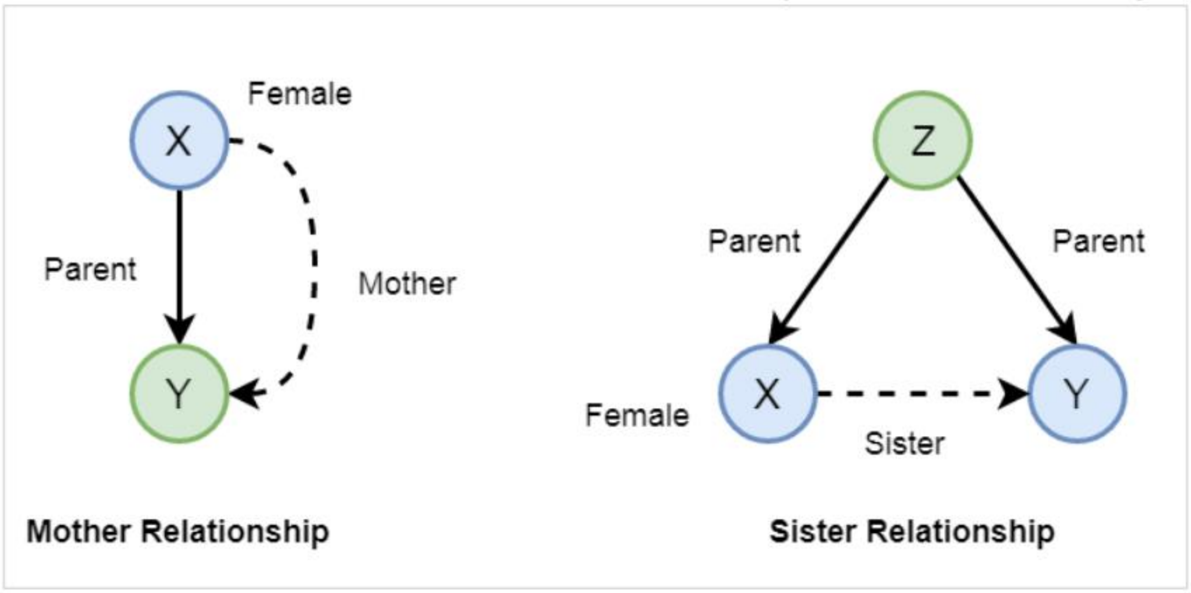
We can define a brother relationship as follows:

Two person are brothers, if,

- They both are male.
- They have the same parent.

Now consider we have the below phrases:

- parent(sudip, piyus).
- parent(sudip, raj).
- male(piyus).
- male(raj).
- brother(X,Y) :- parent(Z,X), parent(Z,Y),male(X), male(Y)



PROGRAM:

```
female(pam).

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

Queries: To Test the Validity of the Clauses

```
?- parent(X,jim).
?- mother(X,Y).
?- haschild(X).
?- sister(X,Y).
?- brother(X,Y).
```

OUTPUT:

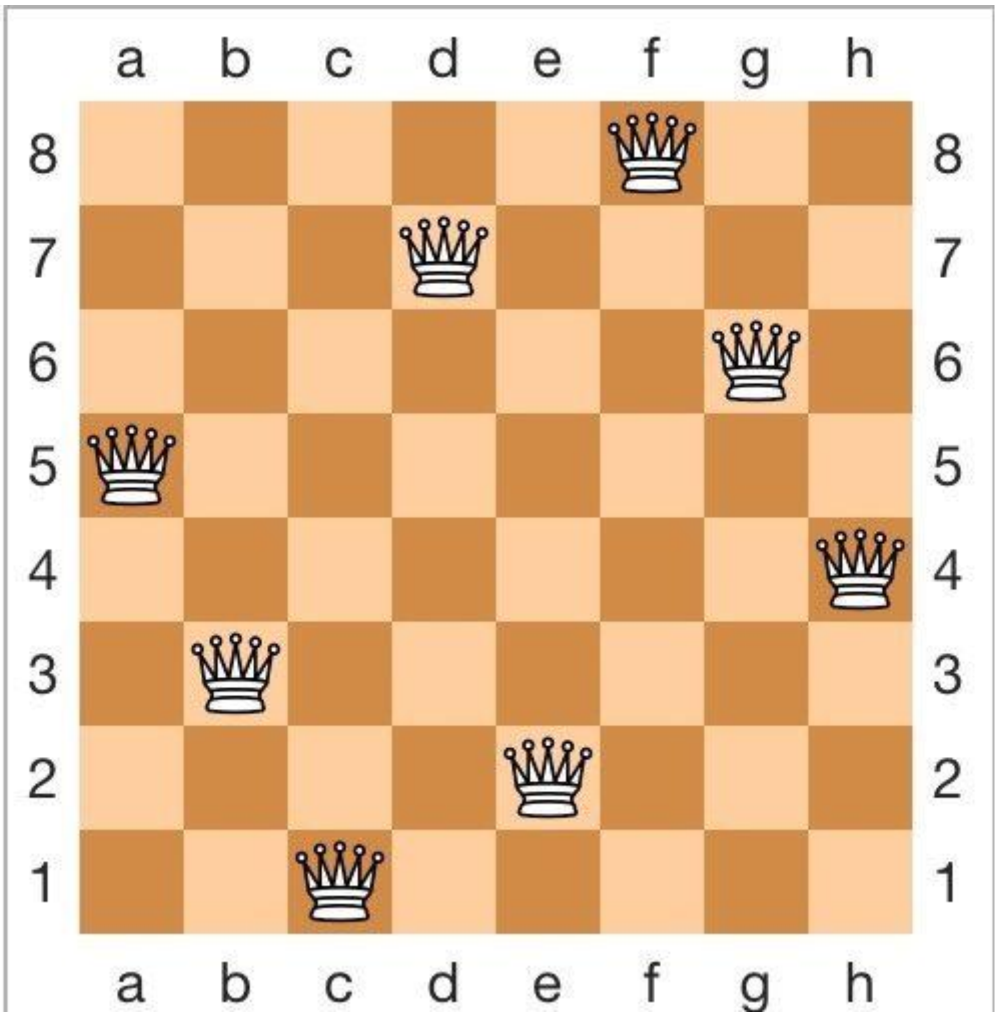
RESULT: The complex relationships has been implemented using Prolog.

EX. No: 3 EIGHT QUEENS PROBLEM

AIM: Write a program to solve Eight Queens problem in Prolog.

Procedure:

- ❖ The "Eight Queens Problem" is a classic problem in computer science that involves placing eight queens on a chessboard such that no two queens are in a position to attack each other.
- ❖ For Solving this Eight Queens Problem we have to use a backtracking approach to find a solution.
- ❖ It starts by placing the first queen on the first row and first column, and then checks if it is attacking any other queen. If not, it moves on to the next queen and repeats the process.
- ❖ If it finds that a queen is attacking another queen, it backtracks to the previous queen and tries a different position for that queen.
- ❖ This process continues until all queens are placed in a non-attacking position or all possibilities have been exhausted.



PROGRAM:

```
commands :-
  write("List of commands:"), nl, nl,
  write("print_coordinates"), nl,
  write("print_board"), nl,
  nl.

print_coordinates :-
  solution(A),
  display_coordinates(A, 1).

print_board :-
  solution(A),
  print_boundry,
  print_columns(A, 1),
  print_column_letters,
  nl.

print_square :- write("| ").
print_queen :- write("|Q").
print_end_quare :- write("|"), nl.
print_boundry :- write("   - - - - - - - - "), nl.
print_column_letters :- write("   h g f e d c b a "), nl.

print_columns([], _).
print_columns([H|T], C) :-
  write(C), write(" "),
  print_row(H, 1),
  C1 is C + 1,
  print_columns(T, C1).

print_row(_, 9) :-
  print_end_quare,
  print_boundry, !.
print_row(X, Col) :-
```

```
  X is Col, print_queen,
  Col1 is Col + 1, print_row(X, Col1).
print_row(X, Col) :-
  X =\= Col, print_square,
  Col1 is Col + 1, print_row(X, Col1).

solution(Queens) :-
 permutation([1,2,3,4,5,6,7,8], Queens),
 safe(Queens).

permutation([],[]).
permutation([H|T], P) :-
 permutation(T, P1),
 mydelete(H, P, P1).

mydelete(X, [X|L], L).
mydelete(X, [Y|L], [Y|L1]) :-
 mydelete(X, L, L1).

safe([]).
safe([Queen|Others]) :-
 safe(Others),
 noattack(Queen,Others,1).

noattack(_,[],_).
noattack(Y, [Y1|L], X1) :-
 Y1 - Y =\= X1,
 Y - Y1 =\= X1,
 X2 is X1 + 1,
 noattack(Y, L, X2).

display_coordinates([], 9).
display_coordinates([H|T], Y) :-
  nth1(H, [h,g,f,e,d,c,b,a], X),
  write("["), write(X),
  write(Y), write("]"), nl,
  Y1 is Y + 1,
  display_coordinates(T, Y1).
```

OUTPUT:

```
?- commands.
List of commands:

print_coordinates
print_board

true.

?- print_coordinates.
[d1]
[g2]
[c3]
[h4]
[b5]
[e6]
[a7]
[f8]
true ;
[c1]
[f2]
[d3]
[b4]
[h5]
[e6]
[g7]
[a8]
true ;
[c1]
[e2]
[b3]
[h4]
[f5]
[d6]
[g7]
[a8]
true .

?- print_board.
   - - - - - - - -
1 | | | | |Q| | | |
  - - - - - - - - -
2 | |Q| | | | | | |
  - - - - - - - - -
3 | | | | | |Q| | |
  - - - - - - - - -
4 |Q| | | | | | | |
  - - - - - - - - -
5 | | | | | | |Q| |
  - - - - - - - - -
6 | | | |Q| | | | |
  - - - - - - - - -
7 | | | | | | | |Q|
  - - - - - - - - -
8 | | |Q| | | | | |
  - - - - - - - - -
  h g f e d c b a

true ;
  - - - - - - - - -
1 | | | | | |Q| | |
  - - - - - - - - -
2 | | |Q| | | | | |
  - - - - - - - - -
3 | | | | |Q| | | |
  - - - - - - - - -
4 | | | | | | |Q| |
  - - - - - - - - -
5 |Q| | | | | | | |
  - - - - - - - - -
6 | | | |Q| | | | |
  - - - - - - - - -
7 | |Q| | | | | | |
  - - - - - - - - -
8 | | | | | | | |Q|
  - - - - - - - - -
  h g f e d c b a

true ;
  - - - - - - - - -
1 | | | | | |Q| | |
  - - - - - - - - -
2 | | | |Q| | | | |
  - - - - - - - - -
3 | | | | | | |Q| |
  - - - - - - - - -
4 |Q| | | | | | | |
  - - - - - - - - -
5 | | |Q| | | | | |
  - - - - - - - - -
6 | | | | |Q| | | |
  - - - - - - - - -
7 | |Q| | | | | | |
  - - - - - - - - -
```

```
8 | | | | | | | |Q|
   - - - - - - - -
   h g f e d c b a

true .
```

RESULT: Thus the Eight Queens problem has been Solved using Prolog language.

EX. No: 4a **SIMPLE CALCULATOR**

AIM: Write a program to implement simple calculator in Prolog.

PROCEDURE:

Represent the following functions as prolog statements.

1. A function add that returns the sum of given two numbers.
2. A function sub that returns the difference of given two numbers.
3. A function mul that returns the product of given two numbers.
4. A function div that returns the quotient of given two numbers.

Clauses in prolog:

```
add(A,B,SUM):-SUM is A+B.

sub(A,B,DIF):-DIF is A-B.

mul(A,B,MUL):-MUL is A*B.

div(A,B,DIV):-DIV is A/B.
```

Queries: To Test the Validity of the Clauses

```
?- add(2,3,SUM).

?- sub(6,4,DIF).

?- mul(10,9,MUL).

?-div(100,5,DIV).
```

OUTPUT

```
?- add(2,3,SUM).

SUM=5


?- sub(6,4,DIF).

DIF=2



?- mul(10,9,MUL).

MUL=90
```

```
?-div(100,5,DIV).
DIV=20
```

RESULT: Thus the simple calculator has been implemented in Prolog successfully.

EX.NO:4b **FACTORIAL**

AIM: To write a program for implementing the given factorial of a prolog program.

PROCEDURE:

Represent the following functions as prolog statements.

1. When the number is 0, its factorial is 1 - This can be done with a simple fact.

2. When the number is greater than zero, compute Number-1, obtain its factorial, and multiplying the result by Number.

CLAUSES IN PROLOG:

```
factorial(0,1).
factorial(N,M) :- N>0,
 N1 is N-1,
 factorial(N1, M1),
 M is N*M1.
```

Queries :To test the validity of the clauses

```
?factorial(5,X).
?factorial(7,X).
?factorial(12,X).
```

OUTPUT

```
?factorial(5,X).
X=120

?factorial(7,X).
X=5040

?factorial(12,X).
X=479001600
```

RESULT: Thus the factorial has been implemented in Prolog successfully.

EX. No: 4c FIBONACCI SERIES

AIM: Write a program to display n numbers of the Fibonacci series in Prolog.

PROCEDURE:

Represent the following as prolog statements.

1. Define the base cases for the first 2 numbers of the Fibonacci series i.e. 0 and 1.
2. Define a recursive function to generate the nth number if the Fibonacci series.
3. Define a predicate to print the series.

Clauses in prolog:

```
% Define the base cases

fib(0, 0).

fib(1, 1).


% Define the recursive rule

fib(N, F) :-

   N > 1,

   N1 is N - 1,

   N2 is N - 2,

   fib(N1, F1),

   fib(N2, F2),

   F is F1 + F2.


% Define a predicate to print the series

fib_series(N) :-

   N > 0,

   fib_series(0, N).


fib_series(N, N).

fib_series(A, N) :-

   A < N,

   fib(A, F),

   write(F), write(' '),

   A1 is A + 1,
    fib_series(A1, N).
```

Queries: To Test the Validity of the Clauses

```
?- fib_series(10).

?- fib_series(7).
```

OUTPUT

```
?- fib_series(10).

0 1 1 2 3 5 8 13 21 34

True.


?- fib_series(7).

0 1 1 2 3 5 8

True.
```

RESULT: Thus the program to display Fibonacci series has been implemented in Prolog successfully.

EX. No: 5a **BEST FIRST SEARCH**

AIM: Write a program to perform best first search in Prolog.

PROCEDURE:

Greedy best first search is an informed search that, instead of measuring distance travelled, measures distance to the goal. This is the heuristic, which has been provided in the code. Note that the heuristic cost for travel to Bucharest is 0, implying that this is the goal and that these heuristic costs will only work with Bucharest as the goal. Greedy best first search is like a depth-first search except that fringe is sorted based on heuristic cost and you must pop the least cost node. After writing codes for the map of Romania, carry out a search from arad to bucharest. Do not forget that identifiers starting with a capital letter are actually variables, do not accidentally type Arad as opposed to arad.

Clauses in prolog:

```
arc(arad,zerind,75). arc(arad,sibiu,140).
arc(arad,timisoara,118).
arc(bucharest,fagaras,211).
arc(bucharest,pitesti,101).
arc(bucharest,giurgiu,90).
arc(bucharest,urziceni,85).
arc(craiova,dobreta,120).
arc(craiova,rimnicu,146).
arc(craiova,pitesti,138).
arc(dobreta,mehadia,75).
arc(dobreta,craiova,120). arc(eforie,hirsova,86).
arc(fagaras,sibiu,99).
arc(fagaras,bucharest,211). arc(giurgiu,bucharest,90).
arc(hirsova,urziceni,98). arc(hirsova,eforie,86).
arc(iasi,neamt,87). arc(iasi,vaslui,92).
arc(lugoj,timisoara,111). arc(lugoj,mehadia,70).
arc(mehadia,lugoj,70). arc(mehadia,dobreta,75).
arc(neamt,iasi,87). arc(oradea,zerind,71).
arc(oradea,sibiu,151). arc(pitesti,rimnicu,97).
arc(pitesti,craiova,138). arc(pitesti,bucharest,101).
arc(rimnicu,sibiu,80). arc(rimnicu,pitesti,97).
arc(rimnicu,craiova,146). arc(sibiu,arad,140).
arc(sibiu,oradea,151). arc(sibiu,fagaras,99).
arc(sibiu,rimnicu,80). arc(timisoara,arad,118).
arc(timisoara,lugoj,111). arc(urziceni,bucharest,85).
arc(urziceni,hirsova,98). arc(urziceni,vaslui,142).
arc(vaslui,iasi,92). arc(vaslui,urziceni,142).
arc(zerind,arad,75). arc(zerind,oradea,71).

% heuristic function: stright line distance to bucharest

% (works only if the goal state is bucharest)
```

```
h(Node, Value) :-     hdist(Node,
Value). h(_,1). % A default value


hdist(arad                    ,366).
hdist(bucharest,                 0).
hdist(craiova                 ,160).
hdist(dobreta                 ,242).
hdist(eforie                  ,161).
hdist(fagaras                 ,178).
hdist(giurgiu          ,        77).
hdist(hirsova                 ,151).
hdist(iasi                    ,266).
hdist(lugoj                   ,244).
hdist(mehadia                 ,241).
hdist(neamt                   ,234).
hdist(oradea                  ,380).
hdist(pitesti          ,        98).
hdist(rimnicu                 ,193).
hdist(sibiu                   ,253).
hdist(timisoara,329).
hdist(urziceni         ,        80).
hdist(vaslui                  ,199).
hdist(zerind   ,374).



best_first([[Goal|Path]|_],Goal,[Goal|Path],0).
best_first([Path|Queue],Goal,FinalPath,N)
:-    extend(Path,NewPaths),     append(Queue,NewPaths,Queue1),
sort_queue1(Queue1,NewQueue),
wrq(NewQueue),     best_first(NewQueue,Goal,FinalPath,M),
   N is M+1.


extend([Node|Path],NewPaths) :-     findall([NewNode,Node|Path],
          (arc(Node,NewNode,_),

          \+ member(NewNode,Path)), % for avoiding loops

          NewPaths).


sort_queue1(L,L2) :-

   swap1(L,L1), !,     sort_queue1(L1,L2).
sort_queue1(L,L).


swap1([[A1|B1],[A2|B2]|T],[[A2|B2],[A1|B1]|T])
:-    hh(A1,W1),     hh(A2,W2),
   W1>W2.

swap1([X|T],[X|V]) :-     swap1(T,V).
```

```
hh(State, Value)
:-     h(State,Value),     number(Value), !.
hh(State, Value) :-    write('Incorrect heuristic
functionh: '),    write(h(State, Value)),
nl,    abort.

wrq(Q) :- length(Q,N), writeln(N).
```

Queries: To Test the Validity of the Clauses

```
?- best_first([[arad]],bucharest,N,P).
```

OUTPUT

```
?- best_first([[arad]],bucharest,N,P).
3
5
5
N=[Bucharest,fagaras,sibiu,arad],
P=3
```

RESULT: Thus the program to perform best first search has been implemented in Prolog successfully.

EX. No: 5b **LINEAR SEARCH**

AIM: Write a program to perform linear search in Prolog.

PROCEDURE:

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection. We need to get the list of elements and the key from user to perform the search.

Clauses in prolog:

```
/*Linear Search in Prolog*/ linearSearch([Key|_], Key).


linearSearch([], _) :- fail. linearSearch([_|Tail], Key) :-
linearSearch(Tail, Key).


go :-
        write('Enter List : '),
         read(List),
        write('Enter Key : '),
         read(Key),
 linearSearch(List, Key),  write('Element Found');write('Element not
found').


:-initialization(go).
```

OUTPUT

```
Enter List : [1,2,3,4,5].
Enter Key : |    4.

Element Found
```

RESULT: Thus the program to perform linear search has been implemented in Prolog successfully.

EX. No: 6a **BREADTH FIRST SEARCH**

AIM: Write a program to perform breadth first search in Prolog.

PROCEDURE:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node. We need to maintain a queue of visited nodes.

Clauses in prolog:

```
%connected(+Start, +Goal, -Weight)

connected(1,7,1). connected(1,8,1).

connected(1,3,1). connected(7,4,1).

connected(7,20,1). connected(7,17,1).

connected(8,6,1). connected(3,9,1).

connected(3,12,1). connected(9,19,1).

connected(4,42,1). connected(20,28,1).

connected(17,10,1).


connected2(X,Y,D) :- connected(X,Y,D).

connected2(X,Y,D) :- connected(Y,X,D).


next_node(Current, Next, Path) :-

connected2(Current, Next, _),     not(member(Next,

Path)). breadth_first(Goal, Goal, _,[Goal]).

breadth_first(Start, Goal, Visited, Path) :-

   findall(X,

          (connected2(X,Start,_),not(member(X,Visited))),

          [T|Extend]),     write(Visited), nl,

append(Visited, [T|Extend], Visited2),     append(Path,

[T|Extend], [Next|Path2]),     breadth_first(Next, Goal,

Visited2, Path2).
```

Queries: To Test the Validity of the Clauses

```
?- breadth_first(1,28,[1],[]).
```

OUTPUT

```
 [1]
[1,7,8,3]
[1,7,8,3,4,20,17]
[1,7,8,3,4,20,17,6]
[1,7,8,3,4,20,17,6,9,12]
[1,7,8,3,4,20,17,6,9,12,42]
[1,7,8,3,4,20,17,6,9,12,42,28]
false.
```

RESULT: Thus the program to perform breadth first search has been implemented in Prolog successfully.

EX. No: 6b **DEPTH FIRST SEARCH**

AIM: Write a program to perform depth first search in Prolog.

PROCEDURE:

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Clauses in prolog:

```
%connected(+Start, +Goal, -Weight)
connected(1,7,1). connected(1,8,1).
connected(1,3,1). connected(7,4,1).
connected(7,20,1). connected(7,17,1).
connected(8,6,1). connected(3,9,1).
connected(3,12,1). connected(9,19,1).
connected(4,42,1). connected(20,28,1).
connected(17,10,1).


connected2(X,Y,D) :- connected(X,Y,D).

connected2(X,Y,D) :- connected(Y,X,D).


next_node(Current, Next, Path) :-     connected2(Current,
Next, _),     not(member(Next, Path)). depth_first(Goal,
Goal, _, [Goal]).
depth_first(Start, Goal, Visited, [Start|Path]) :-     next_node(Start,
Next_node, Visited),
   write(Visited), nl,

   depth_first(Next_node, Goal, [Next_node|Visited], Path).
```

Queries: To Test the Validity of the Clauses

```
?- depth_first(1, 28, [1], P).
```

OUTPUT

```
 [1]
[7,1]
[4,7,1]
[7,1]
[20,7,1]
P = [1, 7, 20, 28]
```

RESULT: Thus the program to perform depth first search has been implemented in Prolog successfully.

EX.NO:7 8-PUZZLE PROBLEM

AIM: To write a program for implementing the 8-puzzle problem python program.

PROCEDURE:

In this puzzle solution of the 8 puzzle problem is discussed.

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.

PYTHON CODE:

```
import copy
from heapq import heappush, heappop
n = 3
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]
class priorityQueue:
    def __init__(self):
        self.heap = []
    def push(self, key):
        heappush(self.heap, key)
    def pop(self):
        return heappop(self.heap)
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:

    def __init__(self, parent, mats, empty_tile_posi,
            costs, levels):
        self.parent = parent
        self.mats = mats
        self.empty_tile_posi = empty_tile_posi
        self.costs = costs
        self.levels = levels
    def __lt__(self, nxt):
        return self.costs < nxt.costs
def calculateCosts(mats, final) -> int:

    count = 0
    for i in range(n):
        for j in range(n):
            if ((mats[i][j]) and
                (mats[i][j] != final[i][j])):
                count += 1
    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:

    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)
    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes
# func to print the N by N matrix
def printMatsrix(mats):

    for i in range(n):
        for j in range(n):
            print("%d " % (mats[i][j]), end = " ")

        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()
def solve(initial, empty_tile_posi, final):
    pq = priorityQueue()

    # Creating the root node
```

```
costs = calculateCosts(initial, final)
root = nodes(None, initial,
            empty_tile_posi, costs, 0)

# Adding root to the list of live nodes
pq.push(root)
while not pq.empty():
    minimum = pq.pop()

    # If the min. is ans node
    if minimum.costs == 0:

        # Printing the path from the root to
        # destination;
        printPath(minimum)
        return

    # Generating all feasible children
    for i in range(n):
        new_tile_posi = [
            minimum.empty_tile_posi[0] + rows[i],
            minimum.empty_tile_posi[1] + cols[i], ]

        if isSafe(new_tile_posi[0], new_tile_posi[1]):

            # Creating a child node
            child = newNodes(minimum.mats,
                            minimum.empty_tile_posi,
                            new_tile_posi,
                            minimum.levels + 1,
                            minimum, final,)
```

```
                # Adding the child to the list of live nodes
                pq.push(child)

initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]
empty_tile_posi = [ 1, 2 ]

# Method call for solving the puzzle
solve(initial, empty_tile_posi, final)
```

OUTPUT:

```
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4

1   2   3
5   8   6
7   0   4

1   2   3
5   8   6
0   7   4
```

RESULT: Thus the 8-puzzle problem has been implemented in python successfully.

EXPNO:8a **SOLVING TOWER OF HANOI PROBLEM**

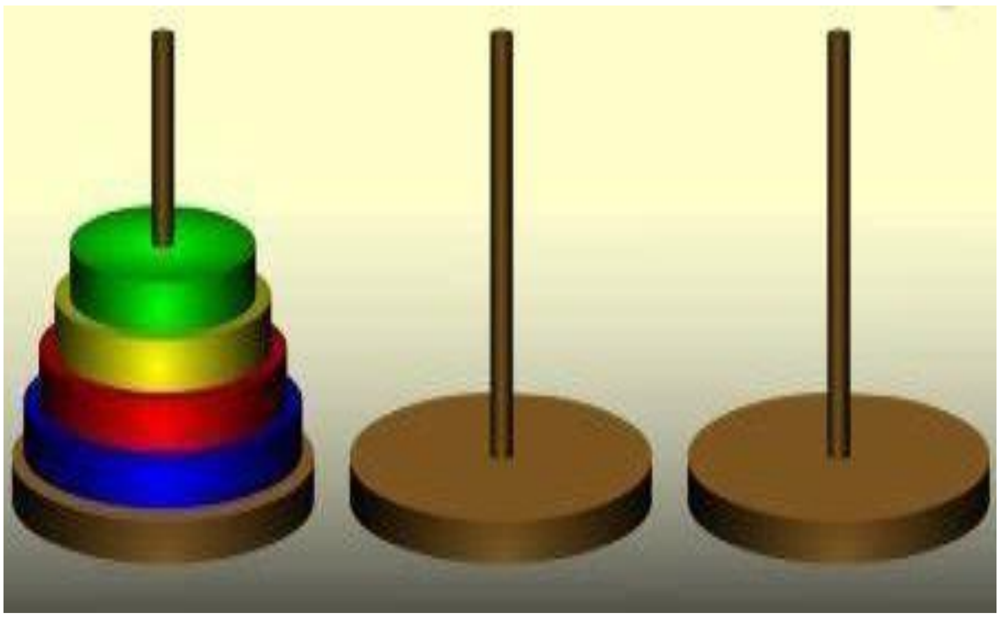
AIM: To write a Program to Implement Tower of Hanoi using python.

PROCEDURE:

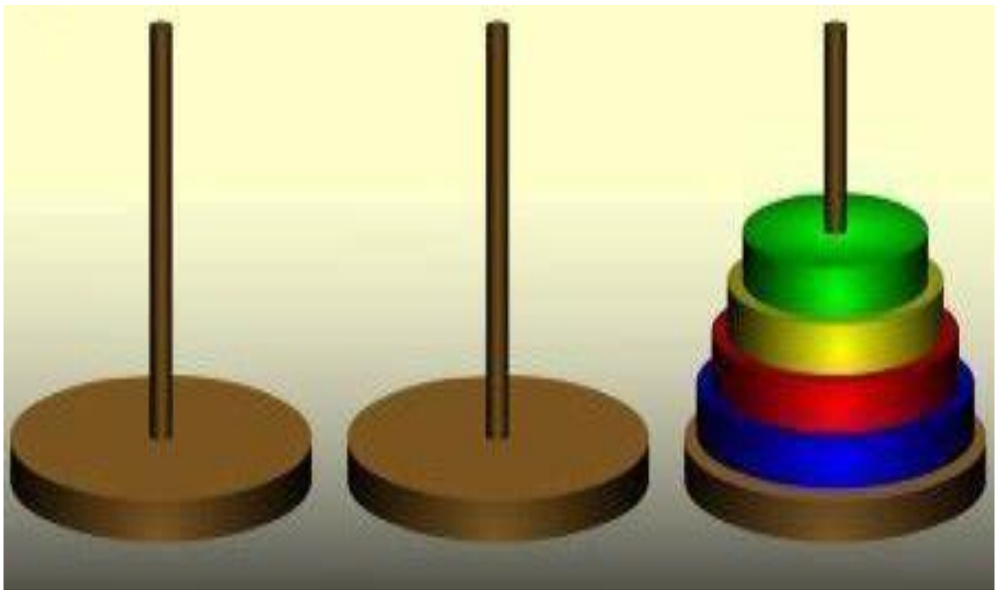
Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

INITIAL-



FINAL-



PROGRAM:

```
def TowerOfHanoi(n , source, destination, auxiliary):
            if n==1:
                      print ("Move disk 1 from source",source,"to destination",destination)
                       return
           TowerOfHanoi(n-1, source, auxiliary, destination)
           print ("Move disk",n,"from source",source,"to destination",destination)
            TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code
n = 4
TowerOfHanoi(n,'A','B','C')
```

OUTPUT:

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A Move disk 2 from
source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B Move disk 1 from
source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```

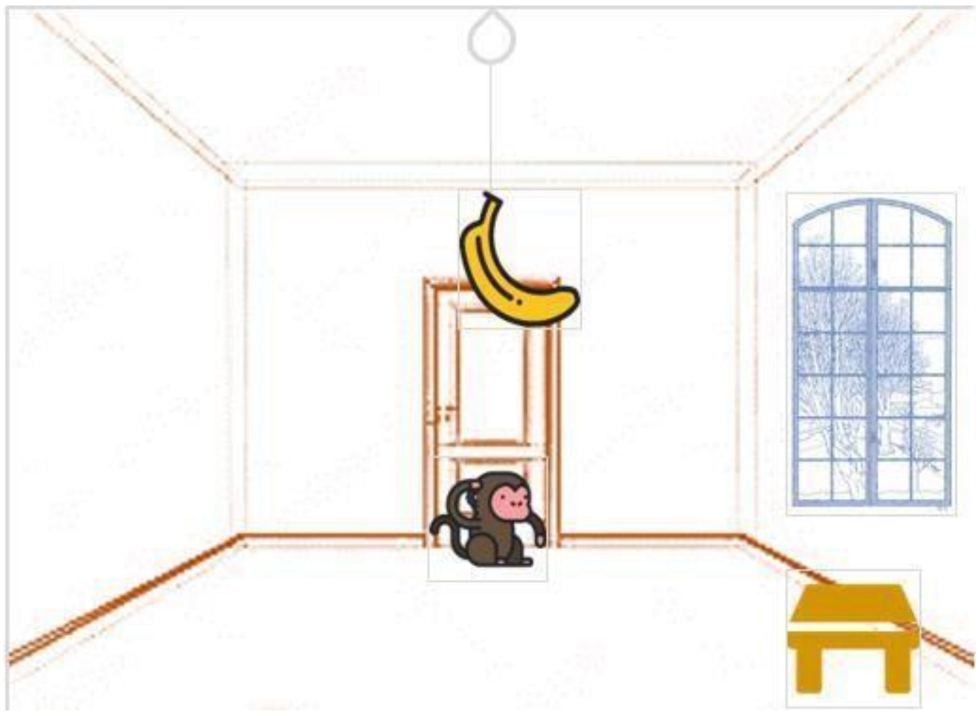
RESULT: Thus implementation of Tower of Hanoi using python was successfully executed.

EXPNO:8b **Solving Monkey Banana problem**

AIM: To write a program to solve monkey banana problem using python.

PROCEDURE:

The Monkey Banana problem is a classic puzzle in artificial intelligence that involves a monkey trying to collect bananas placed in a room while avoiding obstacles. The problem is typically solved using a search algorithm such as breadth-first search, depth-first search, or A* search. In this implementation, the State class represents a state of the problem, which consists of the position of the monkey and the positions of the boxes. The Problem class defines the problem, including the initial and goal states, and the is_goal_state and get_successors methods. The bfs_search function performs a breadth-first search of the problem, using a queue to keep track of the frontier and a set to keep track of the explored states.



PROGRAM:

```
from queue import Queue

# define the state of the problem class State:
    def __init__(self, monkey, boxes):
        self.monkey = monkey
        self.boxes = boxes

    def __eq__(self, other):        return self.monkey == other.monkey and self.boxes
== other.boxs
```

```
    def __hash__(self):         return hash((self.monkey,
tuple(self.boxes)))

    def __repr__(self):         return f"State({self.monkey},
{self.boxes})"

# define the problem class Problem:     def __init__(self,
initial_state, goal_state):         self.initial_state =
initial_state
        self.goal_state = goal_state

    def is_goal_state(self, state):         return state
== self.goal_state

    def get_successors(self, state):
        monkey_row, monkey_col = state.monkey
        boxes = state.boxes
        successors = []

        for row, col in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            new_row, new_col = monkey_row + row, monkey_col + col

            if (new_row, new_col) in boxes:
                box_index = boxes.index((new_row, new_col))
                new_box_row, new_box_col = new_row + row, new_col + col

                if (new_box_row, new_box_col) not in boxes:
                    new_boxes = boxes.copy()                    new_boxes[box_index] =
(new_box_row, new_box_col)                    new_state = State((new_row,
new_col), new_boxes)
                    successors.append(new_state)            else:                if
(new_row, new_col) not in boxes:                    new_state =
State((new_row, new_col), boxes)
successors.append(new_state)

        return successors

# define the breadth-first search algorithm def
bfs_search(problem):
    frontier = Queue()
    explored = set()

    frontier.put(problem.initial_state)

    while not frontier.empty():
        state = frontier.get()

        if problem.is_goal_state(state):
            return state
        explored.add(state)

        for successor in problem.get_successors(state):             if
successor not in explored:                  frontier.put(successor)

    return None

# define the initial and goal states initial_state = State((0, 0),
[(2, 1), (3, 3), (4, 1)])
goal_state = State((4, 4), [(2, 1), (3, 3), (4, 1)])

# define the problem and solve it problem =
Problem(initial_state, goal_state) solution =
bfs_search(problem)

# print the solution if solution is
None:
    print("No solution found.") else:
    print("Solution found:")     print(solution)
```

OUTPUT:

```
Solution found:
State((4, 4), [(2, 1), (3, 3), (4, 1)])
```

RESULT: Thus to solve monkey banana problem using python was successfully executed.

EX.NO:9 ROBOT (TRAVERSAL) PROBLEM USING MEANS END ANALYSIS

AIM: To write a program to solve robot traversal problem using end mean analysis in python.

PROCEDURE:

A robot is placed on a grid with coordinates (x, y). The robot can move up, down, left, or right, but cannot move outside the grid. Given a target position (x_target, y_target), write a Python function to determine the sequence of moves the robot should make to reach the target position.

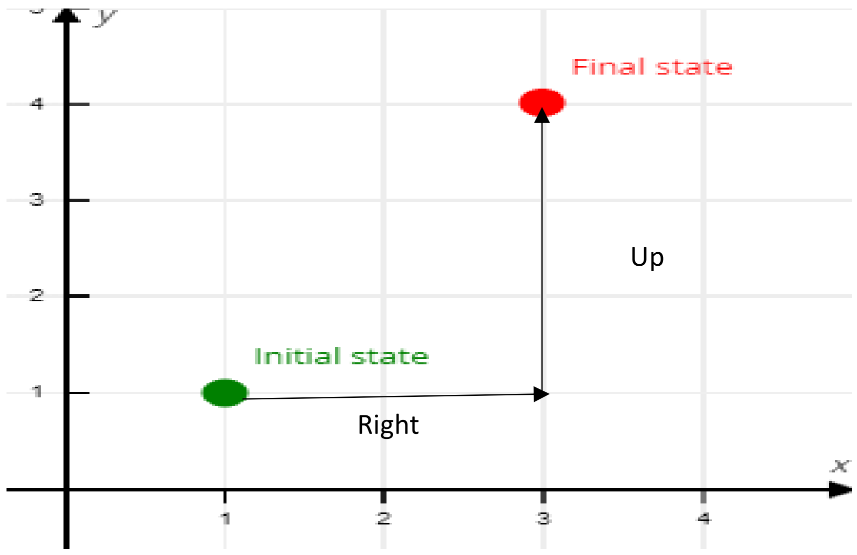
Now, we can break down the problem into subgoals:

1. Check if the current position is the same as the target position. If so, we're done.
2. If not, determine the direction the robot needs to move to get closer to the target.
3. Move the robot in that direction.
4. Repeat steps 1-3 until the robot reaches the target position.

We can use Means-End Analysis to determine the subgoals and how they can be achieved. The Means-End Analysis process involves the following steps:

1. Define the desired end state: the robot is at the target position.
2. Define the current state: the robot is at its current position.
3. Identify the differences between the desired end state and the current state.
4. Identify the subgoals necessary to bridge the gap between the current state and the desired end state.
5. Determine the actions required to achieve each subgoal.
6. Repeat steps 4-5 until the desired end state is reached.

GRAPH:



PYTHON CODE:

```
def robot_traversal(x, y, x_target, y_target):

    # Define the desired end state
```

```
end_state = (x_target, y_target)

# Define the current state
current_state = (x, y)

# Check if the robot is already at the target position
if current_state == end_state:
    return []

# Identify the differences between the desired end state and the current state
dx = x_target - x
dy = y_target - y

# Identify the subgoals necessary to bridge the gap between the current state and the desired end state
subgoals = []
if dx > 0:
    subgoals.append((1, 0))
elif dx < 0:
    subgoals.append((-1, 0))
if dy > 0:
    subgoals.append((0, 1))
elif dy < 0:
    subgoals.append((0, -1))

# Determine the actions required to achieve each subgoal
actions = { (1, 0): 'right', (-1, 0): 'left', (0, 1): 'up', (0, -1): 'down' }
moves = []
for subgoal in subgoals:
    dx, dy = subgoal
    move = actions[(dx, dy)]
    moves.append(move)
    return moves


x = 1

y = 1

x_target = 3

y_target = 4


moves = robot_traversal(x, y, x_target, y_target)


print("Moves to reach target:", moves)
```

OUTPUT:

```
Moves to reach target: ['right', 'up']
```

RESULT: Thus the program to solve robot traversal problem using end mean analysis has been implemented in python successfully.

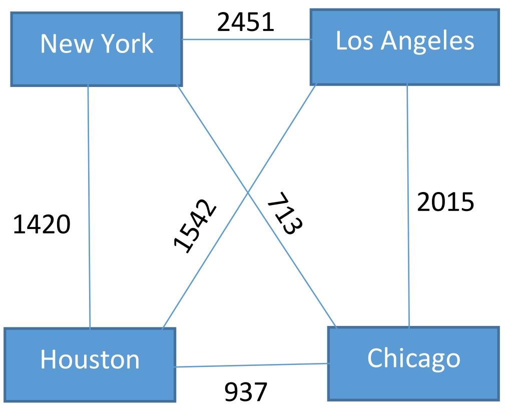
EX.NO:10 TRAVELING SALESMAN PROBLEM

AIM: To write a program to solve Traveling Salesman Problem in python.

PROCEDURE:

The Travelling Salesman Problem (TSP) is a classic optimization problem in computer science. The goal of the TSP is to find the shortest possible route that visits every city exactly once and returns to the starting city.

GRAPH:



PYTHON CODE:

```
import itertools


class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.edges = {}
        for vertex in vertices:
            self.edges[vertex] = {}


    def add_edge(self, src, dest, weight):
        self.edges[src][dest] = weight
        self.edges[dest][src] = weight
```

```
def tsp(graph, start):
    # Create a list of all possible city permutations
    permutations = itertools.permutations(graph.vertices)

    # Initialize the shortest route to infinity
    shortest_route = float('inf')

    # Loop through each permutation and calculate its total distance
    for permutation in permutations:
        distance = 0
        for i in range(len(permutation)-1):
            distance += graph.edges[permutation[i]][permutation[i+1]]

        # Add the distance back to the starting city
        distance += graph.edges[permutation[-1]][permutation[0]]

        # Update the shortest route if this one is shorter
        if distance < shortest_route:
            shortest_route = distance
            best_permutation = permutation

    # Return the shortest route and the best permutation of cities
    return shortest_route, best_permutation

# Example usage:
city_names = ['New York', 'Los Angeles', 'Chicago', 'Houston']
city_graph = Graph(city_names)
city_graph.add_edge('New York', 'Los Angeles', 2451)
city_graph.add_edge('New York', 'Chicago', 713)
city_graph.add_edge('New York', 'Houston', 1420)
city_graph.add_edge('Los Angeles', 'Chicago', 2015)
city_graph.add_edge('Los Angeles', 'Houston', 1542)
city_graph.add_edge('Chicago', 'Houston', 937)


start_city = 'New York'
shortest_route, best_permutation = tsp(city_graph, start_city)
print("Shortest route:", shortest_route)
print("Best permutation:", best_permutation)
```

OUTPUT:

```
Shortest route: 5643
Best permutation: ('New York', 'Los Angeles', 'Houston', 'Chicago')
```

RESULT: Thus the program to Traveling Salesman Problem has been implemented in python successfully.