Q-1).Write a program in Prolog to implement TowerOfHanoi(N) where N represents the number of disks.

**Code**:-

write\_move(N,X,Y) :- write('Move disk'), write(N),

write(' from '), write(X), write(' to '),

write(Y),nl.

move(1, X, Y, \_) :- write\_move(1,X,Y).

move(N, X, Y, Z) :- N > 1,

M is N - 1,

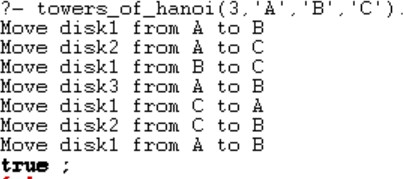
move(M, X, Z, Y), % Move smaller disks to auxiliary rod write\_move(N,X,Y),

% Pr

move(M, Z, Y, X). % Move smaller disks to target rod using auxiliary rod

towers\_of\_hanoi(N, Source, Target, Aux) :- move(N, Source, Target, Aux).

## Output:



Q-2). Write a program to implement the Hill climbing search algorithm in Prolog.

**Code**:-

% Define the tree structure

% tree(Node, LeftSubtree, RightSubtree) tree(a, b, c).

tree(b, d, e).

tree(c, f, g).

tree(d, nil, nil).

tree(e, nil, nil).

tree(f, nil, nil).

tree(g, nil, nil).

% Define the goal node goal(d).

% Define heuristic values for each node (depth of the node) heuristic(a, 4).

heuristic(b, 5).

heuristic(c, 2).

heuristic(d, 1).

heuristic(e, 1).

heuristic(f, 1).

heuristic(g, 1).

% hill\_climbing/2 is the main predicate for Hill Climbing search hill\_climbing(State, Goal) :-

heuristic(State, H), % Calculate the heuristic value for the current state hill\_climbing(State, Goal, H). % Call the helper predicate with the initial heuristic

value

% Base case: if the current state matches the goal state, succeed hill\_climbing(State, Goal, \_) :-

goal(State),

format('Goal state reached: ~w~n', [State]).

% hill\_climbing/3 is the helper predicate for Hill Climbing search hill\_climbing(State, Goal, CurrentH) :-

% Traverse left subtree tree(State, Left, \_),

% If the left subtree exists and its heuristic value is less than the current state,

% continue hill climbing from the left subtree Left \= nil,

heuristic(Left, LeftH), LeftH < CurrentH,

format('Moving from ~w to ~w~n', [State, Left]), hill\_climbing(Left, Goal, LeftH),

!.

hill\_climbing(State, Goal, CurrentH) :-

% Traverse right subtree tree(State, \_, Right),

% If the right subtree exists and its heuristic value is less than the current state,

% continue hill climbing from the right subtree Right \= nil,

heuristic(Right, RightH), RightH < CurrentH,

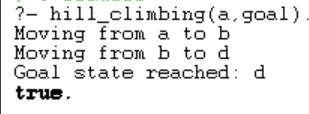
format('Moving from ~w to ~w~n', [State, Right]), hill\_climbing(Right, Goal, RightH),

!.

% If no better move is available hill\_climbing(State, \_, \_) :-

format('No better move from ~w~n', [State]).

## Output:



Q-3). Write a program to implement the Best first search algorithm in Prolog.

**Code**:-

% Define the facts and rules for the tree structure and search algorithm

% Sample tree structure (nodes connected by edges)

edge(a, b, 3). % edge(node1,node2,distance) distance is optional edge(a, c, 4).

edge(b, d, 5).

edge(b, e, 2).

edge(c, f, 6).

edge(c, g, 7).

% Define heuristic values for each node (h(n)) heuristic(a, 10).

heuristic(b, 8).

heuristic(c, 6).

heuristic(d, 4).

heuristic(e, 6).

heuristic(f, 3).

heuristic(g, 2).

search\_path(Start, Closed) :-

% Step 1: Initialize open list with the start node search\_path\_helper([Start], [], Closed).

% Base case: If open list is empty, return the closed list search\_path\_helper([], Closed, Closed).

search\_path\_helper(Open, Closed, FinalClosed) :-

% Step 2: Sort the open list by heuristic value

% predefined predsort predicate

% It is a built-in Prolog predicate that sorts a list according to a custom comparison % predicate. It takes three arguments:Comparison Predicate,Unsorted List,Sorted List.

predsort(compare\_heuristic, Open, SortedOpen),

% Get the node with the lowest heuristic value [Current|RestOpen] = SortedOpen,

% Step 3: Find nodes connected to the current node by edges

% findall/3 is a built-in predicate in Prolog used to find all solutions that satisfy % a goal. It takes three arguments:Template,Goal,Solutions.

findall(Next, edge(Current, Next, \_), Neighbors),

% Step 4: Append neighbors to open list append(RestOpen, Neighbors, UpdatedOpen),

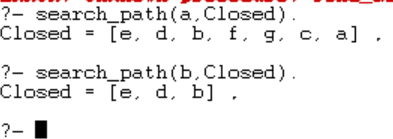
% Recursive call

search\_path\_helper(UpdatedOpen, [Current|Closed], FinalClosed).

% Compare nodes based on their heuristic values compare\_heuristic(Order, Node1, Node2) :-

heuristic(Node1, Heuristic1), heuristic(Node2, Heuristic2), compare(Order, Heuristic1, Heuristic2).

## Output:-



Q-4). Write a program to implement A\* search algorithm in Prolog.

#### Code:-

% Edges between nodes: edge(From, To, Distance) edge(k, a, 1).

edge(k, b, 4).

edge(a, b, 2).

edge(a, c, 5).

edge(a, d, 12).

edge(b, c, 2).

edge(c, d, 3).

% Define heuristic values for each node (h(n)) heuristic(k, 7).

heuristic(a, 6).

heuristic(b, 2).

heuristic(c, 1).

heuristic(d, 0).

% Predicate to find all paths from K to D along with their total cost find\_all\_paths\_with\_cost(K, D, Paths) :-

findall(Path-Cost, path\_with\_cost(K, D, [], Path, Cost), Paths).

% Base case: Reached destination node path\_with\_cost(D, D, Visited, Path, Cost) :-

reverse([D|Visited], Path), calculate\_path\_cost(Path, Cost).

% Recursive case: Explore paths to neighbors path\_with\_cost(Current, D, Visited, Path, Cost) :-

edge(Current, Next, Distance), % Find a neighbor node and its distance

\+ member(Next, Visited), % Ensure the neighbor is not already visited

path\_with\_cost(Next, D, [Current|Visited], Path, RemainingCost),

Cost is RemainingCost + Distance. % Accumulate distance along the path

% Calculate the total cost of a path

calculate\_path\_cost([], 0). % Base case: Total cost of an empty path is 0 calculate\_path\_cost([\_], 0). % Base case: Total cost of a single-node path is 0 calculate\_path\_cost([Node1, Node2|Rest], TotalCost) :-

edge(Node1, Node2, Distance), calculate\_path\_cost([Node2|Rest], RemainingCost), TotalCost is RemainingCost + Distance.

% Predicate to find the path with the minimum cost minimum\_path(K, D, MinPath) :-

find\_all\_paths\_with\_cost(K, D, Paths), % Find all paths min\_member(Path-Cost, Paths), % Find the path with minimum cost

**Output:-**



### Q-5) Write a program to implement the min-max search algorithm in Prolog.

#### Code:-

tree(3, [

tree(5, [

tree(9, []),

tree(8, []),

tree(6, [])

]),

tree(2, [

tree(7, []),

tree(1, []),

tree(4, [])

]),

tree(6, [

tree(1, []),

tree(2, []),

tree(8, [])

])

]).

% minimax(Tree, Depth, Player, Value) minimax(tree(Value, []), \_, \_, Value) :- !. minimax(tree(\_, Children), Depth, Player, Value) :-

Depth > 0,

NextDepth is Depth - 1, findall(ChildValue, (

member(Child, Children), other\_player(Player, OtherPlayer),

minimax(Child, NextDepth, OtherPlayer, ChildValue)

), ChildValues), best\_value(ChildValues, Player, Value).

% Find the best value for the current player

% best\_value(Values, Player, BestValue)

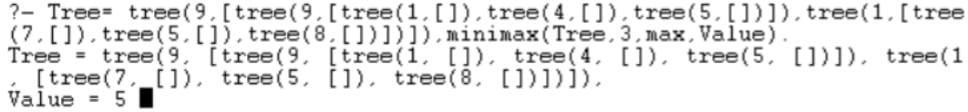
best\_value(Values, max, BestValue) :- max\_list(Values, BestValue).

best\_value(Values, min, BestValue) :- min\_list(Values, BestValue).

% Predicate to determine the other player

% other\_player(Player, OtherPlayer) other\_player(max, min). other\_player(min, max).

#### Output:-



Q-6).Write a program to solve the Water-Jug Problem in Prolog.

#### Code:-

% Define the initial state and the goal state initial\_state((0, 0)).

goal\_state((4, \_)).

% Define the actions possible in the problem action((Jug1, Jug2), fill\_jug1, (5, Jug2)) :-

Jug1 < 5.

action((Jug1, Jug2), fill\_jug2, (Jug1, 3)) :- Jug2 < 3.

action((Jug1, Jug2), empty\_jug1, (0, Jug2)) :- Jug1 > 0.

action((Jug1, Jug2), empty\_jug2, (Jug1, 0)) :- Jug2 > 0.

action((Jug1, Jug2), pour\_jug1\_to\_jug2, (NewJug1, NewJug2)) :- Jug1 > 0,

Total is Jug1 + Jug2, NewJug2 is min(Total, 3),

NewJug1 is Jug1 - (NewJug2 - Jug2).

action((Jug1, Jug2), pour\_jug2\_to\_jug1, (NewJug1, NewJug2)) :- Jug2 > 0,

Total is Jug1 + Jug2, NewJug1 is min(Total, 5),

NewJug2 is Jug2 - (NewJug1 - Jug1).

% Define the predicate to solve the problem using depth-first search solve(State, \_, []) :- goal\_state(State).

solve(State, Visited, [Action|Rest]) :- action(State, Action, NextState),

\+ member(NextState, Visited), solve(NextState, [NextState|Visited], Rest).

**Output:-**



### Q-7) Implement sudoku problem (minimum 9x9 size) using constraint satisfaction in Prolog.

#### Code:-

:- 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 = [A,B,C,D,E,F,G,H,I],

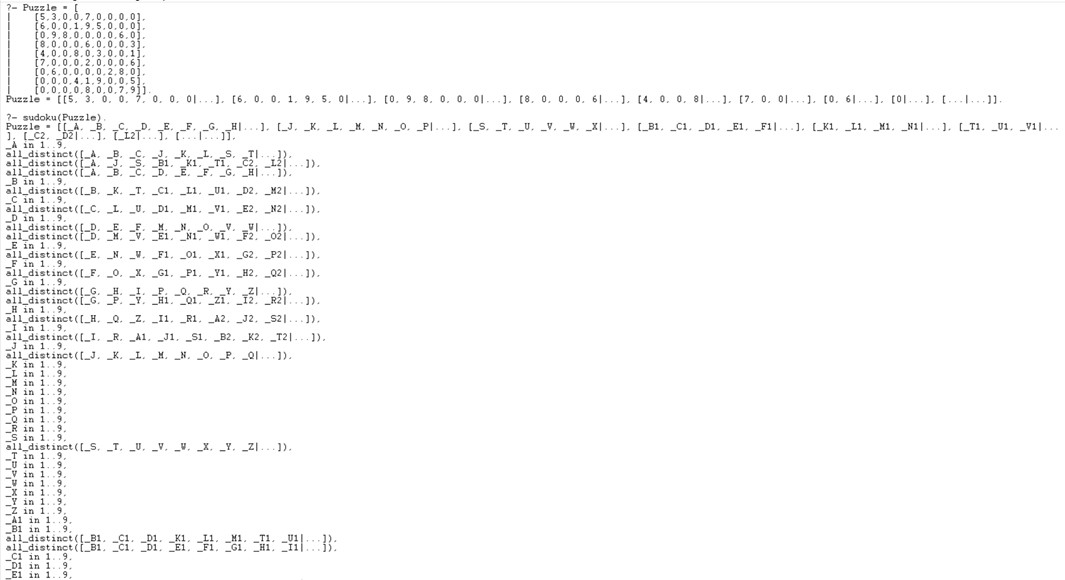
blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

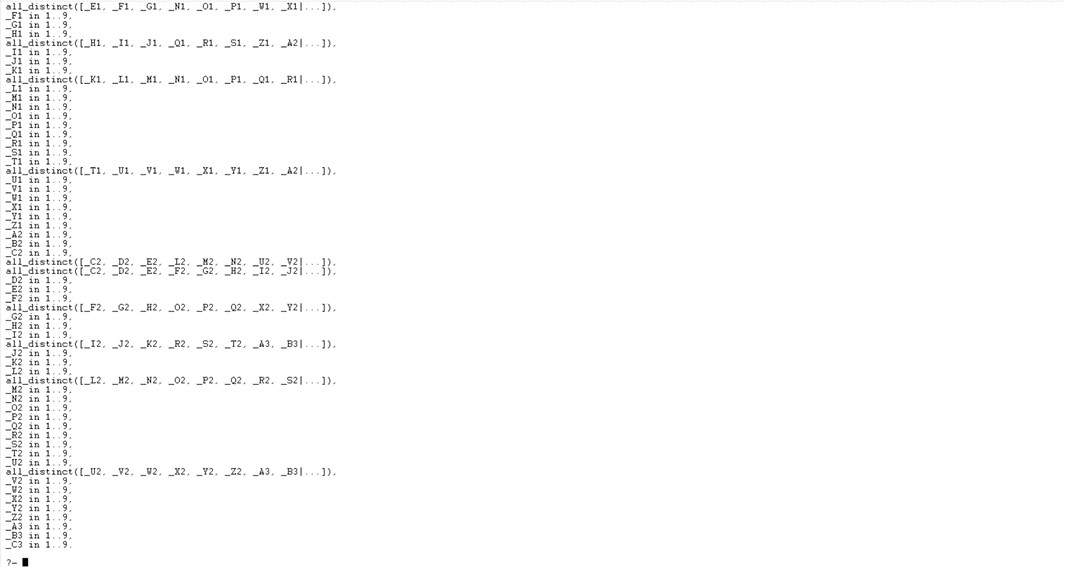
blocks([], [], []).

blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-

all\_distinct([A,B,C,D,E,F,G,H,I]), blocks(Bs1, Bs2, Bs3).

**Output:**





Q8. Write a program to implement the family tree and demonstrate the family relationship.

## Code:-

/\* Define facts about family relationships \*/ male(john).

male(tom). male(peter). male(bob).

female(lisa). female(anna). female(susan). female(emily).

parent(john, tom). parent(john, lisa). parent(lisa, anna). parent(lisa, susan). parent(tom, peter). parent(anna, emily). parent(susan, bob).

/\* Define rules to infer other relationships \*/ father(Father, Child) :-

male(Father), parent(Father, Child).

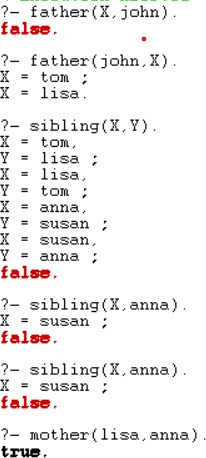
mother(Mother, Child) :- female(Mother), parent(Mother, Child).

sibling(X, Y) :-

parent(Z, X),

parent(Z, Y), X \= Y.

#### Output:-



Q-9).write a prolog program to implement knowledge representation using frames with appropriate examples.

# Code:-

% Define frames for different types of vehicles frame(vehicle,

[ slots: [type, brand, model, color, year]

]).

% Specific instances of vehicles vehicle(car,

[ type: car, brand: honda, model: civic, color: blue, year: 2018

]).

vehicle(truck,

[ type: truck, brand: ford, model: f150, color: black, year: 2020

]).

% Define a predicate to query information about a vehicle vehicle\_info(Type, Info) :-

vehicle(Type, Info).

#### Output:-



Q-10).Write a prolog program to implement conc(L1,L2,L3) where L2 is the list to be appended with L1 to get the resulted list L3.

#### Code:-

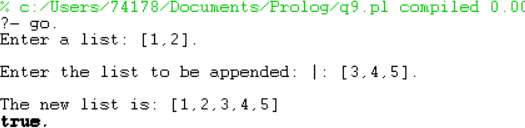
/\*implementation of concat\*/

go:- write('Enter a list: '),read(L1), nl, write('Enter the list to be appended: '),read(L2),nl, conc(L1,L2,L3),

write('The new list is: '),write(L3). conc([],L,L).

conc([H|T1],L2,[H|T3]):- conc(T1,L2,T3).

# Output:-



Q-11).Write a prolog program to implement reverse(L,R) where List L the original list and R is reversed List.

#### Code:-

/\*implementation of reverse\*/ start:-write('Enter The List: '), read(L),

reverselist(L,R),

write('The Reversed List Is: '), write(R).

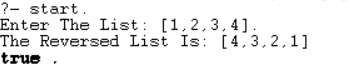
reverselist([],[]).

reverselist([H],[H]).

reverselist([H|T],R):-reverselist(T,R1),conc(R1,[H],R). conc([],L1,L1).

conc([H|T],L2,[H|L3]):-conc(T,L2,L3).

#### Output:-



Q-12).Write a Prolog Program to generate parse tree of a given sentence assuming the grammar required for parsing.

#### Code:-

% Defining the grammar rules.

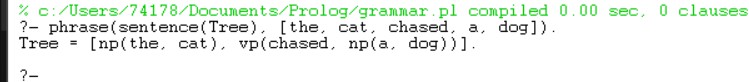
sentence(Tree) --> noun\_phrase(NP), verb\_phrase(VP), {Tree = [NP, VP]}. noun\_phrase(Tree) --> determiner(Det), noun(N), {Tree = np(Det, N)}. verb\_phrase(Tree) --> verb(V), noun\_phrase(NP), {Tree = vp(V, NP)}.

% Lexical rules. determiner(the) --> [the]. determiner(a) --> [a].

noun(cat) --> [cat].

noun(dog) --> [dog]. verb(chased) --> [chased]. verb(ate) --> [ate].

#### Output:-



Q-13).Write a Prolog program to recognize context free grammar aⁿbⁿ.

#### Code:-

% Defining the grammar rules. s --> [].

s --> [a], s, [b].

% Predicate to recognize strings matching the grammar recognize(Input) :- phrase(s, Input).

**Output:-**

