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),

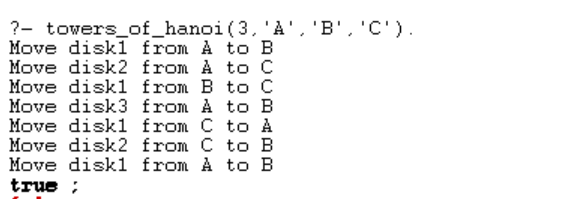
% Pr

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),

!.

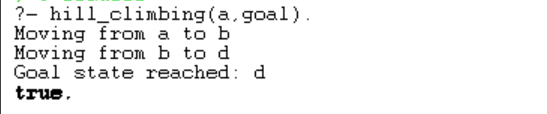
% If no better move is available

hill\_climbing(State, \_, \_) :-

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

fail.

**Output**:



Q-3). Write a program to implement the Best first 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(f).

% 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).

% best\_first\_search/3 is the main predicate for Best-First Search

best\_first\_search(Start, Goal, Path) :-

best\_first\_search([[Start]], Goal, [], Path).

% Base case: If the OPEN list is empty, fail

best\_first\_search([], \_, \_, \_) :-

format('No more nodes to explore. Best-First Search failed.~n'),

fail.

% Base case: If the current node matches the goal state, succeed

best\_first\_search([[Goal|Path]|\_], Goal, \_, Path) :-

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

% Step 1: Choose an initiating node (suppose ‘n’) and place it in the OPEN list.

% Step 2: In case the initiating node is empty, you must stop and return to failure.

% Step 3: Eliminate the node from the OPEN list and place it on the CLOSE list. Here, the node is the lowest value of h(n), i.e., heuristic function.

best\_first\_search([[Node|Path]|Paths], Goal, Closed, FinalPath) :-

delete(Paths, [Node|Path], RemainingPaths),

% Step 4: Expand the node and create its successors.

findall([Successor, Node|Path],

(tree(Node, Left, Right),

member(Successor, [Left, Right]),

not(member(Successor, Path)),

not(member([Successor|\_], Closed))),

Successors),

% Step 5: Check each successor to see whether they are leading to the goal.

(member([Goal|FinalPath], Successors) ->

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

% Step 7: The algorithm analyzes every successor for the evaluation function f(n). Later, it examines whether the nodes are in the OPEN or CLOSED list. In case they do not find the node in either list, it adds them to the OPEN list.

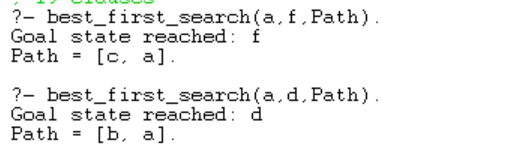
; append(RemainingPaths, Successors, NewPaths),

sort(NewPaths, SortedPaths),

best\_first\_search(SortedPaths, Goal, [[Node|Path]|Closed], FinalPath)).

% Query: best\_first\_search(Start, Goal, Path).

**Output:-**

****

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

**Code:-**

% Define the start state

start\_state(a).

% Define the goal state

goal\_state(d).

% Define the successor states (transitions between states) and their costs

successor(a, b, 2).

successor(b, c, 3).

successor(c, d, 1).

% A\* search algorithm

astar(Path) :-

start\_state(Start),

astar([(0, Start, [])], Path).

% Base case: Goal reached

astar([(Cost, State, Path)|\_], Path) :-

goal\_state(State),

!,

format('Goal reached with cost ~w.~n', [Cost]).

% Recursive step

astar([(\_, State, Path)|Queue], FinalPath) :-

expand(State, Path, Children),

add\_to\_queue(Children, Queue, NewQueue),

astar(NewQueue, FinalPath).

% Expand a node to its children

expand(State, Path, Children) :-

findall((NewCost, NextState, [NextState|Path]),

(successor(State, NextState, StepCost), heuristic(NextState, H), NewCost is H + StepCost),

Children).

% Add children to the queue while maintaining the priority order

add\_to\_queue([], Queue, Queue).

add\_to\_queue([Child|Children], Queue, NewQueue) :-

insert\_ordered(Child, Queue, Queue1),

add\_to\_queue(Children, Queue1, NewQueue).

% Insert an element into a sorted queue based on its priority

insert\_ordered(X, [], [X]).

insert\_ordered(X, [Y|Ys], [X,Y|Ys]) :-

X = (CostX, \_, \_),

Y = (CostY, \_, \_),

CostX =< CostY.

insert\_ordered(X, [Y|Ys], [Y|Zs]) :-

insert\_ordered(X, Ys, Zs).

% Define the heuristic function (distance to the goal)

heuristic(a, 3).

heuristic(b, 2).

heuristic(c, 1).

heuristic(d, 0).

**Output:-**



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

**Code:-**

% Define the game state

state([\_, \_, \_, \_, \_, \_, \_, \_, \_]).

% Define the players

player(x).

player(o).

% Define the moves

move(state([\_, B, C, D, E, F, G, H, I]), 1, state([x, B, C, D, E, F, G, H, I])) :- player(x).

move(state([A, \_, C, D, E, F, G, H, I]), 2, state([A, x, C, D, E, F, G, H, I])) :- player(x).

% ... define the rest of the moves for player x

move(state([\_, B, C, D, E, F, G, H, I]), 1, state([o, B, C, D, E, F, G, H, I])) :- player(o).

move(state([A, \_, C, D, E, F, G, H, I]), 2, state([A, o, C, D, E, F, G, H, I])) :- player(o).

% ... define the rest of the moves for player o

% Define the utility function

utility(state([A, B, C, D, E, F, G, H, I]), Value) :-

(win(state([A, B, C, D, E, F, G, H, I]), x) -> Value = 1;

win(state([A, B, C, D, E, F, G, H, I]), o) -> Value = -1;

draw(state([A, B, C, D, E, F, G, H, I])) -> Value = 0).

% Define the minimax algorithm

minimax(State, Value) :-

utility(State, Value).

minimax(State, Value) :-

findall(NewState, move(State, \_, NewState), NewStates),

best(NewStates, Value).

best([State], Value) :-

minimax(State, Value).

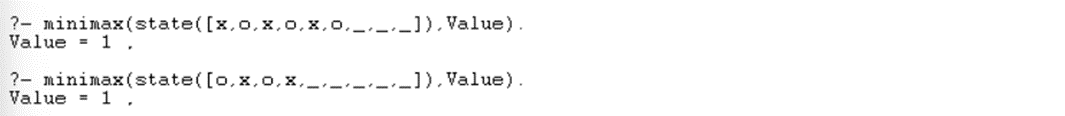
best([State|States], Value) :-

minimax(State, V1),

best(States, V2),

(player(x) -> max(V1, V2, Value); min(V1, V2, Value)).

**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).

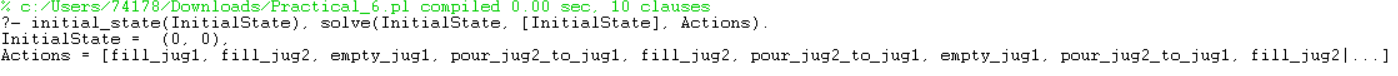
solve(State, Visited, [Action|Rest]) :-

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],

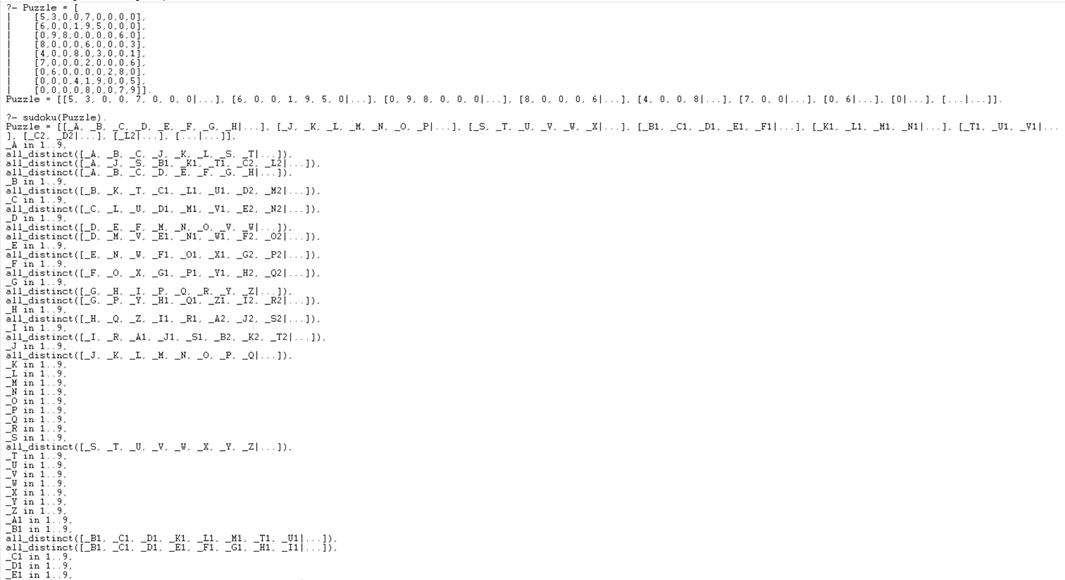
blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

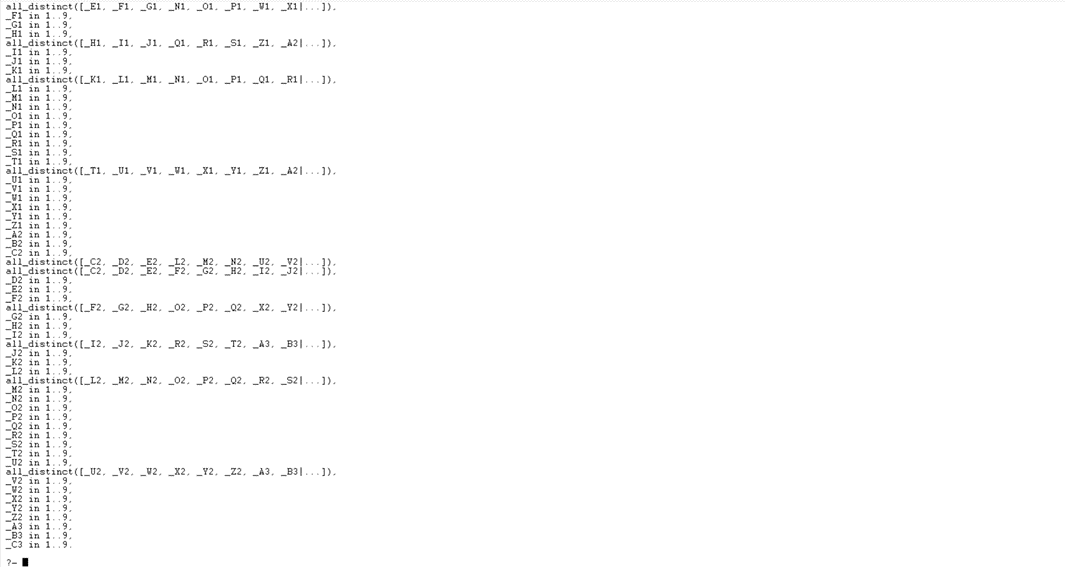
blocks([], [], []).

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).

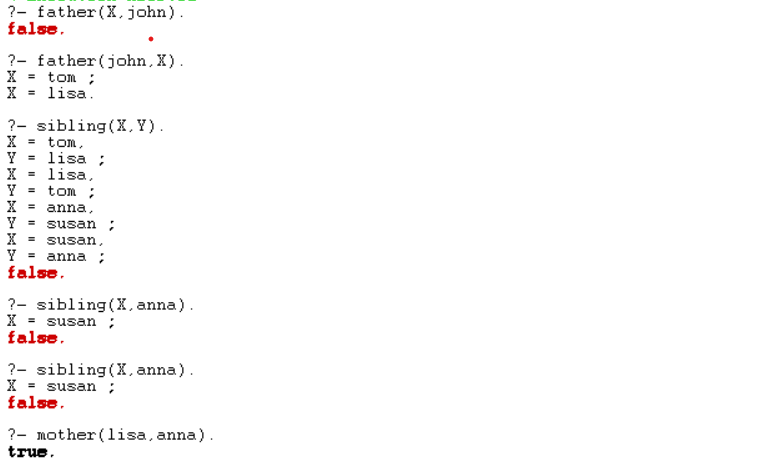
sibling(X, Y) :-

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,

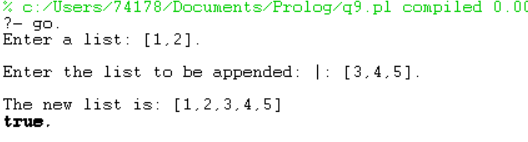
conc(L1,L2,L3),

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([],[]).

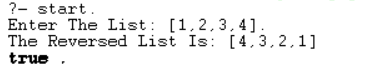
reverselist([H],[H]).

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].

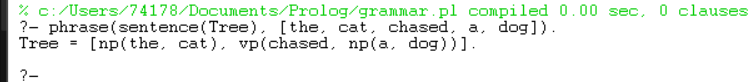
noun(cat) --> [cat].

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:-**

