RAMANUJAN COLLEGE

UNIVERSITY OF DELHI

NAME: SHELTON ROGÉRIO BATINE

ROLL N*:24570077

COURSE: BSC (HONS) COMPUTER SCIENCE

Artificial Intelligence

Practical File

Submitted to: Ms.Bhavya Ahuja **Submitted by: Shelton Batine**

Question 1 Subject:

Write a program in Prolog to implement TowerOfHanoi(N) where N represents the

number of disks

```
 % towerOfHanoi(N) :- Solves the Tower of Hanoi puzzle for N disks
% using pegs A (source), B (auxiliary), and C (destination).

towerOfHanoi(N) :-
    move(N, a, c, b).

% move(N, Source, Destination, Auxiliary)
% Base case: moving 1 disk
move(1, Source, Destination, _) :-
    write('Move disk from '),
    write(Source),
    write(' to '),
    write(Destination),
    nl.

% Recursive case: move N disks
move(N, Source, Destination, Auxiliary) :-
    N > 1,
    N1 is N - 1,
    move(N1, Source, Auxiliary, Destination),
    move(1, Source, Destination, _),
    move(N1, Auxiliary, Destination, Source).
```

Output:

```
?-  towerOfHanoi(3).
ERROR: Unknown procedure: (?-)/1
ERROR:   ?- is the Prolog prompt
ERROR:   See FAQ at https://www.swi-prolog.org/FAQ/ToplevelMode.html
ERROR: In:
ERROR:   [11] throw(error(existence_error(procedure,...),_15880))
ERROR:    [8] correct_goal((?-towerOfHanoi(3)),user,[],_15916) at c:/program files/swipl/boot/dwim.pl:92
ERROR:
ERROR: Note: some frames are missing due to last-call optimization.
ERROR: Re-run your program in debug mode (:- debug.) to get more detail.
?- Move disk from a to c
Move disk from a to b
Move disk from c to b
Move disk from a to c
Move disk from b to a
Move disk from b to c
Move disk from a to c
true
```

Question 2

Write a program to implement the Hill climbing search algorithm in Prolog.

```
% ----------------------------------------
% HILL CLIMBING SEARCH ALGORITHM (Prolog)
% ----------------------------------------

% hill_climb(StartState, GoalState)
% Returns the best state reachable using hill climbing

hill_climb(Start, Goal) :-
    hill_climb_helper(Start, Goal, []).

% Base case: if current state is the goal
hill_climb_helper(State, State, _) :-
    write('Reached goal: '), write(State), nl.

% Recursive case: choose the best neighbor
hill_climb_helper(State, Goal, Visited) :-
    findall(Next, neighbor(State, Next), Neighbors),
    evaluate_list(Neighbors, Scored),
    best(Scored, BestState, _BestScore),
    \+ member(BestState, Visited),
    write('Moving to: '), write(BestState), nl,
    hill_climb_helper(BestState, Goal, [State | Visited]).

% If no better neighbor exists
hill_climb_helper(State, _, _) :-
    write('No better neighbors. Local maximum at: '), write(State), nl.

% ----------------------------------------------------
% Evaluating nodes (example uses heuristic/score value)
% ----------------------------------------------------

evaluate_list([], []).
evaluate_list([State|Rest], [(State,Score)|RestScores]) :-
    heuristic(State, Score),
% Pick the state with the highest score
best([(S,Score)], S, Score).
best([(S,Score)|Rest], BestState, BestScore) :-
    best(Rest, RestState, RestScore),
    ( Score > RestScore ->
        BestState = S,
        BestScore = Score
    ;
        BestState = RestState,
        BestScore = RestScore
    ).
```

Output:

```
[1]  ?-  hill_climb(3, 10).
ERROR: SMoving to: 4
Moving to: 5
Moving to: 6
Moving to: 7
Moving to: 8
Moving to: 9
Moving to: 10
Reached goal: 10
true
```

Question 3

Write a program to implement the Best first search algorithm in Prolog.

```
% ------------------------------------------
% BEST FIRST SEARCH (Greedy Search) IN PROLOG
% ------------------------------------------

% best_first(Start, Goal, Path)
best_first(Start, Goal, Path) :-
    best_first_search([[Start]], Goal, Path).

% If we reach the goal: first node of path is the goal
best_first_search([[Goal | Rest] | _], Goal, Path) :-
    reverse([Goal | Rest], Path),
    write('Goal reached! Path: '), write(Path), nl.

% Main loop
best_first_search([CurrentPath | OtherPaths], Goal, FinalPath) :-
    CurrentPath = [CurrentNode | _],
    findall([NextNode | CurrentPath],
            (edge(CurrentNode, NextNode),
             \+ member(NextNode, CurrentPath)),
            NewPaths),
    evaluate_paths(NewPaths, ScoredPaths),
    merge_paths(OtherPaths, ScoredPaths, Merged),
    best_first_search(Merged, Goal, FinalPath).

% ------------------------------------------
% Heuristic Evaluation
% ------------------------------------------

evaluate_paths([], []).
evaluate_paths([[Node | PathRest] | Other], [(H, [Node | PathRest]) | Scored]) :-
    heuristic(Node, H),
    evaluate_paths(Other, Scored).

% Merge lists by heuristic value (ascending)
```

Output

```
Goal reached! Path: [a, c, f, g]
Path = [a, c, f, g].
```

Question 4

Write a program to implement A* search algorithm in Prolog.

```
% ------------------------------------------
% A* SEARCH ALGORITHM IN PROLOG
% ------------------------------------------

% astar(Start, Goal, Path, Cost)
astar(Start, Goal, Path, Cost) :-
    astar_search([(0, 0, [Start])], [], Goal, Path, Cost).

% Base case: first node in open list reaches Goal
astar_search([(F, G, [Goal|Rest]) | _], _, Goal, Path, F) :-
    reverse([Goal|Rest], Path),
    write('Goal reached! Path = '), write(Path), nl,
    write('Total Cost = '), write(F), nl.

% Main A* loop
astar_search([(F, G, [Node|Rest]) | Open], Closed, Goal, Path, Cost) :-
    findall((F2, G2, [Next,Node|Rest]),
            ( edge(Node, Next, StepCost),
              \+ member(Next, [Node|Rest]),
              G2 is G + StepCost,
              heuristic(Next, H),
              F2 is G2 + H
            ),
            Children),

    append(Open, Children, TempOpen),
    sort(TempOpen, NewOpen),
    astar_search(NewOpen, [Node|Closed], Goal, Path, Cost).
```

Output:

```
Goal reached! Path = [a,b,e,g]
Total Cost = 8
Path = [a, b, e, g],
Cost = 8.
```

Question 5

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

```
% ----- Terminal Node Utility Values -----
% value(Node, UtilityValue)

value(a, 3).
value(b, 5).
value(c, 6).
value(d, 9).
value(e, 1).
value(f, 2).
value(g, 0).
value(h, -1).

% ----- Tree Structure -----
% children(Node, [ListOfChildren])

children(root, [n1, n2, n3]).
children(n1, [a, b]).
children(n2, [c, d]).
children(n3, [e, f, g, h]).

% ----- MINIMAX ALGORITHM -----

% minimax(Node, Player, BestValue)
% Player = max | min

% Case 1: Terminal node → value is known
minimax(Node, _, Value) :-
    value(Node, Value), !.

% Case 2: Max Player
minimax(Node, max, BestValue) :-
    children(Node, ChildList),
    findall(Value,
            (member(Child, ChildList), minimax(Child, min, Value)),
            Values),
    max_list(Values, BestValue).

% Case 3: Min Player
minimax(Node, min, BestValue) :-
    children(Node, ChildList),
    findall(Value,
            (member(Child, ChildList), minimax(Child, max, Value)),
            Values),
    min_list(Values, BestValue).
```

Output:

```
% c:/Users/Hp/Desktop/hill7 compiled 0.00 sec, 0 clauses
|    minimax(root, max, Best).
|
|    ?-minimaxd
|
```

```
Best = 5.
```

Question 6

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

```
% ------------- WATER JUG PROBLEM ----------------
% State is represented as state(Jug4, Jug3)

% capacity of jugs
capacity(4, 3).

% goal state
goal(state(2, _)).

% starting state
start(state(0, 0)).

% ---------------- MOVES ----------------

% Fill the 4-liter jug
move(state(_, B), fill4, state(4, B)).

% Fill the 3-liter jug
move(state(A, _), fill3, state(A, 3)).

% Empty the 4-liter jug
move(state(_, B), empty4, state(0, B)).

% Empty the 3-liter jug
move(state(A, _), empty3, state(A, 0)).

% Pour from 4-liter jug to 3-liter jug
move(state(A, B), pour4to3, state(A2, B2)) :-
    capacity(_, C3),
    Transfer is min(A, C3 - B),
    A2 is A - Transfer,
    B2 is B + Transfer.
```

Output:

```
Solution steps:
fill3
pour3to4
fill3
pour3to4
empty4
pour3to4
fill3
pour3to4
```

Question 7

Implement sudoku problem (minimum 9×9 size) using constraint satisfaction in Prolog.

```
:- use_module(library(clpfd)).

% ---------------- SUDOKU SOLVER ------------------

sudoku(Rows) :-
    length(Rows, 9),
    maplist(same_length(Rows), Rows),

    % Flatten the grid
    append(Rows, Vars),
    Vars ins 1..9,

    % Rows must be distinct
    maplist(all_distinct, Rows),

    % Columns must be distinct
    transpose(Rows, Columns),
    maplist(all_distinct, Columns),

    % 3x3 subgrids must be distinct
    Rows = [R1,R2,R3,R4,R5,R6,R7,R8,R9],
    blocks(R1, R2, R3),
    blocks(R4, R5, R6),
    blocks(R7, R8, R9),

    % Label the solution
    maplist(label, Rows),
    maplist(portray_clause, Rows).

% ---------------- 3x3 BLOCKS ---------------------

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:

```
?- sudoku([
   [5,3,0, 0,7,0, 0,0,0],
   [6,0,0, 1,9,5, 0,0,0],
   [0,9,8, 0,0,0, 0,6,0],

   [8,0,0, 0,6,0, 0,0,3],
   [4,0,0, 8,0,3, 0,0,1],
   [7,0,0, 0,2,0, 0,0,6],

   [0,6,0, 0,0,0, 2,8,0],
   [0,0,0, 4,1,9, 0,0,5],
   [0,0,0, 0,8,0, 0,7,9]
]).
```

Question 8

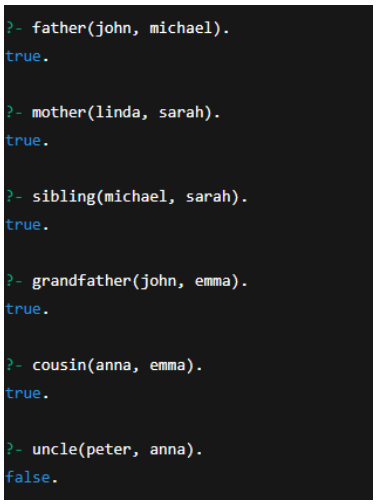
Write a Prolog program to implement the family tree and demonstrate the family relationship.

```
% ---------------------- FACTS ------------------------

% gender
male(john).
male(michael).
male(peter).
male(david).
male(kevin).

female(linda).
female(sarah).
female(emma).
female(julia).
female(anna).

% parents
parent(john, michael).
parent(linda, michael).

parent(john, sarah).
parent(linda, sarah).

parent(michael, emma).
parent(julia, emma).

parent(sarah, david).
parent(kevin, david).

parent(sarah, anna).
parent(kevin, anna).

% ------------------ RELATIONSHIP RULES -------------------


% X is father of Y
father(X, Y) :- male(X), parent(X, Y).

% X is mother of Y
mother(X, Y) :- female(X), parent(X, Y).

% X and Y are siblings
sibling(X, Y) :-
    parent(P, X),
    parent(P, Y),
    X \= Y.

% brother
brother(X, Y) :-
    male(X),
    sibling(X, Y).

% sister
sister(X, Y) :-
    female(X),
    sibling(X, Y).

% grandfather
grandfather(X, Y) :-
    male(X),
    parent(X, Z),
    parent(Z, Y).

% grandmother
grandmother(X, Y) :-
```

```
    female(X),
    parent(X, Z),
    parent(Z, Y).

% child
child(X, Y) :-
    parent(Y, X).

% son
son(X, Y) :-
    male(X),
    parent(Y, X).

% daughter
daughter(X, Y) :-
    female(X),
    parent(Y, X).

% uncle
uncle(X, Y) :-
    male(X),
    sibling(X, P),
    parent(P, Y).
```

Output:

(Example Query)



Question 9

Write a Prolog program to implement knowledge representation using frames with appropriate examples.

```
% -------------------- FRAME DEFINITIONS ----------------------

% frame(FrameName, [slot(SlotName, SlotValue), ...]).

frame(animal, [
    slot(type, living_thing),
    slot(moves, yes),
    slot(needs_food, yes)
]).

frame(bird, [
    is_a(animal),
    slot(has_wings, yes),
    slot(can_fly, yes)
]).

frame(penguin, [
    is_a(bird),
    slot(can_fly, no)
]).

frame(eagle, [
    is_a(bird),
    slot(can_fly, yes),
    slot(color, brown)
]).

frame(fish, [
    is_a(animal),
    slot(lives_in_water, yes),
    slot(has_gills, yes)
]).

frame(salmon, [
    is_a(fish),
    slot(color, silver)
]).

% -------------------- SLOT VALUE RETRIEVAL ----------------------

% get_slot(Frame, SlotName, SlotValue)
% Retrieves a slot value for a frame, supporting inheritance.

get_slot(Frame, SlotName, SlotValue) :-
    frame(Frame, Slots),
    member(slot(SlotName, SlotValue), Slots), !.

% Inheritance rule:
get_slot(Frame, SlotName, SlotValue) :-
    frame(Frame, Slots),
    member(is_a(Parent), Slots),
    get_slot(Parent, SlotName, SlotValue).
```

Output:

(Example queries)

```
?- get_slot(bird, has_wings, Value).
Value = yes.

?- get_slot(penguin, can_fly, Value).
Value = no.

?- get_slot(penguin, moves, Value).
Value = yes.          % inherited from animal

?- get_slot(salmon, has_gills, Value).
Value = yes.

?- get_slot(eagle, type, Value).
Value = living_thing.
```

Question 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.**

```
conc([], L, L).
conc([H|T], L2, [H|R]) :-
    conc(T, L2, R).
```

Output:

```
?- conc([1,2,3], [4,5], R).
```

```
R = [1, 2, 3, 4, 5].
```

Question 11

Write a Prolog program to implement reverse(L, R) where List L is original and List R is reversed list.

```
reverse([], []).
reverse([H|T], R) :-
    reverse(T, RT),
    conc(RT, [H], R).
```

Output:

```
?- reverse([1,2,3,4], R).
```

```
R = [4, 3, 2, 1].
```

Question 12

Write a Prolog program to generate a parse tree of a given sentence in English language assuming the grammar required for parsing.

```
sentence(s(NP, VP)) --> noun_phrase(NP), verb_phrase(VP).

noun_phrase(np(Det, N)) --> det(Det), noun(N).

verb_phrase(vp(V, NP)) --> verb(V), noun_phrase(NP).
verb_phrase(vp(V)) --> verb(V).

det(det(the)) --> [the].
det(det(a)) --> [a].

noun(n(man)) --> [man].
noun(n(woman)) --> [woman].
noun(n(apple)) --> [apple].

verb(v(eats)) --> [eats].
verb(v(sees)) --> [sees].
```

Output:

```
?- phrase(sentence(Tree), [the, man, eats, an, apple]).
```

```
Tree = s(
          np(det(the), n(man)),
          vp(v(eats), np(det(an), n(apple)))
       ).
```

Question 13

Write a Prolog program to recognize context free grammar an bn

```
% recognizes strings of the form a^n b^n

recognize([a,b]).   % base case

recognize([a|T]) :-
    append(Middle, [b], T),
    recognize(Middle).
```

Output

```
?- recognize([a,b]).
```

```
true.
```