**Pract 1:**

%Program1

%red(rose).

%likes(bill ,cindy).

%owns(gold,john).

%Program 2:

c\_to\_f(C, F) :-

F is C \* 9 / 5 + 32.

% Check if the temperature is freezing

freezing(F) :-

F =< 32.

-------------------------------------------------------------------------------------------------------------------------------------

**Pract 2:**

% Pract2:render solutions nicely.

:- use\_rendering(chess).

%% queens(+N, -Queens) is nondet.

%

% @param Queens is a list of column numbers for placing the queens.

% @author Richard A. O'Keefe (The Craft of Prolog)

queens(N, Queens) :-

length(Queens, N),

board(Queens, Board, 0, N, \_, \_),

queens(Board, 0, Queens).

board([], [], N, N, \_, \_).

board([\_|Queens], [Col-Vars|Board], Col0, N, [\_|VR], VC) :-

Col is Col0+1,

functor(Vars, f, N),

constraints(N, Vars, VR, VC),

board(Queens, Board, Col, N, VR, [\_|VC]).

constraints(0, \_, \_, \_) :- !.

constraints(N, Row, [R|Rs], [C|Cs]) :-

arg(N, Row, R-C),

M is N-1,

constraints(M, Row, Rs, Cs).

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

queens([C|Cs], Row0, [Col|Solution]) :-

Row is Row0+1,

select(Col-Vars, [C|Cs], Board),

arg(Row, Vars, Row-Row),

queens(Board, Row, Solution).

/\*\* <examples>

?- queens(8, Queens).

\*/

-------------------------------------------------------------------------------------------------------------------------------------

**Pract3:**

% 3:Updated graph representation with adjacency information

edge(a, b).

edge(a, c).

edge(b, d).

edge(b, e).

edge(c, f).

% Depth-first search predicate

dfs(Node, Visited) :-

dfs(Node, [], Visited).

dfs(Node, Path, Visited) :-

\+ member(Node, Path),

write(Node), nl,

append(Path, [Node], NewPath),

findall(Adjacent, edge(Node, Adjacent), AdjacentNodes),

member(NextNode, AdjacentNodes),

dfs(NextNode, NewPath, Visited).

dfs(\_, Visited, Visited).

% Example usage:

% To find all nodes reachable from node 'a'

% Call: dfs(a, Visited).

------------------------------------------------------------------------------------------------------------------------------------

**Pract4:**

% 4:Graph representation with heuristic values

edge(a, b, 2).

edge(a, c, 3).

edge(b, d, 1).

edge(b, e, 4).

edge(c, f, 2).

% Heuristic values for each node

heuristic(a, 5).

heuristic(b, 3).

heuristic(c, 2).

heuristic(d, 1).

heuristic(e, 4).

heuristic(f, 0).

% Best-First Search predicate

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

best\_first\_search([(0, Start)], Goal, []).

best\_first\_search([], \_, \_) :- write('Goal not reached.').

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

Node = Goal,

write('Goal reached!'), nl,

write('Total cost: '), write(Cost).

best\_first\_search([(Cost, Node) | Rest], Goal, Visited) :-

findall((NewCost, Adjacent), (edge(Node, Adjacent, EdgeCost), \+ member(Adjacent, Visited), heuristic(Adjacent, H), NewCost is Cost + EdgeCost + H), Children),

append(Rest, Children, NewQueue),

sort(NewQueue, SortedQueue),

best\_first\_search(SortedQueue, Goal, [Node | Visited]).

% Example usage:

% To find the path from node 'a' to node 'f'

% Call: best\_first\_search(a, f).

-------------------------------------------------------------------------------------------------------------------------------------

**Pract5:**

YOU Do

Pract 6:

% Initial state of the robot and obstacles

at(robot, position(0, 0)).

at(obstacle1, position(1, 0)).

at(obstacle2, position(2, 1)).

choose\_action(Action).

in\_robot(Obj).

% Goal state for the robot to reach

goal\_state(at(robot, position(2, 2))).

% Define the connected locations and possible directions

connected(position(X, Y), position(X, Y1)) :- Y1 is Y + 1.

connected(position(X, Y), position(X, Y1)) :- Y1 is Y - 1.

connected(position(X, Y), position(X1, Y)) :- X1 is X + 1.

connected(position(X, Y), position(X1, Y)) :- X1 is X - 1.

% Means-End Analysis to achieve the goal

solve :-

goal\_state(at(robot, Goal)),

write('Goal reached!'), nl,

write('Robot is at: '), write(Goal), nl.

solve :-

\+ goal\_state(at(robot, \_)),

choose\_action(Action),

execute\_action(Action),

solve.

% Choosing an action based on the current state

choose\_action(move\_forward) :-

at(robot, Current),

connected(Current, Next),

\+ at(obstacle(\_, \_), Next).

choose\_action(turn\_left).

choose\_action(turn\_right).

choose\_action(pick) :-

at(robot, Position),

at(obstacle(Obj), Position).

choose\_action(drop) :-

at(robot, Position),

in\_robot(Obj).

% Executing the chosen action

execute\_action(move\_forward) :-

at(robot, Current),

connected(Current, Next),

\+ at(obstacle(\_, \_), Next),

write('Robot moves forward to '), write(Next), nl,

retract(at(robot, \_)),

assert(at(robot, Next)).

execute\_action(turn\_left) :-

write('Robot turns left'), nl.

execute\_action(turn\_right) :-

write('Robot turns right'), nl.

execute\_action(pick) :-

at(robot, Position),

at(obstacle(Obj), Position),

write('Robot picks up '), write(Obj), nl,

retract(at(obstacle(Obj), Position)),

assert(in\_robot(Obj)).

execute\_action(drop) :-

at(robot, Position),

in\_robot(Obj),

write('Robot drops '), write(Obj), write(' at '), write(Position), nl,

retract(in\_robot(Obj)),

assert(at(obstacle(Obj), Position)).

% Example usage:

% Call: solve.

-------------------------------------------------------------------------------------------------------------------------------------

**Pract 7:**

% 7:Cities and distances between them

distance(city1, city2, 10).

distance(city2, city3, 25).

distance(city2, city3, 20).

% Predicate to find a tour

tsp(Tour, TotalDistance) :-

permutation([city1, city2, city3], Tour),

valid\_tour(Tour, TotalDistance),

write('Optimal Tour: '), write(Tour), nl,

write('Total Distance: '), write(TotalDistance), nl.

% Predicate to check if a tour is valid and calculate total distance

valid\_tour([\_], 0).

valid\_tour([City1, City2 | Rest], TotalDistance) :-

distance(City1, City2, Dist),

valid\_tour([City2 | Rest], RemainingDistance),

TotalDistance is Dist + RemainingDistance.

% Example usage

% Call: tsp(Tour, TotalDistance).

-------------------------------------------------------------------------------------------------------------------------------------

**Pract 8:**

% Pract8 : Medical imaging data representation

% For simplicity, using a list of features representing imaging data

% Each feature is a pixel value or a feature extracted from the image

% Example features for illustration

features([feature('Intensity', 0.8), feature('Texture', 0.4), feature('Shape', 0.2), feature('Edge', 0.9)]).

% Predicates to define diseases based on imaging features

diagnosis(features) :-

features(Features),

evaluate\_features(Features, Disease),

write('Diagnosis: '), write(Disease), nl.

% Rule to evaluate imaging features and make a diagnosis

evaluate\_features(Features, Disease) :-

has\_intensity\_variation(Features),

has\_texture\_irregularity(Features),

has\_abnormal\_shape(Features),

has\_edge\_irregularity(Features),

Disease = 'Healthy'.

% Rules to check for specific features related to diseases

has\_intensity\_variation(Features) :-

member(feature('Intensity', Intensity), Features),

Intensity > 0.7. % Example threshold, can be adjusted.

has\_texture\_irregularity(Features) :-

member(feature('Texture', Texture), Features),

Texture < 0.5. % Example threshold, can be adjusted.

has\_abnormal\_shape(Features) :-

member(feature('Shape', Shape), Features),

Shape < 0.3. % Example threshold, can be adjusted.

has\_edge\_irregularity(Features) :-

member(feature('Edge', Edge), Features),

Edge > 0.8. % Example threshold, can be adjusted.

% Example usage:

% Call: diagnosis(features).