Practical : 2

:- use\_rendering(chess).

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),

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).

**Query : queens(8,Queens).**

Practical : 3

% Define the capacities of the jugs

capacity(1, 10). % Capacity of jug 1

capacity(2, 7). % Capacity of jug 2

% Define the initial and final states

initial\_state(jugs, jugs(0, 0)).

final\_state(jugs(6, 0)).

% Define legal states

legal(jugs(V1, V2)) :-

capacity(1, C1),

capacity(2, C2),

V1 >= 0, V1 =< C1,

V2 >= 0, V2 =< C2.

% Define possible moves

move(jugs(V1, V2), fill(1)) :-

capacity(1, C1),

V1 < C1.

move(jugs(V1, V2), fill(2)) :-

capacity(2, C2),

V2 < C2.

move(jugs(V1, V2), empty(1)) :-

V1 > 0.

move(jugs(V1, V2), empty(2)) :-

V2 > 0.

move(jugs(V1, V2), transfer(1, 2)) :-

V1 > 0.

move(jugs(V1, V2), transfer(2, 1)) :-

V2 > 0.

% Apply move logic

apply\_move(jugs(V1, V2), fill(1), jugs(C1, V2)) :-

capacity(1, C1).

apply\_move(jugs(V1, V2), fill(2), jugs(V1, C2)) :-

capacity(2, C2).

apply\_move(jugs(V1, V2), empty(1), jugs(0, V2)).

apply\_move(jugs(V1, V2), empty(2), jugs(V1, 0)).

apply\_move(jugs(V1, V2), transfer(1, 2), jugs(V1\_New, V2\_New)) :-

capacity(2, C2),

Transfer is min(V1, C2 - V2),

V1\_New is V1 - Transfer,

V2\_New is V2 + Transfer.

apply\_move(jugs(V1, V2), transfer(2, 1), jugs(V1\_New, V2\_New)) :-

capacity(1, C1),

Transfer is min(V2, C1 - V1),

V1\_New is V1 + Transfer,

V2\_New is V2 - Transfer.

% Depth-First Search (DFS) implementation

dfs(State, Goal, Path, Moves) :-

State = Goal,

reverse(Path, Moves).

dfs(State, Goal, Path, Moves) :-

move(State, Move),

apply\_move(State, Move, NewState),

legal(NewState),

\+ member(NewState, Path), % Avoid cycles

dfs(NewState, Goal, [NewState | Path], Moves).

% Entry point for testing DFS

test\_dfs(jugs, Moves) :-

initial\_state(jugs, Start),

final\_state(Goal),

dfs(Start, Goal, [Start], Moves).

**Query : test\_dfs(jugs,Moves).**

Practical : 4

Code :

% Define winning conditions

win(Board, Player) :- rowwin(Board, Player).

win(Board, Player) :- colwin(Board, Player).

win(Board, Player) :- diagwin(Board, Player).

% Row win

rowwin([Player, Player, Player, \_, \_, \_, \_, \_, \_], Player).

rowwin([\_, \_, \_, Player, Player, Player, \_, \_, \_], Player).

rowwin([\_, \_, \_, \_, \_, \_, Player, Player, Player], Player).

% Column win

colwin([Player, \_, \_, Player, \_, \_, Player, \_, \_], Player).

colwin([\_, Player, \_, \_, Player, \_, \_, Player, \_], Player).

colwin([\_, \_, Player, \_, \_, Player, \_, \_, Player], Player).

% Diagonal win

diagwin([Player, \_, \_, \_, Player, \_, \_, \_, Player], Player).

diagwin([\_, \_, Player, \_, Player, \_, Player, \_, \_], Player).

% Predicate for determining the opponent

other(x, o).

other(o, x).

% Main game loop

game(Board, Player) :-

win(Board, Player), !,

format('Player ~w wins!', [Player]), nl.

game(Board, Player) :-

other(Player, Opponent),

move(Board, Player, NewBoard),

display(NewBoard),

game(NewBoard, Opponent).

% Possible moves

move([b | Rest], Player, [Player | Rest]).

move([Head | Rest], Player, [Head | NewRest]) :-

move(Rest, Player, NewRest).

% Display the board

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

format('~w | ~w | ~w~n', [A, B, C]),

format('~w | ~w | ~w~n', [D, E, F]),

format('~w | ~w | ~w~n~n', [G, H, I]).

% Start a game with an empty board

play :-

initial\_board(Board),

explain,

display(Board),

game(Board, x).

% Initial empty board

initial\_board([b, b, b, b, b, b, b, b, b]).

% Explanation for the user

explain :-

write('Welcome to Tic Tac Toe!'), nl,

write('You play as X. Enter your move as a number (1-9).'), nl,

write('Positions on the board are as follows:'), nl,

display([1, 2, 3, 4, 5, 6, 7, 8, 9]).

% Allow the player to make a move

player\_move(Board, NewBoard) :-

read(N),

nth1(N, Board, b), % Check if position is empty

replace(Board, N, x, NewBoard).

player\_move(Board, Board) :-

write('Invalid move. Try again.'), nl,

player\_move(Board, NewBoard).

% Replace the Nth element in a list

replace([\_|T], 1, X, [X|T]).

replace([H|T], N, X, [H|R]) :-

N > 1,

N1 is N - 1,

replace(T, N1, X, R).

% Computer's move (Best First Search logic)

respond(Board, NewBoard) :-

move(Board, o, NewBoard),

win(NewBoard, o), !.

respond(Board, NewBoard) :-

move(Board, o, NewBoard),

\+ x\_can\_win\_in\_one(NewBoard), !.

respond(Board, NewBoard) :-

move(Board, o, NewBoard).

% Check if X can win in the next move

x\_can\_win\_in\_one(Board) :-

move(Board, x, NewBoard),

win(NewBoard, x).

**Query : play.**

Practical 5

Code :

:- use\_rendering(sudoku).

:- use\_module(library(clpfd)).

sudoku(Rows) :-

length(Rows, 9), % Ensure the board has 9 rows

maplist(same\_length(Rows), Rows), % Ensure the board is 9x9

append(Rows, Vs), % Flatten the 9x9 grid into a single list

Vs ins 1..9, % Each variable must be between 1 and 9

maplist(all\_distinct, Rows), % All rows must have distinct values

transpose(Rows, Columns), % Transpose rows into columns

maplist(all\_distinct, Columns), % All columns must have distinct values

Rows = [A, B, C, D, E, F, G, H, I], % Split rows into 3x3 blocks

blocks(A, B, C), blocks(D, E, F), blocks(G, H, I). % Check 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]), % Ensure 3x3 block values are distinct

blocks(Bs1, Bs2, Bs3). % Recurse for remaining blocks

problem(1,

[[\_, \_, \_, \_, \_, \_, \_, \_, \_],

[\_, \_, \_, \_, \_, 3, \_, 8, 5],

[\_, \_, 1, \_, 2, \_, \_, \_, \_],

[\_, \_, \_, 5, \_, 7, \_, \_, \_],

[\_, \_, 4, \_, \_, \_, 1, \_, \_],

[\_, 9, \_, \_, \_, \_, \_, \_, \_],

[5, \_, \_, \_, \_, \_, \_, 7, 3],

[\_, \_, 2, \_, 1, \_, \_, \_, \_],

[\_, \_, \_, \_, 4, \_, \_, \_, 9]]).

**Query: problem(1, Rows), sudoku(Rows).**

Practical 6 ::

Code:-

% Define the initial position and goal

initial\_state(state(0, 0)). % Starting position (0, 0)

goal\_state(state(3, 2)). % Goal position (3, 2)

% Define possible actions

action(move\_right, state(X, Y), state(X1, Y)) :-

X1 is X + 1, X1 < 5. % Limit grid size

action(move\_left, state(X, Y), state(X1, Y)) :-

X1 is X - 1, X1 >= 0. % Limit grid size

action(move\_down, state(X, Y), state(X, Y1)) :-

Y1 is Y + 1, Y1 < 5. % Limit grid size

action(move\_up, state(X, Y), state(X, Y1)) :-

Y1 is Y - 1, Y1 >= 0. % Limit grid size

% Check if the current state is the goal state

is\_goal(State) :-

goal\_state(State).

% Means-End Analysis to find a path from the initial state to the goal state

means\_end\_analysis(CurrentState, \_, []) :-

is\_goal(CurrentState).

means\_end\_analysis(CurrentState, Visited, [Action | Path]) :-

\+ is\_goal(CurrentState),

\+ member(CurrentState, Visited), % Avoid revisiting states

action(Action, CurrentState, NextState),

means\_end\_analysis(NextState, [CurrentState | Visited], Path).

% Helper to start the analysis

solve(Path) :-

initial\_state(StartState),

means\_end\_analysis(StartState, [], Path).

% Visualize the path on a grid

visualize\_path(Path) :-

initial\_state(StartState),

visualize\_grid(StartState, Path).

% Helper to print the grid

visualize\_grid(CurrentState, Path) :-

grid\_size(GridSize),

length(Row, GridSize),

maplist(=(.), Row),

length(Grid, GridSize),

maplist(=(Row), Grid),

mark\_position(CurrentState, 'R', Grid, TempGrid), % Mark the robot's position

mark\_path(Path, CurrentState, TempGrid, FinalGrid), % Mark the path

goal\_state(GoalState),

mark\_position(GoalState, 'G', FinalGrid, GridWithGoal), % Mark the goal position

print\_grid(GridWithGoal).

% Mark a specific position in the grid

mark\_position(state(X, Y), Symbol, Grid, NewGrid) :-

nth0(Y, Grid, Row),

replace(Row, X, Symbol, NewRow),

replace(Grid, Y, NewRow, NewGrid).

% Mark the path on the grid

mark\_path([], \_, Grid, Grid).

mark\_path([Action | Actions], CurrentState, Grid, NewGrid) :-

update\_position(Action, CurrentState, NextState),

mark\_position(NextState, '\*', Grid, TempGrid),

mark\_path(Actions, NextState, TempGrid, NewGrid).

% Update the robot's position based on the action

update\_position(move\_right, state(X, Y), state(X1, Y)) :-

X1 is X + 1.

update\_position(move\_left, state(X, Y), state(X1, Y)) :-

X1 is X - 1.

update\_position(move\_down, state(X, Y), state(X, Y1)) :-

Y1 is Y + 1.

update\_position(move\_up, state(X, Y), state(X, Y1)) :-

Y1 is Y - 1.

% Replace an element in a list

replace([\_|T], 0, X, [X|T]). % Replace head

replace([H|T], N, X, [H|R]) :-

N > 0,

N1 is N - 1,

replace(T, N1, X, R).

% Print the grid

print\_grid([]).

print\_grid([Row | Rest]) :-

write(Row), nl,

print\_grid(Rest).

% Define the size of the grid

grid\_size(5).

**% Query to solve and visualize the pat**

**% query :-**

**%solve(Path),**

**%visualize\_path(Path).**

Practical 7::

Code :

1 ))

% Define the initial position and goal

initial\_state(state(0, 0)). % Starting position (0, 0)

goal\_state(state(3, 2)). % Goal position (3, 2)

% Define possible actions

action(move\_right, state(X, Y), state(X1, Y)) :- X1 is X + 1, X1 < 5.

action(move\_left, state(X, Y), state(X1, Y)) :- X1 is X - 1, X1 >= 0.

action(move\_down, state(X, Y), state(X, Y1)) :- Y1 is Y + 1, Y1 < 5.

action(move\_up, state(X, Y), state(X, Y1)) :- Y1 is Y - 1, Y1 >= 0.

% Check if the current state is the goal state

is\_goal(State) :- goal\_state(State).

% Means-End Analysis to find a path

means\_end\_analysis(CurrentState, Visited, Path) :-

is\_goal(CurrentState),

Path = [].

means\_end\_analysis(CurrentState, Visited, [Action | Path]) :-

\+ is\_goal(CurrentState),

\+ member(CurrentState, Visited),

action(Action, CurrentState, NextState),

means\_end\_analysis(NextState, [CurrentState | Visited], Path).

% Solve function to start analysis

solve(Path) :-

initial\_state(StartState),

means\_end\_analysis(StartState, [], Path).

% Visualize the path on a grid

visualize\_path(Path) :-

initial\_state(StartState),

visualize\_grid(StartState, Path).

visualize\_grid(CurrentState, Path) :-

grid\_size(GridSize),

length(Row, GridSize),

maplist(=(.), Row),

length(Grid, GridSize),

maplist(=(Row), Grid),

mark\_position(CurrentState, 'R', Grid, TempGrid),

mark\_path(Path, CurrentState, TempGrid, FinalGrid),

goal\_state(GoalState),

mark\_position(GoalState, 'G', FinalGrid, GridWithGoal),

print\_grid(GridWithGoal).

% Mark a specific position in the grid

mark\_position(state(X, Y), Symbol, Grid, NewGrid) :-

nth0(Y, Grid, Row),

replace(Row, X, Symbol, NewRow),

replace(Grid, Y, NewRow, NewGrid).

% Mark the path

mark\_path([], \_, Grid, Grid).

mark\_path([Action | Actions], CurrentState, Grid, NewGrid) :-

update\_position(Action, CurrentState, NextState),

mark\_position(NextState, '\*', Grid, TempGrid),

mark\_path(Actions, NextState, TempGrid, NewGrid).

% Update the position based on action

update\_position(move\_right, state(X, Y), state(X1, Y)) :- X1 is X + 1.

update\_position(move\_left, state(X, Y), state(X1, Y)) :- X1 is X - 1.

update\_position(move\_down, state(X, Y), state(X, Y1)) :- Y1 is Y + 1.

update\_position(move\_up, state(X, Y), state(X, Y1)) :- Y1 is Y - 1.

% Replace element in a list

replace([\_|T], 0, X, [X|T]).

replace([H|T], N, X, [H|R]) :- N > 0, N1 is N - 1, replace(T, N1, X, R).

% Print the grid

print\_grid([]).

print\_grid([Row | Rest]) :-

write(Row), nl,

print\_grid(Rest).

% Define the grid size

grid\_size(5).

**% Example query:**

**% ?- solve(Path), visualize\_path(Path).**

2)) code :

% Facts: Distances between cities

dist(a, b, 10).

dist(a, c, 15).

dist(a, d, 20).

dist(b, a, 10). % Ensure bidirectional

dist(b, c, 35).

dist(b, d, 25).

dist(c, a, 15).

dist(c, b, 35).

dist(c, d, 30).

dist(d, a, 20).

dist(d, b, 25).

dist(d, c, 30).

% Find the shortest path using tsp

tsp(Path, Cost) :-

findall(City, (dist(City, \_, \_); dist(\_, City, \_)), AllCities),

sort(AllCities, Cities), % Ensure unique cities

permutation(Cities, [Start | Rest]),

append([Start | Rest], [Start], Path), % Return to the starting city

calculate\_cost(Path, Cost).

% Calculate the total cost of a path

calculate\_cost([City1, City2 | Rest], Cost) :-

dist(City1, City2, D),

calculate\_cost([City2 | Rest], PartialCost),

Cost is PartialCost + D.

calculate\_cost([\_], 0).

**% Query to run:**

**% ?- tsp(Path, Cost), write('Shortest Path: '), write(Path), nl, write('Total Cost: '), write(Cost).**

**Questions and Answer On Practical :**

### **Practical 1: Study of PROLOG**

1. **What is Prolog, and why is it used in AI?**Prolog (Programming in Logic) is a declarative language focusing on logic and relationships. It's used in AI for tasks like expert systems, natural language processing, and problem-solving due to its powerful pattern-matching and automatic backtracking capabilities.

**Explain the syntax of Prolog with an example.**Prolog syntax consists of facts, rules, and queries.  
**Example**:  
prolog  
Copy code  
male('john').

female('mary').

person(X) :- male(X); female(X).

?- person(X). % Query to find all persons

1. **How does Prolog handle backtracking?**Prolog automatically explores alternative solutions by backtracking. When a query fails, Prolog returns to the previous choice point and tries another path.

### **Practical 2: 8 Queens Problem**

1. **What is the 8 Queens problem?**It involves placing 8 queens on an 8x8 chessboard such that no two queens threaten each other (no same row, column, or diagonal).
2. **How does backtracking work in Prolog?**Prolog tries to place a queen in a valid position row by row. If a conflict arises, it backtracks to the previous row and tries another position.
3. **Why is the 8 Queens problem important in AI?**It demonstrates constraint satisfaction and backtracking, which are crucial for solving complex combinatorial problems.

### **Practical 3: Solve Problem Using Depth First Search (DFS)**

1. **Explain Depth First Search with its advantages and disadvantages.**DFS explores a branch fully before backtracking.
   * **Advantages**: Low memory usage, finds deep solutions quickly.
   * **Disadvantages**: Can get stuck in deep loops, not guaranteed to find the shortest path.
2. **How does DFS solve the Water Jug problem?**DFS systematically explores all possible states by filling, emptying, or transferring water between jugs until it reaches the goal state.
3. **What are the limitations of DFS in large search spaces?**DFS may run indefinitely in infinite or very deep search spaces and is not guaranteed to find the shortest path.

### **Practical 4: Solve Tic Tac Toe Using Best First Search**

1. **What is Best First Search, and how is it different from DFS and BFS?**Best First Search uses a heuristic to prioritize nodes that seem closest to the goal, unlike DFS or BFS, which explore blindly.
2. **How do heuristics work in Tic Tac Toe?**Heuristics evaluate the game board to choose the best move, such as blocking an opponent or completing a winning line.
3. **Can Best First Search guarantee a win? Why or why not?**No, it depends on the opponent's moves and the heuristic's accuracy. It can improve decision-making but doesn't guarantee a win against an optimal player.

### **Practical 5: Solve 8-Puzzle Problem Using Best First Search**

1. **What is the 8-puzzle problem?**It involves sliding tiles on a 3x3 grid to match a goal configuration.
2. **Explain the Manhattan distance heuristic.**The Manhattan distance sums the absolute differences between the current and goal positions of each tile, representing how far each tile is from its goal.
3. **Why is Best First Search suitable for the 8-puzzle problem?**It focuses on the most promising moves, guided by heuristics like Manhattan distance, to efficiently reach the solution.

### **Practical 6: Solve Robot Traversal Using Means-End Analysis**

1. **What is Means-End Analysis?**MEA is a problem-solving strategy that reduces the difference between the current state and the goal by selecting actions that bring them closer.
2. **How does MEA ensure goal achievement?**It repeatedly applies actions to reduce the difference until the goal state is reached, backtracking if necessary.
3. **Compare MEA with other search strategies like DFS or BFS.**MEA focuses on reducing differences directly, making it more efficient for certain problems than DFS (which explores deeply) or BFS (which explores broadly).

### **Practical 7: Solve Traveling Salesman Problem (TSP)**

1. **What is the Traveling Salesman Problem?**TSP involves finding the shortest path that visits a set of cities and returns to the starting city.
2. **How is TSP solved using Prolog?**Prolog generates all permutations of city routes, calculates their total distances, and selects the shortest one.
3. **What are real-world applications of TSP?**TSP is used in logistics, route planning, circuit design, and supply chain optimization.

### **General Questions**

1. **What is the difference between informed and uninformed search strategies?**
   * **Informed Search**: Uses heuristics to guide the search (e.g., Best First Search, A\*).
   * **Uninformed Search**: Explores without guidance (e.g., DFS, BFS).
2. **How does Prolog handle recursion and logical inference?**Prolog uses recursion to solve problems by breaking them into smaller subproblems and logical inference to deduce new facts from existing ones.
3. **Explain the concept of heuristics and give examples where they are used.**Heuristics are rules of thumb to make decisions faster.  
   **Examples**:
   * Manhattan distance in the 8-puzzle problem.
   * Blocking moves in Tic Tac Toe.
4. **How does backtracking help in solving combinatorial problems?**Backtracking explores all possibilities by returning to previous steps if a solution path fails, ensuring all configurations are tested.
5. **Compare different search algorithms (DFS, BFS, Best First Search).**
   * **DFS**: Explores deeply, may not find the shortest path.
   * **BFS**: Explores level by level, guarantees the shortest path.
   * **Best First Search**: Uses heuristics, focuses on promising paths.
6. **Discuss the role of Prolog in AI applications.**Prolog excels in tasks requiring logical reasoning, such as expert systems, natural language processing, and combinatorial problem solving.
7. **Explain the importance of solving combinatorial optimization problems like TSP and 8-puzzle.**These problems help in optimizing resources and solving real-world challenges in logistics, scheduling, and planning efficiently.