**Water-jug (using state space formation):**from collections import deque

import math

def is\_valid(state, capacity\_a, capacity\_b):

a, b = state

return 0 <= a <= capacity\_a and 0 <= b <= capacity\_b

def get\_next\_states(current\_state, capacity\_a, capacity\_b):

a, b = current\_state

states = []

# Fill Jug A

states.append((capacity\_a, b))

# Fill Jug B

states.append((a, capacity\_b))

# Empty Jug A

states.append((0, b))

# Empty Jug B

states.append((a, 0))

# Pour A to B

pour = min(a, capacity\_b - b)

states.append((a - pour, b + pour))

# Pour B to A

pour = min(b, capacity\_a - a)

states.append((a + pour, b - pour))

return states

def is\_solvable(capacity\_a, capacity\_b, target):

# If target is greater than both jugs, it's not possible

if target > max(capacity\_a, capacity\_b):

return False

# Target must be divisible by GCD of jug sizes

return target % math.gcd(capacity\_a, capacity\_b) == 0

def water\_jug\_state\_space(capacity\_a, capacity\_b, target):

start\_state = (0, 0)

visited = set()

parent = {}

queue = deque([start\_state])

visited.add(start\_state)

while queue:

current = queue.popleft()

a, b = current

if a == target or b == target:

path = []

while current != (0, 0):

path.append(current)

current = parent[current]

path.append((0, 0))

path.reverse()

return path

for next\_state in get\_next\_states(current, capacity\_a, capacity\_b):

if next\_state not in visited and is\_valid(next\_state, capacity\_a, capacity\_b):

visited.add(next\_state)

parent[next\_state] = current

queue.append(next\_state)

return None # No solution found

# --- Main Program ---

if \_\_name\_\_ == "\_\_main\_\_":

print("=== Water Jug Problem ===\n")

jug\_a = int(input("Enter capacity of Jug A: "))

jug\_b = int(input("Enter capacity of Jug B: "))

target = int(input("Enter target amount of water: "))

if not is\_solvable(jug\_a, jug\_b, target):

print("\nNo solution is possible with the given jug capacities and target.")

else:

solution = water\_jug\_state\_space(jug\_a, jug\_b, target)

if solution:

print("\nSteps to reach the target:")

for step in solution:

print(f"Jug A: {step[0]} | Jug B: {step[1]}")

else:

print("\nNo solution found through state space search.")  
 **Uninformed search(BFS & DFS):**% Tree edges (like parent → child)

edge(a, b).

edge(a, c).

edge(b, d).

edge(b, e).

edge(c, f).

% dfs(Start, Goal, Path).

dfs(Start, Goal, Path) :-

dfs\_search(Start, Goal, [Start], Path).

% If current node is the goal, return the path

dfs\_search(Goal, Goal, Visited, Visited).

% Otherwise, keep exploring

dfs\_search(Current, Goal, Visited, Path) :-

edge(Current, Next),

\+ member(Next, Visited), % Avoid cycles

dfs\_search(Next, Goal, [Next|Visited], Path).

% bfs(Start, Goal, Path).

bfs(Start, Goal, Path) :-

bfs\_search([[Start]], Goal, RevPath),

reverse(RevPath, Path). % Reverse the path at the end

% If we reach the goal, return the path

bfs\_search([[Goal|Rest]|\_], Goal, [Goal|Rest]).

% Otherwise, keep exploring neighbors

bfs\_search([[Current|Rest]|OtherPaths], Goal, Path) :-

findall([Next, Current | Rest],

(edge(Current, Next), \+ member(Next, [Current|Rest])),

NewPaths),

append(OtherPaths, NewPaths, UpdatedPaths),

bfs\_search(UpdatedPaths, Goal, Path).

%Sample Queries:

%?- dfs(a, e, Path).

%Path = [e, b, a].

%?- bfs(a, e, Path).

%Path = [a, b, e].

**Informed search (A\*):**from queue import PriorityQueue

def a\_star\_algorithm(graph, start, goal, heuristic): """

A\* Algorithm implementation.

Parameters:

graph: Dictionary representing the graph {node: [(neighbor, cost), ...]}.

start: Starting node.

goal: Goal node.

heuristic: Dictionary representing heuristic values for each node.

Returns:

Path from start to goal and the total cost. """

# Priority queue to store (cost, node, path)

open\_set = PriorityQueue()

open\_set.put((0, start, [start])) # (f(n), current\_node, path)

# Track the cost to reach each node

g\_cost = {node: float('inf') for node in graph}

g\_cost[start] = 0

while not open\_set.empty():

# Get the node with the smallest f(n)

f, current, path = open\_set.get()

# If goal is reached, return the path and cost

if current == goal:

return path, g\_cost[goal]

# Explore neighbors

for neighbor, cost in graph[current]:

tentative\_g\_cost = g\_cost[current] + cost

if tentative\_g\_cost < g\_cost[neighbor]:

g\_cost[neighbor] = tentative\_g\_cost

f\_cost = tentative\_g\_cost + heuristic[neighbor]

open\_set.put((f\_cost, neighbor, path + [neighbor]))

# If no path found return None, float('inf')

# Example usage

graph = {

'A': [('B', 1), ('C', 3)],

'B': [('D', 1), ('E', 4)],

'C': [('F', 2)],

'D': [('G', 5)],

'E': [('G', 1)],

'F': [('G', 2)],

'G': []

}

heuristic = {

'A': 7,

'B': 6,

'C': 4,

'D': 2,

'E': 2,

'F': 2,

'G': 0

}

start = 'A'

goal = 'G'

path, cost = a\_star\_algorithm(graph, start, goal, heuristic)

print(f"Path: {path}, Total Cost: {cost}") **Informed search (Best-first):**def best\_first\_search(graph, start, goal, heuristic):

open\_list = [(heuristic[start], start, [start])] # (h(n), current\_node, path)

closed\_list = []

while open\_list:

# Sort open\_list based on heuristic value (lowest first)

open\_list.sort()

h, current, path = open\_list.pop(0)

closed\_list.append(current)

# Print current state

print("Current Node:", current)

print("Open List:", [node for \_, node, \_ in open\_list])

print("Closed List:", closed\_list)

print("-" \* 30)

# Goal check

if current == goal:

print("􈙌􈙇􈙈􈙉􈙊􈙋 Goal Reached!")

print("Path:", " -> ".join(path))

return

# Add neighbors to open\_list if not visited

for neighbor in graph[current]:

if neighbor not in closed\_list and all(neighbor != n for \_, n, \_ in open\_list):

open\_list.append((heuristic[neighbor], neighbor, path + [neighbor]))

print("􎆡 Goal not found!")

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F']

'D': [],

'E': ['G'],

'F': [],

'G': []

}

heuristic = {

'A': 6,

'B': 4,

'C': 5,

'D': 3,

'E': 2,

'F': 4,

'G': 0

}

# Run the search

best\_first\_search(graph, 'A', 'G', heuristic)  
  
**Family tree (Prolog):**

% Facts for parent(Parent, Child).

parent(sevak, anil).

parent(vimala, anil).

parent(anil, sunaina).

parent(leela, sunaina).

parent(anil, sonam).

parent(leela, sonam).

parent(anil, sumit).

parent(leela, sumit).

% Rules to define relationships

grandparent(Grandparent, Grandchild) :-

parent(Grandparent, Parent),

parent(Parent, Grandchild).

sibling(Person1, Person2) :-

parent(Parent, Person1),

parent(Parent, Person2),

person1 \= Person2.

ancestor(Ancestor, Descendant) :-

parent(Ancestor, Descendant).

ancestor(Ancestor, Descendant) :-

parent(Ancestor, Parent),

ancestor(Parent, Descendant).  
  
%sample queries:  
%?-ancestor(X, sonam).

%X=anil

%X=leela

%X=sevak

%X=vimala

%?-grandparent(X, sunaina).

%X=seval

%X=Vimala  
  
**Bayes belief Network:**

% Probabilities

prob(rain, 0.3). % P(Rain = true)

prob(sprinkler\_given\_rain(true), 0.9). % P(Sprinkler = true | Rain = true)

prob(sprinkler\_given\_rain(false), 0.2). % P(Sprinkler = true | Rain = false)

% P(WetGrass = true | Rain, Sprinkler)

prob(wet\_grass(true, true), 0.99). % Rain = true, Sprinkler = true

prob(wet\_grass(true, false), 0.9). % Rain = true, Sprinkler = false

prob(wet\_grass(false, true), 0.9). % Rain = false, Sprinkler = true

prob(wet\_grass(false, false), 0.0). % Rain = false, Sprinkler = false

% Query function: get probability of wet grass given rain and sprinkler

wet\_grass\_probability(Rain, Sprinkler, P) :-

prob(wet\_grass(Rain, Sprinkler), P).

% Example usage:

% ?- wet\_grass\_probability(true, true, P).

% P = 0.99  
  
**Animal Experts system(prolog):**

% Animal Recognition Expert System

% Knowledge Base with rules for animals

animal(lion) :- mammal, carnivore, has\_mane.

animal(tiger) :- mammal, carnivore, has\_stripes.

animal(cheetah) :- mammal, carnivore, has\_spots.

animal(elephant) :- mammal, herbivore, has\_trunk.

animal(zebra) :- mammal, herbivore, has\_stripes.

animal(giraffe) :- mammal, herbivore, has\_long\_neck.

animal(crocodile) :- reptile, carnivore, has\_sharp\_teeth.

animal(tortoise) :- reptile, herbivore, has\_shell.

animal(parrot) :- bird, herbivore, has\_colored\_feathers.

animal(eagle) :- bird, carnivore, has\_sharp\_beak.

% Classification Rules

mammal :- verify(has\_fur), verify(gives\_birth).

reptile :- verify(has\_scales), verify(lays\_eggs).

bird :- verify(has\_feathers), verify(lays\_eggs).

carnivore :- verify(eats\_meat).

herbivore :- verify(eats\_plants).

has\_mane :- verify(has\_mane).

has\_stripes :- verify(has\_stripes).

has\_spots :- verify(has\_spots).

has\_trunk :- verify(has\_trunk).

has\_long\_neck :- verify(has\_long\_neck).

has\_sharp\_teeth :- verify(has\_sharp\_teeth).

has\_shell :- verify(has\_shell).

has\_colored\_feathers :- verify(has\_colored\_feathers).

has\_sharp\_beak :- verify(has\_sharp\_beak).

% Dynamic fact storage for user responses

:- dynamic(yes/1).

:- dynamic(no/1).

% Asking user for verification

verify(Attribute) :-

(yes(Attribute) -> true;

no(Attribute) -> fail;

ask(Attribute)).

% Asking the user and storing the response

ask(Attribute) :-

format('Does the animal have the following attribute: ~w? (yes/no) ', [Attribute]),

read(Response),

nl,

( (Response == yes ; Response == y) -> assertz(yes(Attribute));

assertz(no(Attribute)), fail).

% Identify the animal based on attributes

identify :-

retractall(yes(\_)), % Reset previous answers

retractall(no(\_)),

animal(Animal),

format('The animal is: ~w', [Animal]), nl, !.

% If no match is found

identify :-

write('No matching animal found!'), nl.  
  
%to run this code  
%?-identify.