**Question 1 uninformed**

from collections import deque

def bfs(adjList, startNode, visited):

"""

Performs a breadth-first search on the graph starting from the given node.

Args:

adjList (list): The adjacency list representation of the graph.

startNode (int): The node to start the search from.

visited (list): A list to keep track of visited nodes.

"""

q = deque() # Initialize the queue

visited[startNode] = True # Mark the start node as visited

q.append(startNode) # Add the start node to the queue

while q: # While there are nodes to process

currentNode = q.popleft() # Dequeue a node

print(currentNode, end=" ") # Print the current node

# Iterate through the neighbors of the current node

for neighbor in adjList[currentNode]:

if not visited[neighbor]: # If the neighbor hasn't been visited

visited[neighbor] = True # Mark it as visited

q.append(neighbor) # Add it to the queue

def addEdge(adjList, u, v):

"""

Adds an edge to the graph.

Args:

adjList (list): The adjacency list representation of the graph.

u (int): The source node of the edge.

v (int): The destination node of the edge.

"""

adjList[u].append(v) # Add the edge from u to v

def main():

"""

The main function.

"""

vertices = 5 # Number of vertices in the graph

adjList = [[] for \_ in range(vertices)] # Create an empty adjacency list

# Add edges to the graph

addEdge(adjList, 0, 1)

addEdge(adjList, 0, 2)

addEdge(adjList, 1, 3)

addEdge(adjList, 1, 4)

addEdge(adjList, 2, 4)

visited = [False] \* vertices # Initialize the visited list

print("Breadth First Traversal starting from vertex 0:", end=" ")

bfs(adjList, 0, visited) # Perform BFS starting from vertex 0

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

main() # Execute the main function

**Question 2 informed**

import heapq

def heuristic(a, b):

"""Calculate the Manhattan distance from point a to point b."""

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def greedy\_best\_first\_search(maze, start, end):

"""Perform Greedy Best-First Search in a grid maze."""

open\_list = [] # Priority queue for the nodes to explore

came\_from = {} # Tracks the path history

visited = set() # Keeps track of visited nodes

# Initialize the open list with the start node

heapq.heappush(open\_list, (heuristic(start, end), start))

came\_from[start] = None # Start node has no parent

# Possible movement directions: Right, Down, Left, Up

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

while open\_list:

# Get the node with the lowest heuristic value

current\_heuristic, current = heapq.heappop(open\_list)

# Check if we've reached the end

if current == end:

return reconstruct\_path(came\_from, current)

visited.add(current)

# Explore neighbors

for direction in directions:

neighbor = (current[0] + direction[0], current[1] + direction[1])

# Ensure the neighbor is within bounds and walkable

if (0 <= neighbor[0] < len(maze)) and (0 <= neighbor[1] < len(maze[0])):

if maze[neighbor[0]][neighbor[1]] == 0 and neighbor not in visited:

visited.add(neighbor)

came\_from[neighbor] = current

heapq.heappush(open\_list, (heuristic(neighbor, end), neighbor))

return None # If no path is found

def reconstruct\_path(came\_from, current):

"""Reconstruct the path from start to end."""

path = []

while current is not None:

path.append(current)

current = came\_from[current]

return path[::-1] # Return reversed path

# Example maze: 0 - walkable, 1 - blocked

maze = [

[0, 0, 0, 0, 1],

[0, 1, 1, 0, 1],

[0, 0, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 0, 0]

]

start = (0, 0)

end = (4, 4)

path = greedy\_best\_first\_search(maze, start, end)

print("Path from start to end:", path)

**Question 3 local**

import random

def hill\_climbing(problem):

# Start with an initial state

current\_state = problem.initial\_state()

while True:

# Generate neighbors of the current state

neighbors = problem.get\_neighbors(current\_state)

# If no neighbors, return the current state

if not neighbors:

break

# Choose the best neighbor

next\_state = max(neighbors, key=lambda state: problem.evaluate(state))

# If the best neighbor is not better than the current state, stop

if problem.evaluate(next\_state) <= problem.evaluate(current\_state):

break

# Move to the neighbor state

current\_state = next\_state

return current\_state

# Example problem class

class Problem:

def initial\_state(self):

# Define the initial state (example: random integer between 0 and 100)

return random.randint(0, 100)

def get\_neighbors(self, state):

# Define neighbors of the current state (example: state +/- 1)

return [state - 1, state + 1] if 0 < state < 100 else []

def evaluate(self, state):

# Define the objective function (example: maximize state, target = 90)

return -abs(state - 90)

# Example usage

problem = Problem()

solution = hill\_climbing(problem)

print("Solution found:", solution)

**Question 4 agent archi**

Initialize VirtualAssistant:

Load profiles, NLP engine, task system

Function HandleUserRequest(user\_id, request):

profile = GetUserProfile(user\_id)

goal = IdentifyGoal(ParseRequest(request))

If IsKnownTask(goal):

response = ExecuteTask(goal, profile)

Else:

response = LearnAndHandle(goal)

UpdateUserProfile(user\_id, request, response)

return response

Function ParseRequest(request):

return NLPParser(request)

Function IsKnownTask(goal):

return CheckTaskDatabase(goal)

Function ExecuteTask(goal, profile):

Return TaskExecutor(goal, profile) # Handles specific tasks like setting reminders

Function LearnAndHandle(goal):

return LearnFromInteraction(goal)

Function UpdateUserProfile(user\_id, request, response):

ModifyUserProfile(user\_id, request, response)

# Main loop

For each user\_id in ActiveUsers:

response = HandleUserRequest(user\_id, GetUserRequest(user\_id))

SendResponseToUser(user\_id, response)

**Question 5 prolog monkey**

class MonkeyBananaProblem:

def \_\_init\_\_(self):

# Initial state: (Monkey's position, Monkey has banana?, Banana's position, Monkey on box?)

self.initial\_state = ('floor', False, 'ceiling', False)

self.actions = ['move', 'push\_box', 'climb\_box', 'grab\_banana']

def goal\_test(self, state):

# The goal is when the monkey has the banana

return state[1] == True

def get\_successors(self, state):

successors = []

monkey\_position, has\_banana, banana\_position, on\_box = state

if has\_banana:

return [] # No more actions possible if the monkey has the banana

if monkey\_position == 'floor' and not on\_box:

successors.append(('floor', False, 'ceiling', False, 'move'))

successors.append(('floor', False, 'ceiling', False, 'push\_box'))

if monkey\_position == 'floor' and on\_box == False:

successors.append(('floor', False, 'ceiling', False, 'climb\_box'))

return successors

def result(self, state, action):

monkey\_position, has\_banana, banana\_position, on\_box = state

if action == 'move':

# Monkey moves to where the box is

return ('near\_box', has\_banana, banana\_position, on\_box)

elif action == 'push\_box':

# Monkey pushes the box under the banana

return ('under\_banana', has\_banana, 'ceiling', on\_box)

elif action == 'climb\_box':

# Monkey climbs the box

return (monkey\_position, has\_banana, banana\_position, True)

elif action == 'grab\_banana':

# Monkey grabs the banana when on top of the box

if monkey\_position == 'under\_banana' and on\_box:

return (monkey\_position, True, banana\_position, on\_box)

return state

def search(self):

# Simple breadth-first search (BFS) to solve the problem

from collections import deque

# Initial state of the problem

queue = deque([(self.initial\_state, [])])

while queue:

current\_state, path = queue.popleft()

if self.goal\_test(current\_state):

return path # Return the sequence of actions to reach the goal

# Get all possible actions from the current state

for action in self.actions:

new\_state = self.result(current\_state, action)

if new\_state != current\_state: # If state changes, add to queue

queue.append((new\_state, path + [action]))

return None

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

problem = MonkeyBananaProblem()

solution = problem.search()

if solution:

print(f"Solution found! Actions: {solution}")

else:

print("No solution found.")

**Question 6 prolog 8 puzzle**

import heapq

class PuzzleNode:

def \_\_init\_\_(self, state, parent, move, depth, cost):

self.state = state # Current state of the puzzle

self.parent = parent # Parent node (to trace the solution path)

self.move = move # Move that led to this state (e.g., 'UP', 'DOWN', etc.)

self.depth = depth # Depth of the node in the search tree

self.cost = cost # Cost of reaching this node (g(n) + h(n))

def \_\_lt\_\_(self, other):

return self.cost < other.cost # Priority based on cost (used in the priority queue)

# Manhattan Distance Heuristic (h(n))

def manhattan\_distance(state, goal):

distance = 0

for i in range(1, 9): # Tiles numbered 1 to 8

x1, y1 = divmod(state.index(i), 3)

x2, y2 = divmod(goal.index(i), 3)

distance += abs(x1 - x2) + abs(y1 - y2)

return distance

# Get neighbors by sliding tiles

def get\_neighbors(state):

neighbors = []

blank\_index = state.index(0) # 0 represents the blank space

x, y = divmod(blank\_index, 3) # Convert 1D index to 2D coordinates (row, col)

# Possible moves: UP, DOWN, LEFT, RIGHT

moves = {

'UP': (-1, 0), 'DOWN': (1, 0),

'LEFT': (0, -1), 'RIGHT': (0, 1)

}

for move, (dx, dy) in moves.items():

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < 3 and 0 <= new\_y < 3: # Check if the move is within bounds

new\_index = new\_x \* 3 + new\_y

new\_state = list(state)

new\_state[blank\_index], new\_state[new\_index] = new\_state[new\_index], new\_state[blank\_index] # Swap tiles

neighbors.append((new\_state, move))

return neighbors

# A\* Search Algorithm

def a\_star(start, goal):

open\_list = []

closed\_list = set()

start\_node = PuzzleNode(state=start, parent=None, move=None, depth=0, cost=manhattan\_distance(start, goal))

heapq.heappush(open\_list, start\_node)

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.state == goal:

# Reconstruct the solution path

moves = []

node = current\_node

while node.parent is not None:

moves.append(node.move)

node = node.parent

return moves[::-1] # Reverse the list to get the moves from start to goal

closed\_list.add(tuple(current\_node.state))

# Expand neighbors

for neighbor\_state, move in get\_neighbors(current\_node.state):

if tuple(neighbor\_state) in closed\_list:

continue

# Create a new PuzzleNode for the neighbor

neighbor\_node = PuzzleNode(state=neighbor\_state,

parent=current\_node,

move=move,

depth=current\_node.depth + 1,

cost=current\_node.depth + 1 + manhattan\_distance(neighbor\_state, goal))

heapq.heappush(open\_list, neighbor\_node)

return None # No solution found

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

# Initial and goal states (0 represents the blank space)

start\_state = [1, 2, 3,

4, 0, 5,

6, 7, 8] # Example initial state

goal\_state = [1, 2, 3,

4, 5, 6,

7, 8, 0] # Goal state

# Solve the 8-puzzle using A\* search

solution = a\_star(start\_state, goal\_state)

if solution:

print("Solution found! Moves:", solution)

else:

print("No solution found.")

**Question 7 bayesian network**

from pgmpy.models import BayesianNetwork

from pgmpy.factors.discrete import TabularCPD

from pgmpy.inference import VariableElimination

# Step 1: Define the structure of the Bayesian Network

model = BayesianNetwork([('Rain', 'Traffic'), ('Accident', 'Traffic')])

# Step 2: Define the Conditional Probability Distributions (CPDs)

# CPD for Rain (prior probability of raining)

cpd\_rain = TabularCPD(variable='Rain', variable\_card=2, values=[[0.7], [0.3]])

# CPD for Accident (prior probability of an accident happening)

cpd\_accident = TabularCPD(variable='Accident', variable\_card=2, values=[[0.9], [0.1]])

# CPD for Traffic (conditional on both Rain and Accident)

# Values are filled in order: P(Traffic | Rain, Accident)

cpd\_traffic = TabularCPD(variable='Traffic', variable\_card=2,

values=[[0.9, 0.6, 0.7, 0.1], # P(Traffic = False | ...)

[0.1, 0.4, 0.3, 0.9]], # P(Traffic = True | ...)

evidence=['Rain', 'Accident'],

evidence\_card=[2, 2])

# Step 3: Add CPDs to the model

model.add\_cpds(cpd\_rain, cpd\_accident, cpd\_traffic)

# Step 4: Verify if the model is valid

if model.check\_model():

print("Bayesian Network is valid!")

# Step 5: Perform inference

inference = VariableElimination(model)

# Query the probability of traffic given that it's raining

posterior\_probability = inference.query(variables=['Traffic'], evidence={'Rain': 1})

print(posterior\_probability)

# Query the probability of traffic given that there was an accident and it’s raining

posterior\_probability = inference.query(variables=['Traffic'], evidence={'Rain': 1, 'Accident': 1})

print(posterior\_probability)