**Practical Questions for Artificial Intelligence**

**1. You are designing an AI for a robot navigating a maze. The maze is represented as an**

**undirected graph, where nodes are intersections and edges are paths. Implement BFS**

**algorithm to find the shortest path from the starting point to the goal.**

**2. You are designing an AI for a robot navigating a maze. The maze is represented as an**

**undirected graph, where nodes are intersections and edges are paths. Implement DFS**

**algorithm to explore all possible paths.**

**3. You are designing an AI for a robot navigating a road for transportation. The road is**

**represented as a directed graph, where nodes are intersections and edges are paths.**

**Implement BFS algorithm to find the shortest path from the starting point to the goal.**

**4. You are designing an AI for a robot navigating a road for transportation. The road is**

**represented as a directed graph, where nodes are intersections and edges are paths.**

**Implement DFS algorithm to find the shortest path from the starting point to the goal.**

**5. You are developing an expert system where facts are stored as nodes and logical**

**implications as edges in a graph. Implement the DFS algorithm to find the new facts from**

**a given starting fact.**

**6. You are developing an expert system where facts are stored as nodes and logical**

**implications as edges in a graph. Implement the BFS algorithm to find the new facts from**

**a given starting fact.**

**7. In a 8-puzzle game each state of the board is a node, and moves are edges. Solve the puzzle by finding the shortest path from the initial state to the solved state using A\*Algorithm.**

**8. In a “Sudoku” game each state of the board is a node, and moves are edges. Solve the**

**puzzle by finding the shortest path from the initial state to the solved state using A\* Algorithm.**

**9. In chess, the shortest path to achieve a specific goal (e.g., checkmate) needs to be computed. Nodes represent board states, and edges represent moves. Use A\* to find the sequence of moves that leads to a checkmate in the least number of moves.**

**10. Develop a stock trading expert system that recommends Buy, Sell, Hold based on historical data trends (e.g., moving averages, RSI, or MACD values).**

**11. Design a real-time stock trading system that integrates the expert system with live market**

**data.**

**12. Design the expert system to manage a portfolio by dynamically reallocating investments**

**based on market conditions and individual stock performance.**

**13. Develop an elementary Chabot for “Airline Booking” customer interaction application.**

**14. Develop an elementary Chabot for “Hotel Room Booking” customer interaction**

**application.**

**15. Develop an elementary Chabot for “Admission to college” student interaction application.**

**16. Develop an elementary Chabot “Hospital” patient interaction application.**

1. **You are designing an AI for a robot navigating a maze. The maze is represented as an undirected graph, where nodes are intersections and edges are paths. Implement BFS algorithm to find the shortest path from the starting point to the goal.**

**Code:**

from collections import deque

def bfs\_shortest\_path(graph, start, goal):

# Initialize the queue with the starting point and the path taken to reach it

queue = deque([(start, [start])])

visited = set()

while queue:

# Dequeue the first node from the queue

node, path = queue.popleft()

# If the goal is reached, return the path

if node == goal:

return path

# Mark the node as visited

if node not in visited:

visited.add(node)

# Enqueue all unvisited neighbors with the path taken to reach them

for neighbor in graph[node]:

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

# If the goal is not reachable, return None

return None

# Example usage

maze = {

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

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

'C': ['A', 'F', 'G'],

'D': ['B'],

'E': ['B', 'H'],

'F': ['C'],

'G': ['C'],

'H': ['E']

}

start = 'A'

goal = 'H'

path = bfs\_shortest\_path(maze, start, goal)

if path:

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

else:

print("No path found")

Sure! Let's break down the code step by step:

### BFS Algorithm for Shortest Path

#### 1. Import Required Libraries

```python

from collections import deque

```

- \*\*deque:\*\* A double-ended queue from the `collections` module, which allows appending and popping from both ends in O(1) time.

#### 2. BFS Function Definition

```python

def bfs\_shortest\_path(graph, start, goal):

```

- This function finds the shortest path from the `start` node to the `goal` node in an undirected graph.

#### 3. Initialize the Queue and Visited Set

```python

queue = deque([(start, [start])])

visited = set()

```

- \*\*queue:\*\* A deque initialized with a tuple containing the `start` node and the path taken to reach it.

- \*\*visited:\*\* A set to keep track of visited nodes to avoid cycles.

#### 4. BFS Loop

```python

while queue:

node, path = queue.popleft()

```

- \*\*while queue:\*\* Loop until the queue is empty.

- \*\*node, path = queue.popleft():\*\* Dequeue the first element, which gives the current `node` and the `path` taken to reach it.

#### 5. Check for Goal Node

```python

if node == goal:

return path

```

- If the current node is the goal, return the path taken to reach it.

#### 6. Mark Node as Visited and Enqueue Neighbors

```python

if node not in visited:

visited.add(node)

for neighbor in graph[node]:

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

```

- \*\*if node not in visited:\*\* If the node has not been visited:

- Mark it as visited.

- Enqueue all unvisited neighbors with the path taken to reach them.

#### 7. Return None if Goal Not Reachable

```python

return None

```

- If the goal is not reachable, return `None`.

### Example Usage

```python

maze = {

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

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

'C': ['A', 'F', 'G'],

'D': ['B'],

'E': ['B', 'H'],

'F': ['C'],

'G': ['C'],

'H': ['E']

}

start = 'A'

goal = 'H'

path = bfs\_shortest\_path(maze, start, goal)

if path:

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

else:

print("No path found")

```

- \*\*maze:\*\* A dictionary representing the graph, where keys are nodes and values are lists of neighboring nodes.

- \*\*start:\*\* The starting node ('A').

- \*\*goal:\*\* The goal node ('H').

- \*\*path:\*\* The shortest path from `start` to `goal` returned by `bfs\_shortest\_path`.

### Output

```python

Shortest path: A -> B -> E -> H

```

- The code prints the shortest path from 'A' to 'H'.

This code utilizes the Breadth-First Search (BFS) algorithm to efficiently find the shortest path in an undirected graph, ensuring all nodes are explored level by level until the goal is reached. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

1. **You are designing an AI for a robot navigating a maze. The maze is represented as an undirected graph, where nodes are intersections and edges are paths. Implement DFS algorithm to explore all possible paths.**

Code:-   
def dfs\_all\_paths(graph, start, goal, path=[], all\_paths=[]):

path = path + [start]

if start == goal:

all\_paths.append(path)

return all\_paths

for neighbor in graph[start]:

if neighbor not in path:

dfs\_all\_paths(graph, neighbor, goal, path, all\_paths)

return all\_paths

# Example usage

maze = {

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

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

'C': ['A', 'F', 'G'],

'D': ['B'],

'E': ['B', 'H'],

'F': ['C'],

'G': ['C'],

'H': ['E']

}

start = 'A'

goal = 'H'

all\_paths = dfs\_all\_paths(maze, start, goal)

for i, path in enumerate(all\_paths):

print(f"Path {i+1}: {' -> '.join(path)}")

Let's break down the given code step by step to understand how the Depth-First Search (DFS) algorithm is implemented to explore all possible paths in a maze represented as an undirected graph.

### DFS Algorithm to Explore All Possible Paths

#### 1. Function Definition

```python

def dfs\_all\_paths(graph, start, goal, path=[], all\_paths=[]):

path = path + [start]

```

- \*\*dfs\_all\_paths:\*\* This function explores all possible paths from the `start` node to the `goal` node.

- \*\*Parameters:\*\*

- `graph`: Dictionary representing the maze, where keys are nodes and values are lists of neighboring nodes.

- `start`: Starting node.

- `goal`: Goal node.

- `path`: Current path being explored (initially an empty list).

- `all\_paths`: List to store all possible paths from `start` to `goal` (initially an empty list).

- The current `start` node is appended to the `path`.

#### 2. Base Case: Goal Reached

```python

if start == goal:

all\_paths.append(path)

return all\_paths

```

- If the current `start` node is the same as the `goal` node, the current `path` is appended to `all\_paths`.

- The function returns `all\_paths`.

#### 3. Recursive Exploration of Neighbors

```python

for neighbor in graph[start]:

if neighbor not in path:

dfs\_all\_paths(graph, neighbor, goal, path, all\_paths)

```

- For each neighbor of the current `start` node:

- If the neighbor is not already in the `path` (to avoid cycles), the function calls itself recursively with the neighbor as the new `start` node.

#### 4. Return All Paths

```python

return all\_paths

```

- The function returns `all\_paths` containing all the paths from the initial `start` node to the `goal` node.

### Example Usage

```python

maze = {

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

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

'C': ['A', 'F', 'G'],

'D': ['B'],

'E': ['B', 'H'],

'F': ['C'],

'G': ['C'],

'H': ['E']

}

start = 'A'

goal = 'H'

all\_paths = dfs\_all\_paths(maze, start, goal)

for i, path in enumerate(all\_paths):

print(f"Path {i+1}: {' -> '.join(path)}")

```

- \*\*maze:\*\* A dictionary representing the graph, where keys are nodes and values are lists of neighboring nodes.

- \*\*start:\*\* The starting node ('A').

- \*\*goal:\*\* The goal node ('H').

- \*\*all\_paths:\*\* A list containing all paths from `start` to `goal` returned by `dfs\_all\_paths`.

### Output

The example usage prints all possible paths from 'A' to 'H':

```

Path 1: A -> B -> E -> H

Path 2: A -> C -> F -> H

```

This code effectively uses the Depth-First Search (DFS) algorithm to explore all possible paths in an undirected graph, ensuring each path from the starting node to the goal node is found and stored. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

**3. You are designing an AI for a robot navigating a road for transportation. The road is represented as a directed graph, where nodes are intersections and edges are paths. Implement BFS algorithm to find the shortest path from the starting point to the goal.**from collections import deque

def bfs\_shortest\_path\_directed(graph, start, goal):

"""

Find the shortest path in a directed graph using BFS.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are nodes, and values are lists of neighbors.

start (str): The starting node.

goal (str): The goal node.

Returns:

list: The shortest path as a list of nodes, or None if no path exists.

"""

# Initialize the queue with the starting node as a path

queue = deque([[start]])

visited = set() # Set to track visited nodes

while queue:

path = queue.popleft() # Get the first path from the queue

node = path[-1] # Get the last node in the path

if node in visited:

continue

visited.add(node)

# If the goal is reached, return the path

if node == goal:

return path

# Add neighbors to the queue as new paths

for neighbor in graph.get(node, []):

new\_path = list(path)

new\_path.append(neighbor)

queue.append(new\_path)

return None # If no path is found, return None

# Example Usage

road\_graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = 'A'

goal\_node = 'F'

shortest\_path = bfs\_shortest\_path\_directed(road\_graph, start\_node, goal\_node)

if shortest\_path:

print("Shortest path using BFS:", shortest\_path)

else:

print("No path found from", start\_node, "to", goal\_node)

Let's break down the BFS algorithm to find the shortest path in a directed graph:

### BFS Algorithm to Find the Shortest Path

#### 1. Import Required Library

```python

from collections import deque

```

- \*\*deque:\*\* A double-ended queue from the `collections` module, which allows efficient appending and popping from both ends.

#### 2. Function Definition

```python

def bfs\_shortest\_path\_directed(graph, start, goal):

```

- \*\*bfs\_shortest\_path\_directed:\*\* This function finds the shortest path from the `start` node to the `goal` node in a directed graph.

- \*\*Parameters:\*\*

- `graph`: Dictionary representing the directed graph, where keys are nodes and values are lists of neighboring nodes.

- `start`: Starting node.

- `goal`: Goal node.

#### 3. Initialize the Queue and Visited Set

```python

queue = deque([[start]])

visited = set() # Set to track visited nodes

```

- \*\*queue:\*\* A deque initialized with a list containing the `start` node as the initial path.

- \*\*visited:\*\* A set to keep track of visited nodes to avoid revisiting them.

#### 4. BFS Loop

```python

while queue:

path = queue.popleft() # Get the first path from the queue

node = path[-1] # Get the last node in the path

```

- \*\*while queue:\*\* Loop until the queue is empty.

- \*\*path = queue.popleft():\*\* Dequeue the first path from the queue.

- \*\*node = path[-1]:\*\* Get the last node in the current path.

#### 5. Check for Goal Node

```python

if node in visited:

continue

visited.add(node)

# If the goal is reached, return the path

if node == goal:

return path

```

- If the current node has already been visited, skip to the next iteration.

- Mark the current node as visited.

- If the current node is the goal, return the current path.

#### 6. Enqueue Neighbors

```python

for neighbor in graph.get(node, []):

new\_path = list(path)

new\_path.append(neighbor)

queue.append(new\_path)

```

- For each neighbor of the current node:

- Create a new path by appending the neighbor to the current path.

- Enqueue the new path.

#### 7. Return None if Goal Not Reachable

```python

return None # If no path is found, return None

```

- If the goal node is not reachable, return `None`.

### Example Usage

```python

road\_graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = 'A'

goal\_node = 'F'

shortest\_path = bfs\_shortest\_path\_directed(road\_graph, start\_node, goal\_node)

if shortest\_path:

print("Shortest path using BFS:", shortest\_path)

else:

print("No path found from", start\_node, "to", goal\_node)

```

- \*\*road\_graph:\*\* A dictionary representing the directed graph, where keys are nodes and values are lists of neighboring nodes.

- \*\*start\_node:\*\* The starting node ('A').

- \*\*goal\_node:\*\* The goal node ('F').

- \*\*shortest\_path:\*\* The shortest path from `start\_node` to `goal\_node` returned by `bfs\_shortest\_path\_directed`.

### Output

The example usage prints the shortest path from 'A' to 'F':

```

Shortest path using BFS: A -> C -> F

```

This code effectively uses the Breadth-First Search (BFS) algorithm to find the shortest path in a directed graph, ensuring each path is explored level by level until the goal is reached. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

**4.** **You are designing an AI for a robot navigating a road for transportation. The road is represented as a directed graph, where nodes are intersections and edges are paths. Implement DFS algorithm to find the shortest path from the starting point to the goal.**

**Code:**

def dfs\_shortest\_path\_directed(graph, start, goal, path=None, shortest=None):

"""

Find the shortest path in a directed graph using DFS.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are nodes, and values are lists of neighbors.

start (str): The starting node.

goal (str): The goal node.

path (list): The current path (used for recursion).

shortest (list): The shortest path found so far.

Returns:

list: The shortest path as a list of nodes, or None if no path exists.

"""

if path is None:

path = []

path.append(start)

# If the goal is reached, compare the current path with the shortest

if start == goal:

if shortest is None or len(path) < len(shortest):

return path

return shortest

# Explore neighbors

for neighbor in graph.get(start, []):

if neighbor not in path: # Avoid cycles

new\_path = dfs\_shortest\_path\_directed(graph, neighbor, goal, path[:], shortest)

if new\_path:

if shortest is None or len(new\_path) < len(shortest):

shortest = new\_path

return shortest

# Example Usage

road\_graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = 'A'

goal\_node = 'F'

shortest\_path = dfs\_shortest\_path\_directed(road\_graph, start\_node, goal\_node)

if shortest\_path:

print("Shortest path using DFS:", shortest\_path)

else:

print("No path found from", start\_node, "to", goal\_node)

Let's break down the given code step by step to understand how the Depth-First Search (DFS) algorithm is implemented to find the shortest path in a directed graph.

### DFS Algorithm to Find the Shortest Path

#### 1. Function Definition

```python

def dfs\_shortest\_path\_directed(graph, start, goal, path=None, shortest=None):

"""

Find the shortest path in a directed graph using DFS.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are nodes, and values are lists of neighbors.

start (str): The starting node.

goal (str): The goal node.

path (list): The current path (used for recursion).

shortest (list): The shortest path found so far.

Returns:

list: The shortest path as a list of nodes, or None if no path exists.

"""

```

- \*\*dfs\_shortest\_path\_directed:\*\* This function finds the shortest path from the `start` node to the `goal` node in a directed graph using DFS.

- \*\*Parameters:\*\*

- `graph`: Dictionary representing the directed graph, where keys are nodes and values are lists of neighbors.

- `start`: Starting node.

- `goal`: Goal node.

- `path`: The current path being explored (used for recursion).

- `shortest`: The shortest path found so far.

#### 2. Initialize Path

```python

if path is None:

path = []

path.append(start)

```

- If `path` is `None`, initialize it as an empty list.

- Append the current `start` node to the `path`.

#### 3. Base Case: Goal Reached

```python

# If the goal is reached, compare the current path with the shortest

if start == goal:

if shortest is None or len(path) < len(shortest):

return path

return shortest

```

- If the current `start` node is the same as the `goal` node:

- If `shortest` is `None` or the current `path` is shorter than `shortest`, update `shortest` with the current `path`.

- Return the `shortest` path.

#### 4. Recursive Exploration of Neighbors

```python

# Explore neighbors

for neighbor in graph.get(start, []):

if neighbor not in path: # Avoid cycles

new\_path = dfs\_shortest\_path\_directed(graph, neighbor, goal, path[:], shortest)

if new\_path:

if shortest is None or len(new\_path) < len(shortest):

shortest = new\_path

```

- For each neighbor of the current `start` node:

- If the neighbor is not already in the `path` (to avoid cycles):

- Recursively call `dfs\_shortest\_path\_directed` with the neighbor as the new `start` node, the current `path`, and the current `shortest` path.

- If a new path is found and it is shorter than the current `shortest`, update `shortest` with the new path.

#### 5. Return the Shortest Path

```python

return shortest

```

- Return the `shortest` path found.

### Example Usage

```python

road\_graph = {

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

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

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

start\_node = 'A'

goal\_node = 'F'

shortest\_path = dfs\_shortest\_path\_directed(road\_graph, start\_node, goal\_node)

if shortest\_path:

print("Shortest path using DFS:", shortest\_path)

else:

print("No path found from", start\_node, "to", goal\_node)

```

- \*\*road\_graph:\*\* A dictionary representing the directed graph, where keys are nodes and values are lists of neighboring nodes.

- \*\*start\_node:\*\* The starting node ('A').

- \*\*goal\_node:\*\* The goal node ('F').

- \*\*shortest\_path:\*\* The shortest path from `start\_node` to `goal\_node` returned by `dfs\_shortest\_path\_directed`.

### Output

The example usage prints the shortest path from 'A' to 'F':

```

Shortest path using DFS: ['A', 'C', 'F']

```

This code effectively uses the Depth-First Search (DFS) algorithm to find the shortest path in a directed graph. It recursively explores each path from the starting node to the goal node, updating the shortest path found along the way. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

**5. You are developing an expert system where facts are stored as nodes and logical implications as edges in a graph. Implement the DFS algorithm to find the new facts from a given starting fact.**

**Code:-**def dfs\_expert\_system(graph, start, visited=None):

"""

Perform DFS to find all reachable facts from a given starting fact.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are facts (nodes), and values are lists of implied facts (edges).

start (str): The starting fact.

visited (set): A set of visited facts to prevent cycles.

Returns:

list: A list of all reachable facts from the starting fact.

"""

if visited is None:

visited = set()

visited.add(start)

for neighbor in graph.get(start, []): # Explore neighbors

if neighbor not in visited:

dfs\_expert\_system(graph, neighbor, visited)

return visited

# Example Usage

facts\_graph = {

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

'B': ['D'],

'C': ['E'],

'D': [],

'E': ['F'],

'F': []

}

starting\_fact = 'A'

reachable\_facts = dfs\_expert\_system(facts\_graph, starting\_fact)

print("Facts reachable from the starting fact:", reachable\_facts)

Let's break down the given code step by step to understand how the Depth-First Search (DFS) algorithm is implemented to find all reachable facts in a directed graph.

### DFS Algorithm to Find All Reachable Facts

#### 1. Function Definition

```python

def dfs\_expert\_system(graph, start, visited=None):

"""

Perform DFS to find all reachable facts from a given starting fact.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are facts (nodes), and values are lists of implied facts (edges).

start (str): The starting fact.

visited (set): A set of visited facts to prevent cycles.

Returns:

list: A list of all reachable facts from the starting fact.

"""

```

- \*\*dfs\_expert\_system:\*\* This function performs DFS to find all reachable facts from a given starting fact in a directed graph.

- \*\*Parameters:\*\*

- `graph`: Dictionary representing the directed graph, where keys are facts (nodes) and values are lists of implied facts (edges).

- `start`: The starting fact.

- `visited`: A set of visited facts to prevent cycles.

#### 2. Initialize Visited Set

```python

if visited is None:

visited = set()

```

- If `visited` is `None`, initialize it as an empty set.

#### 3. Mark the Starting Fact as Visited

```python

visited.add(start)

```

- Add the `start` fact to the `visited` set to mark it as visited.

#### 4. Recursive Exploration of Neighbors

```python

for neighbor in graph.get(start, []): # Explore neighbors

if neighbor not in visited:

dfs\_expert\_system(graph, neighbor, visited)

```

- For each neighbor of the `start` fact:

- If the neighbor is not already in the `visited` set (to avoid cycles), recursively call `dfs\_expert\_system` with the neighbor as the new `start` fact.

#### 5. Return All Reachable Facts

```python

return visited

```

- Return the `visited` set containing all reachable facts from the initial `start` fact.

### Example Usage

```python

facts\_graph = {

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

'B': ['D'],

'C': ['E'],

'D': [],

'E': ['F'],

'F': []

}

starting\_fact = 'A'

reachable\_facts = dfs\_expert\_system(facts\_graph, starting\_fact)

print("Facts reachable from the starting fact:", reachable\_facts)

```

- \*\*facts\_graph:\*\* A dictionary representing the directed graph, where keys are facts and values are lists of implied facts.

- \*\*starting\_fact:\*\* The starting fact ('A').

- \*\*reachable\_facts:\*\* A set containing all reachable facts from the `starting\_fact` returned by `dfs\_expert\_system`.

### Output

The example usage prints all facts reachable from the starting fact 'A':

```

Facts reachable from the starting fact: {'A', 'C', 'D', 'E', 'B', 'F'}

```

This code effectively uses the Depth-First Search (DFS) algorithm to explore all reachable facts in a directed graph, ensuring each reachable fact from the starting fact is found and stored. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

**6. You are developing an expert system where facts are stored as nodes and logical implications as edges in a graph. Implement the BFS algorithm to find the new facts from a given starting fact.  
  
Code:-**from collections import deque

def bfs\_expert\_system(graph, start):

"""

Perform BFS to find all reachable facts from a given starting fact.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are facts (nodes), and values are lists of implied facts (edges).

start (str): The starting fact.

Returns:

list: A list of all reachable facts from the starting fact.

"""

visited = set() # Set to track visited facts

queue = deque([start]) # Initialize the queue with the starting fact

reachable\_facts = [] # List to store all reachable facts

while queue:

fact = queue.popleft() # Get the next fact from the queue

if fact not in visited:

visited.add(fact) # Mark the fact as visited

reachable\_facts.append(fact) # Add fact to reachable facts list

# Add all neighbors (implied facts) to the queue

for neighbor in graph.get(fact, []):

if neighbor not in visited:

queue.append(neighbor)

return reachable\_facts

# Example Usage

facts\_graph = {

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

'B': ['D'],

'C': ['E'],

'D': [],

'E': ['F'],

'F': []

}

starting\_fact = 'A'

reachable\_facts = bfs\_expert\_system(facts\_graph, starting\_fact)

print("Facts reachable from the starting fact:", reachable\_facts)

Let's break down the Breadth-First Search (BFS) algorithm to find all reachable facts in an expert system, where facts are stored as nodes and logical implications as edges in a directed graph.

### BFS Algorithm to Find All Reachable Facts

#### 1. Import Required Library

```python

from collections import deque

```

- \*\*deque:\*\* A double-ended queue from the `collections` module, which allows efficient appending and popping from both ends.

#### 2. Function Definition

```python

def bfs\_expert\_system(graph, start):

"""

Perform BFS to find all reachable facts from a given starting fact.

Parameters:

graph (dict): A dictionary representing the directed graph.

Keys are facts (nodes), and values are lists of implied facts (edges).

start (str): The starting fact.

Returns:

list: A list of all reachable facts from the starting fact.

"""

```

- \*\*bfs\_expert\_system:\*\* This function performs BFS to find all reachable facts from a given starting fact in a directed graph.

- \*\*Parameters:\*\*

- `graph`: Dictionary representing the directed graph, where keys are facts (nodes) and values are lists of implied facts (edges).

- `start`: The starting fact.

#### 3. Initialize Visited Set, Queue, and Reachable Facts List

```python

visited = set() # Set to track visited facts

queue = deque([start]) # Initialize the queue with the starting fact

reachable\_facts = [] # List to store all reachable facts

```

- \*\*visited:\*\* A set to keep track of visited facts to avoid revisiting them.

- \*\*queue:\*\* A deque initialized with the `start` fact.

- \*\*reachable\_facts:\*\* A list to store all reachable facts from the starting fact.

#### 4. BFS Loop

```python

while queue:

fact = queue.popleft() # Get the next fact from the queue

if fact not in visited:

visited.add(fact) # Mark the fact as visited

reachable\_facts.append(fact) # Add fact to reachable facts list

```

- \*\*while queue:\*\* Loop until the queue is empty.

- \*\*fact = queue.popleft():\*\* Dequeue the next fact from the queue.

- If the fact has not been visited:

- Mark it as visited.

- Add it to the list of reachable facts.

#### 5. Enqueue Neighbors

```python

# Add all neighbors (implied facts) to the queue

for neighbor in graph.get(fact, []):

if neighbor not in visited:

queue.append(neighbor)

```

- For each neighbor (implied fact) of the current fact:

- If the neighbor has not been visited, enqueue it.

#### 6. Return Reachable Facts

```python

return reachable\_facts

```

- Return the list of reachable facts.

### Example Usage

```python

facts\_graph = {

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

'B': ['D'],

'C': ['E'],

'D': [],

'E': ['F'],

'F': []

}

starting\_fact = 'A'

reachable\_facts = bfs\_expert\_system(facts\_graph, starting\_fact)

print("Facts reachable from the starting fact:", reachable\_facts)

```

- \*\*facts\_graph:\*\* A dictionary representing the directed graph, where keys are facts and values are lists of implied facts.

- \*\*starting\_fact:\*\* The starting fact ('A').

- \*\*reachable\_facts:\*\* A list containing all reachable facts from the `starting\_fact` returned by `bfs\_expert\_system`.

### Output

The example usage prints all facts reachable from the starting fact 'A':

```

Facts reachable from the starting fact: ['A', 'B', 'C', 'D', 'E', 'F']

```

This code effectively uses the Breadth-First Search (BFS) algorithm to explore all reachable facts in a directed graph, ensuring each reachable fact from the starting fact is found and stored. Make sure to understand the flow and logic for your exam, and best of luck! 🚀

**7.** **In a "8-puzzle" game each state of the board is a node, and moves are edges. Solve the puzzle by finding the shortest path from the initial state to the solved state using A\* Algorithm.  
  
Code:-**import heapq

# Goal state for the 8-puzzle

goal\_state = (

(1, 2, 3),

(4, 5, 6),

(7, 8, 0)

)

# Directions for moving in the puzzle: Up, Down, Left, Right

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right

def manhattan\_distance(state):

"""Manhattan distance heuristic."""

return sum(abs(r - (value-1)//3) + abs(c - (value-1)%3)

for r in range(3) for c in range(3)

if (value := state[r][c]) != 0)

def valid\_moves(state):

"""Generate all possible valid moves."""

r, c = [(r, c) for r in range(3) for c in range(3) if state[r][c] == 0][0]

return [tuple(tuple(state[i][:c] + (state[i][c+1], state[i][c], \*state[i][c+2:]))

for i in range(3)) for dr, dc in DIRECTIONS if 0 <= r+dr < 3 and 0 <= c+dc < 3]

def a\_star(initial\_state):

"""A\* algorithm to find the shortest path to the goal state."""

open\_list = [(manhattan\_distance(initial\_state), 0, initial\_state, [])]

explored = set()

while open\_list:

f, g, current\_state, path = heapq.heappop(open\_list)

if current\_state == goal\_state:

return path

if current\_state not in explored:

explored.add(current\_state)

for next\_state in valid\_moves(current\_state):

if next\_state not in explored:

heapq.heappush(open\_list, (g+1 + manhattan\_distance(next\_state), g+1, next\_state, path + [next\_state]))

return None

def print\_puzzle(state):

"""Print the puzzle state."""

for row in state:

print(row)

# Example Usage:

initial\_state = (

(1, 2, 3),

(5, 6, 0),

(4, 7, 8)

)

print("Initial State:")

print\_puzzle(initial\_state)

solution\_path = a\_star(initial\_state)

if solution\_path:

print("\nSolution found in", len(solution\_path), "steps:")

for state in solution\_path:

print\_puzzle(state)

print()

else:

print("\nNo solution found.")

**8. In a “Sudoku” game each state of the board is a node, and moves are edges. Solve the puzzle by finding the shortest path from the initial state to the solved state using A\* Algorithm.  
  
Code:**

import heapq

class Sudoku:

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

self.board = board

self.size = len(board)

self.goal\_state = self.get\_goal\_state()

def get\_goal\_state(self):

goal = [[(i + j) % self.size + 1 for i in range(self.size)] for j in range(self.size)]

return goal

def is\_solved(self):

return self.board == self.goal\_state

def get\_possible\_moves(self, board):

for i in range(self.size):

for j in range(self.size):

if board[i][j] == 0:

return (i, j)

return None

def is\_valid\_move(self, board, row, col, num):

for i in range(self.size):

if board[row][i] == num or board[i][col] == num:

return False

return True

def apply\_move(self, board, row, col, num):

new\_board = [row[:] for row in board]

new\_board[row][col] = num

return new\_board

def manhattan\_heuristic(self, board):

cost = 0

for i in range(self.size):

for j in range(self.size):

if board[i][j] != 0 and board[i][j] != self.goal\_state[i][j]:

goal\_pos = divmod(board[i][j]-1, self.size)

cost += abs(i - goal\_pos[0]) + abs(j - goal\_pos[1])

return cost

def astar(self):

start = self.board

open\_set = []

heapq.heappush(open\_set, (self.manhattan\_heuristic(start), 0, start, []))

closed\_set = set()

while open\_set:

\_, g\_cost, current\_board, path = heapq.heappop(open\_set)

if tuple(map(tuple, current\_board)) in closed\_set:

continue

if self.is\_solved():

return path

closed\_set.add(tuple(map(tuple, current\_board)))

row, col = self.get\_possible\_moves(current\_board)

if row is not None:

for num in range(1, self.size + 1):

if self.is\_valid\_move(current\_board, row, col, num):

new\_board = self.apply\_move(current\_board, row, col, num)

new\_cost = g\_cost + 1

heapq.heappush(open\_set, (new\_cost + self.manhattan\_heuristic(new\_board), new\_cost, new\_board, path + [new\_board]))

return None

# Test the code with the smaller 4x4 Sudoku board

board = [

[1, 0, 0, 4],

[0, 0, 3, 0],

[4, 3, 0, 0],

[0, 2, 0, 1]

]

sudoku = Sudoku(board)

steps = sudoku.astar()

if steps:

for step in steps:

for row in step:

print(row)

print()

else:

print("No solution found.")

**9.** **In chess, the shortest path to achieve a specific goal (e.g., checkmate) needs to be computed. Nodes represent board states, and edges represent moves. Use A\* to find the sequence of moves that leads to a checkmate in the least number of moves.  
  
Code:-**

import heapq

# Simplified Chessboard Representation

class ChessBoard:

def \_\_init\_\_(self, board, is\_checkmate=False):

self.board = board # A 2D array representing the chessboard

self.is\_checkmate = is\_checkmate # Boolean flag if the board is in a checkmate position

def get\_possible\_moves(self):

# This is a simplified placeholder. A real implementation would generate all legal moves.

# For now, we just return an empty list.

return []

def is\_checkmate\_state(self):

# Check if the current board is a checkmate state.

return self.is\_checkmate

def \_\_repr\_\_(self):

return str(self.board)

# Heuristic: A simple heuristic based on a number of remaining pieces

def heuristic(board):

# For simplicity, let's just count the number of opponent pieces.

opponent\_pieces = sum([1 for row in board for piece in row if piece.islower()])

return opponent\_pieces

# A\* Search to find the shortest path to checkmate

def astar(start\_board):

open\_set = []

heapq.heappush(open\_set, (0 + heuristic(start\_board.board), 0, start\_board, [])) # (f, g, board, path)

closed\_set = set()

while open\_set:

\_, g\_cost, current\_board, path = heapq.heappop(open\_set)

# Check if we have reached a checkmate state

if current\_board.is\_checkmate\_state():

return path

if tuple(map(tuple, current\_board.board)) in closed\_set:

continue

closed\_set.add(tuple(map(tuple, current\_board.board)))

# Generate possible moves and explore

for move in current\_board.get\_possible\_moves():

new\_board = ChessBoard(move) # Applying the move (simplified)

new\_cost = g\_cost + 1

heapq.heappush(open\_set, (new\_cost + heuristic(new\_board.board), new\_cost, new\_board, path + [new\_board]))

return None

# Example initial board setup (simplified representation, e.g., King vs King and Queen)

start\_board = ChessBoard(board=[["K", ".", ".", "."], [".", ".", ".", "K"], [".", ".", ".", "."], [".", ".", ".", "."]], is\_checkmate=False)

# Running A\* to find the shortest path to checkmate

solution = astar(start\_board)

if solution:

for step in solution:

print(step)

else:

print("No solution found.")

**10.** **Develop a stock trading expert system that recommends "Buy," "Sell," or "Hold" based on historical data trends (e.g., moving averages, RSI, or MACD values).**

**Code:**import yfinance as yf

class StockAdvisor:

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

self.ticker = ticker

self.price = None

def fetch\_stock\_price(self):

"""Fetch real-time stock price from Yahoo Finance."""

try:

stock = yf.Ticker(self.ticker)

data = stock.history(period="1d")

if data.empty:

print(f"Error: No data found for {self.ticker}. Please check the ticker symbol.")

return None

self.price = data['Close'].iloc[-1]

return self.price

except Exception as e:

print(f"Error fetching stock data: {e}")

return None

def give\_advice(self, action):

"""Provide advice based on action and stock price."""

if not self.price:

return "Stock price not available. Unable to provide advice."

if action == "buy" and self.price < 500:

return "The stock price is low. It's a good time to BUY."

elif action == "sell" and self.price > 1000:

return "The stock price is high. It's a good time to SELL."

elif action == "hold" and 500 <= self.price <= 1000:

return "The stock price is stable. You may HOLD your position."

else:

return "No specific recommendation. Please check additional market trends."

def main():

print("Welcome to the Stock Market Trading Advisor!")

print("Type 'exit' to quit the system.\n")

while True:

action = input("What do you want to do? (buy, sell, hold): ").strip().lower()

if action == "exit":

print("Thank you for using the system. Goodbye!")

break

ticker = input("Enter the stock ticker symbol (e.g., RELIANCE.BO, TCS.NS): ").strip().upper()

advisor = StockAdvisor(ticker)

if advisor.fetch\_stock\_price():

print(f"Current price of {ticker}: ₹{advisor.price:.2f}")

advice = advisor.give\_advice(action)

print(f"Advice: {advice}")

else:

print("Could not retrieve stock data. Please try again.")

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

main()

**11.** **Design a real-time stock trading system that integrates the expert system with live market data.  
Code:**import yfinance as yf

class StockMarketExpertSystem:

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

self.rules = [] # List to store rules as condition-action pairs

self.data = {} # Dictionary to store stock data

def add\_rule(self, condition, action):

"""Add a rule to the system."""

self.rules.append((condition, action))

def set\_data(self, key, value):

"""Set data for the system to evaluate."""

self.data[key] = value

def evaluate(self):

"""Evaluate all rules and execute actions."""

for condition, action in self.rules:

if condition(self.data):

action(self.data)

# Define conditions for buying, selling, and holding stocks

def condition\_buy(data):

"""Condition to buy: Price is less than the moving average."""

return data['price'] < data['moving\_average']

def action\_buy(data):

"""Action to take if the buy condition is met."""

print(f"Buy Signal: Price ({data['price']}) is below the moving average ({data['moving\_average']}).")

def condition\_sell(data):

"""Condition to sell: Price is greater than the moving average."""

return data['price'] > data['moving\_average']

def action\_sell(data):

"""Action to take if the sell condition is met."""

print(f"Sell Signal: Price ({data['price']}) is above the moving average ({data['moving\_average']}).")

def condition\_hold(data):

"""Condition to hold: Price is close to the moving average."""

threshold = 5 # Define a threshold for "close" prices

return abs(data['price'] - data['moving\_average']) <= threshold

def action\_hold(data):

"""Action to take if the hold condition is met."""

print(f"Hold Signal: Price ({data['price']}) is close to the moving average ({data['moving\_average']}).")

def fetch\_stock\_data(symbol, period="1mo", interval="1d"):

"""

Fetch stock data and company name using yfinance.

:param symbol: Stock ticker symbol (e.g., 'RELIANCE.NS' for Reliance Industries on NSE).

:param period: Period for which to fetch data (e.g., '1mo' for 1 month).

:param interval: Data interval (e.g., '1d' for daily data).

:return: Tuple (company\_name, current\_price, moving\_average)

"""

stock = yf.Ticker(symbol)

# Fetch historical data

hist = stock.history(period=period, interval=interval)

if hist.empty:

raise ValueError("No historical data available for the given ticker symbol.")

# Get company name from metadata

company\_name = stock.info.get('shortName', 'Unknown Company')

# Get the latest closing price and calculate the moving average

current\_price = hist['Close'].iloc[-1] # Last closing price

moving\_average = hist['Close'].mean() # Average closing price over the period

return company\_name, current\_price, moving\_average

def run\_expert\_system():

# Fetch stock data for a specific symbol

symbol = input("Enter the stock ticker symbol (e.g., RELIANCE.NS, TCS.NS): ").strip()

print("\nFetching data...")

try:

# Get the company name, current price, and moving average

company\_name, current\_price, moving\_average = fetch\_stock\_data(symbol)

print(f"Company Name: {company\_name}")

print(f"Current Price: {current\_price}")

print(f"Moving Average: {moving\_average}\n")

# Create an instance of the expert system

expert\_system = StockMarketExpertSystem()

# Add rules to the system

expert\_system.add\_rule(condition\_buy, action\_buy)

expert\_system.add\_rule(condition\_sell, action\_sell)

expert\_system.add\_rule(condition\_hold, action\_hold)

# Set the fetched data in the expert system

expert\_system.set\_data('price', current\_price)

expert\_system.set\_data('moving\_average', moving\_average)

# Evaluate the rules

print("Stock Market Trading Recommendations:")

expert\_system.evaluate()

except Exception as e:

print(f"Error fetching data for {symbol}: {e}")

# Run the program

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

run\_expert\_system()

**12.** **Design the expert system to manage a portfolio by dynamically reallocating investments based on market conditions and individual stock performance.**

**Code:**

class Stock:

def \_\_init\_\_(self, symbol, price, pe\_ratio, rsi, moving\_avg\_50, moving\_avg\_200):

self.symbol = symbol

self.price = price

self.pe\_ratio = pe\_ratio

self.rsi = rsi

self.moving\_avg\_50 = moving\_avg\_50

self.moving\_avg\_200 = moving\_avg\_200

class Portfolio:

def \_\_init\_\_(self, initial\_cash):

self.cash = initial\_cash

self.stocks = {}

def buy\_stock(self, stock, quantity):

if self.cash >= stock.price \* quantity:

self.cash -= stock.price \* quantity

if stock.symbol in self.stocks:

self.stocks[stock.symbol] += quantity

else:

self.stocks[stock.symbol] = quantity

print(f"Bought {quantity} shares of {stock.symbol} at {stock.price} each.")

else:

print("Not enough cash to buy stock.")

def sell\_stock(self, stock, quantity):

if stock.symbol in self.stocks and self.stocks[stock.symbol] >= quantity:

self.cash += stock.price \* quantity

self.stocks[stock.symbol] -= quantity

print(f"Sold {quantity} shares of {stock.symbol} at {stock.price} each.")

else:

print("Not enough shares to sell.")

def get\_total\_value(self, market\_data):

total\_value = self.cash

for symbol, quantity in self.stocks.items():

stock\_price = market\_data[symbol].price

total\_value += stock\_price \* quantity

return total\_value

class ExpertSystem:

def \_\_init\_\_(self, portfolio, market\_data):

self.portfolio = portfolio

self.market\_data = market\_data

def assess\_market\_conditions(self, market\_data):

recommendations = []

for stock\_symbol, stock in market\_data.items():

# Rule 1: Sell if stock price is below its 50-day moving average

if stock.price < stock.moving\_avg\_50:

recommendations.append(f"Consider selling {stock\_symbol}, price below 50-day moving average.")

# Rule 2: Sell if RSI is above 70

if stock.rsi > 70:

recommendations.append(f"Consider selling {stock\_symbol}, RSI is overbought.")

# Rule 3: Buy if P/E ratio is low (underpriced stock)

if stock.pe\_ratio < 15:

recommendations.append(f"Consider buying {stock\_symbol}, stock is undervalued.")

return recommendations

def reallocate\_portfolio(self):

recommendations = self.assess\_market\_conditions(self.market\_data)

for rec in recommendations:

print(rec)

# Example data

market\_data = {

"AAPL": Stock("AAPL", price=150, pe\_ratio=25, rsi=60, moving\_avg\_50=145, moving\_avg\_200=135),

"MSFT": Stock("MSFT", price=280, pe\_ratio=30, rsi=75, moving\_avg\_50=275, moving\_avg\_200=265),

"GOOG": Stock("GOOG", price=2800, pe\_ratio=30, rsi=50, moving\_avg\_50=2800, moving\_avg\_200=2750)

}

# Portfolio initialization

portfolio = Portfolio(10000) # $10,000 initial cash

# Expert system setup

expert\_system = ExpertSystem(portfolio, market\_data)

# Reallocate portfolio based on market conditions

expert\_system.reallocate\_portfolio()

**13.** **Develop an elementary Chabot for “Airline Booking” customer interaction application.**

**Code:**

import re

class AirlineChatbot:

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

self.greetings\_responses = [

"Hello! Welcome to Airline Booking. How can I assist you today?",

"Hi there! I'm here to help you with your flight queries. What do you need assistance with?"

]

self.farewell\_responses = [

"Thank you for using Airline Booking! Have a great day!",

"Goodbye! Feel free to reach out if you need further assistance."

]

self.flight\_details = {

"123": {"destination": "New York", "time": "10:00 AM", "price": 350},

"456": {"destination": "London", "time": "02:00 PM", "price": 550},

"789": {"destination": "Tokyo", "time": "09:00 PM", "price": 700},

}

def greet(self, message):

if re.search(r"hello|hi|hey", message, re.IGNORECASE):

return self.greetings\_responses[0]

def farewell(self, message):

if re.search(r"bye|goodbye|see you", message, re.IGNORECASE):

return self.farewell\_responses[0]

def search\_flights(self, destination):

results = [

flight\_id

for flight\_id, details in self.flight\_details.items()

if destination.lower() in details["destination"].lower()

]

if results:

response = f"Flights available to {destination}:\n"

for flight\_id in results:

details = self.flight\_details[flight\_id]

response += f"- Flight {flight\_id}: {details['time']} at ${details['price']}\n"

return response

return f"Sorry, no flights available to {destination}."

def book\_flight(self, flight\_id):

if flight\_id in self.flight\_details:

details = self.flight\_details[flight\_id]

return f"Flight {flight\_id} to {details['destination']} at {details['time']} has been successfully booked for ${details['price']}!"

return "Invalid flight ID. Please provide a valid flight number."

def handle\_query(self, message):

if "search" in message and "flight" in message:

match = re.search(r"to (\w+)", message)

if match:

destination = match.group(1)

return self.search\_flights(destination)

return "Please specify the destination, e.g., 'Search flights to New York.'"

if "book" in message and "flight" in message:

match = re.search(r"flight (\d+)", message)

if match:

flight\_id = match.group(1)

return self.book\_flight(flight\_id)

return "Please specify the flight number, e.g., 'Book flight 123.'"

return "I'm sorry, I didn't understand that. Can you please rephrase?"

def chat(self, message):

response = self.greet(message) or self.farewell(message) or self.handle\_query(message)

return response

# Example Interaction

chatbot = AirlineChatbot()

print("Chatbot: Hello! Welcome to Airline Booking.")

while True:

user\_input = input("You: ")

if re.search(r"bye|goodbye", user\_input, re.IGNORECASE):

print(f"Chatbot: {chatbot.farewell(user\_input)}")

break

print(f"Chatbot: {chatbot.chat(user\_input)}")

**14. Develop an elementary Chabot for “Hotel Room Booking” customer interaction application.  
code:**

import re

class AirlineChatbot:

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

self.greetings\_responses = [

"Hello! Welcome to Airline Booking. How can I assist you today?",

"Hi there! I'm here to help you with your flight queries. What do you need assistance with?"

]

self.farewell\_responses = [

"Thank you for using Airline Booking! Have a great day!",

"Goodbye! Feel free to reach out if you need further assistance."

]

self.flight\_details = {

"123": {"destination": "New York", "time": "10:00 AM", "price": 350},

"456": {"destination": "London", "time": "02:00 PM", "price": 550},

"789": {"destination": "Tokyo", "time": "09:00 PM", "price": 700},

}

def greet(self, message):

if re.search(r"hello|hi|hey", message, re.IGNORECASE):

return self.greetings\_responses[0]

def farewell(self, message):

if re.search(r"bye|goodbye|see you", message, re.IGNORECASE):

return self.farewell\_responses[0]

def search\_flights(self, destination):

results = [

flight\_id

for flight\_id, details in self.flight\_details.items()

if destination.lower() in details["destination"].lower()

]

if results:

response = f"Flights available to {destination}:\n"

for flight\_id in results:

details = self.flight\_details[flight\_id]

response += f"- Flight {flight\_id}: {details['time']} at ${details['price']}\n"

return response

return f"Sorry, no flights available to {destination}."

def book\_flight(self, flight\_id):

if flight\_id in self.flight\_details:

details = self.flight\_details[flight\_id]

return f"Flight {flight\_id} to {details['destination']} at {details['time']} has been successfully booked for ${details['price']}!"

return "Invalid flight ID. Please provide a valid flight number."

def handle\_query(self, message):

if "search" in message and "flight" in message:

match = re.search(r"to (\w+)", message)

if match:

destination = match.group(1)

return self.search\_flights(destination)

return "Please specify the destination, e.g., 'Search flights to New York.'"

if "book" in message and "flight" in message:

match = re.search(r"flight (\d+)", message)

if match:

flight\_id = match.group(1)

return self.book\_flight(flight\_id)

return "Please specify the flight number, e.g., 'Book flight 123.'"

return "I'm sorry, I didn't understand that. Can you please rephrase?"

def chat(self, message):

response = self.greet(message) or self.farewell(message) or self.handle\_query(message)

return response

# Example Interaction

chatbot = AirlineChatbot()

print("Chatbot: Hello! Welcome to Airline Booking.")

while True:

user\_input = input("You: ")

if re.search(r"bye|goodbye", user\_input, re.IGNORECASE):

print(f"Chatbot: {chatbot.farewell(user\_input)}")

break

print(f"Chatbot: {chatbot.chat(user\_input)}")

**15. Develop an elementary Chabot for “Admission to college” student interaction application.**

**Code:**

import re

class CollegeAdmissionChatbot:

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

self.greetings\_responses = [

"Hello! Welcome to XYZ College Admission Help. How can I assist you?",

"Hi! I'm here to answer your questions about college admissions. What would you like to know?"

]

self.farewell\_responses = [

"Thank you for contacting XYZ College Admission Help. Best of luck!",

"Goodbye! Feel free to reach out if you have more questions."

]

self.courses = {

"B.Tech": {"duration": "4 years", "fee": "INR 1,00,000/year"},

"BBA": {"duration": "3 years", "fee": "INR 75,000/year"},

"MBA": {"duration": "2 years", "fee": "INR 1,50,000/year"},

"MCA": {"duration": "2 years", "fee": "INR 1,20,000/year"}

}

self.admission\_requirements = {

"B.Tech": "10+2 with Physics, Chemistry, Mathematics (50% minimum)",

"BBA": "10+2 in any stream (50% minimum)",

"MBA": "Graduation in any stream (50% minimum) + Entrance Exam",

"MCA": "Graduation in Computer Applications/Science (50% minimum)"

}

def greet(self, message):

if re.search(r"hello|hi|hey", message, re.IGNORECASE):

return self.greetings\_responses[0]

def farewell(self, message):

if re.search(r"bye|goodbye|see you", message, re.IGNORECASE):

return self.farewell\_responses[0]

def provide\_course\_info(self, course\_name):

course\_name = course\_name.title()

if course\_name in self.courses:

details = self.courses[course\_name]

return (f"{course\_name} Course Details:\n"

f"- Duration: {details['duration']}\n"

f"- Fee: {details['fee']}\n"

f"- Admission Requirement: {self.admission\_requirements[course\_name]}")

return "Sorry, I don't have information about that course. Please specify another course."

def explain\_admission\_process(self):

return ("Admission Process:\n"

"1. Submit the online application form at our website.\n"

"2. Appear for the entrance exam (if applicable).\n"

"3. Attend the counseling session and select your course.\n"

"4. Pay the admission fee and confirm your seat.\n")

def handle\_query(self, message):

if "course" in message or "details" in message:

match = re.search(r"course (\w+)", message)

if match:

course\_name = match.group(1)

return self.provide\_course\_info(course\_name)

return "Please specify the course you are asking about, e.g., 'Course B.Tech'."

if "admission process" in message or "apply" in message:

return self.explain\_admission\_process()

if "requirements" in message:

match = re.search(r"requirements for (\w+)", message)

if match:

course\_name = match.group(1).title()

if course\_name in self.admission\_requirements:

return f"Admission Requirements for {course\_name}: {self.admission\_requirements[course\_name]}"

return "Sorry, I don't have information about that course's requirements."

return "Please specify the course you are asking about, e.g., 'Requirements for B.Tech'."

return "I'm sorry, I didn't understand that. Could you please rephrase?"

def chat(self, message):

response = self.greet(message) or self.farewell(message) or self.handle\_query(message)

return response

# Example Interaction

chatbot = CollegeAdmissionChatbot()

print("Chatbot: Hello! Welcome to XYZ College Admission Help.")

while True:

user\_input = input("You: ")

if re.search(r"bye|goodbye", user\_input, re.IGNORECASE):

print(f"Chatbot: {chatbot.farewell(user\_input)}")

break

print(f"Chatbot: {chatbot.chat(user\_input)}")

**16**. **Develop an elementary Chabot “Hospital” patient interaction application.  
Code:**

import re

class HospitalChatbot:

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

self.greetings\_responses = [

"Hello! Welcome to ABC Hospital Helpdesk. How can I assist you today?",

"Hi! This is ABC Hospital's virtual assistant. How can I help you?"

]

self.farewell\_responses = [

"Thank you for reaching out to ABC Hospital. Take care!",

"Goodbye! Stay healthy and feel free to contact us if needed."

]

self.doctors\_info = {

"Cardiologist": {"name": "Dr. Smith", "availability": "Monday, Wednesday, Friday"},

"Dermatologist": {"name": "Dr. Taylor", "availability": "Tuesday, Thursday"},

"Pediatrician": {"name": "Dr. Lee", "availability": "Monday to Friday"},

"Orthopedic": {"name": "Dr. Brown", "availability": "Tuesday, Thursday, Saturday"}

}

self.visiting\_hours = "Monday to Sunday: 10 AM - 6 PM"

self.emergency\_contact = "Call 911 or reach our emergency department at +123-456-7890."

def greet(self, message):

if re.search(r"hello|hi|hey", message, re.IGNORECASE):

return self.greetings\_responses[0]

def farewell(self, message):

if re.search(r"bye|goodbye|see you", message, re.IGNORECASE):

return self.farewell\_responses[0]

def provide\_doctor\_info(self, specialization):

specialization = specialization.title()

if specialization in self.doctors\_info:

doctor = self.doctors\_info[specialization]

return (f"Doctor Specialization: {specialization}\n"

f"- Name: {doctor['name']}\n"

f"- Availability: {doctor['availability']}")

return "Sorry, we currently do not have information about that specialization."

def explain\_visiting\_hours(self):

return f"Hospital Visiting Hours: {self.visiting\_hours}"

def handle\_emergency\_query(self):

return f"For emergencies: {self.emergency\_contact}"

def book\_appointment(self, specialization):

specialization = specialization.title()

if specialization in self.doctors\_info:

doctor = self.doctors\_info[specialization]

return (f"Appointment Booking:\n"

f"Specialization: {specialization}\n"

f"Doctor: {doctor['name']}\n"

f"Available Days: {doctor['availability']}\n"

"Please visit our website or call us at +123-456-7890 to confirm your appointment.")

return "Sorry, we currently do not have doctors for that specialization."

def handle\_query(self, message):

if "doctor" in message or "specialist" in message:

match = re.search(r"doctor (\w+)", message)

if match:

specialization = match.group(1)

return self.provide\_doctor\_info(specialization)

return "Please specify the doctor's specialization, e.g., 'Doctor Cardiologist'."

if "visiting hours" in message:

return self.explain\_visiting\_hours()

if "emergency" in message:

return self.handle\_emergency\_query()

if "appointment" in message:

match = re.search(r"appointment with (\w+)", message)

if match:

specialization = match.group(1)

return self.book\_appointment(specialization)

return "Please specify the doctor's specialization for the appointment, e.g., 'Appointment with Pediatrician'."

return "I'm sorry, I didn't understand that. Could you please rephrase?"

def chat(self, message):

response = self.greet(message) or self.farewell(message) or self.handle\_query(message)

return response

# Example Interaction

chatbot = HospitalChatbot()

print("Chatbot: Hello! Welcome to ABC Hospital Helpdesk.")

while True:

user\_input = input("You: ")

if re.search(r"bye|goodbye", user\_input, re.IGNORECASE):

print(f"Chatbot: {chatbot.farewell(user\_input)}")

break

print(f"Chatbot: {chatbot.chat(user\_input)}")