

# AI CONCEPT

Great question! Both **BFS** (Breadth-First Search) and **DFS** (Depth-First Search) are fundamental graph traversal algorithms, and they have **real-world applications** in many fields like web crawling, networking, AI, games, and more.

---

## ● Real-Time Use Cases of BFS (Breadth-First Search)

BFS Use Case	Explanation
<b>1. Shortest Path in Maps</b>	BFS is used in GPS systems (like Google Maps) to find the <b>shortest path</b> from one location to another in unweighted graphs.
<b>2. Social Network Friend Suggestions</b>	Facebook, LinkedIn, etc., use BFS to suggest friends by finding people within <b>N levels of connection</b> (friends of friends).
<b>3. Web Crawlers</b>	Web crawlers use BFS to index web pages level by level, starting from the homepage and moving outward.
<b>4. Broadcasting in Networks</b>	BFS is used in networking to broadcast information/data to <b>all nodes</b> from a source.
<b>5. AI in Games (Shortest Moves)</b>	BFS helps find the shortest sequence of moves in games like chess or puzzles (like solving Rubik's cube in fewest moves).

---

## ● Real-Time Use Cases of DFS (Depth-First Search)

DFS Use Case	Explanation
<b>1. File System Traversal</b>	DFS is used to explore <b>all directories and subdirectories</b> (e.g., when searching for a file).
<b>2. Detecting Cycles in Graphs</b>	DFS helps detect <b>cycles</b> in graphs, useful in compilers, deadlock detection in OS, etc.
<b>3. Topological Sorting</b>	In course scheduling or task dependency resolution, DFS is used to sort tasks in a valid order (e.g., Task A before Task B).

DFS Use Case	Explanation
<b>4. Maze or Puzzle Solving</b>	DFS explores <b>deep paths first</b> and backtracks when necessary — useful in finding all solutions (not just shortest).
<b>5. Connected Components</b>	DFS helps find connected groups in social networks, islands in 2D matrices, etc.

---

## Which One to Use When?

Need	Use
<b>Find shortest path (unweighted)</b>	 <b>BFS</b>
<b>Explore all paths deeply or recursively</b>	 <b>DFS</b>
<b>Memory-efficient for deep graphs</b>	 <b>DFS</b>
<b>Level-order exploration</b>	 <b>BFS</b>

---

Would you like a small real-life example implemented in Python to see one of these in action?

4o

**You said:**

explain bfs and dfs

**ChatGPT said:**

Sure! Let's break down **BFS (Breadth-First Search)** and **DFS (Depth-First Search)** in a simple and clear way — with their meaning, working, differences, and when to use them.

## BFS (Breadth-First Search)

### What is it?

BFS is an algorithm for traversing or searching a graph/tree **level by level** — meaning, it explores all neighbors of a node **before going deeper**.

### How it works:

1. Start from a selected node (called the **source**).

2. Visit the node and add it to a **queue**.
3. Repeatedly:
  - o Remove the node from the queue.
  - o Visit all its unvisited neighbors and add them to the queue.
4. Stop when all nodes are visited.

### **Data Structure Used:**

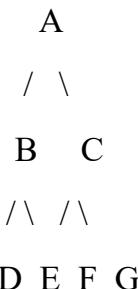
- **Queue (FIFO)**

### **Example:**

In a tree like this:

mathematica

CopyEdit



BFS traversal: A B C D E F G

### **Real-life analogy:**

Imagine a ripple in water: BFS spreads outwards from the center node — layer by layer.

---

```

graph={

'A':['B','C'],
'B':['D','E'],
'C':['F','G'],
'D':['H','I'],
'E:[],
'F':['J','K'],
  
```

```
'G':[],  
'H':[],  
'I':[],  
'J':[],  
'K':[]  
}  
  
# BFS Implementation  
def bfs(visited, graph, node):  
    visited = []  
    queue = []  
  
    visited.append(node)  
    queue.append(node)  
  
    while queue:  
        m = queue.pop(0)  
        print(m, end=" ")  
  
        for neighbour in graph[m]:  
            if neighbour not in visited:  
                visited.append(neighbour)  
                queue.append(neighbour)  
  
    print("\nFollowing is the BFS traversal:")  
bfs([], graph, 'A')
```

```

# DFS Implementation

def dfs(visited, graph, node):

    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

print("\nFollowing is the DFS traversal:")

dfs(set(), graph, 'A')

```

## ● DFS (Depth-First Search)

### 📌 What is it?

DFS is an algorithm that goes **deep into one path** of a graph/tree before backtracking and exploring other paths.

### ⚙️ How it works:

1. Start from the source node.
2. Visit the node, mark as visited.
3. Recursively visit each unvisited neighbor.
4. Backtrack when no unvisited neighbors remain.

### 📦 Data Structure Used:

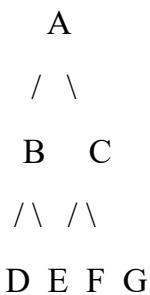
- **Stack (LIFO)** — usually handled **implicitly** with recursion.

### 🧠 Example:

Same tree:

mathematica

CopyEdit



DFS traversal (preorder): A B D E C F G

### Real-life analogy:

Imagine exploring a maze — DFS goes **as deep as possible** down one path before turning back and trying another.

---

## Difference Between BFS and DFS

Feature	BFS	DFS
Strategy	Explore level by level	Explore depth first
Data Structure	Queue	Stack / Recursion
Memory Usage	More (stores levels)	Less (stores deep path)
Shortest Path (Unweighted)	 Yes	 Not guaranteed
Cycle Detection	Yes	Yes
Recursive?	Not usually	Yes, naturally
Faster for Deep Goals		

---

## When to Use Which?

- Use **BFS** when:
  - You need the **shortest path** (like in unweighted graphs).
  - You want to explore nearby nodes first.
- Use **DFS** when:
  - You want to explore all possible paths or components.
  - You want to detect cycles, connected components, or topological order.

## ◆ BFS Function Definition

python

CopyEdit

```
def bfs(visited , graph , node):
```

- Defines the **Breadth-First Search (BFS)** function.
- Parameters:
  - visited: list to track visited nodes.
  - graph: the graph you want to traverse.
  - node: the **starting node**.

python

CopyEdit

```
visited = []
```

```
queue =[]
```

- Initializes visited as an empty list (tracks visited nodes).
- Initializes queue as an empty list (for nodes to be processed next).

python

CopyEdit

```
visited.append(node)
```

```
queue.append(node)
```

- Adds the starting node to both visited and queue.

python

CopyEdit

```
while queue:
```

- Loops until the queue is empty.

python

CopyEdit

```
m = queue.pop(0)
```

- Removes and returns the **first node** from the queue (FIFO — First In First Out).

python

CopyEdit

```
print(m,end=" ")
```

- Prints the current node with a space.

python

CopyEdit

```
for neighbour in graph[m]:
```

- Loops through all neighbors (child nodes) of the current node m.

python

CopyEdit

```
if neighbour not in visited:
```

- Checks if this neighbor hasn't been visited yet.

python

CopyEdit

```
visited.append(neighbour)
```

```
queue.append(neighbour)
```

- Adds the neighbor to both visited and queue to process it later.
- 

### ◆ Calling BFS

python

CopyEdit

```
print("\n BFS Traversal : ")
```

```
bfs([],graph,'A')
```

- Prints a label for BFS.
- Calls the bfs() function with:
  - An empty list for visited (though it's overridden inside).
  - The graph defined above.

- Starting from node 'A'.
- 

### ◆ DFS Function Definition

python

CopyEdit

```
def dfs(visited ,graph, node):
```

- Defines the **Depth-First Search (DFS)** function.
- Parameters:
  - visited: a **set** to store visited nodes.
  - graph: the graph to traverse.
  - node: starting node.

python

CopyEdit

```
if node not in visited :
```

- If the node hasn't been visited yet...

python

CopyEdit

```
    print(node,end=" ")
```

- Print the node.

python

CopyEdit

```
    visited.add(node)
```

- Mark the node as visited (adds it to the set).

python

CopyEdit

```
        for neighbour in graph[node]:
```

```
            dfs(visited,graph,neighbour)
```

- For every neighbor (child node), call `dfs()` **recursively**.
-

## ◆ Calling DFS

python

CopyEdit

```
print("DFS traversal :")
```

```
dfs(set(),graph,'A')
```

- Prints a label for DFS.
- Calls dfs() with:
  - An empty set set() for visited nodes.
  - The graph.
  - Starting from node 'A'.

---

## ✓ Key Concepts Summary

Concept	BFS	DFS
Data Structure	Queue	Stack (implicitly via recursion)
Traversal Order	Level by level	Depth first, then backtrack
Use Case	Shortest path, level order	Topological sort, maze solving

Would you like me to visualize how BFS and DFS traverse this graph step by step?

## Explanation of BFS and DFS

### ◆ Breadth-First Search (BFS)

- **Type:** Level-order traversal.
- **Data structure used:** Queue.
- **How it works:**
  1. Start from the root node.
  2. Visit all its neighbors before moving to their neighbors.

- **Application:** Shortest path in unweighted graphs, peer-to-peer networks.

**Example Output** for your graph:

mathematica

CopyEdit

BFS Traversal:

A B C D E F G H I J K

---

#### ◆ Depth-First Search (DFS)

- **Type:** Deep traversal.
- **Data structure used:** Stack (implicitly via recursion).
- **How it works:**
  1. Start from the root node.
  2. Go as deep as possible in one direction before backtracking.
- **Application:** Maze solving, topological sorting, cycle detection.

**Example Output** for your graph:

mathematica

CopyEdit

DFS Traversal:

A B D H I E C F J K G

---

### ❓ Viva Questions & Answers (with Expected Answers)

#### 📌 Basic Understanding

##### 1. Q: What is BFS?

A: BFS is a graph traversal algorithm that explores neighbors level by level using a queue.

##### 2. Q: What is DFS?

A: DFS is a graph traversal algorithm that goes deep into each path before backtracking, using recursion or a stack.

##### 3. Q: What's the difference between BFS and DFS?

A:

BFS	DFS
Uses Queue	Uses Stack / Recursion
Level-wise traversal	Depth-wise traversal
Better for shortest paths	Better for full exploration

---

## 📌 Code & Logic Questions

4. **Q: What data structure is used in BFS and DFS?**

**A:**

- BFS → Queue
- DFS → Stack (or Recursion)

5. **Q: Why do we use a visited list or set?**

**A:** To avoid revisiting nodes and prevent infinite loops in cyclic graphs.

6. **Q: Is DFS recursive or iterative?**

**A:** DFS can be both, but in your code it is **recursive**.

7. **Q: Can BFS be recursive?**

**A:** BFS is generally implemented using iteration with a queue because recursion doesn't naturally fit its level-based logic.

8. **Q: How will the traversal differ if the graph is directed?**

**A:** In a directed graph, edges have direction, so BFS/DFS will only follow edges from parent to child, not vice versa.

---

## 📌 Advanced Questions

9. **Q: Time complexity of BFS and DFS?**

**A:**

- BFS:  $O(V + E)$
- DFS:  $O(V + E)$   
Where V = vertices, E = edges.

10. **Q: Space complexity of BFS and DFS?**

**A:**

- BFS:  $O(V)$  for queue and visited list
- DFS:  $O(V)$  for recursion stack and visited set

**11. Q: What will happen if we don't mark nodes as visited?**

**A:** It may result in an infinite loop in graphs with cycles.

**12. Q: Can BFS or DFS be used to find connected components?**

**A:** Yes, by applying BFS/DFS to unvisited nodes repeatedly.

**13. Q: How is DFS helpful in topological sorting?**

**A:** DFS can be used to perform topological sorting of a Directed Acyclic Graph (DAG) by recording the finish time of each node.

/

# What is A\* Algorithm?

A\* (pronounced "A-star") is a **graph traversal and pathfinding algorithm** used to find the **shortest path** between a **start node** and a **goal node**.

It combines the **best parts of Dijkstra's Algorithm** (which finds the shortest path) and **Greedy Best-First Search** (which uses heuristics to reach the goal faster).

---

## Key Idea

A\* uses a **cost function** to choose the next node to explore:

- ◆  $f(n) = g(n) + h(n)$ 
    - $f(n)$  = Total estimated cost of the cheapest path through node n.
    - $g(n)$  = Actual cost from the start node to node n.
    - $h(n)$  = Heuristic — estimated cost from node n to the goal (like straight-line distance).
- 

## Data Structures Used

- **Priority Queue** (or Min-Heap): To always explore the node with the **lowest  $f(n)$**  first.
  - **Set or Dictionary**: To track visited nodes and the best cost found so far.
- 

## How A\* Works (Step-by-Step)

1. Start with the **initial node** and put it in the **open set** (priority queue).
2. Calculate  $f(n)$  for each node.
3. While the open set is not empty:
  - Choose the node with the **lowest  $f(n)$** .
  - If it's the goal node, return the path.
  - Otherwise, for each neighbor:
    - Calculate  $g(n)$  (cost from start to this neighbor).
    - Estimate  $h(n)$  using a heuristic (e.g., Euclidean or Manhattan distance).

- Update if the new path is better.
4. Repeat until the goal is reached.
- 

### 📍 Example: Grid Pathfinding

Imagine you're in a grid and want to move from **Start (S)** to **Goal (G)**:

nginx

CopyEdit

S . . . .

. # # ..

. . . . G

Where:

- . is walkable space
- # is an obstacle

A\* uses the shortest path **while avoiding obstacles** and considering both the **actual distance traveled (g)** and the **estimated remaining distance (h)**.

---

### ✅ Advantages of A\*

Benefit	Description
⭐ Accurate	Always finds the <b>shortest path</b> , if one exists (if heuristic is admissible)
🚀 Efficient	Explores fewer paths than Dijkstra's algorithm
🎯 Goal-Directed	Thanks to heuristic guidance ( $h(n)$ )
🔧 Flexible	Can be customized with different heuristics (Euclidean, Manhattan, etc.)

---

### ❗ Limitations

- Can be **slow or memory-intensive** in very large graphs if the heuristic is poor.
- Requires a good **heuristic function** to perform well.

---

## Common Heuristics Used in A\*

Heuristic	Use Case
<b>Manhattan Distance</b>	Grid-based maps (no diagonal movement)
<b>Euclidean Distance</b>	Real-world distance in 2D/3D space
<b>Zero Heuristic</b>	Makes A* behave like Dijkstra's algorithm

---

Would you like me to show a Python code example for A\* on a grid?

4o

**You said:**

```
graph = {  
    'A': {'B': 1, 'C': 3},  
    'B': {'D': 3, 'E': 1},  
    'C': {'F': 5},  
    'D': {'G': 2},  
    'E': {'G': 2},  
    'F': {'G': 1},  
    'G': {}  
}
```

```
heuristic = {  
    'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 1, 'G': 0  
}
```

```
def a_star(start, goal):  
    open_list = {start}  
    g = {start: 0}  
    parents = {start: None}
```

```

while open_list:
    current = min(open_list, key=lambda x: g[x] + heuristic[x]) # Get node with
    lowest cost

    if current == goal:
        path = []
        while current:
            path.append(current)
            current = parents[current]
        return path[::-1] # Return the path from start to goal

    open_list.remove(current)
    for neighbor, cost in graph[current].items():
        temp_g = g[current] + cost
        if neighbor not in g or temp_g < g[neighbor]:
            g[neighbor] = temp_g
            parents[neighbor] = current
            open_list.add(neighbor)

return None

```

```

# Run A* from A to G
print("Path:", a_star('A', 'G'))

```

**ChatGPT said:**

Sure, Kalyan! Here are some **common and insightful questions** an external examiner might ask you about this **A\* algorithm practical**, especially in a viva or lab exam:

---

#### ◆ Basic Understanding

1. **What is the A\* algorithm used for?**
2. **What are the key components of the A\* algorithm?**

3. Explain the formula  $f(n) = g(n) + h(n)$  in your code.
  4. What do  $g$  and heuristic represent in your code?
  5. What is the role of the `open_list` set?
- 

◆ **Code-Specific Questions**

6. Why do you use `min(open_list, key=lambda x: g[x] + heuristic[x])`?
  7. What does the parents dictionary do?
  8. How does your code reconstruct the path at the end?
  9. What happens if there is no path between start and goal?
  10. Why do we check if neighbor not in  $g$  or  $\text{temp\_}g < g[\text{neighbor}]$ ?
- 

◆ **Graph and Heuristic Questions**

11. How is your graph represented in Python?
  12. What kind of graph is this — weighted, unweighted, directed, or undirected?
  13. What is a heuristic function and how did you choose these values?
  14. What would happen if the heuristic values are all zero?
  15. Can the A\* algorithm work without a heuristic? What will it behave like?
- 

◆ **Real-World & Conceptual**

16. Give real-world examples where A\* is used.
  17. What are some common heuristics used in A\*?
  18. How is A\* different from Dijkstra's algorithm?
  19. What is an admissible heuristic? Is your heuristic admissible?
  20. What are the limitations of the A\* algorithm?
- 

✓ **Tips to Prepare**

- Be able to trace the algorithm step-by-step on paper with your given graph.
- Know how the path is reconstructed from goal back to start.

- Understand what would happen with **different heuristic values**.

Let's go step-by-step to **understand the A\* (A-star) algorithm code** you've written for finding the shortest path in a graph using heuristics.

---

## ✓ Overview

- You are trying to find the **shortest path** from node 'A' to 'G' using the **A\* algorithm**.
- The A\* algorithm combines:
  - **Actual cost so far** (called  $g(n)$ )
  - **Heuristic estimate to the goal** (called  $h(n)$ )
  - Total cost  $f(n) = g(n) + h(n)$

## ◆ Input Structures

**Graph with edge weights:**

python

CopyEdit

```
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'D': 3, 'E': 1},
    'C': {'F': 5},
    'D': {'G': 2},
    'E': {'G': 2},
    'F': {'G': 1},
    'G': {}
}
```

- Each key is a node.

- The values are dictionaries: {neighbor: cost}.
  - E.g., from 'A' you can go to 'B' with cost 1 and 'C' with cost 3.
- 

### Heuristic values:

python

CopyEdit

```
heuristic = {
```

```
    'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 1, 'G': 0
```

```
}
```

- Estimated cost from each node to the goal 'G'.
  - Heuristic is used to guide the search.
- 

## 💡 Function Explanation

### Function Signature

python

CopyEdit

```
def a_star(start, goal):
```

- start: the starting node (e.g., 'A')
  - goal: the target node (e.g., 'G')
- 

### Step 1: Initialization

python

CopyEdit

```
open_list = {start}
```

```
g = {start: 0}
```

```
parents = {start: None}
```

- open\_list: Set of nodes to explore (like a priority queue). Starts with 'A'.
- g: Dictionary holding actual cost from start to any node. Initially, cost to reach 'A' is 0.

- parents: To trace the path back. 'A' has no parent.
- 

### Step 2: Loop while open\_list is not empty

python

CopyEdit

```
while open_list:
```

The algorithm runs as long as there are nodes to explore.

---

### Step 3: Pick the best node from open list

python

CopyEdit

```
current = min(open_list, key=lambda x: g[x] + heuristic[x])
```

- Select the node  $x$  in open\_list with the **lowest  $f(x) = g(x) + h(x)$** .
  - This is the A\* principle — choosing the most promising node.
- 

### Step 4: Goal Test

python

CopyEdit

```
if current == goal:
```

```
    path = []
```

```
    while current:
```

```
        path.append(current)
```

```
        current = parents[current]
```

```
    return path[::-1]
```

- If the current node is the goal:
    - Trace back the path using the parents dictionary.
    - Reverse it and return the correct path from start to goal.
- 

### Step 5: Explore Neighbors

python

CopyEdit

```
open_list.remove(current)  
for neighbor, cost in graph[current].items():
```

```
    temp_g = g[current] + cost
```

- Remove current from open\_list (we're done with it).
  - For each neighbor of current, calculate the new cost temp\_g.
- 

## Step 6: Relaxation / Update Logic

python

CopyEdit

```
if neighbor not in g or temp_g < g[neighbor]:
```

```
    g[neighbor] = temp_g
```

```
    parents[neighbor] = current
```

```
    open_list.add(neighbor)
```

- If this path is **better** than a previously found one:
    - Update g value for neighbor.
    - Set current as its parent.
    - Add the neighbor to the open\_list.
- 

## Step 7: If goal not found

python

CopyEdit

```
return None
```

- If open\_list is empty and goal not reached, return None.
- 



## Running the function

python

CopyEdit

```
print("Path:", a_star('A', 'G'))
```

◆ **Output:**

pgsql

CopyEdit

Path: ['A', 'B', 'E', 'G']

- A\* finds the **shortest cost path** from 'A' to 'G' as:  
'A' → 'B' → 'E' → 'G'
- 

📌 **Summary**

**Component Purpose**

$g[x]$  Cost from start to  $x$

$heuristic[x]$  Estimated cost from  $x$  to goal

$f(x)$  Total estimated cost (used for choosing)

parents To reconstruct path after reaching goal

open\_list Nodes yet to be explored

---

Would you like:

- A graph diagram for this A\* example?
- Viva questions based on this A\* practical?

4o

**You said:**

current = min(open\_list, key=lambda x: g[x] + heuristic[x]) explain each word step by step

**ChatGPT said:**

Sure! Let's break down this important A\* line **word by word**:

python

CopyEdit

current = min(open\_list, key=lambda x: g[x] + heuristic[x])

---

- ◆ **1. current =**

- We are assigning the result of the expression to a variable called current.
  - current will be the **node with the lowest total cost** ( $f(x)$ ) from the open\_list.
- 

- ◆ **2. min(open\_list, ...)**

- min() is a built-in Python function.
- It finds the **minimum element** from a sequence based on some logic.
- Here, we are finding the node in open\_list with the **lowest cost**.

Example:

python

CopyEdit

`min([10, 20, 5]) → 5`

But here, we use key=... to customize *how* the minimum is calculated.

---

- ◆ **3. open\_list**

- A **set of nodes** that are candidates to be explored.
- This contains all the "**frontier**" **nodes**, i.e., not yet visited but discovered.

Example:

python

CopyEdit

`open_list = {'A', 'B', 'C'}`

---

- ◆ **4. key=...**

- key is an argument in min() that tells Python how to compare the items.
  - Instead of comparing values directly, it compares key(x) for each x in open\_list.
- 

- ◆ **5. lambda x: ...**

- A lambda function is a short way to define an anonymous function.
  - Here, it means: for each node  $x$  in `open_list`, compute its total cost.
- 

#### ◆ 6. $g[x] + \text{heuristic}[x]$

- This is the  $A^*$  formula:  $f(x) = g(x) + h(x)$ 
    - $g[x]$ : Cost from start to node  $x$  (already calculated).
    - $\text{heuristic}[x]$ : Estimated cost from  $x$  to goal.
  - So this gives the total estimated cost of reaching the goal *via* node  $x$ .
- 

#### ✓ Summary

This whole line:

python

CopyEdit

`current = min(open_list, key=lambda x: g[x] + heuristic[x])`

Means:

**Choose the node from the open list that has the lowest total cost (actual cost so far + estimated cost to goal), and assign it to current.**

---

Would you like me to simulate this with an example `open_list`, `g`, and `heuristic`?

4o

**You said:**

explain code in marathi

**ChatGPT said:**

हो! खाली दिलेला  $A^*$  अल्गोरिदमचा कोड मराठीत स्पष्ट करून सांगतो आहे – ओळी ओळीने.

---

#### ✍ कोड:

python

CopyEdit

`current = min(open_list, key=lambda x: g[x] + heuristic[x])`

---

 मराठीत स्पष्टीकरण:

◆ **current =**

- current हा एक चल (**variable**) आहे, ज्यामध्ये आपण सध्या process करायचा सर्वोत्तम (**cost** कमी असलेला) node ठेवणार आहोत.
- 

◆ **min(open\_list, ...)**

- min() हा Python चा **in-built function** आहे.
- open\_list मध्ये असलेल्या node ऐकी सर्वात कमी खर्चाचा (**कमीत कमी  $f(x)$** ) node शोधतो.

उदाहरण:

python

CopyEdit

```
open_list = {'A', 'B', 'C'}
```

---

◆ **key=...**

- min() मध्ये key= हे parameter वापरून आपण सांगतो की कशाच्या आधारावर minimum node निवडायचा.

---

◆ **lambda x: g[x] + heuristic[x]**

- ही एक **lambda function** आहे – म्हणजे एक छोटसं function जे एका ओळीत लिहितात.
  - x म्हणजे open\_list मधील प्रत्येक node.
  - g[x]: start पासून node x पर्यंतचा **actual** खर्च (real cost).
  - heuristic[x]: x पासून goal पर्यंतचा अनुमानित खर्च (estimate).
  - दोघांचं बेरीज  $f(x)$  म्हणजे एकूण खर्च (A\* formula).
- 

 संपूर्ण ओळ म्हणजे काय?

python

CopyEdit

current = min(open\_list, key=lambda x: g[x] + heuristic[x])

याचा अर्थ:

→ "open\_list मधून असा node निवड जो  $g[x] + \text{heuristic}[x]$  ( $f(x)$ ) कमी आहे, आणि तो current मध्ये ठेव."

---

✓ A\* मध्ये ह्या ओळीचं काम:

ही ओळ decision घेते की पुढे कुठला node expand करायचा – जो सर्वात promising (उपयुक्त) वाटतो.

---

विवरण हवं आहे का कसं  $g$ , heuristic, आणि open\_list बदलत जातात एक उदाहरण वापरून?

40

You said:

yes

ChatGPT said:

चला तर मग, आपण एक लहान उदाहरण घेऊन  $g$ , heuristic, आणि open\_list कसे बदलत जातात हे step-by-step मराठीत समजून घेऊया.

---

✓ Graph (गाठी आणि cost)

python

CopyEdit

graph = {

'A': {'B': 1, 'C': 3},

'B': {'D': 3, 'E': 1},

'C': {'F': 5},

'D': {'G': 2},

'E': {'G': 2},

'F': {'G': 1},

'G': {}

}

### ✓ Heuristic (हुरिस्टिक – अंदाजे अंतर goal पर्यंत)

python

CopyEdit

```
heuristic = {
```

```
'A': 6, 'B': 4, 'C': 4,  
'D': 2, 'E': 2, 'F': 1, 'G': 0  
}
```

---

⌚ उद्दिष्ट: A पासून G पर्यंतची कमी खर्चाची वाट शोधायची

---

🌟 सुरुवात

▣ प्रारंभिक स्थिती:

python

CopyEdit

```
open_list = {'A'}      # सुरुवातीला A node ओपन आहे  
g = {'A': 0}          # A पर्यंतचा cost = 0  
parents = {'A': None}  # A ला कोणी parent नाही
```

---

⌚ पहिला फेरा (Iteration 1)

◆ ओपन लिस्ट:

ini

CopyEdit

```
open_list = {'A'}
```

◆ min():

python

CopyEdit

```
current = min(open_list, key=lambda x: g[x] + heuristic[x])
= min({'A'}, key=lambda x: 0 + 6)
= 'A'
```

- ➡ आता current = 'A'
  - ➡ A वरून पुढे B आणि C ला जाता येतं
- 

◆ Update गाठी:

B:

python

CopyEdit

```
g['B'] = g['A'] + cost = 0 + 1 = 1
parents['B'] = 'A'
open_list.add('B')
```

C:

python

CopyEdit

```
g['C'] = 0 + 3 = 3
parents['C'] = 'A'
open_list.add('C')
```

---

◆ नवीन स्थिती:

ini

CopyEdit

```
open_list = {'B', 'C'}
g = {'A': 0, 'B': 1, 'C': 3}
parents = {'A': None, 'B': 'A', 'C': 'A'}
```

---

➡ दुसरा फेरा (Iteration 2)

◆ **min():**

python

CopyEdit

B → g = 1, h = 4 → f = 5

C → g = 3, h = 4 → f = 7

```
current = min({'B', 'C'}, key=lambda x: g[x] + heuristic[x])
```

= 'B' # कारण f(B) = 5 < 7

→ current = 'B'

→ B कडून D आणि E कडे जातो

---

◆ **Update:**

**D:**

python

CopyEdit

g['D'] = 1 + 3 = 4

parents['D'] = 'B'

open\_list.add('D')

**E:**

python

CopyEdit

g['E'] = 1 + 1 = 2

parents['E'] = 'B'

open\_list.add('E')

---

◆ **नवीन स्थिती:**

ini

CopyEdit

```
open_list = {'C', 'D', 'E'}  
g = {'A': 0, 'B': 1, 'C': 3, 'D': 4, 'E': 2}
```

---

### 💡 तिसरा फेरा (Iteration 3)

python

CopyEdit

C → g = 3, h = 4 → f = 7

D → g = 4, h = 2 → f = 6

E → g = 2, h = 2 → f = 4

➡️ current = 'E'

➡️ E कडून G ला जाता येतं

python

CopyEdit

g['G'] = 2 + 2 = 4

parents['G'] = 'E'

open\_list.add('G')

---

### 💡 चौथा फेरा

python

CopyEdit

open\_list = {'C', 'D', 'G'}

C → f = 7

D → f = 6

G → f = 4

➡️ current = 'G' # हेच आपलं goal आहे

---

 मार्ग वाट शोधा (Backtrack):

python

CopyEdit

G → E → B → A

Final Path:

python

CopyEdit

['A', 'B', 'E', 'G']

---

 निष्कर्ष

स्टेप Open List    g[] अपडेट    f(x)=g+h    चालू नोड

1    {'A'}            A=0            A=6            A

2    {'B', 'C'}        B=1, C=3    B=5, C=7    B

3    {'C', 'D', 'E'}    D=4, E=2    E=4            E

4    {'C', 'D', 'G'}    G=4            G=4            G (goal)

---

हे उदाहरण तुझीं सगळं A\* चं सगळं logic स्पष्ट करतं.

हवं असल्यास मी याचा ग्राफ डायग्राम देखील बनवून देऊ शकतो — हवा आहे का?

## Initial Setup

python

CopyEdit

```
open_list = {A}
```

```
g = {A: 0}
```

```
parents = {A: None}
```

Heuristic values:

python

CopyEdit

```
heuristic = {
```

```
'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 1, 'G': 0
```

```
}
```

---

## ✓ Iteration 1

python

CopyEdit

```
current = min(open_list, key=lambda x: g[x] + heuristic[x])
```

👉 current = A (g=0, h=6, f=6)

Now process neighbors of A: B (cost 1), C (cost 3)

### Update:

python

CopyEdit

```
g = {A: 0, B: 1, C: 3}
```

```
parents = {A: None, B: A, C: A}
```

open\_list = {B, C}

---

## ✓ Iteration 2

Check f-values:

- B → 1 + 4 = 5
- C → 3 + 4 = 7

👉 current = B

Neighbors: D (3), E (1)

**Update:**

python

CopyEdit

$g = \{A: 0, B: 1, C: 3, D: 4, E: 2\}$

parents = {A: None, B: A, C: A, D: B, E: B}

open\_list = {C, D, E}

---

## ✓ Iteration 3

Check f-values:

- C → 3 + 4 = 7
- D → 4 + 2 = 6
- E → 2 + 2 = 4

👉 current = E

Neighbors: G (2) →  $g[G] = g[E] + 2 = 4$

**Update:**

python

CopyEdit

$g = \{A: 0, B: 1, C: 3, D: 4, E: 2, G: 4\}$

parents = {A: None, B: A, C: A, D: B, E: B, G: E}

open\_list = {C, D, G}

---

## Iteration 4

Check f-values:

- C → 3 + 4 = 7
- D → 4 + 2 = 6
- G → 4 + 0 = 4

 current = G  → Goal reached!

Now reconstruct the path using parents:

python

CopyEdit

G ← E ← B ← A

## Final Path:

python

CopyEdit

['A', 'B', 'E', 'G']

---

## Final Output:

pgsql

CopyEdit

Path: ['A', 'B', 'E', 'G']

---

## Summary Table:

**Node g(n) h(n) f(n) = g+h Parent**

A	0	6	6	None
B	1	4	5	A
E	2	2	4	B
G	4	0	4	E

## ◆ 1. Selection Sort

### ✓ Definition:

Selection Sort is a **simple comparison-based sorting algorithm** that repeatedly selects the **smallest (or largest)** element from the unsorted part and moves it to the beginning.

---

### 🧠 Working:

1. Divide the array into **sorted** and **unsorted** parts.
  2. Repeatedly:
    - o Find the **minimum element** in the unsorted part.
    - o Swap it with the first unsorted element.
  3. Repeat until the array is fully sorted.
- 

### ⌚ Example:

Let's sort: [29, 10, 14, 37, 13]

- Pass 1: smallest = 10 → swap with 29 → [10, 29, 14, 37, 13]
  - Pass 2: smallest = 13 → swap with 29 → [10, 13, 14, 37, 29]
  - Pass 3: already in order
  - Pass 4: swap 29 and 37 → [10, 13, 14, 29, 37]
- 

### 💻 Time Complexity:

- Worst Case:  $O(n^2)$
- Best Case:  $O(n^2)$  (no optimization for sorted data)
- Space:  $O(1)$  (in-place)

---

## ◆ 2. Prim's Algorithm

### ✓ Definition:

Prim's Algorithm is a **greedy algorithm** used to find a **Minimum Spanning Tree (MST)** for a connected, weighted, undirected graph.

An MST is a subset of edges that connects **all vertices** with the **minimum total weight, without any cycles**.

---

### 🧠 Working:

1. Start from any node (usually node 0).
  2. Add the **smallest edge** that connects a **visited node** to an **unvisited node**.
  3. Repeat until all nodes are connected.
- 

### 🌐 Real-Life Example:

Used in:

- Network design (cables, water pipes)
  - Road networks
  - Circuit layout design
- 

### ⌚ Steps (Visual Idea):

Graph:

less

CopyEdit

A -1- B

| |

4 3

| |

C -2- D

Start from A:

- A–B (1)
  - B–D (3)
  - D–C (2)  
→ MST edges: AB, BD, DC
- 

### Time Complexity:

- Using adjacency matrix:  $O(V^2)$
- Using Min Heap + Adjacency List:  $O(E \log V)$

## Part 1: Selection Sort

### ◆ Code:

python

CopyEdit

```
def selection_sort(arr):  
    for i in range(len(arr)):  
        min_index = i  
        for j in range(i + 1, len(arr)):  
            if arr[j] < arr[min_index]:  
                min_index = j  
        arr[i], arr[min_index] = arr[min_index], arr[i]  
    return arr
```

```
a1 = [20, 10, 5, 7, 9, 13]
```

```
print(selection_sort(a1))
```

### Step-by-Step Explanation:

- **Selection Sort** is a simple comparison-based sorting algorithm.
- It works by **repeatedly selecting the smallest (or largest) element from the unsorted part and moving it to the beginning.**

### How it works:

1. Start from index  $i = 0$ .
2. Assume the element at  $i$  is the smallest ( $\text{min\_index} = i$ ).
3. Loop  $j$  from  $i+1$  to end and compare values.
4. If a smaller value is found, update  $\text{min\_index}$ .
5. After inner loop ends, swap  $\text{arr}[i]$  with  $\text{arr}[\text{min\_index}]$ .
6. Repeat this for every position in the array.

## Part 2: Prim's Algorithm (for Minimum Spanning Tree)

### ◆ Code:

python

CopyEdit

```
import heapq
```

```
def prim(graph, start):
    mst = [] # Stores edges of the Minimum Spanning Tree
    visited = set([start]) # Track visited nodes
    edges = [(cost, start, to) for to, cost in graph.get(start, {}).items()]
    heapq.heapify(edges) # Create a min-heap for edges

    while edges:
        cost, frm, to = heapq.heappop(edges) # Get edge with least cost
        if to not in visited:
            visited.add(to)
            mst.append((frm, to, cost)) # Add edge to MST
            for to_next, cost2 in graph.get(to, {}).items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost2, to, to_next)) # Add new edges
    return mst
```

## 💡 What is Prim's Algorithm?

Prim's algorithm is used to find the **Minimum Spanning Tree (MST)** for a **connected, weighted, undirected graph**. It ensures all nodes are connected with the **minimum total edge cost** and **no cycles**.

---

## 👉 Let's break it down step-by-step using this graph:

python

CopyEdit

```
graph = {
```

```
    'A': {'B': 2, 'C': 3},  
    'B': {'A': 2, 'C': 1, 'D': 1},  
    'C': {'A': 3, 'B': 1, 'D': 4},  
    'D': {'B': 1, 'C': 4}
```

```
}
```

**Starting at node 'A':**

- Add all edges from A: A-B (2), A-C (3) to the priority queue.
  - Pick the edge with the lowest cost → A-B (2), add it to MST.
  - Now from B, add edges B-C (1), B-D (1).
  - Continue selecting the smallest edge and expand the tree.
- 

## ✓ Output:

python

CopyEdit

```
[('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]
```

This means:

- A is connected to B (cost 2)
- B is connected to C (cost 1)
- B is connected to D (cost 1)

**Total Minimum Cost = 4**

---

 **Summary:**

**Edge Cost**

A - B 2

B - C 1

B - D 1

**Total 4**

---

Would you like me to create a **visual diagram of the graph and MST** for better understanding?

4o

**You said:**

explain prism algo code line by line

**ChatGPT said:**

Sure Omkar! Let's go through the **Prim's Algorithm code line by line** in Python with clear explanation.

---

 **Code:**

python

CopyEdit

import heapq

- ◆ This imports Python's **heapq module** which provides a **priority queue** (min-heap).

We'll use it to always pick the edge with the smallest cost.

---

python

CopyEdit

def prim(graph, start):

- ◆ This defines a function called `prim()` which takes:

- graph: the weighted undirected graph (as a dictionary of dictionaries)
  - start: the starting node (e.g., 'A')
- 

python

CopyEdit

```
mst = []
```

- ◆ mst (Minimum Spanning Tree) will store the **final selected edges** in the MST.
- 

python

CopyEdit

```
visited = set([start])
```

- ◆ visited keeps track of **nodes we've already added** to the MST.
    - We start by visiting the start node.
- 

python

CopyEdit

```
edges = [(cost, start, to) for to, cost in graph.get(start, {}).items()]
```

- ◆ This creates a **list of edges** coming from the start node.
    - Each edge is a tuple: (cost, from, to)
    - graph.get(start, {}) safely gets the neighbors of start.
- 

python

CopyEdit

```
heapq.heapify(edges)
```

- ◆ Converts the list of edges into a **min-heap**, so we can **always extract the smallest edge first**.
- 

python

CopyEdit

while edges:

- ◆ While there are still edges in the heap, we continue building the MST.
- 

python

CopyEdit

```
cost, frm, to = heapq.heappop(edges)
```

- ◆ Pops the **smallest edge** from the heap (edge with lowest cost).
    - frm → from node
    - to → to node
    - cost → weight of the edge
- 

python

CopyEdit

```
if to not in visited:
```

- ◆ We only consider the edge if the to node has **not already been visited**.
    - This avoids forming **cycles**.
- 

python

CopyEdit

```
visited.add(to)
```

- ◆ Add the new node to the visited set, meaning we're including it in the MST now.
- 

python

CopyEdit

```
mst.append((frm, to, cost))
```

- ◆ Add the edge (from, to, cost) to the MST list.
- 

python

CopyEdit

```
    for to_next, cost2 in graph.get(to, {}).items():
```

- ◆ Now we go through **all neighbors** of the newly added node (to)...
- 

python

CopyEdit

```
    if to_next not in visited:
```

- ◆ ...but only consider neighbors we haven't already visited.
- 

python

CopyEdit

```
        heapq.heappush(edges, (cost2, to, to_next))
```

- ◆ Push these new candidate edges to the heap.
    - Again as (cost, from, to) format.
- 

python

CopyEdit

```
    return mst
```

- ◆ After the loop ends, return the list of MST edges.
- 

## 📌 Final Output Example:

python

CopyEdit

```
graph = {
```

```
    'A': {'B': 2, 'C': 3},
```

```
    'B': {'A': 2, 'C': 1, 'D': 1},
```

```
    'C': {'A': 3, 'B': 1, 'D': 4},
```

```
    'D': {'B': 1, 'C': 4}
```

```
}
```

```
print(prim(graph, 'A'))
```

 **Output:**

css

CopyEdit

```
[('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]
```

**Dry Run (Start from node 'A'):**

**Step 0: Initial Setup**

ini

CopyEdit

```
start = 'A'
```

```
visited = {'A'}
```

```
edges = [(2, 'A', 'B'), (3, 'A', 'C)] → heapified (min-heap)
```

```
mst = []
```

---

**Step 1: Pop edge with min cost: (2, 'A', 'B')**

csharp

CopyEdit

B not visited → add to visited → visited = {'A', 'B'}

Add edge to MST: ('A', 'B', 2)

Add neighbors of B:

B → A (already visited) → skip

B → C = 1 → add to heap

B → D = 1 → add to heap

Heap = [(1, 'B', 'C'), (3, 'A', 'C'), (1, 'B', 'D')]

MST = [('A', 'B', 2)]

---

**Step 2: Pop edge: (1, 'B', 'C')**

csharp

CopyEdit

C not visited → visited = {'A', 'B', 'C'}

Add edge to MST: ('B', 'C', 1)

Neighbors of C:

C → A (visited) → skip

C → B (visited) → skip

C → D = 4 → add to heap

Heap = [(1, 'B', 'D'), (3, 'A', 'C'), (4, 'C', 'D')]

MST = [('A', 'B', 2), ('B', 'C', 1)]

---

**Step 3: Pop edge: (1, 'B', 'D')**

java

CopyEdit

D not visited → visited = {'A', 'B', 'C', 'D'}

Add edge to MST: ('B', 'D', 1)

Neighbors of D:

D → B (visited) → skip

D → C (visited) → skip

Heap = [(3, 'A', 'C'), (4, 'C', 'D')]

MST = [('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]

---

**Step 4: Heap has 2 more edges, but both go to visited nodes:**

mathematica

CopyEdit

(3, 'A', 'C') → C already visited

(4, 'C', 'D') → D already visited

Skip them

---

### ❖ Loop Ends:

- All nodes are visited: {'A', 'B', 'C', 'D'}
- Minimum Spanning Tree (MST):

python

CopyEdit

[('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]

## Q1. What is Prim's Algorithm?

### Answer:

Prim's Algorithm is a greedy algorithm that finds the **Minimum Spanning Tree (MST)** for a connected weighted graph. It starts from any one vertex and adds the **lowest weight edge** that connects a new vertex to the tree until all vertices are included.

---

## Q2. What is a Minimum Spanning Tree (MST)?

### Answer:

An MST is a **subset of edges** in a weighted, connected, undirected graph that:

- Connects all the vertices.
  - Has **no cycles**.
  - Has the **minimum possible total edge weight**.
- 

## Q3. Difference between Prim's and Kruskal's Algorithm?

### Answer:

<b>Feature</b>	<b>Prim's Algorithm</b>	<b>Kruskal's Algorithm</b>
Approach	Greedy – builds MST starting from one node	Greedy – picks smallest edge globally
Data Structure	Min-Heap/Priority Queue	Disjoint Set (Union-Find)
Graph Type	Dense graphs	Sparse graphs
Cycle check	Implicit via visited set	Explicit via disjoint sets

---

#### Q4. Explain dry run of your code with given graph.

##### Answer:

(Use the same explanation and example as given above – walk through the MST step by step.)

##### Graph:

```
python
CopyEdit
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'A': 2, 'C': 1, 'D': 1},
    'C': {'A': 3, 'B': 1, 'D': 4},
    'D': {'B': 1, 'C': 4}
}
```

##### Output:

```
python
CopyEdit
[('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]
```

This is the MST with total cost =  $2 + 1 + 1 = 4$ .

---

#### Q5. Which data structures are used in your Prim's code?

##### Answer:

- Set for visited nodes
  - List of tuples for edges
  - heapq (priority queue) to always pick the smallest edge
  - Graph is represented using **adjacency list** with weights
- 

### Q6. Can Prim's Algorithm handle disconnected graphs?

**Answer:**

No. Prim's algorithm only works on **connected graphs**. If the graph is disconnected, it will not be able to reach all nodes, and the MST won't include all vertices.

---

### Q7. What is the time complexity of Prim's Algorithm?

**Answer:**

- Using priority queue (heap): **O(E log V)**
  - Where:
    - E = number of edges
    - V = number of vertices
- 

### Q8. What happens if two edges have the same weight?

**Answer:**

Prim's algorithm will choose **any one of the minimum edges**, depending on how they appear in the data structure. Both are valid as long as the MST condition is satisfied.

---

### Q9. Can Prim's algorithm be applied to directed graphs?

**Answer:**

Prim's Algorithm is **designed for undirected graphs**. In a directed graph, the concept of a **spanning tree** doesn't directly apply because edges have direction.

---

### Q10. What happens if negative weights are present?

**Answer:**

Prim's algorithm **can work with negative weights**, as long as the graph remains

**connected** and **undirected**. The algorithm always picks the **minimum weight edge**, regardless of sign.

---

Do you want me to create a **PDF cheat sheet** of these questions and answers for your exam?

40

**You said:**

also give in marathi

**ChatGPT said:**

नक्कीच, ओंकार! खाली **Prim's Algorithm practical** संदर्भातील **external viva** प्रश्न आणि उत्तरे मराठीत दिले आहेत — यामध्ये थोडक्यात स्पष्टीकरण आणि उदाहरण दिले आहे जे SPPU practical मध्ये उपयुक्त ठेल.

---

◆ **प्रश्न 1: Prim's Algorithm म्हणजे काय?**

उत्तर:

Prim's Algorithm एक greedy algorithm आहे जो **Minimum Spanning Tree (MST)** तयार करतो. हा algorithm एका node पासून सुरुवात करतो आणि प्रत्येक टप्प्यावर सर्वात कमी **weight** असलेला **edge** निवडतो जो एका नवीन node ला जोडतो.

---

◆ **प्रश्न 2: Minimum Spanning Tree (MST) म्हणजे काय?**

उत्तर:

MST म्हणजे एका **connected, weighted graph** मधील अशा edges चा subset:

- जो सर्व nodes जोडतो.
  - कोणताही cycle नसतो.
  - आणि एकूण weight **minimum** असतो.
- 

◆ **प्रश्न 3: Prim's आणि Kruskal's Algorithm मधील फरक सांगा.**

बाब	Prim's Algorithm	Kruskal's Algorithm
-----	------------------	---------------------

सुरुवात	एखाद्या node पासून	सर्वात कमी weight असलेल्या edge पासून
---------	--------------------	---------------------------------------

बाब	Prim's Algorithm	Kruskal's Algorithm
डेटा स्ट्रक्चर	Priority Queue (heapq)	Disjoint Set (Union-Find)
Cycle चा तपास	visited set वापरतो	Union-find वापरून cycle check
वापर	Dense graphs साठी चांगला	Sparse graphs साठी चांगला

---

◆ प्रश्न 4: तुमच्या code ची dry run करून दाखवा.

उत्तर:

(तुमचं code आणि output दाखवा आणि पायरी-पायरीने explain करा.)

उदाहरण:

```
python
CopyEdit
graph = {
    'A': {'B': 2, 'C': 3},
    'B': {'A': 2, 'C': 1, 'D': 1},
    'C': {'A': 3, 'B': 1, 'D': 4},
    'D': {'B': 1, 'C': 4}
}
```

**Output:**

```
python
CopyEdit
[('A', 'B', 2), ('B', 'C', 1), ('B', 'D', 1)]
MST मध्ये cost = 2 + 1 + 1 = 4.
```

---

◆ प्रश्न 5: तुमच्या code मध्ये कोणते data structures वापरले आहेत?

उत्तर:

- set → visited nodes साठी
- list of tuples → edges साठी

- heapq → priority queue (minimum edge निवडण्यासाठी)
  - dictionary of dictionaries → weighted adjacency list म्हणून graph
- 

◆ प्रश्न 6: Prim's algorithm disconnected graph साठी काम करतो का?

उत्तर:

नाही, Prim's Algorithm फक्त **connected graphs** साठीच काम करतो. जर graph disconnected असेल तर MST पूर्ण होणार नाही.

---

◆ प्रश्न 7: Prim's Algorithm ची time complexity काय आहे?

उत्तर:

- Priority Queue वापरल्यास: **O(E log V)**
    - E = edges ची संख्या
    - V = nodes ची संख्या
- 

◆ प्रश्न 8: जर दोन edges चा weight सारखाच असेल तर algorithm काय करतो?

उत्तर:

Algorithm त्या दोनपैकी कोणताही एक edge निवडू शकतो, जो internal order नुसार आधी भेटतो. दोन्ही valid असतात जोपर्यंत MST ची अट पूर्ण होते.

---

◆ प्रश्न 9: Prim's Algorithm directed graph साठी वापरता येतो का?

उत्तर:

Prim's Algorithm फक्त **undirected graph** साठी योग्य आहे. Directed graph मध्ये spanning tree ची संकल्पना वेगळी आहे.

---

◆ प्रश्न 10: जर edge ला negative weight असेल तर algorithm काम करेल का?

उत्तर:

हो, Prim's Algorithm **negative weights** असलेल्या graph वर देखील काम करू शकतो — फक्त graph **connected** असावा आणि **undirected** असावा.

# The N-Queens algorithm

is a classic **backtracking** algorithm used to solve the **N-Queens problem**:

 **Goal:** Place N queens on an  $N \times N$  chessboard such that **no two queens** threaten each other — i.e., no two queens are in the same **row**, **column**, or **diagonal**.

---

## Why Backtracking?

Backtracking is used because:

- It's a **trial-and-error** method.
  - You try to place a queen, and if it leads to a solution, great.
  - If not, **go back** (backtrack) and try a different position.
- 

## Rules:

A queen can attack:

- Vertically (same column)
  - Horizontally (same row)
  - Diagonally (top-left to bottom-right and top-right to bottom-left)
- 

## Basic Idea:

1. Start from the **first row**.
2. Try placing a queen in **each column** of the row.
3. For each attempt, **check if it's safe**:
  - No queen in same column.
  - No queen in same diagonals.
4. If safe, **place the queen** and **move to next row**.
5. If all queens are placed, **you found a solution**.
6. If not, **backtrack** and try next column in previous row.

```

print("Enter the number of queens: ")
N = int(input())

# Chessboard
board = [[0] * N for _ in range(N)]

def is_safe(row, col):
    """Check if it's safe to place a queen at (row, col)"""
    # Check vertical and diagonal conflicts
    for i in range(row):
        if board[i][col] == 1: # Check column
            return False
        if col - (row - i) >= 0 and board[i][col - (row - i)] == 1: # Upper-left diagonal
            return False
        if col + (row - i) < N and board[i][col + (row - i)] == 1: # Upper-right
            diagonal
            return False
    return True

def solve_N_queens(row):
    """Solve N-Queens using backtracking"""
    if row >= N:
        return True

    for col in range(N):
        if is_safe(row, col):
            board[row][col] = 1
            if solve_N_queens(row + 1):
                return True
            board[row][col] = 0

```

```
    board[row][col] = 0 # Backtracking

return False

# Solve the problem starting from row 0
solve_N_queens(0)
```

```
# Print the chessboard
for i in board:
    print(i)
```

**ChatGPT said:**

Sure! Here's a **line-by-line explanation** of your N-Queens Python code in a very simple way:

---

### **Code Explanation:**

python

CopyEdit

```
print("Enter the number of queens: ")
```

```
N = int(input())
```

- Asks the user to enter the number of queens (N).
  - The number of queens also defines the **size of the chessboard** ( $N \times N$ ).
- 

python

CopyEdit

```
# Chessboard
```

```
board = [[0] * N for _ in range(N)]
```

- Creates a 2D list (board) with all values as 0.
- A 0 means **no queen**, and 1 means **queen placed**.
- For example, if  $N = 4$ , the board is:

```
python
CopyEdit
[[0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0],
 [0, 0, 0, 0]]
```

---

```
python
CopyEdit
def is_safe(row, col):
    • This function checks if placing a queen at position (row, col) is safe or not.

python
CopyEdit
for i in range(row):
    • Loop over all previous rows only, because queens are placed row by row.

python
CopyEdit
if board[i][col] == 1:
    return False
    • ✗ Check same column in previous rows. If there's already a queen, it's not safe.

python
CopyEdit
if col - (row - i) >= 0 and board[i][col - (row - i)] == 1:
    return False
    • ✗ Check the upper-left diagonal. If there's a queen, not safe.

python
CopyEdit
if col + (row - i) < N and board[i][col + (row - i)] == 1:
```

```
        return False
```

- ✗ Check the **upper-right diagonal**. Same logic.

python

CopyEdit

```
    return True
```

- ✓ If all checks passed, the position is safe.
- 

python

CopyEdit

```
def solve_N_queens(row):
```

- Recursive function to **place queens row by row**.

python

CopyEdit

```
    if row >= N:
```

```
        return True
```

- Base case: If all rows are filled, return True (solution found).

python

CopyEdit

```
    for col in range(N):
```

- Try placing the queen in every column of the current row.

python

CopyEdit

```
        if is_safe(row, col):
```

```
            board[row][col] = 1
```

- If it's safe to place a queen at (row, col), **place it** (set to 1).

python

CopyEdit

```
        if solve_N_queens(row + 1):
```

```
            return True
```

- Move to the next row recursively. If successful, keep going.

python

CopyEdit

```
    board[row][col] = 0 # Backtracking
```

- If the next step doesn't work, **remove the queen** and try the next column (backtrack).

python

CopyEdit

```
    return False
```

- If no position works in this row, return False to backtrack.
- 

python

CopyEdit

```
# Solve the problem starting from row 0
```

```
solve_N_queens(0)
```

- Start solving from the **first row** (row 0).
- 

python

CopyEdit

```
# Print the chessboard
```

```
for i in board:
```

```
    print(i)
```

- After solving, print the final board with queens (1s) placed safely.
- 

#### ✓ Example Output for N = 4:

text

CopyEdit

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

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

[1, 0, 0, 0]

[0, 0, 1, 0]

Each row has one 1, and no two 1s are in the same row, column, or diagonal.

---

Would you like a visual explanation or diagram for better understanding?

Sure! Let's **dry run** the N-Queens code along with the code logic. We'll walk through it step-by-step **for  $N = 4$** , which is the smallest  $N$  where a solution exists.

---

### **Code with Step-by-Step Dry Run ( $N = 4$ )**

python

CopyEdit

```
print("Enter the number of queens: ")
```

```
N = int(input()) # Suppose user inputs 4
```

So,  $N = 4$ , we need to place 4 queens on a  $4 \times 4$  chessboard.

python

CopyEdit

```
# Chessboard
```

```
board = [[0] * N for _ in range(N)]
```

Initial board:

CopyEdit

```
0 0 0 0
```

```
0 0 0 0
```

```
0 0 0 0
```

```
0 0 0 0
```

---

### **Step 1: Start solving from row 0**

python

CopyEdit

```
solve_N_queens(0)
```

Loop through each column in row 0:

**Try  $\text{col} = 0$ :**

- $\text{is\_safe}(0, 0) \rightarrow \text{True}$

- Place queen at (0, 0)

CopyEdit

1 0 0 0

0 0 0 0

0 0 0 0

0 0 0 0

- Call solve\_N\_queens(1)
- 

## Step 2: Row 1

Try col = 0:

- Conflict in column →

Try col = 1:

- Conflict in upper-left diagonal →

Try col = 2:

- Safe → Place queen at (1, 2)

CopyEdit

1 0 0 0

0 0 1 0

0 0 0 0

0 0 0 0

- Call solve\_N\_queens(2)
- 

## Step 3: Row 2

Try col = 0:

- Conflict in column →

Try col = 1:

- Conflict in upper-right diagonal →

Try col = 2:

- Conflict in column →

Try col = 3:

- Safe → Place queen at (2, 3)

CopyEdit

1 0 0 0

0 0 1 0

0 0 0 1

0 0 0 0

- Call solve\_N\_queens(3)
- 

#### Step 4: Row 3

Try all columns — all positions are unsafe!

- Backtrack:
  - Remove queen from (2, 3)
  - Try next column in row 2

Backtrack again:

- Remove queen from (1, 2)
  - Try next columns in row 1
- 

**Eventually, after trying various options and backtracking:**

A correct configuration is found:

CopyEdit

0 1 0 0

0 0 0 1

1 0 0 0

0 0 1 0

Corresponding output:

python

CopyEdit

[0, 1, 0, 0]

[0, 0, 0, 1]

[1, 0, 0, 0]

[0, 0, 1, 0]

---

### Summary of the Algorithm:

1. Start from row 0.
  2. Try placing a queen in each column.
  3. Use `is_safe()` to ensure the queen doesn't attack any others.
  4. If safe, place the queen and move to the next row.
  5. If stuck, backtrack (remove the queen and try next column).
  6. Repeat until all queens are placed.
- 

Would you like me to generate a **visual diagram** of the steps or board state animations?

4o

## onceptual Questions

### 1. What is the N-Queens problem?

#### Answer:

The N-Queens problem is a classic combinatorial problem where we must place  $N$  queens on an  $N \times N$  chessboard such that no two queens threaten each other — meaning, no two queens share the same row, column, or diagonal.

---

## 2. Which algorithm is used in your code?

**Answer:**

I used **backtracking**, which is a type of recursive algorithm that tries all possible arrangements and "backtracks" when it reaches an invalid state.

---

## 3. What is the purpose of the `is_safe()` function?

**Answer:**

The `is_safe()` function checks if placing a queen at a specific row and column is safe — meaning no queen is already placed in the same column, or in the upper-left or upper-right diagonals.

---

## 4. Why do we only check rows above in `is_safe()`?

**Answer:**

We solve the board row by row from top to bottom. So when placing a queen in the current row, we only need to check for conflicts with queens already placed in previous rows (i.e., rows above).

---

## 5. What is backtracking?

**Answer:**

Backtracking is a trial-and-error method where we build a solution step-by-step and undo ("backtrack") a step if it leads to an invalid state.

---

## 6. What is the base case of the recursion in your code?

**Answer:**

The base case is: if `row >= N`, which means all queens are successfully placed — so we return `True`.

---

## 7. What is the time complexity of the N-Queens problem?

**Answer:**

In the worst case, the time complexity is **O(N!)**, because we try to place a queen in every column of every row, and recursively explore all combinations.

---

## **8. Can there be more than one solution for a given N?**

**Answer:**

Yes. For example,  $N = 4$  has 2 valid solutions. The code can be modified to print all possible solutions instead of just one.

---

## **9. What are the constraints or base cases?**

**Answer:**

- $N = 1 \rightarrow$  Always has 1 solution.
  - $N = 2$  or  $N = 3 \rightarrow$  No solution exists.
- 

## **10. Why is the board initialized with 0s?**

**Answer:**

Zero represents an empty square. When a queen is placed, we set it to 1. This helps us track the board state.

---

## **Coding/Implementation Questions**

### **11. What does $\text{board}[\text{row}][\text{col}] = 1$ mean?**

**Answer:**

It means we are placing a queen at position (row, col) on the chessboard.

---

### **12. What happens in backtracking?**

**Answer:**

If a recursive call fails (returns False), we remove the queen by setting  $\text{board}[\text{row}][\text{col}] = 0$  and try the next column in that row.

---

### **13. How is diagonal conflict detected?**

**Answer:**

By checking:

- Upper-left:  $\text{board}[i][\text{col} - (\text{row} - i)]$
  - Upper-right:  $\text{board}[i][\text{col} + (\text{row} - i)]$   
Only when the index is within bounds.
-

#### **14. How would you modify the code to print all solutions?**

**Answer:**

Store each valid board configuration in a list when  $\text{row} == N$ , instead of returning immediately after finding the first one.

---

#### **15. What is the output of your code for $N = 4$ ?**

**Answer:**

csharp

CopyEdit

[0, 1, 0, 0]

[0, 0, 0, 1]

[1, 0, 0, 0]

[0, 0, 1, 0]

This is one valid configuration of 4 queens.

# What is a Chatbot?

A **chatbot** is a software application designed to simulate human conversation, either via text or voice, allowing users to interact with digital systems as if they were speaking to a real person.

---

## Core Concept of a Chatbot

At its core, a chatbot works like this:

1. **User Input:** The user types or speaks something.
  2. **Processing:** The chatbot processes the input using:
    - o **Rule-based logic** (if input is X, respond with Y)
    - o **AI/NLP** (Natural Language Processing) to understand meaning
  3. **Response Generation:** It creates a reply based on:
    - o Predefined scripts (for rule-based bots)
    - o Machine learning models (for AI bots)
  4. **Reply:** The bot sends the response back to the user.
- 

## Types of Chatbots

Type	Description
Rule-based	Uses predefined rules and decision trees. Simple and fast, but limited.
AI-based (Smart)	Uses NLP and machine learning to understand and generate human-like replies.

---

## Technologies Used

- **NLP Libraries:** NLTK, spaCy, transformers
- **Frameworks:** Rasa, Dialogflow, Microsoft Bot Framework
- **Languages:** Python, JavaScript, Java

- **APIs:** OpenAI (ChatGPT), Google Dialogflow, IBM Watson
- 

### Example of How a Simple Chatbot Works:

**User:** "What's the weather today?"

**Bot:** → Extracts keywords → Recognizes it's a weather query → Calls a weather API  
→ Replies with "It's sunny and 30°C."

```
def chatbot():
```

```
    print("Chatbot: Hi, I'm a chatbot! How can I help you today?")
```

```
while True:
```

```
    user_input = input("You: ").lower()
```

```
    if user_input == "bye":
```

```
        print("Chatbot: Bye! Have a great day.")
```

```
        break
```

```
    elif user_input == "hello":
```

```
        print("Chatbot: Hello there! How are you doing today?")
```

```
    elif user_input == "how are you":
```

```
        print("Chatbot: I'm doing great, thanks for asking. What would you like to see?")
```

```
    elif user_input == "phone":
```

```
        print("Chatbot: Sure! Which brand are you interested in?")
```

```
    elif user_input == "apple":
```

```
        print("Chatbot: Okay. What is your budget?")
```

```
    elif user_input == "100000":
```

```
        print("Chatbot: Great! Here are some top models under ₹100000: iPhone 14, iPhone 13 Pro, iPhone 15 (base).")
```

```
    elif user_input == "ok":
```

```
        print("Chatbot: Let me know if you want to know more about a specific model.")
```

```
    elif user_input == "thank you":  
        print("Chatbot: You're welcome! See you again.")  
    else:  
        print("Chatbot: I didn't understand that. Can you please rephrase?")
```

# Run the chatbot

```
chatbot()
```

**ChatGPT said:**

Here's a **simple explanation** of your chatbot code **line by line**:

---

### ◆ Code Breakdown

python

CopyEdit

```
def chatbot():
```

- This defines a **function** named chatbot() that contains all the chatbot logic.

python

CopyEdit

```
    print("Chatbot: Hi, I'm a chatbot! How can I help you today?")
```

- Prints the initial greeting from the chatbot.

python

CopyEdit

```
    while True:
```

- Starts an **infinite loop** so the chatbot keeps running and waiting for user input until the user types "bye".

python

CopyEdit

```
        user_input = input("You: ").lower()
```

- Takes input from the user, converts it to **lowercase** to make comparison easier (e.g., "Hello" becomes "hello").

---

## Conditional Responses

The chatbot checks for specific keywords:

python

CopyEdit

```
if user_input == "bye":  
    print("Chatbot: Bye! Have a great day.")  
    break
```

- If the user says "bye", the bot replies and **exits the loop** using break.

python

CopyEdit

```
elif user_input == "hello":  
    print("Chatbot: Hello there! How are you doing today?")
```

- Responds to a simple greeting.

python

CopyEdit

```
elif user_input == "how are you":  
    print("Chatbot: I'm doing great, thanks for asking. What would you like to  
see?")
```

- Handles casual small talk.

python

CopyEdit

```
elif user_input == "phone":  
    print("Chatbot: Sure! Which brand are you interested in?")
```

- User wants to explore phones; bot asks for a brand.

python

CopyEdit

```
elif user_input == "apple":  
    print("Chatbot: Okay. What is your budget?")
```

- If user mentions **Apple**, the bot follows up by asking for a budget.

python

CopyEdit

```
elif user_input == "100000":
```

```
    print("Chatbot: Great! Here are some top models under ₹100000: iPhone 14,  
iPhone 13 Pro, iPhone 15 (base).")
```

- Suggests phone models within the ₹100000 budget.

python

CopyEdit

```
elif user_input == "ok":
```

```
    print("Chatbot: Let me know if you want to know more about a specific  
model.")
```

- Continues the conversation if user says "ok".

python

CopyEdit

```
elif user_input == "thank you":
```

```
    print("Chatbot: You're welcome! See you again.")
```

- Responds politely to thanks.

python

CopyEdit

```
else:
```

```
    print("Chatbot: I didn't understand that. Can you please rephrase?")
```

- Default reply if the input doesn't match any known command.

---

## ✓ Summary

This is a **rule-based chatbot** using simple if-elif-else conditions.

- It's **text-based**
- Uses `input()` to read what the user types
- Uses `print()` to respond

- Only understands predefined commands
- 

Would you like an upgraded version that handles more complex queries using AI or NLP?

4o

**You said:**

give me the list and questions that external aske me about chatbot prac and swer them

**ChatGPT said:**

Here's a helpful list of **common viva questions** an external examiner may ask you for your **Chatbot practical**, along with **simple and clear answers**:

---

◆ **1. What is a chatbot?**

**Answer:**

A chatbot is a software program that can simulate a conversation with users, typically through text or voice, to answer questions or perform tasks.

---

◆ **2. What type of chatbot did you implement?**

**Answer:**

I implemented a **rule-based chatbot** using Python. It replies based on fixed responses using if-else statements.

---

◆ **3. How does your chatbot understand user input?**

**Answer:**

The chatbot takes user input using the `input()` function, converts it to lowercase, and matches it with predefined responses using if-elif conditions.

---

◆ **4. What is the purpose of `.lower()` in your code?**

**Answer:**

`.lower()` converts the user input to lowercase so that the chatbot can match it correctly, regardless of how the user types (e.g., "Hello" or "hello").

---

◆ **5. What happens if the user types something unexpected?**

**Answer:**

If the input doesn't match any condition, the chatbot replies with a default message like: "I didn't understand that. Can you please rephrase?"

---

◆ **6. How does the chatbot stop running?**

**Answer:**

When the user types "bye", the chatbot prints a goodbye message and exits the loop using the break statement.

---

◆ **7. Can this chatbot handle complex queries?**

**Answer:**

No, this is a basic chatbot that works with only fixed responses. To handle complex queries, we'd need Natural Language Processing (NLP) or AI-based methods.

---

◆ **8. What are the limitations of your chatbot?**

**Answer:**

- It can't understand user intent.
  - It only works for exact inputs.
  - It doesn't learn from conversations.
  - No use of AI or machine learning.
- 

◆ **9. What are the applications of chatbots in real life?**

**Answer:**

- Customer support (e.g., bank, online shopping)
  - Virtual assistants (e.g., Alexa, Siri)
  - Booking and scheduling (e.g., hotel, doctor)
  - Education and e-learning help
- 

◆ **10. How can you improve this chatbot?**

**Answer:**

- Add more responses
- Use NLP libraries like NLTK or spaCy
- Integrate with AI tools
- Store conversation history
- Add voice input/output

**ChatGPT said:**

## Sure! Let's go step-by-step and explain how **this Employee Performance Evaluation Expert System** works:

# Implement any one of the following Expert System

# IV. Employee performance evaluation

# List of evaluation criteria

```
criteria = ["productivity", "communication", "teamwork", "punctuality",
"quality_of_work"]
```

# Function to get rank input from the user (from -5 to 5)

```
def get_rank_input(criterion):
```

```
    rank = int(input(f"Enter rank for {criterion} (-5 to 5): "))
```

```
    if -5 <= rank <= 5:
```

```
        return rank
```

```
    else:
```

```
        print("Invalid input. Please enter an integer between -5 and 5.")
```

```
    return get_rank_input(criterion)
```

# Function to evaluate total score

```
def evaluate_employee(ranks):
```

```
    return sum(ranks.values())
```

# Main function

```
def main():
```

```
    ranks = {}
```

```
for criterion in criteria:
```

```
    ranks[criterion] = get_rank_input(criterion)
```

```
score = evaluate_employee(ranks)
```

```
print(f"\nTotal Score: {score}")
```

```
if score > 15:
```

```
    print("Performance: Exceptional")
```

```
elif score > 10:
```

```
    print("Performance: Outstanding")
```

```
elif score > 0:
```

```
    print("Performance: Good")
```

```
elif score == 0:
```

```
    print("Performance: Neutral")
```

```
elif score >= -10:
```

```
    print("Performance: Needs Improvement")
```

```
elif score >= -15:
```

```
    print("Performance: Poor")
```

```
else:
```

```
    print("Performance: Critical - Immediate Action Needed")
```

```
main()
```

---

#### ◆ Purpose of the Program

This Python program acts as an **Expert System** to evaluate an employee's performance based on ranks (scores) given for different criteria.

---

#### ◆ Code Explanation

##### 1. List of Evaluation Criteria

python

CopyEdit

```
criteria = ["productivity", "communication", "teamwork", "punctuality",  
"quality_of_work"]
```

- These are the **5** parameters used to evaluate the employee.
  - The user will rate each of these on a scale from **-5 to 5**.
- 

## 2. Function to Get Input with Validation

python

CopyEdit

```
def get_rank_input(criterion):  
  
    rank = int(input(f'Enter rank for {criterion} (-5 to 5): '))  
  
    if -5 <= rank <= 5:  
  
        return rank  
  
    else:  
  
        print("Invalid input. Please enter an integer between -5 and 5.")  
  
    return get_rank_input(criterion)
```

- Prompts the user to enter a **rank for each criterion**.
  - **It checks if the rank is valid** (between -5 and 5). If not, it asks again using recursion.
- 

## 3. Function to Evaluate the Score

python

CopyEdit

```
def evaluate_employee(ranks):  
  
    return sum(ranks.values())
```

- This takes the dictionary of ranks and calculates the **total performance score** by summing all the values.
- 

## 4. Main Function: Execution Starts Here

python  
CopyEdit  
def main():

```
ranks = {}  
for criterion in criteria:  
    ranks[criterion] = get_rank_input(criterion)
```

- An empty dictionary ranks is created.
  - The loop runs through each evaluation criterion and stores the user's input in the dictionary.
- 

## 5. Total Score and Performance Category

python  
CopyEdit

```
score = evaluate_employee(ranks)  
print(f"\nTotal Score: {score}")
```

- Calls the function to calculate the **total score** from the ranks and prints it.
- 

## 6. Performance Decision (Expert Rules)

python  
CopyEdit

```
if score > 15:  
    print("Performance: Exceptional")  
elif score > 10:  
    print("Performance: Outstanding")  
elif score > 0:  
    print("Performance: Good")  
elif score == 0:  
    print("Performance: Neutral")  
elif score >= -10:
```

```
    print("Performance: Needs Improvement")  
elif score >= -15:  
    print("Performance: Poor")  
else:  
    print("Performance: Critical - Immediate Action Needed")
```

- Based on the total score, it **decides the performance level** using a series of **if-elif expert rules**.
- 

## 7. Run the Program

python

CopyEdit

main()

- Finally, it calls the main() function to run the full program.
- 

### ◆ Sample Output

vbnnet

CopyEdit

Enter rank for productivity (-5 to 5): 5

Enter rank for communication (-5 to 5): 4

Enter rank for teamwork (-5 to 5): 3

Enter rank for punctuality (-5 to 5): 2

Enter rank for quality\_of\_work (-5 to 5): 4

Total Score: 18

Performance: Exceptional

**Here's a helpful list of viva/practical questions with their answers that an external examiner might ask you about your Employee Performance Evaluation Expert System practical:**

---

## Basic Questions

---

### **Q1. What is an Expert System?**

**A1.** An Expert System is a computer program that simulates the decision-making ability of a human expert. It uses rules and logic to evaluate data and provide advice or decisions.

---

### **Q2. What is the purpose of your program?**

**A2.** The program evaluates an employee's performance by taking user input (ratings from -5 to 5) for five different criteria and then classifies their performance into levels like "Good", "Outstanding", "Poor", etc.

---

### **Q3. What are the criteria used in your program?**

**A3.** The five criteria are:

- Productivity
  - Communication
  - Teamwork
  - Punctuality
  - Quality of work
- 

### **Q4. What is the input range for each criterion? Why this range?**

**A4.** The input is an integer from -5 to 5.

This range allows both positive and negative feedback, capturing both strengths and weaknesses.

---

### **Q5. How is the final performance score calculated?**

**A5.** The individual ratings for all five criteria are added together using the sum() function to get the total score.

---

### **Q6. What technique is used to evaluate performance in your code?**

**A6.** The program uses rule-based logic (if-elif statements), which is a common method in expert systems to make decisions based on input data.

---

**Q7. What happens if the user enters an invalid input (like 10 or a letter)?**  
**A7. The code checks if the input is between -5 and 5. If not, it displays an error and recursively asks again. (Note: If non-integer input is entered, the code will throw an error unless exception handling is added.)**

---

 **Intermediate to Advanced Questions**

---

**Q8. How is recursion used in your code?**  
**A8. Recursion is used in the `get_rank_input()` function to re-ask the user for input if the previous one was invalid.**

---

**Q9. Can you improve this program? How?**  
**A9. Yes. Improvements can include:**

- Adding exception handling (try-except) for non-integer inputs
  - Saving results to a file
  - Creating a GUI using Tkinter or web interface
- 

**Q10. Why is this an Expert System?**  
**A10. It's an expert system because it uses predefined rules to make a decision or judgment (i.e., employee performance), just like a human HR expert would.**

---

**Q11. What is the difference between AI and an Expert System?**  
**A11. AI is a broader field aiming for human-like intelligence (e.g., learning, reasoning), whereas an expert system is a specific AI program designed to solve problems using rules and logic, without learning on its own.**

---

**Q12. What will be the output if the total score is -16?**  
**A12. The output will be:**  
**Performance: Critical - Immediate Action Needed**