Below is your **COMPLETE A* Algorithm Practical Answer** in perfect exam format:

✔ Theory (simple and exam-ready)
✔ Full code
✔ Line-by-line explanation (easy to tell mam)
✔ Viva questions & answers (long, full marks)

🔵 AI Practical – A* Search Algorithm

🟣 1. THEORY (Explain this to External Examiner)

✔ What is A* Algorithm?

A* (A-Star) is an **informed search algorithm** used to find the **shortest path** between two points.

It is used in:

- Google Maps
- Robot navigation
- Game AI pathfinding
- Network routing
- GPS systems

🧠 How A* Works?

Every node has 3 costs:

1️⃣ g(n)

Actual cost from the start node to current node.

2️⃣ h(n)

Heuristic – estimated cost from current node to goal.
We use **Manhattan Distance**:

h(n) = |x1 − x2| + |y1 − y2|

3️⃣ f(n)

Total cost:

$$f(n) = g(n) + h(n)$$

✔️ A* Uses Two Lists

OPEN List

Nodes that need to be explored (priority queue).

CLOSED List

Nodes already visited and processed.

🟢 Why A* is Best?

- Guarantees **shortest path** (if heuristic is valid)
- Faster than BFS and DFS because it uses heuristics
- Used in real-life pathfinding (games, robots)

🟡 2. FULL PYTHON CODE (Perfect for Practical Exam)

```
import heapq


# Node class to store coordinates and costs
class Node:
    def __init__(self, x, y, g=0, h=0, parent=None):
        self.x = x
        self.y = y
        self.g = g  # Cost from start to node
        self.h = h  # Heuristic cost from node to goal
        self.f = g + h  # Total cost
        self.parent = parent

    def __lt__(self, other):
        return self.f < other.f
```

```
# Heuristic function (Manhattan Distance)
def heuristic(current, goal):
    return abs(current.x - goal.x) + abs(current.y - goal.y)


# Helper function to check if a node is walkable
def is_walkable(grid, x, y):
    return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 0


# A* search algorithm
def a_star(grid, start, goal):
    open_list = []
    closed_set = set()

    start_node = Node(start[0], start[1])
    goal_node = Node(goal[0], goal[1])
    start_node.h = heuristic(start_node, goal_node)
    start_node.f = start_node.g + start_node.h

    heapq.heappush(open_list, start_node)

    while open_list:
        current = heapq.heappop(open_list)

        if (current.x, current.y) == (goal_node.x, goal_node.y):
```

```
            path = []
            while current:
                path.append((current.x, current.y))
                current = current.parent
            return path[::-1]

        closed_set.add((current.x, current.y))

        # Check 4-directional neighbors (up, down, left, right)
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            neighbor_x, neighbor_y = current.x + dx, current.y + dy

            if (neighbor_x, neighbor_y) in closed_set or not is_walkable(grid, neighbor_x,
neighbor_y):
                continue

            g = current.g + 1
            neighbor_node = Node(neighbor_x, neighbor_y, g)
            neighbor_node.h = heuristic(neighbor_node, goal_node)
            neighbor_node.f = neighbor_node.g + neighbor_node.h
            neighbor_node.parent = current

            heapq.heappush(open_list, neighbor_node)

    return None  # No path found


# Example grid: 0 = walkable, 1 = obstacle
```

```
grid = [
  [0, 0, 0, 1, 0],
  [0, 1, 0, 1, 0],
  [0, 1, 0, 0, 0],
  [0, 0, 0, 1, 0],
  [1, 0, 0, 0, 0]
]

start = (0, 0)
goal = (4, 4)

path = a_star(grid, start, goal)
print("Path:", path)
```

🔵 3. LINE-BY-LINE EXPLANATION (VERY SIMPLE & CLEAR)

🔥 A* Algorithm Code – Line-by-Line Explanation (Super Easy)

📌 1. Importing Priority Queue

```
import heapq
```

✔ This imports Python's heap queue module.
✔ A heap queue is a special type of priority queue.
✔ It always removes the item with the smallest value first.
✔ We need it because A* always picks the node with lowest f-cost.

📌 2. Creating Node Class

```
class Node:
```

We create a class named Node.
A node stores all information about one point on the grid.

✔ Constructor of Node

```
def __init__(self, x, y, g=0, h=0, parent=None):
```

This function runs when a node is created.
It takes:

- x → row number
- y → column number
- g → cost from start
- h → heuristic cost to goal
- parent → previous node (used to build path)

✔ Save x and y positions

```
self.x = x
self.y = y
```

Stores the node's coordinates.

✔ Store g and h cost

```
self.g = g
self.h = h
```

- g = distance travelled so far
- h = predicted distance to goal

✔ Calculate total cost f

```
self.f = g + h
```

A* decides the best path using:

f = g + h

✔ Save parent node reference

```
self.parent = parent
```

Helps later to retrace the full path.

✔ Compare nodes based on f-cost

```
def __lt__(self, other):
    return self.f < other.f
```

This means:

"A node with smaller f-cost is better."

Used so heapq knows how to compare nodes.

📌 3. Manhattan Distance Function

```
def heuristic(current, goal):
```

This function calculates the heuristic h(n).

✔ Calculate Manhattan Distance

```
return abs(current.x - goal.x) + abs(current.y - goal.y)
```

Meaning:

- Difference in x direction
- PLUS
- Difference in y direction

This gives a prediction of how far the goal is.

📌 4. Function to Check Walkable Cells

def is_walkable(grid, x, y):

Checks whether the cell is safe to move on.

✔ Condition (boundary + obstacle check)

return 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 0

This checks 3 things:

1. 0 <= x < len(grid) → x inside grid
2. 0 <= y < len(grid[0]) → y inside grid
3. grid[x][y] == 0 → cell is not a wall

If all true → we can walk there.

📌 5. A* Main Function

def a_star(grid, start, goal):

This function contains the full A* algorithm.

✔ Create open list & closed set

open_list = []

closed_set = set()

- open_list = nodes to explore
- closed_set = nodes already visited

✔ Create start & goal nodes

start_node = Node(start[0], start[1])

goal_node = Node(goal[0], goal[1])

Convert start (row, col) into Node objects.

✔ Calculate heuristic for start node

```
start_node.h = heuristic(start_node, goal_node)
start_node.f = start_node.g + start_node.h
```

Initial f = 0 + heuristic.

✔ Push start node into priority queue

```
heapq.heappush(open_list, start_node)
```

Start node enters open list.

📌 6. Main A* Loop

```
while open_list:
```

Repeat until open list becomes empty.

✔ Remove node with smallest f-cost

```
    current = heapq.heappop(open_list)
```

A* always processes the most promising node.

✔ Check if we reached the goal

```
    if (current.x, current.y) == (goal_node.x, goal_node.y):
```

If yes → we found the path!

✔ Reconstruct path by tracing parents

```
        path = []
        while current:
            path.append((current.x, current.y))
            current = current.parent
```

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

We reverse the path because we start from goal backwards.

✔ Mark node as visited

```
closed_set.add((current.x, current.y))
```

Means: "We are done with this node."

📌 7. Checking All Neighboring Nodes

```
for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
```

These represent 4 directions:

- Up
- Down
- Left
- Right

✔ Calculate neighbor coordinates

```
neighbor_x = current.x + dx
neighbor_y = current.y + dy
```

✔ Skip if neighbor is invalid or closed

```
if (neighbor_x, neighbor_y) in closed_set or not is_walkable(grid, neighbor_x, neighbor_y):
    continue
```

We avoid:

- Already visited nodes
- Obstacles
- Out-of-bound cells

✔ Calculate g for neighbor

g = current.g + 1

Every move costs 1 step.

✔ Create neighbor node

neighbor_node = Node(neighbor_x, neighbor_y, g)

✔ Calculate heuristic & f cost

neighbor_node.h = heuristic(neighbor_node, goal_node)

neighbor_node.f = neighbor_node.g + neighbor_node.h

✔ Link parent for path reconstruction

neighbor_node.parent = current

✔ Push neighbor to open list

heapq.heappush(open_list, neighbor_node)

A* will consider it later.

📌 8. If no path found

return None

📌 9. Running Example

grid = [

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

Grid where:

- 0 = walkable
- 1 = wall

Start and end positions

start = (0, 0)

goal = (4, 4)

Run A*

path = a_star(grid, start, goal)

print("Path:", path)

This prints the shortest path from start to goal.

🔥 4. VIVA QUESTIONS & ANSWERS (FULL MARKS ANSWERS)

🟢 1. What is A* Algorithm?

A* is an informed search algorithm that finds the **shortest path** using cost + heuristic.

Points:

1. Combination of Uniform Cost Search and Greedy Best First
2. Uses evaluation function f(n) = g(n) + h(n)
3. Guarantees optimal path
4. Uses priority queue

🟢 2. What is g(n)?

Actual cost from start to current node.

🟢 3. What is h(n)?

Heuristic function (estimated distance to goal).

🟢 4. What is f(n)?

Total cost used by A*:

f = g + h

🟢 5. Why Manhattan Distance used?

Because:

1. Works for grid movement
2. Only horizontal/vertical moves allowed
3. Simple and fast

🟢 6. What is OPEN list?

A priority queue containing nodes to be explored.

🟢 7. What is CLOSED list?

Set of nodes already visited.

🟢 8. Why priority queue used?

Because we always want the node with **lowest f-cost**.

🟢 9. Applications of A*

1. GPS pathfinding
2. Robot navigation
3. Game development
4. Maps and Navigation
5. AI decision making

---

10. Advantages of A*

1. Finds shortest path
2. Very fast
3. Uses heuristic to guide search
4. More efficient than BFS/DFS

---

11. Disadvantages of A*

1. Uses more memory
2. Performance depends on heuristic
