

**Project Report On – Pathfinding Using A\* Algorithm**

**Name:** Himalaya Vashistha

**Branch:** CSEAI-B

**Roll no:** 47

**Date:** 11/03/2025

**Introduction**

The A\* (A-Star) algorithm is a popular and efficient pathfinding algorithm used to find the shortest path between two points in a grid-based environment. It combines the actual distance from the starting point (g\_score) with an estimated distance to the goal (heuristic) to calculate the total estimated cost (f\_score). This allows the algorithm to explore the most promising paths first, making it faster and more effective than other pathfinding methods like Dijkstra’s algorithm.

The A\* algorithm is widely used in games, robotics, and artificial intelligence because of its ability to find the shortest and most efficient path while avoiding obstacles.

**Methodology**

The A\* algorithm works by combining the actual distance from the starting point (g\_score) and an estimated distance to the goal (heuristic) to find the shortest path. The algorithm explores the most promising paths first based on the total estimated cost (f\_score). Below are the detailed steps involved in the process:

**🡪 *Steps of the A\* Algorithm:***

1. **Initialize the grid**
   * Create a grid where 0 represents a free path and 1 represents an obstacle.
   * Define the starting and ending points on the grid.
2. **Create the priority queue**
   * Use a priority queue (heap) to store nodes based on the lowest total cost (f\_score).
   * Add the starting point to the queue with an initial cost of 0.
3. **Initialize the g\_score and f\_score**
   * g\_score → Keeps track of the shortest known distance from the start to the current node.
   * f\_score → Estimated total cost (actual distance + heuristic).
   * Set g\_score of the starting point to 0 and f\_score to heuristic(start, end).
4. **Heuristic Calculation**
   * Use the Manhattan distance: heuristic=∣x1−x2∣+∣y1−y2∣\text{heuristic} = |x\_1 - x\_2| + |y\_1 - y\_2|heuristic=∣x1​−x2​∣+∣y1​−y2​∣
   * This gives an estimate of the distance between the current node and the goal.
5. **Explore the current node**
   * Remove the node with the lowest f\_score from the priority queue.
   * If this node is the goal → Path found!
   * If not, explore neighboring nodes.
6. **Check neighboring nodes**
   * Check up, down, left, and right moves.
   * Ensure the move is within the grid boundary and not blocked by an obstacle.
7. **Calculate tentative g\_score**
   * Compute the cost to reach the neighboring node from the current node.
   * If this is the shortest known path to the neighbor, update the g\_score and f\_score.
8. **Update path and add to queue**
   * If the new path is shorter, update the came\_from record to store the current node.
   * Add the neighbor to the priority queue with the new f\_score.
9. **Repeat until the goal is reached**
   * Keep exploring nodes with the lowest f\_score.
   * Continue until the goal is reached or the priority queue becomes empty.
10. **Reconstruct the shortest path**

* Once the goal is reached, backtrack using came\_from to reconstruct the shortest path.
* Start from the goal and trace back to the starting point.

1. **Handle no path scenario**

* If the queue is empty and the goal is not reached → No path exists!
* Return None or display a message indicating that no path is found.

1. **Display the output**

* If the path is found, display the coordinates of the shortest path.
* Optionally, visualize the path on the grid for better understanding

**CODE:**

#Path finding using A\* Algorithm

import heapq

# This function calculates the "guess" of how far we are from the goal.

# We're using the Manhattan distance because it's easy to calculate and works well in grids.

def heuristic(a, b):

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

# Main A\* function

def astar(grid, start, end):

    rows, cols = len(grid), len(grid[0])

    # Priority queue to keep track of nodes to explore

    # We use a heap to make sure the node with the lowest cost is explored first

    open\_set = []

    heapq.heappush(open\_set, (0, start))  # (total cost, node)

    # Keeps track of which node led to the current node (used to reconstruct the path)

    came\_from = {}

    # g\_score = Actual shortest distance from start to this node

    g\_score = {start: 0}

    # f\_score = Estimated total cost (actual distance + heuristic guess)

    f\_score = {start: heuristic(start, end)}

    while open\_set:

        # Pop the node with the lowest f\_score from the queue

        \_, current = heapq.heappop(open\_set)

        # If we've reached the target, we're done!

        if current == end:

            path = []

            while current in came\_from:

                path.append(current)

                current = came\_from[current]

            path.append(start)  # Include the starting point

            path.reverse()  # Reverse to get the path from start to end

            return path

        # These are the 4 possible moves → Right, Down, Left, Up

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

        for dx, dy in neighbors:

            # New coordinates after the move

            neighbor = (current[0] + dx, current[1] + dy)

            # Check if it's within the grid and not an obstacle

            if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and grid[neighbor[0]][neighbor[1]] == 0:

                # Cost to move from current node to neighbor → here it's always 1 step

                tentative\_g\_score = g\_score[current] + 1

                # If this is the shortest path to the neighbor so far

                if tentative\_g\_score < g\_score.get(neighbor, float('inf')):

                    # Update the shortest path to this neighbor

                    came\_from[neighbor] = current

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

                    # Total estimated cost = real cost + heuristic guess

                    f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, end)

                    # Add it to the queue to explore it later

                    heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

    # If we explored everything and didn't find the goal, return None

    return None

# Example grid (0 = open path, 1 = obstacle)

grid = [

    [0, 1, 0, 0, 0],  # 0 → Walkable, 1 → Blocked

    [0, 1, 0, 1, 0],

    [0, 0, 0, 1, 0],

    [1, 1, 0, 0, 0],

]

# Starting and ending points

start = (0, 0)  # Top-left corner

end = (3, 4)    # Bottom-right corner

# Run the A\* function

path = astar(grid, start, end)

# Output the path

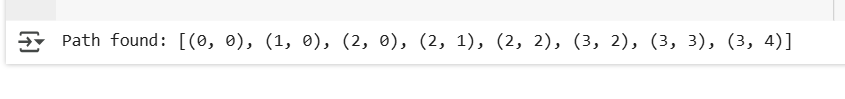
if path:

    print("Path found:", path)

else:

    print("No path found")

**Output Screenshot:**

****

**References:**

🡪 Geeks for Geeks