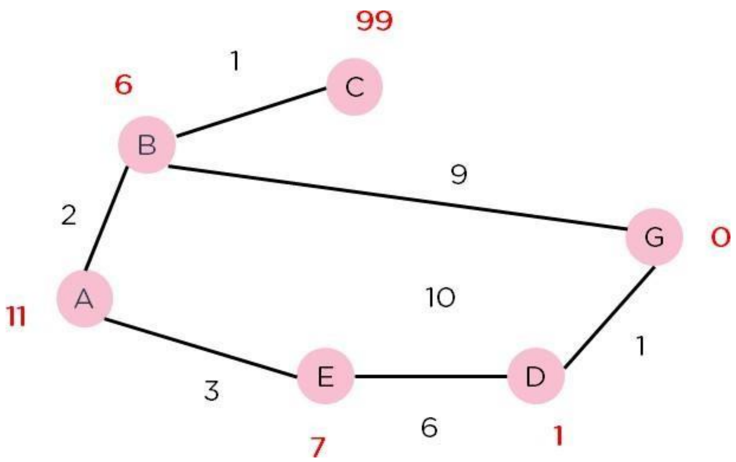
EX.NO:5 **DATE11/9/2024**
Reg.no:220701006

A* SEARCH ALGORITHM

AIM:
To implement A* search algorithm using Python

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.
All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all path's transverse gives you the cost of that route.
Initially, the Algorithm calculates the cost to all its immediate neighboring nodes,n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as:
f(n) = g(n) + h(n), where:
g(n) = cost of traversing from one node to another. This will vary from node to node
h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost.

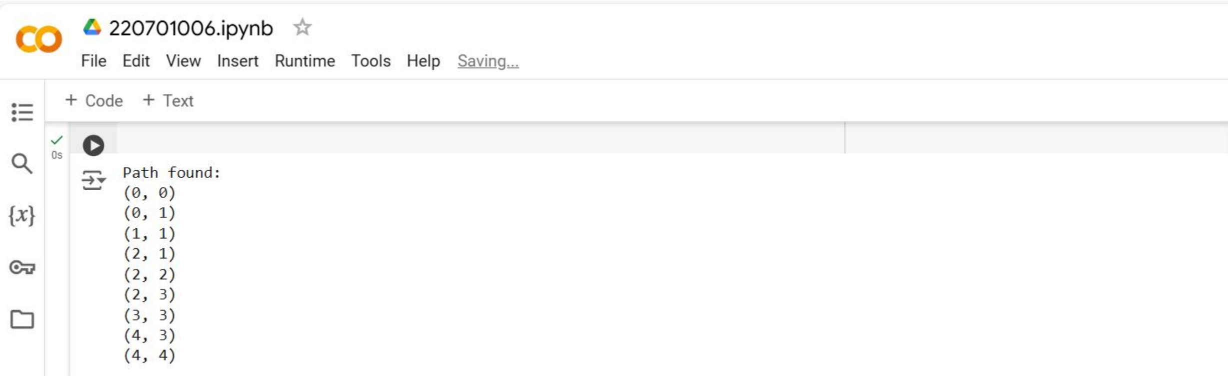


CODE:

```python
from heapq import heappop, heappush

class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to current node
        self.h = 0  # Heuristic (estimated cost from current node to goal)
        self.f = 0  # Total cost (g + h)

    def __eq__(self, other):
        return self.position == other.position

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

def a_star(start, goal, grid):
    # Create start and goal node
    start_node = Node(start)
    goal_node = Node(goal)

    # Open and closed list
    open_list = []
    closed_list = set()

    # Add the start node to open list
    heappush(open_list, start_node)

    # Loop until the open list is empty
    while open_list:
        # Get the node with the lowest f score
        current_node = heappop(open_list)
        closed_list.add(current_node.position)
        # Goal check
        if current_node == goal_node:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        # Generate children (neighbors)
        neighbors = [(0, -1), (0, 1), (-1, 0), (1, 0)]  # Up, Down, Left, Right
        for n in neighbors:
            neighbor_position = (current_node.position[0] + n[0], current_node.position[1] + n[1])

            # Check if the neighbor is within the grid bounds and not an obstacle
            if 0 <= neighbor_position[0] < len(grid) and 0 <= neighbor_position[1]
                    < len(grid[0]) and grid[neighbor_position[0]][neighbor_position[1]] == 0:
                neighbor_node = Node(neighbor_position, current_node)

                # If the neighbor is already in the closed list, skip it
                if neighbor_node.position in closed_list:
                    continue

                # Calculate g, h, and f values
                neighbor_node.g = current_node.g + 1
                neighbor_node.h = abs(neighbor_node.position[0] - goal_node.position[0])
                + abs(neighbor_node.position[1] - goal_node.position[1])  # Manhattan distance
                neighbor_node.f = neighbor_node.g + neighbor_node.h

                # If the neighbor is not in the open list, add it
                if all(neighbor_node != open_node for open_node in open_list):
                    heappush(open_list, neighbor_node)

    return None  # No path found
```

```python
# Example grid (0 = walkable, 1 = obstacle)
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)  # Start position
goal = (4, 4)   # Goal position

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

OUTPUT:



RESULT:

Thus, the A* Search algorithm has been implemented successfully.