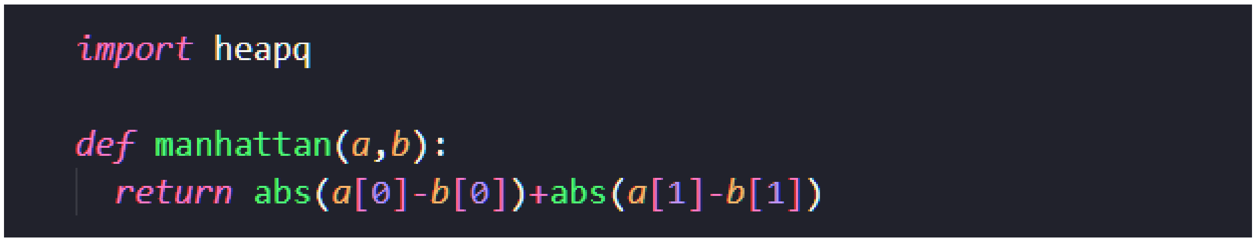
0 means the cell is free to move.

1 means the cell is blocked and cannot be entered.

A robot starts at a specific position (start_x, start_y) and must reach a goal (end_x, end_y).

The robot can move only in four directions: up, down, left, and right.

The task is to find the shortest path from start to goal using the A* search algorithm. If no path exists, the program should return None.



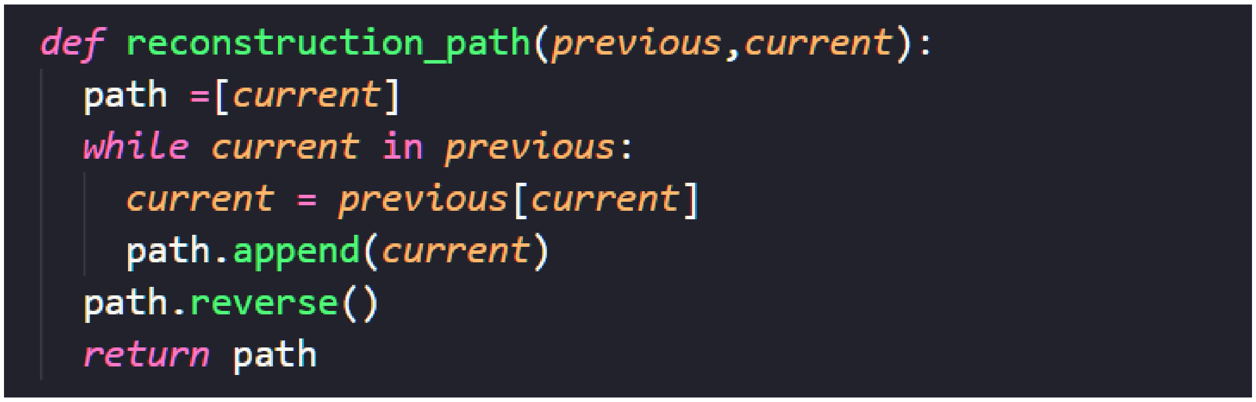
```python
import heapq

def manhattan(a,b):
  return abs(a[0]-b[0])+abs(a[1]-b[1])
```

Heuristic Function (Manhattan Distance)

The code uses the Manhattan distance to estimate how far a cell is from the goal.

Formula: |x1 - x2| + |y1 - y2|

This is appropriate because the robot moves in four directions, and Manhattan distance never overestimates the true cost.



```python
def reconstruction_path(previous,current):
  path =[current]
  while current in previous:
    current = previous[current]
    path.append(current)
  path.reverse()
  return path
```

Path Reconstruction

After reaching the goal, start from the goal and trace back using came_from to the start.

Reverse the collected path so that it goes from start ⟶ goal.

Data Structures Used

1. Priority Queue (heapq)

   Stores cells to explore, prioritized by f_score = g_score + heuristic.

   g_score: cost from start to the current cell.

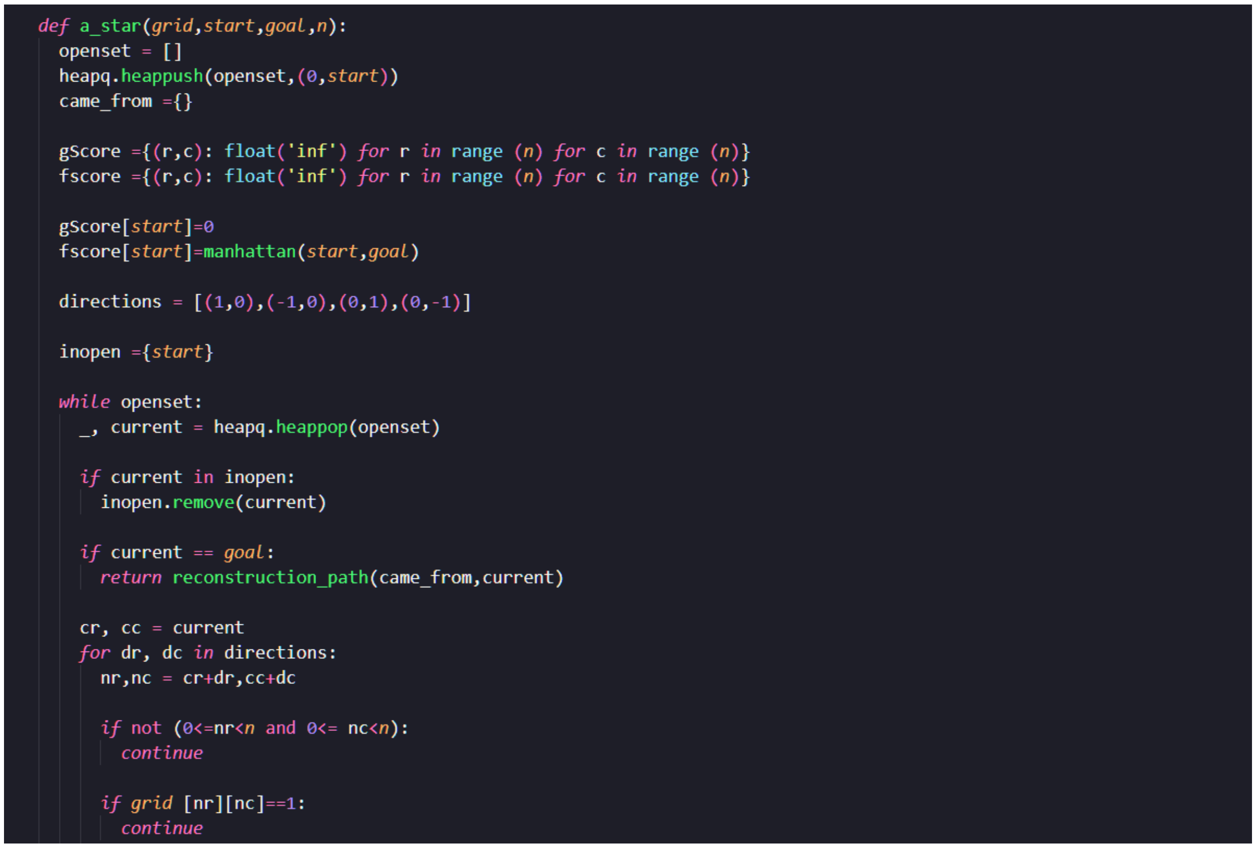
   f_score: estimated total cost from start to goal through the current cell.

2. came_from dictionary

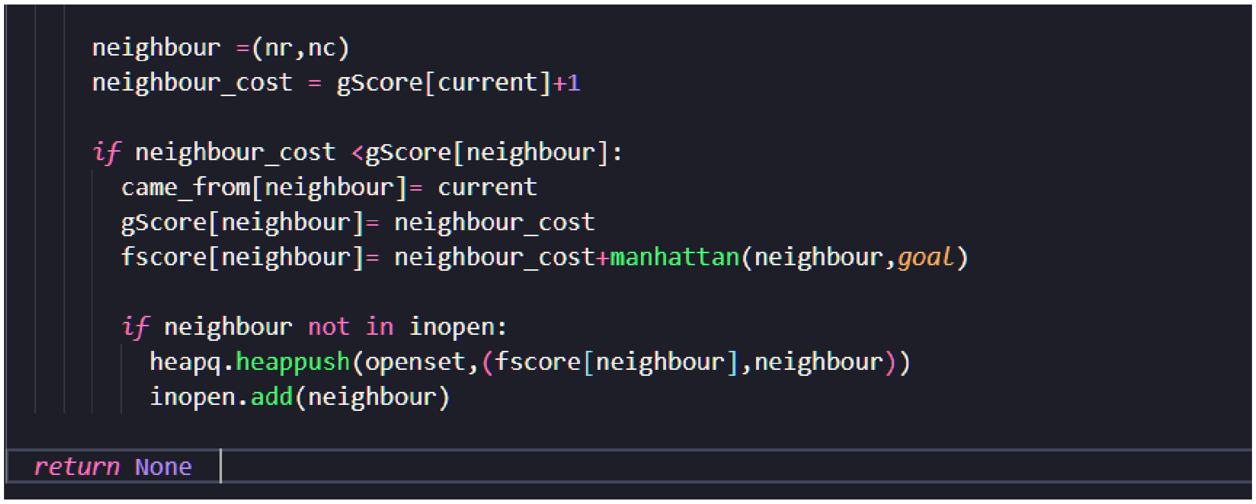
   Stores the parent of each visited cell to reconstruct the path once the goal is reached.

3. inpq set

   Keeps track of which cells are currently in the priority queue to avoid adding duplicates.



```python
def a_star(grid,start,goal,n):
    openset = []
    heapq.heappush(openset,(0,start))
    came_from ={}

    gScore ={(r,c): float('inf') for r in range (n) for c in range (n)}
    fscore ={(r,c): float('inf') for r in range (n) for c in range (n)}

    gScore[start]=0
    fscore[start]=manhattan(start,goal)

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

    inopen ={start}

    while openset:
        _, current = heapq.heappop(openset)

        if current in inopen:
            inopen.remove(current)

        if current == goal:
            return reconstruction_path(came_from,current)

        cr, cc = current
        for dr, dc in directions:
            nr,nc = cr+dr,cc+dc

            if not (0<=nr<n and 0<= nc<n):
                continue

            if grid [nr][nc]==1:
                continue
```



```
        neighbour =(nr,nc)
        neighbour_cost = gScore[current]+1

        if neighbour_cost <gScore[neighbour]:
          came_from[neighbour]= current
          gScore[neighbour]= neighbour_cost
          fscore[neighbour]= neighbour_cost+manhattan(neighbour,goal)

          if neighbour not in inopen:
            heapq.heappush(openset,(fscore[neighbour],neighbour))
            inopen.add(neighbour)

  return None
```

A* Algorithm Logic

1. Start by adding the start cell to the priority queue with f_score = heuristic.

2. Loop while the priority queue is not empty:

  Remove the cell with the lowest f_score (most promising).

  If the current cell is the goal, reconstruct the path using came_from.

  Otherwise, check all four neighbors (up, down, left, right):

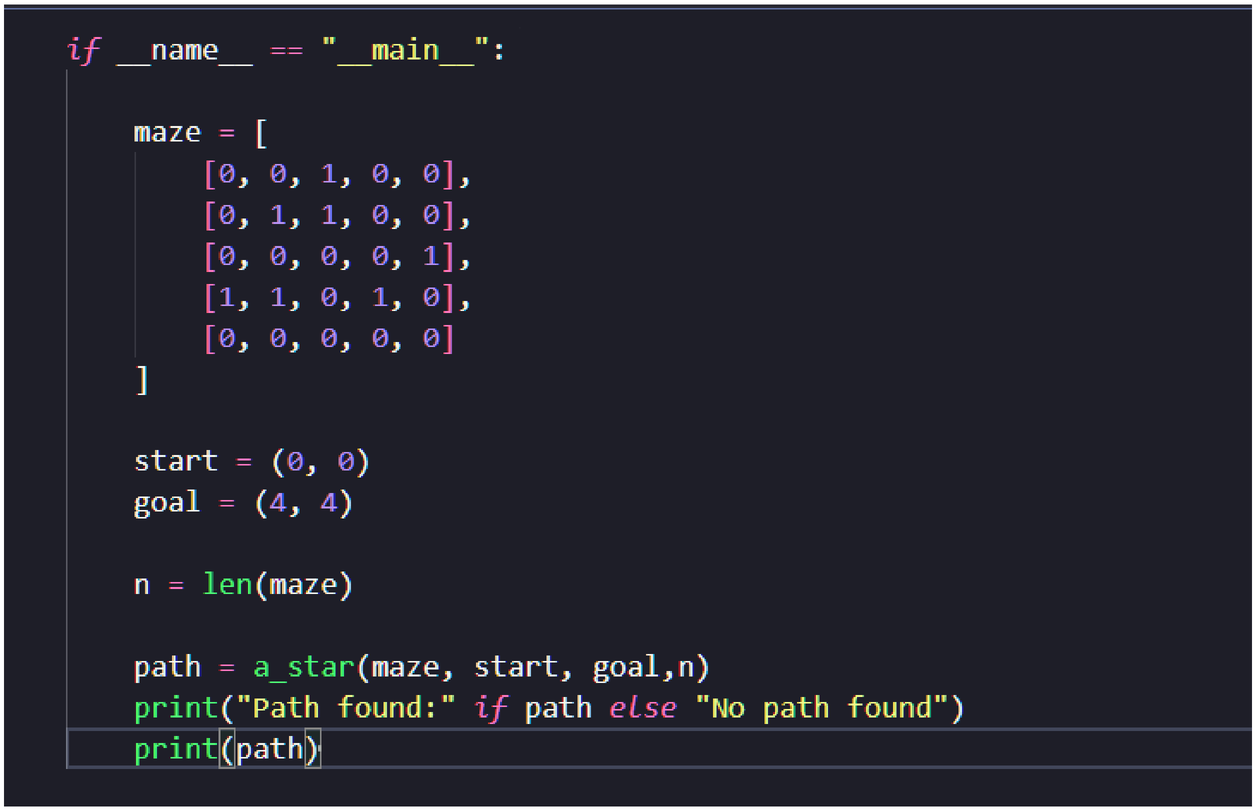
   Ignore cells that are out of bounds or blocked.

   Compute the tentative g_score for the neighbor.

   If this new path is shorter than the previous path to the neighbor:

    Update came_from, g_score, and f_score.

    Add the neighbor to the priority queue if it is not already there.



```
if __name__ == "__main__":

    maze = [
        [0, 0, 1, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 1],
        [1, 1, 0, 1, 0],
        [0, 0, 0, 0, 0]
    ]

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

    n = len(maze)

    path = a_star(maze, start, goal,n)
    print("Path found:" if path else "No path found")
    print(path)
```

Output

If a path is found, print the list of cells (row, col) in order.

If no path exists, print "no path found".