10. Write the python program to implement A\* algorithm

**AIM :** program to implement A\* algorithm

**ALGORITHM :**

1. We create a ‘node’ class to represent each cell in the grid. Each node has attributes for its row, column, cost from start(‘g’) heuristic estimate (‘h’) total cost (‘f’) and a parent node.
2. We initialize the ‘open\_set’ as a priority queue (using heapq) and the ‘closed\_set’ as a set to keep track of visited nodes.
3. We create the start node, set its ‘g’,’h’, and ‘f’ alues, and add it to the ‘open\_set’
4. While the ‘open\_set’ is not empty, we pop the node with the lowest ‘f’ value from the heap.
5. If the current node is the goal node, we reconstruct the path from the goal to the start by following the parent pointers and return the path.
6. Otherwise, we generate the neighbors of the current node and calculate their costs and heuristics. We check if each neighbor is valid and not in the ‘closed\_set’
7. If a neighbor is not in the ‘open\_set’ , we add it. If it is already in the ‘open\_set’ and the new cost is lower, we update its cost and update its position in the heap.
8. If there's no path found, an empty list is returned.

**PROGRAM :**

import heapq

class Node:

def \_\_init\_\_(self, state, parent=None, cost=0, heuristic=0):

self.state = state

self.parent = parent

self.cost = cost

self.heuristic = heuristic

def \_\_lt\_\_(self, other):

return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar(start, goal, neighbors\_func, heuristic\_func):

open\_list = []

closed\_set = set()

start\_node = Node(state=start, cost=0, heuristic=heuristic\_func(start))

heapq.heappush(open\_list, start\_node)

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.state == goal:

path = []

while current\_node:

path.append(current\_node.state)

current\_node = current\_node.parent

return path[::-1]

closed\_set.add(current\_node.state)

for neighbor in neighbors\_func(current\_node.state):

if neighbor in closed\_set:

continue

new\_cost = current\_node.cost + 1

new\_heuristic = heuristic\_func(neighbor)

new\_node = Node(state=neighbor, parent=current\_node, cost=new\_cost, heuristic=new\_heuristic)

existing\_node = next((n for n in open\_list if n.state == neighbor), None)

if existing\_node and new\_cost < existing\_node.cost:

open\_list.remove(existing\_node)

heapq.heappush(open\_list, new\_node)

elif not existing\_node:

heapq.heappush(open\_list, new\_node)

return None

def neighbors(node):

x, y = node

possible\_neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]

return [(nx, ny) for nx, ny in possible\_neighbors if 0 <= nx < 5 and 0 <= ny < 5]

def heuristic(node):

gx, gy = goal

x, y = node

return abs(gx - x) + abs(gy - y)

start = (0, 0)

goal = (4, 4)

path = astar(start, goal, neighbors, heuristic)

if path:

print("Path found:", path)

else:

print("No path found")

**OUT PUT :**

