**EX.NO:5 DATE:4/9/2024**

**Reg.no:220701047**

**A\* SEARCH ALGORITHM**

**AIM:**To implement A\* Algorithm in 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 paths 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

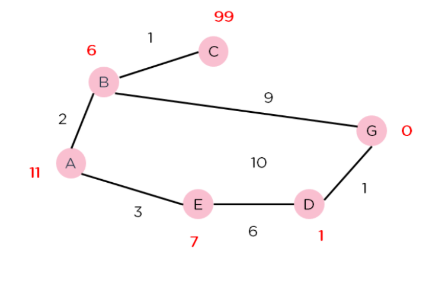
the final cost, then it can be denoted as :

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:**import heappop, heappush

class Node:

def \_init\_(self, position, parent=None):

self.position = position

self.parent = parent

self.g = 0

self.h = 0

self.f = 0

def \_eq\_(self, other):

return self.position == other.position

def \_lt\_(self, other):

return self.f < other.f

def a\_star(start, goal, grid):

start\_node = Node(start)

goal\_node = Node(goal)

open\_list = []

closed\_list = set()

heappush(open\_list, start\_node)

while open\_list:

current\_node = heappop(open\_list)

closed\_list.add(current\_node.position)

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

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

for n in neighbors:

neighbor\_position = (current\_node.position[0] + n[0], current\_node.position[1] + n[1])

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 neighbor\_node.position in closed\_list:

continue

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])

neighbor\_node.f = neighbor\_node.g + neighbor\_node.h

if all(neighbor\_node != open\_node for open\_node in open\_list):

heappush(open\_list, neighbor\_node)

return None

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]

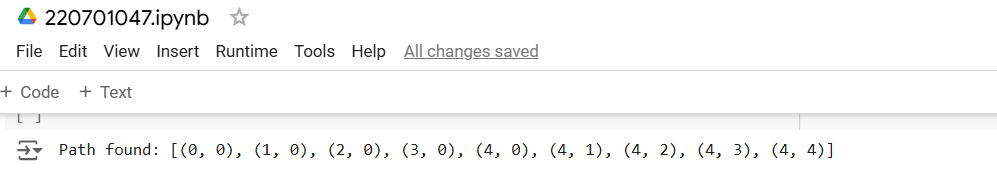
]

start = (0, 0)

goal = (4, 4)

path = a\_star(start, goal, grid)

print("Path found:", path)

**OUTPUT:  
**