**AIM:** Write a program to implement AO\* algorithm

**SOURCE CODE:**

class Graph:

# Initialize the class

def \_\_init\_\_(self, graph\_dict=None, directed=True):

self.graph\_dict = graph\_dict or {}

self.directed = directed

if not directed:

self.make\_undirected()

# Create an undirected graph by adding symmetric edges

def make\_undirected(self):

for a in list(self.graph\_dict.keys()):

for (b, dist) in self.graph\_dict[a].items():

self.graph\_dict.setdefault(b, {})[a] = dist

# Add a link from A and B of given distance, and also add the inverse link if the graph is undirected

def connect(self, A, B, distance=1):

self.graph\_dict.setdefault(A, {})[B] = distance

if not self.directed:

self.graph\_dict.setdefault(B, {})[A] = distance

# Get neighbors or a neighbor

def get(self, a, b=None):

links = self.graph\_dict.setdefault(a, {})

if b is None:

return links

else:

return links.get(b)

# Return a list of nodes in the graph

def nodes(self):

s1 = set([k for k in self.graph\_dict.keys()])

s2 = set([k2 for v in self.graph\_dict.values() for k2, v2 in v.items()])

nodes = s1.union(s2)

return list(nodes)

# This class represent a node

class Node:

# Initialize the class

def \_\_init\_\_(self, name:str, parent:str):

self.name = name

self.parent = parent

self.g = 0 # Distance to start node

self.h = 0 # Distance to goal node

self.f = 0 # Total cost

# Compare nodes

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

return self.name == other.name

# Sort nodes

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

return self.f < other.f

# Print node

def \_\_repr\_\_(self):

return ('({0},{1})'.format(self.name, self.f))

# A\* search

def astar\_search(graph, heuristics, start, end):

# Create lists for open nodes and closed nodes

open = []

closed = []

# Create a start node and an goal node

start\_node = Node(start, None)

goal\_node = Node(end, None)

# Add the start node

open.append(start\_node)

# Loop until the open list is empty

while len(open) > 0:

# Sort the open list to get the node with the lowest cost first

open.sort()

# Get the node with the lowest cost

current\_node = open.pop(0)

# Add the current node to the closed list

closed.append(current\_node)

# Check if we have reached the goal, return the path

if current\_node == goal\_node:

path = []

while current\_node != start\_node:

path.append(current\_node.name + ': ' + str(current\_node.g))

current\_node = current\_node.parent

path.append(start\_node.name + ': ' + str(start\_node.g))

# Return reversed path

return path[::-1]

# Get neighbours

neighbors = graph.get(current\_node.name)

# Loop neighbors

for key, value in neighbors.items():

# Create a neighbor node

neighbor = Node(key, current\_node)

# Check if the neighbor is in the closed list

if(neighbor in closed):

continue

# Calculate full path cost

neighbor.g = current\_node.g + graph.get(current\_node.name, neighbor.name)

neighbor.h = heuristics.get(neighbor.name)

neighbor.f = neighbor.g + neighbor.h

# Check if neighbor is in open list and if it has a lower f value

if(add\_to\_open(open, neighbor) == True):

# Everything is green, add neighbor to open list

open.append(neighbor)

# Return None, no path is found

return None

# Check if a neighbor should be added to open list

def add\_to\_open(open, neighbor):

for node in open:

if (neighbor == node and neighbor.f > node.f):

return False

return True

# The main entry point for this module

def main():

# Create a graph

# Run the search algorithm

path = astar\_search(graph, heuristics, 'Frankfurt', 'Ulm')

print(path)

print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

****