**AI LAB EXP – 5**

**IMPLEMENTATION OF A\* ALGORITHM**

***Submitted by***

**Vinoth S**

**RA1911030010103**

**GitHub Link: https://github.com/vk1308**

**AIM:**

To implement the A\* algorithm using python

**Code:**

*# graph class*

class Graph:

*# init 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 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 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 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)

*# node class*

class Node:

*# init 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):

*# lists for open nodes and closed nodes*

open = []

closed = []

*# a start node and an goal node*

start\_node = Node(start, None)

goal\_node = Node(end, None)

*# add start node*

open.append(start\_node)

*# loop until the open list is empty*

while len(open) > 0:

open.sort() *# sort open list to get the node with the lowest cost first*

current\_node = open.pop(0) *# get node with the lowest cost*

closed.append(current\_node) *# add current node to the closed list*

*# 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 path[::-1]

neighbors = graph.get(current\_node.name) *# get neighbours*

*# loop neighbors*

for key, value in neighbors.items():

neighbor = Node(key, current\_node) *# create neighbor node*

if(neighbor in closed): *# check if the neighbor is in the closed list*

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

*# create a graph*

graph = Graph() *# user-based input for edges will be updated in the upcoming days*

*# create graph connections (Actual distance)*

graph.connect('Frankfurt', 'Wurzburg', 111)

graph.connect('Frankfurt', 'Mannheim', 85)

graph.connect('Wurzburg', 'Nurnberg', 104)

graph.connect('Wurzburg', 'Stuttgart', 140)

graph.connect('Wurzburg', 'Ulm', 183)

graph.connect('Mannheim', 'Nurnberg', 230)

graph.connect('Mannheim', 'Karlsruhe', 67)

graph.connect('Karlsruhe', 'Basel', 191)

graph.connect('Karlsruhe', 'Stuttgart', 64)

graph.connect('Nurnberg', 'Ulm', 171)

graph.connect('Nurnberg', 'Munchen', 170)

graph.connect('Nurnberg', 'Passau', 220)

graph.connect('Stuttgart', 'Ulm', 107)

graph.connect('Basel', 'Bern', 91)

graph.connect('Basel', 'Zurich', 85)

graph.connect('Bern', 'Zurich', 120)

graph.connect('Zurich', 'Memmingen', 184)

graph.connect('Memmingen', 'Ulm', 55)

graph.connect('Memmingen', 'Munchen', 115)

graph.connect('Munchen', 'Ulm', 123)

graph.connect('Munchen', 'Passau', 189)

graph.connect('Munchen', 'Rosenheim', 59)

graph.connect('Rosenheim', 'Salzburg', 81)

graph.connect('Passau', 'Linz', 102)

graph.connect('Salzburg', 'Linz', 126)

*# make graph undirected, create symmetric connections*

graph.make\_undirected()

*# create heuristics (straight-line distance, air-travel distance)*

heuristics = {}

heuristics['Basel'] = 204

heuristics['Bern'] = 247

heuristics['Frankfurt'] = 215

heuristics['Karlsruhe'] = 137

heuristics['Linz'] = 318

heuristics['Mannheim'] = 164

heuristics['Munchen'] = 120

heuristics['Memmingen'] = 47

heuristics['Nurnberg'] = 132

heuristics['Passau'] = 257

heuristics['Rosenheim'] = 168

heuristics['Stuttgart'] = 75

heuristics['Salzburg'] = 236

heuristics['Wurzburg'] = 153

heuristics['Zurich'] = 157

heuristics['Ulm'] = 0

*# run the search algorithm*

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

print("Path:", path)

**Time Complexity:**

The time complexity of A\* Search Algorithm depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: O(b^d), where b is the branching factor (the average number of successors per state). This assumes that a goal state exists at all, and is reachable from the start state; if it is not, and the state space is infinite, the algorithm will not terminate.

**Space Complexity:**

The space complexity of A\* Search Algorithm is roughly the same as that of all other graph search algorithms i.e. O(b^d), as it keeps all generated nodes in memory.

**Output:**







