Name: Krishnaprasad Awala

Roll No: D2215004

Enrolment no: MITU21BTITD006

Class: TYIT CORE 2

**AI LAB Assignment No 3:**

**Aim: To implement A\* algorithm to find shortest path**

**CODE:**

from myutils import \*

infinity = float('inf')

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=0):

self.state = state

self.parent = parent

self.action = action

self.path\_cost = path\_cost

self.depth = 0

if parent:

self.depth = parent.depth + 1

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

return "<Node {}>".format(self.state)

def expand(self, problem):

return [self.child\_node(problem, action)

for action in problem.actions(self.state)]

def child\_node(self, problem, action): # to make node object of each child

next\_state = problem.result(self.state, action)

new\_cost = problem.path\_cost(self.path\_cost, self.state, action, next\_state)

next\_node = Node(next\_state, self, action, new\_cost)

return next\_node

def solution(self): # extracts the path of solution

return [node.state for node in self.path()]

def path(self): # extracts the path of any node starting from current to source

node, path\_back = self, []

while node:

path\_back.append(node)

node = node.parent

return list(reversed(path\_back)) # order changed to show from source to current

class Graph: # For undirected graphs only

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

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

self.directed = directed

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

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

if b is None:

return links

else:

return links.get(b)

class Problem(object):

def \_\_init\_\_(self, initial, goal=None):

self.initial = initial

self.goal = goal

def actions(self, state):

raise NotImplementedError

def result(self, state, action):

raise NotImplementedError

def goal\_test(self, state):

if isinstance(self.goal, list):

return is\_in(state, self.goal)

else:

return state == self.goal

def path\_cost(self, c, state1, action, state2):

return c + 1

class GraphProblem(Problem):

def \_\_init\_\_(self, initial, goal, graph):

Problem.\_\_init\_\_(self, initial, goal)

self.graph = graph

def actions(self, A):

return list(self.graph.get(A).keys())

def result(self, state, action):

return action

def path\_cost(self, cost\_so\_far, A, action, B):

return cost\_so\_far + (self.graph.get(A, B) or infinity)

def h(self, node):

locs = getattr(self.graph, 'locations', None)

if locs:

return int(distance(locs[node.state], locs[self.goal]))

else:

return infinity

def astar\_search(problem):

node = Node(problem.initial)

if problem.goal\_test(node.state):

return node

gval = node.path\_cost

hval = problem.h(node)

nodelist = [{gval + hval:node}]

while nodelist:

entry\_num = closest\_node\_entry\_num(nodelist)

min\_dist = list(nodelist[entry\_num].keys())[0]

closest\_node = nodelist[entry\_num][min\_dist]

print("current nodes:" , nodelist)

print("min dist" , min\_dist , "closest dist", closest\_node )

input("press enter")

if problem.goal\_test(closest\_node.state):

return closest\_node

nodelist.pop(entry\_num)

for child in closest\_node.expand(problem):

gval = child.path\_cost

hval = problem.h(child)

nodelist.append({gval + hval: child})

def closest\_node\_entry\_num(nodelist):

min\_index = 0

min\_dist = list(nodelist[0].keys())[0]

for n in range(1, len(nodelist)):

dist = list(nodelist[n].keys())[0]

if dist < min\_dist:

min\_index = n

min\_dist = dist

return min\_index

def UndirectedGraph(graph\_dict=None):

return Graph(graph\_dict=graph\_dict, directed=False)

romania\_map = UndirectedGraph(

{'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},

'Bucharest': {'Urziceni': 85, 'Pitesti': 101, 'Giurgiu': 90, 'Fagaras': 211},

'Craiova': {'Drobeta': 120, 'Rimnicu': 146, 'Pitesti': 138},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Eforie': {'Hirsova': 86}, 'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Hirsova': {'Urziceni': 98, 'Eforie': 86},

'Iasi': {'Vaslui': 92, 'Neamt': 87},

'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Pitesti': {'Rimnicu': 97, 'Bucharest': 101, 'Craiova': 138},

'Rimnicu': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},

'Urziceni': {'Vaslui': 142, 'Bucharest': 85, 'Hirsova': 98},

'Zerind': {'Arad': 75, 'Oradea': 71},

'Sibiu': {'Arad': 140, 'Fagaras': 99, 'Oradea': 151, 'Rimnicu': 80},

'Timisoara': {'Arad': 118, 'Lugoj': 111},

'Giurgiu': {'Bucharest': 90}, 'Mehadia': {'Drobeta': 75, 'Lugoj': 70},

'Vaslui': {'Iasi': 92, 'Urziceni': 142}, 'Neamt': {'Iasi': 87}})

romania\_map.locations = dict(Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),

Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),

Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),

Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),

Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),

Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),

Vaslui=(509, 444), Zerind=(108, 531))

def solve\_prof(src, dist):

print("\nSolving for "+src + " to " + dist+ "...")

problem = GraphProblem(src, dist, romania\_map)

resultnode = astar\_search(problem)

print("Path taken :", resultnode.path())

print("Path Cost :", resultnode.path\_cost)

solve\_prof('Arad', 'Bucharest')

**OUTPUT:**

