**Practical no -3**

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):

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

next\_node = Node(next\_state, self, action,

problem.path\_cost(self.path\_cost, self.state,

action, next\_state))

return next\_node

def solution(self):

return [node.action for node in self.path()[1:]]

def path(self):

node, path\_back = self, []

while node:

path\_back.append(node)

node = node.parent

return list(reversed(path\_back))

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

return isinstance(other, Node) and self.state == other.state

def \_\_hash\_\_(self):

return hash(self.state)

class Graph:

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

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

self.directed = directed

if not directed:

**Roll no – 17,19**

self.make\_undirected()

def make\_undirected(self):

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

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

self.connect1(b, a, dist)

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

self.connect1(A, B, distance)

if not self.directed:

self.connect1(B, A, distance)

def connect1(self, A, B, distance):

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

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

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

if b is None:

return links

else:

return links.get(b)

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)

def best\_first\_graph\_search(problem, f):

f = memoize(f, 'f')

node = Node(problem.initial)

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

return node

frontier = PriorityQueue('min', f)

frontier.append(node)

explored = set()

while frontier:

node = frontier.pop()

print("popping node : " , node)

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

return node

explored.add(node.state)

for child in node.expand(problem):

print("adding child :", child)

if child.state not in explored and child not in frontier:

frontier.append(child)

elif child in frontier:

incumbent = frontier[child]

print(child , " in frontier. incumbent - ", incumbent)

if f(child) < f(incumbent):

del frontier[incumbent]

frontier.append(child)

return None

def astar\_search(problem, h=None):

h = memoize(h or problem.h, 'h')

return best\_first\_graph\_search(problem, lambda n: n.path\_cost + h(n))

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

def value(self, state):

raise NotImplementedError

def UndirectedGraph(graph\_dict=None):

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

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 find\_min\_edge(self):

m = infinity

for d in self.graph.graph\_dict.values():

local\_min = min(d.values())

m = min(m, local\_min)

return m

def h(self, node):

"""h function is straight-line distance from a node's state to goal."""

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

if locs:

if type(node) is str:

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

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

else:

return infinity

romania\_map = UndirectedGraph(dict(

Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),

Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),

Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),

Drobeta=dict(Mehadia=75),

Eforie=dict(Hirsova=86),

Fagaras=dict(Sibiu=99),

Hirsova=dict(Urziceni=98),

Iasi=dict(Vaslui=92, Neamt=87),

Lugoj=dict(Timisoara=111, Mehadia=70),

Oradea=dict(Zerind=71, Sibiu=151),

Pitesti=dict(Rimnicu=97),

Rimnicu=dict(Sibiu=80),

Urziceni=dict(Vaslui=142)))

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))

romania\_problem = GraphProblem('Drobeta','Oradea', romania\_map)

resultnode = astar\_search(romania\_problem)

print(resultnode.path())

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

**Output:**

**popping node : <Node Drobeta>**

**adding child : <Node Mehadia>**

**adding child : <Node Craiova>**

**popping node : <Node Mehadia>**

**adding child : <Node Drobeta>**

**adding child : <Node Lugoj>**

**popping node : <Node Lugoj>**

**adding child : <Node Timisoara>**

**adding child : <Node Mehadia>**

**popping node : <Node Timisoara>**

**adding child : <Node Arad>**

**adding child : <Node Lugoj>**

**popping node : <Node Craiova>**

**adding child : <Node Drobeta>**

**adding child : <Node Rimnicu>**

**adding child : <Node Pitesti>**

**popping node : <Node Rimnicu>**

**adding child : <Node Sibiu>**

**adding child : <Node Craiova>**

**adding child : <Node Pitesti>**

**popping node : <Node Arad>**

**adding child : <Node Zerind>**

**adding child : <Node Sibiu>**

**adding child : <Node Timisoara>**

**popping node : <Node Sibiu>**

**adding child : <Node Arad>**

**adding child : <Node Fagaras>**

**adding child : <Node Oradea>**

**adding child : <Node Rimnicu>**

**popping node : <Node Zerind>**

**adding child : <Node Arad>**

**adding child : <Node Oradea>**

**popping node : <Node Oradea>**

**[<Node Drobeta>, <Node Craiova>, <Node Rimnicu>, <Node Sibiu>, <Node Oradea>]**

**Path Cost : 497**