**Practical No 1: Implement BFS algorithm for Romanian map Problem**

from collections import deque

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 \_\_lt\_\_(self, node):**

return self.state < node.state

**def expand(self, problem):**

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

foractioninproblem.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

**class Graph:**

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

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

self.directed = directed

if not directed:

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 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.graphdict.values()for k2, v2 in v.items()])

nodes = s1.union(s2)

return list(nodes)

**def UndirectedGraph(graph\_dict=None):**

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

**class Problem(object):**

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

self.initial = initial

self.goal = goal

**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

**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):**

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

**def breadth\_first\_tree\_search(problem):**

frontier = deque([Node(problem.initial)])

print("starting from : ", frontier)

while frontier:

node = frontier.popleft()

print("going to :", node)

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

return node

frontier.extend(node.expand(problem))

return None

mumbai\_map = UndirectedGraph(dict(

kurla=dict(sion=5,santacruz=2),

sion=dict(kurla=5,bandra=3,dadar=4),

dadar=dict(sion=4,bandra=4),

bandra=dict(santacruz=3,sion=3,dadar=4),

santacruz=dict(bandra=3,kurla=2)))

mumbai\_problem = GraphProblem('kurla','dadar', mumbai\_map)

print(breadth\_first\_tree\_search(mumbai\_problem))

**Output:**

**starting from : deque([<Node kurla>])**

**going to : <Node kurla>**

**going to : <Node sion>**

**going to : <Node santacruz>**

**going to : <Node kurla>**

**going to : <Node bandra>**

**going to : <Node dadar>**

**<Node dadar>**

**Roll No: 13,14**