**PRACTICAL NO 1**

**\*Implement Breadth first search algorithm for Romanian map problem.**

**Code:**

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): # to print node objects

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

def expand(self, problem): # to extract children

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)

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

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 Problem(object): # same as given in theory

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

self.initial = initial

self.goal = goal

def actions(self, state):

raise NotImplementedError

def goal\_test(self, state):

return state == self.goal

def result(self, state, action):

raise NotImplementedError

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

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

class GraphProblem(Problem): # subclass of problem, few functions overriden

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

class Graph: # to represent 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()):

print("processing node ...", a)

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

print("-->", a, " connects ", b, " by distance :", dist)

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

links = self.graph\_dict.get(a)

if b is None:

return links

else:

cost = links.get(b)

return cost

def UndirectedGraph(graph\_dict=None): # this function creates graph

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

def breadth\_first\_tree\_search(problem): # our algorithm

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

print("Search begins from : ", frontier)

while frontier:

node = frontier.popleft()

print("Now exploring...", node)

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

return node

x = node.expand(problem)

print("Expanded Nodes :",x)

frontier.extend(x)

return None

#we are giving full description of graph through dictionary. the Graph class is not building any additional links

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}})

print("after construcing grpah - ")

print(romania\_map.graph\_dict)

print("-----\n\n\n")

print("Children of Arad ", romania\_map.get('Arad'))

print("-----\n\n\n")

print("==============BFS Algo=================")

print("distance from arad to Bucharest = ",romania\_map.get('Arad','Bucharest'))

print("-----\n\n\n")

print("=============== BFS Algo ====================")

romania\_problem = GraphProblem('Arad','Bucharest', romania\_map)

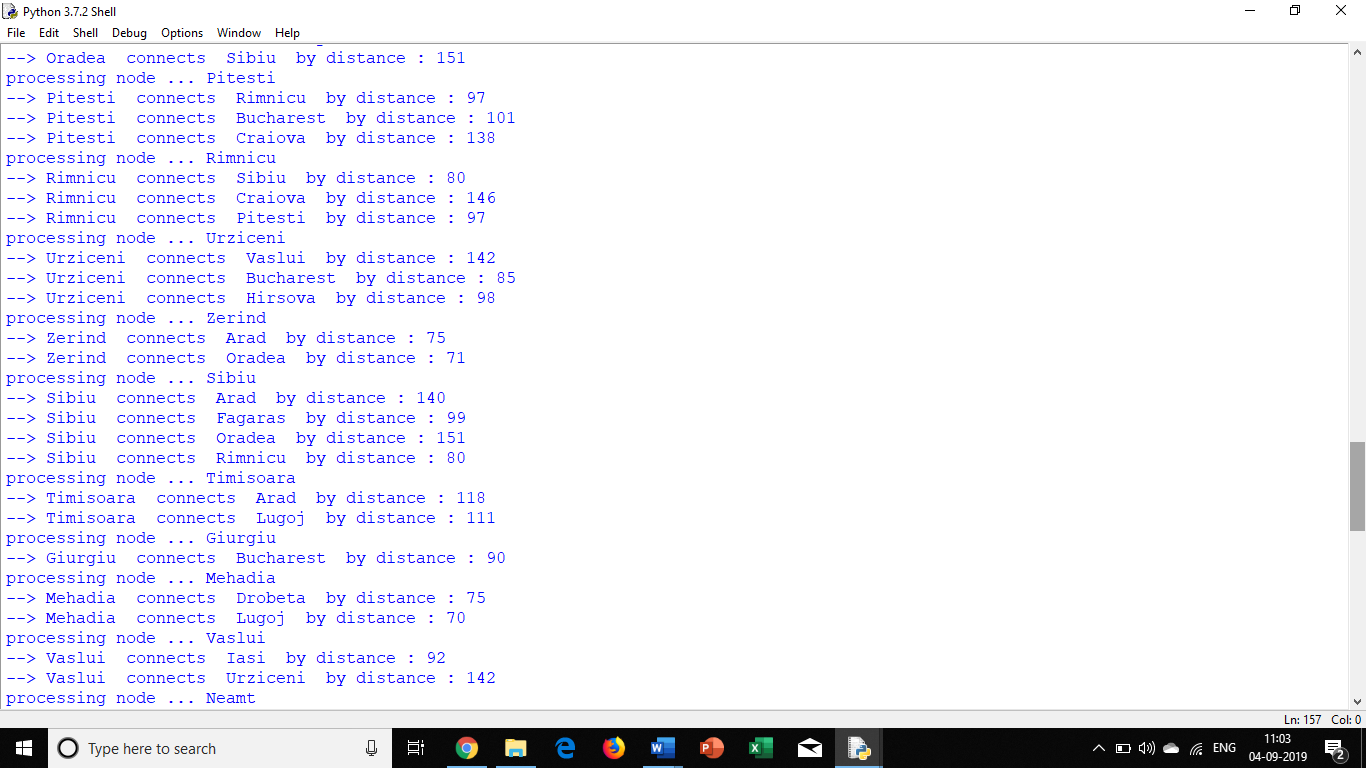
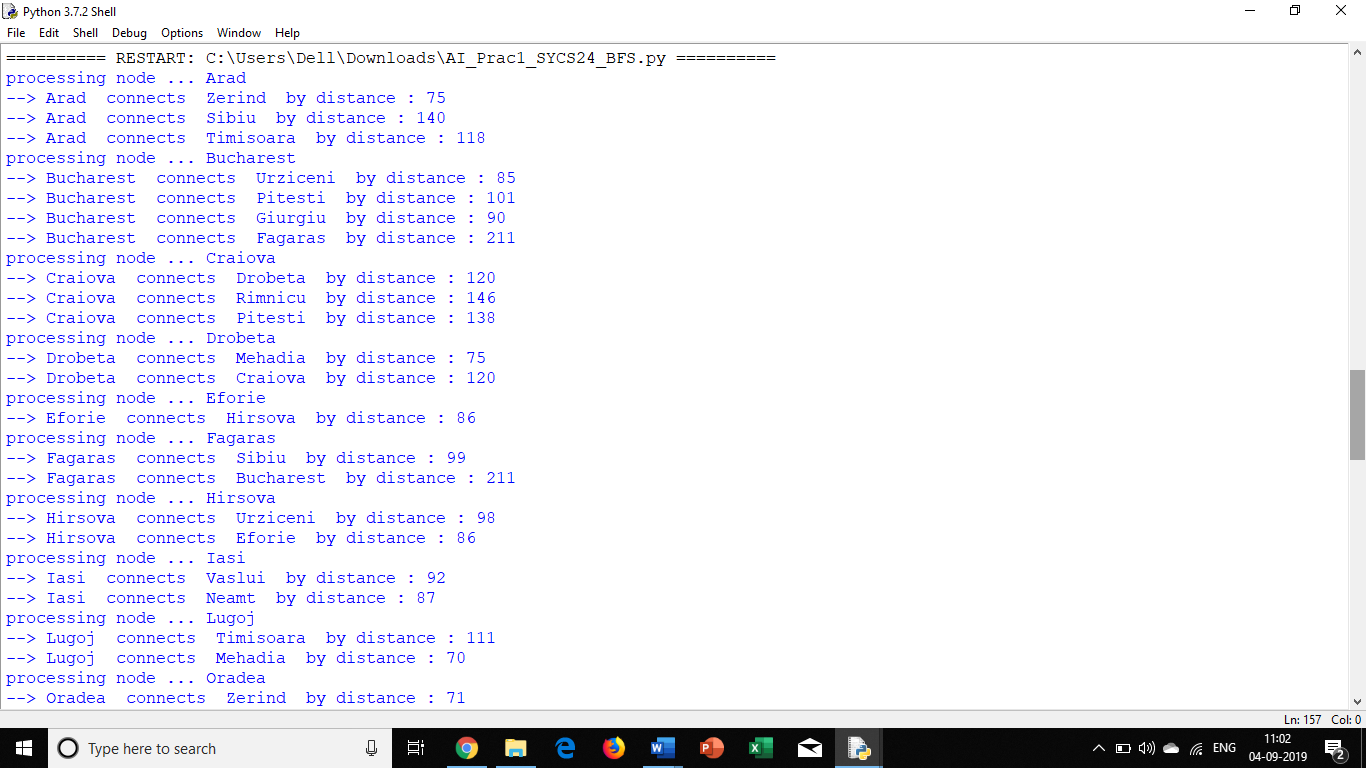
print("Keys of Arad ", romania\_problem.actions( 'Arad'))

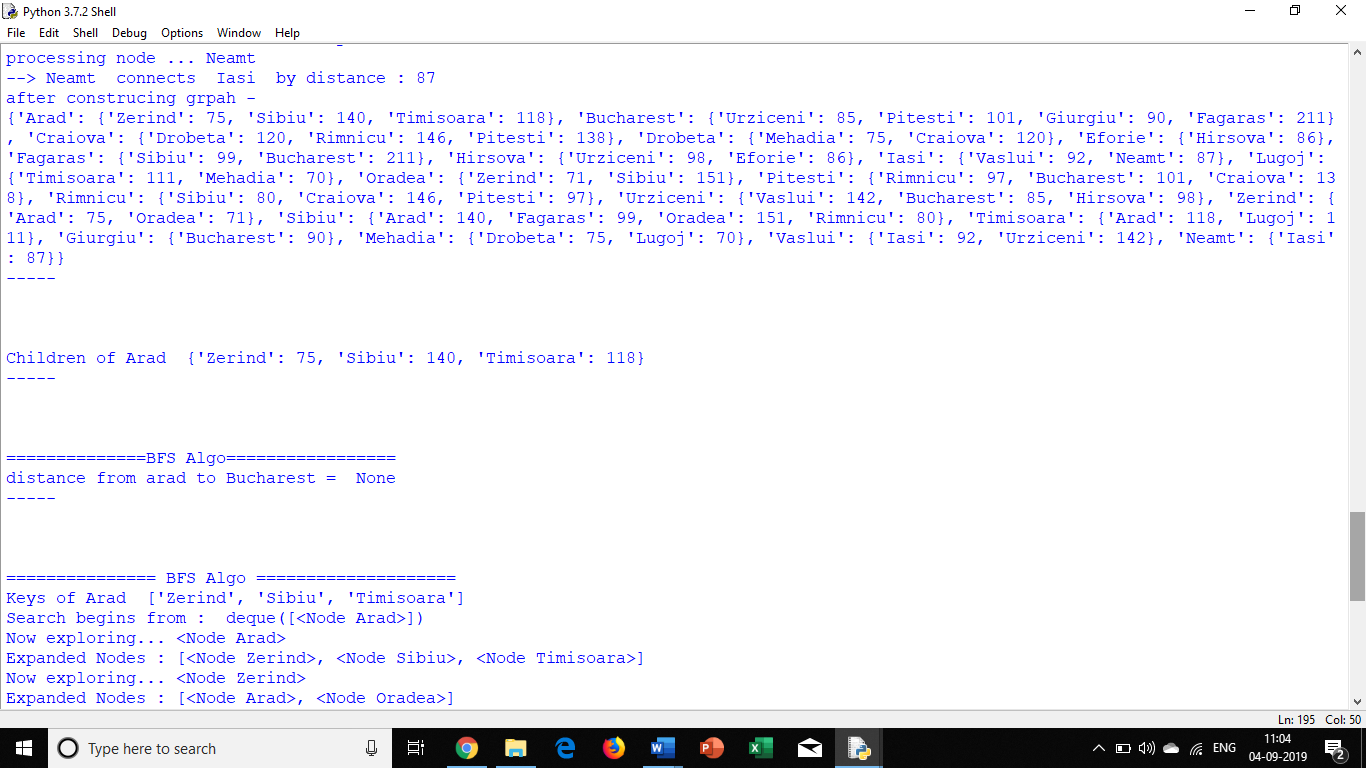
finalnode = breadth\_first\_tree\_search(romania\_problem)

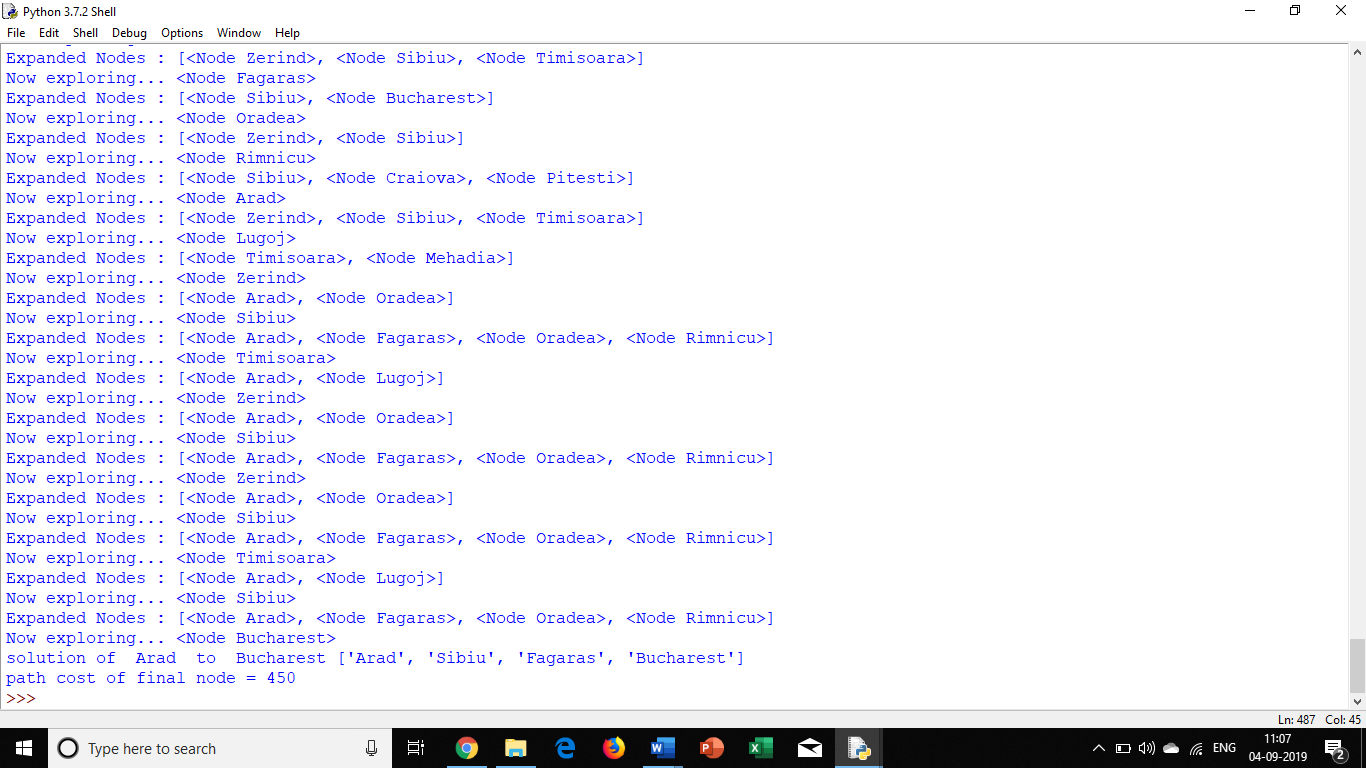
print("solution of ", romania\_problem.initial, " to ", romania\_problem.goal, finalnode.solution())

print("path cost of final node =", finalnode.path\_cost)

**Output:**







**PRACTICAL NO 2**

**\*Implement Iterative deep depth first search for Romanian map problem**

**Code:**

from collections import deque

import sys

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:

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 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 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

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 depth\_limited\_search(problem, limit=50):

def recursive\_dls(node, problem, limit):

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

return node

elif limit == 0:

return 'cutoff'

else:

cutoff\_occurred = False

for child in node.expand(problem):

result = recursive\_dls(child, problem, limit - 1)

if result == 'cutoff':

cutoff\_occurred = True

elif result is not None:

return result

return 'cutoff' if cutoff\_occurred else 'Not found'

return recursive\_dls(Node(problem.initial), problem, limit)

def iterative\_deepening\_search(problem, limit):

for depth in range(0,limit):

print("checking with depth :", depth)

result = depth\_limited\_search(problem, depth)

print("result : ", result)

return result

#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 = 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}})

print("searching from arad to bucharest with level 5...")

romania\_problem = GraphProblem('Arad','Bucharest', romania\_map)

print(iterative\_deepening\_search(romania\_problem, 5))

print("searching from arad to sibiu with level 2...")

romania\_problem = GraphProblem('Arad','Sibiu', romania\_map)

print(iterative\_deepening\_search(romania\_problem, 2))

**Output:**



**PRACTICAL NO 3**

**\*Implement A\* search algorithm for Romanian map problem.**

**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):

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:

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('Arad','Bucharest', romania\_map)

resultnode = astar\_search(romania\_problem)

print(resultnode.path())

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

print('my map...',romania\_map.get('Arad'))

print('my map...',romania\_map.locations)

**Output:**



**PRACTICAL NO 4**

**\*Implement recursive best-first search algorithm for Romanian map problem.**

**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.f=0 #extra variable to represent total 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)

def RecursiveBFS(problem) : #algo 3.26

return RBFS(problem, Node(problem.initial),infinity)

def RBFS(problem, node,f\_limit) : #algo 3.26

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

return [node, None]

successors = []

for child in node.expand(problem):

gval = child.path\_cost

hval = problem.h(child)

child.f = gval+hval #max(s.g + s.h, node.f )) instead of max function function, I am taking only s.g+s.h as I feel this condition may not occur in undirected graph

successors.append(child)

if len(successors) == 0 :

return [None, infinity]

while True:

best = lowest\_fvalue\_node(successors)

if best.f > f\_limit :

return [None, best.f]

alternative = second\_lowest\_fvalue(successors, best.f)

x = RBFS(problem, best, min(f\_limit, alternative))

result = x[0]

best.f = x[1]

if result != None :

return [result, None]

def second\_lowest\_fvalue(nodelist,lowest\_f): # extracts second lowest f value

secondmin\_fval = infinity

for n in range(0,len(nodelist)):

if nodelist[n].f > lowest\_f and nodelist[n].f < secondmin\_fval :

secondmin\_fval = nodelist[n].f

return secondmin\_fval

def lowest\_fvalue\_node(nodelist): # extracts index in list of an entry representing minimum f value

min\_fval = nodelist[0].f

min\_fval\_node\_index=0

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

if nodelist[n].f < min\_fval :

min\_fval\_node\_index = n

min\_fval = nodelist[n].f

return nodelist[min\_fval\_node\_index]

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 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])) ##this line works

else:

return infinity

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=366 , Bucharest=0, Craiova =160, Drobeta=242, Eforie=161, Fagaras=176, Giurgiu=77, Hirsova=151, Iasi=226, Lugoj=244, Mehadia=241, Neamt=234, Oradea=380, Pitesti=100, Rimnicu=193, Sibiu=253, Timisoara=329, Vaslui=199,Urziceni =80, Zerind=374)

# SLD values given in the table, but they dont match if you calculate distance formula from the below locations.

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))

print("\nSolving for arad to bucharest...")

romania\_problem = GraphProblem('Arad','Bucharest', romania\_map)

resultnode = RecursiveBFS(romania\_problem)

if(resultnode[0] != None ):

print("Path taken :" , resultnode[0].path())

print("Path Cost :" , resultnode[0].path\_cost)

print("\n\nSolving for drobeta to vaslui....")

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

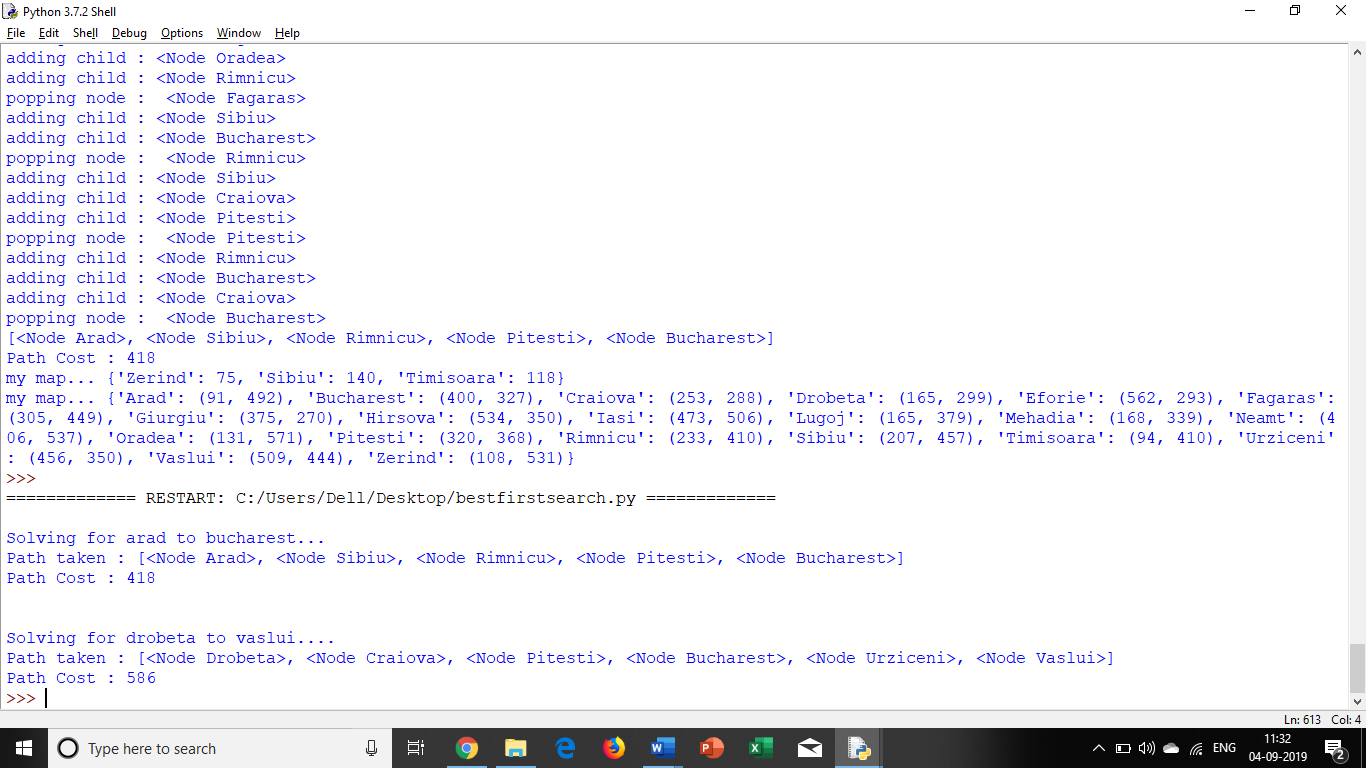
resultnode = RecursiveBFS(romania\_problem)

if(resultnode[0] != None ):

print("Path taken :" , resultnode[0].path())

print("Path Cost :" , resultnode[0].path\_cost)

**Output:**



**PRACTICAL NO 5**

**\*Implement decision tree learning algorithm for the restaurant waiting problem.**

**Code:**

import math

class DataSet:

def \_\_init\_\_(self,ds):

self.dataset = ds

def print(self):

print(self.dataset)

def uniqueAns(self) : # calculates how many unique answers are there

ans\_set = set(self.dataset["Ans"])

return len(ans\_set)

def getMaxOccur(self): # returns a value with maximum occurances of answer

d = {self.dataset["Ans"].count(x):x for x in self.dataset["Ans"]}

keys = list(d.keys())

keys.sort(reverse=True)

if len(keys) >1 and keys[0] != keys[1]:

return d[keys[0]]

else:

return None

def copy(self): # creates a copy of dataset

newdataset = dict()

keys = list(self.dataset.keys())

newdataset = dict.fromkeys(keys,0)

for key in keys :

newdataset[key] = self.dataset[key].copy()

return DataSet(newdataset)

def maxInfoGain(self): #finds a feature with max info gain

features = self.dataset["Features"]

if len(features) >=0 :

maxinfogain = self.infoGain(features[0])

maxfeat = features[0]

i=0

while i<len(features):

ig = self.infoGain(features[i])

if ig>=maxinfogain :

maxinfogain = ig

maxfeat = features[i]

i = i + 1

return maxfeat

else:

return None

def infoGain(self, feat) : # finds info gain of a particular feature

if feat in self.dataset.keys() :

featlist1 = self.dataset[feat]

total = len(featlist1)

featset = set(featlist1)

d = dict.fromkeys(featset, 0)

for item in featlist1 :

d[item] = d[item]+1

branches = self.splitOnFeature(feat)

gain = 0

for key in branches :

dsobj = branches[key]

gain = gain + ( (d[key] / total) \*( dsobj.getEntropy()) )

return self.getEntropy() - gain

else :

print("feature does not exists")

return None

def getEntropy(self) : # calculates entropy

list1 = self.dataset["Ans"]

total = len(list1)

aset = set(list1)

d = dict.fromkeys(aset, 0)

for item in list1 :

d[item] = d[item]+1

ent = 0

for k in aset :

x = d[k]/total

ent = ent + (x \* math.log(x,2))

return -ent

# this function splits original dataset on a feature and creates dictionary whose key values are

#values of the feature with a value as new dataset with that feature removed.

def splitOnFeature(self, feat):

if feat in self.dataset["Features"] :

ans\_set = set(self.dataset[feat])

newfeatures = self.dataset["Features"].copy()

newfeatures.remove(feat)

#create empty replica of dataset without the feature

keys = list(self.dataset.keys())

newdataset = dict()

for akey in keys :

newdataset[akey] = list()

newdataset["Features"] = newfeatures.copy()

newdataset.pop(feat)

branches = dict()

for akey in ans\_set:

branches[akey] = dict()

for key in list(newdataset.keys()) :

branches[akey][key] = newdataset[key].copy()

#copy data from original dataset to new datasets

i=-1

for featval in self.dataset[feat]:

i=i+1

#print("i=", i)

if featval in list(branches.keys()) :

branches[featval]["Ans"].append(self.dataset["Ans"][i])

for nfeat in newfeatures :

branches[featval][nfeat].append(self.dataset[nfeat][i])

for key in branches :

branches[key] = DataSet(branches[key])

return branches

else:

print(feat , " feature is not available")

return None

# main function that calculates the answer

def calculateAns(dsobj, feature, maxoccur, descr ):

branches = dsobj.splitOnFeature(feature)

for key in list(branches.keys()):

newdsobj = branches[key]

for key in list(branches.keys()):

newdsobj = branches[key]

if (newdsobj.uniqueAns() == 1):

print("Answer for " , descr+"-" +feature , " with value =", key , " is :", newdsobj.dataset['Ans'][0])

elif(newdsobj.uniqueAns() == 0):

print("in zero")

print("Answer for " , descr+"-" +feature , " with value =", key , " is :", maxoccur)

elif(newdsobj.uniqueAns() >1 and len(newdsobj.dataset["Features"]) ==0 ) :

print("Answer for " , descr+"-" +feature , " with value =", key , " is :", maxoccur)

else:

newfeat = newdsobj.maxInfoGain()

newmaxoccur = newdsobj.getMaxOccur()

if(newmaxoccur == None) :

newmaxoccur = maxoccur

calculateAns(newdsobj, newfeat, newmaxoccur , descr + ":" + feature +":->" + key +" " )

'''dataset = {

"Ans" :["Mammal", "Mammal", "Reptile", "Mammal", "Mammal", "Mammal", "Reptile", "Reptile", "Mammal", "Reptile"],

"Features":["toothed", "breaths", "legs"],

"toothed" : ["T", "T", "T", "F", "T", "T", "T", "T", "T", "F"],

"breaths" : ["T", "T", "T", "T", "T","T", "F", "T", "T", "T"],

"legs":["T", "T", "F", "T", "T","T", "F", "F", "T", "T"]

}'''

dataset = {

"Ans" :["Wait", "Wait", "Leave", "Wait", "Wait", "Wait", "Leave", "Leave", "Wait", "Leave"],

"Features":["Reservation", "Raining", "BadService"],

"Reservation" : ["T", "T", "T", "F", "T", "T", "T", "T", "T", "F"],

"Raining" : ["T", "F", "T", "T", "T","T", "F", "T", "T", "F"],

"BadService":["F", "F", "T", "F", "F","F", "T", "T", "F", "F"]

}

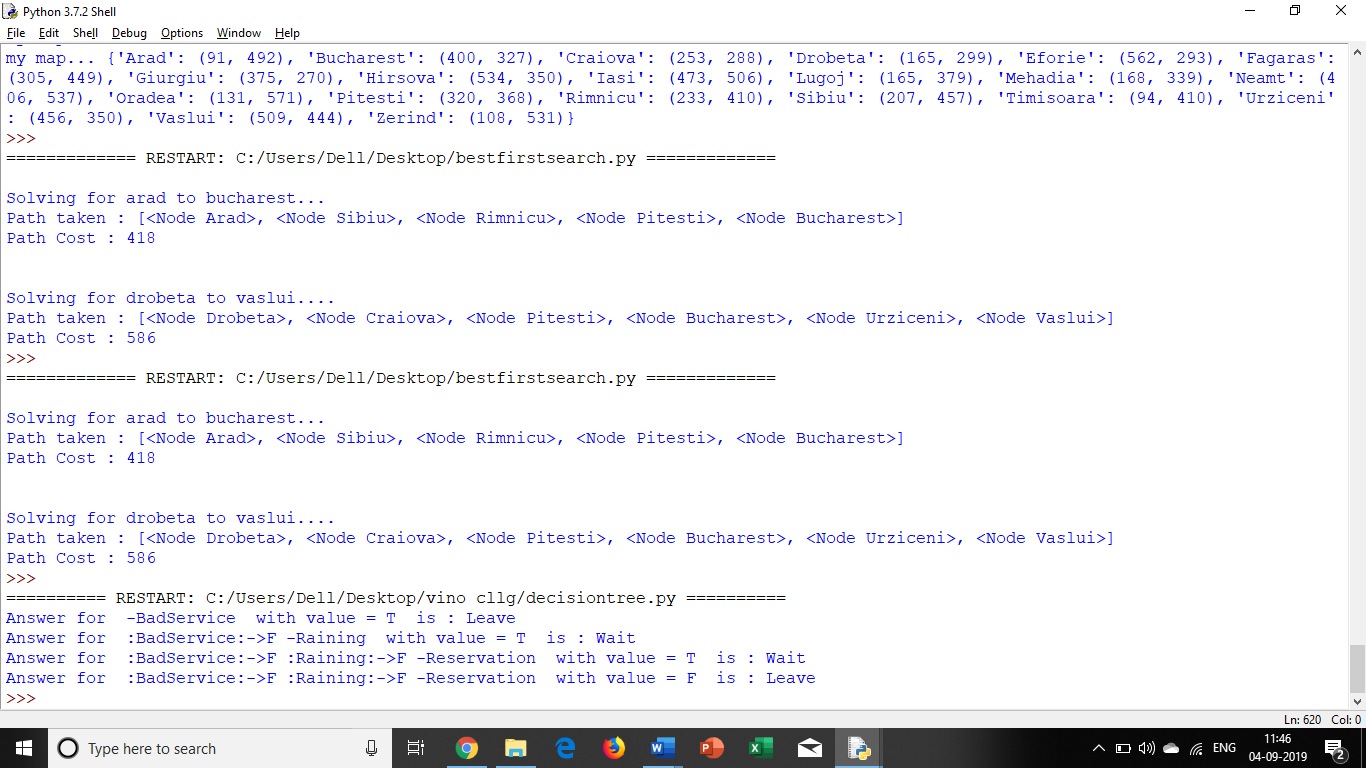
d1 = DataSet(dataset)

if d1.uniqueAns() != 1 :

feat = d1.maxInfoGain()

calculateAns(d1, feat, d1.getMaxOccur(), "")

**Output:**



**PRACTICAL NO 6**

**\*Implement feed forward back propagation neural network learning algorithm for the restaurant waiting problem.**

**Code:**

class Perceptron : # With 2 inputs and 1 output

def \_\_init\_\_(self, a,b, c, tval):

self.x = a # input vector

self.result = b # activation result

self.cresult = c # summation result

self.threshold = tval # threshold value used by activation function

self.w = []

def h(self, tw): # calculating summation(hypothesis function)

hresult= []

for i in range(0 , len(self.result)):

hresult.append(0)

#print("index - ", i, ";", hresult)

for j in range(0,len(tw)):

#print("i=",i, ",j=",j)

hresult[i] = hresult[i] + ( tw[j][i]\*self.x[j][i] )

return hresult

def checkthreshold(self, hresult): # applying activation function on summation result using threshold value

#flag = True

actfun =[]

for i in range(0 , len(self.result)) :

if (hresult[i] <= self.threshold ):

actfun.append(0)

else :

actfun.append( 1)

print("Ans :", hresult)

print("result of act fun:", actfun)

for i in range(0 , len(self.x)) :

if (actfun[i] != self.result[i]) :

return False

return True

def training(self, tw, alpha): #passing w vector and alpha value

i=1

while i<=100 : # Max 100 attempts

print("Attempt :", i)

hresult = self.h(tw)

if(self.checkthreshold(hresult)) : #if training result matches the test result

self.w = tw

print("In Attempt number ", i, ", i got it! I think i have learnt enough. Your w's are --" )

for x in range(0,len(self.w)):

print("w", x, " --> ", self.w[x])

break

i = i +1

# Changing values of w to reduce error/loss using batch gradient descent learning rule given on page 721 eqn 18.6

for j in range(0,len(self.result)) :

for k in range(0, len(tw)):

sum = 0

for n in range(0, len(tw)):

sum = sum + (self.cresult[j] - hresult[j]) \*self.x[n][j]

tw[k][j] = tw[k][j] + alpha\*sum

if(i>=100):

print("I am exhausted, tried 100 iterations! plz change something else...")

a = [ [1,1,1,1], [0,0,1,1] , [0,1,0,1] ] # x vector, x0 is dummy

b = [0,1,1,1] # result of activation function

c = [0.5, 0.7, 1.3, 1.5] # sample h values

p = Perceptron(a,b,c, 0.5) # threshold = 0.5

print("Whether reservation is done =", p.x[0])

print("Whether raining outside =", p.x[1])

print("with threshold value :", p.threshold)

r = p.h([ [0.5,0.5,0.5,0.5], [0.8, 0.8, 0.8, 0.8], [0.2, 0.2, 0.2, 0.2]])

print("status :", p.checkthreshold(r))

print("Example 1 -->") #with alpha as 0.01, you will not get result

p.training( [ [0.7,0.7,0.7,0.7], [0.5, 0.5, 0.5, 0.5], [0.4, 0.4, 0.4, 0.4]], 0.01)

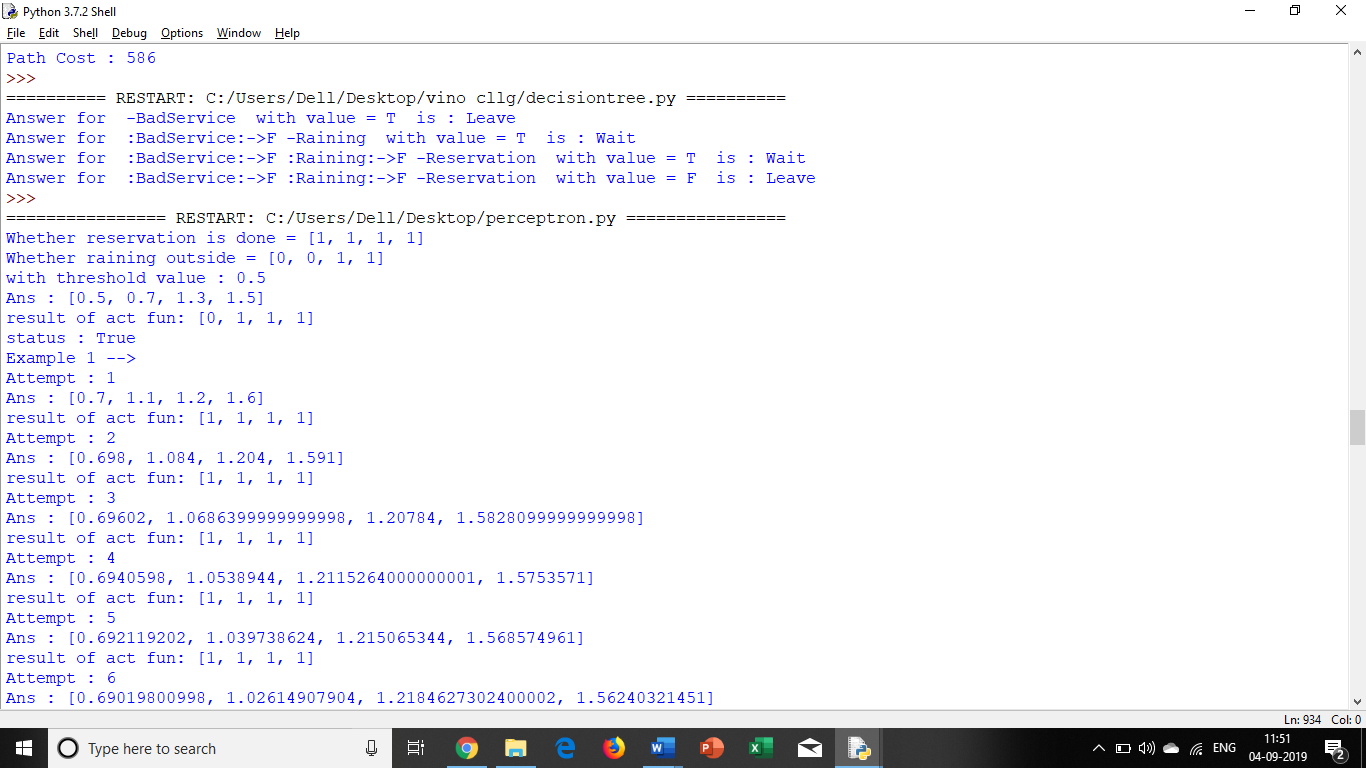
print("Example 2 -->") #with alpha as 0.5, you will not get result

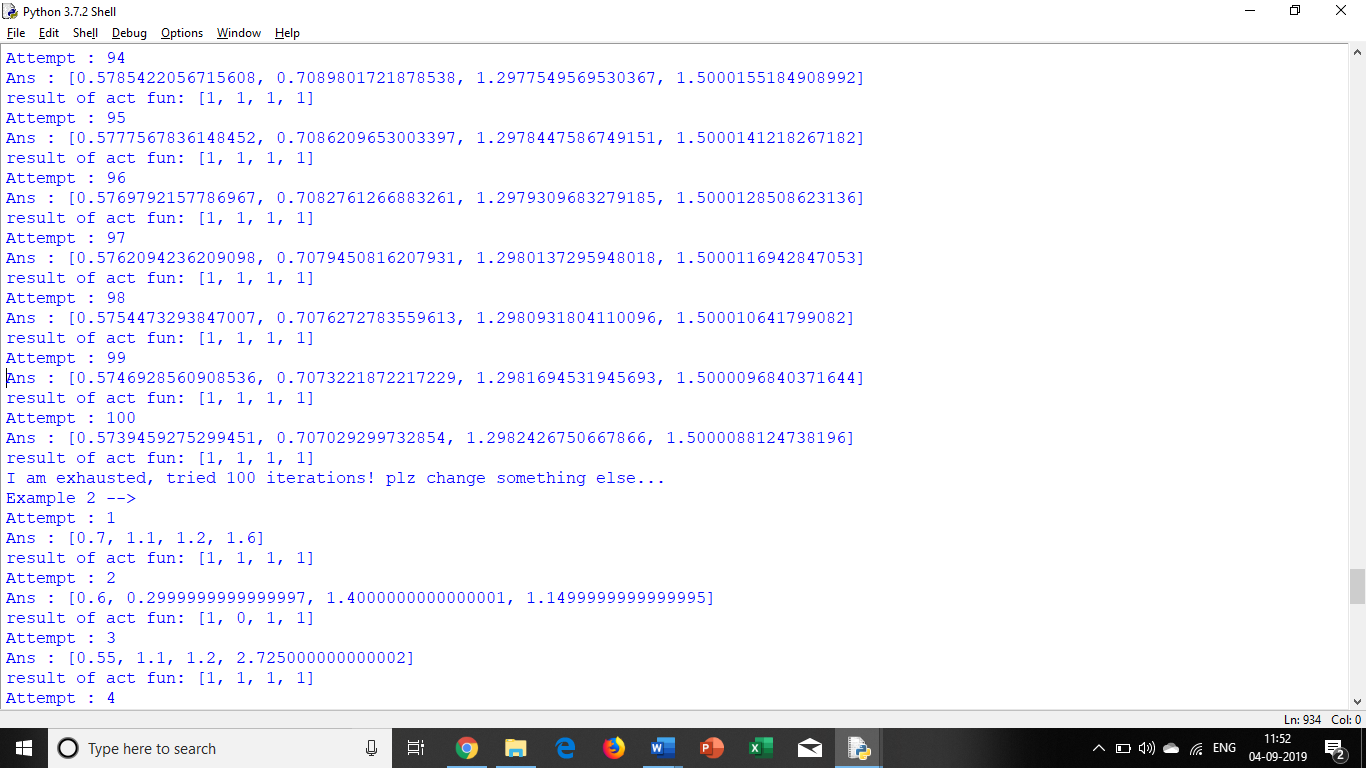
p.training( [ [0.7,0.7,0.7,0.7], [0.5, 0.5, 0.5, 0.5], [0.4, 0.4, 0.4, 0.4]], 0.5)

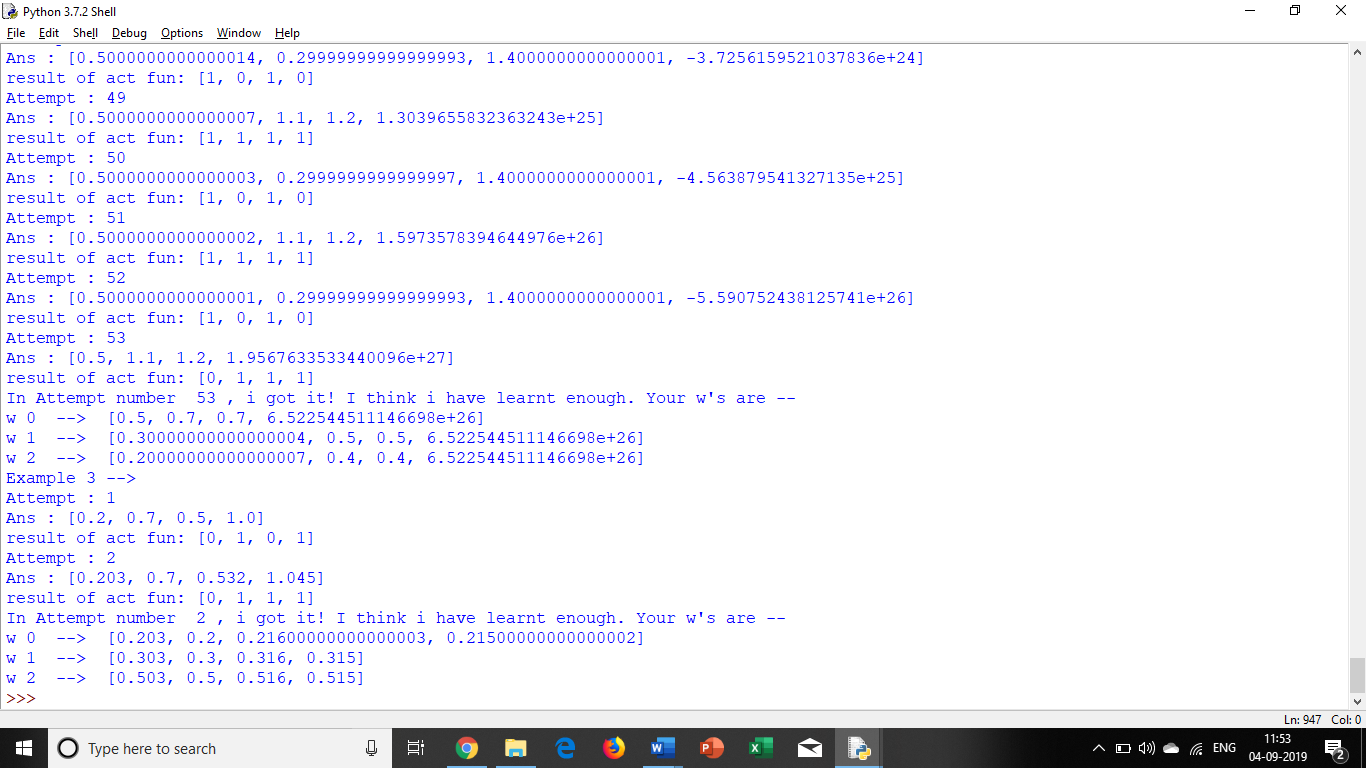
print("Example 3 -->")

p.training( [ [0.2,0.2,0.2,0.2], [0.3, 0.3, 0.3, 0.3], [0.5, 0.5, 0.5, 0.5]], 0.01)

**Output:**







Practical 7

Implement Adaboost ensemble learning algorithm for the restaurant waiting problem

# https://www.analyticsindiamag.com/introduction-to-boosting-implementing-adaboost-in-python/

import pandas as pd

data = pd.read\_csv("apples\_and\_oranges.csv")

pd.options.mode.chained\_assignment = None # default='warn'

print(data)

from sklearn.model\_selection import train\_test\_split

training\_set, test\_set = train\_test\_split(data, test\_size = 0.2,

random\_state = 1)

# first parameter represents data

# second parameter represents size whose value is between 0 to 1,

#how much test data you want

# third parameter represents seed(start point of random number generator)

X\_train = training\_set.iloc[:,0:2].values #column 0 & 1 of all rows

Y\_train = training\_set.iloc[:,2].values #column 2 of all rows

X\_test = test\_set.iloc[:,0:2].values

Y\_test = test\_set.iloc[:,2].values

from sklearn.ensemble import AdaBoostClassifier

adaboost = AdaBoostClassifier(n\_estimators=100, base\_estimator= None,

learning\_rate=1, random\_state = 1)

'''n\_estimators : integer, optional (default=50)

The maximum number of estimators at which boosting is terminated.

In case of perfect fit, the learning procedure is stopped early.

base\_estimator : object, optional (default=None)

The base estimator from which the boosted ensemble is built.

Support for sample weighting is required,

as well as proper classes\_ and n\_classes\_ attributes.

If None, then the base estimator is DecisionTreeClassifier(max\_depth=1)

learning\_rate : float, optional (default=1.)

Learning rate shrinks the contribution of each classifier by learning\_rate.

There is a trade-off between learning\_rate and n\_estimators.

random\_state : int, RandomState instance or None, optional (default=None)

If int, random\_state is the seed used by the random number generator; If RandomState instance, random\_state is the random number generator;

If None, the random number generator is the RandomState instance used by np.random.

'''

adaboost.fit(X\_train,Y\_train)

# Build a boosted classifier from the training set (X, y).

Y\_pred = adaboost.predict(X\_test)

test\_set["Predictions"] = Y\_pred

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(Y\_test,Y\_pred)

'''

Compute confusion matrix to evaluate the accuracy of a classification

By definition a confusion matrix is such that is equal to the

number of observations

known to be in group but predicted to be in group .

Thus in binary classification, the count of true negatives is ,

false negatives is , true positives is and false positives

'''

accuracy = float(cm.diagonal().sum())/len(Y\_test)

print("\nAccuracy Of AdaBoost For The Given Dataset : ", accuracy)

'''

# Load libraries

from sklearn.ensemble import AdaBoostClassifier

from sklearn import datasets

print("j")

# Import train\_test\_split function

from sklearn.model\_selection import train\_test\_split

#Import scikit-learn metrics module for accuracy calculation

from sklearn import metrics

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Split dataset into training set and test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

abc = AdaBoostClassifier(n\_estimators=50,

learning\_rate=1)

# Train Adaboost Classifer

model = abc.fit(X\_train, y\_train)

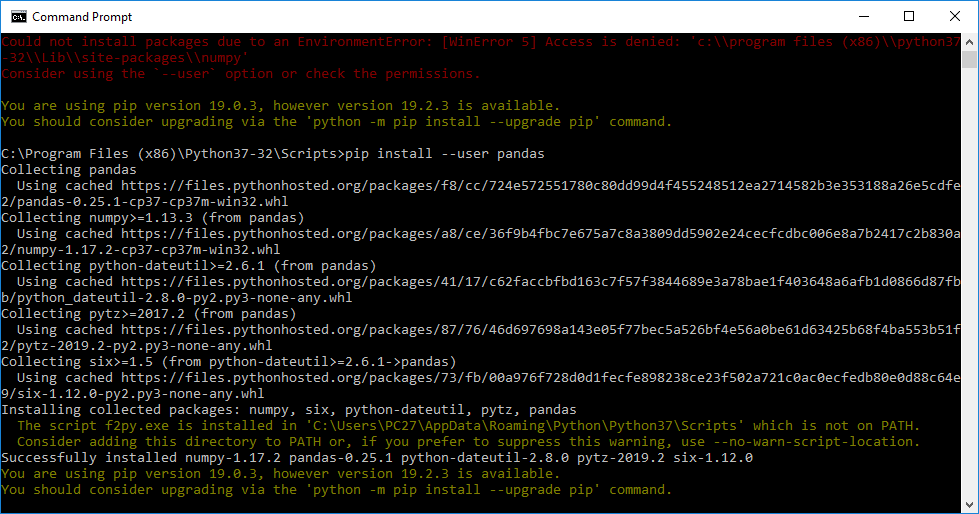
#Predict the response for test dataset

y\_pred = model.predict(X\_test)

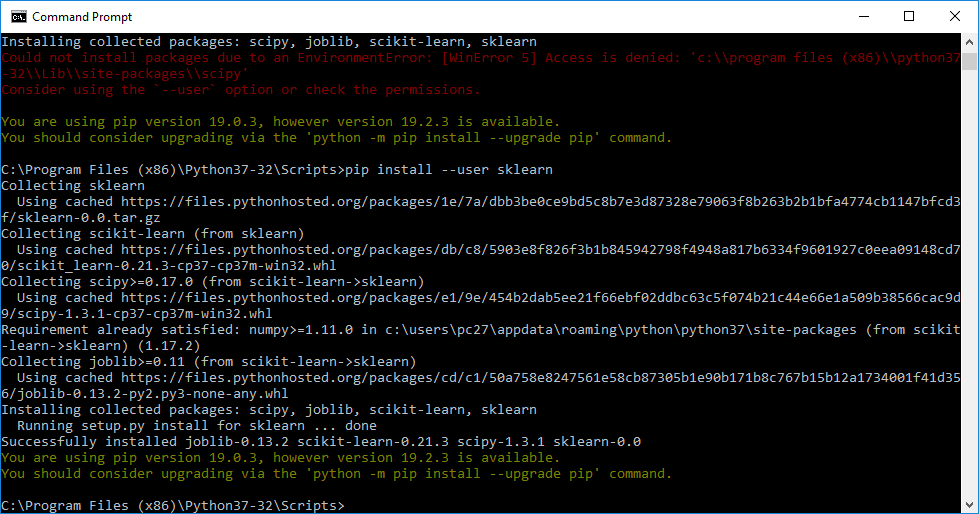
print("Accuracy:",metrics.accuracy\_score(y\_test, y\_pred))

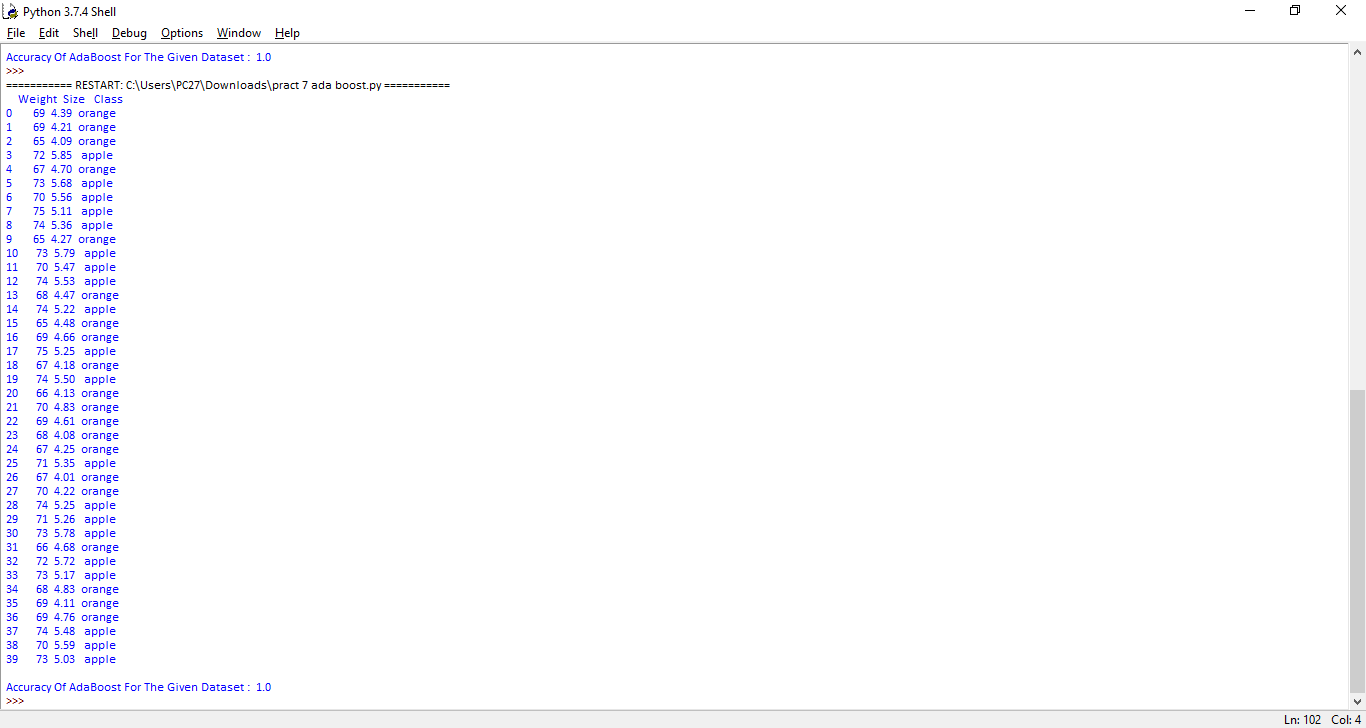
'''

Install pandas



Install sklearn





Practical 8

Implement Naive Bayes’ learning algorithm for the restaurant waiting problem

# https://www.geeksforgeeks.org/naive-bayes-classifiers/

class NaiveBayes:

def \_\_init\_\_(self, f, r):

self.features = f

self.response = r

def predict(self,custcase):

anskeys = list(self.response.keys())

ansvalues = dict.fromkeys(anskeys,0)

print('\n chk ',ansvalues)

for key in anskeys :

ansvalues[key] = self.response[key]

for ikey, ival in custcase.items() :

ansvalues[key] = ansvalues[key] \* self.features[ikey][ival][key]

print(ansvalues)

#calculating MAP

maxkey=""

maxans=-1

for ikey, ival in ansvalues.items():

if ival > maxans :

maxans= ival

maxkey = ikey

return maxkey

#precalculated values from worksheet - "naive bayes classifier working"

response = {"Wait":0.4, "Leave":0.6}

features = {

"Reservation":

{

"Yes" : {"Wait":0.5, "Leave":0.666667},

"No" : {"Wait":0.5, "Leave":0.333333}

} ,

"Time>30":

{

"Yes" : {"Wait":0.25, "Leave":0.83333},

"No" : {"Wait":0.75, "Leave":0.16667}

}

}

nb = NaiveBayes(features, response)

#print("Probability :", nb.features["Reservation"]["Yes"]["Wait"])

#print("Probability :", nb.features["Time>30"]["No"]["Leave"])

resstatus = input("Manager asks Customer, have you reserved table?(Yes/No):")

timestatus = input("Customer asks Manager, Will it take more than 30 mins?(Yes/No):")

custcase = {"Reservation":resstatus, "Time>30":timestatus}

print("Manager predicts that Customer will :" , nb.predict(custcase) )

