Deccan Education Society's
Kirti M. Doongursee College of Arts, Science and Commerce
[NAAC Accredited: "A Grade"]

T.Y.B.Sc. [Computer Science]

Practical Journal

USCS501

Seat Number [ ]

Department of Computer Science and Information Technology

Department of Computer Science and Information Technology
Deccan Education Society's
Kirti M. Doongursee College of Arts, Science and Commerce
[NAAC Accredited: "A Grade"]

C E R T I F I C A T E

This is to certify that Mr. / Miss ____

of T.Y.B.Sc. (Computer Science) with Seat No.________ has completed ________

Practicals of Paper- USCS501 under my supervision in this College during the

year 2022-2023.

Lecturer-In-Charge

Date: / /2022

H.O.D.
Department of
Computer Science & IT

Date:

Examined by:

Date:

Remarks:

____

____

USCS501: Artificial Intelligence				
Sr. No.	**Date**	**Title**	**Page No.**	**Signature**
1		Implement Breadth first search algorithm for Romanian map problem.		
2		Implement Depth first search for Romanian map problem.		
3		Implement Iterative deepening depth first search for Romanian map problem.		
4		Implement a simple tree algorithm for Romanian map problem.		
5		Logistic Regression		
6		Implement decision tree learning algorithm for the restaurant waiting problem.		
7		Implement Naïve Bayes learning algorithm for the restaurant waiting problem. rendering		
8		Implement feed forward back propagation neural network learning algorithm for the restaurant waiting problem.		
9		Ensemble		

CODE: -

```
#Implement Breadth first search 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): # 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 is
        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 result(self, state, action):
        raise NotImplementedError

    def goal_test(self, state):
        return state == self.goal
```

```
    def path_cost(self, c, state1, action, state2):
        return c + 1

    def value(self, state):
        raise NotImplementedError


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

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or infinity)


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 nodes(self):
        nodelist = list()
        for key in self.graph_dict.keys() :
            nodelist.append(key)
        return nodelist

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("------")
print("Children of Arad ", romania_map.get('Arad'))
print("distance from arad to sibiu = ",romania_map.get('Arad','Sibiu'))

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: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac4.py
processing node ... Arad
--> Arad  connects  Zerind  by distance : 75
--> Arad  connects  Sibiu  by distance : 140
--> Arad  connects  Timisoara  by distance : 118
processing node ... Bucharest
--> Bucharest  connects  Urziceni  by distance : 85
--> Bucharest  connects  Pitesti  by distance : 101
--> Bucharest  connects  Giurgiu  by distance : 90
--> Bucharest  connects  Fagaras  by distance : 211
processing node ... Craiova
--> Craiova  connects  Drobeta  by distance : 120
--> Craiova  connects  Rimnicu  by distance : 146
--> Craiova  connects  Pitesti  by distance : 138
processing node ... Drobeta
--> Drobeta  connects  Mehadia  by distance : 75
--> Drobeta  connects  Craiova  by distance : 120
processing node ... Eforie
--> Eforie  connects  Hirsova  by distance : 86
processing node ... Fagaras
--> Fagaras  connects  Sibiu  by distance : 99
--> Fagaras  connects  Bucharest  by distance : 211
processing node ... Hirsova
--> Hirsova  connects  Urziceni  by distance : 98
--> Hirsova  connects  Eforie  by distance : 86
processing node ... Iasi
--> Iasi  connects  Vaslui  by distance : 92
--> Iasi  connects  Neamt  by distance : 87
processing node ... Lugoj
--> Lugoj  connects  Timisoara  by distance : 111
--> Lugoj  connects  Mehadia  by distance : 70
processing node ... Oradea
--> Oradea  connects  Zerind  by distance : 71
--> Oradea  connects  Sibiu  by distance : 151
processing node ... Pitesti
--> Pitesti  connects  Rimnicu  by distance : 97
--> Pitesti  connects  Bucharest  by distance : 101
--> Pitesti  connects  Craiova  by distance : 138
processing node ... Rimnicu
--> Rimnicu  connects  Sibiu  by distance : 80
--> Rimnicu  connects  Craiova  by distance : 146
--> Rimnicu  connects  Pitesti  by distance : 97
processing node ... Urziceni
--> Urziceni  connects  Vaslui  by distance : 142
--> Urziceni  connects  Bucharest  by distance : 85
--> Urziceni  connects  Hirsova  by distance : 98
processing node ... Zerind
--> Zerind  connects  Arad  by distance : 75
--> Zerind  connects  Oradea  by distance : 71
processing node ... Sibiu
--> Sibiu  connects  Arad  by distance : 140
--> Sibiu  connects  Fagaras  by distance : 99
--> Sibiu  connects  Oradea  by distance : 151
--> Sibiu  connects  Rimnicu  by distance : 80
processing node ... Timisoara
--> Timisoara  connects  Arad  by distance : 118
--> Timisoara  connects  Lugoj  by distance : 111
processing node ... Giurgiu
--> Giurgiu  connects  Bucharest  by distance : 90
processing node ... Mehadia
--> Mehadia  connects  Drobeta  by distance : 75
--> Mehadia  connects  Lugoj  by distance : 70
```

```
--> Vaslui  connects  Urziceni  by distance : 142
processing node ... Neamt
--> Neamt  connects  Iasi  by distance : 87
after construcing grpah -
{'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, 'Craiov
a': 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}, 'Nea
mt': {'Iasi': 87}}
-------
Children of Arad  {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118}
distance from arad to sibiu =  140
=============== BFS Algo ====================
Keys of Arad  ['Zerind', 'Sibiu', 'Timisoara']
Search begins from :  deque([<Node Arad>])
Now exploring... <Node Arad>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>, <Node Timisoara>]
Now exploring... <Node Zerind>
Expanded Nodes : [<Node Arad>, <Node Oradea>]
Now exploring... <Node Sibiu>
Expanded Nodes : [<Node Arad>, <Node Fagaras>, <Node Oradea>, <Node Rimnicu>]
Now exploring... <Node Timisoara>
Expanded Nodes : [<Node Arad>, <Node Lugoj>]
Now exploring... <Node Arad>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>, <Node Timisoara>]
Now exploring... <Node Oradea>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>]
Now exploring... <Node Arad>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>, <Node Timisoara>]
Now exploring... <Node Fagaras>
Expanded Nodes : [<Node Sibiu>, <Node Bucharest>]
Now exploring... <Node Oradea>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>]
Now exploring... <Node Rimnicu>
Expanded Nodes : [<Node Sibiu>, <Node Craiova>, <Node Pitesti>]
Now exploring... <Node Arad>
Expanded Nodes : [<Node Zerind>, <Node Sibiu>, <Node Timisoara>]
Now exploring... <Node Lugoj>
Expanded Nodes : [<Node Timisoara>, <Node Mehadia>]
Now exploring... <Node Zerind>
Expanded Nodes : [<Node Arad>, <Node Oradea>]
Now exploring... <Node Sibiu>
Expanded Nodes : [<Node Arad>, <Node Fagaras>, <Node Oradea>, <Node Rimnicu>]
Now exploring... <Node Timisoara>
Expanded Nodes : [<Node Arad>, <Node Lugoj>]
Now exploring... <Node Zerind>
Expanded Nodes : [<Node Arad>, <Node Oradea>]
Now exploring... <Node Sibiu>
Expanded Nodes : [<Node Arad>, <Node Fagaras>, <Node Oradea>, <Node Rimnicu>]
Now exploring... <Node Zerind>
Expanded Nodes : [<Node Arad>, <Node Oradea>]
Now exploring... <Node Sibiu>
Expanded Nodes : [<Node Arad>, <Node Fagaras>, <Node Oradea>, <Node Rimnicu>]
Now exploring... <Node Timisoara>
Expanded Nodes : [<Node Arad>, <Node Lugoj>]
Now exploring... <Node Sibiu>
Expanded Nodes : [<Node Arad>, <Node Fagaras>, <Node Oradea>, <Node Rimnicu>]
Now exploring... <Node Bucharest>
solution of  Arad  to  Bucharest ['Arad', 'Sibiu', 'Fagaras', 'Bucharest']
path cost of final node = 450
```

CODE: -

```
# Implement depth first search for Romanian map problem.
graph = {'Arad': ['Zerind', 'Timisoara', 'Sibiu'],
        'Bucharest': ['Urziceni','Pitesti', 'Giurgiu','Fagaras'],
        'Craiova': ['Dobreta', 'Rimnicu Vilcea', 'Pitesti'],
        'Dobreta': ['Mehadia'],
        'Eforie': ['Hirsova'],
        'Iasai': ['Vaslui','Neamt'],
        'Lugoj': ['Timisoara','Mehadia'],
        'Oradea': ['Zerind','Sibiu'],
        'Pitesti': ['Rimnicu Vilcea','Bucharest','Craiova'],
        'Urziceni': ['Vaslui'],
        'Zerind': ['Oradea','Arad'],
        'Sibiu': ['Oradea','Arad','Rimnicu Vilcea','Fagaras'],
        'Timisoara': ['Arad','Lugoj'],
        'Mehadia': ['Lugoj','Dobreta'],
        'Rimnicu Vilcea': ['Sibiu','Pitesti','Craiova'],
        'Fagaras': ['Sibiu','Bucharest'],
        'Giurgiu': ['Bucharest'],
        'Vaslui': ['Urziceni','Iasai'],
        'Neamt': ['Iasai']}
pc = {('Arad','Zerind'):75,
        ('Arad','Timisoara'):118,
        ('Arad','Sibiu'):140,
        ('Zerind','Oradea'):71,
        ('Zerind','Arad'):75,
        ('Timisoara','Arad'):118,
        ('Timisoara','Lugoj'):111,
        ('Sibiu','Arad'):140,
        ('Sibiu','Rimnicu Vilcea'):80,
        ('Sibiu','Fagaras'):99,
        ('Sibiu','Oradea'):151,
        ('Oradea','Zerind'):71,
        ('Oradea','Sibiu'):151,
        ('Lugoj','Timisoara'):111,
        ('Lugoj','Mehadia'):70,
        ('Rimnicu Vilcea','Sibiu'):80,
        ('Rimnicu Vilcea','Pitesti'):97,
        ('Rimnicu Vilcea','Craiova'):146,
        ('Fagaras','Sibiu'):99,
        ('Fagaras','Bucharest'):211,
        ('Mehadia','Lugoj'):70,
        ('Mehadia','Dobreta'):75,
        ('Pitesti','Rimnicu Vilcea'):97,
        ('Pitesti','Bucharest'):101,
        ('Pitesti','Craiova'):138,
        ('Craiova','Rimnicu Vilcea'):146,
        ('Craiova','Dobreta'):120,
        ('Craiova','Pitesti'):138,
        ('Bucharest','Fagaras'):211,
        ('Bucharest','Bucharest'):0,
        ('Bucharest','Pitesti'):101,
        ('Bucharest','Giurgiu'):90,
        ('Bucharest','Urziceni'):85,
        }
```

```
locs={'Arad': 366,
        'Bucharest': 0,
        'Craiova': 160,
        'Dobreta': 242,
        'Eforie': 161,
        'Iasai': 226,
        'Lugoj': 244,
        'Oradea': 380,
        'Pitesti': 100,
        'Urziceni': 80,
        'Zerind': 374,
        'Sibiu': 253,
        'Timisoara': 329,
        'Mehadia': 241,
        'Rimnicu Vilcea': 193,
        'Fagaras':176,
        'Giurgiu': 77,
        'Vaslui': 199,
        'Neamt': 234
    }

def DFS(g, v, goal, explored, path_so_far,m):
    """ Returns path from v to goal in g as a string (Hack) """
    explored.add(v)
    node=[]
    if v == goal:
        return path_so_far + v
    for w in g[v]:
        if w not in explored:
            f=locs.get(w)+pc.get((v,w))
            if m>f:
                m=f
                print("%i%s%s" %(m,v,w))
                node=w
    p = DFS(g, node, goal, explored, path_so_far + v+'->',m)
    if p:
        return p
    return ""


print(DFS(graph, 'Arad', 'Bucharest', set(), "",1000))
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac2.py
449AradZerind
447AradTimisoara
393AradSibiu
273SibiuRimnicu Vilcea
197Rimnicu VilceaPitesti
101PitestiBucharest
Arad->Sibiu->Rimnicu Vilcea->Pitesti->Bucharest
```

CODE: -

```
# Implement a simple tree algorithm for Romanian map problem.
dict_hn={'Arad':336,'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,'Urziceni':80,'Vaslui':199,'Zerind':374}

dict_gn=dict(
Arad=dict(Zerind=75,Timisoara=118,Sibiu=140),
Bucharest=dict(Urziceni=85,Giurgiu=90,Pitesti=101,Fagaras=211),
Craiova=dict(Drobeta=120,Pitesti=138,Rimnicu=146),
Drobeta=dict(Mehadia=75,Craiova=120),
Eforie=dict(Hirsova=86),
Fagaras=dict(Sibiu=99,Bucharest=211),
Giurgiu=dict(Bucharest=90),
Hirsova=dict(Eforie=86,Urziceni=98),
Iasi=dict(Neamt=87,Vaslui=92),
Lugoj=dict(Mehadia=70,Timisoara=111),
Mehadia=dict(Lugoj=70,Drobeta=75),
Neamt=dict(Iasi=87),
Oradea=dict(Zerind=71,Sibiu=151),
Pitesti=dict(Rimnicu=97,Bucharest=101,Craiova=138),
Rimnicu=dict(Sibiu=80,Pitesti=97,Craiova=146),
Sibiu=dict(Rimnicu=80,Fagaras=99,Arad=140,Oradea=151),
Timisoara=dict(Lugoj=111,Arad=118),
Urziceni=dict(Bucharest=85,Hirsova=98,Vaslui=142),
Vaslui=dict(Iasi=92,Urziceni=142),
Zerind=dict(Oradea=71,Arad=75)
)
import queue as Q
#from RMP import dict_hn

start='Arad'
goal='Bucharest'
result=''


def DLS(city, visitedstack, startlimit, endlimit):
    global result
    found=0
    result=result+city+' '
    visitedstack.append(city)
    if city==goal:
        return 1
    if startlimit==endlimit:
        return 0
    for eachcity in dict_gn[city].keys():
        if eachcity not in visitedstack:
            found=DLS(eachcity, visitedstack, startlimit+1, endlimit)
            if found:
                return found

def IDDFS(city, visitedstack, endlimit):
    global result
    for i in range(0, endlimit):
```

```
        print("Searching at Limit: ",i)
        found=DLS(city, visitedstack, 0, i)
        if found:
            print("Found")
            break
        else:
            print("Not Found! ")
            print(result)
            print("-----")
            result=' '
            visitedstack=[]

def main():
    visitedstack=[]
    IDDFS(start, visitedstack, 9)
    print("IDDFS Traversal from ",start," to ", goal," is: ")
    print(result)


main()
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac3.py
Searching at Limit:  0
Not Found!
Arad
-----
Searching at Limit:  1
Not Found!
 Arad Zerind Timisoara Sibiu
-----
Searching at Limit:  2
Not Found!
 Arad Zerind Oradea Timisoara Lugoj Sibiu Rimnicu Fagaras
-----
Searching at Limit:  3
Not Found!
 Arad Zerind Oradea Sibiu Timisoara Lugoj Mehadia
-----
Searching at Limit:  4
Not Found!
 Arad Zerind Oradea Sibiu Rimnicu Fagaras Timisoara Lugoj Mehadia Drobeta
-----
Searching at Limit:  5
Found
IDDFS Traversal from  Arad  to  Bucharest  is:
 Arad Zerind Oradea Sibiu Rimnicu Pitesti Craiova Fagaras Bucharest
```

CODE: -

```
#Implement a simple tree algorithm for Romanian map problem.
import random
openList=[['Arad']]
closedList=[]
nodeList=
{'Arad':['Sibiu','Timisora'],'Sibiu':['Arad','Timisora','Fagarus'],'Timisora':['Arad','Dorbeta'],'Dorbeta':['Timisora','Cr
aiova'],'Fagarus':['Sibiu','Bucharest'],'Bucharest':['Dorbeta','Fagarus']}

def goalTest(some_node):
    return some_node == 'Bucharest'

def moveGen(some_node):
    return nodeList[some_node]

def SS3():
    while len(openList)>0:
        random.shuffle(openList)
        print("Open list Contains", openList)
        seen = openList.pop(0)
        N = seen[0]
        closedList.append(N)
        print("Picked Node: ", N)
        if goalTest(N):
            print("Goal Found")
            print(seen)
            return
        else:
            neighbours=moveGen(N)
            print("Neighbours of ",N," are : ", neighbours)
            for node in neighbours:
                if(node not in openList) and (node not in closedList):
                    l=[node,seen]
                    openList.append(l)

    print("Goal Not Found")
SS3()
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac4.py
Open list Contains [['Arad']]
Picked Node:  Arad
Neighbours of  Arad  are :  ['Sibiu', 'Timisora']
Open list Contains [['Sibiu', ['Arad']], ['Timisora', ['Arad']]]
Picked Node:  Sibiu
Neighbours of  Sibiu  are :  ['Arad', 'Timisora', 'Fagarus']
Open list Contains [['Timisora', ['Arad']], ['Timisora', ['Sibiu', ['Arad']]], ['Fagarus', ['Sibiu', ['Arad']]]]
Picked Node:  Timisora
Neighbours of  Timisora  are :  ['Arad', 'Dorbeta']
Open list Contains [['Timisora', ['Sibiu', ['Arad']]], ['Dorbeta', ['Timisora', ['Arad']]], ['Fagarus', ['Sibiu', ['Arad']]]]
Picked Node:  Timisora
Neighbours of  Timisora  are :  ['Arad', 'Dorbeta']
Open list Contains [['Dorbeta', ['Timisora', ['Sibiu', ['Arad']]]], ['Dorbeta', ['Timisora', ['Arad']]], ['Fagarus', ['Sibiu', ['Arad']]]]
Picked Node:  Dorbeta
Neighbours of  Dorbeta  are :  ['Timisora', 'Craiova']
Open list Contains [['Craiova', ['Dorbeta', ['Timisora', ['Sibiu', ['Arad']]]]], ['Dorbeta', ['Timisora', ['Arad']]], ['Fagarus', ['Sibiu', ['Arad']]]]
Picked Node:  Craiova
Traceback (most recent call last):
  File "E:\Python\prac4.py", line 34, in <module>
    SS3()
  File "E:\Python\prac4.py", line 26, in SS3
    neighbours=moveGen(N)
  File "E:\Python\prac4.py", line 11, in moveGen
    return nodeList[some_node]
KeyError: 'Craiova'
```

CODE: -

```
# Logistic Regression

# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the datasets
datasets = pd.read_csv('restaurants.csv')
X = datasets.iloc[:, [2,3]].values
Y = datasets.iloc[:, 4].values

# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_Train, X_Test, Y_Train, Y_Test = train_test_split(X, Y, test_size =
0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X_Train = sc_X.fit_transform(X_Train)
X_Test = sc_X.transform(X_Test)

# Fitting the Logistic Regression into the Training set
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(X_Train, Y_Train)

# Predicting the test set results
Y_Pred = classifier.predict(X_Test)

# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(Y_Test, Y_Pred)




# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_Set, Y_Set = X_Train, Y_Train
X1, X2 = np.meshgrid(np.arange(start = X_Set[:,0].min() -1, stop =
X_Set[:, 0].max() +1, step = 0.01),
                     np.arange(start = X_Set[:,1].min() -1, stop =
X_Set[:, 1].max() +1, step = 0.01))

plt.contourf(X1,X2, classifier.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape),
```

```
                alpha = 0.75, cmap = ListedColormap(('red', 'green')))

plt.xlim(X1.min(), X2.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(Y_Set)):
    plt.scatter(X_Set[Y_Set == j, 0], X_Set[Y_Set == j,1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Logistic Regression ( Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

OUTPUT: -

CODE: -

```
#Implement decision tree learning algorithm for the restaurant waiting problem.
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import pandas as pd

dataset = pd.read_csv('restaurants.csv')


train_features = dataset.iloc[:80,:-1]

test_features = dataset.iloc[80:,:-1]


train_targets = dataset.iloc[:80,-1]
test_targets = dataset.iloc[80:,-1]



tree = DecisionTreeClassifier(criterion = 'entropy').fit(train_features,train_targets)


prediction = tree.predict(test_features)


print("The prediction accuracy is: ",tree.score(test_features,test_targets)*100,"%")
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac6.py
The prediction accuracy is:  95.0 %
```

CODE: -

```
#Implement Naïve Bayes learning algorithm for the restaurant waiting problem. rendering
# Gaussian Naive Bayes
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
# load the iris datasets
#Import the dataset
import pandas as pd
dataset = pd.read_csv('restaurants.csv')

#dataset = datasets.load_iris()
# fit a Naive Bayes model to the data
model = GaussianNB()
model.fit(dataset.iloc[:90,0:8],dataset.iloc[:90,-1])
#model.fit(dataset.data, dataset.target)
print(model)
# make predictions
#expected = dataset.target
#predicted = model.predict(dataset.data)

expected = dataset.iloc[:90,-1]
predicted = model.predict(dataset.iloc[:90,0:8])
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac7.py
GaussianNB()
              precision    recall  f1-score   support

           0       0.65      1.00      0.78        20
           1       1.00      0.84      0.91        70

    accuracy                           0.88        90
   macro avg       0.82      0.92      0.85        90
weighted avg       0.92      0.88      0.89        90

[[20  0]
 [11 59]]
```

CODE: -

```
# Implement feed forward back propagation neural network learning
algorithm for the restaurant waiting problem.
from random import seed
from random import randrange
from random import random
from csv import reader
from math import exp

# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
	minmax = list()
	stats = [[min(column), max(column)] for column in zip(*dataset)]
	return stats

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
	for row in dataset:
		for i in range(len(row)-1):
			row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] -
minmax[i][0])

# Split a dataset into k folds
```

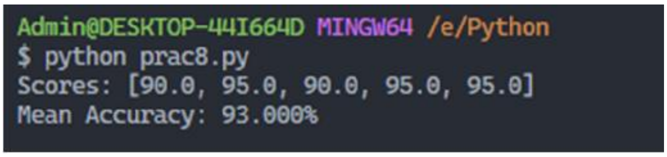
```
def cross_validation_split(dataset, n_folds):
     dataset_split = list()
     dataset_copy = list(dataset)
     fold_size = int(len(dataset) / n_folds)
     for i in range(n_folds):
          fold = list()
          while len(fold) < fold_size:
               index = randrange(len(dataset_copy))
               fold.append(dataset_copy.pop(index))
          dataset_split.append(fold)
     return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
     correct = 0
     for i in range(len(actual)):
          if actual[i] == predicted[i]:
               correct += 1
     return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
     folds = cross_validation_split(dataset, n_folds)
     scores = list()
     for fold in folds:
          train_set = list(folds)
          train_set.remove(fold)
          train_set = sum(train_set, [])
          test_set = list()
          for row in fold:
               row_copy = list(row)
               test_set.append(row_copy)
               row_copy[-1] = None
          predicted = algorithm(train_set, test_set, *args)
          actual = [row[-1] for row in fold]
          accuracy = accuracy_metric(actual, predicted)
          scores.append(accuracy)
     return scores

# Calculate neuron activation for an input
def activate(weights, inputs):
     activation = weights[-1]
     for i in range(len(weights)-1):
          activation += weights[i] * inputs[i]
     return activation

# Transfer neuron activation
def transfer(activation):
     return 1.0 / (1.0 + exp(-activation))
```

```
# Forward propagate input to a network output
def forward_propagate(network, row):
     inputs = row
     for layer in network:
          new_inputs = []
          for neuron in layer:
               activation = activate(neuron['weights'], inputs)
               neuron['output'] = transfer(activation)
               new_inputs.append(neuron['output'])
          inputs = new_inputs
     return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
     return output * (1.0 - output)


# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
     for i in reversed(range(len(network))):
          layer = network[i]
          errors = list()
          if i != len(network)-1:
               for j in range(len(layer)):
                    error = 0.0
                    for neuron in network[i + 1]:
                         error += (neuron['weights'][j] *
neuron['delta'])
                    errors.append(error)
          else:
               for j in range(len(layer)):
                    neuron = layer[j]
                    errors.append(expected[j] - neuron['output'])
          for j in range(len(layer)):
               neuron = layer[j]
               neuron['delta'] = errors[j] *
transfer_derivative(neuron['output'])

# Update network weights with error
def update_weights(network, row, l_rate):
     for i in range(len(network)):
          inputs = row[:-1]
          if i != 0:
               inputs = [neuron['output'] for neuron in network[i - 1]]
          for neuron in network[i]:
               for j in range(len(inputs)):
                    neuron['weights'][j] += l_rate * neuron['delta'] *
inputs[j]
               neuron['weights'][-1] += l_rate * neuron['delta']
```

```
# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):
     for epoch in range(n_epoch):
          for row in train:
               outputs = forward_propagate(network, row)
               expected = [0 for i in range(n_outputs)]
               expected[row[-1]] = 1
               backward_propagate_error(network, expected)
               update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
     network = list()
     hidden_layer = [{'weights':[random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
     network.append(hidden_layer)
     output_layer = [{'weights':[random() for i in range(n_hidden +
1)]} for i in range(n_outputs)]
     network.append(output_layer)
     return network

# Make a prediction with a network
def predict(network, row):
     outputs = forward_propagate(network, row)
     return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent
def back_propagation(train, test, l_rate, n_epoch, n_hidden):
     n_inputs = len(train[0]) - 1
     n_outputs = len(set([row[-1] for row in train]))
     network = initialize_network(n_inputs, n_hidden, n_outputs)
     train_network(network, train, l_rate, n_epoch, n_outputs)
     predictions = list()
     for row in test:
          prediction = predict(network, row)
          predictions.append(prediction)
     return(predictions)

# Test Backprop on Seeds dataset
seed(1)
# load and prepare data
filename = 'restaurants.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
     str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
# normalize input variables
minmax = dataset_minmax(dataset)
normalize_dataset(dataset, minmax)
```

```
# evaluate algorithm
n_folds = 5
l_rate = 0.3
n_epoch = 500
n_hidden = 5
scores = evaluate_algorithm(dataset, back_propagation, n_folds, l_rate,
n_epoch, n_hidden)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

OUTPUT: -



```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac8.py
Scores: [90.0, 95.0, 90.0, 95.0, 95.0]
Mean Accuracy: 93.000%
```

CODE: -

```
# Bagged Decision Trees for Classification
import pandas
from sklearn import model_selection
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
kfold = model_selection.KFold(n_splits=10, random_state=None)
cart = DecisionTreeClassifier()
num_trees = 100
model = BaggingClassifier(base_estimator=cart, n_estimators=num_trees,
random_state=None)
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

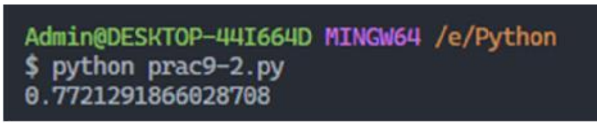
OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac9-1.py
0.766866028708134
```

CODE: -

```
# Random Forest Classification

import pandas
from sklearn import model_selection
from sklearn.ensemble import RandomForestClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
num_trees = 100
max_features = 3
kfold = model_selection.KFold(n_splits=10, random_state=None)
model = RandomForestClassifier(n_estimators=num_trees,
max_features=max_features)
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

OUTPUT: -



```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac9-2.py
0.7721291866028708
```

CODE: -

```
# Extra Trees Classification
import pandas
from sklearn import model_selection
from sklearn.ensemble import ExtraTreesClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
num_trees = 100
max_features = 7
kfold = model_selection.KFold(n_splits=10, random_state=None)
model = ExtraTreesClassifier(n_estimators=num_trees,
max_features=max_features)
```

```
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac9-3.py
0.7720266575529733
```

CODE: -

```
# AdaBoost Classification
import pandas
from sklearn import model_selection
from sklearn.ensemble import AdaBoostClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
num_trees = 30
kfold = model_selection.KFold(n_splits=10, random_state=None)
model = AdaBoostClassifier(n_estimators=num_trees, random_state=None)
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac9-4.py
0.760457963089542
```

CODE: -

```
# Stochastic Gradient Boosting Classification
import pandas
from sklearn import model_selection
from sklearn.ensemble import GradientBoostingClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
num_trees = 100
kfold = model_selection.KFold(n_splits=10, random_state=None)
```

```
model = GradientBoostingClassifier(n_estimators=num_trees,
random_state=None)
results = model_selection.cross_val_score(model, X, Y, cv=kfold)
print(results.mean())
```

OUTPUT: -

```
Admin@DESKTOP-44I664D MINGW64 /e/Python
$ python prac9-5.py
0.7656015037593986
```

CODE: -

```
# Voting Ensemble for Classification
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import VotingClassifier
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
dataframe = pandas.read_csv(url, names=names)
array = dataframe.values
X = array[:,0:8]
Y = array[:,8]
seed = 7
kfold = model_selection.KFold(n_splits=10, random_state=seed,
shuffle=True)
# create the sub models
estimators = []
model1 = LogisticRegression()
estimators.append(('logistic', model1))
model2 = DecisionTreeClassifier()
estimators.append(('cart', model2))
model3 = SVC()
estimators.append(('svm', model3))
# create the ensemble model
ensemble = VotingClassifier(estimators)
results = model_selection.cross_val_score(ensemble, X, Y, cv=kfold)
print(results.mean())
```

OUTPUT: -

```
0.7669514695830485
```