Prac 1

```
import queue as q
from RMP import dict_gn
def bfs(current_city, goal_city, explored_list, exploration_queue):
    explored_list.append(current_city)
    goal_reached = False
    if current_city == goal_city:
        return explored_list, True
    for each_city in dict_gn[current_city].keys():
        if each_city not in explored_list and each_city not in exploration_queue.queue:
            exploration_queue.put(each_city)
    try:
        explored_list, goal_reached = bfs(exploration_queue.get(False), goal_city, explored_list, exploration_queue)
    except q.Empty:
        return explored_list, False
    if goal_reached:
        return explored_list, True
    return explored_list, False


def main():
    start_city = 'Arad'
    goal_city = 'Bucharest'
```

```
    explored_list = []
    exploration_queue = q.Queue()
    exploration_queue.put(start_city)
    goal_reached = False
    explored_list, goal_reached = bfs(exploration_queue.get(False),goal_city,
explored_list, exploration_queue)
    if not goal_reached:
        print('Could not find', goal_city)
    print(explored_list)


main()
```

Prac 2

```
import queue as Q
from RMP import dict_gn
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()
```

Prac 3

```
from RMP import dict_hn
from RMP import dict_gn
import queue as Q
start = 'Arad'
goal = 'Bucharest'
result = ''
def get_fn(citystr):
    cities = citystr.split(" , ")
    hn = gn = 0
    for ctr in range(0, len(cities)-1):
        gn = gn + dict_gn[cities[ctr]][cities[ctr+1]]
    hn = dict_hn[cities[len(cities)-1]]
    return(hn + gn)
def expand(cityq):
    global result
    tot, citystr, thiscity = cityq.get()
    if thiscity == goal:
        result = citystr + " : : " + str(tot)
```

```
        return
    for cty in dict_gn[thiscity]:
        cityq.put((get_fn(citystr + " , " + cty), citystr + " , " + cty, cty))
    expand(cityq)
def main():
    cityq = Q.PriorityQueue()
    thiscity = start
    cityq.put((get_fn(start), start, thiscity))
    expand(cityq)
    print("The A* path with the total is: ")
    print(result)
main()
```

Prac 4

```
import queue as Q
from RMP import dict_gn
from RMP import dict_hn

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

def get_fn(citystr):
    cities=citystr.split(',')
```

```
    hn=gn=0
    for ctr in range(0,len(cities)-1):
        gn=gn+dict_gn[cities[ctr]][cities[ctr+1]]
        #01print("gn",gn)
    hn=dict_hn[cities[len(cities)-1]]
    return(hn+gn)

def printout(cityq):
    for i in range(0,cityq.qsize()):
        print(cityq.queue[i])

def expand(cityq):
    global result
    tot,citystr,thiscity=cityq.get()
    nexttot=999
    if not cityq.empty():
        nexttot,nextcitystr,nextthiscity=cityq.queue[0]
    if thiscity==goal and tot<nexttot:
        result=citystr+'::'+str(tot)
        return
    print("Expanded city----------",thiscity)
    print("Second best f(n)----------",nexttot)
    tempq=Q.PriorityQueue()
    for cty in dict_gn[thiscity]:
```

```
            tempq.put((get_fn(citystr+','+cty),citystr+','+cty,cty))
        for ctr in range(1,3):
            ctrtot,ctrcitystr,ctrthiscity=tempq.get()
            if ctrtot<nexttot:
                cityq.put((ctrtot,ctrcitystr,ctrthiscity))
            else:
                cityq.put((ctrtot,citystr,thiscity))
                break
        printout(cityq)
        expand(cityq)
    def main():
        cityq=Q.PriorityQueue()
        thiscity=start
        cityq.put((999,"NA","NA"))
        cityq.put((get_fn(start),start,thiscity))
        expand(cityq)
        print(result)
    main()
```

Prac 5

```
import numpy as np
import pandas as pd
import sklearn as sk
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
#func importing dataset
def importdata():
    balance_data=pd.read_csv("balance-scale.data")
    #print the dataset shape
    print("Dataset Length : ",len(balance_data))
    print("============check1")
    #printing the dataset observations
    print("Dataset : ",balance_data.head())
    print("============check2")
    return balance_data
#func to split the dataset
def splitdataset(balance_data):
    #seperating the target variable
    X=balance_data.values[:,1:5]
    Y=balance_data.values[:,0]
```

```
    #splitting the dataset into train and test

X_train,X_test,y_train,y_test=train_test_split(X,Y,test_size=0.3,random_state=
100)
    return X,Y,X_train,X_test,y_train,y_test
#function to perform training with entropy
def train_using_entropy(X_train,X_test,y_train,y_test):
    #decision tree with entropy

clf_entropy=DecisionTreeClassifier(criterion="entropy",random_state=100,max
_depth=3,min_samples_leaf=5)
    #performing training
    clf_entropy.fit(X_train,y_train)
    return clf_entropy
def prediction(X_test,clf_object):
    y_pred=clf_object.predict(X_test)
    print("Predicted Values : ")
    print(y_pred)
    return y_pred

def cal_accuracy(y_test,y_pred):
    print("Accuracy : ",accuracy_score(y_test,y_pred)*100)
def main():
    data=importdata()
    X,Y,X_train,X_test,y_train,y_test=splitdataset(data)
```

```
    clf_entropy=train_using_entropy(X_train,X_test,y_train,y_test)
    print("Results using entropy : ")
    y_pred_entropy=prediction(X_test,clf_entropy)
    cal_accuracy(y_test,y_pred_entropy)
main()
```

Prac 6

```
import numpy as np
class NeuralNetwork():
    def __init__(self):
        np.random.seed()
        self.synaptic_weights=2*np.random.random((3,1))-1
    def sigmoid(self, x):
        return 1/(1+np.exp(-x))
    def sigmoid_derivative(self,x):
        return x*(1-x)
    def train(self,training_inputs,training_outputs,training_iterations):
        for iteration in range(training_iterations):
            output=self.think(training_inputs)
            error=training_outputs-output

adjustments=np.dot(training_inputs.T,error*self.sigmoid_derivative(output))
            self.synaptic_weights+=adjustments
    def think(self,inputs):
        inputs=inputs.astype(float)
```

```
        output=self.sigmoid(np.dot(inputs,self.synaptic_weights))
        return output
if __name__=="__main__":
    neural_network=NeuralNetwork()
    print("Beginning randomly generated weights: ")
    print(neural_network.synaptic_weights)
    training_inputs=np.array([[0,0,1],[1,1,1],[1,0,1],[0,1,1]])
    training_outputs=np.array([[0,1,1,0]]).T
    neural_network.train(training_inputs,training_outputs,15000)
    print("Ending weights after training: ")
    print(neural_network.synaptic_weights)
    user_input_one=str(input("User Input One: "))
    user_input_two=str(input("User Input Two: "))
    user_input_three=str(input("User Input Three: "))
    print("Considering new situation:
",user_input_one,user_input_two,user_input_three)
    print("New output data: ")
print(neural_network.think(np.array([user_input_one,user_input_two,user_input
_three])))
```

Prac 7

```
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 makes trees with split number.
#kfold = model_selection.KFold(n_splits=10, random_state=seed)
#n_estimators : This is the number of trees you want to build before predictions.
#Higher number of trees give you better voting optionsand perfomance performance
model = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
#cross_val_score method is used to calculate the accuracy of model sliced into x, y
#cross validator cv  is optional cv=kfold
results = model_selection.cross_val_score(model, X, Y)
print(results.mean())
```