EX.NO:1	UNIFORMED SEARCH ALGORITHMS
DATE:	

AIM:

To implement uniformed search algorithms (BFS, DFS).

1. Implement Breadth First Search (BFS)

ALGORITHM:

1. Create a graph
2. Initialize a starting node
3. Send the graph and initial node as parameters to the bfs function.
4. Mark the initial node as visited and push it into the queue
5. Explore the initial node and add its neighbours to the queue and remove the initial node from the queue
6. Check if the neighbour node of a neighbouring node is already visited
7. If not, visit the neighbouring node neighbours and mark them as visited
8. Repeat this process until all the nodes in a graph are visited and the queue becomes empty

PROGRAM:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    s = queue.pop(0)
    print (s, end = " ")
    for neighbour in  graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
bfs(visited, graph, 'A')
```

OUTPUT:

```
A B C D E F
```

2. Implement Depth First Search (DFS)

ALGORITHM:

1. Start by putting any one of the graph's vertices on top of a stack
2. Take the top item of the stack and add it to the visited list
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack
4. Keep repeating steps 2 and 3 until the stack is empty

PROGRAM:

```
def recursive_dfs(graph, source,path = []):
        if source not in path:
            path.append(source)
            if source not in graph:
                return path
            for neighbour in graph[source]:
                path = recursive_dfs(graph, neighbour, path)
        return path
graph = {"A":["B","C", "D"],
            "B":["E"],
            "C":["F","G"],
            "D":["H"],
            "E":["I"],
            "F":["J"]}
path = recursive_dfs(graph, "A")
print(" ".join(path))
```

OUTPUT:

```
A B E I C F J G D H
```

RESULT:

Thus the above programs are executed and the outputs are verified.

EX.NO:2	INFORMED SEARCH ALGORITHMS
DATE:	

AIM:

To implement informed search algorithms (A*, memory-bounded A*).

1. Implement A* algorithm

ALGORITHM:

1. Firstly, Place the starting node into OPEN and find its f (n) value
2. Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success
3. Else remove the node from OPEN, and find all its successors
4. Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE
5. Goto Step-2
6. Exit

PROGRAM:

```
from collections import deque

class Graph:

    def___init_(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
```

```
closed_list = set([])
g = {}

g[start_node] = 0
parents = {}
parents[start_node] = start_node

while len(open_list) > 0:
    n = None
    for v in open_list:
        if n == None or g[v] + self.h(v) < g[n] + self.h(n):
            n = v;

    if n == None:
        print('Path does not exist!')
        return None
    if n == stop_node:
        reconst_path = []

        while parents[n] != n:
            reconst_path.append(n)
            n = parents[n]

        reconst_path.append(start_node)
        reconst_path.reverse()
        print('Path found: {}'.format(reconst_path))
        return reconst_path

    for (m, weight) in self.get_neighbors(n)
        if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight

        else:
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n

                if m in closed_list:
                    closed_list.remove(m)
                    open_list.add(m)
    open_list.remove(n)
    closed_list.add(n)
```

```
        print('Path does not exist!')
        return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

```
Path found: ['A', 'B', 'D']
['A', 'B', 'D']
```

2. Implement memory-bounded A* algorithm

ALGORITHM:

1. Initialize the CLOSE AND OPEN list
2. Initialize the starting node
3. Find the path with the lowest weight
4. Add previous weight and the current heuristics and weight of the node
5. Find the shortest path with weight for the goal node
6. Exit

PROGRAM:

```
nodes = {
        'A': [['B', 6], ['F', 3]],
        'B': [['A', 6], ['C', 3], ['D', 2]],
        'C': [['B', 3], ['D', 1], ['E', 5]],
        'D': [['B', 2], ['C', 1], ['E', 8]],
        'E': [['C', 5], ['D', 8], ['I', 5], ['J', 5]],
        'F': [['A', 3], ['G', 1], ['H', 7]],
        'G': [['F', 1], ['I', 3]],
        'H': [['F', 7], ['I', 2]],
        'I': [['G', 3], ['H', 2], ['E', 5], ['J', 3]],
        'J': [['E', 5], ['I', 3]]
       }
h = {
    'A' : 10,
```

```
     'B' : 8,
     'C' : 5,
     'D' : 7,
     'E' : 3,
     'F' : 6,
     'G' : 5,
     'H' : 3,
     'I' : 1,
     'J' : 0
    }

def astar(start, goal):
    opened = []
    closed = []
    visited = set()
    opened.append([start, h[start]])
    while opened :
        min = 1000
        val = ''
        for i in opened:
            if i[1] < min:
                min = i[1]
                val = i[0]
        closed.append(val)
        visited.add(val)
        if goal not in closed:
            for i in nodes[val]:
                if i[0] not in visited:
                    opened.append([i[0], (min-h[val]+i[1]+h[i[0]])])
        else:
            break
        opened.remove([val, min])

    closed = closed[::-1]
    min = 1000
    for i in opened:
        if i[1] < min:
            min = i[1]

    lens = len(closed)
    i = 0
    while i < lens-1:
        nei = []
        for j in nodes[closed[i]]:
```

```
            nei.append(j[0])
        if closed[i+1] not in nei:
            del closed[i+1]
            lens-=1
        i+=1
    closed = closed[::-1]
    return closed, min

print(astar('A', 'J'))
```

OUTPUT:

```
(['A', 'F', 'G', 'I', 'J'], 10)
```

RESULT:

Thus the above programs are executed and the outputs are verified.

EX.NO:3	NAÏVE BAYES MODEL
DATE:	

AIM:

To implement Gaussian naïve Bayes model

ALGORITHM:

1. Import necessary libraries and packages
2. Load the dataset
3. Split the dataset into train data and test data
4. Load the Gaussian naïve Bayes algorithm
5. Train the algorithm with train data
6. Test the accuracy of the algorithm

PROGRAM:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=1)
gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)
print("Gaussian Naive Bayes model accuracy(in %):", metrics.accuracy_score(y_test, y_pred)
*100)
```

OUTPUT:

```
Gaussian Naive Bayes model accuracy(in %): 95.0
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:4	BAYESIAN NETWORK
DATE:	

AIM:

To implement Bayesian Network.

ALGORITHM:

1. Import necessary packages and modules.
2. Load the Bayesian model
3. Draw the conditional probability table for each node in the Bayesian network
4. Draw the conditional probability table for posterior probability of burglary if john calls and marry calls and alarm if the burglary happens and earthquake happens

PROGRAM:

```
import pgmpy.models
import pgmpy.inference
import networkx as nx
import pylab as plt
model = pgmpy.models.BayesianModel([('Burglary', 'Alarm'),
                                    ('Earthquake', 'Alarm'),
                                    ('Alarm', 'JohnCalls'),
                                    ('Alarm', 'MaryCalls')])
cpd_burglary = pgmpy.factors.discrete.TabularCPD('Burglary', 2, [[0.001], [0.999]])
cpd_earthquake = pgmpy.factors.discrete.TabularCPD('Earthquake', 2, [[0.002], [0.998]])
cpd_alarm = pgmpy.factors.discrete.TabularCPD('Alarm', 2, [[0.95, 0.94, 0.29, 0.001],
                                                           [0.05, 0.06, 0.71, 0.999]],
                                              evidence=['Burglary', 'Earthquake'],
                                              evidence_card=[2, 2])
cpd_john = pgmpy.factors.discrete.TabularCPD('JohnCalls', 2, [[0.90, 0.05],
                                                              [0.10, 0.95]],
                                             evidence=['Alarm'],
                                             evidence_card=[2])
cpd_mary = pgmpy.factors.discrete.TabularCPD('MaryCalls', 2, [[0.70, 0.01],
                                                              [0.30, 0.99]],
                                             evidence=['Alarm'],
                                             evidence_card=[2])
model.add_cpds(cpd_burglary, cpd_earthquake, cpd_alarm, cpd_john, cpd_mary)
model.check_model()

print('Probability distribution, P(Burglary)')
print(cpd_burglary)
```

```
print('Probability distribution, P(Earthquake)')
print(cpd_earthquake)

print('Joint probability distribution, P(Alarm | Burglary, Earthquake)')
print(cpd_alarm)

print('Joint probability distribution, P(JohnCalls | Alarm)')
print(cpd_john)

print('Joint probability distribution, P(MaryCalls | Alarm)')
print(cpd_mary)
infer = pgmpy.inference.VariableElimination(model)
posterior_probability = infer.query(['Burglary'], evidence={'JohnCalls': 0, 'MaryCalls': 0})
print('Posterior probability of Burglary if JohnCalls(True) and MaryCalls(True)')
print(posterior_probability)

posterior_probability = infer.query(['Alarm'], evidence={'Burglary': 0, 'Earthquake': 0})
print('Posterior probability of Alarm sounding if Burglary(True) and Earthquake(True)')
print(posterior_probability)
```

OUTPUT:

```
Probability distribution, P(Burglary)
+-------------+-------+
| Burglary(0) | 0.001 |
+-------------+-------+
| Burglary(1) | 0.999 |
+-------------+-------+

Probability distribution, P(Earthquake)
+---------------+-------+
| Earthquake(0) | 0.002 |
+---------------+-------+
| Earthquake(1) | 0.998 |
+---------------+-------+

Joint probability distribution, P(Alarm | Burglary, Earthquake)
+------------+---------------+---------------+---------------+---------------+
| Burglary   | Burglary(0)   | Burglary(0)   | Burglary(1)   | Burglary(1)   |
+------------+---------------+---------------+---------------+---------------+
| Earthquake | Earthquake(0) | Earthquake(1) | Earthquake(0) | Earthquake(1) |
+------------+---------------+---------------+---------------+---------------+
| Alarm(0)   | 0.95          | 0.94          | 0.29          | 0.001         |
+------------+---------------+---------------+---------------+---------------+
| Alarm(1)   | 0.05          | 0.06          | 0.71          | 0.999         |
+------------+---------------+---------------+---------------+---------------+
```

```
Joint probability distribution, P(JohnCalls | Alarm)
+--------------+----------+----------+
| Alarm        | Alarm(0) | Alarm(1) |
+--------------+----------+----------+
| JohnCalls(0) | 0.9      | 0.05     |
+--------------+----------+----------+
| JohnCalls(1) | 0.1      | 0.95     |
+--------------+----------+----------+

Joint probability distribution, P(MaryCalls | Alarm)
+--------------+----------+----------+
| Alarm        | Alarm(0) | Alarm(1) |
+--------------+----------+----------+
| MaryCalls(0) | 0.7      | 0.01     |
+--------------+----------+----------+
| MaryCalls(1) | 0.3      | 0.99     |
+--------------+----------+----------+

Posterior probability of Burglary if JohnCalls(True) and MaryCalls(True)
+-------------+-----------------+
| Burglary    |   phi(Burglary) |
+=============+=================+
| Burglary(0) |          0.2842 |
+-------------+-----------------+
| Burglary(1) |          0.7158 |
+-------------+-----------------+
```

```
Posterior probability of Alarm sounding if Burglary(True) and Earthquake(True)
+----------+--------------+
| Alarm    |   phi(Alarm) |
+==========+==============+
| Alarm(0) |       0.9500 |
+----------+--------------+
| Alarm(1) |       0.0500 |
+----------+--------------+
```

RESULT:

Thus the above programs are executed and the output are verified.

EX.NO:5	REGRESSION MODELS
DATE:	

AIM:

To build regression models.

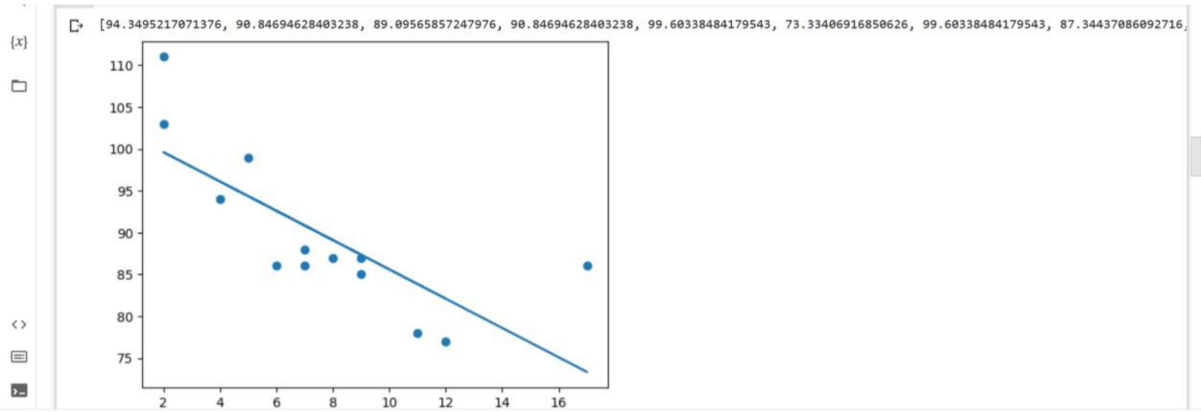
ALGORITHM:

1. Import the necessary packages and modules
2. Create the arrays that represent the values of the x and y axis
3. Create a function that uses the slope and intercept values to return a new value. This new value represents where on the y-axis the corresponding x value will be placed
4. Run each value of the x array through the function. This will result in a new array with new values for the y-axis
5. Draw the original scatter plot
6. Draw the line of linear regression
7. Display the diagram

PROGRAM:

```
import matplotlib.pyplot as plt
from scipy import stats
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
slope, intercept, r, p, std_err = stats.linregress(x, y)
def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))
print(mymodel)
plt.scatter(x, y)
plt.plot(x,mymodel)
plt.show()
```

OUTPUT:



RESULT:

Thus the above program is executed and the output is verified.

EX.NO:6	DECISION TREES AND RANDOM FORESTS
DATE:	

AIM:

To build decision trees and random forests

1. Build decision tree

ALGORITHM:

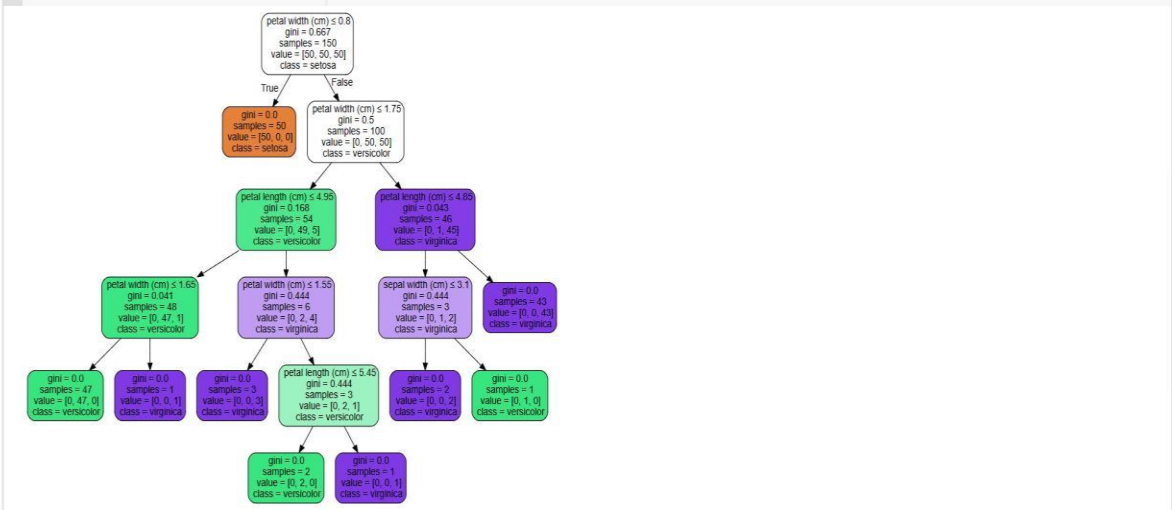
1. Import necessary packages and libraries
2. Load the dataset
3. Load the algorithm decision tree and train the algorithm using the dataset
4. Predict the category of new data
5. Print the graph for the decision tree.

PROGRAM:

```
from sklearn.datasets import load_iris
from sklearn import tree
import graphviz
iris = load_iris()
X, y = iris.data, iris.target
targets = iris.target_names
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
X_pred = [6.7, 3.0, 5.2, 2.3]
y_pred = clf.predict([X_pred])
print("Prediction is: {}".format(targets[y_pred]))
dot_data = tree.export_graphviz(clf, out_file=None,feature_names=iris.feature_names,
                class_names=iris.target_names,
                 filled=True, rounded=True,
                 special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

OUTPUT:

```
Prediction is: ['virginica']
```



2. Build random forest

ALGORITHM:

1. Import necessary packages and libraries
2. Load the dataset
3. Load the algorithm Random Forest and train the algorithm using the dataset
4. Predict the category of new data

PROGRAM:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data, iris.target
targets = iris.target_names
clf = RandomForestClassifier(random_state = 100)
clf = clf.fit(X, y)
X_pred = [6.7, 3.0, 5.2, 2.3]
y_pred = clf.predict([X_pred])
print("Prediction is: {}".format(targets[y_pred]))
```

OUTPUT:

```
Prediction is: ['virginica']
```

RESULT:

Thus the above programs are executed and the outputs are verified.

EX.NO:7	**SVM MODEL**
DATE:	

AIM:

To build SVM (Support Vector Machine) models

ALGORITHM:

1. Import necessary packages and libraries
2. Load the dataset
3. Load the algorithm Support Vector Machine and train the algorithm using the dataset
4. Predict the category of new data

PROGRAM:

```
from sklearn.datasets import load_iris
from sklearn.svm import SVC
iris=load_iris()
X_train = iris.data
y_train = iris.target
targets = iris.target_names
print(targets)
cls = SVC()
cls.fit(X_train, y_train)
X_pred = [5.1, 3.2, 1.5, 0.5]
y_pred = cls.predict([X_pred])
print("Prediction is: {}".format(targets[y_pred]))
```

OUTPUT:

```
['setosa' 'versicolor' 'virginica']
Prediction is: ['setosa']
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:8	ENSEMBLE TECHNIQUES
DATE:	

AIM:

To implement Max voting ensemble technique.

ALGORITHM:

1. Import the necessary modules and packages
2. Load the dataset
3. Load the models(SVM, Random Forest, Decision tree)
4. Combine the models and train them using dataset
5. Predict the category of the new data point.

PROGRAM:

```
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn import tree
from sklearn.ensemble import VotingClassifier
iris=load_iris()
X_train = iris.data
y_train = iris.target
targets = iris.target_names
print(targets)
m1 = tree.DecisionTreeClassifier()
m2 = RandomForestClassifier(random_state = 100)
m3 = SVC()
final_model=VotingClassifier(estimators=[('dt',m1),('rf',m2),('svc',m3)],voting='hard')
final_model.fit(X_train, y_train)
X_pred = [6.7, 3.0, 5.2, 2.3]
y_pred = final_model.predict([X_pred])
print("Prediction is: {}".format(targets[y_pred]))
```

OUTPUT:

```
['setosa' 'versicolor' 'virginica']
Prediction is: ['virginica']
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:9	CLUSTERING ALGORITHMS
DATE:	

AIM:

To implement K-Nearest Neighbor clustering algorithm.

ALGORITHM:

1. Import necessary packages and libraries
2. Load the dataset
3. Load the algorithm k-Nearest Neighbor and train the algorithm using the dataset
4. Predict the category of new data

PROGRAM:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
iris=load_iris()
X_train = iris.data
y_train = iris.target
targets = iris.target_names
print(targets)
cls = KNeighborsClassifier(n_neighbors=5)
cls.fit(X_train, y_train)
X_pred = [6.7, 3.0, 5.2, 2.3]
y_pred = cls.predict([X_pred])
print("The Prediction is")
print("".join(targets[y_pred]))
```

OUTPUT:

```
['setosa' 'versicolor' 'virginica']
The Prediction is
virginica
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:10	EM FOR BAYESIAN NETWORK
DATE:	

AIM:

To implement EM for Bayesian networks.

ALGORITHM:

1. Import necessary libraries and packages
2. Define the bayesian network
3. Generate the true probability distributions P for each node
4. Randomly initialize the estimated probability distributions P^ for each node
5. Perform the E-step and M-step for 32 epochs
6. Plot the log likelihood for each epoch

PROGRAM:

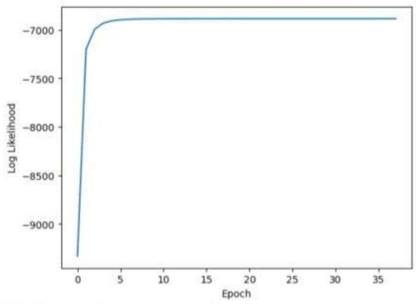
```
import numpy as np
import time
graphNodes = ["a", "b", "c", "d", "e", "f", "g", "h"]

graphNodeIndices = {}
for idx, node in enumerate(graphNodes):
    graphNodeIndices[node] = idx

graphNodeNumStates = {
    "a": 3,
    "b": 4,
    "c": 5,
    "d": 4,
    "e": 3,
    "f": 4,
    "g": 5,
    "h": 4
}

nodesToUpdate = ["a", "b", "c", "d", "e", "f", "g", "h"]

nodeParents = {
    "a": [],
    "b": [],
    "c": ["a"],
    "d": ["a", "b"],
```

```
    "e": ["a", "c"],
    "f": ["b", "d"],
    "g": ["e"],
    "h": ["f"]
}

tensorNodeOrder = {}
for node in graphNodes:
    tensorNodeOrder[node] = [node] + nodeParents[node]
def randomTensorGenerator(shape):
    return np.random.uniform(0.0, 1.0, shape)

def conditionNodeOnParents(probTensor, node, tensorNodeOrder):
    assert(node in tensorNodeOrder)
    inferredDimension = tensorNodeOrder.index(node)
    probTensor = probTensor / np.expand_dims(np.sum(probTensor, inferredDimension), infe
rredDimension)
    return probTensor

np.random.seed(0)
p = {}
for node in graphNodes:
    tensorDimensions = [graphNodeNumStates[x] for x in tensorNodeOrder[node]]
    p[node] = randomTensorGenerator(tensorDimensions)

for node in p:
    p[node] = conditionNodeOnParents(p[node], tensorNodeOrder[node][0], tensorNodeOrder
[node])
    print("p(" + node + "|" + str(nodeParents[node]) + ") dimensions: " + str(p[node].shape))
np.random.seed(int(time.time()))

phat = {}

for node in p:
    phat[node] = randomTensorGenerator(p[node].shape)
    phat[node] = conditionNodeOnParents(phat[node], tensorNodeOrder[node][0], tensorNode
Order[node])
    print("phat(" + node + "|" + str(nodeParents[node]) + ") dimensions: " + str(phat[node].sha
pe))
```

OUTPUT:

```
p(a|[]) dimensions: (3,)
p(b|[]) dimensions: (4,)
p(c|['a']) dimensions: (5, 3)
p(d|['a', 'b']) dimensions: (4, 3, 4)
p(e|['a', 'c']) dimensions: (3, 3, 5)
p(f|['b', 'd']) dimensions: (4, 4, 4)
p(g|['e']) dimensions: (5, 3)
p(h|['f']) dimensions: (4, 4)
phat(a|[]) dimensions: (3,)
phat(b|[]) dimensions: (4,)
phat(c|['a']) dimensions: (5, 3)
phat(d|['a', 'b']) dimensions: (4, 3, 4)
phat(e|['a', 'c']) dimensions: (3, 3, 5)
phat(f|['b', 'd']) dimensions: (4, 4, 4)
phat(g|['e']) dimensions: (5, 3)
phat(h|['f']) dimensions: (4, 4)
```

```
Epoch: 0, Log-Likelihood: -9332.881805615138
Epoch: 1, Log-Likelihood: -7199.713905292975
Epoch: 2, Log-Likelihood: -6993.720926287904
Epoch: 3, Log-Likelihood: -6931.341580007712
Epoch: 4, Log-Likelihood: -6907.326379308647
Epoch: 5, Log-Likelihood: -6896.724197678392
Epoch: 6, Log-Likelihood: -6891.554232318873
Epoch: 7, Log-Likelihood: -6888.846896024085
Epoch: 8, Log-Likelihood: -6887.354740313764
Epoch: 9, Log-Likelihood: -6886.499700392717
Epoch: 10, Log-Likelihood: -6885.9935470337805
Epoch: 11, Log-Likelihood: -6885.684871707819
Epoch: 12, Log-Likelihood: -6885.4910981205785
Epoch: 13, Log-Likelihood: -6885.36587906848
Epoch: 14, Log-Likelihood: -6885.2825697235985
Epoch: 15, Log-Likelihood: -6885.225513754962
Epoch: 16, Log-Likelihood: -6885.185315029962
Epoch: 17, Log-Likelihood: -6885.1562137795645
Epoch: 18, Log-Likelihood: -6885.134602550935
Epoch: 19, Log-Likelihood: -6885.118172014161
Epoch: 20, Log-Likelihood: -6885.105410897078
Epoch: 21, Log-Likelihood: -6885.09530822375
Epoch: 22, Log-Likelihood: -6885.087172924613
Epoch: 23, Log-Likelihood: -6885.080522606069
Epoch: 24, Log-Likelihood: -6885.075013717732
Epoch: 25, Log-Likelihood: -6885.07039690289
Epoch: 26, Log-Likelihood: -6885.066487923169
Epoch: 27, Log-Likelihood: -6885.063148378119
Epoch: 28, Log-Likelihood: -6885.060272690486
Epoch: 29, Log-Likelihood: -6885.05777916803
Epoch: 30, Log-Likelihood: -6885.0556037628585
Epoch: 31, Log-Likelihood: -6885.053695645295
Epoch: 32, Log-Likelihood: -6885.052014017711
Epoch: 33, Log-Likelihood: -6885.0505257893865
Epoch: 34, Log-Likelihood: -6885.04920385731
Epoch: 35, Log-Likelihood: -6885.048025820005
Epoch: 36, Log-Likelihood: -6885.046973004283
Epoch: 37, Log-Likelihood: -6885.046029721253
Complete
```




```
p[a]: [0.29399155 0.38311672 0.32289173]
phat[a]: [0.2574389  0.38565377 0.35690733]
p[h]: [[0.39873888 0.31476471 0.20665257 0.13284953]
 [0.03128721 0.27879818 0.28333014 0.36369784]
 [0.20367005 0.11526534 0.02242566 0.03512425]
 [0.36630386 0.29117177 0.48759164 0.46832838]]
phat[h]: [[3.36946396e-01 2.02987891e-01 1.54407446e-01 8.95918646e-02]
 [3.41693514e-02 3.43958901e-01 3.55072966e-01 4.41662182e-01]
 [1.93191114e-01 9.15878095e-02 4.20099026e-17 4.38732233e-02]
 [4.35693138e-01 3.61465398e-01 4.90519587e-01 4.24872730e-01]]
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:11	NEURAL NETWORK MODEL
DATE:	

AIM:

To build simple Neural network (NN) models.

ALGORITHM:

1. Import the necessary packages and libraries
2. Use numpy arrays to store inputs x and output y
3. Define the network model and its arguments.
4. Set the number of neurons/nodes for each layer
5. Compile the model and calculate its accuracy
6. Print the summary of the model

PROGRAM:

```
from keras.models import Sequential
from keras.layers import Dense, Activation
import numpy as np
x = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([[0], [1], [1], [0]])
model = Sequential()
model.add(Dense(2, input_shape=(2,)))
model.add(Activation('sigmoid'))
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss='mean_squared_error', optimizer='sgd', metrics=['accuracy'])
model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 2)                 6

 activation (Activation)     (None, 2)                 0

 dense_1 (Dense)             (None, 1)                 3

 activation_1 (Activation)   (None, 1)                 0

=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

RESULT:

Thus the above program is executed and the output is verified.

EX.NO:12	DEEP LEARNING NEURAL NETWORK MODEL
DATE:	

AIM:

To build deep learning NN (Neural Network) models.

ALGORITHM:

1. Load the dataset
2. Split the dataset into input x and output y
3. Define the keras model
4. Compile the keras model
5. Train the keras model with the dataset
6. Make predictions using the model

PROGRAM:

```
from numpy import loadtxt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
dataset = loadtxt('https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv', delimiter=',')
X = dataset[:,0:8]
y = dataset[:,8]
model = Sequential()
model.add(Dense(12, input_shape=(8,), activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(X, y, epochs=150, batch_size=10, verbose=0)
predictions = (model.predict(X) > 0.5).astype(int)
for i in range(5):
 print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

OUTPUT:

```
24/24 [==============================] - 0s 2ms/step
[6.0, 148.0, 72.0, 35.0, 0.0, 33.6, 0.627, 50.0] => 1 (expected 1)
[1.0, 85.0, 66.0, 29.0, 0.0, 26.6, 0.351, 31.0] => 0 (expected 0)
[8.0, 183.0, 64.0, 0.0, 0.0, 23.3, 0.672, 32.0] => 1 (expected 1)
[1.0, 89.0, 66.0, 23.0, 94.0, 28.1, 0.167, 21.0] => 0 (expected 0)
[0.0, 137.0, 40.0, 35.0, 168.0, 43.1, 2.288, 33.0] => 1 (expected 1)
```

RESULT:

Thus the above program is executed and the output is verified.