

| **Title:**  **Implementation of Bayesian belief networks** |
| --- |

**Objective:** Implement BBN

**Expected Outcome of Experiment:**

| **CO** | **Outcome** |
| --- | --- |
| **CO1** | Describe and apply supervised learning methods |

**Books/ Journals/ Websites referred: PPTs**

**Theory of BBN algorithm:**

Bayesian Belief Networks (BBNs), also known as Bayesian Networks or Belief Networks, are graphical models that represent probabilistic relationships among variables in a domain. They are based on Bayesian probability theory, which describes how beliefs about uncertain events should change based on the evidence.

Here's a brief overview of the key concepts:

1.**Nodes**: In a BBN, nodes represent variables. These variables can be discrete or continuous and can represent things like the weather, patient health status, or the outcome of a sports game.

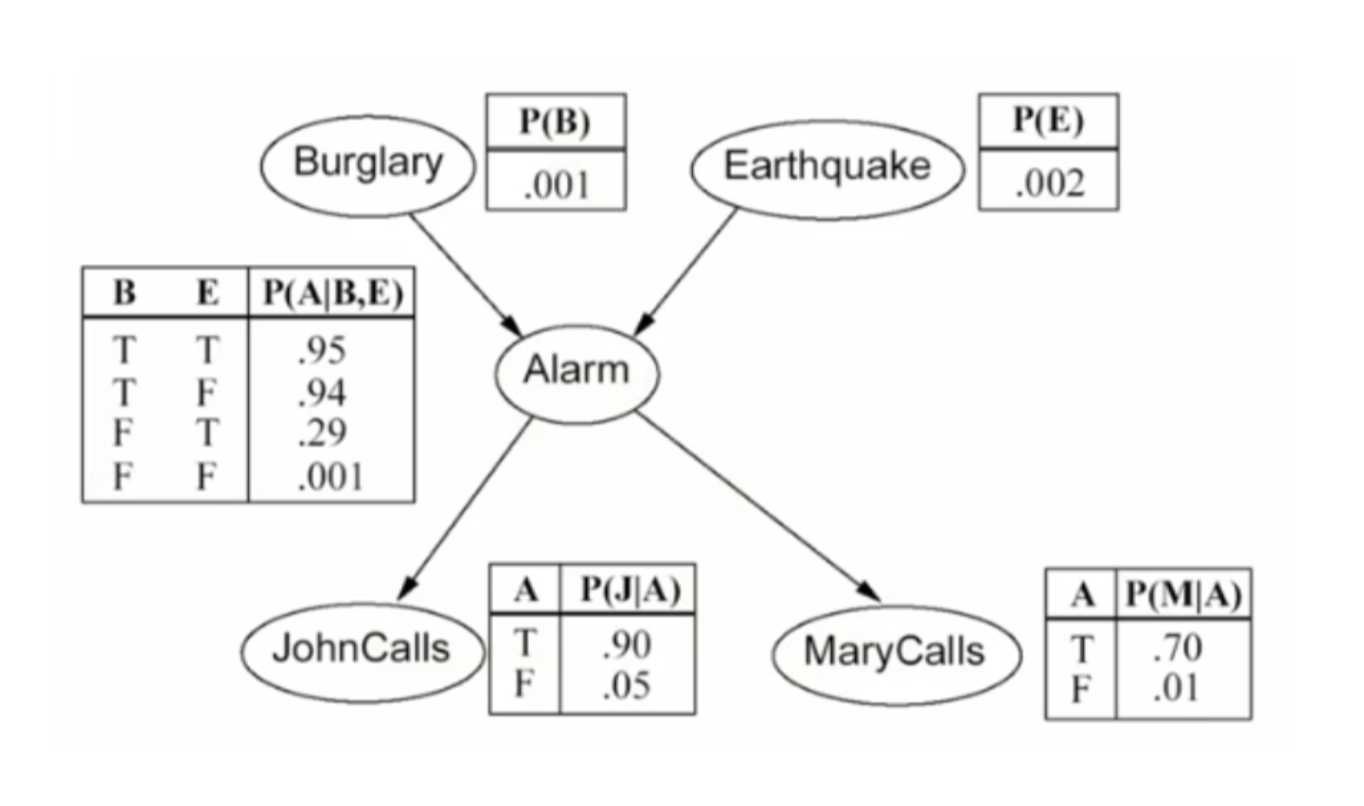
2. **Edges**: Edges between nodes represent probabilistic dependencies between variables. An edge from node A to node B indicates that the probability distribution of B depends on the value of A.

3. **Conditional Probability Tables (CPTs):** Each node in a BBN has a CPT that quantifies the probability of each possible value of the node given the values of its parent nodes. These tables encode the probabilistic relationships in the network.

4. **Inference**: BBNs can be used for probabilistic inference. Given some evidence (observed values of certain variables), the network can calculate the posterior probability distribution of other variables. This is done using algorithms like variable elimination or Gibbs sampling.

5. **Learning**: BBNs can be learned from data. This involves estimating the parameters of the CPTs and the structure of the network from observed data. Learning algorithms include methods like maximum likelihood estimation and Bayesian parameter learning.

**Details of data set used:**

****

**Explanation of API/Tool used for implementation:** Python, no APIs, hardcoded dataset.

import itertools

def get\_binary\_numbers(n):

if(n==1):

return []

binary\_comb =[]

width = len(bin(n)[2:])-1

for i in range(0, n):

binary = bin(i)[2:]

padded\_binary = binary.zfill(width)

binary\_comb.append(padded\_binary)

return binary\_comb

class event:

table:list[float]

parents:list[int]

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

self.table = table

def probability(self,is\_true:int)->float:

if(is\_true=='1'):

if(self.parents==None):

return self.table[0]

p=0

binary\_comb = get\_binary\_numbers(2\*\*len(self.parents))

i=0

print(binary\_comb)

for comb in binary\_comb:

#print(i)

p1=1

for parent\_id in range(len(self.parents)):

p1=p1\*self.parents[parent\_id].probability(comb[parent\_id])

#print(p1)

#print(p1,len(comb))

p1=p1\*self.table[i]

p=p+p1

i+=1

return p

if(is\_true=='0'):

if(self.parents==None):

return 1-self.table[0]

p=0

binary\_comb = get\_binary\_numbers(2\*\*len(self.parents))

i=0

#print(binary\_comb)

for comb in binary\_comb:

#print(i)

p1=1

for parent\_id in range(len(self.parents)):

p1=p1\*self.parents[parent\_id].probability(comb[parent\_id])

#print(p1)

#print(p1,comb,len(comb))

p1=p1\*(1-self.table[i])

#print((1-self.table[i]))

p=p+p1

i+=1

return p

def cond\_probability(self,is\_true:int,condition:list[int])->float:

binary\_string = ''.join(map(str, condition)) # Convert the list to a string

decimal\_number = int(binary\_string, 2) # Convert the binary string to decimal

if(is\_true=='1'):

return self.table[decimal\_number]

if(is\_true=='0'):

return 1-self.table[decimal\_number]

bulgary = event([0.001])

bulgary.parents = None

earthquake = event([0.002])

earthquake.parents = None

alarm = event([0.001,0.29,0.94,0.95])

alarm.parents=[earthquake,bulgary]

john = event([0.05,0.9])

john.parents=[alarm]

mary = event([0.01,0.7])

mary.parents=[alarm]

# print(john.probability("1"))

# print(alarm.cond\_probability("1",[1,1]))

# print(john.cond\_probability("1",[0]))

given = [1,1,1,0,0] # assumes all are given. Else will need to add.

events = [john, mary, alarm, earthquake,bulgary]

def get\_conditional\_probability(given,events):

p=1

i = -1

for event in events:

i+=1

if(event.parents == None):

p =p\*event.probability(str(given[i]))

else:

given\_list = []

for parent in event.parents:

given\_list.append(given[events.index(parent)])

# print(given\_list)

p=p\*event.cond\_probability(str(given[i]),given\_list)

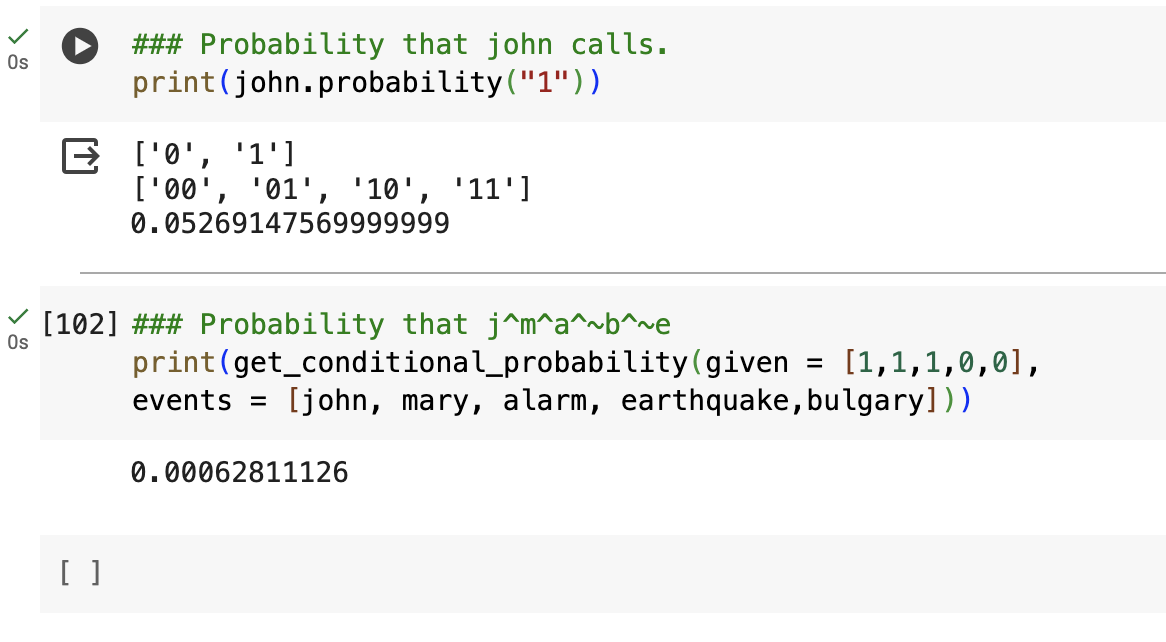
# print(p)

return p

# print(alarm.probability("1"))

# print(john.probability("1",1)\*mary.probability("1")\*alarm.probability("1")\*earthquake.probability("0")\*bulgary.probability("0"))

**Results:**



**Conclusion:** Thus we have implemented Bayesian Belief networks, BBN in Python. We have done that from scratch on a small dataset without using any external libraries. Bayesian Belief Networks provide a flexible and powerful framework for modelling and reasoning about uncertain domains, making them a valuable tool in many areas of research and application.