

In [41]:

```
import numpy as np
import random as rd
import math as mt
import matplotlib as plt
```

## MAPSI - TME - Rappels de Proba/stats

### I- La planche de Galton ( **obligatoire** )

#### I.1- Loi de Bernouilli

Écrire une fonction `bernouilli: float ->int` qui prend en argument la paramètre  $p \in [0, 1]$  et qui renvoie aléatoirement 0 (avec la probabilité  $1 - p$ ) ou 1 (avec la probabilité  $p$ ).

In [42]:

```
def bernouilli(p):
    # votre code
    a=rd.random()
    cond=a<p
    #print(a)
    return (1 if cond else 0)

pass
```

In [43]:

```
## Test
bernouilli( 0.5)
```

Out[43]:

0

#### I.2- Loi binomiale

Écrire une fonction `binomiale: int , float -> int` qui prend en argument un entier  $n$  et  $p \in [0, 1]$  et qui renvoie aléatoirement un nombre tiré selon la distribution  $\mathcal{B}(n, p)$ .

In [44]:

```
def binomiale(n,p):
    # de même loi qu'une somme de variables de Bernouilli indépendantes
    return sum( bernouilli( p) for i in range(n))


pass
```

In [45]:

```
#test  
print(binomiale (100, 0.5))
```

60

### I.3- Histogramme de la loi binomiale

Dans cette question, on considère une planche de Galton de hauteur  $n$ . On rappelle  Planche de Galton que des bâtons horizontaux (oranges) sont cloués à cette planche comme le montre la figure ci-contre.

Des billes bleues tombent du haut de la planche et, à chaque niveau, se retrouvent à la verticale d'un des bâtons. Elles vont alors tomber soit à gauche, soit à droite du bâton, jusqu'à atteindre le bas de la planche. Ce dernier est constitué de petites boîtes dont les bords sont symbolisés par les lignes verticales grises.

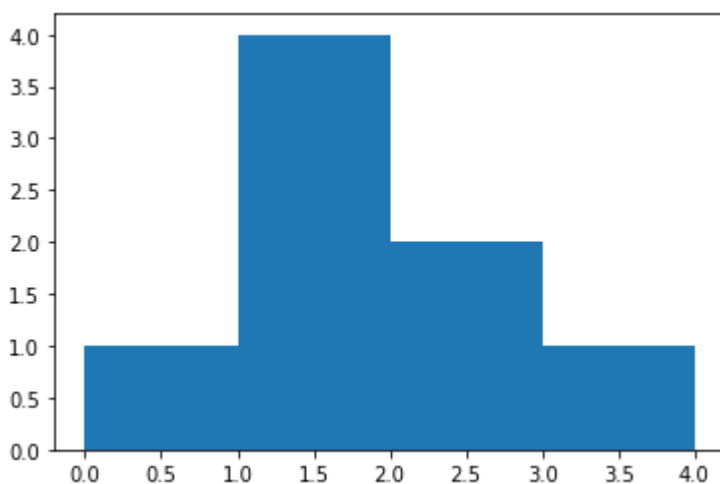
Chaque boîte renferme des billes qui sont passées exactement le même nombre de fois à droite des bâtons oranges. Par exemple, la boîte la plus à gauche renferme les billes qui ne sont jamais passées à droite d'un bâton, celle juste à sa droite renferme les billes passées une seule fois à droite d'un bâton et toutes les autres fois à gauche, et ainsi de suite.

La répartition des billes dans les boîtes suit donc une loi binomiale  $\mathcal{B}(n, 0.5)$ .

Écrire un script qui crée un tableau de 1000 cases dont le contenu correspond à 1000 instanciations de la loi binomiale  $\mathcal{B}(n, 0.5)$ . Afin de voir la répartition des billes dans la planche de Galton, tracer l'histogramme de ce tableau. Vous pourrez utiliser la fonction hist de matplotlib.pyplot:

In [46]:

```
import matplotlib.pyplot as plt  
  
plt.hist ([0,1,2,1,2,4,1,1], 4);  
#l=set([0,1,2,1,2,4,1,1])  
#len(l)
```



Pour le nombre de bins, calculez le nombre de valeurs différentes dans votre tableau.

In [47]:

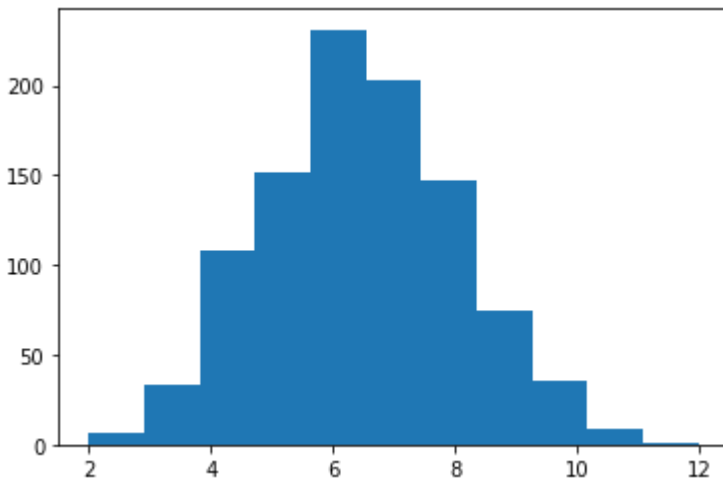
```
# entrée: n: hauteur de la planche de Galton
def Galton(n):
    tab=[binomiale( n, 0.5) for i in range(1000 )]
    return tab
#print(tab)
```

In [48]:

```
def draw_hist(n):
    list=Galton(n)
    nb_values=len(set(list))
    plt.hist(list, nb_values)
```


In [49]:

```
#Test
draw_hist(13)
```



## II- Visualisation d'indépendances ( obligatoire)

### II.1- Loi normale centrée réduite

On souhaite visualiser la fonction de densité de la loi normale. Pour cela, on va créer un  Planche de Galton ensemble de  $k$  points  $(x_i, y_i)$ , pour des  $x_i$  équi-espacés variant de  $-2\sigma$  à  $2\sigma$ , les  $y_i$  correspondant à la valeur de la fonction de densité de la loi normale centrée de variance  $\sigma^2$ , autrement dit  $\mathcal{N}(0, \sigma^2)$ .

Écrire une fonction normale : int , float -> float np.array qui, étant donné un paramètre entier  $k$  impair et un paramètre réel sigma renvoie l' array numpy des  $k$  valeurs  $y_i$ . Afin que l' array numpy soit bien symétrique, on lèvera une exception si  $k$  est pair.

In [50]:

```
def normale ( k, sigma ):  
    #votre code  
    if (k%2==0):  
        raise Exception(' k should be odd')  
    # les valeurs de x que je vais évaluer  
    x=np.linspace(-2*sigma, 2*sigma, k)  
    return x, (1/(mt.sqrt(2*mt.pi)* sigma) * np.exp(-(1/2)*np.power((x /sigma),  
2)))  
    pass
```

In [51]:

```
#print(normale(10, 1)) # Exception well raised  
#test  
print(normale(11, 1))
```

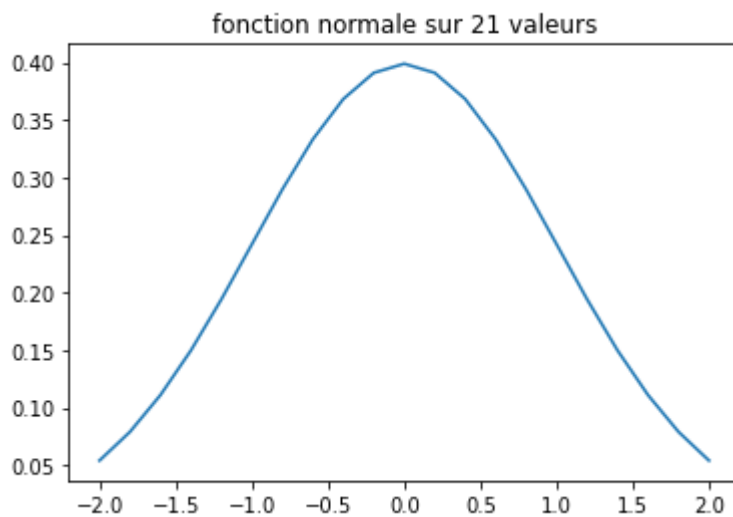
```
(array([-2. , -1.6, -1.2, -0.8, -0.4,  0. ,  0.4,  0.8,  1.2,  1.6,  
2. ]), array([0.05399097, 0.11092083, 0.19418605, 0.28969155, 0.3682  
7014,  
0.39894228, 0.36827014, 0.28969155, 0.19418605, 0.11092083,  
0.05399097]))
```

Vérifier la validité de votre fonction en affichant grâce à la fonction plot les points générés dans une figure.


In [55]:

```
#votre code
k=21
a=normale(k, 1)
b=(1,2)
#print(b[1])
print(a[0])
print(a[1])
plt.title('fonction normale sur '+str(k)+' valeurs')
plt.plot(a[0], a[1])
plt.show()
```

```
[-2.  -1.8 -1.6 -1.4 -1.2 -1.  -0.8 -0.6 -0.4 -0.2  0.   0.2  0.4
 0.6
  0.8  1.   1.2  1.4  1.6  1.8  2. ]
[0.05399097 0.07895016 0.11092083 0.14972747 0.19418605 0.24197072
 0.28969155 0.3332246  0.36827014 0.39104269 0.39894228 0.39104269
 0.36827014 0.3332246  0.28969155 0.24197072 0.19418605 0.14972747
 0.11092083 0.07895016 0.05399097]
```



## II.2- Distribution de probabilité affine

Dans cette question, on considère une généralisation de la distribution uniforme: une  Distribution affine, c'est-à-dire que la fonction de densité est une droite, mais pas forcément horizontale, comme le montre la figure ci-contre.

Écrire une fonction `proba_affine : int , float -> float np.array` qui, comme dans la question précédente, va générer un ensemble de  $k$  points  $y_i, i = 0, \dots, k - 1$ , représentant cette distribution (paramétrée par sa pente `slope`). On vérifiera ici aussi que l'entier  $k$  est impair. Si la pente est égale à 0, c'est-à-dire si la distribution est uniforme, chaque point  $y_i$  devrait être égal à  $\frac{1}{k}$  (afin que  $\sum y_i = 1$ ). Si la pente est différente de 0, il suffit de choisir,  $\forall i = 0, \dots, k - 1$ ,

$$y_i = \frac{1}{k} + \left(i - \frac{k-1}{2}\right) \times slope$$

Vous pourrez aisément vérifier que, ici aussi,  $\sum y_i = 1$ . Afin que la distribution soit toujours positive (c'est quand même un minimum pour une distribution de probabilité), il faut que la pente `slope` ne soit ni trop grande ni trop petite. Le bout de code ci-dessous lèvera une exception si la pente est trop élevée et indiquera la pente maximale possible.

In [58]:

```
def proba_affine ( k, slope ):
    if k % 2 == 0:
        raise ValueError ( 'le nombre k doit etre impair' )
    if abs ( slope ) > 2. / ( k * k ):
        raise ValueError ( 'la pente est trop raide : pente max = ' +
            str ( 2. / ( k * k ) ) )
    #votre code
    i=np.linspace(0, k-1, k)
    return i,(1/k)+(i-(k-1)/2)*slope
```

In [63]:

```
res=proba_affine(11, 0)
print(res[0])
print(res[1])
print ( 'sum '+str(sum(res[1]))+ ' is approximatively equal to 1' )

[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[0.09090909 0.09090909 0.09090909 0.09090909 0.09090909 0.09090909
 0.09090909 0.09090909 0.09090909 0.09090909 0.09090909]
sum 1.0000000000000002 is approximatively equal to 1
```

## II.3- Distribution jointe

Écrire une fonction `Pxy : float np.array , float np.array -> float np.2D-array` qui, étant donné deux tableaux numpy de nombres réels à 1 dimension générés par les fonctions des questions précédentes et représentant deux distributions de probabilités  $P(A)$  et  $P(B)$ , renvoie la distribution jointe  $P(A, B)$  sous forme d'un tableau numpy à 2 dimensions de nombres réels, en supposant que  $A$  et  $B$  sont des variables aléatoires indépendantes. Par exemple, si:

In [67]:

```
PA = np.array ( [0.2, 0.7, 0.1] )
PB = np.array ( [0.4, 0.4, 0.2] )
```

alors Pxy(A,B) renverra le tableau :

```
np.array([[ 0.08,  0.08,  0.04],
          [ 0.28,  0.28,  0.14],
          [ 0.04,  0.04,  0.02]])
```

In [138]:

```
# si A et B sont des variables aléatoires indépendantes
# alors P(A, B)=P(A)*P(B)
```

```
def Pxy(x,y):
    #votre code
    sz1=np.size(x)
    a=np.reshape(x, (1,sz1))
    sz2=np.size(y)
    b=np.reshape(y, (1,sz2))
    #return x.dot(y)
    return np.transpose(a).dot(b)
    pass

''' autre définition possible
def Pxy(x, y):
    result=np.zeros((np.size(x), np.size(y)))
    for i in range(np.size(x)):
        for j in range (np.size(y)):
            result[i, j]=x[i]*y[j]
    return result
'''
```

Out[138]:


```
' autre définition possible\ndef Pxy(x, y):\n    result=np.zeros((n
p.size(x), np.size(y)))\n    for i in range(np.size(x)):\n        fo
r j in range (np.size(y)):\n            result[i, j]=x[i]*y[j]\n
return result\n'
```

In [139]:

```
print(Pxy(PA,PB))
```

```
[[0.08 0.08 0.04]
 [0.28 0.28 0.14]
 [0.04 0.04 0.02]]
```

## II.4- Affichage de la distribution jointe

Le code ci-dessous permet d'afficher en 3D une probabilité jointe générée par la fonction  Distribution jointe précédente. Exécutez-le avec une probabilité jointe résultant de la combinaison d'une loi normale et d'une distribution affine.

Si la commande `%matplotlib notebook` fonctionne, vous pouvez interagir avec la courbe. Si le contenu de la fenêtre est vide, redimensionnez celle-ci et le contenu devrait apparaître. Cliquez à la souris à l'intérieur de la fenêtre et bougez la souris en gardant le bouton appuyé afin de faire pivoter la courbe. Observez sous différents angles cette courbe. Refaites l'expérience avec une probabilité jointe résultant de deux lois normales. Essayez de comprendre ce que signifie, visuellement, l'indépendance probabiliste. Vous pouvez également recommencer l'expérience avec le logarithme des lois jointes.

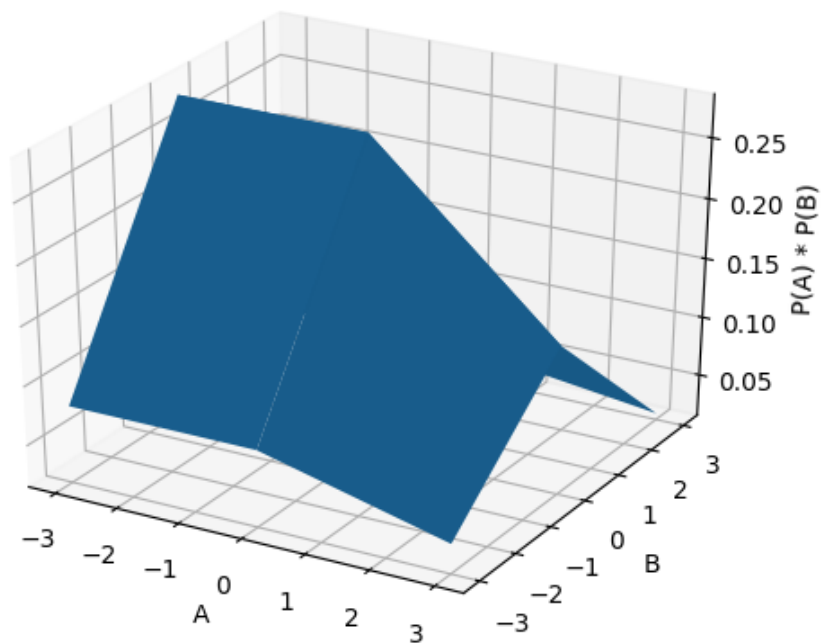
In [134]:

```
from mpl_toolkits.mplot3d import Axes3D
#%matplotlib inline
# essayer '%matplotlib notebook' pour interagir avec la visualisation 3D
%matplotlib notebook
def dessine ( P_jointe ) :
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    x = np.linspace ( -3, 3, P_jointe.shape[0] )
    y = np.linspace ( -3, 3, P_jointe.shape[1] )
    X, Y = np.meshgrid(x, y)
    ax.plot_surface(X, Y, P_jointe, rstride=1, cstride=1 )
    ax.set_xlabel('A')
    ax.set_ylabel('B')
    ax.set_zlabel('P(A) * P(B)')
    plt.show ()
```



In [135]:

```
dessine(np.array([[ 0.08,  0.08,  0.04],  
                  [ 0.28,  0.28,  0.14],  
                  [ 0.04,  0.04,  0.02]]))
```



In [151]:

```
#votre code  
# avec loi normale et loi affine  
Loi_normale=normale(101, 1)  
Loi_affine=proba_affine(101, 0)  
print(Loi_normale)  
print(Loi_affine)
```

```

(array([-2. , -1.96, -1.92, -1.88, -1.84, -1.8 , -1.76, -1.72, -1.6
8,
      -1.64, -1.6 , -1.56, -1.52, -1.48, -1.44, -1.4 , -1.36, -1.3
2,
      -1.28, -1.24, -1.2 , -1.16, -1.12, -1.08, -1.04, -1. , -0.9
6,
      -0.92, -0.88, -0.84, -0.8 , -0.76, -0.72, -0.68, -0.64, -0.6
,
      -0.56, -0.52, -0.48, -0.44, -0.4 , -0.36, -0.32, -0.28, -0.2
4,
      -0.2 , -0.16, -0.12, -0.08, -0.04, 0. , 0.04, 0.08, 0.1
2,
      0.16, 0.2 , 0.24, 0.28, 0.32, 0.36, 0.4 , 0.44, 0.4
8,
      0.52, 0.56, 0.6 , 0.64, 0.68, 0.72, 0.76, 0.8 , 0.8
4,
      0.88, 0.92, 0.96, 1. , 1.04, 1.08, 1.12, 1.16, 1.2
,
      1.24, 1.28, 1.32, 1.36, 1.4 , 1.44, 1.48, 1.52, 1.5
6,
      1.6 , 1.64, 1.68, 1.72, 1.76, 1.8 , 1.84, 1.88, 1.9
2,
      1.96, 2. ]), array([0.05399097, 0.05844094, 0.06315656, 0.
06814357, 0.07340681,
      0.07895016, 0.08477636, 0.09088698, 0.09728227, 0.1039611 ,
      0.11092083, 0.1181573 , 0.12566464, 0.1334353 , 0.14145997,
      0.14972747, 0.15822479, 0.16693704, 0.17584743, 0.18493728,
      0.19418605, 0.20357139, 0.21306915, 0.2226535 , 0.232297 ,
      0.24197072, 0.25164434, 0.2612863 , 0.27086397, 0.28034381,
      0.28969155, 0.29887241, 0.30785126, 0.31659291, 0.32506226,
      0.3332246 , 0.34104579, 0.34849251, 0.35553253, 0.36213488,
      0.36827014, 0.37391061, 0.37903053, 0.38360629, 0.38761662,
      0.39104269, 0.39386836, 0.39608021, 0.39766771, 0.39862325,
      0.39894228, 0.39862325, 0.39766771, 0.39608021, 0.39386836,
      0.39104269, 0.38761662, 0.38360629, 0.37903053, 0.37391061,
      0.36827014, 0.36213488, 0.35553253, 0.34849251, 0.34104579,
      0.3332246 , 0.32506226, 0.31659291, 0.30785126, 0.29887241,
      0.28969155, 0.28034381, 0.27086397, 0.2612863 , 0.25164434,
      0.24197072, 0.232297 , 0.2226535 , 0.21306915, 0.20357139,
      0.19418605, 0.18493728, 0.17584743, 0.16693704, 0.15822479,
      0.14972747, 0.14145997, 0.1334353 , 0.12566464, 0.1181573 ,
      0.11092083, 0.1039611 , 0.09728227, 0.09088698, 0.08477636,
      0.07895016, 0.07340681, 0.06814357, 0.06315656, 0.05844094,
      0.05399097]))
(array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.,
10.,
      11., 12., 13., 14., 15., 16., 17., 18., 19., 20.,
21.,
      22., 23., 24., 25., 26., 27., 28., 29., 30., 31.,
32.,
      33., 34., 35., 36., 37., 38., 39., 40., 41., 42.,
43.,
      44., 45., 46., 47., 48., 49., 50., 51., 52., 53.,
54.,
      55., 56., 57., 58., 59., 60., 61., 62., 63., 64.,
65.,
      66., 67., 68., 69., 70., 71., 72., 73., 74., 75.,
76.,
      77., 78., 79., 80., 81., 82., 83., 84., 85., 86.,
87.,
      88., 89., 90., 91., 92., 93., 94., 95., 96., 97.,

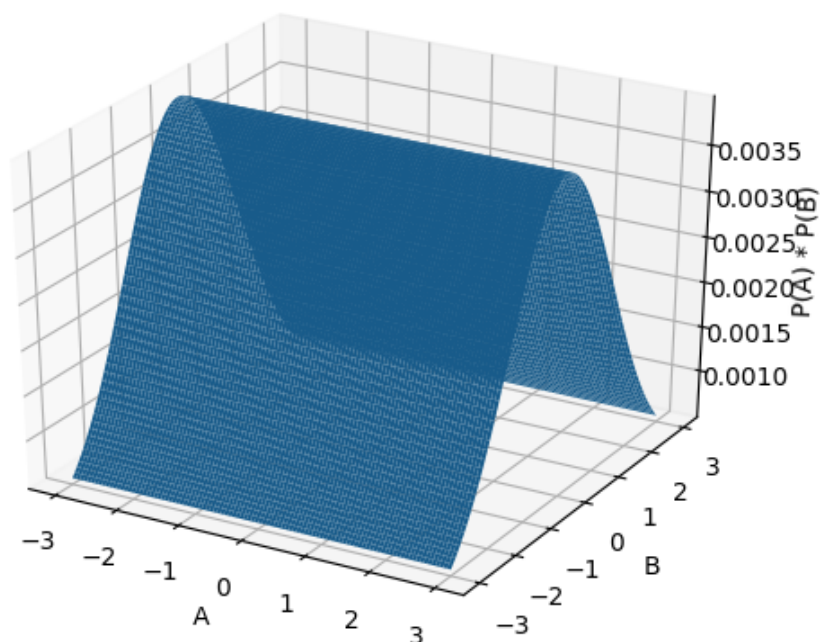
```

In [152]:

```
[[0.00053456 0.00053456 0.00053456 ... 0.00053456 0.00053456 0.00053456]
 [0.00057862 0.00057862 0.00057862 ... 0.00057862 0.00057862 0.00057862]
 [0.00062531 0.00062531 0.00062531 ... 0.00062531 0.00062531 0.00062531]
 ...
 [0.00062531 0.00062531 0.00062531 ... 0.00062531 0.00062531 0.00062531]
 [0.00057862 0.00057862 0.00057862 ... 0.00057862 0.00057862 0.00057862]
 [0.00053456 0.00053456 0.00053456 ... 0.00053456 0.00053456 0.00053456]]
```

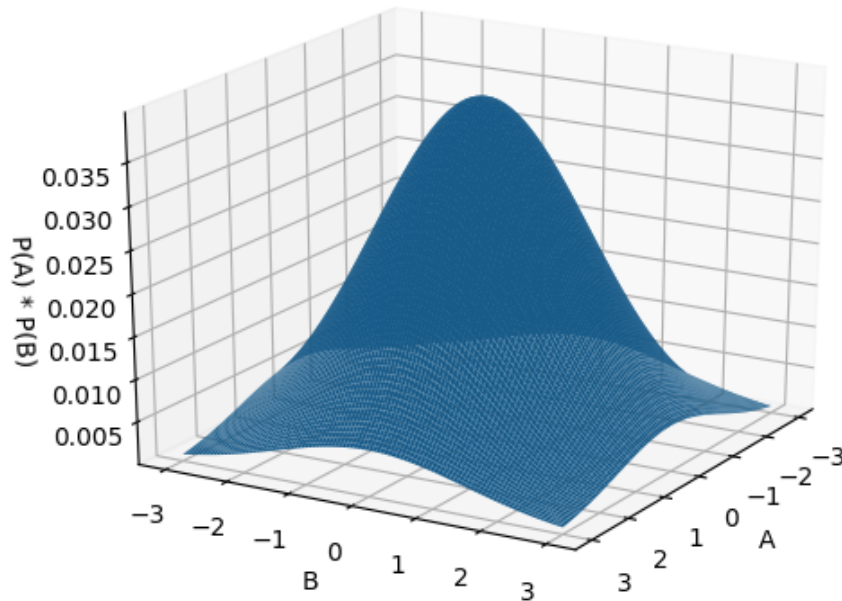
In [153]:

```
#test  
%matplotlib notebook  
dessine(Loi_jointe)
```



In [149]:

```
# avec deux lois normales
Loi_normale1=normale(101, 1)
Loi_normale2=normale(101, 4)
Loi_jointe2=Pxy(Loi_normale1[1], Loi_normale2[1])
%matplotlib notebook
dessine(Loi_jointe2)
```

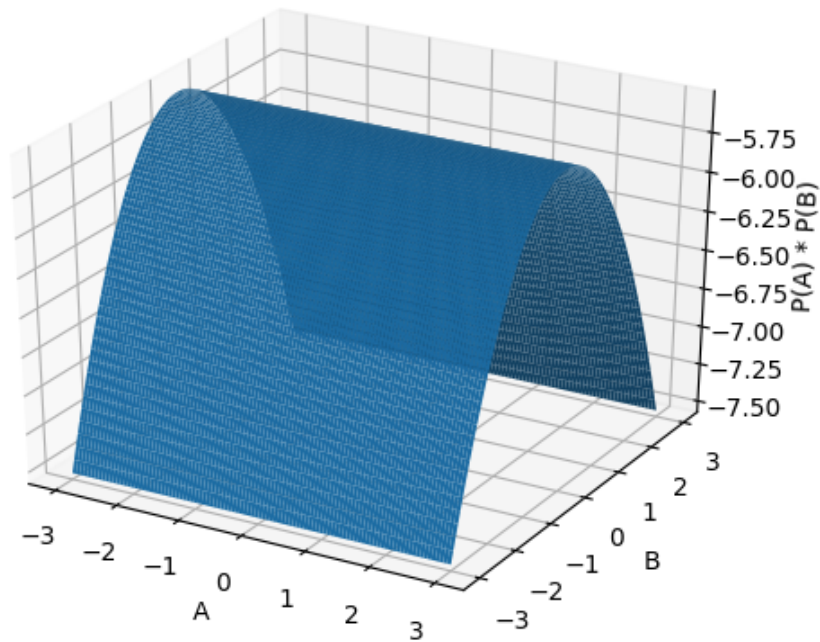


### observation sur la signification visuelle de l'indépendance probabiliste

quand deux variables sont indépendantes, on remarque qu'on a la même forme qui se répète selon un des axes avec un scale différent

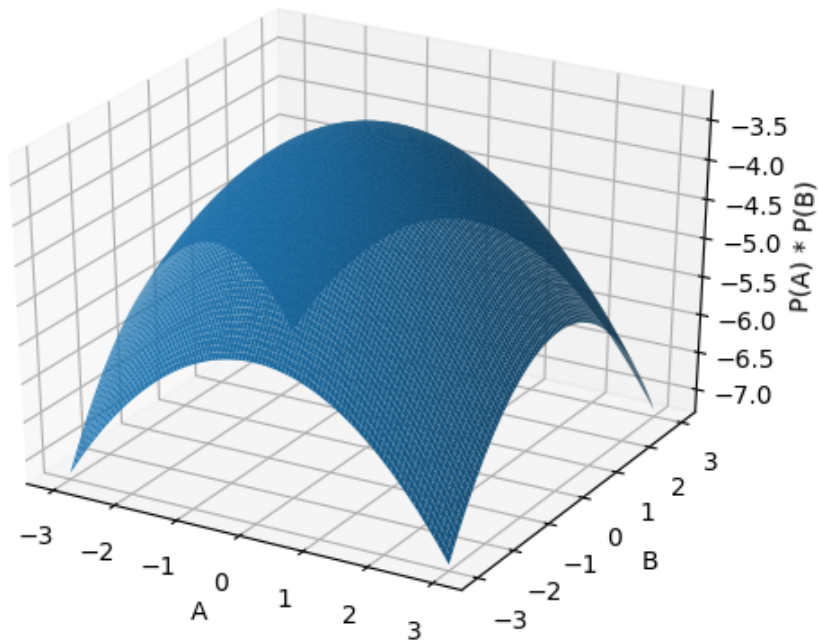
In [154]:

```
#experience avec le logarithme des lois jointes  
dessine(np.log(Loi_jointe))
```



In [155]:

```
dessine(np.log(Loi_jointe2))
```



### III- Indépendances conditionnelles ( **obligatoire** )

Dans cet exercice, on considère quatre variables aléatoires booléennes  $X$ ,  $Y$ ,  $Z$  et  $T$  ainsi que leur distribution jointe  $P(X, Y, Z, T)$  encodée en python de la manière suivante :



In [14]:

```
# creation de P(X,Y,Z,T)
P_XYZT = np.array([[[[ 0.0192,  0.1728],
                      [ 0.0384,  0.0096]],

                    [[ 0.0768,  0.0512],
                      [ 0.016 ,  0.016 ]]],

                  [[[ 0.0144,  0.1296],
                      [ 0.0288,  0.0072]],

                    [[ 0.2016,  0.1344],
                      [ 0.042 ,  0.042 ]]]])
```

Ainsi,  $\forall (x, y, z, t) \in \{0, 1\}^4$ ,  $P\_XYZT[x][y][z][t]$  correspond à  $P(X = x, Y = y, Z = z, T = t)$  ou, en version abrégée, à  $P(x, y, z, t)$ .

### III.1- Indépendance de X et T conditionnellement à (Y,Z)

On souhaite tester si les variables aléatoires  $X$  et  $T$  sont indépendantes conditionnellement à  $(Y, Z)$ . Il s'agit donc de vérifier que dans la loi  $P$ ,

$$P(X, T|Y, Z) = P(X|Y, Z) \cdot P(T|Y, Z)$$

Pour cela, tout d'abord, calculer à partir de  $P\_XYZT$  le tableau  $P\_YZ$  représentant la distribution  $P(Y, Z)$ . On rappelle que

$$P(Y, Z) = \sum_{X, T} P(X, Y, Z, T)$$

Le tableau  $P\_YZ$  est donc un tableau à deux dimensions, dont la première correspond à  $Y$  et la deuxième à  $Z$ . Si vous ne vous êtes pas trompé(e)s, vous devez obtenir le tableau suivant :

```
np.array([[ 0.336,  0.084],
          [ 0.464,  0.116]])
```

Ainsi  $P(Y = 0, Z = 1) = P\_YZ[0][1] = 0.084$

In [15]:

```
#votre code
```

Ensuite, calculer le tableau  $P\_XTcondYZ$  représentant la distribution  $P(X, T|Y, Z)$ . Ce tableau a donc 4 dimensions, chacune correspondant à une des variables aléatoires. De plus, les valeurs de  $P\_XTcondYZ$  sont obtenues en utilisant la formule des probabilités conditionnelles:

$$P(X, T|Y, Z) = \frac{P(X, Y, Z, T)}{P(Y, Z)}$$

In [16]:

```
#votre code
```

Calculer à partir de  $P_{XTcondYZ}$  les tableaux à 3 dimensions  $P_{XcondYZ}$  et  $P_{TcondYZ}$  représentant respectivement les distributions  $P(X|Y, Z)$  et  $P(T|Y, Z)$ . On rappelle que

$$P(X|Y, Z) = \sum_Y P(X, T|Y, Z)$$

In [17]:

```
#votre code
```

Enfin, tester si  $X$  et  $T$  sont indépendantes conditionnellement à  $(Y, Z)$ : si c'est bien le cas, on doit avoir

$$P(X, T|Y, Z) = P(X|Y, Z) \times P(T|Y, Z)$$

### III.2- Indépendance de X et (Y,Z)

On souhaite maintenant déterminer si  $X$  et  $(Y, Z)$  sont indépendantes. Pour cela, commencer par calculer à partir de  $P_{XYZT}$  le tableau  $P_{XYZ}$  représentant la distribution  $P(X, Y, Z)$ .

Ensuite, calculer à partir de  $P_{XYZ}$  les tableaux  $P_X$  et  $P_{YZ}$  représentant respectivement les distributions  $P(X)$  et  $P(Y, Z)$ . On rappelle que

$$P(X) = \sum_Y \sum_Z P(X, Y, Z)$$

Si vous ne vous êtes pas trompé(e),  $P_X$  doit être égal au tableau suivant :

```
np.array([ 0.4,  0.6])
```

In [18]:

```
#votre code
```

Enfin, si  $X$  et  $(Y, Z)$  sont bien indépendantes, on doit avoir

$$P(X, Y, Z) = P(X) \times P(Y, Z)$$

In [19]:

```
#votre code
```

## IV- Indépendances conditionnelles et consommation mémoire (obligatoire)

Le but de cet exercice est d'exploiter les probabilités conditionnelles et les indépendances conditionnelles afin de décomposer une probabilité jointe en un produit de "petites probabilités conditionnelles". Cela permet de stocker des probabilités jointes de grandes tailles sur des ordinateurs "standards". Au cours de l'exercice, vous allez donc partir d'une probabilité jointe et, progressivement, construire un programme qui identifie ces indépendances conditionnelles.

Pour simplifier, dans la suite de cet exercice, nous allons considérer un ensemble  $X_0, \dots, X_n$  de variables aléatoires binaires (elles ne peuvent prendre que 2 valeurs : 0 et 1).

### Simplification du code : utilisation de pyAgrum

Manipuler des probabilités et des opérations sur des probabilités complexes est difficile avec les outils classiques. La difficulté principale est certainement le problème du mapping entre axe et variable aléatoire.

pyAgrum propose une gestion de `Potential` qui sont des tableaux multidimensionnels dont les axes sont caractérisés par des variables et sont donc non ambigus.

Par exemple, après l'initiation du `Potential` PABCD :

In [20]:

```
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

X,Y,Z,T=[gum.LabelizedVariable(x,x,2) for x in "XYZT"]
pXYZT=gum.Potential().add(T).add(Z).add(Y).add(X)
pXYZT[:]=[[[ [ 0.0192, 0.1728],
               [ 0.0384, 0.0096]],
            [ [ 0.0768, 0.0512],
               [ 0.016 , 0.016 ]]],
           [[ [ 0.0144, 0.1296],
               [ 0.0288, 0.0072]],
            [ [ 0.2016, 0.1344],
               [ 0.042 , 0.042 ]]]]
```

On peut alors utiliser la méthode `margSumOut` qui supprime les variables par sommations:

`p.margSumOut(['X','Y'])` correspond à calculer  $\sum_{X,Y} p$

La réponse a question III.1 se calcule donc ainsi :

In [21]:

```
pXT_YZ=pXYZT/pXYZT.margSumOut(['X','T'])
pX_YZ=pXT_YZ.margSumOut(['T'])
pT_YZ=pXT_YZ.margSumOut(['X'])

if pXT_YZ==pX_YZ*pT_YZ:
    print("=> X et T sont indépendants conditionnellement à Y et Z")
else:
    print("=> pas d'indépendance trouvée")
```

=> X et T sont indépendants conditionnellement à Y et Z

La réponse à la question III.2 se calcule ainsi :

In [22]:

```
pXYZ=pXYZT.margSumOut("T")
pYZ=pXYZ.margSumOut("X")
pX=pXYZ.margSumOut(["Y","Z"])
if pXYZ==pX*pYZ:
    print("=> X et YZ sont indépendants")
else:
    print("=> pas d'indépendance trouvée")
```

=> pas d'indépendance trouvée

In [23]:

```
gnb.sideBySide(pXYZ,pX,pYZ,pX*pYZ,
               captions=['$P(X,Y,Z)$', '$P(X)$', '$P(Y,Z)$', '$P(X)\cdot P(Y,Z)$'])
```

		Z	
X	Y	0	1
0	0	0.1920	0.0480
	1	0.1280	0.0320
1	0	0.1440	0.0360
	1	0.3360	0.0840

$P(X,Y,Z)$

X	
0	1
0.4000	0.6000

$P(X)$

	Z	
Y	0	1
0	0.3360	0.0840
1	0.4640	0.1160

$P(Y,Z)$

		Z	
X	Y	0	1
0	0	0.1344	0.0336
	1	0.1856	0.0464
1	0	0.2016	0.0504
	1	0.2784	0.0696

$P(X) \cdot P(Y,Z)$

asia.txt contient la description d'une probabilité jointe sur un ensemble de 8 variables aléatoires binaires (256 paramètres). Le fichier est produit à partir du site web suivant

<http://www.bnlearn.com/bnrepository/> .

Le code suivant permet de lire ce fichier et d'en récupérer la probabilité jointe (sous forme d'une `gum.Potential` ) qu'il contient :

In [24]:

```
def read_file ( filename ) :
    """
    Renvoie les variables aléatoires et la probabilité contenues dans le
    fichier dont le nom est passé en argument.
    """
    Pres = gum.Potential ()
    vars=[]

    with open ( filename, 'r' ) as fic:
        # on rajoute les variables dans le potentiel
        nb_vars = int ( fic.readline () )
        for i in range ( nb_vars ) :
            name, domsize = fic.readline ().split ()
            vars.append(name)
            variable = gum.LabelizedVariable(name,name,int (domsize))
            Pres.add(variable)

        # on rajoute les valeurs de proba dans le potentiel
        cpt = []
        for line in fic:
            cpt.append ( float(line) )
        Pres.fillWith( cpt )
    return vars,Pres

vars,Pjointe=read_file('asia.txt')
# afficher Pjointe est un peu délicat (retire le commentaire de la ligne suivant
e)
# Pjointe

print('Les variables : '+str(vars))
```

Les variables : ['visit\_to\_Asia?', 'tuberculosis?', 'smoking?', 'lung\_cancer?', 'tuberculosis\_or\_lung\_cancer?', 'bronchitis?', 'positive\_Xray?', 'dyspnoea?']

In [25]:

```
# Noter qu'il existe une fonction margSumIn qui, à l'inverse de MargSumOut, élimine
# toutes les variables qui ne sont pas dans les arguments
Pjointe.margSumIn(['tuberculosis?','lung_cancer?'])
```

Out[25]:

	tuberculosis?	
lung_cancer?	0	1
0	0.0006	0.0544
1	0.0098	0.9352

## IV.1- test d'indépendance conditionnelle

En utilisant la méthode `margSumIn` (voir juste au dessus), écrire une fonction `conditional_indep: Potential, str, str, list[str] -> bool` qui rend vrai si dans le `Potential`, on peut lire l'indépendance conditionnelle.

Par exemple, l'appel

```
conditional_indep(Pjointe, 'bronchitis?', 'positive_Xray?',
['tuberculosis?', 'lung_cancer?'])
```

vérifie si `bronchitis` est indépendant de `positive_Xray` conditionnellement à `tuberculosis?` et `lung_cancer?`

D'un point de vue général, on vérifie que  $X$  et  $Y$  sont indépendants conditionnellement à  $Z_1, \dots, Z_d$  par l'égalité :

$$P(X, Y | Z_1, \dots, Z_d) = P(X | Z_1, \dots, Z_d) \cdot P(Y | Z_1, \dots, Z_d)$$

Ces trois probabilités sont calculables à partir de la loi jointe de  $P(X, Y, Z_1, \dots, Z_d)$ .

*Remarque* Vérifier l'égalité  $P=Q$  de 2 `Potential` peut être problématique si les 2 sont des résultats de calcul : il peut exister une petite variation. Un meilleur test est de vérifier  $(P-Q).abs().max() < \epsilon$  avec `epsilon` assez petit.

In [26]:

```
def conditional_indep(P, X, Y, Zs):
    #votre code
    pass
```

In [27]:

```
conditional_indep(Pjointe,
                  'bronchitis?',
                  'positive_Xray?',
                  ['tuberculosis?', 'lung_cancer?'])
```

In [28]:

```
conditional_indep(Pjointe,
                  'bronchitis?',
                  'visit_to_ASia?',
                  [])
```

## IV.2- Factorisation compacte de loi jointe

On sait que si un ensemble de variables aléatoires  $\mathcal{S} = \{X_{i_0}, \dots, X_{i_{n-1}}\}$  peut être partitionné en deux sous-ensembles  $\mathcal{K}$  et  $\mathcal{L}$  (c'est-à-dire tels que  $\mathcal{K} \cap \mathcal{L} = \emptyset$  et  $\mathcal{K} \cup \mathcal{L} = \{X_{i_0}, \dots, X_{i_{n-1}}\}$ ) tels qu'une variable  $X_{i_n}$  est indépendante de  $\mathcal{L}$  conditionnellement à  $\mathcal{K}$ , alors:

$$P(X_{i_n} | X_{i_0}, \dots, X_{i_{n-1}}) = P(X_{i_n} | \mathcal{K}, \mathcal{L}) = P(X_{i_n} | \mathcal{K})$$

C'est ce que nous avons vu au cours n°2 (cf. définition des probabilités conditionnelles). Cette formule est intéressante car elle permet de réduire la taille mémoire consommée pour stocker  $P(X_{i_n} | X_{i_0}, \dots, X_{i_{n-1}})$ : il suffit en effet de stocker uniquement  $P(X_{i_n} | \mathcal{K})$  pour obtenir la même information.

Écrire une fonction `compact_conditional_proba: Potential, str -> Potential` qui, étant donné une probabilité jointe  $P(X_{i_0}, \dots, X_{i_n})$ , une variable aléatoire  $X_{i_n}$ , retourne cette probabilité conditionnelle  $P(X_{i_n} | \mathcal{K})$ . Pour cela, nous vous proposons l'algorithme itératif suivant:

```
K=S
Pour tout X in K:
    Si X indépendante de Xin conditionnellement à K\{X} alors
        Supprimer X de K
retourner P(Xin|K)$
```

Trois petites aides :

1- La fonction précédente `conditional_indep` devrait vous servir...

2- Obtenir la liste des noms des variables dans un `Potential` se fait par l'attribut

`P.var_names`

3- Afin que l'affichage soit plus facile à comprendre, il peut être judicieux de placer la variable  $X_{i_n}$  en premier dans la liste des variables du `Potential`, ce que l'on peut faire avec le code suivant :

```
proba = proba.putFirst(Xin)
```

In [29]:

```
def compact_conditional_proba(P,X):
    #votre code
    pass
```

In [30]:

```
compact_conditional_proba(Pjointe,"visit_to_Asia?")
```

In [31]:

```
compact_conditional_proba(Pjointe,"dyspnoea?")
```

### IV.3- Création d'un réseau bayésien

Un réseau bayésien est simplement la décomposition d'une distribution de probabilité jointe en un produit de probabilités conditionnelles: vous avez vu en cours que  $P(A, B) = P(A|B)P(B)$ , et ce quel que soient les ensembles de variables aléatoires disjoints  $A$  et  $B$ . En posant  $A = X_n$  et  $B = \{X_0, \dots, X_{n-1}\}$ , on obtient donc:

$$P(X_0, \dots, X_n) = P(X_n | X_0, \dots, X_{n-1}) P(X_0, \dots, X_{n-1})$$

On peut réitérer cette opération pour le terme de droite en posant  $A = X_{n-1}$  et  $B = \{X_0, \dots, X_{n-2}\}$ , et ainsi de suite. Donc, par récurrence, on a:

$$P(X_0, \dots, X_n) = P(X_0) \times \prod_{i=1}^n P(X_i | X_0, \dots, X_{i-1})$$

Si on applique à chaque terme  $P(X_i | X_0, \dots, X_{i-1})$  la fonction `compact_conditional_proba`, on obtient une décomposition:

$$P(X_0, \dots, X_n) = P(X_0) \times \prod_{i=1}^n P(X_i | \mathcal{K}_i)$$

avec  $\mathcal{K}_i \subseteq \{X_0, \dots, X_{i-1}\}$ . Cette décomposition est dite "compacte" car son stockage nécessite en pratique beaucoup moins de mémoire que celui de la distribution jointe. C'est ce que l'on appelle un réseau bayésien.

Écrire une fonction `create_bayesian_network` : `Potential` -> `Potential list` qui, étant donné une probabilité jointe, vous renvoie la liste des  $P(X_i | \mathcal{K}_i)$ . Pour cela, il vous suffit d'appliquer l'algorithme suivant:

```
liste = []
P = P(X_0, ..., X_n)
Pour i de n à 0 faire:
    calculer Q = compact_conditional_proba(P, X_i)
    afficher la liste des variables de Q
    rajouter Q à liste
    supprimer X_i de P par marginalisation

retourner liste
```

Il est intéressant ici de noter les affichages des variables de Q: comme toutes les variables sont binaires, Q nécessite uniquement (2 puissance le nombre de ces variables) nombres réels. Ainsi une probabilité sur 3 variables ne nécessite que  $\{2^3 = 8\}$  nombres réels.

In [32]:

```
def create_bayesian_network(P):
    #votre code
    pass
```

In [33]:

```
create_bayesian_network(Pjointe)
```



## IV.4- Gain en compression

On souhaite observer le gain en termes de consommation mémoire obtenu par votre décomposition. Si `P` est un `Potential`, alors `P.toarray().size` est égal à la taille (le nombre de paramètres) de la table `P`. Calculez donc le nombre de paramètres nécessaires pour stocker la probabilité jointe lue dans le fichier `asia.txt` ainsi que la somme des nombres de paramètres des tables que vous avez créées grâce à votre fonction `create_bayesian_network`.

In [34]:

```
# votre code
```

## V- Applications pratiques (optionnelle)

La technique de décomposition que vous avez vue est effectivement utilisée en pratique. Vous pouvez voir le gain que l'on peut obtenir sur différentes distributions de probabilité du site :

<http://www.bnlearn.com/bnrepository/> (<http://www.bnlearn.com/bnrepository/>)

Cliquez sur le nom du dataset que vous voulez visualiser et téléchargez son `.bif` ou `.dsl`. Afin de visualiser le contenu du fichier, vous allez utiliser `pyAgrum`. Le code suivant vous permettra alors de visualiser votre dataset: la valeur indiquée après "domainSize" est la taille de la probabilité jointe d'origine (en nombre de paramètres) et celle après "dim" est la taille de la probabilité sous forme compacte (somme des tailles des probabilités conditionnelles compactes).

In [35]:

```
# chargement de pyAgrum
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb

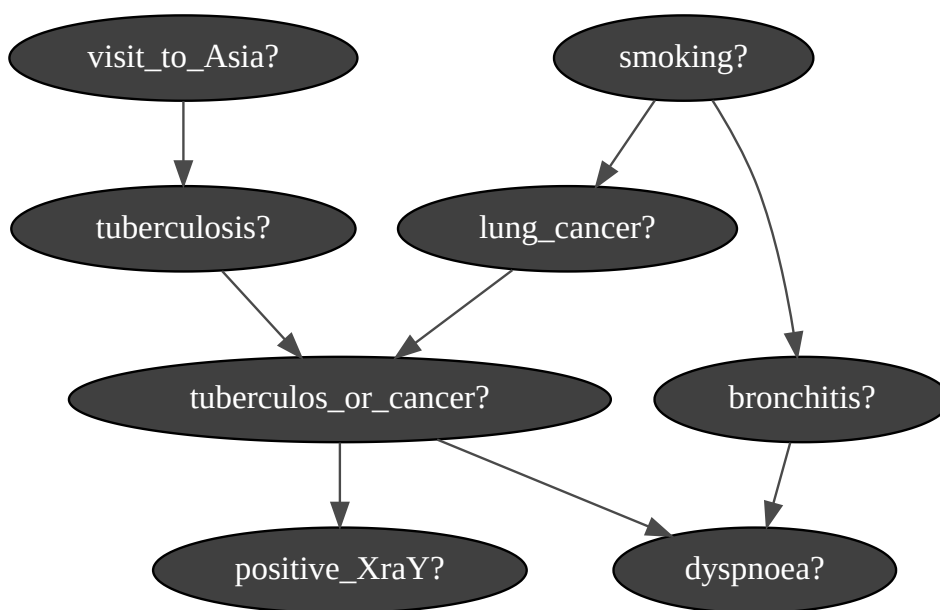
# chargement du fichier bif ou dsl
bn = gum.loadBN ( "asia.bif" )

# affichage de la taille des probabilités jointes compacte et non compacte
print(bn)

# affichage graphique du réseau bayésien
bn
```

BN{nodes: 8, arcs: 8, domainSize: 256, dim: 36}

Out[35]:



In [ ]: