Laboratoire #1

INF8225 - Intelligence artificielle : techniques probabilistes et d'apprentissage

Hiver 2017
Département de génie Informatique
École Polytechnique de Montréal

Dernière mise à jour: 6 février 2017

Gabriel Lepetit-Aimon 1865327

Introduction

Ce laboratoire est une introduction à l'utilisation des réseaux de Bayes et des Factors Graphs pour le calcul de probabilités jointes puis marginales et enfin conditionnelles. Dans un premier temps, nous étudierons la dépendance marginale et conditionnelle entre les variables d'un même graphe. Nous détaillerons trois phénomènes issus de ces dépendances. Dans un second temps, nous nous concentrerons sur un réseau de Bayes donné nottament pour en extraire sa distribution. Enfin, nous implémenterons l'algorithme *sum-product* et recalculerons toutes ces probabilités...

J'ai choisi de coder ce LAB en Python puisque c'est le langage que je devrai utiliser pour mener à bien mon projet de maitrise. N'ayant jamais réellement codé en Python auparavant, j'ai donc saisi cette opportunité pour me familiariser avec ce langage. Par conséquent, certaine des fonctionnalités que je propose dépassent le cadre du TP : réseaux et variables aléatoires quelconques (pas forcément binaires). Ce rapport simplifie donc parfois mon code original (qui est joint à ce rapport) en se concentrant sur l'idée générale derrière les algorithmes sans s'attarder sur mon implémentation. Une courte documentation de l'interface de ces packages est cependant disponible en annexe.

Question 1

Explaining Away

Le phénomène de explaining away résulte de la différence entre probabilités conditionnelle et marginale. Prenons le réseau de Bayes simple présenté sur la figure 1 et générer par le code suivant (voir Annexe B pour plus de détail sur la création de réseau) :

source: question1.py

```
4 explAwayNet = bn.Network("Explaining Away Net")
5 A = explAwayNet.createNode('A', 0.27)
6 B = explAwayNet.createNode('B', 0.11)
7 C = explAwayNet.createNode('C', [0.9,0.7,0.6,0.1], parents=[A, B])
```

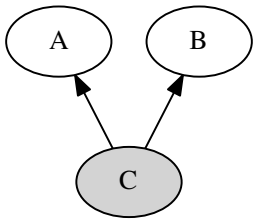

FIGURE 1 – Réseau Bayesien illustrant le phénomène *Explaining Away*

On peut facilement démontrer que A et B sont marginalement indépendantes :

$$\mathrm{P}(\mathsf{A},\mathsf{B}) = \sum_{\mathsf{C}} \left(\mathrm{P}(\mathsf{A})\,\mathrm{P}(\mathsf{B})\,\mathrm{P}(\mathsf{C}\,|\,\mathsf{A},\mathsf{B}) \right) = \mathrm{P}(\mathsf{A})\,\mathrm{P}(\mathsf{B}) \sum_{\mathsf{C}} \left(\mathrm{P}(\mathsf{C}\,|\,\mathsf{A},\mathsf{B}) \right) = \mathrm{P}(\mathsf{A})\,\mathrm{P}(\mathsf{B}) \times 1$$

$$\mathrm{P}(\mathsf{A},\mathsf{B}) = \mathrm{P}(\mathsf{A})\,\mathrm{P}(\mathsf{B})$$

Par contre ces deux variables ne sont pas conditionnellement indépendantes par rapport à C.

$$P(A,B\,|\,C) = \frac{P(A,B,C)}{P(C)} = \frac{P(A)\,P(B)\,P(C\,|\,A,B)}{P(C)} \qquad \text{donc} \qquad \frac{P(A)\,P(B)}{P(C)} = \frac{P(A,B\,|\,C)}{P(C\,|\,A,B)}$$

$$\begin{aligned}
P(A\,|\,C)\,P(B\,|\,C) &= \frac{P(A,C)}{P(C)} \times \frac{P(B,C)}{P(C)^2} \\
&= \frac{1}{P(C)^2} \times \sum_B \left(P(A)\,P(B)\,P(C\,|\,A,B)\right) \times \sum_A \left(P(A)\,P(B)\,P(C\,|\,A,B)\right) \\
&= \frac{1}{P(C)^2} \times P(A) \sum_B \left(P(C\,|\,A,B)\right) \times P(B) \sum_A \left(P(C\,|\,A,B)\right) \\
&= \frac{P(A)\,P(B)}{P(C)} \times \frac{1}{P(C)} \sum_B \left(P(C\,|\,A,B)\right) \sum_A \left(P(C\,|\,A,B)\right) \\
P(A\,|\,C)\,P(B\,|\,C) &= P(A,B\,|\,C) \times \frac{1}{P(C)\,P(C\,|\,A,B)} \sum_B \left(P(C\,|\,A,B)\right) \sum_A \left(P(C\,|\,A,B)\right)
\end{aligned}$$

On a donc bien : $P(A\,|\,C)\,P(B\,|\,C) \neq P(A,B\,|\,C)$. Ce résultat est vérifiable numériquement :

source: question1.py

```
18 p = explAwayNet.p(A==True, C==True) * explAwayNet.p(B==True, C==True)
19 print("P(A|C)*P(B|C)=" + str(p))
20 p = explAwayNet.p((A==True)&(B==True), C==True)
21 print("P(A,B|C)=" + str(p))
```

```
P(A|C)*P(B|C)=0.166149151515
P(A,B|C)=0.0915128898627
```

En pratique, cette dépendance conditionnelle entraine : $P(A\,|\,C) \neq P(A\,|\,C,B)$. C'est ce phénomène que l'on appelle *Explaining Away* et qui est visible sur le réseau précédent : *source:* question1.py

```
10 # Pour l'explication de l'algorithme de calcul, voir la question 2
11 p = explAwayNet.p(A==True)
12 print("P(A)=" + str(round(p,4)*100) + '%')
13 p = explAwayNet.p(A==True, C==True)
14 print("P(A|C)=" + str(round(p,4)*100) + '%')
15 p = explAwayNet.p(A==True, (C==True)&(B==True))
16 print("P(A|C,B)=" + str(round(p,4)*100) + '%')
```

```
P(A)=27.0%
P(A|C)=58.51%
P(A|C,B)=32.23%
```

On peut retrouver ce résultat intuitivement. Si deux maladies A et B ont le même symptôme C et que ce symptome est observé ($C == True$) il est probable qu'au moins une des deux maladies est été contractée. Autrement dit la probabilité qu'un patient soit atteint d'une maladie augmente si il en possède les symptomes ($P(A\,|\,C) > P(A)$). Mais si on est en plus certain que le patient a contracté la deuxième maladie ($B == True$), l'observation des symptomes est expliquée, et la probabilité qu'il ai aussi contracté la première maladie diminue ($P(A\,|\,C,B) < P(C,B)$). Après tout, il faut vraiment qu'il soit malchanceux pour avoir contracté les deux maladies simultanément...

Serial Blocking

Étudions maintenant un autre réseau simple présenté sur la figure 2 et générer par le code suivant :

source: question1.py

```
27 serialBlockNet = bn.Network("Serial Blocking Net")
28 A = serialBlockNet.createNode('A', 0.80)
29 B = serialBlockNet.createNode('B', [0.99, 0.13], parents=A)
30 C = serialBlockNet.createNode('C', [0.2, 0.7], parents=B)
```

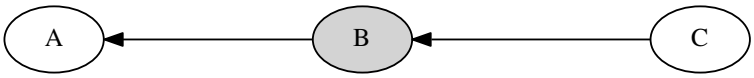

FIGURE 2 – Réseau Bayesien illustrant le phénomène *Serial Blocking*

Cette fois A et C sont bien conditionnellement indépendantes par rapport à B. En effet, on a :

$$P(A,C\,|\,B) = \frac{P(A,B,C)}{P(B)} = \frac{P(A)\,P(B\,|\,A)\,P(C\,|\,B)}{P(B)}$$

$$P(A\,|\,B)\,P(C\,|\,B) = \frac{P(A,B)}{P(B)} \times \frac{P(C,B)}{P(B)} = \frac{\sum_C (\,P(A)\,P(B\,|\,A)\,P(C\,|\,B)\,)}{P(B)} \times \frac{\sum_A (\,P(A)\,P(B\,|\,A)\,P(C\,|\,B)\,)}{\sum_A \left(P(A)\,P(B\,|\,A) \sum_C (\,P(C\,|\,B)\,) \right)}$$

$$= \frac{P(A)\,P(B\,|\,A) \sum_C (\,P(C\,|\,B)\,)}{P(B)} \times P(C\,|\,B) \frac{\sum_A (\,P(A)\,P(B\,|\,A)\,)}{\sum_A (\,P(A)\,P(B\,|\,A)\,)}$$

$$\text{donc} \qquad P(A\,|\,B)\,P(C\,|\,B) = \frac{P(A)\,P(B\,|\,A)\,P(C\,|\,B)}{P(B)} = P(A,C\,|\,B)$$

Par définition, A et C sont donc bien conditionnellement indépendantes par rapport à B. Autrement dit, si B est observé, une observation sur C n'apporte aucune information sur la valeur de A.

Ce phénomène appelé *Serial Blocking*, est observable numériquement $P(A\,|\,B) = P(A\,|\,B,C)$:

source: question1.py

```
33 p = serialBlockNet.p(A==True, B==True)
34 print("P(A|B)=" + str(round(p,4)*100) + '%')
35 p = serialBlockNet.p(A==True, (C==True)&(B==True))
36 print("P(A|B,C)=" + str(round(p,4)*100) + '%')
```

```
P(A|B)=96.82%
P(A|B,C)=96.82%
```

On peut aussi vérifier que $P(A\,|\,B) \times P(C\,|\,B) = P(A,C\,|\,B)$:

source: question1.py

```
38 p = serialBlockNet.p(A==True, B==True) * serialBlockNet.p(C==True, B==True)
39 print("P(A|B)*P(C|B)=" + str(p))
40 p = serialBlockNet.p((A==True)&(C==True), B==True)
41 print("P(A,C|B)=" + str(p))
```

```
P(A|B)*P(C|B)=0.193643031785
P(A,C|B)=0.193643031785
```

Divergent Blocking

Le phénomène de *divergent blocking* est très simulaire au Serial Blocking, mais intervient lorsqu'on observe un parent commun de deux variables aléatoires. Un cas simple de cette configuration est présenté en figure 3 généré par le code : *source:* question1.py

```
46 serialBlockNet = bn.Network("Divergent Blocking Net")
47 B = serialBlockNet.createNode('B', 0.50)
48 A = serialBlockNet.createNode('A', [0.89, 0.17], parents=B)
49 C = serialBlockNet.createNode('C', [0.23, 0.72], parents=B)
```

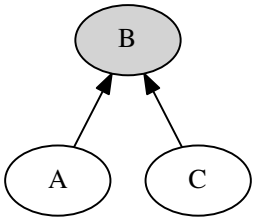

FIGURE 3 – Réseau Bayesien illustrant le phénomène *Divergent Blocking*

Comme pour le Serial Blocking on peut démontrer l'indépendance conditionnelle de A et C par rapport à B :

$$\mathrm{P}(\mathsf{A},\mathsf{C}\,|\,\mathsf{B}) = \frac{\mathrm{P}(\mathsf{A},\mathsf{B},\mathsf{C})}{\mathrm{P}(\mathsf{B})} = \frac{\mathrm{P}(\mathsf{A}\,|\,\mathsf{B})\,\mathrm{P}(\mathsf{B})\,\mathrm{P}(\mathsf{C}\,|\,\mathsf{B})}{\mathrm{P}(\mathsf{B})} = \mathrm{P}(\mathsf{A}\,|\,\mathsf{B})\,\mathrm{P}(\mathsf{C}\,|\,\mathsf{B})$$

De manière évidente on donc bien que l'observation d'un noeud isole ses enfants les uns des autres. L'observation de l'un deux n'apportera aucune information sur la probabilité d'occurence des autres. Cette invariance est vérifiable numériquement grâce à notre exemple :

source: question1.py

```
52 p = serialBlockNet.p(A==True, B==True)
53 print("P(A|B)=" + str(round(p,4)*100) + '%')
54 p = serialBlockNet.p(A==True, (C==True)&(B==True))
55 print("P(A|B,C)=" + str(round(p,4)*100) + '%')
```

```
P(A|B)=89%
P(A|B,C)=89%
```

On peut aussi vérifié que $\mathrm{P}(\mathsf{A}\,|\,\mathsf{B}) \times \mathrm{P}(\mathsf{C}\,|\,\mathsf{B}) = \mathrm{P}(\mathsf{A},\mathsf{C}\,|\,\mathsf{B})$: *source:* question1.py

```
57 p = serialBlockNet.p(A==True, B==True) * serialBlockNet.p(C==True, B==True)
58 print("P(A|B)*P(C|B)=" + str(p))
59 p = serialBlockNet.p((A==True)&(C==True), B==True)
60 print("P(A,C|B)=" + str(p))
```

```
P(A|B)*P(C|B)=0.2047
P(A,C|B)=0.2047
```

Question 2

Cette question va nous permettre d'étudier les calculs de probabilités sur un réseau de Bayes par la méthode brutale.

a) Modélisation du réseau de Bayes

Le réseau étudié peut être généré grâce au code suivant : *source:* question2.py

```
2  import BayesNetwork as bn
3
4  bayesNet = bn.Network()
5  C = bayesNet.createNode("C", 0.001)
6  T = bayesNet.createNode("T", 0.002)
7  A = bayesNet.createNode("A", [0.95, 0.94, 0.29, 0.001], parents=[T, C])
8  M = bayesNet.createNode("M", [0.90, 0.05], parents=A)
9  J = bayesNet.createNode("J", [0.70, 0.01], parents=A)
10 bayesNet.drawGraph(show=True)
```

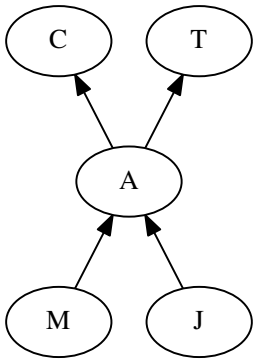

FIGURE 4 – Réseau de Bayes

Avant d'expliciter le fonctionnement de l'algorithme de calcul de probabilités (via un appel à la méthode `p(evenement, sachant)`), on peut vérifier pour quelques valeurs, la cohérence des résultats avec notre modèle ou avec le phénomène d'Explaining Away : *source:* question2.py

```
13 p = bayesNet.p(A==True, (T==True)&(C==False))
14 print("P(A|T,!C)=" + str(round(p,4)))
15 p = bayesNet.p(J==True, A==True)
16 print("P(J|A)=" + str(round(p,4)))
17 p = bayesNet.p(C==True)
18 print("P(C)=" + str(round(p,4)))
19 p1 = round( bayesNet.p(C==True, A==True) * bayesNet.p(T==True, A==True), 5)
20 p2 = round( bayesNet.p((C==True)&(T==True), A==True), 5)
21 print("P(C|A)*P(T|A)=" + str(p1) + " differe de p(C,T|A)="+ str(p2))
```

```
P(A|!C,T)=0.29
P(J|A)=0.7
P(C)=0.001
P(C|A)*P(T|A)=0.08629 diffère de p(C,T|A)=0.00076
```

b) Calcul de probabilités jointes et de distribution

Que ce soit pour calculer la distribution des probabilités ou pour implémenter un algorithme général, il au préalable pouvoir calculer une probabilité jointe. Dans un réseau de Bayes, une probabilité jointe est donné par :

$$\mathrm{P}(\mathsf{X}_0, \mathsf{X}_1, ..., \mathsf{X}_n) = \prod_{i=0}^{n} \left(\mathrm{P}(\mathsf{X}_i \,|...) \right)$$

Si `variables` est la liste des `VariableNode` d'un réseau et si `event` est l'évènement dont on veut calculer la probabilité jointe `probaJointe`, cette formule ce traduit en Python par :

```
probaJointe = 1
for var in variables:
    probaJointe *= var.probaAt(event)
```

C'est en répétant cette opération pour tous les évènements possibles que `distribution()` calcule la distribution du réseau (voir c) pour l'explication du listage des évènements) :

source: question2.py

```
print([str(round(_*100,2))+'%' for _ in bayesNet.distribution()])
bayesNet.drawDistribution(show=True) # Affiche l'histogramme
```

```
['0.0%', '0.04%', '0.06%', '0.06%', '0.0%', '0.0%', '0.0%', '0.05%', '0.0%', '0.0%',
'0.01%', '0.01%', '0.0%', '0.0%', '0.0%', '0.95%', '0.0%', '0.02%', '0.03%', '0.03%',
'0.0%', '0.01%', '0.0%', '4.93%', '0.0%', '0.0%', '0.0%', '0.0%', '0.0%', '0.13%',
'0.01%', '93.67%']
```

Les résultats sont présentés sur le tableau 1 et l'histogramme est visible sur la figure 5. On remarque que la très faible probabilité d'occurence de C et T concentre la distribution des probabilités sur le cas où aucune variable aléatoire n'est vraie ($\mathrm{P}(!\,\mathsf{C}, !\,\mathsf{T}, !\,\mathsf{A}, !\,\mathsf{M}, !\,\mathsf{J}) = 93.67\%$).

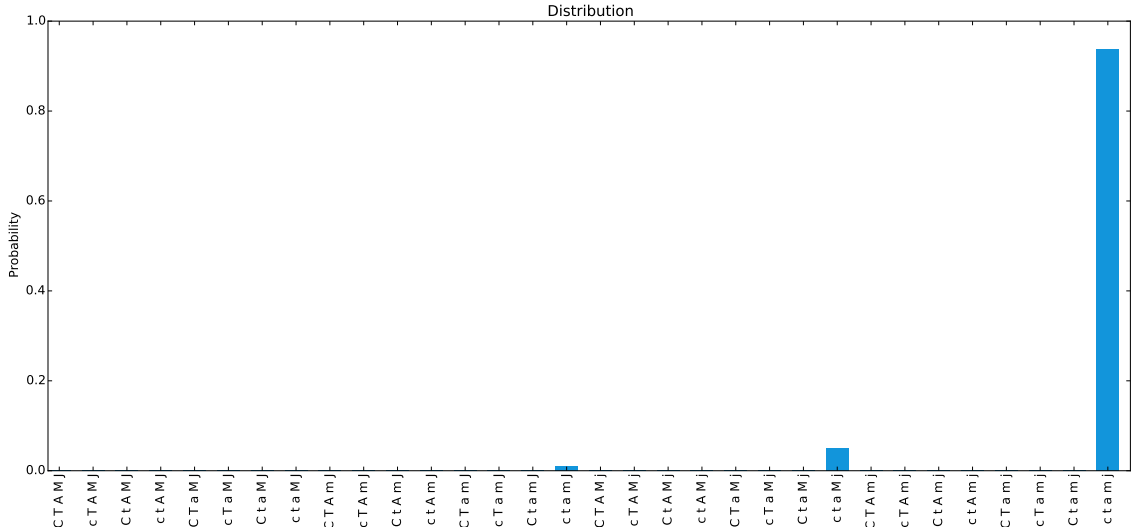

FIGURE 5 – Distribution des probabilités du réseau

		A				non A			
		T		non T		T		non T	
		C	non C	C	non C	C	non C	C	non C
J	M	0.0%	0.04%	0.06%	0.06%	0.0%	0.0%	0.0%	0.05%
	non M	0.0%	0.0%	0.01%	0.01%	0.0%	0.0%	0.0%	0.95%
non J	M	0.0%	0.02%	0.03%	0.03%	0.0%	0.01%	0.0%	4.93%
	non M	0.0%	0.0%	0.0%	0.0%	0.0%	0.13%	0.01%	93.67%

TABLE 1 – Distribution des probabilités du modèle étudié

c) Calcul de probabilités marginales conditionnelles

Dans un réseau de Bayes, une probabilité marginale s'exprime comme une somme de probabilités jointes. Par exemple :

$$\mathrm{P}(\mathsf{C} = Vrai, \mathsf{A} = Faux) = \sum_{\mathsf{T},\mathsf{M},\mathsf{J}} \left(\mathrm{P}(\mathsf{C} = Vrai,\ \mathsf{T},\ \mathsf{A} = Faux,\ \mathsf{M},\ \mathsf{J}) \right)$$

Aussi, pour pouvoir calculer une probabilité marginale quelconque il faut pouvoir décliner son évènement (par exemple : `(C==True)&(A==False)`) en une liste de sous-évènements atomiques où la valeur de chacune des variables est bien définie. C'est ce que réalise la méthode `Event.listVecEvents()` du package `probatoolbox.py` que j'ai codé pour l'occasion.

source: question2.py

```
32 event = (C==True)&(A==False) # Impose C=0 et A=1 et retourne un objet Event
33 for _ in event.listVecEvents():
34     print(_)                 # Les vecteurs ont comme base [C,T,A,M,J]
```

```
[0, 0, 1, 0, 0]
[0, 1, 1, 0, 0]
[0, 0, 1, 1, 0]
[0, 1, 1, 1, 0]
[0, 0, 1, 0, 1]
[0, 1, 1, 0, 1]
[0, 0, 1, 1, 1]
[0, 1, 1, 1, 1]
```

Je précise que j'ai choisi d'avoir `True` comme première valeur pour mes variables aléatoires puis `False`. Par conséquent l'index de la valeur `True` est `0` et l'index pour la valeur `False` est `1`, même si c'est contre-intuitif... Ces indexes ne servent de toutes façons que d'identifiant aux valeurs d'une variables aléatoires (qui peuvent aussi bien être des booléens, que des chaines de caractères ou des nombres flottants...)

En considérant que les `Event` stockent leurs états dans une liste nommée `space` ayant autant de cases que notre modèle de variables aléatoires et contenant chacunes, l'ensemble des valeurs que la variable peut prendre. Par exemple dans le cas de variable binaire cet ensemble serait `[0,1]` pour les variables non définis, `[0]` pour les variables valant `True`, et `[1]` pour les variables valant `False`. Pour l'évènement défini plus haut on aurait donc :

```
print( ((C==True)&(A==False)).space ) # Dans la base [C,T,A,M,J]
```

```
[[0], [0,1], [1], [0,1], [0,1]]
```

L'implémentation de la méthode `Event.listVecEvents()` pourrait alors se simplifier par :

```
def listVecEvents(self)
    # Initialisation du premier evenement atomique et de la varyingVar
    currentEvent = []
    varyingVar = [] # Liste des indexes des variables a faire varier
    for varId, values in enumerate(self.space):
        currentEvent.append(values[0])
        if len(self.values) > 1:
            varyingVar.append(varId)

    # Iteration a travers tous les evenements atomiques
    rList = [currentEvent]
    while(True):
        found = False;
        for varId in varyingVar:
            currentValue = currentEvent[varId]
            # On lit les valeurs acceptables pour cette variable
            values = self.space[varId]
            # On cherche la prochaine valeur acceptable
            while values[-1] >= currentV+1 and not found:
                currentV+=1
                if currentV in values:
                    found = True
            # Si une valeur acceptable a ete trouve on modifie currentEvent
            if found:
                currentEvent[varId] = currentV
                break   # et on sort de la boucle
            else:   # sinon on la remplace par la premiere valeur acceptable
                currentEvent[varId] = values[0]
            # et on passe a la variable suivante

        if found:   # Si on a un evenement on l'ajoute a la liste
            rList.append(list(currentEvent))
        else:   # Sinon on sort de la boucle
            break
    return rList
```

En réalité les valeurs possibles qu'un évènement attribue à une variable sont stockées sous forme d'ensemble (de `set`) ce qui complexifie l'implémentation de cette méthode. La version originale de `listVecEvents()`, visible dans le fichier `probatoolbox.py` est donc un peu plus complexe et complète. Elle permet nottament, via un paramètre optionnel `base` de modifier la base dans laquelle les vecteurs sont exprimés (la base par défaut suit l'ordre selon lequel les variables ont été déclarées)

Avec cette liste d'évènement atomique, sommer les probabilités jointes associées à chacun d'eux pour obtenir la probabilité marginale devient un jeu d'enfant. Cette opération est réalisée dans la méthode de la classe `Network` :

`computeInconditionnalProbability(self, event)` *source:* BayesNetwork.py

```
234         # Calcul de probabilite dans un espace non logarithmique
235         proba = 0
236         for e in event.listEvents(): # Pour chaque evenement atomique
237             pVar = 1
238             for varNode in self.nodes:  # Pour chaque variable du modele
239                 pVar *= varNode.probaAt(e)
240             proba += pVar
241         return proba
```

Enfin, pour implémenter la méthode `Network.p(event, knowing)` qui peut aussi calculer des probabiltés conditionnelles, il suffit d'appliquer la règle de bayes :

$$P(A \mid B) = \frac{P(A \cup B)}{P(B)}$$

En Python, on obtient : (sachant que l'opérateur `+` de la classe `Event` a été surchargé en opérateur d'union)

source: BayesNetwork.py

```
182     def p(self, event, knowing=None, method=""):
183         if not event.isValid:
184             return 0
185         if method == "":    # choisit la m thode de calcul (normal ou log)
186             method = self.defaultComputeMethod
187         if knowing is None:
188             return self.computeInconditionnalProbability(event, method=method)
189         else:
190             if knowing == event + knowing:
191                 return 1    # Si event est un sous-evenement de knowing, proba=1
192             p = self.p(knowing+event, method=method)
193             pK = self.p(knowing, method=method)
194             return p / pK
```

Finalement, on peut calculer les probabilités demandées :

source: question2.py

```
37 toCompute = [(M==True)&(J==False), (M==False)&(J==True), (M==True)&(J==True)]
38 toCompute+= [(M==False)&(J==False), M==True, J==True]
39 for knowing in toCompute:
40     p = bayesNet.p(C==True, knowing)
41     print("p(C=True|"+knowing.printVars()+") = " + str(round(p*100,3)) + "%")
```

```
p(C=True|M=True & J=False) = 0.513%
p(C=True|M=False & J=True) = 0.688%
p(C=True|M=True & J=True) = 28.417%
p(C=True|M=False & J=False) = 0.009%
p(C=True|M=True) = 1.628%
p(C=True|J=True) = 5.612%
```

d) Calcul des probabilités marginales inconditionnelles

Afin de calculer ces probabilités, on peut soit utiliser l'algorithme précédent :

source: question2.py

```
47 for node in bayesNet.nodes:
48     var = node.var
49     p = bayesNet.p(var==True)
50     print("p("+var.name+"=True) = "+str(round(p*100,3))+'%')
```

```
p(C=True) = 0.1%
p(T=True) = 0.2%
p(A=True) = 0.252%
p(M=True) = 5.214%
p(J=True) = 1.174%
```

Ou alors directement extraire ces informations du graphes en calculant, une à une les probabilités marginales des enfants en fonction de celles des parents :

source: question2.py

```
53 pC = 0.001
54 pT = 0.002
55 pA = 0.95*pT*pC + 0.94*(1-pT)*pC + 0.29*pT*(1-pC) + 0.001*(1-pT)*(1-pC)
56 pM = 0.90*pA + 0.05*(1-pA)
57 pJ = 0.70*pA + 0.01*(1-pA)
58 for _ in [[pC,'C'],[pT,'T'],[pA,'A'],[pM,'M'],[pJ,'J']]:
59     print("p(" + _[1] + "=True) = " + str(round(_[0] * 100, 3)) + '%')
```

```
p(C=True) = 0.1%
p(T=True) = 0.2%
p(A=True) = 0.252%
p(M=True) = 5.214%
p(J=True) = 1.174%
```

Dans les deux cas, les résultats sont similaires...

e) Équations de calcul de probabilité

Si on utilise la définition d'une probabilité marginale, on a :

$$\begin{aligned}
\mathrm{P}(\mathsf{J}) &= \sum_{\mathsf{A}}\sum_{\mathsf{C}}\sum_{\mathsf{T}}\sum_{\mathsf{M}} \left(\mathrm{P}(\mathsf{J}\,|\,\mathsf{A})\,\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{T})\,\mathrm{P}(\mathsf{M}\,|\,A)\right) \\
&= \sum_{\mathsf{A}}\sum_{\mathsf{C}}\sum_{\mathsf{T}} \left(\mathrm{P}(\mathsf{J}\,|\,\mathsf{A})\,\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{T})\sum_{\mathsf{M}}\left(\mathrm{P}(\mathsf{M}\,|\,A)\right)\right) \\
&= \sum_{\mathsf{A}} \left(\mathrm{P}(\mathsf{J}\,|\,\mathsf{A})\sum_{\mathsf{C}}\sum_{\mathsf{T}}\left(\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{T})\right)\right) \\
\text{or} \qquad \mathrm{P}(\mathsf{A}) &= \sum_{\mathsf{C}}\sum_{\mathsf{T}}\left(\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{T})\right) \\
\text{donc} \qquad \mathrm{P}(\mathsf{J}) &= \sum_{\mathsf{A}}\left(\mathrm{P}(\mathsf{J}\,|\,\mathsf{A})\,\mathrm{P}(\mathsf{A})\right)
\end{aligned}$$

On peut facilement retrouver ce résultat en appliquant une marginalisation au théorème de Bayes :

$$\begin{aligned}
\mathrm{P}(\mathsf{J}) &= \sum_{\mathsf{A}} \left(\mathrm{P}(\mathsf{J},\mathsf{A}) \right) && \text{(marginalisation)} \\
&= \sum_{\mathsf{A}} \left(\mathrm{P}(\mathsf{J}\,|\,\mathsf{A}) \times \mathrm{P}(\mathsf{A}) \right) && \text{(théorème de Bayes)} \\
\mathrm{P}(\mathsf{J}) &= \mathrm{P}(\mathsf{J}\,|\,\mathsf{A}=Vrai)\,\mathrm{P}(\mathsf{A}=Vrai) \;+\; \mathrm{P}(\mathsf{J}\,|\,\mathsf{A}=Faux)\,\mathrm{P}(\mathsf{A}=Faux)
\end{aligned}$$

avec
$$\begin{aligned}
\mathrm{P}(\mathsf{A}) &= \sum_{\mathsf{C}}\sum_{\mathsf{T}} \left(\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{T}) \right) \\
\mathrm{P}(\mathsf{A}) = {}& \mathrm{P}(\mathsf{A}\,|\,\mathsf{C}=Vrai,\mathsf{T}=Vrai)\;\mathrm{P}(\mathsf{C}=Vrai)\;\mathrm{P}(\mathsf{T}=Vrai) \\
&+ \mathrm{P}(\mathsf{A}\,|\,\mathsf{C}=Faux,\mathsf{T}=Vrai)\;\mathrm{P}(\mathsf{C}=Faux)\;\mathrm{P}(\mathsf{T}=Vrai) \\
&+ \mathrm{P}(\mathsf{A}\,|\,\mathsf{C}=Vrai,\mathsf{T}=Faux)\;\mathrm{P}(\mathsf{C}=Vrai)\;\mathrm{P}(\mathsf{T}=Faux) \\
&+ \mathrm{P}(\mathsf{A}\,|\,\mathsf{C}=Faux,\mathsf{T}=Faux)\;\mathrm{P}(\mathsf{C}=Faux)\;\mathrm{P}(\mathsf{T}=Faux)
\end{aligned}$$

Toujours en partant de la définition d'une probabilité marginale, on a :

$$\begin{aligned}
\mathrm{P}(\mathsf{C}\,|\,\mathsf{J}=Vrai) &= \sum_{\mathsf{A}}\sum_{\mathsf{T}}\sum_{\mathsf{M}} \left(\mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A})\,\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{T})\,\mathrm{P}(\mathsf{M}\,|A) \right) \\
&= \mathrm{P}(\mathsf{C}) \sum_{\mathsf{A}}\sum_{\mathsf{T}} \left(\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A})\,\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{T}) \sum_{\mathsf{M}} \left(\mathrm{P}(\mathsf{M}\,|A) \right) \right) \\
\mathrm{P}(\mathsf{C}\,|\,\mathsf{J}=Vrai) &= \mathrm{P}(\mathsf{C}) \sum_{\mathsf{A}} \left(\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A}) \sum_{\mathsf{T}} \left(\mathrm{P}(\mathsf{A}\,|\,\mathsf{C},\mathsf{T})\,\mathrm{P}(\mathsf{T}) \right) \right)
\end{aligned}$$

Après développement on obtient :

$$\begin{aligned}
\mathrm{P}(\mathsf{C}\,|\,\mathsf{J}=Vrai) = {}& \mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A}=Vrai)\,\mathrm{P}(\mathsf{A}=Vrai|\,\mathsf{C},\mathsf{T}=Vrai)\,\mathrm{P}(\mathsf{T}=Vrai) \\
&+ \mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A}=Vrai)\,\mathrm{P}(\mathsf{A}=Vrai|\,\mathsf{C},\mathsf{T}=Faux)\,\mathrm{P}(\mathsf{T}=Faux) \\
&+ \mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A}=Faux)\,\mathrm{P}(\mathsf{A}=Faux|\,\mathsf{C},\mathsf{T}=Vrai)\,\mathrm{P}(\mathsf{T}=Vrai) \\
&+ \mathrm{P}(\mathsf{C})\,\mathrm{P}(\mathsf{J}=Vrai|\,\mathsf{A}=Faux)\,\mathrm{P}(\mathsf{A}=Faux|\,\mathsf{C},\mathsf{T}=Faux)\,\mathrm{P}(\mathsf{T}=Faux)
\end{aligned}$$

Question 3 : Sum Product

L'algorithme *sum-product* a été implanté dans le package `FactorGraph.py`. Je présenterai dans un premier temps les structures de données que j'ai utilisé puis la propagation des messages au sein du réseau durant la compilation, enfin le calcul des messages effectué par les noeuds.

Architecture et structure de données

Les noeuds d'un *Factor Graph*, qu'ils représentent une variable aléatoire ou une fonction, partagent tous le même mécanisme de propagation de message. C'est donc la classe mère `Node` en combinaison avec la classe `Message` qui se chargeront de ce mécanisme (voir figure 6). L'ensemble des noeuds sont stockés dans un graphe `Graph` qui est chargé de superviser la compilation du réseau (via la méthode `compile()`), et donc le calcul des probabiltés (on retrouve la même interface `p(event, knowing)`).

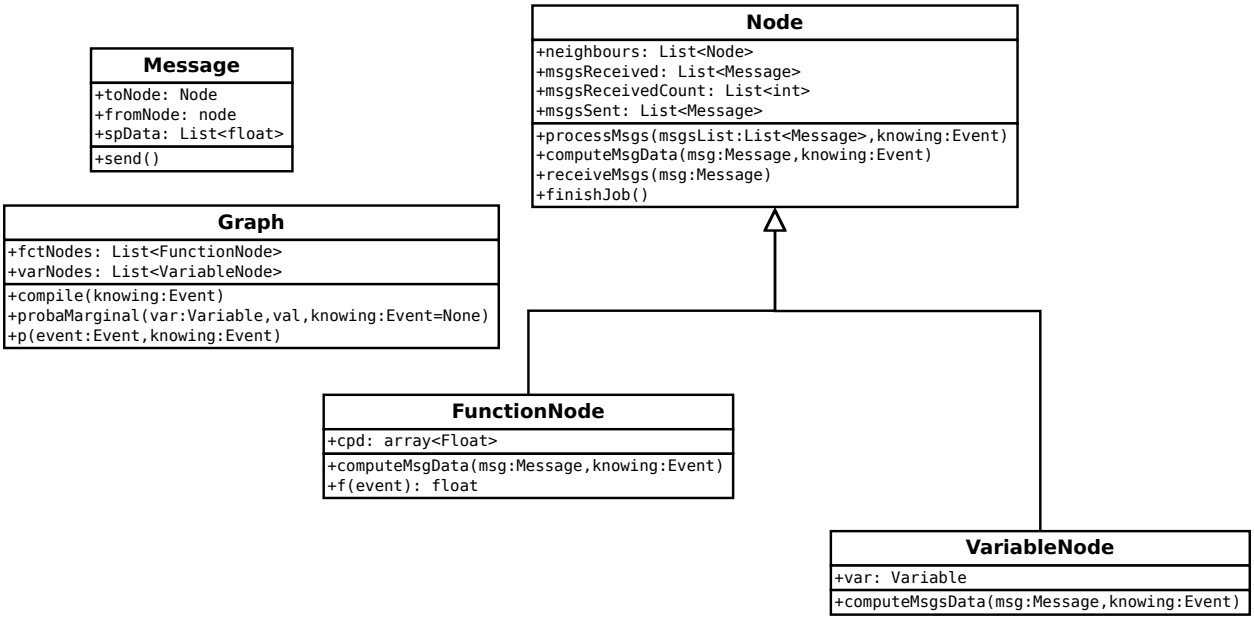

FIGURE 6 – Diagramme UML des structure de données

La méthode `graphFromBayesNet(bayesNet)` permet de convertir un réseau de Bayes en Graphes de Facteur (voir résultat en figure 7) :

source: question3.py

```
5  bayesNet = bn.Network()
6  C = bayesNet.createNode("C", 0.001)
7  T = bayesNet.createNode("T", 0.002)
8  A = bayesNet.createNode("A", [0.95, 0.94, 0.29, 0.001], parents=[T, C])
9  M = bayesNet.createNode("M", [0.90, 0.05], parents=A)
10 J = bayesNet.createNode("J", [0.70, 0.01], parents=A)
11
12 graph = fg.graphFromBayesNet(bayesNet)
13 graph.drawGraph(show=True)
```

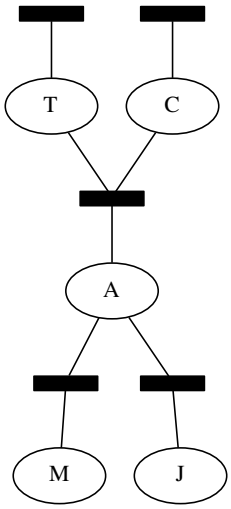

FIGURE 7 – Graphes de Facteur

Compilation et propagation des messages

La compilation d'un graphe consiste en une boucle infinie qui passe en revue tous les noeuds pour générer tous les messages générable dans l'état actuel du réseau puis déclenche l'envoi de ces messages. Une version très simplifié de la méthode `compile()` pourrait donc être :

```
msgs = []
for node in self.varNodes + self.fctNodes:
    node.processMsgs(msgs)

while msgs:
    for m in msgs
        m.send()

    msgs = []
    for node in self.varNodes + self.fctNodes:
        node.processMsgs(msgs)

for node in self.varNodes + self.fctNodes:
    node.finishJob(msgs)
```

Puisque lorsqu'il est construit, un message à connaissance de son destinataire `toNode`, il est capable de déclencher son envoi : *source:* `FactorGraph.py`

```
    def send(self):
        self.toNode.receiveMsg(self)
```

La méthode `receiveMsg` de la classe `Node` prend alors le relai et stocke le dit message dans `receivedMsgs` à l'index du noeud émetteur. Elle mets ensuite à jour les compteurs de réception de messages de tous les autres noeuds : *source:* `FactorGraph.py`

```
    def receiveMsg(self, msg):
        for id, node in enumerate(self.neighbours):
            if node is msg.fromNode:
```

```
110                 self.msgsReceived[id] = msg
111             else:
112                 self.msgsReceivedCount[id] += 1
```

Ces compteurs de réception sont ensuite utilisés dans la méthode `Node.processMsgs(msgs)` pour savoir si un message peut être envoyé ou non : plus précisément si le compteur de réception associé à un noeud voisin est supérieur ou égal au nombre de voisin -1, alors un message doit être envoyé à ce noeud. Une fois envoyé le compteur est décrémenté aux nombres de voisins -2 de tel sorte que le prochain message reçu déclenche la réémission d'un message vers ce voisin.

source: FactorGraph.py

```
114     def processMsgs(self, msgsList, knowing, makeAssumption=0, method='base'):
115         ngbCount = len(self.neighbours)
116         hasConverged = True
117 
118         for outId, receivedCount in enumerate(self.msgsReceivedCount):
119             if receivedCount >= ngbCount-1-makeAssumption:
120                 # Si tous les messages ont ete recus
121                 # on genere un message pour ce noeud
122                 msg = Message(self, self.neighbours[outId])
123                 self.computeMsgData(msg, knowing=knowing, methodName=method)
124                 if self.msgsSent[outId] == msg:
125                     continue
126                 hasConverged = False
127                 self.msgsSent[outId] = msg
128                 self.msgsReceivedCount[outId] = ngbCount - 2
129                 msgsList.append(msg)
130             elif self.msgsSent[outId] is None:
131                 hasConverged = False
132         return hasConverged
```

Dans cette implémentation `hasConverged` et `makeAssumption` permettent la gestion des réseaux cycliques que nous ne détaillerons pas ici.
On peut donc dorénavant compiler notre réseau et suivre la propagation des messages. On constate que les messages envoyés correspondent bien à ceux attendus et présenté dans le cours. La trace des message est de la forme `emmetteur -> destinataire: valeur`, et les nombres tout à gauche correspondent au compteur de tour de la boucle principale de `compile()`.

source: question3.py

```
15 toCompute = [(M==True)&(J==False), (M==False)&(J==True), (M==True)&(J==True)]
```

```
1:      M -> f(A,M): [1, 1]
        J -> f(A,J): [1, 1]
        f(C) -> C: [0.001, 0.999]
        f(T) -> T: [0.002, 0.998]

2:      C -> f(C,T,A): [0.001, 0.999]
        T -> f(C,T,A): [0.002, 0.998]
        f(A,M) -> A: [1.0, 1.0]
        f(A,J) -> A: [1.0, 1.0]

3:      A -> f(C,T,A): [1.0, 1.0]
        f(C,T,A) -> A: [0.0025164419999999998, 0.99748355799999999]
```

```
 4:      A -> f(A,M): [0.0025164419999999998, 0.99748355799999999]
        A -> f(A,J): [0.0025164419999999998, 0.99748355799999999]
        f(C,T,A) -> C: [1.0, 0.99999999999999989]
        f(C,T,A) -> T: [1.0, 1.0]

5:      C -> f(C): [1.0, 0.99999999999999989]
        T -> f(T): [1.0, 1.0]
        f(A,M) -> M: [0.052138975700000006, 0.94786102429999997]
        f(A,J) -> J: [0.011736344980000001, 0.98826365502000002]
```

Calcul de la valeur d'un message

L'une des lignes les plus importantes de la méthode `Node.processMsgs()` est certainement :

source: FactorGraph.py

```
123                 self.computeMsgData(msg, knowing=knowing, methodName=method)
```

Cette ligne appele la fonction `Node.computeMsgData(msg)` pour affecter au message sa valeur. Cette méthode est redéfinie dans `VariableNode` et dans `FunctionNode` pour permettre un comportement différent pour ces deux types de noeuds. Ainsi les `VariableNode` devront renvoyer le produit des messages reçus depuis les autres noeuds. Quant à elles, les `FunctionNode` doivent en plus multiplier ce produit par la valeur de la fonction puis sommer les résulats.
Une version simplifié de `FunctionNode.computeMsgData(msg, knowing)` :

```
1  spData = []
2  for valueId in msg.toNode.valuesId():
3      # Pour chaque valeur de la variable du noeud destinataire
4      sp = 0
5
6      # on somme selon l'evenement joint:
7      event = (msg.toNode.var == valueId) & knowing
8
9      for e in event.listVecEvents(self.neighboursBase):
10         spTemp = self.f(e)
```

```
    for i, m in enumerate(self.msgsReceived):
        if m is None or (m.fromNode is msg.toNode):
                continue # On saute le noeud destinataire
        # On multiplie la valeur du message correspondant a l'evenement atomique
        spTemp *= m.spData[e[i]]

        sp += spTemp
    spData.append(sp)
msg.spData = spData

msg.spData = spData
```

En réalité la méthode `FunctionNode.computeMsgData(msg, knowing)` tel qu'implémentée dans le package `FactorGraph.py` est plus complexe. Elle supporte le calcul dans un espace logarithmique, le calcul d'un message alors que tous les noeuds voisins n'ont pas encore transmis le leur (pour gérer les graphes cycliques), ainsi que l'algorithme *Max-Sum*...
Une version simplifié de `VariableNode.computeMsgData(msg)` :

```
spData = []
for valueId in self.valuesId():
    # Pour chaque valeur de la variable du noeud
    sp = 1  # on calcule le produit des messages
    for m in self.msgsReceived:
        if m is None or (m.fromNode is msg.toNode):
            continue # On saute le noeud destinataire
        sp *= m.spData[valueId]
    spData.append(sp)

msg.spData = spData
```

Finalisation et lecture des probabilités

Une fois que tous les messages ont été transmis, on peut simplement lire la probabilité marginale d'une variable en multipliant tous les messages que sont noeud à reçus ou envoyés. Ou de manière plus optimal multiplier le message envoyé à une fonction par le message reçu de cette fonction. Dans le cas de probabilité conditionnelle, il faut en plus normalisé la valeur des probabilités obtenues pour que leur somme soit égales à 1. Cette tâche est répartie entre la méthode `Node.finishJob()` et `Graph.probaMarginal(var, value, knowing=None, conditional=True)` elle même appeler par la méthode `Graph.p(event, knowing)`. Le fonctionnement de ces méthodes ne sera pas étudié en détail dans ce rapport, leur enjeux étant de gérer de manière optimale le calcul de probabiltés marginales, conditionnelles, marginales conditionnelles et quelconques...

Validation de l'algorithme

Pour finir, on peut vérifier la validité de l'algorithme en calculant les probabilités déjà évaluées à la question précédente : *source:* question3.py

```
toCompute = [(M==True)&(J==False), (M==False)&(J==True), (M==True)&(J==True)]
toCompute+= [(M==False)&(J==False), M==True, J==True]
for knowing in toCompute:
    p = graph.p(C==True, knowing)
    print("p(C=True|"+knowing.printVars()+") = " + str(round(p*100,3)) + "%")
print("")
for node in graph.varNodes:
    var = node.var
    p = graph.p(var == True)
    print("p(" + var.name + "=True) = " + str(round(p * 100, 3)) + '%')
```

```
p(C=True|M=True & J=False) = 0.513%
p(C=True|M=False & J=True) = 0.688%
p(C=True|M=True & J=True) = 28.417%
p(C=True|M=False & J=False) = 0.009%
p(C=True|M=True) = 1.628%
p(C=True|J=True) = 5.612%

p(C=True) = 0.1%
p(T=True) = 0.2%
p(A=True) = 0.252%
p(M=True) = 5.214%
p(J=True) = 1.174%
```

Au niveau de la précision numérique, les résultats de l'algorithme de sum-product sont similaires à ceux obtenus avec la méthode brutale. Bien sur, la complexité du calcul est plus faible avec cet algorithme qu'avec la méthode brutale. Nottament lorsqu'on ajoute, une à une, des observations sur le graphe : les phénomènes de *Serial Blocking* ou de *Divergent Blocking* permettent de limiter la propagation des messages pour mettre à jour le réseau et donc gagner un temps de calcul considérable. Bien que ces mécanismes n'aient pas été présentés dans ce rapport, mon implémentation de l'algorithme de sum product supporte ces mises-à-jour partielles du réseau via la méthode `Graph.addObserved(event)`.

A Le package : *probatoolbox.py*

B Le package : *BayesNetwork.py*

Le package `BayesNetwork` contient une classe `Network` qui représente notre réseau (un ensemble de `VariableNode`). L'ajout de noeud à un `Network` est réalisé par l'appel de la méthode `createNode(nom, CPD, parents=[])` qui renvoie une référence vers la variable aléatoire construite. Le paramètre `CPD` prend la forme d'un tableau contenant la probabilité conditionnelle que la variable soit vrai sachant la valeur de ses parents (la probabilité complémentaire est inférée). Le premier élément est la probabilité sachant que tous les parents sont vrais, la deuxième que seul le premier parent est faux, puis seul le deuxième puis seul les deux premiers et ainsi de suite en suivant une incrémentation binaire.