Génération de codes d'inférences probabilistes

Marvin Lavechin Pierre Stefani

Table des matières

1 Présentation du projet

De Mars à Août 2015, encadré par Mr Pierre Henri Wuillemin, chercheur à l'Université Pierre et Marie Curie, nous avons travaillé sur la génération d'un code pour calculer des probabilités dans un réseau bayésien.

Les réseaux bayésiens sont des faisceaux de probabilités, ordonnés par des liens de parentés. Les éléments du réseau bayésien sont liés entre eux par des probabilités conditionnelles. Dans ces réseaux, on définit les éléments connus : évidences (ou vraisemblances) et les éléments dont on souhaite connaître la probabilité : les *targets*. La structure de réseau bayésien, et d'arbre de jonction (que nous étudierons plus en profondeur) permet la diffusion de l'information des évidences vers les targets, aussi appelé l'inférence. Dans un cadre classique, ce calcul de probabilités selon des observations : $P(X|e)$ est simple. Toutefois, les modèles étant très vite complexes, l'inférence probabiliste devient un problème NP-difficile.

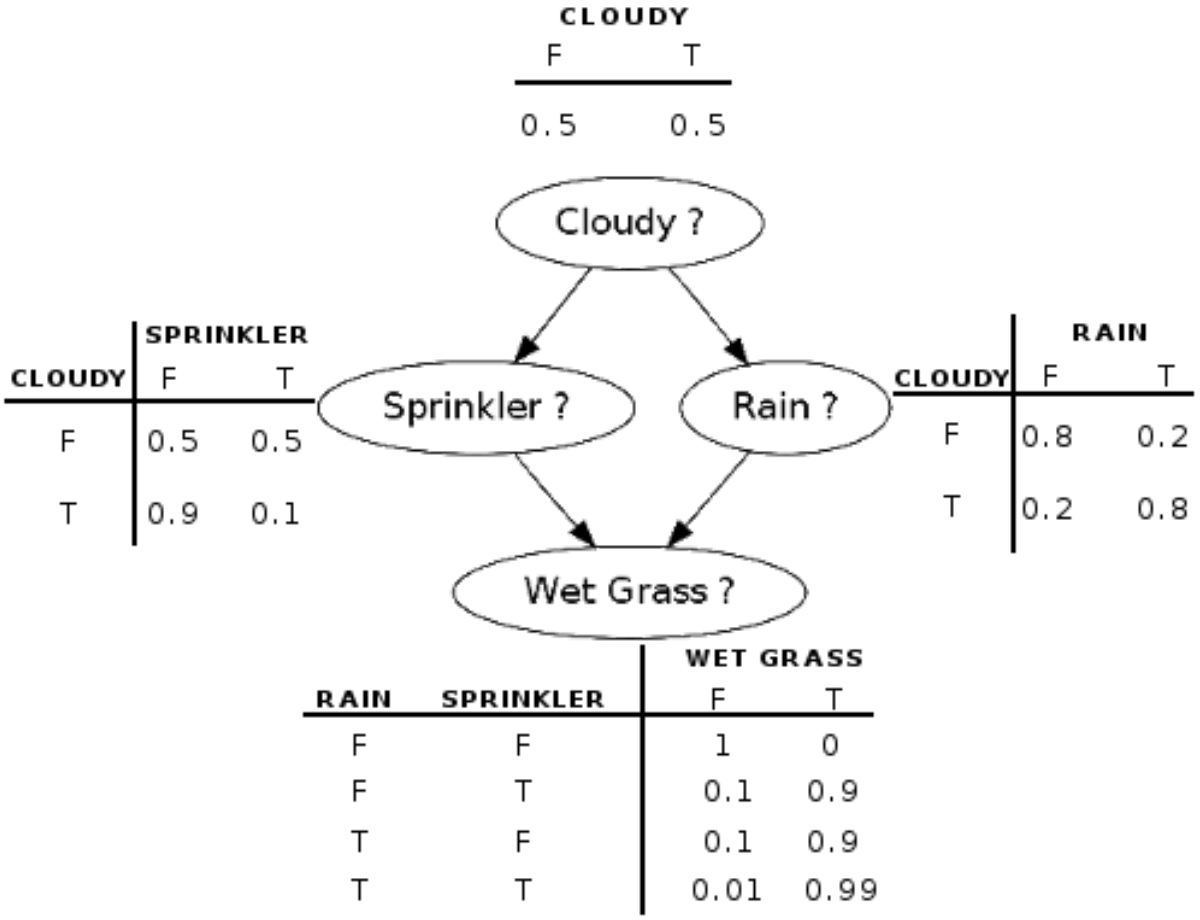

FIGURE 1 – Exemple de réseau bayésien et probabilités conditionnelles des variables

Le projet *metaGenBayes* part d'un réseau bayésien, d'évidences, de targets et d'un langage cible, et écrit dans ce langage souhaité le code calculateur de $P(targets|evidences)$.

2 Présentation du problème et éléments théoriques

2.1 Calcul de probabilité

2.1.1 Quelques formules

Comme explicité dans la courte présentation, l'inférence probabiliste s'appuie sur des probabilités conditionnelles, régies par quelques lois de calcul qu'il est important d'introduire :
Soient A et B deux événements quelconques d'un même univers. On s'intéresse à ce que devient la probabilité de A lorsqu'on apprend que B est déjà réalisé. On note $P(A|B)$ et on lit "probabilité de A sachant B". La définition mathématique de $P(A|B)$ est :

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}, \quad P(B) \neq 0$$

(Formule de Bayes)

Cette formule s'écrit aussi : $P(A|B) \quad \alpha \quad P(A) \times P(B|A)$ [1]
Ceci nous amène à considérer deux autres formules. Soit $(A_i)_{i \in I}$ un système complet d'événements de probabilités non nulles. Alors pour tout événement B on a :

$$P(B) = \sum_{i \in I} P(A_i \cap B) = \sum_{i \in I} P(A_i)P(B|A_i)$$

(Formule des probabilités totales)

Si $P(A_1 \cap \ldots \cap A_n) \neq 0$, on a :

$$P(A_1 \cap \ldots \cap A_n) = \prod_{i=1}^{n} P(A_i|A_{i+1} \cap \ldots \cap A_n)$$

(Formule des probabilités composées)

2.1.2 Calcul au sein d'une table de probabilité jointe

Soient A_1, A_2, A_3 trois variables aléatoires binaires. On peut écrire une table listant toutes les combinaisons possibles de ces 3 variables, qui sont au nombre de 2^3.
En effet, on a :

$$\begin{aligned} &(A_1 = true \quad A_2 = true \quad A_3 = true) \\ &(A_1 = false \quad A_2 = true \quad A_3 = true) \\ &(A_1 = true \quad A_2 = false \quad A_3 = true) \\ &\qquad\qquad\qquad\qquad\qquad\qquad etc... \end{aligned}$$

Pour chacune des combinaisons, supposons que nous disposions de la probabilité jointe de cette combinaison.
Si l'on veut la probabilité $P(A_1 \cap A_2)$, la formule des probabilités totales nous dit qu'il suffit qu'on somme les probabilités des combinaisons de la table pour laquelle A_1 est vrai et A_2 l'est aussi.
Si l'on veut la probabilité $P(A_1|A_2)$, la formule de Bayes nous dit qu'il suffit qu'on somme toutes les probabilités des combinaisons de la table pour lesquelles A_1 est vrai et A_2 est vrai en divisant par la somme des probabilités des combinaisons de la table pour lesquelles A_2 est vrai.
On peut donc à partir de la table des probabilités jointes déduire n'importe quelle probabilité conditionnelle. Les tailles de ces tables évoluant de manière exponentielle, il devient impossible d'utiliser simplement cette approche pour le calcul d'inférences probabilistes.

1. $P(A|B)$ est la loi à posteriori, $P(A)$ la loi à priori et $P(B|A)$ la vraisemblance, tandis que P(B) sert de coefficient normalisateur

2.2 Arbres de Jonction

Les arbres de jonctions sont un aspect théorique important de ce projet. La diffusion de l'information dans les réseaux bayésiens est très vite rendue impossible par le nombre de noeuds et la présence de nombreux cycles. La structure d'arbre de jonction transforme un réseau bayésien en graphe non orienté constitué de 'super-noeuds' connectant plusieurs noeuds du réseau bayésien. Afin de conserver l'information, des séparateurs sont placés entre les cliques de l'arbre pour satisfaire la propriété suivante : *Deux cliques U et V possédant un ensemble S de probabilités communes, sont séparés, lors de leur liaison dans l'arbre de jonction, par cet ensemble S*

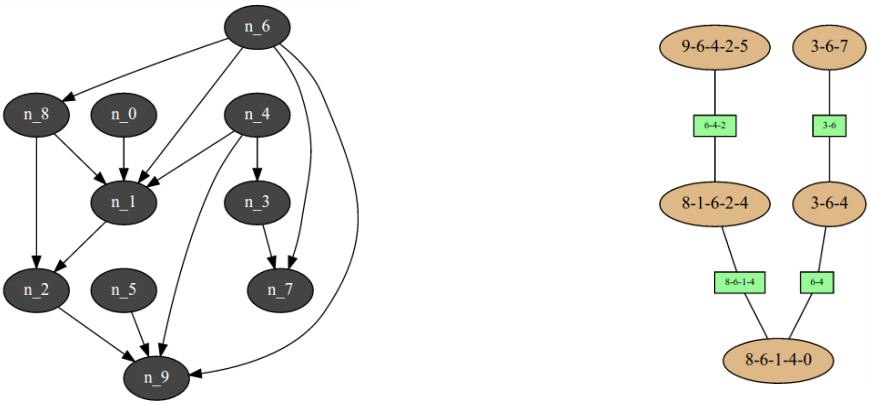

FIGURE 2 – Un réseau bayésien généré (à gauche), et un arbre de jonction possible pour ce réseau (à droite)

Créer cet arbre de jonction passe d'abord par une étape de **moralisation** : créer un arc entre deux parents du même élément. Dans l'exemple de la Figure 2, un lien est crée entre n2 et n5, deux des parents de n9. Une fois cette moralisation effectuée, la création de *supercliques* regroupant des élèments connectés du réseau bayésien est quasiment finie. Toute orientation est ensuite enlevée, et la dernière étape, dite de **triangulation**, enlève toute possibilité de cycles à la nouvelle structure, en ajoutant des cordes aux cycles présents. Les arbres de jonctions regroupent désormais plusieurs probabilités, et l'on associera à ces ensembles leurs probabilités respectives avec des *potentiels* dans le lexique des réseaux bayésiens, qui seront simplement des tableaux de probabilités dans nos langages plus courants.

2.3 Diffusion de l'information dans les arbres de jonctions

Rappelons notre principal objectif : à partir d'une évidence donnée pour une probabilité A, nous devons calculer la nouvelle probabilité de l'évènement B : l'information de A va donc devoir circuler dans le réseau. La structure d'arbres de jonctions va grandement simplifier ce que l'on appelera la *diffusion* de l'information. La clique de départ, celle pour laquelle a été transmise l'information (ce choix arbitraire n'a d'influence que sur le temps de réponse du programme) possède un potentiel Φ contenant les probabilités des variables de notre réseau. Pour passer d'une clique A à la clique B, l'information est projetée de A à B selon les séparateurs comme le montre la Figure 3 ci-dessous.

La projection de A sur B suit d'abors la formule de projection sur le séparateur :

$$\Psi^{'}_{AB} = \sum_{X_i \in \Phi_A, X_i \notin \Psi_{AB}} \Phi_A$$

Puis du séparateur à la clique B :

$$\Phi^{'}_{B} = \Phi_B * \Psi^{'}_{AB}$$

Dans le cas de la figure 3, cela donne :

$$\Phi^{'}_{n3,n6,n4} = \Phi_{n3,n6,n4} * \sum_{n7} \Phi_{n3,n6,n7}$$

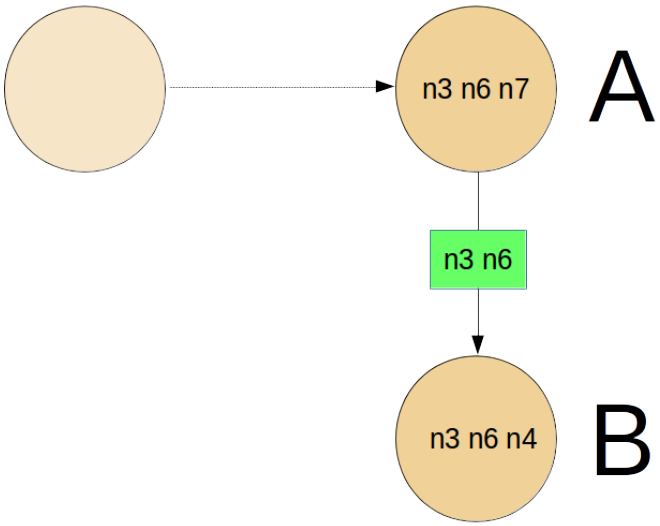

FIGURE 3 – Exemple de diffusion d'une clique A vers une clique B dans un arbre de jonction

Ainsi, à partir des évidences fournies à certaines cliques de l'arbre de jonction, toute l'information sera diffusée vers une clique principale, appelée clique racine, dans une phase que nous avons appelée **absorption**. Cette clique racine sera ensuite le point de départ de la **diffusion** de l'information clique par clique vers les noeuds contenant les targets.
La phase de diffusion est légèrement différente de l'absorption. En effet, il faut d'abord collecter l'information de toutes les cliques avoisinantes (sauf celle vers qui l'on diffuse) pour actualiser les potentiels : Considérons l'exemple ci-contre :

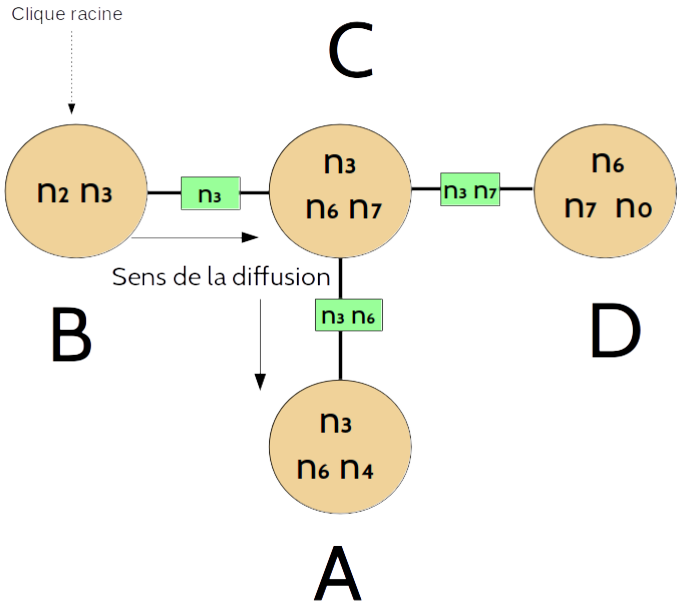

FIGURE 4 – Exemple pour la phase de collecte d'informations de la diffusion

Admettons que B et C aient reçu l'information de la diffusion, qui doit maintenant être projetée sur A, clique contenant une target. Il faut 'collecter' l'information dans les cliques voisines, c'est à dire, pour aller de C à A, récupérer les potentiels Φ_B et Φ_D, en plus de Φ_C. Une fois Φ_C actualisé de la sorte, on peut appliquer les formules de projections sur le séparateur puis vers D comme vu précedemment pour l'absorption. La clique D et ses potentiels ont alors bien reçu l'information de tout le réseau bayésien.
Une fois arrivé sur la clique contenant la target, il faut pouvoir passer d'un potentiel contenant en réalité plusieurs probabilités à la probabilité d'un évènement. Cela consiste à marginaliser : Soit une target $T \in C$, une clique de notre arbre,

$$P(T) = \sum_{X \in C, X \neq T} \Phi_C$$

Une fois la probabilité de la target, $P(T)$, marginalisé, le résultat final est la normalisation de $P(T)$ selon ses parents.

Voici donc la théorie qui encadre notre projet. L'idée importante de *MetaGenBayes* est la construction d'un métacode, qui va parcourir l'arbre de jonction et créer un tableau de toutes les instructions nécessaires pour l'inférence : multiplication par des séparateurs, marginalisations de potentiels, création de variables, etc Cette étape, appelée **Compilation** permettra ensuite de très facilement écrire le code calculatoire, étape par étape, de $P(targets|evidences)$ dans le langage cible.

3 Compilation et génération : du résau bayésien au code calculatoire

3.1 Compilation : introduction et résultat

Pour pouvoir générer en différents langages une même requête, il était nécessaire de passer par une étape dite de compilation, entre les entrées de l'utilisateur et l'écriture dans le langage spécifié. Le résultat de la compilation, un tableau d'instructions, est compréhensible par tous les générateurs de langages, et le déchiffrage de ce dernier correspondra à la *génération*. A partir de nos connaissances en réseaux bayésiens, et après avoir travaillé sur quels types de calculs nous allions devoir utiliser, sept opérations de bases sont apparues. Ce sont ces opérations que l'on retrouve dans chacun des index du tableau d'instructions,

avec leurs arguments. Voici un example d'instruction pour la création d'un potentiel Φ_{0135} contenant les variables 0, 1, 3 et 5 du réseau bayésien :

['CPO', $'Phi-0-1-3-5'$, [0,1,3,5]]

. Voici les opérations principales :

— L'initialisation des cpts (probabilités de chacune des variables). Instruction particulière avant toute générétion, permettant la copie dans des variables locales des tableaux de probabilités du réseau bayésien.
— La création d'un potentiel, code CPO, qui crée un potentiel dont le *nom* et les *variables* qu'il contient sont passés en paramètres.
— L'instruction ASE, qui initialise un potentiel lié à une évidence. Le *nom*, la *valeur* et enfin l'*index* de l'évidence dans le dictionnaire sont les arguments.
— La multiplication d'un potentiel par une cpt, MUC. Pour la génération, le *nom* du potentiel, ses *variables* sont stockés, ainsi que la *variable* dont le tableau de probabilités sera récupéré.
— La multiplication entre deux potentiels, MUL. Là encore, *noms* et *variables* sont ajoutés au tableau d'instruction.
— L'instruction MAR pour la marginalisation d'un potentiel selon un autre potentiel. Les mêmes paramètres que pour la multiplication entre deux potentiels sont stockés.
— La normalisation d'un potentiel, identifié par son *nom*.

Le fichier compiler.py comprend un métacode qui effectue les opérations nécessaires pour ajouter correctement les instructions au tableau petit à petit. Ci-dessous un exemple d'un tableau d'instruction crée avec le compilateur.

```
['CPO', 'Phi26_55_c0', [26, 55]]
['ADV', '26', 'Phi26_55_c0']
['ADV', '55', 'Phi26_55_c0']
['FIL', 'Phi26_55_c0', 1]
['CPO', 'Phi50_26_c1', [50, 26]]
['ADV', '50', 'Phi50_26_c1']
['ADV', '26', 'Phi50_26_c1']
['FIL', 'Phi50_26_c1', 1]
['CPO', 'Phi37_38_39_c2', [37, 38, 39]]
['ADV', '37', 'Phi37_38_39_c2']
['ADV', '38', 'Phi37_38_39_c2']
['ADV', '39', 'Phi37_38_39_c2']
['FIL', 'Phi37_38_39_c2', 1]
['CPO', 'Phi26_45_c3', [26, 45]]
['ADV', '26', 'Phi26_45_c3']
['ADV', '45', 'Phi26_45_c3']
['FIL', 'Phi26_45_c3', 1]
['CPO', 'Phi32_33_34_31_c4', [32, 33, 34, 31]]
['ADV', '32', 'Phi32_33_34_31_c4']
['ADV', '33', 'Phi32_33_34_31_c4']
['ADV', '34', 'Phi32_33_34_31_c4']
['ADV', '31', 'Phi32_33_34_31_c4']
['FIL', 'Phi32_33_34_31_c4', 1]
['CPO', 'Phi26_53_c5', [26, 53]]
```

FIGURE 5 – Premières ligne du tableau d'instruction après compilation

3.2 La compilation pas à pas

Avant de commencer à définir les potentiels à créer, les chemins de diffusion, etc, deux opérations sont nécessaires au bon fonctionnement du métacode.

D'une hard evidence à une soft evidence

Dans le langage probabiliste, on distingue deux types d'informations sur une proobabilité. Une soft evidence est une vraisemblance dont on connait chacune des valeurs. Par exemple, si la probabilité qu'il pleuve aujourd'hui selon qu'il fasse beau, couvert ou orageux est : $P(Pluie|Temps) = [0.1, 0.5, 0.8]$. Celles-ci sont directement implémentables, en passant le tableau de valeurs au potentiel de l'évidence. L'autre type de vraisemblance, l'hard evidence, est une certitude. Avant toute opération de compilation, on transforme l'input hard evidence en un tableau de valeurs plus classiques où toutes les probabilités sont nulles sauf la certitude, qui vaut 1.

Choix d'une clique principale

Comme vu dans la présentation, le coeur de la compilation repose dans l'absorption de l'information par une clique qui diffusera ensuite vers les cliques contenant les targets. Le choix de cette clique racine est important pour le temps de calcul du programme. Une première séléction se fait sur les cliques contenant une target, évitant ainsi au moins un parcours de diffusion dans l'arbre de jonction. Parmi les cliques candidates, celle ayant le maximum de voisins est choisie, optimisant la diffusion de l'information et éliminant les cliques isolées qui créeraient de nombreux calculs intermédiaires. Voici l'algorithme utilisé pour définir cette clique racine, vers qui tout converge :

Algorithm 1 Trouver la clique racine

```
VoisinsMax = −1
for I ∈ cliques do
  for X ∈ targets do
    if X ∈ variables(I) then
      if nbVoisins(I) > voisinsMax then
        VoisinsMax ← nbVoisins(I)
        CliqueRacine ← I
      end if
    end if
  end for
end for
return CliqueRacine
```

3.2.1 Création et Initialisation des potentiels

Dans la bibliothèque pyAGrum, conçue pour travailler avec réseaux bayésiens et arbres de jonctions, chaque clique est considéré comme un potentiel, initialisé avec la commande *pyAgrum.potential()* sur lequel peuvent s'appliquer les fonctions préconçues de multiplication, marginalisation etc de la bibliothèque pyAgrum. Ces structures n'existant pas dans les autres langages, il faudra initialiser les potentiels de cliques sous forme de tableaux de probabilités. C'est le but de la première partie du métacode de compilation, qui va parcourir l'arbre de jonction, sachant les évidences fournies et les targets demandés, et donnera au tableau d'instructions les variables à créer. Lors de la diffusion de l'information, comme vu dans l'exemple de la figure 4 un peu plus haut, il est nécessaire de collecter l'information, et donc les potentiels, des cliques environnantes afin de diffuser vers la clique suivante. Or, ces potentiels environnants seront modifiés lors du calcul, puisqu'ils diffusent eux aussi de l'information. Afin de garder une version inaltérée du tableau de probabilités d'une clique, nous créeons en plus des potentiels dits de diffusion, c'est ceux-ci qui seront modifiés.

3.2.2 Absorption et Diffusion de l'information

Le métacode commence par appeler la fonction *parcours* qui va parcourir en profondeur l'arbre de jonction et créer deux listes : une pour l'absorption, l'autre pour la diffusion. Cette fonction récurrente commence depuis la clique racine et retourne un booléen. Elle séléctionne d'abord ses voisins, et vérifie si ceux-ci contiennent une target pas encore rencontrée (une liste des targets est progressivement mise à jour). Si c'est le cas, un couple (clique, target) est enregistré pour signifier à la diffusion que cette

clique contient ou mène vers une target. L'appel récursif permet donc d'obtenir une liste d'absorption qui signalera au métacode le chemin à suivre jusqu'à la clique racine, puis une liste de diffusion qui elle donne le chemin de la clique racine vers les cliques les plus proches contenant les targets.

Algorithm 2 Parcours(jt, targets, racine, n)

```
liste = Voisins(jt,n)
intersection = False
if racine ∈ liste then
  liste.remove(racine)
end if
if len(liste) == 0 then
  return False
end if
for I ∈ liste do
  absorption.append([i,n])
  if isTarget(jt, targets, i) then
    targets.remove(i)
  end if
  rec = parcours(jt,targets,n,i)
  tar = rec or isTarget(jt,targets,i)
  if tar then
    diffusion.insert(0,[n,i])
    intersection = True
  end if
  return tar or intersection
end for
```

Une fois ces listes correctement définies, elles entrent en paramètres des fonctions suivantes du métacode, qui détermineront les opérations à effectuer pour l'inférence, c'est à dire la transmission de l' information selon le chemin voulu. Un premier parcours de la liste d'absorption crée les potentiels de séparateurs pour l'envoi d'un message d'une clique à une autre. Comme vu dans la partie théorique plus haut, pour cette opération, nous devons marginaliser le potentiel séparateur selon la clique 'émettrice' puis multiplier le potentiel de la clique 'réceptrice' par ce potentiel séparateur marginalisé. La fonction *sendMessAbsorb* réalise toute ces opérations. Aprés l'ajout de la commande de création du potentiel séparateur au tableau d'instruction, la fonction détermine l'intersection des deux cliques concernées pour créer le potentiel du séparateur, avant d'ajouter les consignes de marginalisation et multiplication selon les bon potentiels de cliques. Les potentiels sont correctement récupérés par l'identifiant unique du tableau d'absorption. Illustrons à nouveau avec un exemple clair :

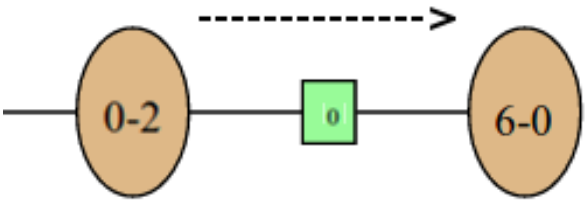

FIGURE 6 – Deux cliques et leurs variables, et un message à passer par le séparateur.

A ce stade de la compilation, le métacode parcourt la liste pour l'absorption et arrive à cet étape, où l'information doit être transmise du potentiel Φ_{0-2} au potentiel Φ_{6-0}. Une consigne pour la création du potentiel séparateur est crée, $\Psi_{0-2>6-0}$, et les variables du séparateur y sont ajoutés : la variable 0 ici. Ensuite, on ajoute au tableau d'instructions les commandes de marginalisation selon la clique 2-0 : [MAR, $\Psi_{0-2>6-0}$, Φ_{0-2}, [0], [2,0]] et de multiplication de la clique 6-0 par le séparateur : [MUL, Φ_{6-0}, $\Psi_{0-2>6-0}$, [6,0], [0]].

Ainsi, toute l'information sera absorbée par la clique racine selon le chemin défini.

Passons à la diffusion vers les targets. Si diffusion il y a (elle n'a pas lieu d'être si une seule target est demandée par exemple), un algorithme similaire à celui de l'absorption régit le passage de l'information d'une clique à l'autre. Les instructions de création de potentiel séparateur sont ajoutées, les variables du séparateur y sont insérées. Comme vu précedemment, l'information doit être collecté dans les cliques voisines avant de pouvoir passer à la clique suivante. Une fonction de collecte a donc été implémenté, qui récupère l'information de tous les voisins d'une clique donnée, sauf un, passé en argument, et qui est dans notre cas la clique vers qui l'inférence est effectuée. L'inférence se termine lorsque la liste de diffusion a été parcourue.

3.2.3 Sortie et résultats

La dernière étape du métacode définit les instructions de renvoi du résultat. La target situé dans la clique racine, dont le potentiel avait été traité différemment puisque la collecte d'informations se fait sur tous les voisins de la clique racine, et sans diffusion de l'information ensuite, est aussi un cas à part en sortie. Les instructions données sont en revanche les mêmes, que la target appartienne à la clique principale ou non. Un potentiel final est crée, la variable correspondant à la target ajoutée, puis les commandes de marginalisation de la target selon les variables de la clique et de normalisation sont transmises au tableau d'instructions.

Récapitulons l'algorithme global de compilation :

Algorithm 3 compilation(jt, targets, evidences)

absorption = ()
diffusion = ()
cliquesTargets = ()
racine = mainClique(jt, targets)
targetsParcours = ()
{targetsParcours sera modifiée dans parcours, on lui supprime de plus les targets déjà contenues dans la clique racine}
$parcours(jt, targetsParcours, racine, racine)$
Fonctions pour la création de potentiels correctement labélisés
for $i \in range(absorption)$ **do**
  $sendMessAbsorb(jt, absorption[i][0], absorption[i][1])$
  {sendMessAbsorb ajoute au tableau les instructions pour le passage d'une clique A à une clique B : multiplication de potentiels, création de potentiels séparateurs, etc}
end for
if $diffusion$ **then**
  for $i \in range(diffusion)$ **do**
    $sendMessDiffu(jt, diffusion[i][0], diffusion[i][1], cliquesTargets)$
    {sendMessDiffu collecte d'abord les potentiels des cliques voisines puis ajoute les instructions nécessaires au tableau}
  end for
  for $i \in cliquesTargets$ **do**
    $collectAroundCliqueTarget(jt, i, diffusion)$
    {Cette fonction permet les dernères multiplications et marginalisations de potentiels avant la clique target, et labélise correctement les variables}
  end for
end if
for $i \in targets$ **do**
  de marginalisation et normalisation de $P(i)$
end for

Rappelons l'objectif du projet : créer dans plusieurs langages des fichiers de code calculatoire de probabilités. A l'aide du métacode vu ci-dessus, un tableau d'instructions a été crée, dans un pseudo-langage sciemment conçu pour être 'traduit' par les générateurs, ces modules qui vont permettre le passage des instructions aux lignes de code en python, Javascript, php, etc Voilà le résultat de la génération de la

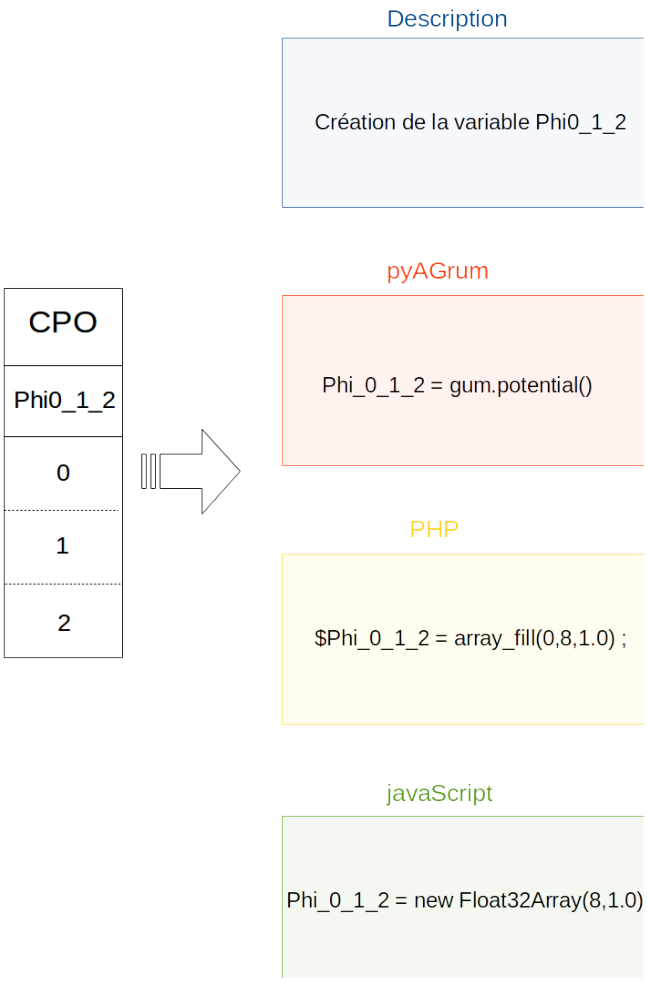

FIGURE 7 – De la compilation à la génération en différents langages

commande $'CPO, Phi_{0-1-2}, 0, 1, 2'$ dans les différents langages. Pour la création de potentiels, les variables du potentiel sont des paramètres du tableau d'instructions afin de calculer les tailles des tableaux à allouer en PHP et Javascript par exemple.

3.3 La Génération

Afin de tester notre métacode de compilation, le premier langage implémenté est le Python avec la bibliothèque de réseaux bayésiens **pyAgrum**. La plupart des structures de données et opérations existent déjà dans pyAgrum, comme les potentiels, la multiplication, marginalisation ou normalisation de ces derniers. Il s'agit donc d'une simple traduction d'une commande de notre tableau d'instructions en fonction pyAgrum. Ainsi, $[MUL, \Phi_{0-1-2}, \Psi_{0-1xx1-3}]$ devient $\Phi_{0-1-2}.multiplicateBy(\Psi_{0-1xx1-3})$. Chaque instruction du tableau a sa fonction correspondante dans les fichiers de génération, fonction précisant quelle chaîne de caractères à retourner. Cette chaîne correspond à la ligne de code voulue par le tableau d'instructions. Le coeur de la génération se passe dans la fonction dite *genere*, qui ouvre le fichier, y écrit l'header passé en configuration par l'utilisateur, puis la fonction qui va calculer la ou les probabilités. Pour ce faire, *genere* parcourt le tableau et vérifie chaque code d'instruction avant de les renvoyer à la fonction du générateur correspondant, avec les arguments suivant le code d'instructions. Voici à quoi ressemble la fonction genere dans **pyAgrumGenerator** :

```
def genere(self, bn, targets, evs, comp, nameFile, nameFunc, header):
    stream = open(nameFile,'w')
    stream.write("import pyAgrum as gum\nimport numpy as np\n")
    stream.write("'''Generated on : "+time.strftime('%m/%d/%y %H:%M',time.localtime())+"'''\n\n\n")
    stream.write(header+"\n\n")
    stream.write("def "+nameFunc+"(evs):\n")
    stream.write("\tres = {}\n")
    stream.write(self.initCpts(bn))
    for cur in comp:
        act = cur[0]
        if act == 'CPO':
            stream.write(self.creaPot(cur[1],cur[2]))
        elif act == 'ADV':
            stream.write(self.addVarPot('v'+cur[1],cur[2]))
        elif act == 'ASE':
            stream.write(self.addSoftEvPot(cur[1],cur[2],cur[3],cur[4]))
        elif act == 'MUC':
            stream.write(self.mulPotCpt(bn,cur[1],cur[2],cur[3]))
        elif act == 'MUL':
            stream.write(self.mulPotPot(cur[1],cur[2],cur[3],cur[4]))
        elif act == 'MAR':
            stream.write(self.margi(cur[1],cur[2],cur[3],cur[4],cur[5]))
        elif act == 'NOR':
            stream.write(self.norm(cur[1]))
            stream.write("\tres['"+cur[2]+"']=np.copy("+str(cur[1])+"[:])\n")
        elif act == 'FIL':
            stream.write(self.fill(cur[1],cur[2]))
        elif act == 'EQU':
            stream.write(self.equa(cur[1],cur[2]))
    stream.write("\treturn res")

    stream.close()
```

FIGURE 8 – La fonction genere de pyAgrumGenerator

Afin d'inscrire le projet dans un cadre plus général, une génération en python utilisant numpy, une bibliothèque plus courante, a été implémenté. PHP et javascript complètent les langages supportés. Pour pouvoir utiliser la fonction de calcul sans le réseau bayésien en paramètre, la première étape de la génération consiste à créer des variables locales pour toutes les probabilités conditionnelles du réseau. Les différences de traduction des commandes du tableau d'instructions sont ensuite principalement syntaxiques, notamment pour la création et l'initialisation des tableaux. La seule différence importante est l'utilisation en javascript de tableaux à une dimension, linéarisés. Lors de la génération des opérations de multiplication de potentiels, marginalisation, etc.., un parcours des variables est nécessaire pour définir la taille totale du tableau à une dimension.
L'efficacité est relative au langage souhaité pour l'utilisation des réseaux bayésiens, l'objectif du projet étant de fournir diverses options pour des codes calculatoires de probabilités. De plus, une importante partie du temps de calcul, notamment pour la compilation, dépend de la taille du réseau bayésien, des évidences et des targets choisies.

4 Utilisation et conclusion

Le projet utilise un fichier de configuration YAML, où l'utilisateur renseigne les différentes entrées du programme : réseau bayésien, évidences, targets, langage, nom du fichier à créer et de la fonction de calcul de probabilités, et éventuellement un header. L'éxecutable **metaGenBayes.py** importe ensuite le module de compilation **Compiler** et crée le tableau d'instructions. Il importe ensuite le module de **Generator** correspondant au langage souhaité. Tous ces éléménts sont disponibles sur le répértoire github du projet : `www.github.com/pierrestefani/metagenbayes`

Des optimisations peuvent encore améliorer ce projet. S'il fournit une base solide pour la génération dans d'autres langages, l'étape de compilation peut être sensiblement amélioré en modifiant le réseau bayésien d'origine pour une meilleure efficacité. En effet, une partie importante du réseau bayésien est souvent indépendante des probabilités qui nous intéressent : évidences et targets. Il serait donc possible de réduire le réseau avant tout parcours ou calcul.

5 Documentation

5.1 Compilation

5.1.1 Création et initialisation de potentiels

creationPotentialsAbsorp(bn, jt, absorp) : créer un potentiel qui sera utilisée pour l'absorption.

initPotentialsAbsorp(bn,jt, absorp) : initialise les potentiels utilisés pour l'absorption en affectant une table de probabilité conditionnelle à une clique contenant toutes les variables de cette table.

evsPotentials(bn, jt , evs, absorb) : créer et initialise les potentiels associés aux évidences puis ajoute l'information dans les potentiels précédemment crées avec initPotentialsAbsorp.

creaIniOnePotDif(bn, jt, ca, cb, indexca, absorp) : créer et initialise le potentiel de diffusion qui enverra son message de la clique "**ca**" à la clique "**cb**".

creaIniOnePotTar(bn, jt, ca, absorp) : créer et initialise le potentiel de diffusion qui contient une ou plusieurs targets.

creaIniPotentialsDiffu(bn, jt, diffu, cliquesTar, targets, absorp) : créer et initialise tous les potentiels nécessaires à la diffusion à l'aide des deux fonctions précédentes.

5.1.2 Absorption, diffusion

isTarget(bn,jt,target,n) : parcourt la liste des targets et la liste des variables de la clique n. Si une variable est une target, alors elle renvoie l'index de la target, sinon elle renvoie -1.

mainClique(bn,jt,target) : renvoie la clique racine vers laquelle on va faire converger l'information. Celle-ci doit contenir au moins une target, et doit avoir le maximum de voisins.

parcours(bn, jt, targetmp, n, r, absorp, diffu, cliquesTar) : parcourt l'arbre de jonction en profondeur et renvoie la liste des instructions à effectuer pour l'absorption et la diffusion.

sendMessAbsorp(bn, jt, ca, cb, indexca, absorp) : Envoie le message de l'absorption de la clique "**ca**" à la clique "**cb**" de la manière suivante :

1. Créer le potentiel séparateur
2. Marginalise le potentiel séparateur selon le potentiel "**ca**".
3. Multiplie le potentiel "**cb**" par le potentiel séparateur.

collectAroundCliq(bn, jt, ca, index, diffu) : collecte l'information manquante autour de la clique "**ca**". L'information manquante étant celle n'apparaissant pas dans la liste de la diffusion.

collectAroundCliqTar(bn, jt, ca, diffu) : collecte toute l'information autour de la clique contenant une target.

sendMessDiffu(bn, jt, ca, cb, index, diffu, cliquesTar) : Envoie le message de la diffusion de la clique "**ca**" à la clique "**cb**" de la manière suivante :

1. Collection l'information manquante autour de "**ca**" à l'aide de **collectAroundCliq**.
2. Créer le potentiel séparateur
3. Marginalise le potentiel séparateur selon le potentiel "**ca**".
4. Si "**cb**" ne contient pas de target, multiplie le potentiel "**cb**" par le potentiel séparateur. Sinon, ne fait rien (**collectAroundCliqTar** s'en occupera).

inference(bn, jt, absorp, diffu, targets, targetmp, cliquesTar) : réalise l'absorption et la diffusion puis s'occupe de collecter l'information autour des cliques contenant une target.

output(bn,jt,target, absorp, diffu, cliquesTar) : Retourne le résultat des targets de la manière suivante :

1. Créer un potentiel associé à la target **t**.
2. Marginalise ce potentiel selon le potentiel de target contenant **t**.
3. Normalise le potentiel selon **t**.

5.2 Génération

Ces fonctions renvoient en réalité des chaînes de caractères, qui composeront le code calculatoire. Les définitions de fonctions correspondent à ce qu'effectuent ces mêmes chaînes de caractères.
nameCpt(bn, var) : Labélise le tableau de probabilités conditionnelles de la variable var : P var sachant (ses parents).

initCpts(bn) : Crée le tableau de probabilités de chaque variable du réseau avec *nameCpt* et le remplit avec les variables correspondantes.

creaPot(bn,nomPot,varPot) : Crée le potentiel nomPot contenant les variables varPot.

addSoftEvPot(evid,nomPot,index,value) : Ajoute dans le potentiel nomPot l'évidence evid.

mulPotCpt(bn,nomPot,var,varPot) : Multiplie le potentiel correspondant à nomPot par la cpt identifié dans le réseau bayésien par var.

mulPotPot(bn,nomPot1,nomPot2,varPot1,varPot2) : Multiplie le potentiel 1 par le potentiel 2. Les variables de chaque potentiel sont nécessaires pour le calcul de dimensions et d'index.

margi(bn,nomPot1,nomPot2,varPot1,varPot2) : Marginalise le potentiel 1 selon le potentiel 2.

norm(nomPot) : Normalise le potentiel.

genere(bn,targets,evs,comp,nameFile,nameFunc,header) : Crée le fichier nameFile, y ajoute l'header et le prototype de nameFunc. Parcourt ensuite le tableau comp et renvoie chaque instruction de ce dernier à l'une des fonctions décrites ci-dessus.

5.3 Fonctions de tests

randomEvidenceGenerator(bn, prop) : Retourne un dictionnaire de soft evidences à partir d'un réseau bayésien et d'une proportion prop de cliques candidates.

targetSelecter(bn, evs) : Selectionne des targets qui ne sont pas déjà des évidences dans le réseau bayésien.

errorMarginCheck(generated,reference,targets,epsilon) : Vérifie que les valeurs obtenues dans le fichier généré generated sont dans la marge d'erreur epsilon avec la référence, et ce pour chaque target.