Université Claude Bernard – École Centrale de Lyon

Master Data Science

Modèles graphiques probabilistes
TP1 – Utilisation de BNLearn

Élèves

Damien Douteaux

Yann Vaginay

Enseignants

Alexandre Aussem

Maxime Gasse

Table des matières

Liste des figures

Liste des tables

Introduction

Ce rapport présente les différents travaux réalisés dans le cadre du premier TP de Modèles Graphiques Probabilistes sur le package BNLearn de R. Pour découvrir ce package, nous débuterons par des essais de commandes basiques sous R et quelques tests d'indépendances conditionnelles.

Ensuite, nous nous intéresserons à réaliser de l'inférence dans les réseaux Bayésiens à partir de ce package. En particulier, nous discuterons les avantages et les inconvénients d'une inférence exacte par rapport à une inférence approchée. Des exemples de calculs seront également menés, notamment pour réaliser un calcul avec la fonction probabiliste *do*.

Enfin, une fois la phase de formation sur BNLearn plus aboutie, nous nous intéresserons à un algorithme de construction de graphe à partir des données, l'algorithme PC. Nous coderons cet algorithme en R, et le testerons sur le jeu de données tests `alarme`. Nous comparerons également ce graphe avec d'autres graphes correpsondant aux mêmes données.

Remarque 1 : Tous les codes présentés dans ce rapport sont disponibles dans les fichiers R de cette archive. Les fichiers correspondand sont indiqués au fil du rapport.

Remarque 2 : Nous sommes tous les deux en double diplôme master Data Science en parallèle de notre troisième année à l'École Centrale de Lyon.

1 • Prise en main

1.1 Travail sous R

L'ensemble des codes utilisés et montrés dans cette partie sont disponibles dans le script `partie0.r`.

Pour commencer, réalisons quelques remarques et commentaires sur les commandes fournies. Nous passerons l'étape d'installation des packages.

```
library("bnlearn")

# Charge la base de données alarm
data("alarm")
```

Ce premier code nous permet de charger la librairie qui sera utilisée dans ce TP (`bnlearn`), ainsi qu'un jeu de données de tests appelé `alarm`.

```
# Infos sur les colonnes (nos variables)
ncol(alarm)
colnames(alarm)
```

Nous obtenons ainsi les informations suivantes :

- *Nombre de colonnes* Nous retrouvons ici la valeur `xxx`.
- *Nom des colonnes* En observant les valeurs, on retrouve celles disponibles dans la description de la base de données `alarm`.

```
# Infos sur les lignes (nos observations)
nrow(alarm)
rownames(alarm)
```

De manière similaire aux commandes précédentes, nous obtenons ici les informations (nombre et noms) des lignes de ce jeu de données.

```
# Dix premières lignes, 5 premières colonnes
alarm[1:10, 1:5]
```

Cette commande nous permet de récupérer une sous-matrice du jeu de données. Le filtrage est ici uniquement réalisé à partir des indices de colonnes et de lignes, mais pas de leurs noms.

```
# lignes 3, 5, 1, colonnes "ANES", "HIST" et "MINV"
alarm[c(3, 5, 1), c("ANES", "HIST", "MINV")]
```

Enfin, cette dernière colonne nous permet de récupérer des éléments du jeu de données donc les « coordonnées » sont dans la liste des numéros de ligne fournis et des noms de colonnes fournis.

1.2 Travail avec `bnlearn`

L'ensemble des codes utilisés et montrés dans cette partie sont disponibles dans le script `partie1.r`.

1.2.1 *Test d'indépendance conditionnelle*

Le package `bnlearn` permet de faire des tests d'indépendance conditionnelle. Pour cela, on utilise les commandes suivantes :

```
ci.test(x = "PAP", y = "SHNT", z = as.character(NULL), data = alarm, test = "mi")

res = ci.test(x = "PAP", y = "SHNT", z = "PMB", data = alarm, test = "mi")
res$statistic
res$p.value
```

Pour l'interprétation des résultats, il nous faut alors regarder la p-valeur du test statistique réalisé. Ainsi, le test vise à vérifier la véracité de l'hypothèse $\mathcal{H}_0$: $(X \perp\!\!\!\perp Y \mid Z)$ (où Z est éventuellement l'ensemble vide). Nous avons alors (seuil à 5%) :

$$\text{p-valeur} \begin{cases} > 0,05, \text{ on conserve l'hypothèse } \mathcal{H}_0, \text{ il y a indépendance} \\ \leq 0,05, \text{ on rejette l'hypothèse } \mathcal{H}_0, \text{ il n'y a pas indépendance} \end{cases}$$

On remarquera également qu'il est possible de récupérer la valeur du test statistique

Avec les deux exemples fournis ci-dessus, on obtient alors :

- *Test de* PAP $\perp\!\!\!\perp$ SHNT $\mid \varnothing$ On trouve une p-valeur inférieure à $2,2 \times 10^{-16}$, ainsi on rejette l'hypothèse d'indépendance.
- *Test de* PAP $\perp\!\!\!\perp$ SHNT $\mid$ PMB On trouve une p-valeur de $0,26$, ainsi on conserve l'hypothèse d'indépendance.

On en déduit donc les éléments suivants :

$$\text{PAP} \not\perp\!\!\!\perp \text{SHNT} \mid \varnothing \qquad\qquad \text{PAP} \perp\!\!\!\perp \text{SHNT} \mid \text{PMB}$$

Pour vérifier le résultat, nous regardons alors l'aspect du graphe d'`alarm`, qui est disponible sur le site du package `bnlearn`. Ce graphe est proposé en Figure 1.

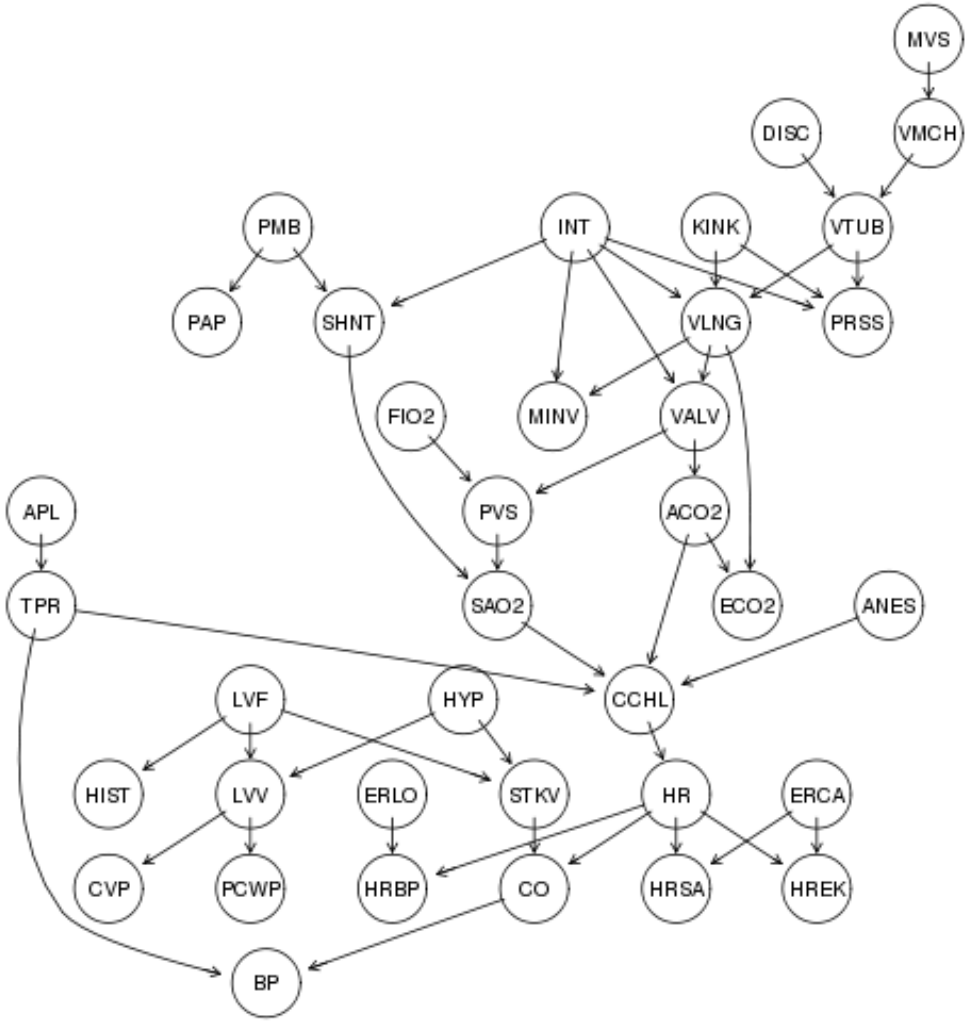


FIGURE 1 • Le graphe « théorique » du jeu de données `alarm`

Dans ce graphe, on regarde alors la structure autours des variables PAP, SHNT et PMB. Cette structure est détaillée à la Figure 2.

Il s'agit en fait d'une structure où PMB est une cause latente de PAP et SHNT. Ains, les résultats sont cohérents, il n'y a indépendance que si l'on connaît la cause latente (PMB).

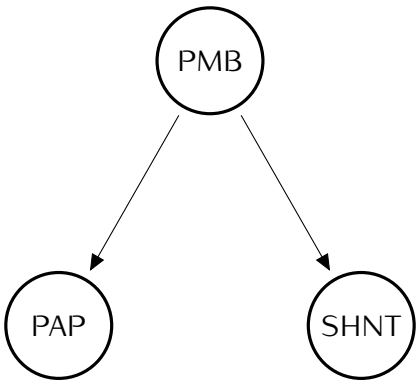


FIGURE 2 • La structure de graphe entre PAP, SHNT et PMB

1.2.2 *Relation entre deux variables*

En plus du test d'indépendance, `bnlearn` nous permet d'inspected les relations entre deux variables.

Étude de la variable PAP « seule » On peut étudier cette variable de manière indépendante à l'aide de diverses requêtes.

- *Somme des occurences* On peut sommer les occurences de chaque valeur de la variable. Un exemple est fourni ici :

```
> table(alarm[, "PAP"])

  HIGH    LOW NORMAL
  1151   1012  17837
```

- *Affichage de l'histogramme* Pour une visualisation plus graphique, on peut également demander à R de ploter les histogrammes des occurences comme ci-dessous.

```
  plot(alarm[, "PAP"])
```

L'histogramme obtenu est alors :

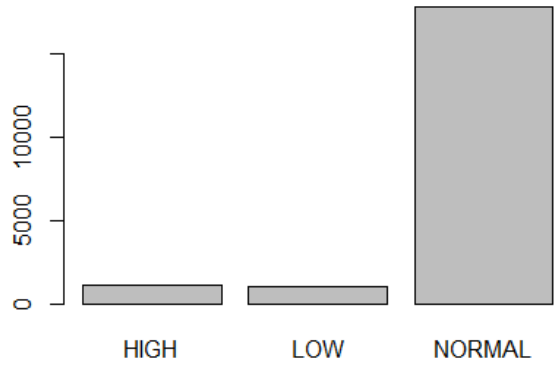


FIGURE 3 • Exemple d'histogramme d'occurence obtenu pour une variable avec R

- *Calcul des probabilités* Enfin, on peut calculer la probabilité d'obtenir les différentes valeurs de la variable. Ces probabilités correspondent au nombre d'occurence d'une valeur divisés par le nombre total d'échantillons. On obtient par exemple :

```
> prop.table(table(alarm[, "PAP"]))

```

```
   HIGH     LOW   NORMAL
0.05755 0.05060 0.89185
```

On vérifie en particulier que la somme de ces probabilités est bien égale à un.
La même étude est proposée avec SHNT, nous ne la détaillerons pas ici, dans la manière où elle n'apporte pas beaucoup d'information supplémentaire.

Étude de la relation entre PAP et SHNT Enfin, il est possible de réaliser une étude croisée de ces deux variables. On peut décomposer cette étude en plusieurs étapes :

- *Probabilité des paires de valeurs* Pour cela, on réutilise la fonction `prop`. On obtient alors par exemple :

```
> prop.table(ct)
         SHNT
PAP          HIGH   NORMAL
  HIGH    0.01265 0.04490
  LOW     0.00480 0.04580
  NORMAL  0.08555 0.80630
```

On remarque bien que la somme de toutes les variables de la table est bien égale à un.

- *Marginalisation* Ensuite, il est possible d'obtenir les mêmes tables, mais en marginalisant une variable par rapport à une autre. Deux exemples sont proposés ci-dessous :

```
> prop.table(ct, margin = 1)
         SHNT
PAP              HIGH      NORMAL
  HIGH    0.21980886 0.78019114
  LOW     0.09486166 0.90513834
  NORMAL  0.09592420 0.90407580

> prop.table(ct, margin = 2)
         SHNT
PAP              HIGH      NORMAL
  HIGH    0.12281553 0.05005574
  LOW     0.04660194 0.05105909
  NORMAL  0.83058252 0.89888517
```

Dans le premier cas, on calcule $p(SHNT \mid PAP)$, ie. on a marginalisé par rapport à PAP. Ceci est bien visible ici où la somme par ligne (ie. en sommant les valeurs de SHNT à PAP fixé) est égale à un.
Le second cas est analogue, mais dans le sens inverse, ie. pour $p(PAP \mid SHNT)$, ou les sommes valent un par colonne.

1.2.3 *Quelques calculs d'indépendance*

Nous allons ici répondre à la vérification des indépendance qui nous ont été demandées. Nous réutiliserons pour cela les éléments de code vus précédemment sans plus de détails.

STKV ⫫ HR | ∅

```
> ci.test(x = "STKV", y = "HR", z = as.character(NULL), data = alarm, test = "mi")$p.value
[1] 0.1313335
```

Dans ce cas, on trouve que p-valeur$= 0,13 > 0,05$, on conserve donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est vraie.

STKV ⫫ HR | CO

```
> ci.test(x = "STKV", y = "HR", z = "CO", data = alarm, test = "mi")$p.value
[1] 0
```

Dans ce cas, on trouve que p-valeur$= 0 < 0,05$, on rejette donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est fausse.

HR ⫫ CO | ∅

```
  > ci.test(x = "HR", y = "CO", z = as.character(NULL), data = alarm, test = "mi")$p.value
[1] 0
```

Dans ce cas, on trouve que p-valeur= $0 < 0,05$, on rejette donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est fausse.

HR ⫫ CO | STKV

```
> ci.test(x = "HR", y = "CO", z = "STKV", data = alarm, test = "mi")$p.value
[1] 0
```

Dans ce cas, on trouve que p-valeur= $0 < 0,05$, on rejette donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est fausse.

CO ⫫ STKV | ∅

```
> ci.test(x = "CO", y = "STKV", z = as.character(NULL), data = alarm, test = "mi")$p.value
[1] 0
```

Dans ce cas, on trouve que p-valeur= $0 < 0,05$, on rejette donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est fausse.

CO ⫫ STKV | HR

```
> ci.test(x = "PAP", y = "SHNT", z = "HR", data = alarm, test = "mi")$p.value
[1] 2.097088e-29
```

Dans ce cas, on trouve que p-valeur= $2,09 \times 10^{-29} < 0,05$, on rejette donc l'hypothèse $\mathcal{H}_0$, ie la relation proposée est fausse.

Conclusion En résumé, nous avons donc établi que :

STKV ⫫ HR | ∅ STKV ⫫̸ HR | CO HR ⫫̸ CO | ∅ HR ⫫̸ CO | STKV

CO ⫫̸ STKV | ∅ CO ⫫̸ STKV | HR

On peut vérifier ces résultats dans le graphe d'alarm. La structure qui nous intéresse est celle de la Figure 4.

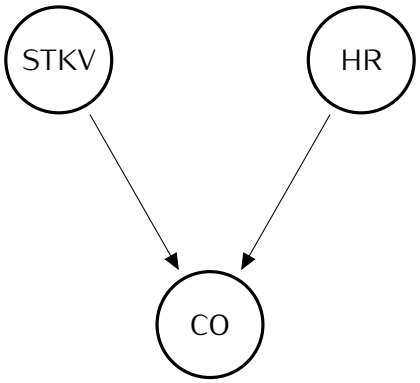
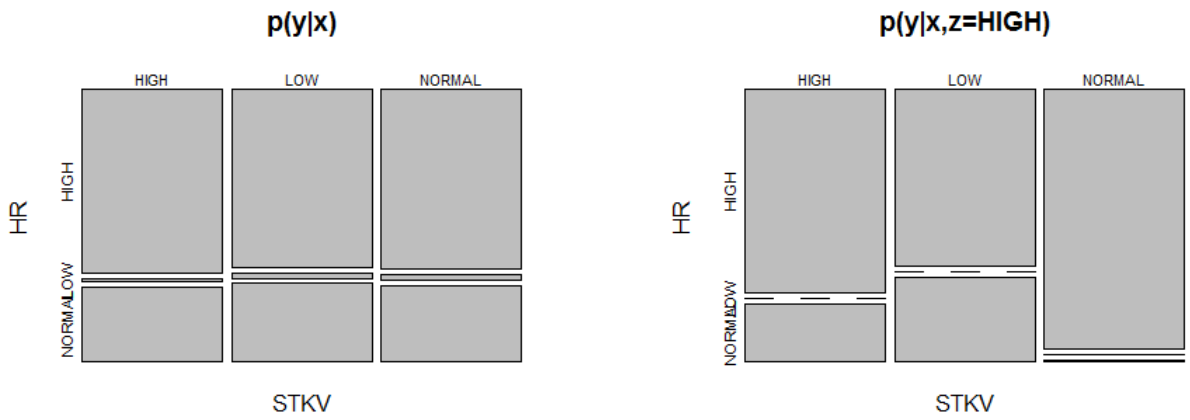


Figure 4 • La structure de graphe entre STKV, CO et HR

Il existe également un chemin entre CO et HR, mais ce dernier n'intervient pas dans la vérification des indépendances ci-dessous. On remarquera que la différence entre STKV ⫫ HR | ∅ et STKV ⫫ HR | CO se fait par ouverture de la V-structure, où on « casse » alors l'indépendance entre les variables.

1.2.4 *Inspection d'une relation d'indépendance*

Pour terminer, nous allons illustrer avec des données l'indépendance (resp. non indépendance) entre STKV et HR ne sachant rien (resp. sachant CO).



(a) STKV $\perp\!\!\!\perp$ HR | $\varnothing$ **(b)** STKV $\not\perp\!\!\!\perp$ HR | CO

Figure 5 • Histogrammes avec marginalisation pour l'indépendance entre STKV et HR avec ou sans connaissance de CO

En utilisant le code fourni, nous pouvons afficher les deux histogrammes avec marginalisation. Ces derniers vous sont proposés à la Figure 5.

On observe bien ici les résultats d'indépendance. En effet, dans le premier cas (Figure 5a), les probabilités d'observation de HR ne dépendent pas de la valeur de STKV. Dans le second cas (Figure 5b), on observe des distributions différentes, signe qu'il n'y a plus indépendance entre les deux variables.

2 • Inférence dans un réseau Bayésien

L'ensemble des codes utilisés et montrés dans cette partie sont disponibles dans le script `partie2.r`.

2.1 Construction du réseau

Avant d'aller plus loin dans l'inférence dans un réseau Bayésien, on commence par construire le réseau avec `bnlearn` et l'afficher.

Le code fourni dans le sujet nous permet d'obtenir le graphe suivant :

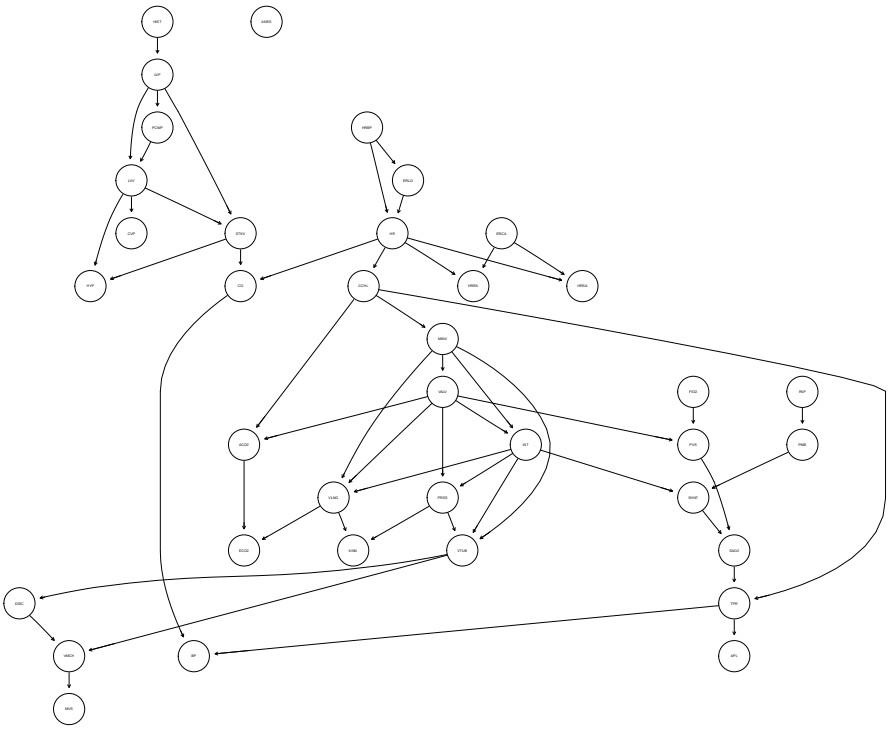


FIGURE 6 • Graphe obtenu en utilisant l'algorithme `hc` de `bnlearn`

Une remarque immédiate est que ce graphe est différent de celui théorique évoqué à la Figure 1 (on observe par exemple un nœud isolé, ce qui n'était pas le cas). L'algorithme créé donc un graphe proche mais pas équivalent à celui proposé initiallement.

Enfin, la deuxième partie du code nous permet d'afficher la loi multinomiale de CO, ie. les probabilités d'observer CO en fonction de ses variables voisines dans le graphe (STKV et HR) :

```
  Parameters of node CO (multinomial distribution)

Conditional probability table:

, , HR = HIGH

        STKV
CO                HIGH          LOW       NORMAL
  HIGH    0.976675468 0.007922082 0.947990788
  LOW     0.012511002 0.798479203 0.009622014
  NORMAL  0.010813529 0.193598716 0.042387197

, , HR = LOW

        STKV
```

```
CO               HIGH          LOW       NORMAL
  HIGH   0.040650407 0.005688282 0.007522332
  LOW    0.260162602 0.988623436 0.962740950
  NORMAL 0.699186992 0.005688282 0.029736718

, , HR = NORMAL

        STKV
CO               HIGH          LOW       NORMAL
  HIGH   0.686109815 0.008781063 0.010658701
  LOW    0.009954892 0.957135916 0.037542133
  NORMAL 0.303935293 0.034083021 0.951799166
```

2.2 Inférence approchée

Dans un premier temps, nous pouvons exécuter des inférence de probabilité en utilisant la commande `cpquery()`. Nous fournissons ci-dessous des exemples de plusieurs exécutions pour chaque ligne proposée.

Inférence de $p(\text{STKV} = \text{'high'} \mid \text{HR} = \text{'low'})$

```
> cpquery(bn, event = (STKV == "HIGH"), evidence = (HR == "LOW"))
[1] 0.04276316
[1] 0.03754266
[1] 0.05084746
[1] 0.03984064
[1] 0.07166124
[1] 0.04794521
```

On observe des valeurs différentes à chaque exécution, mais qui reste bien du même ordre de grandeur.

Inférence de $p(\text{STKV} = \text{'high'} \mid \text{HR} = \text{'low'}, \text{CO} = \text{'low'})$

```
> cpquery(bn, event = (STKV == "HIGH"), evidence = (HR == "LOW" & CO == "LOW"))
[1] 0.02341137
[1] 0.01851852
[1] 0
[1] 0.01915709
[1] 0.006920415
[1] 0.006968641
```

Le constat est le même que pour le cas précédent, bien que la variance semble plus élevée ici.

Explication des observations En regardant la documentation de la fonction `cpquery`, on observe qu'il y a utilisation d'un filtre de Monte Carlo pour le calcul. Ce dernier va ainsi introduire de l'aléatoire dans les calculs (mais les simplifier !), d'où les variations observées.

2.3 Inférence exacte

En utilisant le fichier fourni, les résultats de l'inférence exacte donneent des résultats constants, comme on peut le voir sur des exécutions.

2.3.1 *Méthode de calcul*

Dans un premier temps, nous allons préciser la méthode de calcul utilisée. Le code suggéré était le suivant :

```
p = exact.dist(bn, event = c("STKV", "HR", "CO"), evidence = TRUE)
sum(p["HIGH", "LOW", ]) / sum(p[, "LOW", ])
sum(p["HIGH", "LOW", "LOW"]) / sum(p[, "LOW", "LOW"])
```

La méthode proposée est de commencer par calculer des probabilités pour une loi multinomiale, et c'est l'étape la plus coûteuse. Une fois ces probabilités calculées, on peut inférer nos deux probabilités de tests (ce sont les mêmes qu'à la section précédente) en utilisant les formules suivantes :

$$p(\text{STKV} = \text{"high"} \mid \text{HR} = \text{"low"}) = \frac{p(\text{STKV} = \text{"high"}, \text{HR} = \text{"low"})}{p(\text{HR} = \text{"low"})}$$

$$p(\text{STKV} = \text{"high"} \mid \text{HR} = \text{"low"} \wedge \text{CO} = \text{"low"}) = \frac{p(\text{STKV} = \text{"high"}, \text{HR} = \text{"low"}, \text{CO} = \text{"low"})}{p(\text{HR} = \text{"low"}, \text{CO} = \text{"low"})}$$

Ces formules sont celles qui sont appliquées dans les sommes des lignes 2 et 3 du code précédent.

Les valeurs obtenues sont alors les suivantes :

```
> sum(p["HIGH", "LOW", ]) / sum(p[, "LOW", ])
[1] 0.04179577
> sum(p["HIGH", "LOW", "LOW"]) / sum(p[, "LOW", "LOW"])
[1] 0.01159239
```

2.3.2 *Comparaison des valeurs obtenues*

Les valeurs obtenues sont ici les mêmes à chaque exécution. On peut les comparer aux valeurs moyennes obtenues à la section précédente. Nous avons alors les résultats de la Table 1.

Type d'inférence	$p(\text{STKV} = \text{"high"} \mid \text{HR} = \text{"low"})$	$p(\text{STKV} = \text{"high"} \mid \text{HR} = \text{"low"}, CO = \text{"low"})$
Approchée	0,0484	0,0124
Exacte	0,0418	0,0116
Taux d'erreur	15%	6%

TABLE 1 • Comparaison des résultats d'inférence obtenus avec les deux types d'inférence

Dans la Table 1, les valeurs fournies pour l'inférence approchée proviennet d'une moyenne des valeurs proposées plus tôt dans le rapport. On observe donc que les variations observées avec la première méthode de calcul fournissait des valeurs moyennes avec jusqu'à 15% de taux d'erreur, ce qui peut être problématique pour des applications où l'on désirerait des calculs assez fin.

2.3.3 *Quelle méthode choisir ?*

Un cas intéressant est celui de l'inférence suivante :

```
p = exact.dist(bn, event = c("INT", "APL"), evidence = TRUE)
```

En effet, le temps d'exécution de cette ternière est (très) important pour la méthode exacte (supérieur à une minute sur nos machines), alors qu'un calcul approché fournit un résultat presque immédiat. Le code de la version approchée vous est proposé ci-dessous :

```
cpquery(bn, event = (INT == "NORMAL"), evidence = (APL == "TRUE"))
```

Ainsi, le choix de la méthode doit se faire selon un compromis entre rapidité et précision. La décision devra donc se prendre en relation avec le métier pour lequel on travaille, ses attentes et ses moyens.

2.4 Do-calculus

2.4.1 *Calcul d'une distribution conditionnelle*

Dans un premier temps, nous allons calculer la distribution conditionnelle de $p(\text{HYP} \mid \text{STKV})$. Nous proposons pour cela le code suivant :

```
p2 = exact.dist(bn, event = c("HYP", "STKV"), evidence = TRUE)
M_STKV = matrix(rep(colSums(p2),2), nrow=2, byrow=TRUE)
distrib = p2 / M_STKV
distrib
```

Comme nous l'avons déjà vu, nous commençons dans un premier temps par calculer les probabilité des différentes paires de valeurs que nous stockons dans p2.

Suite à cela, nous utilisons la relation $p(\text{HYP} \mid \text{STKV}) = \frac{p(\text{HYP}, \text{STKV})}{p(\text{STKV})}$ pour calculer chaque terme de la distribution de probabilité. Cependant, afin de gagner du temps, nous réalisons tous ces calculs sous forme matricielle, pour accélérer leur déroulé.

On remarquera également que les lignes 2 et 3 de ce code sont équivalentes à la ligne suivante : lstlisting[language=R] prop.table(p2, margin=2)

L'affichage de `distrib` est alors le suivant :

```
> distrib
       STKV
HYP            HIGH         LOW      NORMAL
  FALSE 0.96213631 0.42396571 0.87447904
  TRUE  0.03786369 0.57603429 0.12552096
```

Le résultat semble cohérent dans la mesure où les sommes de probabilité par colonne sont égales à 1, ce qui traduit bien une marginalisation par rapport à STKV.

2.4.2 *Exemple d'un calcul de « do »*

Nous allons désormais réaliser le calcul de la distribution de probabilité de $p(\text{HYP} \mid do(\text{STKV}))$. Pour utiliser la variable d'ajustement, il nous faut trouver une variable de sommation. En regardant le graphe de la Figure 1, on que le nœud par lequelle peut passer de la corrélation serait celui associé à la variable LVV, comme l'illustre la reprise de cette partie du graphe en Figure 7.

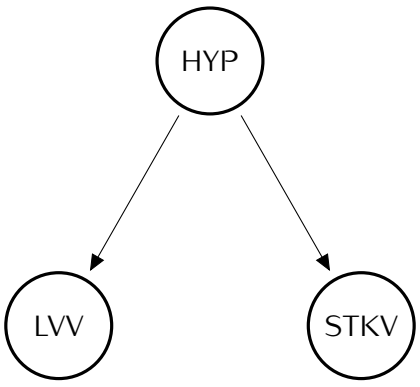


FIGURE 7 • Partie du graphe utilisée pour le calcul du « do »

L'application de la formule pour supprimer cette variables confondante (LVV) est alors la suivante :

$$\begin{aligned} p(\text{HYP} \mid do(\text{STKV})) &= \sum_{\ell \in \text{val(LVV)}} p(\text{HYP} \mid \text{STKV}, \text{LVV} = \ell) \cdot p(\text{LVV} = \ell) \\ &= p(\text{HYP} \mid \text{STKV}, \text{LVV} = \text{"High"}) \cdot p(\text{LVV} = \text{"High"}) \\ &\quad + p(\text{HYP} \mid \text{STKV}, \text{LVV} = \text{"Normal"}) \cdot p(\text{LVV} = \text{"Normal"}) \\ &\quad + p(\text{HYP} \mid \text{STKV}, \text{LVV} = \text{"Low"}) \cdot p(\text{LVV} = \text{"Low"}) \end{aligned}$$

Il nous reste alors à formaliser ceci avec un code R, qui vous est proposé ci-dessous :

```
p3 = exact.dist(bn, event = c("HYP", "STKV", "LVV"), evidence = TRUE)
margin.table(prop.table(p3, margin=c(2,3)) * aperm(array(margin.table(p3, 3), dim
    =c(3,3,2))), margin=c(1,2))
```

La première ligne calcule les probabilité d'observer les différentes paires de variables. Dans la suite, la seconde ligne réalise la somme attendue sur les valeurs de LVV pour chaque couple de valeurs de HYP et STKV.

On obtient alors le résultat suivant :

```
       STKV
HYP            HIGH         LOW      NORMAL
  FALSE 0.92609867 0.63796186 0.83371679
  TRUE  0.07390133 0.36203814 0.16628321
```

Le calcul semble cohérent, avec les sommes des valeurs par colonnes qui donnent toutes 1, signe que la marginalisation est bien réalisée par rapport à do(STKV). On remarque bien que ces valeurs diffèrent de celles obtenues pour le calcul précédent, signe que la corrélation a bien été éliminées de ces calculs pour ne conserver que la composante de causalité.

3 • L'algorithme PC

3.1 Construction du squelette

L'ensemble des codes utilisés et montrés pour la création du squelette du graphe sont disponibles dans le script `partie3_V1.r`.

Pour commencer, nous allons détailler les étables de l'algorithme permettant d'obtenir le squelette global du graphe.

3.1.1 *Création du graphe complet*

Dans un premier temps, l'algorithme suggère partir d'un graphe non dirigé complet. Ceci est réalisé par le code ci-dessous :

```
vars = colnames(alarm)
g = empty.graph(vars)
for (x in vars) {
  for (y in setdiff(vars, x)) {
    g = set.edge(g, from = x, to = y)
  }
}
graphviz.plot(g)
```

À titre indicatif, le graphe complet obtenu est proposé à la Figure 8.

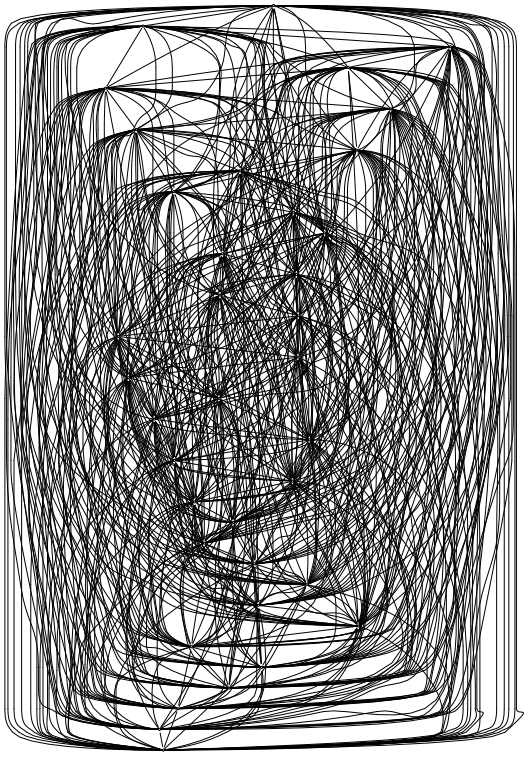


FIGURE 8 • Graphe complet obtenu à l'initialisation de l'algorithme PC

3.1.2 *Première étape de simplification*

Le code proposé Dans un premier temps, nous allons pour chaque paire de variables X et Y tester leur indépendance conditionnelle ($X \perp\!\!\!\perp Y \mid \varnothing$). S'il y a bien indépendance, l'arc est supprimé.

Le code que nous fournissons pour cette étape est alors le suivant :

```
for (i in 1:(length(vars)-1)) {
  x = vars[i]
  cat("Traitement de X = ",x,"\n")
  for (j in (i+1):(length(vars))) {
    y = vars[j]
    if(ci.test(x = x, y = y, z = as.character(NULL), data = alarm, test = "mi")$p
        .value > .01){
      g <- drop.edge(g, from=x, to=y)
    }
  }
}
```

L'exécution de ce code n'est pas immédiate et dure environ une minute. L'ajout de la commande `cat` permet cependant de vérifier le déroulé de l'exécution.

Explication du code La structure du code est assez immédiate, avec un parcours du graphe par valeurs de X et Y comme demandé dans l'énoncé. Ensuite, pour chaque couple, on teste l'indépendance conditionnelle à l'aide de `ci.test`, et l'on fixe le seuil de décision à $0,01$ (au lieu de $0,05$ usuellement, pour accélérer le déroulement de l'algorithme en conservant moins d'arcs). Dans le cas où l'hypothèse d'indépendance conditionnelle est conservée, on supprime l'arc testé à l'aide de `drop.edge`.

Résultat de l'exécution À l'issue de ce code, on obtient le graphe de la Figure 9.

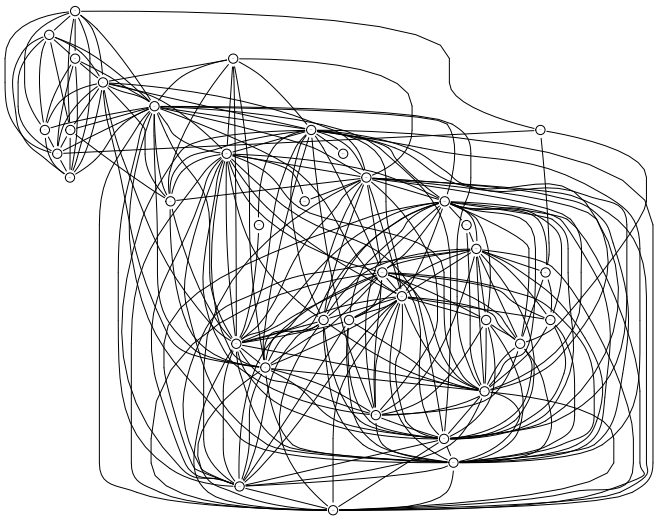


FIGURE 9 • Graphe obtenu après test des indépendances conditionnelles $X \perp\!\!\!\perp Y \mid \varnothing$

En comparaison de la Figure 8, on observe qu'une grande partie des arrêtes ont été supprimées par cette simple itération.

Remarque d'optimisation Dans le code proposé, on remarque que le parcours selon les valeurs de X et Y a été tronqué. En effet, au lieu de parcourir tous les X puis tous les $Y \neq X$, nous utilisons un parcours par indices. L'idée est ici que la première version va nous faire tester deux fois chaque arête, ce qui n'est pas utile.

Ainsi, pour un indice donné pour X (notons le i), l'indice pour Y (noté j) est pris dans $[\![i+1; N]\!]$, pour éviter les doublons. Cette remarque sera particulièrement utile pour la suite de l'algorithme.

3.1.3 *Deuxième étape de simplification*

Le code proposé Après avoir testé les indépendance de la forme $X \perp\!\!\!\perp Y \mid \varnothing$, nous pouvons tester celles de la forme $X \perp\!\!\!\perp Y \mid Z$ où Z est une variable adjacente à X (ie. un « voisin » dans le graphe).

Nous proposons alors de manière transparente vis-à-vis de ce qui précède le code suivant :

```r
for (x in vars) {
  cat('Traitement de X = ',x,'\n')
  for (y in g$nodes[[x]]$nbr){
    # On évite un cas inutile
    parentsnodes = setdiff(g$nodes[[x]]$nbr, y)
    if(length(parentsnodes) >= 1){
      for (z in combn(parentsnodes, 1)){
        if(ci.test(x = x, y = y, z = z, data = alarm, test = "mi")$p.value > .01)
            {
          g = drop.edge(g, from=x, to=y)
          # On fait un break car il ne sert à rien de tester les autres z, on
              saît déjà
          # qu'il faut supprimer l'arc
          break
        }
      }
    }
  }
}
```

Ici aussi, l'exécution de ce code n'est pas immédiate et va durer deux à trois minutes. Nous vérifions encore une fois le déroulé de l'exécution à l'aide de la commande `cat`.

Explication du code La structure du code est proche du premier proposé. On retrouve deux boucles principales, une pour itérer sur X et l'autre sur Y. La différence vient cependant de la méthode de parcours. Si dans le premier code nous avions utilisé des indices pour gagner du temps d'exécution, cela n'a pas été nécessaire ici. Ainsi, Y est désormais pris dans les voisins de X, et donc toute arête supprimée ne pourra par ré-apparaître une deuxième fois au calcul.

Concernant l'intérieur de ces deux boucles, on remarque une troisième boucle sur Z, qui est lui aussi pris dans les voisins de X différents de Y. Pour chaque Z, on va alors faire les tests d'indépendances. Dans le cas où on trouve une p-valeur qui permet de conserver l'hypothèse d'indépendance, on supprime l'arc et on arrête là la boucle sur Z, puisque les autres calculs ne sont plus utiles (l'arc est supprimé !).

Résultat de l'exécution À l'issue de ce code, on obtient le graphe de la Figure 10.

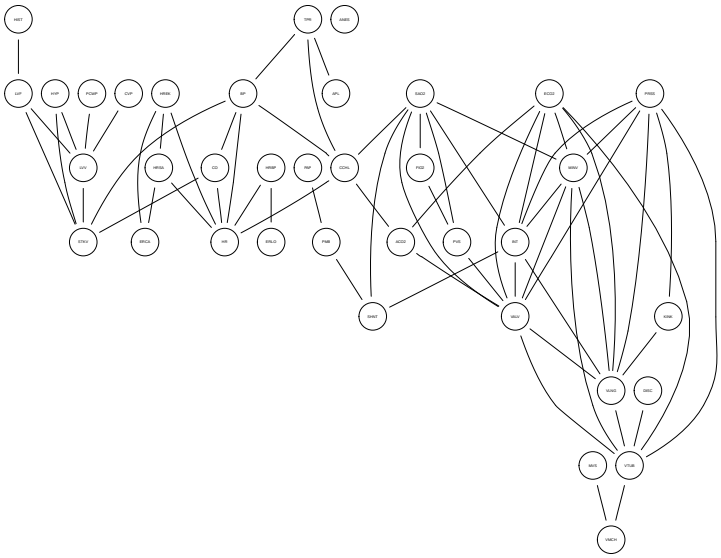


FIGURE 10 • Graphe obtenu après test des indépendances conditionnelles $X \perp\!\!\!\perp Y \mid Z$

3.1.4 *Généralisation de l'étape précédente*

Code proposé La suite de l'algorithme revient à répéter l'étape précédente pour des valeurs de $|Z|$ croissantes. Il suffit alors d'ajouter une boucle englobant le code précédent pour toutes les tailles possibles. Ceci est réalisé avec le code ci-dessous :

```
for (k in 1:(length(vars)-2)){
  cat('Itération pour k = ',k,'\n')
  for (x in vars) {
    cat('X = ',x,'\n')
    for (y in g$nodes[[x]]$nbr){
      # On évite un cas inutile
      parentsnodes = setdiff(g$nodes[[x]]$nbr, y)
      if(length(parentsnodes) >= k){
        for (z in combn(parentsnodes, k)){
          if(ci.test(x = x, y = y, z = z, data = alarm, test = "mi")$p.value >
             .01){
            Z_x_y[i,j] = z
            g = drop.edge(g, from=x, to=y)
            # On fait un break car il ne sert à rien de tester les autres z, on
               saît déjà
            # qu'il faut supprimer l'arc
            break
          }
        }
      }
    }
  }
}
```

Justification de l'intervalle de k On remarque que k prend ses valeurs dans $[\![1; N-2]\!]$. La borne supérieure est fixée à $N-2$, dans la mesure où l'on test *à minima* $X \perp\!\!\!\perp Y \mid Z$ avec $|Z| = 1$. Ainsi, deux variables seront à minima prise outre X, donc on ne pourra jamais chercher des Z de cardinal supérieur à cette valeur.

Temps d'exécution Ce dernier est important pour cette partie du code. Cependant, on observe que l'élagage se termine au bout de trois itérations. Ainsi, pour ce cas particulier, il serait possible de faire un `break` global à partir de ce nombre d'itération pour limiter le temps calcul important sinon (supérieur à 10 minutes).

3.1.5 *Évolution des graphes*

Pour conclure sur cette partie d'élagage du graphe complet initial, nous proposons en Figure 11 une vue côte à côte des graphes obtenus aux différentes étapes.

Le graphe final est en fait celui obtenu à la dernière itération proposée. Ainsi, la convergence vers le graphe final est très rapide, les tests d'indépendance les plus « basiques » réalisant la majorité du travail.

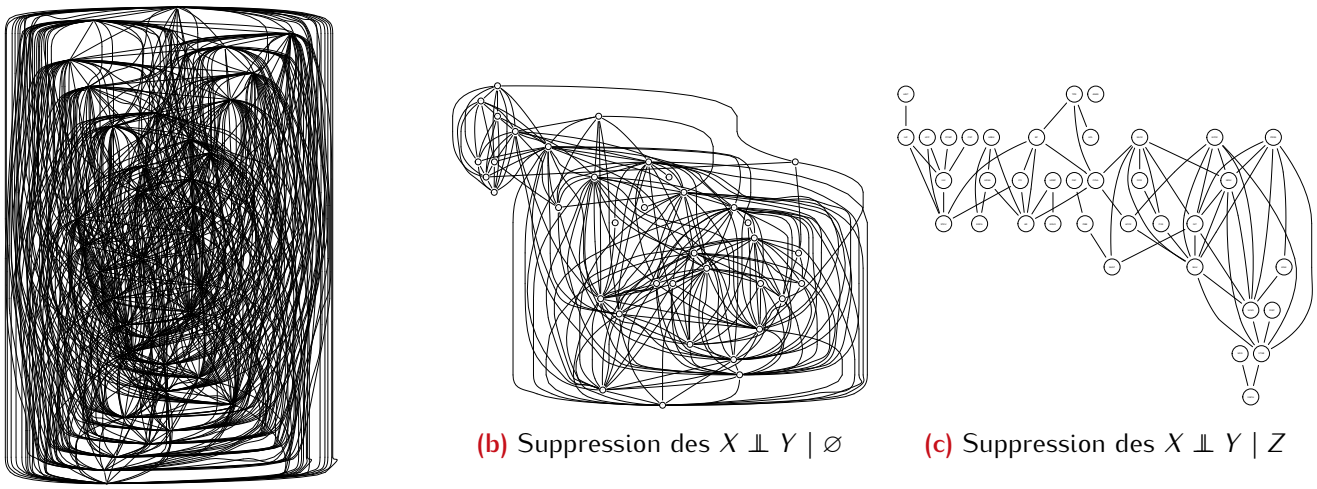
3.2 Orientation des arcs

L'ensemble des codes utilisés et montrés pour l'orientation des arcs du graphe sont disponibles dans le script `partie3_V2.r`. Ce script reprend aussi la partie de création du squelette pour y introduire $Z_{X,Y}$.

3.2.1 *Modification du code précédent*

Code proposé Dans un premier temps, nous allons modifier les codes précédents afin de stocker chaque ensemble $Z_{X,Y}$ qui a permis de retirer l'arc $X - Y$. Le code modifié vous est proposé ci-dessous :

```
vars = colnames(alarm)
g = empty.graph(vars)
for (x in vars) {
```



(a) Graphe complet initial

(b) Suppression des $X \perp\!\!\!\perp Y \mid \varnothing$

(c) Suppression des $X \perp\!\!\!\perp Y \mid Z$

FIGURE 11 • Évolution du graphe au fur et à mesure des itérations

```
  for (y in setdiff(vars, x)) {
    g <- set.edge(g, from = x, to = y)
  }
}

Z_x_y = matrix(-1, nrow=length(vars), ncol=length(vars));

for (i in 1:(length(vars)-1)) {
  x = vars[i]
  cat('Traitement de X = ',x,'\n')
  for (j in (i+1):(length(vars))) {
    y = vars[j]
    if(ci.test(x = x, y = y, z = as.character(NULL), data = alarm, test = "mi")$p
       .value > .01){
      g <- drop.edge(g, from=x, to=y)
      Z_x_y[i,j] = 0
    }
  }
}

for (k in 1:(length(vars)-2)){
  cat('Itération pour k = ',k,'\n')
  for (x in vars) {
    i = match(c(x),vars)
    cat('   Traitement de X = ',x,'\n')
    for (y in g$nodes[[x]]$nbr){
      j = match(c(y),vars)
      # On évite un cas inutile
      parentsnodes = setdiff(g$nodes[[x]]$nbr, y)
      if(length(parentsnodes) >= k){
        for (z in combn(parentsnodes, k)){
          if(ci.test(x = x, y = y, z = z, data = alarm, test = "mi")$p.value >
             .01){
            Z_x_y[i,j] = z
            cat("      Suppression de l'arc : X=",x,"   Y=",y,"   Z=",z,"\n")
            g = drop.edge(g, from=x, to=y)
            # On fait un break car il ne sert à rien de tester les autres z, on
               sait déjà qu'il faut supprimer l'arc
            break
          }
```

```
        }
      }
    }
  }
}

graphviz.plot(g)
```

Définition de `Z_x_y` Les principales modifications sont la définition de $Z_{X,y}$. La forme retenue ici est celle d'une matrice de dimension N initialisée à -1. Les conventions proposées pour cette matrice sont alors les suivantes :

- $Z_{X,Y}[i,j] = -1$ Alors l'arc est toujours présent dans le graphe.
- $Z_{X,Y}[i,j] = 0$ Alors l'arc a été supprimé par un test d'indépendance de type $X_i \perp\!\!\!\perp Y_j \mid \varnothing$.
- $Z_{X,Y}[i,j] = \{Z_1, ..., Z_\ell\}$ Alors l'arc a été supprimé par un test d'indépendance de type :

$$X_i \perp\!\!\!\perp Y_j \mid Z_1, ..., Z_\ell$$

De plus, la structure de notre code nous assure que nous ne travaillerons qu'avec la moitié supérieur de cette matrice (sa moitié inférieure aura donc en permanence des cœfficients de valeurs -1.

On remarquera également la manière de récupérer les indices i et j à partir des labels des variables en utilisant `match(c(x),vars)` qui nous renvoie l'index de la valeur dans `vars`.

Exemple d'exécution À titre d'exemple, nous fournisson une sous-matrice de `Z_x_y` pour illustrer l'aspect de cette matrice après les itérations proposées.

```
> Z_x_y
      [,1] [,2]  [,3]  [,4] [,5]  [,6]   [,7]   [,8]   [,9]   [,10] [,11]  [,12] [,13]
 [1,] "-1" "LVV" "LVF" "0"  "CO"  "LVV"  "0"    "0"    "0"    "HYP" "0"   "0"   "0"
 [2,] "-1" "-1"  "LVF" "0"  "CO"  "LVV"  "0"    "0"    "0"    "0"   "0"   "0"   "0"
 [3,] "-1" "-1"  "-1"  "0"  "LVF" "LVF"  "0"    "0"    "0"    "0"   "0"   "0"   "0"
 [4,] "-1" "-1"  "-1"  "-1" "-1"  "CCHL" "CCHL" "CCHL" "CCHL" "0"   "0"   "0"   "0"
 [5,] "-1" "-1"  "-1"  "-1" "-1"  "-1"   "HR"   "HR"   "HR"   "0"   "0"   "0"   "0"
 [6,] "-1" "-1"  "-1"  "-1" "-1"  "-1"   "HR"   "HR"   "HR"   "0"   "MINV" "0"   "0"
 [7,] "-1" "-1"  "-1"  "-1" "-1"  "-1"   "-1"   "HR"   "HR"   "0"   "CCHL" "0"   "0"
 [8,] "-1" "-1"  "-1"  "-1" "-1"  "-1"   "-1"   "-1"   "-1"   "0"   "HRSA" "0"   "0"
 [9,] "-1" "-1"  "-1"  "-1" "-1"  "-1"   "-1"   "-1"   "-1"   "0"   "HREK" "0"   "0"
```

On retrouve les différents éléments mentionnés ci-avant dans la définition de cette matrice.

3.2.2 *Mise en place des V-Structures*

Code proposé L'algorithme proposé est de réaliser une orientation en V-structure $X \to W \leftarrow Y$ dans le cas où X et Y ne sont pas liés par un arc et où $W \notin Z_{X,Y}$.

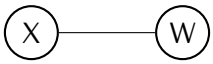
Pour cela, on propose alors le code suivant :

```
for (x in vars) {
  xx = match(c(x),vars)
  for (w in setdiff(g$nodes[[x]]$nbr, c(g$nodes[[x]]$children, g$nodes[[x]]$
     parents))) {
    for(y in setdiff(g$nodes[[w]]$nb, c(g$nodes[[w]]$children, g$nodes[[w]]$
       parents))) {
      yy = match(c(y),vars)
      tryCatch({
        if(Z_x_y[xx,yy] != -1 && !w%in%Z_x_y[xx,yy]){
            g = set.arc(g, from = x, to = w)
            g = set.arc(g, from = y, to = w)
        }
      }, error = function(e){})
    }
  }
}
```

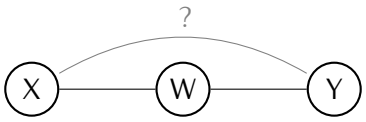
Explication du code L'idée est ici de parcourir toutes les variables X (dont on va récupérer l'indice dans $Z_{X,Y}$) ; puis pour chacune d'entre elle, parcourir les variables W qui ne sont pas au bout d'un graphe orienté partant ou allant vers cette instance. Ceci est réalisé en utilisant la différence d'ensemble `setdiff` et les fonction renvoyant les enfants et les parents d'un nœud dans un graphe orienté :

```
setdiff(g$nodes[[x]]$nbr, c(g$nodes[[x]]$children, g$nodes[[x]]$parents))
```

Ainsi, nous allons parcourir tous les arcs de la forme suivante :



Comme on parcourt toujours notre graphe sans répétition, on peut s'intéresser soit aux voisins de W uniquement. On parcourt donc tous les voisins Y de W (qui sont sélectionnées de manière analogue à l'ensemble des W). Nous sommes donc dans la situations suivante :



En particulier, aucun des arcs n'est orienté. La question est alors de savoir s'il existe un arc entre X et Y (en gris sur le schéma précédent). Ceci peut être lu directement via notre matrice $Z_{X,Y}$, en effet, si cette arc existe alors « $Z_{X,Y} = -1$ » (arc non supprimé). Ceci justifie donc l'intérêt d'avoir récupéré les indices de X et Y.

Si l'arc n'existe pas, il ne nous reste plus qu'à vérifier que $W \notin Z_{X,Y}$. Si cette situation se présente bien, alors on peut créer la V-structure, sinon on ne touche pas aux arcs.

Sortie de l'algorithme Après avoir appliqué ce code, on obtient le graphe partiellement orienté de la Figure 12.

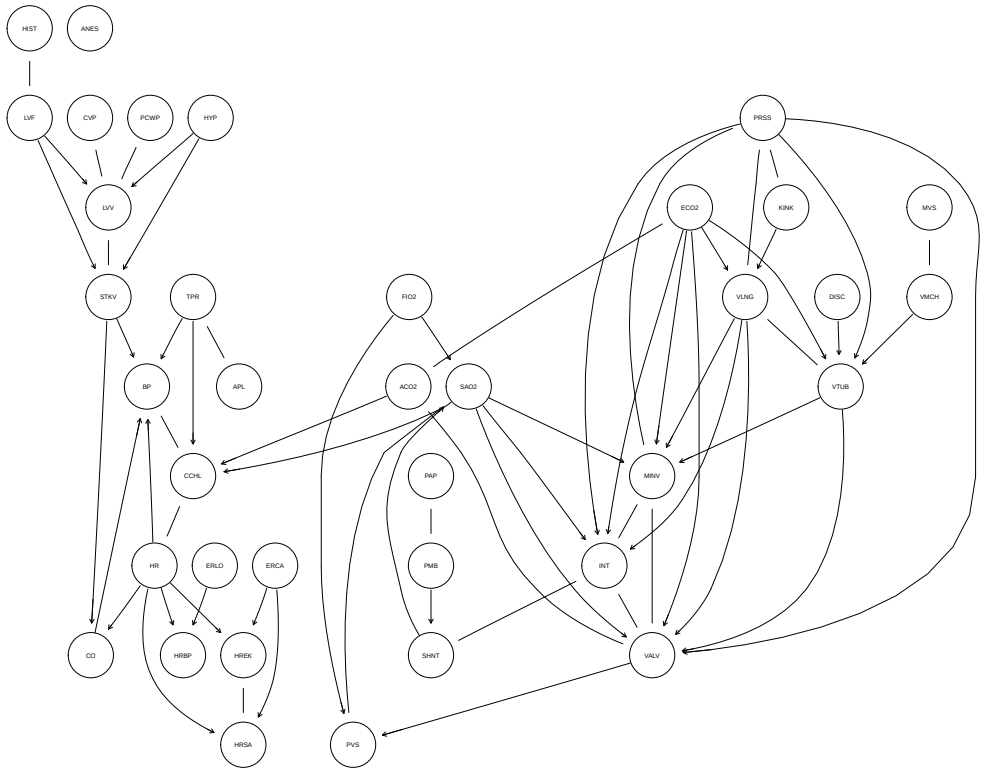


FIGURE 12 • Première étape de l'orientation des arcs

On remarque qu'il reste encore un nombre important d'arcs non orientés, qui seront traités dans la section suivante.

Remarque sur les problèmes de cycles Pour terminer avec cette partie, on notera l'utilisation d'un `tryCatch`. En effet, ce code ne garantit pas de ne pas créer de cycles. Ainsi, on échapper les cas problématiques avec cette commande, qui seront laissés non orientés et traités à l'étape suivante.

3.2.3 *Cas des arcs encore non orientés*

Code proposé Pour réaliser cette étape, nous proposons de parcourir tous les arcs non orientés restants. Pour chaque arc détecté, on regarde alors si l'orienter peut créer une V-structrue. Si ce n'est pas le cas, on créé l'arc. Après avoir passé tous les arcs en revue, certains auront été laissé de côté, qui seront traité après « au mieux ».

```
# Premier cas favorable où on ne risque par créer de V-structure.
# X-Y n'est pas orienté
for (x in vars) {
  for (y in setdiff(g$nodes[[x]]$nbr, c(g$nodes[[x]]$children, g$nodes[[x]]$
      parents))) {
    if (length(g$nodes[[x]]$parents) == 0) {
      tryCatch({
        g = set.arc(g, from = y, to = x)
      }, error = function(e) {
        if (length(g$nodes[[y]]$parents) == 0) {
          g = set.arc(g, from = x, to = y)
        }
      })
    }
  }
}

# Pour les arcs restants on fait au mieux pour éviter de créer des cycles
# On risque ici de créer de nouvelles V-structure...
for (x in vars) {
  for (y in setdiff(g$nodes[[x]]$nbr, c(g$nodes[[x]]$children, g$nodes[[x]]$
      parents))) {
    tryCatch({
      g = set.arc(g, from = x, to = y)
    }, error = function(e) {
      g = set.arc(g, from = y, to = x)
    })
  }
}
```

Le code est assez similaire à celui de l'étape précédente. On remarquera ici aussi l'utilisation de `tryCatch` pour récupérer d'éventuelles erreurs avec des créations involontaires de cycles.

Sortie de l'algorithme Après avoir exécuter cette dernière partie du code, nous obtenons notre graphe final, qui est représenté Figure 13.

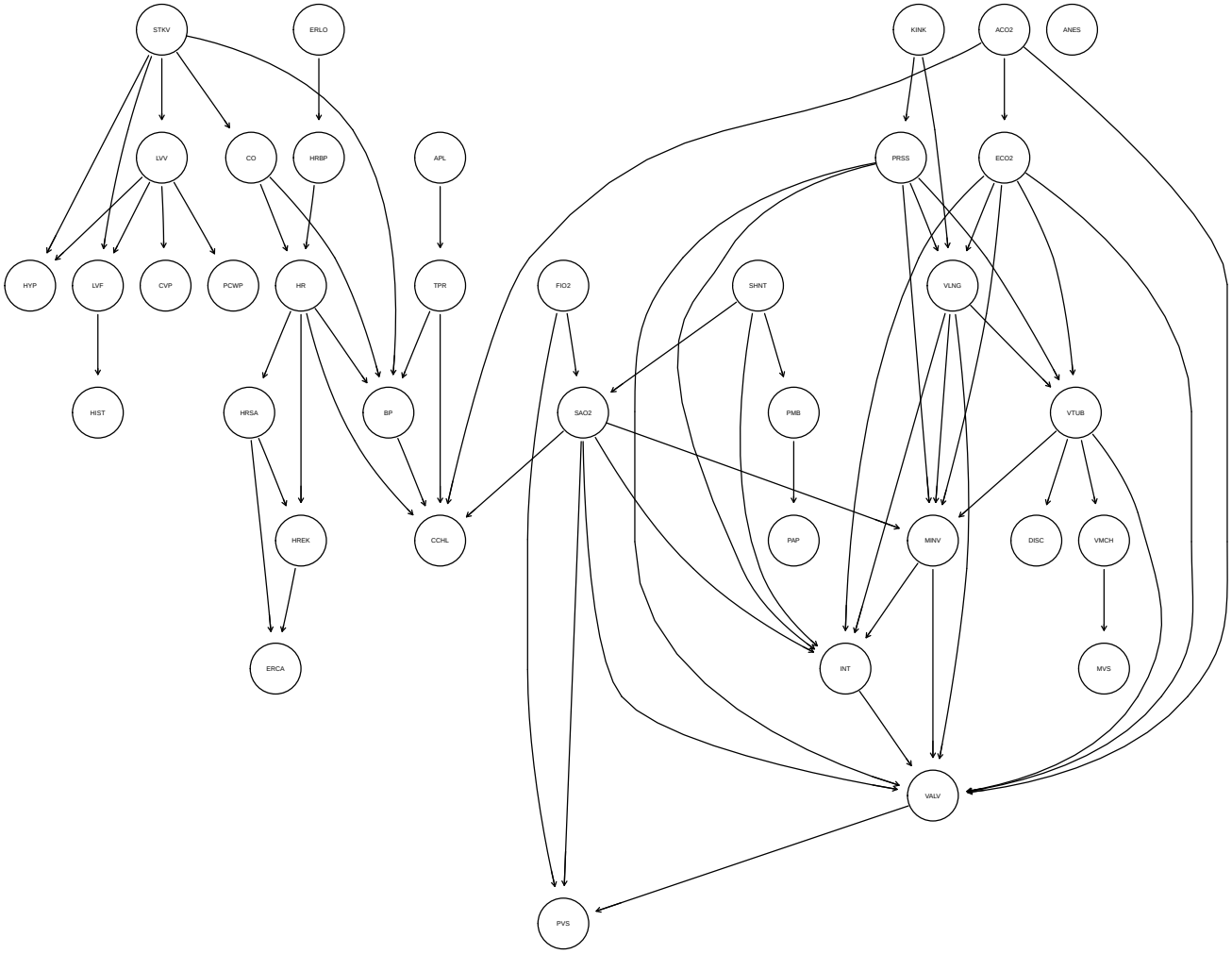


FIGURE 13 • Graphe obtenu à la fin de l'algorithme PC

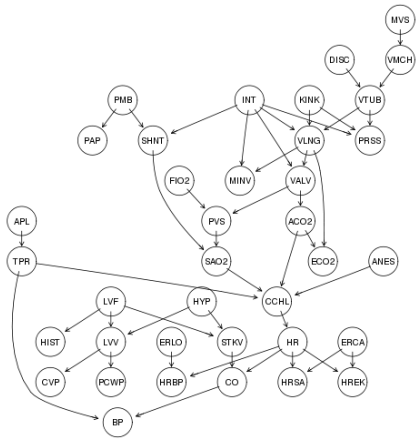
4 • Comparaison des graphes

4.1 Les trois graphes obtenus

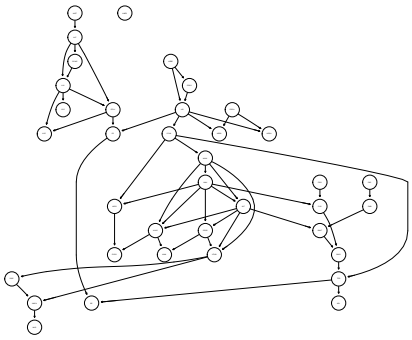
En définitive, nous avons vu trois graphes différents pour le même jeu de données :

- *Graphe « théorique »* Le graphe représentant les données dans le manuel de BNLearn.
- *Graphe obtenu avec* HC Ce graphe est obtenu avec un des algorithmes de construction de graphe de `bnlearn`.
- *Graphe obtenu avec PC* Ce graphe est celui que nous avons obtenu en codant l'algorithme PC à la main.

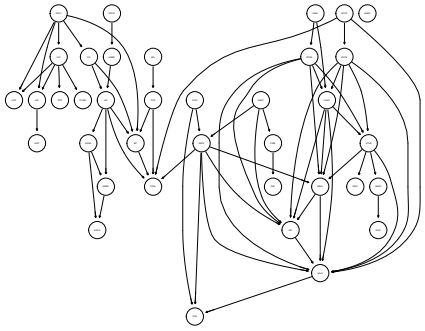
Pour faciliter la comparaison, ces trois graphes sont représentés Figure 14.



(a) Graphe théorique



(b) Graphe obtenu avec HC



(c) Graphe obtenu avec PC

FIGURE 14 • Comparaison des graphes obtenus avec les trois méthodes

4.2 Éléments de comparaison

Le premier élément qui saute aux yeux est que notre algorithme semble avoir conservé trop d'arcs dans le graphe. Cependant, notre graphe semble plus proche de celui obtenu avec BNLearn que de celui théorique. Par exemple, on retrouve le fait que la variable ANES soit solitaire.

De manière générale, la comparaison semble difficile sans utiliser de moyen informatique. On peut cependant estimer que la qualité de notre arbre est inférieur à celle obtenue avec HC, en particulier à cause du trop grand nombre d'arcs conservés.

4 • Conclusion

Lors de ce travail, nous avons pu appliquer les différentes notions vues lors des CMs, en particulier concernant l'inférence et les relations d'indépendances.

De plus, ce travail nous a permis d'explorer un algorithme de calcul de modèle graphique, l'algorithme PC. Si notre code ne fournit pas un arbre aussi bon que ceux disponibles via des algorithmes pré-fournis (comme l'algorithme `hc`), il fournit un premier modèle de réflexion et d'étude sur les données.

Un point de notre implémentation qu'il serait intéressant d'améliorer et l'optimisation des performances du code. En effet, il devrait être possible d'éviter certains cas dans le parcours pour éliminer les arcs. De même, la dernière étape pour éviter de créer des V-structures pourraient être affinées, afin d'éviter un maximum notre cas par défaut où l'on fait « au mieux ».