Sorbonne Université

Projet P-A.N.D.R.O.I.D.E

Dépôt Github : `https://git.io/J39ZB`

Branch and Bound pour les Diagrammes d'influence

Auteurs:
David PINAUD
Emilie BIEGAS

Encadrant:
M. Pierre-Henri WUILLEMIN

8 Mai 2021

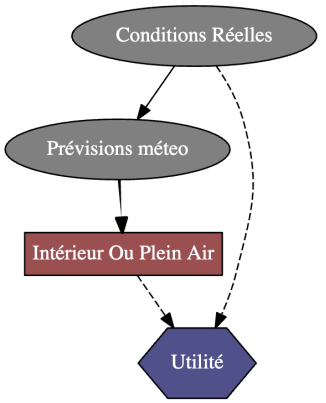
Table des Matières

1 Introduction

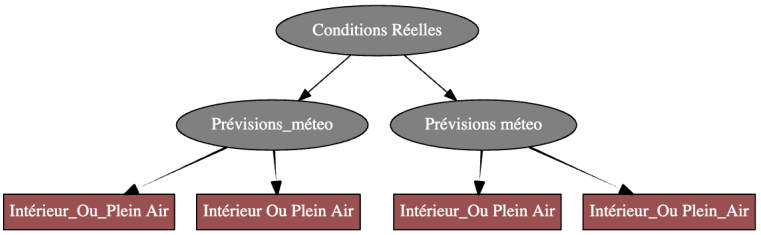
1.1 Les diagrammes d'influence (ID)

Un diagramme d'influence est une représentation graphique et mathématique compacte d'une situation de décision. Il s'agit d'une généralisation d'un réseau bayésien, dans lequel non seulement les problèmes d'inférence probabiliste, mais aussi les problèmes de prise de décision peuvent être modélisés et résolus. Le diagramme d'influence est une bonne alternative aux arbres de décision qui peuvent devenir extrêmement complexes. En effet, elles permettent de représenter des arbres de décision symétriques de façon compacte. On se place dans ce projet dans le cadre de l'espérance du maximum d'utilité.

Considérons, par exemple, une situation de décision concernant le choix d'une activité (de domaine : sport en plein air ou sport en intérieur) prise selon la prévision météorologique qui dépend, elle, des conditions réelles (leurs domaines est : pluvieux ou ensoleillé). Le diagramme d'influence 1 (a) ci-dessous permet de représenter cette situation et est équivalent à l'arbre de décision de la figure 1 (b).



(a) Diagramme d'influence



(b) L'arbre de décision symétrique associé

Figure 1: ID et Arbre de décision

Plus précisément, les diagrammes d'influence (ID) sont des graphes dirigés acycliques présentant trois types de nœuds. Les nœuds de *chance*, illustrés graphiquement par un ovale, représentent des variables aléatoire (dont le domaine est fini et non vide). Les nœuds de *décision*, illustrés graphiquement par un rectangle, représentent des variables de décision (dont le domaine est fini et non vide) tandis que les nœuds d'*utilité*, illustrés graphiquement par un octogone ou un hexagone, représentent une fonction d'utilité locale exprimant la préférence. Lorsqu'il y a plusieurs nœuds d'utilité, l'utilité totale en est la somme (c'est une décomposition partiellement additive de l'utilité).

Les arcs de ce graphe représentent une dépendance entre les nœuds et ont une signification différente en fonction du type de nœud à son extrémité. En effet, si un arc pointe vers un nœud de chance, cela représente une dépendance probabiliste; si il pointe vers un nœud de décision, il a un but informatif; enfin, si il pointe vers un nœud d'utilité, il représente une dépendance fonctionnelle.

Les IDs respectent deux hypothèses, une hypothèse de régularité (les décisions sont ordonnées dans le temps) et une hypothèse dite non-oubliant (chaque décision est conditionnée par

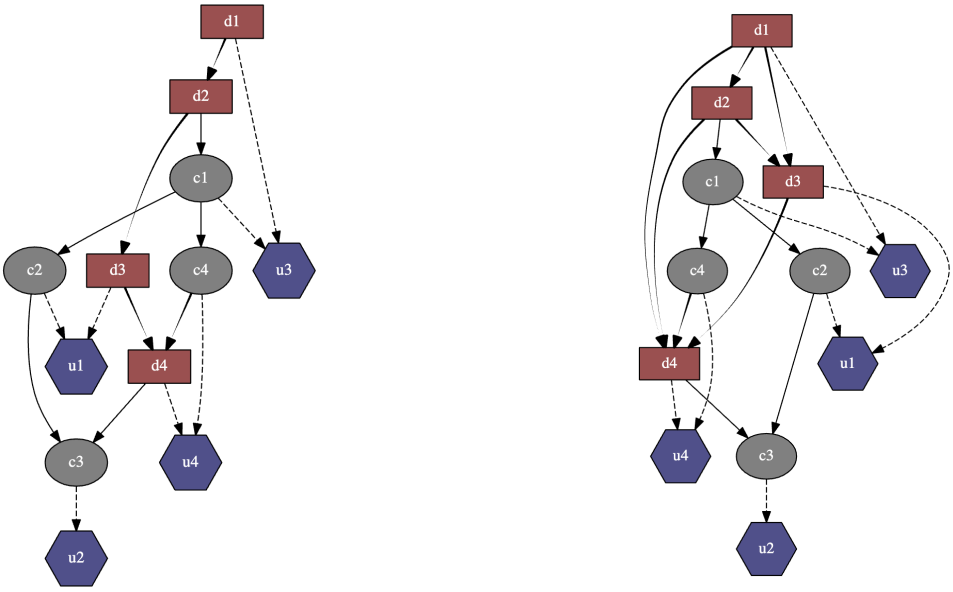
toutes les observations de décisions antérieures). L'ensemble des instanciations des nœuds de chance pour les décisions antérieures d'une certaine décision est appelé l'*historique*.

1.2 Les LIMIDs

Les *LImited-Memory Influence-Diagrams* (LIMIDs) sont des diagrammes d'influence qui assouplissent les deux hypothèses précédemment citées.

Tout d'abord, les LIMIDs assouplissent l'hypothèse non-oubliant de manière à conditionner une décision sur un nombre limité d'observations et de décisions antérieurs pertinentes (pour un compromis qualité/complexité).

Ensuite, assouplir l'hypothèse de régularité permet par exemple de modéliser la coopération de problèmes de décision multi-agents où un agent n'est pas au courant de décision d'autres agents. Les IDs forment alors un sous-ensemble des LIMIDs. On peut en voir un exemple ci-dessous.



(a) Diagramme d'influence

(b) LIMID associé à l'ID

Figure 2: ID et LIMID

1.3 Limites des implémentations actuelles

Les méthodes de résolution exactes actuelles de LIMIDs ou IDs sont limités par leur complexité en temps et en espace. Pour exemple, considérons un ID qui modélise un robot dans un labyrinthe représenté par une matrice $n \times m$ composé de cases *mur*, de cases *libre* et d'une case *objectif*. On représente le labyrinthe comme ci-dessous, les cases grises, blanches, et étoilés étant respectivement les cases *mur*, *libre*, et *objectif*.

Le robot est initialement placé sur une des cases *libre* et l'objectif du robot est d'atteindre la case *objectif*. À chaque étape, le robot peut se déplacer dans toutes les directions (nord,

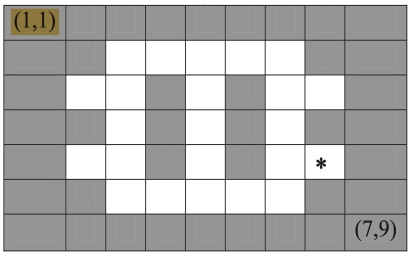


Figure 3: Un exemple de représentation d'un labyrinthe 9×7

sud, est, ouest et les diagonales) d'une seule case ou choisir de ne pas se déplacer. Il possède quatre capteurs pointés vers les quatre cardinaux qui indiquent au robot la présence d'un mur ou non.

À chaque étape, le robot choisit une direction cardinale où faire un pas puis le mouvement du robot suit une mesure de probabilité :

- Un pas vers la case voulu a une probabilité $pBouger$ de réussir.
- Échouer de bouger survient avec probabilité $pEchecBouger$.
- Un pas vers un *mur* a une probabilité de 0.
- À chaque étape, il y a une chance que le robot fasse un mouvement erratique :
  - Faire un pas vers la droite ou vers la gauche (c'est-à-dire vers l'est ou vers l'ouest si son choix est de se diriger vers le nord), s'il est possible de le faire, survient avec une probabilité de $pCote$ dans chacun des deux cas.
  - Faire un pas en arrière (c'est-à-dire vers le sud si son choix est de se diriger vers le nord), s'il est possible de le faire, survient avec une probabilité $pArriere$.
- Les autres probabilités sont normalisées afin de former une distribution de probabilité.

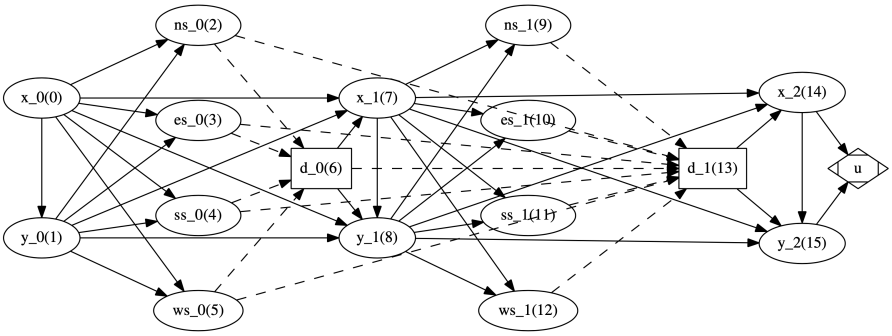


Figure 4: L'ID modélisant le problème du robot à deux étapes

A l'étape i, les nœuds de chance x_i et y_i représentent les coordonnées du robot sur la grille, ns_i, es_i, ss_i, ws_i les capteurs du robot dans les sens nord, est, sud et ouest respectivement. Les nœuds d_i sont les nœuds de décision. Pour visualiser le caractère exponentiel de la complexité en espace, voici un tableau récapitulatif des tailles des arbres de jonctions selon le nombre d'étapes ainsi que les graphes associés.

Nombre d'étapes	Tree-width (largeur de l'arbre)	taille en mémoire de l'arbre (en Go)
2	11	0,002
3	15	0,038
4	20	0,787
5	24	12
6	29	984
7	34	76 000
8	37	246 000
9	42	19 687 000
10	46	315 000 000

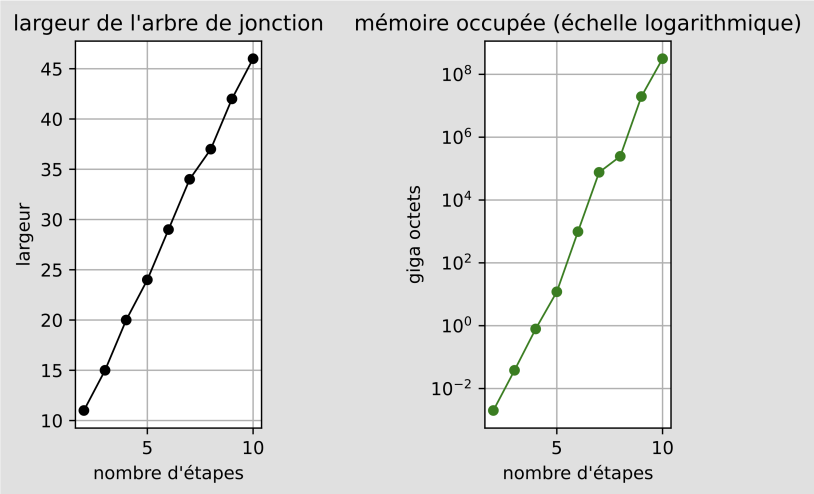


Figure 5: Graphes sur la complexité en temps et espace des méthodes actuelles

On parvient à résoudre cet ID avec les méthodes exactes actuelles (une machine avec 128Go de mémoire vive) sur des instances du problèmes comprenant jusqu'à 5 étapes. Au delà de 5 étapes, la mémoire nécessaire et le temps pour trouver la solution explose.

Il n'est pas envisageable de concevoir une machine capable de stocker autant de données pour résoudre de façon exacte cet ID. Il y a donc un réel besoin d'avoir une implémentation permettant de résoudre plus rapidement les IDs et en nécessitant moins d'espace.

1.4 But du projet

Ce projet avait un cadre prospectif et était orienté recherche. Ni nous ni notre encadrant ne savions quels en seraient les aboutissants. Pour mener à bien ce projet, nous avons donc d'abord eu besoin de nous familiariser avec le sujet et de trouver les documents adéquats pour répondre à nos questions. Après cela, nous avions donc pour but d'étudier et d'implémenter un algorithme de Branch and Bound pour résoudre des LIMIDs afin de proposer une solution à l'explosion en espace des algorithmes actuels. Nous nous appuierons fortement sur les travaux de Yuan et Hensen [4], et d'Acid et Campos [6] qui posent les bases et explicitent les étapes de construction de l'algorithme. Celui-ci devra être documenté et testé. Enfin, il est envisagé d'intégrer l'algorithme dans la librairie Python PyAgrum.

Ce sujet d'implémentation étant orienté recherche, nous n'avons pas de réel cahier des charges et avons décidé de l'inclure directement à ce rapport. Ce dernier correspond à la partie 4.1.

2 Algorithme de Branch and Bound pour la résolution de LIMIDS

2.1 Description informelle

Les LIMIDs sont résolus en trouvant une stratégie qui maximise l'utilité prévue. L'algorithme étudié dans ce projet résout les LIMIDs en les convertissant en un graphe ET/OU puis en effectuant un parcours en profondeur sur ces derniers.

Les nœuds ET sont alors des variables aléatoires correspondant aux nœuds chances qui sont informatifs à un nœud de décision et les nœuds OU sont des nœuds de décision (représentant des alternatives de décision), enfin les feuilles sont les nœuds d'utilités.

Durant le parcours en profondeur, un chemin racine-feuille représente une instance des nœuds d'information et de décision. Ces nœuds sont générés à la volée, ce qui permet de limiter l'utilisation de la mémoire. Les bornes supérieures et les évaluations des solutions candidates sont aussi calculées à la volée, permettant d'élaguer les chemins non prometteurs [3]. Les calculs des bornes supérieures sont faites sur un ID relaxé, dont la construction est détaillée plus tard dans le rapport.

Il faut souligner que cet algorithme n'applique pas classiquement le Branch and Bound comme dans le problème du sac à dos par exemple (où les nœuds sont des sacs avec plus ou moins d'objets)dont une branche peut être élaguée si une solution courante est meilleure que la borne supérieure de cette branche.

En effet, l'élagage se produit uniquement sur les branches sortant de nœuds de décision (ceux qui ne sont pas des feuilles) et non tous les nœuds. L'arbre se compose aussi de nœuds chances mais on ne peut pas élaguer leurs branches car ces nœuds construisent toutes les instances possibles du LIMID. On élague donc seulement sur les branches sortant des nœuds de décision car leurs évaluations sont commensurables puisqu'elles ont le même contexte. En contrepartie, un nœud de décision ne peut pas faire élaguer une branche d'un autre nœud de décision car elles ne possèdent pas le même contexte : si une branche d'un nœud de décision est élaguée car un autre nœud de décision est mieux évaluée, c'est une alternative du LIMID qu'on n'exploreras pas et donc on ne va pas résoudre entièrement le LIMID.

2.2 Implémentation des arbres ET/OU

Les implémentations actuelles des arbres ET/OU ne sont pas satisfaisantes au regard des besoins de l'algorithme, nous avons donc implémenté notre version de l'arbre. Cette implémentation contient trois classes : une classe modélisant un graphe ET/OU, une classe modélisant les nœuds ET (nœuds de chance) et enfin une classe modélisant les nœuds OU (nœuds de décision).

Dans l'arbre ET/OU, un nœud OU (décision) est créé pour chaque alternative d'instanciation des nœuds chances parents de ce nœud, et cela pour tous les nœuds de décision différents dans l'ID. C'est-à-dire que chaque nœud de décision dans l'ID a plusieurs contextes de nœuds chances possibles et tous sont représentés dans le graphe ET/OU par un nœud OU correspondant, à la suite du chemin des nœuds ET correspondants au contexte. Notre arbre se structure donc en couche, chaque couche étant constituée du même nœud de décision plusieurs fois (une fois par contexte possible) précédé de ses nœuds parents chance instanciés.

Ensuite, chaque nœud de décision va construire à la volée des branches au nombre égal, au maximum, au nombre de possibilités de décision de ce nœud (au maximum car il peut y avoir élagage). Ces fils vont être structurés de la même manière que précédemment en créant tous les contextes possibles pour le prochain nœud de décision. Soulignons encore qu'on ne peut ainsi pas élaguer entre nœuds de couches différentes mais seulement entre fils d'un même nœud de décision.

La structure détaillée de l'implémentation est accessible en annexe.

2.3 Implémentation de l'algorithme

2.3.1 ID relaxé et calcul des bornes

Pour permettre l'élagage, nous devons calculer les bornes supérieures et inférieures pour toute les valeurs d'instantiation possibles des nœuds de ET (décision).La borne inférieure est la meilleure valeur courante calculée pour un nœud OU. Pour sa borne supérieure, il faut résoudre un ID relaxé. Ce dernier est créé en ajoutant des informations à l'ID de départ et en simplifiant l'ID en supprimant les arcs non requis.

Les informations ajoutées sont sous la forme d'arcs entre des nœuds d'information et des nœuds de décision. Pour savoir quels arcs à ajouter à un nœud de décision, il faut calculer son *sufficient information set* (SIS).

L'ID relaxé est alors créé en deux étapes : tout d'abord, le SIS de chaque nœud de décision est calculé dans l'ordre temporel inverse des décisions (donné par le décideur) D_n, ..., D_1 faisant de chaque nœud chance dans leur SIS correspondant un de leur nœud parent. Enfin, les arcs d'information non requis sont supprimés de l'ID.

2.3.1.1 Sufficient Information Set (SIS)

Le calcul du SIS a été un point majeur de l'implémentation de l'algorithme et nous a pris une grande partie du temps alloué au développement.

Nous nous sommes basé sur l'article d'Acid et Campos [6] qui propose une méthode basé sur l'algorithme de Ford et Fulkerson afin de trouver l'ensemble séparant entre deux nœuds dans un graphe non dirigé et l'article de Yuan et al. [4] qui nous proposé les ensembles de nœuds sur lesquels appliquer l'algorithme.

Nous avons implémenté 3 versions de l'algorithme :

- La première se basait sur la construction complète d'un graphe auxiliaire à partir de l'ID, sur lequel on pouvait appliquer Ford et Fulkerson.
- La deuxième, plus performante, utilisait quant à elle un algorithme sur un graphe moralisé ancestral où on applique Ford et Fulkerson de manière implicite.
- La troisième, dont la majorité du code a été fourni par M. Wuillemin, était assez similaire à la deuxième mais présentait néanmoins une structure plus modulaire qui permettait aussi de calculer des ensembles séparants pour des réseaux bayésiens. Il contenait aussi des éléments qui permettait de mieux débugger et de comprendre le comportement de l'algorithme de Acid et Compos [6].

Nous soulignons le fait que la compréhension des articles était délicat, d'une part à cause de notre inexpérience sur ce sujet et d'autre part à cause de l'écriture confuse et expéditive de la part d'Acid et Campos [6]. L'implémentation de cet algorithme est complète mais il reste des points confus qui restent à discuter tel que le choix final des nœuds dans le SIS, la méthode de parcours du graphe moralisé ancestral dans certaines parties de l'algorithme ou encore la validité de la théorie derrière l'adaptation de l'algorithme Ford and Fulkerson. Il est d'autant plus important pour le projet de bien comprendre cette partie au vu de l'importance d'une relaxation informative du diagramme d'influence.

2.3.2 Branch and bound

Après avoir construit l'ID relaxé, nous pouvons commencer à appliquer l'algorithme.

L'idée de départ était d'appliquer un Branch and Bound sur un *graphe* ET/OU construit à partir d'un LIMID donné, d'évaluer les solutions candidates et de calculer les bornes supérieures avec un algorithme itératif utilisant les arbres de jointures fort construit eux aussi à partir de l'ID.

Par manque de temps, nous nous sommes résolus à :

- Utiliser un arbre ET/OU au lieu d'un graphe ET/OU. En effet, l'idée originale était de construire un graphe ET/OU au lieu d'un arbre ET/OU pour la résolution de LIMID car ils offraient des optimisations en temps et en espace en exploitant le fait que, dans un LIMID, comme l'hypothèse du non oubliant est assoupli, beaucoup de décisions seront prises dans le même contexte. Ce qui augmenterait à la fois le temps de calcul en faisant des inférences plusieurs fois, mais aussi l'espace requis car on doit stocker plus de nœuds. Le graphe ET/OU devait en plus des nœuds ET et OU, avoir des nœuds spéciaux qui facilite l'implémentation de l'algorithme.
- Calculer les bornes supérieures et les évaluations avec de l'inférence exacte complète à chaque fois, ce qui augmente grandement notre complexité en temps (qui va même devenir le goulot d'étranglement de l'algorithme).

Nous devons donc commencer par construire l'arbre ET/OU, et ce à la volée. Pour cela, nous commençons par choisir dans l'ordre de décision donné avec le LIMID, le premier nœud de décision d_0.

Nous générons alors les alternatives possibles d'instanciations des parents de d_0 que nous appellerons *couche* de d_0. Cette couche est composée de nœuds ET (chance), nous ajoutons alors des nœuds OU (décision) correspondant à d_0 sur les feuilles de cette couche. Pour exemple, si d_0 est le nœud (6) de décision dans l'ID de la figure 2, ses parents sont ns_0, es_0, ss_0, ws_0. Ceux-là ont pour domaine $[True, False]$ donc il y a $2^4 = 16$ alternatives possibles pour d_0. Lorsque cela est fait, s'il y a plus d'un seul nœud de décision dans le LIMID, nous calculons la borne supérieure des nœuds OU générés (pour toutes les valeurs de son domaine) et nous générons la couche pour les nœuds de décision suivants dans l'ordre de décision donné pour une branche d'un des nœuds OU qu'on vient de créer (on garde en mémoire, pour les nœuds OU, quelles branches ont été développées, évaluées, ou élaguées). Lorsque nous arrivons au dernier nœud de décision, il faut faire une inférence pour tous les nœuds de décision de cette couche feuille, pour tous les éléments de domaines de ces nœuds afin qu'on puisse remonter par un processus similaire à l'induction arrière dans la résolution d'un arbre de décision (la valeur d'un nœud chance est la moyenne pondérée par les probabilités postérieures des valeurs de ses fils et la valeur d'un nœud de décision est le maximum des valeurs de ses enfants). Lorsque cette induction arrière qui remonte la couche atteint son nœud de décision père, nous pouvons regarder si on peut élaguer en comparant les bornes supérieures des autres branches du nœud de décision avec l'évaluation de la couche (on met aussi à jour la meilleure solution courante pour ce nœud de décision s'il le faut). Lorsque un nœud de décision a toutes ses branches évaluées ou élaguées, il est *processed*. Lorsque tous les nœuds de décision dans une couche sont *processed*, nous pouvons faire l'induction arrière pour cette couche. Nous répétons ces étapes jusqu'à atteindre la racine de l'arbre ET/OU.

Si, par contre, il n'y a qu'un seul nœud de décision, la première couche est alors une couche feuille dans l'arbre ET/OU et nous procédons comme dans le premier cas.

À la fin d'une exécution, nous avons donc un arbre ET/OU dont les nœuds de décision ont une décision optimale et une valeur d'utilité pour cette décision, les nœuds chances ont des probabilités postérieures associées ainsi qu'une valeur. Nous avons donc calculé pour toutes les alternatives une décision optimale.

2.4 Test

Nous avons écrit plusieurs test unitaires, notamment pour tester l'algorithme de calcul du SIS. Celui-ci calcule en réalité l'ensemble séparant minimal dans un réseau bayésien. Nous avons donc testé sur des réseaux bayésiens générés aléatoirement si l'ensemble séparant minimal de deux nœuds tirés au hasard font bien de ces deux nœuds des nœuds indépendants. Nous avons aussi testé le branch and bound sur des IDs aléatoires afin de vérifier qu'il prenne en charge tous types de diagrammes.

2.5 Documentation

Une documentation des classes et des méthodes est disponible en annexe sous format PDF, elle a été rédigée en anglais dans le style numpy. Une version en ligne est aussi disponible sur `https://davidpinaud.github.io/PANDROIDE_InfDiag_BandB/`

3 Évaluation de l'algorithme Branch and Bound

3.1 Performances en temps

Sur l'exemple de la figure 2, l'ID relaxé construit avec les SIS était mauvais et donnais des valeurs (les bornes supérieures) de MEU très élevées comparés aux évaluations des solutions candidates. Il n'y avait donc pas d'élagage possible et l'exécution était très longue : 25 minutes comparé à moins d'une seconde avec la méthode classique.

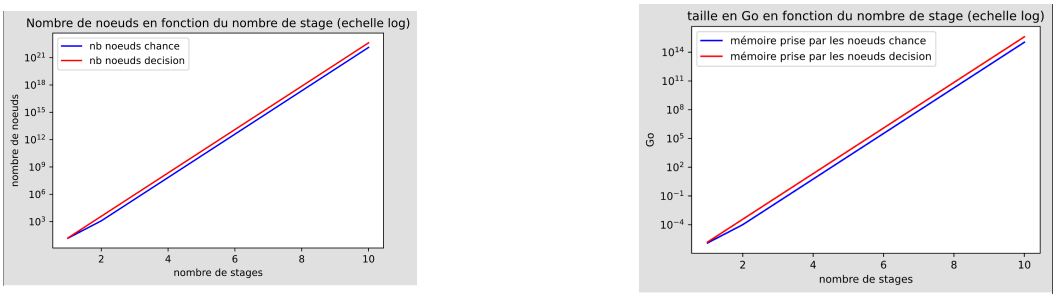
La performance en temps dépend largement de la qualité de la relaxation et donc de la borne supérieure qui influe directement sur le nombre d'inférences à faire par les élagages qu'elle permet. Elle est aussi très impactée par le fait que nous avions choisi de faire des inférences exactes à chaque calcul de bornes ou d'évaluation au lieu d'implémenter la méthode itérative.

Les performances en temps du branch and bound comparées à celles de l'algorithme Shafer Shenoy sont mauvaises aussi. Ce dernier est exponentiellement plus rapide proportionnellement au nombre de nœududs et d'arcs dans le LIMID. Ceci est peut être du à la mauvaise optimisation de notre code, couplé aux concessions sur l'algorithme initial décidées, mais aussi au fait que le branch and bound a été écrit en Python et l'implémentation de l'algorithme de Shafer Shenoy en C++.

3.2 Performances en espace

La performance en espace de notre algorithme dépend largement de la structure du LIMID. En effet, l'arbre est d'autant plus profond qu'il y a de nœuds de décision et d'autant plus large que le nombre de parents des nœuds de décision et leur domaine. Nous rappelons aussi que la performance en espace est impactée par notre choix d'utiliser des arbres ET/OU et non des graphes ET/OU. Elle dépend aussi des potentiels des nœuds et de la qualité du relaxé : plus le relaxé est bon, plus on élaguera.

À titre de comparaison avec l'exemple de la figure 2, voici le nombre de nœuds créés et la quantité de mémoire occupée dans le graphe ET/OU en fonction du nombre de stages.



(a) Nombre de nœuds/Stage

(b) Mémoire/Stage

Figure 6: Performance en espace

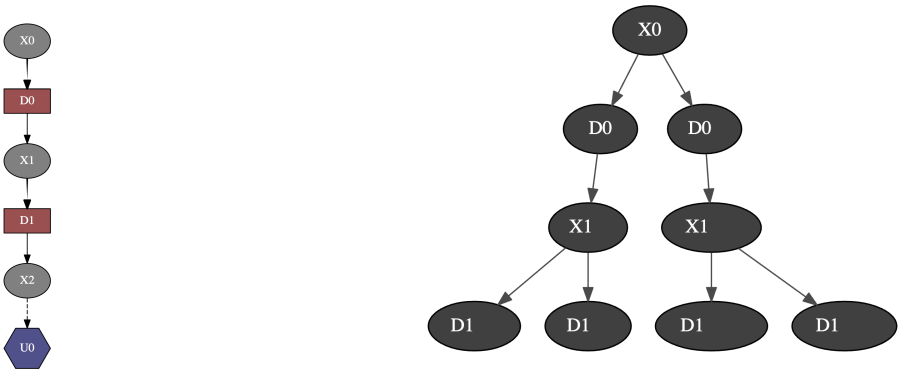
Remarquons que c'est la même complexité exponentielle que les algorithmes de résolutions actuels qui ressort. C'est parce que le LIMID relaxé n'était pas informatif et aucun élagage n'a été fait ! Cet exemple souligne encore l'importance du relaxé, en effet, l'élagage peut théoriquement diminuer de façon drastique le nombre de branches explorées. Sur des exemples

où des élagages ont pu être fait, nous observons une réduction souvent drastique (souvent plus de 40%) de la quantité de mémoire occupée et du nombre de nœuds créés dans l'arbre ET/OU. En comparaison directe avec l'implémentation de Shafer Shenoy en C++, le branch and bound occupe beaucoup plus d'espace sur le disque. Cette différence de performance est sûrement due au choix du langage d'implémentation et à l'optimisation de l'algorithme.

3.2.1 Exemples

Voici des exemples illustrant les performances en temps et en espace ainsi que l'importance de l'élagage :

3.2.1.1 Exemple 1



(a) Diagramme d'influence

(b) L'arbre ET/OU construit pendant le branch and bound

Figure 7: ID et Arbre ET/OU

En effet, voici l'arbre ET/OU complet si aucun élagage n'est fait :

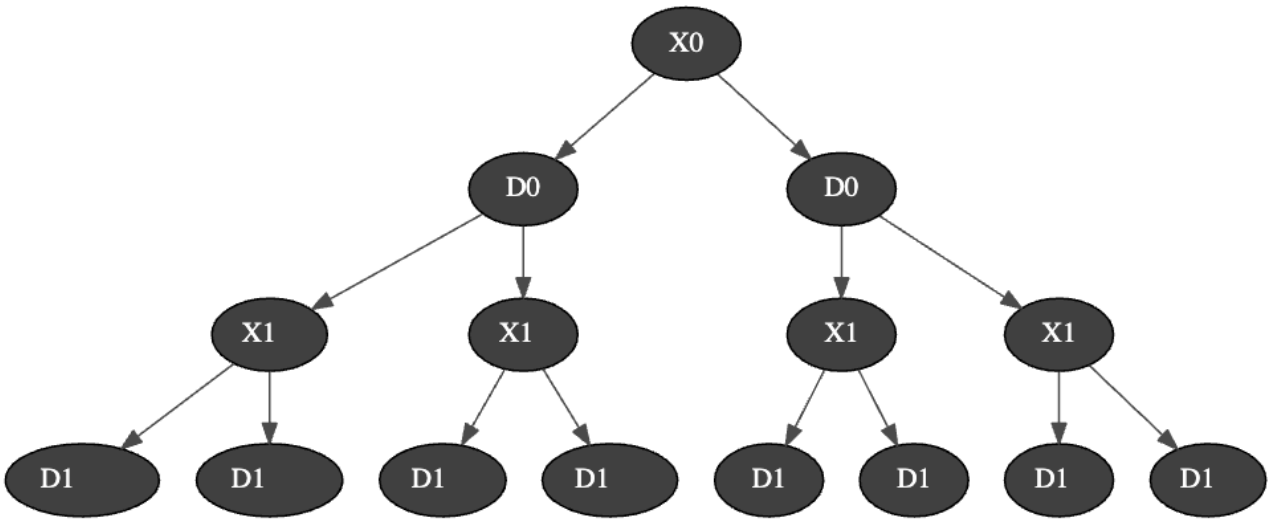


Figure 8: Arbre ET/OU complet

Une branche des deux nœuds D_0 est élaguée, ce qui nous fait économiser la construction de 6 nœuds, représentant une économie d'espace de 40%. Cela nous libère aussi de $4 \times 2 = 8$ calculs d'inférence exacte (fait sur les nœuds D_1), ce qui représente 50% des calculs d'inférence et de deux remontées de sous-arbres (qui économise aussi du calcul car on n'a plus à calculer les valeurs des nœuds chances X_1 avec les probabilités à posteriori).

En comparaison avec l'inférence Shafer Shenoy, notre algorithme a fait l'inférence en 0.00956 secondes comparé à 0.00054 pour la méthode état de l'art soit $17,7$ fois plus rapide.

Shafer Shenoy a construit un arbre de jonction de largeur 2 et a utilisé environ 32 octets de mémoire tandis que le branch and bound a utilisé un arbre ET/OU de profondeur 4 et environ 432 octets soit 32 fois plus d'espace occupé. Shafer Shenoy étant écrit en C++ et le branch and bound en Python, il est aisé de comprendre l'écart de performance entre les deux algorithmes.

3.2.1.2 Exemple 2

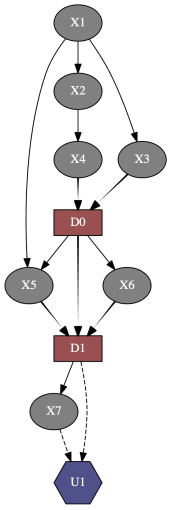
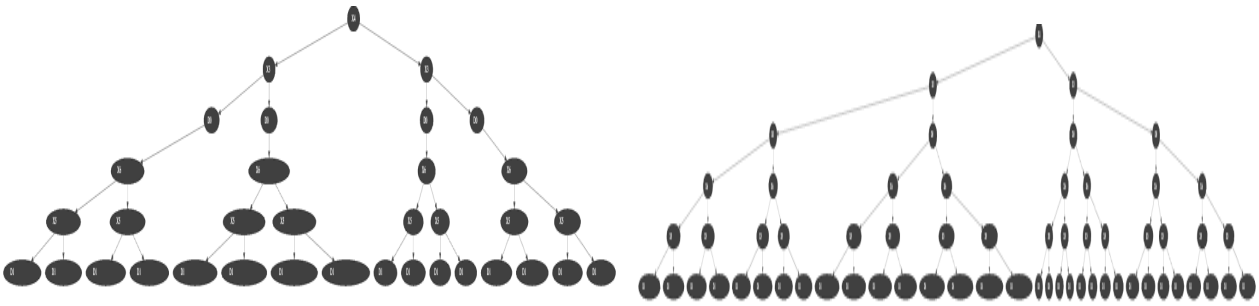


Figure 9: Diagramme d'influence



(a) L'arbre ET/OU construit pendant le branch and bound

(b) Arbre ET/OU complet

Figure 10: Arbre ET/OU complet et élagué

Le branch and bound a élagué 4 branches, ce qui a permis de ne développer que 35 nœuds au lieu de 63 pour un temps d'exécution de 0.07885 secondes et un arbre de profondeur 6

occupant 1680 octets comparé à un arbre de jonction de largeur 3 occupant environ 64 octets et un temps d'exécution de 0.00126 secondes pour Shafer Shenoy. Le branch and bound est donc 204 fois plus lent et occupe 26 fois plus d'espace que la méthode actuelle.

3.2.1.3 Exemple 3

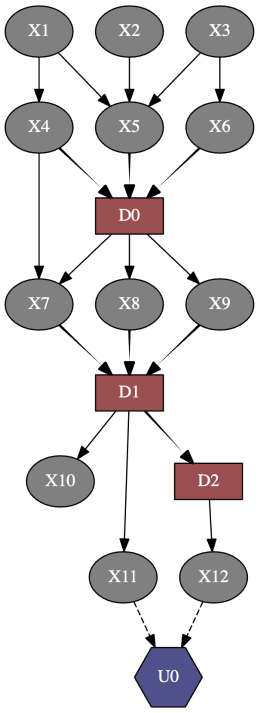


Figure 11: Diagramme d'influence

Le branch and bound a élagué 8 branches, ce qui a permis de ne développer que 135 nœuds au lieu de 255 pour un temps d'exécution de 0.56199 secondes et un arbre de profondeur 7 occupant 6480 octets comparé à un arbre de jonction de largeur 4 occupant environ 128 octets et un temps d'exécution de 0.00275 secondes pour Shafer Shenoy. Le branch and bound est donc 62 plus lent et occupe 50 fois plus d'espace que la méthode actuelle.

4 Intégration possible de l'algorithme dans PyAgrum

4.1 Déliverables du projet

Le projet a pour but de réaliser :

- Un état de l'art et stabilisation des algorithmes pour la résolution avec Branch and Bound, en particulier l'analyse du calcul des bornes et des probabilités postérieures.
- Une implémentation efficace et compacte des arbres ET/OU.
- Une intégration de l'algorithme correctement documentée (style python en anglais) dans l'API pyAgrum vérifiant l'implémentation des méthodes (liste non exhaustive) :

  - makeInference : Réalise l'inférence sur un LIMID.
  - PosterieurUtility : Retourne la probabilité postérieure d'un nœud.
  - LIMInfluenceDiagram : Retourne le LIMID associé à l'instance de la classe.
  - optimalDecision : Retourne la décision optimale pour un certain nœud décision basée sur le critère MEU.
  - MEU : Retourne l'utilité maximale espérée de l'inférence.
  - junctionTree : Retourne l'arbre de jonction associé au LIMID
  - andOrTree : Retourne le graphe de jonction développé
  - une fonction pour visualiser l'arbre ET/OU

- Une série de tests unitaires effectués sur les méthodes avec l'API TestUnit.
- Une série de benchmark validant l'utilité de notre implémentation de la méthode branch and bound face aux méthodes existante.

5 Conclusion

En conclusion, après avoir testé les algorithmes actuels sur un exemple, nous avons implémenté un algorithme de résolution de LIMID basé sur le branch and bound qui fait honneur à la promesse de performance en terme d'espace de ce type d'approche (et qui pourrait également l'être en terme de temps de calcul en utilisant un algorithme incrémental), malgré la faible performance en comparaison directe avec l'algorithme de Shafer Shenoy dû à l'optimisation insuffisante de notre code.

Nous avons souligné le rôle majeur de la relaxation dans les performances et montré que l'algorithme pour la création de celle-ci n'est pas encore très bien compris au vu des difficultés que l'on a rencontré pendant l'implémentation du programme concernant le calcul de SIS.

Nous avons donc appris à avoir un œil critique sur les documents étudiés et nous avons pu approfondir des notions dans le domaine d'aide à la prise de décision. Nous avons également appris à utiliser plus en profondeur la librairie PyAgrum et sa documentation.

Enfin, nous remercions M.Pierre-Henri Wuillemin pour ses contributions et ses conseils sans lesquels nous n'aurions sans doute pas pu mener ce projet à terme.

6 Bibliographie

1 C.Yuan and E.Hansen, « Efficient Computation of Jointree Bounds for Systematic MAP Search. », IJCAI Int. Jt. Conf. Artif. Intell., p. 1989, janv. 2009.

2 D.Nilsson and S.Lauritzen, « Evaluating influencediagrams using LIMIDs », Proc. Sixt. Conf. Uncertain. Artif. Intell., janv. 2013.

3 A.Khaled, E.Hansen, and C.Yuan, « Solving Limited-Memory Influence Diagrams Using Branch-and-Bound Search », Int. Symp. Artif. Intell. Math. ISAIM 2012, sept. 2013.

4 C.Yuan, X.Wu, and E.Hansen, Solving MultistageInfluence Diagrams using Branch-and-Bound Search. 2012, p. 700.

5 R.Dechter and R.Mateescu, « AND/OR searchspaces forgraphical models », Artif. Intell.,vol. 171, p. 73-106, févr. 2007, doi: 10.1016/j.artint.2006.11.003.

6 S.Acid and L.M.deCampos, « An Algorithm for Finding Minimum d-Separating Sets in Belief Networks », in Proceedings of the twelfth Conference of Uncertainty in Artificial Intelligence, 1996, p. 22.

7 D.D.Maua, C.P.deCampos, and M.Zaffalon, « Solving Limited Memory Influence Diagrams », J. Artif. Intell. Res., vol. 44, p. 97-140, mai 2012, doi: 10.1613/jair.3625.

8 D.R.Morrison, S.H.Jacobson, J.J.Sauppe, and E.C.Sewell, « Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning », Discrete Optim., vol. 19, p. 79-102, févr. 2016, doi: 10.1016/j.disopt.2016.01.005.

9 R.A.Howard, J.E.Matheson, and Strategic Decisions Group, Éd.,Readings on the principles and applications of decision analysis, Repr. Menlo Park, Calif: SDG, 1989.

10 A.J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, « An introduction to decision tree modeling », J. Chemom., vol. 18, no 6, p. 275-285, 2004, doi: https://doi.org/10.1002/cem.873.

11 D. Geiger, T. Verma, and J. Pearl, « d-Separation: From Theorems to Algorithms », in Machine Intelligence and Pattern Recognition, vol. 10, M. Henrion, R. D. Shachter, L. N. Kanal, et J. F. Lemmer, Éd. North-Holland, 1990, p. 139-148.

12 David Kahle, Terrance Savitsky, Stephen Schnelle, and Volkan Cevher, « Junction tree algorithm », Stat, Volume 631, 2008.

7 Annexes

7.1 Liens divers

Documentation :

- WEB : `https://davidpinaud.github.io/PANDROIDE_InfDiag_BandB/`
- PDF : `https://github.com/davidPinaud/PANDROIDE_InfDiag_BandB/tree/master/docs/documentation`

Carnet de Bord :

- PDF : `https://github.com/davidPinaud/PANDROIDE_InfDiag_BandB/blob/master/docs/pdf/Carnet%20de%20bord.pdf`

Dépendances :

- Python >3.8.5 :`https://www.python.org/downloads/`
- PyAgrum >0.20.2 : `https://gitlab.com/agrumery/aGrUM/blob/0.20.2/wrappers/pyAgrum/doc/sphinx/index.rst`
- NumPy >v1.20.0 : `https://numpy.org/install/`
- pydotplus >2.0.2 : `https://pypi.org/project/pydotplus/`
- Matplotlib >3.4.1: `https://matplotlib.org/stable/users/installing.html`

7.2 Structre des éléments de l'arbre ET/OU

Les nœuds de décision ont pour attributs :

- Un identifiant au sein de l'arbre ET/OU
- Un dictionnaire de contexte dont la clef est l'identifiant du nœud (dans le diagramme d'influence, ce qui est possible même s'il y a plusieurs nœuds correspondant au même nœud de décision, puisqu'on regarde le contexte dans l'ID et non pas dans l'arbre ET/OU) et dont la valeur est la valeur d'instanciation
- Une valeur d'instanciation du nœud
- Deux identifiants (celui du nœud correspondant dans l'ID pour retrouver à quel nœud dans l'ID celui ci corresponds, et un autre identifiant pour le graphe ET/OU)
- Une liste de parents.
- Un dictionnaire de bornes supérieures dont la clef est la valeur du domaine et dont la valeur est un tuple (moyenne, variance).
- Un dictionnaire d'enfants dont la clef est la valeur du domaine et dont la valeur est l'enfant, un dictionnaire d'évaluations dont la clef est la valeur du domaine et dont la valeur est un tuple (moyenne, variance).

- Un support (domaine du nœud).
- Une décision optimale (une valeur du domaine du nœud de décision).
- La valeur de cette décision (la moyenne du MEU).
- Une liste de nœuds élagués.
- L'inférence qui a été faite sur l'instanciation qui a mené à ce nœud.
- La liste des enfants qui ont été évalués.

Les nœuds de chance ont pour attributs :

- Un identifiant.
- Un support.
- Un dictionnaire d'enfants dont la clef est la valeur du support et dont la valeur est l'enfant.
- Une valeur d'instanciation, une liste de parents, et leurs valeurs.
- Un dictionnaire de probabilités postérieures dont la clef est la valeur du support et dont la valeur est la probabilité.
- Un dictionnaire de contexte dont la clef est l'identifiant du parent, et dont la valeur est le domaine.
- Un identifiant au sein de l'arbre ET/OU.

L'arbre (classe andOrGraph) a pour attributs :

- Le diagramme d'influence que l'on utilise pour l'arbre ET/OU.
- Un nœud chance racine.
- Une liste de nœuds.
- Une liste de nœuds chance.
- Une liste de nœuds de décision.
- Un identifiant au sein de l'arbre ET/OU.

7.3 Documentation version PDF (voir page suivante)

PANDROIDE Branch and bound method for LIMID Inference

Release 1.0.0

David PINAUD, Emilie BIEGAS

May 09, 2021

CONTENTS:

- genindex
- modindex

File containing a class that allows the encapsulation of an influence diagram and its resolution by a branch and bound method

class bandbLIMID.**BranchAndBoundLIMIDInference**(*ID*, *OrdreDecision*, *verbose=False*)
Class that allows the encapsulation of an influence diagram and its resolution by a branch and bound method

ID
The influence diagram to encapsulate

Type pyAgrum.InfluenceDiagram

OrdreDecision

Type list of decision node ids in the order in which the decisions are taken in the influence diagram

SIS(*decisionNodeID*, *ID*)
Function that returns the Sufficient information set for a given decision node and influence diagram

Parameters

- **decisioNodeID** (*int*) – the id of the decision node
- **ID** (*pyAgrum.InfluenceDiagram*) – the ID on which to base the creation of the graph

Returns **SIS** – The sufficient information set of the given decision node in the given influence diagram

Return type set of ints

addCouche(*index*, *root*, *parents_chanceID*, *pile*)
Function that creates a new branch given a root node and the index of the decision node that is the bottom layer to be

Parameters

- **index** (*int*) – index of the decision node that is going to be the bottom layer in the branch
- **root** (chanceNode) – the node that serves as a root of the subtree
- **parents_chanceID** (*list*) – list of the parents of the decision node, used to build the tree before adding the decisions nodes as bottom layer
- **pile** (*list*) – the list of decision nodes still to be processed

Returns the list of nodes that constitutes the new branch

Return type list

branchAndBound()
Function that executes the branch and bound algorithm. It generated the and or graph on the fly and calculates the MEU for every decision nodes/chance nodes for every instanciation possible

checkNameTaken(*bn*, *name*)
Utility function for viewAndOrGraphNoCuts, checks if a name is taken in the BN (we need it because the and or tree has a lot of nodes with the same name)

Parameters

- **bn** (*pyAgrum.BayesNet*) – the bn to check

- **name** (*str*) – the name to check

Returns true if the name is in the bn

Return type bool

createCoucheChance(*parents*, *root*, *contexte*)
Function that builds a new branch by adding recursively layers of chance nodes (only, decision nodes are added with the createCoucheDecision function)

Parameters

- **parents** (*list*) – the set of parents of the decision node for which we are branching (not the one from where we branch but the ones that will be created)
- **root** (chanceNode) – the chance node that is the root of the subtree
- **contexte** (*dict*) – the instanciation path of the root

createCoucheDecision(*root*, *decisionNodeID*, *contexte*, *pile*, *couche*)
Function that builds the layer of decision node at the base of a new branch

Parameters

- **root** (chanceNode) – the root of the subtree
- **decisionNodeID** (*int*) – the id of the decision node (in the ID) for which we must build the layer
- **contexte** (*dict*) – the instanciation path of a decision node
- **pile** (*list*) – the list of decision nodes to expand during the branch and bound, new decisions nodes are added as they are created
- **couche** (*list*) – the branch for which we must create the layer

Returns the branch but now with a layer of decision nodes

Return type list

createRelaxation()
Function that creates the relaxation of the encapsulated influence diagram by adding informations to the ID through the SIS and removing non-required arcs

Returns The relaxed influence diagram

Return type pyAgrum.InfluenceDiagram

createRest(*bn*, *decisionNodeCreated*, *i*)
Function that creates the rest of the BN for viewAndOrGraphNoCuts

Parameters

- **bn** (*pyAgrum.BayesNet*) – the BN
- **decisionNodeCreated** (*list*) – the decision created in a layer
- **i** (*int*) – the index of the current decision node that is being developped

evaluate(*ID*, *evidence*)
Function that makes the inference over an ID given evidence

Parameters

- **ID** (*pyAgrum.InfluenceDiagram*) – The influence diagram to evaluate
- **evidence** (*set of {key:nodeID,value:probability} (cpt)*) – the evidence to set in the inference

Returns The inference object with makeInference() already called

Return type pyAgrum.ShaferShenoyLIMIDInference

findCoucheDeNoeudDeDecision(*decisionNode*, *couches*)
Utility function that allows to find the branch of a certain decision node

Parameters

- **decisionNode** (decisionNode) – the decision node for which we want to find the branch
- **couches** (*list*) – the list of all the branches in the and/or tree

Returns the branch where the decision node is

Return type list or None

findLigneAuDessus(*ligne*)
function that allows to find the layer above the layer "ligne" in a branch

Parameters **ligne** (*list*) – the layer of nodes (chance of decision nodes) for which we wish to find the layer above

Returns the layer of nodes above

Return type list

fromIDToMoralizedAncestral(*decisionNodeID*, *ID*)
Function that, given an influence diagram and a decision node id, creates the corresponding moralized ancestral undirected graph and adds a source and a well node. It is used to generate a graph on which the SIS algorithm can work to return the SIS of the decision node given.

Parameters

- **decisioNodeID** (*int*) – the id of the decision node
- **ID** (*pyAgrum.InfluenceDiagram*) – the ID on which to base the creation of the graph

Returns

- **MoralizedAncestral** (*pyAgrum.UndiGraph*) – the moralized ancestral undirected graph generated
- **alphaXid,BetaYid** (*int*) – the source and well added to the graph

getBNFromID(*idiag: pyAgrum.pyAgrum.InfluenceDiagram*)
Function that gives us the bayesian network for finding posterior probability when doing the backwards inductio, part of the branch and bound

Parameters **idiag** (*pyAgrum.InfluenceDiagram*) – the ID for which we want the bayesian network

Returns the bayesian network generated

Return type pyAgrum.BayesNet()

getDecisionOpt(*decisionNode*)
Function that, given a decisionNode from the And/Or Graph, returns the optimal decision and its MEU value. It is ONLY used internally and is part of the branch and bound algorithm. It is used only on decision nodes that are leaf nodes in the And/Or Graph that are already evaluated. Do NOT use this function to get the optimal decision of a decision Node.

Parameters **decisionNode** (andOrGraph.decisionNode) – the decision node, part of the And/Or Graph

Returns

- **decisionOpt** (*Object*) – The optimum decision
- **valeurDecisionOptimale** (*float*) – the value of the optimal decision

getDomain(*NodeID*)
Function that returns the domain in which this node can instanciate

Parameters **NodeID** (*int*) – the id of the node

Returns Domain of the node

Return type list

getNameFromID(*idNode*)
Function that returns the name of a node in the ID from its id

Parameters **idNode** (*int*) – the id of the node

Returns the name of the node

Return type str

getNamesFromID(*listId*)
Function that returns the name of nodes in the influence diagram given their ids

Parameters **listId** (*list of int*) – list of the ids of the node

Returns list of the names of the nodes

Return type list of str

getParents_chanceID(*decisionNodeID*, *nodeADevID*)
Function that returns the parents of a decision node (that is the leaf the branch we want to create)

Parameters

- **decisionNodeID** (*int*) – the id of the decision node in the and or graph
- **nodeADevID** (*int*) – the id of the decision node in the ID

Returns list of parents of the decision node

Return type list

getSIS(*decisioNodeID*)
Function that returns the Sufficient Information Set of a decision node

Parameters **decisioNodeID** (*int*) – the id of the decision node

Returns the Sufficient Information Set of the decision Node

Return type set of ints

induction(*ligne*)
Function that allows to go up through a layer in a branch given a layer ligne. It calculates the values of the chance nodes when the layer consist of chance nodes and the MEU of decision nodes when it consist decision node

Parameters **ligne** (*list*) – layer in which we want to start the going up process

Returns the layer above the layer ligne

Return type list

inductionArriere(*couche*, *pile*, *couches*, *indexPile*)
Function that is called when the algorithm arrives at decision nodes that are leafs in the and/or graph, it allows to go back up through the branch while computing the values of the chance nodes and the MEU of the decision nodes It also prunes branches that upper bound are smaller than the best evaluation. It is a recursive function that uses the induction function.

Parameters

- **couche** (*list*) – the branch we wish to go up through
- **pile** (*list*) – the list of decisions nodes to expand
- **couches** (*list*) – the list of branches present in the and/or tree
- **indexPile** (*int*) – the next decision node to expand

Returns if the algorithm reaches the root of the and or graph, it returns None and the algoritm has finished, otherwise, it returns an int that is the index of next decision node to expand in the pile

Return type int or None

isAllDecisionNodeProcessed(*couches*)
Function that checks if all the decision nodes are processed

Parameters **couches** (*list of list*) – list of all the branches in the and or graph

Returns true if all the decision nodes are processed, false otherwise

Return type bool

setVerbose(*verbose: bool*) → None
Function that allows to set the verbose parameter, true will print the trace, false will not

Parameters **verbose** (*bool*) – the parameter

viewAndOrGraph()
Creates a BN that allows to visualize the and or graph (without branches that have been cut)

Returns the BN

Return type BayesNet

viewAndOrGraphNoCuts()
Creates a BN that allows to visualize the complete and or graph (with branches that have been cut).

Returns the BN

Return type BayesNet

viewCreateCoucheChance(*bn*, *s*, *parents*, *idNodeIDParent*, *idNodeBNParent*)
creates a layer of chance node for viewAndOrGraphNoCuts

Parameters

- **bn** (*pyAgrum.BayesNet*) – the bn in which to add the nodes
- **s** (*str*) – utility string to modulate the name of the nodes (prevent reuse)
- **parents** (*list*) – list of chance nodes that are parents of the decision node
- **idNodeIDParent** (*int*) – id in the influence diagram of the parent of the chance node that is being developped
- **idNodeBNParent** (*int*) – id in the bayesian network of the parent of the chance node that is being developped

viewCreateDecisionCouche(*bn*, *decisionNode*, *root*)
creates a layer of decision node for viewAndOrGraphNoCuts

Parameters

- **bn** (*pyAgrum.BayesNet*) – the bn in which to add the nodes
- **decisionNode** (*int*) – the id of the decision node to add in the layer
- **root** (*int*) – the id of the root of the layer (a chance node)

Returns list of ids of the decision node in the bn

Return type list

class andOrGraph.**andOrGraph**(*ID*, *root*)
Class that emulates an And/Or Graph (in reality its a tree) .. attribute:: ID

The influence diagram we use for the And/Or Graph

type pyAgrum.InfluenceDiagram

root
the root of the And/Or Graph

Type *andOrGraph.chanceNode*

noeuds
The list of nodes in the graph

Type list of andOrGraph.chanceNode and andOrGraph.decisionNode

noeudsChance
The list of andOrGraph.chanceNode in the graph

Type list of andOrGraph.chanceNode

noeudsDecision
The list of andOrGraph.decisionNode in the graph

Type list of andOrGraph.decisionNode

IDNoeudDecisionAndOr
Integer that serves to give ids to the decision nodes (different to their influence ids)

Type int

addNoeudChance(*noeud*)
Function that adds a andOrGraph.chanceNode to the And/Or Graph

Parameters **noeud** (*andOrGraph.chanceNode*) – the chance node to add

addNoeudDecision(*noeud*)
Function that adds a andOrGraph.decisionNode to the And/Or Graph It also sets its And/Or Graph id

Parameters **noeud** (*andOrGraph.decisionNode*) – the decision node to add

getID()
Getter function for the ID attribute

Returns The influence diagram we use for the And/Or Graph

Return type pyAgrum.InfluenceDiagram

getIDNoeudAndOr()
Getter function for the IDNoeudAndOr attribute

Returns Integer that serves to give ids to the decision nodes (different to their influence ids)

Return type int

getNoeud()
Getter function that returns all the And/Or Graph nodes

Returns The list of nodes in the graph

Return type list of andOrGraph.chanceNode and andOrGraph.decisionNode

getNoeudChance()
Getter function for the noeudsChance attribute

Returns The list of andOrGraph.chanceNode in the graph

Return type list of andOrGraph.chanceNode

getNoeudDecision()
Getter function for the ID attribute

Returns The influence diagram we use for the And/Or Graph

Return type pyAgrum.InfluenceDiagram

getNoeudDecisionAndOrIDs()
Getter function that returns all the And/Or Graph ids

Returns The ids of the nodes in the And/Or Graph

Return type list of ints

getNoeudWithIdAndOr(*id_andOr*)
Getter function for that returns a decision node given its And/Or Graph id (not the Influence Diagram one)

Returns the decision node corresponding to the given id

Return type *andOrGraph.decisionNode*

getRoot()
Getter function for the root attribute

Returns the root of the And/Or Graph

Return type *andOrGraph.chanceNode*

setRoot(*root*)
Setter function for the root attribute

Parameters **root** (andOrGraph.chanceNode) – the root of the And/Or Graph

class andOrGraph.**chanceNode**(*Id*, *support*, *parent*, *valeurParent*, *contexte*, *id_andOr*)
AND node for the And/Or Graph .. attribute:: Id

the id of this node in the corresponding influence Diagram

type int

support
the domain of this node

Type list

valeur
the value of this node calculated during the induction process

Type float

parent
the parent of this node in the andOrGraph

Type *chanceNode* or *decisionNode*

probabilitiesPosteriori
The posterior probabilities calculated during the induction process ; key=a domainValue of this node value: a float

Type dict

contexte
The instanciation context of this node ; key = id of a node in the influence diagram, value = the instanciation value of said node

Type dict

id_andOr
the id of this node in the andOrGraph

Type int

childs
children of this node ; key=domainValue, value=chanceNode or decisionNode

Type dict

getChilds()
childs of the node in the And/Or Graph :returns: key=domainValue, value=chanceNode or decisionNode :rtype: dict

getContexte()
The instanciation context of this node

Returns key = id of a node in the influence diagram, value = the instanciation value of said node

Return type dict

getId_andOr()

Returns the id of the node in the AND/OR Graph

Return type int

getNodeID()

Returns the id of the node in the influence diagram

Return type int

getParent()
parent of the node

Returns the parent of the node

Return type *chanceNode* or *decisionNode*

getProbabilitiesPosteriori()
The posterior probabilities calculated during the induction process

Returns key=a domainValue of this node value: a float

Return type dict

getSupport()

Returns domain of the node

Return type list

getValeur()
the value of the chance node calculated in the induction process

Returns value of the chance node

Return type float

class andOrGraph.**decisionNode**(*Id*, *contexte*, *parent*, *support*, *id_andOr*)
OR node for the And/Or Graph .. attribute:: Id

the id of this node in the corresponding influence Diagram

type int

support
the domain of this node

Type list

decisionOptimale
the optimal decision of this node

Type any

ValeurDecisionOptimale
the value of the optimal decision of this node calculated during the induction process

Type float

parent
the parent of this node in the andOrGraph

Type *chanceNode* or *decisionNode*

contexte
The instanciation context of this node ; key = id of a node in the influence diagram, value = the instanciation value of said node

Type dict

id_andOr
the id of this node in the andOrGraph

Type int

doNotDevelop
list of domain values that should not be developped (its upper bound is smaller than the evaluation of another branch corresponding to a domain value of this node)

Type list

inference
the inference object on which we used to calculated the value of the node (only if this node is a leaf)

Type pyAgrum.ShaferShenoyLIMIDInference

enfants
children of this node ; key=domainValue, value=chanceNode or decisionNode

Type dict

evaluation
evaluation of this node ; key=domainValue, value=(mean,variance)

Type dict

borneSup
upper bounds for the branches of this node (one for each domain value and only if this node is a leaf), key=domainValue, value=(mean,variance) ;

Type dict

getBorneSup()
Getter function for the upper valuation of the node (only is not a leaf)

Returns key=domainValue, value=(mean,variance)

Return type dict

getContexte()
The instanciation context of this node

Returns key = id of a node in the influence diagram, value = the instanciation value of said node

Return type dict

getDecisionOptimale()
Returns the optimal decision for this node

Returns the optimal decision of this node, a value of its domain

Return type any

getEnfants()

Returns key=domainValue, value=chanceNode or decisionNode

Return type dict

getEvaluation()
Getter function for the valuation of the node

Returns key=domainValue, value=(mean,variance)

Return type dict

getId_andOr()

Returns the id of the node in the AND/OR Graph

Return type int

getInference()
returns the Shafer Shenoy Object used to make the inference on this node (only if it is a leaf node)

Returns the inference object

Return type pyAgrum.ShaferShenoyLIMIDInference

getNodeID()

Returns the id of the node in the influence diagram

Return type int

getParent()
parent of the node

Returns the parent of the node

Return type *chanceNode*

getSupport()

Returns domain of the node

Return type list

getValeurDecisionOptimale()
return the value of the optimal decision

Returns the value

Return type float

PYTHON MODULE INDEX

INDEX

I

M

N

O

P

R

S

V