Sorbonne Université
Master ANDROIDE

Implémentation des “dynamic Bayesian Networks" dans pyAgrum

UE de projet M1

Doruk OZGENC - Haya MAMLOUK

2025

Table des matières

I. Introduction

Les **réseaux bayésiens dynamiques (dBN)** sont une extension des réseaux bayésiens classiques. Ils modélisent non seulement les dépendances conditionnelles entre variables aléatoires à un instant donné, mais aussi l'évolution de ces variables au fil du temps. Les dBN sont essentiels pour analyser des systèmes séquentiels où le facteur temps joue un rôle clé, comme la reconnaissance vocale, la bioinformatique, le diagnostic médical ou la prévision de séries temporelles.

Il existe plusieurs outils pour travailler avec les réseaux bayésiens, dont la librairie **pyAgrum**, qui propose un ensemble de fonctionnalités pour la modélisation, l'inférence et la visualisation de réseaux bayésiens. Cependant, son support des réseaux bayésiens dynamiques reste limité, notamment parce qu'elle ne permet que des transitions temporelles sur un seul pas de temps. Cette contrainte limite son utilisation pour des systèmes dynamiques plus complexes nécessitant la prise en compte de dépendances sur plusieurs périodes.

Pour remédier à cette limite, nous avons conçu et implémenté un module qui étend pyAgrum afin de faciliter la modélisation de réseaux bayésiens dynamiques multi-pas de temps (**k-Timestep Bayesian Networks**, ou **kTBN**). Ce module permet de représenter des séries temporelles multidimensionnelles satisfaisant une propriété de Markov d'ordre $p \geq 1$. Il intègre également un système d'inférence dynamique, qui ajuste les prédictions au fil des nouvelles observations, ainsi que des outils de visualisation tirant parti des fonctionnalités de pyAgrum.

Le projet a été réalisé sous la supervision de **Pierre-Henri WUILLEMIN**. Le code source est disponible sur GitHub : https://github.com/HayaMamlouk/dynamic-Bayesian-Networks

II. État de l'art

A. Réseaux Bayésiens (BN)

Un **réseau bayésien (BN)** est un modèle graphique probabiliste qui représente un ensemble de variables aléatoires et leurs dépendances conditionnelles à l'aide d'un **graphe orienté acyclique (DAG)**. Il offre une représentation compacte de la **distribution de probabilité conjointe (JPD)** du système, permettant un raisonnement efficace sous incertitude.

- **Factorisation de la distribution conjointe**

Pour un ensemble de variables $X = X_1, X_2, \ldots, X_n$ structuré par un DAG, la distribution de probabilité conjointe se factorise comme suit :

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i \mid Pa(X_i))$$

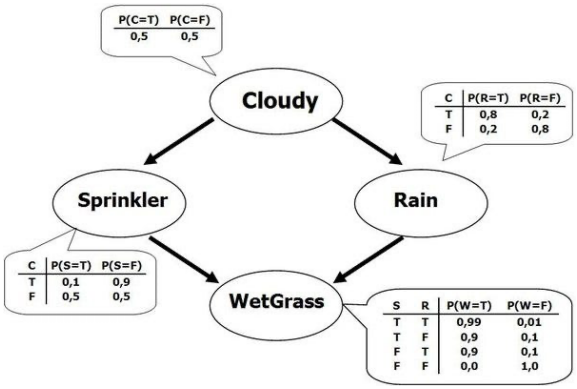
où $Pa(X_i)$ désigne l'ensemble des parents de X_i dans le graphe.

- **Propriété de Markov locale**

Un réseau bayésien satisfait la propriété de Markov locale :

$$X_v \perp\!\!\!\perp X_{V \setminus de(v)} \mid X_{Pa(v)}, \quad \forall v \in V$$

Chaque variable est conditionnellement indépendante de ses non-descendants, compte tenu de ses parents. Cette propriété rend les réseaux bayésiens particulièrement efficaces pour l'inférence probabiliste



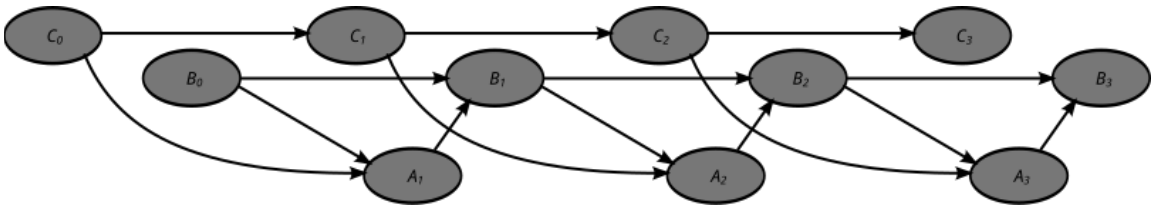
Document 1 : Représentation simple d'un réseau bayésien

- **Implémentation dans pyAgrum**

La librairie **pyAgrum** est un outil Python open-source dédié aux modèles graphiques probabilistes. Elle repose sur la bibliothèque C++ aGrUM et fournit une interface de haut niveau pour la modélisation, l'apprentissage, l'inférence et la visualisation des réseaux.

B. Réseaux Bayésiens Dynamiques (dBN)

Un **réseau bayésien dynamique (dBN)** est une extension des BN qui modélise l'évolution des variables au fil du temps. Contrairement à un BN classique, qui capture un instantané statique des dépendances, un dBN représente comment ces dépendances changent **à travers plusieurs pas de temps.**



Document 2 : Représentation simple d'un réseau bayésien dynamique

- **Composants essentiels d'un dBN**

1. **Variables aléatoires temporelles** : à chaque pas de temps t, l'ensemble $\mathbf{X}_t = X_t^1, X_t^2, \ldots, X_t^n$ décrit l'état du système

2. **Dépendances intra-tranche** : entre variables du même instant t.

3. **Dépendances inter-tranches** : entre variables de tranches temporelles différentes (par exemple, entre t et $t-1$, ou plus généralement entre t et $t-k$.

4. **Distributions conditionnelles (CPT)** : associées à chaque variable, conditionnées par ses parents (dans la même tranche et/ou des tranches précédentes).

- **Structure temporelle et factorisation**

Sous l'hypothèse de Markov d'ordre 1 : $P(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \ldots, \mathbf{X}_0) = P(\mathbf{X}_t \mid \mathbf{X}_{t-1})$, la distribution conjointe sur toutes les tranches $t = 0$ à T se factorise ainsi :

$$P(\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_T) = P(\mathbf{X}_0) \prod_{t=1}^{T} P(\mathbf{X}_t \mid \mathbf{X}_{t-1}) .$$

Pour des dBN de plus haut ordre (Markov d'ordre k), la dépendance s'étend jusqu'à X_{t-k}:

$$P(\mathbf{X}_t \mid \mathbf{X}_{t-k}, \ldots, \mathbf{X}_{t-1})$$

Ce cas général permet de modéliser des dépendances temporelles longues, fréquentes dans des applications réelles.

- **Implémentation dans pyAgrum**

La librairie pyAgrum propose des outils de modélisation et d'inférence pour les dBN via le module `pyAgrum.lib.dynamicBN`, permettant notamment la création de **2-Time-slice Bayesian Networks (2TBN)**, l'affichage des tranches temporelles, le suivi de l'évolution des variables et l'inférence sur des séquences temporelles. Toutefois, avant notre projet, le support se limitait aux transitions mono-pas de temps (2TBN) et n'intégrait pas les modèles d'ordre supérieur (**k > 2**). Les distributions conditionnelles étaient représentées par des objets **Tensor** (anciennement appelés *Potential*).

- **Visualisation et inférence**

Pour la visualisation des structures et des CPT (Conditional Probability Tables), ainsi que pour l'exécution d'inférences, pyAgrum propose également le module `pyAgrum.lib.notebook` (visualisation graphique des Timeslice, CPT et résultats d'inférence).

III. Contribution

Le travail réalisé a consisté à concevoir et implémenter une extension de la librairie pyAgrum afin de permettre la modélisation et l'inférence sur des réseaux bayésiens dynamiques (dBN) à plusieurs pas de temps (**k > 2**), fonctionnalité absente de la version originale de la bibliothèque.

À cette fin, deux modules ont été développés, `DynamicBaseNet.py` et `notebook.py`. Le module `DynamicBaseNet.py` intègre plusieurs classes structurées selon une approche orientée objet en Python, notamment les classes **DynamicBayesNet** et **dTensor.**

Les contributions spécifiques sont les suivantes :

(1) **Extension de la modélisation** des dBN à des dépendances temporelles de plus haut ordre (k > 2).
(2) Développement d'un s**ystème d'inférence dynamique** compatible avec ces modèles étendus.
(3) Refonte de l'**ergonomie utilisateur**:
   - Introduction d'une **représentation normalisée des variables** sous forme de couples (nom de variable, indice temporel).
   - **Uniformisation de cette représentation** dans l'ensemble de la modélisation, de l'inférence et de la visualisation.
   - **Simplification de la création et de la manipulation des objets** (variables, arcs, CPT), remédiant à la lourdeur et à l'incohérence de la version native de pyAgrum.

Cette contribution améliore considérablement la convivialité (**user-friendliness**) et la cohérence de l'interface, simplifiant la modélisation et l'analyse de réseaux dynamiques complexes. Les résultats expérimentaux obtenus sur des exemples synthétiques démontrent que notre approche permet de modéliser efficacement des dépendances temporelles complexes tout en réalisant des inférences précises. Par ailleurs, nos choix d'implémentation garantissent une compatibilité complète avec les outils standards de **pyAgrum**, tout en offrant une interface plus intuitive et accessible pour les utilisateurs travaillant sur des processus temporels.

IV. Travail effectué

A. Implémentation de la modélisation des dBNs

Pour la modélisation des dBN, nous avons conçu la classe **DynamicBayesNet**, destinée à représenter et manipuler des réseaux bayésiens dynamiques (kTBNs) avec une dimension temporelle. Cette classe encapsule un objet gum.BayesNet() de pyAgrum, qui sert de structure sous-jacente. Elle gère un ensemble de variables atemporelles ainsi que l'horizon temporel k. Une partie essentielle de notre contribution a porté sur la gestion des dépendances temporelles multi-tranches et sur la conception d'une interface utilisateur intuitive.

Un changement fondamental concerne la représentation des variables temporelles. Dans BayesNet de pyAgrum, chaque variable est simplement identifiée par une chaîne de caractères, et la représentation temporelle n'est disponible que pour deux tranches spécifiques : l'instant initial 0 et l'instant suivant t, ce qui limite son usage aux réseaux bayésiens dynamiques à deux tranches temporelles (**2TBN**). Pour introduire cette notion temporelle généralisée (k>2), nous avons créé le concept de **séparateur**, qui permet de distinguer automatiquement, dans un nom de variable, sa partie « nom » et son « indice temporel ». Par défaut, nous utilisons le symbole **"#"** (par exemple, "A#1"). Cependant, l'utilisateur peut choisir un autre symbole si nécessaire, notamment pour éviter les conflits avec des noms de variables contenant déjà un "#".

Grâce à cette abstraction, l'utilisateur peut définir les variables de manière naturelle sous forme de couples **(nom, time slice)**, par exemple ("A", 1), sans jamais manipuler directement la syntaxe interne de BayesNet. Lorsque l'utilisateur entre une variable comme ("A", 1), notre module la **traduit automatiquement** en "A#1" pour l'intégrer à BayesNet. Inversement, lorsqu'une variable doit être présentée à l'utilisateur, elle est convertie du format interne "A#1" vers le couple ("A", 1). Ce mécanisme de traduction est assuré par les fonctions `_userToCodeName` et `_codeToUserName`.

Cette infrastructure permet de manipuler facilement l'ajout et la suppression de variables, l'ajout et la suppression d'arcs, l'accès aux tables de probabilité conditionnelle (CPT), et la génération automatique des CPT.

L'ensemble de ces opérations est uniformisé grâce à cette nouvelle représentation des variables, ce qui simplifie considérablement la modélisation et garantit une interface cohérente, conviviale et pleinement compatible avec les outils standards de pyAgrum

```
dBN = DynamicBayesNet(6)

a = gum.LabelizedVariable("A","A",2)
b = gum.LabelizedVariable("B","B",2)

dBN.add(a)
dBN.add(b)

dBN.addArc(("A", 1), ("A", 2))
dBN.addArc(("A", 1), ("B", 2))
dBN.addArc(("A", 2), ("B", 2))
```

Document 3 : Exemple simple de la création d'un kTBN

La classe **`dTensor`**, définie dans le module `DynamicBayesNet.py` encapsule un **Tensor** (anciennement *Potential*) de pyAgrum, et l'adapte aux spécificités des dBNs. Elle permet notamment de gérer les CPT sur plusieurs tranches temporelles, en assurant la compatibilité avec le nommage des variables temporelles et les transformations liées au séparateur. Elle joue un rôle crucial pour garantir que les CPT restent accessibles, cohérents et exploitables aussi bien dans le contexte du modèle que dans l'affichage.

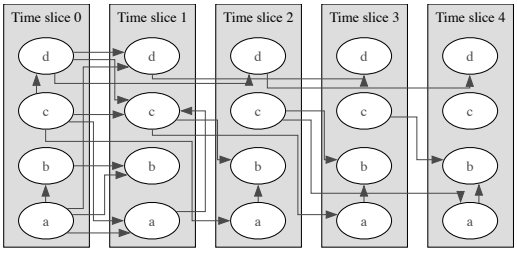
B. Visualisation

Pour la visualisation des kTBNs, nous avons développé un ensemble de fonctions regroupées dans le module **`notebook.py`**, en grande partie inspirées de celles de pyAgrum, mais adaptées à notre format user-friendly. Ces fonctions permettent d'afficher les réseaux dynamiques en tranches temporelles, ainsi que les tables de probabilité conditionnelle (**CPT**), tout en garantissant que les entrées et les sorties soient cohérentes avec notre représentation utilisateur basée sur des couples *(nom, time slice)*.

Une composante essentielle de cette visualisation est la fonction **`unrollKTBN`**, qui permet de "dérouler" un kTBN sur un horizon temporel donné, c'est-à-dire de construire explicitement un réseau bayésien équivalent en dupliquant les variables pour chaque time slice et **en ajoutant les dépendances temporelles entre elles**. Cette étape est particulièrement utile pour visualiser la structure globale du réseau ou pour effectuer certaines inférences classiques sur une structure dépliée.

	a,2					
c,0	0	1	2	3	4	5
0	0.1445	0.2187	0.0938	0.2080	0.2333	0.1017
1	0.3178	0.1178	0.2098	0.0238	0.0070	0.3238
2	0.1625	0.0317	0.2427	0.1785	0.1928	0.1918
3	0.2345	0.1839	0.1023	0.2683	0.1496	0.0615
4	0.0555	0.0433	0.0798	0.4224	0.3826	0.0164
5	0.0101	0.1367	0.3000	0.0134	0.3101	0.2298

Document 4 : Visualisation d'une CPT



Document 5 : Visualisation d'un unrolled kTBN

La fonction unrollKTBN s'appuie sur un objet clé de notre architecture : la classe `dTensor`, qui joue un rôle central notamment lors du remplissage des tables de probabilité conditionnelle (CPT). Elle permet de gérer correctement les dépendances temporelles en assurant la cohérence des dimensions et des noms de variables sur plusieurs tranches temporelles.

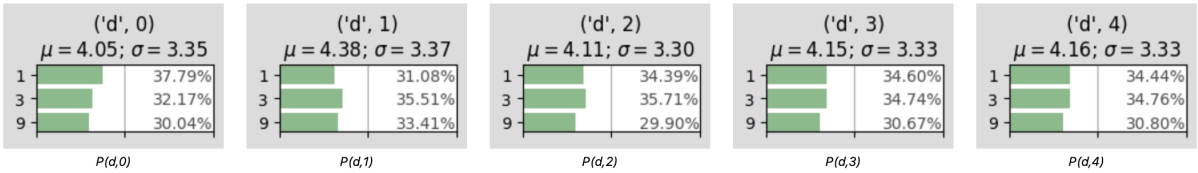
À noter que toutes les fonctions de visualisation sont conçues pour être utilisées dans un environnement **Jupyter Notebook**, conformément aux conventions de pyAgrum. Pour permettre également l'affichage des CPT en dehors de cet environnement, notamment dans un terminal, nous avons implémenté une méthode `__str__` dédiée, qui génère une représentation textuelle des CPT dans un format clair et user-friendly, respectant la même logique de nommage des variables basée sur les couples *(nom, time slice)*.

C. Inférence probabiliste dans les dBNs

Pour l'inférence, nous avons adopté une approche classique consistant à effectuer les calculs sur des dBNs **déroulés** (unrolled). Les fonctions d'inférence ont été adaptées afin de garantir la cohérence avec notre format **user-friendly**, à la fois en entrée et en sortie. La majorité des fonctions repose sur un wrapping des outils d'inférence existants de pyAgrum, que nous avons adaptés pour prendre en charge notre concept de séparateur et notre représentation des variables sous forme de couples *(nom, time slice)*.

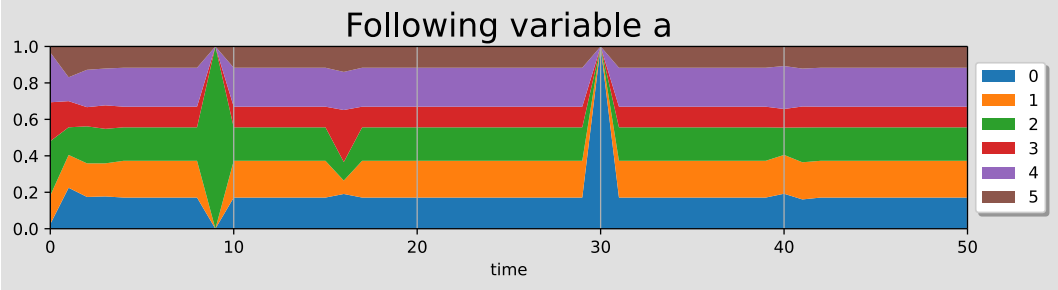
Deux fonctions principales ont été développées dans ce cadre :

- **`getPosterior(bn, evs, target)`**, qui calcule la distribution postérieure d'une variable cible sous forme d'un histogramme, en s'appuyant sur un objet Tensor.



Document 6 : La distribution postérieure d'une variable d'un unrolled dBN

- **`plotFollow(lovars, kTBN, T, evs)`**, qui permet de visualiser l'évolution d'un ensemble de variables au cours du temps en tenant compte d'un horizon T et d'un ensemble d'évidences.



Document 7 : Evolution d'une variable au cours du temps

Ces fonctions assurent une interaction fluide avec les dBNs unrolled, tout en préservant la simplicité d'usage et la lisibilité attendues par l'utilisateur final.

D. Tests et étude de cas

1. Tests

Afin de garantir la **robustesse** et la **validité** de notre implémentation des réseaux bayésiens dynamiques, nous avons mis en place une suite de **tests unitaires** utilisant le module Python unittest.

Ces tests couvrent l'ensemble des fonctionnalités essentielles de notre modèle :

- **Ajout de variables** : nous vérifions que les variables sont correctement ajoutées à toutes les tranches temporelles définies par l'horizon k, et qu'elles apparaissent bien dans le graphe dynamique construit.
- **Ajout et suppression d'arcs** : nous testons la création d'arcs entre variables à travers les tranches temporelles, ainsi que leur suppression correcte du graphe.
- **Suppression de variables** : les tests s'assurent qu'une variable supprimée disparaît bien de toutes les tranches, et que les dépendances associées sont également supprimées.
- **Création de CPTs** et accès : nous testons la possibilité de remplir et interroger des tables de probabilités conditionnelles (CPTs), en vérifiant la cohérence des résultats.
- **Propagation des CPTs lors de l'unrolling** : nous vérifions que les valeurs des CPTs sont correctement copiées et réutilisées dans les tranches temporelles étendues lorsque le modèle est déplié (unroll).
- **Compatibilité avec des séparateurs personnalisés** : notre implémentation permet d'utiliser un séparateur autre que "#" pour les noms des variables temporelles (par exemple "|"). Des tests spécifiques assurent que toutes les opérations (ajout, CPT, unroll) restent valides avec ces séparateurs alternatifs.
- **Structures plus complexes** : un test construit un réseau avec plusieurs variables interconnectées sur plusieurs tranches, afin de vérifier la stabilité du modèle même dans des cas d'usage avancés.

En résumé, nous avons poussé notre code à ses limites à travers ces tests. Cela nous a permis de détecter certaines erreurs : par exemple, l'utilisation de séparateurs personnalisés ne fonctionnait pas correctement dans toutes les fonctions. Grâce à cette phase de test, nous avons pu réviser notre logique de parsing, et corriger le code pour que toutes les fonctions soient compatibles ainsi que s'assurer qu'aucun ajout ou changement affecte le code.

2. Etude de cas : modélisation d'un scénario météorologique

Afin d'évaluer la praticité et l'utilisabilité de notre bibliothèque dans un contexte d'application réaliste, nous avons conçu un scénario fictif mais crédible, centré sur la modélisation d'un système météorologique évoluant dans le temps. Cette expérimentation, réalisée dans le notebook `experiments.ipynb`, avait pour principal objectif de valider l'usage effectif de notre modèle dans un cas concret, au-delà des simples tests unitaires.

Nous avons simulé un environnement météorologique avec quatre variables symboliques : `SunnyWithClouds`, `RainingCatsAndDogs`, `StormOfTheCentury` et SnowingSoMuchThereMayBeAnAvalanche.

Ces variables sont intégrées dans un dBN de profondeur k=3, avec de nombreuses dépendances temporelles allant jusqu'à deux tranches d'écart. L'objectif était de mettre à l'épreuve la **flexibilité de la construction du réseau**, l'utilisation de **noms complexes**, ainsi que l'implémentation des **arcs inter-temporels** et des méthodes rapides comme addFast.

Le modèle ainsi construit a été enrichi par la génération aléatoire des CPTs via generateCPTs(), avant d'être déroulé sur un horizon de 5 tranches. Nous avons ensuite mené différentes inférences : des calculs de **postérieurs temporels** pour évaluer les probabilités à chaque instant, et des **visualisations dynamiques** de l'évolution de certaines variables en fonction d'observations partiellement injectées dans la séquence temporelle.

Ce scénario a également joué un rôle **clé dans la validation des fonctions d'affichage**, qui ne peuvent pas être testées efficacement à l'aide de simples tests unitaires. En mettant en œuvre un cas d'usage réel, nous avons pu observer le comportement visuel du système : lisibilité des graphes, intégrité des noms de variables, cohérence temporelle, etc.

V. Conclusion

Ce travail a permis d'étendre significativement les capacités de la librairie pyAgrum en introduisant la prise en charge des réseaux bayésiens dynamiques à **k tranches temporelles** (k-TBNs), avec **k>2**. Avant ce projet, pyAgrum ne permettait de modéliser que des réseaux dynamiques à deux tranches (2-TBNs), ce qui limitait fortement les possibilités d'analyse temporelle. Notre contribution rend désormais possible la modélisation, l'inférence et la visualisation de modèles dynamiques plus riches et réalistes, s'étendant sur plusieurs pas de temps.

Nous avons développé un ensemble cohérent de modules facilitant la construction de k-TBNs, leur manipulation et leur visualisation. Ces outils conservent la compatibilité avec l'écosystème pyAgrum tout en introduisant une approche plus intuitive et flexible pour représenter les dépendances temporelles. L'expérience utilisateur a également été au cœur de notre démarche, avec une interface claire, un affichage graphique soigné et des fonctions accessibles pour analyser les distributions postérieures dans un contexte temporel étendu.

Le mécanisme de séparateur, souvent mentionné dans notre documentation, était un levier technique au service de cet objectif. Il a permis d'encoder proprement la dimension temporelle dans les noms de variables, afin de rendre possible cette **généralisation à k tranches**.

Un temps important a été consacré à comprendre le fonctionnement interne de pyAgrum. Plutôt que de repartir de zéro, nous avons dû bâtir notre solution sur la base des structures et des fonctions existantes de pyAgrum. Cela a nécessité une connaissance approfondie de l'architecture de la bibliothèque, ce qui a rendu notre travail plus itératif et progressif, avec de fréquents ajustements pour assurer une intégration cohérente avec les fonctionnalités existantes.

Une perspective d'évolution naturelle serait l'implémentation de **dBNs non stationnaires**, permettant à l'utilisateur de spécifier différentes familles de réseaux pour différents intervalles temporels. Cette extension soulève des questions intéressantes, notamment sur la manière de gérer des cycles alternés (par exemple, phases de traitement et de repos dans un protocole médical, ou cycles économiques expansion/récession) ou des transitions conditionnelles entre familles. Ce type de modélisation nécessitera une implémentation soigneusement pensée, tenant compte de multiples cas d'usage.

Bibliographie

- **Dynamic Network Models for Forecasting**
  Paul Dagum, Adam Galper, Eric Horvitz
  Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence (UAI), 1992
  Link

- **Dynamic Bayesian Networks: A State Of The Art**
  V. Mihajlovic, M. Petkovic
  University of Twente, 2001
  Link

- **Dynamic Bayesian Networks in PyAgrum**
  PyAgrum Documentation
  Link

- **Dynamic Bayesian Networks: A Tutorial**
  PyAgrum Notebooks
  Link

- **PyAgrum 2.1.0 Documentation**
  Link

- **PyAgrum Tutorial**
  Link

- **aGrUM/PyAgrum Official Website**
  Link

- **Bayesian Networks with pyAgrum - Artificial Intelligence and Machine Learning**
  Johannes Maucher
  Link

- **Dynamic Bayesian Network Overview**
  ScienceDirect Topics
  Link

Annexe A : Cahier des charges

1. Objectif

L'objectif de ce projet est d'étendre la bibliothèque pyAgrum afin de prendre en charge les **réseaux bayésiens dynamiques** (dBNs) à **plusieurs tranches temporelles** (e.g., 2, 3 ou 4 pas de temps). Cette extension comprendra :

- L'implémentation de nouvelles fonctions pour gérer les dépendances multi-temporelles.
- L'amélioration des **capacités de visualisation** et **d'inférence**.
- La fourniture de **tests unitaires** et de **documentation** claire et fonctionnelle.

2. Échéances Clés

- **Date limite de rendu :** 9 mai 2025
- **Soutenance du projet :** 15 mai 2025

3. Découpage du Projet

3.1 Modélisation de la Classe DynamicBayesNet

Attributs

- kTBN : Instance de pyAgrum.BayesNet représentant le réseau bayésien dynamique sous-jacent.
- variables : Ensemble des noms de variables atemporelles présentes dans le réseau.
- k : Nombre de tranches temporelles (horizon temporel).
- separator : Caractère utilisé pour séparer le nom de la variable et l'indice temporel (par défaut : #).
- name_display_mode : Mode d'affichage des noms de variables (par défaut : "compact").

Méthodes

Méthode	But	Exemple
add(self, v)	Ajoute une variable v (instance de pyAgrum.Variable) au réseau, en la répliquant sur toutes les tranches temporelles.	(1) a=gum.LabelizedVariable("A", "A", 2) (2) dbn.add(a) - Ajoute A#0, A#1, ..., A#(k-1)
addFast(self, var_description)	Ajoute rapidement une variable en utilisant la syntaxe rapide de pyAgrum.	dbn.addFast("B[0,1]") - Ajoute B#0, B#1, ..., B#(k-1)
addArc(self, tail, head)	Ajoute un arc dirigé du nœud tail vers le nœud head. Chaque nœud est représenté par un tuple (nom_variable, indice_temps).	dbn.addArc(("A", 0), ("B", 1)) Ajoute un arc de A#0 vers B#1

eraseArc(self, tail, head)	Supprime l'arc dirigé du nœud tail vers le nœud head.	dbn.eraseArc(("A", 0), ("B", 1)) - Supprime l'arc de A#0 vers B#1
erase(self, var)	Supprime la variable nommée var de toutes les tranches temporelles, ainsi que tous les arcs associés.	dbn.erase("A") - Supprime A#0, A#1, ..., A#(k-1) et les arcs associés
cpt(self, var)	Retourne la CPT (objet dTensor) de la variable spécifiée par le tuple (nom_variable, indice_temps)	cpt_A1 = dbn.cpt(("A", 1)) - Récupère la CPT de A#1
generateCPTs(self)	Génère aléatoirement les CPTs pour la structure actuelle du réseau.	dbn.generateCPTs()

3.2 Classe dTensor

Le principal objectif de cette classe est de rendre l'objet Potential de pyAgrum compatible avec le dBN en exposant une interface basée sur des tuples pour accéder et modifier les CPTs.

Attributs

- potential : L'objet pyAgrum.Potential sous-jacent représentant la table de probabilité conditionnelle (CPT).
- separator
- name_display_mode

Méthodes

Méthode	But	Exemple
__getitem__(self, key)	Permet l'indexation conviviale des CPTs en utilisant des tuples pour accéder aux valeurs des états des variables	dBN.cpt((‘A’, 1) - Accéde à la CPT de la variable (“A”,1)
__setitem__(self, key, value)	Permet de modifier les valeurs dans le CPT en utilisant des clés conviviales sous forme de dictionnaires avec des tuples.	dBN.cpt(('A', 1))[{‘B’, 0) : 1}] = valeur
fillWith(self, value)	Remplir le CPT avec une valeur constante ou une liste de valeurs. Si une liste est fournie, elle doit correspondre à une version aplatie du CPT.	dBN.cpt(('A', 2)).fillWith([0, 1])

3.3 Visualisation et affichage

La visualisation est un aspect essentiel du projet, l’objectif est de fournir à l’utilisateur des outils de représentation clairs, cohérents et centrés sur une interface lisible, sans exposer les noms internes des variables générés automatiquement par la bibliothèque.

Objectifs

- Représenter visuellement la structure d'un kTBN, en distinguant explicitement les dépendances temporelles entre tranches.
- Afficher les tables de probabilité conditionnelle (CPT) de manière lisible, en masquant les noms internes bruts des variables au profit d'un format compréhensible pour l'utilisateur.
- Assurer une intégration fluide avec les environnements interactifs comme Jupyter Notebook, tout en prévoyant une alternative utilisable dans un terminal.

Fonctionnalités prévues

Déroulement temporel		
unrollKTBN(dbn, nbr)	Construction d'un réseau bayésien statique équivalent à un kTBN pour un horizon donné, avec dupliquation des variables et des arcs temporels, permettant une visualisation globale ou une inférence classique.	unrolled_dbn = unrollKTBN(dBN, T)
Affichage des graphes		
showKTBN(dbn)	Visualisation du kTBN via une conversion en format DOT, en conservant les noms des variables au format utilisateur.	showKTBN(dBN) - Affichage dans un Jupyter Notebook
showUnrolled(bn)	Visualisation du réseau résultant du déroulement, avec une présentation temporellement cohérente.	showUnrolled(unrolled_dbn) - Affichage dans un Jupyter Notebook
Affichage des CPTs		
showCPT(dbn,var)	Présentation des tables de probabilité conditionnelle avec des noms user-friendly selon la convention (nom, time slice).	showCPT(dBN, ('A', 1) - Affichage dans un Jupyter Notebook
__str__ de dTensor	Génération automatique d'une représentation textuelle claire et structurée des CPTs, lisible dans un environnement non graphique.	print(dBN.cpt(('B', 2))) - Affichage string

Contraintes de nommage et de lisibilité

Tous les outils de visualisation doivent respecter rigoureusement le format utilisateur pour les noms de variables : aucun nom brut (nom interne utilisé en interne par pyAgrum) ne doit apparaître**.**

Le nom des variables doit être généré dynamiquement à partir de l'attribut `display_name_mode`, qui propose trois styles d'affichage :

- **compact** : " A, 1" — concis et pratique.
- **reverse** : " 1, A" — utile pour les noms longs.

- **classic** : " ('A', 1)" — conforme à la représentation des variables .

Le système de visualisation doit être compatible avec la séparation temporelle (separator) définie dans le kTBN, et capable de s'adapter à tout horizon temporel défini dynamiquement.

3.4 Inférence

L'inférence dans notre réseau bayésien dynamique multi-temporel (kTBN) repose sur le déroulement explicite du réseau (unrolling) sur un nombre fini de tranches temporelles. Ce choix permet d'appliquer les algorithmes d'inférence classiques fournis par pyAgrum, tout en maintenant une interface cohérente avec la représentation utilisateur des variables.

Fonctionnalités prévues

- **Inférence ponctuelle (getPosterior)** : reçoit un réseau déroulé, une cible et des observations exprimées dans le format utilisateur (nom, time_slice).

  Exemple : getPosterior(dbn, target=('d', 1),evs={}),"P(d,1)")

- **Suivi temporel (plotFollow)** : Permet de tracer l'évolution des distributions d'une ou plusieurs variables cibles au fil du temps, à partir d'un kTBN, d'un horizon temporel T et d'un ensemble d'évidences.

  Exemple : plotFollow(["a","b", "c", "d"], dBN, T=51, evs={('a',9):2, ('a',30):0, ('c',14):0, ('b',40):0, ('c',50):1})

3.5 dBNs non stationnaires (si le temps le permet)

Étend les réseaux bayésiens dynamiques (dBNs) pour prendre en charge des structures variables dans le temps, en associant à chaque dBN d'une famille une période d'utilisation spécifique. Cela signifie qu'au lieu d'avoir une structure fixe, le modèle peut adapter ses dépendances et ses paramètres en fonction des différentes phases d'utilisation. Au fil du temps, différents dBNs issus de la famille sont appliqués selon la période définie, permettant ainsi au réseau de mieux capturer l'évolution des dynamiques du système.

3.6 Tests unitaires (Obligatoire)

Objectif : Garantir la validité fonctionnelle via des tests et une documentation.

Développer des tests unitaires pour toutes les fonctions implémentées.

---

4. Calendrier du projet (12 semaines)

Phase	Durée
Cahier des charges	2 semaines
Modélisation	3 semaines
Visualisation	3 semaines
Inférence	2 semaines

Tests unitaires	1 semaine
Modèle non stationnaire	Temps restant

Annexe B : Documentation

La documentation complète des modules développés dans le cadre de ce projet est disponible en ligne. Cliquez pour la consulter.

Annexe C : Manuel utilisateur

Un notebook interactif intitulé tutorial.ipynb est disponible dans le dépôt GitHub. Il guide l'utilisateur pas à pas dans la prise en main du module, à travers des exemples concrets d'utilisation sur des modèles dynamiques.

En complément, le fichier README.md du dépôt fournit les informations nécessaires à l'installation, à l'utilisation du module, ainsi qu'une vue d'ensemble de ses fonctionnalités.