RAPPORT

Comparaisons de méthodes de résolutions de Processus de Décision Markoviens sur des instances issues de langage de modélisation

Master ANDROIDE
Sorbonne Université

Réalisé par

Zeyu TAO
Jiahua LI

Dans le cadre de l'UE
PANDROIDE

Travail encadré par

Emmanuel Hyon
Pierre-Henri Wuillemin

2025

Contents

1 Introduction

1.1 Objectif du projet

L'objectif principal de ce projet est de comparer l'efficacité des méthodes de résolution des Processus de Décision Markovien (MDP) implémentées dans la librairie Marmote .

1.2 Composition de l'équipe

L'équipe de développement est constituée des étudiants suivants:

- Zeyu TAO
- Jiahua Li

Dans le cadre de ce projet, initialement, tous les membres de l'équipe collaboreront pour identifier et étudier les différents langages de description. Ensuite, chaque membre prendra en charge le développement et les tests du parseur pour un langage de description spécifique.

2 Description de la demande

2.1 Les objectifs

L'objectif principal de ce projet est de comparer l'efficacité des méthodes de résolution des Processus de Décision Markovien (MDP) implémentées dans la librairie Marmote et MDP Toolbox.

Et plus concrètement, les objectifs suivants sont demandés:

- **Étude des langages de description des Processus de Décision Markoviens (MDP)**
  - Rechercher et étudier divers langages de description tels que PDDL, ses extensions PPDDL, RDDL, et JANI.
  - Déterminer si ces langages sont toujours en usage. Identifier les langages adaptés spécifiquement aux chaînes de Markov et aux processus de décision markoviens.
  - Évaluer l'activité autour de ces langages, y compris la disponibilité de bibliothèques de gestion, de benchmarks, et de modèles existants.
- **Sélection et analyse détaillée de 1 à 2 langages de description**
  - Sélectionner un ou deux langages de description
  - Analyser la sémantique de ces langages sélectionnés pour comprendre leur structuration et fonctionnement.
- **Développement d'un parseur**
  - Concevoir et implémenter un parseur pour convertir les instances décrites dans les langages sélectionnés en un format compatible avec les solveurs de la librairie Marmote ou MDP Toolbox.
- **Tests et comparaisons**
  - Collecter et analyser des instances de MDP disponibles sur Internet.
  - Exécuter des benchmarks pour évaluer l'efficacité des méthodes de résolution implémentées.
  - *Si le temps et les ressources le permettent*, intégrer les langages sélectionnés dans la librairie Marmote.

2.2 Délivrable

Le projet final proposé sera

- Code source : Un ensemble de scripts Python, soigneusement commentés, implémentant l'intégralité du processus.

- Rapport détaillé : Un document explicatif décrivant la méthode employée, le langage utilisé, les choix réalisés, les résultats obtenus, ainsi qu'une analyse approfondie des performances des différentes méthodes.
- Soutenance : Une présentation synthétique mettant en avant les objectifs de l'étude, les méthodes mises en œuvre et les résultats obtenus, accompagnée d'une analyse critique.

2.3 Contraintes de coûts

En raison des exigences académiques de la Sorbonne Université, ce projet est un projet universitaire qui ne dispose pas de ressources financières allouées. Il est entièrement géré par les étudiants et les enseignants impliqués dans le cadre de l'unité d'enseignement.

Pour ce projet, nous disposons d'une équipe de développement composée de Zeyu TAO, Jiahua LI. L'équipe est dirigée par deux enseignants encadrants: Emmanuel Hyon et Pierre-Henri Wuillemin. Pour les ressources matérielles, nous disposons d'une connexion internet, des ordinateurs personnels pour chaque membre de l'équipe. Nous avons aussi accès au GitHub pour les dépôts en ligne et pour la gestion de nos versions et la collaboration sur le projet.

2.4 Contraintes de délais

Un rendu du code source doit être livré le 9 Mai 2025 et il y aura une soutenance le 15-16 Mai 2025.

2.5 Planning

- **Février à mi-mars** : Finalisation du choix des langages de description à utiliser. Cette phase comprend également la collecte et l'étude approfondie de la documentation pertinente (bibliographie) concernant les solveurs et les langages de description.
- **Mi-mars** : Décision définitive sur le choix du solveur.
- **Mi-mars à mi-avril** : Développement du parseur pour les langages sélectionnés. Cette tâche est cruciale et nécessite une attention particulière pour assurer la compatibilité avec les solveurs.
- **Fin avril** : Achèvement du développement du parseur. À ce stade, les tests préliminaires (et l'intégration initiale avec la librairie Marmote si possible) doivent être en cours ou achevés.
- **Début mai** : Phase finale de tests et de peaufinage du code en préparation pour la livraison finale. Cette période inclut également la préparation de la présentation pour la soutenance.

3 Concepts de base

3.1 Chaîne de Markov (MC)

Une chaîne de Markov est un outil mathématique permettant de modéliser un processus stochastique dans lequel l'état futur ne dépend que de l'état courant (chaîne de Markov d'ordre 1). Cette propriété, appelée propriété de Markov est particulièrement utile, car elle permet de représenter des dépendances entre les variables aléatoire à l'aide d'une matrice de transition de taille NxN, où N est la cardinalité de l'espace d'états. Cette représentation compacte offre un faible coût de calcul et de mémoire.

3.1.1 Définition: Propriété de Markov

Soit $X = \{x_1, x_2, \cdots, x_T\}$ une séquence de variables aléatoires prenant des valeurs dans un ensemble fini d'états $Q = \{q_1, q_2, \cdots, q_N\}$.

Une chaîne de Markov (MC) est caractérisée par deux composantes principales:

- Une matrice de transition A: $a_{q_i,q_j} = P(x_{t+1} = q_j \mid x_t = q_i)$
- Un vecteur de probabilités initiales Π: $\pi_{q_i} = P(x_1 = q_i)$

La séquence X est une chaîne de Markov si elle vérifie la propriété suivante:

$$P(x_{t+1} = q_j \mid x_1, x_2, \cdots, x_t = q_i) = P(x_{t+1} = q_j \mid x_t = q_i)$$

Cela signifie que l'état futur x_{t+1} ne dépend conditionnellement que de l'état courant x_t et non des états précédents $x_1, x_2, \cdots, x_{t-1}$.

En particulier, en utilisant la propriété de Markov, la probabilité jointe de la séquence X peut être factorisée de manière suivante:

$$\begin{aligned} P(X) &= P(x_1, x_2, \cdots, x_T) \\ &= P(x_1)P(x_2 \mid x_1) \cdots P(x_T \mid x_1, x_2, \cdots, x_{T-1}) \\ &= P(x_1)P(x_2 \mid x_1) \cdots P(x_T \mid x_{T-1}) \\ &= \pi_{x_1} \prod_{t=2}^{T} a_{x_{t-1}, x_t} \end{aligned}$$

3.1.2 Propriété de stationnarité

Soit μ une distribution de probabilité sur l'ensemble fini d'états Q, c'est-à-dire, $\forall q \in Q, 0 \leq \mu_q \leq 1$ et $\sum_{q \in Q} \mu_q = 1$. Elle est dite stationnaire, ou invariante, si:

$$\mu = \mu A$$

Cela signifie que la probabilité des états (μ) ne change plus au cours du temps.

3.2 Processus de décision markovien (MDP)

Un processus de décision markovien (MDP) est un modèle stochastique utilisé pour modéliser des problèmes de décision séquentielle dans un environnement incertain (stochastique) où un agent prend des décision en fonction de l'état courant du système et reçoit des récompenses en fonction des actions qu'il choisit et les états dans lesquels il se trouve. Les MDP sont alors des chaînes de Markov (MC) dont les transitions d'un état vers un autre sont contrôlées par l'action choisie et l'état courant, et sont évaluées par une valeur de récompense.

3.2.1 Définition: MDP

Un MDP est défini par un quadruplet (S, A, p, r), où:

- Un ensemble fini d'états S, qui représente tous les états possibles du système.
- Un ensemble fini d'actions A, qui représente toutes les actions possibles de l'agent dans le système.

  Nous notons également, un sous ensemble $A_s \subseteq A$, toutes les actions disponibles pour l'agent dans l'état s.
- Une fonction de transition p, qui représente des probabilités de transition d'un état vers un autre:

  $$p(s_{t+1} \mid s_1, a_1, s_2, a_2, \cdots, s_t, a_t) = p(s_{t+1} \mid s_t, a_t)$$

  (propriété de Markov).
- Une fonction de récompenses r, qui associé à tout paire d'état et d'action un valeur réelle: $r(s_t, a_t) \in \mathbb{R}$.

Résoudre un MDP revient à trouver une politique optimale π^*, qui maximise les récompenses totales cumulées au cours du temps, selon un critère de performance donné (moyen, actualisé ou total). La politique optimale π^* est une fonction qui guide l'agent de manière optimale dans le choix de ses actions en fonction de l'état courant. Elle peut être déterministe ou stochastique:

- Une politique déterministe $\pi^*(s) : S \longmapsto A$, qui associe à chaque état $s \in S$ une action unique $a \in A$.
- Une politique stochastique $\pi^*(s, a) : S\text{x}A \longmapsto [0, 1]$, qui associe à chaque état $s \in S$ une distribution de probabilité sur les actions.

3.2.2 Critères de performance

- **Critère γ pondéré**

  Le critère γ pondéré maximise la somme actualisée des récompenses sur un horizon infini, où chaque récompense est pondérée par un facteur d'actualisation $\gamma \in [0, 1)$. De manière intuitive, ce critère donne un poids

plus important à la récompense immédiate et de moins en moins aux récompenses futures. Il est couramment utilisé en apprentissage par renforcement. La fonction de valeur est donnée par la formule suivante:

$$\forall s \in S, \forall \gamma \in [0,1) \qquad V^\pi(s) = \mathbb{E}^\pi[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s] = \mathbb{E}^\pi[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s]$$

- **Critère moyen**

  Le critère moyen maximise la somme moyenne des récompenses sur un horizon infini:

  $$\lim_{N\to\infty} \frac{1}{N} \mathbb{E}[\sum_{t=0}^{N-1} r_t \mid s_0 = s]$$

  , où $\frac{1}{N}$ est un facteur de normalisation, qui permet d'éviter la divergence de la somme. De manière informelle, nous pouvons voir cela comme un critère pondéré par γ avec un facteur de normalisation, où $\gamma = 1$, c'est-à-dire que toutes les récompenses, qu'elles soient immédiates ou futures, ont la même importance pour l'agent.

- **Critère total**

  Le critère total maximise la somme totale des récompenses sur un horizon fini ou infini. La fonction de valeur est donnée par la formule suivante:

  $$\forall s \in S \qquad V^\pi(s) = \mathbb{E}[\sum_{t=0}^{T} r_t | s_0 = s]$$

  , où T peut être fini ou infini.

3.2.3 Algorithmes de résolution

Il existe plusieurs approches pour résoudre des modèles de processus de décision markovien (MDP). Par exemple, les méthodes basées sur la programmation dynamique, telles que value iteration et policy iteration, les algorithmes d'apprentissage par renforcement, comme Q-learning, SARSA, actor-critic et l'approche Dyna, ainsi que la programmation linéaire dans des cas particuliers. Dans le cadre de ce projet, nous nous concentrerons principalement sur les méthodes de programmation dynamique, car nous supposons que l'environnement est parfaitement connu, c'est-à-dire, les récompenses r et les probabilités de transition p sont connues.

- **Itération sur les valeurs**

  L'algorithme d'itération sur les valeurs est basé sur la propriété de stationnarité de la fonction de valeur. Par exemple, sous le critère γ pondéré,

nous cherchons une fonction de valeur V^*, telle que:

$$V^*(s) = \max_{a \in A}\{r(s,a) + \gamma \sum_{s' \in S} p(s' \mid s,a)V^*(s')\}, \forall s \in S$$

Le principe de l'algorithme d'itération sur les valeurs est de partir d'une fonction de valeur V_0, initialisée par défaut, et d'itérer jusqu'à la convergence vers une fonction de valeur V^*.

Voici les différentes approches possibles:

- **Algorithme d'itération sur les valeurs**
- **Algorithme d'itération sur les valeurs - Gauss Seidel**
- **Algorithme d'itération sur les valeurs relatives**

- **Itération sur les politiques**

  L'algorithme d'itération sur les politiques est également basé sur la propriété de stationnarité de la fonction de valeur.

  Il est constitué de deux étapes principales, prenant toujours l'exemple avec le critère γ pondéré. Dans un premier temps, nous cherchons une fonction de valeur V_n, telle que:

$$V_n(s) = r(s, \pi_n(s)) + \gamma \sum_{s' \in S} p(s' \mid s,a)V_n(s'), \forall s \in S$$

  , où π_n est la politique optimale à l'instant n. Pour cette étape, soit nous pouvons résoudre un système d'équation linéaire, ce qui donne une solution exacte, soit utiliser une version itérative, ce qui donne une solution approchée.

  Dans un second temps, nous calculons une nouvelle politique optimale π_{n+1} en fonction de la fonction de valeur calculée. L'algorithme répète ces deux étapes jusqu'à la convergence, c'est-à-dire $\pi_n = \pi_{n+1}$.

  Voici les différentes approches possibles:

  - **Algorithme d'itération sur les politiques**
  - **Algorithme modifié d'itération sur les valeurs**

- **Algorithme classique de la programmation dynamique**

  Pour des modèles MDP avec le critère total à horizon fini T, il suffit, à partir de la dernière étape, de calculer les fonctions de valeur optimales $V_1^*, V_2^*, \cdots, V_T^*$ en utilisant la formule:

$$V_{n+1}^*(s) = \max_{a \in A}\{r(s,a) + \sum_{s'} p(s' \mid s,a)V_n^*(s')\}, \forall s \in S$$

De-même, à chaque étape de l'algorithme, nous calculons également la politique optimale $\pi_1^*, \pi_2^*, \cdots, \pi_T^*$ en utilisant la formule:

$$\pi_{T-1-n}(s) = \text{argmax}_{a \in A}\{r(s,a) + \sum_{s'} p(s' \mid s,a) V_n^*(s')\}, \forall s \in S$$

Le résultat final est une liste de politiques optimales $[\pi_1^*, \pi_2^*, \cdots, \pi_T^*]$, où chaque politique π_{T-1-t}^* spécifie, pour chaque instant t, l'action à effectuer en fonction de l'état courant s.

Nous remarquons que la politique, dans le cadre du critère total à horizon fini, dépend également du temps, autrement dit, elle n'est pas stationnaire.

3.3 Langages de description

Les langages de description sont des formalismes définissant une syntaxe et une sémantique précises, destinés à représenter de manière structurée les systèmes dynamiques, qu'ils soient stochastiques ou déterministes. Dans le cadre des chaînes de Markov (MC) et des processus de décision markoviens (MDP), ces langages permettent de spécifier explicitement les états, transitions, probabilités, actions possibles ainsi que les récompenses associées, facilitant ainsi leur analyse mathématique, leur simulation ou encore leur vérification formelle. L'objectif principal de ces langages est de fournir un cadre normalisé, interprétable directement par des outils logiciels spécialisés (solveurs, vérificateurs, simulateurs), afin d'étudier les propriétés des systèmes modélisés.

3.3.1 Description de nos recherches

Les encadrants du projet nous ont initialement proposé deux langages de description : **JANI** et **PPDDL**. Toutefois, une recherche complémentaire a été réalisée afin d'identifier d'autres langages potentiellement plus intéressants ou mieux adaptés à notre étude des chaînes de Markov (MC) et des processus de décision markoviens (MDP), ainsi qu'aux outils logiciels associés.

Cette recherche documentaire a principalement été menée à l'aide des mots-clés suivants :

- solver Markov decision processes
- benchmark Markov decision process
- modeler Markov decision process
- modeling Markov decision process
- description language for stochastic dynamic problems
- comparison PDDL and other description languages

Les ressources documentaires ont été consultées à l'aide des outils suivants :

- **Google**, pour identifier les ressources générales, documentations techniques et outils logiciels existants.
- **Google Scholar**, afin d'accéder à des articles scientifiques récents et pertinents concernant les langages de description et les méthodes associées aux MDP.
- **ChatGPT**, dans l'objectif d'obtenir rapidement des résumés introductifs, des clarifications techniques, ainsi que des orientations pour des recherches plus approfondies.

Cette démarche méthodologique nous a permis d'obtenir une vue complète et actualisée du domaine étudié, facilitant ainsi le choix des langages et des outils à utiliser pour la réalisation effective de ce projet.

3.3.2 Recensement des langages

- **PDDL** (Planning Domain Definition Language) : Utilisé principalement pour la planification automatisée dans des environnements déterministes. Il permet de décrire des domaines de planification en spécifiant les actions, les préconditions et les effets.
- **PPDDL** (Probabilistic PDDL) : Une extension de PDDL qui supporte les transitions d'état probabilistes, permettant ainsi de modéliser des environnements incertains dans les problèmes de planification.
- **RDDL** (Relational Dynamic Influence Diagram Language) : Conçu pour modéliser des décisions dynamiques dans des environnements complexes, en intégrant des aspects relationnels et probabilistes. Il est notamment utilisé dans les compétitions internationales de planification probabiliste (IPPC).
- **JANI** : Un langage basé sur JSON pour l'interaction avec des modèles stochastiques et décisionnels, supportant divers types de formalismes de modélisation. Il facilite l'interopérabilité entre différents outils de vérification et d'analyse.
- **PRISM** : Langage utilisé par le model checker PRISM pour spécifier des modèles probabilistes tels que les chaînes de Markov à temps discret (DTMC), les chaînes de Markov à temps continu (CTMC) et les processus de décision markoviens (MDP). Il est basé sur le formalisme des modules réactifs et permet l'analyse de propriétés exprimées en logiques temporelles probabilistes.
- **MODEST** : Langage de modélisation comportementale pour les systèmes stochastiques temporisés, permettant la représentation conjointe d'aspects probabilistes et temps réel, ainsi que de décisions non déterministes. Il est soutenu par le Modest Toolset, qui offre des outils pour la modélisation et l'analyse de tels systèmes.

- **GSPN** (Generalized Stochastic Petri Net) : Extension des réseaux de Petri stochastiques, intégrant des transitions immédiates et temporisées, utilisée pour l'analyse de la performance et de la fiabilité des systèmes dynamiques à événements discrets.
- **PGCL** (Probabilistic Guarded Command Language) : Langage de programmation formel combinant des choix non déterministes et probabilistes, adapté à la spécification et à la vérification de programmes probabilistes. Il est utilisé pour raisonner sur les propriétés quantitatives des programmes, notamment dans le contexte de la vérification formelle.

3.3.3 Choix de JANI et RDDL

Après une analyse comparative préliminaire, nous avons décidé d'utiliser JANI et RDDL comme langages principaux pour la suite de notre étude.

D'une part, JANI présente une excellente compatibilité, de nombreux langages tels que PRISM ou PPDDL pouvant être convertis au format JANI via des outils dédiés(par exemple Storm pour PRISM). Ceci permet à JANI de disposer d'un large nombre de ressources disponibles pour les tests et les benchmarks.

D'autre part, RDDL, issu de l'évolution des langages PDDL et PPDDL, et utilisé comme langage principal dans les *International Probabilistic Planning Competitions* (IPPC) depuis 2011, offre des fonctionnalités plus étendues. En outre, la communauté logicielle associée à RDDL est particulièrement riche et bien développée, ce qui constitue un avantage significatif pour le développement ultérieur de notre parseur.

3.3.4 JANI : Références et Ressources

- **Spécification du format JANI** :
  `https://jani-spec.org/`
  Ce site fournit la spécification officielle du format JANI.
- **Syntaxe de JANI** :
  `https://docs.google.com/document/d/1BDQIzPBtscxJFFlDUEPIo8ivKHgXT8_X6hz5quq7jK0/edit?tab=t.0`
  Ce document fournit une description détaillée de la syntaxe du langage JANI, notamment la signification des mots-clés, la structure des modèles, les types de données. Il constitue une référence essentielle pour comprendre comment rédiger ou interpréter un fichier JANI.

3.3.5 RDDL : Références et Ressources

- **Guide du langage RDDL**:

  `https://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf`

  Ce document décrit le langage RDDL, utilisé pour spécifier des problèmes de planification probabiliste. RDDL permet de modéliser des processus

décisionnels markoviens (MDP) et des MDP partiellement observables (POMDP) avec des variables paramétrées.

- **RDDLSim – Simulateur RDDL en Java**:

  `https://github.com/ssanner/rddlsim`

  RDDLSim est une implémentation en Java d'un parseur et simulateur pour RDDL, offrant une architecture client/serveur pour l'évaluation des modèles. Il a été utilisé dans plusieurs compétitions internationales de planification probabiliste.

- **pyRDDLGym – Environnement Python pour RDDL**:

  `https://github.com/pyrddlgym-project/pyRDDLGym`

  pyRDDLGym est une boîte à outils Python permettant de simuler et d'interagir avec des environnements définis en RDDL. Elle facilite l'intégration avec des algorithmes d'apprentissage par renforcement et offre des interfaces compatibles avec OpenAI Gym environnements.

- **Compétitions Internationales de Planification Probabiliste (IPPC)**:
  - IPPC 2011 : `https://users.cecs.anu.edu.au/~ssanner/IPPC_2011/`
  - IPPC 2014 : `https://ssanner.github.io/IPPC_2014/`
  - IPPC 2015 : `http://users.cecs.anu.edu.au/~ssanner/IPPC_2014/`
  - IPPC 2018 : `https://ipc2018-probabilistic.bitbucket.io/`
  - IPPC 2023 : `https://ataitler.github.io/IPPC2023/`

  Ces compétitions évaluent les systèmes de planification probabiliste sur divers benchmarks. RDDL a été le langage principal utilisé pour la spécification des domaines dans ces compétitions, offrant ainsi une référence pour les chercheurs et développeurs dans le domaine.

3.3.6 Analyse comparative approfondie : JANI vs RDDL

Après une étude plus approfondie, nous avons constaté que le langage **JANI** présente des avantages notables en matière de lisibilité et de traitement. Basé sur le format JSON, il propose une structure de données claire, interprétable et extensible, ce qui facilite notre développement de parseurs.

À l'inverse, le langage **RDDL** nécessite la définition manuelle d'une grammaire formelle et la mise en œuvre d'un analyseur syntaxique dédié, représentant ainsi une charge de développement importante. Bien qu'il existe des outils comme *RDDLSim*, un parseur et simulateur RDDL en Java, leur intégration dans notre projet pose plusieurs problèmes. Notre objectif étant de tester des méthodes dans une bibliothèque Python, l'usage de composants Java introduit une complexité technique non négligeable (interopérabilité, dépendances, configuration). De plus, cette solution imposerait aux utilisateurs de notre projet

la gestion simultanée d'environnements Java et Python, ainsi que l'installation de nombreuses bibliothèques, ce qui nuirait à la reproductibilité et à la facilité de déploiement.

Par ailleurs, nous avions initialement supposé qu'il serait possible de convertir un fichier RDDL en PPDDL, puis d'utiliser un convertisseur existant pour obtenir une version au format JANI. Cette hypothèse avait motivé notre choix d'approfondir le format JANI, dans l'idée qu'il pourrait élargir notre ensemble de benchmarks.

Cependant, une analyse plus poussée a révélé que cette conversion générale de RDDL vers PPDDL n'est pas réalisable. RDDL est en effet un langage beaucoup plus expressif que PPDDL, notamment grâce à sa capacité à modéliser des fonctions de transition paramétriques complexes. En pratique, seule une version instanciée — appelée *ground RDDL* — peut être traduite en PPDDL, ce qui limite fortement l'intérêt de cette chaîne de conversion dans une approche générique.

Compte tenu de l'ensemble de ces considérations, nous avons choisi de concentrer nos efforts sur le langage **JANI**.

3.4 Recensement des repository pour le benchmark de MDP ou de MC

Comme le précisent les auteurs de JANI : *« We do not expect users to create jani-model files manually. Instead, they will be automatically generated from higher-level and domain-specific languages. »* Autrement dit, la majorité des fichiers JANI ne sont pas écrits manuellement, mais générés automatiquement à partir de langages plus expressifs ou spécifiques à un domaine.

Dans cette optique, plusieurs plateformes ont été créées pour collecter et partager des modèles JANI issus de diverses sources. Voici quelques référentiels notables où l'on peut trouver de nombreux fichiers au format JANI:

- **Bibliothèque de modèles JANI sur GitHub** :
  `https://github.com/ahartmanns/jani-models`
  Cette bibliothèque propose une collection de modèles formels au format JANI, illustrant diverses structures et propriétés. Bien que cette collection ne soit plus activement maintenue, elle reste une ressource utile pour comprendre l'application du format JANI.
- **Quantitative Verification Benchmark Set** :
  `https://qcomp.org/benchmarks/index.html`
  Ce site est une collection extensive de modèles probabilistes, incluant des chaînes de Markov, des arbres de défaillance et des réseaux de Petri stochastiques, tous disponibles au format JANI. Ces benchmarks sont utilisés pour évaluer et comparer les outils de vérification quantitative.
- **Bibliothèque de benchmarks IMITATOR** :

`https://www.imitator.fr/library.html`
IMITATOR propose une bibliothèque de benchmarks comprenant 65 cas avec 137 modèles différents et 235 propriétés. Certains de ces modèles sont disponibles au format JANI, offrant ainsi des exemples supplémentaires pour l'étude et la vérification de systèmes temporisés paramétriques.

Une grande partie des modèles que nous avons recensés pour les benchmarks sont en réalité des variantes du problème classique du *stochastic shortest path*. En effet, les questions traitées dans ces contextes consistent principalement à atteindre un état cible — où toutes les conditions sont satisfaites — ou à ordonnancer une série d'actions de manière optimale.

Cela constitue une limitation importante : ces modèles couvrent essentiellement des scénarios de planification avec un objectif de destination (planning terminal), ce qui ne reflète pas l'ensemble des dynamiques possibles des MDP. Il est donc nécessaire de garder à l'esprit cette restriction lors de l'interprétation des résultats expérimentaux issus de ces benchmarks.

3.5 Solveur MDP

3.5.1 Marmote

Marmote est une bibliothèque développée par Inria pour la modélisation et la résolution de chaînes de Markov et de processus de décision markoviens. Sa composante **MarmoteMDP**, dédiée aux MDP, a été développée par Emmanuel Hyon.

3.5.2 MDP Tool Box

MDP Toolbox est une bibliothèque multi-plateforme permettant de résoudre des MDP en temps discret à l'aide d'algorithmes tels que value iteration, policy iteration et Q-learning.

4 Contribution

4.1 Analyse de RDDL

4.1.1 Brève description de la syntaxe

Un fichier RDDL se compose de trois sections (ou blocs) principales, telles que **domain**, **non-fluents** et **instance**.

La section **domain** définit l'ensemble des variables représentant un état du problème, les actions possibles, les probabilités conditionnelles de transition entre les états - interprétables comme un réseau bayésien dynamique (DBN), c'est-à-dire, $P(s_{t+1} \mid s_t) = \prod_{x \in V} P(x_t \mid \mathrm{pa}(x_t))$, où s_{t+1} et s_t représentent respectivement les états aux instants $t+1$ et t, V l'ensemble des variables, et $\mathrm{pa}(x)$ sous-ensemble des variables non-indépendantes de x_t - la fonction de

récompense immédiate, ainsi que les contraintes appliquées aux états et aux actions.

La section **non-fluents** définit les valeurs des constantes (variables non-fluents) ainsi que les instances des objets définis de manière externe (i.e. user-defined).

La section **instance** définit l'état initial du problème à résoudre, la valeur de l'horizon (i.e. il décrit un MDP à horizon fini) et le facteur d'actualisation.

Voici la structure minimale d'une instance RDDL:

```
domain <domain name> {
    requirements { <requirement>, ... };
    types { <object name>: object, ... };
    pvariables {
        <pvar name> (<object>, ...): {
            <type fluent>, <type value>, default = <value>
        };
        ...
    };
    cpfs {
        <<type fluent> name>' (?<param>, ...) = <expr>;
        ...
    };
    reward = <expr>;
}

non-fluents <non-fluents name> {
    domain = <domain name>;
    objects {
        <object name>: { <object inst>, ... };
        ...
    };
    non-fluents {
        <non-fluents name> (<object inst>, ...) = <value>;
        ...
    };
}

instance <instance name> {
    domain = <domain name>;
    non-fluents = <non-fluents name>;
    init-state { <state-fluent name> = <value>, ... };
    max-nondef-actions = <int>;
    horizon = <int>;
    discount = <real>;
}
```

4.2 Analyse de Jani

4.2.1 Brève description de la syntaxe

Dans un fichier JANI, la section `edges` de l'automate décrit les comportements de transitions possibles entre les états du système. Chaque transition (ou *edge*) est définie par une condition booléenne (`guard`), un identifiant d'action optionnel (`action`), et une ou plusieurs destinations (`destinations`), chacune associée à une probabilité de transition ainsi qu'à une règle d'affectation de variables (`assignments`).

Lorsque l'on considère les états comme étant uniquement définis par les valeurs des variables (en ignorant les `locations` qui peuvent être aussi considérées comme une variable implicite), le modèle se réduit à un système de transitions piloté par les variables.

Le processus de lecture d'un fichier JANI peut ainsi être interprété comme une exploration de l'espace d'états, à partir des valeurs initiales des variables. Concrètement, on commence par ajouter l'état initial à une file d'attente `open` et on utilise un ensemble `closed` pour enregistrer les états déjà visités. L'algorithme procède ensuite par parcours en largeur : pour chaque état extrait de `open`, on parcourt tous les `edges` et on vérifie si la condition `guard` est satisfaite. Si c'est le cas, on applique les affectations de chaque `destination` pour générer de nouveaux états (nouvelles combinaisons de variables), que l'on ajoute à `open` s'ils n'ont pas encore été explorés. En parallèle, on enregistre la probabilité de chaque transition dans une structure de graphe.

Ce processus se poursuit jusqu'à ce que la file `open` soit vide, signifiant que tous les états atteignables ont été explorés. Le résultat final est un graphe orienté, dont les nœuds correspondent à toutes les combinaisons de valeurs de variables atteignables, et les arcs représentent les transitions probabilistes entre ces états. Ce graphe constitue la base pour créer les objets de Marmote.

4.2.2 Extension du format JANI : JaniR

Dans le format JANI standard, le champ `reward` n'est pas obligatoire. En effet, les fichiers que nous utilisons sont principalement liés à des problèmes de plus court chemin, dans lesquels la matrice de récompense dépend directement de l'objectif défini dans la section `properties`. Par conséquent, les fichiers JANI existants ne contiennent que des relations de transitions et des actions, sans spécification explicite des récompenses.

Le format **JaniR** a été conçu comme une extension du format JANI afin de permettre la représentation complète d'un objet MDP tel que défini dans la bibliothèque Marmote. Les objets MDP de Marmote sont plus génériques et nécessitent une définition explicite des récompenses pour chaque transition.

Dans ce contexte, JaniR impose les contraintes et ajouts suivants :

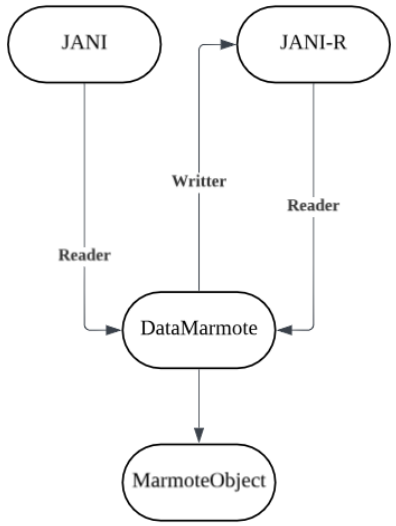

Figure 1: Architecture de conversion vers MarmoteObject

- **Récompenses obligatoires** : chaque transition (`edge`) doit contenir un champ `reward`. Si ce champ est absent, une valeur par défaut de `reward = 0` est appliquée.
- **Ajout de mots-clés** :
  - `criterion` : indique le sens de l'optimisation.
    - `min` : le `reward` est interprété comme un coût.
    - `max` : le `reward` est interprété comme un gain.
  - `type` : précise la nature du MDP modélisé, avec les valeurs possibles :
    - `DiscountMDP`
    - `AverageMDP`
    - `TotalMDP`
    - `FiniteHorizonMDP`

Ainsi, cette extension répond aux exigences structurelles de Marmote pour la modélisation et le stockage. Un fichier JaniR permet de recréer exactement le même objet MDP que celui représenté en mémoire dans Marmote.

4.3 Implémentation

4.3.1 Développement du parseur

Nous avons développé un parseur dédié pour extraire les informations nécessaires à partir de fichiers au format `JANI` ou `JaniR` en vue de générer des objets Marmote. Ce parseur repose sur un module central, appelé `reader`, dont la principale fonctionnalité est la suivante :

- **Entrée** : un fichier au format JANI ou JaniR ;
- **Sortie** : un dictionnaire Python structuré contenant toutes les informations indispensables à la création d'un objet Marmote (états, transitions, récompenses, type de modèle, etc.).

4.3.2 Intégration avec Marmote

Pour automatiser la construction des objets Marmote, nous avons conçu la classe `DataMarmote`, qui prend en entrée la sortie du `reader`.

- **Fonction principale** : la méthode `createMarmoteObject()` analyse le type de modèle spécifié (DTMC, DiscountedMDP, etc.) et instancie automatiquement le bon objet Marmote.
- **Fonction complémentaire** : la méthode `saveAsJaniRFile()` permet de sauvegarder l'objet sous forme de fichier JaniR, conforme à notre extension du format JANI.

Choix de conception : Nous avons délibérément choisi de ne pas sérialiser directement les objets Marmote. En effet :

- L'API Marmote ne fournit pas d'interfaces d'accès (*getter*) permettant de lire facilement les données internes d'un objet ;
- La sérialisation directe aurait introduit une complexité importante, en particulier pour les matrices et structures internes.

Ainsi, la classe `DataMarmote` joue le rôle d'une structure intermédiaire, contenant les données nécessaires à la reconstruction d'un objet Marmote.

4.3.3 Implémentation avec le dictionnaire

Le traitement des matrices de transition a été conçu de manière à faciliter l'exportation et la réutilisation des données.

Dans les fichiers JANI que nous manipulons, les matrices de transition sont généralement très creuses : le nombre moyen de transitions non nulles par état est faible (souvent entre 1 et 2). Or, la classe `SparseMatrix` de Marmote, bien qu'adaptée à la modélisation efficace des matrices creuses, ne fournit pas de méthode directe permettant d'accéder à ses éléments non nuls. Cela complique les opérations de conversion vers le format JaniR, qui nécessitent une itération complète sur les n^2 cellules.

Pour contourner cette difficulté, nous avons adopté une représentation temporaire des transitions sous forme de dictionnaire imbriqué dans la classe `DataMarmote`, selon la structure suivante :

```
{
    état_source_1: {
        état_cible_1: {"probability": p, "reward": r},
        ...
    },
    ...
}
```

Cette représentation permet de :

- Construire efficacement un objet `SparseMatrix` à partir des seules transitions non nulles ;
- Générer un fichier JaniR sans parcourir toute la matrice, avec une complexité en $O(m)$ où m est le nombre de transitions effectives.

5 Benchmark et résultats

5.1 Description du benchmark

Nos fichiers de test sont répartis en deux catégories distinctes :

- **Fichiers avec constantes intégrées** : tous les paramètres (par exemple `max_iteration`) sont définis en dur dans le fichier.
- **Fichiers paramétrables** : certaines constantes doivent être fournies dynamiquement, généralement liées à la taille de l'espace d'états.

Cette seconde catégorie nous permet de générer, à partir d'un même modèle structurel, une famille d'instances MC ou MDP de taille variable. Grâce à cela, nous avons pu observer que la structure et la taille des matrices influencent fortement le temps de calcul des méthodes de Marmote.

Méthodologie de test Pour chaque classe de modèle (MC et MDP), nous avons sélectionné un fichier représentatif, puis généré plusieurs instances en faisant varier un paramètre clé lié à la taille (comme le nombre d'états n).

- **Chaînes de Markov (MC)** :
  Nous avons comparé les performances de deux méthodes de calcul de distribution stationnaire :
  - `StationaryDistribution()`
  - `StationaryDistributionRLGL()`

  Pour chaque valeur de n, nous avons mesuré le temps d'exécution de chaque méthode.

- **Processus de décision markoviens (MDP)** :
  Pour chaque type de MDP, nous avons évalué plusieurs algorithmes de résolution :
  - `ValueIteration`
  - `ValueIterationGS`
  - `PolicyIterationModified`
  - `PolicyIterationModifiedGS`

  Remarques spécifiques :
  - Pour les `FiniteHorizonMDP`, seul `ValueIteration` est applicable.
  - Pour les `AverageMDP`, nous utilisons `RelativeValueIteration` à la place de `ValueIterationGS`.

  Chaque fonction est exécutée 30 fois pour chaque instance afin d'obtenir une moyenne représentative.

Enregistrement des résultats:

Tous les résultats sont stockés dans des fichiers CSV contenant les informations suivantes:

- type de modèle (MC ou MDP)
- paramètres utilisés
- nombre d'états n
- nom de la méthode
- temps d'exécution (moyenne sur 30 exécutions)

Ces données facilitent l'analyse systématique et la génération de graphiques comparatifs par la suite.

5.2 Résultats pour les chaînes de Markov

Nous avons comparé deux méthodes de calcul de la distribution stationnaire sur un même modèle MC en faisant varier le nombre d'états n:
`StationaryDistribution()` (abrégée SD) et `StationaryDistributionRLGL()` (abrégée RLGL).

Cependant, il est important de noter que les résultats produits par ces deux méthodes ne sont pas directement comparables. En effet, notre chaîne de Markov contient plusieurs états absorbants, ce qui la rend non ergodique. Dans ce cas, la distribution stationnaire dépend de la distribution initiale choisie.

La méthode `StationaryDistribution()` utilise par défaut une distribution initiale concentrée sur l'état 0 (c'est-à-dire une probabilité de 1 pour cet état et 0 ailleurs), tandis que `StationaryDistributionRLGL()` utilise uniquement

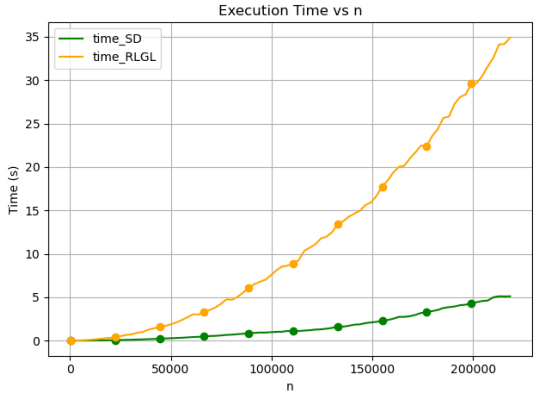

Figure 2: Temps d'exécution en fonction de n pour SD et RLGL

une distribution initiale uniforme sur l'ensemble des états. Par conséquent, les vecteurs stationnaires produits sont différents, bien que les deux méthodes soient mathématiquement valides dans leur contexte respectif.

Les paramètres expérimentaux sont les suivants :

- Limite maximale d'itérations : 1000 pour RLGL, 2000 pour SD (fixée en interne).
- Chaque méthode est exécutée 30 fois pour chaque valeur de n, et le temps moyen est reporté.
- **Remarque** : le nombre d'itérations utilisé par chaque méthode n'est pas disponible directement dans les objets retournés par Marmote. Ces valeurs ont été observées via des sorties *log* affichées lors de l'exécution. Par conséquent, les comparaisons concernant le nombre d'itérations reposent uniquement sur des tests ponctuels et ne font pas l'objet d'une moyenne statistique rigoureuse.

Sur le plan des performances, les expérimentations ont révélé les tendances suivantes :

- Pour les petites valeurs de n, la méthode SD effectue environ dix fois plus d'itérations que RLGL, mais reste globalement plus rapide.
- RLGL converge en un nombre d'itérations réduit, même lorsque n devient grand, mais chaque itération semble plus coûteuse en temps de calcul.
- Lorsque n dépasse 150 000, le temps d'exécution de RLGL devient significativement supérieur à celui de SD, malgré un nombre d'itérations plus faible.

En résumé, RLGL offre une convergence plus rapide en nombre d'itérations,

mais semble présenter une complexité algorithmique par itération plus élevée que SD.

5.3 Résultats pour les MDP

Tous les algorithmes ont été évalués avec un critère de convergence fixé à $\epsilon = 10^{-8}$ et un nombre maximal d'itérations global limité à 5000.

Pour les méthodes `PolicyIterationModified` et `PolicyIterationModifiedGS`, deux paramètres supplémentaires sont requis :

- `delta` : le critère de convergence interne pour les itérations sur les valeurs
- `maxInIteration` : le nombre maximal d'itérations internes autorisées.

Afin d'étudier l'influence de ces paramètres sur les performances, nous avons testé les trois combinaisons suivantes :

$$(0{,}01,\ 500), \quad (10^{-4},\ 1500), \quad (10^{-7},\ 5000)$$

5.3.1 Finite Horizon MDP

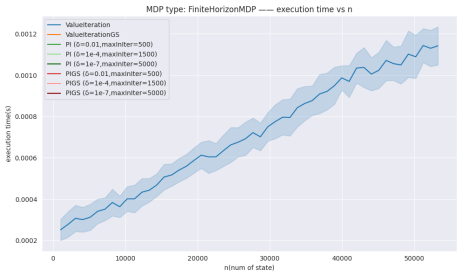

Figure 3: Temps d'exécution en fonction de n pour FiniteHorizonMDP

La figure 3 présente le temps d'exécution d'algorithme appliquée au modèle `FiniteHorizonMDP` en fonction du nombre d'états n.

Cependant, Marmote ne propose qu'une seule méthode d'implémentation pour ce type de MDP : `ValueIteration`. Il n'est donc pas possible, dans ce cas, de comparer plusieurs algorithmes de résolution. Néanmoins, la courbe montre clairement que le temps d'exécution augmente de manière modérée avec la taille du modèle.

5.3.2 Discounted MDP

Les figures 4 représente le temps d'exécution de différentes d'algorithmes appliquées au modèle `DiscountedMDP` en fonction du nombre d'états n.

Nous constatons que la méthode d'itération sur les valeurs simple produit un temps de calcul beaucoup plus significatif que les méthodes d'itération sur les politiques. Cela s'explique par les méthodes d'itération sur les politiques reposant sur l'amélioration de la politique. À chaque itération, elles évaluent la politique en calculant la valeur de la fonction V, puis améliorent la politique

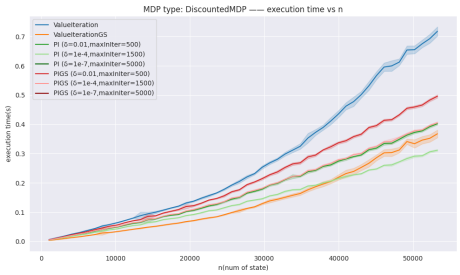

Figure 4: Temps d'exécution en fonction de n pour DiscountedMDP

en fonction de la valeur de V obtenue. Bien que cette approche présente un coût computationnel beaucoup plus élevé que les méthodes d'itération sur les valeurs, elle permet d'accélérer la convergence de la politique.

Nous avons vérifié cette conjecture en examinant le log affiché par l'instance Marmote et nous avons bien observé que les méthodes d'itération se terminent bien plus tôt que celles d'itération sur les valeurs.

Nous avons également remarqué un phénomène intéressant: bien que les méthodes d'itération sur les valeurs effectuent beaucoup plus d'itérations que celles sur les politiques, nous observons que le temps de calcul produit par la méthode d'itération sur les valeurs - version Gauss Seidel - est comparable à celui des méthodes d'itération sur les politiques, voire bien meilleur que certaines d'entre elles en termes de temps de calcul.

5.3.3 Total Reward MDP

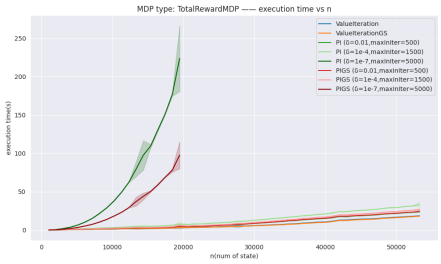

Figure 5: Temps d'exécution en fonction de n pour TotalRewardMDP

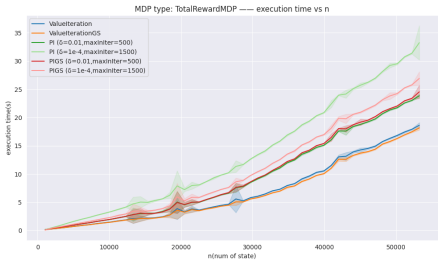

Figure 6: Zoomer sur la partie en bas

Les figures 5 et 6 représentent le temps d'exécution de différentes d'algorithmes appliqués au modèle `TotalRewardMDP` en fonction du nombre d'états n.

Nous remarquons les méthodes d'itération sur les valeurs sont plus performantes, en termes de temps de calcul, que les méthodes d'itération sur les politiques. Cette observation est complètement différente et contradictoire de

celle du `DiscountedMDP`. Après avoir examiné le log affiché par l'instance, nous constatons qu'aucune de ces méthodes n'arrive à converger vers l'optimum.

Cela peut s'expliquer par la différence de nature entre ces deux modèles. En effet, dans un modèle `DiscountedMDP`, les récompenses à long terme ont moins d'importance en raison de la présence d'un facteur d'actualisation, tandis que pour dans un modèle `TotalRewardMDP`, l'objectif est d'optimiser la récompense total reçu tout au long du parcours. Cela entraîne donc à une convergence plus lente de la fonction de valeur V. Par conséquent, un modèle `TotalRewardMDP` nécessite plus d'itérations qu'un modèle `DiscountedMDP`. En particulier, dans notre cas, 5000 itérations ne suffisent pas à assurer la convergence.

5.3.4 Average MDP

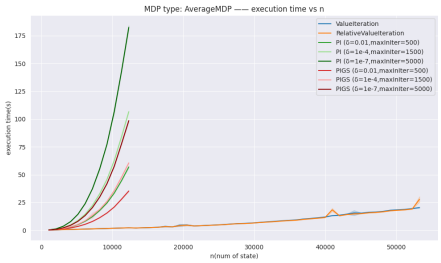

Figure 7: Temps d'exécution en fonction de n pour AverageMDP

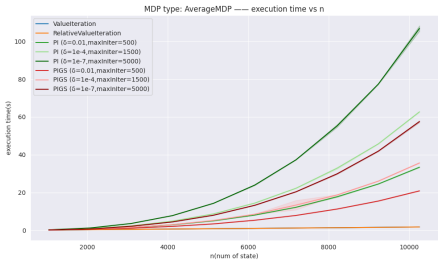

Figure 8: Zoomer sur la partie à gauche

Les figures 7 et 8 présentent les temps d'exécution mesurés pour différentes méthodes appliquées au modèle `AverageMDP`, en fonction du nombre d'états n. Pour des raisons de lisibilité, la figure 8 fournit un zoom sur la zone où $n \leq 10\,000$.

Nous observons que les méthodes utilisant la variante `Gauss-Seidel` (suffixe `GS`) sont plus performantes que leurs versions standard. Cela s'explique par une meilleure efficacité de calcule grâce à la mise à jour séquentielle des valeurs, propre à la stratégie Gauss-Seidel.

Les méthodes de type `ValueIteration` (y compris la version `Relative`) conservent une très bonne stabilité de temps d'exécution et ne semblent pas souffrir de l'augmentation de la taille du modèle, contrairement aux méthodes d'itération sur les politiques dont la complexité augmente rapidement avec n.

6 Conclusion

6.1 Limites du format JANI

Le format `JANI` ne permet pas de spécifier explicitement une distribution initiale sur l'ensemble des états. Par défaut, l'état initial est déterminé uniquement par les valeurs initiales de toutes les variables, ce qui correspond à un seul état bien défini dans le graphe de transition. Ainsi, aucune notion de distribution initiale probabiliste n'est intégrée dans la spécification JANI.

Dans notre extension `JaniR`, nous proposons de remédier à cette limitation en ajoutant un champ optionnel permettant de définir une distribution initiale explicite sur plusieurs états. Cette modification permet une modélisation plus complète, notamment pour les chaînes de Markov non ergodiques.

6.2 Limites de Marmote

6.2.1 Limite 1

Dans le cas des chaînes de Markov (MC), la distribution stationnaire est, en théorie, indépendante de l'état initial dès lors que la chaîne est irréductible et apériodique. Cependant, dans notre modèle, la chaîne contient plusieurs états absorbants, ce qui la rend non ergodique. Dans ce contexte, la distribution stationnaire dépend fortement de l'état initial.

Or, les deux méthodes de calcul proposées par Marmote — `StationaryDistribution()` et `StationaryDistributionRLGL()` — produisent des résultats différents. La première utilise une distribution initiale concentrée sur l'état 0, tandis que la seconde adopte une distribution initiale uniforme. Ces choix influencent fortement les résultats lorsque la chaîne possède plusieurs composantes absorbantes.

N'ayant pas accès au code source de Marmote, nous supposons que ces méthodes ne prennent pas explicitement en compte la présence de plusieurs états absorbants, ce qui explique la divergence observée. Par conséquent, comme ces deux méthodes ne calculent pas exactement le même objet mathématique, la comparaison de leurs résultats ne permet pas d'évaluer objectivement la performance ou la qualité de l'une par rapport à l'autre. **Les courbes obtenues ne doivent donc pas être interprétées comme un indicateur fiable d'efficacité algorithmique.**

Références

- marmote : https://marmote.gitlabpages.inria.fr/marmote/

Annexes

Nous présentons ici la syntaxe complète de JANI (JSON Automata Network Interface), en particulier pour les modèles MDP et DTMC que nous avons étudiés tout au long du projet, ainsi que les extensions proposées pour le langage JANI-R. En effet, un fichier JANI est un document structuré au format JSON, organisé autour des principaux composants suivants:

- Déclaration du type de modèle ("type").

  Le champ "type" indique le type de modèle utilisé, tels que "dtmc" (Discrete Time Markov Chain), "mdp" (Markov Decision Process), ou encore d'autres types de modèles. Cependant, dans le cadre de ce projet, nous ne les étudions pas.

  Lors de la conversion d'une instance DataMarmote en fichier JANI-R, le type est précisé de manière plus spécifique, tels que AverageMDP, TotalRewardMDP, DiscountedMDP, FiniteHorizonMDP ou MarkovChain. Notons que certains paramètres spécifiques à ces modèles sont également introduits, tels que "criterion" (critères d'optimisation pour les modèles MDP), "gamma" (facteur d'actualisation pour les modèles DiscountedMDP et FiniteHorizonMDP) et "horizon" (utilisé pour le modèle FiniteHorizonMDP).

- Déclaration des actions ("actions").

  Ce champ contient la liste des actions utilisées dans le modèle. Toutefois, cette liste peut être incomplète, voire absente: une transition sans action explicite est alors interprétée comme une action silencieuse. La taille des actions du modèle construit est déterminée par le nombre maximal de transitions sortantes d'un état.

  Dans le langage JANI-R, les actions silencieuses ne sont pas autorisées: toute transition déclenchée par une action doit obligatoirement être associée à une action explicite, préalablement déclarée.

- Déclaration des variables ("variables")

  Notons qu'un état du modèle est représenté par la combinaison des valeurs des variables non-transitoires et des locations (nous détaillons ce point dans la section suivante). Le nombre total d'états possibles correspond alors au produit des cardinaux des domaines des variables et des locations.

  Le type des variables non-transitoires est typiquement entier ou booléen, avec des domaines finis. Cette restriction ne s'applique pas aux variables transitoires, dont les valeurs ne participent pas à la définition de l'état.

Chaque variable possède une valeur initiale. En l'absence de valeur explicitement définie, sa valeur initiale est considérée comme l'ensemble des valeurs possibles dans son domaine. L'état initial du modèle est donc déterminé par la combinaison de toutes les valeurs initiales des variables non-transitoires, ainsi que des locations initiales des automates.

- Déclaration des constantes ("constantes")

  Chaque constante du modèle est associée à une valeur de son type. Lorsqu'une constante ne possède pas de valeur, cela signifie qu'elle doit être fournie au moment de l'instanciation.

- Déclaration des automates ("automata")

  Un modèle JANI peut contenir un ou plusieurs automates, cependant, JANI-R n'en autorise qu'un seul. Chaque automate comprend:

  - `locations`: les états internes de l'automate.
  - `initial-location`: l'état initial de l'automate.
  - `edges`: la liste des transitions, chacune définie par :
    - une `action` (facultative en JANI, mais obligatoire dans JANI-R).
    - une `guard`: expression booléenne précisant les conditions de déclenchement.
    - des `destinations`: états cibles avec une probabilité, un ensemble d'affectations, ainsi qu'une récompense si le modèle est en JANI-R.

  Dans un fichier JANI, l'état global du système n'est pas uniquement défini par les valeurs des variables non-transitoire. Il est en réalité constitué de deux composantes principales :

  - la **location** actuelle de chaque automate (similaire à un état de contrôle)
  - la **valeur actuelle de chaque variable non-transitoire déclarée**

  Ainsi, un **état complet** est une combinaison :

  $$\text{état} = (\text{location}, \text{valeurs des variables non-transitoire})$$

  On peut considérer la `location` comme une forme de **variable implicite**, qui ne figure pas dans la section `"variables"`, mais qui joue un rôle équivalent dans la définition des états.

  Dans la plupart des modèles rencontrés, chaque automate ne possède qu'une seule `location`: les transitions d'états sont alors entièrement déterminées par les variations des variables. Cependant, il existe également des fichiers JANI où plusieurs `locations` sont utilisées, et dans ce cas, elles

participent à la définition de l'espace d'états. À l'extrême, un modèle peut même être défini uniquement par les `locations`, sans aucune variable déclarée.

Nous pouvons considérer la `location` comme une forme de **variable implicite**, qui n'apparaît pas dans la section `"variables"`, mais qui joue un rôle équivalent dans la définition des états.

Important : Le fichier JANI ne liste pas explicitement tous les états ou transitions. Le graphe de transitions est reconstruit dynamiquement en :

1. Générant l'état initial à partir des variables.
2. Parcourant les `edges` : si une garde est satisfaite dans un état, on applique les affectations pour créer les états suivants.
3. Répétant ce processus jusqu'à ce qu'aucun nouvel état n'apparaisse.

Les états atteints forment l'espace d'états accessible (qui peut être strictement plus petit que l'espace total des combinaisons de variables).

Lorsque plusieurs automates sont présents, ils peuvent interagir via des actions synchronisées.

- Déclaration des propriétés ("properties")

  Ce champ définit les objectifs de vérification du modèle, tels que les propriétés PCTL, LTL ou reachability. Dans notre projet, ces propriétés sont traduites en conditions de récompense pour guider l'apprentissage de politique.