Travail d'Étude et de Recherche

Identification d'états se répétant infiniment souvent dans les structures de Kripke

19 mai 2025

Auteurs
Boudjémâa Salah : boudjemaa.salah@etud.univ-evry.fr
Max Fillere : max.fillere@etud.univ-evry.fr

Enseignant encadrant
Johan Arcile : johan.arcile@univ-evry.fr

Table des matières

Introduction

Ce projet s'inscrit dans le domaine de la vérification de modèles (model checking). Un ensemble de méthodes utilisé pour vérifier que les comportements d'un système modélisé respectent des propriétés spécifiées. La structure des systèmes étudiés est souvent modélisée par des graphes orientés que l'on nomme **structure de Kripke**. De nombreuses méthodes sont appliquées sur ces structures afin d'identifier certaines propriétés ou comportements. C'est le cas de la ***layered based exploration***, une technique d'exploration des états du système qui repose sur une décomposition du graphe en couches successives. Pour cet outil, l'identification d'un état ou d'un groupe d'états présents infiniment souvent est nécessaire. Ces éléments se caractérisent par leur présence dans toutes les traces d'exécution du graphe. C'est-à-dire qu'ils sont présents dans tous les chemins possibles du graphe. Une propriété intéressante de tels états est que leur suppression rend le graphe acyclique.

Objectif du Travail d'Étude et de Recherche

L'objectif du projet est donc d'implémenter un ensemble d'algorithmes capables d'identifier un état ou un groupe d'états présents infiniment souvent dans une structure de Kripke donnée.

Deux approches ont été étudiées :

— Une première méthode repose sur l'intersection des cycles pour déterminer s'il existe un état présent infiniment souvent.

— Un algorithme glouton qui étudie la cyclicité du graphe en fonction de la suppression d'un ou plusieurs états, en lien avec le ***Directed Feedback Vertex Set (DFVS)***.

Le problème du DFVS consiste à identifier le plus petit ensemble de sommets dont la suppression rend un graphe orienté acyclique. Ce problème est **NP-complet**, c'est-à-dire que trouver une solution demande un temps exponentiel, tandis que vérifier une solution se fait en temps polynomial.
Ce rapport présente les fondements des problèmes étudiés, les algorithmes développés, les choix d'implémentation ainsi que les résultats obtenus.

1 Problèmes étudiés

1.1 Structure de Kripke

Une structure de Kripke est un outil utilisé en model checking pour représenter le comportement d'un système. Il s'agit d'un graphe orienté dont les nœuds sont des états accessibles du système et les arcs représentent les transitions entre ces états. Une fonction d'étiquetage associe à chaque état un ensemble de propositions logiques.

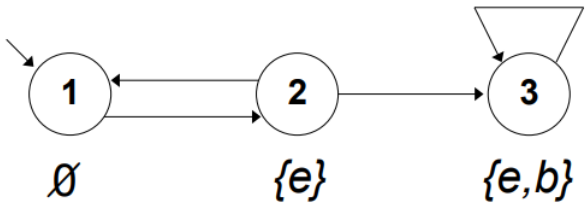
Définition formelle : Une structure de Kripke est notée $K = (Q, q_0, T, AP, L)$ où :

- Q : ensemble fini des états,
- q_0 : état initial, $q_0 \in Q$,
- T : ensemble fini des transitions, $T \subseteq Q \times Q$,
- AP : ensemble fini des propositions atomiques,
- $L : Q \to 2^{AP}$: fonction d'étiquetage.

Une exécution de K représente une séquence infinie d'états à partir de q_0 et en suivant les transitions T. Tout état doit avoir au moins une transition sortante pour éviter les blocages.

L'objectif est de vérifier si une propriété logique Φ est satisfaite par la structure.

Exemple du ressort :



$K = (Q, q_0, T, AP, L)$
$Q = \{1, 2, 3\}$
$q_0 = 1$
$T = \{(1, 2), (2, 1), (2, 3), (3, 3)$
$AP = \{e, b\}$(e : étendu, b : bloqué)
$L = \{1 : \emptyset, 2 : \{e\}, 3 : \{e, b\}\}$

Une propriété possible serait : « le ressort peut-il rester dans une situation où il est étendu et bloqué ? ». Dans cet exemple simple, on peut conclure que oui. Pour des cas plus complexes, des méthodes spécifiques sont nécessaires.

1.2 NP-complet

Un problème est dit NP-complet lorsqu'il est complet pour la classe NP :

> Un problème est complet pour une classe de complexité lorsqu'il fait partie de cette classe, et si sa résolution permet de résoudre les autres problèmes de la classe aussi efficacement.
>
> La classe NP (non déterministe polynomial) fait référence à un type de problème dont on peut vérifier une solution rapidement (temps polynomial). Mais trouver une solution optimale se fait en temps exponentiel en fonction des données d'entrées. Dans notre exemple, cela signifie qu'il est « rapide » (temps polynomial) de déterminer si le graphe est acyclique. Mais il n'est pas possible de déterminer un ensemble minimal d'états à retirer, dans un temps autre qu'exponentiel.

1.3 Layered Based Exploration

Le layered based exploration (exploration par couche) est une méthode d'exploration d'un graphe qui consiste à l'analyser couche par couche, à partir de l'état initial, en mêlant parcours en profondeur et parcours en largeur. Une couche étant un ensemble d'états ayant au même moment un certain nombre de paramètres en commun.

Cette méthode nécessite, pour être appliquée, que soit identifié un état ou un groupe d'états présents infiniment souvent dans toutes les traces d'exécutions. Ce qui nous renvoie au problème du Directed Feedback Vertex Set.

1.4 Directed Feedback Vertex Set (DFVS)

Le directed feedback vertex set (DFVS) est un problème qui s'inscrit dans le domaine de la théorie des graphes. L'objectif est de trouver le plus petit sous-ensemble de sommets dont la suppression rend le graphe acyclique. Ce problème appartient à la classe des problèmes NP-complets, ce qui signifie qu'il est difficile de trouver, en temps polynomial, une solution sur de grands graphes.

Définition formelle : Soit un graphe $G = (V, E)$, avec :

V : ensemble des sommets,

E : ensemble des arcs.

Objectif : trouver $S \subseteq V$ de taille minimale tel que $G \setminus S$ soit acyclique.

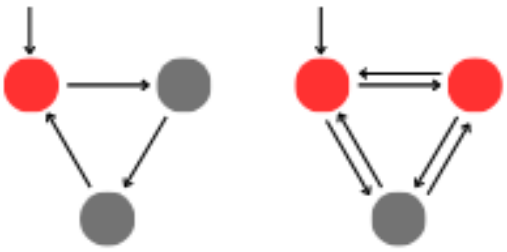
Observation :

Si un seul état est présent infiniment souvent, alors il appartient à tous les cycles :

$$\forall c \in C,\ e \in c$$

Si un groupe d'états est présent infiniment souvent, alors chaque cycle contient au moins un de ces états :

$$\forall c \in C,\ s \cap c \neq \emptyset$$



En retirant les états marqués en rouge, les graphes deviennent acycliques.

2 Implémentation

Pour le développement du projet, nous avons décidé d'utiliser le langage Python. Les librairies disponibles en Python sont les plus adaptées pour répondre aux problématiques du sujet.

2.1 Librairies

Networkx
C'est une librairie dédiée à la création et la manipulation de graphes. Elle permet de modéliser des graphes avec différents paramètres, notamment l'orientation et la pondération de ces derniers. On y retrouve également des algorithmes pour réaliser des calculs de chemin, ou des calculs de flots. Dans notre cas, la librairie a été utilisée afin de créer et visualiser des structures de Kripke (graphe orienté). Cela est rendu possible grâce à sa complémentarité avec Matplotlib.

Matplotlib
C'est une large librairie principalement utilisée dans la visualisation 2D d'une grande variété de graphiques, diagramme... Dans le contexte du projet, la librairie a été utile pour visualiser, dans une fenêtre graphique, les différents graphes générés. Cette visualisation a permis, dans un premier temps, de vérifier la bonne construction du graphe. Dans un second temps, elle a permis de veiller au bon fonctionnement de méthodes modifiant la structure du graphe, comme la méthode de suppression de sommet/arc.

2.2 Classe Graphe

L'objet principal de notre projet est donc le graphe. C'est pourquoi il nous a paru naturel de créer une classe Graphe, afin d'y appliquer différentes opérations qui conduiront à une solution à la problématique initiale. Nous considérerons, pour ce projet, que tous les graphes générés par l'utilisateur respecteront la forme de la structure de Kripke, c'est-à-dire que tout état a, au moins, une transition sortante.

2.3 Attributs :

états : liste des sommets du graphe

transitions sortantes : Dictionnaire de transitions avec

clé : état

valeur : liste d'états sortants

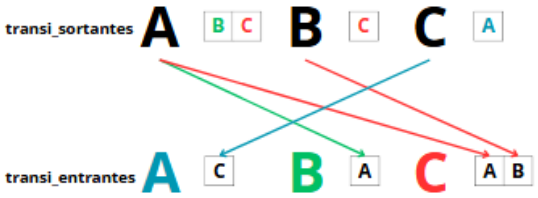
état initial : état initial de la structure de Kripke (utile seulement pour la visualisation)

transitions entrantes : Dictionnaire de transitions avec

clé : état

valeur : liste d'états sortants

Le dictionnaire est créé à partir de celui des transitions sortantes Pour chaque état E du graphe, on parcourt les clés K du dictionnaire des transitions sortantes, afin d'ajouter cette même clé K en tant que valeur dans la liste des transitions entrantes de notre état E.



cycles : Liste 2D des cycles présents dans le graphe.

etats_dans_cycle : Liste des états présents au moins dans un cycle du graphe. Autrement dit, l'ensemble des états formant les cycles du graphe. Ces deux derniers attributs sont calculés respectivement par les méthodes :

trouver_cycle()

get_etats_dans_cycles

De cette manière, le calcul est réalisé une unique fois, seul l'attribut sera utilisé dans les méthodes.

2.4 Méthodes :

Pour veiller au bon déroulement des méthodes, nous avons créé un fichier de test. Pour ce faire, nous avons généré différents graphes de petites tailles, avec des propriétés distinctes. Notamment la complétude du graphe, le nombre de cycles ou la présence d'auto-boucles.

L'avantage des graphes de taille réduite, associé à la visualisation, est la possibilité de déduire le résultat attendu pour chaque méthode. En effet, ces dernières étant déterministes, le résultat sera le même à chaque exécution.

Magic methods :
En python, les méthodes magiques sont des méthodes au nom prédéfini qui permettent d'adapter des objets créés aux méthodes natives du langage.
Dans notre cas, elles nous ont permis de réaliser des tests de nos méthodes.

__eq__ (equal) : Permet de comparer deux graphes
La méthode compare un à un les attributs du graphes, non issus de l'execution d'une méthode

__str__ (string) : Permet d'afficher les attributs du graphe
La méthode "print" les attributs : états, transitions sortantes et transitions sortantes

visualiser () :

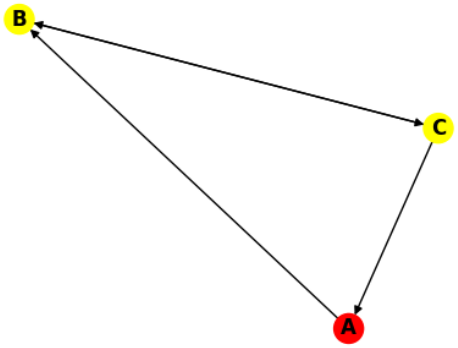
Objectif :
La méthode permet d'avoir un rendu visuel du graphe crée, on y retrouve les états, les transitions ainsi que l'état initial qui est mis en avant (sommet de couleur différente). La méthode est basée sur l'usage des librairies Networkx et Matplotlib.

Fonctionnement :
La création de l'affichage se fait en 5 étapes majeures :

— Création de l'objet
— (graphe orienté de networkx)
— Ajout des états de l'objet Graphe créé plus tôt

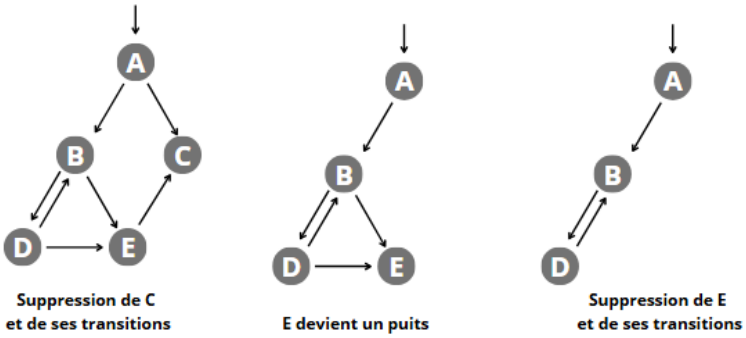
— Ajout des transitions sortantes, après les avoir restructurées (dictionnaire état : liste de transitions sortantes → liste de tuples (A, B))
— Choix de la couleur des sommets
— Création de l'objet et de la fenêtre d'affichage avec les attributs créés précédemment

```
graphe = Graphe(["A", "B", "C"], {"A": ["B"], "B": ["C"], "C": ["A","B"]}, "A")
graphe.visualiser()
```



supprimer_etat(etat_a_supp) :

Objectif :
La méthode permet de supprimer un état du graphe, les transitions dont il fait partie, ainsi que les puits engendrés.
Fonctionnement :
Ce processus se fait en plusieurs étapes :
— Suppression de l'état dans la liste d'état
— Suppression des occurrences de l'état dans le dictionnaire des transitions entrantes.
— Suppression de l'état dans le dictionnaire des transitions sortantes
— Suppression des occurrences de l'état dans le dictionnaire des transitions sortantes. Si la liste est vidée, c'est qu'on a trouvé un puit. La méthode de suppression est donc appelée de manière récursive sur l'état concerné
— l'état est ensuite supprimé du dictionnaire des transitions entrantes.



Par la suite, la méthode a été supprimée, car une version plus performante du code a été implémentée.

get_etats_dans_cycles () :

Objectif :
La méthode permet de déterminer les états présents dans, au moins, un cycle du graphe.

Retour :
Liste d'états ([str]) → [A,B,C].

Fonctionnment :
Pour cela, on crée un dictionnaire vide. En parcourant les cycles du graphe, état par état, on crée un couple état :état Il suffit de récupérer les clés du dictionnaire.

etat_infini_brute_force () :

Objectif :
La méthode permet de retourner s'il existe un état, pour qui quand on le retire, le graphe devient acyclique. Si un tel état n'existe pas, un message est renvoyé.

Retour :

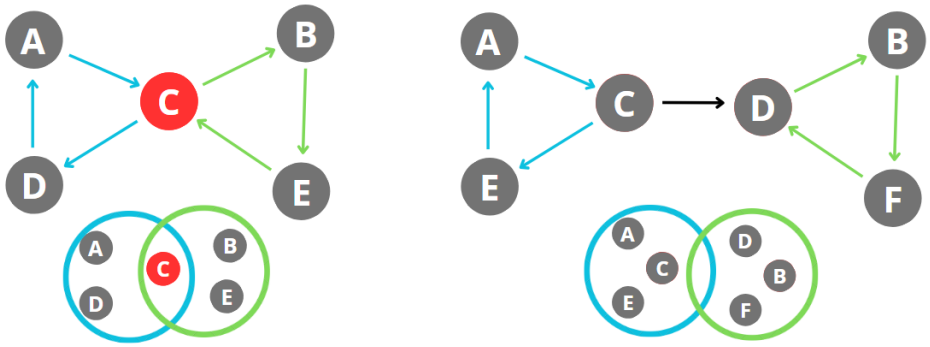
- Un état (str) → A
- Un message (str) → "plus de 1 état présent infiniment souvent"

Fonctionnment :
Le fonctionnement est basé sur le retour de la méthode précédente. En effet, la particularité d'un état présent infiniment souvent est qu'il est présent dans tous les cycles du graphe. C'est une condition nécessaire pour annuler la présence de tous les cycles par la suppression de l'état.

Afin de trouver cet état, il faut donc calculer l'intersection de tous les cycles. Deux résultats sont alors possibles :

L'intersection est non vide : n'importe quel état de l'intersection satisfait la propriété et peut être retourné.

L'intersection est vide : Il n'y a pas d'état présent infiniment souvent, il faut donc chercher un groupe d'état satisfaisant la propriété.



etat_infinis_glouton () :

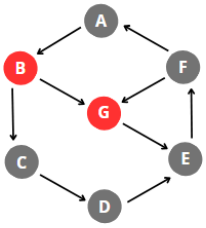
Objectif :
Cette méthode est l'implémentation d'un algorithme glouton permettant de déterminer un ensemble d'états présents infiniment souvent.

Retour :
Une liste [(str)] des etats trouvés

Fonctionnment :
Tant qu'il y a des cycles dans la liste de cycles (graphe cyclique) :

On sélectionne l'état qui a le plus de transitions sortantes (la sélection est faite parmi les états formant les cycles).

On supprime de la liste des cycles tous ceux qui comporte l'état sélectionné.



cycles : [ABCDEF, ABGEF, EFG]

t	1	2
ε	B	G
cycles	[EFG]	[]
états choisis	[B]	[G]

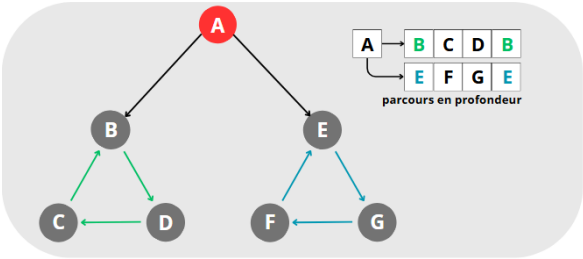
S'il n'y a qu'un seul état présent infiniment souvent, alors il est possible que le retour soit le même que la méthode précédente.

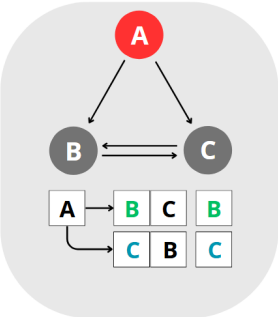
trouver_cycles () :

Objectif :
Cette méthode permet de trouver tous les cycles du graphe.

Retour :
Une liste [[(str)]] des cycles trouvés

Fonctionnment :
L'algorithme est basé sur un parcours en profondeur. Il enregistre le chemin en cours et détecte un cycle dès qu'un état est visité une nouvelle fois.





A noter que cette méthode détecte les cycles plusieurs fois s'ils sont accessibles depuis plusieurs branches. Il y a donc une étape intermédiaire qui fait office de filtre, afin de n'enregistrer le cycle que s'il n'a jamais été vu.

3 Complexité algorithmique

Nous allons, pour chacune de nos méthodes, étudier la complexité dans le pire cas. Nous poserons les variables suivantes :

n : le nombre d'états

c : le nombre de cycles

supprimer_etat (etat_a_supp) :

La méthode est composée de deux boucles sur les états entrants et sortants de l'état à supprimer.
Dans le pire des cas, chacune des boucles parcourt une liste de tous les sommets, alors une exécution de la méthode aura une complexité $O(2n)$ soit $O(n)$.
Cependant, la méthode peut être appelée de manière récursive si un état devient un puit. Ce qui fait que la méthode sera appelée sur $n-1$ états, puis possiblement sur $n-2$ états. Ainsi de suite jusqu'à ce qu'il n'y ait plus d'états dans le graphe.
Il y aura alors :

$$\frac{n(n+1)}{2} \text{ itérations}$$

On en déduit alors une complexité globale de :

$$O(n^2)$$

Rappelons que la méthode n'est plus utilisée, ce qui réduit la complexité du code.

get_etats_dans_cycles ()) :

Pour cette méthode, on parcourt chaque état de chaque cycle trouvé grâce à deux boucles imbriquées. On pose alors n et c respectivement le nombre d'états et le nombre de cycles de la structure.

On en déduit une complexité de :

$$O(n \times c)$$

etat_infini_brute_force ()) :

La complexité de cet algorithme réside dans le parcours des cycles afin d'en faire l'intersection. Pour ce faire, on observe une première boucle sur la liste de cycles qui effectue des intersections avec les cycles suivants. Cette opération d'intersection est une boucle implicite, puisque l'on va vérifier, pour chaque état des cycles suivants, s'il est présent dans la liste actuelle.

On en déduit une complexité de :

$$O(c \times n)$$

etat_infini_glouton ()) :

Cet algorithme est construit sur une boucle principale, tant qu'un cycle existe. Dans cette boucle, on parcourt tous les cycles une fois.

Dans le pire cas, il y a un cycle par état (auto-boucle), donc la boucle principale s'exécuterait n fois. La boucle secondaire s'exécute c - (i - 1) fois, avec i le nombre d'exécutions de la boucle principale.
Ce qui donne une complexité de :

$$O(2 \times \frac{c(c+1)}{2}) \text{ soit } O(c^2)$$

Dans la boucle principale, l'état à supprimer est mis à jour en fonction du nombre de transitions (on choisit celui qui en a le maximum). Cependant, ce calcul est fait en appelant la méthode etats_dans_cycles, qui a, rappelons-le, une complexité de :

$$O(n \times c)$$

On déduit alors, une complexité globale pour la méthode de :

$$O((n \times c) \times c + c^2) \text{ soit } O(n \times c^2)$$

trouver_cycles ()) :

Pour cette fonction, le pire cas serait un graphe complet de taille n.

On peut dans un premier temps déterminer le nombre de cycles d'un tel graphe. Ici, ABC et ACB seront considérés comme le même cycle, car retirer un état dans l'un des deux, aura le même impact sur le deuxième.

Dans un graphe complet de taille n, chercher les cycles de taille k correspond à calculer k parmi n.
La somme de tous les cycles est donc :

$$\sum_{k=2}^{n}(k \; parmis \; n) = \sum_{k=2}^{n} \frac{n!}{(n-k)!k!}$$

La méthode peut repérer le même cycle à partir de tous les états, ce qui signifie que l'on peut observer toutes les permutations de chacun des cycles trouvés au fur et à mesure de la boucle while.

Le nombre d'exécutions est donc calculé de la manière suivante :

$$n + \sum_{k=2}^{n}(\frac{n!}{(n-k)!k!} \times k!) = n + \sum_{k=2}^{n}(\frac{n!}{(n-k)!})$$

(On ajoute n, car la pile incrémente chaque compteur jusqu'à dépasser le nombre de voisins pour chaque état.)

Si on considère le cas où les graphes présentent des auto-boucles, il faut savoir que la méthode comptera cette auto-boucle dès que l'état est présent dans le chemin. Il faudra donc ajouter cette valeur au nombre d'exécutions calculé précédemment.

$$\frac{nb\_executions}{n}$$

Finalement, la complexité dans le pire cas est en :

$$O(n!)$$

On sait que $O(n!) > O(2^n)$.

Cependant, dans la plupart des cas, le graphe a un nombre de cycles modéré. On peut donc conclure que dans les pires cas, une solution brute force serait meilleure. Mais dans le cas d'un graphe de grande taille avec peu de cycles, cette méthode, étant linéaire sur le nombre de cycles, est meilleure.

On peut donc réduire la complexité à :

$$O(c \times n)$$

__init__ :

La méthode d'initialisation à également une complexité qu'il ne faut pas négliger. La création du premier dictionnaire de transition entrante se fait en une boucle sur les états, soit une complexité de :

$$O(n)$$

Le deuxième dictionnaire est rempli grâce à une boucle imbriquée, la première sur les états, la seconde sur les clés du dictionnaire des transitions sortantes, autrement dit les états. La complexité est donc de :

$$O(n^2)$$

Finalement, on peut ajouter les méthodes trouver_cycles et get_etats_dans_cycles qui sont utilisées pour remplir les attributs du graphe. On détermine alors une complexité de :

$$O(n^2 + 2 \times c \times n) \text{ soit } O(n(2c + n))$$

En conclusion, la complexité pour le calcul d'un état présent infiniment souvent est :

__init__ : $O(n(2c + n))$
etat_infini_brute_force : $O(c^2)$
etat_infinis_glouton : $O(c \times n)$

Somme : $O(n^2 + c^2 + 3cn)$ soit $O(n^2 + c^2))$

4 Tests et Benchmark

4.1 Tests

Pour vérifier le bon fonctionnement des algorithmes, une phase de tests unitaires a été mise en place. De cette manière, chaque modification du code a été validée une fois que les tests étaient aussi validés. Cette série de tests a été mise en place à l'aide de la librairie **unittest** de Python. Les tests couvrent les méthodes :

get_etats_dans_cycles

etat_infini_brute_force

etas_infinis_glouton

trouver_cycles

Chaque test compare le résultat renvoyé par la méthode avec une valeur de sortie attendue. Les tests étant réalisés sur des graphes de petite taille (4-5 états max), il est aisé de déterminer le résultat attendu. Les différents graphes générés couvrent un ensemble de caractéristiques pouvant créer des comportements particuliers. On note la présence ou non d'auto-boucle, des graphes complets ou non. Cependant, on considérera que chaque graphe généré par l'utilisateur respecte les propriétés d'une structure de Kripke, à savoir notamment que chaque état du graphe présente au moins une transition sortante.

4.2 Benchmark

Afin d'évaluer les performances de différentes méthodes au fur et à mesure des versions, une méthode de benchmark a été implémentée, suivant le principe suivant.
Deux ensembles de graphes créés avec chacune des versions du code. Ces graphes sont sensiblement les mêmes que ceux utilisés pour les tests. C'est-à-dire qu'ils couvrent différentes propriétés que peut avoir un graphe.

Les méthodes :

etat_infini_brute_force

etats_infinis_glouton

trouver_cycles

sont exécutées un nombre défini de fois, pour une meilleure précision des résultats.

5 Conclusion

En conclusion, pour répondre à la problématique de ce Travail d'Étude et de Recherche, nous avons implémenté deux éléments principaux :

— une classe `Graphe`, dont la construction repose sur la définition formelle des structures de Kripke, afin de modéliser les systèmes étudiés ;

— deux méthodes permettant de trouver un état ou un groupe d'états présents infiniment souvent dans toutes les traces d'exécution.

Les études de complexité ont permis de mettre en évidence les avantages et les limites de chaque méthode. Dans l'ensemble, une solution *brute force* est plus précise, mais devient inefficace dès que l'on traite des graphes de grande taille ou fortement cycliques. À l'inverse, la méthode *états infinis glouton* constitue une approche plus rapide et conceptuellement simple, mais elle est, par définition, approximative dans ses résultats.

Ces constats ouvrent des perspectives intéressantes. Il serait pertinent d'explorer d'autres types d'implémentations, notamment en s'appuyant sur l'analyse des sous-arbres ou d'autres sous-structures du graphe.

6 Bibliographie

Références

[1] Stephan MERZ, *An introduction to model checking*. Inria, 2008.

[2] *NetworkX documentation*, 2005.

[3] J. Arcile, H. Klaudel, R. Devillers, *Dynamic exploration of multi-agent systems with periodic timed tasks*. Fundamenta Informaticae, 2020.

[4] Angrick, B. Bals, K. Casel, S. Cohen, T. Friedrich, N. Hastrich, T. Hradilak, D. Issac, O. Kißig, J. Schmidt, L. Wendt, *Solving Directed Feedback Vertex Set by Iterative Reduction to Vertex Cover*. lipics, 2023.