Sorbonne Université
Master ANDROIDE

Rapport projet
Allocation en ligne de tâches

COCOMA

Rayan Perotti-Valle
Malo Thomasson

Février 2025

Table des matières

1 Introduction

Dans le cadre de ce projet COCOMA, nous avons étudié et modélisé un problème d'allocation en ligne de tâches dans une flotte de taxis, impliquant plusieurs agents. Notre objectif principal était de minimiser la somme des coûts des trajets ainsi que des déplacements entre ces trajets. Pour y parvenir, nous avons exploré et comparé différentes méthodes afin d'identifier la plus optimale.

Le projet était structuré en trois phases. Dans la première, nous avons implémenté l'environnement de test. La seconde partie consistait à modéliser le problème sous la forme d'un DCOP. Enfin, la dernière phase portait sur l'étude et la mise en œuvre de différents protocoles de coordination multi-agents.
Dans les chapitres suivants de ce rapport, nous détaillerons les différentes parties de notre projet en expliquant nos choix de modélisation pour chacune d'elles. Enfin, nous conclurons par une présentation et une analyse de nos résultats.

Lien GitHub du projet

2 Partie 1

Dans cette partie, nous avons mis en place les différents éléments nécessaires à la modélisation du problème d'allocation de tâches. Pour cela, nous avons opté pour une approche en programmation orientée objet avec Python. Nous avons ainsi défini trois classes principales : la classe **Task**, la classe **Taxi** et enfin la classe **Simulation**. De plus, le fichier *config.py* contient toutes les variables globales nécessaires à notre modélisation.

Nous avons choisi la programmation orientée objet, car elle nous semble être l'approche la plus adaptée pour modéliser ce problème de manière claire et modulaire. Ce dernier repose principalement sur deux entités : les taxis et les tâches, en dehors de la classe **Simulation**, qui sert à la définition des algorithmes et des fonctions associées, à la génération des tâches ainsi qu'à la visualisation. L'approche objet permet d'encapsuler les comportements et les données propres à chaque entité, améliorant ainsi la lisibilité et la maintenabilité du code.

De plus, comme précisé dans l'énoncé, chaque taxi et chaque tâche possèdent leurs propres attributs, et la programmation orientée objet offre une flexibilité pour gérer ces caractéristiques.Elle facilite l'ajout de nouvelles fonctionnalités.

1. Classe **Task** :

   — Cette classe représente une tâche qui sera allouée à un taxi. Elle contient différents attributs (id, point de départ, point d'arrivée), comme décrit dans le sujet.

2. Classe **Taxi** :

   — Cette classe représente un taxi qui possède plusieurs attributs, notamment la liste des tâches qui lui sont allouées, sa position actuelle ainsi que sa route à suivre. De plus, elle intègre diverses méthodes essentielles au fonctionnement d'un taxi dans notre modélisation.

   Parmi ces méthodes, on retrouve la fonction *update*, qui met à jour la position du taxi à chaque pas de temps et veille à ce qu'il suive correctement sa route. La classe comprend également la fonction *calculate_total_route_cost*, qui évalue le coût total du trajet du taxi, ainsi que *plan_route*, chargée de déterminer le meilleur ordonnancement des tâches en effectuant toutes les permutations possibles de la route afin de sélectionner la plus optimale.

   Nous avons également implémenté la fonction *build_route_from_current_tasks*, qui construit la route en suivant l'ordre actuel des tâches, sans effectuer de permutations. Selon l'algorithme utilisé, cette méthode est appelée à chaque ajout d'une nouvelle tâche à la route du taxi.

3. Classe **Simulation**

   — Nous avons choisi d'utiliser Pygame pour visualiser notre modélisation, car il offre une solution simple et efficace pour représenter graphiquement notre simulation. Cette visualisation nous permet d'avoir un aperçu immédiat des ajouts effectués et de mieux comprendre l'évolution du système en temps réel. Grâce

à cette approche, nous pouvons identifier plus rapidement les événements et comportements émergents, tels que la répartition des taxis ou l'optimisation des trajets, ce qui facilite le débogage et l'amélioration du modèle. Dans cette classe, on retrouve tous les paramètres de la simulation. Toutes les variables globales utilisées pour configurer la simulation se trouvent dans le fichier *config.py*. De plus, on retrouve la fonction *generate_task*, qui crée de nouvelles tâches à allouer, ainsi que d'autres fonctions que nous détaillerons par la suite. Nous avons aussi ajouté la possibilité de mettre la simulation en pause afin de pouvoir s'y retrouver lorsque le nombre de taxis et de tâches devient trop important.

— La simulation avec Pygame se présente de la manière suivante : les ronds rouges représentent les taxis, avec leur *id* écrit au-dessus. Les tâches sont représentées par un rond vert pour le point de départ et un rond bleu pour l'arrivée, les deux points étant reliés par un segment noir. Au départ de la simulation, un certain nombre de tâches sont créées et allouées, puis de nouvelles tâches apparaissent tous les T pas de temps ($TASK\_INTERVAL$ dans *config.py*). On trace ensuite la route de chaque taxi en connectant toutes les tâches d'un taxi dans l'ordre dans lequel il va les compléter. Lorsqu'une tâche est en cours d'exécution, elle passe en orange. Les taxis suivent leurs routes, et la tâche disparaît une fois terminée. Les nouvelles tâches sont directement ajoutées à la route des taxis, avec un réarrangement de l'ordre des tâches si nécessaire. Ci-dessous, vous trouverez un exemple de simulation avec 3 taxis et 5 tâches.

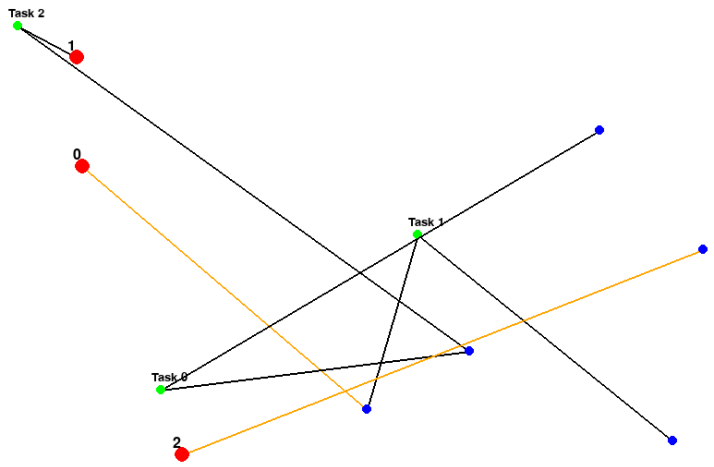

FIGURE 1 – Capture d'écran d'une simulation avec 3 taxis

Dans la première partie, nous devions également développer un algorithme de planification qui calcule la meilleure allocation de tâches. Pour cela, nous avons réalisé un algorithme *greedy* (*greedy_task_assignment*) qui teste toutes les permutations de tâches possibles et retourne celle avec le meilleur coût total. On utilise ensuite cette permutation pour parcourir la liste de tâches et allouer aux taxis les tâches les plus proches

3 Partie 2

Dans cette partie, nous devions aborder le problème de coordination multi-agents entre taxis à l'aide d'un DCOP. Pour cela, nous avons procédé en 3 étapes. Dans un premier temps, nous avons mis en place la fonction *generate_ dcop* qui permet, à partir de la liste de taxis et des nouvelles tâches, de modéliser le problème d'allocation en DCOP dans un fichier **dcop.yaml**. Ensuite, la fonction *solve_ dcop* permet la résolution du DCOP obtenu avec la fonction *generate_ dcop*. Enfin, avec la fonction *attribution_ dcop*, nous attribuons les tâches aux taxis comme assigné dans le résultat de *solve_ dcop*.

1. Pour ce qui est de la modélisation du problème, nous avons fait le choix de représenter les tâches comme des variables et les taxis comme le domaine de ces variables. La raison derrière ce choix repose sur le fait qu'une tâche peut être allouée à un seul taxi, ce qui signifie que chaque variable aura une seule valeur dans le domaine, facilitant ainsi la modélisation. En ce qui concerne les contraintes, nous en avons identifié trois types.
   (a) La contrainte de préférence : Elle sert uniquement de mécanisme de sécurité dans le cas où deux taxis se situent exactement à la même distance d'une tâche. Dans ce cas, la préférence permet d'attribuer la tâche au taxi ayant l'indice le plus petit.
   (b) La contrainte de distance : ici, on ajoute la distance entre deux tâches au coût total si elles sont assignés à un même taxi.
   (c) La contrainte liée au coût : Pour cette contrainte, la fonction *cost_ dcop* ajoute le coût de la tâche à allouer avec :
      — soit la distance entre le point d'arrivée de la tâche en cours d'exécution et le point de départ de la nouvelle tâche, si le taxi est en plein milieu d'une tâche.
      — soit la plus courte distance entre la position actuelle du taxi et le point de départ de la tâche, ou entre tout points d'arrivée des tâches déjà attribuées et le début de la nouvelle tâche, si le taxi n'est pas en train d'exécuter une tâche.

      Cette contrainte permet d'éviter un calcul erroné en supposant une position incorrecte pour le taxi. De plus, elle garantit que le taxi termine bien la tâche commencée avant de recalculer un nouvel itinéraire avec les nouvelles tâches allouées.
2. Ensuite, pour la résolution, notre fonction *solve_ dcop* prend en paramètres le fichier **dcop.yaml** généré par la fonction *generate_ dcop*, ainsi que l'algorithme DCOP à utiliser pour la résolution. Elle exécute ensuite la commande correspondant à l'algorithme spécifié en paramètre, puis récupère le résultat du DCOP dans le fichier *results.json*, dont le contenu est retourné par la fonction.

3. Enfin, on peut maintenant récupérer l'allocation générée par le DCOP. Pour cela, on récupère l'objet *assignment* du fichier *results.json*. Cette allocation se présente sous la forme suivante :

```
{
    "Tache_0": "T0",
    "Tache_1": "T1",
    "Tache_2": "T0",
    "Tache_3": "T1",
    "Tache_4": "T1"
}
```

`"Tache_0": "T0"` signifie que la tâche 0 a été assignée au taxi 0. Cet objet permet d'associer les tâches aux taxis correspondants, et la simulation se poursuit jusqu'à l'ajout de nouvelles tâches, où le processus sera répété.

4 Partie 3

4.1 Protocoles implémentés

Cette partie vise à implémenter des protocoles de négoctiation permettant aux agents de coordonner l'allocation des tâches.

4.1.1 Enchères Parallèles (PSI)

— **Principe** : Les taxis soumettent des offres en parallèle sur toutes les tâches disponibles. Chaque tâche est attribuée au taxi ayant l'offre la plus basse.
— **Problème** : Un taxi proche géographiquement et possédant beaucoup de tâches partout sur la carte peut monopoliser toutes les tâches, créant un déséquilibre de charge.

4.1.2 Enchères Séquentielles (SSI)

— **Principe** : Les tâches sont attribuées une par une. À chaque itération, le taxi avec le coût marginal le plus bas remporte la tâche.
— **Problème** : Effet "boule de neige" : un taxi accumulant des tâches voit ses coûts marginaux diminuer (car proche des destinations précédentes), lui permettant de gagner encore plus de tâches.

4.1.3 Enchères Séquentielles avec Regret

— **Principe** : Priorisation des tâches où la différence entre le meilleur et deuxième meilleur coût (*regret*) est la plus élevée.
— **Limite** : Bien que réduisant les mauvaises allocations précoces, le protocole ne résout pas complètement l'accumulation de tâches.

4.2 Limites des Protocoles

— **Problème central** : Les 3 protocoles tendent à favoriser les taxis déjà chargés, conduisant à des allocations sous-optimales.
— **Exemple** : Un taxi avec 5 tâches peut avoir un coût marginal bas pour une 6ème tâche située près de la destination d'une de ses tâches.

4.3 Améliorations Proposées

Une pénalité proportionnelle au nombre de tâches actuelles est ajoutée au coût marginal :

$$\text{Pénalité}(n_{\text{tâches}}) = \alpha \cdot n_{\text{tâches}} \quad (\alpha = \text{coefficient, 50 dans notre code})$$

Ceci décourage les taxis surchargés de surenchérir systématiquement.

5 Résultats

5.1 En fonction du temps total pour finir l'ensemble des 50 tâches

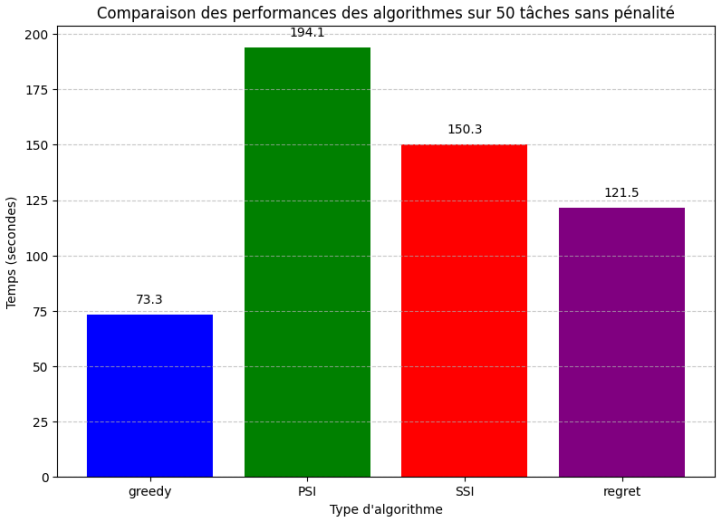

FIGURE 2 – Comparaison du temps nécessaire pour les pour accomplir 50 tâches, sans pénalité.

On fait maintenant la même execution mais avec pénalité sur la fonction *insertion_heuristic*, qui va influer seulement PSI, SSI, regret.

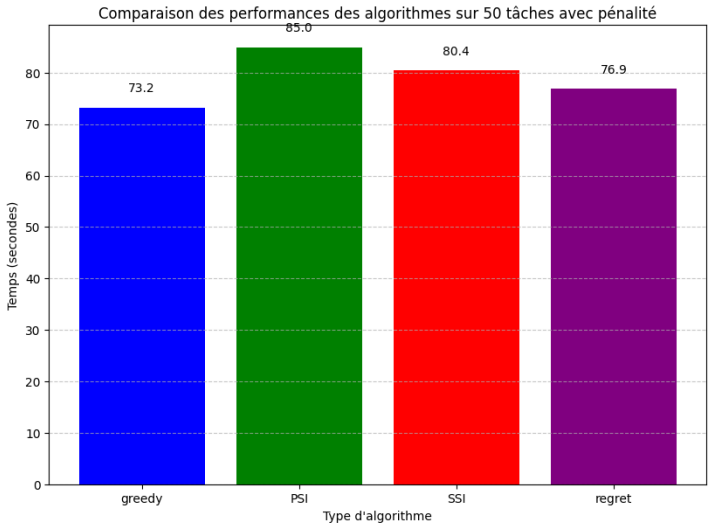

FIGURE 3 – Comparaison du temps nécessaire pour les pour accomplir 50 tâches, avec pénalité.

Paramètres du fichier `config.py` utilisés pour générer les deux histogrammes ci-dessus :

- **NUM_TAXIS** = 3
- **TASK_INTERVAL** = 5000 (5 sec)

— **NUM_TASKS_SPAWN** = 5
— **TAXI_SPEED** = 200

On observe que, en général, GREEDY termine plus rapidement. Cependant, si trop de tâches sont présentes sur la carte, GREEDY devient inefficace, car il calcule l'ensemble des permutations, ce qui entraîne une complexité en $n!$ (où n est le nombre de tâches).

Ainsi, GREEDY est bien adapté pour un petit ensemble de tâches. On remarque également que SSI avec regret est plus performant que SSI sans regret en termes de temps d'exécution, tandis que PSI est le plus lent.

Avec pénalité, l'ordre de performance reste le même, mais les écarts de temps sont réduits.

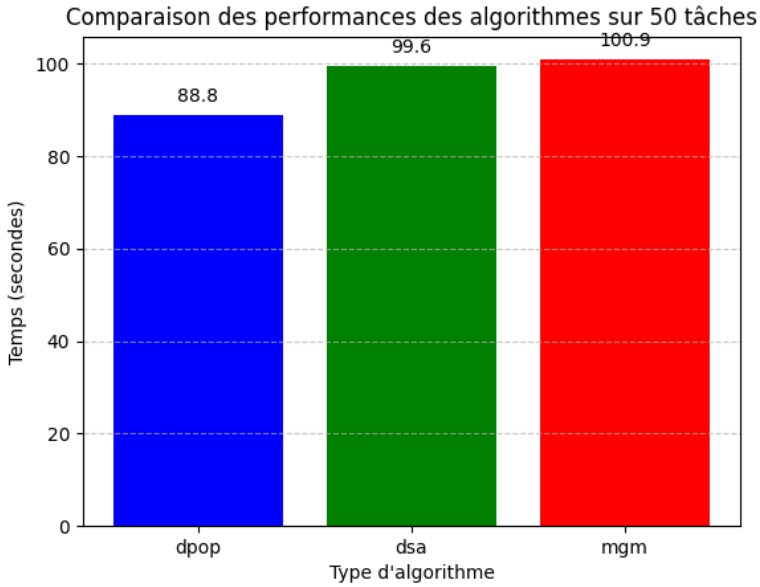

FIGURE 4 – Comparaison du temps nécessaire pour les pour accomplir 50 tâches, avec DCOP.

Paramètres du fichier `config.py` utilisés pour générer les deux histogrammes ci-dessus :

— **NUM_TAXIS** = 3
— **TASK_INTERVAL** = 8000 (8 sec)
— **NUM_TASKS_SPAWN** = 5
— **TAXI_SPEED** = 200

On constate que l'algorithme DPOP est plus rapide que DSA et MGM. Cependant, ces trois algorithmes restent globalement plus performants que PSI, SSI et SSI avec regret sans pénalité. Il convient toutefois de nuancer cette comparaison, car les algorithmes DCOP ont été exécutés avec un intervalle de 8 secondes entre les tâches.

Concernant la modification de cet intervalle entre les tests, elle résulte uniquement de contraintes liées à la simulation. En effet, avec un intervalle de 5 secondes, la simulation rencontrait des dysfonctionnements, car l'allocation des tâches n'avait pas suffisamment de temps pour s'exécuter avant l'apparition des tâches suivantes.

5.2 En fonction du coût moyen des routes

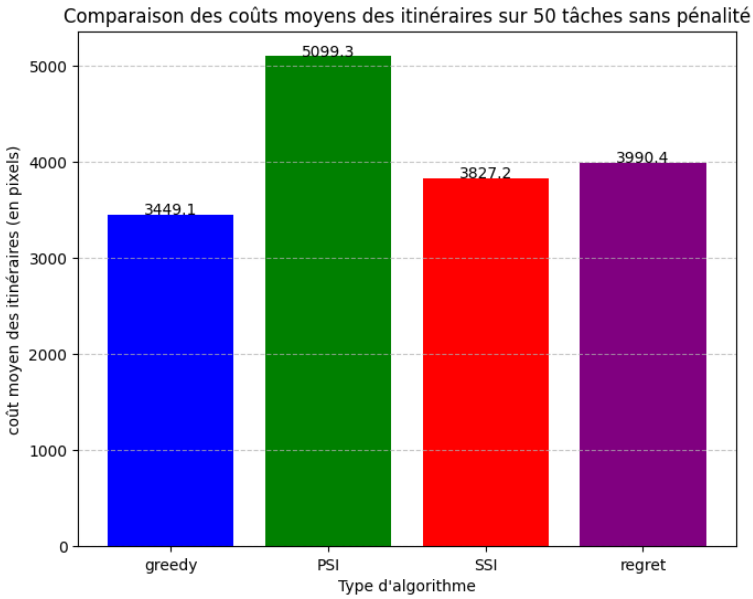

FIGURE 5 – Comparaison du coût moyen pour accomplir 50 tâches sans pénalité.

On fait maintenant la même execution mais avec pénalité sur la fonction *insertion_heuristic*, qui va influer seulement PSI, SSI, regret.

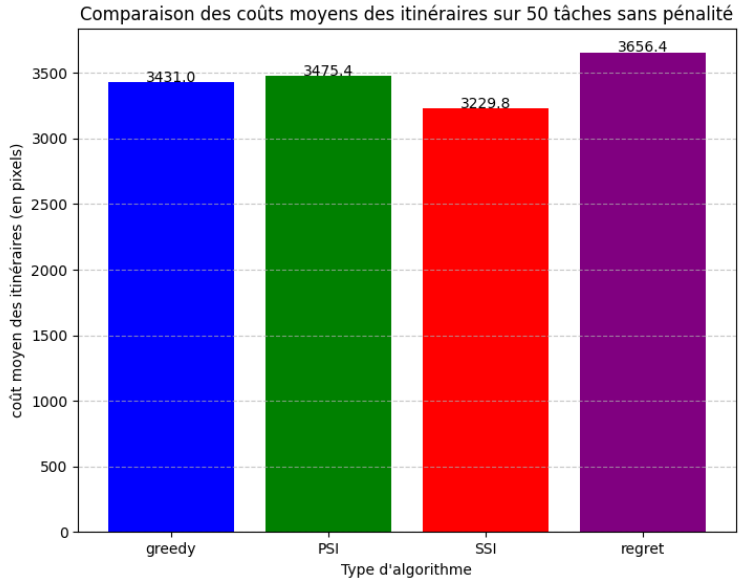

FIGURE 6 – Comparaison du coût moyen pour accomplir 50 tâches avec pénalité.

Paramètres du fichier `config.py` utilisés pour générer les deux histogrammes ci-dessus :

- **NUM_TAXIS** = 3
- **TASK_INTERVAL** = 5000 (5 sec)

— **NUM_TASKS_SPAWN** = 5
— **TAXI_SPEED** = 200

On observe que PSI est l'algorithme le plus coûteux en termes de taille moyenne du chemin sans pénalité.

GREEDY offre les meilleures performances en l'absence de pénalité.

Cependant, avec pénalité, SSI présente le coût moyen de chemin le plus faible, tandis que REGRET affiche le plus élevé.

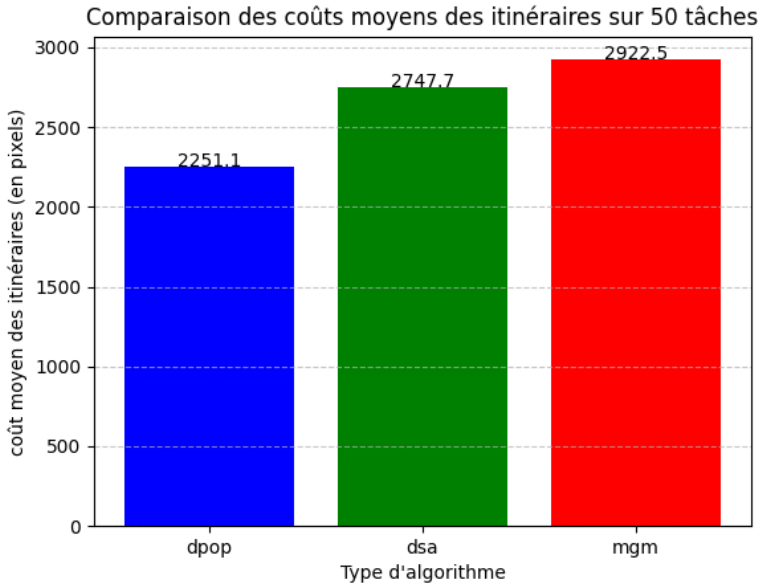

FIGURE 7 – Comparaison du coût moyen pour accomplir 50 tâches avec DCOP.

Paramètres du fichier `config.py` utilisés pour générer l'histogrammes ci-dessus :

— **NUM_TAXIS** = 3
— **TASK_INTERVAL** = 8000 (8 sec)
— **NUM_TASKS_SPAWN** = 5
— **TAXI_SPEED** = 200

Comme pour le temps, on observe une meilleure taille moyenne de chemin avec l'algorithme DPOP. De plus, on observe un plus grand écart entre DSA et MGM pour le coût moyen de la file d'attente. On peut donc en conclure que DSA trouve des meilleures solutions mais prend autant de temps que MGM.

6 Conclusion

Dans ce projet, nous avons étudié et implémenté différentes approches pour résoudre le problème d'allocation en ligne de tâches dans une flotte de taxis. En passant par une modélisation en programmation orientée objet, l'utilisation des DCOPs et l'implémentation de divers protocoles de coordination multi-agents, nous avons pu comparer les performances et les limites de chaque méthode.

Nos résultats ont montré que les méthodes basées sur les enchères offrent une solution rapide mais parfois sous-optimale en raison d'une répartition inégale des tâches entre taxis. L'ajout d'une pénalité proportionnelle au nombre de tâches en attente a permis de réduire cet effet négatif et d'améliorer l'équilibrage de la charge de travail. De plus, la modélisation DCOP a permis une allocation plus optimale des tâches, bien que plus coûteuse en termes de calcul.