03 FEVRIER 2025

INITIATION A L'INTELLIGENCE ARTIFICIELLE
SAE

Brenann JOLY

Farid MARI

Table des matières

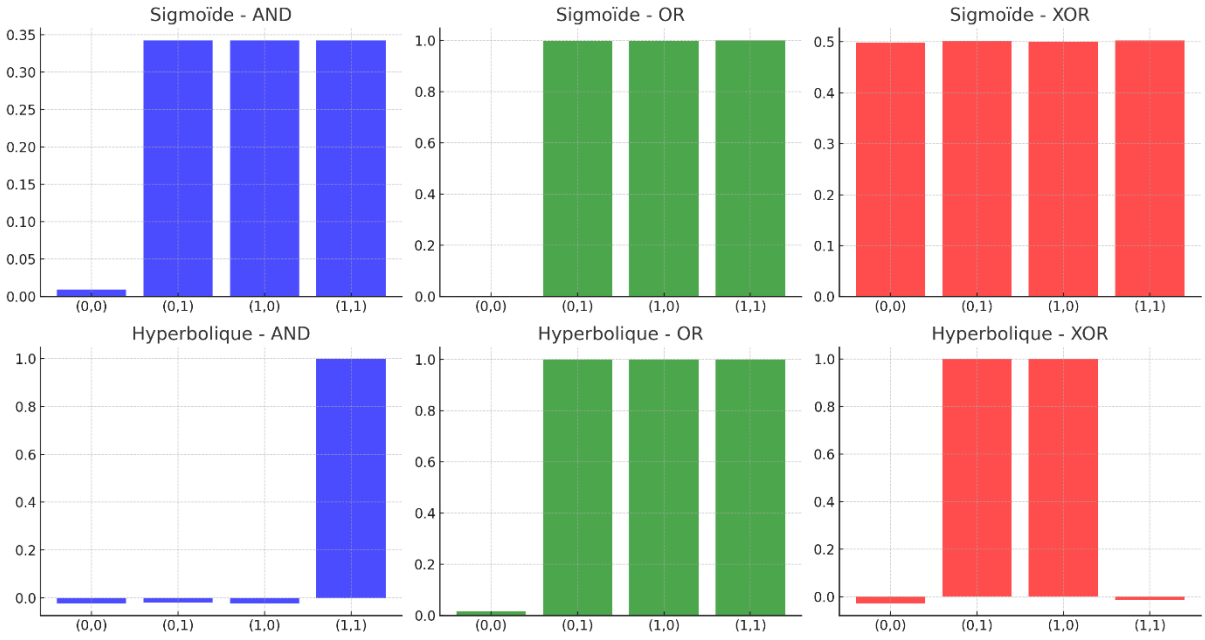
PARTIE 1
Approfondir les algorithmes

I. Perceptrons multi-couches

A. Un peu de programmation ?

Tableaux des résultats

Fonction d'activation	Entrée (x1, x2)	AND(x1, x2)	OR(x1, x2)	XOR(x1, x2)
Sigmoïde	(0, 0)	0.0092	0.0020	0.4979
Sigmoïde	(0, 1)	0.3422	0.9991	0.5017
Sigmoïde	(1, 0)	0.3422	0.9991	0.5006
Sigmoïde	(1, 1)	0.3422	1.0000	0.5026
Hyperbolique	(0, 0)	-0.0219	0.0158	-0.0274
Hyperbolique	(0, 1)	-0.0210	0.9999	0.9999
Hyperbolique	(1, 0)	-0.0230	0.9999	0.9999
Hyperbolique	(1, 1)	0.9992	0.9999	-0.0130



Les résultats des tests montrent les sorties du perceptron multi-couches en fonction des différentes fonctions d'activation (sigmoïde et tangente hyperbolique) pour les tables de vérité AND, OR et XOR.

Analyse des résultats :

1. **Fonction sigmoïde :**
    - Les sorties sont comprises entre 0 et 1.
    - Pour AND(0,0), la sortie est très proche de 0.
    - Pour OR(1,1), la sortie est très proche de 1.
    - XOR produit des valeurs autour de 0.5, indiquant que l'apprentissage peut ne pas être optimal.
2. **Fonction tangente hyperbolique :**
    - Les sorties sont comprises entre -1 et 1.
    - AND(0,0) donne une sortie négative très faible.
    - OR(1,1) est proche de 1.
    - XOR donne des valeurs positives ou négatives selon les entrées.

Observations :

- La sigmoïde a tendance à produire des valeurs centrées autour de 0.5 pour le XOR, ce qui peut être un problème pour un apprentissage correct.
- La tangente hyperbolique semble mieux séparer les classes avec des valeurs plus contrastées.
- XOR étant un problème non linéaire, il est souvent difficile à apprendre sans une architecture adaptée (nombre de couches, neurones, taux d'apprentissage).

B. Comparaison entre MLP et KNN

1) Classification des chiffres manuscrits

Résultats des tests :

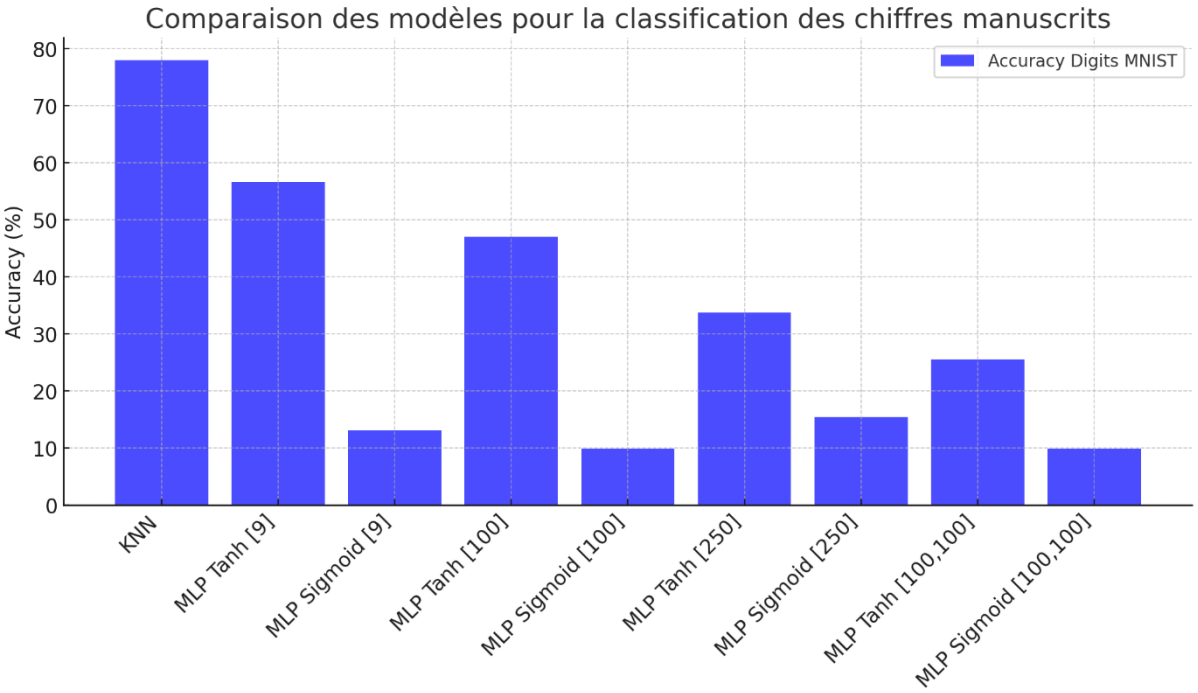
KNN

- Nombre d'images : **1000**
- Taux de réussite : **78%**
- Nombre de réussites : **789**
- Nombre d'échecs : **211**

MLP (Multi-Layer Perceptron)

Configuration	Fonction d'activation	Neurones cachés	Taux de réussite
1	tanh	[9]	56.6%
2	sigmoid	[9]	13.1%
3	tanh	[100]	47.1%
4	sigmoid	[100]	9.9%

5	tanh	[250]	33.8%
6	sigmoid	[250]	15.5%
7	tanh	[100, 100]	25.6%
8	sigmoid	[100, 100]	9.9%



Analyse des résultats :

- Le **KNN** surpasse globalement le **MLP** pour la reconnaissance des chiffres manuscrits.
- Concernant le MLP, la fonction **tanh** offre de meilleures performances que **sigmoid**.
- Une seule couche cachée avec **9 neurones** est la plus efficace pour **tanh**, tandis que **250 neurones** sont plus performants pour **sigmoid**.
- L'ajout d'une seconde couche cachée ne semble pas améliorer significativement les performances du MLP.

---

2) Classification des vêtements (Fashion MNIST)

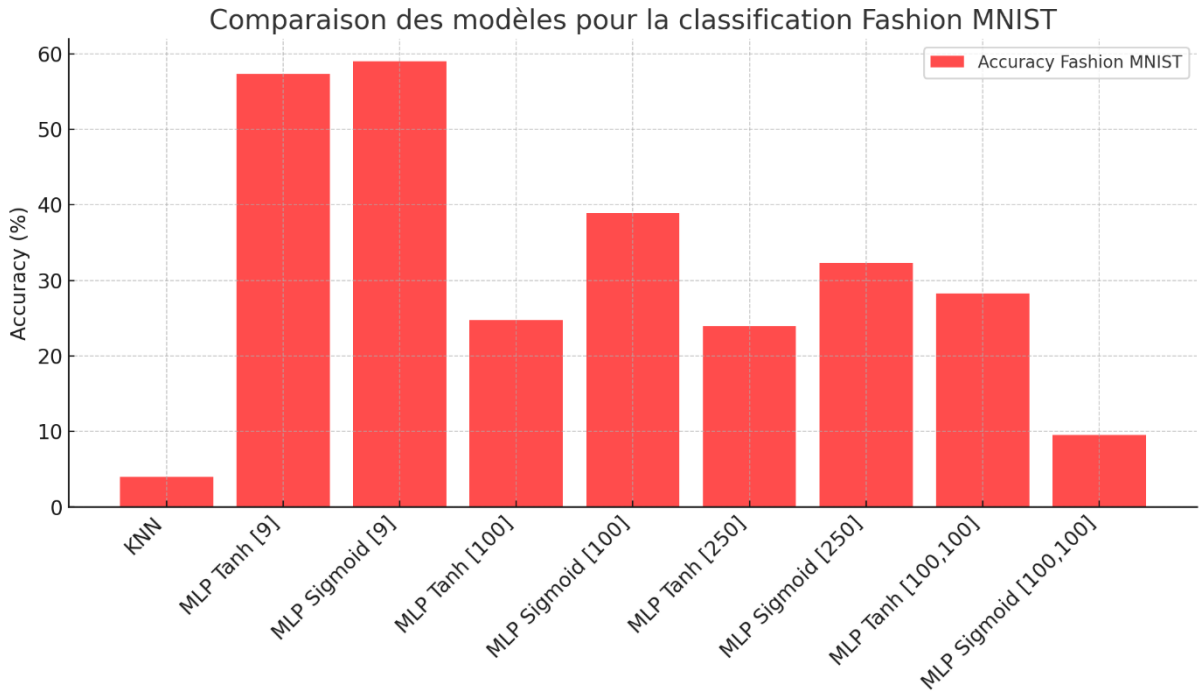
Résultats des tests :

KNN

- Nombre d'images : **1000**
- Taux de réussite : **4%**
- Nombre de réussites : **42**
- Nombre d'échecs : **958**

MLP (Multi-Layer Perceptron)

Configuration	Fonction d'activation	Neurones cachés	Taux de réussite
1	tanh	[9]	57.3%
2	sigmoid	[9]	59.0%
3	tanh	[100]	24.7%
4	sigmoid	[100]	38.9%
5	tanh	[250]	23.9%
6	sigmoid	[250]	32.3%
7	tanh	[100, 100]	28.3%
8	sigmoid	[100, 100]	9.5%



Analyse des résultats :

- Le **KNN** obtient un taux de réussite très faible (**4%**), ce qui s'explique par la ressemblance entre les différents vêtements.
- En revanche, le **MLP** offre des performances significativement meilleures.
- Contrairement aux chiffres manuscrits, la fonction **sigmoid** semble donner des résultats légèrement meilleurs que **tanh**.
- Une seule couche cachée avec **9 neurones** semble la plus efficace, indépendamment de la fonction d'activation.

Conclusion générale

- Pour la **classification des chiffres manuscrits**, le **KNN** est plus performant que le **MLP**.
- Pour la **classification des vêtements (Fashion MNIST)**, le **MLP** est nettement supérieur au **KNN**.
- La fonction **tanh** est généralement plus efficace que **sigmoid**, sauf pour le dataset Fashion MNIST où **sigmoid** semble légèrement mieux adapté.
- Le choix du nombre de neurones cachés a un impact significatif, mais l'ajout d'une seconde couche ne semble pas améliorer les performances.

Ainsi, le choix entre **KNN** et **MLP** dépend du type de données à classifier, chaque méthode ayant ses avantages et ses limites.

PARTIE 2
Approfondir les algorithmes

Introduction

Étudier les algorithmes de planification

Dans cette section, nous allons explorer et comparer les performances de plusieurs algorithmes de recherche en intelligence artificielle. Ces algorithmes jouent un rôle clé dans la résolution de problèmes de planification et de prise de décision dans des contextes variés et souvent complexes. L'objectif est d'évaluer leurs atouts, leurs limites et d'identifier les scénarios où ils s'avèrent les plus performants.

Algorithmes étudiés :

- **Recherche en largeur (BFS)** : Explore méthodiquement tous les nœuds d'un niveau avant de passer au suivant, assurant ainsi une recherche complète.
- **Recherche en profondeur (DFS)** : Se concentre sur un chemin en profondeur avant de revenir en arrière, ce qui peut être plus rapide mais parfois inefficace dans des graphes denses.
- **Recherche en coût uniforme (UCS)** : Tient compte des coûts des actions pour garantir une solution optimale.
- **Recherche gloutonne (GFS)** : Exploite une heuristique pour guider la recherche rapidement vers l'objectif, sans toujours garantir l'optimalité.
- **A*** : Combine le coût passé et une estimation heuristique du coût restant pour équilibrer rapidité et optimalité.

Méthodologie :

Pour comparer ces algorithmes, nous avons réalisé des tests sur des problèmes générés par la classe Dummy, avec des paramètres variés pour moduler leur complexité (taille du graphe et facteur de branchement). Les performances ont été évaluées selon trois critères principaux :

- **Nombre de nœuds explorés** : Indique l'efficacité de l'algorithme dans son exploration de l'espace des états.
- **Temps d'exécution** : Mesure la rapidité de l'algorithme pour parvenir à une solution.
- **Coût de la solution** : Évalue la qualité de la solution trouvée.

Comme demandé, les tests ont été menés sur des instances de problèmes simples (taille réduite, facteur de branchement faible) et des problèmes complexes (grands graphes, facteur de branchement élevé) afin de comparer leurs performances dans différents scénarios.

Objectifs de l'analyse :

1. Identifier les forces et faiblesses de chaque algorithme, en fonction de la nature du problème.
2. Mettre en évidence l'algorithme le plus adapté selon la complexité et les contraintes du problème.
3. Illustrer ces observations.

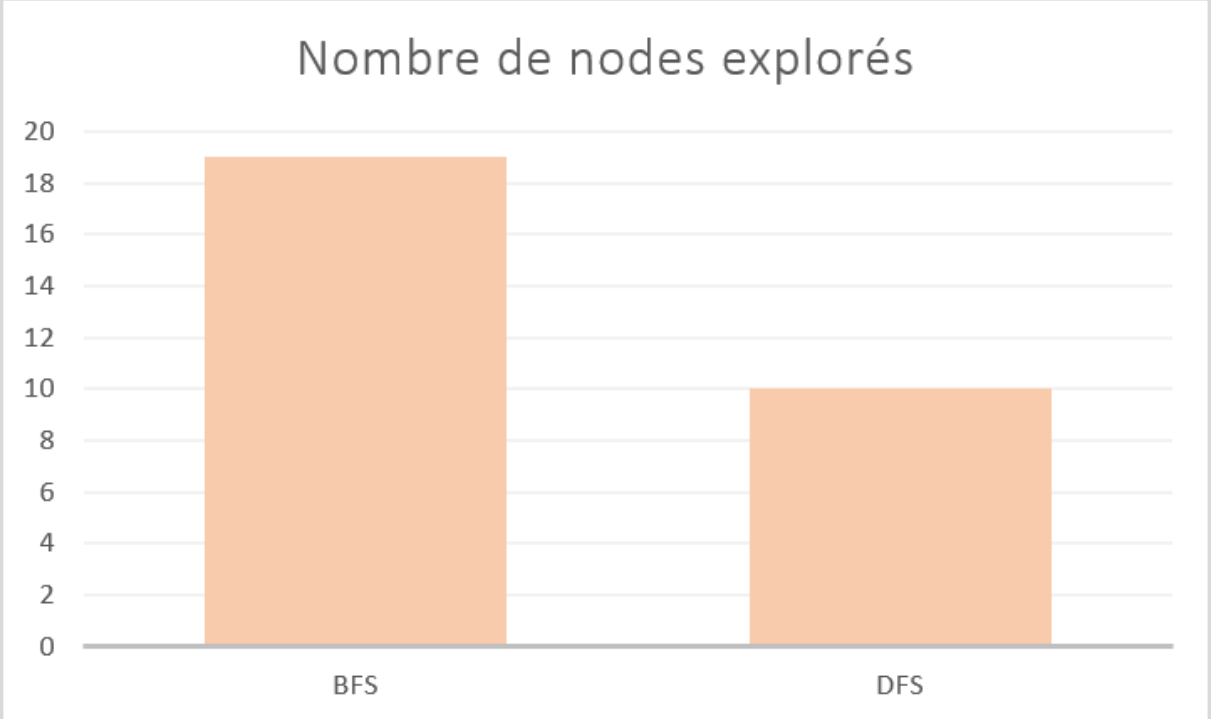
4. On va donc débuter par l'analyse des performances sur des problèmes simples avant de passer à des cas plus complexes. Cela nous permettra d'observer comment chaque algorithme s'adapte à l'augmentation des contraintes et de mieux comprendre leurs comportements respectifs.

1. BFS vs DFS

Problème vac

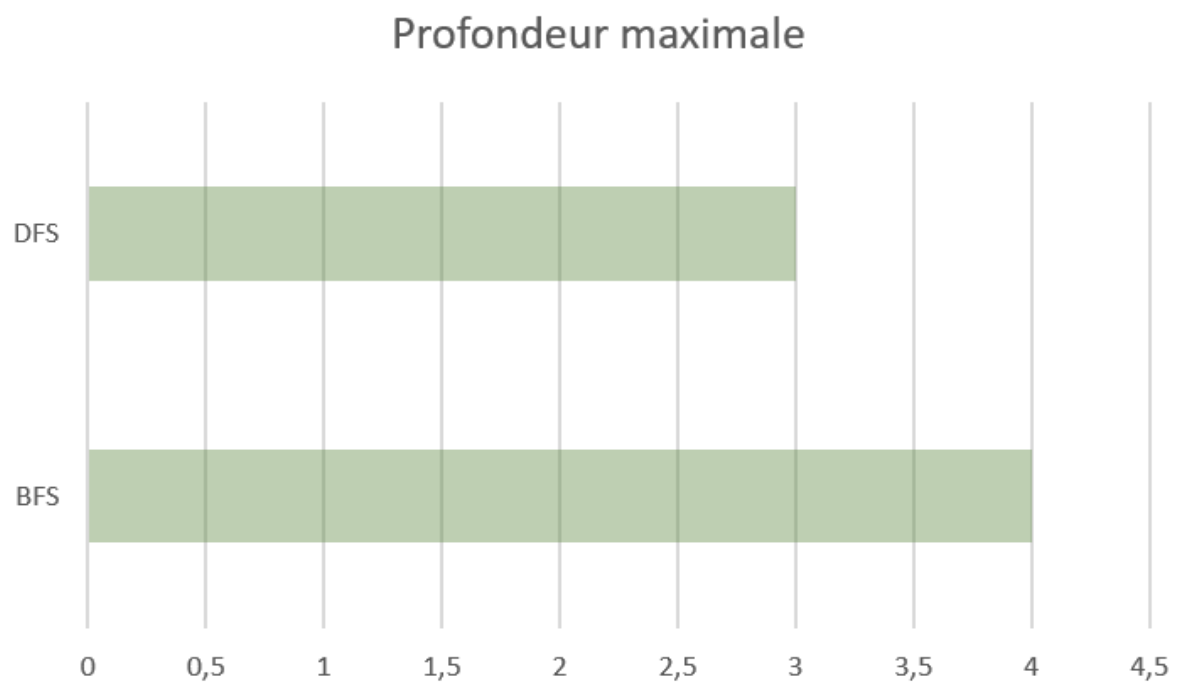
Pour débuter nos analyses on a choisi de comparer les algorithmes de **BFS** et de **DFS**. Ces deux approches constituent des stratégies de recherche assez fondamentales et sont différentes principalement dans leur manière d'explorer l'espace des états. BFS explore tous les nœuds d'un niveau avant de passer au suivant, tandis que DFS suit un chemin en profondeur jusqu'à ce qu'il atteigne une impasse, avant de revenir en arrière pour explorer d'autres options comme nous l'avons vu.

Nombre de nœuds explorés



Nous constatons que BFS explore **19 nœuds**, contre **10 nœuds** pour DFS. Cette différence s'explique par la manière dont chaque algorithme parcourt l'arbre de recherche.

Profondeur maximale



Nous constatons que **BFS** atteint une profondeur maximale de **4**, alors que que **DFS** s'arrête à une profondeur maximale de **3**.

Ca peut refléter l'approche systématique de BFS qui garantit de parcourir tous les niveaux de manière exhaustive, contre l'approche plus ciblée de DFS qui s'arrête dès qu'une solution est trouvée.

On peut donc en conclure que pour des problèmes simples, la différence de temps d'exécution entre BFS et DFS est assez marginale.

Problème vac : difficulté croissante

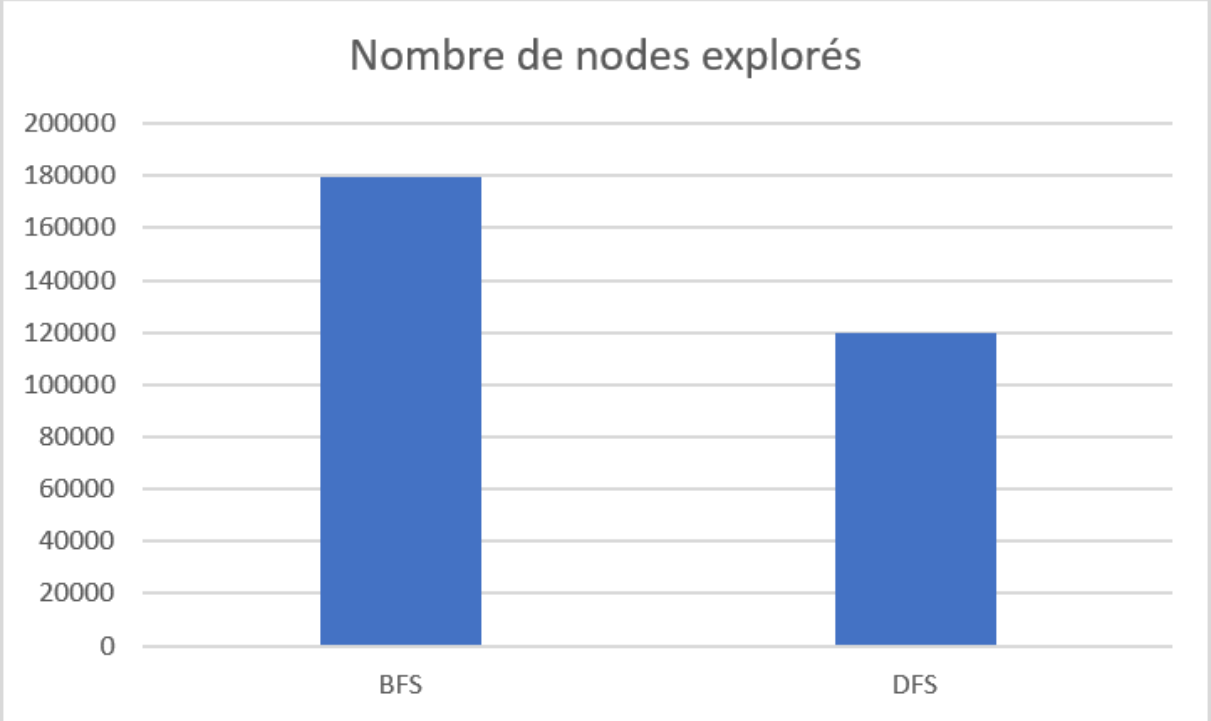
Nous avons testé BFS et DFS sur le problème vac avec des paramètres de difficulté croissante (augmentation de -n et -k). Cependant les résultats obtenus restent exactement les mêmes, quels que soient les paramètres. Le nombre de nœuds explorés, la profondeur maximale, le coût de la solution n'ont pas changé.

On peut d'ailleurs expliquer cela par la nature du problème vac. L'espace d'états est simple et linéaire, avec peu de possibilités d'augmentation de complexité.

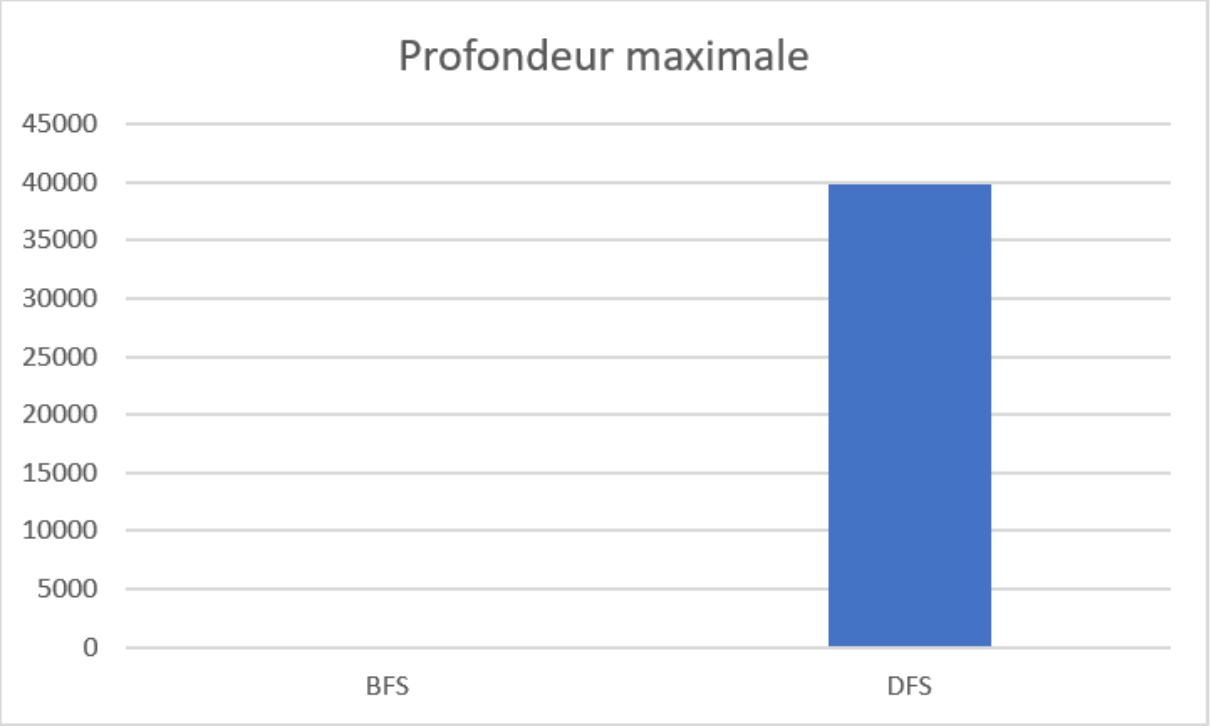
Problème puz

Nombre de nœuds explorés

Pour le problème puz, on observe une différence significative dans le nombre de nœuds explorés entre les algorithmes BFS et DFS :



Profondeur maximale



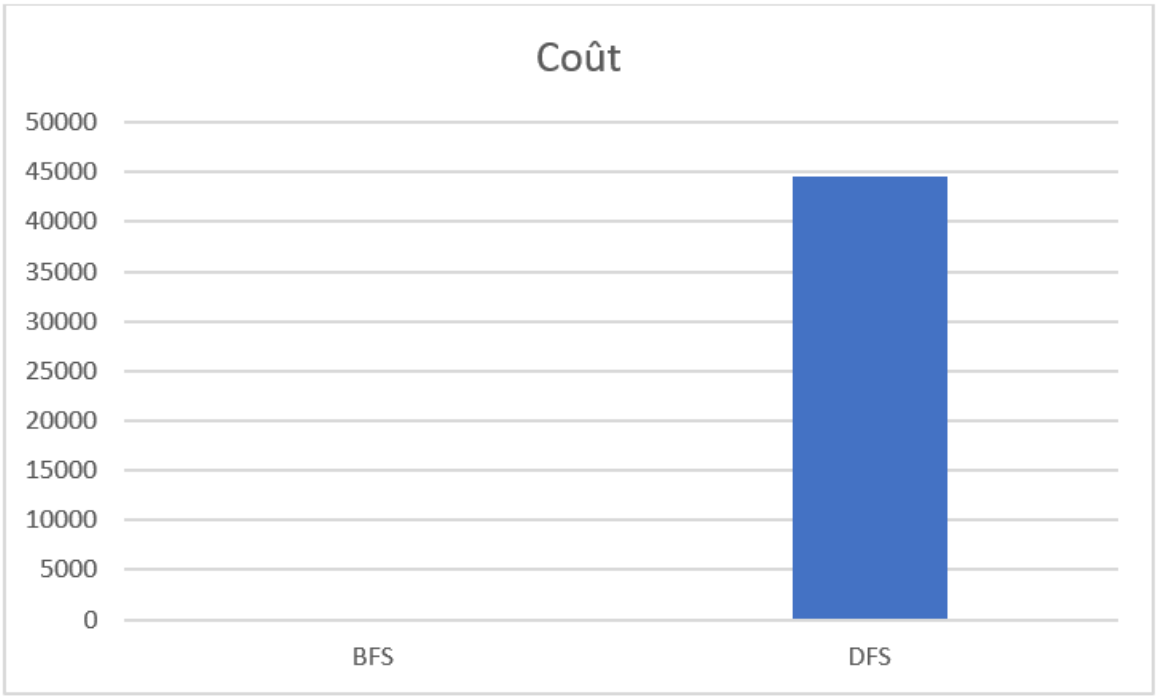
(BFS est à 22)

Ces résultats mettent en évidence une différence fondamentale dans la manière dont BFS et DFS explorent l'espace des états, en particulier en termes de **profondeur maximale atteinte** :

> **BFS (22)** : La profondeur maximale atteinte par BFS est de 22, car en effet cet algo explore tous les nœuds d'un niveau avant de passer au suivant mais une fois la solution trouvée il ne s'aventure pas inutilement plus loin dans l'arbre

DFS (39796) : En revanche DFS atteint une profondeur maximale de **39 796**, ce qui reflète sa stratégie d'exploration en profondeur. Donc dans ce cas, DFS a continué à explorer un chemin bien plus profond que nécessaire, car il ne revient en arrière qu'après avoir atteint un « mur » disons. Cela peut entraîner une exploration inefficace dans des problèmes où la solution se trouve à des niveaux pas trop profonds.

Coût de la solution



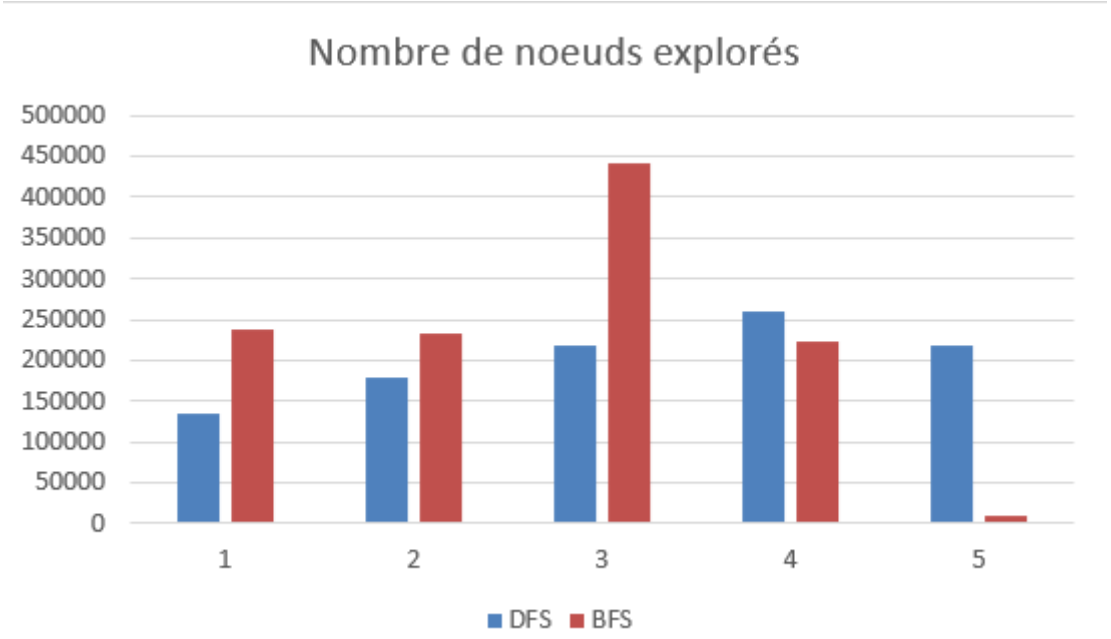
(BFS est à 23)

Ce résultat montre clairement que DFS n'est pas adapté pour trouver des solutions optimales en termes de coût, surtout dans des problèmes complexes où plusieurs chemins sont possibles. BFS, en revanche, garantit une solution optimale, bien que cela se fasse au prix d'un plus grand nombre de nœuds explorés.

Problème puz : difficulté croissante

Nous avons exécuté les algorithmes **BFS** et **DFS** sur le problème **puz**, en augmentant progressivement la difficulté avec **n = 100** et **k = 3**. Voici les résultats obtenus :

Nombre de nœuds explorés

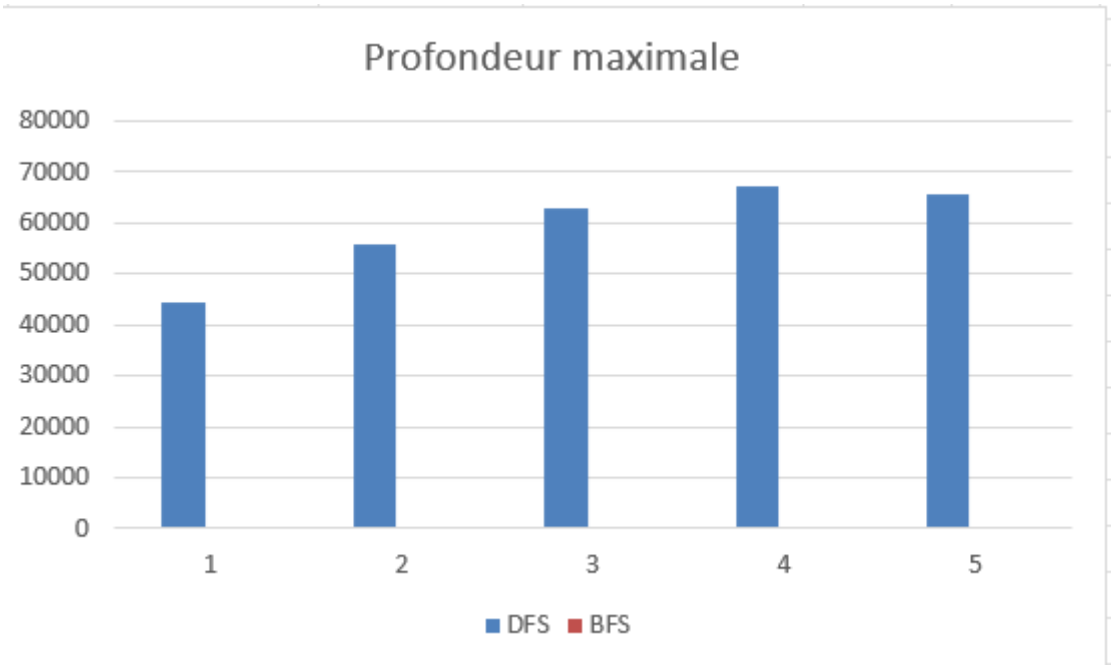


Nous constatons que DFS explore généralement **moins de nœuds** que BFS, sauf pour certains essais où BFS affiche un nombre de nœuds plus faible. Cela s'explique par les stratégies d'exploration de chacun qu'on a mentionnées précédemment.

Ces résultats confirment que **DFS peut être plus économique en termes de nœuds explorés**, mais cela **se fait souvent au détriment d'autres facteurs**.

Profondeur maximale

En plus du nombre de nœuds explorés, nous avons mesuré la **profondeur maximale atteinte** par chaque algorithme sur le problème **puz** avec **n = 100** et **k = 3**. Voici les résultats obtenus :



DFS atteint des profondeurs **beaucoup plus élevées**, de l'ordre de **40 000 à 66 000**, alors que **BFS reste toujours sous 30**.

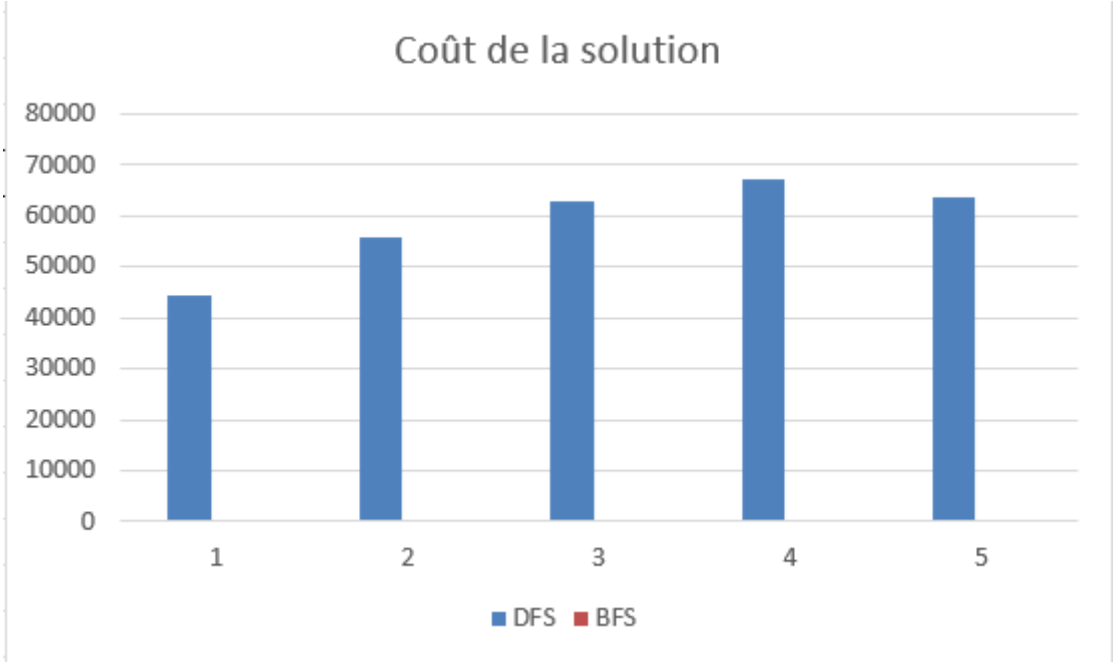
Finalement, si la solution se trouve à la surface, on voit que DFS est inefficace.

Coût de la solution

Le coût des solutions obtenues montre une **différence significative** entre **DFS** et **BFS**.

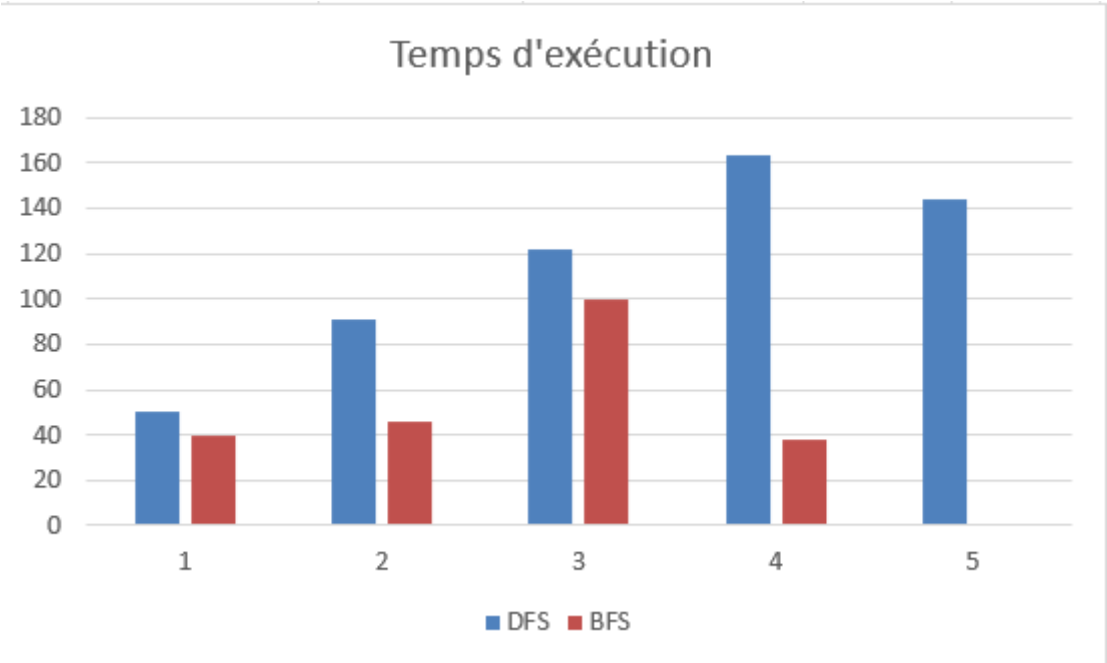
- **DFS** produit systématiquement des solutions avec un coût très élevé, dépassant les **44 000** dans tous les cas. Cela s'explique par le fait qu'il explore un chemin en profondeur sans toujours garantir qu'il s'agit du plus optimal.
- **BFS**, en revanche, génère des solutions avec un **coût beaucoup plus faible** et plus variable. Dans certains cas, il obtient des valeurs proches de **0,16**, ce qui indique qu'il trouve des chemins bien plus courts.

Comme on peut le voir ici :



Temps d'exécution

Le **temps d'exécution** des algorithmes **DFS** et **BFS** montre des variations à analyser :



- **DFS** affiche des **temps globalement plus élevés** avec des valeurs allant de **50 à 163 secondes**.
- **BFS** quant à lui présente **une grande variabilité**, avec des temps allant de **0,16 à quasiment 100 secondes**.

On est quand même tombés sur une valeur intéressante, le **0,16s** pour BFS, qui indique un cas où l'algorithme a trouvé une solution très rapidement, sans aller bien loin. Ca souligne son efficacité dans certaines situations.

BFS vs UCS vs DFS

Nous allons comparer trois algorithmes de recherche, **BFS, UCS, et DFS** sur d'autres problèmes afin d'observer leurs performances et leurs différences en termes d'exploration et de coût de solution.

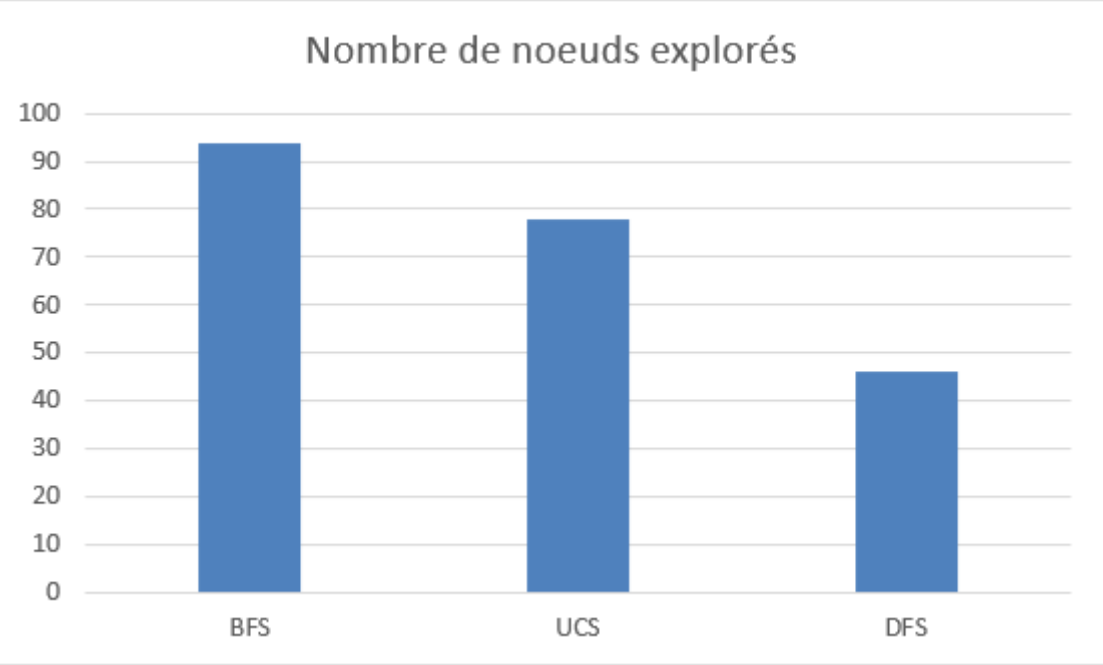
Nous allons nous focaliser sur **deux types de problèmes** :

- **Dum**.
- **Map**.

Les premiers résultats obtenus montrent une petite différence notable entre les trois algorithmes en termes de nombre de nœuds explorés :

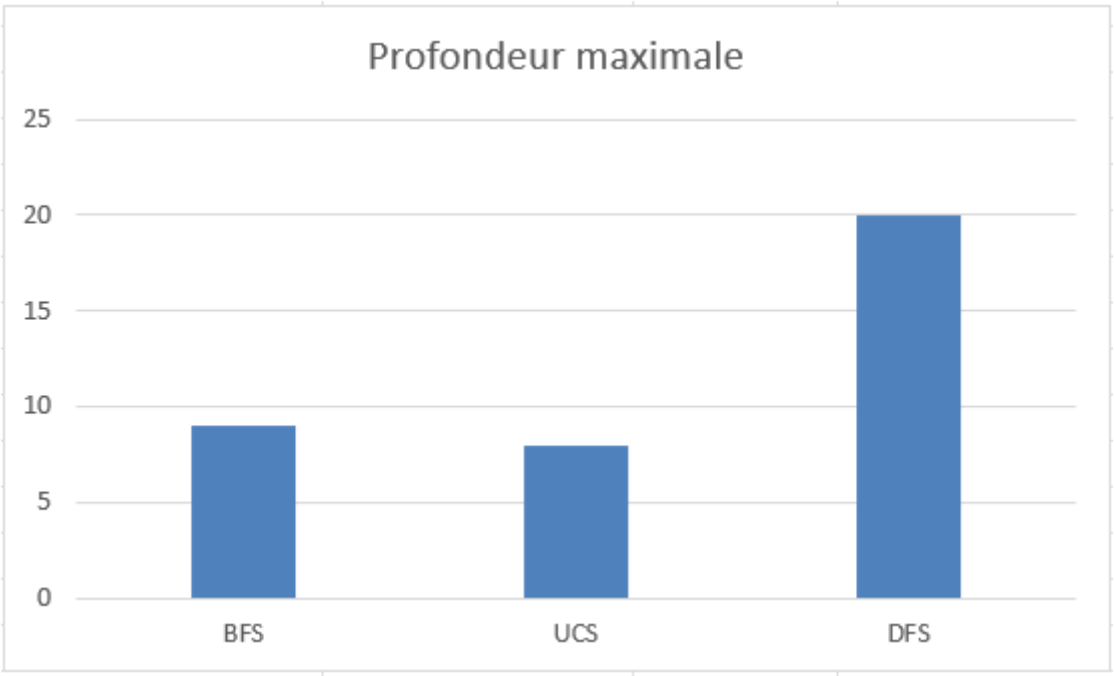
Problème dum

Nombre de nœuds

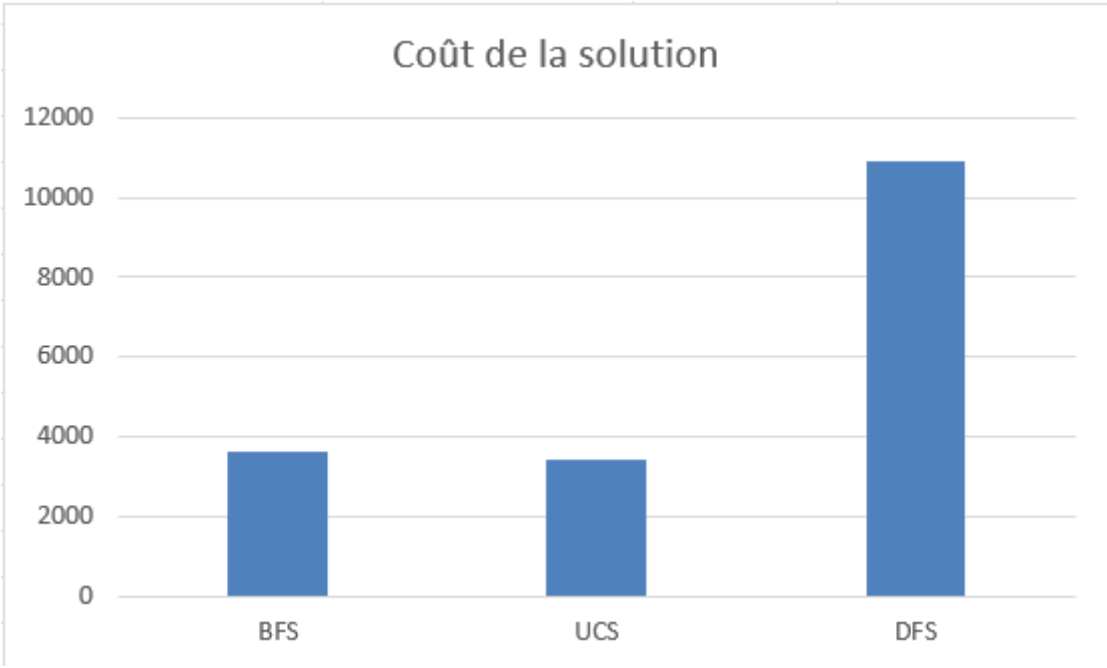


UCS explore plus de nœuds que DFS mais reste derrière BFS. Ce résultat est logique : UCS priorise le coût minimal, ce qui peut l'amener à examiner plus de chemins avant de trouver une solution optimale. BFS, en explorant tous les nœuds d'un niveau avant de descendre, il tend à générer plus d'explorations. Et puis on a DFS, qui à l'inverse explore moins de nœuds car il suit un seul chemin en profondeur avant de revenir en arrière si nécessaire. On verra si cette tendance reste telle qu'elle avec des problèmes plus complexes.

Profondeur maximale



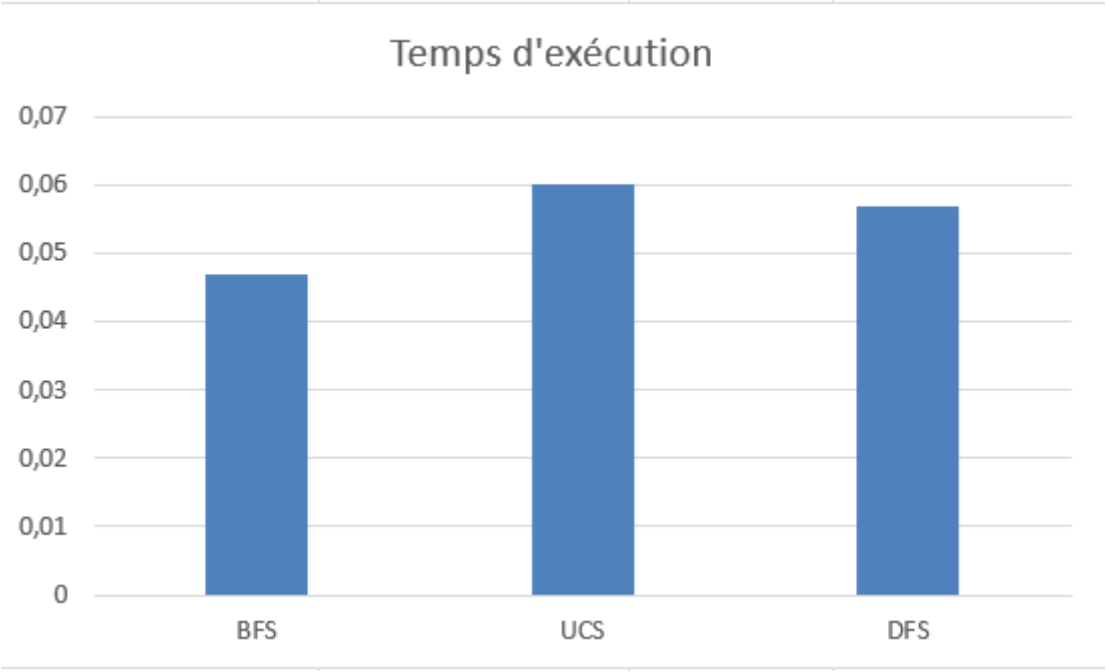
Coût de la solution



UCS obtient le coût le plus bas, ce qui est logique puisqu'il privilégie toujours le chemin le moins coûteux. BFS suit de près, tandis que DFS affiche un coût nettement plus élevé.

Temps d'exécution

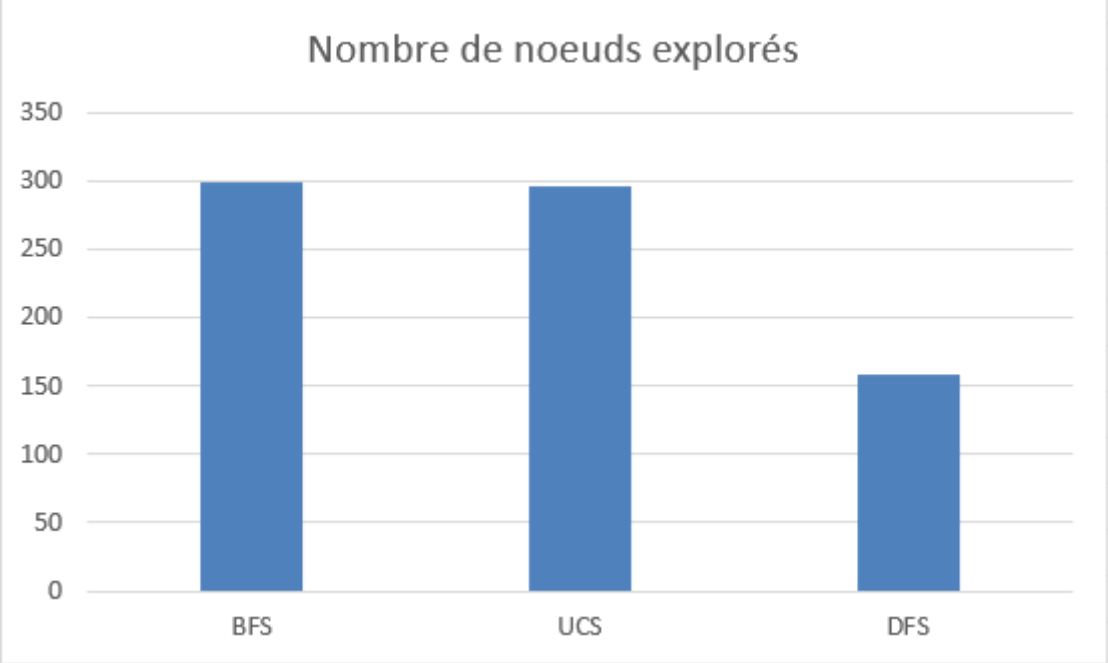
Des délais relativement courts et identiques pour ce probleme, sans difficulté.



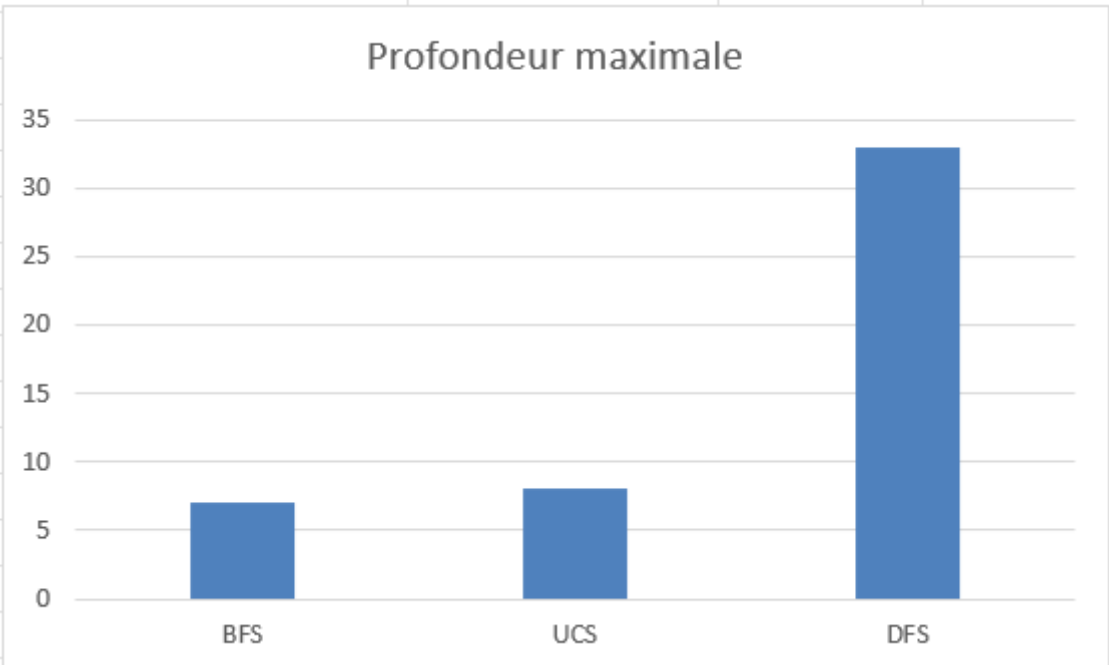
problème dum : difficulté croissante

Avec une difficulté croissante (n=200, k = 3), nous allons observer comment BFS, UCS et DFS se comportent sur le problème *dum*. L'objectif est de voir si les tendances précédemment observées se maintiennent et si l'écart entre les algorithmes se creuse à mesure que la complexité du problème augmente.

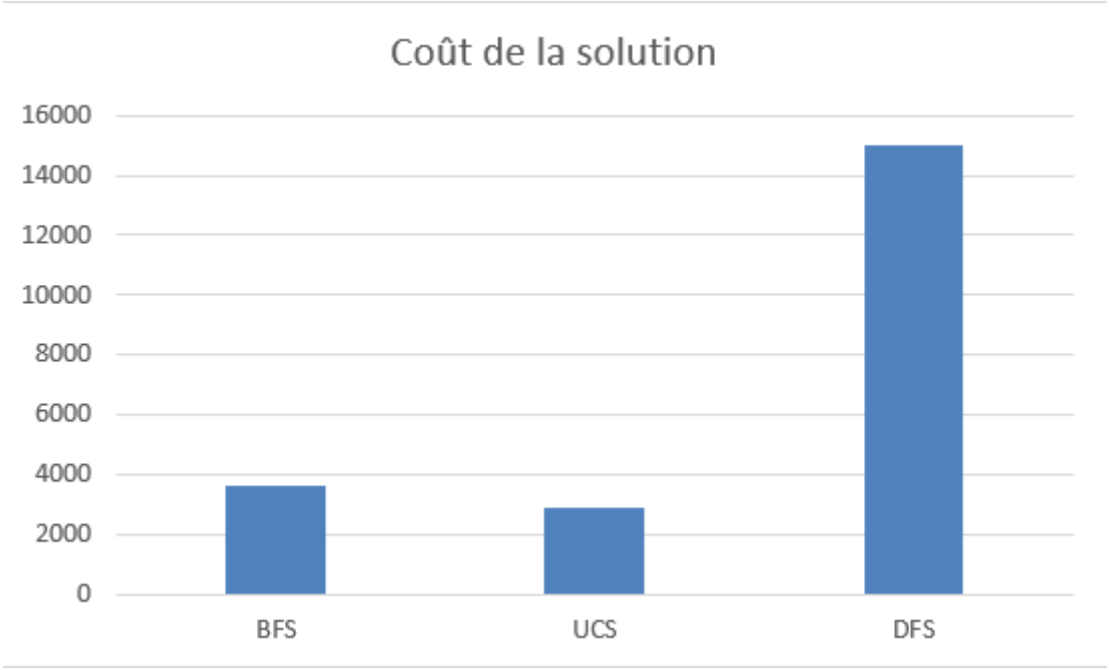
Nombre de nœuds



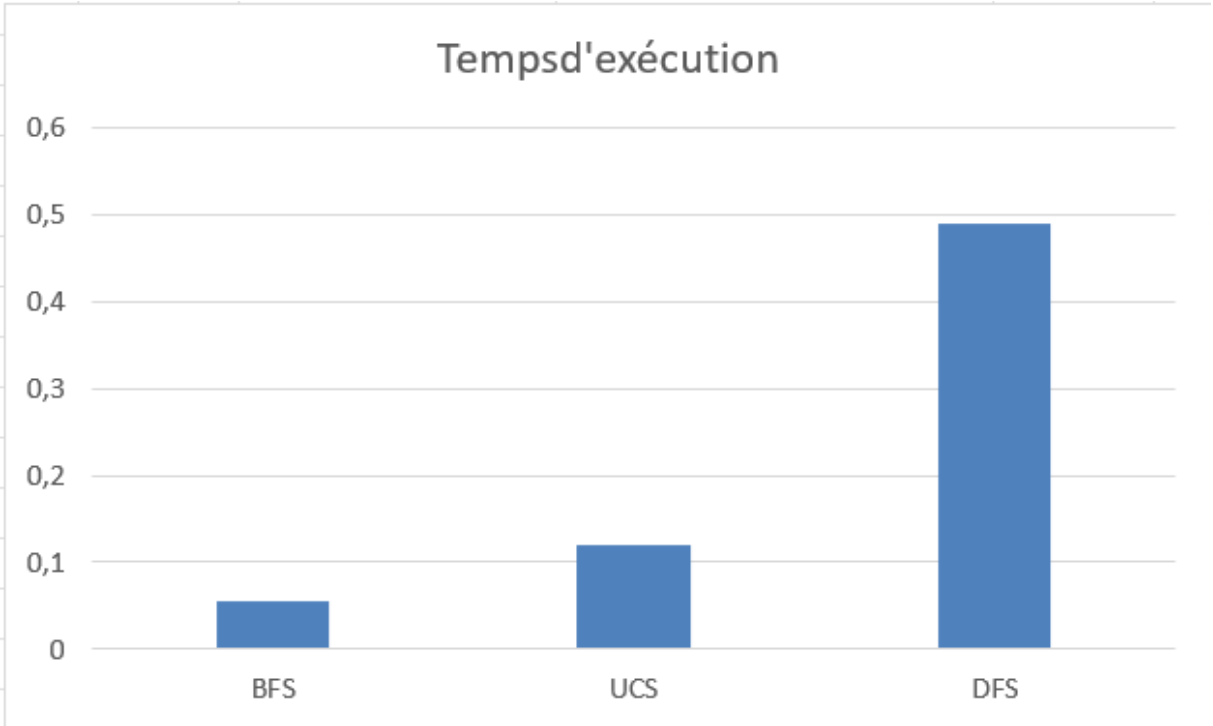
Profondeur maximale



Coût de la solution



Temps d'exécution

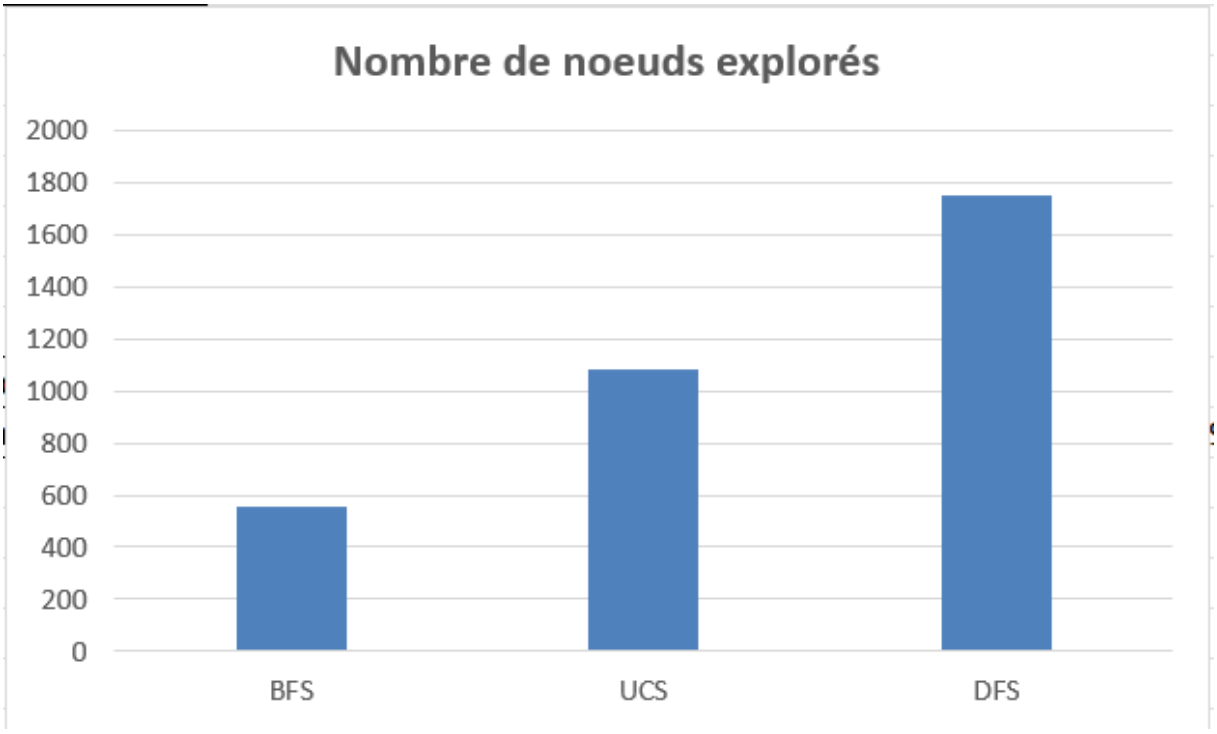


UCS apparaît comme **le meilleur choix** pour ce type de problème, car il trouve **une solution optimale tout en limitant le nombre de nœuds explorés et le coût**. BFS reste une approche sûre mais moins optimisée, tandis que DFS, bien qu'explorant moins de nœuds, est trop imprévisible et coûteux.

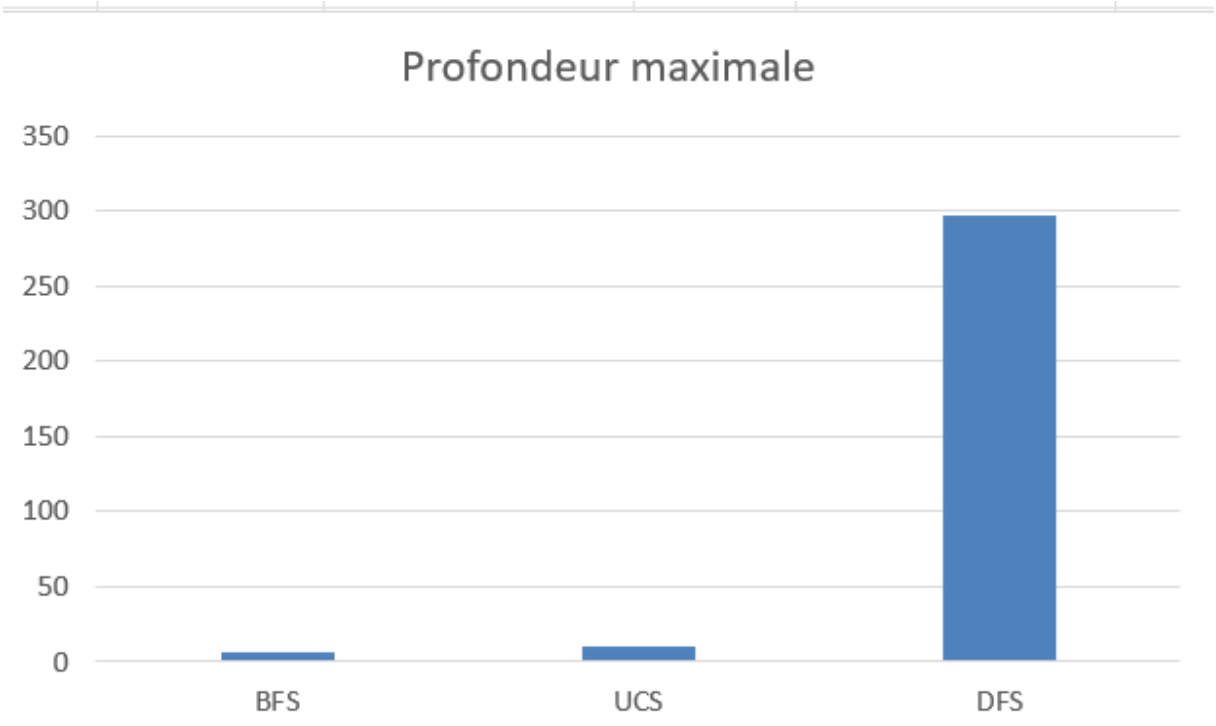
A présent, allons plus loin avec, cette fois-ci, n = 1000.

On obtient les résultats suivants :

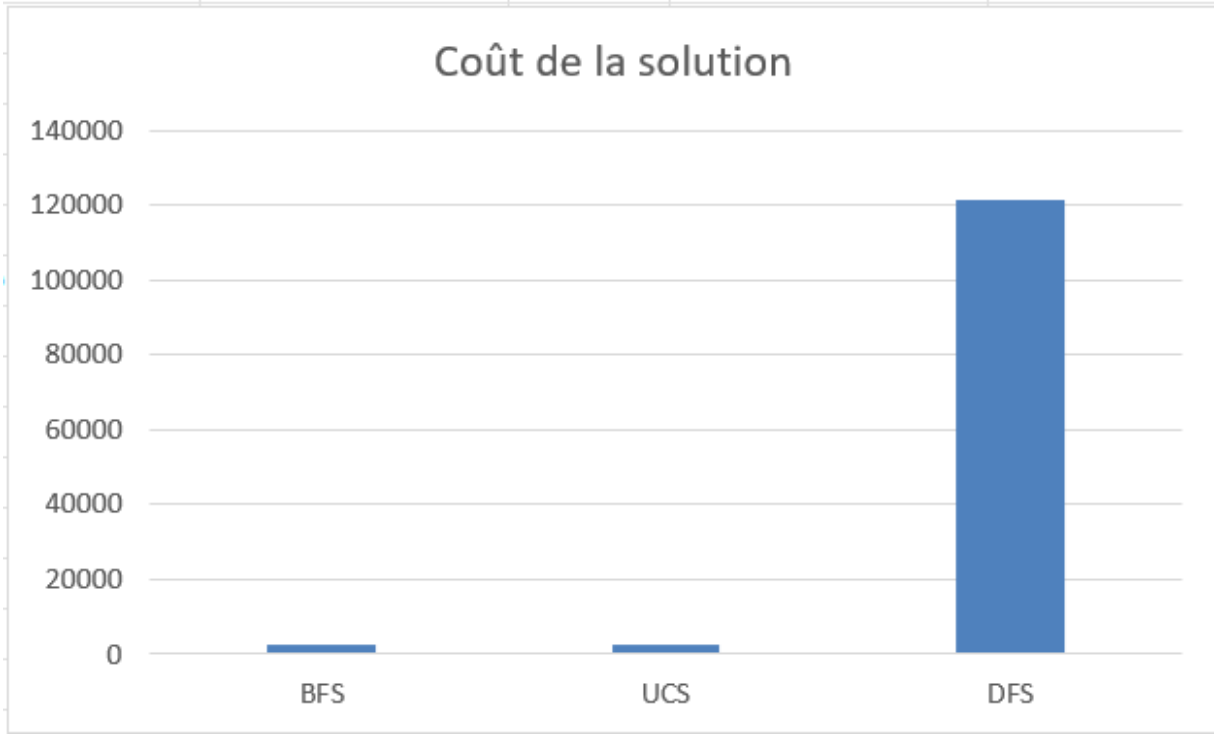
Nombre de nœuds



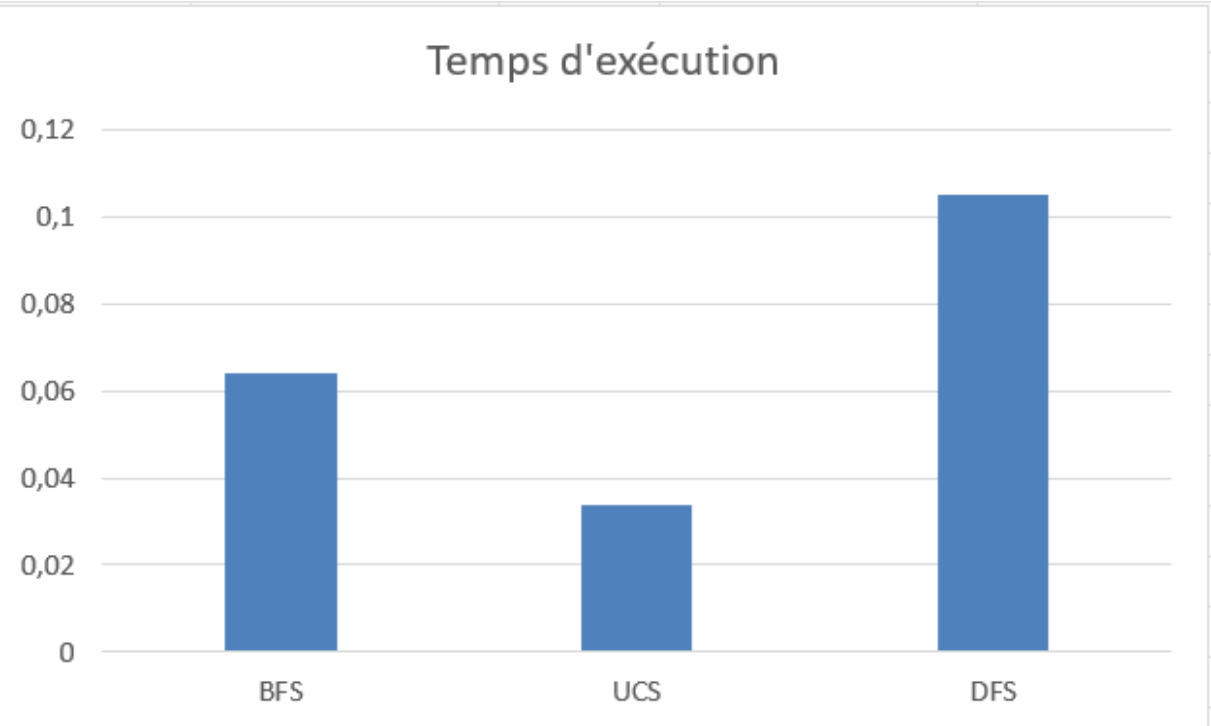
Profondeur maximale



Coût de la solution



Temps d’exécution



Avec une augmentation de la taille du problème (n=1000, k=3), on observe des tendances déjà aperçues auparavant :

BFS explore relativement peu de nœuds (554) et atteint une profondeur maximale limitée (6). Il trouve une solution de coût optimal (2466.57), mais met un peu plus de temps que UCS (0.064 sec).

UCS explore plus de nœuds (1085) mais atteint une profondeur plus importante (10). Il retrouve exactement le même coût que BFS, par contre il s’exécute plus rapidement (0.034 sec).

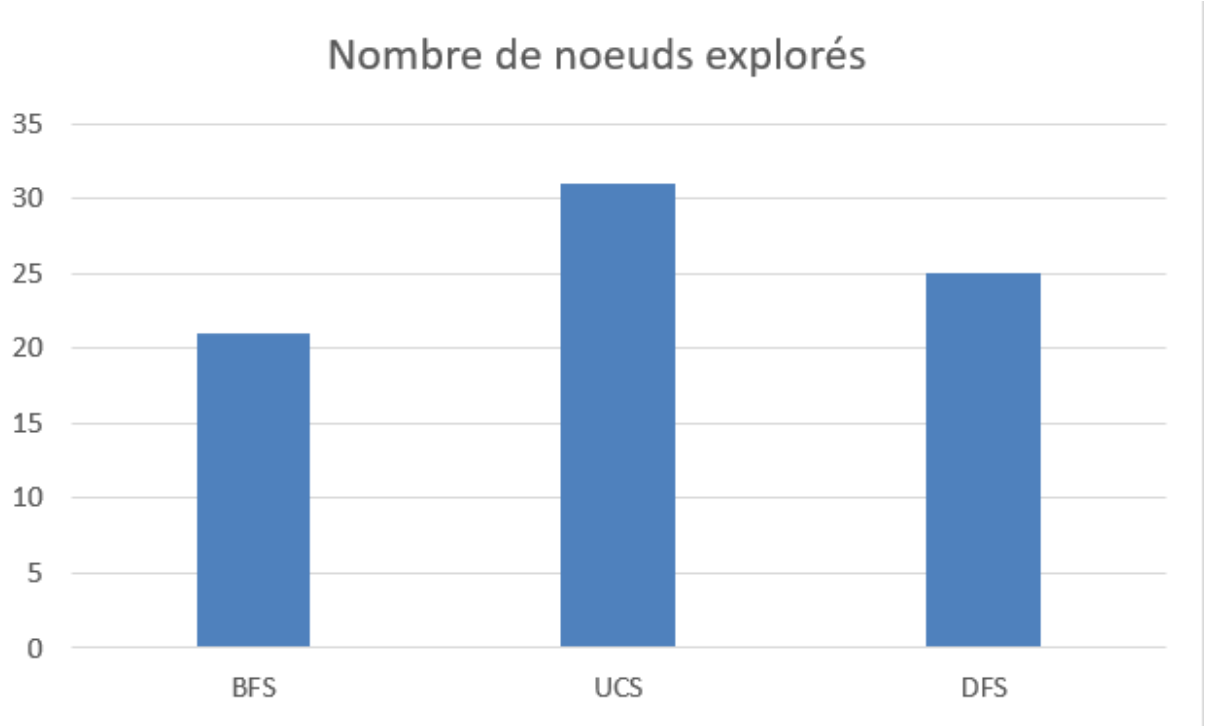
DFS explore beaucoup plus de nœuds (1745) et va beaucoup plus profondément (297), ce qui confirme son comportement exploratoire qui n’est pas optimal. Le coût de la solution trouvée est extrêmement élevé (121600.07). Il met aussi plus de temps à s'exécuter (0.105 sec).

En conclusion, BFS et UCS trouvent des solutions optimales avec un coût bien inférieur à DFS. UCS semble plus efficace en termes de temps et de profondeur explorée. DFS, lui, continue à explorer inutilement en profondeur, menant à une solution de mauvaise qualité.

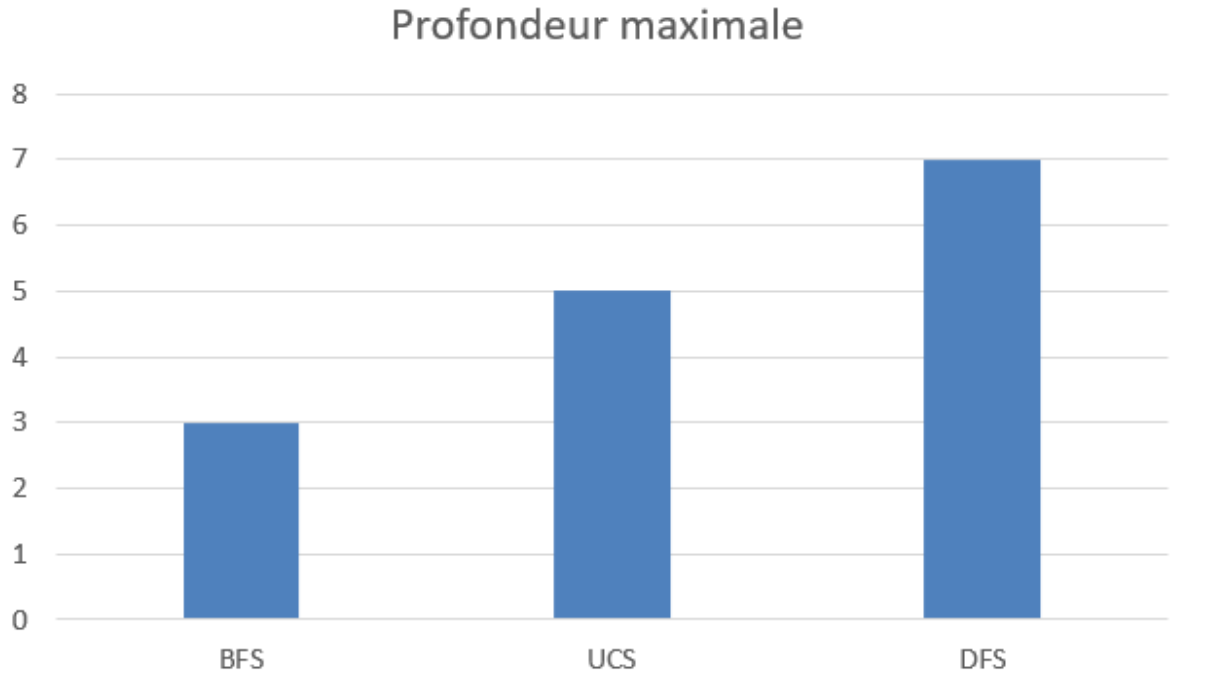
Problème map

Après avoir testé les algorithmes sur le problème dum, nous nous intéressons maintenant au problème map, qui représente un scénario de planification de trajets.

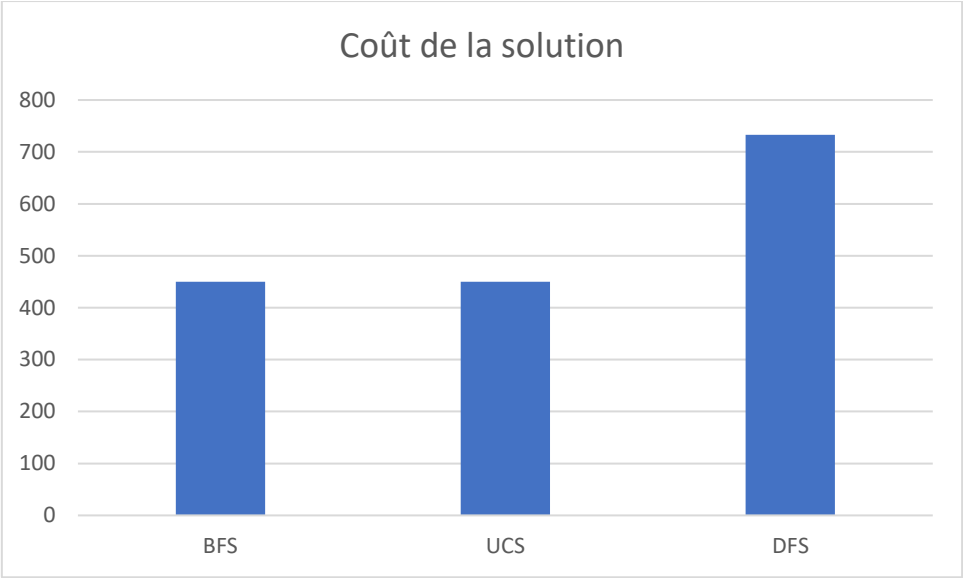
Nombre de nœuds



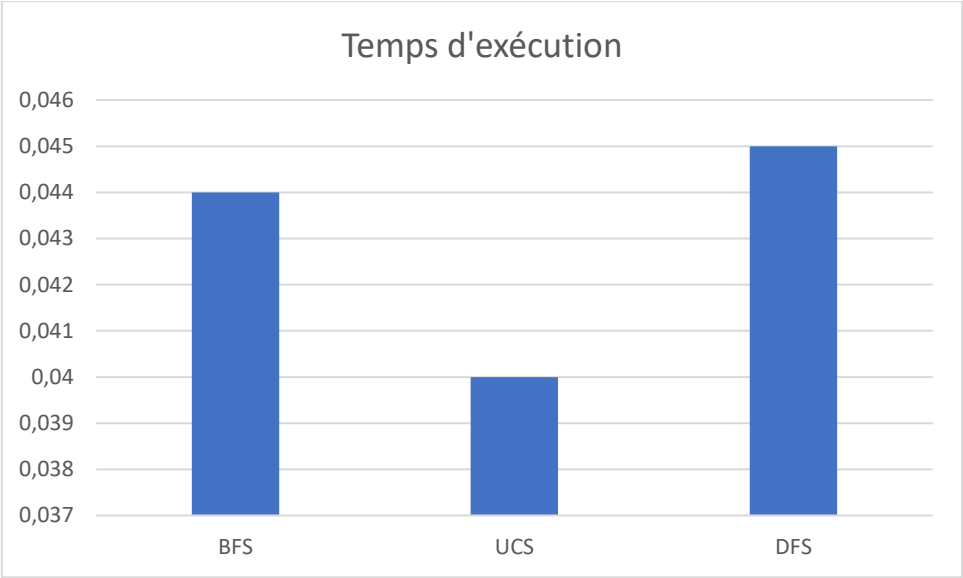
Profondeur maximale



Coût de la solution



Temps d'exécution



BFS explore **21 nœuds**, trouve une solution de **coût 450**, et s'exécute en **0.044 sec**. Il maintient une profondeur assez faible (**3**).

UCS explore **31 nœuds**, atteint une profondeur plus importante (**5**), mais retrouve le même **coût optimal (450)** que BFS. Il s'exécute bien plus rapidement (**0.004 sec**).

DFS explore **25 nœuds**, atteint une **profondeur de 7**, mais trouve une solution beaucoup plus coûteuse (**733**). Son exécution est légèrement plus lente (**0.045 sec**), et il montre encore une fois ses limites pour une solution optimale.

BFS vs GFS vs UCS

Après avoir analysé BFS, UCS et DFS sur les problèmes précédents, nous allons maintenant **introduire GFS** pour voir l'impact de l'utilisation d'une **heuristique** sur l'exploration et la qualité des solutions.

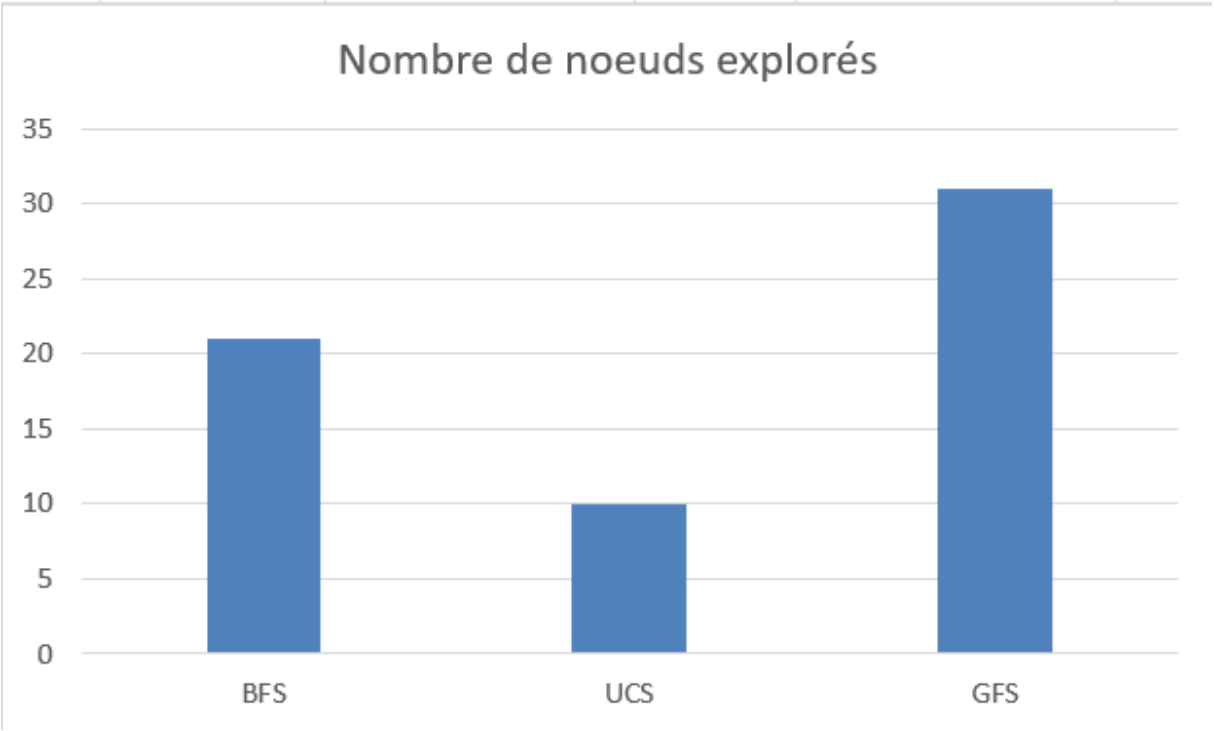
Nous allons tester ces trois algorithmes sur **deux types de problèmes** :

1. **map**
2. **puz**

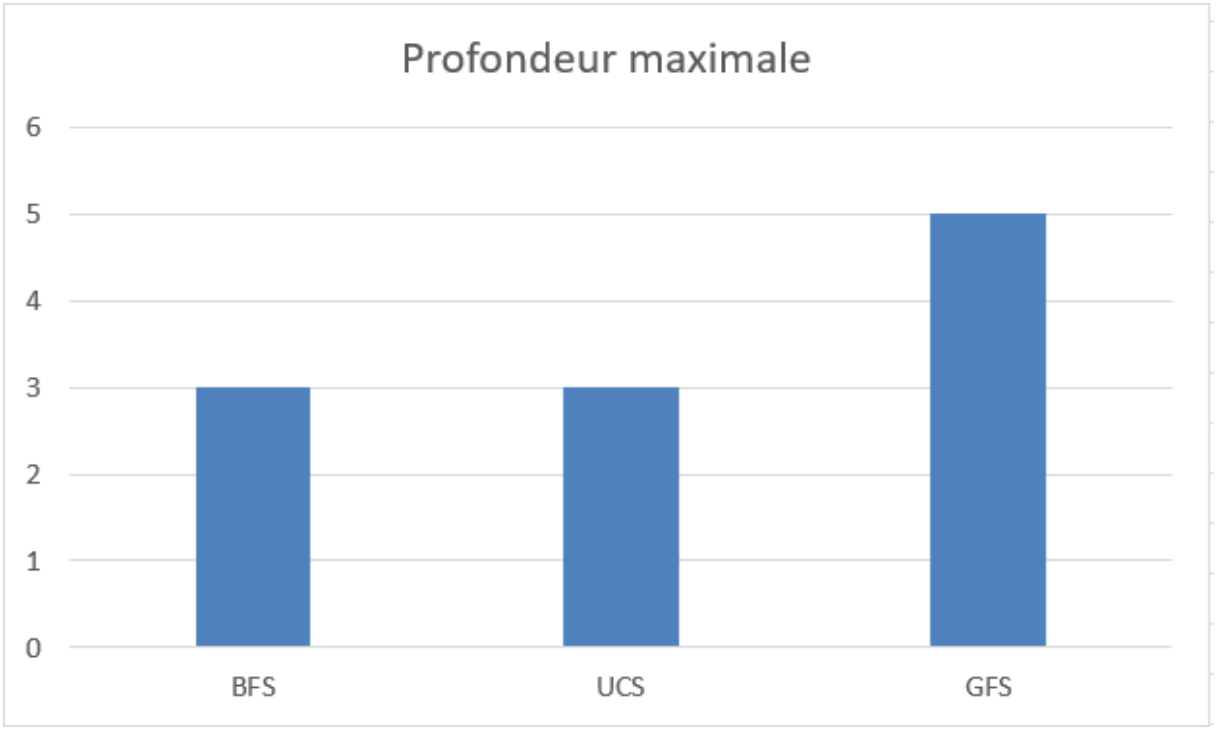
L'objectif est d'analyser si l'heuristique de **GFS** permet une exploration plus rapide que BFS et UCS, et si elle aboutit à des solutions de bonne qualité malgré l'absence de garantie d'optimalité.

Problème map

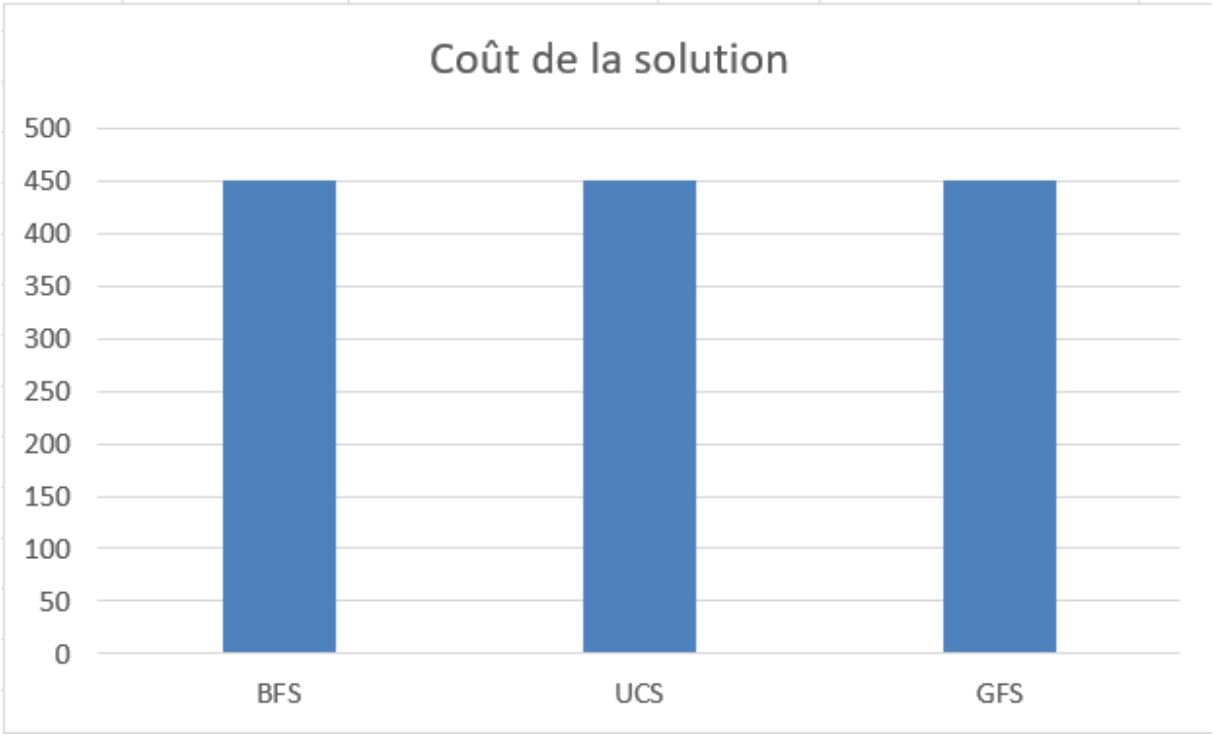
Nombre de nœuds



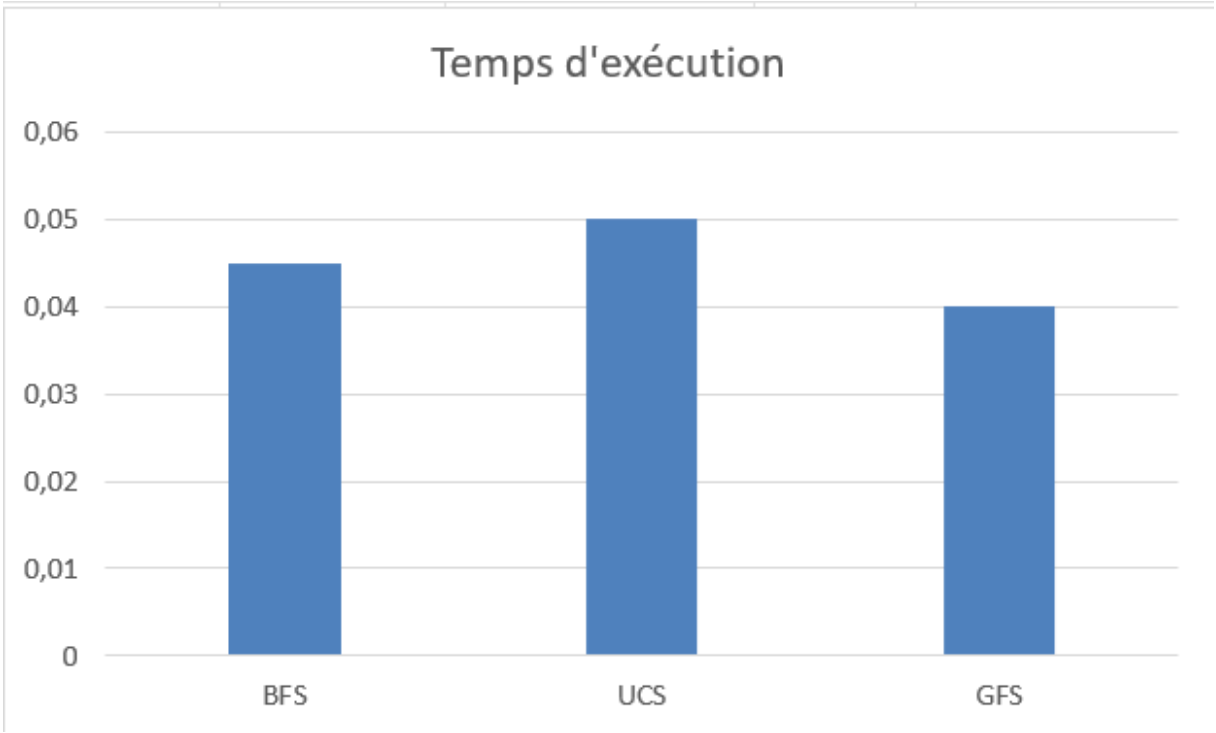
Profondeur maximale



Coût de la solution



Temps d’exécution



On observe ici que **BFS, GFS et UCS trouvent la même solution** avec un coût de **450.0**.

GFS explore seulement 10 nœuds, soit **moins de la moitié de BFS (21 nœuds)** et **trois fois moins que UCS (31 nœuds)**. Son approche heuristique lui permet donc de se focaliser plus rapidement sur la bonne direction.

BFS explore 21 nœuds, ça confirme son approche.

UCS explore 31 nœuds, ce qui peut sembler assez étonnant vu son objectif d'optimisation.

Temps d’exécution :

GFS explore **moins de nœuds** et offre un bon compromis rapidité/efficacité mais il n'est pas forcément **le plus rapide** en exécution.

UCS explore plus de nœuds mais il **est légèrement plus rapide** en temps d'exécution.

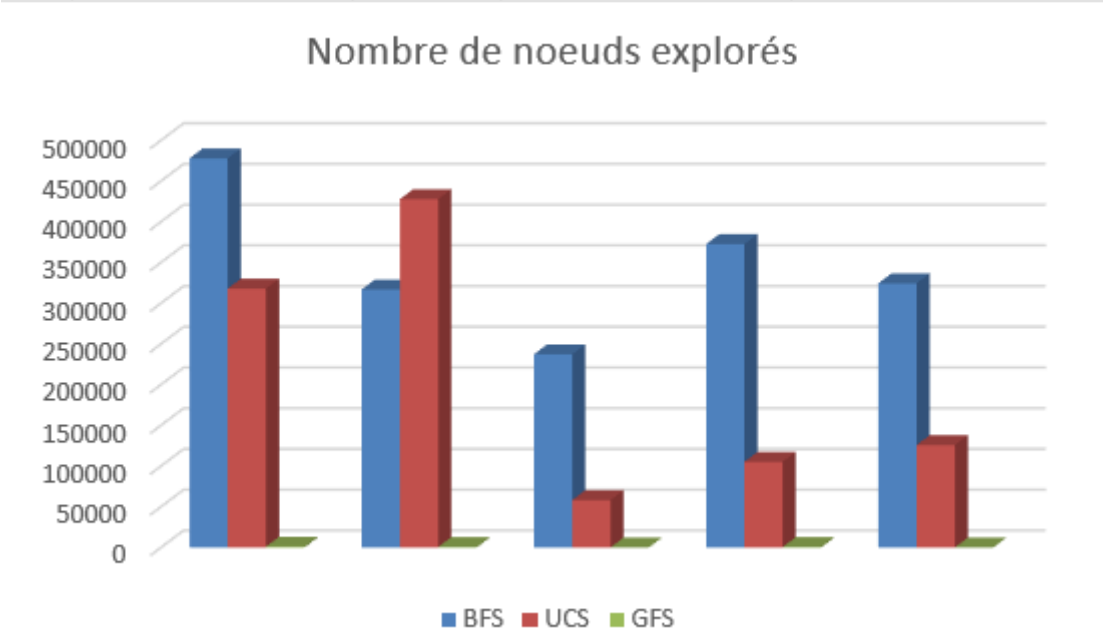
BFS est clairement **le plus lent** et explore **beaucoup plus de nœuds** inutilement.

Problème puz

On passe maintenant à l’évaluation des algorithmes sur le problème **puz**.

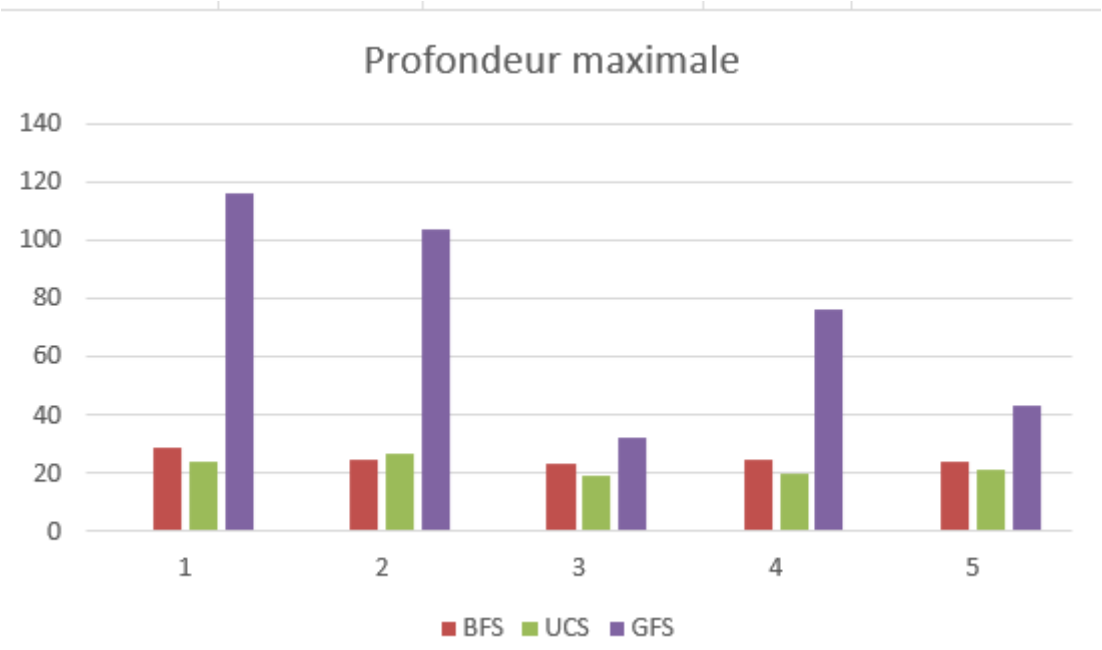
Ce problème étant plus complexe que **map**, on va voir **si l'heuristique de GFS lui permet d'explorer moins de nœuds** tout en trouvant une solution efficace.

Nombre de nœuds



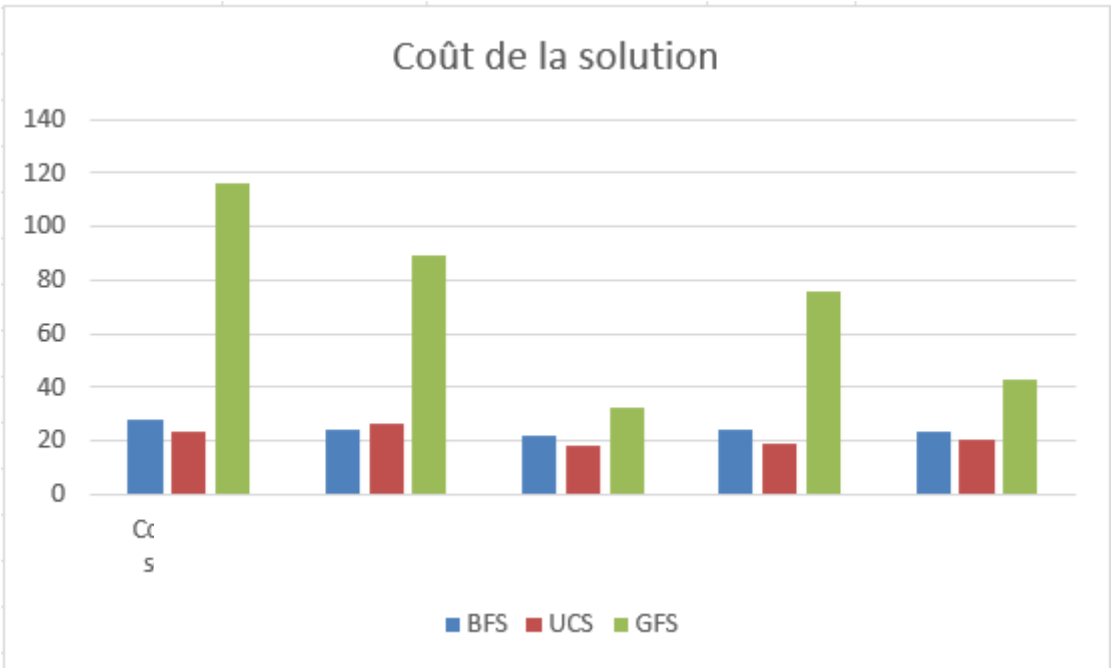
BFS et UCS explorent un très grand nombre de nœuds, avec UCS parfois dépassant BFS. UCS explore plus de nœuds dans certains cas, et GFS par contre explore un nombre de nœuds très réduit.

Profondeur maximale



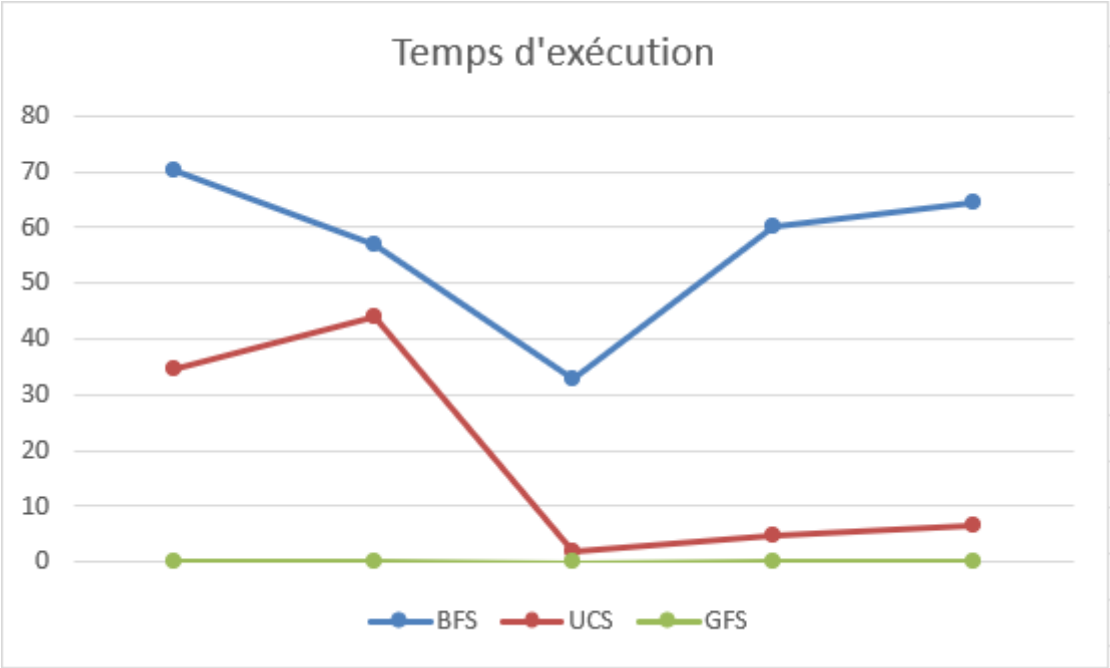
La profondeur maximale atteinte varie selon l'algorithme. **DFS et UCS restent dans des profondeurs raisonnables**, tandis que **GFS atteint parfois des valeurs très élevées**.

Coût de la solution



Le coût des solutions trouvées est systématiquement plus bas avec UCS, ce qui est logique. BFS donne des coûts proches mais légèrement plus élevés, mais GFS produit des solutions avec des coûts bien plus élevés. Il va vite, mais il peut choisir des chemins bien plus coûteux en ignorant la meilleure route.

Temps d'exécution



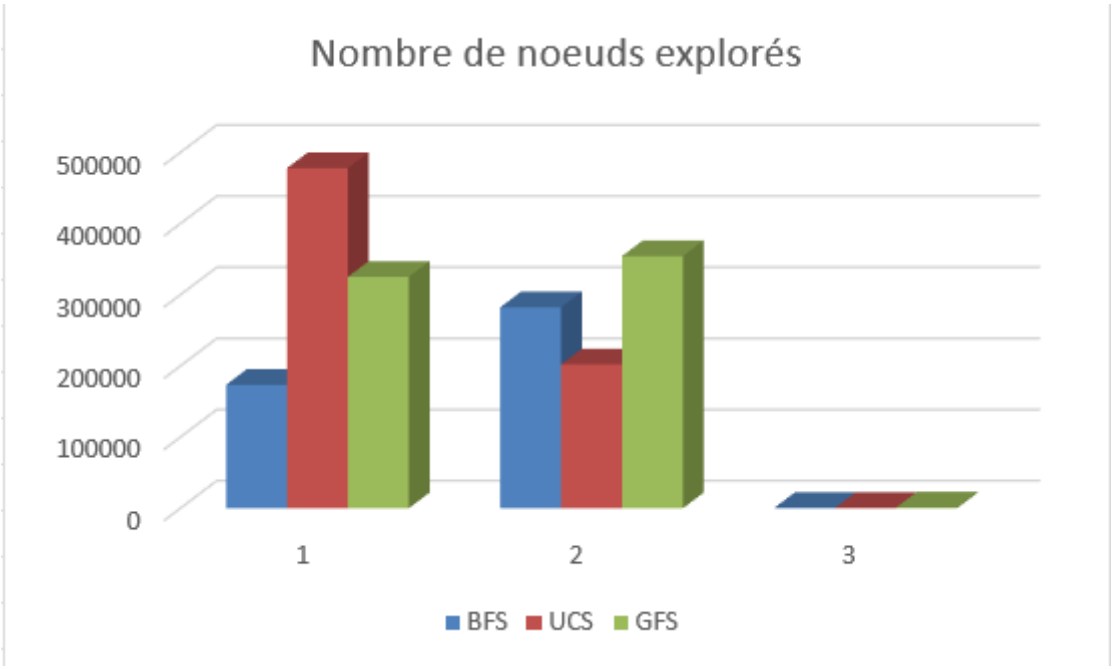
Le temps d'exécution reflète bien les tendances observées : BFS et UCS prennent beaucoup plus de temps que GFS, en raison du grand nombre de nœuds explorés. UCS est généralement plus rapide que BFS car il évite de traiter des choses inutiles, puis on a GFS qui est certes extrêmement rapide, mais cette rapidité se fait au détriment de l'optimalité de la solution.

En clair, GFS est un excellent choix pour des problèmes où on cherche juste une solution rapide, mais pas forcément optimale.

Problème puz : difficulté croissante

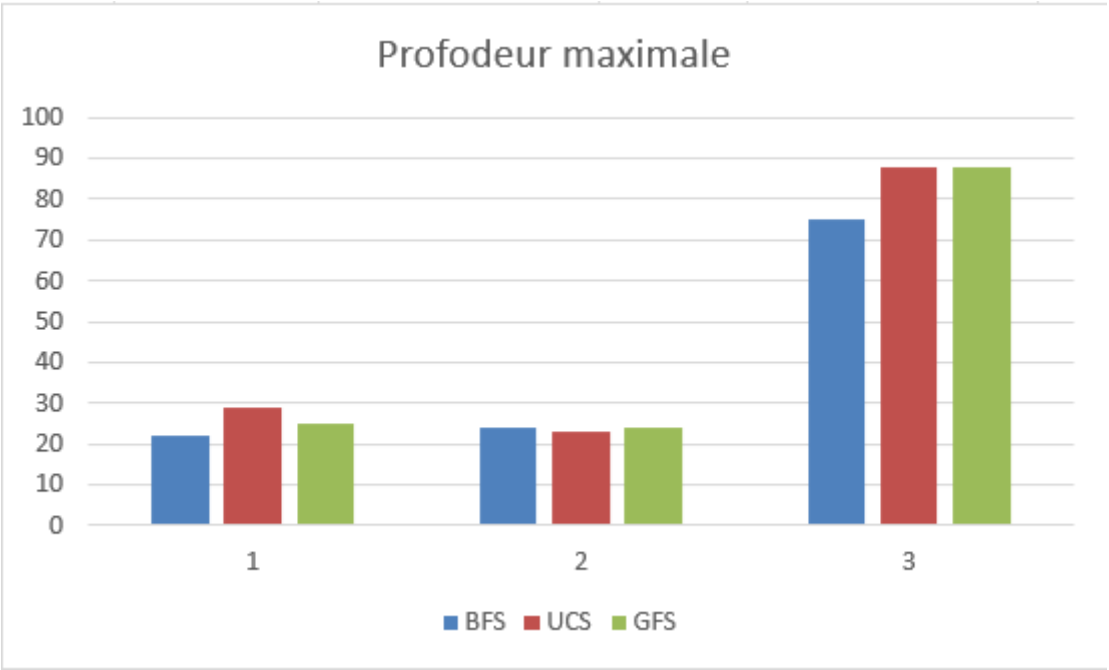
Nous allons maintenant **augmenter la difficulté** en passant à **n = 10 000 et k = 3**, sur 3 essais.

Nombre de nœuds



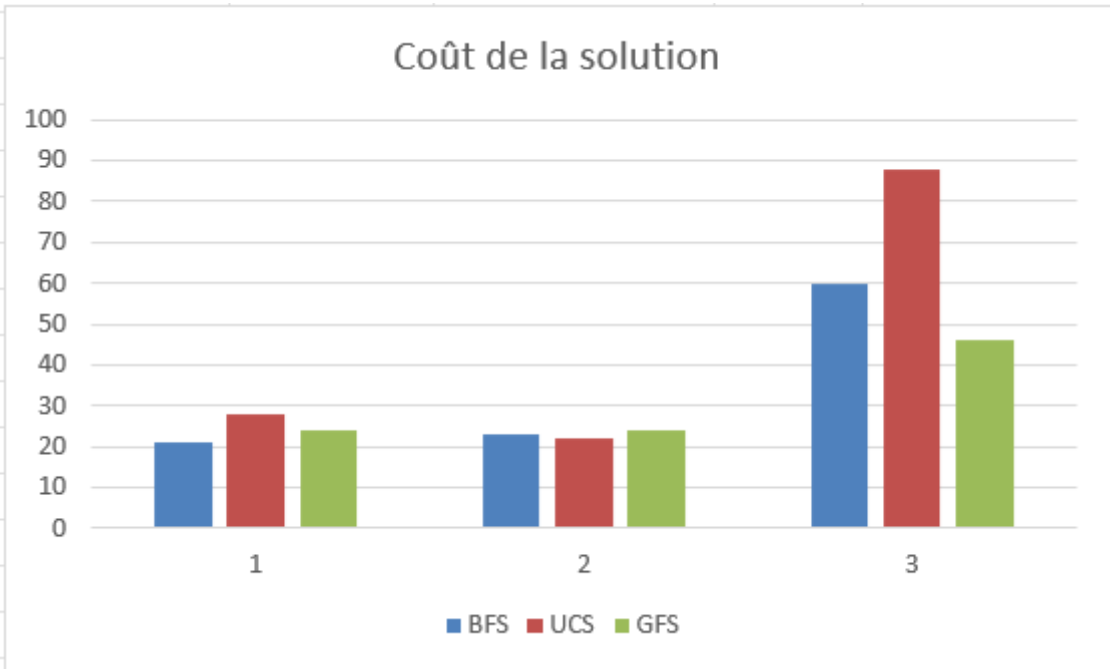
Profondeur maximale

- **UCS et GFS** atteignent des profondeurs similaires (~24 à 29).
- **BFS**, en revanche, atteint des profondeurs moindres



Coût de la solution

- **UCS** trouve la solution **la moins coûteuse** (22 ou 23 selon les cas).
- **BFS** trouve des solutions légèrement plus chères.
- **GFS** donne des solutions **beaucoup plus coûteuses** (jusqu'à 88), confirmant que son approche gloutonne **ne garantit pas d'optimalité**.

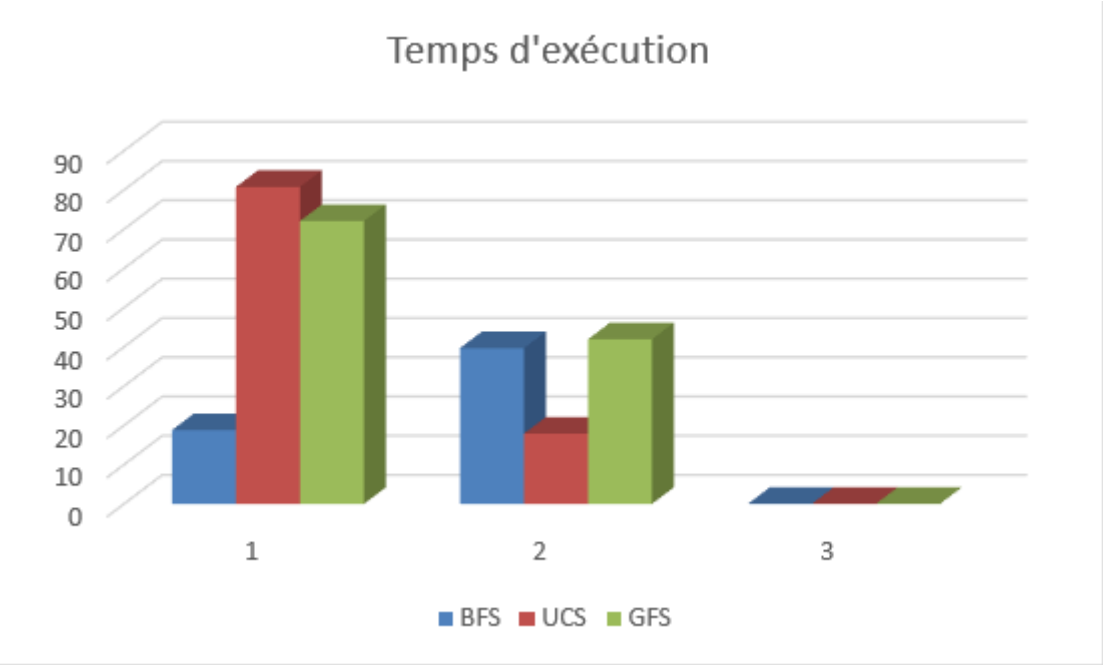


Temps d'exécution

GFS est toujours le plus rapide avec un temps d'exécution extrêmement faible (~0.015 à 0.024 s).

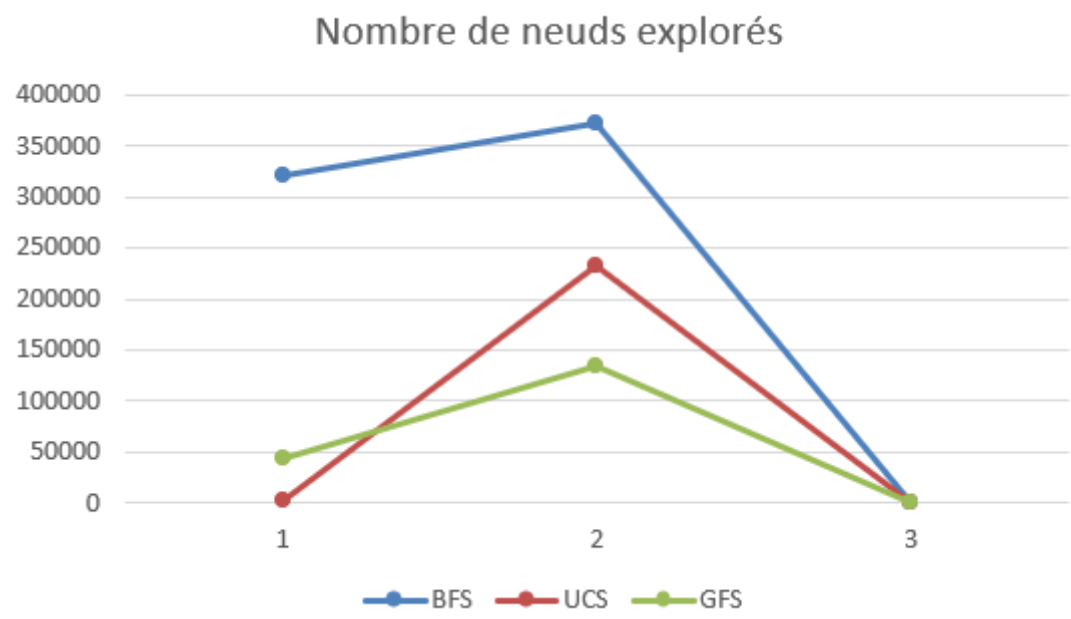
BFS est relativement efficace (~18 s), mais il reste plus lent que GFS.

UCS peut être soit rapide (~17 s) soit très lent (~80 s), en fonction de la configuration du graphe et de la variabilité des coûts des transitions.



A présent, on passe à une **difficulté** de **n = 100 000 et k = 5**, sur 3 essais.

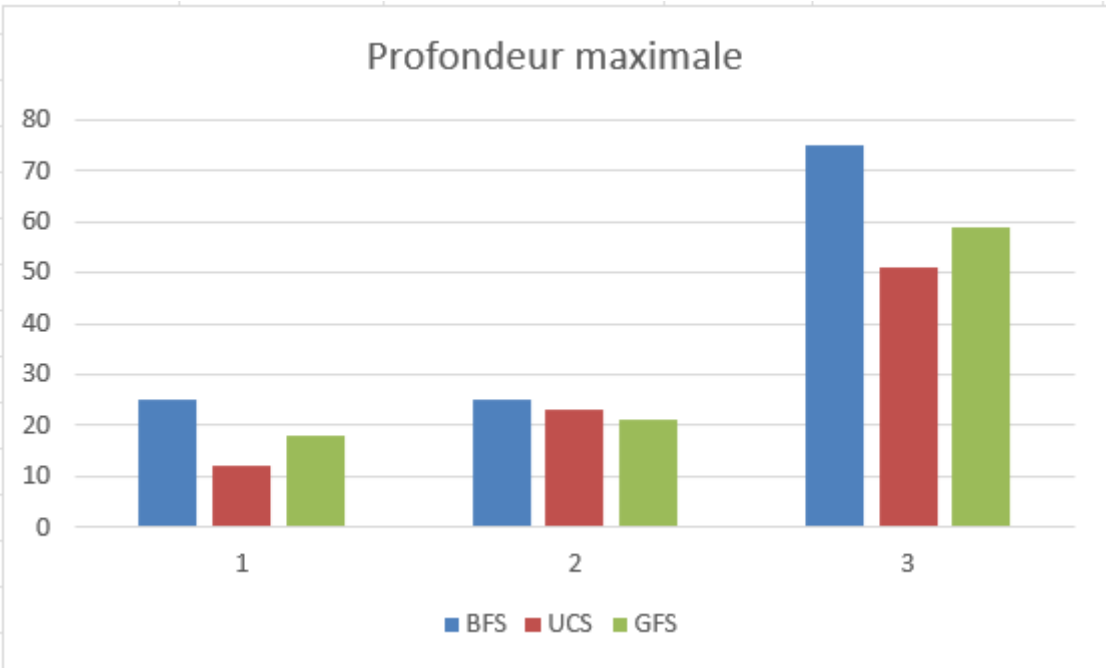
Nombre de nœuds



BFS et UCS explorent un nombre de nœuds **très élevé** (320k-370k pour BFS et UCS en moyenne), ce qui montre leur exhaustivité dans la recherche.

GFS explore beaucoup moins de nœuds (958 à 587).

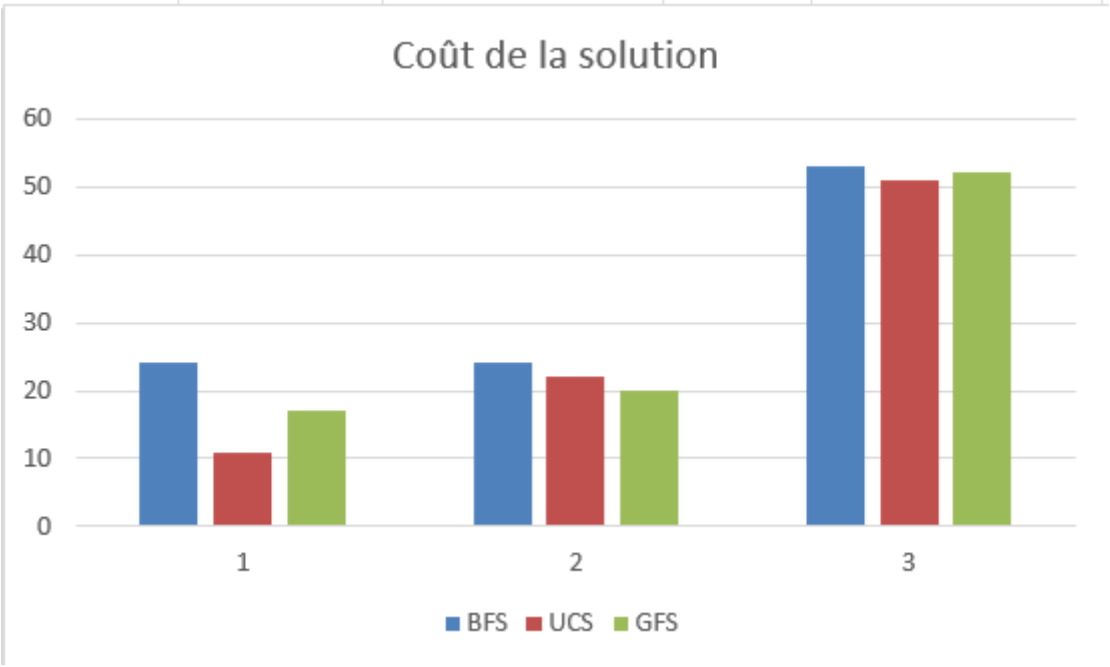
Profondeur maximale



BFS et UCS atteignent des profondeurs similaires (~25)

GFS atteint des profondeurs plus variables, parfois beaucoup plus grandes (jusqu'à 75), ce qui montre qu'il peut suivre des chemins très longs sans forcément garantir une solution efficace.

Coût de la solution

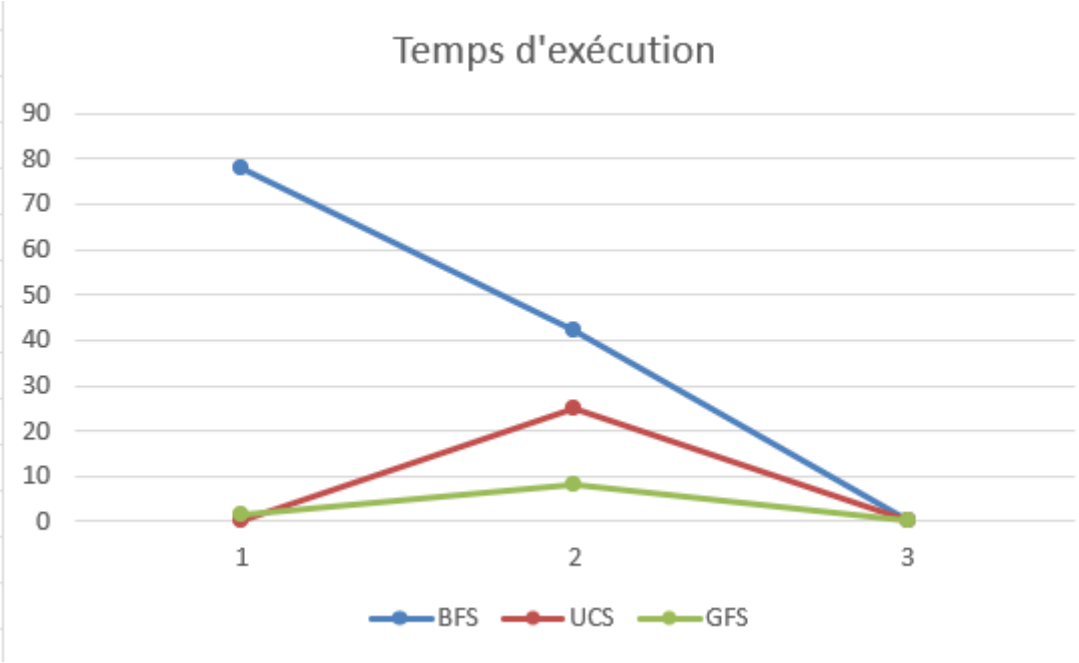


UCS obtient systématiquement la meilleure solution (coût minimum ~11-22), ce qui est attendu puisqu'il optimise le coût.

BFS trouve des solutions plus coûteuses (~24), mais reste cohérent dans ses résultats.

GFS est le pire en termes de coût de solution, souvent dépassant les **50** (jusqu'à 53), confirmant qu'il ne cherche pas l'optimalité.

Temps d'exécution



BFS est le plus lent, avec des temps élevés (jusqu'à 77s).

UCS est bien plus rapide dans certains cas (0.075s), mais peut aussi être lent (environ 25s), ce qui dépend fortement du graphe et des coûts de transitions.

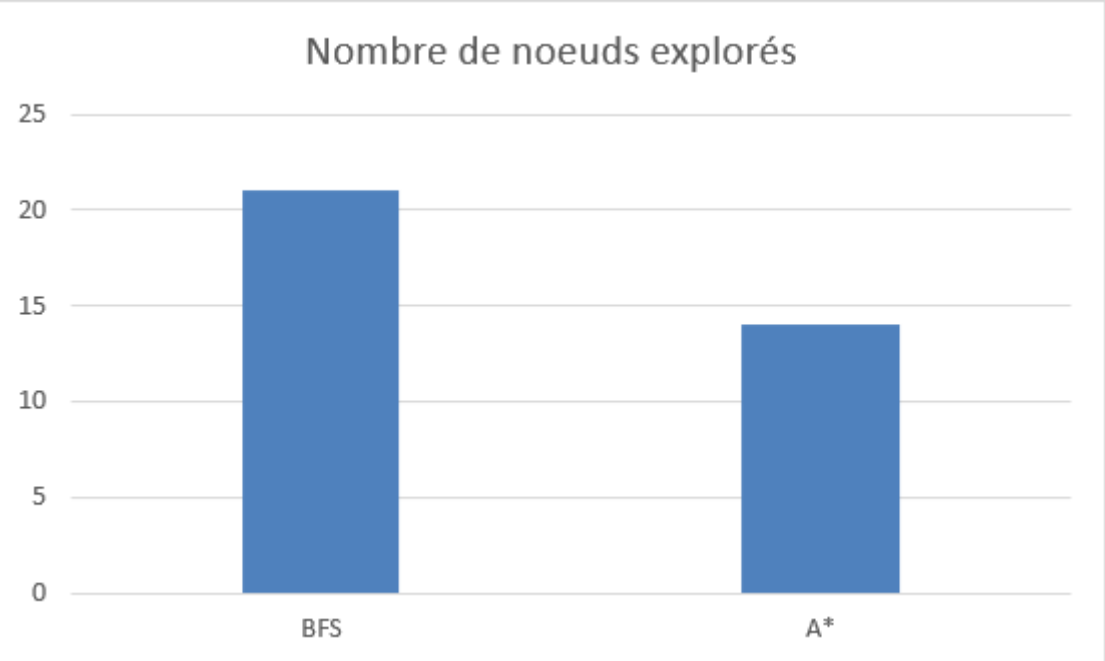
GFS reste extrêmement rapide (environ 0.02s - 0.014s), mais cela au détriment de la qualité de la solution.

BFS vs A*

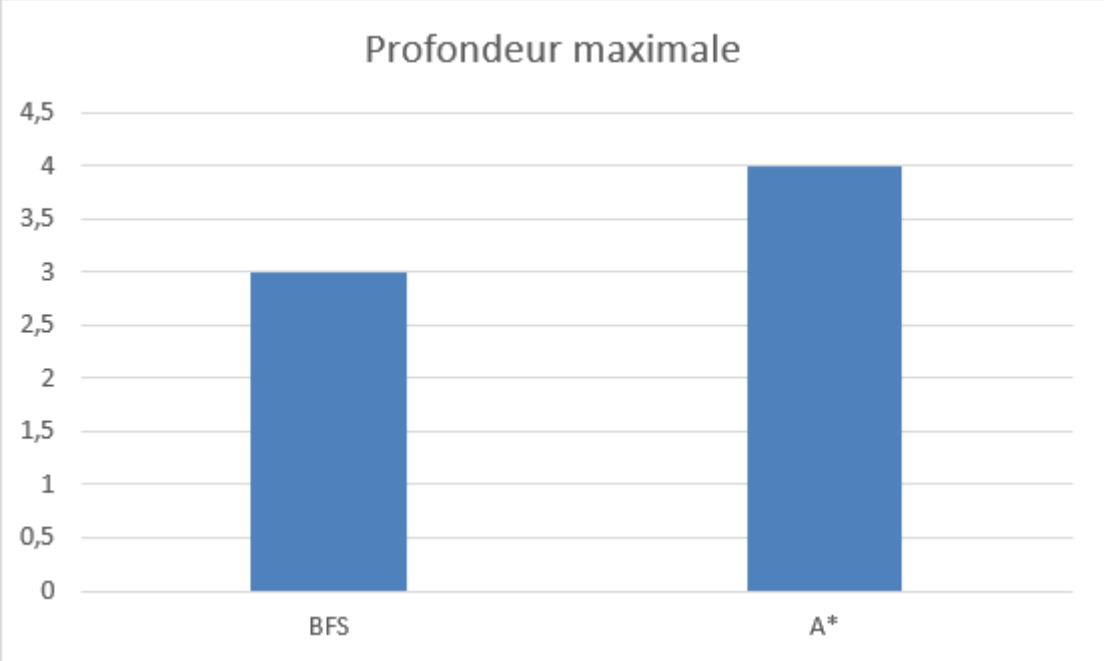
Nous allons maintenant comparer **BFS** et **A*** afin d'évaluer l'impact de l'heuristique sur la recherche de solutions. A* combine le coût passé (**g**) et une heuristique (**h**) pour minimiser le coût total estimé du chemin. L'objectif est d'observer si A* permet une exploration plus efficace et une meilleure optimisation des solutions sur différents types de problèmes. Nous commencerons par tester cela sur le problème **map**.

Problème map

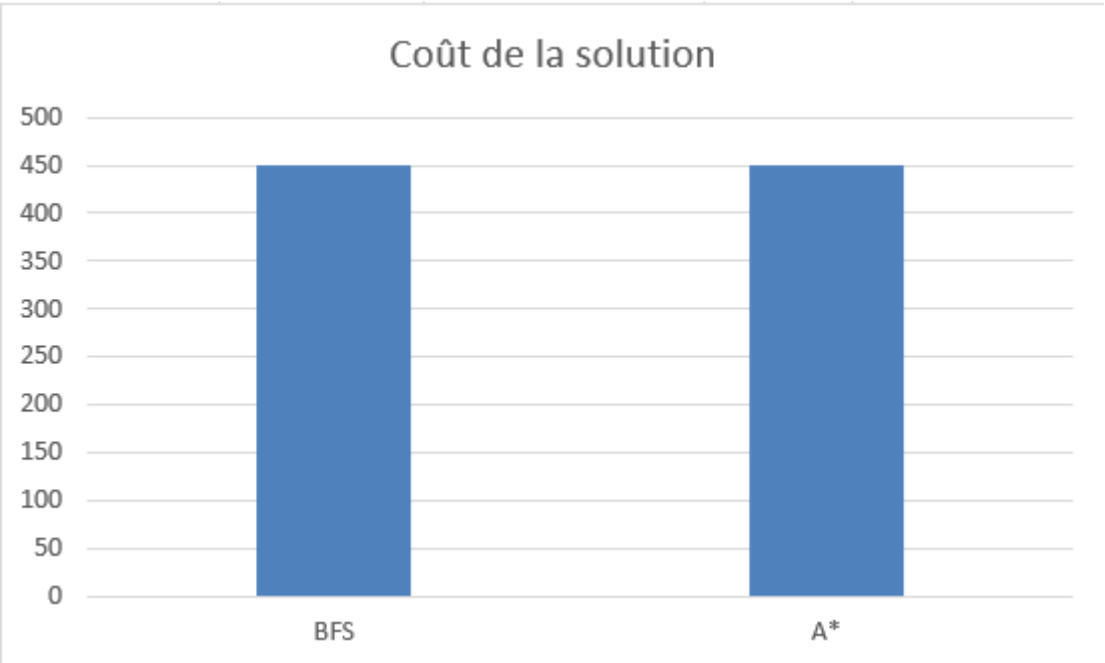
Nombre de nœuds



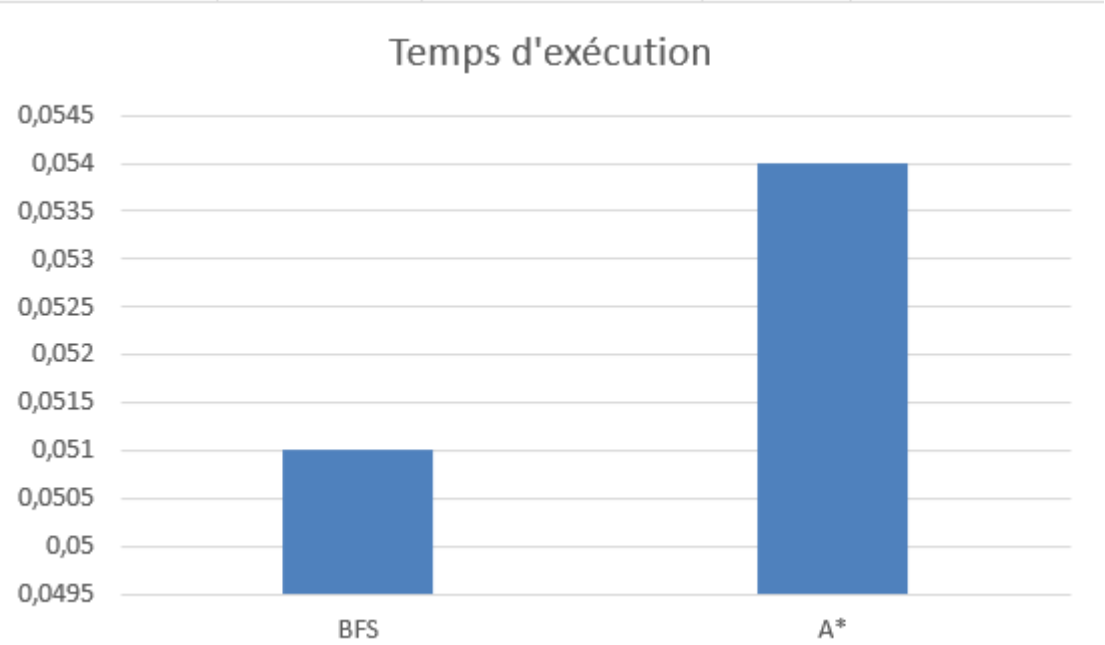
Profondeur maximale



Coût de la solution



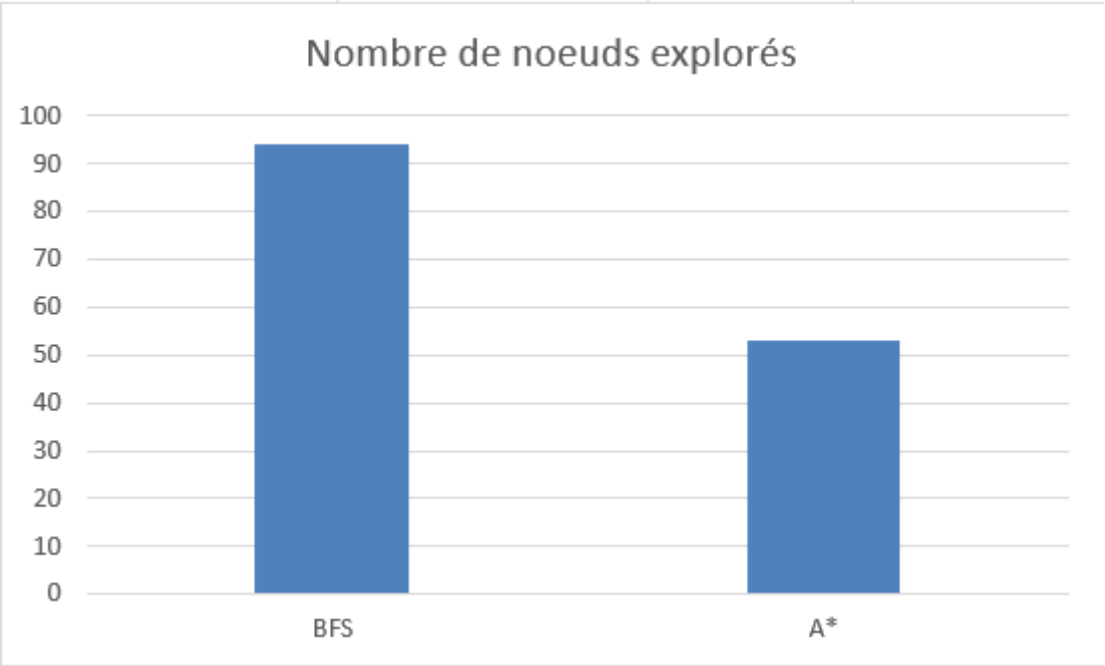
Temps d'exécution



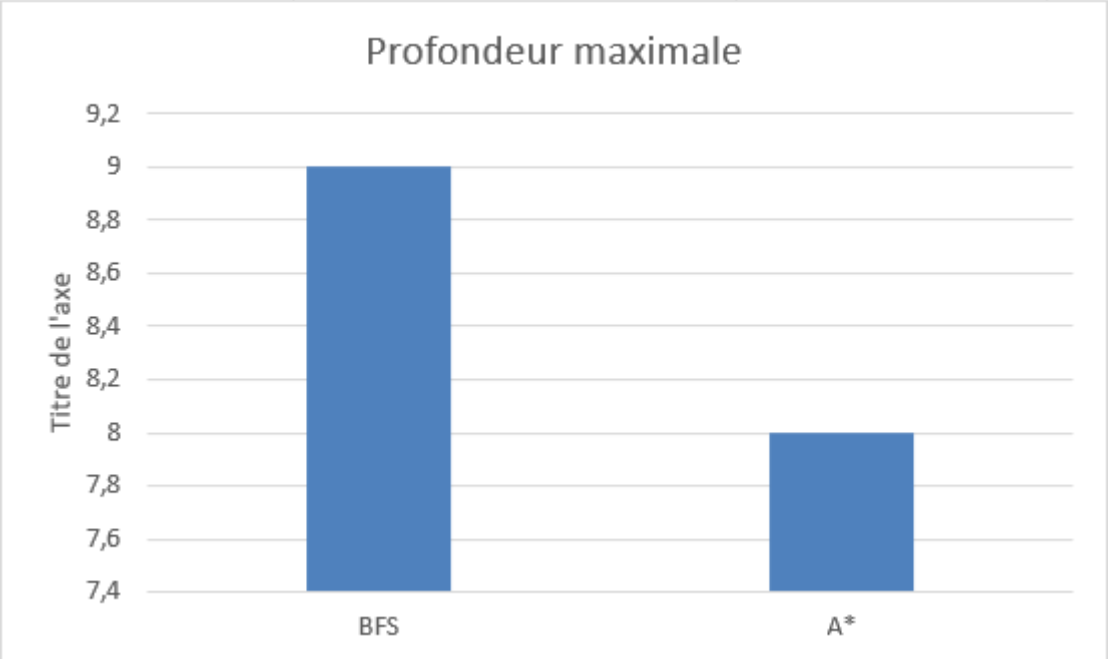
Le test entre BFS et A* sur ce problème montre des écarts plutôt minimes. A* explore légèrement moins de nœuds (14 contre 21), ce qui est attendu puisqu'il utilise une heuristique pour guider la recherche. Cependant la profondeur maximale atteinte par A* est un peu plus élevée (4 contre 3), ce qui peut être dû à l'ordre d'exploration des nœuds. Concernant le coût de la solution, les deux algorithmes fournissent exactement le même résultat (450). Enfin, le temps d'exécution est quasi identique, avec une très légère différence en faveur de BFS. Ces résultats indiquent que pour des problèmes simples, l'avantage d'A* en termes d'optimisation des nœuds explorés reste limité.

Problème dum

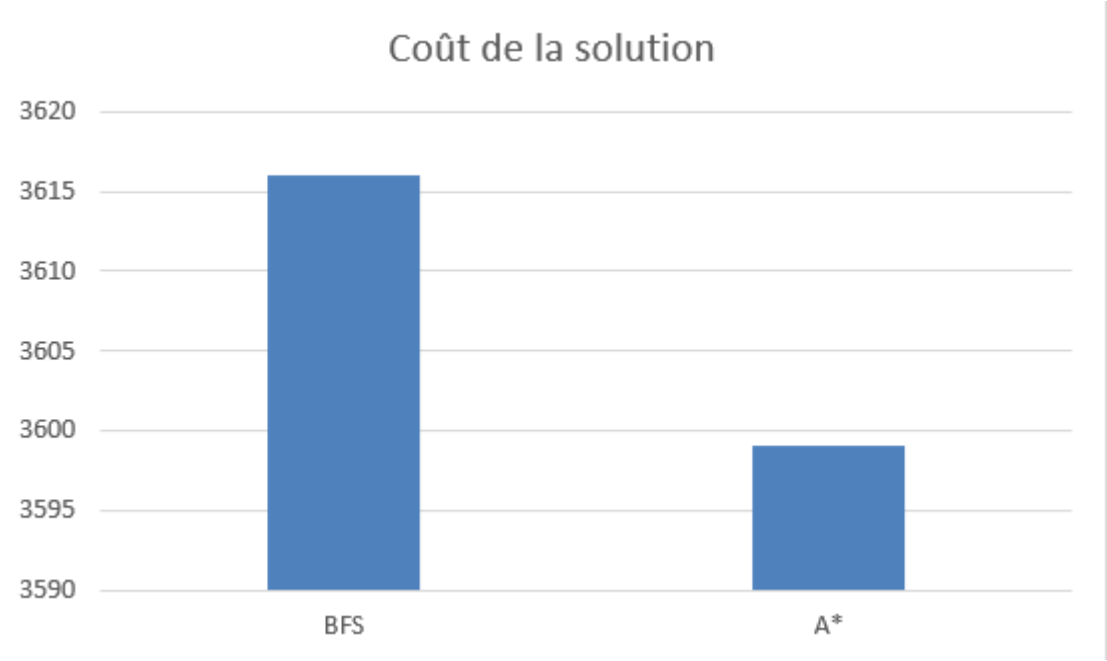
Nombre de nœuds



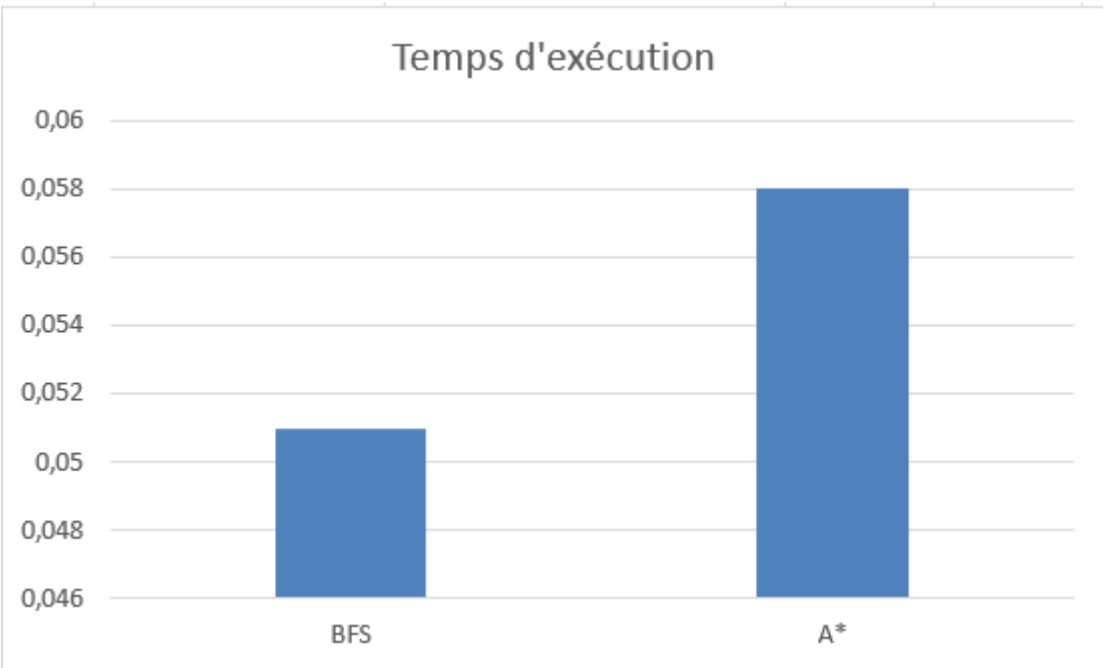
Profondeur maximale



Coût de la solution



Temps d'exécution

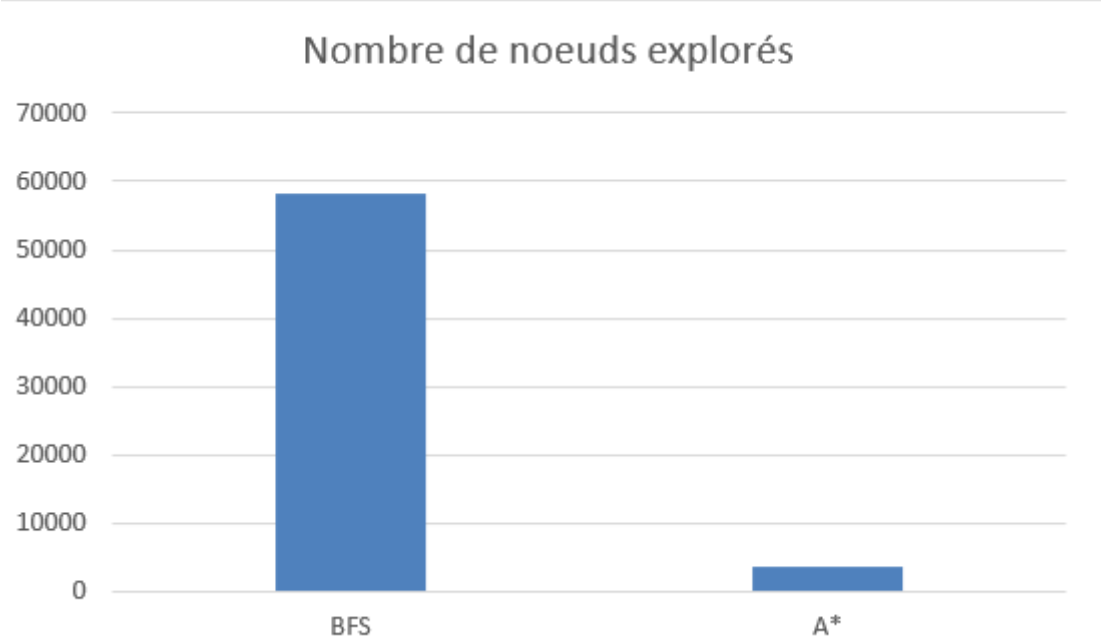


Dans ce test avec le problème *dum*, A* montre une amélioration notable en réduisant le nombre de nœuds explorés (53 contre 94 pour BFS). On confirme ici que l'heuristique aide à cibler plus efficacement les bons chemins. En revanche, la profondeur maximale atteinte est similaire (8 pour A* contre 9 pour BFS), suggérant que les deux méthodes suivent des trajectoires assez proches dans cet exemple. Niveau coût de la solution, A* trouve une solution légèrement meilleure (3599 contre 3616). Enfin, le temps d'exécution reste comparable, A* étant un peu plus lent que BFS (0,058 sec contre 0,051 sec).

Problème dum : difficulté croissante

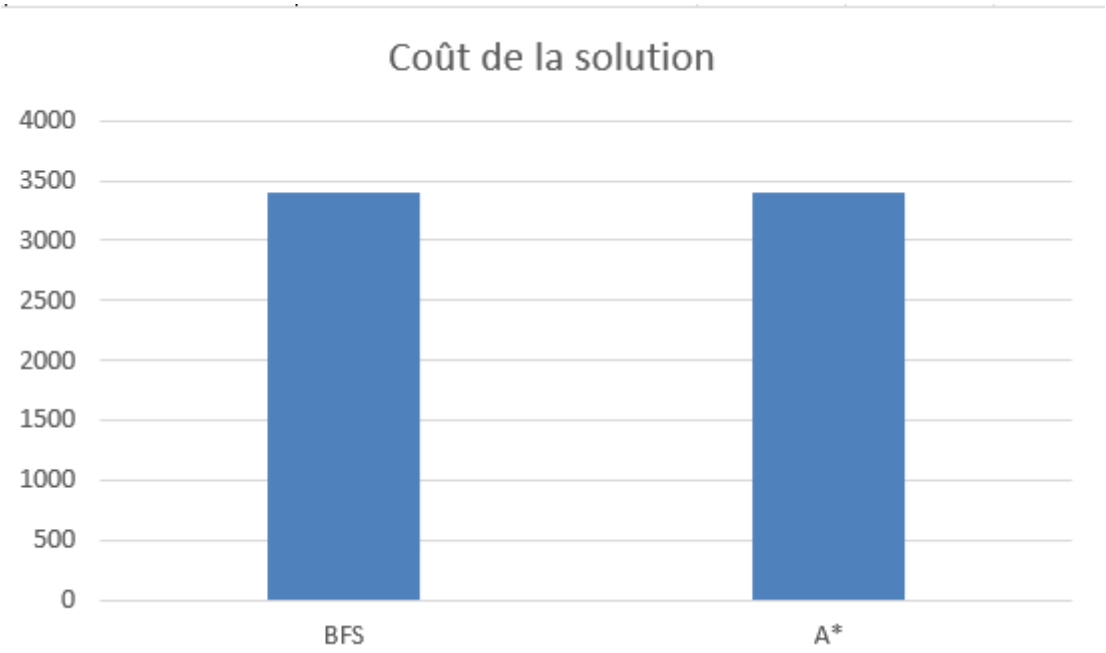
Testons avec une difficulté significative : n = 100 000, k = 3

Nombre de nœuds explorés



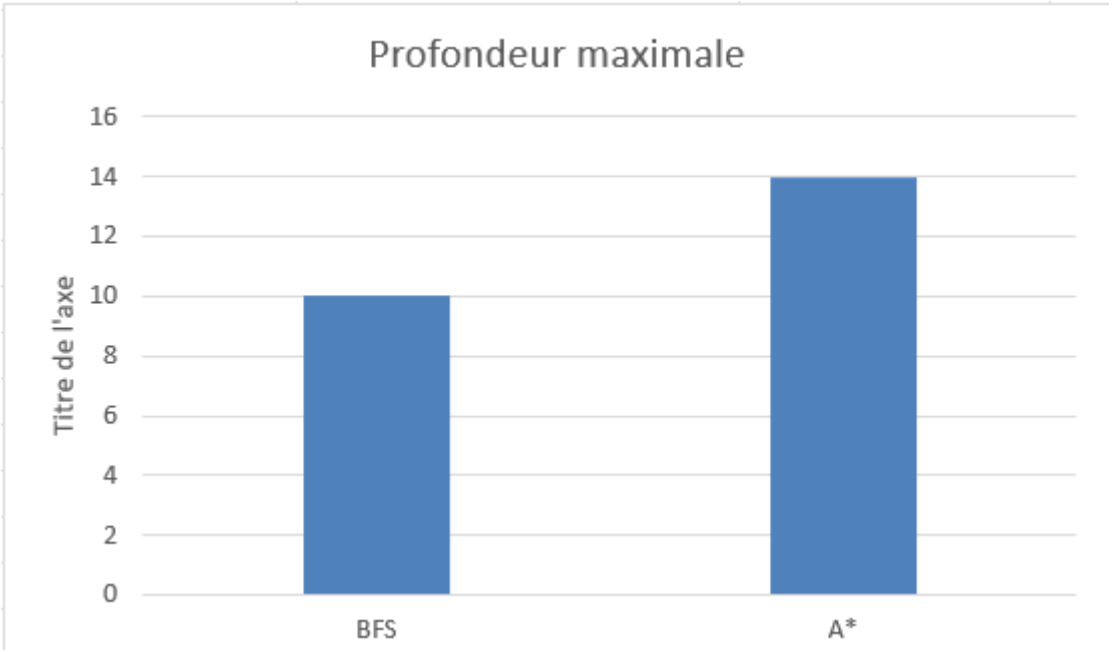
A* démontre une efficacité impressionnante en réduisant considérablement le nombre de nœuds explorés : **3 797 contre 58 205 pour BFS**. Cela met clairement en évidence l'avantage de l'heuristique dans A*, qui oriente rapidement la recherche vers la solution au lieu d'explorer tout l'espace comme BFS.

Coût de la solution



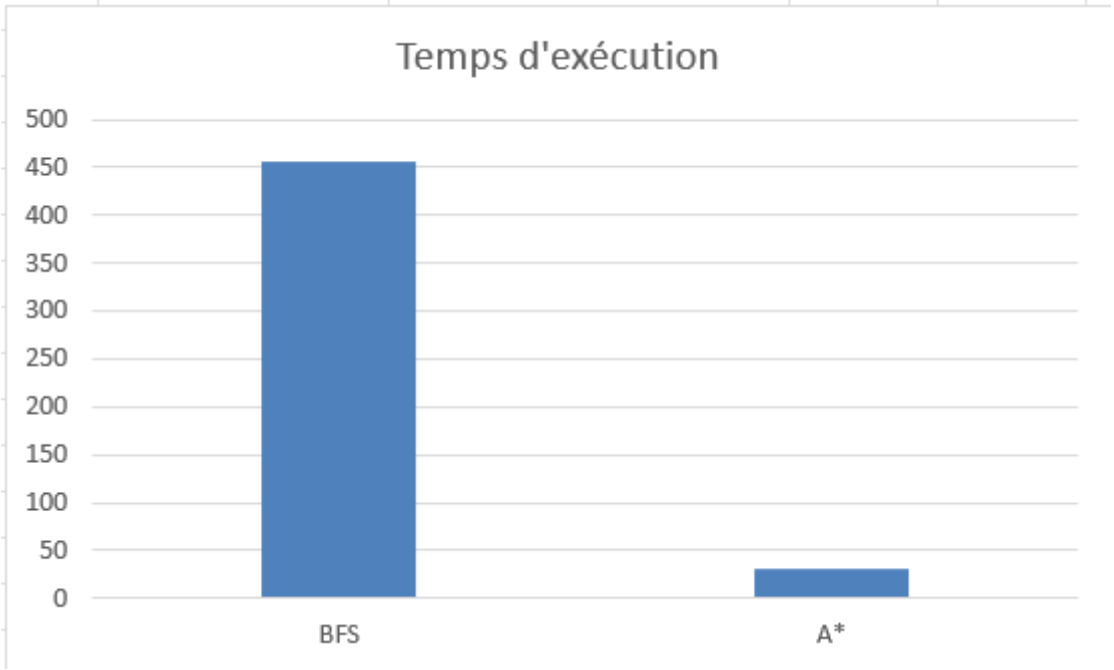
Les deux algorithmes trouvent une solution avec **exactement le même coût de 3 394**.

Profondeur maximale



A* atteint une profondeur de **14**, légèrement plus élevée que BFS (**10**). Cela peut s'expliquer par le fait qu'A* explore d'abord les chemins les plus « prometteurs » disons, même s'ils sont plus longs, tandis que BFS explore sans se soucier du coût.

Temps d'exécution



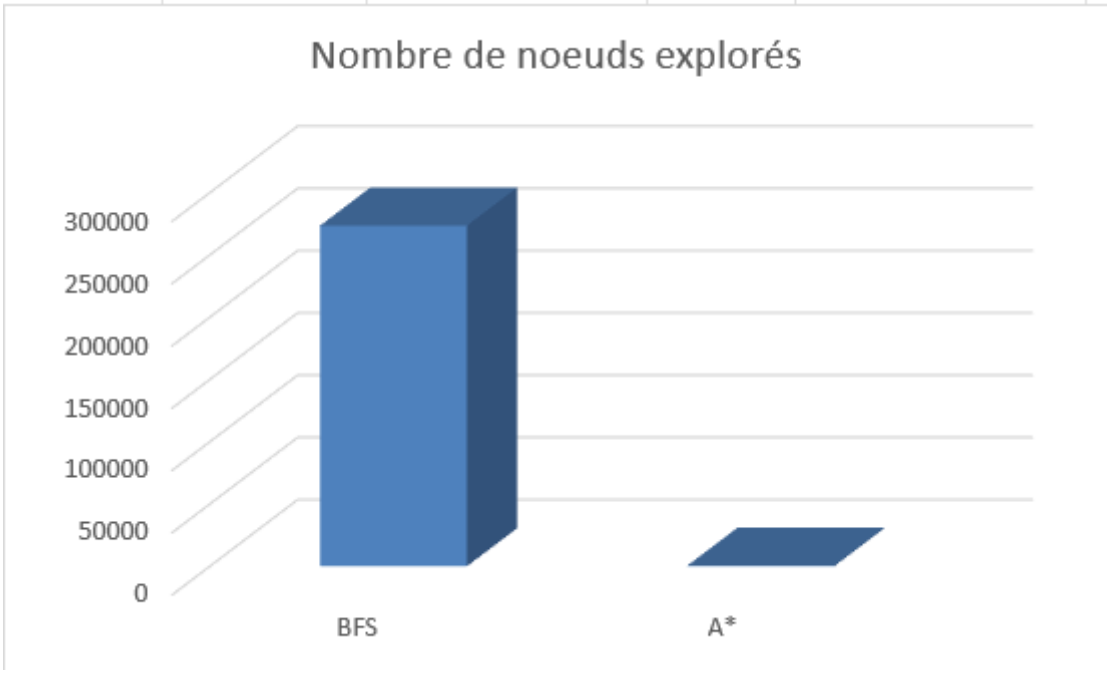
A* est **plus de 10 fois plus rapide** que BFS (**30,676 sec contre 456,127 sec**). Ce résultat est une conséquence directe du nombre réduit de nœuds explorés par A*, ce qui lui permet de converger bien plus rapidement que BFS.

Avec une grosse difficulté, A* confirme sa supériorité par rapport à BFS. Il explore beaucoup moins de nœuds et trouve toujours une solution optimale.

Problème puz

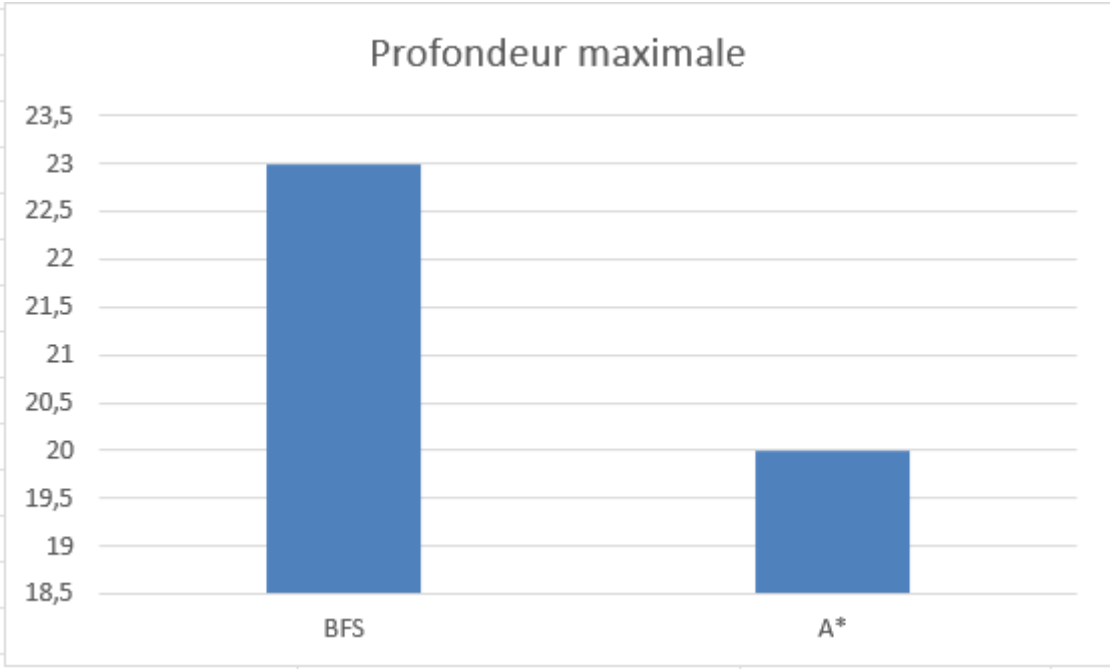
Traitons le problème puz maintenant.

Nombre de nœuds explorés



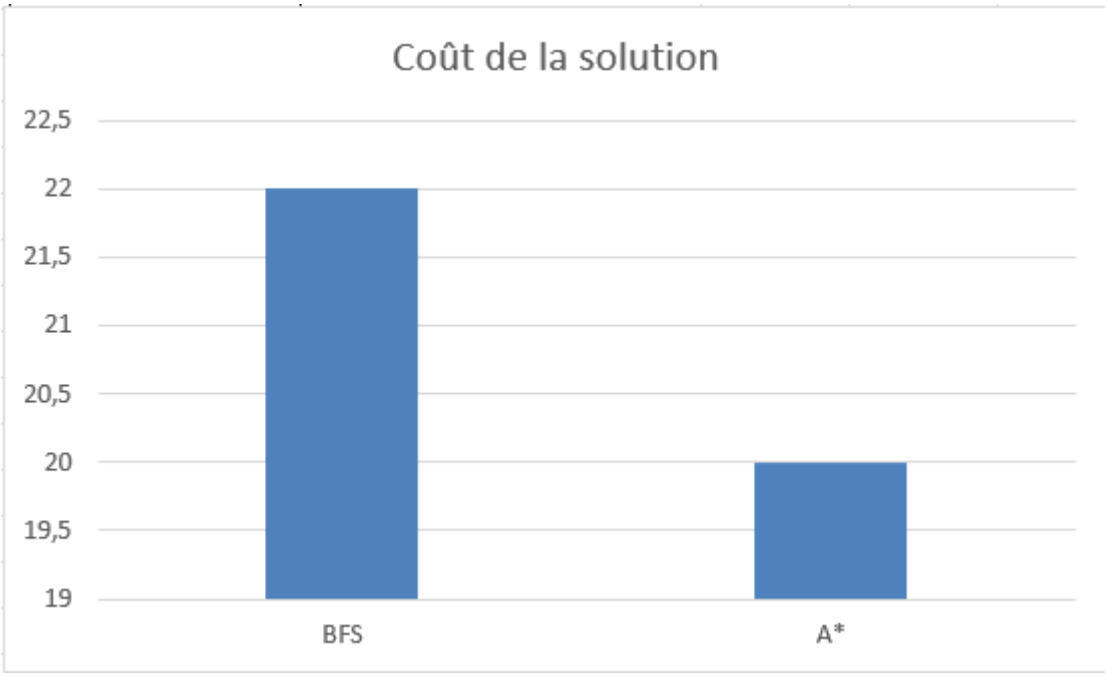
A* est **extrêmement efficace** ici, explorant **seulement 430 nœuds** contre **273 491 pour BFS**.

Profondeur maximale



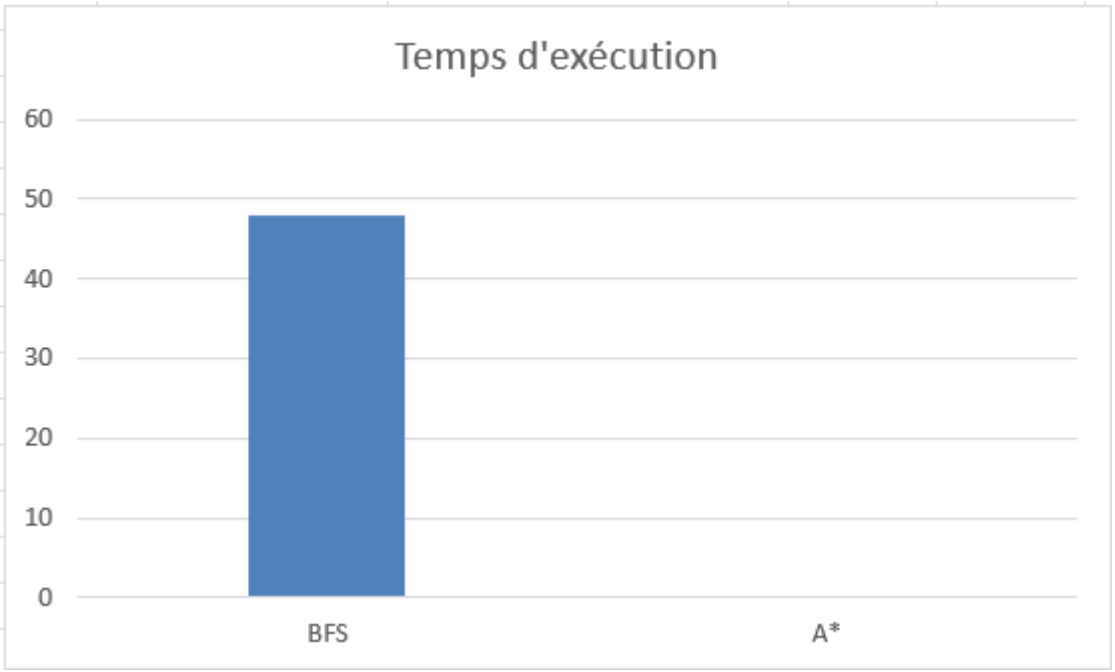
A* atteint une profondeur légèrement moindre (**20 contre 23** pour BFS). Cela signifie qu'A* trouve une solution plus tôt en sélectionnant des bons chemins dès le départ.

Coût de la solution



A* trouve une solution légèrement meilleure avec un coût de **20**, contre **22 pour BFS**.

Temps d'exécution

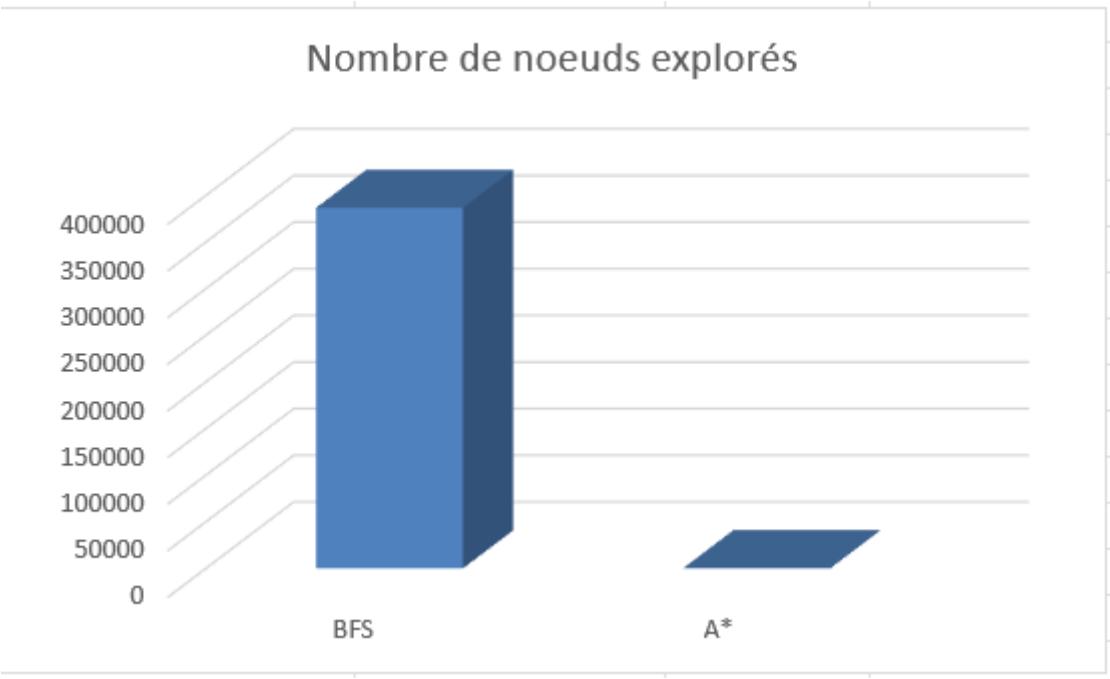


A* est **près de 775 fois plus rapide** que BFS (**0,062 sec contre 48,082 sec**), ce qui est énromé.

Probleme puz : difficulté croissante

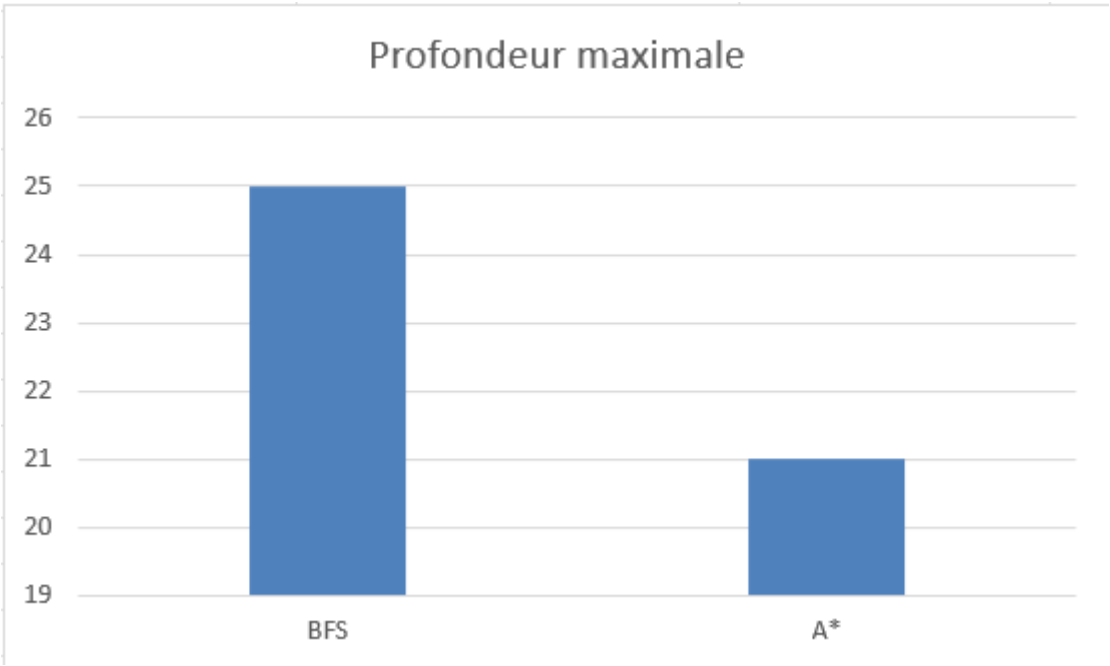
Continuons sur le même problème, avec une difficulté croissante (n=1000).

Nombre de nœuds explorés



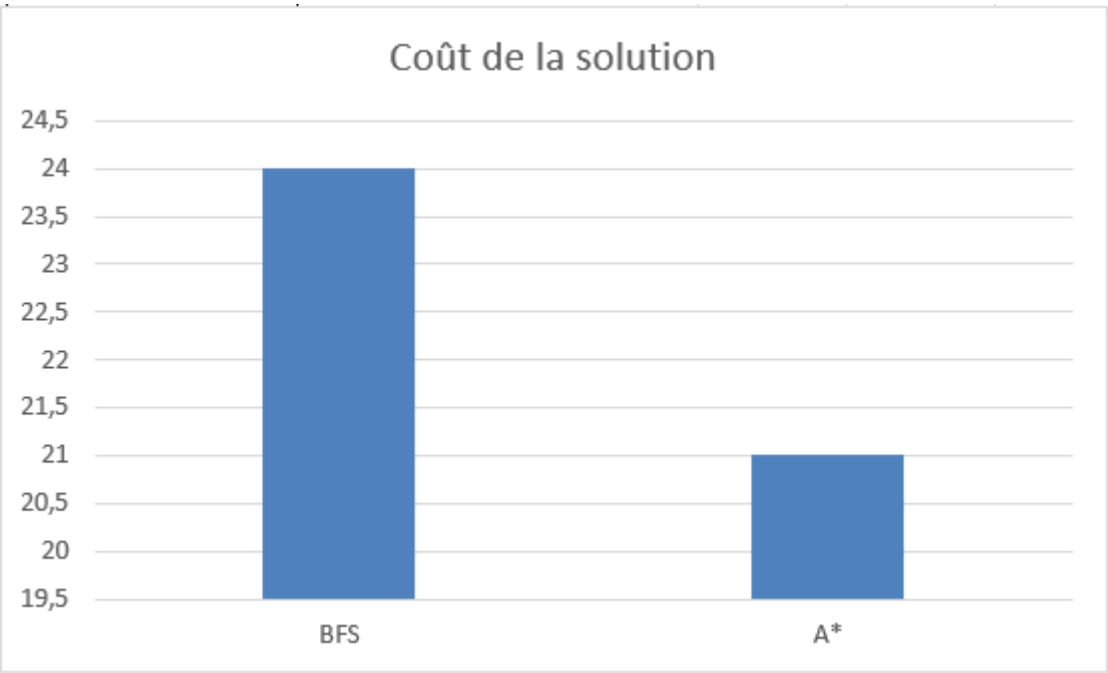
A* explore **seulement 299 nœuds**, alors que BFS en explore **386 667**. L'heuristique permet donc une recherche **beaucoup plus ciblée**.

Profondeur maximale



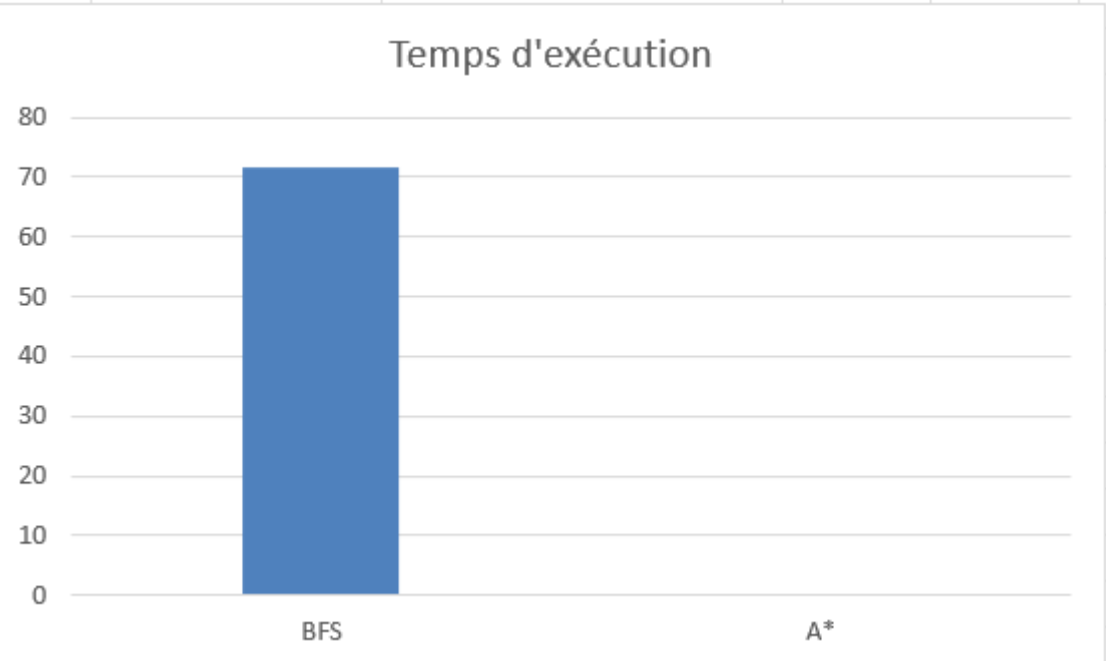
A* atteint la solution en **21 niveaux** contre **25 pour BFS**.

Coût de la solution



A* trouve une solution légèrement meilleure avec un coût de **21**, contre **24** pour BFS.

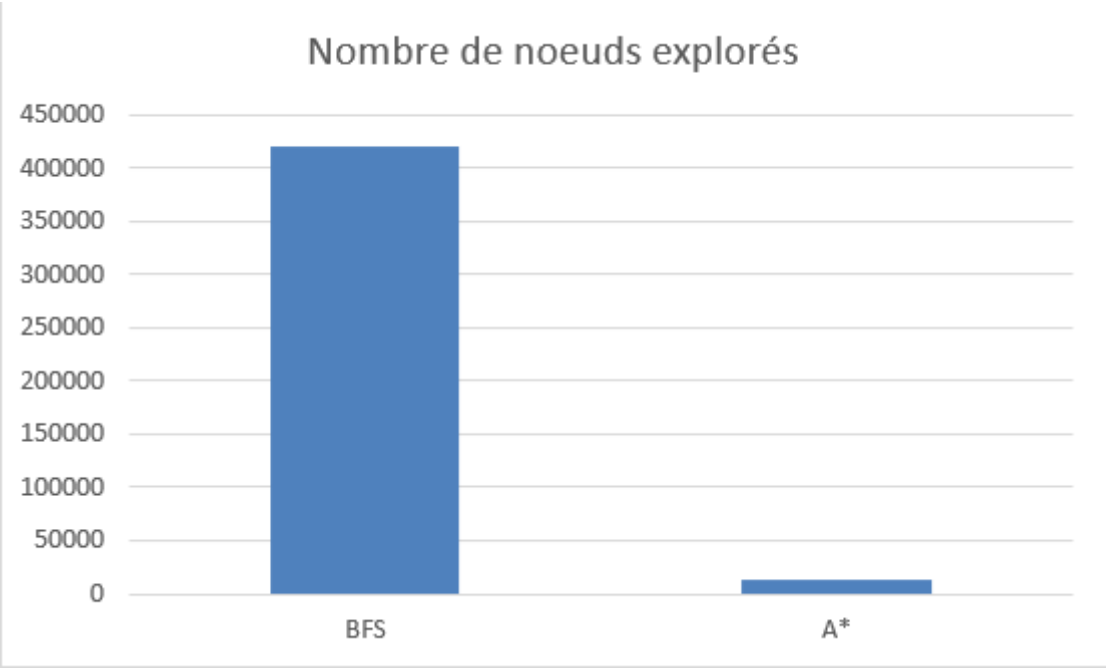
Temps d'exécution



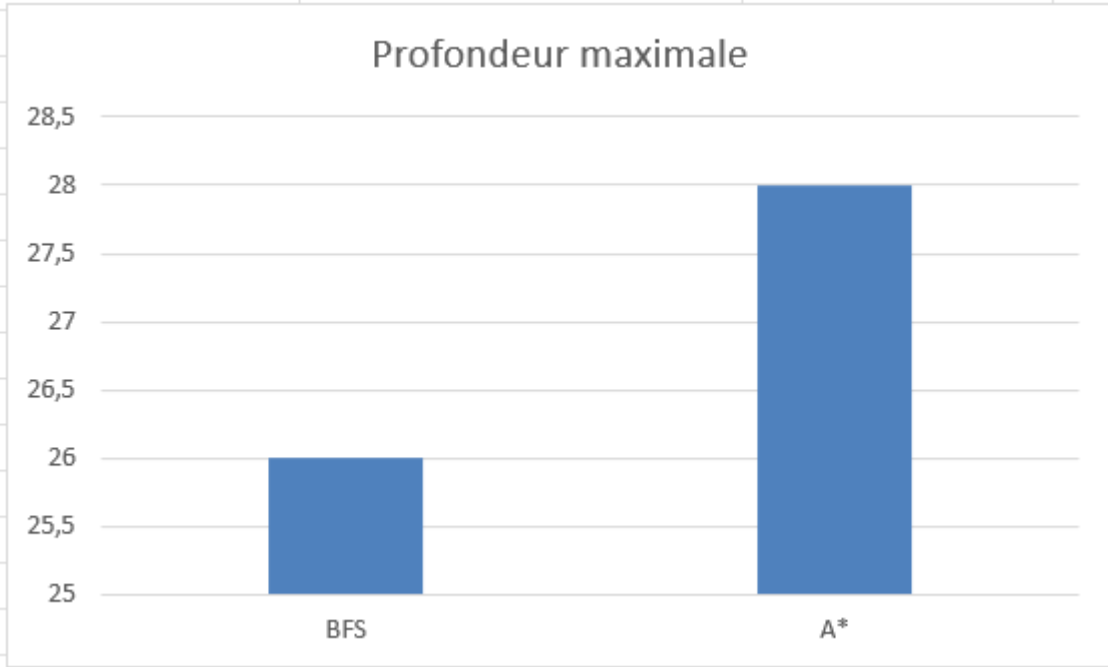
A* est **1255 fois plus rapide** que BFS (**0,057 sec contre 71,551 sec**), prouvant son énorme avantage en termes de performances.

Avec une difficulté nettement plus grande : **n=1 000 000, k = 5**.

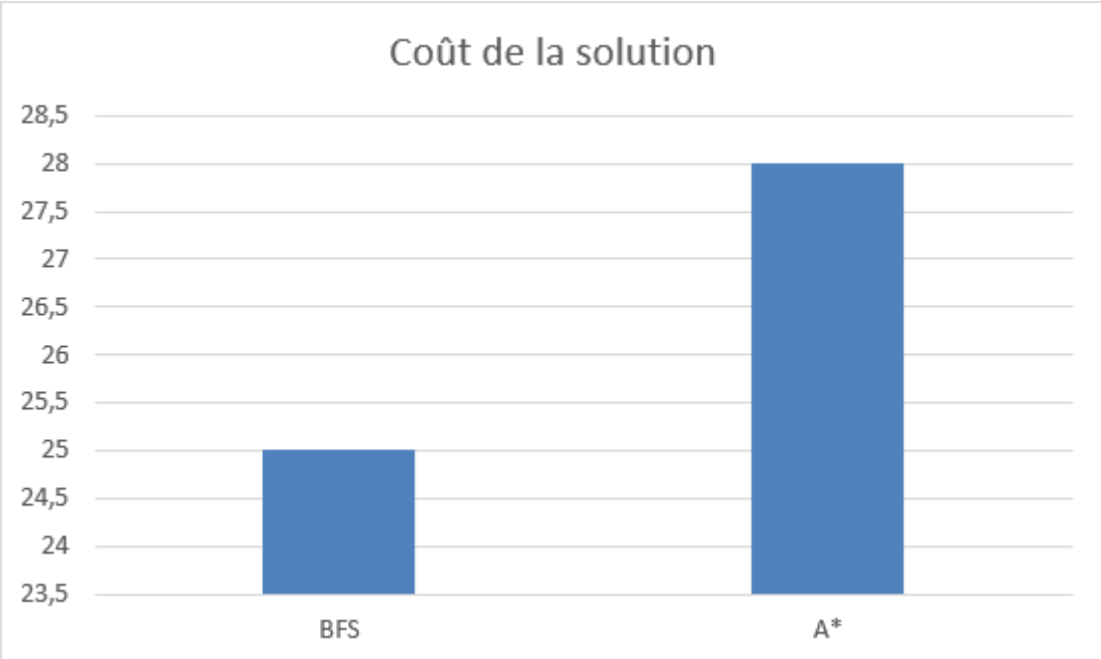
Nombre de nœuds explorés



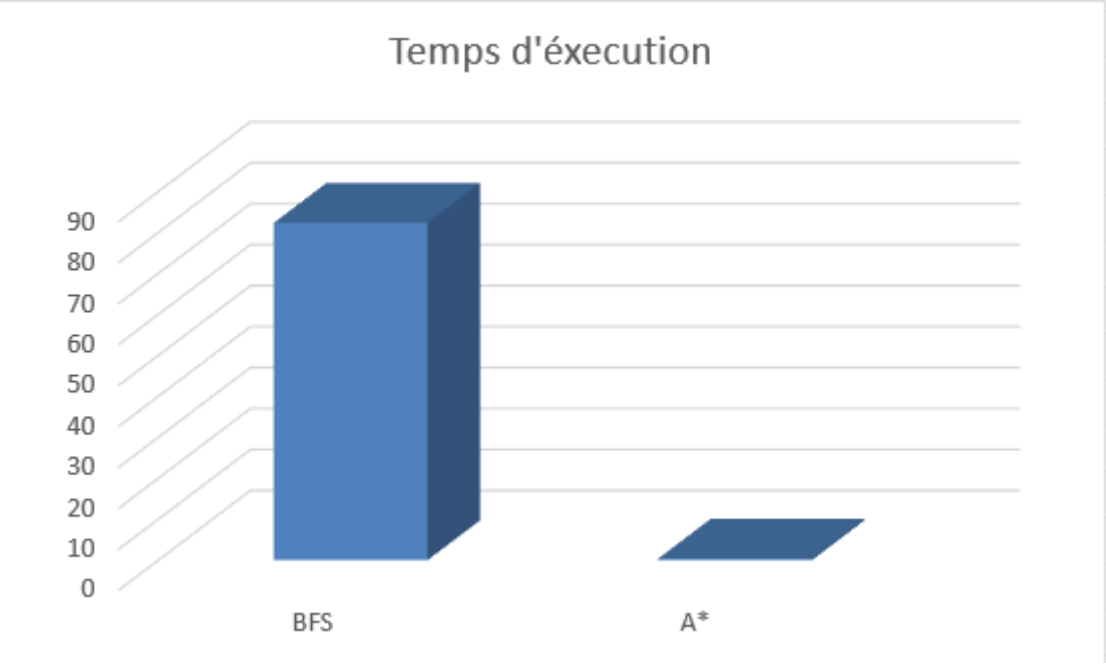
Profondeur maximale



Coût de la solution



Temps d'exécution



Même si A* n'a pas trouvé **la solution optimale** en termes de coût, son **efficacité en nombre de nœuds explorés et en temps d'exécution reste impressionnante**. L'heuristique fait donc bien son travail en réduisant considérablement la complexité du problème.

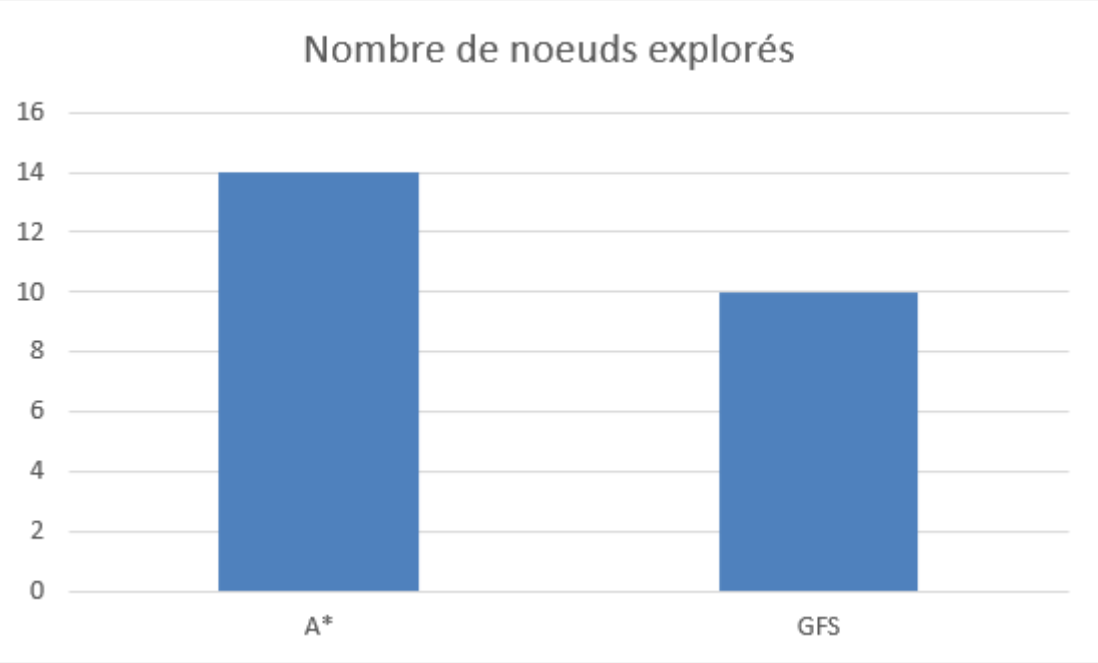
A* vs GFS

Nous allons maintenant comparer **A*** et **GFS** afin d'évaluer l'impact de l'heuristique et du coût sur la recherche de solutions.

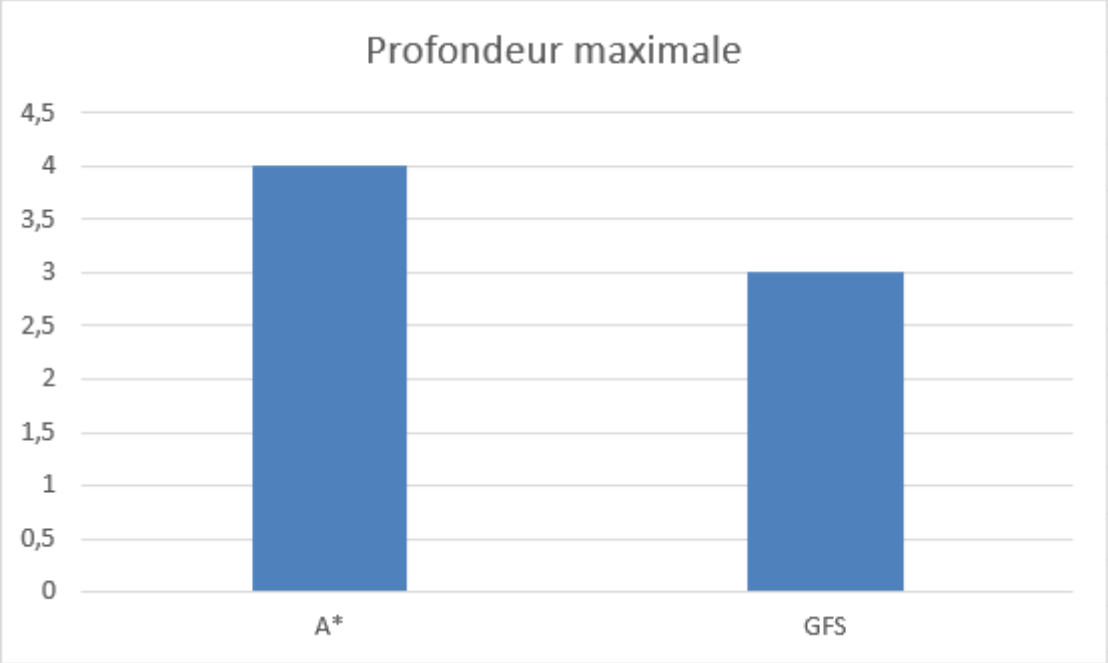
L’objectif est de voir **si A*** parvient à trouver de meilleures solutions que GFS en termes de coût et d'exploration, et d'observer leur efficacité respective sur des problèmes différents. Nous allons commencer par le problème **map**.

Problème map

Nombre de nœuds explorés

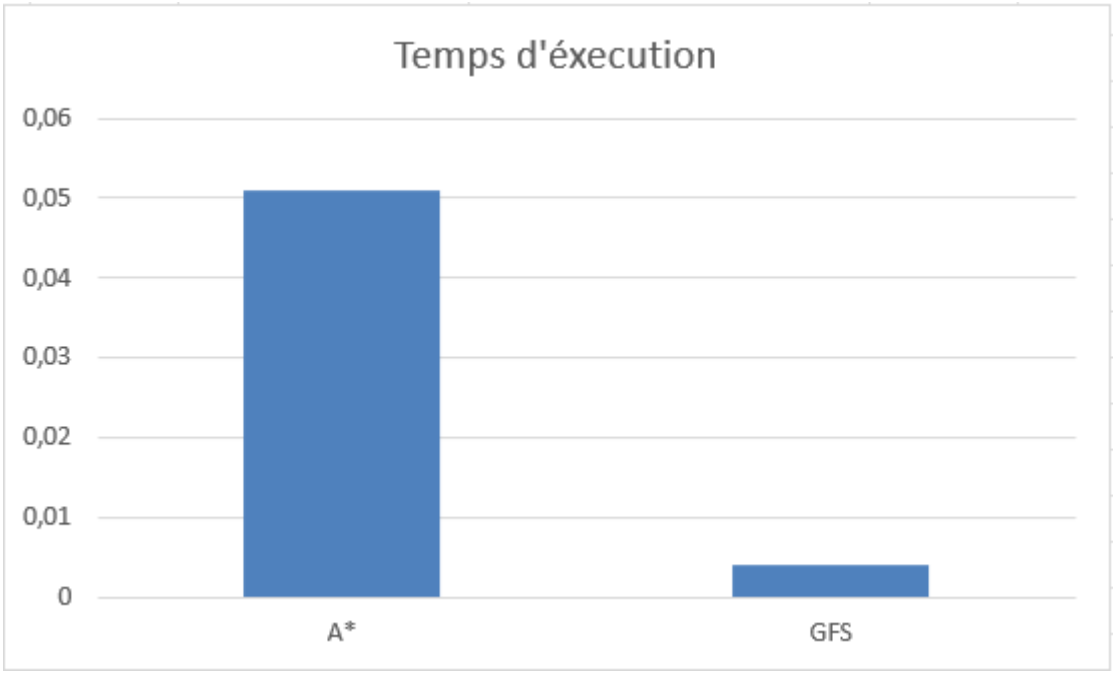


Profondeur maximale



Coût de la solution

Temps d'exécution



Dans cette première comparaison entre **A*** et **GFS** sur le problème **map**, nous observons que GFS explore un peu moins de nœuds (10 contre 14 pour A) et atteint une profondeur légèrement inférieure (3 contre 4). Cela s'explique par le fait que **GFS suit uniquement l'heuristique**.

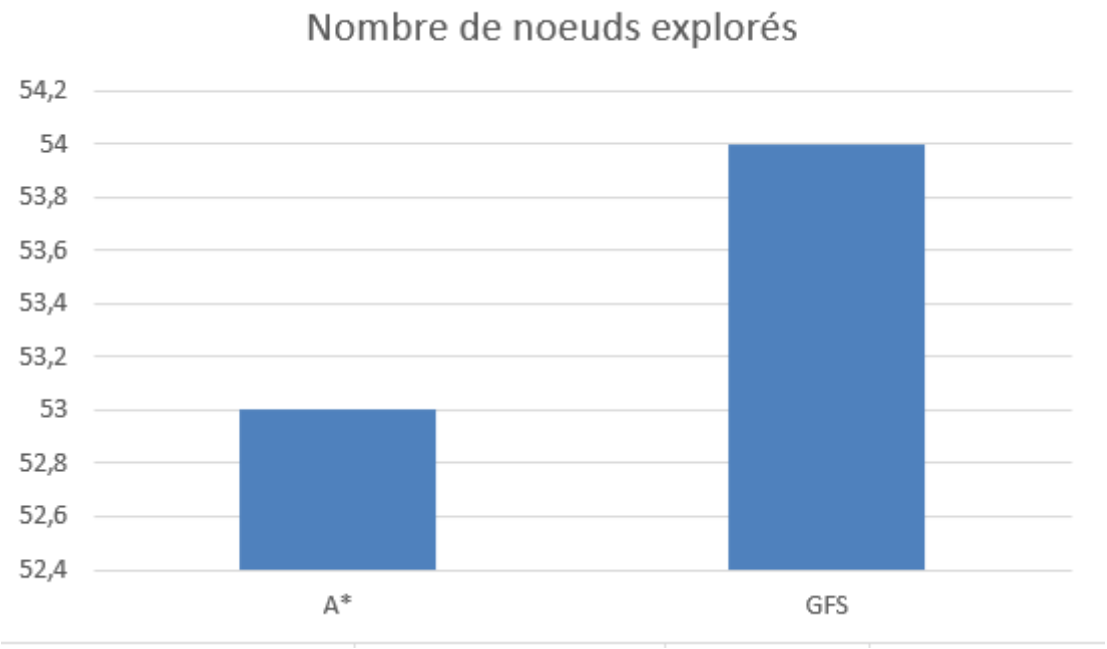
Cependant, **les deux algorithmes trouvent la même solution avec un coût identique (450)**, ce qui peut signifier que dans ce cas précis l'heuristique seule est suffisante pour atteindre une solution optimale. **GFS est plus rapide** en termes de temps d'exécution (**0,004s contre 0,051s pour A***).

Toutefois, cette observation devra être nuancée avec des problèmes plus complexes où le coût réel des chemins pourrait jouer un rôle plus déterminant.

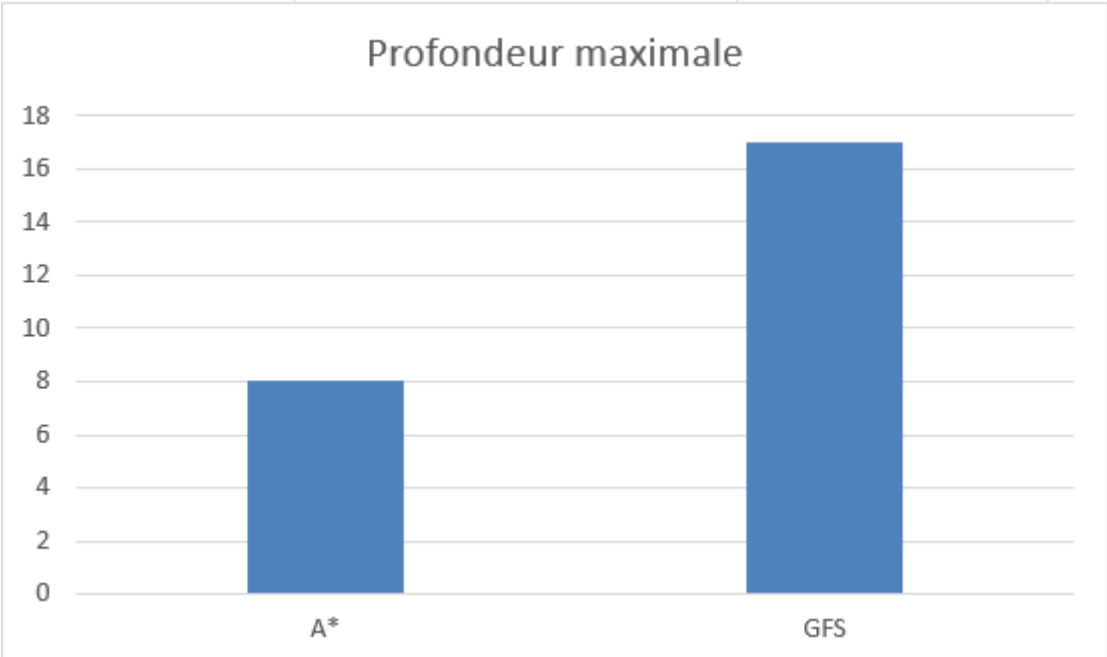
On va d'ailleurs tout de suite changer de problème, avec le **dum**.

Problème dum

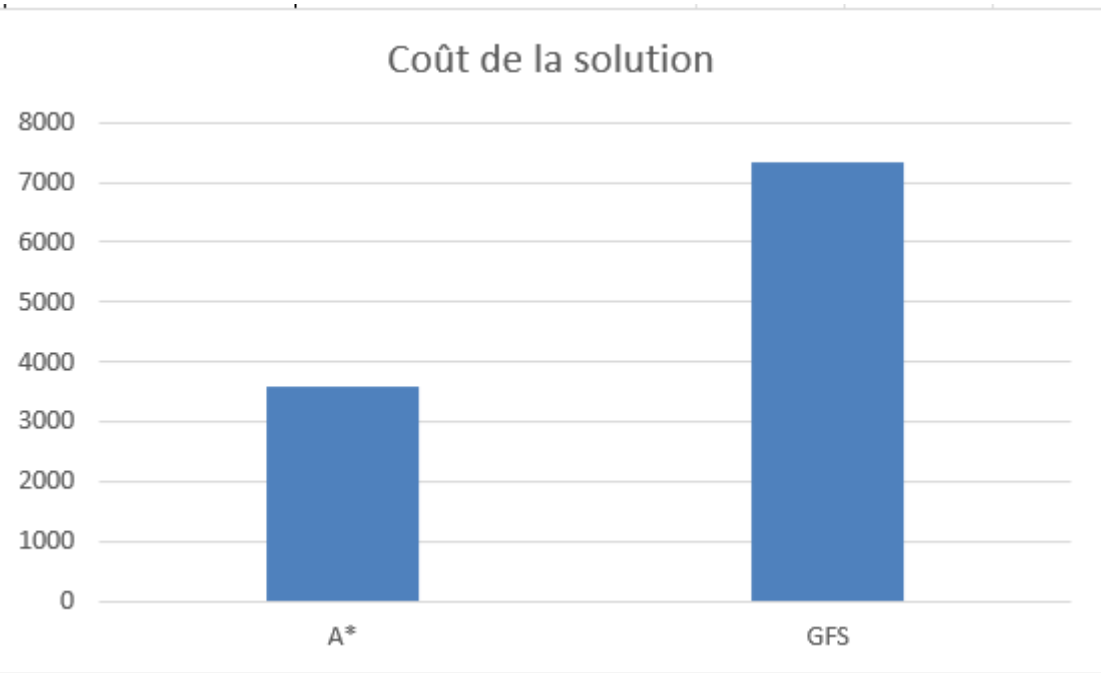
Nombre de nœuds explorés



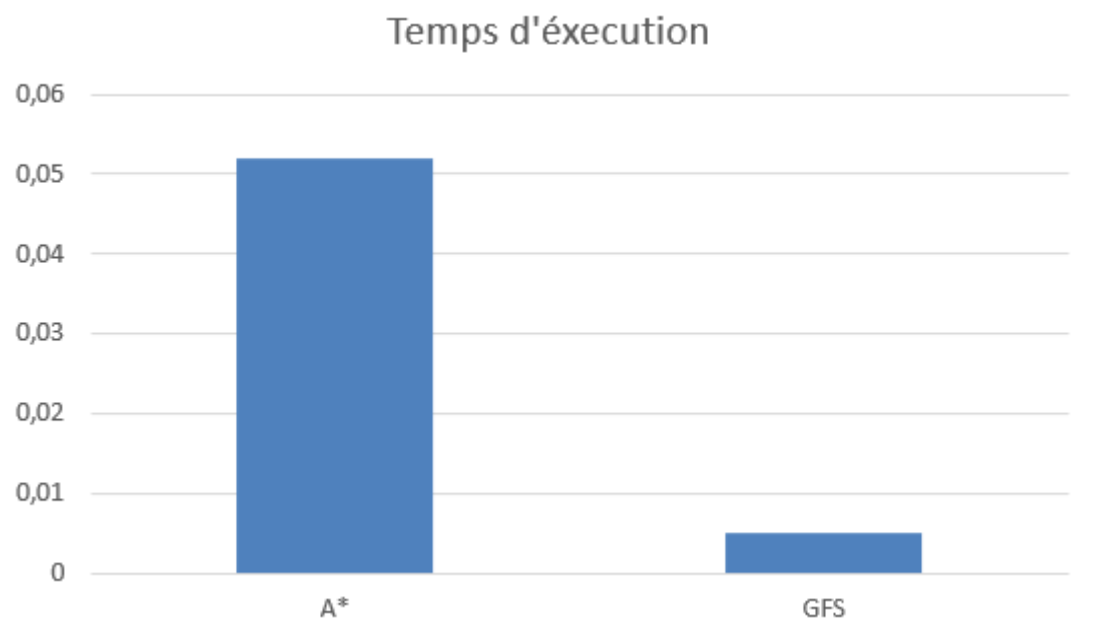
Profondeur maximale



Coût de la solution



Temps d'exécution



Sur le problème dum, **A*** et **GFS** explorent quasiment le même nombre de nœuds (**53 contre 54**), mais **GFS descend beaucoup plus profondément dans l'arbre (17 contre 8)**.

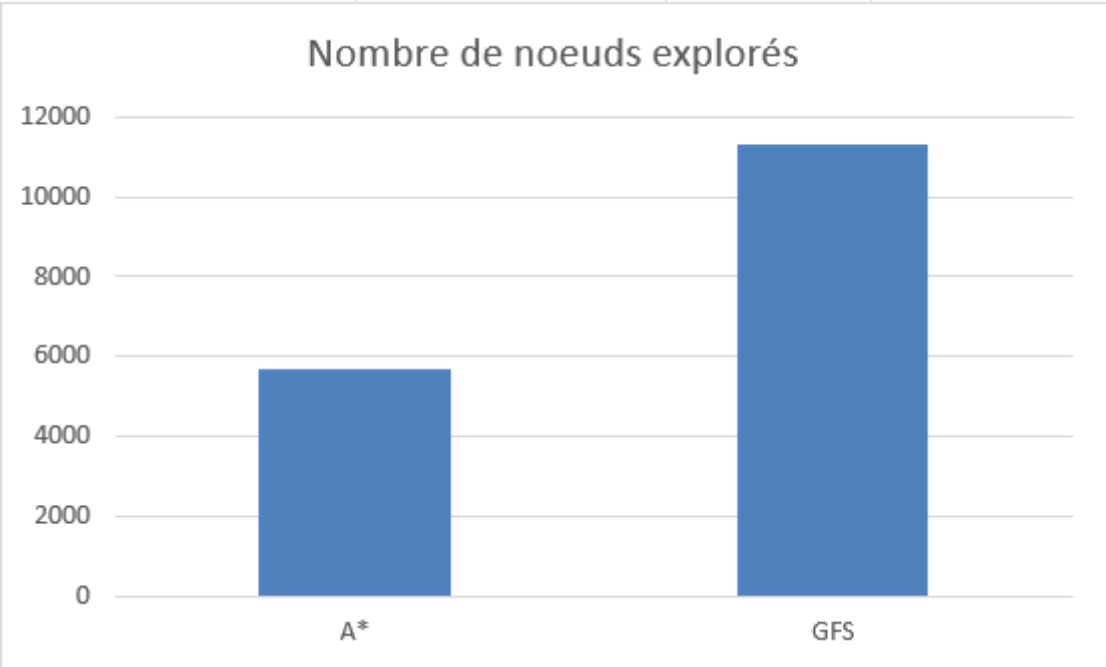
La différence majeure réside dans le coût de la solution : A* trouve une solution bien plus optimale (3599 contre 7355 pour GFS), confirmant que GFS ne garantit pas une solution de coût minimal.

En revanche, GFS est **bien plus rapide en temps d'exécution** (0,005s contre 0,052s), ce qui confirme son efficacité en termes de rapidité, mais au détriment de la qualité de la solution.

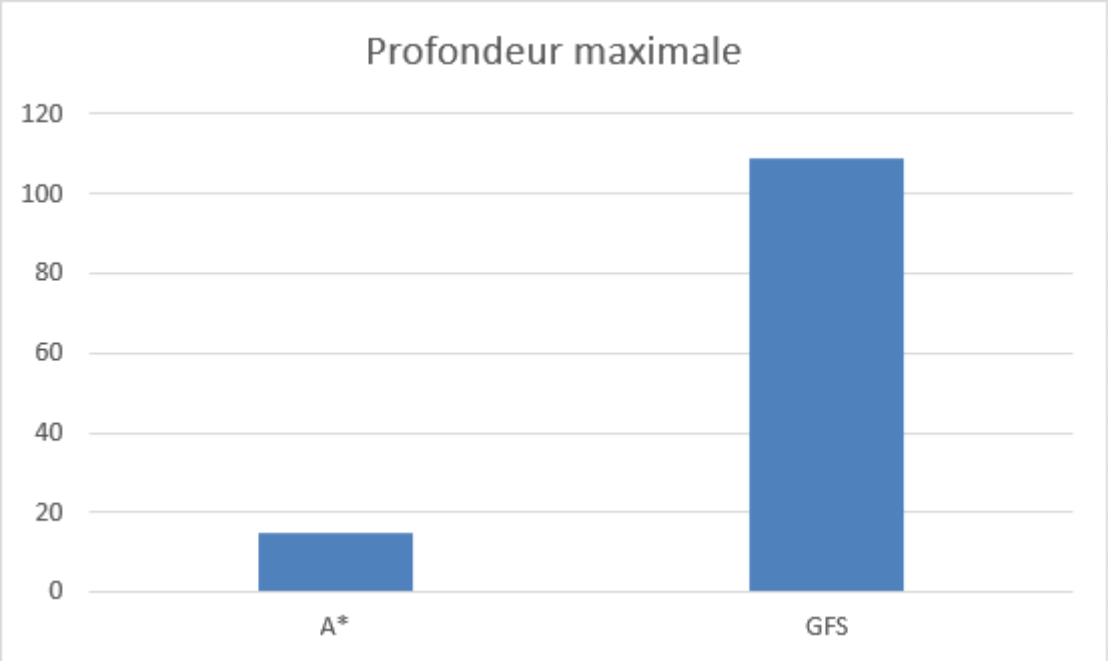
Augmentons un peu la difficulté, pour atteindre n = 1000, et un facteur k = 3.

Problème dum : difficulté croissante

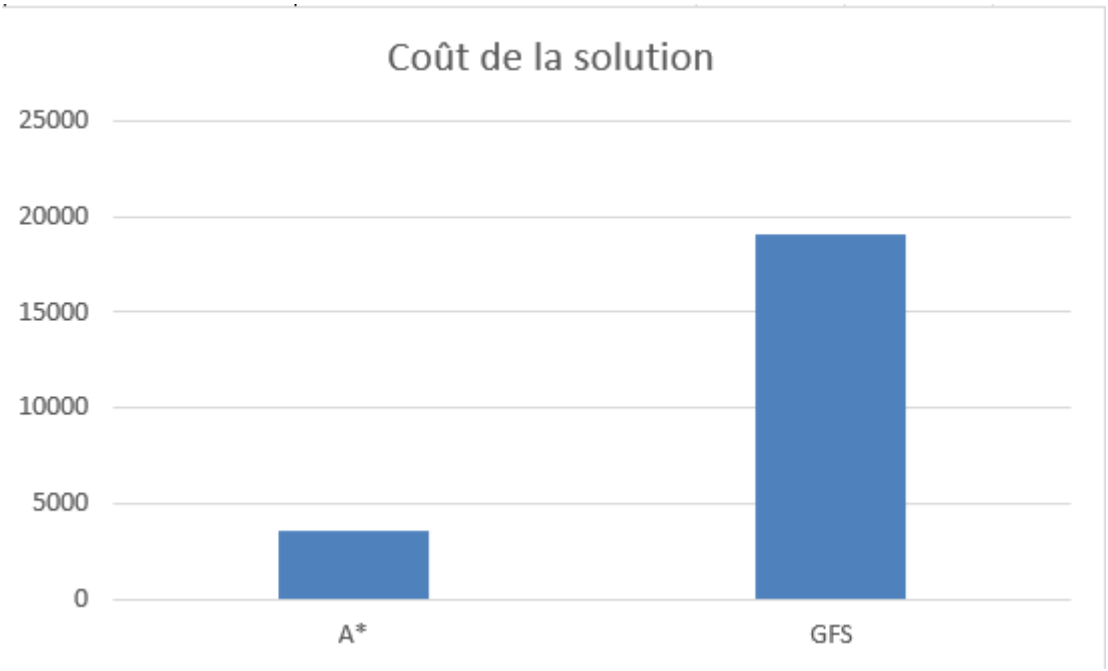
Nombre de nœuds explorés



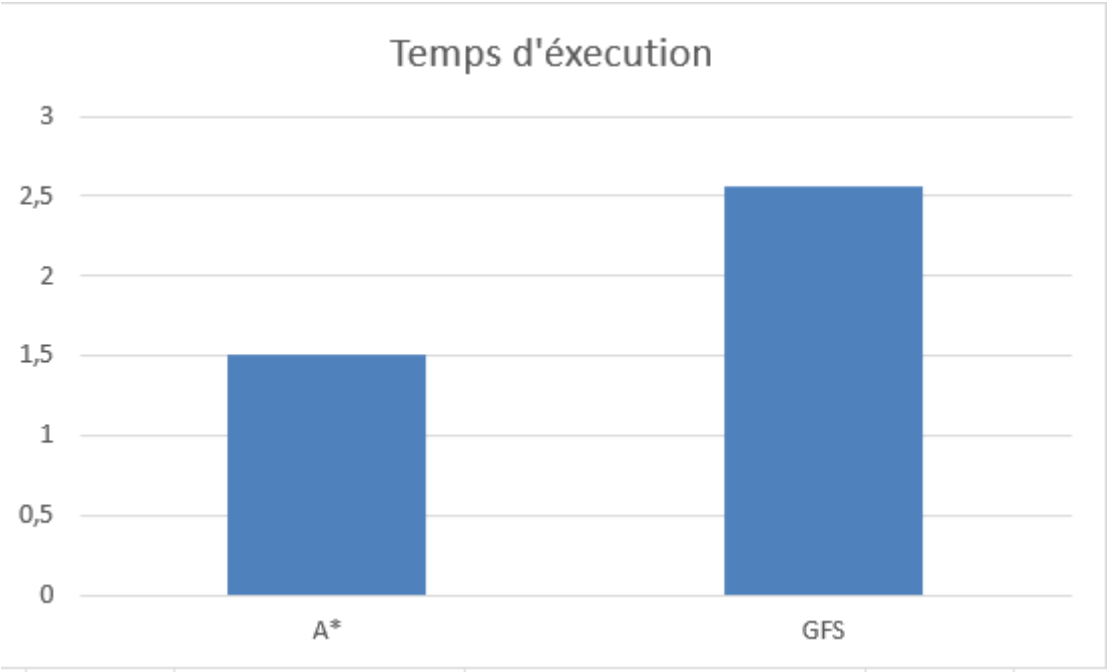
Profondeur maximale



Coût de la solution

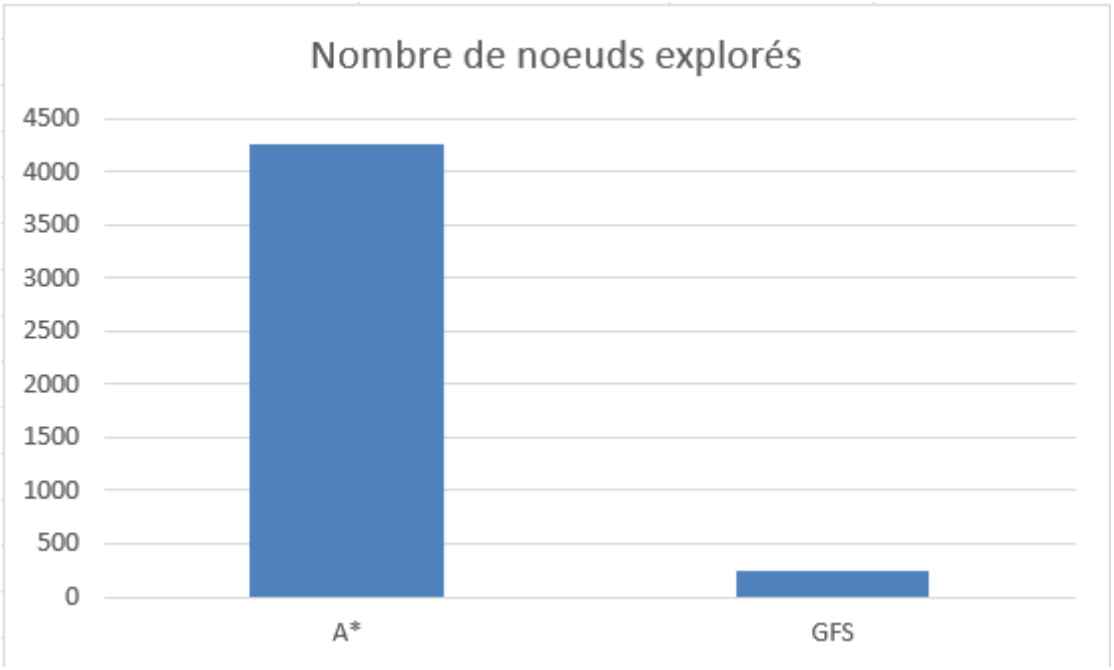


Temps d'exécution

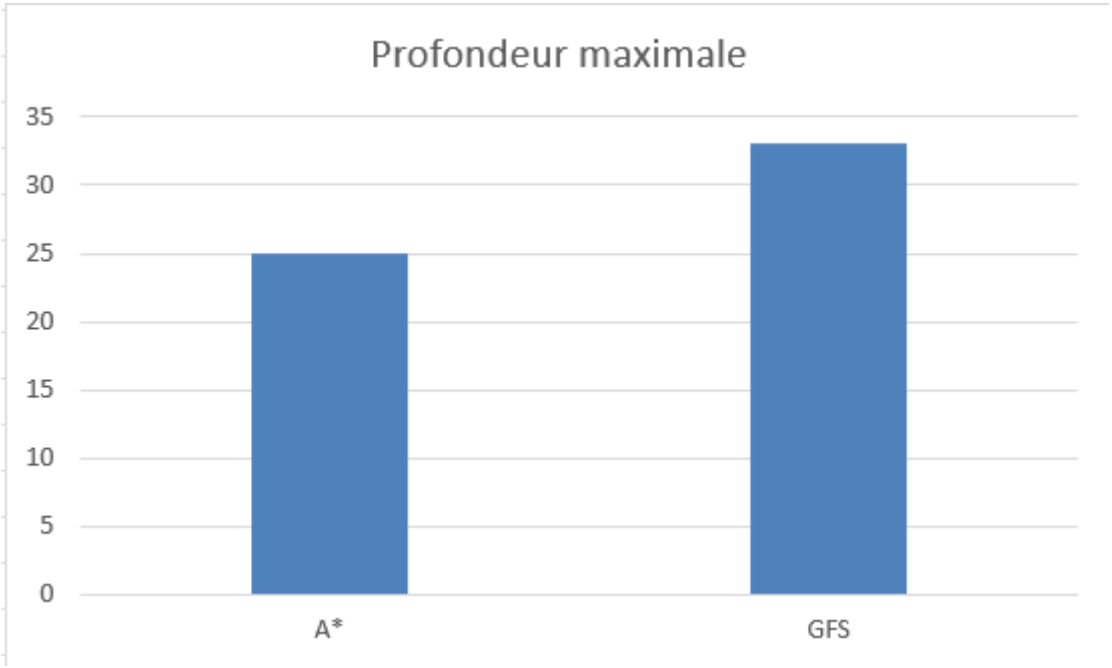


Problème puz

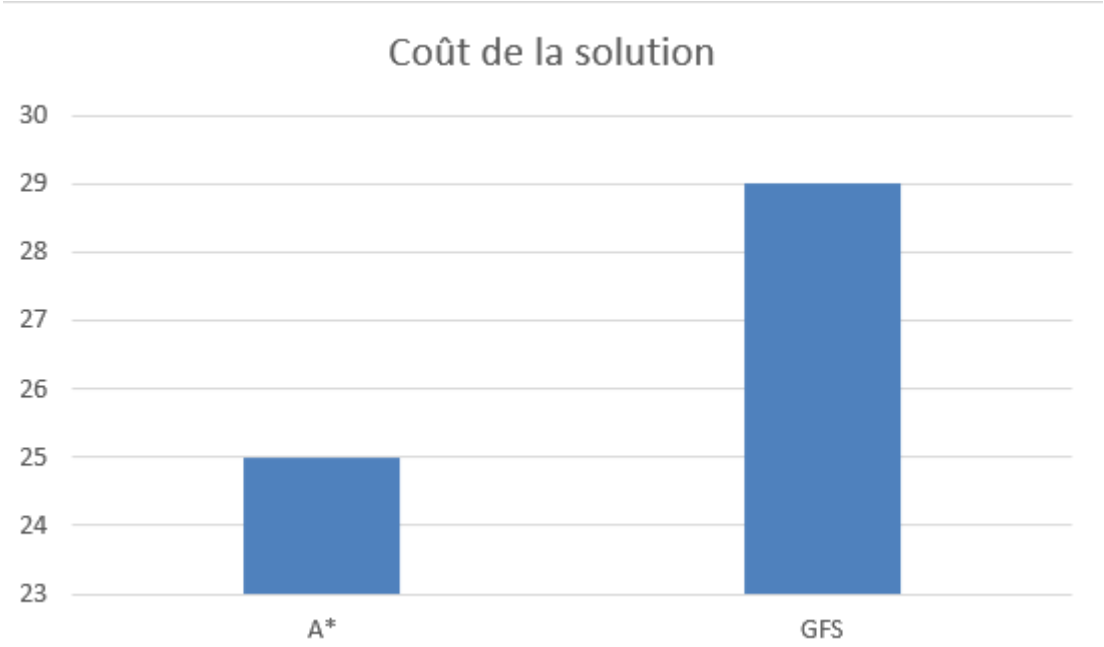
Nombre de nœuds explorés



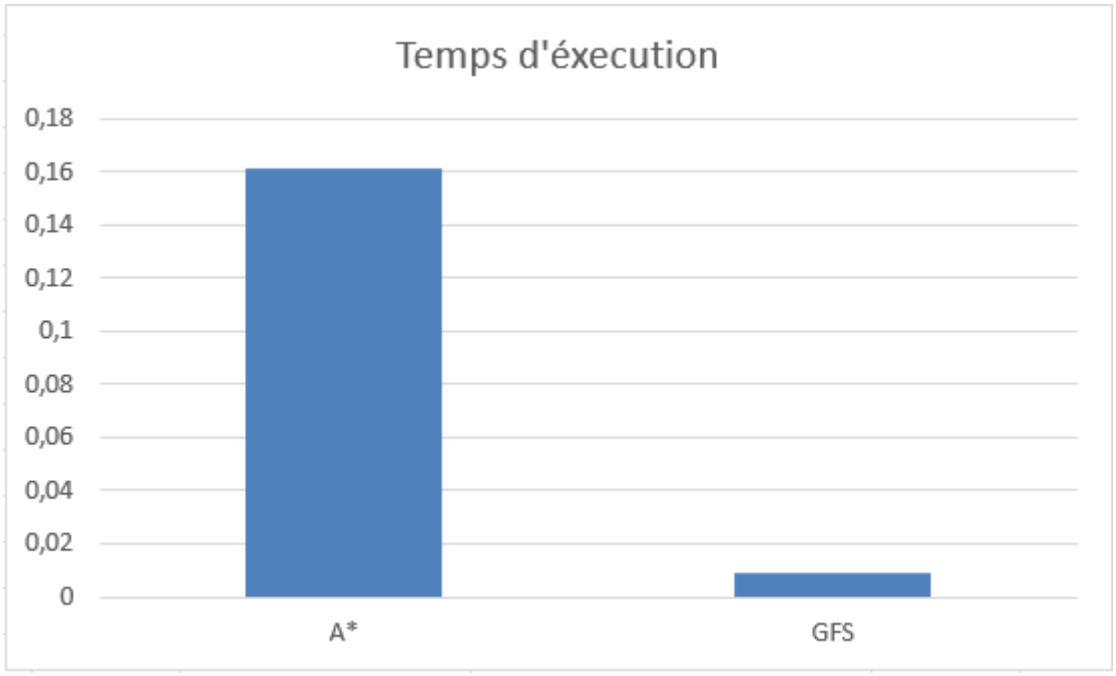
Profondeur maximale



Coût de la solution



Temps d'exécution

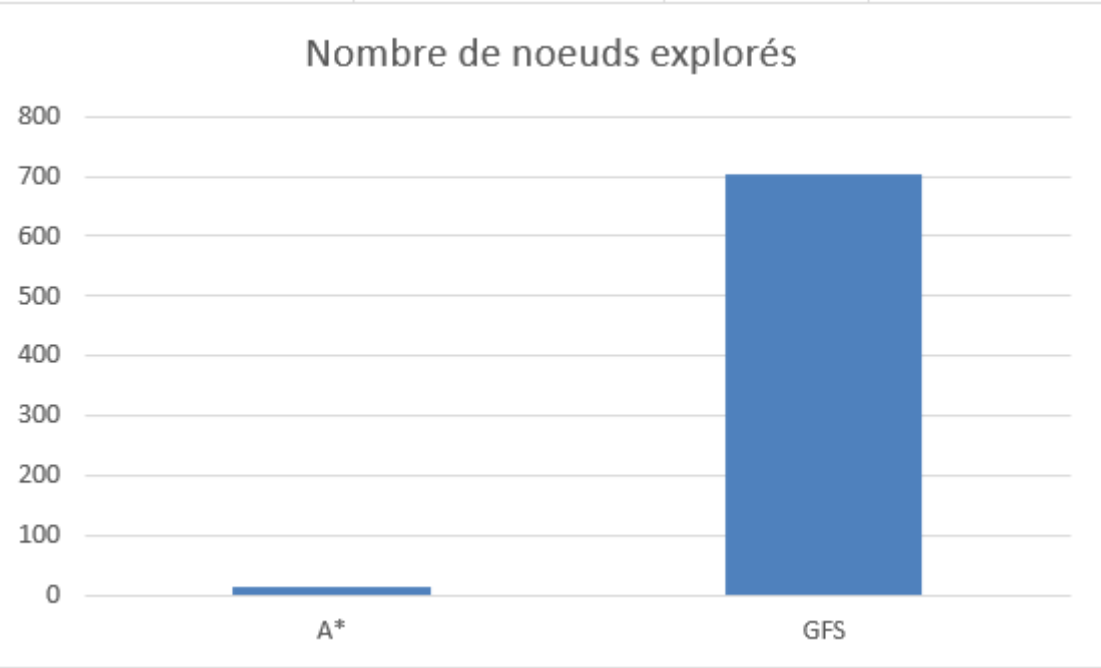


A* trouve une meilleure solution mais prend plus de temps car il évalue mieux les chemins. GFS est très rapide, mais peut donner une solution sous-optimale. Dans un problème comme puz, où l'optimisation est cruciale, A* est préférable malgré un coût en temps plus élevé.

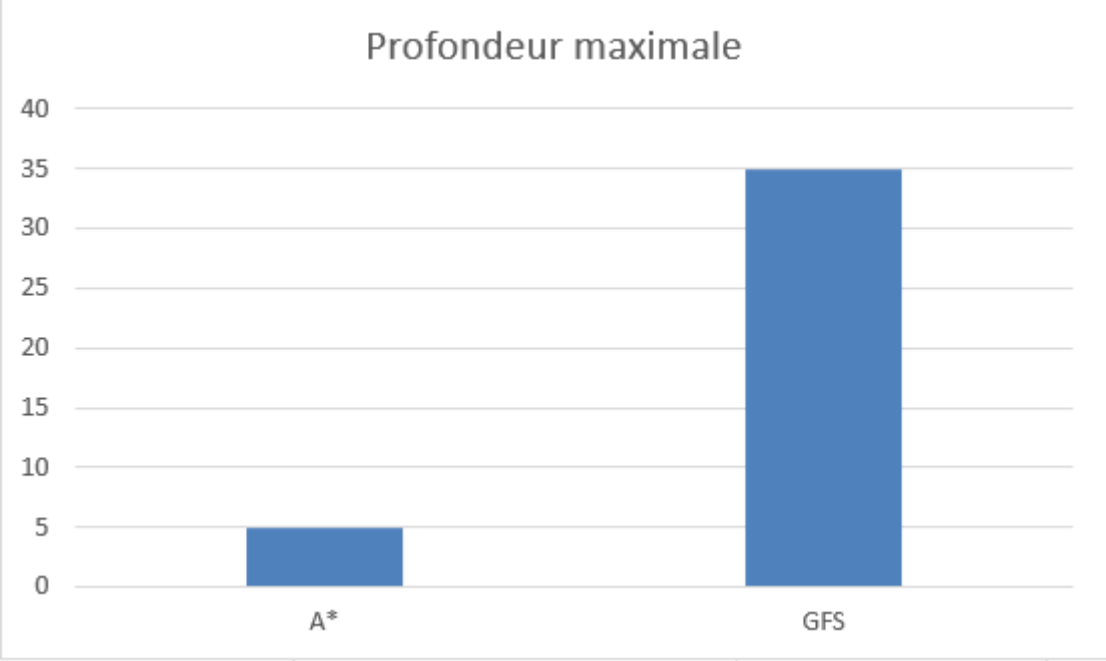
Problème puz : difficulté croissante

avec n = 100 000, k = 5

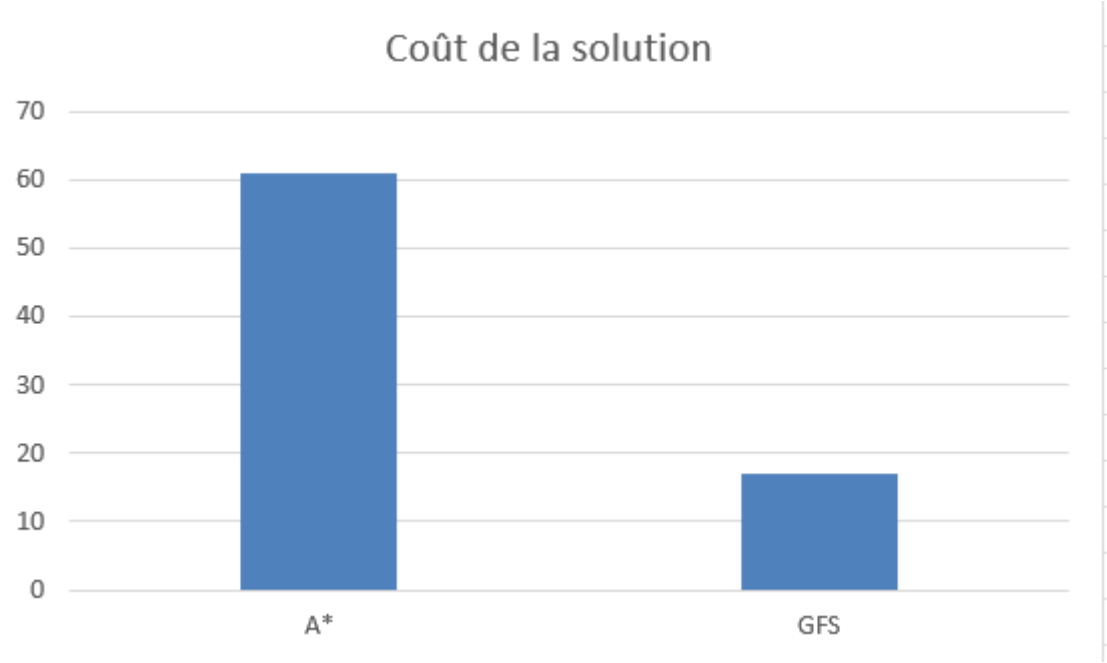
Nombre de nœuds explorés



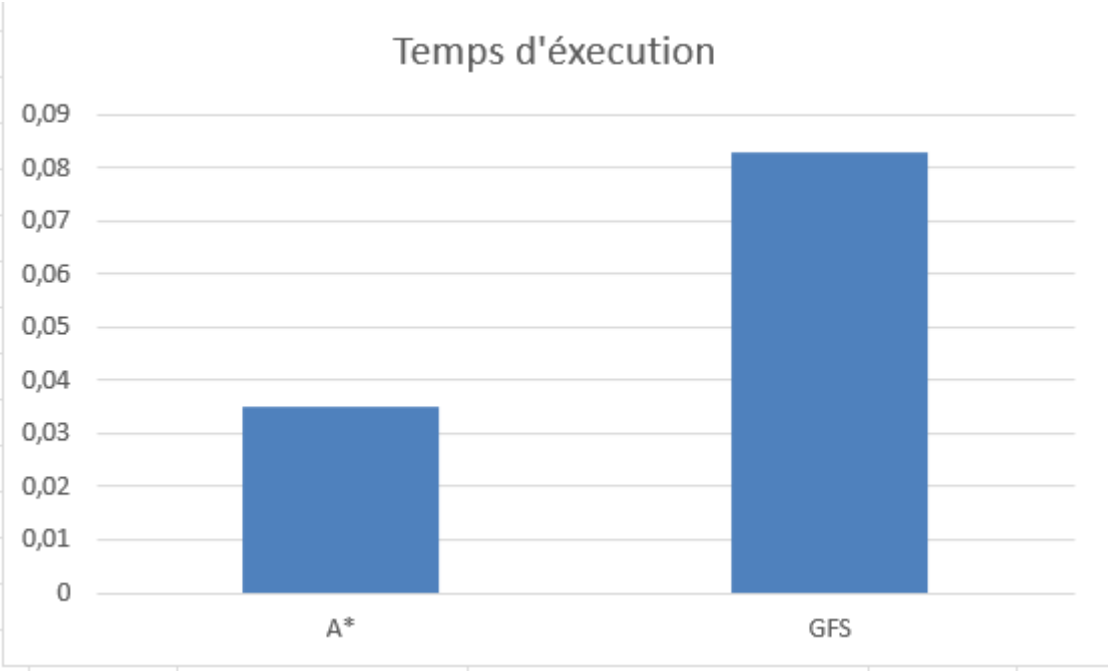
Profondeur maximale



Coût de la solution



Temps d'exécution



A* est nettement plus efficace en termes de nombre de nœuds explorés et de temps d'exécution.

GFS trouve une solution à coût plus bas, mais son comportement reste moins fiable car il ne garantit pas une solution optimale et explore plus de nœuds inutilement.

A* vs UCS

Nous allons maintenant comparer **A*** et **UCS** afin d'évaluer l'impact de l'heuristique dans la recherche de solutions optimales.

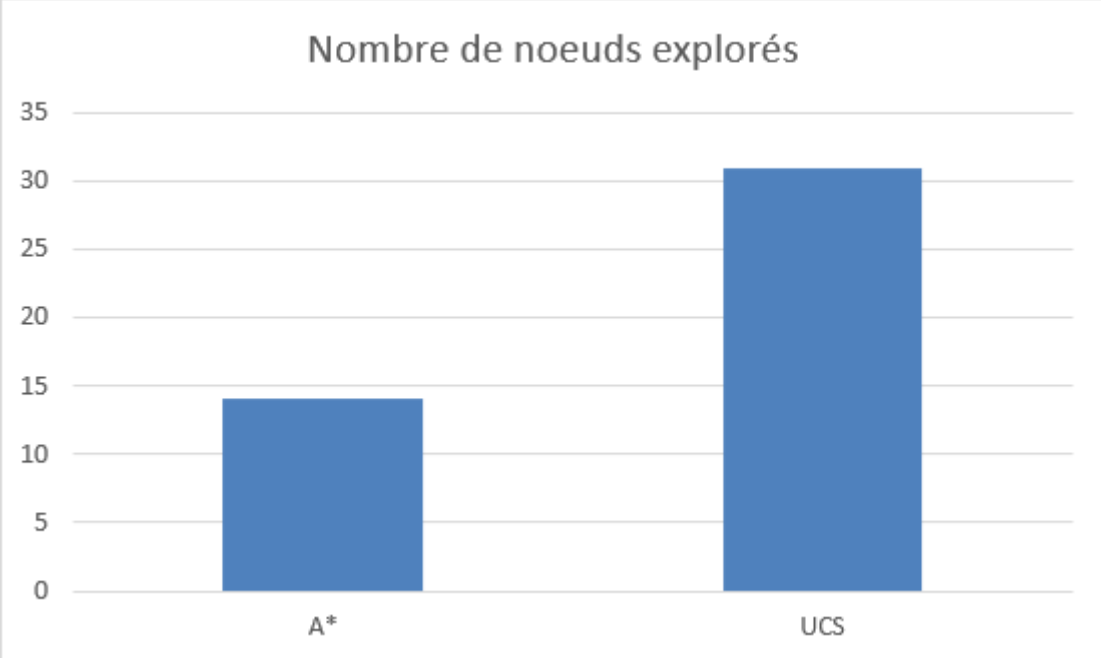
Pour cela, nous allons tester plusieurs types de problèmes :

- **map**
- **vac**
- **dum**
- **puz**

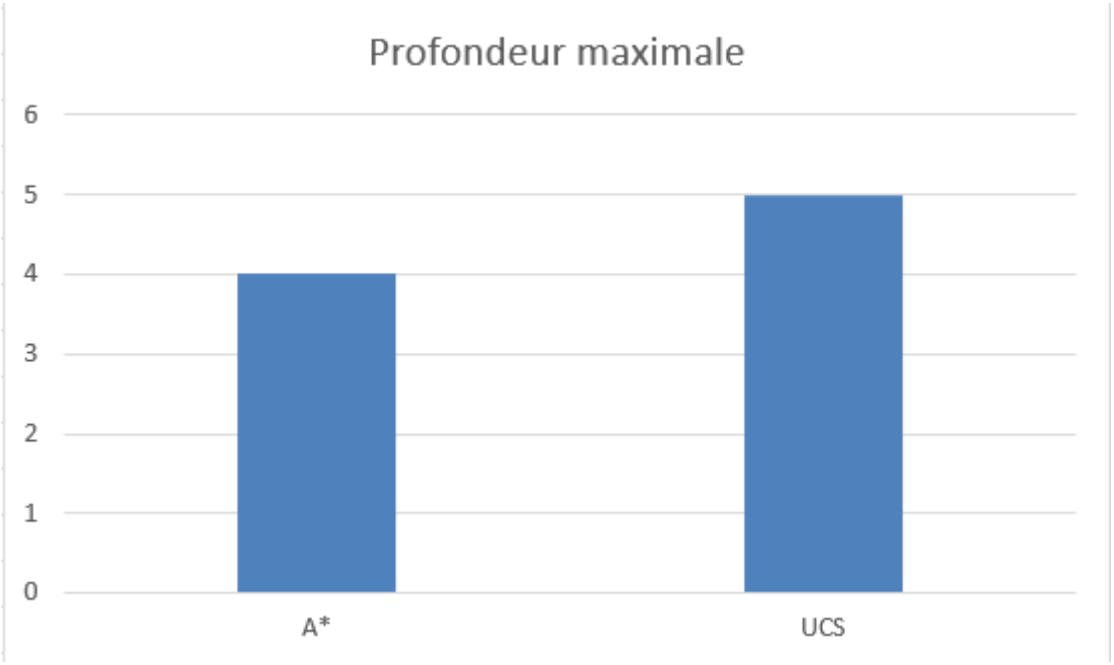
L'objectif est de voir dans quelles conditions A* offre un gain significatif par rapport à UCS.

Problème map

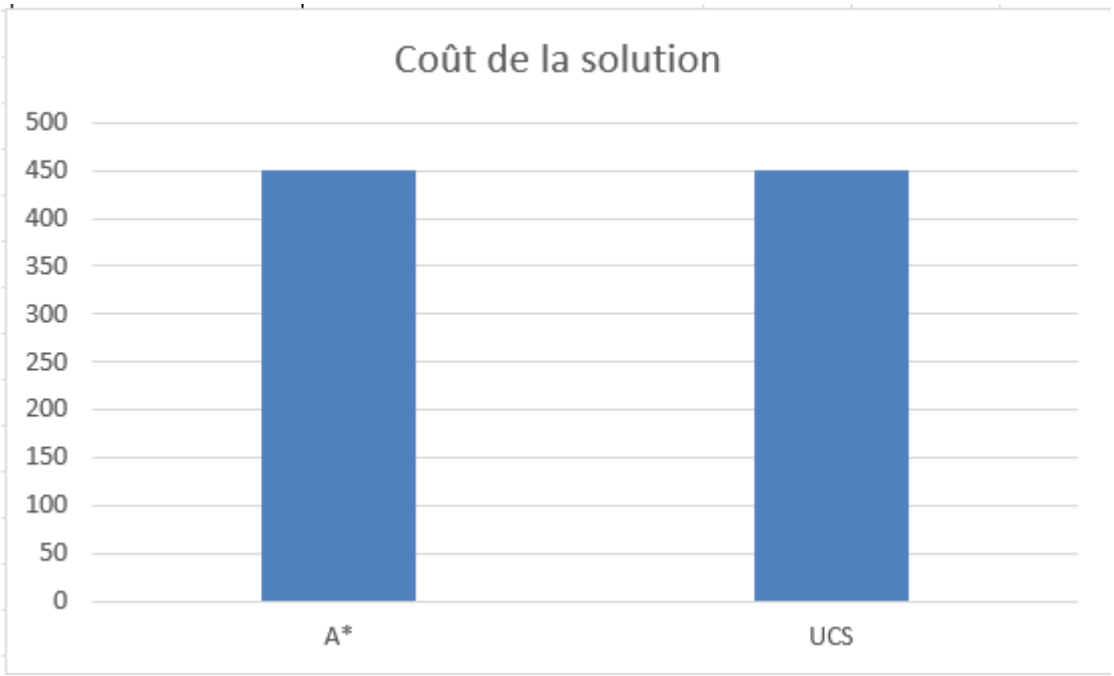
Nombre de nœuds explorés



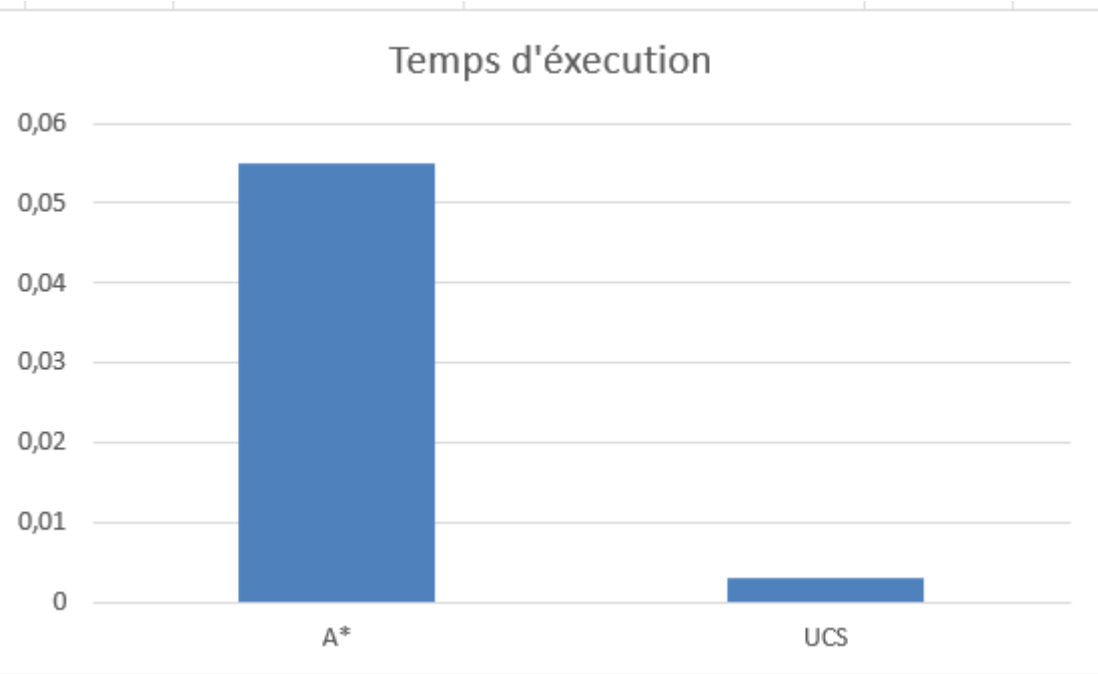
Profondeur maximale



Coût de la solution



Temps d’exécution



L'algorithme **A*** a exploré **moins de nœuds** (14 contre 31 pour UCS), ce qui montre l’impact positif de l’heuristique dans la réduction de l’espace de recherche. Par contre, la profondeur maximale atteinte est légèrement inférieure pour **A***.

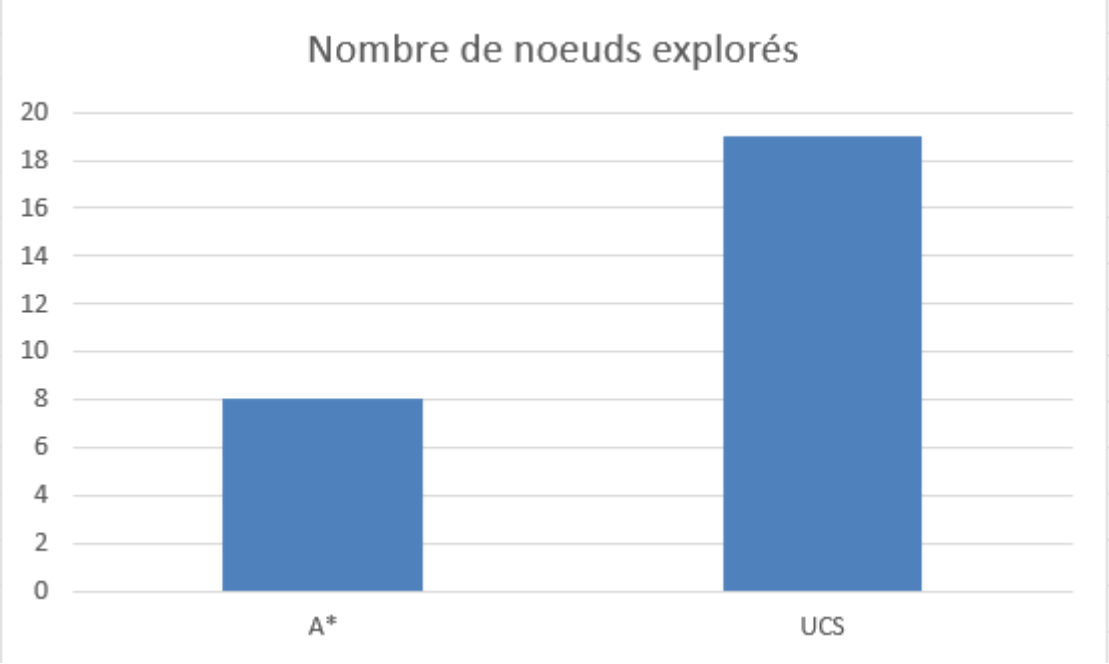
Concernant le **coût de la solution**, les deux algorithmes donnent la même valeur (**450**), ce qui est logique puisque **UCS** garantit l’optimalité, et **A***, avec une bonne heuristique, devrait toujours converger vers le même résultat.

Enfin sur le **temps d’exécution**, UCS est significativement plus rapide (0.003 sec contre 0.055 sec pour A*), ce qui est assez surprenant en réalité étant donné que A* a exploré moins de nœuds.

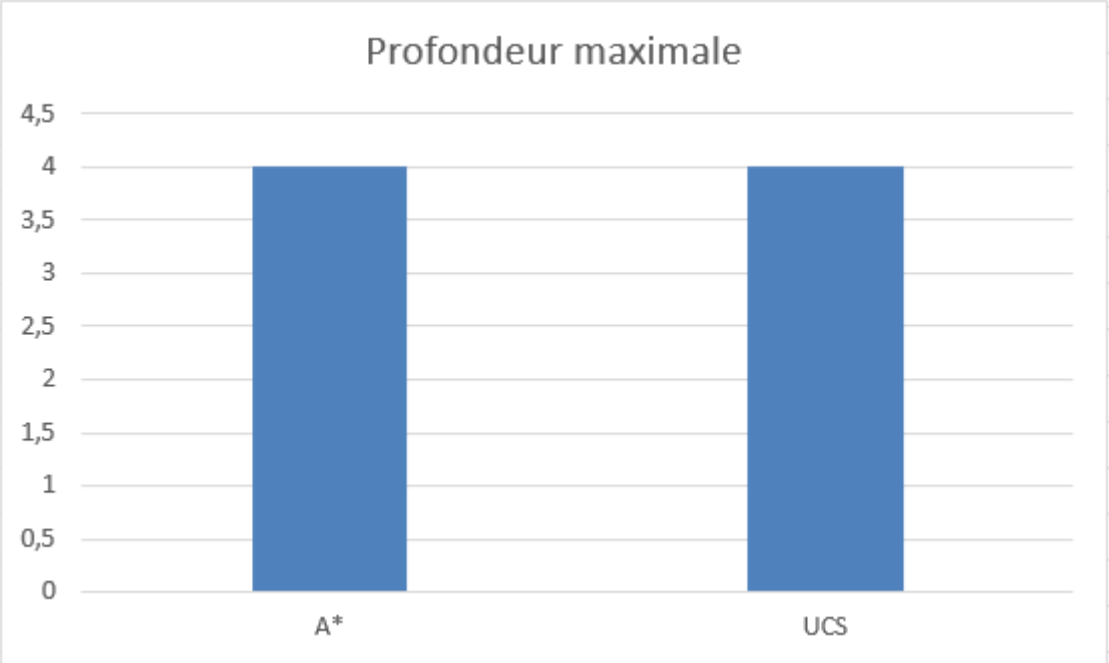
En résumé, **A*** explore plus intelligemment et trouve le même chemin optimal que **UCS**, mais avec un léger surcoût en temps d’exécution. Ce point devra être observé sur des problèmes plus complexes pour voir si l’avantage d’**A*** en termes de nœuds explorés se traduit réellement par un gain de performance.

Probleme vac

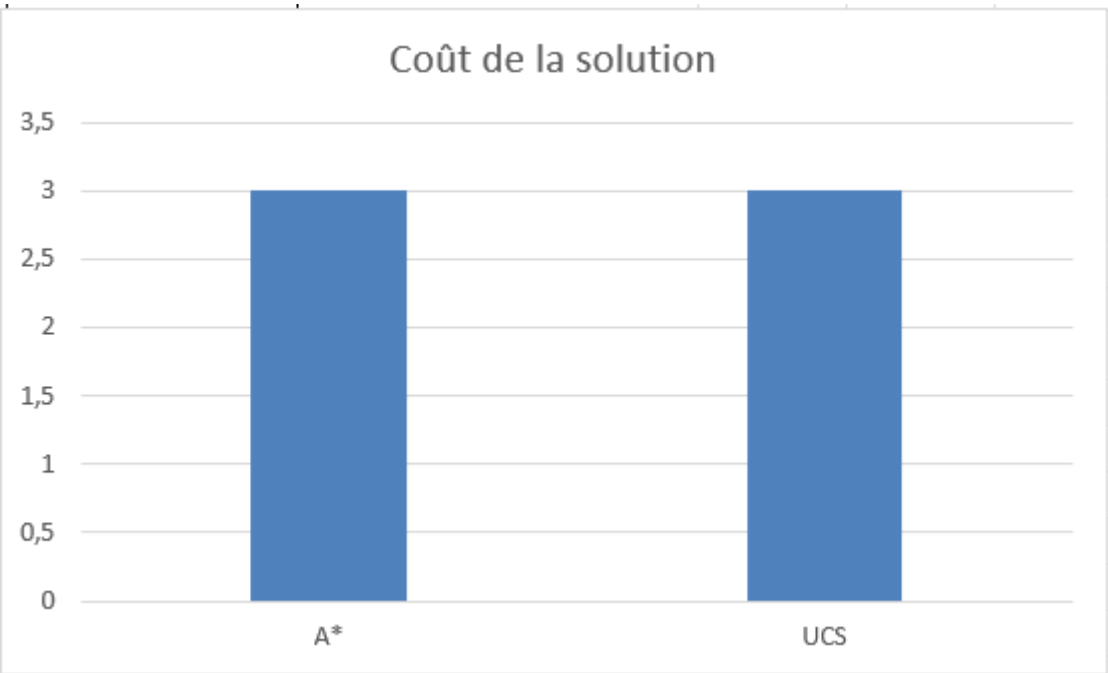
Nombre de nœuds explorés



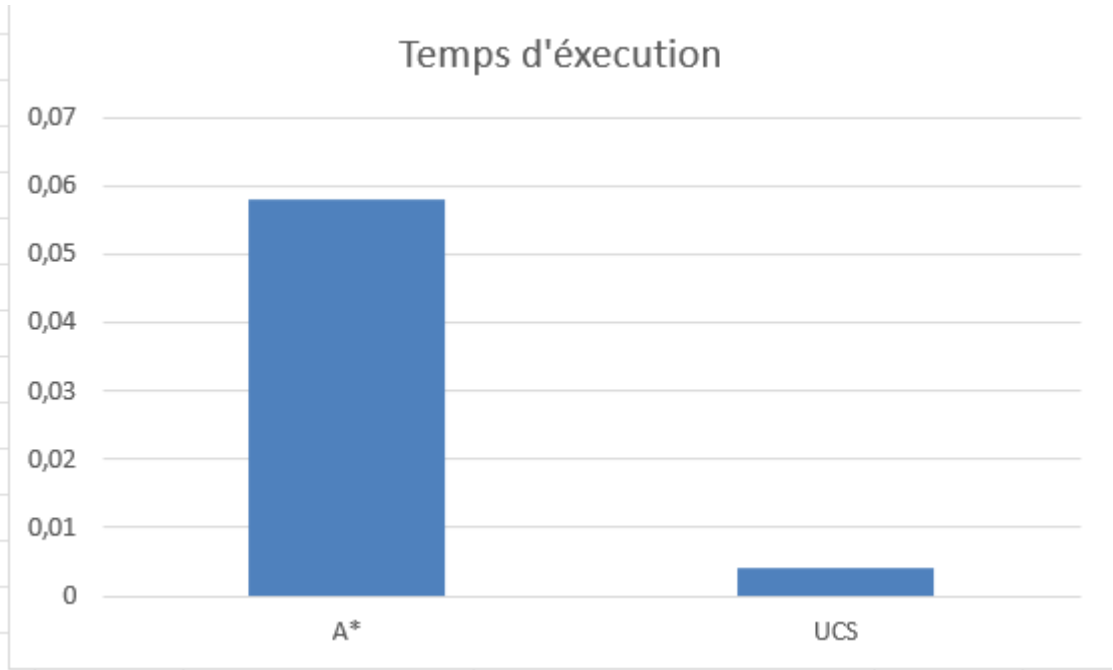
Profondeur maximale



Coût de la solution



Temps d'exécution



Sur ce problème, **A*** explore **moins de nœuds** (**8** contre **19** pour UCS), ce qui montre que l'heuristique aide bien à guider la recherche cette fois. Mais la **profondeur maximale** reste identique (**4**), ce qui suggère que les deux algorithmes trouvent le même chemin optimal sans grosse différence.

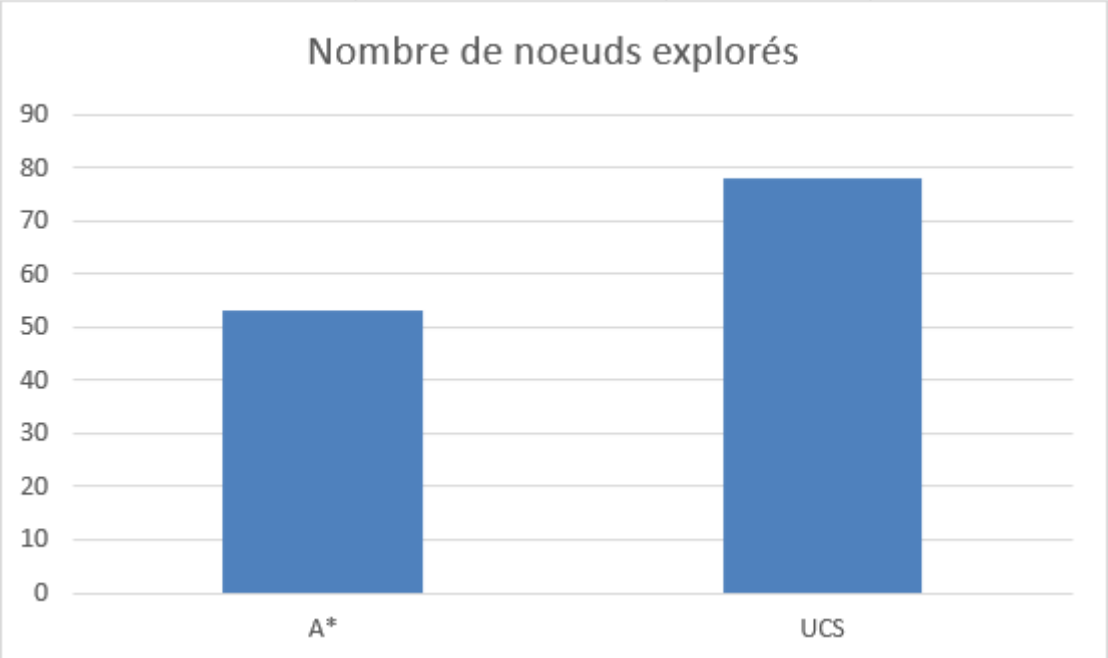
Le **coût de la solution** est pareil (**3** pour les deux).

Par contre en **temps d'exécution UCS** est nettement plus rapide (**0.004 sec** contre **0.058 sec** pour A*).

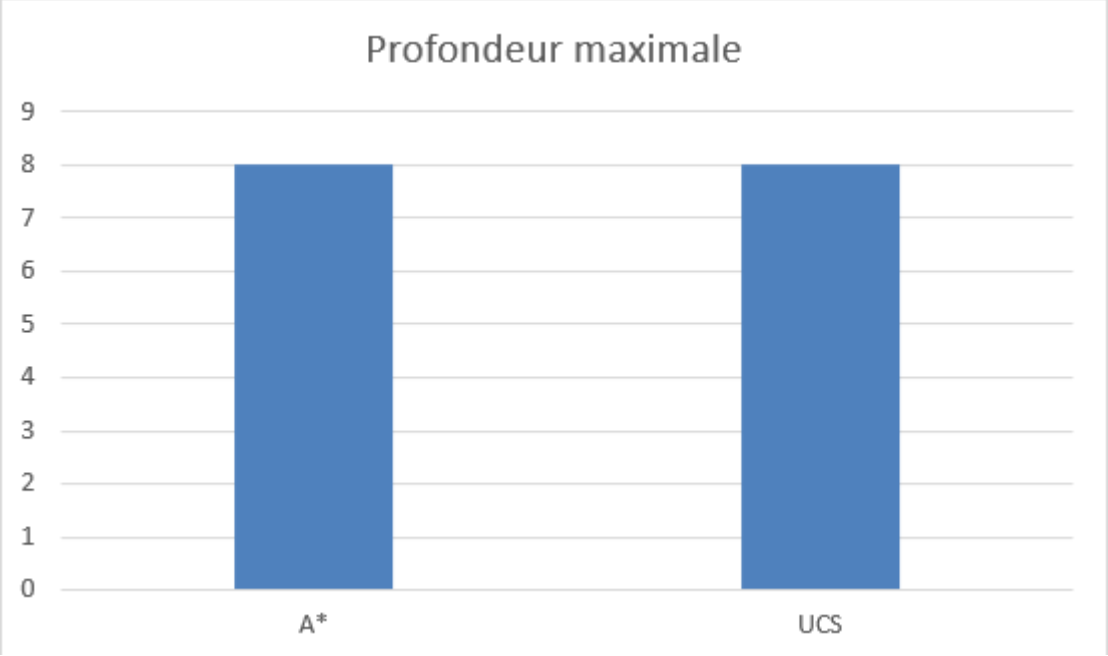
Sur ce type de problème, où l'espace d'états est limité et où le chemin optimal est facilement trouvable, l'heuristique d'**A*** n'apporte pas un gain décisif, et **UCS reste plus efficace en temps de calcul.**

Probleme dum

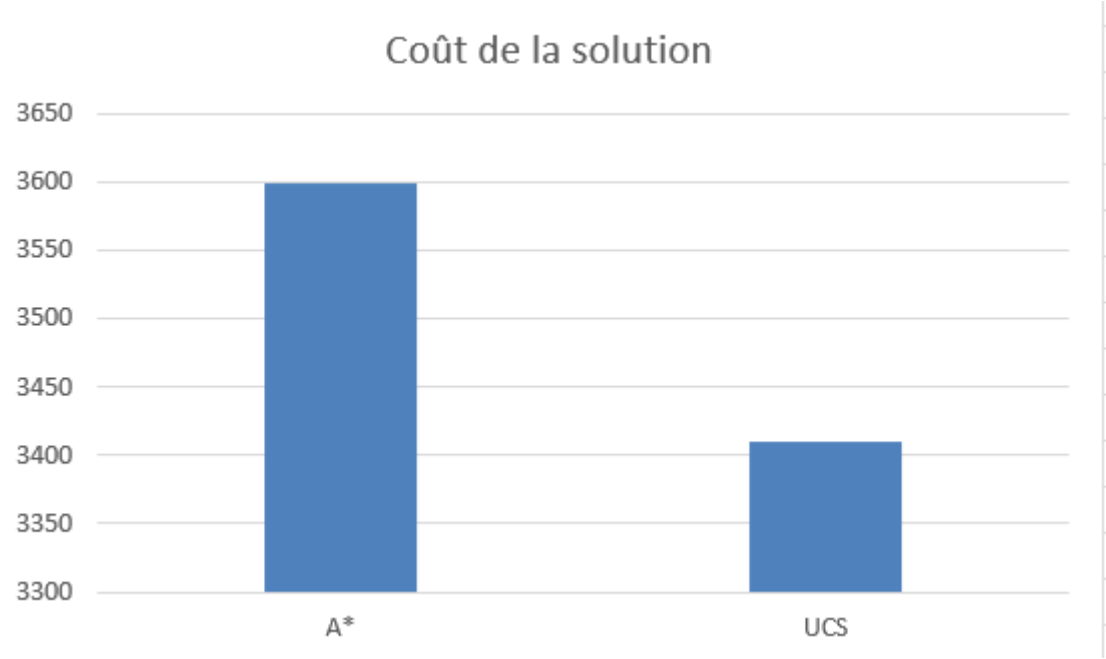
Nombre de nœuds explorés



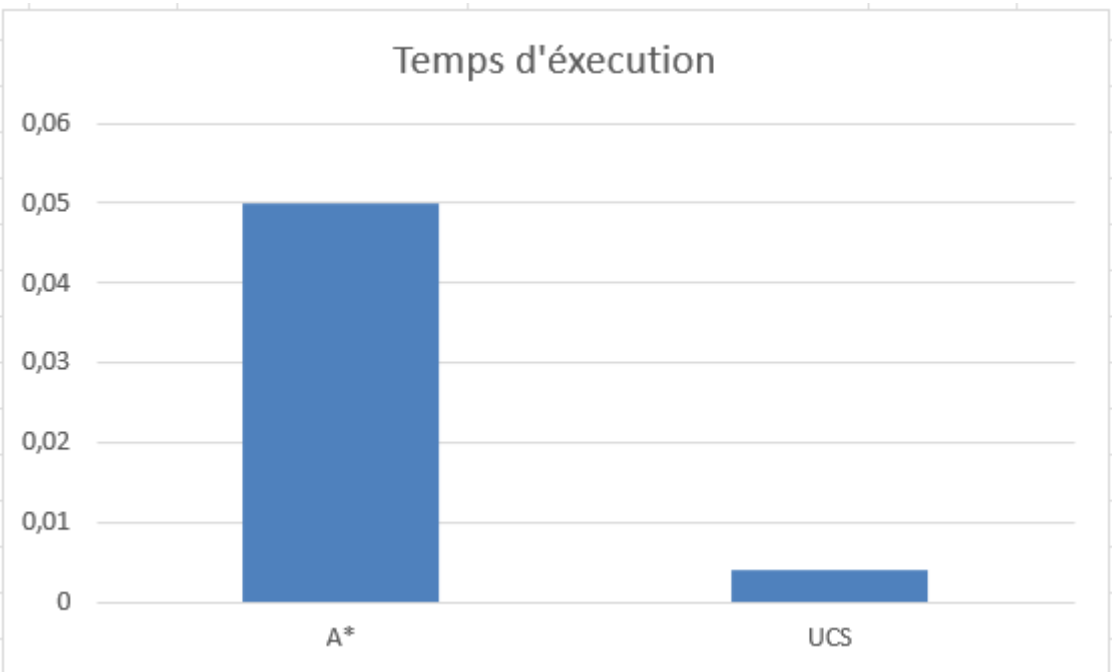
Profondeur maximale



Coût de la solution



Temps d'exécution



Problème dum, très grosse difficulté

Nous avons tenté l'expérience d'un très gros problème avec dum. Afin d'évaluer les performances des algorithmes sur une instance particulièrement complexe, nous avons laissé tourner l'exécution pendant deux heures pour comparer leurs comportements respectifs.

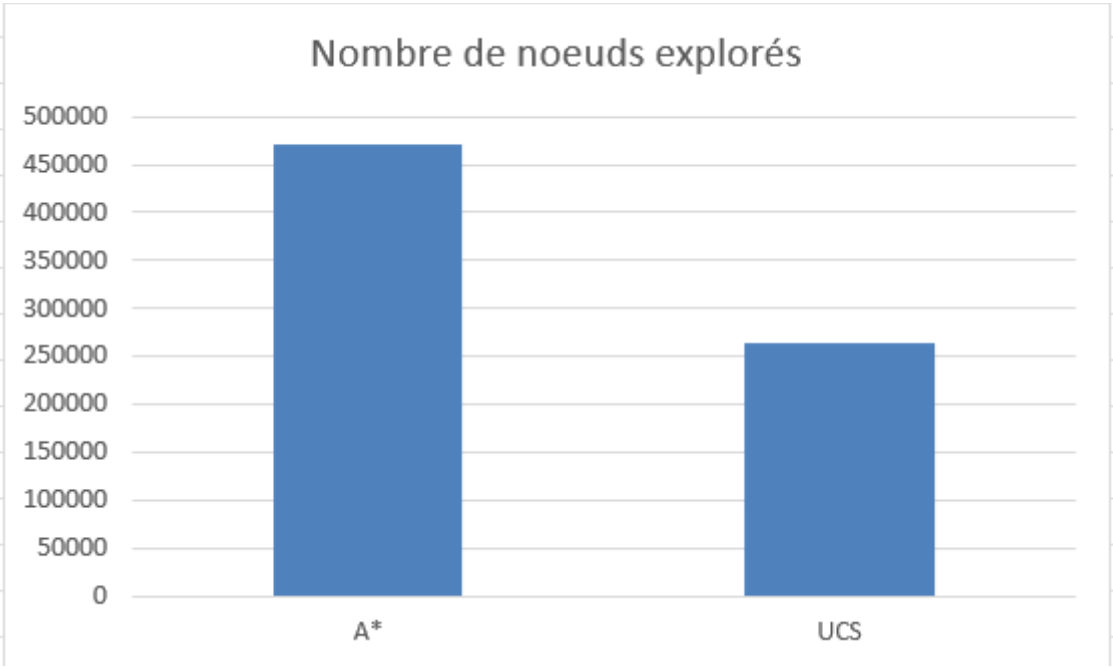
Exécution avec UCS :

```
PS C:\Users\marse\Documents\SAE_IA\src> java LancerRecherche.java -prob dum -algo ucs -n 100000 -k 5
Instance de Dummy crée avec 100000 noeuds et un facteur de branchement de 5. Graine aléatoire 1234
Solution: {0} > Goto 2617 > Goto 70558 > Goto 37117 > Goto 12997 > Goto 94331 > Goto 89767 > Goto 40994 > Goto 60882 > Goto 99999 {99999}
Solved ! Explored 471832 nodes. Max depth was 18. Solution cost is 3434.7652601318714
Temps n'écessaire 4092.187 sec.
```

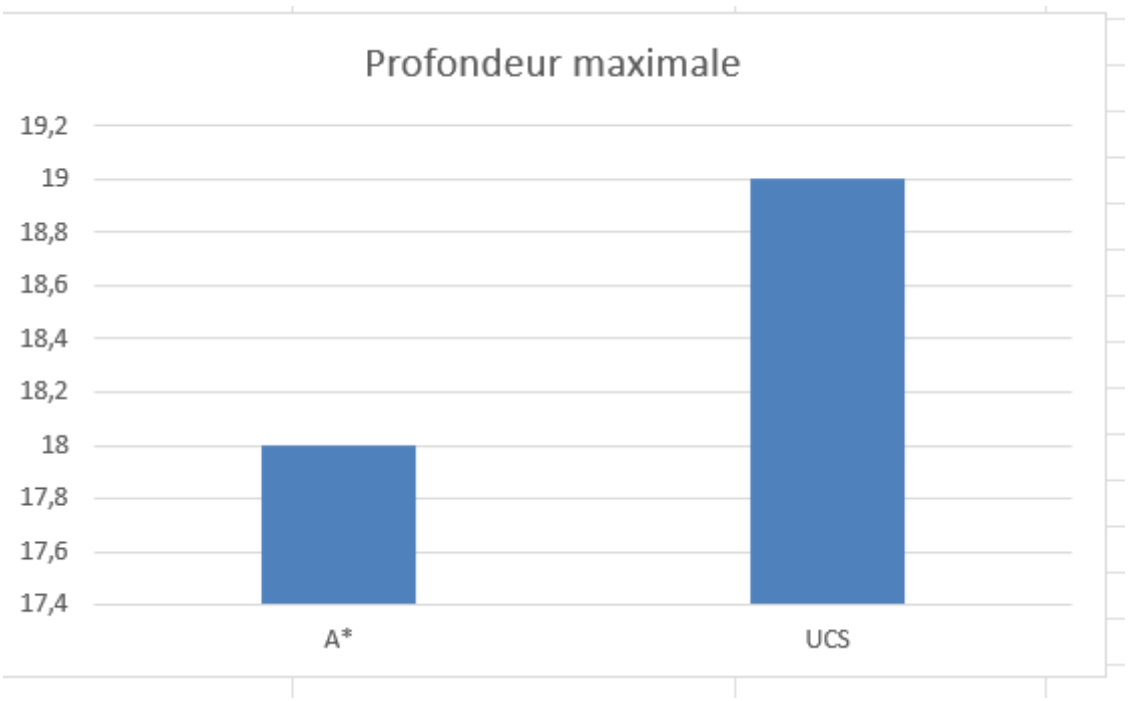
Exécution avec A* :

```
PS C:\Users\marse\Documents\SAE_IA\src> java LancerRecherche.java -prob dum -algo astar -n 100000 -k 5
Instance de Dummy crée avec 100000 noeuds et un facteur de branchement de 5. Graine aléatoire 1234
Solution: {0} > Goto 20771 > Goto 60900 > Goto 41157 > Goto 28901 > Goto 61520 > Goto 46294 > Goto 30259 > Goto 37614 > Goto 80993 > Goto 60882 > Goto 99999 {99999}
Solved ! Explored 264514 nodes. Max depth was 19. Solution cost is 4617.94720402011
Temps n'écessaire 4365.532 sec.
```

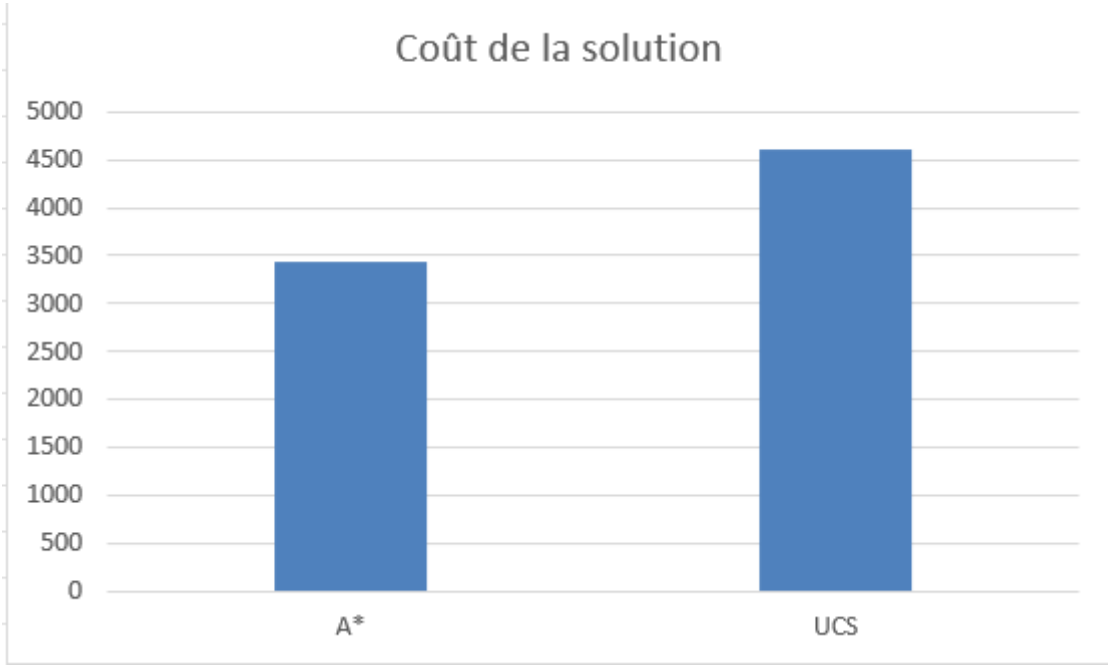
Nombre de nœuds explorés



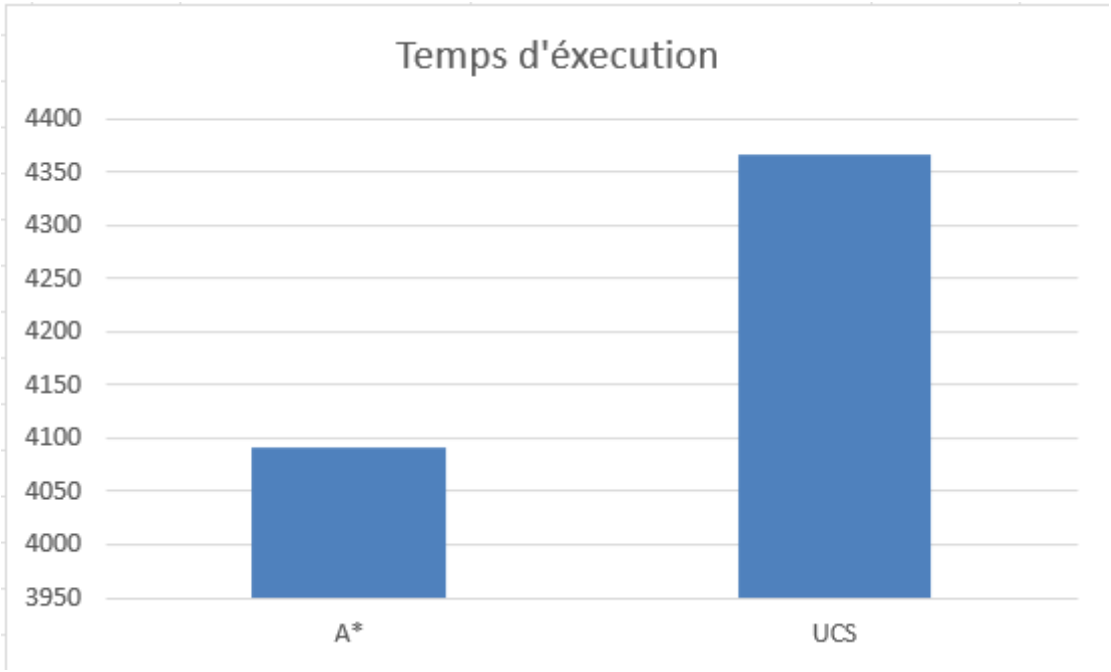
Profondeur maximale



Coût de la solution



Temps d’exécution



L'expérience sur un problème de très grande taille avec dum montre plusieurs tendances marquantes entre A* et UCS. D’abord, bien que A* explore un nombre plus important de nœuds que UCS (471832 contre 264514 ici), il arrive à trouver une solution de meilleure qualité, avec un coût inférieur (3434 contre 4617). Ca confirme que l'heuristique utilisée dans A* lui permet de mieux guider la recherche vers une solution optimisée.

Concernant la profondeur maximale, les valeurs restent assez proches (18 pour A* contre 19 pour UCS), ce qui montre que les deux algorithmes ont suivi des chemins relativement identiques en termes de nombre d'étapes. Par contre, la différence de coût souligne l'efficacité de A* pour minimiser la distance parcourue.

Enfin le temps d'exécution reste élevé dans les deux cas (plus de 4000 secondes !!!), avec un léger avantage pour A*, qui termine plus rapidement qu'UCS. On peut expliquer ça par l'optimisation offerte par l'heuristique.

En conclusion, A* s'avère plus efficace sur des problèmes de grande ampleur en réduisant le coût de la solution tout en conservant une profondeur de recherche comparable et un temps d'exécution légèrement inférieur à celui de UCS.

Étudier les algorithmes de jeux

Nous allons commencer par comparer MinMax et Alpha-Beta sur un jeu simple : **Tic-Tac-Toe**. Ce jeu est un excellent point de départ car il permet d'observer clairement la différence entre ces deux algorithmes sans nécessiter une puissance de calcul excessive.

Nous avons vu que l'algorithme MinMax explore systématiquement l'ensemble des possibilités jusqu'à une profondeur donnée, alors que Alpha-Beta optimise cette recherche en éliminant les branches non prometteuses. L'objectif est de voir si Alpha-Beta permet une réduction significative du nombre d'états explorés sans affecter la qualité du jeu.

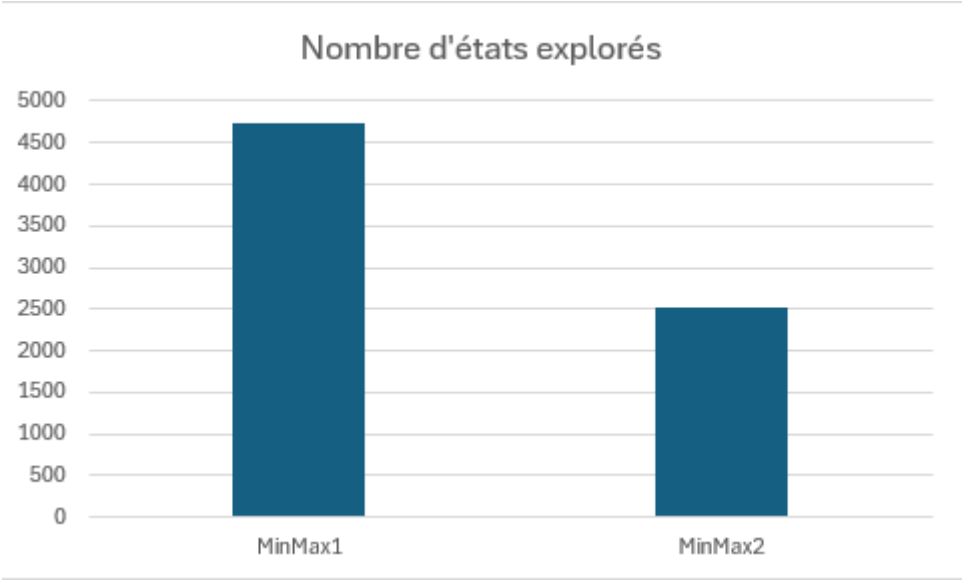
Exécutions Faciles

Ces tests utilisent des jeux et des paramètres qui permettent des exécutions relativement rapides. L'objectif ici est de comparer les algorithmes sur des instances dont l'espace de recherche reste correct grâce à une profondeur limitée ou une taille réduite du plateau avec les options données.

TicTacToe – MinMax vs MinMax

MATCH NUL

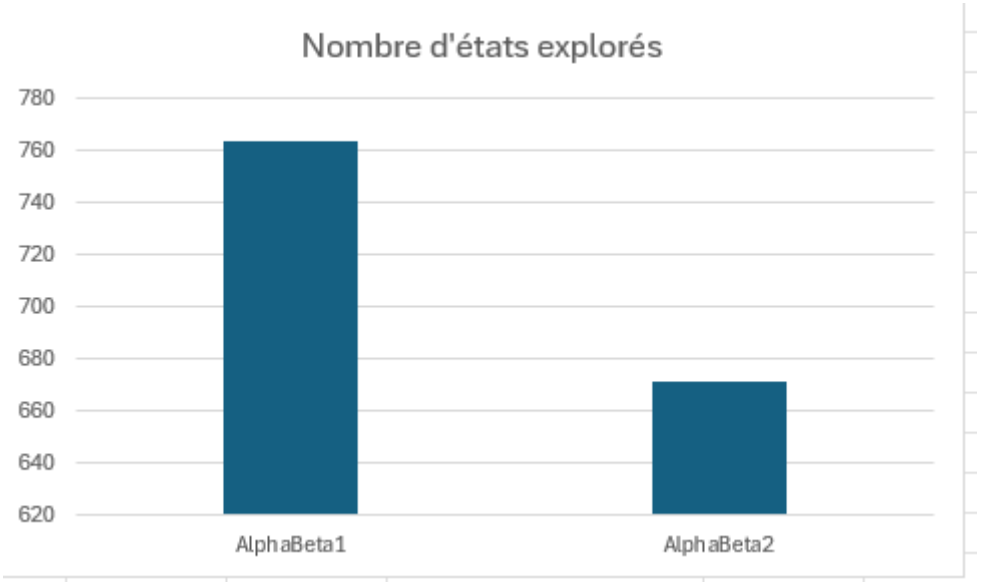
Temps de calcul = 0.164s



TicTacToe – AlphaBeta vs AlphaBeta

MATCH NUL

Temps de calcul = 0.245s



Les résultats obtenus montrent clairement l'impact de Alpha-Beta par rapport à MinMax.

- **MinMax** explore **4743** et **2526** états respectivement pour les deux, ce qui montre une exploration complète de l'espace de jeu. Cette approche garantit une prise de décision optimale, mais au prix d'un coût très élevé.
- **Alpha-Beta** par contre réduit énormément le nombre d'états explorés (**764** et **671**), ce qui confirme son efficacité à couper les branches inutiles et à accélérer la recherche sans affecter la qualité de la décision. En plus le temps d'exécution est un peu amélioré malgré la complexité supplémentaire.

Finalement, même sur un jeu simple comme TicTacToe, Alpha-Beta démontre son intérêt en rendant l'exploration plus efficace.

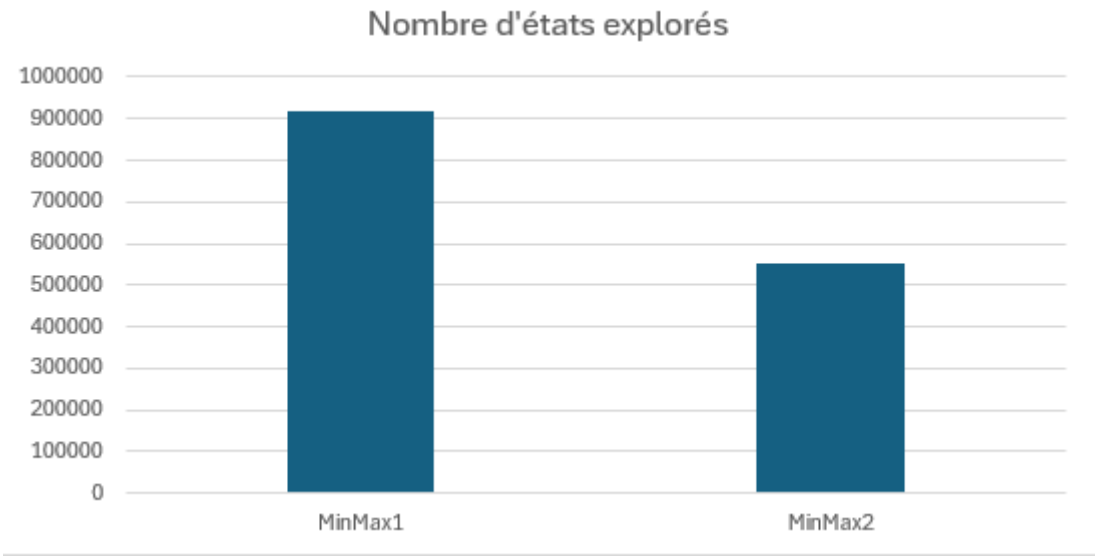
Mnk – MinMax vs MinMax

Nous allons maintenant tester MinMax et Alpha-Beta sur une instance plus complexe : le jeu MnkGame en configuration 5×5 avec 4 en ligne. En limitant la profondeur à 5, on cherche à observer si Alpha-Beta parvient à explorer moins d'états tout en maintenant une performance de jeu équivalente à MinMax.

VAINQUEUR → MinMax1

Temps de calcul = 3.89s

4 coups

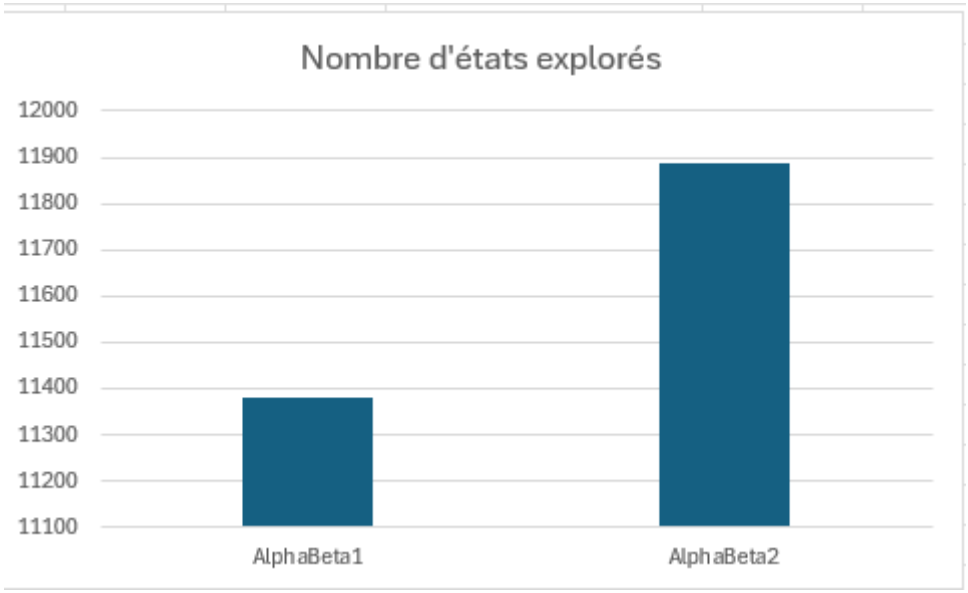


Mnk – AlphaBeta vs AlphaBeta

VAINQUEUR → AlphaBeta1

Temps de calcul = 0.19s

3 coups



La comparaison entre MinMax et Alpha-Beta sur MnkGame (en 5×5, et 4 en ligne) montre parfaitement l'intérêt d'Alpha-Beta. MinMax explore un nombre énorme d'états (plus de **900 000** pour le premier joueur), alors que Alpha-Beta arrive à obtenir une solution en explorant à peine **11 000 états**.

Cette réduction se traduit aussi par une grosse accélération du temps d'exécution : **3,89 secondes pour MinMax contre seulement 0,19 seconde pour Alpha-Beta**. La partie s'est terminée en **moins de coups** pour Alpha-Beta aussi.

Ce test nous montre queAlpha-Beta est une optimisation essentielle pour les jeux qui nécessitent une exploration en profondeur, surtout lorsque l'espace des états devient exponentiellement plus grand.

Connect4 – AlphaBeta vs AlphaBeta

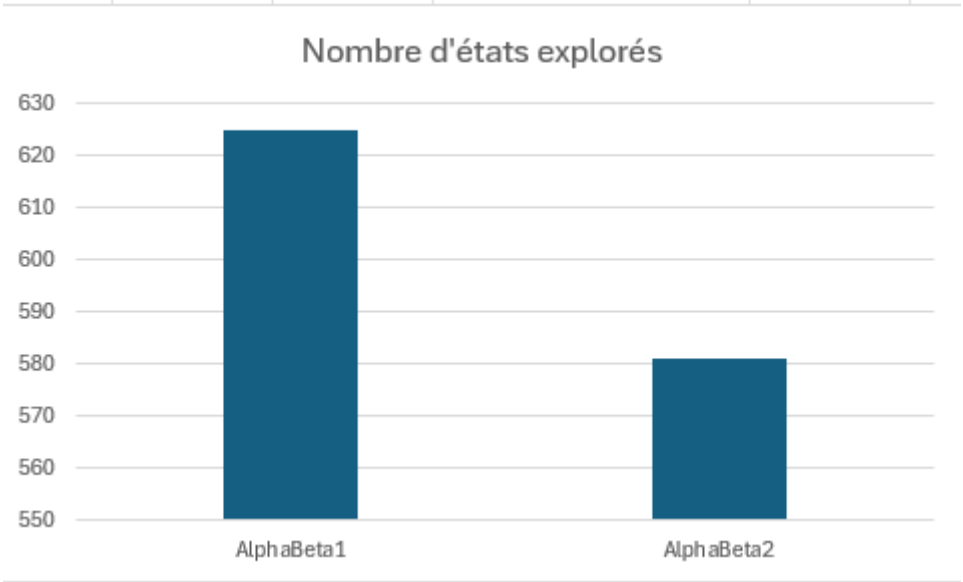
Nous allons maintenant observer l'impact de la **profondeur** sur les performances de l'algorithme Alpha-Beta dans un jeu de complexité modérée comme **Connect4**. En augmentant progressivement la profondeur de recherche, on pourra alors mesurer l'évolution du **temps d'exécution** et du **nombre d'états explorés**. L'objectif est de voir dans quelle mesure l'augmentation de la profondeur affecte la capacité de l'algorithme à prendre des décisions optimales tout en restant efficace.

(-d 4)

VAINQUEUR → AlphaBeta2

Temps de calcul = 0.154s

21 coups

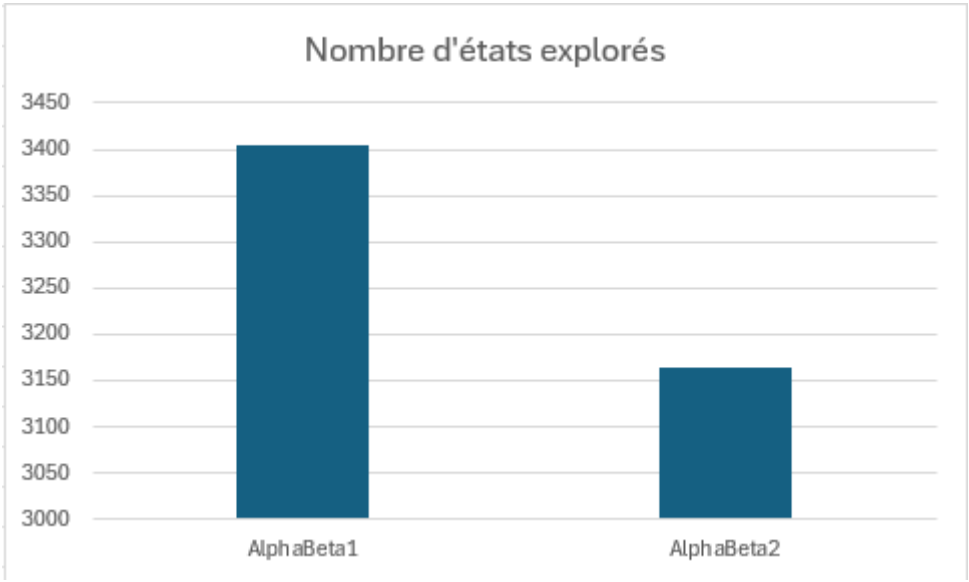


(-d 6)

VAINQUEUR → AlphaBeta2

Temps de calcul = 0.255s

19 coups



Les résultats montrent que l'augmentation de la **profondeur de recherche** impacte significativement le **nombre d'états explorés** et le **temps d'exécution** d'Alpha-Beta.

- Avec une **profondeur de 4**, l'algorithme explore environ **600 états** en moins de **0.16 secondes**.
- En passant à une **profondeur de 6**, le nombre d'états explorés est **multiplié par 5-6**, atteignant environ **3 400 états**, mais le **temps d'exécution augmente légèrement** à **0.25 secondes**.

Cette augmentation reste plutôt maîtrisée ensoi, ce qui nous confirme qu'Alpha-Beta parvient à explorer plus profondément sans explosion exponentielle du temps de calcul.

Exécutions Difficiles

Nous allons maintenant tester **Alpha-Beta** sur **Connect 4** avec des profondeurs plus élevées.

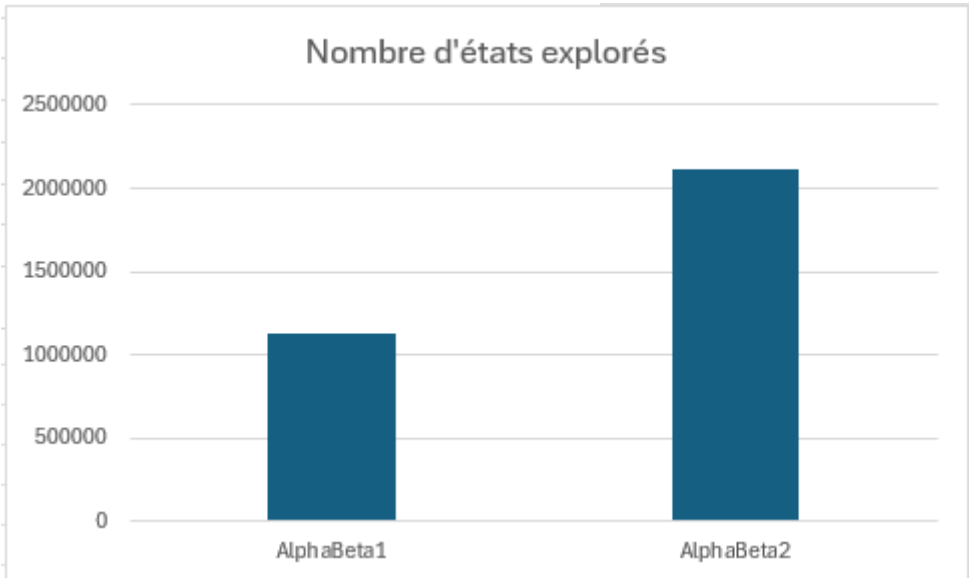
En augmentant la profondeur, Alpha-Beta devrait être en mesure de **mieux anticiper les coups adverses**, mais cela entraînera une **hausse du coût**. On verra si cet équilibre entre qualité et performance est intéressant ici.

(-d 12)

VAINQUEUR → AlphaBeta2

Temps de calcul = 27.134s

16 coups

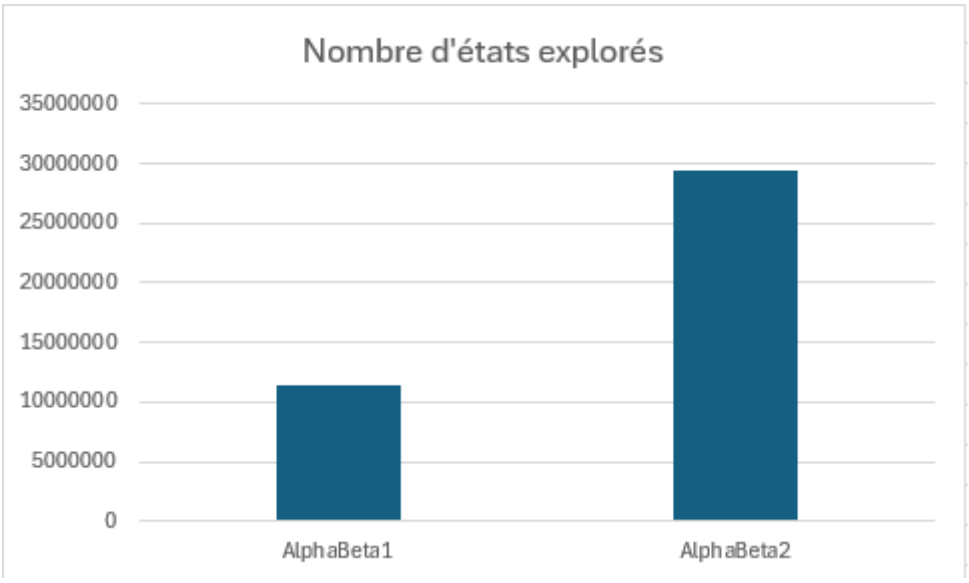


(-d 14)

VAINQUEUR → AlphaBeta2

Temps de calcul = 337.136s

21 coups



Ces tests montrent clairement l'impact de l'augmentation de la profondeur sur **Alpha-Beta**.

Avec une profondeur de **12**, l'algorithme explore déjà **plusieurs millions d'états** en une trentaine de secondes, et à **14**, le nombre d'états explose à **plus de 40 millions**, qui a entrainé par ailleurs un temps d'exécution beaucoup plus grand.

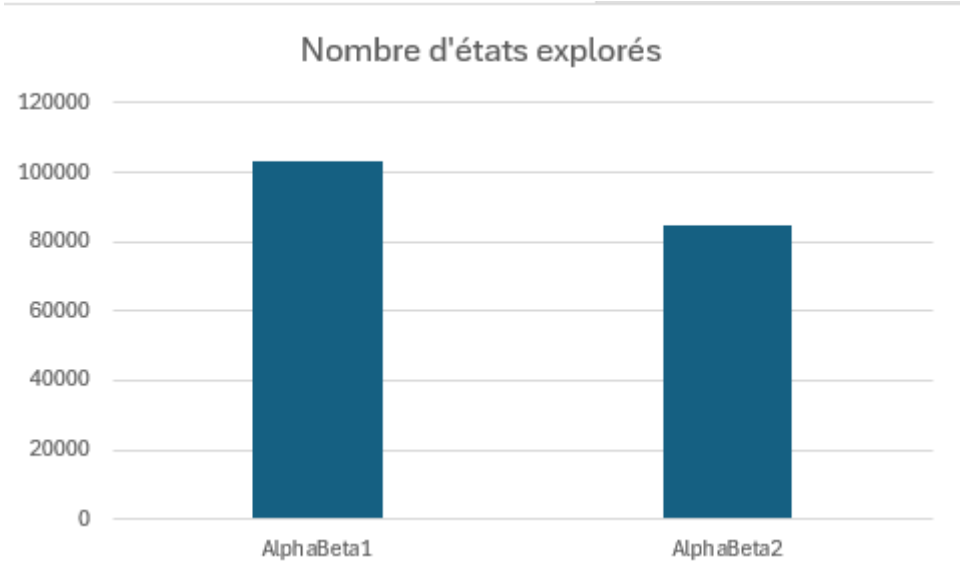
Bien que la recherche plus profonde permette d'affiner la qualité des coups et d'anticiper davantage, les résultats que nous observons nous confirment que le coût devient vite un facteur limitant, surtout pour des jeux complexes comme **Connect 4**.

Mnk (5x5, 4) – AlphaBeta vs AlphaBeta

VAINQUEUR → AlphaBeta1

Temps de calcul = 1.044s

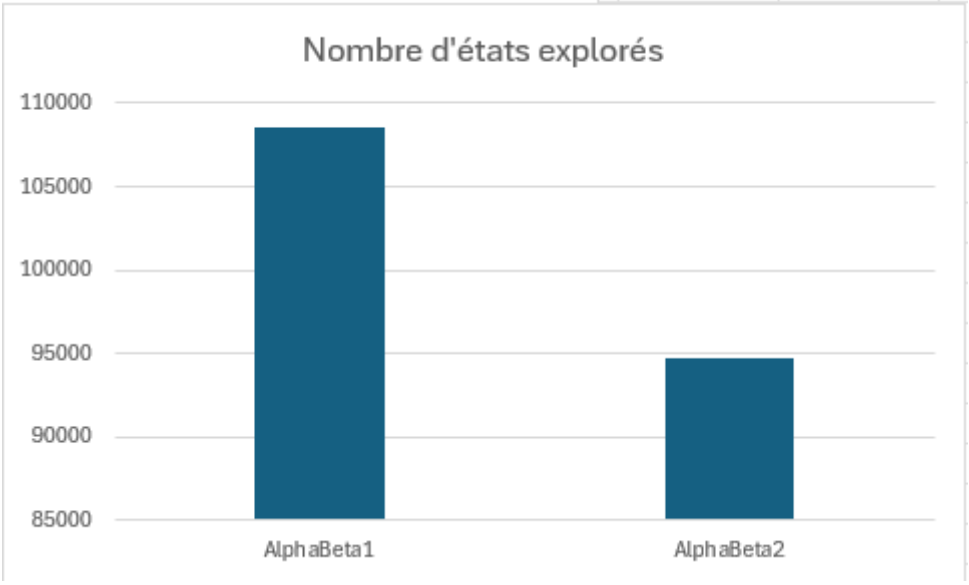
8 coups



VAINQUEUR → AlphaBeta1

Temps de calcul = 1.044s

8 coups



La comparaison des deux fonctions d'évaluation met en évidence un léger gain en précision, mais une augmentation du nombre d'états explorés.

- **Première exécution (fonction d'évaluation par défaut) :**
  - **Temps d'exécution :** 1.044 s
  - **États explorés :** 103 189 (Joueur1) / 84 563 (Joueur2)
- **Deuxième exécution (nouvelle fonction d'évaluation) :**
  - **Temps d'exécution :** 1.377 s
  - **États explorés :** 108 543 (Joueur1) / 94 657 (Joueur2)

Notre fonction d'évaluation semble légèrement plus efficace pour évaluer les positions, mais elle explore plus de nœuds que la version précédente. Cela peut être dû au fait que les évaluations intermédiaires modifient les choix de l'algorithme, ce qui pousserait Alpha-Beta à examiner plus de branches.

L'impact sur le temps d'exécution reste modéré, avec une légère augmentation du temps de calcul (+0.3s). Cependant, cela peut être négligeable dans un contexte plus complexe.

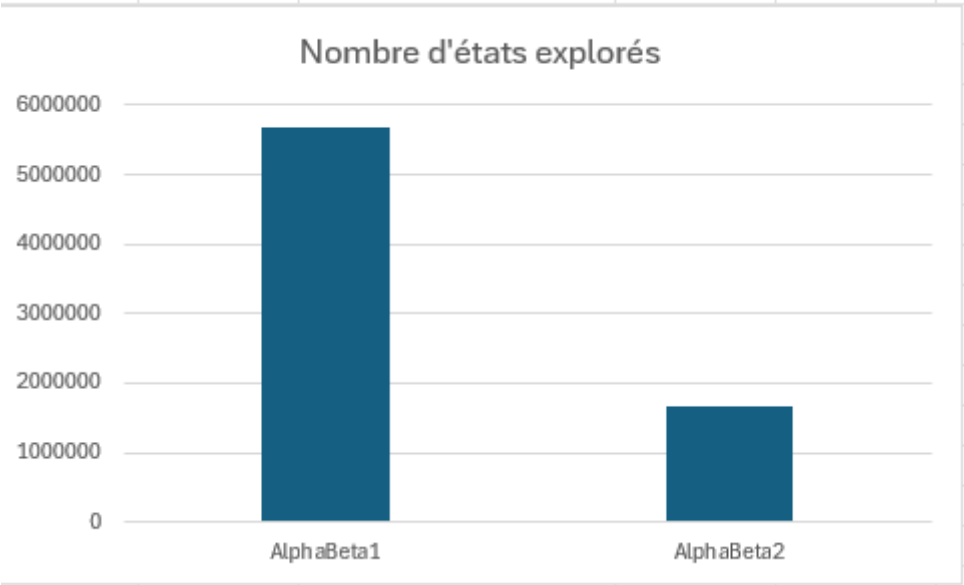
Elle ne réduit pas drastiquement le nombre d'états explorés, mais elle apporte une meilleure précision des décisions.

Gomoku (6x6, 5 en ligne, profondeur 3) – AlphaBeta vs AlphaBeta

VAINQUEUR → AlphaBeta1

Temps de calcul = 607.813 s

22 coups



Même avec une profondeur réduite à 3, le temps d'exécution reste extrêmement long. Cela montre que le facteur de branchement du Gomoku est bien plus élevé que dans les jeux précédents (Tic-Tac-Toe, MnkGame ou Connect 4).

- Nombre d'états explorés très élevé (**plusieurs millions**).
- Alpha-Beta fonctionne bien mais n'arrive pas à réduire suffisamment l'espace de recherche pour rendre l'exécution fluide.
- Augmenter la profondeur serait impraticable car le temps d'exécution exploserait.

Gomoku, même sur un plateau réduit est trop coûteux en calcul. En fait il faudrait intégrer des heuristiques plus avancées et réduire davantage la profondeur pour que l'algorithme soit réellement utilisable dans un contexte pratique.

Conclusion

Alpha-Beta améliore nettement MinMax, mais n'est pas une solution miracle pour tous les jeux malheureusement. L'optimisation de la profondeur et des fonctions d'évaluation reste essentielle pour obtenir un bon équilibre entre temps de calcul et qualité des décisions.
Dans les jeux très complexes comme Gomoku, on pourrait se dire que d'autres approches sont nécessaires, car même Alpha-Beta ne suffit pas à maîtriser l'exponentielle.