AAIA TP 3IF - Monde des Blocs

Florian Rascoussier Christine Solnon

2 avril 2025

Introduction

Le problème du monde des blocs est un cas classique en intelligence artificielle et en planification automatique. Il consiste à manipuler un ensemble de blocs disposés sur différentes piles pour atteindre une configuration spécifique.

Dans sa modélisation standard, le monde des blocs est représenté comme un problème de planification, où chaque configuration de blocs représente un état. Le but est de trouver une séquence d'actions (mouvements de blocs d'une pile à une autre) permettant de passer de l'état initial à l'état final désiré. Ceci est typiquement représenté sous la forme d'un graphe d'états, où chaque nœud correspond à une configuration de blocs et chaque arête à un mouvement possible entre deux configurations.

Les algorithmes A* et AWA* sont utilisés pour résoudre ce problème en cherchant le chemin le plus court dans le graphe d'états. A* est un algorithme de recherche informé qui utilise des heuristiques pour guider la recherche vers l'état final de manière efficace. AWA*, quant à lui, est une variante d'A* permettant une recherche anytime, adaptative en fonction des ressources disponibles.

Ce travail pratique (TP) se concentre sur l'application et la comparaison de ces algorithmes dans le contexte du monde des blocs. Nous discuterons en détail la mise en œuvre de ces algorithmes, l'importance de choisir des heuristiques adéquates, et comment elles peuvent drastiquement influencer les performances et l'efficacité de la recherche. L'objectif est de comprendre non seulement comment fonctionnent ces algorithmes de planification, mais aussi d'apprécier l'impact significatif des heuristiques pour guider la recherche vers des solutions optimales ou satisfaisantes dans des délais raisonnables.

En somme, ce TP offre une exploration approfondie du monde des blocs comme un microcosme pour comprendre les défis et les stratégies de la planification et de l'optimisation dans les systèmes intelligents.

Table des Matières

Table des figures

Codes et programmes

1 Partie 1 : Modélisation du problème

Dans cette partie, nous nous intéressons à la modélisation du problème du *Monde des Blocs* en tant que problème de planification.

1.1 Définition et déclaration des variables

Voici une liste des variables utilisées, avec leur domaine et leur signification :

- n : le nombre de blocs
- k : le nombre de piles
- E : un état donné du Monde des Blocs
- $actions(E)$: l'ensemble de toutes les actions possibles pour l'état E
- $t(E, i \to j)$: le nouvel état obtenu en appliquant l'action $i \to j$ à l'état E
- $G = (S, A)$: le graphe d'états, où S est l'ensemble des états possibles et A est l'ensemble des transitions entre les états

1.2 Question 1

Combien d'actions différentes sont-elles possibles pour l'état E_1 ?

Supposons que l'état E_1 dispose de $k = 3$ piles non-vides. Chaque pile peut envoyer un bloc vers $k - 1$ autres piles. Donc, pour chaque pile, il y a $k - 1$ actions possibles. Par conséquent, le nombre total d'actions possibles pour l'état E_1 est $(k - 1) \times k = 6$ actions.

1.3 Question 2

Étant donné un état E de n blocs sur k piles, quelle est la taille maximale de $actions(E)$?

Pour maximiser le nombre d'actions possibles, chaque pile doit contenir au moins un bloc, permettant ainsi des déplacements de blocs entre toutes les piles. Ainsi, chaque pile peut envoyer un bloc à $k - 1$ autres piles. Le nombre total d'actions possibles est donc $k(k - 1)$.

1.4 Question 3

Étant donné un état E de n blocs sur k piles ayant v piles vides, quelle est la taille de $actions(E)$?

Avec v piles vides, il reste $k - v$ piles contenant des blocs. Chaque pile active peut envoyer un bloc à $k - 1$ autres piles, donc le nombre total d'actions est $(k - v)(k - 1)$.

1.5 Question 4

Quel est l'ordre de grandeur du nombre total d'états différents possibles ?

Le nombre total d'états est exponentiel par rapport au nombre de blocs et linéaire par rapport au nombre de piles.

Pour s'en convaincre, on peut considérer que pour atteindre l'état final, chaque bloc doit être déplacé au moins une fois. Considérons cette situation qui est donc une borne minimale du chemin optimal parmi le graphe d'états possibles. Dans ce cas, à chaque niveau de profondeur, on a vu avec la question 2 qu'il y a $k(k-1)$ actions possibles au maximum. Ainsi, chaque état à chaque niveau possède $k(k-1)$ états fils. On peut donc en déduire que le nombre total d'états est de l'ordre de $O((k(k-1))^n) = O((k^2-k)^n) = O(k^{2n})$.

On se retrouve alors avec une borne minimale du nombre maximum d'états possible en $O(k^{2n})$, ce qui est exponentiel par rapport au nombre de blocs et linéaire par rapport au nombre de piles.

2 Partie 2 : Définition du graphe d'états

2.1 Question 5

Le graphe d'états G est-il orienté ?

La question est ambigüe. Selon le contexte réel derrière la question, le graphe d'états G peut être orienté ou non. En effet, si l'on considère le graphe d'états comme un graphe de planification, alors il est orienté. En revanche, si l'on considère le graphe d'états comme un graphe de recherche de plus court chemin, alors il n'est pas orienté.

Ici, on attend "non orienté" comme réponse, car le graphe d'états est généralement utilisé pour trouver le plus court chemin entre l'état initial et l'état final, et n'est donc pas orienté.

2.2 Question 6

Quels sont les algorithmes qui peuvent être utilisés pour rechercher ce plus court chemin ?

Plusieurs algorithmes peuvent être utilisés pour rechercher le plus court chemin, tels que Bellman-Ford, BFS, DFS et Dijkstra, en fonction des spécificités du graphe (poids, orientation, etc.).

Attention, on ne pourra pas utiliser l'algorithme TopoDAG car le graphe n'est pas un DAG (Directed Acyclic Graph) car il contient des cycles. En effet, par exemple, l'état final peut typiquement être atteint par plusieurs états prédédecesseurs différents.

2.3 Question 7

Quel est l'algorithme le plus efficace pour rechercher ce plus court chemin ?

Dans le contexte d'un graphe non pondéré comme celui du Monde des Blocs, BFS (Breadth-First Search) est généralement le plus efficace, car il trouve le plus court chemin en explorant uniformément autour du nœud source.

DFS est un autre algorithme de parcours de graphe qui se concentre sur l'exploration en

profondeur d'une branche du graphe avant de revenir en arrière pour explorer d'autres branches. Il peut également être utilisé pour trouver le plus court chemin.

Chacun de ces algorithmes a ses avantages et inconvénients. Mais surtout, ils peuvent être améliorés en utilisant des heuristiques pour guider la recherche vers des solutions plus optimales.

2.4 Question 8

Quelle est la complexité en temps de cet algorithme par rapport à $|S|$ et $|A|$?

La complexité temporelle de BFS est linéaire, soit O(|S| + |A|), où |S| est le nombre de sommets (états) et |A| est le nombre d'arêtes (actions) dans le graphe.

2.5 Question 9

Quelle est la complexité en temps de cet algorithme par rapport au nombre de blocs (n) et de piles (k) ?

La complexité temporelle de l'algorithme par rapport au nombre de blocs (n) et de piles (k) est exponentielle, car le nombre total d'états possibles croît exponentiellement avec l'augmentation de n et k (voir question 4, section 1.5).

3 Partie 3 : Heuristiques pour le monde des blocs

3.1 Question 10

On commence par exécuter le programme de recherche de plus court chemin pour 4 piles et un nombre de blocs variant de 6 à 8. On obtient les résultats suivants :

```
$ make run
Enter the number of stacks: 4
Enter the number of blocs: 6
Optimal solution of length 7 found in 12982 iterations and 0.060902 seconds
[...]
$ make run
Enter the number of stacks: 4
Enter the number of blocs: 7
Optimal solution of length 9 found in 188569 iterations and 1.04769 seconds
[...]
$ make run
Enter the number of stacks: 4
Enter the number of blocs: 8
Optimal solution of length 11 found in 2224481 iterations and 15.6446 seconds
[...]
```

Code 1 – Résultats de l'exécution du programme de recherche de plus court chemin

Les résultats montrent une augmentation exponentielle du nombre d'itérations et du temps d'exécution avec l'augmentation du nombre de blocs. Pour améliorer les performances de l'algorithme, on peut adopter des heuristiques adéquates. Trois heuristiques sont introduites :

- h_1 : Nombre de blocs ne se trouvant pas sur la dernière pile.
- h_2 : Nombre de blocs ne se trouvant pas sur la dernière pile, plus deux fois le nombre de blocs b tels que b se trouve sur la dernière pile mais devra nécessairement être enlevé de cette pile pour ajouter et/ou supprimer d'autres blocs sous lui.
- h_3 : Nombre de blocs de E_i ne se trouvant pas sur la dernière pile, plus le nombre de blocs se trouvant au-dessus de chaque bloc ne se trouvant pas sur la dernière pile.

Pour qu'une heuristique soit admissible, elle doit toujours sous-estimer le coût réel pour atteindre l'objectif. Formellement, une heuristique h est admissible si pour tout état E, $h(E) \leq \delta(E, E')$ où $\delta(E, E')$ est la distance réelle entre E et l'état cible E'.

Une heuristique h est dite plus informée qu'une heuristique h' si, pour tout état E, $h(E) \geq h'(E)$, tandis que les deux heuristiques sont incomparables s'il existe deux états E et E' tels que $h(E) < h'(E)$ et $h(E') > h'(E')$.

3.2 Question 11

L'heuristique h_1 est-elle admissible ?

L'heuristique h_1 est admissible, car le nombre de blocs ne se trouvant pas sur la dernière pile représente une limite inférieure du nombre de mouvements nécessaires pour atteindre l'état final. En effet, chaque bloc doit au moins une fois être déplacé vers la dernière pile, sous-estimant ainsi le coût réel pour atteindre l'objectif.

3.3 Question 12

L'heuristique h_2 est-elle admissible ?

L'heuristique h_2 est également admissible. Elle prend en compte le coût de déplacement des blocs qui sont sur la dernière pile, mais doivent être déplacés pour permettre à d'autres blocs de se placer en dessous. En multipliant ce nombre par deux, on reste dans une estimation inférieure du coût réel. En effet, pour chaque bloc sur la dernière pile, mais n'étant pas à la bonne hauteur, il faudra au minimum 2 déplacements de ce bloc considéré pour l'enlever puis le remettre sur la dernière pile. Il s'agit d'une estimation minimum, car chaque bloc ainsi déplacé peut nécessiter plus de 2 mouvements pour atteindre sa position finale.

3.4 Question 13

L'heuristique h_2 est-elle plus informée, moins informée, ou incomparable par rapport à l'heuristique h_1 ?

L'heuristique h_2 est plus informée que l'heuristique h_1. En plus de compter les blocs ne se trouvant pas sur la dernière pile, elle prend en compte le coût supplémentaire pour déplacer les blocs déjà sur la dernière pile, mais qui doivent être temporairement déplacés. Ainsi, h_2 donnera une valeur de coût égale ou supérieur à celle fournie par h_1, et donne donc une meilleure estimation du coût réel. Elle est donc plus informée.

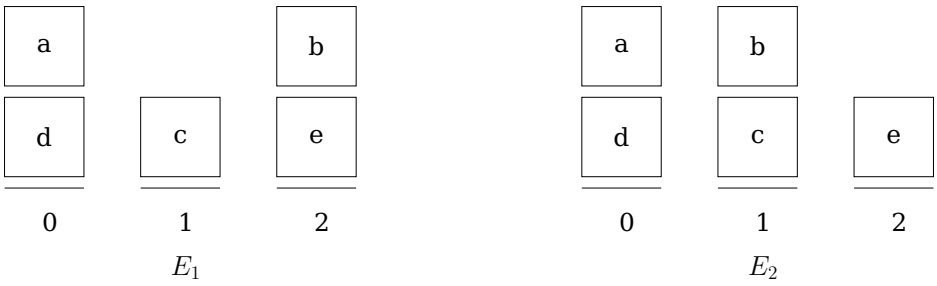

Figure 1 – Exemples de deux états E_1 et E_2, pour $k = 3$ et $n = 5$.

.

3.5 Question 14

L'heuristique h_3 est-elle admissible ?

L'heuristique h_3 n'est PAS admissible. Elle compte non seulement les blocs qui ne sont pas sur la dernière pile, mais aussi le nombre de blocs qui doivent être déplacés pour permettre le placement d'autres blocs. Il existe des cas dans lesquels h_3 fournit une surestimation du coût réel, car elle ne prend pas en compte les déplacements multiples ni l'ordre optimal de déplacement des blocs.

Un contre exemple simple consiste à considérer un exemple avec 2 piles et 2 blocs sur la pile 0, de tel sorte que ces blocs peuvent être placés directement sur la pile 1. Dans ce cas, h_3 donnera une valeur de $2 + 1 = 3$ (2 blocs pas sur la dernière pile plus le bloc du dessous qui a un bloc au dessus de lui), alors que le coût réel est de 2.

3.6 Question 15

L'heuristique h_3 est-elle plus informée, moins informée, ou incomparable par rapport à l'heuristique h_2 ?

Tandis que h_2 compte de manière spécifique les déplacements nécessaires pour les blocs sur la dernière pile, h_3 se concentre sur une vue plus globale de tous les blocs qui ne sont pas à leur place. En fonction de la situation, par exemple si la dernière pile est déjà bien remplie, h_2 peut donner une meilleure estimation du coût réel que h_3 puisqu'il reste peu de blocs sur les autres piles. C'est bien sûr l'inverse quand la dernière pile est vide. Ainsi, les deux heuristiques sont incomparables.

On peut ainsi trouver des contres-exemples :

$$\exists E_1, E_2 \in E \mid h_2(E_1) < h_2(E_2) \wedge h_3(E_1) > h_3(E_2).$$

Dans l'illustration suivante 1, on peut voir :

$$h(E_1) = \begin{cases} h_1(E_1) = 3 \\ h_2(E_1) = 5 \\ h_3(E_1) = 4 \end{cases} \qquad h(E_2) = \begin{cases} h_1(E_2) = 4 \\ h_2(E_2) = 4 \\ h_3(E_2) = 6 \end{cases}$$

On a donc $h_2(E_1) > h_3(E_2) \wedge h_2(E_1) < h_3(E_2)$, ce qui montre que les deux heuristiques sont incomparables.

3.7 Question 16 : heuristique h_4

Quelle est la longueur (en nombre d'actions) de la solution optimale pour $n = 16$ blocs et $k = 4$ piles ?

On commence par implémenter les trois heuristiques dans le programme de recherche de plus court chemin. On obtient nos premiers résultats :

```
$ make run
[...]
g(4, 8, heuristic0):
Optimal solution of length 11 found in 2224481 iterations and 15.1703 seconds
g(4, 8, heuristic1):
Optimal solution of length 11 found in 61468 iterations and 0.242329 seconds
g(4, 8, heuristic2):
Optimal solution of length 11 found in 169 iterations and 0.000934 seconds
g(4, 8, heuristic3):
Optimal solution of length 12 found in 261465 iterations and 1.12496 seconds
```

Code 2 – Résultats de l'exécution du programme de recherche de plus court chemin avec 3 premières heuristiques pour $k = 4$ piles et $n = 8$ blocs.

On remarque d'après les résultats obtenus 2 que l'heuristique h_2 est la plus performante, car elle trouve la solution optimale en un nombre d'itérations et de temps d'exécution très faible. On peut donc la considérer comme la meilleure heuristique parmi les trois.

On ajoute alors une quatrième heuristique h_4 basée sur h_2, elle-même une amélioration de h_1. Pour cela, considérons par exemples les états E_1 et E_2 suivants :

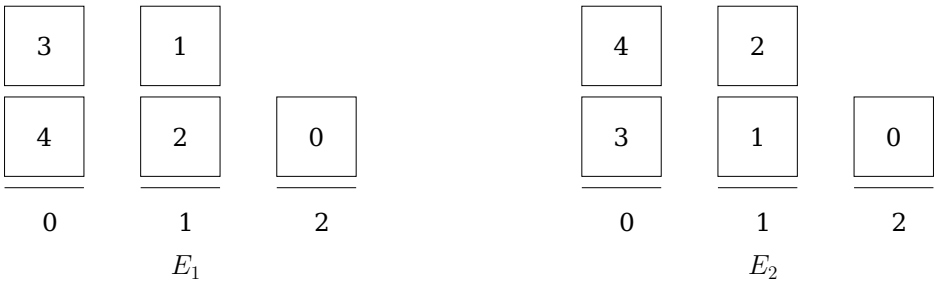

Figure 2 – Exemples de deux états E_1 et E_2, pour $k = 3$ et $n = 5$, illustrant l'intérêt derrière h_4.

On peut constater à la main que l'état E_2 est à 7 mouvements de l'état cible, alors que l'état E_1 en a besoin de beaucoup plus.

On définit alors une telle heuristique h_4 comme suit :

- **Nombre de blocs non dans la dernière pile :** Correspondant à h_1, cette partie compte tous les blocs qui ne sont pas placés dans la dernière pile. Plus il y a de blocs éloignés de leur destination finale, plus le nombre de mouvements requis sera potentiellement important.
- **Deux fois le nombre de blocs sur la dernière pile à déplacer :** Cette mesure reprends h_2 en donnant plus de poids aux blocs qui doivent être déplacés depuis la dernière pile. Cela

reflète le coût potentiellement élevé en termes de mouvements pour déplacer ces blocs, notamment s'il faut les déplacer temporairement ailleurs avant de les remettre en ordre.

- **Évaluation de l'arrangement des blocs sur toutes les autres piles :** Pour chaque pile (sauf la dernière), on augmente la somme heuristique pour chaque bloc qui n'est pas plus grand que le bloc situé en dessous. Ainsi, un bloc plus petit ou de taille égale reposant sur un bloc plus grand augmente la valeur heuristique, indiquant un état moins idéal. Cela capture l'arrangement des blocs dans les piles et favorise les états où ils sont déjà dans un ordre croissant vers le but.

En combinant ces éléments, l'heuristique h_4 fournit une meilleure sous-estimation que h_1 ou h_2, en saisissant davantage de nuances dans l'agencement des blocs. Cette sous-estimation est importante car elle garantit que l'heuristique reste admissible et ne surestime jamais le coût réel pour atteindre l'objectif, ce qui est une propriété essentielle pour guarantir l'optimalité de A*.

On obtient les résultats suivants pour cette nouvelle heuristique :

```
$ make run
[...]
g(4, 8, heuristic4):
Optimal solution of length 11 found in 50 iterations and 0.000341 seconds
```

Code 3 – Résultats de l'exécution du programme de recherche de plus court chemin avec l'heuristique h_4 pour $k = 4$ piles et $n = 8$ blocs.

Comme on peut s'y attendre, l'heuristique h_4 est plus performante que les trois premières heuristiques. Elle trouve la solution optimale en un nombre d'itérations et de temps d'exécution très faible. On peut donc la considérer comme la meilleure heuristique parmi les quatre. C'est cette heuristique qui sera utilisée pour la suite des expériences.

Voici mes temps de calculs pour $n = 16$ blocs et $k = 4$ piles :

```
$ make run
[...]
g(4, 16, heuristic2):
Optimal solution of length 25 found in 1566532 iterations and 21.5822 seconds
g(4, 16, heuristic4):
Optimal solution of length 25 found in 19890 iterations and 1.32529 seconds
```

Code 4 – Résultats de l'exécution du programme de recherche de plus court chemin avec l'heuristique h_4 pour $k = 4$ piles et $n = 8$ blocs.

On peut voir que la longueur de la solution optimale est de 25 actions.

3.8 Question 16 bis

On peut encore améliorer l'heuristique h_4 en une nouvelle version h_5. En effet, considérons les états E_1 et E_2 suivants :

3.9 Question 17

Quelle est la longueur (en nombre d'actions) de la solution optimale pour $n = 20$ blocs et $k = 4$ piles ?

On exécute le programme de recherche de plus court chemin avec l'heuristique h_4 pour $n = 20$ blocs et $k = 4$ piles. On obtient les résultats suivants :

```
$ make run
[...]
g(4, 20, heuristic4):
Optimal solution of length 32 found in 306805 iterations and 3.11438 seconds
```

Code 5 – Résultats de l'exécution du programme de recherche de plus court chemin avec l'heuristique h_4 pour $k = 4$ piles et $n = 20$ blocs.

La longueur de la solution optimale est de 32 actions.

4 Partie 4 : Implémentation de AWA*

Cette section présente l'algorithme A* ainsi que deux de ses variantes : WA* (Weighted A*) et AWA* (Anytime WA*). Ces algorithmes sont utilisés pour la recherche de chemins optimaux dans un graphe à partir d'un état initial vers un ou plusieurs états finaux, en combinant le coût déjà accumulé et une estimation du coût restant.

5 A* et ses versions WA* et AWA*

Afin de trouver la longueur de la solution optimale pour autant de calculs nécessaires, on transforme le programme de recherche de plus court chemin en passant de *A** à *AWA** (Anytime Weighted A*). AWA* est une variante de A* qui permet de trouver une solution optimale en un temps limité. Il s'agit d'une version pondérée de A*, où le poids w est défini par l'utilisateur. Ainsi, AWA* peut être utilisé pour trouver une solution optimale en un temps limité, ou pour trouver une solution sous-optimale en un temps limité.

- **Algorithme WA* (Weighted A*) :** Version très légèrement modifiée de l'algorithme A*, où la valeur de mise en file d'attente dans le tas est davantage pondérée en faveur de l'heuristique. Ceci transforme techniquement A* en une sorte de Recherche Gloutonne au Meilleur Premier (Greedy Best First Search). Ainsi, lorsque $w = 1$, la distance depuis l'origine et la distance heuristique vers le but sont pondérées de manière égale et l'algorithme se comporte exactement comme A*. Lorsque $w > 1$, l'algorithme trouve une solution plus rapidement mais perd la garantie d'optimalité. L'objectif est d'itérer vers un $w = 1$.
- **Algorithme AWA* (Anytime Weighted A*) :** Variante améliorée et anytime de l'algorithme WA*. L'objectif ici est de mettre à jour une borne supérieure de la solution. Cette approche permet à l'algorithme de fournir une solution valide à tout moment du processus de recherche et d'améliorer cette solution au fur et à mesure du temps ou des itérations disponibles.

5.1 Algorithme A*

L'algorithme A* repose sur une fonction d'évaluation

$$f(n) = g(n) + h(n),$$

où :

- $g(n)$ représente le coût du chemin depuis l'état initial jusqu'à l'état n,
- $h(n)$ est une fonction heuristique qui estime le coût minimal restant pour atteindre un état final.

Le pseudocode de l'algorithme A* est présenté ci-dessous :

Algorithm 1 A*

$OPEN \leftarrow \{s_{init}\}$
$CLOSED \leftarrow \emptyset$
while $OPEN \neq \emptyset$ **do**
    Sélectionner l'état $n \in OPEN$ tel que $f(n)$ est minimal
    if n est un état final **then**
        return le chemin de s_{init} à n
    end if
    Retirer n de $OPEN$ et l'ajouter à $CLOSED$
    for chaque successeur s de n **do**
        if s n'est pas dans $CLOSED$ ou si une meilleure solution est trouvée pour s **then**
            Mettre à jour $g(s)$ et calculer $f(s) = g(s) + h(s)$
            Ajouter s à $OPEN$
        end if
    end for
end while
return *échec* (aucune solution n'est trouvée)

5.2 Version WA* (Weighted A*)

WA* est une variante d'A* dans laquelle la fonction heuristique est surpondérée par un paramètre $w \geq 1$. La fonction d'évaluation devient :

$$f(n) = g(n) + w \times h(n).$$

Lorsque $w = 1$, on retrouve l'algorithme A*. L'utilisation d'un $w > 1$ tend à accélérer la recherche en favorisant l'exploration des nœuds proches de la solution, mais cela peut compromettre l'optimalité de la solution trouvée.

Le pseudocode de WA* est similaire à celui d'A*, en remplaçant simplement la fonction f :

Algorithm 2 WA*

$OPEN \leftarrow \{s_{init}\}$
$CLOSED \leftarrow \emptyset$
while $OPEN \neq \emptyset$ **do**
    Sélectionner l'état $n \in OPEN$ tel que $f(n) = g(n) + w \times h(n)$ est minimal
    if n est un état final **then**
        return le chemin de s_{init} à n
    end if
    Retirer n de $OPEN$ et l'ajouter à $CLOSED$
    for chaque successeur s de n **do**
        if s n'est pas dans $CLOSED$ ou si une meilleure solution est trouvée pour s **then**
            Mettre à jour $g(s)$ et calculer $f(s) = g(s) + w \times h(s)$
            Ajouter s à $OPEN$
        end if
    end for
end while
return *échec*

5.3 Version Anytime WA* (AWA*)

L'approche Anytime WA* (AWA*) vise à fournir des solutions de manière progressive, sans attendre la fin complète de l'exploration. L'idée est la suivante :

- On initialise une borne supérieure B à $+\infty$.
- Chaque fois qu'un état final n est généré et que le coût $g(n)$ est inférieur à B, le chemin menant à n est immédiatement enregistré comme solution et la borne B est mise à jour par $g(n)$.
- Cette borne permet d'élaguer l'arbre de recherche en éliminant les états dont la valeur $f(n)$ excède B, réduisant ainsi l'espace de recherche et améliorant l'efficacité globale.

Le pseudocode de l'algorithme AWA* est présenté ci-dessous :

Algorithm 3 AWA*

$OPEN \leftarrow \{s_{init}\}$
$CLOSED \leftarrow \emptyset$
$B \leftarrow +\infty$
while $OPEN \neq \emptyset$ **do**
    Sélectionner l'état $n \in OPEN$ tel que $f(n) = g(n) + w \times h(n)$ est minimal
    if n est un état final **et** $g(n) < B$ **then**
        Enregistrer le chemin de s_{init} à n
        Mettre à jour la borne supérieure $B \leftarrow g(n)$
    end if
    if $f(n) \geq B$ **then**
        passer (élagage de l'état)
    end if
    Retirer n de $OPEN$ et l'ajouter à $CLOSED$
    for chaque successeur s de n **do**
        if s n'est pas dans $CLOSED$ ou si une meilleure solution est trouvée pour s **then**
            Mettre à jour $g(s)$ et calculer $f(s) = g(s) + w \times h(s)$
            Ajouter s à $OPEN$
        end if
    end for
end while
return la meilleure solution enregistrée

5.4 Question 18

Quelle est la longueur (en nombre d'actions) de la solution optimale pour $n = 25$ blocs et $k = 5$ piles ?

Après implémentation de AWA*, et en utilisant h_4 et h_5, on obtient les résultats suivants :

```
$ make run
[...]
AWA*(k = 5, n = 25, heuristic number = 4):
starting upper bound = 2147483647
Found solution of length 44 in 884 iterations and 0.02478 seconds, upper bound = 44, weight = 2
Found solution of length 43 in 1152 iterations and 0.029052 seconds, upper bound = 43, weight = 2
Found solution of length 42 in 1829 iterations and 0.040759 seconds, upper bound = 42, weight = 2
Found solution of length 41 in 4140 iterations and 0.083494 seconds, upper bound = 41, weight = 2
Found solution of length 40 in 6950 iterations and 0.13297 seconds, upper bound = 40, weight = 2
Found solution of length 39 in 40141 iterations and 0.809311 seconds, upper bound = 39, weight = 2
Found solution of length 38 in 100838 iterations and 2.044 seconds, upper bound = 38, weight = 2
Found solution of length 37 in 173164 iterations and 3.49765 seconds, upper bound = 37, weight = 2
Optimal solution of length 37 found in 2460289 iterations and 51.5927 seconds, weight = 2
```

Code 6 – Résultats de l'exécution du programme de recherche de plus court chemin avec l'heuristique h_4 et *AWA** pour $k = 5$ piles et $n = 25$ blocs.

```
$ make run
[...]
AWA*(k = 5, n = 25, heuristic number = 5):
starting upper bound = 2147483647
Found solution of length 42 in 1539 iterations and 0.027061 seconds, upper bound = 42, weight = 2
Found solution of length 41 in 1678 iterations and 0.029173 seconds, upper bound = 41, weight = 2
Found solution of length 40 in 1774 iterations and 0.030814 seconds, upper bound = 40, weight = 2
Found solution of length 39 in 8472 iterations and 0.148832 seconds, upper bound = 39, weight = 2
Found solution of length 38 in 11642 iterations and 0.203649 seconds, upper bound = 38, weight = 2
Found solution of length 37 in 19889 iterations and 0.353376 seconds, upper bound = 37, weight = 2
Optimal solution of length 37 found in 1066478 iterations and 19.9276 seconds, weight = 2
```

Code 7 – Résultats de l'exécution du programme de recherche de plus court chemin avec l'heuristique h_5 et *AWA** pour $k = 5$ piles et $n = 25$ blocs.

On peut voir que la longueur de la solution optimale est de 37 actions.

Acronymes

AAIA Algorithmique Avancée pour l'Intelligence Artificielle et les graphes. 2

Bibliographie additionnelle

[1] Christine Solnon. *Première partie : Algorithmique avancée pour les graphes*. AAIA. CITI Lab, INSA Lyon dépt. Informatique. 2016. url : `http://perso.citi-lab.fr/csolnon/supportAlgoGraphes.pdf` (visité le 28/12/2023).