UNBLOCK ME

CSC_42021_EP
Conception et analyse d'algorithmes

16 février 2025

Hubert LEROUX & Elliot THOREL

1 INTRODUCTION

Ce projet a été développé dans le cadre du cours CSC_42021_EP - Conception et analyse d'algorithmes de l'école polytechnique. Il est encadré par M. Coupechoux. Le but du projet est d'implémenter un algorithme efficace pour la résolution du problème du **rush hour** aussi connu sous le nom de **unblock-me**, dont on peut trouver une description détaillée ici. Il s'agit d'un casse-tête de déplacements où le but est d'extirper la voiture rouge d'un embouteillage automobile. De plus, nous nous intéresserons majoritairement dans ce projet à la recherche de **solution optimale**, c'est-à-dire minimisant le nombre de déplacements.

Le jeu peut facilement se transformer en un problème de graphe. La recherche du chemin le plus rapide pour la voiture rouge se fait alors en implémentant des algorithmes de type **Dijkstra**. Puis on peut explorer différentes heuristiques pour essayer d'être plus rapide dans notre recherche du chemin optimal.

2 PRÉPARATION DU JEU

2.1 Question 1

La première question consiste à récupérer les données d'un fichier texte et à vérifier qu'elles correspondent bien à une instance du problème. Il faut donc vérifier que deux véhicules ne partagent pas de case et qu'aucun véhicule ne sort de la grille.

Cette question est également l'occasion de décider la structure de données que nous utiliserons pour tout le problème. Nous avons fait le choix de travailler essentiellement avec une **dict-list** pour représenter chaque grille, chaque voiture étant alors représentée par un **dictionnaire** indiquant ses caractéristiques. Cependant, nous pouvons aussi représenter la grille comme une **matrice**, où chaque case représente une case de la grille.

Les deux représentations donnent chacune des informations complémentaires, l'une permettant d'avoir facilement des informations sur une voiture en particulier, l'autre permettant de facilement regarder le contenu d'une case. Pour cette raison, nous avons fait le choix de travailler avec les **deux structures en parallèle**, ce qui nous semble plus efficace en termes de temps d'exécution malgré la redondance des données.

2.2 Question 2

Ici, le but est d'avoir une représentation graphique de la grille. Nous nous sommes tenus à quelque chose de simple en nous servons de **print** python et en nous appuyant sur la **représentation matricielle** de la grille.

3 RÉSOLUTION DU PROBLÈME, SOLUTION INSPIRÉE DE DIJKSTRA

3.1 Question 3

Nous cherchons maintenant une méthode pour résoudre le problème du *rush hour*. Pour cela, nous voyons le problème comme un problème de graphe, où les nœuds sont les différentes configurations que peuvent prendre la grille au cours de la partie. Une arête entre deux nœuds indique que l'on peut passer d'une configuration à l'autre en un déplacement. Nous avons ainsi essentiellement ramené notre problème en un problème de recherche

du **plus court chemin dans un graphe**. Nous allons donc dans un premier temps résoudre le problème avec l'algorithme de Dijkstra.

```
Data: initial_game_state
Result: True if the red car can exit, False otherwise
file ← FIFO([initial_game_state]) ;
seen ← ∅ ;
while file ≠ ∅ do
    game_state ← file.pop() ;
    if Winning_configuration(game_state) then
        return True ;
    end
    if game_state ∉ seen then
        seen.add(game_state) ;
        for next_game_state ∈ game_state.next_possible_moves() do
            file.push(next_game_state) ;
        end
    end
end
return False ;
```

Algorithm 1: Brute force algorithm

3.1.1 • Sur les structures de données utilisées

Nous avons donc ici deux structures de données intéressantes.

— **seen** est une structure qui permet de se souvenir quels nœuds ont déjà été explorés. Elle doit être capable d'**ajouter des configurations** et surtout de **déterminer la présence d'une configuration**. Nous avons donc décidé d'utiliser **une table de hachage** pour son implémentation, afin de pouvoir effectuer ces opérations en temps constant.

— **File** est une structure qui permet de gérer l'**ordre** dans lequel on explore les différentes configurations. Ici, comme on effectue un parcours en profondeur (cas de l'algorithme de Dijkstra où toutes les arrêtes sont de même poids), une **file de priorité** suffit.

3.1.2 • Complexité

Notre implémentation nous permet d'exécuter toutes les opérations en temps constant. La complexité est donc en $\mathcal{O}$(**nombre de pushs**), c'est-à-dire $\mathcal{O}$(**nombre d'arêtes du graphe**)

Essayons d'exprimer cela en fonction de n la taille de la grille et de nb_cars le nombre de voitures. Chaque voiture peut prendre moins de n positions dans la grille. Le nombre de configurations est donc majoré par n^{nb_cars} et le nombre de coups est majoré par $nb\_cars * n$. On peut donc majorer la complexité par du $\mathcal{O}(n^{nb\_cars+1} * nb\_cars)$ ce qui est une estimation assez large, car elle ne prend pas en compte le fait que les voitures se bloquent les unes les autres.

3.2 Question 4

Pour déterminer, tous les mouvements possibles, on regarde simplement voiture par voiture les mouvements possibles. Le stockage double des données sous forme de grille et de liste de voitures rend cette tâche assez efficace.

3.3 Question 5

Pour ce qui est de la table de hachage, nous avons de plus créé une table adaptée à nos données. Nous avons décidé de créer une table de hachage de taille fixe environ égale à 1 000 000. Pour la fonction de hachage, nous avons considéré la surjection de l'ensemble des configurations possibles dans $\mathbb{N}$ suivantes : $\texttt{hash} = \prod_i^{\texttt{nb\_cars}} p_i^{d_i+1}$ où p_i désigne le $i^{\text{ème}}$ nombre premier et d_i désigne la distance de la voiture i au bord haut ou gauche selon l'orientation de la voiture. Notre fonction de hachage s'obtient alors en prenant le résidu modulo un nombre premier très grand (la taille de notre table de hachage).

3.4 Question 6

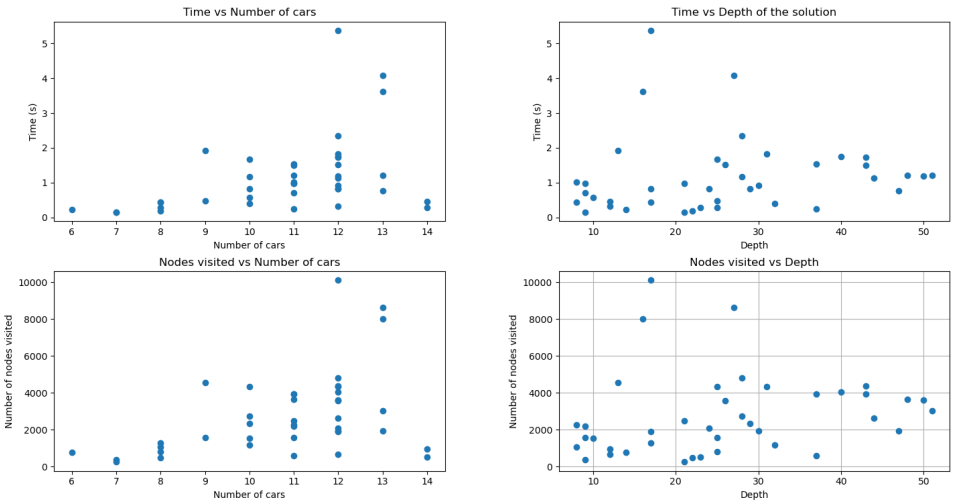

Figure 1 – Étude des performances de l'algorithme

L'algorithme tourne en moins d'une seconde pour la majorité des problèmes et explore jusqu'à 10 000 configurations. Si, comme on pouvait s'y attendre, un nombre élevé de voiture tend à rendre l'algorithme plus lent, la longueur de la solution finale n'est pas vraiment un facteur majeur de la complexité du problème. (1)

3.5 Question 7

Pour générer la solution optimale, on conserve en plus les **antécédents** de chaque configuration. Une fois une configuration gagnante trouvée, on est donc capable de retrouver tout le chemin parcouru, en remontant antécédent par antécédent à la configuration initiale.

Pour conserver les antécédents, nous avons utilisé **une table de hachage**, similaire à celle utilisée pour stocker les nœuds déjà parcourus. Nous nous sommes rendus compte qu'il était même possible de n'utiliser qu'une seule table de hachage pour réaliser à la fois le stockage des antécédents et la conservation en mémoire des nœuds déjà explorés. Cela nécessitait cependant une légère modification de l'algorithme de la question 3.

Au lieu de gérer le stockage dans **seen** lorsque le nœud sort de la file, on le gère lorsque le nœud entre dans la file. Cette modification conserve la validité de l'algorithme, car on effectue un parcours en largeur et non un Dijkstra plus général.

4
UTILISATION D'HEURISTIQUES (INSPIRÉ DE L'ALGORITHME A*)

4.1 Question 8

4.1.1 • Explication de l'algorithme

L'idée de l'algorithme est de se servir d'une distance heuristique qui estime le nombre de coups nécessaires pour atteindre la solution.

L'algorithme est essentiellement le même qu'aux questions précédentes. La seule différence est que plutôt de stocker les nœuds à parcourir dans une FIFO, on les stocke dans une file de priorité qui nous permet de parcourir les nœuds par priorité croissante ($\texttt{priorite}(s) = \texttt{h}(s) + \texttt{depth}(s)$).

Une description plus précise de l'algorithme se trouve en A.

4.1.2 • Choix d'implémentation

Nous ne pouvons pas nous servir de l'astuce de la question 7 de gérer l'appartenance à **seen** lors de l'ajout à la file. Nous allons donc utiliser deux tables de hachages en parallèle, une table **seen** et une table **antécédent**.

Nous avons utilisé des **tas** pour gérer plus simplement les priorités.

4.1.3 • Preuve de l'algorithme

La preuve de l'algorithme est essentiellement similaire à une preuve de l'algorithme A*. La preuve détaillée est donnée en B.

4.1.4 • Avec l'heuristique nulle

Avec une heuristique constante égale à 0, la priorité vaut toujours la profondeur. L'algorithme est donc très similaire à l'utilisation d'une FIFO (algorithme de la question 3). Il se peut cependant que pour une même profondeur, les états ne soient pas parcourus dans le même ordre selon l'implémentation de la file de priorité.

4.2 Question 9

À chaque mouvement, on ne peut déplacer qu'une voiture. En particulier, on ne peut donc changer le nombre de voitures bloquantes que de 1.

Avec deux états voisins s et s', on a donc $h(s) \leq h(s') + 1$. Par récurrence, on obtient que l'heuristique h est consistante.

4.3 Question 10

Pour générer les courbes de la figure 2, nous avons utilisé l'heuristique nulle pour le parcours en largeur afin de pouvoir comparer au maximum sur une base similaire. On observe que l'heuristique est pertinente dans la mesure où elle permet toujours de diminuer le nombre d'états explorés parfois jusqu'à un facteur 2. Cependant, les temps associés au calcul de la distance heuristique rendent les gains en temps d'exécution assez peu importants en moyenne. Il arrive même qu'il soit plus intéressant d'utiliser un parcours en largeur.

Lors du choix de l'heuristique, il faudra donc trouver un compromis entre le gain en nombre d'états explorés que permet l'heuristique et la complexité de calcul de l'heuristique.

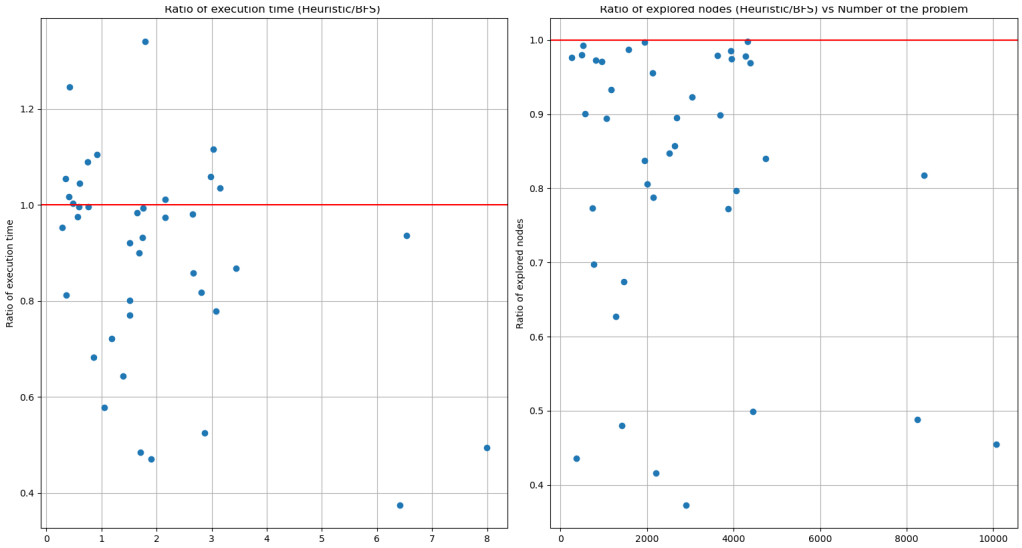

Figure 2 – Comparaison des performances de l'algorithme heuristique par rapport à un simple parcours en largeur

4.4 Question 11

Pour la recherche de nouvelles heuristiques admissibles, nous avons principalement étudié **deux types d'heuristiques**.

— Des heuristiques "normalisées" dont la valeur est comprise entre 0 et 1.
— Des heuristiques inspirées de celle du nombre de voitures bloquant la voiture rouge.

Pour ce qui est des heuristiques "normalisées", la démonstration de leur consistance est immédiate. Cependant, à l'instar de l'heuristique nulle, **leur intérêt est assez limité**. En effet, l'algorithme qui en résulte est fondamentalement un BFS ; on contrôle juste quel nœud sera exploré en priorité **à profondeur égale**. Un exemple de ce type d'heuristique est la distance normalisée de la voiture rouge à l'objectif.

Pour ce qui est des heuristiques inspirées du nombre de voitures bloquantes, l'idée est de chercher à déterminer le nombre de voitures **bloquant indirectement** la voiture rouge, pour une certaine définition de "indirectement". La consistance de l'heuristique vient alors du fait qu'on ne peut déplacer qu'une voiture par mouvement et que le nombre de voitures bloquant indirectement la voiture rouge peut donc varier d'au plus 1 à chaque mouvement.

Dans l'heuristique originale, on détermine les voitures bloquant la voiture rouge. Une première idée est donc de regarder quelles voitures bloquent ces voitures-ci. On cherche donc à déterminer le nombre minimum de voitures à faire disparaître pour permettre aux voitures bloquantes directement la voiture rouge de pouvoir se déplacer hors du chemin de cette dernière. En sommant les voitures à faire disparaître, les voitures bloquantes et le mouvement de la voiture rouge, on obtient alors une nouvelle heuristique consistante.

On peut prolonger cette idée en recommençant ce qu'on a fait pour les voitures bloquant directement, mais cette fois-ci pour celles bloquant indirectement.

— Plus précisément, on réfléchit en essayant de construire la solution dans le sens inverse. (On commence donc par le déplacement de la voiture rouge de la position initiale à la case d'arrivée).
— Chaque déplacement d'une voiture se fait en ignorant les autres voitures présentes et uniquement dans le but de libérer une certaine case (il peut y avoir un ou deux coups permettant à une voiture de libérer une case avec une perturbation minimale, selon si la voiture est coincée par un bord de la grille ou non)
— On regarde ensuite les autres voitures bloquant ce déplacement et on cherche à les libérer à leur tour.
— Une voiture libérée disparaît de la grille afin de ne pas bloquer les mouvements des autres voitures. Ceci garantit qu'on ne déplace chaque voiture au maximum une fois

Le calcul de cette heuristique est **assez lourd**. En effet, l'algorithme est similaire à l'algorithme du **rush**

hour dans la mesure où c'est un parcours de graphe de différentes configurations possibles. Cependant, le problème est nettement plus simple que le problème du **rush hour** car le fait que les voitures disparaissent assure que la profondeur ne dépasse pas le nombre de voitures. De plus, chaque voiture n'a au plus que deux déplacements possibles contrairement au **rush hour** où il peut en avoir $O(n)$.

En pratique, cette heuristique est **inutilisable** sur certaines grilles, car la distance heuristique est trop longue à calculer dans certaines configurations. Cependant, dans les grilles où elle est calculable, elle réduit significativement le nombre d'états à explorer.

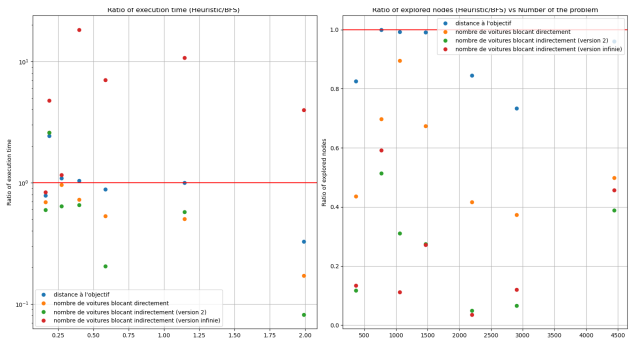

FIGURE 3 – Comparaison des performances des algorithmes heuristiques

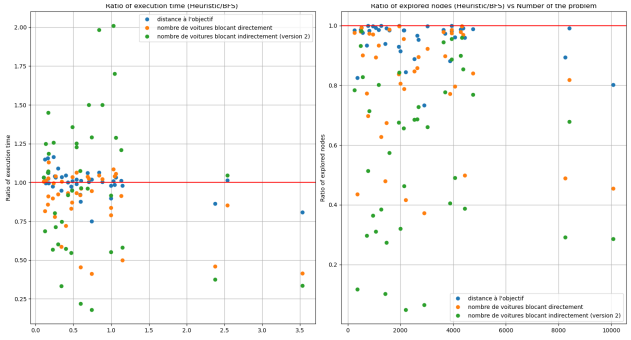

FIGURE 4 – Comparaison des performances des algorithmes heuristiques sur un jeu d'exemples plus important

La distance à l'objectif étant une distance heuristique normalisée à des performances très proches d'un simple parcours en largeur. On remarque cependant une légère amélioration.

L'heuristique complexe de calcul du nombre de véhicules bloquant indirectement donne de bonnes performances en termes de nombre de nœuds explorés, mais sa complexité la rend assez mauvaise du point de vue du temps d'exécution.

Les deux autres heuristiques semblent être de bon compromis. Elles se calculent suffisamment rapidement pour que cela ne soit pas handicapant à l'exécution de l'algorithme et diminuent (parfois significativement) le nombre de nœuds à explorer.

5 CONCLUSION

L'objectif de ce projet était d'implémenter une solution au problème du **rush hour**. Pour cela, nous avons donc transformé ce problème en un problème de plus court chemin dans un graphe puis implémenté des algorithmes de parcours de graphe (d'abord BFS puis de type A*).

La difficulté du projet résidait dans le fait d'optimiser au maximum le temps d'exécution. Pour cela nous avons utilisé des structures de données adaptées (table de hachage, file de priorité, représentation redondante des configurations, ...) et nous avons étudié la pertinence des heuristiques utilisées dans l'algorithme de type A*.

L'étude de ce problème est loin d'être inutile, la recherche du problème du **plus court chemin** étant un problème fondamental en informatique utilisé tous les jours par des millions d'utilisateurs sur des applications comme **Google Maps**.

A
ALGORITHME HEURISTIQUE

```
Data: h, initial_game_state
Result: True if the red car can exit, False otherwise
file ← PriorityQueue() ;
file.push((h(initial_game_state), 0, initial_game_state)) ;
seen ← ∅ ;
while file ≠ ∅ do
    priority, depth, game_state ← file.pop() ;
    if Winning_configuration(game_state) then
        return True ;
    end
    if game_state ∉ seen then
        seen.add(game_state) ;
        for next_game_state ∈ game_state.next_possible_moves() do
            file.push(h(next_game_state) + depth + 1, depth + 1, next_game_state) ;
        end
    end
end
return False ;
```

Algorithm 2: Heuristic algorithm

B
PREUVE DE L'ALGORITHME HEURISTIQUE

Pour commencer, la preuve de la **terminaison** est essentiellement la même que celle de l'algorithme de la question 3. Chaque configuration entre au maximum une fois dans la file et il y a un nombre fini d'état possible ce qui assure que l'algorithme se termine.

Maintenant, nous allons diviser la preuve de la **validité** en deux étapes. Tous d'abord, nous allons montrer que si une solution existe, une solution est trouvée par l'algorithme. Ensuite, nous allons montrer que la solution renvoyée par l'algorithme minimise le nombre d'étapes.

1) Encore, une fois, la preuve est la même que pour l'algorithme de la question 3. Nous allons raisonner par l'absurde en supposons qu'il existe une solution et que l'algorithme ne l'a pas trouvé (il a donc vidé entièrement la file). Nous allons représenter la solution par la succession de ses états $(s_0, ..., s_k)$. On a ainsi $\forall i \in [\![0, (k-1)]\!], s_{i+1} \in \texttt{next\_moves}(s_i)$.

Regardons les états qui sont entrés (et donc sortis !) de la file. Nécessairement, s_0 est entré dans la file et s_k n'est pas entré (sinon une solution aurait été trouvé). On peut donc considérer le plus petit i tel que s_i soit entré dans la file, mais pas s_{i+1}. L'existence d'un tel i est absurde, car s_i a nécessairement été pop et a donc obligatoirement ajouté s_{i+1} à la file.

2) Ici, la preuve est similaire à celle qu'on pourrait faire pour l'algorithme de la question 3, mais on va devoir utiliser les propriétés de l'heuristique qui nous sont données.

Dans un premier temps, nous allons montrer par induction structurelle les deux propriétés suivantes :

— les nœuds sont toujours pop par priorité croissante

— lorsque l'algorithme pop un nœud, la profondeur que l'algorithme a donné à ce nœud est sa vraie profondeur minimale.

Initialisation : Au début de l'algorithme, il n'y a aucun nœud pop. Tout est donc bien vérifié Après la première étape de l'algorithme, un seul nœud a été pop et donc l'ordre de pop a bien été respecté. Par ailleurs, le seul nœud ayant été pop est la configuration initiale et sa profondeur est bien 0.

Hérédité : On suppose que les n premiers nœuds ont bien été pop par priorité croissante et qu'il n'y a pas eu de mauvais *assignement* de profondeurs dans les nœuds qui ont déjà été pop. Considérons ce qu'il se passe lorsque l'on *pop* le nœud suivant s_{n+1}.

— Pour le premier critère, il suffit de vérifier que la priorité de s_{n+1} est supérieure ou égale à la priorité du nœud précédemment pop s_n. Il y a deux cas possibles :
  — soit s_{n+1} n'est pas arrivé dans la file suite au pop de s_n. Dans ce cas avant le pop de s_n, les deux états étaient dans la file de priorité et donc la priorité de s_{n+1} est supérieure ou égale à la priorité de s_n. (sinon il aurait été pop à sa place)
  — soit s_{n+1} est arrivé dans la file suite au pop de s_n. Dans ce cas-là, on a $\texttt{depth}(s_{n+1}) = \texttt{depth}(s_n) + 1$ et la consistance de l'heuristique nous donne $\texttt{h}(s_n) \leq \texttt{h}(s_{n+1}) + 1$ (car $k_{s_n,s_{n+1}} = 1$. On a donc bien $\texttt{priority}(s_n) \leq \texttt{priority}(s_{n+1})$.

)

— Pour le second critère, il suffit de vérifier qu'à s_{n+1} a bien été assignée la bonne profondeur. Comme $s_{n+1} \neq s_0$, la profondeur de s_{n+1} a été obtenue comme $\texttt{depth}(s_{n+1}) = p + 1$ où $p = \texttt{depth}(s')$ pour un certain s' "voisin" de s_{n+1} qui a déjà été *pop*. Par hypothèse de récurrence, la profondeur de s' est correcte. $\texttt{depth}(s_{n+1})$ est donc une borne supérieure de la vraie profondeur minimale de s_{n+1}.

  Raisonnons par l'absurde et supposons que la vraie profondeur q de s_{n+1} soit strictement inférieure à $p+1$. Il existe donc un chemin d'états $(c_0, ..., c_q)$ tel que $c_0 = s_0$ et $c_q = s_{n+1}$. Pareillement à la preuve du premier critère, on peut considérer le plus petit i tel que c_i ait été pop avant l'étape n, mais pas c_{i+1}. On a $\texttt{priority}(c_{i+1}) = \texttt{depth}(c_i)+1+\texttt{h}(c_{i+1})$. Or par hypothèse de récurrence, la profondeur de c_i est la vraie profondeur. Donc $\texttt{depth}(c_i) \leq i$. Par ailleurs, par consistance de l'heuristique, $\texttt{h}(c_{i+1}) \leq \texttt{h}(c_q) + q - i - 1$. On a donc $\texttt{priority}(c_{i+1}) \leq \texttt{h}(c_q) + q = \texttt{h}(s_{n+1}) + q$

  Ceci signifie qu'à l'étape $n+1$, il y a dans la file un nœud c_{i+1} avec une priorité inférieure à $\texttt{h}(s_{n+1}) + q$, mais que c'est le nœud s_{n+1} qui est pop avec une priorité supérieure de $\texttt{h}(s_{n+1}) + p$. Ceci est en contradiction avec le fonctionnement des files de priorité.

Ceci achève de démontrer par induction structurelle les deux propriétés.

Pour conclure, nous allons considérer un algorithme similaire excepté qu'au lieu de s'arrêter lorsqu'il trouve une solution, il continue jusqu'à avoir épuisé entièrement la file de priorité puis renvoie la première solution trouvée. Il est clair que ce nouvel algorithme renvoie exactement le même résultat que l'ancien. De plus, la preuve ci-dessus permet également de montrer que cet algorithme valide les deux propriétés énoncées.

Ces propriétés nous permettent alors d'affirmer que parmi toutes les solutions trouvables par l'algorithme, l'algorithme va renvoyer celle de priorité minimum (première propriété). Or pour une solution, l'heuristique vaut 0, donc la priorité est égale à la profondeur calculée. La profondeur calculée est égale à la vraie profondeur (deuxième propriété). Donc l'algorithme renvoie bien une solution de profondeur minimum.