SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Sorbonne Université

Projet Python

Développement d'une méthode heuristique pour la résolution d'un problème de trajectoire avec recouvrement

---

Kevin Ancourt - Adrien Talatizi

Master 2 IMPE-Meca
2019-2020

Table des matières

1 Introduction

Nous avons mené ce projet en parallèle afin de développer un code qui marche et de le réécrire en optimisé, d'où la différence entre le premier code envoyé publiquement et celui-ci. On garde cependant toutes les fonctions en faisant l'économie de certaines données suppressibles, ceci dans le but d'accélérer le code.

Le problème auquel nous faisons face n'admettant pas forcément de solution, nous avons opté pour une méthode heuristique en combinant l'aléatoire au principe de recouvrement *glouton* "ϵ-*greedy*". C'est à dire qu'on part à probabilité ϵ dans la direction qui contient le plus de cases non nettoyées (avec une selection aléatoire en cas d'égalité) ; et qu'on part dans une autre direction aléatoire avec une probabilité $1-\epsilon$. Cette méthode est utilisée par exemple pour trouver une solution au problème du voyageur de commerce (bien que des méthodes comme *Dijkstra* en Monte-Carlo soient également très efficaces). L'aléatoire intervient également dans notre problème puisque pour selection du robot à déplacer.

Pourquoi l'aléatoire ?
Nous avons, dans un premier temps, souhaité évaluer la complexité du problème, et le nombre d'itérations nécessaires à une résolution parfaite. Prenons notre quatrième configuration : 3 robots, 3 directions maximum chacun (= 6 combinaisons) et une solution à 16 itérations. Si nous voulons couvrir tous les cas possibles dans le but d'avoir la certitude de ressortir la meilleure trajectoire, il nous aurait fallu évaluer **au minimum** 6^{16} cas, soit environs $3 \cdot 10^{12}$ cas, et stocker les différents chemins dans un fichier texte pour les essayer un à un. C'était inconcevable : la puissance d'un processeur étant de l'ordre de 10^9 Hz, il nous aurait fallu au minimum 10^3 secondes = 17 minutes environs, uniquement pour parcourir les données. Or, on ne connait normalement pas la solution de tels problèmes, il aurait fallu partir d'un nombre d'itérations beaucoup plus grand !

Remarque : Dans cette première estimation du temps de calcul, on considère que la complexité de tester un cas est $O(1)$, ce qui n'est pas du tout le cas : le temps de calcul nécessaire pour tester tous les cas est en réalité bien supérieur (accéder à la case, colorier les cases parcourues etc).

Ainsi, dans notre programme, rien n'assure théoriquement la découverte de la meilleure solution. La génération de nombres sur PC n'étant pas parfaitement aléatoire, on ne peut même pas affirmer qu'à l'infini, on tombera sur le bon chemin. Cependant, le module `random` de Python est un générateur de nombres pseudo-aléatoires qui permet d'approcher longtemps un comportement aléatoire. De ce fait, on estime pouvoir approcher une *bonne* solution, même s'il serait trop long de prouver que c'est la meilleure (comme expliqué précédemment).

Note : La logique de notre programme est celle-ci : "Trouver à coup sûr la meilleure solution serait trop long, nous assurons d'en trouver une très bonne en un temps réduit."

Afin d'économiser du temps sur des cas simples, l'utilisateur peut modifier une valeur qui influe sur le temps de calcul : le nombre de chemins maximum qu'on teste. En effet, les premiers cas sont assez rapides, on peut donc se permettre de générer 5000 chemins. Tandis que pour les derniers cas tests, il en faudra 100 à 200 fois plus...

2 Présentation des fonctions

2.1 ReadingFile

La fonction `ReadingFile` prend en entrée une chaîne de caractères qui correspond au nom du fichier `.txt` qu'on souhaite lire. On lit le fichier et on renvoie 5 données : `nx` le nombre de lignes ; `ny` le nombre de colonnes ; `nd` le nombre d'aspirateurs ; `list_line` qui, pour chaque case, contient l'hexadécimal décrivant la présence de murs ; et enfin `robots` qui décrit la position actuelle des robots, ainsi que leur couleur. Ci-dessous une capture pour l'appel de *"Case_Aspi_R_4.txt"* :

```
ROBOTS :  [['B', '0', '0'], ['R', '4', '0']]
LISTLINE :  [['9', '5', '1', '1', '5', '3'], ['A', '9', '0', '0', '3', 'A'],
 ['8', '0', '0', '0', '0', '2'], ['8', '0', '0', '0', '0', '2'], ['8', '6',
'8', '0', '0', '2'], ['C', '5', '4', '4', '4', '6']]
```

FIGURE 1 – Exemple de list_line et robots

2.2 InitGrid

La fonction `InitGrid` prend en argument les dimensions de la grille `nx` et `ny` pour renvoyer la matrice `grid`, t.q. `grid[i][j]` $= \mathbb{1}_{Aspi\_sur\_grid_{i,j}}$ au départ. Cette grille se verra complétée au cours du programme. Elle contient 1 là où un aspirateur est déjà passé, 0 sinon.

2.3 AuthorizedMove

`AuthorizedMove` prend en argument la position `(x,y)` d'un robot, `list_line`, `robots` et `(X,Y)` les coordonnées actualisées de tous les robots. Elle renvoie `W,S,E,N`, valant chacun 0 ou 1 selon la présence d'un mur ou d'un apirateur dans la direction correspondante (1 si la direction est verrouillée, 0 sinon).

2.4 Direction_Chosen

`Direction_Chosen` prend 4 entrées (`W,S,E,N`) décrivant les directions disponibles. Elle trie les directions disponibles et en prend une aléatoirement (appelée `direction` $\in \{0, 1, 2, 3\}$) parmi elles (grâce à la fonction `choice` de `random`) qu'elle renvoie en plus de `W,S,E,N` concaténés en liste.

2.5 Movement

Cette fonction déplace l'aspirateur choisi dans une direction donnée sur 1 case qui est coloriée en conséquence. Sont renvoyées la nouvelle position de l'aspirateur `(x,y)` et la grille actualisée `grid`.

2.6 Movement_Loop

En plus d'actualiser `test` (le chemin déjà parcouru), cette fonction boucle dans une direction donnée sur `Movement` jusqu'à atteindre un mur ou un aspirateur et retourne : la position de l'aspirateur actualisée, la grille mise à jour et `test` qui décrit le chemin déjà parcouru à présent actualisé.

Notons que c'est lors de cette fonction qu'est mis en place la stratégie de ϵ-Glouton, puisqu'on choisit d'agir de façon gloutonne selon une probabilité.

2.7 Glout

Cette fonction met en place le système *Glouton*. C'est une simulation de plusieurs déplacements qui compte pour chaque direction le nombre de cases non nettoyées. On verouille ainsi les directions qui possèdent strictement moins de cases à nettoyer que les autres, et on les renvoie à travers `W,S,E,N`.

2.8 CheckGrid

Cette fonction est très simple, elle compte le nombre de cases non nettoyées sur la grille et le renvoie. Elle servira comme test d'arrêt sur la complétion d'un chemin.

2.9 Try

Cette fonction est préliminaire au programme final `mainprog`. On dirige ainsi jusqu'à complétude du remplissage de la grille (ou dépassement du nombre de déplacements maximum) des aspirateurs pris aléatoirement qu'on bouge selon les directions disponibles (avec un ϵ-Glouton). On renvoie le chemin alors parcouru `test`, le nombre d'itérations `it` et la grille `grid`.

2.10 mainprog

Cette fonction est notre fonction principale. Elle prend en argument un nom de fichier `filename` ainsi qu'un nombre maximum de chemins à essayer `i_max`, fixé par défaut à 10^6. Elle essaie `i_max` chemins différents. La taille de ceux-ci rétrécit en fonction de la taille minimale obtenue sur un chemin complétant toute la grille ; cela afin de ne pas tester inutilement des chemins longs. On affiche ainsi : le **nombre maximum de déplacements**, le **chemin** qui correspond et la **grille finale**, normalement bien remplie (donc ne contenant que des 1). Ci-dessous un exemple de sortie pour le cas test 0 : `"Case_Aspi_R_0.txt"`.

```
6
['BE', 'RE', 'BN', 'RS', 'BW', 'RW']
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

FIGURE 2 – Exemple de sortie de `mainprog` pour le cas test 0.

3 Résumé du fonctionnement du programme

— On lit nos données.
— On choisit successivement un aspirateur et une direction (donnée par ϵ-Glouton) jusqu'à constituer un chemin complétant la grille.
— On répète l'opération suffisamment de fois (par défaut 10^6) pour être sûr de tomber sur un chemin relativement court.
— On renvoie la grille, un chemin parcouru parmi les plus courts et le nombre de déplacements ainsi effectués.

4 Résultats

Nous avons testé les 8 cas tests à notre disposition lors de ce projet, nous avons obtenu les résultats suivants :

— Le test 0, nous avons un chemin de taille 6 : ['BE', 'BS', 'RE', 'BW', 'RN', 'RW'] obtenu en 0.32 secondes

— Le test 1 nous avons obtenue un chemin de taille 6 : ['BE', 'BS', 'BW', 'RE', 'BN', 'BE'] en 0.37 secondes

— Le test 2 nous avons obtenu un chemin de taille 10 : ['RN', 'RE', 'RS', 'BS', 'BE', 'BN', 'BE', 'BS', 'RE', 'RN'] en 0.58 secondes

— Le test 3 nous avons obtenu un chemin de taille 16 : ['YE', 'BE', 'YS', 'BS', 'YW', 'RE', 'RN', 'RW', 'BW', 'BN', 'RS', 'BE', 'YN', 'BS', 'RE', 'BW'] en 225.37 secondes

— Le test 4 nous avons obtenu un chemin de taille 12 : ['RE', 'BS', 'RN', 'RE', 'RS', 'BE', 'RE', 'RN', 'RW', 'BN', 'RE', 'RS'] en 563.20 secondes

— Le test 5 nous avons obtenu un chemin de taille 12 : ['BW', 'BS', 'BE', 'BN', 'BW', 'RS', 'RE', 'BS', 'RW', 'RN', 'RW', 'RS'] en 28.43 secondes

— Le test 6 nous avons obtenu un chemin de taille 14 : ['RS', 'BE', 'BS', 'BW', 'BN', 'RW', 'BW', 'BS', 'BE', 'BN', 'BE', 'BS', 'RE', 'RN'] en 152.15 secondes

— Le test 7 nous avons obtenu un chemin de taille 14 : ['BE', 'RS', 'RW', 'BS', 'BN', 'RN', 'RE', 'BE', 'RS', 'BS', 'RE', 'RN', 'RW', 'RS'] en 36.85 secondes

Nous avons bien trouvé tous les résultats attendus. Par ailleurs, nous avons décidé de faire le dessin de deux chemins (le cas 4 et le cas 6) afin de démontrer le bon fonctionnement de notre algorithme, voir les deux figures ci-dessous :

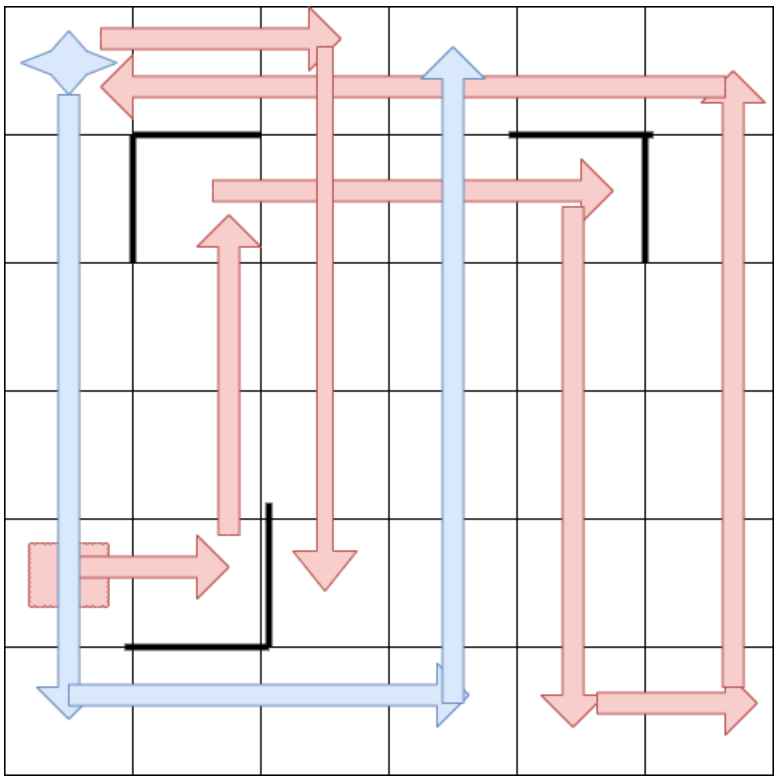

FIGURE 3 – Chemin optimal pour le cas test 4.

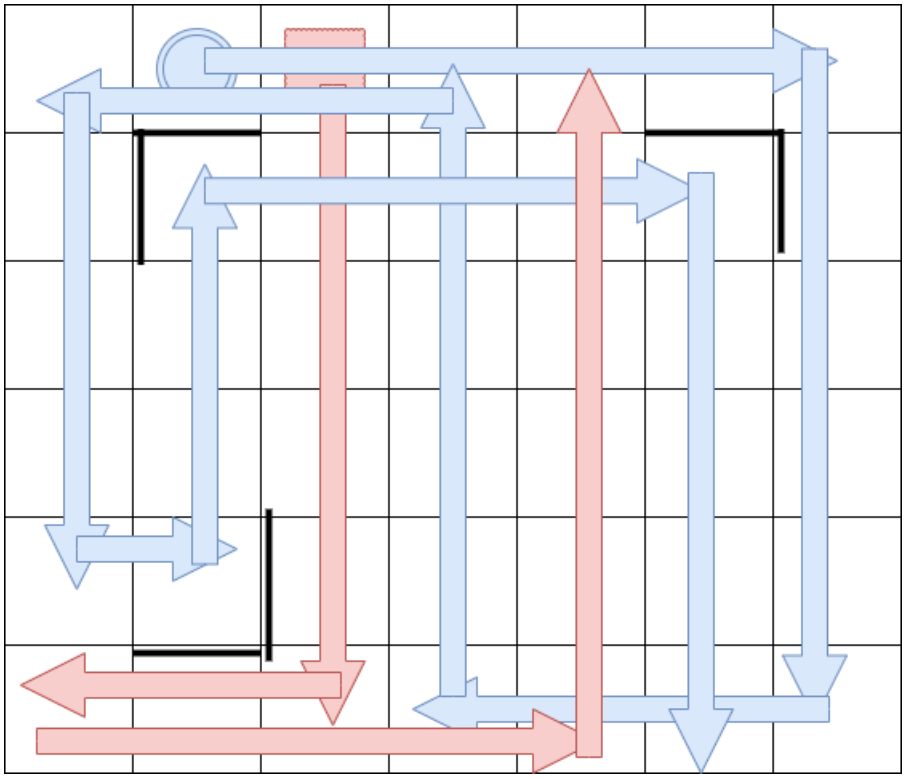

FIGURE 4 – Chemin optimal pour le cas test 6.

On remarque que nos deux chemins recouvrent effectivement la grille, nous avons donc trouvé des solution viables.

5 Conclusion

Malgré l'absence de formules mathématiques nous permettant de connaitre la longueur d'un chemin optimal, en combinant un algorithme déterministe et un algorithme aléatoire, nous sommes tout de même en mesure de trouver une solution à notre problème initial. Afin d'assurer une certaine robustesse, il faudrait dans l'idéal choisir un nombre d'itérations assez grand afin de laisser le temps à l'algorithme de converger. Par ailleurs, nous choisissons un ϵ proche de 0.5, afin de ne pas négliger l'effet *aléatoire* sur des cas compliqués (où celui-ci se révèle efficace) ni l'effet *glouton* qui permet un remplissage efficace de la grille.

Nous estimons que ce problème se trouve dans la classe des problèmes NP complets, ce qui signifie que nous devrions essayer toutes les stratégies afin de déterminer laquelle est la meilleure. Il n'était pas concevable de mettre en place de telles recherches, nous avons donc cherché un algorithme robuste et convergeant avant 10 minutes.