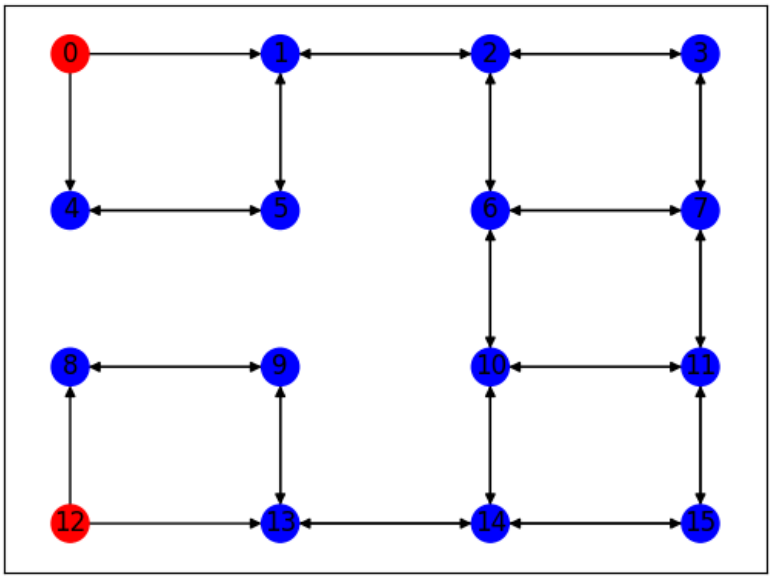
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Sorbonne Université

Projet Python

Quelques algorithmes pour résoudre un problème de recherche de chemin dans un graphe



Kenneth Assogba - Alexis Squarcioni

Master 2 IMPE-Meca
2019-2020

Table des matières

Introduction

L'objectif du projet est de coder en Python3 un algorithme qui donnera les instructions de direction à un ou plusieurs robots à partir du plan d'une pièce afin de passer au moins une fois dans toutes les cases avec un nombre de déplacement minimal.

Pour atteindre cet objectif, nous devons dans un premier temps lire et traiter les données issues du plan de la pièce. Ensuite, il faudra stocker ces données dans des structures de données adéquates. Il s'agira enfin d'écrire un algorithme qui donnera les instructions de déplacement aux robots afin de nettoyer la pièce de façon optimale. En outre, nous aurons à vérifier et valider notre code à l'aide de cas tests.

1 Structures de données

Nous avons plusieurs informations a stocker et a mettre a jour au fur et a mesure de nos déplacements, notamment : la position des robots, les cases nettoyés et les déplacements possibles. Nous prendrons l'exemple de Case_Aspi_R_2 pour illustrer.

•La position des robots est un dict de clé : robotColor et valeur : robotPosition.

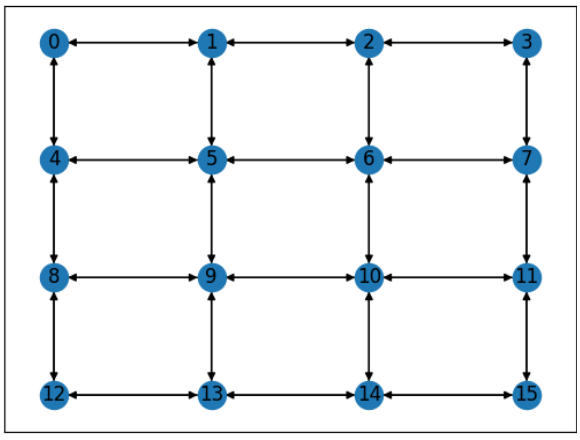
```
robots = {'B': 0, 'R': 12}
```

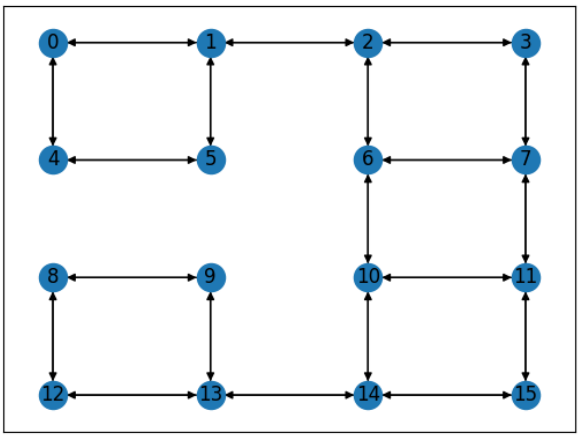
•Les cas nettoyés sont représentés par une liste. Les cas sont numérotés (par ligne) à partir de 0. La valeur est 1 pour une case nettoyée et sinon 0 :

```
cleanedBoxes = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
```

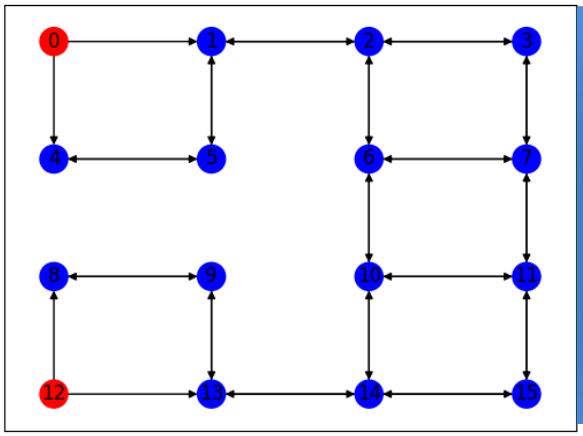
1.1 Déplacements possibles

Nous avons choisi de représenter notre grille par un graphe orienté. Ce graphe représente les déplacements possibles en partant d'une case en tenant compte des murs internes, des murs externes et des robots. Nous le stockons dans un dict car : la recherche dans un dict est de complexité $O(1)$ [1]. Ainsi nous avons successivement les graphes suivants : 1) sans murs et sans robot ; 2) avec murs et sans robot ; 3) avec murs et robots.





1. 5 tips to speed up your Python code. https://pybit.es/faster-python.html




```
graph = {0: [-1, 4, 1, -1], 1: [-1, 5, 2, -1],
         2: [1, 6, 3, -1], 3: [2, 7, -1, -1],
         4: [-1, -1, 5, -1], 5: [4, -1, -1, 1],
         6: [-1, 10, 7, 2], 7: [6, 11, -1, 3],
         8: [-1, -1, 9, -1], 9: [8, 13, -1, -1],
         10: [-1, 14, 11, 6], 11: [10, 15, -1, 7],
         12: [-1, -1, 13, 8], 13: [-1, -1, 14, 9],
         14: [13, -1, 15, 10], 15: [14, 11]}
```

Nous créons enfin une classe *Cleaning* contenant nos données et méthodes.

2 Algorithmes

Les robots sont toujours choisis aléatoirement parmi ceux qui peuvent se déplacer. Puisque tous nos algorithmes sont aléatoires, nous relançons la méthode *Cleaning.cleaning()* plusieurs fois et ne retenons que le chemin optimal. Ce nombre d'itération dépends de la position des murs internes, du nombre de robots et de leur positions. Après étude des cas tests qui nous ont été soumis, nous fixons le nombre d'itérations pour chacun d'eux via la fonction *setParameters* afin d'éviter les calculs inutiles sur les cas simples. Pour les cas tests inconnus, un nombre d'itérations de sécurité=500000 est fixé, le temps calcul est tout de même raisonnable (<2min). Dans la suite, nous décrivons les stratégies élaborées et notre façon de les jouer.

2.1 Aléatoire uniquement

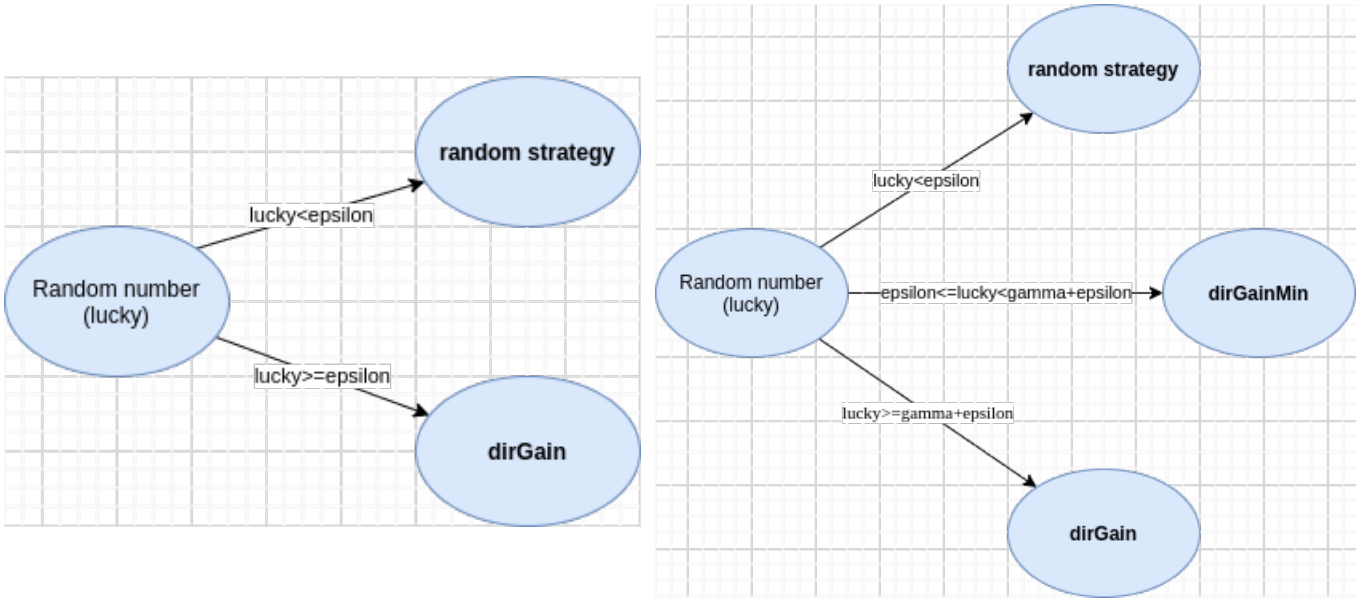
Ici, après le choix du robot, une direction aléatoire parmi les directions possibles est choisie. On peut théoriquement obtenir une réponse optimale pour tous les problèmes en temps infini et les performances en temps de calcul sont médiocres. Python utilise Mersenne Twister comme générateur et a une période de $2^{19937} - 1$. Une amélioration consisterait a interdire les allés-retour.

2.2 Glouton uniquement

La stratégie random étant très coûteuse, nous avons pensée a une heuristique afin de nettoyer le maximum de cases. *dirGain* détermine pour un robot donné la direction de gain maximal, c'est a dire la direction dans laquelle le robot nettoies le plus de cases. En cas d'égalité, un choix aléatoire est fait. Cette stratégie gloutonne revient a faire le choix optimal a chaque itération. Les temps de calculs ont été divises par 10, néanmoins cette stratégie est loin d'être globalement optimale (ou ne l'est que dans des cas particuliers) d'autant que en amont, le choix du robot est fait aléatoirement.

2.3 Epsilon-greedy

Nous introduisons un peu d'aléatoire via un paramètre *epsilon*. À chaque étape, un nombre aléatoire $\in [0, 1)$ est généré par la méthode *random.random()*. Si le nombre est inférieur à epsilon à cette étape, le robot choisit une direction aléatoire et s'il est supérieur à epsilon à cette étape, le robot choisit une action basée sur *dirGain*. Une amélioration consisterai a faire varier epsilon selon l'itération afin de plus ou moins influencer le greedy.



2.4 Epsilon-Gamma-greedy

Puisque nous n'obtenions pas des résultats satisfaisant sur certains cas tests, nous avons décidé d'implémenter une heuristique supplémentaire. En effet la solution optimale du cas test 4 revient a faire nécessairement de petits déplacements au début. Cette nouvelle heuristique : *dirGainMin* détermine la direction de gain minimal non nul. En cas d'égalité, un choix aléatoire est fait. Pour faire jouer aléatoirement nos 3 stratégies, nous introduisons un nouveau paramètre γ. Désormais, la stratégie random sera jouée selon la probabilité ϵ, *dirGainMin* selon la probabilité γ. Les paramètres gamma et epsilon étant variables selon le cas test, nous les réglons avec la méthode *setParameters*.

2.5 Petites améliorations

Chaque fois qu'un chemin de taille plus petite que les précédents est obtenu, sa taille est enregistrée dans *lenoptimal*. Des qu'un nouveau chemin atteint le *lenoptimal* courant il est stoppé. Cela permet d'économiser beaucoup de calculs inutiles.
Nous avons également écrit un destructeur pour supprimer les copie de *robots*. L'efficacité de cela semble minime puisque Python détruit déjà les objets non utilises. Enfin nous remarquons que les temps de calculs sont divises par 20 (voir plus) en commentant les nombreux print (d'où le booléen verbose), nous avons remarqué que les print consomment beaucoup grâce a la commande *python -m cProfile*.

3 Résultats et performance

3.1 Résultats

Nous présentons les résultats que nous avons obtenus. L'algorithme étant aléatoire, nous pouvons obtenir un nouveau chemin optimal a chaque exécution. Le nombre d'itérations est choisi afin d'augmenter fortement la probabilité de trouver un chemin optimal sans pour autant sacrifier le temps de calcul. Les tests sont effectues sur une Machine Debian 10 pourvue d'un Intel Core i5-7200U CPU @ 2.50GHz × 4 avec 7.7 GiB de mémoire. IW=Murs internes. Les cas tests étoilés * sont ceux ou nous obtenons de meilleurs résultats que dans les tests unitaires fournis par l'enseignant. Enfin, le temps d'exécution de tous les tests unitaires par *unittest* est de **124.7s**.

	nx*ny*nd	**IW**	**Depl**	**iter**	ϵ	γ	**Time(s)**	**Path**
0	4*3*2	0	6	200	0.1	0	0.02	RE BE RS RW RN RE
1	4*4*2	0	6	200	0.1	0	0.02	RE RN BE RW BS BW
2	4*4*2	4	10	10000	0.1	0	1.71	BS BE RN BN RE BE BS RS BW BN
3	6*6*3	8	16	200000	0.15	0.2	67.81	YE YS BE BS YW RE RN BW RW RS YN BN BE BS BW RE
4	6*6*2	6	**12***	60000	0.15	0.8	14.77	**RE BS RN RE RS BE RE RN RW BN RE RS**
5	6*6*2	6	12	20000	0.1	0.1	4.97	RS BE BS RE BN BW BS BE BN RN RW RS
6	6*7*2	6	**14***	200000	0.2	0.1	64.42	**RS BE BS BW RW BN BW BS BE BN BE BS RE RN**
7	6*7*2	6	14	40000	0.15	0	11.87	BE RS RW RN RE RS BS RN RE RS BE BN BW BS

3.2 Améliorations envisagées

Un enjeu important est le choix judicieux du robot a déplacer. Nous n'avons pas encore trouve de stratégie adéquate pour cela. Ensuite, il aurait été intéressant d'améliorer *setParameters* afin d'obtenir le nombre d'itérations, les paramètres epsilon et gamma en se basant les les données de la grille plutôt que sur le nom du cas test.
Enfin nous avons songé à implémenter un algorithme de colonie de fourmis, c'est a dire marquer dans chaque case le nombre de fois qu'elle a été parcourue et aller de préférence dans les directions ou nous sommes le moins allées. Mais nous ne savions quelle stratégie appliquer au début du jeu.

Conclusion

Du point de vue des résultats, nous avons bien obtenu les résultats escomptés. C'est-à-dire que nous avons écrit un algorithme qui détermine le chemin optimal pour une pièce donnée. Ces résultats ont été validés par huit cas tests.
D'autre part, ce projet nous a permis d'améliorer nos connaissances en Python. Nous avons pu implémenter différents algorithmes que nous avons à chaque fois testé. Ce qui nous a permit de vérifier et valider notre code à la manière de ce qui se fait dans un bureau d'étude. Notre code a

également été documenté en anglais et en français. Enfin, nous avons rédigé un rapport détaillé sur les résultats que nous avons obtenus et les méthodes utilisées.