

Projet *Labyrinthe*

Algorithmes et Structures de Donnée

Juan-Carlos Barros et Daniel Kessler

14 mai 2021

Cours d'*Algorithmes* et *Structures de donnée*

Projet *Labyrinthe*

- *Algorithme*
A*
- *Structure de Donnée*
Priority Queue

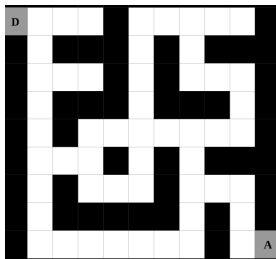


Table des matières

1 Quel algorithme pour résoudre quel problème ?

- Choix du problème
- Choix de l'Algorithme

2 Algorithme A*

- Pseudo-Code
- Heuristique et Priorité
- Structure de données "Priority Queue"
- Idée de preuve
- Exemple de résolution
- Complexité

3 Tests avec Python

4 Conclusion

5 Références

Un labyrinthe, plusieurs problèmes

- Cherche-t-on un chemin quelconque ?
 - ▶ Oui, on ne traversera le labyrinthe qu'une seule fois.
 - ▶ **Non, on veut le chemin le plus court**, pour peut-être le réutiliser.
- Connait-on les coordonnées de la sortie dès le départ ?
 - ▶ **Oui, et cette information pourra nous aider.**
 - ▶ Non, le lieu de la sortie fait partie des inconnues.

Un problème, plusieurs solutions

- Breadth-First Search

- ▶ garantit de trouver une solution si elle existe
- ▶ solution optimale si tous les pas sont égaux

- Dijkstra

- ▶ choisit où explorer selon les distances déjà parcourues
- ▶ garantit de trouver le plus court chemin

- A*

- ▶ nécessite de connaître les coordonnées de la sortie
- ▶ choisit où explorer selon les distances déjà parcourues et la distance à la sortie

Algorithme A*

pseudo-code

Démarrer une file d'attente avec la cellule de départ D , une liste de prédécesseurs avec $\{D : Nil\}$ et une liste de coûts d'accès avec $\{D : 0\}$.

Tant que la file d'attente n'est pas vide,

- extraire (pop) la cellule prioritaire C de la file d'attente
- si C est la cellule d'arrivée A , retourner le chemin qui y amène (via backtracking sur les prédécesseurs)
- sinon, pour chaque voisin V de C qui n'est pas déjà accessible à moindre coût
 - ▶ mémoriser le prédécesseur de V et le coût d'accès à V
 - ▶ ajouter V à la file d'attente

Heuristique et Priorité

La priorité d'une cellule C en attente est le coût estimé d'un chemin complet passant par cette cellule.

$$\text{priorité} = \text{coût_réel} (D \rightarrow C) + \text{coût_estimé} (C \rightarrow S)$$

La distance restante depuis une cellule jusqu'à l'arrivée doit être estimée *sans jamais la surestimer*.

- La **distance de Manhattan** $|\Delta x| + |\Delta y|$ est un bon estimateur si les mouvements permis sont horizontaux et verticaux.
- Une **heuristique nulle** ramène A^* à l'algorithme de Dijkstra (ou Breadth-First Search sur grille carrée).

File d'attente : “Priority Queue”

- Structure permettant insertion avec priorité et “pop” rapide de l’élément prioritaire
- Implémentation en Python en tant que **binary heap** avec le module *heapq*
- Dans cette implémentation, vérifier si vide en $O(1)$, insertion et “pop” en $O(\log(n))$ où n est le nombre d’objets en attente¹

1. cf. <https://www.cs.princeton.edu/wayne/kleinberg-tardos/pdf/BinomialHeaps.pdf>

Preuve de l'algorithme (grandes lignes)

L'heuristique $h(C)$ qui estime le chemin restant depuis une cellule C doit satisfaire deux conditions.

- ❶ **Monotonicté** : sorte d'inégalité triangulaire faible

$h(C_1) \leq r(C_1, C_2) + h(C_2)$ où C_1, C_2 sont deux cellules, $r(C_1, C_2)$ est la distance réelle entre elles.

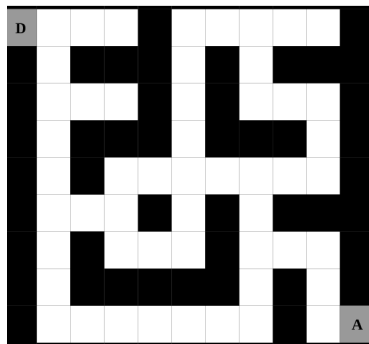
Cette propriété garantit de trouver la sortie, sans se “perdre” dans des boucles éventuelles.

- ❷ **Admissibilité** : l'heuristique ne surestime jamais une distance

Cela garantit qu'il n'y a pas de chemin plus court que celui trouvé.

Par contradiction : s'il y en avait un, il aurait été estimé correctement ou sous-estimé, et serait donc prioritaire par rapport à un chemin complet trop long, vu qu'**un chemin complet a une priorité calculée uniquement en coût réel.**

Exemple de résolution



0	1	2	3		15	16	17	18	19	
---	---	---	---	--	----	----	----	----	----	--

Complexité

Le pire des cas sera réalisé par un labyrinthe dont le meilleur chemin revient souvent en arrière (s'éloigne de l'arrivée). Dans ce cas, aucun gain n'est réalisé par rapport à l'algorithme de Dijkstra (ou "heuristique nulle").

Dans ce cas, on aura visité toutes les N cellules. Les coûts de lecture/écriture dans la file d'attente sont en $O(\log(n))$ où n est le nombre d'éléments dans la "frontière" d'exploration, donc $n \sim O(\sqrt{N})$. Les autres opérations sont à coût comparable ou moindre. L'ensemble de la recherche sera donc en $O(N \log(N))$.

Cependant, le "worst case" ne rend pas justice à A^* dont le but est justement d'éviter la plupart du temps le "worst case" avec un bon choix d'heuristique.

Tests avec Python

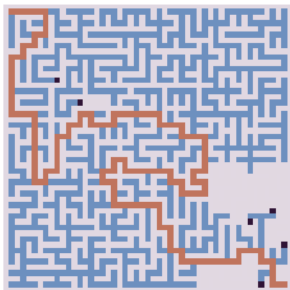
```
marge = QueuePrioritaire(grid.start)
cout_reel = {grid.start: 0}
parent = {grid.start: None}

while True:
    noeud_courant = marge.pop()
    if noeud_courant is None:
        raise ValueError("la grille n'a pas de solution")
    if noeud_courant == grid.out:
        break # chemin optimal trouvé
    # ... traiter noeud courant
```

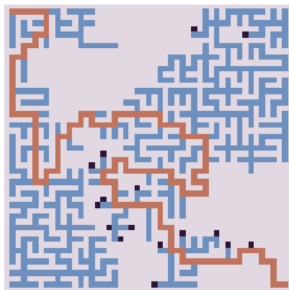
la suite dans : https://github.com/Dalker/ASD_labyrinthe/

Tests avec Python

A* with null heuristic



A* with Manhattan heuristic



Tests avec Python

```
* Comparaison heuristique nulle vs Manhattan distance *  
solveur1 = heuristique 0, solveur2 = heuristique Manhattan  
30x30 : generate=0.1019s solve1=0.0060s solve2=0.0022s  
40x40 : generate=0.1814s solve1=0.0096s solve2=0.0086s  
50x50 : generate=0.5002s solve1=0.0228s solve2=0.0204s  
60x60 : generate=0.8701s solve1=0.0280s solve2=0.0199s  
70x70 : generate=1.0461s solve1=0.0451s solve2=0.0391s  
80x80 : generate=1.4218s solve1=0.0563s solve2=0.0423s
```

Conclusion

- Fourmis
- Robot-Aspirateur

Références

- Liste des sources consultées :
`https://github.com/Dalker/ASD_labyrinthe/wiki/Sources`
- Notre implémentation en Python, avec tests temporels et “tests visuels” (animations) :
`https://github.com/Dalker/ASD_labyrinthe/tree/main/implementation`