

# Projet *Labyrinthe*

## Algorithmes et Structures de Donnée

Juan-Carlos Barros et Daniel Kessler

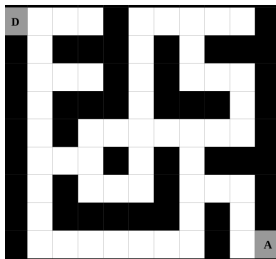
31 mai 2021

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

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

## Projet *Labyrinthe*

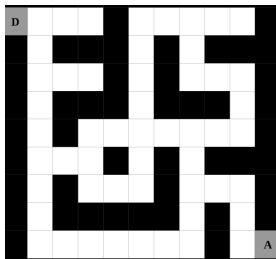
- Algorithme
- Structure de Donnée



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

## Projet *Labyrinthe*

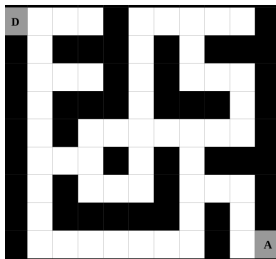
- Algorithme  
A\*
- Structure de Donnée



# 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

- Ants
- Robots

## 5 Références

# Un labyrinthe, plusieurs problèmes

- Cherche-t-on un chemin quelconque ?

# 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.



# 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.

# 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 ?

# 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.

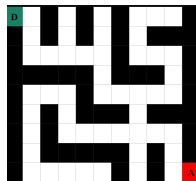
# Un labyrinthe, plusieurs problèmes

- 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 (même coût)



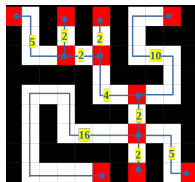
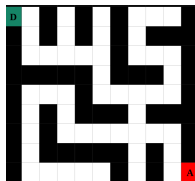
# Un problème, plusieurs solutions

- Breadth-First Search

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

- Dijkstra

- ▶ choisit où explorer selon les distances déjà parcourues
- ▶ garantit de trouver le plus court chemin (en tenant compte des coûts)



# Un problème, plusieurs solutions

- Breadth-First Search

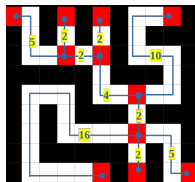
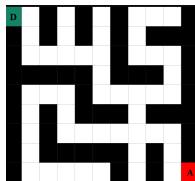
- ▶ garantit de trouver une solution si elle existe
- ▶ solution optimale si tous les pas sont égaux (même coût)

- Dijkstra

- ▶ choisit où explorer selon les distances déjà parcourues
- ▶ garantit de trouver le plus court chemin (en tenant compte des coûts)

- A\*

- ▶ nécessite de connaître les coordonnées de la sortie
- ▶ choisit où explorer selon les distances déjà parcourues et une estimation de 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  $\{D : 0\}$ .



# 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,

# 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

# 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)

# 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

# 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  $V$  (donc  $C$ ) et le coût d'accès à  $V$

# 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  $V$  (donc  $C$ ) et le coût d'accès à  $V$
  - ▶ insérer (push)  $V$  dans la file d'attente

# Heuristique et Priorité

La priorité d'une cellule  $C$  en attente est le coût total (estimé) d'un chemin passant par cette cellule.  $\text{priorité} = \text{coût\_réel} (D \rightarrow C) + \text{coût\_estimé} (C \rightarrow A)$

# Heuristique et Priorité

La priorité d'une cellule  $C$  en attente est le coût total (estimé) d'un chemin passant par cette cellule.  $\text{priorité} = \text{coût\_réel} (D \rightarrow C) + \text{coût\_estimé} (C \rightarrow A)$

La distance restante depuis une cellule jusqu'à l'arrivée doit être estimée *sans jamais la surestimer*.  $\rightarrow$  choix d'heuristique !



# Heuristique et Priorité

La priorité d'une cellule  $C$  en attente est le coût total (estimé) d'un chemin passant par cette cellule.  $\text{priorité} = \text{coût\_réel} (D \rightarrow C) + \text{coût\_estimé} (C \rightarrow A)$

La distance restante depuis une cellule jusqu'à l'arrivée doit être estimée *sans jamais la surestimer*.  $\rightarrow$  choix d'heuristique !

- La **distance de Manhattan**  $|\Delta x| + |\Delta y|$  est un bon estimateur si les mouvements permis sont horizontaux et verticaux.

# Heuristique et Priorité

La priorité d'une cellule  $C$  en attente est le coût total (estimé) d'un chemin passant par cette cellule.  $\text{priorité} = \text{coût\_réel} (D \rightarrow C) + \text{coût\_estimé} (C \rightarrow A)$

La distance restante depuis une cellule jusqu'à l'arrivée doit être estimée *sans jamais la surestimer*.  $\rightarrow$  choix d'heuristique !

- 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 si on a un coût identique pour chaque mouvement).

# File d'attente : “Priority Queue”

- Structure permettant “push” 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  $O(1)$ , “push” et “pop” en  $O(\log(n))$  où  $n$  est le nombre d’objets en attente<sup>1</sup>

---

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.

# Preuve de l'algorithme (grandes lignes)

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

- ① **Monotonicité** : 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.

# Preuve de l'algorithme (grandes lignes)

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

- 1 **Monotonicité** : 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.

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

# Preuve de l'algorithme (grandes lignes)

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

- ① **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.

- ② **Admissibilité** : l'heuristique ne surestime jamais une distance

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

# Preuve de l'algorithme (grandes lignes)

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

- ① **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.

- ② **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 :



# 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.

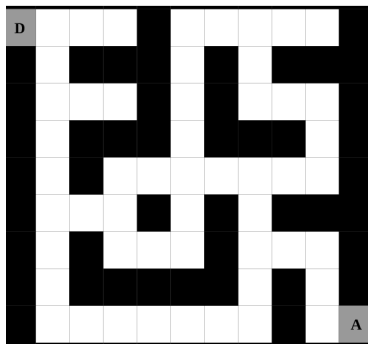
Cette propriété garantit de trouver la sortie, sans se “perdre” dans des boucles éventuelles.

- 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 en coût réel. (coût heuristique de cellule A = 0)**

# Exemple de résolution



## Objectif

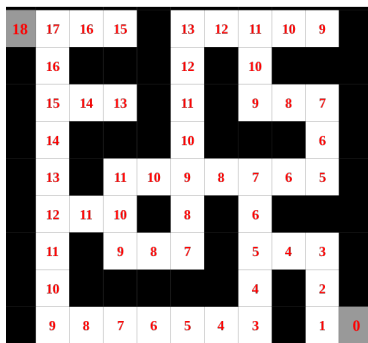
Trouver le chemin le plus court entre le  
Départ et l'Arrivée

# Exemple de résolution

0	1	2	3		15	16	17	18	19	
	2				14		18			
	3	4	5		13		19	18	17	
	4				12				16	
	5		9	10	11	12	13	14	15	
	6	7	8		12		14			
	7		9	10	11		15	16	17	
	8						16		18	
	9	10	11	12	13	14	15		19	20

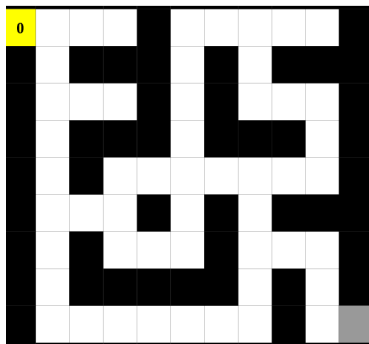
Chaque noeud est à une distance réelle du départ, qui sera découverte en cours de route.

# Exemple de résolution



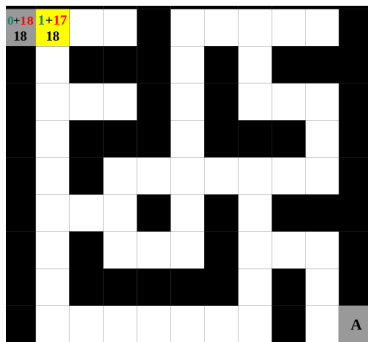
Estimation de la distance réelle de l'arrivée pour chaque cellule (Heuristique "Manhattan").

# Exemple de résolution



Le départ est mis en file d'attente, avec une priorité 0.

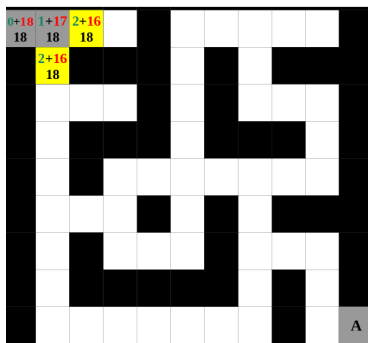
# Exemple de résolution



Le seul voisin est évalué :

- coût réel pour y accéder : 1
- coût heuristique pour la suite : 17
- coût heuristique total (priorité) : 18

# Exemple de résolution

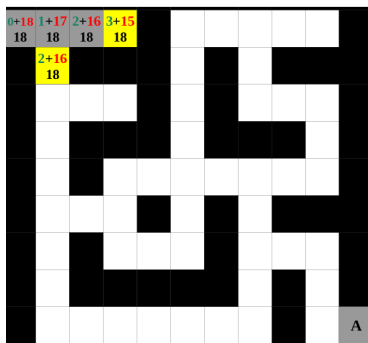


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



Légende :

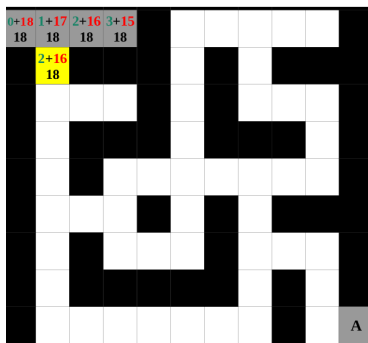
- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois deux choix ont la même priorité, le choix est arbitraire.



# Exemple de résolution



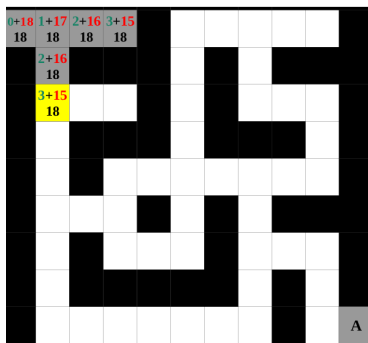
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois deux choix ont la même priorité, le choix est arbitraire.

# Exemple de résolution

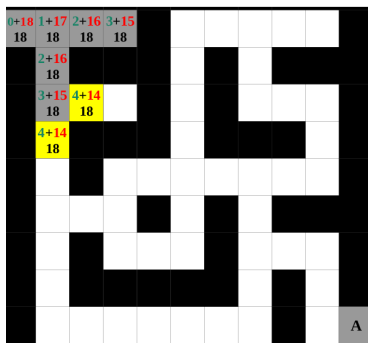


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

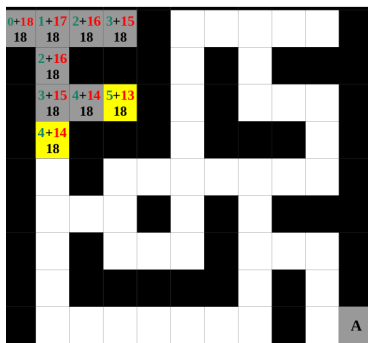


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

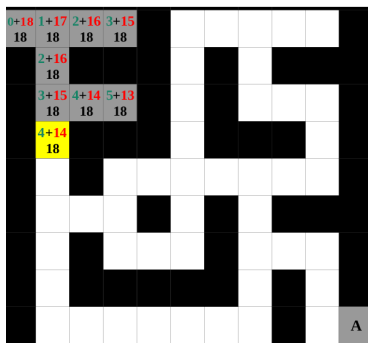


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

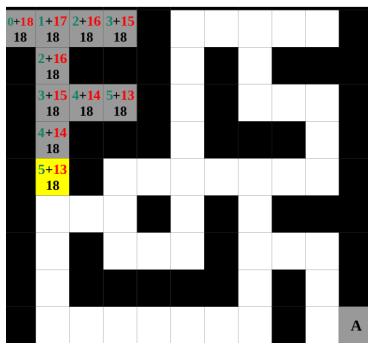


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

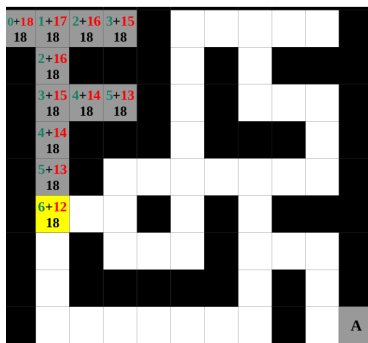


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

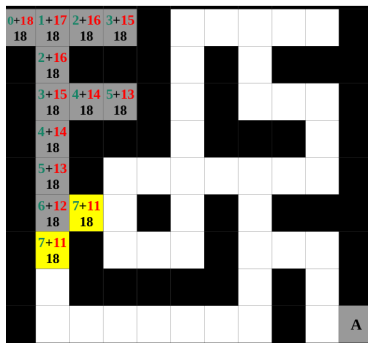


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



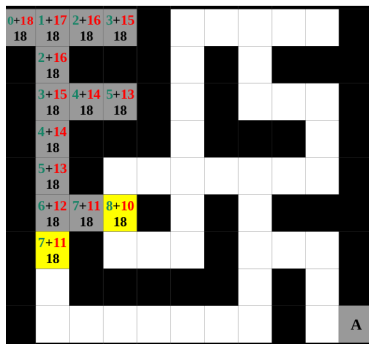
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.



## Exemple de résolution

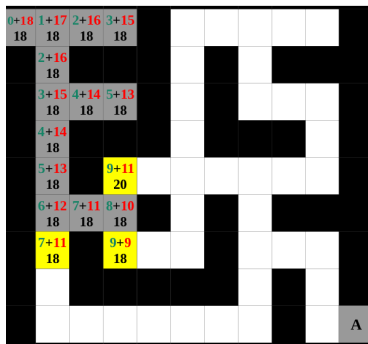


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

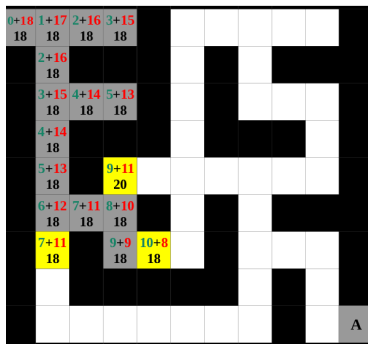


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

## Exemple de résolution



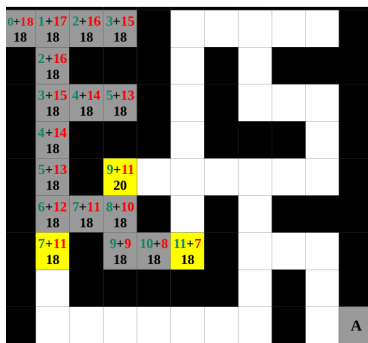
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



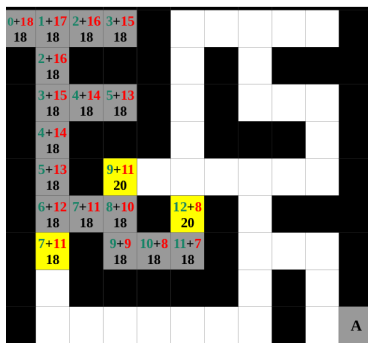
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



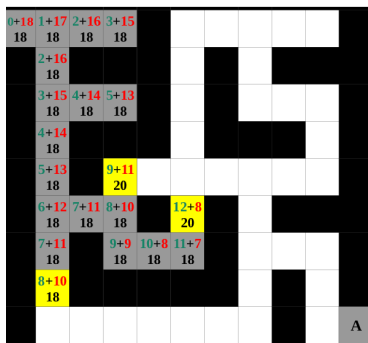
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



Légende :

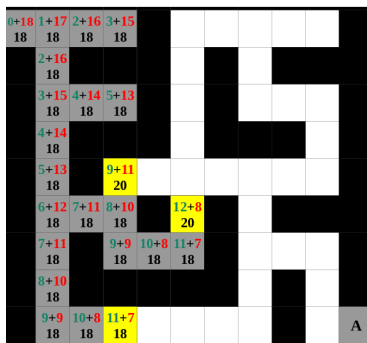
- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.



# Exemple de résolution



Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



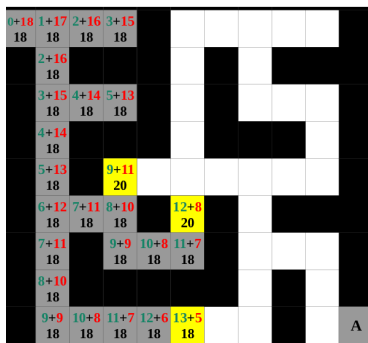
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



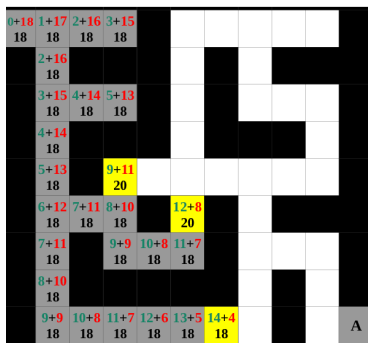
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



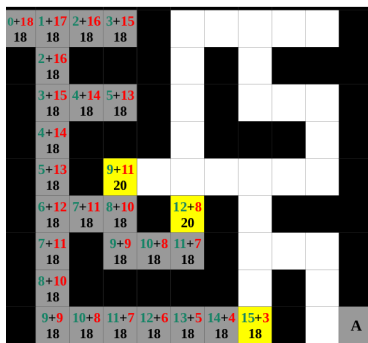
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



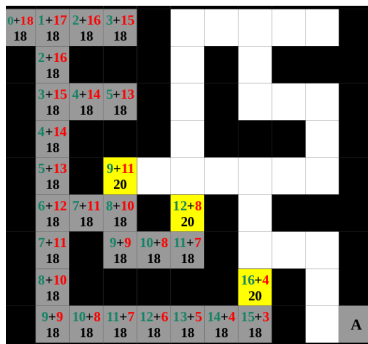
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution



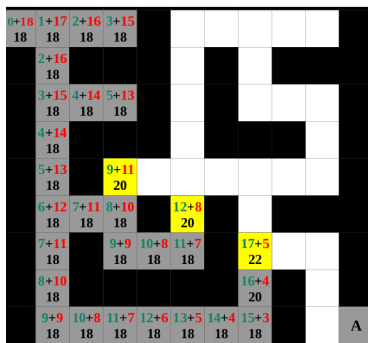
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois la priorité participe vraiment au choix.

# Exemple de résolution

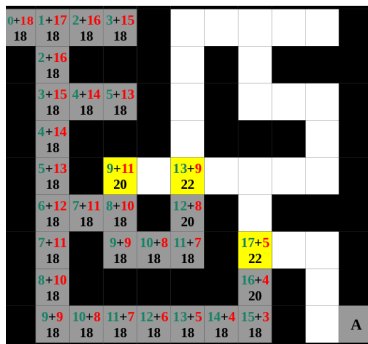


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



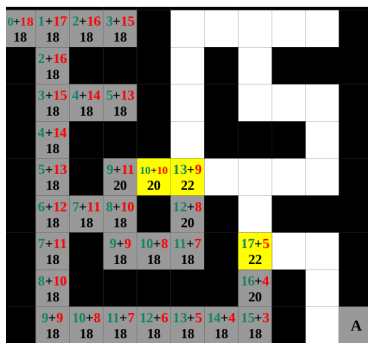
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.



# Exemple de résolution

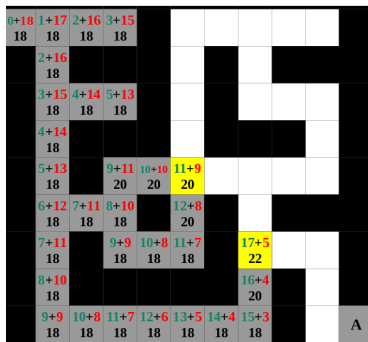


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

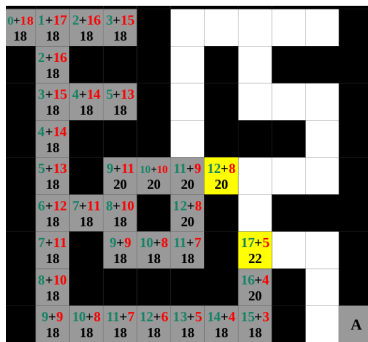


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



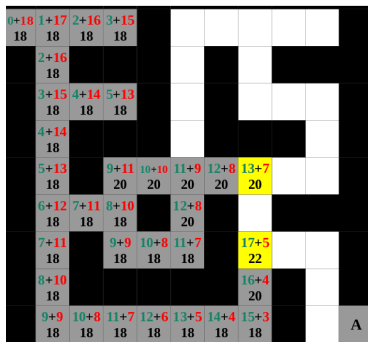
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois un nouveau chemin améliore l'accès à une même cellule.

# Exemple de résolution

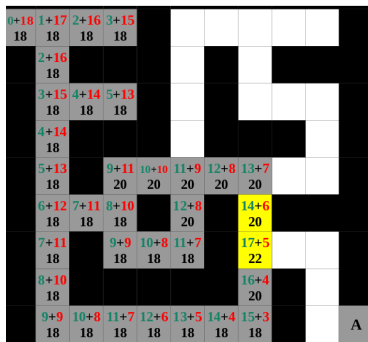


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

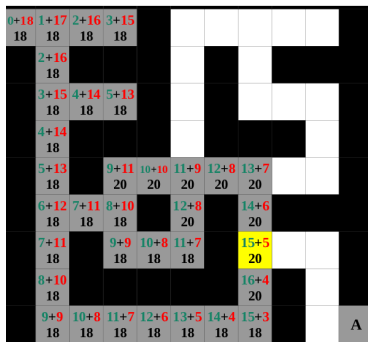


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

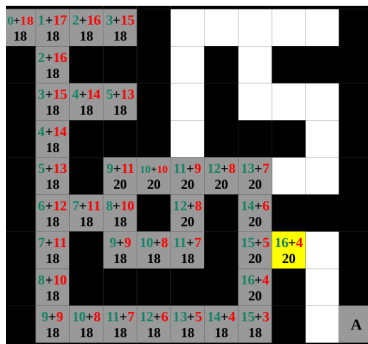


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



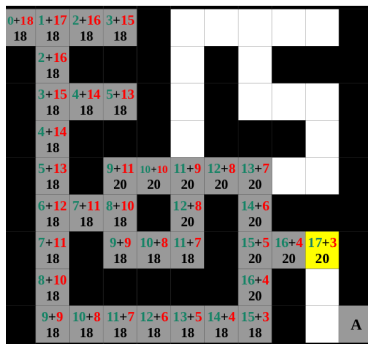
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

Parfois un nouveau chemin améliore l'accès à une même cellule.

## Exemple de résolution



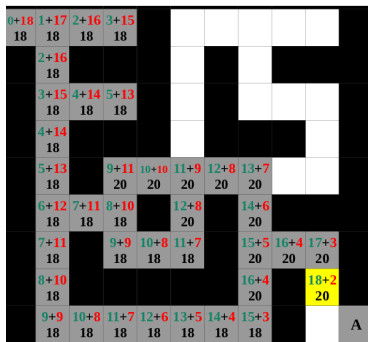
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.



# Exemple de résolution

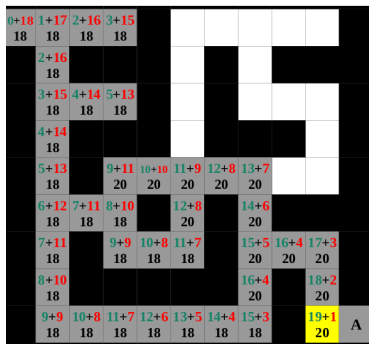


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

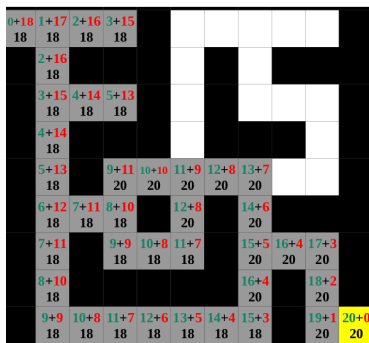


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution

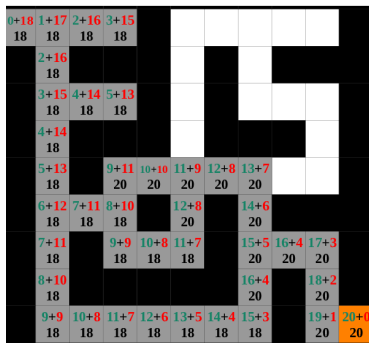


Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

# Exemple de résolution



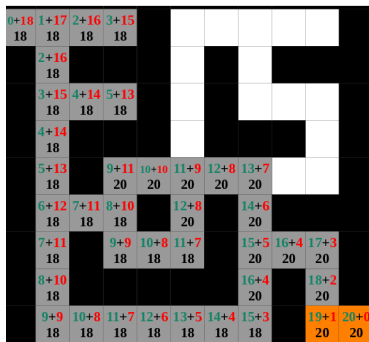
Légende :

- coût réel jusqu'ici
- coût heuristique pour la suite
- coût heuristique total

L'algorithme poursuit son chemin.

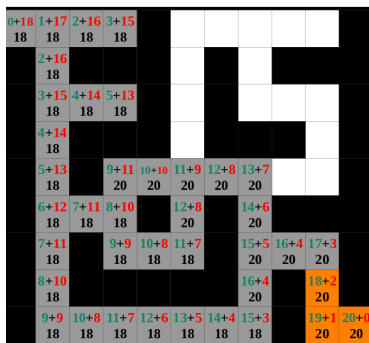
Un chemin vers la sortie a été trouvé!

# Exemple de résolution



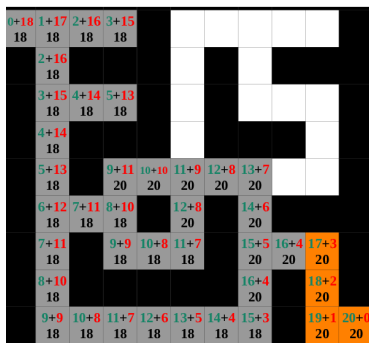
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



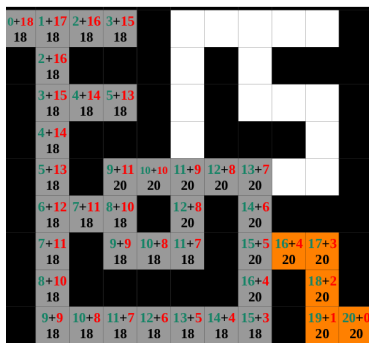
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

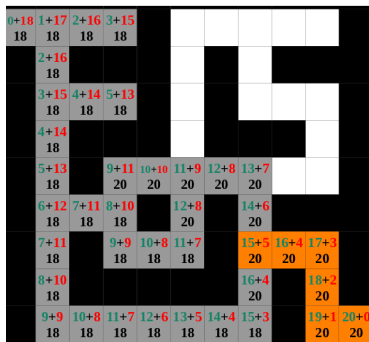
# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

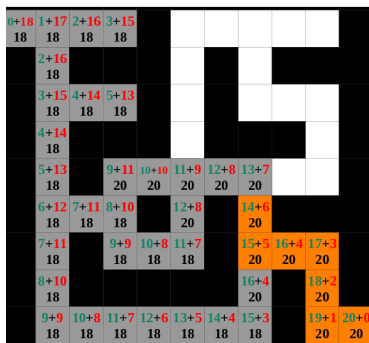


# Exemple de résolution



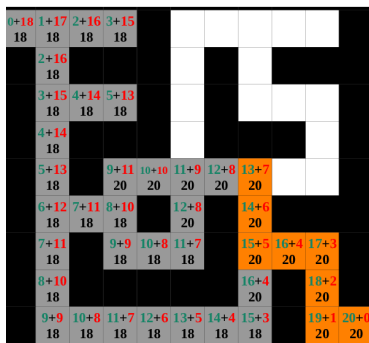
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



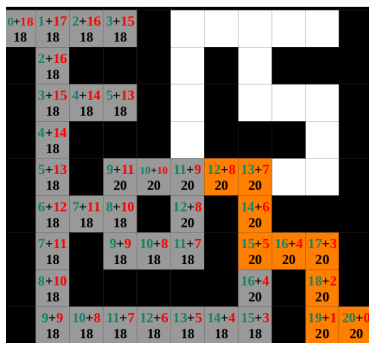
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



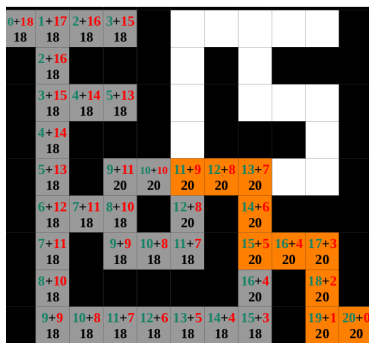
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



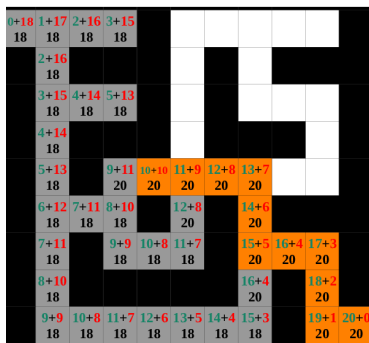
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



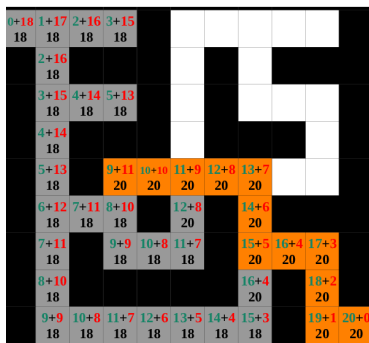
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



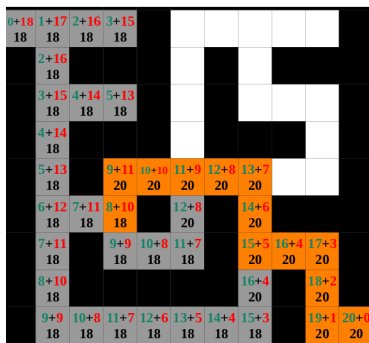
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

## Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

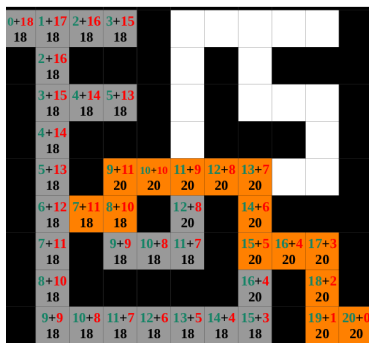
## Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

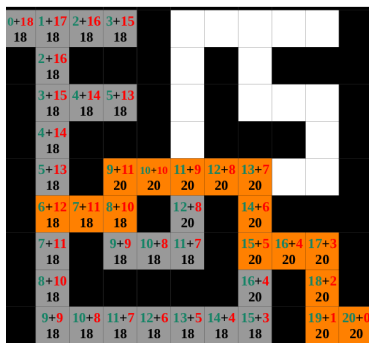


# Exemple de résolution



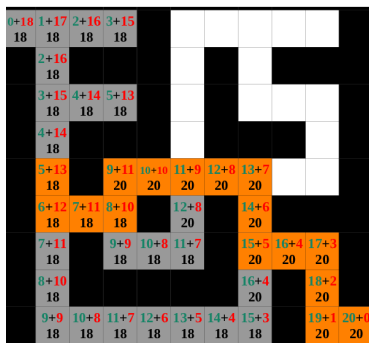
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



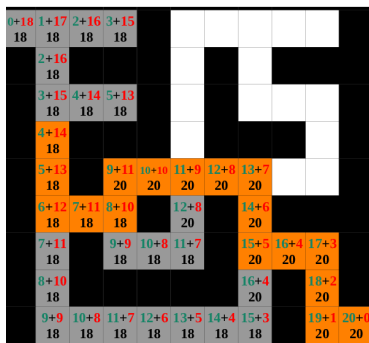
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



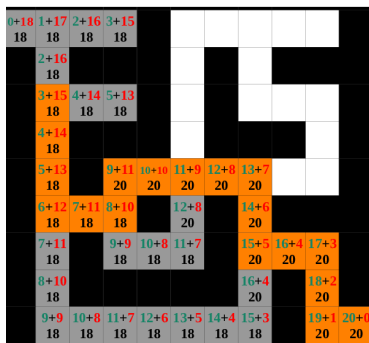
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



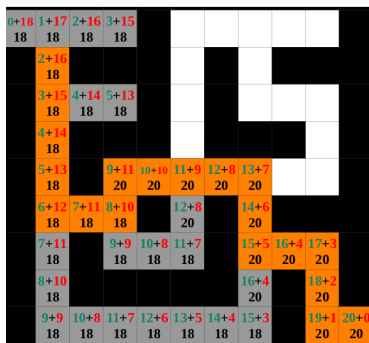
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



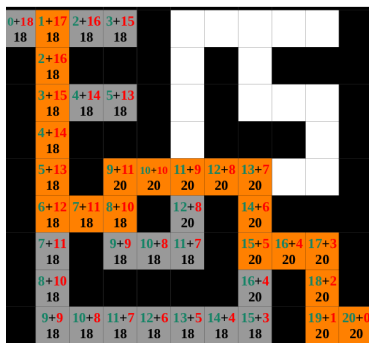
Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

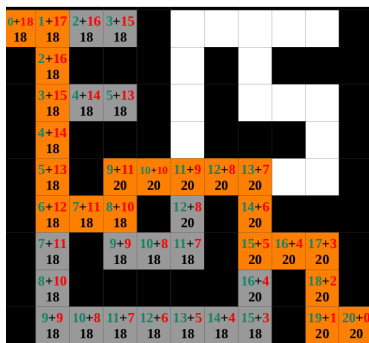


# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Exemple de résolution



Un chemin vers la sortie a été trouvé!  
Backtracking pour reconstituer le chemin

# Complexité de l'algorithme - 1

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").

# Complexité de l'algorithme - 1

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").

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.

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente
- Le nombre de **pop** et de **push** sera donc au maximum  $N$ .



## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente
- Le nombre de **pop** et de **push** sera donc au maximum  $N$ .
- Le nombre de **raffraichissements** sera au maximum de  $A$  où  $A$  = nombre de mouvements possibles.

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente
- Le nombre de **pop** et de **push** sera donc au maximum  $N$ .
- Le nombre de **raffraichissements** sera au maximum de  $A$  où  $A$  = nombre de mouvements possibles.
  - ▶  $O(A) \sim O(N)$  vu que chaque cellule possède au maximum 4 voisins atteignables.

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente
- Le nombre de **pop** et de **push** sera donc au maximum  $N$ .
- Le nombre de **raffraichissements** sera au maximum de  $A$  où  $A$  = nombre de mouvements possibles.
  - ▶  $O(A) \sim O(N)$  vu que chaque cellule possède au maximum 4 voisins atteignables.

### complexité algorithmique

$$N \cdot (\text{pop} + \text{push}) + A \cdot (\text{raffraichissement})$$

## Complexité de l'algorithme - 2

- “worst case” → visite toutes les  $N$  cellules.
  - ▶ **Pop** pour chaque cellule visitée, **push** de chaque voisin atteignable (si pas encore pushé).
  - ▶ Sinon, **raffraichissement** éventuel de la liste d'attente
- Le nombre de **pop** et de **push** sera donc au maximum  $N$ .
- Le nombre de **raffraichissements** sera au maximum de  $A$  où  $A$  = nombre de mouvements possibles.
  - ▶  $O(A) \sim O(N)$  vu que chaque cellule possède au maximum 4 voisins atteignables.

### complexité algorithmique

$$N \cdot (\text{pop} + \text{push}) + A \cdot (\text{raffraichissement})$$

L'implémentation de la liste d'attente est cruciale. La complexité de **push**, **pop** et **raffraichissement** dépendent en effet du type de file mise en oeuvre !

# Complexité - structure de données

complexité algorithmique

$$N \cdot (pop + push) + A \cdot (rafraichissement)$$

# Complexité - structure de données

## complexité algorithmique

$$N \cdot (\text{pop} + \text{push}) + A \cdot (\text{raffraichissement})$$

- simple tableau : **push** et **raffraichissement** en  $O(1)$ , **pop** en  $O(N)$

## complexité totale avec simple tableau

$$N \cdot (O(N) + O(1)) + A \cdot O(1) = O(N^2).$$

# Complexité - structure de données

## complexité algorithmique

$$N \cdot (\text{pop} + \text{push}) + A \cdot (\text{raffraichissement})$$

- simple tableau : **push** et **raffraichissement** en  $O(1)$ , **pop** en  $O(N)$

## complexité totale avec simple tableau

$$N \cdot (O(N) + O(1)) + A \cdot O(1) = O(N^2).$$

- queue prioritaire binaire : **push** et **raffraichissement** en  $O(\log(N))$ , **pop** en  $O(1)$ .

## complexité totale avec queue prioritaire binaire

$$N \cdot (O(1) + O(\log(N))) + A \cdot O(\log(N)) = O(N \cdot \log(N))$$

# 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
```

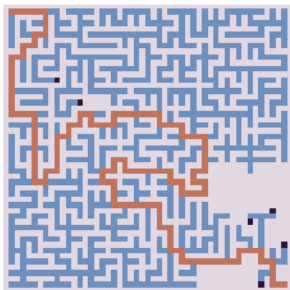
la suite dans : [https://github.com/Dalker/ASD\\_labyrinthe/](https://github.com/Dalker/ASD_labyrinthe/)

- 1 cloner le dossier implementation
- 2 exécuter visual\_test.py et time\_test.py

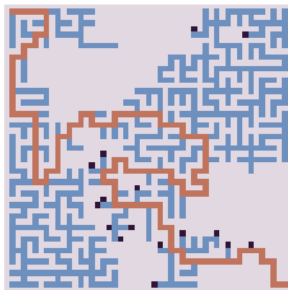


# Tests avec Python

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 - Autres "Path Finding" Ants

«ACO» : Ant Colony Optimisation

# Conclusion - Autres "Path Finding" Ants

«ACO» : Ant Colony Optimisation

- imite la nature avec dépôt de phéromones

# Conclusion - Autres "Path Finding" Ants

«ACO» : Ant Colony Optimisation

- imite la nature avec dépôt de phéromones
- évaporation des phéromones → disparition des chemins longs

# Conclusion - Autres "Path Finding" Ants

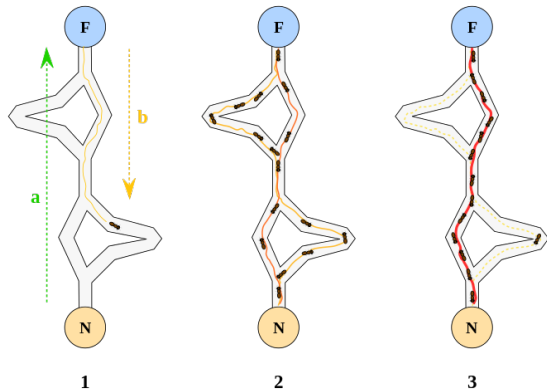
«ACO» : Ant Colony Optimisation

- imite la nature avec dépôt de phéromones
- évaporation des phéromones → disparition des chemins longs
- Avantage : adaptable si le labyrinthe évolue au cours du temps !

# Conclusion - Autres "Path Finding" Ants

## «ACO» : Ant Colony Optimisation

- imite la nature avec dépôt de phéromones
- évaporation des phéromones → disparition des chemins longs
- Avantage : adaptable si le labyrinthe évolue au cours du temps !
- Désavantage : nécessite un grand nombre d'individus



Par Johann Dréo  
(User :Nojhan) —  
Travail personnel, CC  
BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?rid=815564>

# Conclusion - Autres "Path Finding" Robots

Robots aspirateurs



# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)
- Transmission de gènes, via reproduction

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)
- Transmission de gènes, via reproduction
- Quelques mutations puis verdict du calcul d'adaptabilité

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)
- Transmission de gènes, via reproduction
- Quelques mutations puis verdict du calcul d'adaptabilité
- Les "individus" moins performants disparaissent

# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)
- Transmission de gènes, via reproduction
- Quelques mutations puis verdict du calcul d'adaptabilité
- Les "individus" moins performants disparaissent
- Avantage : Adaptable si le labyrinthe se modifie



# Conclusion - Autres "Path Finding" Robots

## Robots aspirateurs

- But un peu différent car ils doivent passer partout 1 seule fois
- Mais ils doivent aussi trouver leur chemin d'un point A vers un point B
- Ils utilisent souvent des algorithmes évolutionnistes
- Gènes = paramètres à optimiser (chemins)
- Transmission de gènes, via reproduction
- Quelques mutations puis verdict du calcul d'adaptabilité
- Les "individus" moins performants disparaissent
- Avantage : Adaptable si le labyrinthe se modifie
- Désavantage : Beaucoup d'individus et de générations pour parvenir à un bon résultat

# 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`