## 1. Relation entre les algorithmes de recherche (4 pts)

### 1.1 Démontrer les affirmations suivantes

- La recherche par respiration, la recherche en profondeur et la recherche à coût uniforme sont des cas particuliers de la recherche du meilleur en premier.

La recherche du meilleur en premier (Best-First Search) est une stratégie de recherche qui explore les nœuds en fonction d'une fonction d'évaluation. La recherche par respiration (Breadth-First Search, BFS) explore les nœuds par niveau de profondeur, la recherche en profondeur (Depth-First Search, DFS) explore les nœuds par profondeur, et la recherche à coût uniforme (Uniform-Cost Search, UCS) explore les nœuds par coût cumulé minimal. Chacune de ces recherches peut être vue comme une variante de la recherche du meilleur en premier en utilisant une fonction d'évaluation spécifique.

- Le coût uniforme est un cas particulier de A\.

L'algorithme A utilise une fonction d'évaluation $f(n) = g(n) + h(n)$, où $g(n)$ est le coût du chemin depuis le nœud initial jusqu'au nœud $n$, et $h(n)$ est une heuristique estimant le coût du chemin depuis $n$ jusqu'au but. Si $h(n) = 0$ pour tous les nœuds, alors $f(n) = g(n)$, ce qui correspond à la recherche à coût uniforme.

### 1.2 Propriété de l'heuristique

Prouver que si une heuristique est cohérente, alors elle doit être admissible.

Une heuristique $h(n)$ est cohérente si pour chaque nœud $n$ et chaque successeur $n'$ généré par une action $a$, $h(n) \leq c(n, a, n') + h(n')$, où $c(n, a, n')$ est le coût de l'action $a$ menant de $n$ à $n'$. Une heuristique est admissible si pour chaque nœud $n$, $h(n) \leq h^(n)$, où $h^(n)$ est le coût réel du chemin optimal de $n$ au but.

Pour prouver que la cohérence implique l'admissibilité, on peut utiliser l'induction sur la profondeur des nœuds. Si $h(n)$ est cohérente, alors pour le nœud initial $n_0$, $h(n_0) \leq h^(n_0)$. En supposant que $h(n) \leq h^(n)$ pour tous les nœuds à une certaine profondeur, on peut montrer que cela reste vrai pour les nœuds à la profondeur suivante, ce qui prouve l'admissibilité.

**1.3 A contre Dijkstra**

Expliquons si un algorithme est ou non un cas particulier de l'autre, justifions notre réponse.

L'algorithme A peut être vu comme une généralisation de l'algorithme de Dijkstra. L'algorithme de Dijkstra trouve le chemin de coût minimal dans un graphe pondéré sans heuristique, ce qui correspond à A avec \( h(n) = 0 \). Donc, Dijkstra est un cas particulier de A.

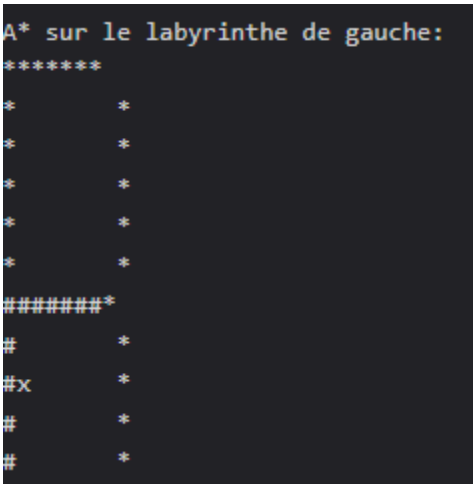
Donner une heuristique cohérente pour ce problème.

Pour le problème du labyrinthe, une heuristique cohérente pourrait être la distance de Manhattan (somme des différences absolues des coordonnées) entre la position actuelle et la position cible, en ignorant les murs.

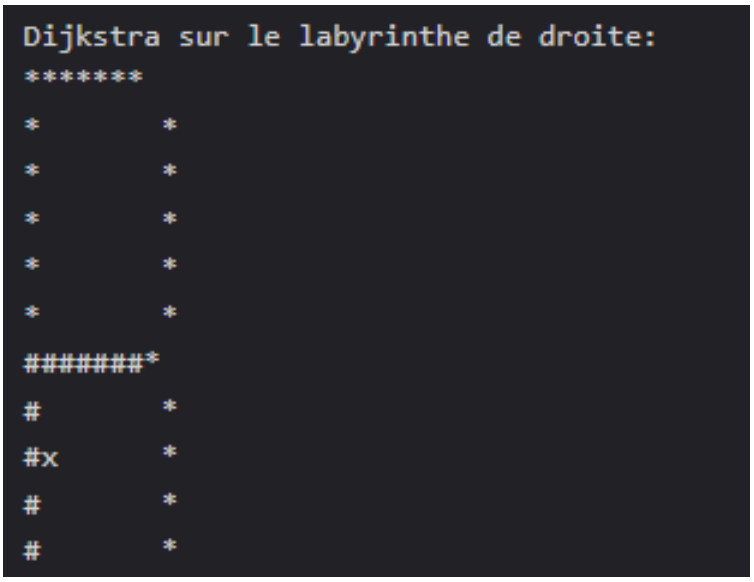
Afficher dans le labyrinthe de gauche les états (positions du plateau) qui sont inclus dans la frange lors d'une exécution de la recherche de graphe A avec une heuristique de distance de Manhattan (ignorant les murs).

Pour cela, nous devons simuler l'algorithme A en utilisant la distance de Manhattan comme heuristique et marquer les positions incluses dans la frange à chaque étape.

Montrons sur le labyrinthe de droite les positions du plateau visitées par l'algorithme de Dijkstra modifié.




Pour cela, nous devons simuler l'algorithme de Dijkstra modifié et marquer les positions visitées à chaque étape.




**Explication de notre représentation Python des états.**

On peut représenter l'état du jeu Sokoban comme une matrice 2D où chaque élément représente une position du labyrinthe. Par exemple, utiliser une liste de listes en Python.

Formulons ce problème comme un problème de recherche (notre fonction successeur doit être décrite très précisément). Quel est le facteur de ramification de notre modèle ?

Le problème de Sokoban peut être formulé comme un problème de recherche où l'état initial est la configuration initiale du labyrinthe, et l'état but est la configuration où toutes les boîtes sont sur les positions cibles. La fonction successeur génère les états accessibles en déplaçant l'avatar dans les 4 directions possibles. Le facteur de ramification est de 4, car il y a 4 directions possibles de déplacement.

Quelle est la complexité de notre fonction de successeur ? Combien de successeurs y a-t-il au maximum ? Comment est-ce lié au facteur de ramification ?

La complexité de la fonction de successeur est \( O(1) \) car elle génère un nombre constant de successeurs (4 au maximum). Le nombre maximum de successeurs est lié au facteur de ramification, qui est de 4.

Existe-t-il d'autres situations similaires ? Pourquoi est-il important d'identifier ces situations dans notre fonction de succession ?

Oui, il existe d'autres situations sans issue, comme les configurations où une boîte est coincée contre un mur sans possibilité de la déplacer. Il est important d'identifier ces situations pour éviter d'explorer des chemins sans issue, ce qui réduit le temps de recherche.

Décrire les heuristiques possibles (non triviales) pour atteindre un état objectif (avec référence le cas échéant). Notre(nos) heuristique(s) est(sont) admissible(s) et/ou cohérente(s) ? Donnons la complexité ?

Une heuristique possible est la somme des distances de Manhattan entre chaque boîte et sa position cible la plus proche. Cette heuristique est admissible car elle ne surestime jamais le coût réel. La complexité de cette heuristique est $O(n)$, où $n$ est le nombre de boîtes.

Donnons une limite supérieure au nombre d'états.

La limite supérieure au nombre d'états est $4^{mn}$, où $m$ et $n$ sont les dimensions du labyrinthe, car chaque position peut être dans 4 états différents (vide, mur, boîte, cible).