

# Sokoban : explorer un graphe avec $A^*$

INFORMATIQUE COMMUNE - TP n° 3.8 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ☞ utiliser  $A^*$  pour explorer le graphe d'un jeu avec une heuristique
- ☞ imaginer et coder des heuristiques admissibles pour  $A^*$

## A Sokoban

Sokoban est un jeu de réflexion dans lequel le joueur doit **déplacer des caisses en les poussant** sur un plateau **pour les placer sur des cibles**, des positions fixes spécifiques. La partie est gagnée lorsque toutes les caisses ont été déplacées sur les cibles.

- Le joueur se déplace sur le plateau et peut pousser les caisses, **mais pas les tirer**. Le joueur doit donc être stratégique dans ses déplacements pour éviter de coincer des caisses contre les murs. Il peut pousser une caisse dans les directions horizontale et verticale, **mais il ne peut pousser qu'une seule caisse à la fois**.
- Le plateau est bordé de **murs** qui empêchent le joueur de se déplacer et d'accéder à certaines zones. Des murs intérieurs peuvent également limiter les zones de déplacement des caisses.
- Certaines cases du plateau sont marquées comme des **cibles**.
- Les caisses doivent être placées sur ces cases pour gagner la partie.

Un exemple de configuration de départ du jeu est donné sur la figure 1.

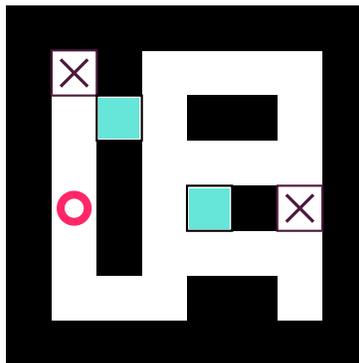


FIGURE 1 – Exemple de configuration de départ du jeu Sokoban. Le joueur est un cercle rouge, les cibles des croix et les caisses des rectangles en cyan.

## B Modélisation

Le plateau du jeu Sokoban de la figure 1 est modélisé par une carte sous la forme d'un dictionnaire comme décrit ci-dessous.

```
def init_level():
    taille = (8, 8)
    murs = {(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7),
            (1, 0), (1,2), (1, 7),
            (2, 0), (2, 4), (2, 5), (2, 7),
            (3, 0), (3,2) ,(3, 7),
            (4, 0), (4, 2), (4, 5), (4, 7),
            (5, 0), (5, 2), (5, 7),
            (6, 0), (6,4), (6,5), (6, 7),
            (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7)}
    cibles = {(1, 1), (3, 6)}
    caisses = {(2, 2), (4, 4)}
    joueur = (3, 1)
    return {"taille": taille, "murs": murs, "cibles": cibles, "joueur": joueur, "
           caisses": caisses}
```

murs, caisses et cibles sont des `set` Python, c'est-à-dire des ensembles d'éléments. Cette structure proche du dictionnaire au niveau de la syntaxe est hors programme, mais permet de traiter rapidement l'appartenance d'un élément à un ensemble, opération répétée de nombreuses fois lors des algorithmes que nous allons mettre en place.

Pour afficher le plateau de Sokoban, on utilise la fonction suivante :

```
def affiche(carte):
    nl, nc = carte["taille"]
    print(' ', end='')
    for i in range(nl):
        print(i, end='')
    for i in range(nc):
        print("\n" + str(i), end=' ')
    for j in range(nl):
        if (i, j) in carte["murs"]:
            if i==0 and j==0:
                print('┌', end='')
            elif i==0 and j== nc-1:
                print('┐', end='')
            elif i==nl -1 and j== 0:
                print('└', end='')
            elif i==nl-1 and j== nc-1:
                print('┘', end='')
            elif i==0 and (i+1,j) in carte["murs"]:
                print('┴', end='')
            elif i == nl-1 and (i-1,j) in carte["murs"]:
                print('┬', end='')
            elif j==0 and (i,j+1) in carte["murs"]:
                print('├', end='')
            elif j == nc-1 and (i,j-1) in carte["murs"]:
                print('┤', end='')
            elif j == 0 or j == nc -1:
                print('│', end='')
            elif i==0 or i == nl-1:
                print('─', end='')
```

```

        print('-', end='')
    elif j == 0 or j == nc -1:
        print('|', end='')
    else:
        print('\u2588', end='')
    elif (i, j) in carte["cibles"]:
        print('\033[31m\u272A\033[0m', end='')
    elif (i, j) in carte["caisses"]:
        print('\033[34m\u25C8\033[0m', end='')
    elif (i, j) == carte["joueur"]:
        print('\033[32m\u25C9\033[0m', end='')
    else:
        print(' ', end='')

print()

carte = init_level()
affiche(carte)

```

- B1. Écrire une fonction de signature `mvt_valide(carte: dict, pos: tuple[int, int]) -> bool` qui renvoie `True` si le déplacement du joueur est valide, c'est-à-dire si `pos` n'est pas un mur ou si `pos` ne sort pas du cadre du jeu. Cette fonction renvoie `False` dans le cas contraire.

On se donne les directions des déplacements possibles du joueur sous la forme suivante :

```

NORD = (-1, 0)
SUD = (1, 0)
EST = (0, 1)
OUEST = (0, -1)
DIRECTIONS = [NORD, SUD, EST, OUEST]

```

Une direction est donc un tuple.

- B2. Écrire une fonction de signature `pos_suivante(carte, direction) -> dict` qui renvoie un dictionnaire vide si le joueur ne peut pas jouer dans le sens de `direction` et une nouvelle carte sinon. Cette nouvelle carte comportera la nouvelle position du joueur ainsi que les nouvelles positions des caisses éventuellement déplacées par son mouvement. On pourra utiliser `deepcopy` pour copier `carte` en profondeur. On pourra utiliser les méthodes suivantes pour ajouter ou retirer d'un ensemble :

```

nouvelle_carte["caisses"].remove((posp))
nouvelle_carte["caisses"].add(posb)

```

- B3. Écrire une fonction de signature `resolu(carte) -> bool` qui renvoie `True` si le jeu a été résolu et `False` sinon.
- B4. Écrire une fonction de signature `distance_manhattan(pos1, pos2)` qui calcule la distance de Manhattan de deux positions du Sokoban.

## C Exploration du graphe avec $A^*$

On souhaite utiliser l'algorithme  $A^*$  pour résoudre le Sokoban. Dans cet objectif, on construit au fur et à mesure le graphe du jeu (cf. figure 2). Chaque sommet correspond à une position des caisses et du joueur sur le plateau de jeu.

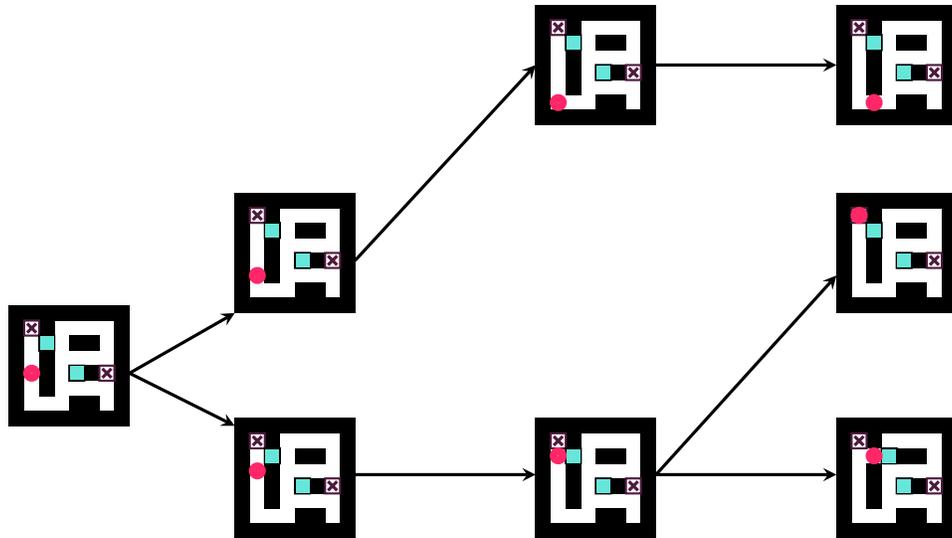


FIGURE 2 – Une partie du graphe de jeu d’après le point de départ de la figure 1 à gauche

- C5. Proposer une fonction de signature `hash_carte(carte)` qui renvoie un clef permettant d’identifier un sommet du graphe de jeu. Cette clef est utilisable comme clef d’un dictionnaire Python.
- C6. Écrire une fonction de signature `heuristique(carte) -> int` qui renvoie une estimation de la distance à parcourir de `carte` jusqu’à la condition de gain du jeu. Le score généré sera la somme :

- de la distance minimale des caisses aux cibles,
- de la distance minimale du joueur aux cibles,

On justifiera brièvement le caractère admissible de cette heuristique.

- C7. Écrire une fonction de signature `astar(carte)` qui implémente l’algorithme  $A^*$  et trouve la succession de déplacements nécessaires pour résoudre le Sokoban. Cet algorithme renvoie :
- None si aucune solution n’a été trouvée.
  - `parents`, `deja_vus`, `carte` si une solution a été trouvée, où
    - `parents` est un dictionnaire qui recense le parent d’un sommet sur le chemin emprunté, c’est-à-dire le sommet par lequel il a été découvert,
    - `deja_vus` est un dictionnaire des sommets déjà empruntés,
    - `carte` est la carte finale du Sokoban résolu.

Proposer une assertion qui vérifie si l’heuristique est monotone. Cela permet de s’assurer que le chemin est trouvé de manière optimale.

- C8. Écrire une fonction de signature `jouer()` qui affiche les étapes de la solution trouvée par l’algorithme `astar`. La tester sur plusieurs configurations de départ.