Sokoban : exploration et heuristiques

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 adminissibles 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. Le 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 daccéder 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.

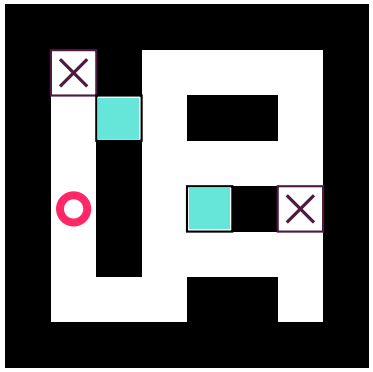


FIGURE 1 – Exemple de niveau du jeu Sokoban. Le joueur est un cercle rouge, les cibles des croix et les caisses des rectangles en cyan.

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}
```

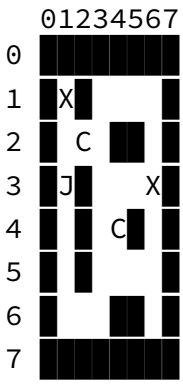
`murs`, `caisses` et `cibles` sont des `set` Python, c'est-à-dire des ensembles d'éléments. Cette structure proche du dictionnaire au niveau de la syntaxe est hors programme, mais permet de traiter rapidement l'appartenance d'un élément à un ensemble, opération répétée de nombreuses fois lors des algorithmes 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"]:
                print('\u2588', end='')
            elif (i, j) in carte["cibles"]:
                print('X', end='')
            elif (i, j) in carte["caisses"]:
                print('C', end='')
            elif (i, j) == carte["joueur"]:
                print('J', end='')
            else:
                print(' ', end='')
    print()

carte = init_level()
affiche(carte)
```

Ces instructions produisent normalement la figure suivante :



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.

Solution :

```
def mvt_valide(carte, pos):
    nl, nc = carte["taille"]
    ligne, col = pos
    return 0 < ligne < nl and 0 < col < nc and (ligne, col) not in carte["
        murs"]
```

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)
```

Solution :

```
def pos_suivante(carte, direction):
    dl, dc = direction
```

```
        lj, cj = carte["joueur"]
        nposj = lj + dl, cj + dc
        nouvelle_carte = deepcopy(carte)  # copie en profondeur de la carte
        # Coup possible ?
        if not mvt_valide(carte, nposj):
            return {}
        # Cas où il y a une caisse
        if nposj in carte["caisses"]:
            nposb = lj + 2 * dl, cj + 2 * dc  # position suivante de la boite
            # Coup possible ?
            if not mvt_valide(carte, nposb):
                return {}
            # Déplacement : case de la boite
            nouvelle_carte["caisses"].remove((nposj))
            nouvelle_carte["caisses"].add(nposb)
        # Déplacement : case du joueur
        nouvelle_carte["joueur"] = nposj
        return nouvelle_carte
```

B3. Écrire une fonction de signature `resolu(carte)-> bool` qui renvoie `True` si le jeu a été résolu et `False` sinon.

Solution :

```
def resolu(carte):
    for boite in carte["caisses"]:
        if boite not in carte["cibles"]:
            return False
    return True
```

B4. Écrire une fonction de signature `distance_manhattan(pos1, pos2)` qui calule la distande de Manhattan de deux positions du Sokoban.

Solution :

```
def distance_manhattan(pos1, pos2):
    return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])
```

B5. Écrire une fonction de signature `heuristique(carte)-> float` qui renvoie un score décrivant la carte par rapport à l'objectif du jeu. Le score généré sera la somme :

- de la distance minimale des caisses aux cibles,
- de la distance minimale du joueur aux cibles,
- du nombre de directions selon lesquelles les caisses sont bloquées dans leur déplacement par des murs ou d'autres caisses.

On pourra pondérer ces différents éléments et on justifiera brièvement le caractère admissible de cette heuristique.

Solution : Les distances de Manhattan ne tiennent pas compte des murs ou des caisses. C'est pourquoi cette heuristique ne surestime jamais la distance aux cibles.

```
def heuristique(carte):
    score = 0
    # Trouve toutes les caisses et cibles
    for bpos in carte["caisses"]:
        dmin = math.inf
        for cpos in carte["cibles"]:
            d = distance_manhattan(bpos, cpos)
            if dmin > d:
                dmin = d
        score += d
    score *= 2.5
    dmin = math.inf
    for cpos in carte["cibles"]:
        d = distance_manhattan(carte["joueur"], cpos)
        if dmin > d:
            dmin = d
    score += dmin

    penalite = 0
    for boite in carte["caisses"]:
        if boite not in carte["cibles"]:
            # Vérifie les directions de poussée possibles
            bloquees = 0
            for dligne, dcol in DIRECTIONS:
                pos_suiv = (boite[0] + dligne, boite[1] + dcol)
                # Pénalise si la direction de poussée ou la position
                    derrière est bloquée
                if not mvt_valide(carte, pos_suiv) or pos_suiv in carte["
                    caisses"]:
                    bloquees += 1
            # Pénalise fortement si la boîte est presque impossible à
                déplacer
            penalite += bloquees * 2
    return score
```

B6. 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.
- `sommets, parents, deja_vus, carte` si une solution a été trouvée.
  - `sommets` est la liste des tuples `(cartes, direction)` qui documente les sommets explorés au cours de l'algorithme.
  - `parents` est un dictionnaire qui recense les parents d'un sommet sur le chemin emprunté.
  - `deja_vus` est un dictionnaire des sommets déjà empruntés.
  - `carte` est la carte finale du Sokoban résolu.

Une fonction pour créer une empreinte (hash) d'un Sokoban est nécessaire pour s'assurer facilement qu'on a déjà rencontré la même carte. On pourra utiliser :

```
def hash_carte(carte):
```

```
    return (tuple(carte["caisses"]), tuple(carte["joueur"]))
```

Solution :

```
def astar(carte):
    deja_vus = {}
    h0 = heuristique(carte)
    clef = hash_carte(carte)
    pq = queue.PriorityQueue()
    pq.put((h0, clef, carte, None))
    sommets = []
    parents = {clef: clef}
    while not pq.empty():
        d, key, carte, dir = pq.get()  # transfert
        sommets.append((carte, dir))
        # print(carte, dir)
        # affiche(carte)
        if resolu(carte):  # early exit !
            return sommets, parents, deja_vus, carte
        for dir in DIRECTIONS:
            suivante = pos_suivante(carte, dir)
            if suivante != {}:
                clef = hash_carte(suivante)
                if not clef in deja_vus:
                    deja_vus[clef] = suivante
                    hs = heuristique(suivante)
                    parents[clef] = key
                    pq.put((d + hs, clef, suivante, dir))
    return None
```

B7. Écrire une fonction de signature `jouer()` qui joue selon la solution trouvée par l'algorithme `astart`.

Solution :

```
def jouer():
    carte = init_level_2()
    try:
        print("Début !")
        s, p, d, c = astar(carte)
        solutions = [c]
        start = hash_carte(c)
        seen = set()
        while p[start] != start and start not in seen:
            seen.add(start)
            solutions.append(d[start])
            start = p[start]
        solutions.reverse()
        for c in solutions:
            affiche(c)
    except Exception as e:
        print(e,"Pas de solutions")
```