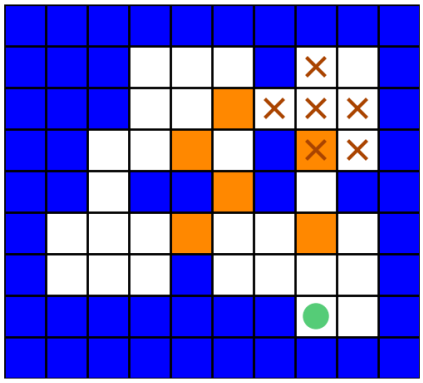
Résolution de Sokoban

Mickaël Péchaud (mickaelpechaud@protonmail.com)

Février 2021



Un niveau de *Sokoban*

- Documentez vos fonctions.
- Testez au fur et à mesure les fonctions que vous écrivez !

Vous travaillerez dans le fichier *sokoban.py* fourni.

Règles

Sokoban est un jeu video de réflexion historique, dans lequel un personnage doit pousser des caisses afin de les placer dans certaines positions cibles. On parlera de *position gagnante* lorsque toutes les caisses seront correctement placées.

Plus précisément, un *niveau* est la donnée :

- d'une *carte* de départ, i.e. d'un tableau à deux dimensions, où chaque case peut
  - être vide (V)
  - contenir un mur (M)
  - contenir le personnage (P)
  - ou contenir une caisse (C)
- et d'une liste de cases correspondant aux *cibles*. Les cibles sont en même nombre (non-nul) que les caisses, et ne peuvent pas être situées sur une case contenant un mur.

Le personnage peut se déplacer horizontalement ou verticalement vers toute case vide. Il peut également pousser une caisse, si la case située derrière celle-ci est vide. Il est interdit de pousser deux caisses simultanément.

Les cases des quatre bords de la carte devront nécessairement contenir des murs.

Codage

- Une carte est codée comme une liste de listes de V, M, C ou P. On adopte la même convention d'indiçage que pour les matrices : si `c` est une carte, `c[0][0]` désigne le contenu de la case située en haut à gauche, `c[1][0]` le contenu de la case située immédiatement en dessous, etc.
- Les cibles sont codées comme une liste de *coordonnées*, chaque coordonnée étant un couple d'entiers : un numéro de ligne, et un numéro de colonne.

Le fichier *sokoban.py* contient trois exemples de niveaux, que vous pourrez utiliser pour tester vos fonctions.

Vous disposez également de fonctions `affiche_carte` et `affiche_carte_leger`, permettant d'afficher des cartes de façon un peu plus agréable que sous forme de liste de listes.

Le but de ce TP est d'écrire un programme permettant de résoudre un niveau de *Sokoban* - de préférence en un nombre minimal de déplacements de caisses.

Sauf mention contraire, les fonctions que vous écrirez fonctionneront *en place*, c'est-à-dire modifieront la carte qui leur est passée en argument plutôt que de créer une nouvelle carte.

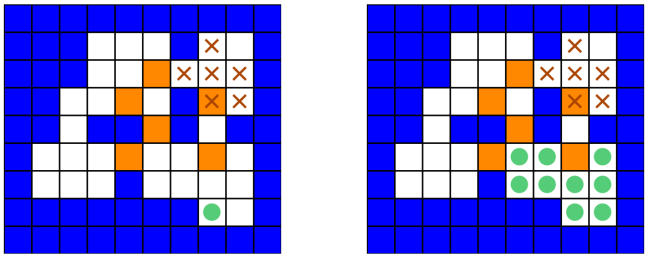
Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".

Premières fonctions

1. Compléter la fonction `dim`, qui prend en argument une carte et renvoie le couple d'entiers (w, h), w étant la largeur de l'image, et h sa hauteur (en nombre de cases).
   On pourra ainsi dans la suite utiliser l'instruction `(w, h) = dim(c)` pour récupérer les dimensions d'une carte `c`.
2. Compléter la fonction `succes`, qui prend en argument une carte et un ensemble de cibles, et renvoie vrai si et seulement si chaque caisse est sur une position cible.
   On garantira une complexité linéaire en le nombre de caisses.

Dans la suite, nous allons nous intéresser uniquement aux mouvements des caisses, et donc considérer que le personnage peut se déplacer instantanément dans toute zone de cases vides contigües.
Nous allons pour cela «cloner» le personnage dans toute la zone de cases vides contigües à laquelle il appartient.

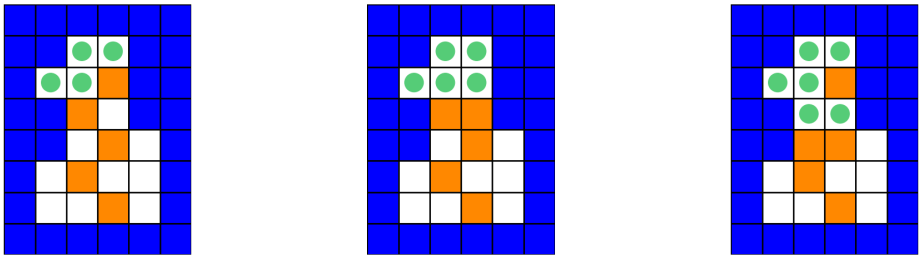
3. Compléter la fonction `cases_libres_voisines`, qui prend en arguments une carte et des entiers i, j correspondant aux coordonnées d'une case (qui n'est pas au bord de la carte), et renvoie la liste des coordonnées des cases vides voisines de (i, j).
4. Compléter la fonction `clone_personnage`, qui prend en arguments une carte et des entiers i, j correspondant aux coordonnées de la occupée par le personnage, et mets la valeur P dans toutes les cases vides accessibles sans bouger de caisse.
   On utilisera un algorithme de parcours itératif.
   Discuter l'importance du choix de cet algorithme.
5. Implémenter une version récursive `clone_personnage_recursif` effectuant la même chose que la fonction précédente. À quel type de parcours cela correspond-il ?



Une carte avant et après le clonage de personnage

Détermination des cartes accessibles

Nous allons maintenant écrire une série de fonctions permettant de déterminer les cartes accessibles à partir d'une carte donnée en poussant une caisse.



Une position, et les deux positions accessibles en poussant une caisse.

6. Compléter la fonction `coordonnees_caisses`, qui prend en argument une carte, et renvoie la liste des coordonnées des caisses y apparaissant.
7. Lire, expliquer rapidement et compléter la fonction `poussable`, qui prend en argument une carte `c`, des coordonnées `i` et `j` et une direction (`HAUT`, `BAS`, `GAUCHE` ou `DROITE`), et qui détermine s'il est possible de pousser la caisse en position (i, j) dans la direction indiquée. On supposera qu'une caisse est bien présente dans la carte à cette position, et que le personnage a été cloné dans la carte.
8. Compléter la fonction `supprime_tous_personnages`, qui prend en argument une carte, et remplace tous ses personnages par des cases vides.
9. Compléter la fonction `pousser`, qui prend en argument une carte `c`, des coordonnées `i` et `j` et une direction, et qui **renvoie une nouvelle carte** obtenue en poussant la caisse en position (i, j) dans la direction indiquée.

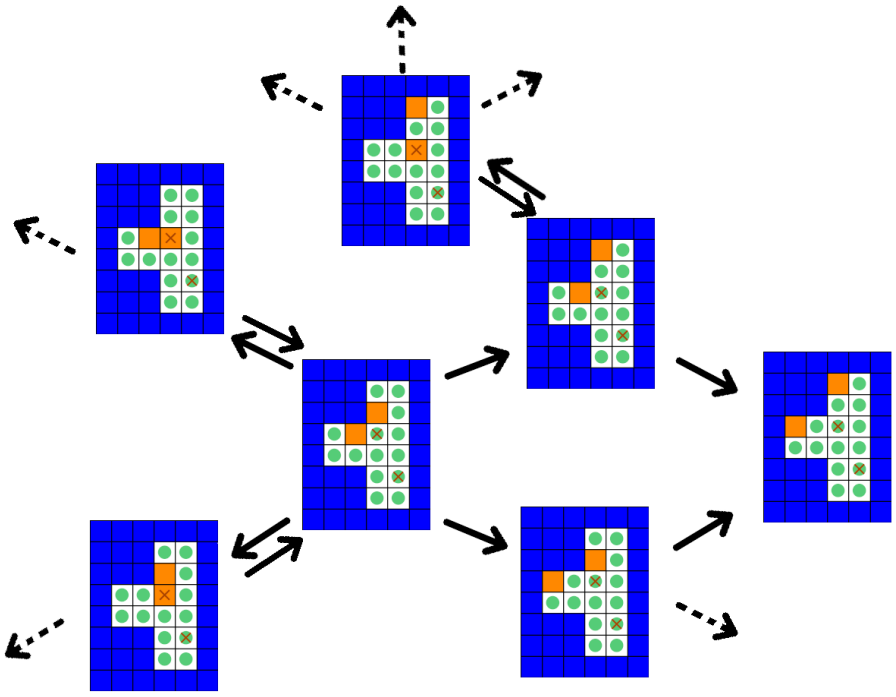
Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".

On supposera qu'il y a effectivement une caisse en position (i, j), et que celle-ci est poussable.
La carte renvoyée ne contiendra qu'un personnage en position (i, j).

10. Compléter enfin la fonction `cartes_accessibles`, qui prend en argument une carte où le personnage a été cloné, et renvoie la liste des cartes accessibles en poussant une caisse (où le personnage a été également cloné).

Résolution

Nous allons maintenant coder un algorithme de résolution complet.
Pour ce faire, nous allons voir chaque carte comme un sommet d'un graphe orienté. Deux sommets sont reliés par un arc si et seulement si il est possible de passer d'une carte à l'autre en poussant une caisse.
Nous allons parcourir ce graphe des cartes jusqu'à ce qu'à atteindre la position gagnante.



Début du graphe des cartes correspondant au premier niveau.

Pour cela, nous allons maintenir deux structures de données.

L'ensemble des cartes actives, implémenté par une *liste* de sommets `actifs`. Les nouvelles cartes seront ajoutées en fin de liste, et la prochaine carte à explorer sera la dernière de la liste. La liste est donc utilisée pour implémenter une structure de *pile* des cartes actives – ou *LIFO* : **L**ast **I**n **F**irst **O**ut ;

L'ensemble des cartes déjà explorées – i.e. dans `actifs`, ou qui ont été dans `actifs` dans le passé – implémenté par un *dictionnaire* `dejavu`. Les clés seront des codes de cartes, obtenus à l'aide de la fonction `code` fournie [1]. Les valeurs seront dans l'immédiat mises à `None` [2].

On rappelle sous forme de pseudocode le principe général d'un algorithme de parcours :

1. Il est impossible d'utiliser directement une carte comme clé d'un dictionnaire, car il s'agit d'un objet *mutable*.
2. On pourrait utiliser directement le type `set` de *Python*. Nous aurons cependant besoin d'avoir un dictionnaire dans les questions suivantes.

Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".

Pseudocode : Algorithme de parcours

- Ajouter la carte de départ à `actifs` et `dejavu`.
- Tant que `actifs` est non-vide :
  - extraire une carte `c` de `actifs`
  - pour chaque carte accessible à partir de `c` qui n'est pas dans `dejavu`, rajouter cette carte à `actifs` et `dejavu`.

11. À quel type de parcours de graphe correspond l'algorithme décrit ci-dessus ?

12. Compléter la fonction `nombre_positions`, qui prend en argument une carte, parcourt le graphe des positions de la façon décrite ci-dessus et renvoie le nombre total de positions accessibles. Vérifier que pour les cartes 1, 2 et 3, ces nombres sont respectivement 122, 348 et 6476 (en comptant la position de départ).

13. Copier coller votre code dans la fonction `resoluble`, puis le modifier afin que `resoluble` renvoie *vrai* si et seulement si le niveau passé en argument a une solution. La fonction *affichera* de plus le nombre d'itérations effectuées.

On souhaite maintenant calculer un *chemin* résolvant le niveau, c'est-à-dire une liste de cartes allant de la position de départ à la position gagnante. Pour ce faire, à chaque carte `c` stockée dans `dejavu`, on associera comme valeur la carte «parente» de `c`, c'est-à-dire celle à partir de laquelle `c` a été obtenue.
Une fois la position gagnante obtenue, il suffira donc de remonter successivement les «parentes» jusqu'à la position initiale.

14. Compléter la fonction `chemin` correspondante.

Plus courts chemins

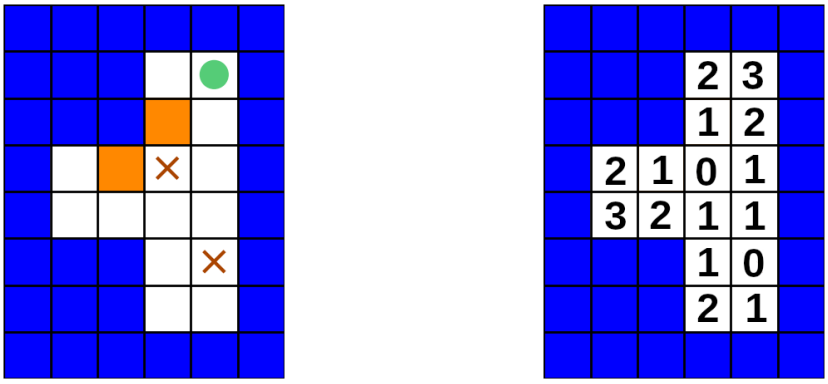
On souhaite maintenant obtenir un plus court chemin. Pour cela, nous allons simplement remplacer la structure de *pile* d'`actifs`, où le sommet traité est le dernier entré, par une structure de *file*, où le sommet traité est le premier entrée.
Pour cela, nous allons utiliser le type `collections.deque`, dont les méthodes utiles sont données en annexe. Vous en trouverez également un exemple d'utilisation dans le fichier.

15. Quel est le type de parcours ainsi implémenté ?

16. Compléter la fonction `plus_court_chemin` correspondante.

Exploration guidée

Nous allons maintenant accélérer la vitesse de résolution en utilisant l'algorithme $A^\star$.
La première étape va être de calculer la carte des distances de chaque case à la cible la plus proche.



Un niveau, et la carte des distances correspondante.

Pour cela, on utilise un parcours en largeur des cases.

Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".

Pseudocode : calcul de la distance à la plus proche cible

- Créer une carte des distances initiale, où les distances sont initialisées à une valeur infinie.
- Initialiser les distances des cases correspondant aux cibles à 0. Ajouter ces cases à une liste `actifs`.
- Tant que `actifs` est non-vide :
  - extraire une case `c` de `actifs` de distance minimale
  - pour chaque case (ne contenant pas un mur) voisine de `c` ayant une distance infinie, la rajouter à `actifs`, et mettre sa distance à cette de `c` plus un.

17. Compléter la fonction `carte_distances` correspondante.

À l'aide d'une telle carte des distances, on peut, pour une position donnée, obtenir une minoration du nombre de déplacements de caisses nécessaire pour arriver à la position gagnante : la somme des distances des cases où se trouvent les caisses. En effet, chaque caisse doit être déplacée au moins autant de fois que la distance de la case sur laquelle elle se trouve. Cette minoration est appelée *heuristique* de la position.

18. Compléter la fonction `heuristique`, qui prend en argument une carte et la carte des distances correspondante, et renvoie l'heuristique de la carte.

L'algorithme A^* est une variation sur le principe de l'algorithme de parcours en largeur. Au lieu de parcourir les sommets du graphe des cartes de façon concentrique, par distances à la position de départ croissantes, on les parcourt dans l'ordre des $d + h$ croissants, où

- d est la distance du sommet au sommet de départ ;
- h est l'heuristique du sommet.

$d + h$ est appelée *priorité*.

L'heuristique permet de guider l'algorithme dans la direction de la position gagnante, et l'on peut prouver qu'il trouve une solution optimale.

Voici le pseudocode de l'algorithme $A^\star$.

Pseudocode : Algorithme $A^\star$

- Ajouter la carte de départ à `actifs` et `dejavu`. Mettre sa distance à 0.
- Tant que `actifs` est non-vide :
  - extraire une carte de plus basse priorité `c` de `actifs` ;
  - pour chaque carte accessible à partir de `c` qui n'est pas dans `dejavu`, l'ajouter à `actifs` et `dejavu`, et mettre sa distance à celle de c plus un ;
  - pour chaque carte accessible à partir de `c` qui est dans `dejavu` et dont la distance est plus grande que celle de c plus un, mettre à jour sa distance.

`actifs` est ici une *file de priorité*, c'est-à-dire une structure surlaquelle on peut effectuer les opérations suivantes :
- tester si elle est **vide** ;
- **ajouter** un élément avec une certaine priorité ;
- **extraire** l'élément de plus basse priorité ;
- **mettre à jour la priorité** d'un élément présent.

Vous trouverez dans le fichier *taquin.py* une implémentation d'une file de priorité [3].

19. Compléter la fonction `plus_court_chemin_astar`, qui implémente l'algorithme $A^\star$ pour calculer une solution optimale.
Indication : on stockera comme valeurs du dictionnaire des couples (distance, parente).
Tester, et comparer le nombre d'itérations nécessaires pour trouver un plus court chemin avec celle du parcours en largeur.

20. ($\star$) Proposer (et éventuellement coder) une meilleure heuristique !

3. Le type `queue.priorityQueue` existe en *Python*, mais il n'est pas possible d'y mettre une valeur à jour, ce qui limite son utilisation pour les algorithmes de parcours.

Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".

Annexe

Voici les fonctions et méthodes sur les listes, dictionnaires et deque dont l'usage est autorisé – ainsi que les complexités que vous utiliserez pour les calculs de complexité (en fonction de la longueur de la liste n). Tout autre fonction dont vous auriez besoin doit être implémentée !

Fonctions et méthodes sur les listes

Opération	Exemple	Complexité
Création d'une liste vide	`l=[]`	$O(1)$
Accès direct	`l[0]`	$O(1)$
Longueur	`len(l)`	$O(1)$
Concaténation	`l1+l2`	$O(n1+n2)$
Ajout en fin de liste	`l.append(1)`	$O(1)$
Suppression en fin de liste	`l.pop()`	$O(1)$
Extraction de tranche	`l[1 :10]`	$O(n)$, où n est la longueur de la tranche.
Répétition	`[0]*k`	$O(n)$, où n est la longueur de la liste créée.
Création par compréhension	`[k**2 for k in range(n)]`	$O(n)$ si l'expression est évaluée en temps constant

Fonctions et méthodes sur les dictionnaires

Opération	Exemple	Complexité
Création	`d = {cle : valeur}`	$O(1)$
Test d'appartenance d'une clé	`cle in d`	$O(1)$
Ajout d'un couple clé/valeur	`d[cle] = valeur`	$O(1)$
Valeur correspondant à une clé	`d[cle]`	$O(1)$

Fonctions et méthodes sur les deque

Opération	Exemple	Complexité
Création	`q = deque()`	$O(1)$
Ajout à la fin	`q.append(e)`	$O(1)$
Suppression au début	`e = q.popleft()`	$O(1)$
Longueur	`len(q)`	$O(1)$

Ce document est mis à disposition selon les termes de la licence Creative Commons "Attribution - Pas d'utilisation commerciale 4.0 International".