

Unblock me : Résolution du jeu en python

Par Yannick NANA et André PIRES, L3MIASHS Sciences cognitives, Université de Lille.

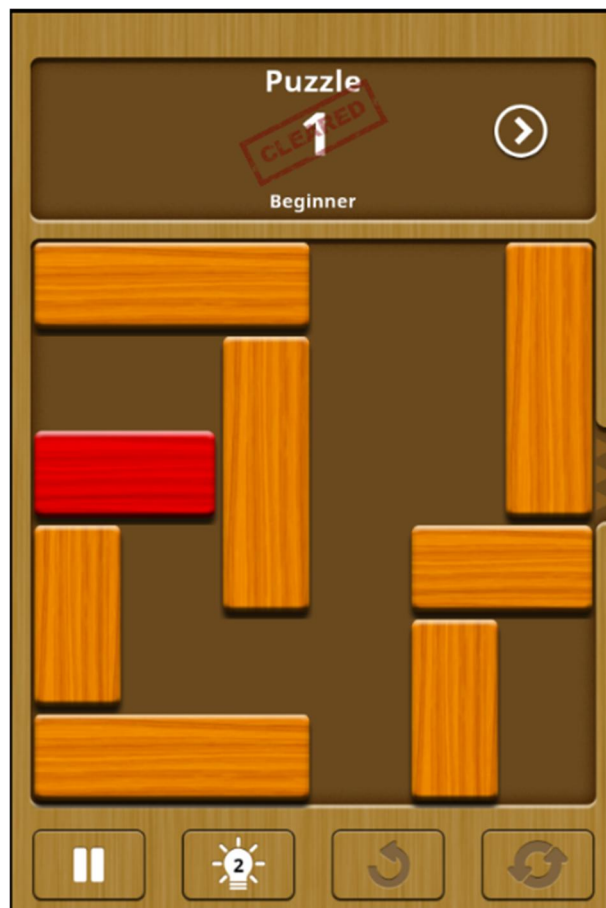
1.Description du domaine choisi

1.1 Règles du jeu

L'interface du jeu Unblock Me est composée d'une grille de taille 6x6, sur laquelle des blocs sont places. Le joueur doit déplacer les blocs horizontaux de gauche à droite, et les blocs verticaux de haut en bas afin de débloquer le chemin du bloc rouge (figure), menant à la sortie à droite. Le bloc rouge se situe toujours sur la ligne de la sortie.

Le problème concerne le déplacement des blocs qui permettrait au bloc directeur d'atteindre la position finale.

Le bloc traditionnellement de couleur rouge, censé atteindre la position finale sera désigné par « bloc directeur » tandis que les autres en tant que « obstacles ».



Capture d'écran du jeu Unblock Me par Kira Games

2. Présentation de l'algorithme codé, des fonctionnalités implémentées

2.1 Résolution du problème

2.1.1 Formalisation

La grille sera représentée sous la forme d'une matrice NxN, ici $n = 4$.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Chaque bloc inséré occupera 2 cases de la matrice.

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Chaque état sera représenté sous la forme d'une liste qui contiendra : plusieurs listes de coordonnées. L'état ci-dessus sera donc représenté par :

```
etat = [[1, 0, 1, 1], [0, 2, 1, 2]]
```

Pour chaque état, le bloc directeur sera toujours le premier élément de la liste et vu qu'il sera toujours à la même position, ses coordonnées seront toujours [1, 0, 1, 1] en début de partie.

2.1.2 Fonctionnalités implémentées

La formalisation du jeu tourne autour de la classe Blocs, qui contient les fonctions suivantes :

Actions applicables à un état pour un bloc donné :

- move_down, move_up, move_right, move_up

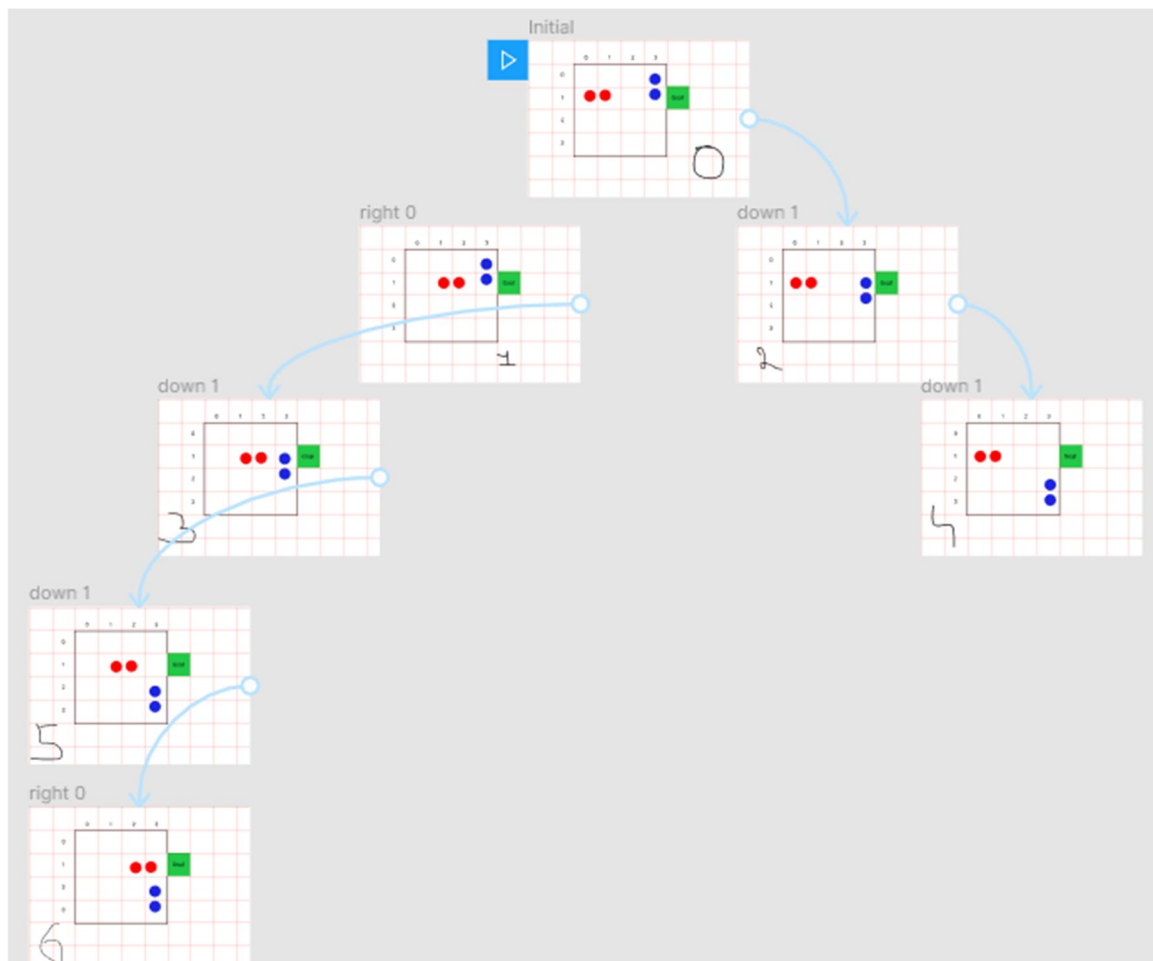
Preconditions:

- precond_down, precond_up, precond_right, precond_up

Les algorithmes de recherche en profondeur se basent sur les fonctions vues en cours. La recherche en largeur a été programmée en suivant l'algorithme du diaporama. Nous avons modifié les variables retournées par chaque algorithme afin qu'elles puissent au mieux servir à la représentation visuelle de la solution (retourne l'action de l'opérateur plutôt que le nom).

L'algorithme de recherche en largeur parcourt tous les nœuds possibles tandis que la Classe Nœud garde en mémoire, pour chaque nœud :

- L'état (les coordonnées des blocs de la matrice).
- Le label de son parent direct (racine = 0).
- Le mouvement (action de l'opérateur) qui a permis d'y arriver.



Pour retrouver le cheminement de la solution on effectue une remontée successive en partant du nœud solution au nœud racine. Pour la simulation ci-dessus, l'algorithme retournera les mouvements qui ont permis d'atteindre le nœud 6.

3. DOCS

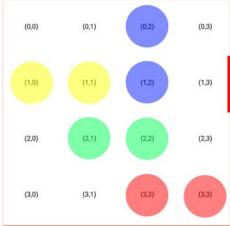
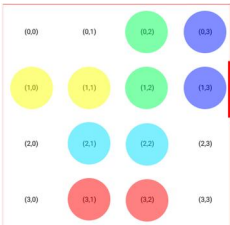
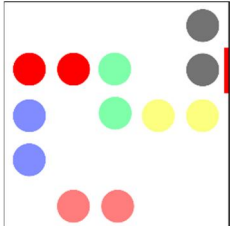
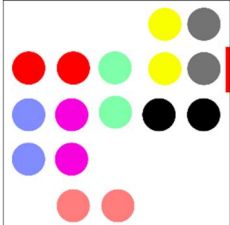
3.1 Fonctions

Label	Paramètres	Return	Effet
<code>est_final(e)</code>	état	Booléen	Vérifie si l'état est final.
<code>make_board(size)</code>	taille	plateau vide	Construction de la grille de jeu en fonction de la taille souhaitée
<code>fill_board(etat)</code>	état	plateau	Construit le plateau à partir d'un état.
<code>show(plateau)</code>	plateau	None	Affiche la grille de façon visuelle.
<code>copie(e)</code>	état	état copiée	Copie l'état.
<code>make_operateurs(blocs)</code>	blocs	operateurs	Constructions des opérateurs disponibles pour les blocs initialises
<code>move_down(self, e)</code>	bloc, état	état modifié	Effectue le déplacement du bloc dans l'état spécifié.
<code>move_up(self, e)</code>	bloc, état	état modifié	Effectue le déplacement du bloc dans l'état spécifié.
<code>move_right(self, e)</code>	bloc, état	état modifié	Effectue le déplacement du bloc dans l'état spécifié.
<code>move_left(self, e)</code>	bloc, état	état modifié	Effectue le déplacement du bloc dans l'état spécifié.
<code>precond_down(self, e)</code>	bloc, état	Booléen	Vérifie si le déplacement est possible.
<code>precond_right(self, e)</code>	bloc, état	Booléen	Vérifie si le déplacement est possible.
<code>precond_up(self, e)</code>	bloc, état	Booléen	Vérifie si le déplacement est possible.
<code>precond_left(self, e)</code>	bloc, état	Booléen	Vérifie si le déplacement est possible.
<code>show_result(solution, e)</code>	solution, état	None	Effectue de façon séquentielle tous les mouvements menant à l'état final.

4. Performance

La Recherche en profondeur libre pas possible car l'arbre est trop grand. Le blocage s'effectue vers une profondeur de 14 (programme tourne continuellement).

Initialement la grille faisait 4x4 et le nombre de blocs était limite à 4. Le programme final permet d'augmenter la taille de la matrice et le nombre de blocs. Le plus grand problème que nous avons teste possédait 8 blocs sur une matrice 5x5.

Taille	SETS	Algorithme	Temps	Nombre de coups
4x4 4 blocs		Profondeur limitée	48.08ms	7
		Profondeur mémoire	26.96ms	13
		Largeur	0.0s	7
4x4 5 blocs		Profondeur limitée	841.51ms	8
		Profondeur mémoire	61.95ms	20
		Largeur	1.99ms	8
5x5 6 blocs		Profondeur limitée	3995.67ms	9
		Profondeur mémoire	37.97ms	91
		Largeur	31.98ms	9
5x5 8 blocs		Profondeur limitée	Trop long	14
		Profondeur mémoire	56 ms	185
		Largeur	175 ms	14

Bilan : La recherche en largeur la plus adaptée car elle donne une solution rapide pour un nombre de coups minimum.

Scenario de tests

Toutes les modifications s'effectuent dans le fichier *unblockme.py*. Le rafraichissement du terminal a été conçu pour Windows (modification nécessaire des lignes 74 & 80 pour le lancer sur Linux) :

```
system("cls")
```

Pour changer de problème à résoudre il faut modifier trois points :

La taille de la grille :

```
# Création de la grille de jeu
empty_board = make_board(5)
```

Les coordonnées des blocs à insérer dans la grille de jeu. Le fichier *sets.py* contient des exemples. Le dossier *boards* contient des images associées à chaque exemple.

```
# Initialisation des blocs
Blocs([1, 0, 1, 1])
Blocs([2, 0, 3, 0])
Blocs([1, 2, 2, 2])
```

Et enfin choisir l'algorithme de résolution :

```
# Lancement de la résolution

# solution = recherche_en_profondeur_memoire(Blocs.initial, partial(est_final, len(empty_board)), operateurs_disponibles, [])

solution = recherche_en_profondeur_limitee(Blocs.initial, partial(est_final, len(empty_board)), operateurs_disponibles, 14)

# solution = recherche_en_largeur(Blocs.initial, partial(est_final, len(empty_board)), operateurs_disponibles, [], False)
```

5. Conclusions

Difficultés rencontrées

- Au début on modifiait constamment les instances des blocs, nous nous sommes rendus compte que modifier les mêmes instances plusieurs fois en parallèles n'était pas une bonne idée au vu des multiples problèmes que cela a créé.
- Vu la nature du jeu, nos operateurs se devaient de prendre 2 paramètres (le bloc à déplacer et l'état de la matrice). Ce problème a été résolu grâce à l'utilisation de Partials
- Nous avons rencontré des problèmes sur la nature de l'état qui pouvait soit contenir
 - o Chaque ligne du plateau. Cette option paraissait évidente parce que nous avons essayé de représenter ce que nous voyons lorsqu'on jouait.
 - o Les coordonnées de chaque bloc. Cette option est meilleure car elle donne des informations sur la position de chaque bloc et facilite la détection de collisions. Le seul inconvénient était que notre fonction *est_final()* ne pouvait pas détecter la taille

de notre grille uniquement a partir des coordonnes du bloc et ne pouvait donc pas s'adapter à différente grille. Il aurait donc fallu rajouter la taille de la grille en paramètre. L'utilisation de Partials a résolu ce problème.

- Recherche en largeur :
 - Remontée de l'état final au nœud racine
 - Utilisation de récursivité comme pour la profondeur ne permettait pas de transférer la liste des nœuds parcourus a toutes les boucles While qui s'effectuaient en parallèle. Cette méthode a produit un algorithme qui donnait une solution avec autant de coups que la recherche en profondeur mémoire mais en + rapide.

Améliorations possibles :

- Possibilité d'insérer des blocs de 3 cases, comme dans le vrai jeu Unblock Me.
- Interface graphique de résolution sur Pygame ou en javascript
- Possibilité de jouer au jeu avec un lancement de l'algorithme à un instant T pour servir d'aide au joueur s'il se trouve dans une impasse.
- Algorithme de recherche en largeur pas au point, certaines variables ont le même rôle (ex : fermes et Nœud.etats), ce qui augmente peut-être le temps de recherche.