Lartigau Théo
Siahaan–Gensollen Rémy

Rapport du projet de programmation (Swap Puzzle)

SOMMAIRE

I Introduction

Ce rapport présente nos (Lartigau Théo et Siahaan–Gensollen Rémy) travaux sur le projet de programmation de première année, le Swap Puzzle. Le code est accessible sur GitHub à l'adresse

`https://github.com/codcordance/ensae-prog24`

I.1 Structure

Globalement, le projet a été agencé sous la forme d'un package python nommé `swap_puzzle`. Le package principal contient notamment le module `Graph` et le module `Grid` qui ont été implémentés, et un module `SPUtils` qui contient des fonctions utilitaires pour la résolution du problème.

Elle contient également un module `solver` : celui-ci fournit une classe abstraite `Solver` munie d'une méthode abstraite `solver.work`, prenant en entrée la taille d'une grille et son statut[1], mais pas l'objet grille en lui-même, et renvoyant une liste de swaps. Les différents « solveurs » sont donc des implémentations de cette classe et de cette méthode. Ainsi le « solveur » n'interagit jamais directement avec l'objet grille.

[1] Ainsi qu'un accumulateur pour optimiser les calculs, se référer à la documentation de `solver.work` et de `solver.solve`.

Dans le subpackage (`swap_puzzle.solvers`) sont placés les « solveurs » implémentés. Un autre subpackage (`swap_puzzle.exception`) contient de plus des implémentations d'exceptions qui sont utilisées dans le projet. Les initialisateurs permettent d'importer directement les classes : typiquement, l'instruction

```
from swap_puzzle import *
```

importe les classes `Grid`[2], `Graph`, les exceptions, les « solveurs »,.... Elle importe également (sauf pour le module d'interface graphique) les bibliothèques standards

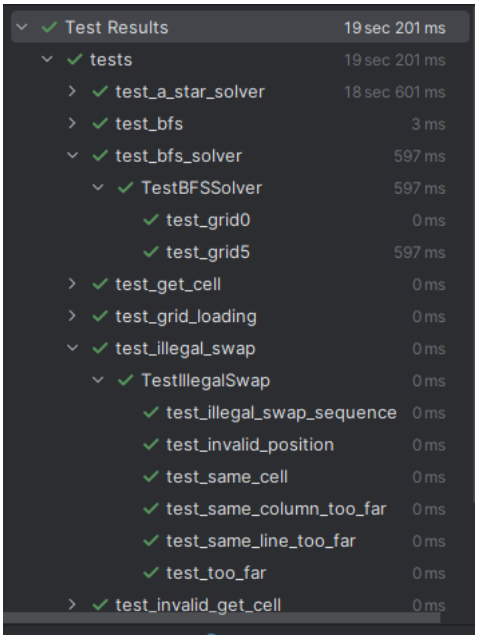
[2] On pourrait donc écrire `from swap_puzzle import Grid`, par exemple.

utilisées. L'import des subpackage est également implémenté pour ne charger que les modules nécessaires.

I.2 Code

Un dossier `tests` est présent à la racine et contient un ensemble de tests unitaire. Il est basé sur le jeu de tests qui avait été fourni et que nous avons enrichi. Il contient un total de 28 tests qui passent tous[3] :

[3] Sur un environnement conda, sur l'un de nos ordinateurs personnels, l'ensemble des tests passent en 19.30 secondes



Le code a été globalement documenté et type-hint. Une documentation a été générée par Sphynx et est accessible sur github pages à l'adresse `https://codcordance.github.io/prog24/index.html`. Il est parfois commenté lorsque le code est moins intuitif.

Nous avons également apporté un soin à l'optimisation des calculs : les « solveurs » utilisent typiquement un accumulateur (ou un cache dans le cas de A*) qui leur permet de minimiser le nombre d'opérations (au prix d'un peu de mémoire). Nous avons également veillé à faire attention lors de notre utilisation des primitives : on utilise par exemple les itérateurs de python et les listes par compréhension qui consomment moins de ressources que d'autres méthodes. Par exemple, plutôt que décrire `l[::-1]` pour inverser la liste `l`, ce qui s'exécute en une complexité temporelle en $\mathcal{O}(n)$ et spatiale en $\mathcal{O}(n)$, on utilisera `reversed{l}`, ce qui s'éxécute en une complexité temporelle en $\mathcal{O}(1)$ et spatiale en $\mathcal{O}(1)$.

I.3 Interface graphique

L'interface graphique repose sur la librairie `Pygame`. L'interface est initialisée directement avec une grille et un solver donnés et peut prendre deux états : `idle` et `animating` en fonction de ce qui doit être affiché. En toute circonstance, la méthode `draw_grid` est appelée à chaque frame. Lorsque l'interface est lancée, l'ensemble des « swaps » qui forment la solution du puzzle est présenté à l'utilisateur dans une animation. La méthode `trigger_swap` met l'interface graphique dans l'état `animating` et lance l'animation de l'échange entre deux grilles qui est définie dans la méthode `animate_swap`. Lorsque l'animation est finie et tant

qu'il reste des swaps à animer dans la solution[4], la méthode `trigger_swap` est de nouveau appelée jusqu'à ce que l'ensemble de la solution soit présenté. L'interface graphique rentrera alors dans l'état `idle` de manière définitive.

[4] La solution est traitée comme une queue contenant des swaps, une queue étant une structure de données de type premier entrant, premier sortant ou « FiFo »

La classe abstraite `Solver` permet l'utilisation d'un callback qui peut également être utilisé pour représenter le cheminement de l'algorithme plutôt que la seule solution qu'il renvoie. Il faut pour cela une méthode que l'on nommerait `push_swap` et qui ajouterai les swaps tentés par le « solveur » à la queue de swaps à animer.

II Résolution naïve

La résolution naïve fonctionne comme un tri à bulle. Pour cela, on va « aplatir » l'état pour passer d'une liste de liste (la liste de chaque ligne) à une grande liste. On pourrait initialement penser à juste concaténer les lignes de la grille :

$$\begin{bmatrix} \texttt{1, 2, 3} \\ \texttt{4, 5, 6} \\ \texttt{7, 8, 9} \end{bmatrix} \quad \longrightarrow \quad \texttt{[1, 2, 3, 4, 5, 6, 7, 8, 9]}$$

Cependant ici, un swap entre deux éléments adjacents de la liste ne serait pas systématiquement valide : on pourrait typiquement échanger `3` et `4`. Pour palier ce problème, on peut inverser une ligne sur 2 :

$$\begin{bmatrix} \texttt{1, 2, 3} \\ \texttt{4, 5, 6} \\ \texttt{7, 8, 9} \end{bmatrix} \quad \longrightarrow \quad \texttt{[1, 2, 3, 6, 5, 4, 7, 8, 9]}$$

Ceci entraîne un autre problème : trier la liste aplatie dans cet état amènerait la grille a être résolue en « serpent » :

$$\texttt{[1, 2, 3, 4, 5, 6, 7, 8, 9]} \quad = \quad \begin{bmatrix} \texttt{1, 2, 3} \\ \texttt{6, 5, 4} \\ \texttt{7, 8, 9} \end{bmatrix}$$

Pour régler ce problème, on fait intervenir dans l'algorithme de tri un ordre différent de celui usuel, qui inverse une ligne sur deux. Dans l'exemple ci-dessus exemple, il s'agirait de l'ordre $\prec_\alpha$ tel que

$$1 \prec_\alpha 2 \prec_\alpha 3 \prec_\alpha 6 \prec_\alpha 5 \prec_\alpha 4 \prec_\alpha 7 \prec_\alpha 8 \prec_\alpha 9$$

On arrive alors à résoudre correctement la grille. Du fait de ces calculs, avec $n \times m$ la dimension de la grille, cet algorithme opère (dans le pire cas) temporellement en $\mathcal{O}\Big((nm)^2\Big)$ et spatialement en $\mathcal{O}(nm)$.

III Résolution Breadth-first search

La deuxième résolution implémentée est celle avec l'algorithme *Breadth-first search*. On considère qu'un état de la grille est assimilable à un élement de $\mathfrak{S}(\llbracket 1, n \times m \rrbracket)$, et on encode chacun des états par une chaîne de caractère[5] pour pouvoir les utiliser comme sommets d'un graphe.

[5] voir `state_to_str` et `str_to_state`.

On construit donc un graphe tel que chaque sommet correspond à une permutation de la grille, et on détermine ensuite quels sommets sont voisins : deux sommets sont reliées par une arrête si les états correspondant sont égaux à un

swap près (c'est-à-dire si l'on peut passer de l'un à l'autre avec un swap). On applique enfin l'algorithme BFS pour obtenir le chemin entre la grille de départ et la grille d'arrivée, ce qui correspond à une séquence de swaps que l'on peut opérer sur la grille.

En terme de correction, puisque les poids du graphe obtenu sont tous égaux (graphe uniforme), l'algorithme BFS assure que l'on trouve le plus court chemin, et donc ici le nombre minimum de swaps. Cependant, il faut déterminer si chaque sommet est voisin avec un autre. Puisqu'il y a $|\mathfrak{S}(\llbracket 1, n \times m \rrbracket)| = (n \times m)!$ sommets, on a donc $((n \times m)!)^2$ arrête à vérifier. Une arrête se vérifie en $\mathcal{O}((n \times m)!)$, donc l'algorithme s'éxécute temporellement en $\mathcal{O}((n \times m)!)^3$, ce qui est extrêmement long pour des grandes grilles. On ne le teste donc que sur des grilles de maximum 6 éléments.

Pour optimiser ce calcul, on pourrait sauvegarder pour chaque dimension de grille le graphe déjà construit. Celle solution demanderait un peu d'espace, mais permettrait d'économiser énormément de temps. Une autre solution consiste à ne pas chercher absolument le meilleur chemin, mais un chemin fonctionnel et relativement court, à l'aide de l'algorithme $A*$.

IV Résolution A*

La résolution A* est une implémentation de l'algorithme $A*$ appliquée à notre problème. Plutôt que de calculer initialement toutes les arrêtes du graphe, on se munit d'une heuristique, correspondant au nombre de différences avec l'état final[6], et on ne détermine si deux sommets sont voisins que lors de l'éxécution de l'algorithme.

[6] voir `cell_difference`

Ce changement permet nottament de trouver la solution rapidement lorsqu'il y a un faible nombre de swaps à opérer. Par exemple, on peut l'exécuter sur l'exemple 2 de grille qui a été fourni, alors que le solveur BFS met trop de temps. Cependant, notre implémentation du calcul des voisins est mal faite, et entraîne un nombre déraisonnables de calculs inutiles, même avec l'utilisation d'un cache.