Projet de programmation

ENSAE Paris - 2023/24

Auteurs : *Mounir FARHOUN, Samuel NORSA*

Ce document a pour but de décrire notre projet de programmation en langage Python, orienté vers l'implémentation d'un algorithme de tri d'une grille.

Sommaire

Vue d'ensemble et squelette du projet

Le projet consiste en l'implémentation d'un algorithme de tri d'une grille de taille n x m, où chaque case contient un nombre différent des autres, compris entre 1 et n x m. On souhaitera, au fil du rapport, présenter les différentes méthodes utilisées pour arriver à nos fins, en les rendant de plus en plus optimales. On utilisera d'abord un tri **naïf**, un tri **BFS (breadth-first search)**, puis l'algorithme **A***.

Le squelette du projet était composé :

- d'une classe *Graph* minimaliste
- d'une classe *Grid*
- d'une classe *Solver*
- d'un fichier *main.py*, utile pour les tests.

Nous avons mis en place un dépôt *GitHub* commun afin de faciliter la mise à jour de nôtre projet, et avons donc dû nous familiariser avec le *Terminal*.

La classe Grid et première solution naïve

Implémentation de méthodes essentielles

Nous avons commencé par implémenter **trois** méthodes cruciales pour la suite de notre projet, à savoir `swap`, `swap_seq` et `is_sorted`. Elles ont respectivement pour but :

- d'effectuer un échange entre deux cellules de la grille lorsque cela est permis
- d'effectuer une série de swap
- de vérifier si la grille actuelle correspond à la grille triée.

Nous ne revenons pas sur l'implémentation de ces méthodes, comme elles ne présentent pas de difficulté que nous ne pouvons expliquer à l'oral.

Une première solution naïve

L'idée que nous avons retenue derrière cette première solution est de placer chaque nombre de la grille à sa place, selon l'ordre croissant. Nous commencerons donc par placer le nombre 1, puis 2, et ainsi de suite.

Pour ce faire, nous déterminons déjà les coordonnées actuelles du nombre, à l'aide de la méthode `find_index`, conçue avec deux boucles imbriquées ; puis les coordonnées de ce nombre dans la liste triée, à l'aide de `find_index_when_sorted`, qui consiste en la construction de la grille triée puis en l'utilisation de `find_index` sur celle-ci, sur le nombre qui nous intéresse.

Enfin, attaquons-nous à la construction du *solver* naïf, qui comportera deux méthodes : `correct_spot`, et `get_solution`. La première méthode a pour but de prendre en entrée un nombre, puis de le placer comme si la grille était triée. La seconde itère ce processus sur tous les nombres de la liste, en ajoutant à chaque itération le/les swap/s effectués dans une liste. Nous récupérerons donc `self.n` x `self.m` listes, dont les éléments seront ajoutés à une liste finale (`path`), qui contiendra la liste de tous les swaps effectués.
Passons en revue ces deux méthodes, en expliquant leur fonctionnement, puis en évaluant leur complexité.

La méthode `correct_spot` vérifie premièrement que le nombre n'est effectivement pas à sa place. Si ce n'est pas le cas, il commencera par le placer sur la bonne colonne. Elle détermine ainsi si le nombre est à gauche ou à droite de la bonne colonne, ainsi que le nombre de colonnes qui les sépare, et appliquera `swap_seq` autant de fois qu'il le faut. Une fois cela fait, on applique à nouveau ce raisonnement sur les lignes. Dès lors, après des mouvements en "L", le nombre sera là où il faut.

Pour finir, `get_solution` applique `correct_spot` sur tous les éléments de la grille de départ grâce à une boucle for, puis crée la liste de tous les swaps effectués. Voilà donc finie nôtre première ébauche de solution, dont la complexité est quadratique (on effectue des tests sur tous les éléments de nôtre grille).

Optimisation de la longueur du chemin

Une approche intéressante consiste à visualiser la résolution d'une grille comme une exploration dans un réseau spécifique. Dans ce réseau, chaque nœud représente un état possible de la grille, et les connexions entre les nœuds indiquent qu'un échange valide peut être effectué entre les états correspondants. Ainsi, résoudre la grille en trouvant le chemin le plus court dans ce réseau revient à naviguer depuis le nœud représentant l'état initial de la grille jusqu'au nœud où la grille est complètement ordonnée. Dans cette optique, utilisons un algorithme de type BFS (*breadth-first search*, ou parcours en largeur). On ne reviendra pas sur le principe de l'algorithme, seulement la façon dont nous l'avons implémenté.

Dès lors, on crée, au sein de la méthode BFS deux listes, l'une qui va contenir les nœuds visités (`visited`) et l'autre qui contient ceux qui vont l'être (`queue`). Tant que `queue` n'est pas vide, on extrait de cette dernière le prochain nœud à visiter. On récupère les voisins de ce nœud, et voyons si l'un d'entre eux a déjà été visité. En bouclant sur les voisins du nœud actuel, on copie le chemin précédent et y ajoute le voisin considéré dans l'itération. Si ce voisin se trouve être la destination, on s'arrête ici.

Pour ce qui est du calcul de la complexité de notre algorithme BFS, on pourra dire que, dans le pire des cas, chaque sommet est visité une unique fois et que chaque arête n'est pas empruntée plus d'une fois. En somme, on peut dire que sa complexité est un $O(S+A)$, où S représente le nombre de sommets et A le nombre d'arêtes.

Par ailleurs, on peut noter que les questions 5. et 8. ont été abordées en même temps : en effet, notre algorithme est "optimisé" en ce qu'il ne visite pas le graphe dans son intégralité, mais s'est limité à la destination, et donc à la partie du graphe qui y correspond.

Occupons-nous maintenant d'optimiser BFS en créant le graphe de tous les états possibles.

Avant tout, on importe `permutations` de la librairie `itertools` qui nous permettra d'arriver à nos fins. On nommera la méthode qui nous retourne toutes les grilles possibles en partant d'un état donné `permutations_to_grid`. Celle-ci consiste en la création de toutes les permutations possibles de {1, 2, … , self.m x self.n}. On initialise ensuite une liste vide, que l'on va remplir à coup de tuple de tuple qui seront issus de toutes les permutations possibles, à l'aide d'une boucle. Ainsi, notre liste contiendra tous les états possibles, et il suffira d'appeler `grids` dans le terminal pour voir toutes les grilles possibles en partant d'un état donné.

L'algorithme A*

Quelques mots en préambule : plusieurs fonctions apparaissent au début de notre fichier `main.py`, notamment des fonctions de "conversion". Cela est dû au fait que plusieurs des procédés que nous allons utiliser dans la suite de cette section ne s'appliquent que sur des objets hashables.

On ne reviendra pas en détail sur le fonctionnement de l'algorithme A*, mais allons revenir sur les choix/subtilités de notre programme.

Premièrement, pourquoi avoir utilisé `heapq` ? En effet, à première vue, on peut se demander pourquoi ne pas fonctionner comme dans BFS, où l'on ajoute et extrait les voisins successivement jusqu'à arriver à notre destination. Ici, on ajoute un "critère" d'ajout et de suppression des tas (ce terme n'est pas utilisé par hasard) `open_set` et `closed_set`. Le premier tas fait référence aux nœuds à explorer, alors que le second fait référence aux nœuds déjà évalués. Le dit critère est en fait un score, un coût estimé, et l'on choisira, dans A*, le nœud qui aurait le coût le moins élevé (la variable `f_score` de notre algorithme).

La prochaine question à laquelle nous allons répondre est : comment calcule-t-on ce score ? Habituellement, on dit que le `f_score` d'un noeud dans un algorithme A* est la somme de deux autres caractéristiques de ce nœud : sa distance au "nœud destination", et le coût que cela représente, que l'on identifie à une distance mathématique et à une "fonction heuristique". En termes de distance réelle, notre choix s'est porté sur la distance constante égale à 1, en ce que le déplacement d'un nœud à un autre "coûte" un swap. Enfin, notre heuristique est la distance euclidienne ("à vol d'oiseau"), plutôt que celle de Manhattan. Nous avons choisi cette dernière car nous avons pensé que la valeur numérique de cette heuristique, entre deux noeuds du graphe, était représentative de l'écart réel entre ces deux grilles, mais aussi car elle nous aidait à visualiser cela comme un espace "physique", et donc à garder un caractère moins abstrait.

Pour finir, nous avons construit l'algorithme de façon à récupérer le chemin le plus court d'un nœud du graphe à un autre, lorsque celui-ci existe, et à le renvoyer grâce à `reconstruct_path`.

Tests et version jouable

Nous avons ajouté deux fichiers qui n'étaient initialement pas présents : `test_temporaire.py` et `game.py`.

Le premier présente un test respectif pour les trois algorithmes présentés dans ce rapport, accompagnés du temps qu'il ont pris à s'exécuter et ce qu'ils ont retourné. On observe, à chaque test réalisé, qu'A* est le plus rapide, comme attendu.

Pour conclure, nous avons ajouté un fichier contenant une version interactive de l'algorithme. Un état initial est engendré, puis, tant que la grille n'est pas dans son état triée, le joueur peut, à l'aide du terminal, effectuer des *swaps* jusqu'à l'objectif. *Pygame* n'ayant pas fonctionné, nous avons fait le choix de représenter ces fenêtres sous *matplotlib.pyplot*. Le joueur rentre les coordonnées des cases qu'il veut échanger, si cela est permis, puis l'état de la grille s'affiche à nouveau et ainsi de suite.