Iris Beaussant, Lucy Neveu

Projet de programmation

- swap puzzle -

Note: Le fichier main.py contient la liste des différentes fonctions ainsi que leur emplacement
Tout le code est commenté avec des explications. Les réponses aux questions du sujet sont également en commentaire du code.

Le code se trouve à l'emplacement suivant: https://github.com/irisbeaussant/ensae-prog24

I Introduction

A-Objectif

L'objectif du projet est de résoudre le problème suivant:

On dispose d'une grille de m lignes et n colonnes. Chaque case contient un entier de 1 à mn. Il faut trouver la séquence de mouvements la plus courte possible afin de ranger la grille, i.e. les nombres sont à la bonne position. Les mouvements acceptés sont les échanges de cases voisines, horizontalement ou verticalement.

B-Ressources

On dispose de plusieurs classes dont la classe Grid qui permet de définir un objet de type grille qui a comme attributs le nombre de colonnes et de lignes de la grille, et la grille sous forme de listes de listes. La classe Graph permet de définir un objet de type graphe avec comme attributs les sommets, le nombre de sommets, les arêtes et leur nombre ainsi qu'un graphe sous forme de dictionnaire.

On dispose également de plusieurs grilles de tailles différentes, qui permettent de faire des tests dont certains sont également fournis.

II-Code

A-Implémentation des fonctions permettant de réaliser les mouvements

L'ensemble des fonctions citées dans cette sous-partie se trouvent dans le fichier grid.py.

La fonction permettant de réaliser les mouvements de cases, c'est-à-dire les échanges de cases voisines, est la fonction **swap**. Celle-ci prend comme arguments les deux cases à échanger. Cette fonction vérifie d'abord que les cases sont bien voisines, soit horizontalement soit verticalement. Si cette condition n'est pas vérifiée, la fonction renvoie un message d'erreur. Sinon, les deux valeurs sont échangées à l'intérieur de la liste de listes.

La fonction **swap_seq**, prenant en argument une liste de couples de coordonnées de cases, permet de réaliser une série d'échanges de cases à l'aide de la fonction **swap** définie ci-dessus.

La fonction **is_sorted** permet de vérifier si une grille donnée est bien rangée, i.e. les cases sont bien dans le bon ordre. Puisqu'une grille de taille m*n contient tous les entiers de 1 à mn, il suffit de regarder si la case de coordonnées (i,j) contient bien un nombre strictement plus petit que la case de coordonnées (i,j+1). C'est ainsi que la fonction is_sorted vérifie si la grille est rangée. Il y a deux boucles for, si la grille est bien rangée le coût est de m*(n-1), sinon le coût est inférieur et dépend de la position de la première case mal placée.

Les tests **test_swap.py** et **test_is_sorted.py** permettent de vérifier le bon fonctionnement de ces fonctions.

B-Solution naïve

L'idée est ensuite d'implémenter une première méthode permettant de résoudre la grille. Cette méthode se trouve dans la classe **Solver**.

La fonction **get_solution** permet de trouver une solution de façon naïve. Elle consiste à trouver chaque nombre de la grille en commençant par 1 et à les mettre à la bonne place, un à la fois. La fonction trouve donc le 1 avec la fonction trouver, puis calcule la position qu'il doit avoir dans la grille résolue et le déplace verticalement puis horizontalement pour le placer au bon endroit. La fonction recommence ensuite avec le 2 et ainsi de suite jusqu'à la résolution de la grille.

Cette fonction, bien qu'elle soit utilisable quel que soit l'état initial de la grille, renvoie une solution naïve car elle ne prend pas en compte que les swaps faits pour déplacer une grille peuvent également servir à mettre une autre case au bon emplacement. En réalité, la grille peut donc être résolue en effectuant moins de swaps.

Le nombre de swaps effectués dépend de l'état initial de la grille. Il peut donc être relativement faible si la grille est presque triée, ou assez élevé si chaque case est à l'opposé de sa bonne position par rapport à la diagonale par exemple. Dans ce dernier cas on peut imaginer qu'il faudra alors (m+n) swaps pour mettre les premières cases au bon emplacement puis m+n-2 pour la rangée du dessous, et ainsi de suite.

Si la grille est triée la complexité sera de $(m*n)^2$ car on ne rentre pas dans la boucle while car la condition n'est jamais vérifiée. On entre dans la boucle for m*n fois puis il faut utiliser la fonction trouver à chaque boucle, celle-ci ayant une complexité d'au plus m*n selon l'emplacement de la case. Dans le pire des cas la complexité sera en $O((m+n)*(mn)^2)$. Cette valeur est surestimée car il ne faudra (m+n) swaps que pour une case se trouvant dans un coin de la grille et devant se trouver dans le coin opposé.

C-Représentation graphique

On peut visualiser une grille grâce à matplotlib. On peut en effet créer un tableau que l'on remplit avec les nombres. La fonction se nomme **representation_graphique_bis** et se trouve dans la classe **Grid**.

La figure ci-dessous montre ce que renvoie la fonction lorsque la grille donnée en argument est de taille 4*4 et est triée.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

D-Algorithme BFS

On souhaite maintenant trouver une solution plus optimale en termes de longueur de chemin que celle donnée par l'algorithme naïf abordé dans la partie B. Pour cela on implémente une méthode BFS (Breadth-first search) qui permet de trouver le chemin le plus court permettant de résoudre le problème. Cet algorithme se base sur l'utilisation d'un graphe. Celui-ci est représenté par un dictionnaire contenant les nœuds comme clés et leurs voisins comme valeurs associées. L'algorithme visite les nœuds en parcourant d'abord ceux qui sont à distance 1 de la source, puis à distance 2 et ainsi de suite jusqu'à trouver le nœud de destination - ici la grille rangée.

1-Première implémentation

On commence par implémenter une méthode BFS indépendante du problème traité. Celle-ci se trouve dans la classe **Graph** du fichier graph.py.

La méthode **bfs** prend en argument une source et une destination qui sont des entiers.On commence par initialiser une file à l'aide de Queue() avec la valeur de la source, ainsi qu'une liste correspondant aux sommets visités et un dictionnaire parents qui permettra de reconstituer le chemin parcouru jusqu'au noeud de destination.

La complexité de la méthode **bfs** est au pire en O(nombre d'arêtes + nombre de sommets). En effet, chaque sommet est ajouté au plus une fois à la liste des sommets visités, chaque arête est parcourue au plus une fois et chaque sommet est parcouru au plus une fois.

2-BFS pour les grilles

Une fois la méthode bfs mise en place, on peut l'adapter au parcours de graphe dont les clés et valeurs associées sont des graphes. La méthode **chemin_le_plus_court** se trouve dans la classe **Grid**.

On crée d'abord un graphe avec pour clés les grilles et comme valeurs associées les grilles voisines. La fonction **liste_noeuds_a_relier** renvoie les couples de grilles qui sont voisines (i.e. qui ne diffèrent que d'un swap). Il faut également s'assurer que les éléments du dictionnaire soient hashables (non mutables), on transforme ainsi les listes de listes qui représentent les grilles en des tuples. Il suffit ensuite d'appliquer la méthode bfs.

La fonction **chemin_le_plus_court** renvoie le chemin qui permet de passer de la source à la destination. La fonction **swaps_a_faire** la complète en renvoyant les swaps nécessaires pour réaliser le chemin.

3-BFS avec construction progressive du dictionnaire

La méthode **chemin_le_plus_court** n'est pas optimale car il construit le dictionnaire complet, or il ne faut pas forcément parcourir tout le graphe pour atteindre la destination. La solution est alors de construire le dictionnaire au fur et à mesure, pour ne construire que les parties dont on a besoin. LA fonction correspondante se nomme **bfs_bis** et se trouve dans la classe **Grid**.

On commence par initialiser le dictionnaire g. On construit ensuite la partie du dictionnaire correspondant à chaque sommet que l'on visite au moment où on le visite. Pour cela on cherche les grilles voisines du sommet que l'on traite. La première fonction qui servait à trouver les grilles voisines, **liste_noeuds_a_relier**, n'était pas optimale et ne permettait pas d'utiliser **bfs bis** avec des grilles de dimension supérieure à 2*2. On définit donc **liste_noeuds_a_relier_bis**. Cette fonction, au lieu de parcourir toutes les grilles possibles et de vérifier si c'est bien un voisin de la grille étudiée, va construire tous les voisins possibles.

E-Algorithme A*

L'algorithme A* permet d'optimiser encore plus la recherche de chemin. La méthode **A_star** se trouve dans la classe **Grid**. L'algorithme est un amélioration de BFS car il permet d'explorer en priorité certains chemins en utilisant une heuristique.

L'heuristique choisie consiste à calculer la distance de manhattan entre le sommet étudié et la destination, et de la diviser par deux. La division par deux est nécessaire car l'heuristique doit être une borne inférieure de la distance restante. Or pour échanger deux cases il suffit de faire un seul swap. D'autres heuristiques sont proposées dans la partie III mais celle choisie semble être la mieux adaptée.

Pour coder cette méthode on reprend le code de la méthode BFS avec construction du dictionnaire au fur et à mesure. Au lieu d'utiliser une file on utilise une file prioritaire qui contient des couples (distance à la destination, sommet). Ainsi les sommets sont parcourus dans l'ordre croissant de leur distance à la grille triée. L'heuristique est calculée grâce à la fonction **borne_inf_a_dst** se trouvant dans la classe **Grid**.

swaps_A_star complète la méthode **A_star** en renvoyant les swaps nécessaires pour réaliser le chemin le plus court.

III- Pour aller plus loin

A-Barrières

Il est possible de rendre la résolution de la grille plus difficile en ajoutant des barrières. Si deux cases sont séparées par une barrière alors on ne peut pas effectuer de swap entre celles-ci.

On commence par définir les barrières grâce à une liste de quadruplets (i,j,k,l) tels qu'il existe une barrière entre les cases de coordonnées (i,j) et (k,l).

La fonction **sep_par_barrieres**, se trouvant dans la classe **Grid**, prend en argument deux grilles voisines et la liste des barrières. Elle renvoie False si le swap pour passer d'une grille à l'autre n'est pas possible à cause d'une barrière, True sinon. La fonction **barrieres** correspond à la méthode **A_star** mais prend en compte les barrières. Ainsi avant d'ajouter un voisin au dictionnaire, la fonction vérifie si une barrière n'empêche pas le swap.

B-Cas particulier de la grille de taille 1*n

Le cas de la grille de taille 1*n revient à trier une liste. Puisque les cases ne peuvent être déplacées que par des swaps il faut une fonction qui trie la liste en comparant les cases deux par deux. Le tri bulle paraît alors adapté. La fonction **tri_bulles** se trouve dans la classe **Grid**.

La complexité est en $O(n^2)$ dans le pire des cas, en $O(n)$ dans le meilleur des cas, i.e. si la liste est déjà triée.

C-Heuristiques alternatives

L'heuristique choisie dans l'algorithme A* est la distance de Manhattan divisée par 2. On pourrait imaginer utiliser la distance de Manhattan et prendre la racine carré du résultat. Cette heuristique serait bien une borne inférieure de la distance à la destination. Cependant pour que l'algorithme soit le plus efficace il faut que l'heuristique soit la plus grande possible tout en restant inférieure à la véritable distance à la destination. Prendre la racine carrée renvoie un résultat plus petit que la division par deux. Ainsi l'heuristique choisie est plus efficace.

Une autre idée serait de prendre la distance euclidienne. Cependant l'algorithme ne permet pas de faire de swaps en diagonale. Ainsi la distance euclidienne est peu représentative de la véritable distance.

D-Difficulté

Il est possible de choisir le niveau de difficulté de la grille. C'est ce que fait la fonction **difficulte** de la classe **Grid**. L'utilisateur choisit le niveau de difficulté entre 1, 2 ou 3. La difficulté est mise en place en faisant varier le nombre de swaps. Des swaps sont faits aléatoirement à partir d'une grille triée afin de constituer le problème.