Rapport : Projet Programmation

Hugo Morand et Eve Samani

Mars 2024 - ENSAE Paris

Table des matières

1 Introduction : problème de tri de grille

On considère une grille m×n, où $m \geq 1$ et $n \geq 2$ sont des entiers représentant respectivement le nombre de lignes et de colonnes de la grille. La grille contient des carreaux numérotés de 1 à m*n. Dans l'état initial, les carreaux sont disposés arbitrairement dans la grille, l'objectif est de trouver la plus courte séquence de mouvements qui les amènent à être ordonnés en ligne (i.e., la première ligne contient 1,. . ., n ; la 2ème n + 1,. . .2n, etc.).

Les mouvements autorisés sont des échanges de carreaux consécutifs verticalement ou horizontalement ; c'est à dire qu'on a le droit d'échanger les carreaux (i_1, j_1) et (i_2,j_2) si et seulement si $i_1 = i_2$ et $|j_1 - j_2| = 1$ ou $j_1 = j_2$ et $|i_1 - i_2| = 1$. On n'autorise pas les échanges par les bords extérieurs (entre la première et dernière ligne, ou la première et dernière colonne).

Dans un premier temps, l'objectif de ce projet est d'implémenter des algorithmes renvoyant le chemin le plus court pour ordonner la grille. Ensuite, l'objectif sera de réduire leur complexité afin de pouvoir résoudre des grilles assez grandes en un temps raisonnable.

2 Séance 1 : une solution naïve dans la classe *Solver*

Lors de cette première séance, nous avons réalisé un algorithme naïf glouton pour ordonner la grille. Nous avons donc au préalable préparé l'environnement de travail de tout ce projet de programmation, puis nous avons commencé à implémenter la classe *Grid* et ensuite la classe *Solver*.

2.1 Environnement de travail : la classe *Grid*

La classe *Grid* regroupe plusieurs fonctions de base :

1. La méthode __init__ :
   — Elle définit les objets utilisés dans la classe Grid. On a donc deux entiers m et n correspondant au nombre de lignes et de colonne respectivement. Il y a aussi la variable state qui est une liste de liste correspond à l'état de la grille. Enfin, on instancie également la classe *Graph* qui sera utilisée plus tard dans les fonctions *bfs_grids* et *bfs_grid_upgrade*
2. Les méthodes d'affichage :
   — Les méthodes __str__ et __repr__ permettent d'avoir un affichage clair d'une grille lorsque l'on effectue des tests dans le *main.py*.
3. Fonctions qui effectuent des opérations élementaires sur la grille :
   — La fonction *swap* permet de vérifier si l'échange entre deux cellules est possible et si c'est le cas, l'échange est effectué sinon la fonction renvoie un message d'erreur.
   — La fonction *swap_seq* effectue plusieurs swap consécutifs dans une grille. C'est donc une fonction qui utilise la méthode *swap*.
   — La fonction *is_sorted* vérifie si la grille est ordonnée ou non. Cette fonction renvoie un booléen qui est True si la grille est ordonnée et False si ce n'est pas le cas.

Nous utiliserons ces fonctions et notamment la fonction *swap* lors des algorithmes de résolution de la grille (*Solveur*,*BFS*,*bfs_upgrade* et *A**) afin d'ordonner la grille.

2.2 Algorithme glouton : Classe *Solver*

L'algorithme glouton implique la recherche de solution d'optimisation "locale", coup après coup. Cependant, il nous a fallu déterminer quelle approche prendre afin d'arriver au plus vite à notre grille finale triée.

Afin de construire notre *Solver*, nous avons dû déterminer la méthode de lecture de la grille. Notre façon d'opérer est la suivante : nous allons placer une à une, dans l'ordre croissant, les valeurs de 1 à m*n dans la grille. Ainsi, pour chaque valeur, nous allons trouver sa position dans la grille, puis ensuite la comparer avec la position finale dans laquelle elle doit être pour qu'on ait une grille ordonnée. On va donc effectuer d'abord les échanges de cellules sur les colonnes jusqu'à ce que la valeur soit dans la bonne colonne puis on la "remonte" sur la bonne ligne. On va répéter ce mécanisme pour chacune des valeurs, afin d'atteindre une grille ordonnée.

La méthode *get_solution* de la classe *Solver* effectue cela et renvoie la liste des échanges effectués pour ordonner la grille. Cette méthode modifie également la grille pour qu'elle soit ordonnée. On vérifie tout cela sur différentes grilles dans le fichier *main.py* avec des tests sur des grilles de tailles différentes.

Le point positif de cette approche est qu'elle fonctionne pour des grilles de dimensions plus importantes (5x5, 6x6). Cependant, son point négatif est que le *Solver* ne trouve pas le chemin

le plus court, mais un chemin parmis d'autres afin d'arriver à la grille solutionnée (i.e triée).

La complexité de la fonction *get_ solution* est de l'ordre de $(m * n)^2$. Nous avons estimé cette complexité à partir des deux boucles *for* imbriquées allant jusqu'à m et n respectivement. On a ajouté à cela le nombre de déplacement maximal possible pour ordonner la grille qui est $m*n$ échanges. On peut alors remarquer que dès lors que la taille de la grille augmente, le nombre de calcul de l'algorithme augmente de façon exponentielle.

3 Séance 2 : Première version du BFS (breadth-first search)

On s'intéresse désormais à la recherche d'une solution de grille de longueur minimale. Pour cela, la méthode BFS permet de trouver le plus court chemin d'un graphe avec un parcours en largeur. Cela signifique que l'on va explorer un noeud, puis ces successeurs puis les successeurs des successeurs etc...

La compréhension de la méthode BFS est assez simple mais son implémentation pour le *swap puzzle* ne l'a pas forcément été car il fallait faire le lien entre le graphe et l'utilisation d'un dictionnaire. Dans un second temps, il a fallu comprendre comment à partir du dictionnaire, nous pouvions récupérer le chemin le plus court entre un noeud de départ et un noeud d'arrivé.

3.1 Implémentation de l'algorithme BFS sur la classe *Graph*

Dans cette section, on va détailler la façon d'implémenter la fonction *bfs* de la classe *Graph*.

La fonction *bfs* prend en argument un noeud de départ et un noeud d'arrivé. Dans cette méthodes, on va créer un nouveau dictionnaire qui aura pour clé un noeud et en valeur son noeud antécédant associé. On va aussi créer deux listes : la première *visited* sert à marquer les noeuds du graphe déjà visités pour ne pas les ajouter plusieurs fois au dictionnaire, et la deuxième *queue* sert de liste d'attente pour les noeuds qui n'ont pas encore été ajoutés au dictionnaire.

Une fois que le graphe contenu dans le dictionnaire *self.graph* a été parcouru, on a donc le dictionnaire *path* contenant les noeuds et leurs antécédents qui a été créé. On peut alors "remonter" ce dictionnaire afin d'en extraire le chemin le plus court. Le chemin le plus court est extrait sous réserve que le noeud final fasse partie des clés du dictionnaire *path*, c'est-à-dire que le noeud final doit admettre un antécédent.

Cette fonction a été testée dans le *main.py* sur les graphes en input en considérant différents cas possibles : le cas où le chemin existe et le cas où le chemin n'existe pas. On a également vérifié si les chemins sont de même longueur lorsque l'on inverse les noeuds de départ et d'arrivée.

3.2 Création des fonctions intermédiaires pour faire le BFS sur la classe *Grid*

Dans cette partie, le graphe est constitué de noeuds représentant les états de la grille. Deux noeuds sont reliés entre eux si les grilles sont voisines, *i.e* il existe un swap réalisable pour passer d'une grille à l'autre. Ainsi, nous devons poser des fonctions intermédaires afin de construire le graphe sous forme de dictionnaire, ce qui nous permettra de lui appliquer la méthode *bfs* de la classe *Graph* pour ordonner notre grille.

Tout d'abord, nous avons créé la fonction *all_ grid*. Cette méthode nous permet de créer toutes les configurations possibles de grilles à partir d'une grille donnée. Le nombre de configuration

possible d'une grille est *(m*n) !*. Cette fonction renvoie une liste contenant toutes les configurations de grilles possibles qui sera utilisée pour créer le dictionnaire dans la fonction *init_ dict*. Il est aussi intéressant de noter que la grille ordonnée est toujours le premier élément de cette liste.

Ensuite, nous avons créé la méthode *test_ grids_ adjacents* qui prend en variable deux grilles et renvoie un booléen : *True* si les deux grilles sont voisines, *i.e* si un échange permet de passer de l'une à l'autre, et *False* si les grilles ne sont pas voisines.

Nous avons aussi la fonction *init_ dict* qui permet de créer le dictionnaire correspondant au graphe sur lequel on va appliquer le BFS. Cette fonction renvoie un dictionnaire contenant l'ensemble des configurations possibles de la grille en clé et en valeur associée une liste vide dans laquelle on va ajouter les grilles voisines avec la méthode suivante pour compléter le dictionnaire. Nous avons "hasher" ces "listes de listes", correspondant aux grilles, en les transformant en "tuples de tuples" afin qu'elles puissent être ajouté en clé du dictionnaire. Nous avons trouvé les lignes de commandes grâce à des recherches internet.

Enfin, nous avons la fonction *add_ grids_ adjacents_ in_ dict* qui utilise les trois fonctions précédentes. Une fois que l'on a le dictionnaire initialisé créé avec la fonction *init_ dict*, on le complète en y ajoutant les grilles voisines. Pour cela, nous allons parcourir toutes les grilles dans la liste renvoyée par la fonction *all_ grid*, et avec la fonction *test_ grids_ adjacents*, nous testons deux à deux si les grilles sont adjacentes. Si c'est le cas, on "hashe" les grilles sous forme de tuples et on les ajoute l'une et l'autre en voisine (*i.e* en valeur) dans le dictionnaire. Ainsi, la fonction renvoie un dictionnaire complet qui représente le graphe associé au problème de la grille et que l'on va pouvoir appliquer à la méthode bfs de la classe *Graph*.

3.3 Mise en application du BFS sur la classe *Grid*

Grâce à toutes ces fonctions, nous avons pu implémenter le bfs via la fonction *bfs_ grids* sur une grille afin de trouver le chemin le plus court pour ordonner une grille. Dans cette fonction, il suffit de définir correctement le noeud de départ et celui d'arrivée, qui sont respectivement l'état actuel de la grille, *i.e self.state*, et la grille ordonnée. Ensuite, il nous suffit d'associer le dictionnaire, créé par la fonction *add_ grids_ adjacents_ in_ dict* qui correspond au graphe de la grille, à la variable *self.graph* de la classe *Graph*. Cela nous permet d'effectuer le *bfs* sur le bon dictionnaire afin de résoudre la grille. On va donc récupérer la liste renvoyée par la fonction *bfs*. On transforme ensuite les grilles sous forme de tuples en "liste de liste" afin d'obtenir un affichage plus parlant lorsque l'on teste dans le *main.py*. Enfin, la fonction *bfs_ grids* modifie la grille en parcourant la liste contenant le chemin pour ordonner la grille.

Petit problème : cette méthode fonctionnne uniquement sur des grilles de dimension *2*2* car celle-ci nécessite de créer l'ensemble du graphe qui contient $(m * n)!$ noeuds. Ainsi, dès que la taille de la grille augmente, le graphe, et donc le dictionnaire, devient très important. Les calculs sont alors très longs et n'aboutissent pas à une résolution dans un temps raisonnable. Cela peut même renvoyer une erreur dans le cas où le nombre de calcul pour parcourir le dictionnaire est infinement grand.

Concernant la complexité de cette fonction, on peut faire une estimation de l'ordre de $(m*n)!$. Cela correspond au "pire cas", celui où l'on doit passer par toutes les grilles possibles pour ordonner la grille.

4 Séance 3 : Seconde version du BFS (breadth-first search)

Dans un second temps, nous avons ensuite essayé de développer une version améliorée du BFS, en créant un dictionnaire contenant uniquement la partie "utile" du graphe, celle qui permet de résoudre la grille. L'objectif de ce second BFS est donc de réduire la taille du dictionnaire correspondant à notre graphe afin de permettre une résolution plus rapide. Nous avons également essayé d'optimiser et de réduire les fonctions intermédiaires, notamment celles permettant de trouver les grilles voisines à partir d'une grille donnée.

4.1 Création de nouvelles fonctions intermédiaires pour optimiser le BFS

Nous avons créé deux nouvelles fonctions intermédiaires afin de gagner du temps de calcul pour le bfs amélioré. Ces trois fonctions sont *generate_ordered_matrix*, *get_adjacent_grids* et *dico_upgrade_bfs*. Les deux premières fonctions nous permettent de ne pas utiliser la fonction *all_grid* et donc de pouvoir travailler sur des grilles plus grandes en limitant les calculs. La troisième fonction va créer le dictionnaire contenant uniquement la partie du graphe servant à résoudre la grille.

La fonction *generate_ordered_matrix* renvoie la grille ordonnée de taille *m*n*. Cela nous permet de disposer directement de la grille d'arrivée.

La fonction *get_adjacent_grids* prend une grille *g* en paramètre et renvoie une liste contenant les grilles voisines à la grille *g*. On évite de comparer toutes les grilles entre elles après les avoir toutes générées ici. On a donc un gain de temps de calcul considérable.

Enfin, la méthode *dico_upgrade_bfs* crée et renvoie le dictionnaire sur lequel on applique la fonction *bfs_grids_upgrade*, afin de résoudre notre problème de grille. Cette fonction reprend des idées évoquées précedemment. Tout d'abord, les listes sont "hasher" en "tuples de tuples" de manière à pouvoir être ajoutées en clé du dictionnaire que l'on va créer. Pour créer ce dictionnaire, on utlise les listes *queue* et *marquer* : la première pour mettre en liste d'attente les voisins du noeud que l'on traite, et la seconde pour enregistrer les noeuds déjà ajoutés au dictionnaire et ne pas les ajouter plusieurs fois. Au niveau de la procédure, on ajoute les voisins de la grille de départ dans le dictionnaire. Puis on répète une boucle qui s'arrêtera lorsque le noeud d'arrivée, *i.e* le noeud correspondant à la grille ordonnée, sera en clé du dictionnaire. Dans cette boucle, on marque les noeuds que l'on ajoute au dictionnaire et on génère ses voisins qu'on ajoute également au dictionnaire.

4.2 Mise en application du BFS amélioré sur la classe *Grid*

Afin d'avoir un BFS pouvant résoudre des grilles de tailles supérieures à *2*2*, il est nécessaire d'utiliser les fonctions décrites précedemment pour réduire le temps de calcul et disposer d'un dictionnaire, donc d'un graphe plus petit, en se restreignant à la partie du graphe utile pour résoudre notre problème de grille.

La fonction *bfs_grids_upgrade* correspond exactement à la fonction *bfs_grids* à l'exception du dictionnaire que l'on associe à la variable *self.graph* de la classe *Graph* qui est le dictionnaire provenant de la fonction *dico_upgrade_bfs*. Celui-ci est normalement d'une taille inférieure au dictionnaire précédent retourné par la fonction *add_grids_adjacents_in_dict*. Les dictionnaires

sont de même taille *si et seulement si* on ajoute la grille ordonnée dans le dictionnaire en dernier, *i.e* après avoir parcouru et ajouté tous les autres états de grilles possibles dans le dictionnaire renvoyé par la fonction *bfs_ grids_ upgrade*. C'est fortement improbable.

Cette fonction nous permet de résoudre maintenant des grilles allant jusqu'à la dimension *3*3* mais aussi des grilles de taille *2*4* par exemple. Le temps de résolution dépend de l'état dans lequel est la grille *3*3*. Autrement dit, si l'on peut l'ordonner en quelques coups seulement. Dans nos tests, on remarque qu'une grille *3*3* (et qui nécessite de nombreux échanges pour être ordonnée) met environ 20 secondes avant d'être résolue, mais une autre grille nécessitant moins d'échange peut se résoudre en quelques secondes.

Concernant la complexité de l'algorithme, on peut l'estimer inférieure à $(m * n)!$. Le seul cas où elle serait égale à $(m * n)!$ est le cas où le dictionnaire contiendrait toutes les grilles en clé, ce qui est improbable.

Mais cette méthode ne résoud pas les grilles de tailles supérieures ou égales *4*4* qui demande trop de calcul. Nous avons donc implémenté une méthode heuristique avec l'algorithme A*.

5 Séance 4 : Méthode heuristique avec l'algorithme A*

L'algorithme A* repose sur l'idée de comparer les noeuds en fonction d'une heuristique, c'est-à-dire une "distance", qui les séparent de l'arrivé afin d'explorer en priorité les noeuds ayant plus de potentiels pour arriver rapidement à l'état final. Les noeuds ayant le plus de potentiel sont ceux dont l'heuristique avec le noeud final, c'est-à-dire la grille ordonnée, est la plus faible.

On a donc dans un premier temps défini la fonction *heuristic* qui prend comme paramètre une grille *current_ state* et renvoie un entier correspondant à la "distance" séparant la grille actuelle de la grille triée (finale). Pour calculer cette distance, on compare la position de chaque numéro par rapport à sa position dans l'état final. On somme tous les écarts de position pour tous les nombres de la grills et on obtient la valeur correspondant à la "distance" entre la grille actuelle et finale.

Une fois l'heuristique calculée, on peut implémenter la fonction *solve* qui résoudra la grille avec la méthode A*. Dans ce programme, on utilise une file de priorité qui sera ordonnée en fonction des heuristiques des noeuds afin d'étudier les noeuds les plus prometteurs en priorité. Cette liste de priorité va contenir des tuples ayant 3 éléments (heuristique du noeud actuel, le noeud actuel, liste contenant le chemin entre la grille de départ et le noeud actuel).

L'algorithme va donc effetctuer une boucle en ajoutant les noeuds voisins au noeud étudié dans la file de priorité en fonction de leur heuristique. On ajoute ces noeuds dans la liste de priorité en les associant à leur heuristique et en complétant la liste contenant le chemin entre la grille de départ et le noeud étudié. Cette boucle s'arretera quand le noeud étudié sera le noeud d'arrivé, c'est-à-dire la grille ordonnée. Une fois le chemin trouvé, la fonction parcours la liste le contenant et modifie la grille afin de l'ordonner.

La complexité de cet algorithme est inférieure à celle de *bfs_ grids_ upgrade* car le programme fonctionne sur des grilles de tailles supérieures à *3*3* mais nous n'avons pas réussi à la calculer. On aurait également pu se renseigner sur d'autres heuristiques pour voir si elles auraient été plus efficaces mais celle-ci semble bien fonctionner car elle permet de résoudre des grilles *4*4* et *5*5*

6 Interface graphique du projet : Pygame (Séance 5)

Concernant l'interface graphique, nous avons commencé le code pygame sans pouvoir réellement le tester car nous n'arrivions pas à visualiser la grille sur Onyxia.

7 Ressources complémentaires utilisées

— Wikipédia (BFS, Algorithm de Dijkstra, Algorithme A*)
— Algorithms by Dasgupta, Papadimitriou, and Vazirani (Berkeley)
— Cours "Introduction aux méthodes heuristiques", Télécom Nancy, J.-F. Scheid