Projet de programmation

Adam BACHKIRI & Léo LEROY

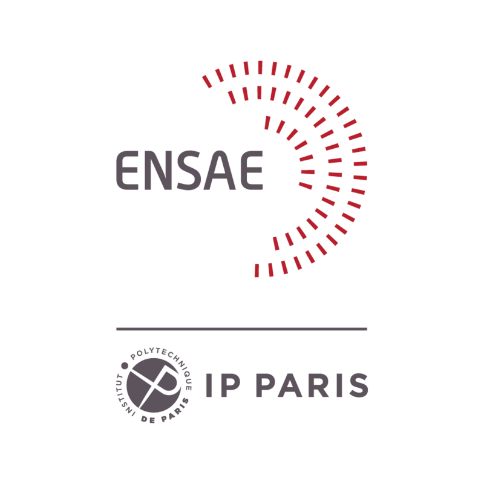


Table des matières

[I. Introduction 1](#_Toc160985640)

[II. Le fichier grid 1](#_Toc160985641)

[III. Le fichier graph 2](#_Toc160985642)

[IV. Le fichier solver 3](#_Toc160985643)

[V. Le fichier priorityQueue 3](#_Toc160985644)

[VI. Le fichier interactive 4](#_Toc160985645)

# Introduction

On considère une grille m×n, où m ≥ 1 et n ≥ 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 mn. 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 (i1, j1) et (i2, j2) si et seulement si i1 = i2 et |j1 − j2| = 1 ou j1 = j2 et |i1 − i2| = 1. On n’autorise pas les échanges par les bords extérieurs (entre la première et dernière ligne, ou la première et dernière colonne).

Outils utilisés : Nous avons utilisé le langage de programmation Python depuis l‘éditeur de code Visual Studio Code. Visual Studio Code nous permettait de travailler en collaboration en connectant notre travail à nous repositoire GitHub. Visual Studio Code nous était accessible depuis la plateforme cloud éphémère Onyxia, d’où la nécessité de stocker notre code sur GitHub.

Voici le lien du GitHub : <https://github.com/Mr-Dator/ensae-prog24>

# Le fichier grid

La première étape de ce projet consistait à créer une classe pour l’objet central de notre étude : les grilles. La formalisation des grilles par une classe permet de regrouper des attributs et des fonctions (appelée méthodes) à un concept unique (les grilles) : c’est l’encapsulation. L’avantage d’une classe par rapport à la définition de plusieurs fonctions indépendantes est qu’elle permet de maintenir l’état interne de l’objet et évite de passer constamment des paramètres entre les fonctions.

Nous avons utilisés les libraries *random*, *sys*, *hashlib* et *copy*.

Pour l’initialisation de la classe grille, nous définissons les dimensions m et n de la grille et on lui attribue une grille ordonnée par défaut si aucune autre grille n’est rentrée. La méthode *\_\_str\_\_* nous permet d’afficher l’état de la grille, c’est-à-dire son contenu, sous forme de texte. Cela s’avère particulièrement utile pour d’autres algorithmes ci-dessous. La méthode *grid\_from\_files* permet de créer une grille à partir d’un fichier text pour pouvoir effectuer nos tests sur des exemples donnés.

Pour la permutation de cellules, nous avons définis plusieurs méthodes. Tout d’abord, *swap* permet d’effectuer une permutation entre deux cellules spécifiées selon les conditions initiales (les seuls swaps autorisés sont ceux pour les cellules adjacentes). En cas de swap non autorisé, l’output est « Not allowed ». La méthode *swap\_seq* permet d’effectuer une séquence de permutation spécifiée à partir d’une liste donnée.

La méthode *is\_sorted* permet de vérifier si la grille est triée, l’output est True le cas échéant.

La méthode *hash* converti l’état de la grille en une chaine de caractères spécifiques afin de pouvoir effectuer des opérations de hashages. Ceci permet d’identifier une grille entière à un seul numéro ce qui rend la comparaison entre les grilles plus aisée. La méthode *unHash* permet de faire l’inverse.

Pour la gestion de l’aléatoire, nous avons créé trois méthodes. La méthode *randomSwap* permet de générer un swap aléatoirement. On choisit une paire (i,j) aléatoirement et une direction aléatoire pour effectuer le swap. Cette méthode est réutilisée ensuite via *gridScrambledness* et *gridDifficulty* : la première méthode permet de quantifier l’état de désordre dans la grille et la deuxième méthode permet de générer plusieurs swaps aléatoires afin d’obtenir le niveau de désordre souhaité selon le niveau de difficulté (1 : 40% des cases sont en désordre, 2 : entre 40% et 90% des cases sont en désordre, ou 3 : plus de 90% des cases sont en désordre). *gridScrambledness* calcule en effet le pourcentage de cellules qui sont en désordre dans une grille. L’utilisation d’un pourcentage plutôt que d’une quantité fixe de case en désordre permet de réutiliser l’algorithme pour des grilles de taille différente.

La méthode *gridNeighbors* retourne une liste des grilles voisines résultant d’une seule permutation.

Deux méthodes sont utilisées pour le calcul de norme. La méthode *cellNorm* calcule la norme d’une cellule par rapport à sa position initiale tandis que la méthode *gridNorm* calcule la norme totale de la grille en additionnant les normes de toutes les cellules. Le calcul de la norme d’une cellule consiste à faire la somme de la valeur absolue i moins sa valeur cible et la valeur absolue j moins sa valeur cible. Ces méthodes sont utiles pour l’implémentation des algorithmes de résolution automatique d’une grille.

# Le fichier graph

On définit une classe Graph qui prend en argument la liste des nœuds du graphe, un dictionnaire pour représenter les adjacences des nœuds, le nombre de nœuds et d’arrêtes totales, et la liste de toutes les arrêtes. Par défaut, le graphe obtenu est une liste vide. La méthode *\_\_str\_\_* permet d’afficher le graphe sous forme de liste de voisins pour chaque nœud. *\_\_repr\_\_* affiche une représentation du graphe avec le nombre de nœud et d’arrête. La méthodes *add\_edge* permet d’ajouter des arrêtes entre deux nœuds. *graph\_from\_files* permet de transformer en graphe un fichier text pour tester nos algorithmes.

Les trois dernières méthodes permettent de participer à la résolution automatique de la grille. La méthodes *bfs* permet de trouver le chemin le plus court entre deux nœuds en utilisant en parcours en largeur.

L’algorithme BFS (pour *Breadth-First Search*) procède à un parcourt en largeur du graphe. On procède ainsi : en partant d’un premier nœud (le départ), on ajoute tous ses voisins dans une file et on note le nœud actuel comme leur prédécesseur puis, pour chacun de ses voisins on recommence l’opération en ajoutant leurs voisins, s’ils n’ont pas déjà été visités dans la liste des nœuds à visiter. On peut visualiser l’exécution de cet algorithme comme l’avancée d’une frontière, s’étendant à chaque exécution de l’algorithme. Si le graphe est connexe, on finira par atteindre le nœud cible et il suffit alors de remonter la chaine des prédécesseurs, en partant de l’arrivée, pour obtenir le chemin désiré. Ici les nœuds sont les grilles et le graphe connecte les grilles selon leur similarités (en fonction de leur distance, d’après des normes définies).

La méthode *bfsOpti* modifie l’algorithme pour *bfs* en utilisant une optimisation, en effet au lieu de générer l’intégralité du graphe (ce qui est coûteux, d’autant plus qu’il contient (n\*m)! nœuds), elle utilise la méthode *gridNeighbors* issue de la classe *Grid* pour trouver les voisins d’un nœud dans le graphe et donc générer uniquement la partie “utile” du graphe.

Finalement, la méthode *aStar* permet une implémentation de l’algorithme A\* pour trouver le chemin le plus court entre deux nœuds dans le graphe. On utilise une file de priorité pour gérer les nœuds à explorer en fonction de la priorité. Pour cela on utilise la classe *PriorityQueue*. De plus cette méthode repose sur des heuristiques pour estimer la distance séparant les grilles de la grille résolue. On a ici utilisée un équivalent de la norme 1 pour les grilles mais une version pour une norme α quelconque a aussi été implémentée. L’algorithme est donc similaire à celui du parcours en largeur à la différence que l’on tient compte de la distance séparant les voisins de l’arrivée pour décider si oui ou non on explore un éventuel chemin.  
La norme 1 utilisée ici est calculée comme suit : Pour chaque cellule (i, j) on calcule les valeurs (iTarget, jTarget) de l’emplacement où devrait se trouver la cellule et on calcule alors la quantité ci,j=0,5\*(|i-iTarget| +|j-jTarget|). On somme cette quantité pour chaque cellule de la grille est on obtient alors un nombre qui sert de distance entre la grille et la grille rangée. Le calcul est similaire pour une norme α la distance ayant pour valeur

La file de priorité utilisée dans le code repose sur l’implémentation de la structure de tas proposée par le module “heapq” de python. Un tas est un arbre binaire presque complet à gauche ordonné. Autrement dit, il s’agit d’un arbre dont chaque nœud possède au plus 2 fils (binaire) et dont tous les niveaux sont remplis (complet) sauf éventuellement le dernier qui l’est partiellement depuis la gauche (presque complet à gauche). Cet arbre est de plus ordonné, c’est-à-dire que la valeur d’un nœud est toujours inférieure à celles de ses fils (dans le cas des tas implémenté avec “heapq”). On pourrait aussi vouloir que la valeur d’un nœud soit supérieure à celles de ses fils dans d’autres cas. Avec cette structure on peut obtenir l’élément de plus petite priorité (c’est-à-dire le plus proche) en retirant la racine de l’arbre puis en créant un nouveau tas à partir des 2 sous-arbres pour conserver la structure de tas.

# Le fichier solver

La classe *Solver* permet de résoudre la grille selon différents algorithmes.

Nous avons mis en place une résolution naïve de la grille. L’idée est de comparer la valeur de la cellule avec ses voisines pour effectuer un swap. Pour chaque cellule, on compare avec la cellule à gauche et en haut et on effectue un échange si nécessaire via la méthode *solveCell*. Sachant qu’un swap a une complexité en O(1) et que pour placer correctement une cellule on effectue O(m\*n) swaps, cette méthode a une complexité temporelle en O((m\*n)2)

Une version alternative de solution naïve a aussi été implémenté, elle procède en mettant au bon emplacement la cellule contenant la valeur 1, puis la valeur 2, .... jusqu’à la valeur m\*n-1. Sachant qu’un swap a une complexité en O(1) et que pour placer correctement une cellule on effectue O(m\*n) swaps, cette méthode a une complexité temporelle en O((m\*n)2)

La méthode *testSolution* permet de tester si une solution résout correctement la grille en créant une copie de la grille initiale et en appliquant la séquence de swaps associée (cf. Classe *Grid*). Finalement, on vérifie si la grille est triée en utilisant *is\_sorted*. Elle effectue m\*n test d’égalités en O(1). Cette méthode a donc une complexité en O(m\*n)

La méthode *getSolution* retourne la séquence de swaps qui a trié la grille.

L’algorithme BFS : on utilise la recherche en largeur (BFS). On crée un graphe représentant l’espace d’états possibles à partir de la grille initiale et on effectue une recherche BFS pour trouver la séquence de swaps menant à la grille triée. Dans le pire cas la complexité est en O(m\*n + (m\*n)!) = O((m\*n)!)

De même la méthode solveA renvoie la solution obtenue par la méthode aStar.  
Les méthodes du module “heapq” ayant été implémenté avec une complexité en O(log(n)) avec n la taille de la liste de priorité, l’algorithme A\* a une complexité temporelle en O(m\*n\*log((m\*n)!))

# Le fichier priorityQueue

C’est dans ce fichier qu’est implémentée les méthodes permettant d’utiliser la file de priorité. Le code repose en majorité sur le module *heapq* de python qui propose une implémentation des tas. On utilise la structure de tas implémentée par ce module et le fait que python puisse comparer les tuples pour implémenter les files de priorités. Pour cela on utilise des tas contenant des tuples de la forme (priorité, valeur). Cela combiné aux méthodes du module *heapq* permet d’obtenir une file de priorité fonctionnelle et avec une complexité raisonnable.

# Le fichier interactive

Le but de ce fichier est de fournir un affichage interactif pour jouer et résoudre une grille. Mais il condense aussi tout ce qui a été fait précédemment. On a utilisé la libraries *Pygame* pour représenter visuellement la grille. Ci-dessous une capture d’écran.

Une image contenant capture d’écran, texte, nombre, carré

Description générée automatiquement

Le programme commence par créer une instance de la classe *Grid* pour pouvoir afficher quelque chose en début de partie. Le programme détecte les clics de souris pour permettre à l’utilisateur de sélectionner des cellules à échanger. Les échanges sont effectués quand l’utilisateur sélectionne deux cellules. Un menu interactif est affiché à droite de la grille pour permettre à l’utilisateur de modifier certaines options :

* On peut régler la largeur et la longueur de la grille à partir des deux boutons en haut à droite. Un clic gauche augmente le nombre alors qu’un clic droit le diminue.
* La difficulté permet de désordonner la grille, de manière plus ou moins aléatoire selon le niveau de difficulté choisi.
* Enfin, le dernier bouton permet de résoudre la grille de manière automatique, avec BFS ou A\*.

La boucle principale du programme s’occupe des mises à jour continues de la fenêtre *Pygame*. La fonction *draw\_menu* est appelée pour afficher le menu et ses options. La fenêtre est rafraichie à une fréquence de 8 images par secondes (*clock.tick(8)*).

Pour rendre l’utilisation du programme plus intéressante nous avons aussi ajouté de multiples sons qui permettent à l’utilisateur d’avoir un retour sonore. Ainsi lorsqu’un swap est valide un son approprié est joué tandis que si le swap n’est pas autorisé un son d’erreur est joué. De la même manière un son de victoire est joué lorsque la grille est rangée et d’autres sons sont joués lorsque les différents boutons sont pressés. De plus une musique de fond est présente pour compléter le tout.

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement