ENSAE

Projet de programmation

Rapport final

Auteurs :
Avner EL BAZ
Stanislas Dekerle

10 mars 2024

Table des matières

Résumé

Le rapport ci-dessous détaille les différentes méthodes que nous avons implémentées pour trier une grille de nombres en une grille triée par ordre croissant. Nous avons d'abord mis en place une méthode "gloutonne", puis plusieurs méthodes basées sur le parcours d'un graphe associé à la grille. Enfin nous avons implémenté une interface de jeu pour résoudre ces grilles.

1 Présentation du problème

2	1
3	4

$\rightarrow$

1	2
3	4

FIGURE 1 – Résolution d'une grille 2x2

Le cadre d'étude est une grille de m lignes et n colonnes. Chaque case de cette grille contient l'un des nombres compris entre 1 et m*n. Les mouvements autorisés sont les échanges entre cases voisines.
Le problème consiste à trier cette grille pour que les cases contiennent les nombres par ordre croissant pour les lignes. On souhaite obtenir la plus courte liste d'échanges pour trier la liste.

2 Algorithme glouton

2.1 Principe théorique

La première méthode de résolution de la grille est une méthode naïve. On parcourt chaque case de la grille par ordre croissant. On commence par la première case. Si le nombre présent dans la case correspond à celui attendu, on passe à la suivante. Sinon, on localise la case où il se situe et on échange ces deux cases, en enchaînant plusieurs échanges consécutifs si besoin. Pour ne pas déranger les cases déjà parcourues, on aligne horizontalement la case du nombre avec sa destination, puis on échange verticalement les cases. Ainsi, la partie déjà parcourue est triée et on trie la partie à parcourir case à case.

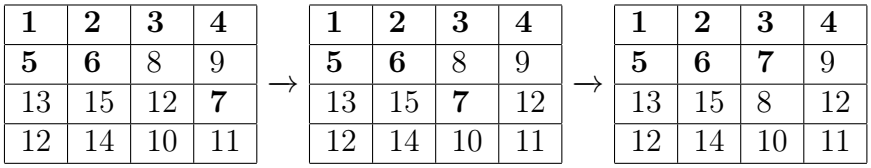

FIGURE 2 – Traitement de la case 7 avec la méthode gloutonne

Cette méthode est efficace mais effectue des échanges superflus. Dans l'exemple précédent, le *8* est éloigné de sa case. Il faudra ainsi plus d'échanges plus tard pour qu'il soit à sa case. Nous étudierons donc des méthodes plus rapides et plus efficaces par la suite.

2.2 Implémentation

2.2.1 Classe Grid

Pour implémenter cet algorithme, nous avons créé une classe ***grid***. Celle-ci est dotée des attributs suivants : deux variables numériques (*self.m* et *self.n*) contentant respectivement le nombre de lignes et le nombre de colonnes de la grille ; et une liste *self.state* contenant l'état de la grille sous la forme d'une liste de listes. Ces variables permettent de caractériser la grille.

Ensuite, nous avons implémenté les fonctions nécessaires à la manipulation de la grille. Ces fonctions sont les suivantes :

- swap : échanger les nombres de deux cases adjacentes
- test valid swap : vérifier la validité d'un échange lors de l'appel de la fonction *swap*
- move seq : créer la séquence d'échanges nécessaires pour échanger les nombres de deux cases non nécessairement adjacentes (en suivant la méthode détaillée plus tôt, voire figure 2)
- swap seq : exécuter une séquence d'échanges
- is sorted : vérifier si une grille est triée

2.2.2 Classe Solver

La résolution de la grille se fait à l'aide d'une classe ***solver*** dédiée. Celle-ci se construit à partir de la grille à résoudre. Ses attributs sont les suivants :

self.g est une copie de la grille, *self.state*, *self.m* et *self.n* correspondent aux variables homonymes de la classe *grid*.

Nous implémentons également certaines fonctions nécessaires à la résolution. Celles-ci sont les suivantes :

- find : trouver la case où se trouve le nombre correspondant à une certaine case
- fetch : utiliser la fonction *find* pour situer un nombre et utiliser la fonction *move seq* pour retourner la séquence d'échanges à effectuer pour rapatrier le nombre à sa case

Enfin, la fonction *get solution* permet de résoudre la grille en utilisant toutes ces fonctions. Cette fonction parcourt chaque case de la grille par ordre croissant. Pour chaque case, nous appelons la fonction *fetch* et nous obtenons la séquence d'échanges nécessaire pour rapatrier le nombre associé à cette case. Nous exécutons cette séquence sur la grille et nous ajoutons cette séquence à la liste *solution*. Avant de passer à la case suivante, nous utilisons la fonction *is sorted* pour nous assurer que la grille n'est pas triée pour pouvoir continuer. Si la grille est triée, nous sortons de la boucle et revoyons la liste *solution*.

Il est bon de noter que cette méthode de résolution a une compléxité élevée. En effet, la complexité de cet algorithme est en $O(m^2n^2)$. Ainsi, la solution "gloutonne" fonctionne mais est lente, inefficace et effectue un grand nombre d'échanges inutiles voire contreproductifs.

3 Parcours de graphe

3.1 Principe théorique et construction du graphe

Le problème peut être vu comme un parcours de graphe entre le noeud représenté par la grille de départ et le noeud représenté par la grille triée de même dimension. Concrètement, nous partons de la grille de départ. Nous lui associons un hachage non mutable unique et nous effectuons tous les échanges possibles sur cette grille. A chacune de ces grilles, nous associons un hachage et créons un noeud associé à cette grille et une arête entre chacun de ces nouveaux noeuds et le noeud de départ.

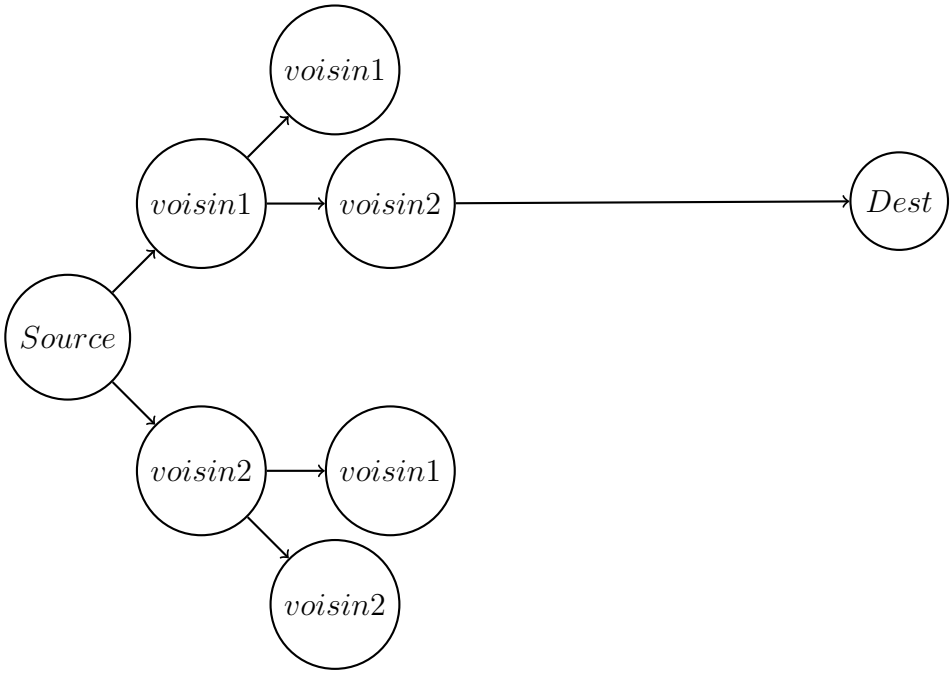

FIGURE 3 – Représentation sous forme de graphe

En supposant que nous gardons en mémoire l'échange reliant deux noeuds, la résolution du problème consiste à trouver le chemin le plus court entre le noeud de départ associé à la grille initiale et le noeud d'arrivée associé à la grille triée.
Ainsi, en associant un hachage unique à chaque grille, on peut construire un graphe connexe en reliant ensemble les noeuds partageant un échange valide.

3.2 Implémentation du graphe

L'implémentation du graphe se fait dans la classe **Graph**. Il est important de noter que nous travaillons sur la branche **alt graph**. Cette classe est munie des attributs suivants :

- nodes : liste des noeuds du graphe
- graph : dictionnaire contenant la liste d'adjacence de chaque noeud
- nb nodes : entier contenant le nombre de noeuds du graphe
- nb edges : entier contenant le nombre d'arêtes du graphe
- edges : liste des arêtes du graphe

— vertices : dictionnaire contenant l'échange nécessaire pour passer d'un noeud à un autre

Penchons-nous désormais sur les fonctions de cette classe. Pour construire le graphe, nous avons besoin de hacher chacune des grilles. Pour cela, nous implémentons la fonction *hash* dans la classe **Grid**. De plus, nous implémentons la fonction *add edge* qui permet d'ajouter rapidement une arête entre deux noeuds au graphe sans avoir à vérifier que ces noeuds sont dans le graphe ou si cette arête existe déjà.

La construction du graphe se fait à l'aide de la fonction *construct grid graph*. On lui fournit la grille initiale.
Si *self.graph* est vide, nous l'initialisons avec une liste vide pour la clé du hachage de la grille source. Nous fixons *nb nodes* à 1 et ajoutons le hachage de la grille source à *self.nodes*.
Nous initialisons une file d'attente à double extrémité avec la grille initiale. Tant qu'elle est vide, nous effectuons tous les échanges possibles sur le premier élément de la file d'attente. Nous obtenons les voisins de la grille. Nous la hashons, l'ajoutons à la file d'attente si son hashage ne figure pas dans *self.graph* et ajoutons les arêtes associées si elles ne figurent pas dans *self.edges*, et leur valeur dans *self.vertices*.

Ainsi, on obtient un graphe constitué de noeuds ayant pour identifiant le hachage de leur grille, un dictionnaire contenant les arêtes et un dictionnaire contenant les échanges reliant ces noeuds. Ce dernier dictionnaire est nécessaire car nous ne pouvons pas retrouver une grille à partir de son hachage et donc nous ne pouvons pas connaître l'échange liant deux noeuds liés.

3.3 Implémentation du parcours de graphe

3.3.1 Parcours en largeur

Le premier algorithme de parcours de graphe implémenté est un simple parcours en profondeur. Le principe est simple : nous commençons au noeud source, puis nous visitons tous ses voisins. Et pour chacun de ces voisins, nous visitons chacun de ses voisins. Nous répétons le processus jusqu'à atteindre le noeud de destination.

Nous implémentons cet algorithme dans la classe **Solver**. Nous construisons d'abord le graphe, puis nous effectuons le parcours.

Seulement, nous créons l'intégralité du graphe alors que nous pourrions nous arrêter lorsque la grille est résolue. Pour cela, nous combinons la fonction *construct grid graph* et la fonction *bfs* pour obtenir la fonction *construct grid graph bfs*. Ainsi, nous construisons le graphe à la volée et nous évitons de construire le graphe au-delà de la destination.

3.3.2 Algorithme A*

L'algorithme A* est similaire à l'algorithme précédent. La différence principale est le tri de la file d'attente à chaque ajout selon une heuristique. Pour cela, nous utilisons une structure de *heap* pour modifier la file d'attente tout en conservant une bonne complexité. L'heuristique que nous avons choisie est la distance de Manhattan.

Comme pour le parcours en largeur, nous avons d'abord implémenté la fonction *a star* utilisée en tandem avec *construct grid graph* puis la fonction *construct a star*, qui construit le graphe à la volée.

Cet algorithme est bien plus rapide et permet de prioriser les échanges les plus prometteurs. Ainsi, non seulement, nous nous arrêtons lorsque la grille est résolue, mais nous partons dans la bonne direction générale de résolution de la grille.

4 Interface graphique

4.1 Affichage de la grille

Pour l'affichage de la grille, nous avons créé la classe **Game**, héritée de la classe **Grid**, et utilisé le module *PyGame* comme suggéré. La grille est composée de différentes cases prenant en paramètres les valeurs de la grille. Plusieurs interfaces ont été implémentées par la suite pour améliorer l'expérience de l'utilisateur.

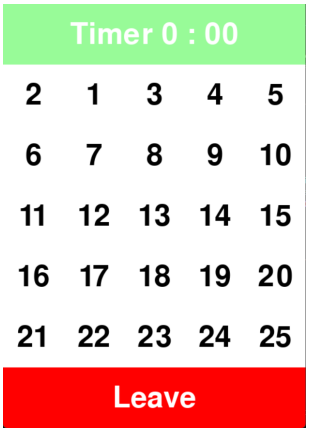

FIGURE 4 – Affichage pour une grille 5x5

Nous avons ainsi implémenté des affichages pour :
- L'accueil
- Le choix du niveau de la grille
- Le choix du niveau de jeu
- Le système de victoire et de défaite
- Le choix de recommencer

4.2 Sélection du niveau de la grille

Le choix du niveau de la grille est basé sur le calcul de son heuristique pour approximer, grâce à la distance de Manhattan, le minimum d'échange de case à effectuer pour trier la grille. La fonction *grid level* permet de créer de manière itérative une grille d'heuristique choisie en effectuant des échanges dans une grille triée initialement jusqu'à obtenir cette heuristique.

La fonction *choose level* donne la possibilité à l'utilisateur de choisir le niveau de la grille grâce à un champ de saisie.

4.3 Sélection du niveau du jeu

Le choix du niveau du jeu est basé sur le temps accordé à l'utilisateur pour effectuer un swap. La fonction *level grid* permet de créer une grille de manière itérative en effectuant des échanges dans une grille triée initialement jusqu'à

obtenir l'heuristique choisie. Le temps par échange est de 10 secondes pour le niveau facile à 0.5 secondes pour les plus chevronnés. Ce choix est donné à l'utilisateur grâce à la fonction *difficulty*. Un décompte est ensuite calculé comme la multiplication du niveau de la grille (nombres d'échanges réalisés par A*) et du temps accordé par échange.

4.4 Résultats

Si le joueur parvient à trier la grille avant la fin du décompte. La fonction *Results* informe au joueur de sa réussite. On lui indique par la suite le nombre optimal d'échanges et une résolution en temps réel de la grille grâce à la fonction *BestSol*. Enfin grâce à la fonction *retry* le joueur aura la possibilité de recommencer une partie.

4.5 Contraintes

Une contrainte que nous pouvons imposer à la résolution est l'interdiction d'effectuer certains échanges. Il s'agirait de placer des obstacles entre différentes cases. Ainsi, il nous suffit de modifier la fonction *test valid swap* pour inclure une condition de validité supplémentaire. Cette condition est contenue dans la variable *self.barriers* de la classe **Grid**.

Cependant, il est possible de rendre une grille impossible à résoudre en la séparant en plusieurs parties isolées par exemple. Dans ce cas-là, les algorithmes de résolution sont inefficaces et parcourent l'entièreté du graphe sans retourner la solution.

La résolution de ce problème a été traitée, par la fonction *valid barriers*, en testant pour une grille donnée et une liste d'échanges interdits, la possibilité de parcourir toutes les cases de cette grille grâce à l'algorithme de recherche en largeur. Il est important de préciser que cet algorithme est appliqué sur une grille et non sur un graphe construit comme précédemment. Cela permet d'avoir un test très rapide, car le fait de créer un graphe est extrêmement coûteux en termes de mémoire, et cela même en utilisant l'algorithme *a star*.