COMPTE RENDU DU PROJET DE PROGRAMMATION

Enzo MORAN et Alexis DAHLEN

SOMMAIRE :

Introduction

Le problème de réorganisation d'une grille m×n, où m ≥ 1 et n ≥ 2, pour aligner les carreaux dans un ordre prédéfini constitue un cas d'étude fascinant. Ce compte rendu de projet explore cette problématique et présente une approche algorithmique pour résoudre ce défi de manière efficace.

Le projet s'articule autour de la conception et de l'implémentation d'un algorithme de réorganisation de grille, ainsi que de son application pratique à travers une interface utilisateur conviviale.

L'objectif de cette introduction est de trouver la séquence de mouvements la plus courte pour ordonner les chiffres contenus dans la grille de telle sorte que la première ligne contienne les carreaux numérotés de 1 à n, la deuxième de n + 1 à 2n, et ainsi de suite. Nous examinerons les principaux aspects du problème, ainsi que les enjeux algorithmiques et techniques qui sous-tendent sa résolution.

Nous allons au cours de cette étude, considérer différents types d'algorithmes de résolution qui s'appliquent sur des graph. Pour pouvoir les utiliser afin de résoudre notre problématique, nous avons du déterminer un graph et ses nœuds. La direction que nous avons choisi est la suivante : le graph est constitué de l'ensemble des grilles atteignables grâce à des modifications autorisées de la grille rangée.

Analyse des différents algorithmes

Breadth-First Search :

L'algorithme de recherche "Breadth-First Search" (BFS) (ou "parcours en largeur" en français) est utilisé pour explorer et traverser efficacement une structure de données en suivant un parcours en largeur plutôt qu'en profondeur.
Dans le contexte des grilles et des graphes, BFS commence par explorer tous les voisins d'un nœud donné avant de passer aux voisins de ces voisins, et ainsi de suite. Cela signifie qu'il explore d'abord les nœuds les plus proches du nœud de départ avant de se déplacer vers les nœuds plus éloignés.

Pour appliquer un BFS a notre problématique nous avons décidé de prendre comme arguments un graph *self*, un nœud source *src* et un nœud destination *dst*. Les clés de *self* sont des grilles de type *Grid* et ses valeurs sont les listes des voisins de la clé associée.

Pour ce qui est du parcours du graph, nous utilisons une file d'attente *queue*. Prenons le premier élément de *queue* : *current*, s'il se trouve que *current* est en réalité *dst*, cela signifie qu'on a trouvé un chemin reliant *src* et *dst* et qu'on peut

directement sortir de la boucle *while*, sinon, on vérifie que celui-ci n'a pas déjà été visité ou mis dans *queue*, et si ce n'est pas le cas, on peut l'ajouter à queue et mémoriser *current* comme son antécédent.

Il est possible que l'algorithme se termine sans qu'on ait pour autant réussi à trouver un chemin reliant *src* et *dst. V*oilà pourquoi nous avons décidé d'introduire le booléen *found_path* qui prend la valeur *True* si un chemin a été trouvé et *False* sinon.

Après avoir vérifier que *found_path* est vrai, nous pouvons nous concentrer sur la reconstitution de la solution. Pour cela, nous avons décidé d'utiliser le dictionnaire *father* qui permet de garder en mémoire l'antécédent de chacun des nœuds. Toutefois, afin d'utiliser un dictionnaire comme mémoire, il était nécessaire d'utiliser des objets non mutable comme clés. Voilà pourquoi nous avons décidé d'introduire une fonction dans la classe Grid : *hash* qui permet de transformer une liste de liste (objet mutable) et tuple de tuple (objet non-mutable) afin de pouvoir utiliser nos grilles comme clés de *father*. Le dictionnaire *father* associe donc à un tuple de tuple (représentant une grille), la grid grâce à laquelle on est venue à l'observer (i.e. son *father*).

Il suffit alors de remonter d'antécédent en antécédent depuis *dst* jusqu'à obtenir *src* afin d'obtenir notre solution.

Déterminons maintenant la complexité de notre algorithme BFS.
L'initialisation a une complexité de O(1), car elle consiste principalement à initialiser des structures de données comme une file d'attente (*queue*) et un dictionnaire (*father*).
La boucle *while* itère jusqu'à ce que tous les nœuds accessibles aient été explorés ou jusqu'à ce que le nœud de destination soit trouvé. Dans le pire des cas, chaque nœud du graphe est visité une fois, ce qui donne une complexité de O((m*n)!) où m et n sont les dimensions de notre grille *dst*.
Enfin pour chaque nœud visité, tous ses voisins sont explorés pour déterminer s'ils ont été déjà visités ou s'ils sont déjà dans la file d'attente, cette opération prendra O(E) où E est le nombre d'arêtes du graph.

E = nb de nœuds * nb de voisins d'un nœud = (m*n)! * (m(n-1) + (m-1)n)

Donc O(E) = O(m*n*(m*n)!)
Finalement la complexité totale de notre programme est :

O((m*n)! + m*n*(m*n)!) = O(m*n*(m*n)!)

Nous ne considerons pas ici le coût associé à la création du graph qui serait en O(mn!)

Breadth-First Search Improved :

D'autre part, nous avons codé un algorithme *BFS_improved* qui intègre la création du graph à notre programme. On peut ainsi générer uniquement les nœuds qu'on veut explorer et non pas la totalité des nœuds comme on le faisait pour le BFS classique.

Pour cela nous avons implémenté une fonction *neighbors* qui prend en argument une G*rid* et renvoie une liste contenant toutes les *Grid* qui sont à un swap d'écart de la *Grid* initiale.

Pour ce qui est de la complexité de ce programme, lorsqu'on la calcule, on considère la « pire » des situations, dans ce cas bien précis, le *BFS_improved* doit générer tous les nœuds et devient équivalent au *BFS* classique. Les deux algorithmes ont donc théoriquement la même complexité mais en réalité, le *BFS_improved* est nettement plus rapide.

A* :

L'algorithme A* vise à parcourir intelligemment le graphe afin d'éviter un coût en termes de complexité trop élevé, voire de le minimiser autant que possible.

Pour ce faire, A* mesure le coût pour aller du nœud source jusqu'au nœud qu'il étudie, ainsi que celui pour aller de ce nœud jusqu'au nœud destination. Chaque nœud du graphe aura donc un coût qui lui est associé, selon le parcours du graphe. Nous commençons donc par étudier le coût de tous les voisins de notre source, puis nous entamons le parcours du graphe par le nœud dont le coût est le plus faible, et ainsi de suite.

Pour appliquer ce nouvel algorithme à notre problème, nous avons commencé par définir la distance que nous utiliserons pour mesurer le coût jusqu'à la grille source. Nous avons choisi de représenter cette distance par la distance de Manhattan. Nous avons attribué un coût de 1 pour passer d'un voisin à l'autre, étant donné qu'ils se trouvent à 1 swap l'un de l'autre. Afin de pouvoir réutiliser toutes les fonctions définies dans la classe Grid, nous avons décidé de prendre deux instances de cette classe comme arguments de la fonction A*, ainsi que le graphe.

Comme pour BFS, nous avons créé une file d'attente mais contrairement à BFS, chaque élément de cette file d'attente possède 4 arguments :

- Le nœud qu'il représente sous forme d'une grille.
- Le coût qu'il a fallu utiliser pour aller de la source jusqu'à celui-ci.
- Le coût total associé à ce nœud.
- Le père qui a généré ce nœud sous forme d'une grille.

A* permet d'explorer plusieurs chemins menant au même nœud, donc mettre le coût total est nécessaire afin de définir un chemin optimal. De même, comme le coût pour passer d'un nœud à l'autre est constant, définir le coût pour aller de la source à ce nœud permet de donner le coût suivant pour les nœuds qui seront générés à partir de celui-ci. Enfin, mettre le père du nœud permettra de remonter le chemin une fois le nœud destination trouvé.

Afin de parcourir le graphe de façon optimale, la file d'attente va être triée selon le coût total de chaque nœud, ce qui permettra de parcourir en priorité les nœuds les plus prometteurs.

Cependant, contrairement à BFS, nous pouvons retomber plusieurs fois sur le même nœud, mais en ayant parcouru deux chemins différents. Nous avons donc créé une disjonction de cas lors du traitement d'un nœud pour savoir si celui-ci avait déjà été visité par le passé :

- Si nous avions déjà visité ce nœud, nous comparons le coût total de ces deux nœuds et gardons uniquement celui dont le coût est le plus faible dans la *closed_list*. En effet, la closed_list contient les nœuds que nous avons déjà visités selon le chemin optimal pour les atteindre.
- S'il n'y en a pas, nous créons ses voisins, les ajoutons dans la file d'attente et mettons le nœud dans la *closed_list*.

Voici la correction :

Étant donné que tous les éléments sur lesquels nous travaillons sont des instances de la classe *Grid*, nous effectuons des comparaisons entre les éléments à l'aide de la fonction hash() ou de leur état.

Afin d'éviter la création d'une boucle infinie une fois la destination trouvée, notamment entre un père qui aurait un fils qui serait à son tour son père, nous avons décidé de ne pas recréer le nœud père lorsque nous générons les voisins du nœud sur lequel l'algorithme travaille. Pour cela, nous avons dû extraire l'information du père du nœud. Néanmoins, comme le nœud source n'a pas de père, nous avons dû créer une exception lorsque l'algorithme crée ses premiers voisins.

Comme pour BFS, nous avons implémenté un booléen *found_path* qui prend la valeur « True » si nous avons trouvé un chemin qui relie la source et la destination, ou « False » sinon. Si nous avons trouvé un chemin, nous remontons le graphe en passant à chaque fois par le père optimal grâce à la *closed_list*.

Complexité :

La fonction *neighbors* à une complexité de O(m*n)

Pour L'ajout et la suppression d'éléments dans les listes peuvent avoir une complexité de) O(log(p), ou p est le nombre d'éléments dans la liste des nœuds ouvert. Donc dans le pire des cas la complexité de l'ajout et la suppression d'éléments est de O(log((m*n)!)

Complexité des opérations de tri : Le tri des listes ouvertes peut avoir une complexité de O(p*log(p)) et donc au pire des cas O((m*n)!*log((m*n)!)

La complexité de la recherche de l'élément avec la plus petite heuristique dans la liste ouverte : La recherche de l'élément avec la plus petite heuristique dans la liste ouverte peut avoir une complexité de O(p) et donc au pire des cas de O((m*n)!)

Ainsi la complexité de l'algorithme A étoile est au pire de O(m*n*log((m*n)!))

Pygame :

Nous allons rapidement voir ce que nous avons fait dans pygame.

Le but ici était de faire un jeu interactif avec le joueur. Pour cela, nous avons commencé par définir la taille de la fenêtre du jeu ainsi que la méthode d'affichage de la grille sur laquelle nous jouons.

Après avoir défini cela, il fallait mettre en place les interactions du joueur avec la fenêtre. Le joueur peut soit arrêter de jouer soit effectuer des swaps. Nous avons donc commencé par créer des variables qui nous permettaient de connaître dans quel état était le jeu. Nous avons utilisé la variable *running* qui vaut *True* si nous sommes en train de jouer et *False* sinon. Comme pour la classe *Grid*, nous effectuons un swap dans la grille uniquement si ce swap est valable. Le joueur doit donc cliquer successivement sur deux cases où le swap est possible afin de l'effectuer.

Une fois tout ceci implémenté, il nous suffit de vérifier l'état de la grille afin de savoir si le joueur a réussi à trier la grille ou non. Une fois la grille triée, le jeu s'arrête et nous affichons l'écran de victoire.

Après avoir implémenté le jeu conformément aux recommandations, nous avons mis en place un niveau de difficulté. Plusieurs choix s'offraient à nous quant à ce qui déterminerait la difficulté d'une grille. Nous avons décidé de partir sur la taille de la grille comme critère de difficulté.