PROJET DE PROGRAMMATION

Réalisé par
Rudy Lysa, Neau Arthur

Projet de programmation réalisé dans le cadre d'un cours à l'ENSAE sous la responsabilité de Patrick Loiseau et Simon Mauras

ENSAE Paris
Mars 2024

I Introduction

Le problème initial concernait la résolution d'une grille en utilisant le langage de programmation Python. L'objectif était de réorganiser les cases d'une grille selon un ordre spécifique, en prenant comme entrée une grille initialement ordonnée de manière aléatoire. L'aspect crucial de cette tâche résidait dans la recherche de méthodes efficaces avec une complexité computationnelle minimale pour parvenir à cette réorganisation.

II Méthode naïve

L'intérêt de la méthode naïve est de trouver une méthode qui fonctionne tout le temps sans prendre en compte la complexité. Elle permet de s'assurer qu'au moins une méthode existe. Notre idée a donc été de placer les éléments les un après les autres dans l'ordre croissant. On place d'abord le 1, puis le 2 ... De ce fait, les cases déjà placées ne seront pas déplacées pendant les prochaines étapes. Le nombre de swaps de cette méthode dans le pire cas est

$$S_{n,m} = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1}(n-i+1)+(m-j+1) \ = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} i+j \ = \frac{m\cdot n\cdot(m+n-2)}{2}$$

En effet, dans le pire cas, le prochain numéro que l'on doit placer (i, j) est dans la case (n, m) (en bas à droite). Dans ce cas, il faut faire (n-i+1)+(m-j+1) swaps. De plus, la complexité obtenue est de l'ordre de O(n*m), ce qui indique une solution non-optimale. La principale limitation de cet algorithme réside dans sa complexité, ce qui restreint son utilisation pour des grilles de grande taille. En raison du temps de calcul, il n'est pas facilement praticable pour des grilles dont la taille dépasse 2*2. Nous sommes donc confrontés à la nécessité de trouver une approche plus efficace.

III Résolution par parcours d'un graphe

III.1 BFS simple

Grâce à un parcours dans un graphe, on peut trouver le plus court chemin entre la grille initiale et la grille finale. On utilise ici la méthode bfs (parcours en largeur) pour trouver le plus court chemin entre la source (src) et la destination (dst).
Pour atteindre cet objectif, il est nécessaire de créer le graphe contenant tous les états possibles. Cette opération nécessite l'implémentation de méthodes permettant de convertir entre des objets non hashables et des objets hashables. Ces méthodes, ‘to_hashable(self)‘ et ‘from_hashable(self, state)‘, sont essentielles dans ce processus. Les fonctions cruciales pour la réalisation du graphe sont ‘grille(self)‘ et ‘all(self)‘. Ces dernières sont responsables de la construction du

graphe complet, comprenant tous les états possibles. Ensuite, l'application de la recherche en largeur (BFS) 'bfs2(self)' nous fournit la séquence des grilles à visiter pour atteindre la destination (dst). Cette approche permet de naviguer dans le graphe en explorant les états de manière systématique et en trouvant le chemin optimal vers la destination.
Par rapport à l'algorithme naïf, cette méthode permet d'aller jusqu'à la grille 4*2 mais pas au-delà. Cependant, les perspectives d'évolution de cette méthode vont nous permettre de trouver des solutions pour de plus grandes grilles et plus rapidement (cf. II.2 et II.3).
Le nombre de nœuds dans le graphe est (m*n)!. En effet, dans la première case, on a m*n possibilités, puis m*n-1 dans la deuxième, ainsi de suite ... Pour calculer le nombre d'arêtes du graphe, il faut prendre en compte chaque case (intérieur, sur un bord ou dans un coin). Pour chaque arête à l'intérieur, on peut faire 4 mouvements, sur un bord, seulement 3 et dans un coin, 2. De plus, il y a 4 coins, 2*(m+n)-8 cases sur un bord et m*n-2*(m+n)+4 à l'intérieur.

$$\begin{aligned} Nb_{aretes} &= \frac{1}{2}(4 \cdot 2 + (2 \cdot (m+n) - 8) \cdot 3 + (m \cdot n - 2 \cdot (m+n) + 4) \cdot 4) \\ &= 2 \cdot m \cdot n - m - n \end{aligned}$$

Il ne faut pas oublier de diviser par 2 car on a compté toutes les arêtes 2 fois. En effet, s'il y a une arête entre A et B, alors cette arête va aussi de B vers A. Néanmoins, cette approche demeure peu efficace puisqu'elle exige la création intégrale du graphe avant son exploration.

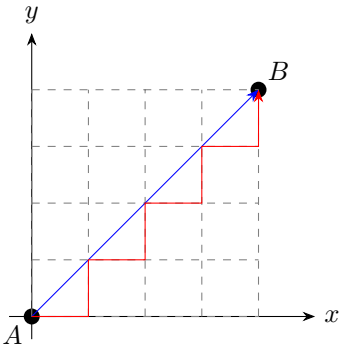
III.2 BFS amélioré

Plutôt que de créer le graphe dans son intégralité, même les sommets que le parcours ne visiterait jamais, on peut créer le graphe au fur et à mesure du parcours jusqu'à ce qu'on trouve la destination (dst) : 'bfs3 (self, dst)'. Dans le bfs simple, on utilisait la méthode 'all(self)' qui construisait le graphe entièrement à partir des permutations, et la recherche en largeur commençait après. Désormais, un dictionnaire est utilisé pour suivre les états visités au fur et à mesure de la recherche, et le graphe est construit dynamiquement pendant que la recherche progresse.
Cela nous permet d'aller plus loin dans la taille des grilles utilisées. Cette méthode est suffisamment performante pour calculer les swaps à faire pour les grilles jusqu'à la taille 3*3 et même la première grille 4*4 (grid3).

III.3 Algorithme A*

Toujours dans une recherche de performance et de rapidité de calcul, nous allons maintenant implémenter l'algorithme A*. Cette méthode fonctionne avec une fonction de coût, appelée heuristique. C'est une approximation par le calcul de la distance qu'il reste à parcourir dans le graphe pour atteindre la destination. Pour chaque nouveau voisin, l'algorithme compare les coûts liés à l'atteinte de

la destination et visite le voisin ayant le coût minimal.
Nous avons choisi de commencer par l'heuristique 'nombre de positions incorrectes' : heuristique(self). On somme le nombre de cases qui ne sont pas à leur place et on divise par 2 pour que l'heuristique soit un inf du nombre de swaps. En effet, avec un swap, on peut remettre 2 cases à leur bonne position.
Pour améliorer l'efficacité et la rapidité de l'algorithme, nous avons essayé 2 autres heuristiques heuristique1(self) et heuristique2(self) qui sont respectivement la distance de manhattan et la distance euclidienne.



Distance euclidienne

Distance de Manhattan

Toutes ces heuristiques servent donc dans la méthode 'bfs4(self,dst)' qui renvoie donc l'état final et le chemin pour y parvenir. Voici donc un tableau pour comparer la rapidité des heuristiques avec le calcul grâce au bfs4.

Grille	Positions incorrectes	Manhattan	Euclidienne
grid0.in	0.001	0.001	0.001
grid1.in	0.003	0.002	0.002
grid2.in	0.006	0.005	0.006
grid3.in	0.006	0.006	0.006
grid4.in	0.044	0.048	0.048
grid5.in	0.687	0.697	0.698

Table 1: Tableau comparatif des temps de calcul (en s)

On remarque donc que même pour les plus grandes grilles, le temps de calcul reste très faible. On peut donc conclure que l'algorithme A* est le plus performant de tous. Avec cette taille de grille, on ne peut pas juger de la rapidité des heuristiques.

IV Interface graphique

Nous avons donc réussi à trouver une méthode suffisamment efficace pour résoudre le puzzle. Ce qui semble ensuite intéressant est de permettre aux joueurs de comprendre l'intérêt du jeu. Pour ce faire, nous avons utilisé le module pygame pour faire apparaitre une interface graphique (photos ci dessous). Les joueurs peuvent donc déplacer les cases les unes après les autres pour compléter le jeu.

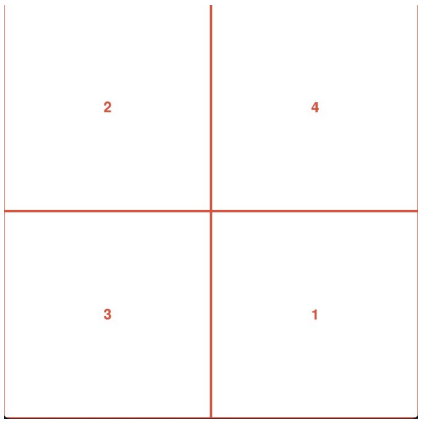


Figure 1: grid0.in

Figure 2: grid0.in résolue

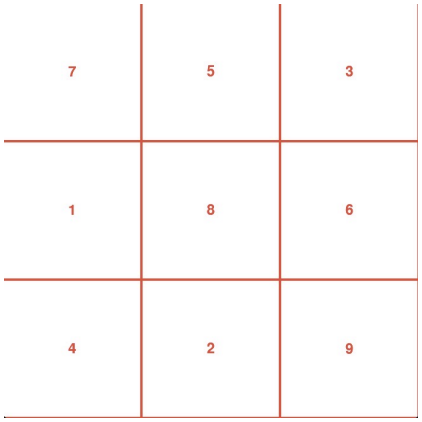


Figure 3: grid2.in

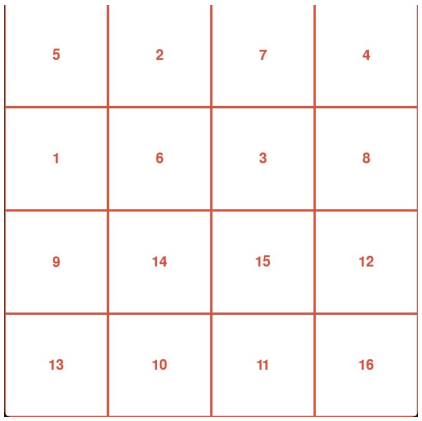


Figure 4: grid3.in

V Conclusion

La résolution du problème du swap puzzle a été abordé à travers différentes approches algorithmiques. Initialement, une méthode naïve a été mise en œuvre pour organiser les éléments de la grille, suivie d'une réflexion sur la représentation de la résolution comme un parcours de graphe. L'utilisation de l'algorithme de recherche BFS a été explorée pour trouver un chemin optimal entre l'état initial et l'état final de la grille. L'évolution naturelle du projet a conduit à la proposition d'une amélioration, l'algorithme A*, visant à optimiser la recherche de solutions en explorant des chemins plus prometteurs en priorité grâce à des heuristiques. Pour continuer cette démarche de recherche, nous aurions pu chercher d'autres jeux similaires et essayer de comparer les différentes méthodes utilisées ou encore optimiser l'interface utilisateur.