Rapport de notre projet de programmation

Malo David, Roman Bataille

Mars 2024

Résumé

Dans le présent document, nous nous intéressons à la résolution du *swap puzzle*. Dans un premier temps, nous présentons et expliquons le code que nous avons produit sur git pour résoudre ce problème (disponible à l'adresse suivante : `https://github.com/rbataille300/ensae-prog24`). Puis, nous calculons et discutons des complexités de nos différents algorithmes. Enfin, nous présentons brièvement chacune des méthodes et fonctions utilisées.

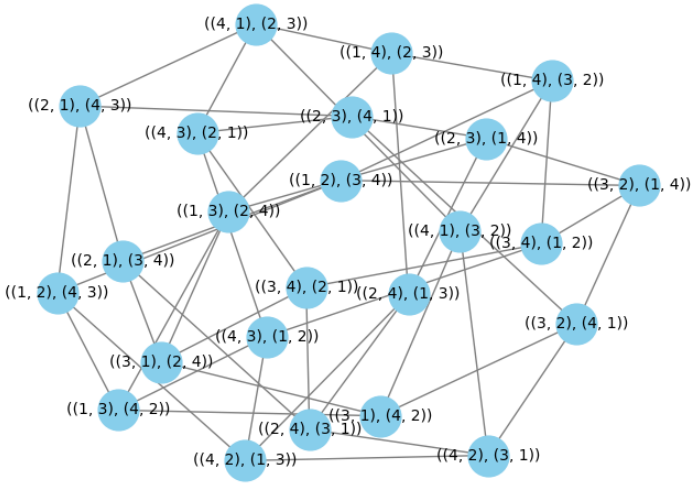

Table des matières

1 Explications de nos choix algorithmiques et compte-rendu

1.1 Implémentation des méthodes de base de la classe `Grid` (question 2)

L'implémentation des méthodes `swap`, `swap_seq` et `is_sorted` nous a dans un premier temps familiarisé avec la nature du problème donné. En effet, elle nous a permis d'acquérir une intuition quant à la problématique que la résolution du swap puzzle pose, ainsi qu'une compréhension du type des objets que l'on manipule : cellules, swap, grille etc.

Aussi, nous avons été ammené à traduire sous forme de condition (booléen dans Python) la licéité d'un swap : les swaps par dessus les bords de la grilles n'étant pas autorisés. À cet effet, nous avons utilisé la condition présentée et mise en contexte ci-dessous :

```
#this condition corresponds to a valid swap (one side apart + no border move)
(i1==i2 and abs(j1-j2)==1) or (j1==j2 and abs(i1-i2)== 1)
#when this condition is True, it means the swap is legit.
#in the method swap, we use it as follows :

def swap(self, cell1, cell2):
        etat = self.state
        #We proceed to define the line and column numbers of both cells.
        i1=cell1[0]
        j1=cell1[1]
        i2=cell2[0]
        j2=cell2[1]
        if (i1==i2 and abs(j1-j2)==1) or (j1==j2 and abs(i1-i2)== 1): #the condition
            etat[i1][j1], etat[i2][j2] = etat[i2][j2], etat[i1][j1]
        else:
            raise Exception("The swap isn't valid")
        return etat
```

FIGURE 1 – Explication de la condition de validité du swap

1.2 Implémentation d'une méthode de résolution naive (question 3)

Afin de résoudre naivement (mais à tous les coups) une grille, nous avons adopté la stratégie suivante : *à condition que la grille soit valide*, on considère le carreau n°1 (il y en a un total de $m \times n$ où n et m correspondent respectivement au nombre de colonnes et au nombre de lignes de la grille) et s'il n'est pas à la bonne place, on effectue des swaps horizontaux jusqu'à ce qu'il se trouve sur la bonne colonne, puis des swaps verticaux jusqu'à ce qu'il se trouve sur la bonne ligne. On passe ensuite au carreau suivant. L'algorithme prend fin lorsque l'on traite le dernier carreau. La grille est alors ordonnée.

À cet effet, nous sommes ammené à déterminer les coordonnées (`i0` et `j0`) de notre carreau s'il se trouvait dans la grille ordonnée (c'est à dire à sa position souhaitée). Pour cela, nous avons choisi d'effectuer une disjonction de cas, en fonction de si le carreau une fois à la bonne place touche ou non l'extrémité droite de la grille. En effet, une telle disjonction permet de déterminer `i0` et `j0` plus facilement (cf. Figure 2).

```
#The condition below is equivalent to the cell touching the right border
(element+1)%self.grid.n
#If it's True, it means that it touches the right border.

#Thus, the definition of i0 and j0 is as follows :
if (element+1)%self.grid.n != 0:
    (i0, j0) = ((element+1)//self.grid.n, ((element+1)%self.grid.n)-1)
else:
    (i0,j0) = ((element+1)//self.grid.n-1, self.grid.n-1)
```

FIGURE 2 – Explication de la méthode de détermination de `i0` et `j0`

Note 1.1. Il est important de remarquer que dans le code nous utilisons `element+1` car dans la méthode `get_solution`, `element` varie de 0 à $n \times m - 1$ à cause de son implémentation utilisant `range()`.

Note 1.2. Cette méthode n'est pas optimale, et nous l'avons montré en utilisant un exemple simple pour lequel la méthode renvoit une séquence de swap correcte mais plus longue qu'une solution que l'on peut trouver à la main. En voici un exemple :

Considérons la grille suivante : `[[2,3],[1,4]]`
Notre algorithme effectue 4 swaps et passe par le chemin suivant avant d'obtenir la grille ordonnée :
`[[2,3],[1,4]] -> [[1,3],[2,4]] -> [[1,3],[4,2]] -> [[1,2],[4,3]] -> [[1,2],[3,4]]`
Seulement, en effectuant deux swaps on obtient le chemin suivant plus court :
`[[2,3],[1,4]] -> [[3,2],[1,4]] -> [[1,2],[3,4]]`
Notre algorithme naif n'est donc pas optimal, comme attendu.

1.3 Implémentation d'une représentation graphique de la grille (question 4)

Afin de représenter graphiquement une grille, nous avons choisi d'utiliser la librairie `matplotlib.pyplot` car nous en connaissions la syntaxe. Nous avons dû tester plusieurs tailles de police et avons fini par choisir `fontsize=12` qui est un bon compromis entre lisibilité et facilité de représentation lorsque la grille possède des dimensions importantes. Nous avons choisi de représenter la grille en noir pour suivre la représentation adoptée dans la documentation du professeur.

Plus précisemment, nous avons décidé de placer les numéros des carreaux en des coordonnées entières, puis de dessiner les lignes horizontales et verticales en passant par des demi-entiers en utilisant la fonction `ax.plot()`.

Nous avons rencontré un problème avec la fonction `plt.show` car notre grille ne s'affichait pas. Nous avons donc décidé de sauvegarder en local notre représentation sous format png grâce à la fonction `plt.savefig`, ce qui a résolu le problème. Nous avons par ailleurs fait la même chose dans la méthode `plot_graph` (cf. `??`).

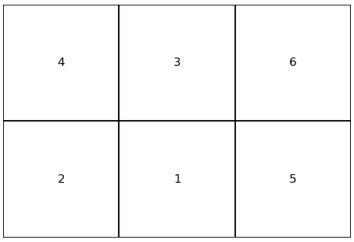

FIGURE 3 – Exemple de représentation graphique d'une grille non ordonnée de taille 2×3

1.4 Implémentation de l'algorithme BFS pour des graphes (question 5)

Dans notre implémentation de l'aglorithme BFS dans la classe `Graph`, nous avons utilisé des ensembles avec la fonction `set` ainsi qu'une queue à deux extrémités (`deque`) dont on se sert comme une file pour stocker les noeuds à explorer : à chaque itération, nous retirons un nœud de la file, explorons ses voisins non visités et les ajoutons à la file. L'algorithme BFS est utilisé pour parcourir le graphe en largeur, en explorant d'abord tous les nœuds à un niveau donné avant de passer au niveau suivant.

Nous avons testé la méthode `bfs` en utilisant les fichiers d'entrée `graph1.in` et les fichiers de sortie attendus `graph1.path.out`. Nous avons ainsi vérifié si les chemins retournés par notre implémentation correspondaient bien aux chemins attendus.

En implémentant cet algorithme, nous avons rapidement fait le lien avec le problème de la résolution du swap puzzle. En effet, en considérant le graphe dont les noeuds sont tous les états possibles de la grille, et dont les arrêtes relient les états tels qu'un swap valide permet de passer de l'un à l'autre, nous pourrions trouver la solution optimale en terme de nombre de swaps à coup sûr (cf. question 6 et 7).

1.5 Représentation des noeuds correspondants aux états de la grille (question 6)

Nous avons réfléchi à une représentation des nœuds correspondant à tous les états de la grille en utilisant des tuples de tuples. Les tuples sont immuables et donc hashables, ce qui est nécessaire pour les utiliser comme clés dans les ensembles et les dictionnaires en Python. De plus, cela garantissait l'immuabilité des états de la grille, ce qui est important pour les manipulations dans les algorithmes de recherche de chemin comme BFS et A*. La méthode `generate_possible_states` - qui génère tous les états possibles de la grille - repose sur la fonction `permutations` de la librairie `itertools`.

1.6 Implémentation d'une méthode qui trouve une solution de longueur optimale pour le swap puzzle (question 7)

Nous avons combiné les résultats des questions précédentes pour implémenter une méthode qui trouve une solution de longueur optimale pour le swap puzzle en construisant d'abord le graphe de tous les états possibles de la grille, puis en appliquant l'algorithme BFS sur ce graphe. Pour cela, nous avons utilisé des méthodes telles que `find_optimal_solution`, qui utilise BFS pour trouver la solution optimale, et `generate_possible_states`.

Nous avons comparé les résultats de cette méthode avec ceux de la méthode naïve, consistant à parcourir tous les états possibles de la grille. La méthode utilisant BFS sur le graphe est plus efficace en termes de temps et d'espace, car elle explore uniquement les états nécessaires pour atteindre la solution optimale.

Nous avons utilisé la librairie `matplotlib` ainsi que la librairie `networkx` pour représenter le graphe. Ce choix nous a permis d'implémenter la méthode `plot_graph` efficacement. Nous avons dû expérimenter afin de déterminer la taille des noeuds, l'épaisseur des arrêtes ainsi que la taille de la police idéales. Nous avons fait le choix de couleurs dont le contraste permet la lecture du graphe aisément.

Note 1.3. Dès que la dimension de la grille est trop importante, le graphe devient illisible.

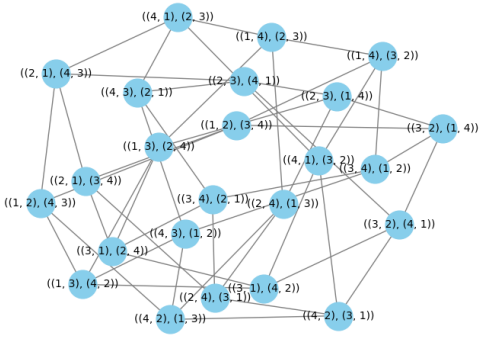

FIGURE 4 – Représentation du graphe pour une grille 2×2.

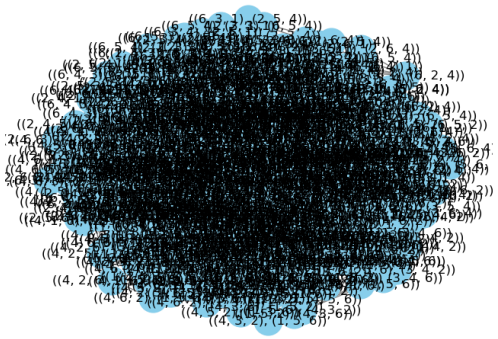

FIGURE 5 – Représentation du graphe pour une grille 2×3.

1.7 Implémentation de BFS spécifiquement pour le swap puzzle (question 8)

Nous avons proposé une nouvelle implémentation de BFS spécifiquement pour le swap puzzle afin de ne visiter que la partie du graphe nécessaire pour atteindre le nœud de destination, ce qui est particulièrement avantageux pour les grilles de grande taille. Cela permet d'économiser de l'espace mémoire en évitant de construire le graphe complet de tous les états possibles de la grille. Nous avons utilisé des méthodes telles que `get_neighbors`, qui génère uniquement les voisins valides d'un état donné, et `find_optimal_solution_2`, qui implémente BFS de manière à ne visiter que les nœuds nécessaires pour atteindre le nœud de destination.

1.8 Implémentation de l'algorithme A* pour la résolution du swap puzzle

Nous avons implémenté cet algorithme en utilisant une file de priorité (`heapq`) pour stocker les nœuds à explorer. Nous avons recherché des heuristiques pour améliorer l'efficacité de la recherche de la solution optimale (de longueur minimale). Nous avons implémenté des heuristiques telles que la distance de Manhattan, le nombre de carreaux mal placés et une combinaison linéaire d'heuristiques. Ces heuristiques ont été utilisées pour évaluer la distance entre l'état actuel et l'état cible, ce qui a permis à l'algorithme A* de prioriser les nœuds les plus prometteurs en termes de distance restante jusqu'à la solution.

Nous avons utilisé des méthodes telles que `heuristic_manhattan_distance`, `heuristic_misplaced_tiles` et `heuristic_linear_combination` pour calculer les heuristiques utilisées par l'algorithme A*. Nous avons également implémenté des méthodes telles que `find_optimal_solution_astar` pour appliquer l'algorithme A* à la résolution du swap puzzle.

L'algorithme A* permet de trouver une solution optimale (de longueur minimale) plus rapidement en général que BFS en explorant en priorité les chemins les plus prometteurs. L'utilisation d'heuristiques plus développées pourrait améliorer davantage les performances de l'algorithme en réduisant le nombre de nœuds à explorer.

1.9 Création d'une interface graphique et d'un niveau de difficulté contrôlé

Nous avons réussi à implémenter un jeu dont la difficulté est selectionnée par l'utilisateur au moyen de sa souris. Il est possible d'y choisir la dimension de la grille à résoudre. L'utilisateur choisit alors les swaps qu'il veut faire en cliquant sur les cellules correspondantes. Un mode permet de jouer avec des *barrières* (représentées en rouge) qui empêchent certains swaps. Lorsque Lorsque la grille est ordonnée, un écran de félicitation apparait.

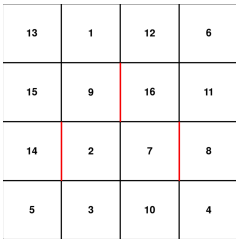

FIGURE 6 – Interface de jeu avec les barrières pour une grille 4×4

2 Calculs de complexité

2.1 Complexité de la méthode `generate_possible_states(self)`

La méthode génère toutes les permutations des positions des cellules dans la grille, ce qui donne une complexité en temps exponentielle en fonction du nombre de lignes m et de colonnes n.

$$O(m \times n \times m! \times n!)$$

2.2 Complexité de la méthode `find_optimal_solution(self)`

La méthode construit un graphe de tous les états possibles de la grille, où chaque nœud représente un état de la grille et chaque arête représente un échange valide. Ensuite, elle applique la recherche BFS (Breadth-First Search) sur ce graphe, ce qui donne une complexité exponentielle en fonction du nombre d'états possibles de la grille $(m \times n)!$ et de la longueur du chemin optimal $m \times n$).

$$O((m \times n)! \times (m \times n))$$

2.3 Complexité de la méthode `find_optimal_solution_2(self)`

La méthode est similaire à `find_optimal_solution`, mais elle stocke directement les états de grille dans le chemin plutôt que des paires d'états. Par conséquent, la complexité est la même que pour `find_optimal_solution`.

$$O((m \times n)! \times (m \times n))$$

2.4 Complexité de la méthode `find_optimal_solution_astar(self)`

La méthode applique l'algorithme A* au graphe des états possibles de la grille en utilisant l'heuristique de distance de Manhattan. La complexité en temps est dominée par la complexité de la recherche dans un graphe exponentiel, multipliée par le coût logarithmique de la manipulation de la file de priorité dans l'algorithme A*.

$$O((m \times n)! \times (m \times n) \times \log((m \times n)!))$$

3 Description générale de toutes les méthodes et fonctions

`swap_seq(self, cell_pair_list)` :
Effectue une série d'échanges basée sur la liste fournie de paires de cellules et met à jour l'état de la grille en conséquence.

`generate_possible_states(self)` :
Génère toutes les permutations des positions des cellules pour représenter les différents états de la grille et renvoie un ensemble contenant tous les états possibles.

`find_optimal_solution(self)` :
Construit un graphe où les nœuds représentent les états de grille possibles et les arêtes représentent les échanges valides et applique BFS pour trouver le chemin le plus court de l'état initial à l'état ordonné.

`state_after_swap(self, state, cell1, cell2)` :
Étant donné l'état actuel et deux positions de cellules, calcule le nouvel état après avoir échangé les valeurs des cellules.

`find_optimal_solution_2(self)` :
Similaire à `find_optimal_solution`, mais la représentation du chemin est modifiée pour stocker directement les états de grille au lieu de paires d'états.

`is_sorted(self, state)` :
Détermine si l'état de grille donné est dans l'ordre croissant.

`get_neighbors(self, state)` :
Renvoie une liste des états voisins qui peuvent être atteints en effectuant des échanges valides.

`heuristic_manhattan_distance(self, state, target_state)` :
Calcule la distance de Manhattan entre l'état actuel et l'état cible comme estimation du nombre de mouvements nécessaires pour atteindre la cible.

`find_optimal_solution_astar(self)` :
Applique l'algorithme A* pour trouver le chemin le plus court de l'état initial à l'état trié en utilisant l'heuristique de distance de Manhattan.

Et pour l'interface graphique, le jeu et les fonctions de base...

`shuffle_grid` :
Cette fonction prend en paramètre la taille de la grille et renvoie une grille mélangée de cette taille. Elle génère d'abord une liste de nombres de 1 à `size`^{2}, puis les mélange aléatoirement. Ensuite, elle divise cette liste en sous-listes de taille `size` pour former la grille mélangée.

`draw_grid` :
Cette fonction prend en paramètre la grille, la taille des cellules et la police pour afficher les nombres. Elle remplit l'écran en blanc, puis parcourt chaque cellule de la grille. Pour chaque cellule, elle dessine un rectangle et affiche le numéro à l'intérieur.

`swap_cells` :
Cette fonction prend en paramètre la grille et les coordonnées de deux cellules à échanger. Elle échange les valeurs des deux cellules dans la grille.

`main` :
Cette fonction prend en paramètre la taille de la grille et la taille des cellules. Elle initialise la police, mélange la grille, et initialise les variables de jeu. Elle exécute la boucle principale du jeu, qui gère les événements utilisateur (clics de souris) pour effectuer les mouvements. Une fois que la grille est ordonnée, elle affiche un écran de félicitations avec le nombre de mouvements effectués.

`start_game` :
Cette fonction prend en paramètre la difficulté sélectionnée par le joueur. En fonction de la difficulté, elle appelle la fonction main avec la taille de la grille appropriée.

`Fonction show_difficulty_select` :
Cette fonction affiche à l'écran les options de difficulté ("Facile", "Moyen", "Difficile"). Elle attend l'entrée de l'utilisateur (clic de souris). En fonction de la position du clic de souris, elle renvoie la difficulté sélectionnée.

`create_barriers(size)` :
Cette fonction choisit aléatoirement $n - 1$ barrières pour une grille $n \times n$.