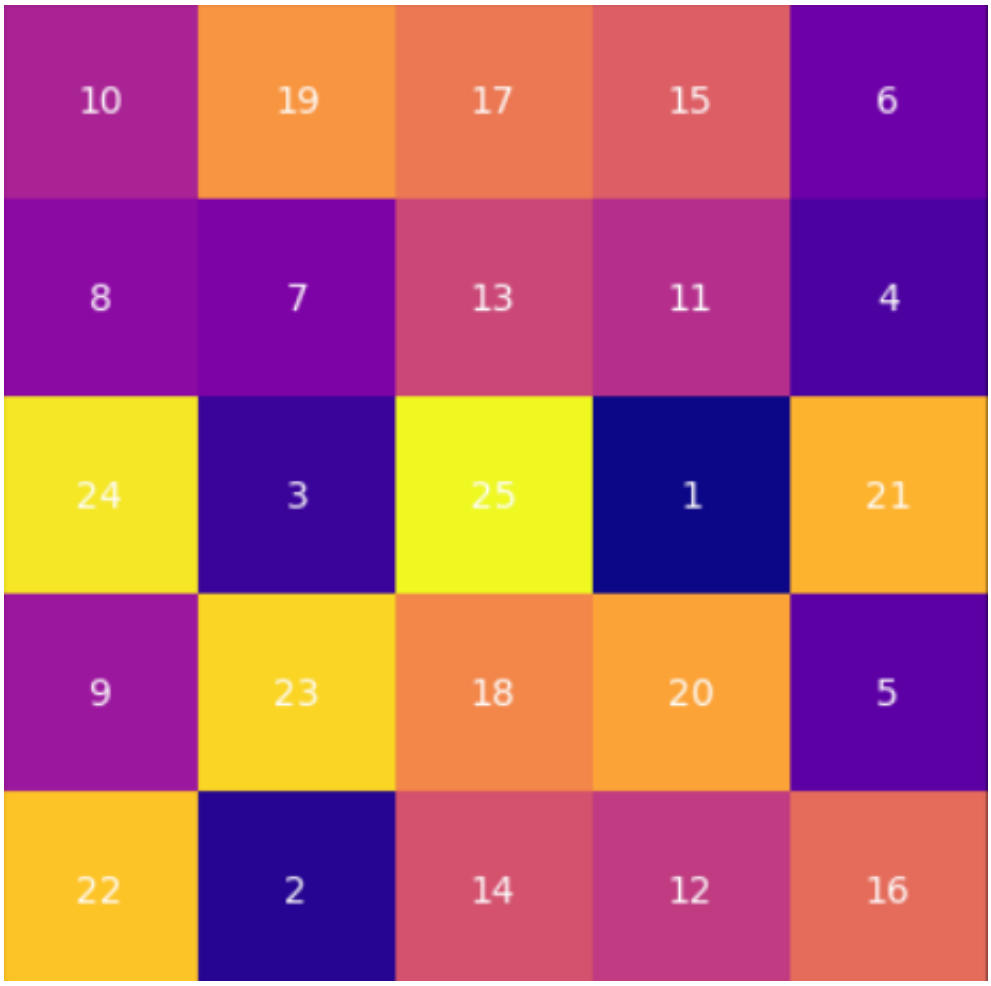
Projet de Programmation

Chardon-Denizot Lisa, Roche Antoine

Github



()

Contents

Contents I

1 Fonctions pythons 1

1.1 Fonctions de base 1

1.2 Graphes 2

2 Appendix 5

1 Fonctions pythons

1.1 Fonctions de base

1.1.1 `swap`, `swap_seq` et `is_sorted` :

`swap` est en $O(1)$, à noter que nous avons créé une exception particulière qui indique lorsqu'un coup n'est pas autorisé. `swap_seq` repose de manière directe sur swap et a une complexité en $O(|l|)$ où $|l|$ est la taille de la liste des swaps à effectuer. Enfin, pour `is_sorted`, on parcourt simplement toute la grid en comparant l'élément à son élément suivant dans la grid en prenant en compte le saut de ligne éventuel. Puisque nous parcourons au plus toute la grid, la complexité est en $O(mn)$ où m et n sont les dimensions (lignes, colonnes) de la grid (complexité linéaire).

Les tests de ces fonctions se situent dans le fichier `test_solver.py`

1.1.2 Résolution naïve de la grille :

Pour la résolution naïve de la grille, nous effectuons simplement une boucle sur les entiers entre 1 et mn et les plaçons au fur et à mesure à la place où ils devraient être dans une grille triée (les calculs reviennent à effectuer une divison euclidienne). A chaque fois que nous déplacons une tuile nous gardons en mémoire le swap effectué et renvoyons la liste finale de swap. Il convient de noter que nous sommes garanti de la correction de cette fonction car les swaps qui sont effectués par la suite ne vont pas "déranger" les tuiles qui ont déjà été bien rangées précédemment car les swaps vont de haut en bas et de la droite vers la gauche en rangeant les chiffres par ordre croissant. Finalement on obtient une complexité en $O((mn)^2(m+n))$, en effet, on doit trier au pire toutes les tuiles pour sortir de la boucle `while` et il y en a mn, à chaque fois que nous entrons dans le `while` les deux boucles `for` représentent mn opérations et dans les boucles `while` internes on fait au plus n swaps entre colonnes et m swaps entre lignes. Ce choix algorithmique s'explique intuitivement par le fait que c'est une approche naturelle que l'on pourrait prendre si l'on devait résoudre une telle grid. L'algorithme s'appliquant à toute grid indépendamment de ses caractéristiques, on peut toujours trouver une solution de la grid d'une longueur arbitraire. Enfin, l'algorithme ne renvoie pas la solution optimale en terme de longueur en général, Toutefois on peut fournir la borne supérieure (grossière) de $mn(m+n)$ swaps pour la taille des solutions qu'elle renvoie (elle doit trier au plus mn tuiles et pour chaque tuile la fonction effectue au plus $m+n$ swaps dans le cas extrême ou il faut remonter la tuile de toutes les lignes et toutes les colonnes).

La fonction est testée dans le fichier `test_solver.py`

1.1.3 Représentation graphique de la grille :

Malheureusement onyxia (plateforme de l'Insee) ne permet pas d'afficher avec matplotlib sur vscode. Nous avons toutefois testé la fonction (exactement la même en copiant collant le code) dans un Jupyter Notebook et elle fonctionne. Un exemple d'affichage de cette fonction est d'ailleurs utilisé en page de garde de ce rapport et en Appendix seront insérées plusieurs autres exemples d'affichage. A priori, la complexité de la fonction est un $O(mn)$ dûe à la double boucle `for`. L'aspect graphique du projet n'a pas été beaucoup plus approfondi dans la suite du projet.

1.2 Graphes

1.2.1 Fonction `bfs` :

Le bfs en lui même est en $O(|V|+|E|)$ où $|V|+E|$ est la taille du graphe que l'on parcourt. La version du bfs est modifiée légèrement ici avec la liste `prev` qui contient les parents des nouveaux noeuds que l'on découvre. Cette liste permet de reconstruire les plus courts chemins dans le graphe car on peut repartir du sommet d'arrivée, regarder son prédécesseur, puis regarder le prédécesseur du prédecésseur etc... A noter que l'ajout de cette liste `prev` à la fonction ne change aucunement la complexité en temps puisque la mise à jour des éléments d'une liste se fait ici en temps constant. Il n'y a pas de choix algorithmiques particuliers à évoquer ici, le bfs étant écrit en général de manière plus ou moins similaires partout. On peut cependant noter l'utilisation d'une liste python et donc l'utilisation de la fonction `append` pour mettre à jour la file de priorité, ce qui est critiquable au niveau complexité puisque à chaque ajout d'un élément dans la liste, python doit recalculer tous les indices des éléments (les décaler de 1) ce qui induit une complexité en linéaire en la taille de la queue supplémentaire. Il reste dans le code des fonctions telles que `graph_from_grid` et `permutations` qui ne servent plus mais permettaient de mettre en place la première version du BFS donc nous les avons laissées.

La fonction est testée dans le fichier `test_solver.py`

1.2.2 Fonctions auxiliaires du bfs :

Reconstruction du chemin :

Une fois que nous obtenons la liste `prev` à l'issue du bfs, il nous faut reconstruire le chemin dans le graphe, c'est ici qu'intervient la fonction `get_path` qui reconstruit la suite des sommets visités du sommet source au sommet destination et donc le plus court chemin. On obtient immédiatement une complexité en $O(|l|)$ où $|l|$ est la taille de la liste `prev` (dans le pire des cas ou il faut parcourir toute la liste), le `path.reverse` effectué à la fin étant lui aussi un $O(|l|)$

Hashage des états de la grid :

Comme nous n'étions pas vraiment familier avec Python, nous ne savions pas au moment où nous avons écrit notre fonction de hashage que les tuples python étaient hashables et donc que nous pouvions simplement utiliser un dictionnaire pour les hash. Ainsi nous avons manuellement écrit une fonction de hashage. Cela revient à trouver une injection f des états de la grid dans $\mathbb{N}$. Pour cela, nous avons décidé d'écrire les états de la grid en base $mn+1$. Ainsi la grille ligne valant 2-1-3 (m=1, n=3) a pour hash $2 * 4^0 + 1 * 4^1 + 3 * 4^3$, par l'unicité de l'écriture d'un nombre dans une base, l'application est bien une injection et on peut calculer le hash et sa fonction réciproque (avec des modulos) en temps linéaire soit $O(mn)$.

Calculs des voisins dans le graphe :

Pour construire le graphe au fur et à mesure nous avons décidé d'écrire une fonction qui à partir d'une grille calcule toutes les grilles "adjacentes". Ainsi `adj_grid` repose sur la fonction `nextperm` qui fait cela pour une case de la grid fixée. `nextperm` est en $O(mn)$ en raison de la copie de grid et de sa transformation en liste (toutes deux en $O(mn)$). Puisque `adj_grid` fait cela en pour tout $(i, j) \in [\![1, m]\!] \times [\![1, n]\!]$, on est en $O((mn)^2)$.

Complexité finale du BFS :

Malgré l'utilisation de toutes les fonctions décrites plus haut, la complexité du BFS pour trouver la suite des swaps pour résoudre la grille reste dominée par la complexité de la génération du graphe qui est en $O(mn(mn!))$. En effet, générer tout le graphe revient à générer toutes les permutations possibles de la grille et il y en a $mn!$, or le nombre d'arêtes créées est en $O(mn(mn!))$ car chaque état a toujours mn cases et chaque case peut être échangée des 4 côtés possibles donc chaque sommet est de degré au plus $4mn$ donc la somme de tous les degrés est au plus $4mn(mn)!$ or la somme des degrés de tous les sommets d'un graphe est égale au double de son nombre d'arête donc finalement le nombre d'arêtes total est bien un $O(mn(mn!))$. On gagne cependant en complexité en pratique car on ne génère pas forcément tout le graphe mais la complexité au pire est toujours la même.

1.2.3 Optimisation avec A* :

Nous avons compactifié le A* en y ajoutant directement la méthode `get_path` à l'intérieur. D'autre part, la logique est la même que pour le bfs où l'on génère le graphe, seul l'ajout de l'heuristique change. La complexité au pire reste toujours la même.

Heuristiques considérées :

- Distance pour la norme infinie sur la matrice associée à la grid (très peu efficace).
- Distance pour la norme 1 sur la matrice associée à la grid (la plus efficace pour nous).
- Nombre d'échange maximal à faire pour mettre une tuile au bon endroit (heuristique admissible qui nous garantit d'ailleurs d'optenir un plus court chemin).
- Heuristique donnée par la différence des hash (plus un test qu'autre chose et très très peu efficace).

1.2.4 Création d'une grille de niveau de difficulté contrôlée :

Nous avons fini par ajouter une dernière fonction permettant de générer de manière aléatoire une grid dont la difficulté de résolution est soit 1 - facile, 2 - moyenne ou 3 - difficile, il est difficile de réellement évaluer la difficulté de résolution d'une grille mais nous avons opté pour le choix de partir d'une grille de base aléatoire puis de choisir des tuiles au hasard de cette grille et de les permuter. En difficulté facile nous faisons 3 échanges, en moyenne 5 et en difficile 10. Cette approche fonctionne pour des grilles de plus petite taille (entre 3×3 et 5×5) mais devient moins intéressantes pour des grilles très grandes. Enfin, cette fonction a une complexité en $O(mn)$ car on génère tout d'abord la grille de taille mn triée.

1.2.5 Cas particulier et algorithmes :

Cas 1 $\times$ n :

Cela revient à trier une liste en n'échangeant que les cases adjacentes, on peut alors faire un tri par insertion qui convient car on n'échange que avec des cases adjacentes et on obtient une solution en $O(n^2) = O((mn)^2)$ dans ce cas particulier ce qui est plus efficace que la méthode naïve.

2 Appendix

Puisque nous avons utilisé dans le code des fonctions Python de base telles que `sum` et `list.copy` nous donnons ici comment nous aurions pu les implémenter nous même :

2.0.1 Code supplémentaire :

```
def sum(l):
    s = 0
    for e in l:
        s += e
    return e

def list.copy(l):
    n = len(l)
    l_ = [0 for i in range(n)]
    for i in range(n):
        l_[i] = l[i]
    return l_
```

2.0.2 Affichage avec la fonction `show_grid` :