Rapport du projet de programmation

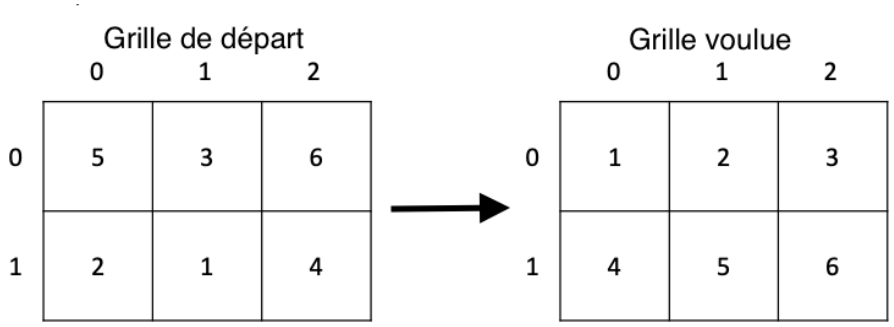
Lien du projet Github : https://github.com/atbrt/projet_programmation

1. Introduction : Présentation du problème du swap puzzle

A. Description du problème

Dans le cadre de ce projet, nous nous intéressons au problème du swap puzzle. Imaginons une grille composée de m lignes et n colonnes, où m est supérieur ou égal à 1 et n est supérieur ou égal à 2. Chaque case de cette grille est occupée par un carreau numéroté de 1 à mn. Au début, ces carreaux sont disposés de manière aléatoire dans la grille, créant un désordre total. Notre objectif est alors de trouver la séquence la plus courte de mouvements pour les aligner en ligne. En d'autres termes, nous devons obtenir une disposition où la première ligne contient les nombres de 1 à n, la deuxième de n + 1 à 2n, et ainsi de suite.

Les règles du puzzle sont plutôt simples. Les seuls mouvements autorisés sont des échanges de carreaux consécutifs (swaps), soit verticalement soit horizontalement. Ces échanges sont permis uniquement entre des carreaux adjacents, ce qui signifie qu'il est possible d'échanger deux carreaux si et seulement si leur position est adjacente sur la grille dans le sens vertical ou horizontal. Toutefois, il est important de noter que les échanges ne sont pas autorisés entre les carreaux situés sur les bords extérieurs de la grille.



B. Applications du problème

La résolution de ce puzzle présente plusieurs aspects qui sont pertinents en intelligence artificielle (IA) et en recherche opérationnelle.

En IA, ce type de problème sert souvent de cas d'étude pour développer et tester des algorithmes de recherche et d'optimisation. Par exemple, comme on le verra plus tard, la recherche de la séquence de swaps la plus courte pour réorganiser la grille peut être abordée avec diverses techniques comme la recherche en profondeur, la recherche en largeur, ou encore l'algorithme A*.

Par ailleurs, la résolution de ce puzzle peut également être appliquée dans le domaine de la recherche opérationnelle pour modéliser et résoudre des problèmes de planification. Par exemple, la grille de carreaux

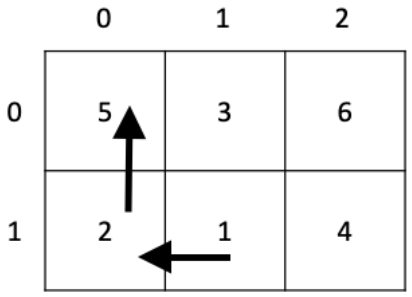
peut être vue comme une représentation abstraite d'un problème de logistique où les carreaux représentent des tâches à accomplir et les mouvements correspondent aux opérations nécessaires pour les réaliser dans un ordre spécifique. En optimisant la séquence de mouvements pour réorganiser la grille, on peut appliquer ces principes à des problèmes concrets de gestion de la chaîne d'approvisionnement, d'ordonnancement de production ou de routage dans des réseaux logistiques [1].

2. Méthodologie algorithmique

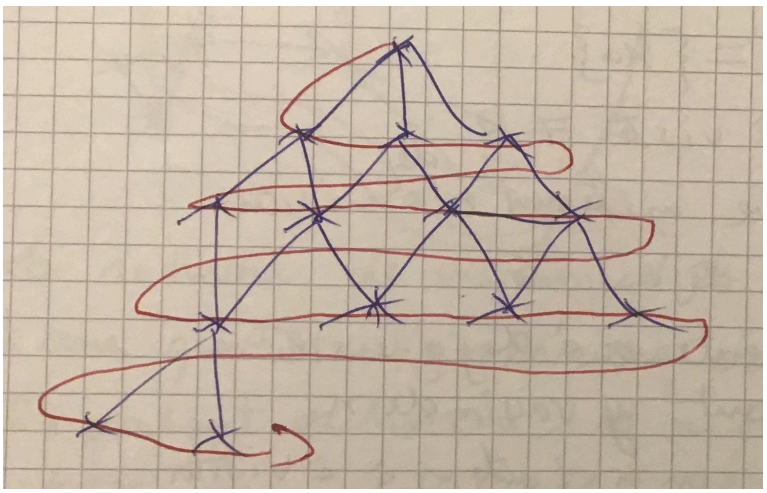
A. Description des méthodes algorithmiques

Afin de résoudre ce problème, le sujet a choisi de nous guider, en partant des solutions les plus évidentes aux solutions les plus complexes, tout en améliorant la qualité de la solution.

Pa Â La première méthode consiste en une méthode naïve : nous souhaitons ordonner une grille de nombre, et un des premiers reflexes lorsqu'on se trouve devant ce puzzle, c'est d'ordonner la grille en commençant par mettre le 1 à sa place, puis le 2, etc ... Cette méthode, bien que non optimale, nous assure d'arriver à la solution, car le puzzle est ordonné du haut vers le bas et de gauche à droite, donc lorsque nous mettons le nombre à sa bonne colonne et qu'on le fait remonter jusqu'à sa bonne ligne, nous sommes assurés que cela ne viendra pas perturber les numéros déjà bien placés.



La seconde méthode proposée par l'énoncé utilise l'algorithme BFS (*Breadth-First Search,* parcours en largeur en français). Pour cela, il faut visualiser notre problème comme un graphe, dans lequel nous partons d'un certain nœud et nous voulons arriver à un autre nœud, celui qui correspond à la grille ordonnée. Les nœuds représentent donc les différentes grilles possibles, et les voisins d'un certain nœud sont les grilles qui peuvent être obtenues à partir de notre nœud par un unique swap. Dès lors, le parcours en largeur semble plausible pour trouver la solution à notre problème : on commence par explorer un nœud source, ici notre grille de départ, puis ses successeurs, puis les successeurs non explorés des successeurs, etc... Nous utilisons donc une file dans laquelle est placé le premier sommet puis ses voisins non encore explorés. Les nœuds déjà visités sont retenus en mémoire afin d'éviter qu'un même nœud soit exploré plusieurs fois.



Le principe de fonctionnement d'une file s'appuie sur le principe du premier arrivé, premier sorti (FIFO : First In, First Out) : les nouveaux éléments à traiter sont stockés en fin de file et ceux qui vont être traités sont en début de file. C'est le même principe qu'une file d'attente à la caisse d'un supermarché.

Dans un premier temps, nous avons donc créé le graphe de toutes les grilles possibles, puis l'avons parcouru en utilisant l'algorithme BFS. Pour trouver la solution, il nous suffit alors de remonter les "parents" de chaque grille, c'est-à-dire la grille que l'on a parcouru avant la grille où l'on se situe, afin de remonter, depuis la grille ordonnée, à la grille source.

Néanmoins, cette méthode n'est pas optimale, car elle requiert de retenir en mémoire le graphe de toutes les grilles possibles, ce qui est très coûteux en espace. Dès lors, cette méthode ne fonctionne pas sur des grilles au-delà de la taille 3x3.

Cependant, pour contourner ce problème, nous avons modifié l'algorithme BFS, de telle manière que l'on construit le graphe au-fur-et-à-mesure qu'on le parcourt : en effet, cela nous permet une grande amélioration, car notre algorithme BFS s'arrête dès que nous nous situons sur la grille ordonnée, donc seulement une infime partir du graphe est parcourue. Ainsi, pour chaque grille, nous ne nous intéressons uniquement à ses voisins.

Mais le problème de l'algorithme BFS est que l'on parcourt "naïvement" les voisins des grilles, sans essayer de chercher quel voisin nous permettrait d'arriver plus rapidement à destination, ce qui fait que la solution n'est toujours pas optimale. En effet, nous parcourons le graphe en cercles concentriques autour du noeud source : d'abord les noeud à distance 1, puis ceux à distance 2, etc.. Pour résoudre ce problème, nous avons donc utilisé un algorithme A*, qui utilise une heuristique. Une heuristique est une première approximation de la distance à la destination : elle pourrait être imaginée comme une distance à vol d'oiseau à la destination. Nous allons ainsi préférer dans notre parcours du graphe les noeud à heuristique basse par rapport à ceux qui sont à heuristique haute. Nous construisons alors le chemin le plus court de la grille de départ à la grille d'arrivée, en construisant le chemin de proche en proche, de manière à minimiser la "distance" à l'arrivée à chaque fois. Cette méthode, à condition de choisir une heuristique qui fonctionne, permet d'arriver à trouver la solution du problème de manière quasi-optimale.

B. Description de l'implémentation

Pour implémenter l'algorithme de la méthode naïve, nous avons codé une fonction swap qui permet d'échanger deux cases, et une fonction qui permet de trouver, pour chaque élément de la grille, ses coordonnées actuellement, et une autre fonction qui calcule ses coordonnées dans la grille ordonnée. Ainsi, pour chaque numéro dans l'ordre croissant, on regarde s'il est trop à gauche ou à droite en numéro de colonne, puis on le swap pour qu'il soit sur la bonne colonne, et enfin on le remonte vers sa position voulue, le tout en utilisant la fonction swap codée précédemment.

Pour implémenter BFS, on l'implémente dans la classe Graph. Premièrement, si la grille est déjà celle qu'on veut, alors aucun swap n'est à effectuer. On utilise ensuite deux listes, une pour marquer ceux qu'on a visité et une qui contient ceux qu'il nous reste à visiter, ainsi qu'un dictionnaire de parents (auquel on ajoute la valeur None à la clé source, afin de pouvoir la repérer quand on remontera les parents ensuite), qui nous permettra de remonter à la solution. Nous bouclons ensuite avec un while, tant que la file des sommets à visiter est non vide, et on prend le premier élément de la file à chaque fois, et on regarde ses voisins. Pour chaque voisin qui n'a pas été encore visité, on l'ajoute à la fin de la file, on l'ajoute à la liste des noeuds visités (afin de ne pas le remettre ensuite dans les parents) et on ajoute son parent dans le dictionnaire. Enfin, on arrête la boucle si le noeud sur lequel on se situe est celui voulu. Dès lors, on remonte les parents on partant de la grille d'arrivée, et on remonte progressivement le dictionnaire tant qu'on n'a pas atteint la valeur None, c'est à dire le noeud de départ. Ensuite, on inverse la liste afin de mettre le chemin des noeuds visités à l'endroit.

Afin de pouvoir l'utiliser sur les grilles de nombres, nous avons donc dû construire le graphe de tous les sommets possibles. Premièrement, nous avons créé une fonction pour transformer les grilles en objets hashable afin de pouvoir les utiliser dans les dictionnaires. Comme il peut y avoir des nombres à deux chiffres, nous avons choisi d'utiliser des chaînes de caractère, avec un "n" entre chaque nombre, et un "/" pour repérer les fins de ligne, à l'aide de deux boucles for. Ensuite, nous avons créé une fonction qui permet de créer un objet de classe Grid depuis une liste, en utilisant un compteur qui compte l'indice où l'on est dans la liste, et deux boucles for, qui rajoutent des listes à une liste définie précédemment à chaque fin d'exécution de la deuxième boucle, afin de transformer la liste en tableau.

Concernant la construction du graphe des sommets possibles, nous avons créé toutes les permutations possible des entiers de 1 à mn, en utilisant un algorithme récursif qui est initialisé à la liste à 1 élément qui renvoie le tableau avec uniquement cet élément. On utilise ensuite un appel récursif, donc à la fin des appels récursifs, on se sera retrouvé dans le cas de l'initialisation, où Lp contient une liste de la forme [[e]], puis on case l'élément précédent avant ou après, cela donne une liste de deux permutations, à laquelle on va encore faire des permutations en ajoutant l'élément précédent dans ces listes, etc. Au final, on arrivera à des tailles de permutations de len(E), et en construisant au fur et à mesure par récursivité les permutations en augmentant leur taille et en casant à chaque fois les éléments dans tous les emplacements possibles, on aura obtenu la liste des permutations.

On crée dès lors une fonction qui permet de transformer toutes les grilles possibles en objet de classe Grid, en utilisant la fonction qui permet d'obtenir les permutations possibles sur la liste de 1 à mn, et en utilisant la fonction de liste à grid. Ensuite, on crée une fonction qui permet de savoir si deux grilles sont

accessibles depuis un swap, en utilisant deux boucles for, et en regardant les positions des éléments qui sont différents entre les deux grilles, puis en regardant si ces positions sont des cases adjacentes.

Nous pouvons enfin créer le graphe des grilles, en utilisant toutes les grilles possibles, et en les parcourant deux à deux à l'aide de deux boucles for, et si deux grilles sont liées par un swap, alors on les ajoute dans un dictionnaire qui stocke pour chaque grille, toutes ses grilles voisines, hashable grâce à la fonction de hashage. On crée la grille que l'on souhaite grâce à une fonction qui transforme la liste des entiers de 1 à mn en objet de type grid grâce à "de liste à grid". On utilise enfin le BFS déjà construit, en ajoutant les arêtes entre les noeuds en parcourant le dictionnaire du graphe. On obtient ainsi un objet de type graphe, donc on peut utiliser BFS, et on renvoie les swaps grâce à une fonction qui permet d'acquérir le swap en regardant les positions des cases différentes entre les deux grilles et en les ajoutant dans un tuple.

Pour implémenter la version améliorée de BFS, nous avons créé un algorithme qui calcule tous les voisin d'une grille donnée, en effectuant tous les swaps possibles sur chaque ligne et sur chaque colonne, à l'aide de boucles for sur la taille de la grille. Ensuite, nous avons créé une fonction qui permet de passer d'une grille de type hashable à une grille sous forme de tableau, en utilisant des split sur la chaîne de caractère afin d'enlever les "n" et les "/", et on repère la fin d'une ligne grâce à ce dernier symbole. Enfin, pour implémenter la version améliorée de BFS, nous reprenons quasiment identiquement l'algorithme BFS déjà implémenté dans graph.py, mais en parcourant uniquement les voisins du sommet sur lequel on se situe, en transformant notre sommet en objet de type Grid grâce à la fonction "de hashage à grille", puis en recherchant tous ses voisins grâce à la fonction implémentée pour. Ainsi, dans cette nouvelle version, nous ne parcourons que la partie du graphe qui nous intéresse au lieu de le garder en mémoire.

Enfin, pour implémenter A*, nous avons créé une heuristique basée sur la distance de Manhattan : pour chaque élément, nous regardons ses coordonnées (x, y) et ses coordonnées voulues (xv, yv), et on fait la somme abs(xv-x)+abs(yv-y), puis nous sommons cela sur tous les éléments de la grille. Pour l'algorithme A*, nous commençons par définir Dejavu qui est est un dictionnaire qui enregistre si un nœud a été visité, Distance qui est un dictionnaire tenant compte de la distance de chaque noeud au noeud initial, Parents pour remonter la solution, et avisiter, qui est une file de priorité basée sur le module heapq, où chaque élément est un tuple contenant la somme de la distance depuis le nœud de départ et de l'heuristique estimée pour atteindre le nœud de destination, ainsi que le noeud.

Nous créons ensuite une boucle while tant que la liste avisiter est non vide, puis on extrait le nœud avec la priorité la plus élevée de la file avisiter. Si ce nœud est la destination, l'algorithme s'arrête. Sinon, pour chaque voisin du nœud actuel non encore visité, l'algorithme calcule si le chemin à travers le nœud actuel est plus court que le chemin connu précédemment. Si c'est le cas, il met à jour la distance du voisin, ajoute le nœud actuel comme parent du voisin, et ajoute le voisin à la file de priorité avec la nouvelle priorité calculée. Une fois la destination atteinte, l'algorithme nous renvoie la solution de la même manière que pour BFS, c'est-à-dire en remontant les parents.

Enfin, nous avons utilisé le module Pygame afin de créer une interface graphique, qui permet de jouer au jeu en cliquant sur les cases à échanger, ainsi que de voir la solution donnée par A*. Pour cela, nous avons dessiné des cases, mis les nombres dedans, et selon le clic de l'utilisateur, nous obtenons les positions de ce clic, et connaissant la taille des cases, nous pouvons remonter au numéro de ligne et de colonne du clic,

et ainsi obtenir la case cliquée. Nous avons aussi créé plusieurs niveaux de difficulté, basés sur l'heuristique estimée à la grille d'arrrivée, connaissant l'heuristique de la pire grille possible qui est celle qui est complètement inversée.

Nous implémenté deux autres versions de A*, en modifiant uniquement la liste des voisins qu'on parcourt sur chaque noeud. Pour le premier, nous interdisons un échange entre deux cases données par l'utilisateur, donc nous avons créé une autre fonction qui cherche les voisins, mais en retirant ceux qui sont obtenues par ce swap interdit en créant une condition if. Dans le second, nous générons aléatoirement des swaps interdits à chaque étape du jeu, en créant une autre fonction qui recherche les voisins autour d'un noeud en interdisant un certain échange aléatoire trouvé grâce au module random et une fonction qui permet de trouver les cases adjacentes (à l'aide d'une boucle for, en regardant toutes les directions possibles), de la même manière, à l'aide d'une condition if, en reprenant la fonction de recherche de voisins déjà existante.

3. Analyse de la complexité

A. Complexité de la méthode naïve :

À chaque itération, la position actuelle et la position désirée de chaque carreau sont déterminées, ce qui nécessite O(m*n) opérations. Ensuite, dans le pire des cas, chaque carreau doit être déplacé à travers sa ligne ou sa colonne jusqu'à sa position correcte, ce qui représente jusqu'à O(m*n) opérations par déplacement. Comme cette opération est répétée pour chaque carreau, le nombre total d'opérations dans la boucle principale est O((m*n)^2). Ainsi, la complexité temporelle totale de l'algorithme est de O((m*n)^2). Cela signifie que pour les grilles de grande taille, le temps d'exécution de l'algorithme augmentera rapidement. Il était donc essentiel d'optimiser cet algorithme pour des performances optimales.

B. Complexité de l'algorithme BFS :

On sait que la complexité de l'algorithme BFS est de O(nombre d'arêtes+nombre de sommets). Il nous suffit donc de calculer le nombre d'arêtes et le nombre de sommets de notre graphe. D'une part, il y'a autant de noeuds que de permutations de la liste des entiers de 1 à m*n, soit (m*n)! noeuds. D'autre part, chaque noeud possède m*(n−1)+n*(m−1) voisins par un swap (m*(n-1) swaps horizontaux, et n*(m-1) swaps verticaux). Ainsi il y'a au total (n*m)!*(m*(n−1)+n*(m−1)) arrêtes dans notre graphe et donc l'algorithme BFS a une complexité de O((n*m)!*(m×(n−1)+n×(m−1))+1)=O((m*n)!(2m*n-m-n+1)). Comparativement, la méthode naïve implique potentiellement un nombre beaucoup plus élevé de swaps, mais est plus rapide en raison de sa simplicité. La méthode BFS, bien que plus efficace en termes de nombre de swaps, est donc plus coûteuse en termes de temps de calcul en raison de la construction du graphe et du parcours BFS.

4. Résultats et évaluation de l'algorithme de résolution

En utilisant la grille donnée par l'énoncé du projet, nous allons comparer la solution donnée par chacun des algorithmes, ainsi que le temps pris pour retourner la solution, à l'aide du module time.

```
grille=Solver(2, 3, [[5, 3, 6], [2, 1, 4]])
t = time.perf_counter()
print(grille.get_solution())
print("Voici le temps de la méthode naïve")
print(time.perf_counter() - t)
```

- La méthode naïve nous renvoie la solution : [((1, 1), (1, 0)), ((1, 0), (0, 0)), ((1, 1), (0, 1)), ((1, 1), (1, 2)), ((1, 2), (0, 2)), ((1, 1), (1, 0))] avec un temps de 0.0114 secondes.
- Le BFS amélioré nous renvoie la solution : [((0, 0), (1, 0)), ((0, 1), (1, 1)), ((0, 0), (0, 1)), ((1, 1), (1, 2)), ((1, 0), (1, 1)), ((0, 2), (1, 2))] avec un temps de 0.06 secondes, tandis que le BFS classique ne fonctionne pas.
- A* nous renvoie la solution : [((0, 0), (1, 0)), ((1, 0), (1, 1)), ((0, 2), (1, 2)), ((0, 1), (0, 2)), ((0, 0), (0, 1)), ((0, 0), (1, 0))] avec un temps de 0.0115 secondes.

Nous observons donc que la méthode naïve s'avère être la solution la plus optimale sur des petites grilles, aussi bien en terme de longueur de solution que de temps.

Enfin, soit la grille grille=Solver(6, 6, [[34, 20, 26, 1, 25, 13], [12, 35, 11, 31, 7, 27], [6, 2, 17, 36, 30, 21], [9, 29, 8, 3, 16, 18], [32, 15, 4, 22, 14, 19], [24, 33, 23, 10, 28, 5]])

- La méthode naïve nous renvoie une solution de taille 124 en un temps de 0.01 secondes.
- A* nous renvoie une solution de taille 102 en un temps de 6.5 secondes.

Nous observons donc que l'algorithme A* nous renvoie une solution plus optimale en terme de longueur, mais en un temps plus long que la méthode naïve.

5. Conclusion

Grâce à ce projet, nous avons pu découvrir et implémenter diverses méthodes de parcours de graphe, mais aussi appris à créer des interfaces graphiques à l'aide du module Pygame.

En comparant les résultats proposés par la méthode naïve, BFS et A*, nous observons que la méthode naïve est très peu coûteuse en temps, mais que A* propose une solution de longueur plus optimale que cette dernière, alors que BFS ne fournit ni une solution rapide ni optimale en longueur.

Enfin, il pourrait être intéressant de chercher d'autres heuristiques pour l'algorithme A*, afin de le rendre plus optimal en terme de complexité en temps.

6. Sources :

[1]:https://www.supplychaininfo.eu/dossier-optimisation-logistique/comment-utiliser-recherche-operationnelle-optimisation-logistique/