ENSAE 1A – Projet de programmation 2024

Groupe 4 : Yasmine Houri

Binôme :
Anna TRETYAKOVA
Eliott VALETTE

Lien GitHub du projet : https://github.com/VS-EV/ensae-prog24

Séance 1 :

- Implémentation de la méthode *swap* :

Par souci de clarté nous avons divisé cette méthode en deux parties, une première méthode statique *is_swap_valid* vérifiant la validité d'un swap entre deux cellules grâce à la condition nécessaire et suffisante énoncé page 3. Puis nous avons effectué ce swap dans une affectation simultanée pour assurer le bon déroulement de l'échange de position.

- Implémentation de la méthode *swap_seq* :

Une boucle for nous permet aisément de parcourir toute la liste de tuple de 2 cellules et d'effectuer le swap entre ces deux dernières grâce à la méthode ci-dessus.

- Implémentation de la méthode *is_sorted*:

En utilisant les propriétés de l'initialisation des objets de la classe Grid, il suffit de comparer l'état de self avec l'état d'un objet Grid dont on n'a pas précisé l'état initial.

Tests :

```
class Test_Swap(unittest.TestCase):
    def test_grid1(self):
        grid = Grid.grid_from_file("input/grid1.in")
        grid.swap((3,0), (3,1))
        self.assertEqual(grid.state, [[1, 2], [3, 4], [5, 6], [7, 8]])

    def test_grid1_seq(self):
        grid = Grid.grid_from_file("input/grid1.in")
        grid.swap_seq([((3,0), (3,1)), ((3,0), (3,1))])
        self.assertEqual(grid.state, [[1, 2], [3, 4], [5, 6], [8, 7]])

class Test_IsSorted(unittest.TestCase):
    def test_grid1(self):
        grid = Grid.grid_from_file("input/grid1.in")
        self.assertFalse(grid.is_sorted())
        grid.swap((3,0), (3,1))
        self.assertTrue(grid.is_sorted())
```

- Implémentation d'une méthode naïve de résolution :

Nous avons choisi de procéder de la manière suivante pour chacun des entiers (dans l'ordre croissant)[3] de la grille : Trouver l'entier i dans la grille[1], s'il n'est pas à sa place, effectuer une série de swap pour le déplacer vers celle-ci en suivant le schéma suivant : déplacer i horizontalement pour le placer dans sa colonne dédiée pour enfin le faire remonter a sa place[2].

Ce fonctionnement nous garantit de ne pas déranger les entiers allant de 1 à i-1, en effet la grille se rangeant du haut vers le bas et de gauche vers la droite, chaque swap se fera nécessairement avec un entier pas encore rangé.

Pour ce faire nous avons implémenté les méthodes suivantes dans la classe Solver :

- *find_coordinates_x* (1)
- *drag_x* (2)
- *get_solution* (3) *:* A l'aide d'une boucle *for*, chacun des entiers de la grille sont déplacés un par un à leur place respective.

Or le rendu attendu étant une séquence de swaps, tout au long de ce processus, une variable globale *sequence_swaps* se voit incrémenté d'un tuple de deux cellule à chaque fois qu'un swap entre celles-ci est effectué. *sequence_swaps* est alors enfin retournée une fois la boucle *for* épuisée.

Complexité :

La méthode de résolution implémentée utilise des boucles et des swaps pour déplacer les entiers dans la grille. La complexité de cet algorithme dépend du nombre *n* d'entiers dans la grille et du nombre de swaps nécessaires pour les déplacer à leurs positions correctes. En effet *get_solution* utilise une boucle for allant de 1 à n, a chaque rang, drag_x fait appel à find_coordinates qui elle-même utilise deux boucles *for* imbriquées de longueur moyenne chacune sqrt(n). Ainsi nous estimons que *get_solution* a une complexité suivant l'ordre de grandeur O*(n* sqrt(n)^2)* c'est-à-dire O(n^2).

Tests :

```
grid_file_name = "input/grid1.in"


class Test_GetSolution(unittest.TestCase):
    def test_sorted_grid(self):
        grid = Grid.grid_from_file(grid_file_name)
        grid.state=[[1, 2], [3, 4], [5, 6], [7, 8]]
        print(grid.state)
        solver = Solver(grid.m, grid.n, [[1, 2], [3, 4], [5, 6], [7, 8]])
        solution = solver.get_solution()
        self.assertEqual(solution, [])

    def test_partially_sorted_grid(self):
        grid = Grid.grid_from_file(grid_file_name)
        grid.state=[[1, 2], [3, 4], [5, 6], [8, 7]]
        solver = Solver(grid.m, grid.n, grid.state)
```

```
        solution = solver.get_solution()
        # Dans ce cas, seul un swap est nécessaire pour trier la grille.
        self.assertEqual(solution, [((3, 1), (3, 0))])

    def test_unsorted_grid(self):
        grid = Grid.grid_from_file(grid_file_name)
        grid.state=[[2, 1], [4, 3], [5, 6], [8, 7]]
        solver = Solver(grid.m, grid.n, grid.state)
        solution = solver.get_solution()
        # Dans ce cas, plus d'un swap est nécessaire pour trier la grille.
        self.assertTrue(len(solution) > 1)
```

- Représentation de la grille

Grace à une méthode faisant appel à la bibliothèque *matplotlib.pyplot*, la méthode *plot_grid* représente graphiquement l'état de la grille.

Séance 2 :

- Implémentation de la méthode bfs dans la classe Graph :

Pour cette étape, nous avons implémenté la méthode bfs dans la classe Graph. Cette méthode prend en paramètre le nœud de départ et le nœud cible. Elle parcourt le graphe à partir du nœud de départ jusqu'à ce qu'elle atteigne le nœud cible ou qu'elle explore tous les nœuds accessibles (retournant None le cas échéant). Pendant ce processus, elle maintient une file d'attente (queue) des nœuds à explorer.

Exploration du graphe avec BFS :

L'algorithme BFS explore le graphe couche par couche, en partant du nœud de départ. À chaque étape, il visite tous les voisins du nœud actuel avant de passer aux voisins de ces voisins, et ainsi de suite. Cette approche garantit que le premier chemin trouvé entre le nœud de départ et le nœud cible est le plus court. Une fois que le nœud cible est trouvé, nous retournons le chemin le plus court sous forme d'une liste de nœuds traversés.

Variables utilisées :

path: Une liste contenant le chemin actuel exploré à partir du nœud source jusqu'au nœud courant.
queue: Une file d'attente (queue) des nœuds à explorer.
(Initialement, ces deux variables contiennent logiquement seulement le nœud source.)
all_paths: Un dictionnaire qui stocke tous les chemins découverts jusqu'à présent. Chaque clé est un nœud et sa valeur est le chemin complet jusqu'à ce nœud. Elle représente l'arborescence de l'exploration de bfs, couche après couche.
visited: Une liste pour enregistrer les nœuds déjà visités.

Processus :

Path est mis à jour à chaque étape pour inclure le nœud courant ainsi que les nœuds précédemment visités sur le chemin jusqu'à ce nœud. Cette variable représente une branche de l'arborescence. Elle sert a actualiser le dictionnaires des chemins *all_paths*
queue: Commence avec le nœud source. À chaque étape, un nœud est retiré de la file d'attente pour être exploré (*.pop(0)*), et ses voisins sont ajoutés à la file d'attente.
visited: À chaque étape, le nœud actuel est ajouté à la liste des nœuds visités pour éviter de revisiter les mêmes nœuds.

A chaque étape de ce processus, la boucle while vérifie si le nœud objectif dans un chemin, si oui, queue est vide, on sort de la boucle et retourne le chemin, sinon le processus continue

Dans la classe Solver, la méthode *get_solution_bfs* parcours chacun des nodes pour renvoyer un fichier output au format path.out

Tests :

```
class Test_BFS(unittest.TestCase):
    def test_graph1(self):
        graph = Graph.graph_from_file("input/graph1.in")
        Solver.get_solution_bfs(graph)
        with open("tests/bfs.txt", "r") as file:
            resultat_observe = file.read()


        with open("input/graph1.path.out", "r") as file:
            resultat_attendu = file.read()


        self.assertEqual(resultat_observe, resultat_attendu)
```

- Chemin optimal pour Swap_puzzle

Avant d'utiliser bfs dans le contexte de swap puzzle nous avons défini une multitude de méthodes :
get_nodes : qui, selon la taille de self, renvoie un dictionnaire dont les clés sont des entiers de 1 à fact(m*n) et chaque entier est associé à une valeur, un état unique d'une grille (m,n). Pour ce faire nous avons utilisé la bibliothèque itertools donnant accès a la fonction permutations.
are_neighbours : Qui comment son nom l'indique vérifie si il existe un unique swap tel que 2 grilles soient égales. Cette méthode fait appel à :
generate_swapped : utilisant *copy.deepcopy,* cette dernière renvoie une grille swappée sans affecter la grille initiale.
get_neighbours : parcourant le dictionnaire des nœuds, renvoie un dictionnaire associant les nœuds voisins à chaque nœuds en appelant successivement la méthode are_neighbours.

Enfin la méthode *generate_graph* s'appuyant des fonctions précédentes renvoie un object graph dont les nodes et les edges sont tous renseignés. Nous tenons à noter que nous avons rajouté une condition *(neighbor_key,node_key) not in graph.edges* prévenant de l'apparition de doublons dans les voisins, et ainsi améliorant la clarté ainsi que l'efficacité.

La méthode *get_solution_bfs* évoquée précedemment nous permet alors d'obtenir le résultat souhaité.

Tests :

```
class Test_BFS(unittest.TestCase):
    def test_graph1(self):
        grid_file_name = "input/grid0.in"
        grid = Grid.grid_from_file(grid_file_name)
        graph_from_grid = grid.generate_graph()
        Solver.get_solution_bfs(graph_from_grid)
        with open("tests/bfs.txt", "r") as file:
            resultat_observe = file.read()

        with open("tests/grid1_verified_output.txt", "r") as file:
            resultat_attendu = file.read()

        self.assertEqual(resultat_observe, resultat_attendu)
```

Complexité :
A chaque exécution du *bfs* sur une *grid* dont l'état est de taille (m,n), factorielle(m*n) nœuds sont créés. En effet il existe factorielle(m*n) permutations d'entiers allant de 1 à m*n et donc autant d'états différents i.e de nœuds.

Pour calculer le nombre total de swaps valides dans une grille (m,n), nous pouvons utiliser la formule suivante : $(m-1)\times n+(n-1)\times m$. En effet Cela prend en compte les swaps horizontaux et verticaux possibles. Le terme $(m-1)\times n$ correspond au nombre de swaps horizontaux, et le terme $(n-1)\times m$ correspond au nombre de swaps verticaux. Nous pouvons constater que ce calcul avait été fait implicitement dans la méthode *are_neighbours,* ou nous effectuons 2 boucles *for* successives chacune imbriquant une autre boucle *for* d'une complexité de $(m-1)\times n+(n-1)\times m$.

Ainsi, bien que la méthode bfs donne un résultat bien plus performant que la méthode naïve, sa complexité impliquant une factorielle rend difficile d'envisager une utilisation pour des grandes matrices.

- Bfs optimisé

En raison d'une compréhension tardive de ce qui était attendu nous ne sommes pas parvenu à effectuer cette question, cependant voici notre raisonnement :
Le fonctionnement du bfs ci-dessus n'est pas optimisé car il prend en entrée l'intégralité du graph, ce qui ne nous permets pas avec les méthodes actuelles d'améliorer son efficacité. Il nous aurait fallu définir une fonction pour obtenir les voisins d'un nœud, sans passer par une recherche dans le dictionnaire des nœuds, par exemple en définissant une méthode retournant tous les swaps valides, ce qui nous permettrait de déterminer les voisins d'un nœud en appliquant successivement ces swaps. Le fonctionnement du bfs n'en serait que peu changé, simplement les voisins ne sont calculés que lorsque cela est

nécessaire, ce qui réduirait grandement le nombre de calculs dans le cas de matrices de grande taille.

Séance 3,4 :

- Implémentation de A*

Le principe de la fonction A* est d'améliorer le BFS en explorant en priorité les chemins les plus prometteurs afin de renvoyer plus rapidement le chemin le plus court. Pour cela, nous avons tout d'abord importé la bibliothèque *heapq* on établit une queue qui va contenir des tuples formés par une grille et l'estimation de la longueur du chemin restant entre cette grille et la solution finale. On va de cette façon prioriser nos recherches.

En résumé, on part de notre grille initiale, on regarde ses voisins, pour chaque voisin qui n'a pas encore été visité, on estime sa distance à la solution finale. On sélectionne le voisin qui est le plus proche de la solution finale et on réitère le processus jusqu'à avoir une grille ordonnée. cet algorithme nous permet de ne pas visiter toutes les branches du graphe mais de se diriger toujours vers celle qui semble être la plus courte.

Tests :

```
class TestAStar(unittest.TestCase):
    def test_a_star_solution(self):
        # Define an initial grid
        grid_file_name = "input/grid0.in"
        grid = Grid.grid_from_file(grid_file_name)

        # Get the solution using A*
        solution = grid.optimized_solver_astar(grid,grid.m,grid.n)

        # Define the expected solution
        expected_solution = [
            (2, 4, 3, 1),
            (2, 1, 3, 4),
            (1, 2, 3, 4)
        ]

        # Check if the generated solution matches the expected solution
        self.assertEqual(solution, expected_solution)
```

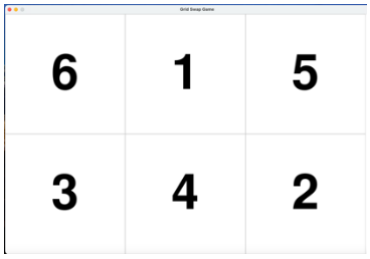
Pygame :

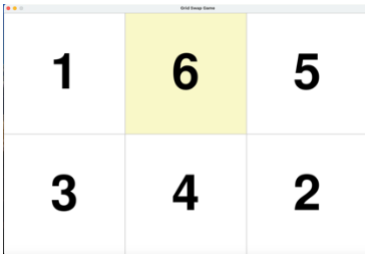
Dans la partie Pygame de notre projet, nous avons développé une interface utilisateur pour le jeu de swap de grille. Le but du jeu est de réorganiser une grille désordonnée en

effectuant des swaps de cases adjacentes. Pour effectuer un swap, l'utilisateur clique sur une case avec la souris, puis la tire vers une case adjacente.

Nous avons mis en place une boucle principale qui gère les événements utilisateur, tels que les clics de souris. Lorsqu'un clic est détecté, le programme identifie la case sélectionnée et permet à l'utilisateur de la déplacer vers une case adjacente en relâchant le clic de souris. Pendant ce temps, la grille est constamment mise à jour et affichée à l'écran, permettant à l'utilisateur de suivre visuellement ses progrès. De plus, une vérification est effectuée pour déterminer si la grille est rangée dans l'ordre croissant, auquel cas un message de félicitations est affiché, indiquant le nombre de swaps effectués pour résoudre le puzzle. Si la grille est petite, une suggestion est également affichée, montrant combien de swaps auraient été nécessaires pour résoudre le puzzle de manière optimale.

Par ailleurs, il vous suffit de cliquer sur la touche r, pour relancer une partie avec une grille nouvellement mélangée tandis que si vous appuyez sur t, une nouvelle grille est chargée.





Congratulations! You beat the game in 7 swaps!

However it could have been beaten in 5