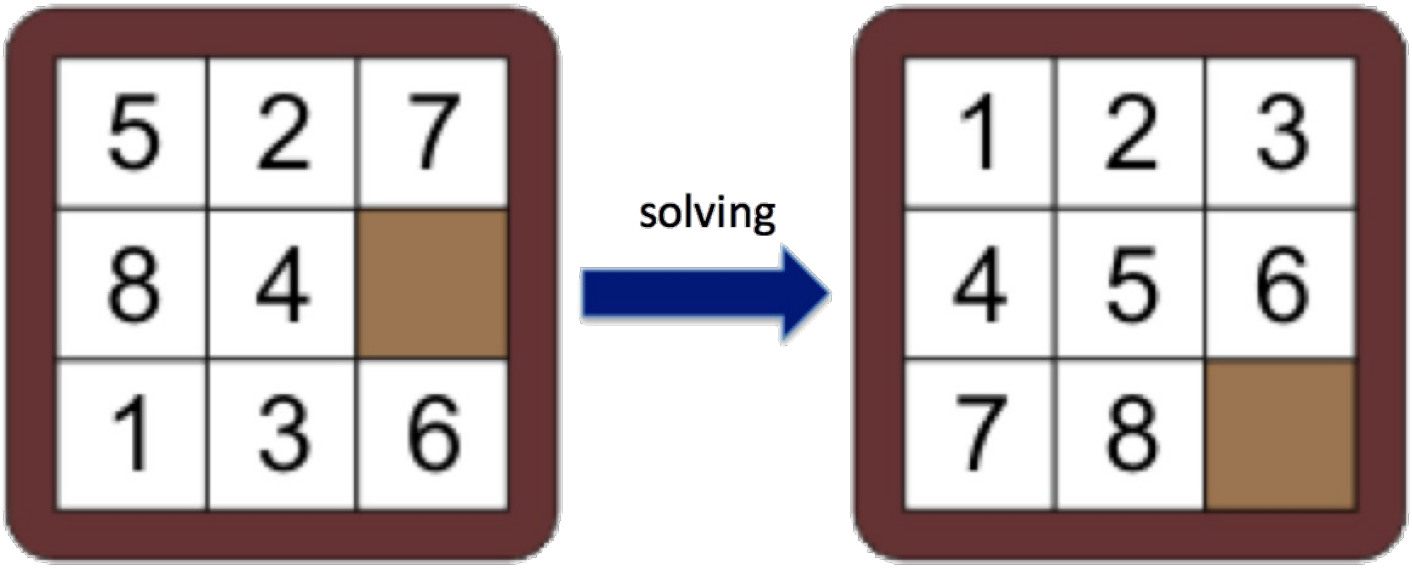
Compte rendu de projet

Swap Puzzle

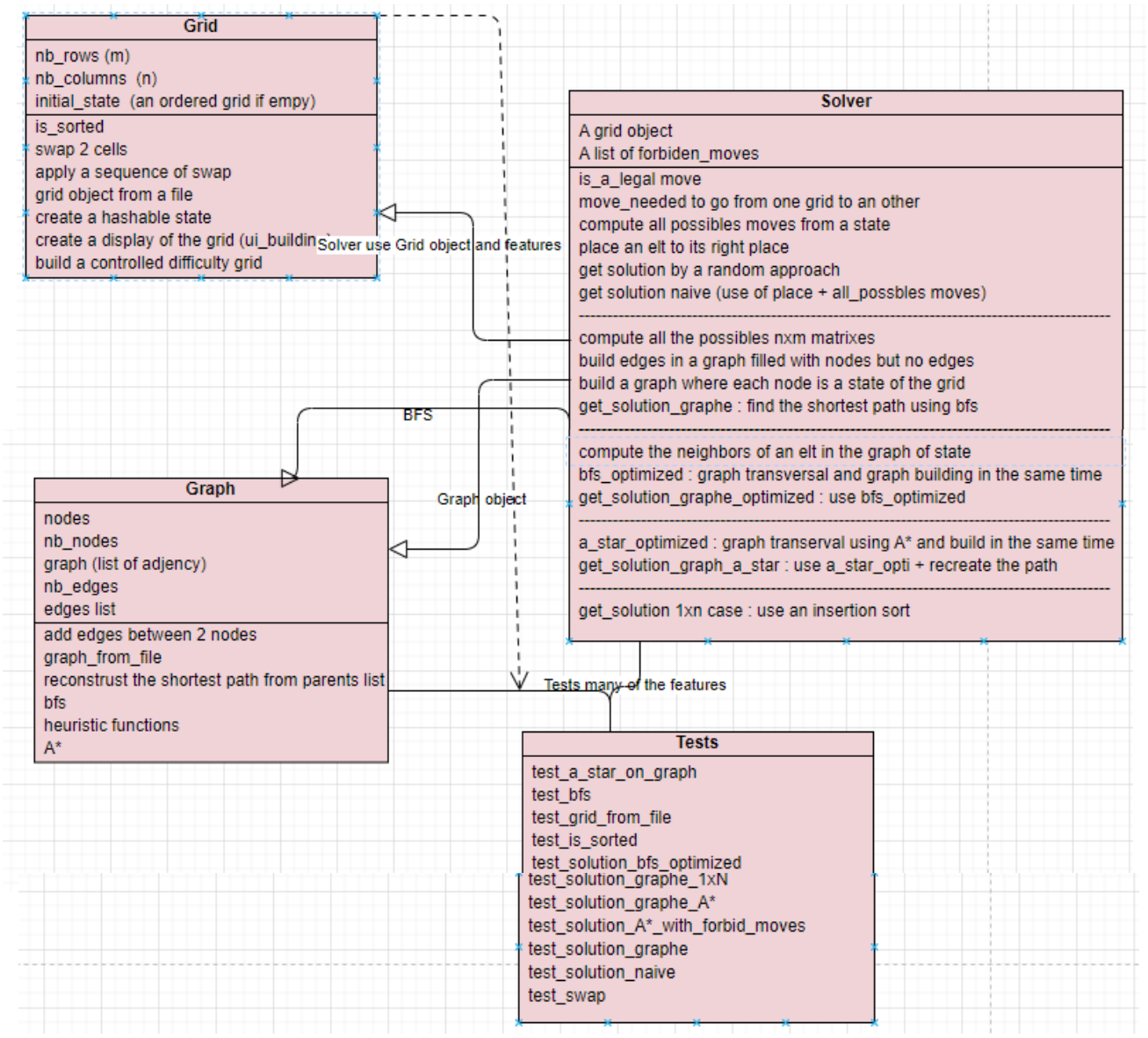


Célian MARSALA
Quentin FOSALVA

ENSAE

INSTITUT POLYTECHNIQUE DE PARIS

Diagramme de classe



Choix Algorithmique et complexité

Grid :

- is_sorted() :
    création d'une grille bien ordonnée - O(taille grille) = O(nm)
    comparaison de deux tableaux de même taille O(mn)

Donc O(nm)

- swap : O(1)

- swap_seq : O(nombre de swap à effectuer) = O(len(cell_pair_list))

- hashable_state : la conversion d'une liste en tuple est en temps linéaire donc O(n*m) car chaque inner_list contient n élement

Graph :

- add_edge :
    vérification que l'arrête n'est pas présente O(len(self.edges))
    verification si les noeuds sont déjà présent dans le graphe O(1)

Donc O(len(self.edges))

- bfs :
    Complexité temporelle -> O(self.nb_edges+self.nb_nodes)
    Complexité spatiale -> O(self.nb_nodes)

- h_wrong_place :
    Complexité temporelle -> double boucle for -> O(nm)
    Complexité spatiale -> création de la grille ordonnée ->O(nm)

Cette heuristique est admissible (testé en pratique en comparant à la solution renvoyé par le BFS) cependant elle n'est clairement pas cohérente ce qui n'optimise pas la complexité de notre code.

- manhattan :
    On utilise l'heuristique de Manhattan, qui est une heuristique classique dans l'implémentation de A*. La distance de Manhattan consiste à compter pour chaque pion du tableau le nombre de cellules

qui sépare sa position courante de sa position finale ; puis à sommer ce nombre pour tous les pions du tableau de jeu. Cette heuristique a l'avantage d'en plus d'être admissible, d"être cohérente ce qui en pratique divise par 10 le temps de résolution. (15 sec pour wrong place contre 0,2 en moyenne pour manhattan).

- A* :

Pour que notre algorithme renvoie un chemin de poids minimal (i.e un plus court chemin car la fonction de poid des arêtes est constante à 1), l'heuristique doit être admissible i.e : $\text{pour tout sommet } v,\ h(v) \leq d(v, but)$

De plus si l'heuristique est cohérente i.e :

$$\text{pour toute arête } v \to v',\ h(v) \leq \rho(v \to v') + h(v')$$

alors la complexité de A* est O(|E| +|V|) avec E = edges et V =vertex Sinon, elle peut être exponentielle en la taille du graphe.

Concernant **l'implémentation**, nous avons utilisé un dictionnaire qui renvoie +infini quand la valeur est manquante pour implémenter dist qui stock l'actuelle + courte distance entre deux sommets. On a aussi utilisé une file de priorité à laquelle on a rajouté une méthode decrease_or_push qui baisse la prio si l'élement est déjà dans la file et qui l'ajoute sinon.

Solver :

- legal_move :

Prend deux positions et renvoie un booléen pour savoir si le swap des deux positions est possible. Cette fonction vérifie aussi que le move n'est pas dans les coups interdits (on pense à vérifier les deux (y,x) et (x,y)) O(|self.forbiden_moves|)

- possible_moves :

Renvoie tous les mouvements possible à partir de l'état de self.grid sous forme de liste. Pour chaque élement (i,j) de la grille, on calcule dans une liste les swaps autour de lui (Haut, Bas, Gauche, Droite) puis on regarde parmis ces swaps lesquels sont valides grâce à la fonction

legal_move. Les swaps valides sont ajoutés à une liste de tous les coups possibles qui sera renvoyé à la fin de la fonction.
O(n*m)

- get_soltion_naive_random:
Tant que la grille n'est pas triée on calcule tous les mouvements possible, on en choisit un, on l'éxecute et on recommence.

- place :
Fonction qui place x sans faire aucun swap donc sans faire bouger les elts 1,2,..,x-1. Pour cela on crée une grille ordonné dans laquelle on trouve la bonne place à laquelle x devrait être (on pourrait éviter cette création en utilisant des divisions euclidiennes et des modulos). Puis une fois cette bonne place trouvé, on va effectuer tous les moves possibles pour nous rapprocher de cette bonne place sans bouger les eléments inférieurs.

Complexité de cette fonction en O(mn)

- get_solution_naive :
On place l'élément 1 puis l'élément 2 etc... jusqu'à n*m. La fonction place nous assure que une fois que l'élement 1 est placé par exemple la mise en place des élements i, i>1 ne perturebera pas la place de 1. Ce qui nous assure au final une grille triée. Cependant la solution n'est pas minimale.

Complexité en O((mn)^2)

- generate_matrice :
Renvoie dans une liste toutes les matrices sous forme hashable possible de taille n*m

- build_edges :
Pour un état donnée state d'une grille on calcule tous ses voisins (i.e les grilles qui sont à un mouvement de cette état) et on ajoute une arête entre eux.

- build_graph :

on génére toutes les matrices taille n*m, on crée un graphe dont les noeuds sont toutes ces possibles matrices. Puis on crée les arêtes grâce à la fonction build_edges applique sur chaque noeud.
Complexité en O((nm)!)

- get_solution_graphe :
On construit le graphe des états comme expliqué ci dessus puis on applique un BFS qui nous renvoie la liste des états par lesquels on est passé. Il nous suffit plus que d'en tirer les swaps réalisés pour reconstruire la solution minimale.
Complexité en O((nm)!) à cause de la construction du graphe.

Pour tirer parti de cette remarque, nous n'allons plus construire le graphe dans son entiereté puis après le parcourir, nous allons en même temps que nous le parcourons le construire.

- compute_neighbor :
Pour un state et un graphe donné cette fonction va calculer et ajouter au graphe tous les voisins de ce state (i.e toutes les autres grilles distantes de 1 coup).
Complexité en O(nm)

- bfs_optimized :
Implemente un parcours en largeur qui construit le graphe lors de son exploration. On aurait pu optimiser la complexité spatiale en choisisant des dictionnaires plutôt que des listes pour stocker les noeuds vus et les distances.
La complexité dans le pire cas n'est pas amélioré car on pourrait imiaginer un cas pathologique où on se retrouve à construire le graphe en entier cependant en pratique le temps d'éxecution de cette méthode est bien plus rapide que le bfs classique.

- get_solution_graphe_optimized :
Utilise bfs_optimized puis recrée la liste des swaps effectués. Cette fonction renvoie une solution optimale.

- a_star_optimized :

De la même manière que bfs_optimized on a modifié la fonction A* pour créer le graphe en même temps qu'on le parcours. On a utilisé le module heapq pour implémenter la file de priorité.

Résultats

test_a_star_on graph : 0.02s
test_bfs : 0.024s
test_solution_bfs_optimized : 0.39s
test_solution_1xn : 0.06s
test_solution_graphe_A_star_manhattan : 0.05s
test_solution_graph_A_star_wrong_place : 13.2s
test_solution_naive : 0.03s