RAPPORT PROJET D'INFORMATIQUE

Par Adrien Barrau et Diane Sainteny

Ce compte-rendu se veut presque entièrement chronologique.

1 CLASSE GRID

1.1 get_solution_naive

Une solution très simple est de parcourir la matrice de taille (m,n) avec un compteur k allant de 1 à (m*n). Pour chaque coefficient (i,j) ayant pour valeur k, on trouve les coefficient (i1,i2) où serait situé la valeur k si la grille était résolue. A l'aide d'une succession de swaps horizontaux (vers la droite ou la gauche) on déplace la valeur k sur la bonne abscisse, idem pour l'ordonné. Il est crucial d'effectuer ces opérations dans cet ordre sous peine de "détruite" le début du trie. C'est ce que fait la fonction go_to(self,(i,j),(i1,j1)) qui renvoi une liste de swaps [voir grid.py ligne 162 qui renvoi une liste de swaps] On réutilise la fonction go_to dans la fonction get_solution_naive(self) qui renvoi la liste complète des swaps à effectuer.

1.2 les fonctions permettant la créations de murs

Nous n'avons pas eu le temps de créer de fonction test pour ces fonction. Cependant, le jeu marche bien quand on les implémente.

identification et origin_id : ces deux fonctions associent un identifiant unique à chaque case du tableau (ou permettent de retrouver la case grâce à son identifiant). Elles servent à simplifier la fonction creation_walls() décrite juste après.

creations_walls(self, nombre) : cette fonction prend en argument le nombre de murs à créer. Pour les créer, dans une boucle, on génère à chaque fois deux nombre aléatoire qui correspondent aux identifiants de deux cases. Il faut deux conditions pour qu'ils soient validés : que les deux cases soient mitoyennes et qu'il n'y ait pas déjà un mur validé au même endroit.

2 CLASSE GRAPH

2.1 bfs

L'idée est de créer un graphe dont les noeuds sont des tuples de tuples car ils sont de type hashable (on convertit préalablement les matrices qui correspondent aux états de la grille en tuples de tuples avec une fonction matrice_into_tuple). Une fois le graphe créé, il n'y plus qu'à chercher le plus court chemin entre la grille de départ src, et la grille d'arrivée dst. Notons que la création du graphe nécessite de générer (m*n) ! noeuds, soit 479 millions de noeuds pour une grille de dimensions (4,3). On ne pourra pas espérer mieux que de résoudre des grilles de taille (3,3) en un temps raisonnable avec cette méthode. L'algorithme BFS explore tous les voisins du nœud actuel, si l'un d'eux est le but, alors on renvoie le chemin utilisé en parcourant les ancêtres du nœud avec le dictionnaire des pères (dict_pere). On peut visualiser l'algorithme comme une onde qui se déplace dans le graphe. Cela donne alors l'idée de lancer deux BFS simultanément sur le graphe, l'un partant de la source et l'autre du but, puis de recomposer les deux chemins a l'endroit ou les parcours se rencontrent. La complexité en temps (nombres de calculs) du BFS est en O(|A|+|N|) avec |A| le nombres d'arêtes et |N| le nombre de nœuds.Or |N|=(m*n) ! (un peu moins pour new_bfs), et |A|=|N|((m-1)n+(n-1)*m)=(2mn-m-n) (m*n) ! .Le BFS est linéaire en le nombre d'arêtes et de noeuds, mais ces derniers explosent quand m,n grandissent : Complexite en O(mn(m*n) !). On sera vite limité par la taille de la grille.

2.2 new_bfs

On fait de même mais sans créer tout le graphe avant de le parcourir. Cela permet de ne pas générer des nœuds qui de toute manière ne seront pas utilisés par BFS. On obtient un algorithme plus rapide. Dans le pire cas, la complexité reste inchangée. Notons que le temps d'exécution de new_BFS sera toujours inférieur à BFS car on ne génère pas le graphe avant de le parcourir.

2.3 a_star

On utilise la distance de Manhattan (multiplié par un certain poids) en tant qu' heuristique pour estimer la distance d'un nœud jusqu'à la solution. Par distance de Manhattan entre deux matrices on entend la distance de manhattan coefficients par coefficients. Ainsi, au lieu de parcourir brutalement les nœuds comme dans BFS, on choisit a chaque étape le voisin le plus prometteur. Nous en avons testé d'autres heuristiques, comme la distance euclidienne, mais elles donnent toutes de moins bons résultats en particulier en grande dimension. Cependant, rien n'exclut la possible existence d'une bien meilleure heuristique que la distance de Manhattan. C'est dans la fonction heuristique2 (graph.py) qu' on peut faire varier le fameux poids, voir même trouver un formule qui dépend de m et n. On a donc une heuristique qui depend de la dimension de la grille ! On remarque que l'algorithme A* gagne en vitesse mais perd en précision lorsqu'on augmente le poid.Il faut alors modifier à la main la valeur du poids pour ne pas attendre trop longtemps. C'est un compromis entre la solution naïve et le BFS. Il est alors intéressant d'automatiser le processus, en fonction notamment de la taille de la matrice, ou bien simplement en fonction de la valeur de l'heuristique de la source La complexité de A star dépend de l'heuristique, mais dans le pire cas, A star ne fait pas mieux que BFS. En revanche , en pratique , la complexité est bien meilleure.

La fonction adjacent_grids (grid.py) prend en argument une grille, et renvoie une liste d' états des grilles atteignables en un swap. Pour cela on considère tous les swaps possibles en itérant de la gauche vers la droite et de haut en bas (pour ne pas avoir de doublons !). Dans bfs,new_bfs et a_star, cette fonction est essentielle pour trouver les voisins d'un nœud.

On va résoudre le jeux pour des petites tailles comme (2,2) et (3,2) pour créer une base de données de solutions. Cela remplace la série de tests que nous avions fait puisque cela montre que toutes les fonctions intermédiaires fonctionnent. On place donc ces fichiers dans tests.

2.4 Comparaison des performances des deux algorithmes

Pour déterminer la performance de nos modèles on se base deux critères : La vitesse d'exécution (liée à la complexité), et la précision de la réponse. On sait déjà que le BFS donne le plus court chemin, donc s'il termine il donne nécessairement une solution parfaitement précise. En revanche, il met plus de temps à s'exécuter que la solution naïve, qui elle est imprécise. A* est un compromis entre rapidité et précision. Pour en être certain je vais sélectionner au hasard des matrices de taille (m,n) et toutes les résoudres avec ces trois méthodes. Je vais ensuite comparer les deux critères majeurs. Tout ceci se trouve dans le fichier solver.py. Pour faire les test, il suffit de modifier liste_mat avec les arguments de la fonction generate_random_matrices(m,n,nb). Notons ici que tous les tests ont été fait sans toucher au poids dans la fonction heuristique2. Par tâtonnement, on pourrait optimiser le score de a_star pour chaque dimension, mais l'objectif est d'avoir une formule générale fixe pour le poids. On choisit poids=10**(m+n-4)+m+n Les résultats pour cette valeur du poids sont disponibles dans le fichier conclusion.txt dans le dossier tests.

3 CLASSE GRAPHICS

3.1 display_plt(tableau)

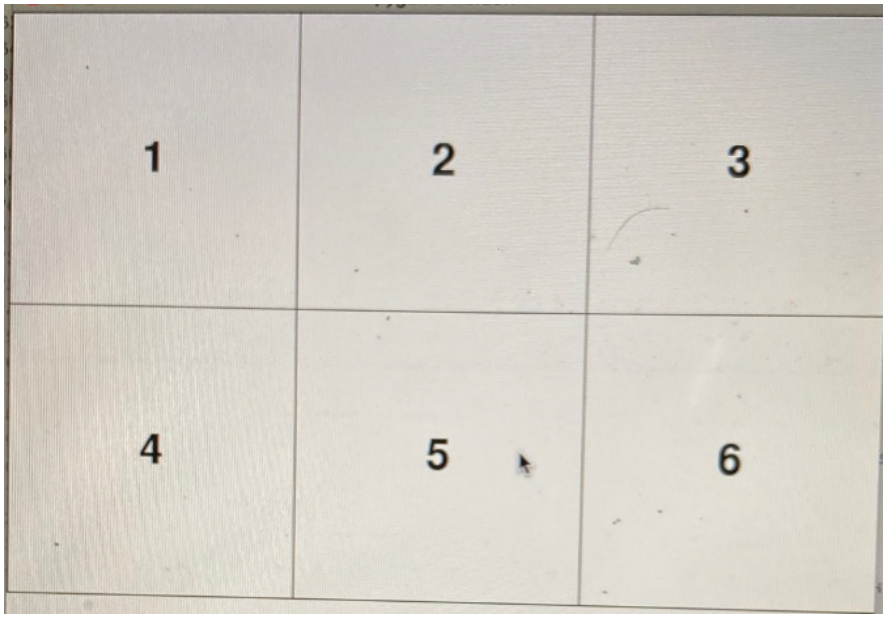
Ce premier affichage est un affichage simple, non intéractif. Ce n'est pas ce dernier que nous utiliserons dans la fonction finale permettant de jouer au jeu. Nous avons commencé par créer et placer des compartiments de texte, avant de tracer sur le fond une grille pour ajouter un aspect esthétique, avec un espacement correspondant à l'espacement des nombres entre eux.

3.2 display_pygame(tableau, to_do)

Il s'agit cette fois de l'affichage intéractif utilisé en partie dans le jeu final. Il y a deux arguments dans la fonction. "Tableau" désigne la grille dessinée et représentée. "To_do" désigne le nombre de swaps minimal pour ordonner la grille, calculé grâce aux algorithmes précédents. Le but est d'ordonner la grille en faisant ce nombre minimal de swaps.

L'espacement des lignes et des colonnes, tracées cette fois avant de placer les nombre dans des cases, dépend du nombre de lignes et de colonnes du TABLEAU et est calculé par rapport à la grandeur de la fenêtre.

L'affichage s'arrête et un message "You lose" s'affiche si l'utilisateur a effectué le nombre de swaps calculé par l'algorithme mais n'a pas résolu la grille. L'affichage s'arrête et affiche au contraire "You win" si l'utilisateur a réussi à parvenir à une grille ordonnée. A chaque swap, les deux cases échangées sont recouvertes par un carré blanc puis on réecrit le nouveau numéro par dessus. J'ai modélisé la même fonction avec des murs.



4 JEU PRINCIPAL DANS LA PARTIE MAIN

Pour commencer, quand on lance le jeu, les règles s'affichent, afin d'expliquer comment marche le jeu mais aussi en quoi consistent les différents niveaux.

Ensuite, c'est à l'utilisateur de choisir non seulement la dimension de la grille qu'il veut résoudre, mais aussi la difficulté. Il y a trois difficulté : une facile, une moyenne et une difficile. Pour les solutions faciles et difficiles, le but est de trouver grâce aux algorithmes le nombre de swaps optimal, puis de le stocker dans solution pour le mettre comme paramètre de la fonction display_pygame. L'algorithme a_star étant moins précis et l'algorithme new_bfs étant plus lent, nous avons décidé d'utiliser l'un ou l'autre pour résoudre la grille selon la taille de cette dernière. Pour le niveau difficile, il n'y a pas de calcul de nombre de swaps optimal, même si nous aurions aimé le faire et nous détaillons plus haut comment nous aurions fait. Par contre, il y a des murs qui empêchent plusieurs cases d'être échangées. Le but est de résoudre des grilles en moins de vingt swaps étant donné ces murs.