Rapport projet de programmation

Titouan Constance et Imade Haddadi

Avril 2025

1 Le jeu

Le jeu que nous étudions est un jeu où nous avons une grille n par m, chaque case possède une couleur (blanche, rouge, bleu, verte ou noir) et une valeur qui est un entier supérieur à 1. Nous pouvons apparier les cases adjacentes (sans compter les diagonales) selon les couleurs des cases :

- **Noirs** : Aucune case
- **Blancs** : Blancs, Bleus, Rouges, Verts
- **Bleus** : Bleus, Rouges, Blancs
- **Rouges** : Rouges, Bleus, Blancs
- **Verts** : Blancs, Verts

Sachant que v(i,j) représente la valeur de la case (i,j), le score associé à un choix de paire de cellules est représenté par la formule suivante :
Score $= \sum_{((i_1,j_1),(i_2,j_2))\in \text{Paires}} |v(i_1,j_1) - v(i_2,j_2)| + \sum_{(i,j)\notin \text{Cellules utilisées}} v(i,j)$

Nous cherchons alors ici à venir minimiser ce score avec un appariement optimal des cellules Nous modélisons informatiquement la grille dans une class Grid

2 La class Grid

2.1 Représentation informatique de la grille

Les valeurs et les couleurs de chaque cases sont enregistrées dans des liste de liste color et value telles que value[i][j] représente la valeur de (i,j), color[i][j] l'indice de la couleur de la case

2.2 Méthode de base de la class

La classe possède des méthodes de base :

- __str__() : permet de représenter la matrice dans le terminal sous forme de text
- is_valid_pair(i1,j1,i2,j2) : renvoie vrai si la pair ((i1,j1),(i2,j2)) est valide
- cost(pair) : Renvoie la valeur de la pair
- plot() : Affiche la grille via la fonction imshow du module matplotlib. Pour tenir compte du fait que l'indice de la ligne "augmente" vers le bas à l'inverse de matplotlib, il est nécessaire de renverser la liste selon l'indice
- all_pairs() : renvoie la liste de toutes les paires possible en regardant chaque paire possible et vérifiant que textis_valid_pair(i1,j1,i2,j2) renvoie vraie

3 Une première approche naïve

3.1 Méthode Greedy

La première méthode utilisée pour résoudre le jeu est l'algorithme glouton.

Nous avons besoin de quelques fonctions intermédiaires :

- `remove(pair, p)` : enlève l'élément `pair` de la liste `p`. La complexité est en $O(n)$ où n est la longueur de `p`.
- `index_min(l)` : renvoie l'indice du minimum de la liste `l`. Sa complexité est en $O(n)$.

L'idée générale de l'algorithme est de choisir à chaque étape la paire (c_1, c_2) avec le coût le plus faible dans `all_pairs`, et de retirer ensuite les cellules c_1 et c_2 de `all_pairs`. On initialise une liste vide `sol`. À chaque itération, on prend le couple le moins coûteux grâce à `index_min`, puis on retire les deux cellules de `all_pairs` avec `remove`. Ce couple est ajouté à la liste `sol`. Le processus se répète jusqu'à ce que `all_pairs` soit vide.

3.2 Pourquoi ça finit ?

Le nombre de cellules étant fini, le nombre de paires possibles est également fini. À chaque étape, on enlève au moins un élément de la liste `all_pairs`, ce qui garantit que la boucle se termine après $O(|G|)$ étapes, où $|G|$ est le nombre de paires possibles.

3.3 Complexité

La complexité de l'algorithme greedy est en $O((n \cdot m)^4)$, où n et m sont les dimensions de la grille. En effet, la création de la liste des coûts prend $O(|G|)$, de même que la recherche du minimum et la suppression d'une paire. Or, $|G|$ est $O((n \cdot m)^2)$, et l'algorithme itère cette boucle au plus $O(|G|)$ fois, d'où une complexité en $O((n \cdot m)^4)$ dans le pire des cas.

3.4 Temps de calcul

Les temps de calcul ont été calculés et représenté dans la figure 1 de l'index. Nous pouvons notamment noter une forte évolution du temps de calcul lors du passage aux grille 11 et plus

3.5 Optimalité de la méthode gloutonne

D'après le tableau 1 de l'index, l'algorithme glouton n'est pas optimal globalement car les scores diffèrent, mais seulement localement. En effet, à chaque étape, il choisit la paire qui minimise le score uniquement à cet instant. Il ne prend pas en compte les paires qui seront formées dans les étapes suivantes.
Ainsi, le solveur naïf n'est pas mauvais dans ses résultats mais n'est clairement pas optimal.

Pour cela, nous cherchons un algorithme de recherche qui retournerait une solution exacte, une première idée est l'algorithme de Ford-Fulkerson

4 Une deuxième approche par les graphes de flow

4.1 Le solveur bipartite

Le solveur bipartite s'appuie sur l'algorithme de Ford-Fulkerson, un algorithme de flux maximal. Nous initialisons un graphe bipartite dirigé entre un point de départ et un puit. Pour cet algorithme, toutes les valeurs des cellules sont égales à 1, afin de saturer la contrainte dès qu'une paire est formée, évitant ainsi que des cellules soient partagées entre deux paires.

Pour cette méthode, nous avons développé plusieurs fonctions auxiliaires :

- `Is_even(pair)` : renvoie vrai si la cellule `pair` est paire, c'est-à-dire si $i + j$ est pair. La complexité en O(1).
- `Adjacency_dictionary(p)` : construit un dictionnaire d'adjacence pour la liste `p`. La complexité est en O(n) où n est la taille de la liste p.
- `Extended_graph(C, G)` : renvoie un graphe orienté à partir de `G` et du couplage `C`.La complexité est en O(n*m) où n*m est la taille de la grille.

- `Rev(l)` : renverse la liste `l` en O(n) où n est la taille de la liste l.
- `Exists_path(G, s, p, visited, path)` : trouve le chemin de la source s au puit p dans le graphe G. La complexité est de O(V+E) où V est le nombre de noeuds et E le nombre d'arêtes dans le graphe G.
- `Edges(path)` : renvoie les arêtes qui composent le chemin `path` en O(n) où n est la taille de path.
- `Augmenting_path(C,G)` : renvoie un chemin augmentant dans le graphe étendu. La complexité est en O(V+E) où V est le nombre de noeuds et E le nombre d'arêtes dans le graphe G.
- `Symmetric_difference(path, C)` : renvoie la différence symétrique entre `path` et `C`. La complexité est en O(n*m) où n et m sont les tailles respectives de path et C.

Une fois ces fonctions auxiliaires implémentées, nous avons écrit la fonction `run()` pour trouver un couplage parfait à l'aide du graphe bipartite.

4.2 Termination et Complexité

En définissant $n = S_1$ et $m = S_1$, la complexité du solver bipartite est en $O((n+m) \cdot n)$. En effet, on peut montrer que le couplage maximal est de taille au plus $\frac{n}{2}$, ce qui assure qu'il y au plus n itérations. Par ailleurs, à chaque itération, on appelle la fonction Augmenting_path(C,G) qui est en O(n+m) (qui correspond, en fait, à la recherche d'un chemin entre deux sommets dans un graphe orienté).
A chaque itération, la taille du couplage augmente strictement d'un, ce qui assure sa terminaison. Ceci explique que cet algorithme finit pour n'importe quelle grille, même pour les grilles de taille importante (avec, de fait, un temps plus important).

4.3 Comparaison des temps entre Hongrois et Bipart

Le tableau 2 de l'index montre que le solveur Bipart est beaucoup plus efficace sur les grilles qu'elle peut résoudre. Cependant cela reste dans des cas très spécifique, le temps de calcul de l'algorithme Hongrois reste raisonnable

5 L'approche finale : L'algorithme Hongrois

5.1 Méthode Hongroise

Cette dernière approche s'appuie sur l'algorithme Hongrois qui est un algorithme d'allocation à partir d'une matrice carrée des poids. L'algorithme trouve les couples (i,j) tels que la somme des poids est minimisé. Cette fonction nécessite les fonctions auxiliaires suivantes

- `Initialisation(M)` : initialise la matrice M en soustrayant à chaque ligne son minimum, puis sur les colonnes en $O(n^2)$.
- `Unslashed_zeros(M, slashed)` : retourne un dictionnaire d où $d[i]$ contient le nombre de zéros sur la ligne i qui ne sont pas dans le dictionnaire `slashed` en en $O(n^2)$.
- `Index_min(d, visited)` : renvoie la clé de la valeur minimale dans le dictionnaire d, qui n'est pas dans `visited` en O(n*m) où m est la taille de `visited`.
- `Step1(M)` : renvoie deux dictionnaires, `outlined` et `slashed`, correspondant respectivement aux zéros encadrés et barrés dans la matrice M.
- `Step2(M)` : renvoie deux dictionnaires, `marked_row` et `marked_col`, correspondant respectivement aux lignes et colonnes marquées.
- `Step3(M)` : modifie la matrice M en soustrayant/ajoutant une valeur minimale positive.

La matrice est de plus initialisée lors de la création d'un objet de la classe. Chaque case (i, j) est liée à un entier unique comme étant sa position dans la liste `case_impaire` ou `case_paire`. La matrice est alors initialisée avec des 0, on regarde alors pour chaque paire si elle est valide et on initialise son poids à $(|p_1 - p_2|) - p_1 - p_2$, où p_1 et p_2 sont les valeurs des cases (i_1, j_1) et (i_2, j_2). Finalement, on ajoute la valeur absolue de la plus petite valeur à toute la grille pour la décaler vers les positifs. Ainsi tous les anciens poids à 0 deviennent les plus grand de la grille car ce sont ceux que l'on veut éviter.
Finalement on rajoute des colonnes ou des lignes de 0 pour rendre carré la matrice.

5.2 Termination et Complexité de la méthode hongroise

L'algorithme s'arrête lorsque tous les zéros dans la matrice sont couverts, et le couplage parfait est trouvé.
La complexité de l'algorithme est en $O(n^3)$ où n est le nombre de cases paires dans la grille. A chaque étape, l'algorithme augmente la taille du couplage d'un jusqu'à avoir une taille égale au nombre de cases paires. Ceci justifie, à priori, la terminaison de l'algorithme.

6 Pour aller plus loin

6.1 L'algorithme scipy

Un autre solveur basé sur le solveur `linear_sum_assignement` de sicpy a été rapidement implémenté. Il fonctionne de manière similaire à l'algorithme hongrois, avec en entrée une matrice mais de manière beaucoup plus rapide avec une

meilleure complexité temporelle. Avec ce module, on arrive à résoudre toutes les grilles en un temps moindre.

6.2 L'interface de jeu

Une interface graphique a été implémenté pour pouvoir jouer. La première couche graphique est réalisé avec tkinter et consiste en un éffacement des frames puis un nouveau build des boutons pour la nouvelle frame ou l'on va.
Seuls deux modes de jeux sont disponible : un mode solo player et un mode two player.
L'interface de choix des grilles ainsi que la grille jouable est réalisée à partir du module pygame et s'appuie sur le dessin, à chaque itération de la boucle principale, des différents élèments que nous plaçons aux coordonées choisies.
Cette interface est totalement dynamique et s'adapte à la taille de l'écran

6.3 Documentation

Une attention particulière a été portée sur l'intégration de docstring pour chaque fonction. Cela permet derrière une implémentation de Sphinx pour créer une documentation HTML.

6.4 Nos résultats et futures réflexions

Notre algorithme glouton marche avec toutes les grilles mais n'est pas, à priori optimal.
Notre algorithme bipart fonctionne également avec toutes les grilles avec la condition de n'avoir que des cellules dont la valeur vaut 1. A partir de la grille21, l'algorithme finit mais prend plusieurs minutes avant de terminer. Une idée pour l'améliorer pourrait être de construire le graphe étendu une seule fois (avant la boucle tant que) puis de le modifier en fonction du couplage obtenu. Dès lors, on n'aurait plus à re-parcourir la liste des `all_pairs` à chaque itération.
Enfin, notre algorithme hongrois fonctionne sur toutes les grilles jusqu'à la grille 19 où il semble bloqué. Une idée serait de re-implémenter la méthode hongroise puisqu'en utilisant le module scipy, l'algorithme termine.

6.5 Idée de variation du jeu

Une piste de recherche de variation des règles du jeu pourrait être de changer les règles d'appariement par couleur, ainsi nous pouvons imagine un cycle bleu s'apparie avec vert qui s'apparie avec rouge qui s'apparie avec bleu, de même les blancs s'apparient avec tous sauf noirs et les noirs avec aucune

7 Index

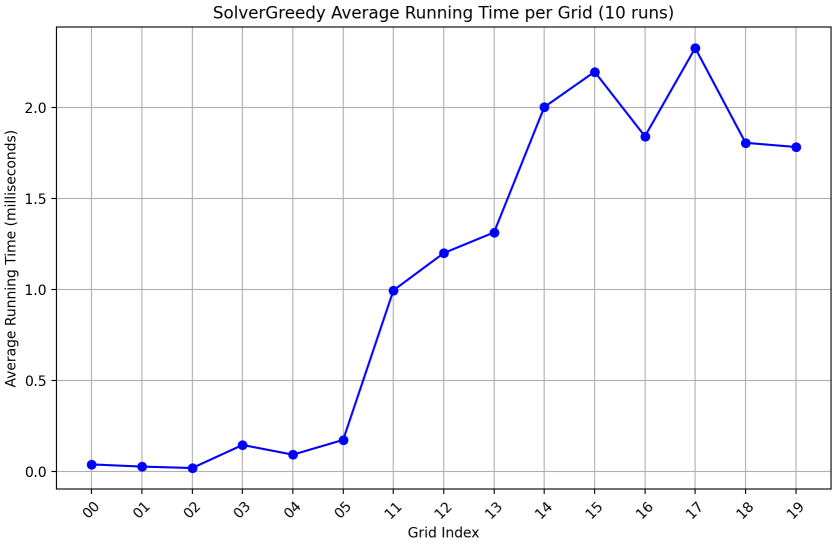

Figure 1: Temps de calcul d'une solution selon la grille

Score selon la grille	Grid00	Grid01	Grid02	Grid03	Grid04	Grid05	Grid15	Grid16	Grid17	Grid18
Greedy	14	8	1	2	6	41	31	32	280	273
Optimum	12	8	1	2	4	35	21	28	256	259

Table 1: Score de la solution trouvé par le solveur Greedy pour certaine grille

Temps execution	**Grid00**	**Grid01**	**Grid02**	**Grid03**	**Grid13**	**Grid14**	**Grid15**	**Grid16**	**Grid18**
Bipart	NA	NA	0.0001	0.0003	0.01	0.015	0.019	0.014	NA
Hongrois	0.02	0.02	0.02	0.01	4.21	7.43	6.61	6.61	30.87

Table 2: Temps d'éxecution des solveurs Bipart et Hongrois