Rapport - Projet de programmation

Une histoire de grilles et de couleurs

Auteurs : Mohamed Iyed MOKLINE, Olivier DE BOISSIEU

Référent : Simon MAURAS

Avril 2025

Table des matières

Introduction

Ce rapport présente les choix algorithmiques que nous avons réalisés lors de la résolution du problème qui s'offrait à nous, ainsi que leurs complexités. Nous y détaillons de plus les extensions que nous avons ajoutées, ainsi que les résultats obtenus.

Structure du code

Au sein du dossier code :

— grid.py : Implémentant les fonctions demandées lors des séances 1 et 2 main.py :

— solver.py : implémentation des différents algorithmes de résolution, dont l'algorithme glouton, l'algorithme de résolution optimal (Hongrois) et dans l'algorithme de flot mis à place lors de la deuxième séance

— visual grid.py : implémentation d'une représentation graphique du jeu, et de trois modes de jeu qui sont 1 joueur, 2 joueurs, 1 joueur VS AI

— minmax.py : implémentation de l'algorithme minmax utilisé dans le mode de jeu adversarial contre l'IA

Au sein du dossier input :

— Les grilles de test fournies

— Une grille notopti.in qui présente un contre exemple à l'optimalité de l'algorithme glouton introduit en séance 1.

Au sein du fichier tests :

— test_grid_from_file.py : un fichier python implémentant les tests unitaires réalisés au cours du développement pour vérifier les fonctions implémentées

1 Algorithmes de résolution

Au cours de ce projet, nous avons introduit plusieurs algorithmes afin de résoudre le problème. Après une approche simpliste visant à proposer une méthode naïve de résolution (Algorithme glouton), nous avons traité un cas particulier du problème avec un algorithme de flot, avant de trouver une solution optimale basée sur l'application de l'algorithme hongrois en ramenant notre problème à un problème d'assignement.

Ces trois solutions ont été implémentées dans le fichier solver.py sous forme d'héritiers de la classe Solver fournie.

1.1 Algorithme glouton SolverGreedy

Dans une première approche, nous nous sommes concentrés sur une approche gloutonne : puisque le but du jeu est de minimiser le score, lequel est calculé comme la somme des valeurs des cases non choisies, et la valeur absolue de la différence de la valeur des paires de cases choisies ; notre algorithme considère les paires qu'il peut sélectionner, et choisit donc celle avec la plus petite différence.

1.1.1 Présentation

Pour cela, on a construit la classe SolverGreedy, dont la fonction *run* va considérer chaque paire possible, pas encore sélectionnée, et considérer son coût. Elle va ensuite choisir celle dont le coût est minimal, puisque c'est ce que l'on veut. Si le coût de la prendre (coût de la différence des valeurs) est plus faible que celui de ne pas la prendre (somme des valeurs), alors elle est sélectionnée, et l'algorithme continue d'évaluer les paires. Sinon, l'algorithme s'arrête, puisque toute autre paire choisie ne ferait qu'augmenter le score que nous cherchons à minimiser. Cet algorithme est récursif dans son implémentation.

1.1.2 Non optimalité

Cet algorithme n'est pas optimal, en effet, par son caractère glouton, il est évident que le choix d'une paire de coût minimal peut se faire au détriment d'un choix plus judicieux à l'avenir.

Pour preuve, voici une grille non optimale pour cet algorithme (dans le dossier input, c'est la grille *notopti.in*). Ci-dessous une représentation graphique :

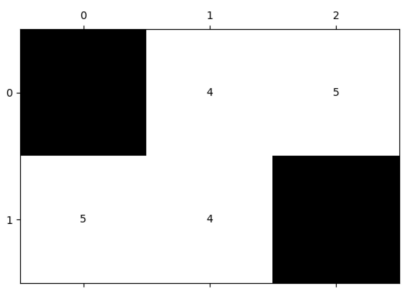

FIGURE 1 – Grille non optimale pour l'algorithme glouton

Notre algorithme va d'abord sélectionner les cases du centre, de coordonnées (0, 1) et (1, 1), et puis il ne pourra plus en sélectionner, pour un score final de 10.

Alors que de façon optimale, ce sont les deux paires ((0, 1),(0, 2)) et ((1, 0),(1, 1)) qui devraient être sélectionnées, pour un score de 2.

1.1.3 Complexité

Par l'usage qu'il fait de la récursivité et du tri de liste, ainsi que les appels multiples à la fonction *all_pairs* du module *grid*, cet algorithme a de plus une complexité élevée : $O((n \times m)^3 log(n \times m))$.

1.2 Algorithme de Ford-Fulkerson SolverMatching

Dans le cas plus restreint où toutes les cases ont pour valeur 1, nous avons été invités à utiliser l'algorithme de Ford-Fulkerson, en assimilant ce problème à un problème de flot.

L'algorithme implémente une approche basée sur la recherche en profondeur (DFS) pour trouver un appariement biparti maximal. La complexité principale provient de deux étapes :

(1) la construction du graphe biparti, qui explore chaque cellule de la grille et ses voisins en $O(nm)$, et

(2) la recherche de l'appariement maximal par DFS, qui dans le pire des cas peut nécessiter $O(nm)$ appels récursifs avec un parcours de $O(nm)$ pour chaque cellule,

conduisant à une complexité globale en $O((nm)^2)$.

1.3 Algorithme hongrois SolverMaxWeightMatching

1.3.1 Présentation générale

Le problème proposé consiste à former des paires de cellules valides sur une grille colorée, tout en maximisant une fonction de score. Chaque cellule est caractérisée par une couleur et une valeur entière positive. Certaines paires sont interdites (par exemple, celles incluant une cellule noire), d'autres sont restreintes à l'adjacence immédiate, et des règles de compatibilité basées sur les couleurs s'appliquent.

Pour résoudre ce problème dans le cas général, où les valeurs peuvent être arbitraires, nous avons choisi d'implémenter l'algorithme hongrois. Cet algorithme résout de manière optimale les problèmes d'affectation pondérée sur des graphes bipartis, ce qui correspond exactement à notre modélisation du problème.

1.3.2 Réduction à un problème de matching maximal pondéré

La grille est transformée en un graphe biparti pondéré. On répartit les cellules en deux ensembles disjoints selon la parité de la somme de leurs indices $(i + j)$:

— U = cellules avec $(i + j)$ pair

— V = cellules avec $(i + j)$ impair

Chaque arête représente une paire de cellules adjacentes et compatibles, avec un poids calculé comme :

$$\text{Poids}(u, v) = 2 \times \min(\text{valeur}(u), \text{valeur}(v))$$

Le but est de trouver un ensemble de paires disjointes (matching) maximisant la somme des poids. Comme l'algorithme hongrois minimise les coûts, on transforme la matrice de poids $W(i, j)$ en une matrice de coûts $C(i, j) = \text{maxWeight} - W(i, j)$ afin de maximiser le matching pondéré.

1.3.3 Variante : suppression de la contrainte d'adjacence pour les cellules blanches

Nous avons également étudié une variante du problème : les cellules blanches peuvent être appariées entre elles même si elles ne sont pas adjacentes.

Cela ajoute de nombreuses arêtes supplémentaires dans le graphe biparti (entre toutes les paires blanches valides), augmentant potentiellement le score obtenu. Cette extension est intégrée dans le solveur `SolverMaxWeightMatching2`.

1.3.4 Résultats expérimentaux

Nous avons comparé les scores obtenus avec et sans la variante sur plusieurs grilles :

Instance	Score standard	Score variante (blancs non adjacents)
grid00	12	12
grid11	26	2
grid12	19	3
grid13	22	6
grid14	27	21
grid15	21	17
grid17	256	228
grid18	259	237
grid19	248	208

La variante produit toujours un meilleur score comme attendu.

1.3.5 Performances et complexité

L'algorithme hongrois a une complexité théorique en $\mathcal{O}(N^3)$, où N est le nombre de nœuds du graphe biparti.

Dans notre cas, $N \approx n \times m$, la taille de la grille. On obtient donc une complexité globale :

$$\mathcal{O}((n \times m)^3)$$

Ainsi, le temps de calcul augmente très rapidement avec la taille de la grille. Nous avons constaté que le programme devient significativement lent à partir de la grille `grid21`, en particulier pour la version étendue avec les appariements blancs non adjacents, car elle rend la matrice plus dense.

1.3.6 Conclusion

Grâce à une modélisation précise du problème et à l'utilisation de l'algorithme hongrois, nous obtenons une solution optimale et généralisable pour différentes variantes du problème.

La version avec contraintes classiques reste la plus efficace en temps et souvent en qualité.

2 Extensions

2.1 Implémentation graphique du mode 1 joueur

2.1.1 Mode graphique interactif

Dans le cadre du projet, nous avons implémenté un mode graphique interactif permettant à un joueur de former lui-même des paires sur la grille. Cette extension, développée avec `Pygame`, permet d'explorer manuellement le problème et de comparer son intuition aux résultats d'un algorithme exact.

2.1.2 Choix algorithmique

Ce mode n'intègre pas de stratégie algorithmique complexe, mais repose sur une logique utilisateur simple :

— Le joueur sélectionne deux cellules ;

— Si la paire est valide (adjacente, autorisée, non utilisée), elle est ajoutée ;

— Une ligne visuelle (jaune) relie les deux cellules.

L'interface contient deux boutons fonctionnels : **Annuler** (pour retirer la dernière paire) et **Terminer** (pour lancer l'évaluation des performances).

2.1.3 Résultats obtenus

À la fin de la session, deux scores sont affichés :

— Le score utilisateur, calculé sur les paires choisies ;

— Le score optimal, obtenu automatiquement via le `SolverMaxWeightMatching` (algorithme hongrois).

Cela permet de mesurer l'écart entre les décisions humaines et la solution optimale.

2.1.4 Intérêt de l'extension

Cette extension est précieuse pour :

— Comprendre visuellement les contraintes du problème ;

— Expérimenter des stratégies humaines (gourmande, par valeurs élevées) ;

— Comparer intuition et optimalité.

Elle constitue donc un complément pédagogique utile, bien que non essentiel à la résolution algorithmique du problème.

2.2 Mode deux joueurs

Nous avons choisi de réaliser la question 4 (Séance 4) et donc d'implémenter un mode deux joueurs, et d'y ajouter ensuite un algorithme contre lequel jouer (section suivante).

Nous avons étendu l'interface graphique pour permettre une partie à deux joueurs. À tour de rôle, chaque joueur sélectionne une paire de cellules valide. Les paires sont enregistrées et affichées à l'écran, et chaque joueur conserve ses propres paires. Une fois la partie terminée, chaque score est calculé comme la somme des valeurs des cellules

appariées. Le gagnant est celui qui obtient le score total le plus faible, conformément à l'objectif de minimisation.

Cette version introduit une dimension compétitive et stratégique, et permet de tester les décisions humaines dans un cadre de jeu équilibré.

2.3 Mode contre l'AI : algorithme de minimax

Afin d'introduire un algorithme adversarial, nous nous sommes basés sur l'algorithme du *Minimax*, classiquement utilisé sur des problèmes de ce genre, mais qui a le désavantage d'étudier toutes les possibilités d'un jeu. (Le jeu ne terminant que lorsqu'il n'y a plus de possibilités d'action.)

2.3.1 Description

L'algorithme étant classique, et son implémentation directement conforme à la description qui en est faite dans la littérature, nous n'en rappellerons que les grandes lignes : il consiste à évaluer récursivement les actions possibles, en considérant que l'adversaire cherche à maximiser son score (ici le score est défini de telle sorte qu'il est maximum si le joueur gagne et minimal si son adversaire gagne). Le meilleur choix est alors celui qui minimise les pertes du joueur tout en supposant que l'adversaire cherche au contraire à les maximiser.

2.3.2 Choix de l'heuristique

Pour cela, il va donc évaluer récursivement toutes les actions, en leur attribuant un score selon une heuristique correspondante à celle décrite ci-dessus (Victoire :1, Défaite :-1, par exemple).

Notre choix d'heuristique a été le suivant : ($v(c)$ la valeur de la cellule c dans la grille)

$$\Big(\sum_{c_1,c_2 \in \text{Couples choisis par l'adversaire}} |v(c_1) - v(c_2)|\Big) - \Big(\sum_{c_1,c_2 \in \text{Couples choisis par l'algorithme}} |v(c_1) - v(c_2)|\Big)$$

Ainsi, notre algorithme cherche bien à avoir le score de grille minimal, soit en augmentant le coût pour l'adversaire du choix de ses combinaisons (membre de gauche), soit en minimisant le coût de ses propres choix.

De ce fait, cet algorithme est très efficace sur les grilles de petite taille (4x4, etc.), mais devient très vite très long dès lors que la taille augmente.

L'une des façons de résoudre ce problème est d'implémenter une limite de récursion (MAXIMUM_RECURSION_DEPTH) pour éviter que l'algorithme ne teste toutes les possibilités : lorsqu'il dépasse cette limite, il renvoie une heuristique intermédiaire en fonction de la grille. C'est ce que nous avons fait. Nous avons conservé la même heuristique comme heuristique intermédiaire, jugeant que c'était aussi une bonne appréciation de la valeur des choix réalisés à ce stade.

Annexe : Images de l'extension visuelle

FIGURE 2 – Menu de l'implémentation graphique

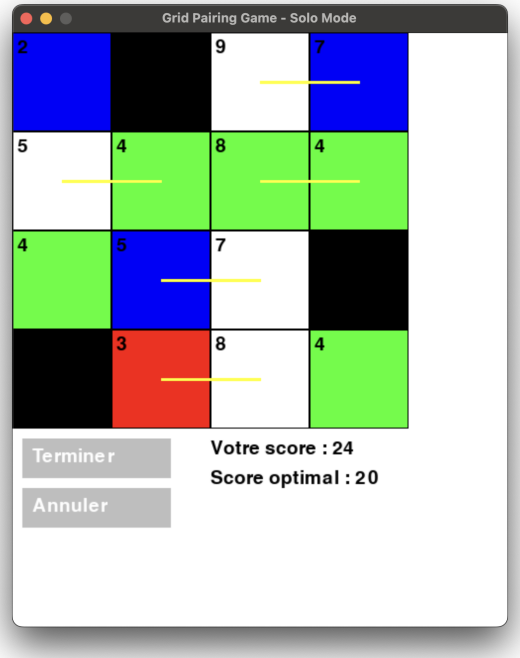

FIGURE 3 – Mode à 1 joueur : fin de jeu après avoir cliqué sur Terminer

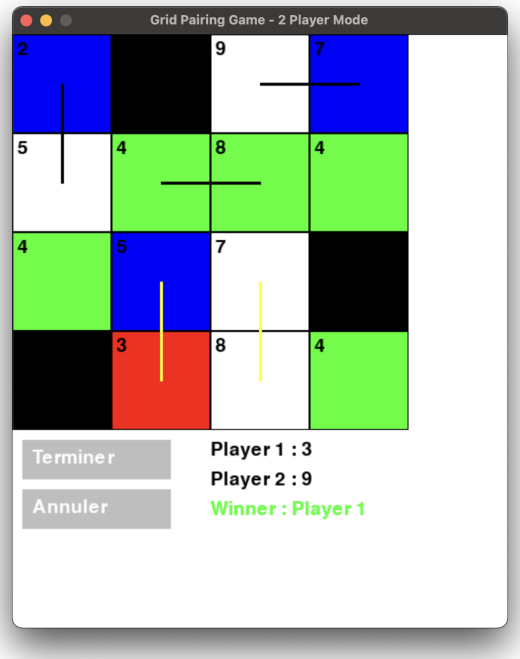

FIGURE 4 – Mode à 2 joueurs

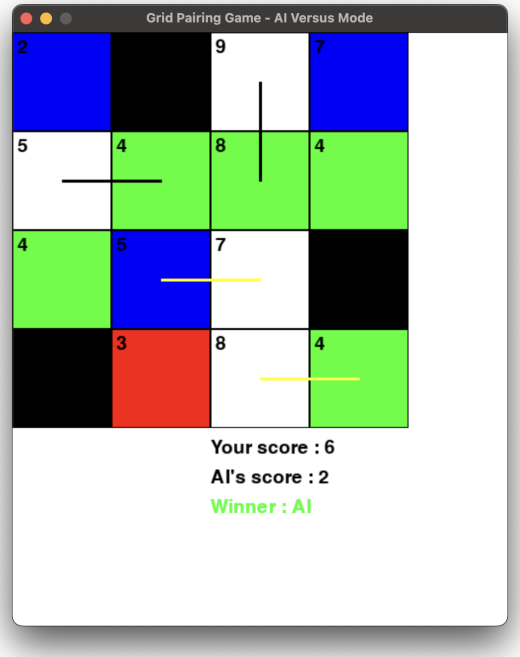

FIGURE 5 – Mode contre l'IA : victoire de l'algorithme minimax