Rapport du projet de programmation

Ralph NADER et Maxime HEBERT

2 avril 2025

Introduction

Dans ce projet, nous nous attaquons à un problème d'optimisation sur une grille, où le but est de trouver l'appariement optimal des cellules (ie celui minimisant le score). Il se déroule sur une grille de taille $n \times m$, avec $n \geq 1$ et $m \geq 2$, où chaque cellule possède deux attributs : une couleur et une valeur (un entier positif).

Les couleurs des cellules sont codées comme suit :

0 : blanc ('w'), 1 : rouge ('r'), 2 : bleu ('b'), 3 : vert ('g'), 4 : noir ('d')

Les règles pour former des paires de cellules sont définies de la façon suivante. Une cellule ne peut être impliquée que dans **une seule paire** à la fois et ne peut être appariée qu'à une celle **adjacente**. À cela s'ajouter un appairement suivant des couleurs (défini plus tard).

Une fois les paires formées conformément aux règles d'appariement, le score est calculé de la manière suivante :

1. Pour chaque paire, on prend la différence en valeur absolue entre les valeurs des deux cellules et on l'ajoute au score.

2. La valeur des cellules non appariées (sauf les cellules noires) sont ajoutées à ce score.

Le but du jeu est donc de minimiser ce score tout en respectant toutes les contraintes d'appariement.
Nous nous proposons ensuite d'aller au-delà en changeant les règles du jeu puis en proposant un model à deux joueurs!

SOMMAIRE

1 Description du jeu

1.1 Structure du code

Notre code est organisé avec différents fichiers séparant la gestion de la grille de jeu, la résolution de la grille, ou encore son implémentation graphique. La structure est la suivante :

- **grid.py** : Ce fichier déclare la classe `Grid` et implémente les fonctions utiles au respect des règles.
- **solver.py** : Ce fichier contient l'implémentation des solveurs, que nous détaillerons plus bas.
- **test_grid_from_file.py** : Ce fichier teste les fonctions implémentées dans les autres fichiers.
- **pygames.py** : Ce fichier implémente l'interface graphique du jeu.

1.2 Classe grid

1.2.1 Attributs principaux

La classe `Grid` possède plusieurs attributs qui définissent ses propriétés :

- **n** : Nombre de lignes dans la grille.
- **m** : Nombre de colonnes dans la grille.
- **color** : Une matrice $n \times m$ contenant les couleurs des cellules, stockées sous forme d'entiers (de 0 à 4).
- **value** : Une matrice $n \times m$ contenant les valeurs numériques associées aux cellules.
- **colors_list** : Une liste associant chaque valeur de `color` à une couleur spécifique comme définies dans l'introduction.

1.2.2 Méthodes principales

La classe `Grid` propose plusieurs méthodes pour manipuler et gérer la grille que nous présentons ci-dessous.

1.2.3 Initialisation et affichage

- **__init__(n, m, color=[], value=[])** : Initialise une grille de n lignes et m colonnes. Par défaut, toutes les cellules ont la couleur blanche (0) et la valeur 1, sauf si des paramètres spécifiques indiquent le contraire.
- **__str__()** : Retourne une représentation textuelle de la grille, affichant la couleur et la valeur de chaque cellule.
- **plot()** : Génère une représentation graphique de la grille à l'aide de matplotlib, en coloriant chaque cellule selon sa couleur et en affichant sa valeur au centre.

1.2.4 Manipulation et vérification des cellules

- **is_forbidden(i, j)** : Vérifie si la cellule située à la position (i, j) est noire (4), ce qui la rend inaccessible.
- **cost(pair)** : Calcule le coût entre deux cellules définies par une paire $((i_1, j_1), (i_2, j_2))$. Ce coût est donné par la valeur absolue de la différence entre leurs valeurs.
- **compatible_color(paire1, paire2)** : Vérifie si deux cellules sont compatibles en fonction de leurs couleurs. Par exemple :
  - Les cellules vertes (3) ne peuvent se lier qu'à des blanches (0) ou d'autres vertes.
  - Les cellules bleues (2) et rouges (1) peuvent s'associer entre elles ou avec une cellule blanche (0).
  - Les cellules blanches peuvent s'associer à toutes les couleurs sauf le noir.

1.2.5 Génération et lecture des grilles

- **all_pairs()** : Génère une liste de toutes les paires de cellules voisines qui respectent les règles de compatibilité.
- **grid_from_file(file_name, read_values=False)** : Permet de charger une grille à partir d'un fichier texte. Le fichier doit contenir :
  - Une ligne avec $n\,m$ (dimensions de la grille).
  - n lignes contenant les couleurs de chaque cellule.
  - Optionnellement, n lignes supplémentaires avec les valeurs des cellules.

2 Les différents solveurs

2.1 Solveur glouton ('Greedy')

Pour commencer, nous avons choisi d'implémenter un solveur glouton. Ce solveur suit une approche myope, en prenant des décisions uniquement basées sur le gain immédiat. À chaque étape, il sélectionne la paire de cellules qui semble offrir la meilleure valeur selon un critère prédéfini, sans chercher à optimiser le résultat global du problème. Dans notre cas, il choisit la paire qui minimise le score si celle-ci est prise, c'est-à-dire celle qui donne le score le plus bas dans le cas où le jeu s'arrêterait après ce coup.

Ce modèle nous permet d'obtenir une solution en $O((n \cdot m)^2)$, mais il n'est pas optimal. Par exemple, pour la grille 00, le score obtenu est de 14, alors qu'il est possible d'atteindre un score de 12 en prenant les paires suivantes :

$$\{((0,0),(1,0)),((0,1),(0,2)),((1,1),(1,2))\}$$

Il est possible de trouver la solution optimale en testant toutes les combinaisons possibles et en calculant le score pour chacune d'elles. Cela nécessite de vérifier $2^{(n \cdot m/2)}$ combinaisons possibles, ce qui donne une complexité de $O(n \cdot m \cdot 2^{(n \cdot m/2)})$.

2.2 Algorithme de Ford-Fulkerson

2.2.1 Méthode optimale basée sur les graphes

Nous proposons une méthode optimale basée sur les graphes, valable lorsque toutes les cellules de la grille ont une valeur égale à 1. L'objectif est de maximiser le nombre de paires sélectionnées, car chaque paire n'a pas de coût, tandis que les cellules non appariées ont un coût de 1.

Le problème se modélise comme un appariement dans un graphe biparti, où les cellules sont divisées en deux ensembles : les cellules paires (où $i+j$ est pair) et les cellules impaires (où $i+j$ est impair). Une cellule paire ne peut être appariée qu'avec une cellule impaire, en respectant les contraintes d'adjacence.

Un graphe biparti est alors construit, avec les cellules paires à gauche et les cellules impaires à droite, et des arêtes sont ajoutées entre celles pouvant être appariées. L'objectif est de trouver un appariement maximal, soit un sous-ensemble d'arêtes où chaque sommet est connecté à une seule arête.

2.2.2 Résolution avec l'algorithme de Ford-Fulkerson

Pour résoudre ce problème, nous utilisons l'algorithme de Ford-Fulkerson, qui permet de trouver la solution optimale. L'algorithme fonctionne de la manière suivante :

1. Construction d'un graphe résiduel avec des capacités initiales correspondant aux contraintes du problème.
2. Recherche d'un chemin augmentant dans le graphe (par exemple via DFS ou BFS).
3. Mise à jour du flux en augmentant ou en réduisant la capacité des arêtes sur ce chemin.
4. Répétition de ces étapes jusqu'à ce qu'aucun flux supplémentaire ne puisse être ajouté.

Cette méthode garantit une solution optimale avec une complexité de $O((n \cdot m)^2)$ dans le pire des cas.

2.3 Solveur général - Maximum weight matching

2.3.1 Présentation

Dans le cadre d'un graphe pondéré, le problème de correspondance de poids maximum consiste à en trouver une dans laquelle la somme des poids est maximisée. Parmi les différents algorithmes répondant efficacement à ce problème, on se propose de travailler avec celui de **maximum weight matching**.

2.3.2 Implémentation

Dans un premier temps, nous devons transformer notre grille en un graphe afin de pouvoir exécuter notre algorithme de correspondance de poids maximum. Comme l'algorithme cherche à maximiser le poids des arêtes, nous devons définir le poids des arêtes de manière à ce que :

$$\arg\max \sum w(i,j) = \arg\min \text{score}$$

Ainsi, nous avons :

$$w(i,j) = \text{valeur de la cellule } i + \text{valeur de la cellule } j - \text{score}(i,j)$$

(La preuve est disponible en annexe)

Nous avons décidé d'utiliser la fonction déjà implémentée dans la librairie `networkX`. Avec cette implémentation, nous trouvons une solution optimale, même dans des cas où les poids sont hétérogènes, avec une complexité de $O((n \cdot m)^3)$.

La fonction `nx.max_weight_matching()` de NetworkX utilise l'algorithme de Blossom pour résoudre le problème de l'appariement de poids maximal dans un graphe général. Voici un résumé de son fonctionnement :

1. **Recherche d'un appariement initial** : L'algorithme commence par un appariement initial, souvent arbitraire, puis l'améliore progressivement en ajoutant et retirant des arêtes via des chemins augmentants. Cela permet de commencer à maximiser la somme des poids des arêtes de l'appariement.

2. **Gestion des cycles impairs ("Blossoms")** : Si des cycles impairs (blossoms) sont présents, l'algorithme les "contracte" en un sommet unique. Cela simplifie le graphe en éliminant les cycles complexes, tout en maintenant les relations essentielles entre les sommets.

3. **Augmentation de l'appariement** : L'algorithme continue de rechercher des chemins augmentants, chaque fois améliorant l'appariement en augmentant la somme des poids, jusqu'à ce que l'appariement ne puisse plus être amélioré.

4. **Retour de l'appariement optimal** : Une fois l'appariement optimal atteint, l'algorithme retourne l'ensemble des arêtes avec la somme des poids maximale possible.

3 Pour aller pour loin

3.1 Variante du jeu

Une autre approche consiste à envisager une variante où la contrainte d'adjacence est levée pour les paires de cellules blanches. Autrement dit, une cellule blanche peut être appariée avec des cellules adjacentes de n'importe quelle couleur (à l'exception des cellules noires), ou même avec n'importe quelle autre cellule blanche, qu'elle soit adjacente ou non.

Pour résoudre ce problème, il suffit d'ajouter les arêtes supplémentaires dans le graphe pour tenir compte de ces nouvelles règles. Une fois le graphe adapté, il est possible d'utiliser le solveur de maximum weight matching pour trouver la solution optimale.

3.2 Différents modes de jeu

Différents modes de jeu sont présents dans ce projet :

- Le traditionnel joueur contre joueur dont l'utilité est principalement réservée au loisir.
- Le joueur contre environnement. Dans ce cas, il est nécessaire d'implémenter un algorithme capable de jouer de manière optimale. Plutôt que de calculer les attracteurs, ce qui serait trop complexe compte tenu du grand nombre de coups possibles, nous avons choisi de nous orienter vers un algorithme minimax.
- IA contre IA : où les algorithmes se comportent de façon identique au cas de l'environnement dans le mode "1 joueur".

3.2.1 Approche heuristique - Minimax

L'algorithme Minimax est une méthode classique pour résoudre des jeux à somme nulle. Il repose sur une évaluation récursive des états de jeu afin de choisir la meilleure action en anticipant les coups de l'adversaire.

Puisque l'ensemble des états S est trop grand pour être exploré exhaustivement, les techniques utilisées consistent à approximer une stratégie gagnante, en guidant la recherche de coups à jouer à l'aide d'une heuristique. Pour cela, on introduit la notion de gain (ou score) : chaque état $s \in S$ est doté d'une valeur score(s), qui en général est :

- 0 si c'est un match nul ;
- $+\infty$ si c'est une position gagnante pour Alice ;
- $-\infty$ si c'est une position gagnante pour Bob ;

Si jamais les positions gagnantes n'ont pas été identifiées, ou que le score est une fonction plus complexe, le score de s n'est alors connu que pour les états terminaux, et est approximé à l'aide d'une heuristique h pour les autres sommets. Intuitivement :

- si $h(s)$ est proche de 0, la partie semble équilibrée ;
- si $h(s) > 0$, Alice semble gagner ;
- si $h(s) < 0$, Bob semble gagner.

Ainsi, l'IA va essayer de maximiser son gain, et le joueur va essayer de le minimiser. Dans notre cas, nous prenons $h(s) =$ score de IA $-$ score du joueur. Ainsi, notre heuristique h nous donne bien une approximation de la situation.

Le nombre de coup possible à jouer peut être très grand, donc pour optimiser le temps de calcul nous mettons en place un élagage $\alpha - \beta$. L'élagage $\alpha - \beta$ est une optimisation de l'algorithme Minimax permettant en pratique d'éviter l'évaluation d'un grand nombre de nœuds, en coupant des branches dont on sait qu'elles ne participeront pas à la stratégie optimale. Plus précisément, on va parfois détecter qu'il est inutile de calculer eval(n) pour certains petits-fils n du nœud courant. Pour cela, on rajoute à l'algorithme Minimax deux arguments : α et β, tels que, lors du calcul de eval(n), on ait $\alpha \leq \text{eval}(n) \leq \beta$. Au départ, on lance l'algorithme avec $\alpha = -\infty$ et $\beta = +\infty$, et ces valeurs vont :

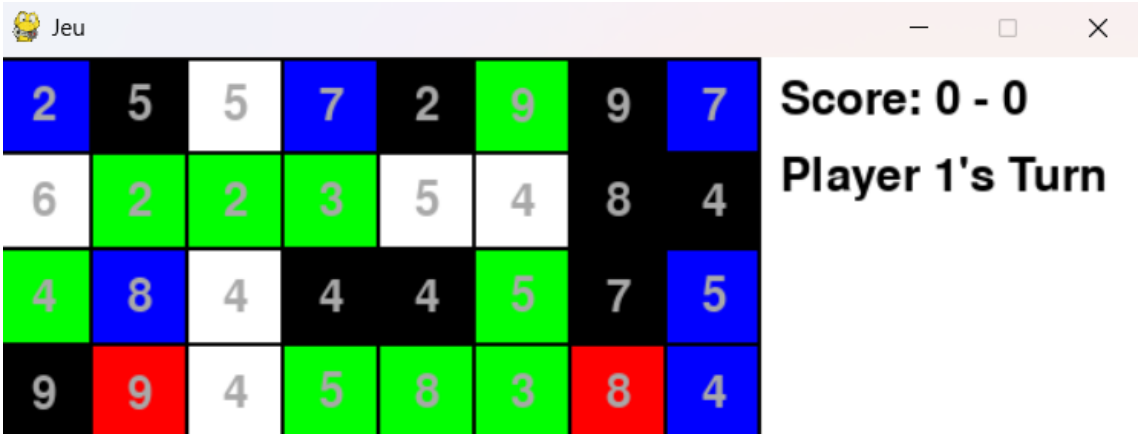
- être raffinées et passées à travers les appels récursifs ;
- être utilisées pour ne pas effectuer certains appels récursifs.

Pour ce dernier point, on va effectuer ce qu'on appelle des coupures α et des coupures β.

3.3 Implémentation du jeu sur pygame

On se propose maintenant d'aborder l'aspect graphique du jeu implémenté à l'aide du module *pygame*. L'idée ici est bien sûr de fournir un confort visuel et esthétique quelque ce soit le mode de jeu (IA contre IA, 1 joueur ou encore joueur contre joueur).

Parmi les différentes fonctionnalités dont disposera l'interface, il y aura la possibilité de suivre l'état du jeu en temps réel, et encore d'interagir directement avec les cellules dans les modes de jeu qui le requièrent. Cela facilite également la compréhension des actions possibles/comportement des algorithmes.



D'autres images sont disponibles en annexe.

4 Annexe

4.1 Preuve du poids des arêtes

$$
\begin{aligned}
\arg\min_M \text{score}(i,j) &= \arg\max_M -\text{score}(i,j) \\
&= \arg\max_M -\sum_M s(i,j) - \sum_{M^c} v(i) + v(j) \\
&= \arg\max_M -\sum_M cost(i,j) + \sum_M v(i) + v(j) - \text{Somme totale} \\
&= \arg\max_M \sum v(i) + v(j) - cost(i,j)
\end{aligned}
$$

4.2 Interface graphique

4.3 Comparaison du temps d'éxécution

Grille	Glouton	Ford Fulkerson	Maximum weight matching
11	0.01	0.04	0.01
12	0.01	0.05	0.01
13	0.01	0.06	0.01
14	0.01	0.07	0.01
15	0.02	0.08	0.01
16	0.02	0.06	0.01
21	.	.	41.79
22	.	.	42.83

Table 1: Comparaison des temps d'éxécution des différrents solveurs (en seconde)

Le solveur basé sur le maximum weight matching est plus rapide que l'algorithme de Ford-Fulkerson, bien que sa complexité théorique soit plus élevée. Cette différence de performance s'explique par le fait que des bibliothèques comme NetworkX, qui implémente cet algorithme, utilisent des langages bas niveau tels que C ou Cython. Ces langages sont beaucoup plus rapides que du Python pur, ce qui permet à l'algorithme de s'exécuter plus rapidement, même avec une complexité théorique plus importante.

4.4 Comparaison du score

Grille	Glouton	Ford Fulkerson	Maximum weight matching
11	26	26	26
12	21	19	19
13	22	22	22
14	29	27	27
15	23	21	21
18	286	.	256
21	.	.	1686
22	.	.	1689

Table 2: Comparaison des scores des différrents solveurs