Rapport du projet de programmation

Allan PARIENTE, Joël KHAYAT

Date de rendu : 2 avril 2025

Table des matières

1 Précisions sur le rendu

Le rendu est composé de trois dossiers `assets`, `code`, `input` et `tests`.
Le dossier `assets` contient tous les fichiers (images) nécessaires pour le programme.
Le dossier `tests` contient des fichiers tests chacun permettant de tester une classe.
Le dossier `input` contient des grilles pouvant être utilisées pour des tests unitaires.
Le dossier `code` contient six modules :

- `main.py` contient les instructions permettant de lancer le code sur certains de tests unitaires présents dans le dossier `test`.
- `grid.py`, où sont développées toutes les méthodes basiques de gestion de la grille.
- `solver.py` contenant les différents solvers, implémentés sous forme de classe.
- `ford_fulkerson_algo.py` contenant les fonctions permettant d'exécuter l'algorithme de FORD-FULKERSON décrit en annexe.
- `hungarian_algo.py` contenant les fonctions permettant d'exécuter l'algorithme de KHUN-MUNKRES (ou hongrois) décrit en annexe. Il s'agit d'une version débogguée et commentée inspirée d'une version trouvée sur Internet, que nous nous sommes appropriée pour passer à l'échelle.
- `graphic_version.py` contenant la classe `PlotResolution` permettant d'afficher les résolutions graphiques des grilles par les solvers ou une interface interactive permettant à un joueur de jouer au jeu.
- `hungarian_version_intermediaire.py` qui est une annexe proposant un code supplémentaire de l'algorithme Hongrois développé par nos soins, reposant sur la définition originale, mais dans une version intermédiaire qui a été abandonnée parce qu'elle ne passait pas à l'échelle des grandes matrices de coût (`shape=(20++,20++)`).

La méthode pour exécuter un solver sera toujours la méthode `run()`. Plusieurs types d'affichages graphiques sont proposés, tous développés à partir de la librairie Python `pygame`. L'appel à la méthode `plot()` de la classe `Grid()` présente dans le module `grid` permet l'affichage d'une grille. L'appel à la méthode `launch()` de la classe `PlotResolution()` affiche un menu dans lequel l'utilisateur choisit un solver ou bien choisi de jouer lui même. Dans le premier cas s'affiche la résolution progressive de la grille par le solver choisi, dans le deuxième cas, la grille s'affiche et l'utilisateur résout lui-même la grille en cliquant sur les cases qu'il souhaite choisir.

L'ensemble des codes doit être exécuté depuis le dossier racine.

2 Description des différents modules développés

Dans le code on utilise très souvent des ensembles (type `set`) et des dictionnaires (type `dict`) au lieu d'utiliser des listes. De cette manière les tests d'appartenance aux ensembles et de recherche dans les dictionnaires est en $O(1)$ et leur implémentation permet d'accélérer grandement les algorithmes de ce projet à l'exécution.

2.1 Le module Grid

On crée un objet de la classe `Grid()` par la donnée du nombre de lignes de la grille (`int`), du nombre de colonnes (`int`), les couleurs des cases (`list[list[int]]`) (où les entiers sont entre 0 et 4), et les valeurs des cases (`list[list[int]]`). `removed` est une

liste représentant l'ensemble des cellules déjà choisies lors de la résolution de la grille. `plot_removed` est une liste représentant l'ensemble des cellules ne devant pas être affichées lors de l'affichage graphique dans la classe `PlotResolution()` du module `solver` car déjà choisies. `rect_list`, `cells_list` et `selected_cells` sont des attributs utilisés lors de l'affichage graphique afin de lister les objets `Pygame.Rect`, les cellules, et les cellules sélectionnées à une étape.

La méthode `plot()` permet d'afficher l'objet de la classe `Grid` avec la librairie `pygame`. La sous-fonction `draw_grid()` est définie dans la méthode `plot()` permet de mettre à jour la fenêtre affichée afin de dessiner la grille avec `pygame`. Un logo et un texte sont aussi affichés en dessous de l'affichage graphique. La variable `running` permet de détecter si l'utilisateur a fermé la fenêtre (en cliquant sur la croix ou en appuyant sur la touche `f` du clavier).

La méthode `self.is_forbidden(i,j)` retourne `True` si et seulement si la cellule `(i,j)` ne peut pas être sélectionnée (si c'est une cellule noire, et si elle a déjà été sélectionnée).

La méthode `cost(pair)` détermine le coût de la sélection de la paire.

La méthode `all_pairs()` détermine la liste des paires pouvant être sélectionnées. Elle fait appel à une sous-fonction `is_color_matching_forbidden(i1,j1,i2,j2)` qui renvoie `True` si et seulement si la paire `((i1,j1),(i2,j2))` ne peut pas être sélectionnée pour cause de mauvais appariement de couleurs.

La méthode `grid_from_file(file_name, read_values)`, (où `False` est par défaut affecté à `read_values`) permet de créer un objet de la classe `Grid()` à partir des fichiers présents dans le dossier `input`.

2.2 Le module solver

Un objet de la classe `grid.Grid()` est passé en paramètre pour créer un objet de la classe `Solver()`. La méthode `score()` détermine le score de la liste des paires choisies `pairs`.

2.2.1 La classe SolverGreedy()

La classe `SolverGreedy()` hérite de la classe `Solver()` et l'appel à la méthode `run()` exécute un algorithme glouton pour résoudre la grille.

Complexité

À chaque étape, l'appel à la méthode `all_pairs` est en $O(n \cdot m))$ puis on sélectionne la paire de coût maximal (où le coût est défini par $v_i + v_j - |v_i - v_j|$) en $O(|\texttt{all\_pairs}|) = O(nm)$. Il y a au plus autant d'étapes que de paires pouvant être sélectionnées c'est à dire en $O(nm)$. La complexité de l'algorithme glouton est donc $O(n^2m^2)$.

2.2.2 La classe Graph()

Pour créer un objet de classe `Graph` est passé en paramètre le nombre de nœuds (`int`) et la liste des arêtes du graphe (`list(tuple(int, int, int))`) où chaque arête est un tuple composé des deux nœuds reliés.

La méthode `bfs(source, target, parent)` détermine s'il existe un chemin dans le graphe du nœud `source` au nœud `target`, et met à jour la liste `parent` par effet de bord.

La méthode `ford_fulkerson(source, target)` applique l'algorithme de Ford Fulkerson au graphe qui doit être biparti et renvoie le flot maximal dans ce graphe et met à jour la matrice `adjency` par effet de bord.

2.2.3 La classe SolverMaxMatching()

La classe `SolverMaxMatching()` hérite de la classe `Solver()`. L'appel à la méthode `self.run()` permet d'exécuter la résolution d'une grille dont toutes les valeurs sont fixées à 1, en appliquant l'algorithme de FORD-FULKERSON par l'appel à la méthode `ford_fulkerson` de la classe `Graph` à un problème de maximisation de flot dans un graphe biparti, équivalent à notre problème d'appariement dans la grille (voir 3.2).

3 L'algorithme de FORD-FULKERSON et l'équivalence avec le problème initial

Voir dans l'annexe pour la description de l'algorithme de FORD-FULKERSON

3.1 Complexité de l'algorithme de FORD-FULKERSON

On trouve un chemin augmentant p à chaque itération, d'un coût $O(|E|)$ par itération. De plus le nombre d'itérations est borné par le nombre de sommets et arcs : $O(|V||E|)$. Donc **la complexité de l'algorithme de FORD-FULKERSON est** $O(|E| \cdot |V||E|) = O(|V||E|^2)$.

3.2 Equivalence entre le problème d'appariement des cellules de la grille avec un problème de maximisation de flot

Les cellules de la grille sont toutes fixées à 1, par conséquent, comme le coût de sélection d'une paire est donc de zéro, alors que laisser une cellule non appariée coûte 1, le problème d'appariement minimisant le score équivaut donc à sélectionner l'appariement de plus grande cardinalité dans la grille. On transforme alors le problème en un problème de maximisation du flot dans un graphe biparti :

— On construit deux ensembles de noeuds A et B correspondant à l'ensemble des cellules paires et impaires respectivement, c'est à dire dont la somme des composantes est paire et est impaire respectivement.

— On relie un noeud de l'ensemble A à un noeud de l'ensemble B avec une arête orientée de capacité 1 si et seulement si la paire associée dans la grille peut être sélectionnée.

On applique ensuite l'algorithme de FORD-FULKERSON à ce graphe biparti et on obtient l'appariement de cardinalité maximale dans ce graphe (il s'agit de l'ensemble des arêtes qui ont été renversées, car cela veut dire qu'une partie du flot est passée par cette arête).

On récupère alors les paires de cellules de la grille associées à chacune des arêtes du graphe sélectionné. Il s'agit bien de l'appariement maximal puisque l'algorithme de FORD-FULKERSON permet de maximiser le flot dans le graphe biparti dont toutes les

arêtes sont de capacité 1, c'est-à-dire qu'il sélectionne le sous-graphe du graphe biparti de plus grande cardinalité, dont aucun nœud (hors `source` et `target`) n'est un sommet de plus de deux arêtes (dont une la relie à `source` ou à `target`) : il s'agit bien de l'appariement de plus grande cardinalité entre les paires de la grille ne se recouvrant pas !

Calcul de la complexité temporelle :

On crée le graphe biparti en `len(all_pairs())` $\leq nm$. Comme alors le nombre de sommets du graphe $V = nm$ et son nombre d'arêtes $E \leq 4nm$ (car chaque nœud est relié à au plus 4 autres nœuds, correspondant aux cellules adjacentes), alors on en déduit que l'algorithme a une complexité en $O((nm)^3)$ (voir calcul de la complexité de l'algorithme hongrois en annexe).

4 Généralisation de la solution pour des grilles quelconques et l'algorithme de Khun-Munkres

Les cases de la grille susceptibles d'être appariées sont classées en deux catégories, selon la parité de la somme de leurs coordonnées. Une matrice des coûts est ensuite construite de la manière suivante :

- La matrice a une taille de $n_p \times n_i$, où n_p est le nombre de cases dites "paires" et n_i le nombre de cases dites "impaires", où la parité correspond à la parité de la somme des coordonées de la case. Elle est initialisée avec des zéros.
- Pour chaque combinaison de case paire et de case impaire, si ces deux cellules ne sont pas adjacentes, la valeur correspondante dans la matrice des coûts reste à zéro.
- Si les cases i et j respectent les conditions d'adjacence et respectent les contraintes de couleur, la valeur de la cellule correspondante dans la matrice est calculée comme le coût d'appariement de ces deux cellules, c'est-à-dire $|v_i - v_j| - v_i - v_j$ car le choix d'une cellule coûte $|v_i - v_j|$ mais permet d'éviter de laisser deux cases non appariées, de coût $v_i + v_j$ dans le score final.

Une fois cette matrice des coûts construite, l'algorithme hongrois (aussi appelé algorithme de Khun-Munkres, et dont l'explication détaillée se trouve en annexe) est appliqué pour résoudre le problème d'affectation de manière optimale. L'algorithme trouve la meilleure correspondance entre les cases paires et impaires, en minimisant le coût total d'appariement. Le résultat est une liste de paires de cellules appariées, accompagnée du score minimal associé à cette affectation optimale.

Calcul de la complexité temporelle :

On construit la matrice en $O(n_p n_i)$ si n_p est le nombre de cellules paires et n_i le nombre de cellules impaires puis on l'injecte dans l'algorithme hongrois qui a une complexité temporelle en $O(max(n_p, n_i)^3)$ car la matrice est de taille $max(n_p, n_i) \times max(n_p, n_i)$. Comme $n_p \approx n_i \approx \frac{n}{2}$ où n est le nombre de cellules de la grille, on en déduit que **la complexité temporelle de l'algorithme vaut** $O(n^3)$.

5 L'affichage graphique de la résolution et la version interactive

L'affichage graphique de la résolution est géré par la classe `PlotResolution()`.

L'appel à la méthode `launch()` affiche un menu dans lequel l'utilisateur peut choisir le solver pour résoudre la grille, ou bien peut choisir de résoudre lui-même la grille. Dans le cas où un des solvers `Greedy` ou `Ford-Fulkerson` ou `Hungarian` est choisi, on fait appel à la méthode `run_solver(bot_name)` qui résout la grille avec le solver choisi puis affiche graphiquement l'évolution progressive de la grille, en marquant un temps d'arrêt entre chaque étape. Chaque étape fait appel à la méthode `plotStep(bot)` du module `Grid()`, dont l'idée générale du code est la même que la méthode `plot()` du même module, qui permet d'afficher la grille à l'état actuel, en prenant bien garde de ne pas afficher les cellules déjà sélectionnées, présentes dans la liste `plot_removed`, attribut de la classe `Grid`. Une autre fonctionnalité additionnelle de `plotStep()` est que les deux cellules sélectionnées à l'étape courante, dans la liste `selected_cells` attribut de la classe `Grid` sont mises en évidence par un cadre bleu clair. L'argument `bot` est une chaîne de caractères permettant d'identifier quel solver a été choisi et `bot_name` est une chaîne de caractères qui sera affichée en dessous de la grille, afin de spécifier quel algorithme résout la grille. Ensuite la fonction `plot_score(score)` est appelée afin d'afficher le score dans une nouvelle fenêtre.

6 Quelques mesures de temps à l'exécution

Afin de mesurer les temps à l'exécution des différents solver, nous avons utilisé le module python `time` et exécuté les programmes sur un processeur `13th Gen Intel(R) Core(TM) i5-1340P 1.90 GHz`.
Les résultats calculés à 1e-5 près sont listés dans le tableau suivant :

Grid	SolverGreedy	SolverMaxMatching	SolverHungarian	SolverHungarianScipy
00	4.6740e-04	6.6990e-03	2.9243e-03	3.2902e-04
01	4.3824e-04	3.7520e-03	3.2158e-03	2.8014e-04
02	5.0700e-04	3.502e-03	4.2123e-03	2.7514e-04
03	7.011e-04	2.5088e-02	1.0004e-03	4.6873e-04
04	4.9320e-04	2.393e-02	2.4347e-03	3.2282e-04
05	7.602e-04	2.6205e-02	1.0037e-03	5.5337e-04
11	2.8180e-03	3.6795e+00	8.7099e-03	1.9658e-03
12	3.991e-03	4.179e+00	1.0954e-02	2.3305e-03
13	3.042e-03	4.4123e+00	1.1103e-02	5.6026e-03
14	1.8301e-02	5.9072e+00	1.6740e-02	2.0270e-03
15	9.2916e-03	6.6202e+00	1.7605e-02	2.2068e-03
16	1.7864e-02	6.3119e+00	1.3216e-02	1.9939e-03
17	7.3652e-03	6.5987e+00	1.3373e-02	2.1582e-03
18	3.0422e-03	6.2587e+00	1.6714e-02	2.6760e-03
19	7.5119e-03	6.653e+00	1.8298e-02	2.322e-03
21	3.7264e+01		8.6574e+01	6.825e+00
22	3.9238e+01		8.5356e+01	8.0155e+00
23	3.4534e+01		1.0098e+02	6.8498e+00
24	6.7267e+01		1.5967e+02	1.2666e+01
25	7.0560e+01		1.7080e+02	1.3193e+01
26	7.1229e+01		1.474e+02	1.2363e+01
27	6.4843e+01		1.6744e+02	1.2910e+01
28	6.6114e+01		1.9987e+02	1.2989e+01
29	6.4185e+01		1.7383e+02	1.3118e+01

Pour SolverMatchMaxing, le temps d'exécution pour les grilles suivant la grille 19 n'a pas été relevé pour des raisons techniques. En effet, pour les autres solvers, le rapport de temps d'exécution entre les grilles 19 et 21 est de l'ordre de 1000. Puisque pour SolverMatchMaxing, le temps d'exécution pour la grille 19 est d'environ 6,5 secondes, le temps d'exécution pour la grille 21 serait d'environ 2 heures...

`SolverHungarianScipy` permet de comparer nos résultats à ceux qu'on obtient en utilisant la fonction `linear_sum_assignment` de la librairie `scipy.optimize`. Nous constatons que le solver utilisant la fonction `linear_sum_assignment` est en moyenne 10 fois plus rapide que le solver codé manuellement.

7 Vers une suite du projet ?

7.1 Nouvelles règles...

Dans le cas où les règles d'appariements sont changées, et les cellules blanches peuvent être appariées à n'importe quelles cellules de la grille, hormis les cellules noires, la fonction `all_pairs` peut être modifiée facilement, mais la difficulté réside dans le calcul de la solution optimale. Le graphe équivalent au problème n'est alors plus biparti, puisque deux cellules paires ou deux cellules impaires peuvent maintenant être appariées l'une à l'autre. L'algorithme hongrois ne fonctionnant que pour des graphes bipartis, on serait alors dans une impasse... C'est pourquoi on pourrait faire appel à l'algorithme de Blossom qui est détaillé dans l'annexe, et qui lui, ne nécessite pas de graphe biparti, mais fonctionne dans

le cas général d'un graphe orienté pondéré.

7.2 L'adversarial

Afin de créer un jeu adversarial, c'est-à-dire joueur contre joueur, on peut modifier les règles par exemple de façon à ce que les joueurs doivent maximiser leur score. Afin de développer un algorithme qui pourrait jouer à ce jeu contre un autre joueur, on s'est intéressé sur le plan théorique à l'algorithme minimax de Von Neumann. Il paraît certain que nous pourrions arriver à des résultats substantiels en se contentant de descendre de deux couches dans l'arbre des possibles, mais afin d'avoir un algorithme certain de battre son adversaire, il faudrait parcourir en totalité cet arbre des possibilités, ce qui, bien entendu, n'est pas réalisable en un temps correct.

Annexes

A L'algorithme de Ford-Fulkerson

A.1 Cadre et objectif de l'algorithme de Ford-Fulkerson

Un réseau de flot est représenté par un graphe orienté $G = (V, E)$, où :

— V est l'ensemble des sommets,

— E est l'ensemble des arcs,

— Chaque arc (u, v) a une capacité $c(u, v) \geq 0$,

— Il existe une source s (départ du flot) et un objectif t (destination).

L'objectif est de maximiser le flot f de s à t, en respectant :

1. **Contrainte de capacité** : $0 \leq f(u, v) \leq c(u, v)$ pour tout (u, v),
2. **Conservation du flot** : $\sum_u f(u, v) = \sum_w f(v, w)$ pour tout $v \neq s, t$.

A.2 Description de l'algorithme de Ford-Fulkerson

Algorithm 1 Algorithme de Ford-Fulkerson

Entrée : Graphe $G = (V, E)$, capacités $c(u, v)$, source s, objectif t
Sortie : Flot maximum f
Initialiser $f(u, v) = 0$ pour tout $(u, v) \in E$
while il existe un chemin augmentant p dans le graphe résiduel **do** G_f
    $c_f(p) \leftarrow \min\{c_f(u, v) \mid (u, v) \in p\}$ ▷ Capacité résiduelle minimale
    for chaque arc (u, v) dans p **do**
        $f(u, v) \leftarrow f(u, v) + c_f(p)$
        $f(v, u) \leftarrow f(v, u) - c_f(p)$ ▷ Mise à jour arc inverse
    end for
end while
Retourner $\sum_{v\,:\,(s,v)\in E} f(s, v)$

B L'algorithme de Kuhn-Munkres

B.1 Cadre et objectif

L'algorithme de Kuhn-Munkres, aussi appelé algorithme hongrois vise à résoudre un problème d'affectation consistant à associer n agents à n tâches, représentés par une matrice de coût C de taille $n \times n$, où $C[i, j]$ est le coût d'assigner l'agent i à la tâche j. L'objectif est de minimiser le coût total de l'affectation, c'est-à-dire de trouver une permutation des lignes ou colonnes telle que la somme des éléments sélectionnés (un par ligne et par colonne) soit minimale.

B.2 Description de l'algorithme hongrois

Voici le pseudo-code de l'algorithme hongrois :

Algorithm 2 Algorithme hongrois

Entrée : Matrice de coût C de taille $n \times n$
Sortie : les n zéros indépendants
for chaque ligne i de C **do** ▷ Réduction de la matrice
    $min\_ligne \leftarrow \min(C[i,:])$
    $C[i,:] \leftarrow C[i,:] - min\_ligne$
end for
for chaque colonne j de C **do**
    $min\_col \leftarrow \min(C[:,j])$
    $C[:,j] \leftarrow C[:,j] - min\_col$
end for
while le nombre de zéros indépendants $< n$ **do** ▷ Couverture des zéros
    Trouver un ensemble minimal de lignes et colonnes couvrant tous les zéros
    if nombre de lignes et colonnes couvrantes $< n$ **then**
        $min\_non\_couvert \leftarrow$ plus petit élément non couvert
        Soustraire $min\_non\_couvert$ aux éléments non couverts
        Ajouter $min\_non\_couvert$ aux éléments à l'intersection des lignes et colonnes couvrantes ▷ Ajustement de la matrice
    end if
end while
Retourner la liste des n zéros indépendants

Il est important de noter que l'algorithme hongrois nécessite une matrice carrée. Dans l'implémentation de notre code on a donc choisi d'ajouter des lignes ou des colonnes factices au début de l'algorithme remplies de la valeur zéro. On doit ensuite prendre bien soin de ne retourner que les cases de la matrice qui ne sont pas de coût zéro (car dans le cas précis de l'utilisation de notre algorithme, tous les coûts des cellules pouvant être appariées sont strictement négatifs).

Complexité temporelle

Pour une matrice de taille $n \times n$:

On réduit les lignes (*resp.* les colonnes) de la matrice : le coût de la recherche du minimum est en $O(n)$ puis on le retranche aux valeurs de la ligne (*resp.* de la colonne) en $O(n)$. On effectue ces opérations pour les n lignes et les n colonnes, donc pour un coût total de $O(n^2)$.

Couvir les zéros s'effectue en une complexité de $O(n^3)$ car chaque itération de la boucle `While` ajoute un zéro indépendant, il y a donc au plus n opérations, et à chaque itération, le parcours de la matrice coûte $O(n^2)$.

Dans le cas où le nombre de lignes et colonnes couvrantes est strictement inférieur à n on ajuste la matrice en créant des parmi dans les lignes et colonnes non couvertes en $O(n^2)$.

On identifie les zéros indépendants en $O(n^2)$.

La complexité totale de l'algorithme est donc $O(n^3)$.

B.3 L'algorithme de Blossom

L'algorithme Blossom est une méthode qui permet de résoudre le problème de couplage maximum dans un graphe non orienté. Le principe repose sur la recherche d'un ensemble maximal d'arêtes disjointes (un couplage) qui maximise le nombre de sommets couverts. Cet algorithme est particulièrement puissant car il gère les graphes généraux, y compris ceux contenant des cycles de longueur impaire, grâce à l'introduction du concept de *blossoms* (fleurs).

Une fleur est une structure formée par un cycle impair dans le graphe, qui peut être ramenée à un seul super-sommet (i.e. contractée) pour simplifier la recherche d'un chemin augmentant. Lorsqu'un tel chemin est trouvé, il permet d'augmenter la taille du couplage en alternant entre arêtes appariées et non appariées. Le mécanisme clé consiste à détecter ces fleurs, les contracter, résoudre le problème sur le graphe réduit, puis décontracter les fleurs pour obtenir une solution valide dans le graphe original.

Algorithm 3 Algorithme Blossom pour le couplage maximum

Entrée : Un graphe non orienté $G = (V, E)$
Sortie : Un couplage maximum M
Initialiser $M \leftarrow \emptyset$ (couplage vide)
while il existe un chemin augmentant P dans G par rapport à M **do**
    if un cycle impair (fleur) est détecté dans P **then**
        Contracter la fleur en un super-sommet
        Mettre à jour G et M dans le graphe contracté
        Rechercher un nouveau chemin augmentant dans le graphe modifié
    else
        Augmenter M en utilisant P : $M \leftarrow M \oplus P$
    end if
end while
Décontracter toutes les fleurs pour restaurer G et ajuster M
Retourner M

Il est important de noter que l'algorithme de Blossom nécessite une matrice carrée. Dans l'implémentation on peut donc choisir d'ajouter des lignes ou des colonnes factices au début de l'algorithme remplies de valeurs très grandes (10^9 par exemple). Remplir la matrice de valeurs infinies pouvait conduire le code à ne pas finir, on doit donc préférer laisser des valeurs réelles mais très grandes, en prenant bien soin de ne pas retourner les couples composés de noeuds factice lors du renvoi de l'assignement optimal.

Calcul de la complexité temporelle :

L'algorithme Blossom résout le problème du couplage maximum dans un graphe non orienté $G = (V, E)$, où $n = |V|$ est le nombre de sommets et $m = |E|$ est le nombre d'arêtes. La complexité temporelle atteint $O(n^2 m)$ (source Wikipédia).