ENSAE 1A – Projet de programmation 2025

Responsable du cours: Patrick Loiseau
patrick.loiseau@inria.fr

Version du 9 février 2025

1 Instructions et recommandations générales

1.1 Objectifs

Le projet de programmation a pour objectif global de vous aider à acquérir une expérience de programmation par la pratique, via la résolution d'un problème particulier (décrit dans la section suivante). Plus spécifiquement, il est attendu au cours du projet que vous acquériez un certain nombre de compétences incluant :

— Apprendre à structurer du code pour un projet de moyenne envergure
— Apprendre à chercher en ligne les informations, les documentations, et les librairies utiles
— Apprendre à analyser un problème et à le traduire en terme algorithmique
— Apprendre à analyser la complexité d'un algorithme
— Apprendre à manipuler quelques algorithmes et structures de données classiques
— Apprendre à debugger, tester, et optimiser du code
— Se familiariser avec les bonnes pratiques de code et avec un environnement d'édition.

1.2 Environnement

Le projet est fait en **Python** (obligatoirement). On utilisera la notion de classe, avec une structure minimale imposée (voir plus loin), et on utilisera des tests unitaires. Pour l'ensemble du projet, on demande l'utilisation de l'éditeur **vscode**. Il peut être utilisé en local sur les machines des salles informatiques. Il est également possible d'utiliser vscode sur une machine personnel ou sur un serveur cloud type onyxia, mais dans ce cas les élèves doivent pouvoir installer de façon autonome ce dont ils ont besoin.

Une documentation sur l'usage de vscode à l'ENSAE est disponible ici :

https://moodle.ip-paris.fr/mod/book/view.php?id=122644&chapterid=2226,

ainsi qu'en PDF sur pamplemousse.

1.3 Évaluation et rendus

Le projet **doit être fait en binôme ; les deux membres d'un même binôme doivent obligatoirement appartenir au même groupe de TP**. Vous devez vous inscrire avec votre binôme, **avant le début de la séance 2**, en envoyant un email à votre chargé de TP donnant le nom du binôme. Lors de la séance 2, les étudiants n'ayant pas inscrit leur binôme seront appariés arbitrairement avec un autre étudiant du groupe.

Le projet contient 5 séances de TP. Les deux premières séances de TP seront relativement guidées, les deux suivantes plus ouvertes, la dernière servira à l'évaluation finale. On demande 2 rendus :

Un rendu intermédiaire, dû le dimanche 02/03 (23h59) : Après la 2ème séance de TP (juste avant la 3ème), on demande le rendu de l'ensemble du code écrit jusque-là, testé et documenté. Le code doit être déposé sur moodle en suivant le lien correspondant à son groupe de TP, sous la forme d'une archive `.zip` unique contenant tout le code.

Important : Pour le rendu intermédiaire comme pour le rendu final, déposer **une seule archive `.zip` par binôme**. Elle peut être déposer par n'importe lequel des membres du binôme. Elle doit par contre être déposée dans le bon groupe de TP.

Un rendu final, dû le dimanche 30/03 (23h59) : Une semaine avant la dernière séance de TP, on demande le rendu final de l'ensemble du code du projet, documenté, accompagné d'un rapport d'environ 4-6 pages au format PDF. Le rapport doit être concis, clair et précis, et décrire succinctement les choix algorithmiques et les résultats et justifier la complexité des algorithmes utilisés. Le code doit être déposé sur moodle en suivant le lien correspondant à son groupe de TP, sous la forme d'une archive `.zip` unique contenant tout le code ainsi que le rapport au format PDF.

Lors de la dernière séance de TP, le 07/04, chaque chargé de TP passera voir tous les binômes de son groupe pour une séance de questions destinée à évaluer la compréhension des deux membres du binôme. Il ne sera pas demandé de faire une présentation du projet (pas de slide), mais il faudra pouvoir faire une démo si demandé.

La présence aux TPs est obligatoire et sera vérifiée au début de chaque séance. Des pénalités seront appliqués pour les absences. La notation inclura 3 points (sur 20) pour le rendu intermédiaire. Nous serons particulièrement attentifs au fait que le code soit testé et bien documenté. Le reste (17 points) sera attribué sur la base du rendu final et de la séance de questions lors du dernier TP. Nous tiendrons compte d'un certain nombre de critères : qualité du code, clarté de la documentation, pertinence des choix algorithmiques, analyse des performances algorithmiques, respect des bonnes pratiques, etc. Il est important de noter que la qualité du code (et de sa documentation)—ainsi que la compréhension par les deux membres du binôme—est un critère essentiel dans la notion, plus encore que la quantité de code produite.

L'objectif du cours est d'acquérir une compétence en programmation. Celle-ci est essentielle pour le reste des études (et la vie professionnelle de la plupart d'entre vous). En conséquence, et contrairement à d'autres cours focalisés sur des applications de ces compétences, **le recours à ChatGPT ou à tout autre IA générative équivalente est strictement interdit et lourdement sanctionné par le règlement de l'école (tout comme le plagiat).** Les chargés de TP testeront de façon détaillée la connaissance par chacun des deux membres du binôme. Ils le feront par des questions précises et ciblées, mais ils pourront aussi par exemple introduire des erreurs dans le code et demander de les corriger, voir demander de coder devant eux une fonction ou d'en modifier une.

1.4 Ressources utiles

Le projet suppose une connaissance du language Python acquise au premier semestre. Toutefois, en cas de besoin, il existe de nombreuses ressources en ligne et de nombreux cours et livres sur la programmation en Python, tels que *Introduction to programming in Python* de R. Sedgewick, K Wayne, et R. Dondero. Pour ce qui est des bonnes pratiques, il existe de nombreuses ressources. Citons en particulier :

- Les documents de la communauté Python, e.g., le PEP8 (qui contient une section sur le nommage) et le PEP257 (sur l'écriture de documentation)
- Le Hitchhiker's Guide to Python
- Les supports de Lino Galiana (enseignant du cours Python pour la data science de deuxième année) sur la qualité du code

Il pourra être utile de s'y référer au long du projet. Enfin, les élèves pourront être amenés à vouloir consulter des livres d'algorithmique ; il en existe là aussi de nombreux (e.g., *Introduction to Algorithms* de Cormen et al., *Algorithm Design* de Kleinberg et Tardos, ou *Algorithms* de Dasgupta, Papadimitriou, et Vazirani).

1.5 Contact des chargés de TP

- Groupe 1 : Patrick Loiseau, email : `patrick.loiseau@inria.fr`
- Groupe 2 : Solal Rapaport, email : `solal.rapaport@telecom-paris.fr`
- Groupe 3 : Ziyad Benomar, email : `ziyad.benomar@ensae.fr`
- Groupe 4 : Simon Mauras, email : `simon.mauras@gmail.com`
- Groupe 5 : Lucas Baudin, email : `lucas.baudin@ensae.fr`
- Groupe 6 : Melissa Tamine, email : `melissa.tamine@ensae.fr`
- Groupe 7 : Marius Potfer, email : `marius.potfer@ensae.fr`
- Groupe 8 : Youssef Ouazzani Chahdi, email : `youssef.ouazzani-chahdi@centralesupelec.fr`
- Groupe 9 : Axel Xerri, email : `axel.xerri@ensae.fr`

2 Description du problème et du code fourni

2.1 Description du problème

On considère une grille $n \times m$, où $n \geq 1$ et $m \geq 2$ sont des entiers représentant respectivement le nombre de lignes et de colonnes de la grille. La grille contient des cellules de coordonnées (i, j) où $i \in \{0, \cdots, n-1\}$ est l'indice de ligne et $j \in \{0, \cdots, m-1\}$ est l'indice de colonne. A chaque cellule est associée 2 attributs :

— une couleur $c(i, j)$ (ou `color` dans le code). La couleur est un entier dans $\{0, 1, 2, 3, 4\}$ et chaque entier correspond à une couleur suivant le mapping donné dans l'attribut `colors_list` :
  0 : blanc (white, ou `'w'` en Python)
  1 : rouge (red, ou `'r'` en Python)
  2 : bleu (blue, ou `'b'` en Python)
  3 : vert (green, ou `'g'` en Python)
  4 : noir (black, ou `'k'` en Python)

— une valeur $v(i, j)$ (ou `value` dans le code). La valeur est un entier positif.

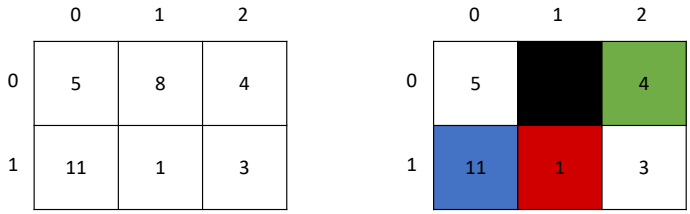

FIGURE 1 – Examples de gilles 2×3 (i.e, $n = 2$, $m = 3$).

Le problème est le suivant : on prend des paires de cellules [1] avec les contraintes suivantes :

— On ne peut prendre une paire de cellules que si les deux cellules sont adjacentes, c'est à dire soit l'une au dessus de l'autre, soit l'une à côté de l'autre. Formellement, on a le droit de prendre la paire des cellules (i_1, j_1) et (i_2, j_2) si et seulement si $i_1 = i_2$ et $|j_1 - j_2| = 1$; ou $j_1 = j_2$ et $|i_1 - i_2| = 1$. On n'autorise pas à prendre une paire entre la première et dernière ligne, ou la première et dernière colonne (i.e., la grille n'est pas circulaire).
— Les couleurs imposent les contraintes suivante (en plus de la contrainte d'adjacence ci-dessus) :
  — On n'a jamais le droit de prendre une cellule noire (i.e., une paire contenant une cellule noire).
  — On peut apparier une cellule blanche avec n'importe quelle autre couleur (sauf noire). On peut apparier une cellule bleue avec une cellule bleue ou rouge ou blanche. On peut apparier une cellule rouge avec une cellule bleue ou rouge ou blanche. On peut apparier une cellule verte seulement avec une cellule verte (ou blanche).

Chaque cellule ne peut être prise que dans une seule paire. L'objectif est de choisir une liste (valide) de paires à prendre pour minimiser le score calculé de la façon suivante : on somme, pour chaque paire $((i_1, j_1), (i_2, j_2))$, la différence (en valeur absolue) des valeurs des cellules $|v(i_1, j_1) - v(i_2, j_2)|$ et on ajoute à cela la somme des valeurs des cellules non prises dans une paire, sauf les cellules noires. [2]

2.2 Description du code fourni

Dans cette section, on décrit brièvement la base de code fournie. On rappelle toutefois qu'il est bien souvent indispensable de **consulter le code lui-même (et sa documentation en commentaire)** pour en comprendre les détails. La base de code est fournie dans une archive `.zip` disponible sur pamplemousse. **Note : il est crucial de ne pas modifier les noms des fonctions données en les implémentant, ainsi que de bien respecter les instructions données dans la suite.**

1. On peut imaginer que ce sont des cartes d'un jeu, installées en grille, et qu'on prend des paires de 2 cartes.
2. Les cellules noires sont des cellules "interdites", on n'ajoute pas leur valeur au score. Si on voit le problème comme un jeu de cartes installées sur une grille, on peut imaginer que les cellules noires correspondent à des espaces sans carte.

On fourni un squelette de classe `Grid` définissant une object de type grille. Un object `grid` de type `Grid` contient les attributs `n` et `m`, et les attributs `color` et `value` qui contiennent les couleurs et valeurs des cellules de la grille au format liste de liste. Par example, la grille à gauche de la Figure 1 correspondra à un attribut `color` égal à

```
[[0, 0, 0], [0, 0, 0]],
```

et à un attribut `value` égal à

```
[[5, 8, 4], [11, 1, 3]].
```

Il y a également un attribut `colors_list` qui donne la correspondance entre les couleurs au format numérique et les couleurs elles-même.

La classe `Grid` contient également quelques premières fonctions, en particulier celle qui permet de créer un objet de type `Grid` à partir d'un fichier de type `grid00.in` dont le format suit la structure ci-dessous.

On fourni également une ébauche de classe `Solver` dans le fichier `solver.py`. Ici, on impose simplement de retourner la solution sous le format d'une liste de tuple représentant des paires où chaque paire est un tuple de 2 tuples représentants les cellules de la paire. Par example, une solution pour la grille de la Figure 1 gauche pourra être

```
[((0, 0), (1, 0)), ((0, 1), (1, 1)), ((0, 2), (1, 2))]
```

Cette solution est stockée dans l'attribut `pairs` de l'objet de type `Solver`. Pour aider à comprendre la structure, on fourni un premier solver qui ne prend aucune paire (il est donc très mauvais, mais la liste de paires retournée est valide).

Enfin le fichier `main.py` a pour but d'appeler et de faire tourner les fonctions développées dans les fichiers `grid.py` et `solver.py` pour résoudre le problème.

Description des fichiers d'entrée `gridxy.in`, fournis Les fichiers d'entrées (dans le répertoire `input`) dont le nom est de la forme `gridxy.in` pour 2 entiers x et y représente une grille et ont le format suivant :

- la première contient $n \quad m$ (2 entiers séparés par un espace)
- les n lignes suivantes contiennent m entiers séparés par un espace correspondant aux couleurs des m cellules de la ligne correspondante
- optionnellement les n lignes suivantes contiennent m entiers séparés par un espace correspondant aux valeurs des m cellules de la ligne correspondante. Si ces n lignes ne sont pas présentes (i.e., le fichier n'a que $n+1$ lignes), il faut interpréter comme : toutes les valeurs sont égales à 1.

Par example, la grille à gauche de la Figure 1 sera encodée par le fichier `grid00.in`

```
2 3
0 0 0
0 0 0
5 8 4
11 1 3
```

et celle de droite par le fichier `grid01.in`

```
2 3
0 4 3
2 1 0
5 8 4
11 1 3
```

On fourni plusieurs fichiers d'entrée :

- Les fichiers `grid0y.in` correspondent à des petites grilles :
  - `grid00.in`, `grid01.in`, `grid02.in` sont des grilles 2×3 pour des petits tests
  - `grid03.in`, `grid04.in`, `grid05.in` sont des grilles 4×8 pour des tests plus grands mais qu'on pourra quand-même regarder à la main
- Les fichiers `grid1y.in` sont des grilles 10×20. Les trois première sont en noir et blanc avec des valeurs 1 partout, les trois suivantes ont toutes les couleurs avec des valeurs 1 partout, les trois dernières ont toutes les couleurs et des valeurs entre 1 et 10.

— Les fichiers `grid2y.in` sont des grilles 100×200. De même les trois première sont en noir et blanc avec des valeurs 1 partout, les trois suivantes ont toutes les couleurs avec des valeurs 1 partout, les trois dernières ont toutes les couleurs et des valeurs entre 1 et 10.

Les différents fichiers d'entrée pourront être utilisés tout au long du projet quand c'est approprié.

Description des tests fournis Nous fournissons dans le répertoire `tests` un test préliminaire très basique dans le fichier `test_grid_from_file.py`. Ce test porte sur la méthode `grid_from_file` qui est implémentée et devrait donc passer avec succès. Le but est essentiellement de vous donner un example de structure de test unitaire sur lequel vous pouvez vous appuyer pour développer d'autres tests unitaires. **Il est de votre responsabilité (cela fait parti du projet) d'implémenter de nombreux autres tests beaucoup plus exhaustifs tout au long du développement de votre code.** Vous devez mettre ces tests dans le repertoire `tests`.

3 Quelques rappels utiles

Le texte ci-dessous rappelle simplement quelques éléments permettant d'aborder efficacement le travail à faire sur le projet.

3.1 Environnement de travail

On demande d'utiliser vscode et on rappelle qu'une documentation se trouve dans le PDF sur pamplemousse, ou sur moodle à cette adresse :

https://moodle.ip-paris.fr/mod/book/view.php?id=122644&chapterid=2226.

3.2 Résumé sur l'utilisation du terminal

Le terminal est une interface en ligne de commande (par opposition aux interfaces graphiques) qui permet d'interagir avec le système d'exploitation en écrivant des instructions. Dans ce TP, vous pouvez utiliser les fonctionnalités de VSCode pour exécuter du code python (le bouton « lecture » en haut à droite) mais il peut être utile d'exécuter votre code via le terminal. Il est très pratique de connaitre aussi les commandes pour se déplacer d'un répertoire à un autre ou lister les fichiers d'un répertoire.

Depuis VSCode, vous pouvez ouvrir un terminal en utilisant le menu en haut à gauche, puis en sélectionnant « Terminal » et « New Terminal ». Sur la partie inférieure de votre écran, vous obtenez l'invite de commandes qui est de la forme :

```
PS W:\Documents\ensae-prog25 >
```

où `W:\Documents\ensae-prog25` est le dossier actuel, `PS` désigne le nom du shell (ici PowerShell de Windows). La présentation peut varier sur d'autres systèmes, mais il y a (presque) toujours au moins le nom du dossier courant.

Une première commande : `pwd` La commande `pwd` permet d'afficher le chemin du dossier dans lequel on se trouve. Si vous tapez la commande et validez avec la touche Entrée, vous obtenez un résultat de la forme :

```
W:\Documents\testgit\ensae-prog25
```

Lister les fichiers du dossier La commande `ls` (pour ***list***) permet d'afficher les fichiers et dossiers du dossier courant.

Changer de dossier avec `cd` La commande `cd` permet de changer de dossier (c'est un acronyme de ***c**hange **d**irectory*).

Le dossier parent s'écrit `..`, ainsi la commande `cd ..` permet d'aller dans le dossier `W:\Documents`.

Exécuter du code python La commande `python` permet d'exécuter un fichier `.py`.

```
W:\Documents\testgit\ensae-prog25 > python mondossier/main.py
<exécution du code python>
```

où `mondossier` est le nom du dossier qui contient votre fichier `main.py`.

Le dossier où on exécute le code python est important : certains scripts python peuvent être exécutés dans leur dossier directement (c'est le cas de `main.py`) tandis que d'autres sont prévus pour être exécuté uniquement dans le dossier parent. C'est particulièrement important lorsqu'on utilise des chemins *relatifs*, par exemple :

```
open("input/fichier.in")
```

ne fonctionne que si on est dans le dossier parent qui contient le dossier `input`.

Si à l'inverse le shell est dans le dossier `tests`, alors il faut utiliser :

```
open("../input/fichier.in")
```

i.e. on indique à python qu'il faut d'abord se déplacer dans le dossier parent (avec `..`) puis dans le dossier `input` avant d'ouvrir `fichier.in`.

3.3 Classes

En Python, tout est un objet. Il existe plein de types d'objets prédéfinis, tels que les entiers, les listes, les fonctions, etc. Les classes servent à définir de nouveaux types d'objets.

Définition et instanciation de la classe Dans notre cas, on veut définir le type d'objet correspondant à une grille, dans la classe `Grid` du fichier `grid.py` (le plus souvent, chaque classe est dans son propre fichier). On peut alors instancier la classe, c'est-à-dire créer un objet de la classe, avec l'instruction suivante :

```
grid = Grid(2, 3, [], [])
```

Dans ce cas, la variable `grid` contient maintenant un objet de type grille. Les entiers 2 et 3 et les listes vides `[]` sont les argument du constructeur de la classe `Grid`.

Attributs et méthodes Cet objet a des attributs auxquels on accède par la notation `.`, par exemple `grid.n` pour l'attribut nombre de lignes. Enfin, on peut définir des méthodes, c'est-à-dire des fonctions qui agissent sur un élément de la classe, auxquelles on accède aussi par la notation `.`. La définition des attributs et des méthodes de la classe `Grid` est faite sous la ligne `class Grid`. Par exemple, une méthode `is_forbidden` sera définie dans la classe de façon similaire à une fonction :

```
class Grid:
    ...
    def is_forbidden(self, i, j):
        code de la methode
```

Le premier argument de la méthode est obligatoirement `self` et représente l'objet sur lequel la méthode est appelée.

La méthode peut-être exécutée sur l'objet `grid` avec :

```
grid.is_forbidden(1, 0)
```

où `1` correspond au premier argument (ligne de la cellule) et `0` au deuxième(colonne de la cellule).

Dans notre cas, cet appel retourner `True` si la cellule (i, j) dans la grille `grid` est noire et `False` sinon.

Il existe également des méthodes dites de classe, qui n'agissent pas sur un objet de la classe mais sur la classe elle-même. Elles sont précédés par le décorateur `@classmethod`. C'est le cas de la méthode `grid_from_file` qui crée un objet de type `Grid` par lecture d'un fichier d'input.

```
    @classmethod
    def grid_from_file(cls, file_name, read_values=False):
        ...
        return grid
```

Celle-ci s'appelle comme suit :

```
grid = Grid.grid_from_file(file_name)
```

Notez qu'ici, comme pour les fonctions, `read_values=False` signifie que `False` est la valeur par défaut de l'argument `read_values` (c'est à dire la valeur qu'il prend si aucune n'est donnée).

Il est important de savoir écrire des classes, en particulier des méthodes dans les classes. Cela a été vu au 1er semestre et doit être acquis. Toutefois, en cas d'oubli, voici un lien vers courte introduction aux classes : https://courspython.com/classes-et-objets.html (vous pouvez aussi simplement vous referez au cours du 1er semestre).

3.4 Tests unitaires

Les tests unitaires permettent de tester les fonctions écrites de façon automatique. On les écrits dans un répertoire séparé, en utilisant le framework `unittest` de Python. Une courte introduction est donnée par exemple ici https://gayerie.dev/docs/python/python3/unittest.html.

Dans ce projet, on fourni un premier test qui montre la structure d'un test. **ATTENTION : Il est nécessaire et très important de complémenter ces tests en en écrivant d'autres pour tester toutes les fonction que vous coderez**. Notez qu'en complément des tests unitaires qui sont une façon systématisée de tester les fonctions, il est souvent très utile d'écrire dans le `main.py` des petits bouts de code qui appellent les fonction et en affiche le résultat, pour tester qu'il correspond bien à ce qu'on attend.

Les tests unitaires peuvent être exécutés soit avec le framework de vs-code, soit plus simplement en exécutant le fichier Python correspondant, par example en tapant dans un terminal

```
python tests/test_grid_from_file.py
```

Attention toutefois, les tests sont écrit de telle façon qu'ils ne sont executables que depuis le répertoire racine `ensae-prog25`.

4 Organisation détaillée des séances

L'organisation des séances proposée ci-dessous a pour but de vous aider à avancer de façon guidée au début et de vous donner plus de liberté à la fin. Il est toutefois possible d'avancer plus vite !

Séance 1 : 27/01

Dans la première séance, on va préparer l'environnement de travail, coder des fonctions de base de la classe Grid, et coder une première solution naïve du problème. On pourra en supplément commencer à produire une représentation graphique agréable du jeu. Noter qu'en dehors des contraintes imposées sur les fonctions demandées et les résultats qu'elles doivent retourner, vous êtes libres d'ajouter toutes les méthodes que vous jugez utile dans la classe `Grid` ou `Solver`.

Question 1. Commencer par préparer l'environnement de travail suivant les indications la section précédente et du chargé de TP. Ne continuez pas plus loin si vous n'avez pas entièrement lu les pages précédentes.

Question 2. Explorer le code fourni dans les classes `Grid` et `Solver`. Pour cela, utiliser le fichier `main.py`, le faire tourner et explorer les différentes méthodes et attributs soit en ajoutant des commandes dans ce fichier, soit en tapant des commandes dans un terminal interactif.

Question 3. Implémenter dans la classe `Grid` les méthodes `is_forbidden` (qui indique si une case est noire ou non), `cost` (qui retourne le cout d'une paire) et `all_pairs` (qui retourne une liste de toutes les paires valides, c'est-à-dire respectant les contraintes énoncées dans la définition du problème en Section 2). On rappelle qu'il est possible d'implémenter d'autres méthodes intermédiaires appelées par ces méthodes si vous le jugez utile. Implémenter des tests unitaires pour tester ces fonctions à minima avec les grilles `grid00.in`, `grid01.in`, et `grid02.in`.

Rappel : vous pouvez vous aider de la documentation du code pour savoir ce qu'une méthode doit prendre en entrée et/ou retourner. N'oubliez pas également qu'il est indispensable de bien documenter tout votre code.

Question 4. Implémenter dans la classe `Solver` une méthode naïve/simple de résolution du problème (par exemple un algorithme glouton). Pour cela, créer une nouvelle classe (e.g., `SolverGreedy`) similaire à la classe `SolverEmpty` qui hérite de la classe `Solver`. On conseille également d'implémenter la function `score` de la classe `Solver`.
Évaluer la complexité de cette méthode en temps de calcul. Calculer le score final de votre solution. Est-il minimal ? (Cette dernière question nécessite une réflexion, vous pouvez par exemple chercher à trouver des exemples de grilles "difficiles", c'est-à-dire conduisant votre algorithme à prendre de mauvaises décisions.)
Vous pouvez proposer plusieurs solutions et les comparer. Est-il possible de trouver la solution optimale et si oui, avec quelle complexité ?

Note : on ne cherchera pas à être très formel sur le calcul de la complexité. L'idée est d'évaluer l'ordre de grandeur du nombre d'opérations. Il peut être utile dans le processus de reflexion de considérer certains cas particuliers de grilles.

Question 5. Implémenter une représentation graphique de la grille dans la méthode `plot` de la classe `Grid`.

On suggère ici de commencer par une représentation simple avec la librairie `matplotlib`.

Séance 2 : 10/02

La séance précédente devrait avoir permise de résoudre le problème avec une méthode simple, mais pas nécessairement optimale en score (en tout cas pas en un temps raisonnable). On propose maintenant une méthode basée sur les graphes permettant de trouver la solution optimale, dans le cas où les valeurs des cellules de la grille sont toutes les mêmes (par simplicité on suppose qu'elles sont toutes égales à 1). Dans ce cas, le problème se simplifie : l'objectif est simplement de prendre le plus grand nombre de paires possibles puisque chaque paire a un coût nul alors que les cellules non prises dans une paire ont un cout de 1.
L'idée est de voir le problème de sélectionner des paires comme un problème d'appariement (matching) : sélectionner une paire de cellules correspond à apparier ces deux cellules ensemble. On peut diviser les cellules en

deux ensembles : les cellules paires (i.e., les cellules (i, j) telles que $i + j$ est paire) et les cellules impaires (i.e., les cellules (i, j) telles que $i + j$ est impaire). [3] Du fait de la contrainte d'adjacence, une cellule paire ne peut être appariée qu'avec une cellule impaire et vice-versa. On peut donc construire un graphe biparti [4] avec les cellules paires à gauche et les cellules impaires à droite, et mettre une arête entre deux cellules si et seulement si elles peuvent être appariées ensemble (en respectant la contrainte d'adjacence et les contraintes de couleurs)—voir un exemple sur la Figure 2. Une liste de paires valide dans notre problème initial est alors un appariement dans ce graphe biparti, c'est-à-dire un sous-ensemble d'arêtes tel que chaque sommet est incident à au plus une arête du sous-ensemble.

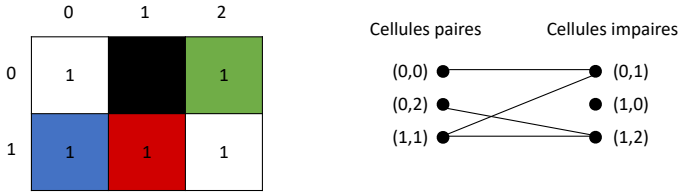

FIGURE 2 – Examples de transformation d'une grille en graphe.

Avec tout ceci en place, le problème initial de recherche d'une liste de paires qui minimise le coût total est équivalent à la recherche d'un appariement de cardinalité maximale dans le graphe biparti construit ci-dessus. C'est un problème classique en informatique et il existe des algorithmes efficaces pour le résoudre. En particulier, on peut utiliser l'analogie à un problème de flot et l'algorithme de Ford-Fulkerson pour le résoudre (parfois appelé algorithme des chemins « augmentants » dans le cas de graphes bipartis, l'idée étant de partir d'un appariement ou flot vide, et de chercher à l'augmenter itérativement). Les chargés de TP donneront une brève introduction à cet algorithme en début de séance 2. On peut aussi consulter les sections 7.1 et 7.5 de ce livre, ou tout autre ressource en ligne.

Question 6. Implémenter un solveur basé sur le matching maximal pour résoudre le problème dans le cas où les valeurs de chaque cellule sont toutes égales à 1. Tester sur les grilles `grid2.in`, etc. et comparer aux solutions de la séance 1. Discuter la complexité de votre algorithme.

On rappelle qu'il y a un rendu intermédiaire après la séance 2 (voir deadline précise ci-dessus). Il est attendu que toutes les questions ci-dessus soient abordées. Toutefois, il est tout à fait possible que certains les finissent plus tôt, on pourra alors évidemment passer directement à la suite sans attendre la séance 3. On rappelle toutefois qu'on prête une attention particulière à la qualité du code : organisation, bonnes pratiques, documentation, commentaires, etc. et qu'il ne faut pas passer trop vite à la suite avant d'avoir bien fait le début.

Séance 3 et 4 : 03/03 et 24/03

Ces 2 séances n'ont pas de structure imposée. On impose toutefois d'aborder le premier point, c'est-à-dire la résolution du problème dans le cas général. Ensuite, on pourra étendre le projet dans toutes les directions possibles ; on donne ici des suggestions :

1. Implémenter un solveur pour résoudre le problème dans le cas général, c'est-à-dire si les valeurs peuvent être des entiers arbitraires. Discuter la complexité. Pour cette question on suggère une piste : réfléchir au problème du "maximum weight matching" pour lequel il existe des algorithmes efficaces (que vous pouvez trouver facilement en cherchant en ligne).
2. Etudier une variante du problème. Par exemple, on peut considérer la variante où on lève la contrainte d'adjacence pour les paires de cellules blanches, c'est à dire qu'une cellule blanche peut être appariée aux cellules adjacentes de n'importe quelle couleur (sauf noire) ou à n'importe quelle cellule blanche qu'elle soit adjacente ou non.

3. Comme sur un échiquier, mais on évitera ici de parler de cellules noires et blanches puisqu'elles sont déjà un attribut couleur.
4. Un graphe est biparti si son ensemble de sommets V peut être divisé en deux sous-ensembles disjoints V_{gauche} et V_{droite} tels que chaque arête a une extrémité dans V_{gauche} et l'autre dans V_{droite}.

3. Aspects graphiques : même si l'objectif principal reste la partie algorithmique du projet, on pourra proposer une interface graphique (e.g., permettant de jouer de façon interactive en prenant des paires). On conseille de commencer par regarder la documentation de `pygame`.
4. Considérer la version à 2 joueurs : chaque joueur prenant une paire à tout de rôle. Implémenter un algorithme jouant de façon optimale.

Tout autre création complémentaire restant dans l'objectif du projet de programmation sera récompensée. On insiste toutefois à nouveau pour finir sur un point crucial : la qualité du code est primordiale dans le projet. Il ne faut donc pas se précipiter pour en faire "beaucoup" au détriment de la faire bien.

Séances 5 : 07/04

On rappelle que la deadline pour le rendu final (incluant le rapport au format PDF) est le dimanche 30/03 à 23h59. La séance 5, qui est après la deadline pour le rendu final, sera consacré à l'évaluation. Les enseignants auront lu les projets et passeront dans les groupes pour poser des questions. Cela sera aussi l'occasion de donner un retour aux étudiants sur leurs projets.