RO 203

Rapports de projet

DRÉAN Alexandre, BELHAJ Ayoub

29/04/2024

Table des matières

Introduction

Ce rapport traite du projet réalisé dans le cadre de ce cours. Ces projets permettent de mettre en pratique ce qui a été appris en théorie pendant les cours et TD, ils sont aussi l'occasion de découvrir un nouveau language de programmation, le **julia**

1 Premier Jeu : Rectangles (3 points)

1.1 Description du jeu

Retangles est un jeu dans lequel les joueurs sont confrontés à une grille de jeu de dimensions variables, généralement de taille $m \times n$. Dans la grille sont placés quelques des nombres, indiquant le nombre de cases que doit contenir le rectangle qui les contient. Les joueurs doivent alors diviser la grille en rectangles de différentes tailles, en suivant ces indications de nombres. Chaque rectangle doit être rempli de manière à ce qu'il n'y ait aucune case vide à l'intérieur. De plus, aucun rectangle ne doit se chevaucher. Le jeu se termine lorsque la grille est entièrement divisée en rectangles et que tous les rectangles sont correctement remplis.

1.2 Modélisation en PLNE

$$
\begin{aligned}
\min_{x \in \mathcal{X}} \quad & 1 \\
\text{s.c.} \quad & \forall k \in [[1,p]], \quad \sum_{l=1}^{q} x_{kl} = 1 \\
& \forall (i,j) \in [[1,n]] \times [[1,n]], \quad \sum_{k=1}^{p} \sum_{l=1}^{|R_k|} R_{kl}(i,j) \cdot x_{kl} = 1
\end{aligned} \tag{1}
$$

La première contrainte cherche à obtenir exactement un rectangle par nombre non nul dans notre grille :

- x_{kl} étant une variable binaire permettant d'indiquer si le rectangle l est actif pour le nombre k.
- q étant le nombre de rectangles possibles pour le nombre k dans la grille.

La seconde contrainte cherche à ce qu'aucun des rectangles ne se chevauche, il ne faut donc qu'aucun des pixels de la grille ne puisse être couvert par plus d'un rectangle actif :

- $R_{kl}(i,j)$ est un terme de contrainte qui vaut 1 si le pixel à la position (i,j) est couvert par le rectangle l associé au nombre k, et 0 sinon.
- x_{kl} est une variable binaire indiquant si le rectangle l est actif pour le nombre k.
- $|R_k|$ est le nombre de rectangles possibles pour le nombre k dans la grille.
- $n \times n$ est le nombre total de pixels dans la grille.

1.3 Implémentation

Dans cette partie nous décrirons les idées qui ont permis d'élaborer le code mais nous n'explicterons pas le code lui même, pour le consulter voir dans **./jeu2 (singles)/src**

1.3.1 Generation de Grilles

Le but de cette partie est de générer une grille dont on est sûr de la faisabilité. Afin d'assurer cela, il suffit de générer des solutions au jeu que l'on va masquer avant de les fournir comme entrée de notre fonction de résolution. La génération se fait donc en plusieurs étapes, d'abord on génère une grille de taille $m \times n$ contenant exclusivement des zéros. Par la suite on va couper en deux rectangles aléatoires notre grille et répéter cela sur les rectangles ainsi créés (on effectue cette étape un nombre de fois choisit, en fonction de la finesse de découpage que l'on recherche). Par la suite, on choisit aléatoirement une des cases de ce rectangles pour y inscrire la taille du rectangle à la place de 0. Lorsque l'on obtient ces rectangles, on se préoccupe de la résolution de la PLNE.

1.3.2 Resolution de Grilles

L'enjeu principale lors de la résolution est de générer l'ensemble des rectangles possibles pour chaque case dont la valeur est non nulle. On parcourt la grille et lorsque l'on rencontre un nombre non nul, on crée une list de paires de diviseurs, matérialisant les différents formats de rectangles possible. Par la suite, pour chacun de ces formats possibles, il faut fixer un retangle grace à son coin supérieur gauche et son coin inférieur droit.

1.4 Résultats

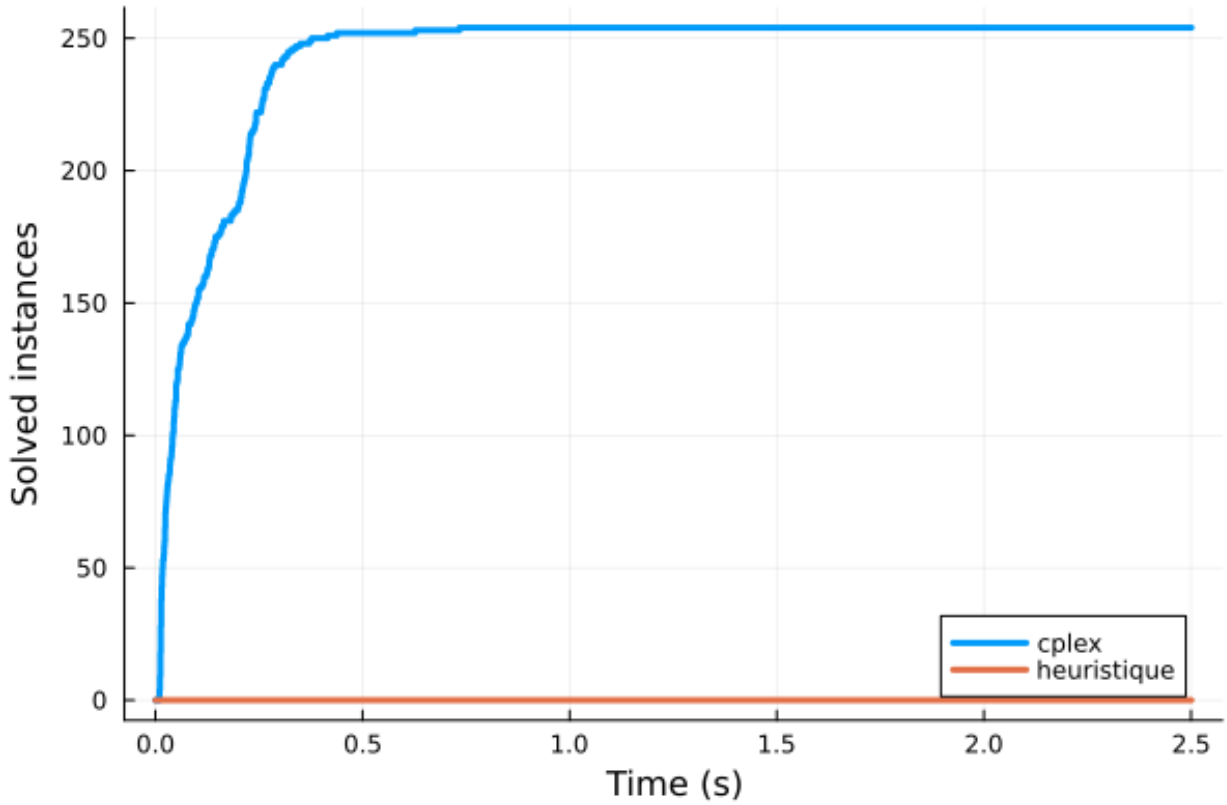

FIGURE 1 – Performances obtenus

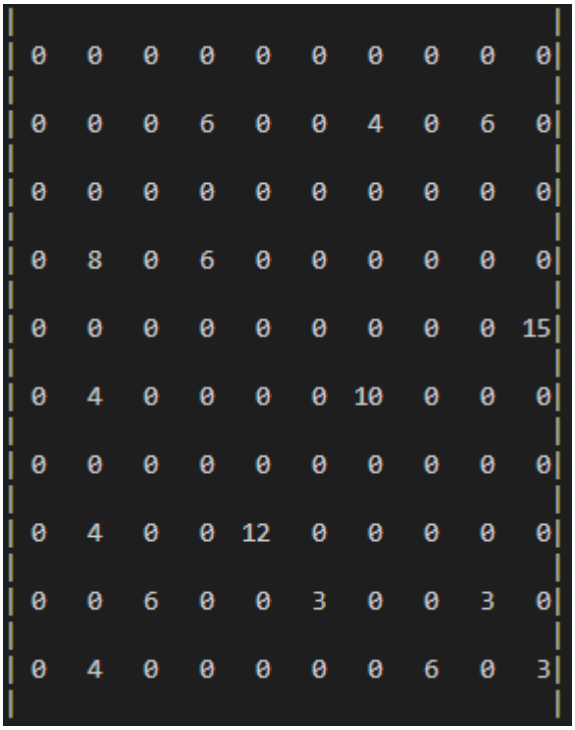

FIGURE 2 – Grille 10x10

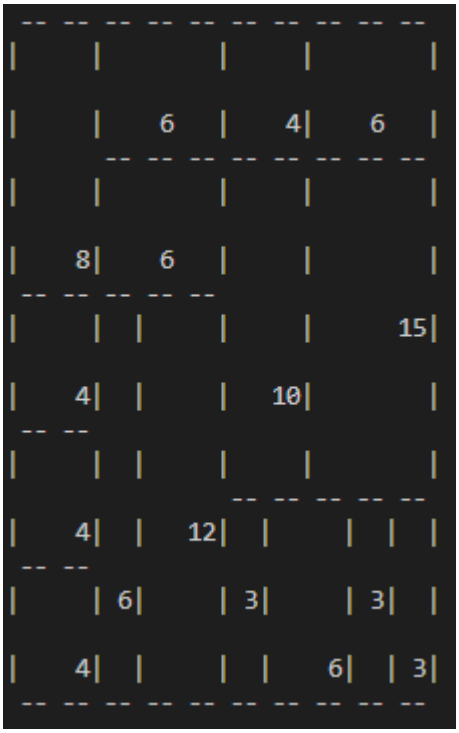

FIGURE 3 – Solution

2 Second jeu : Singles (7 points)

2.1 Description du jeu

blabla

2.2 Modélisation en PLNE

On représente une grille de taille (n, n) par un tableau représentés par les variables $G_{i,j}$ avec $1 \leq i, j \leq n$. Pour chaque case de cette grille, on introduit une variables binaire $x_{i,j}$ qui vaut 1 si on décide de noircir la case (i, j) et 0 sinon. On cherche donc des solutions dans l'ensemble $\mathscr{X}$ des matrices à coefficient binaire et de taille (n, n). La modélisation en PLNE choisie est donnée ci-dessous.

$$
\begin{aligned}
\min_{x \in \mathscr{X}} \quad & \sum_{1 \leq i,j \leq n} x_{i,j} \\
\text{s.c.} \quad & \forall j \in [[1,n]] \forall i, i' \in [[1,n]]^2 G_{i,j} = G_{i',j} \Rightarrow x_{i,j} + x_{i',j} \geq 1 \\
& \forall i \in [[1,n]] \forall j, j' \in [[1,n]]^2 G_{i,j} = G_{i,j'} \Rightarrow x_{i,j} + x_{i,j'} \geq 1 \\
& \forall i \in [[1,n-1]] \forall j \in [[1,n]] x_{i,j} + x_{i+1,j} \leq 1 \\
& \forall j \in [[1,n-1]] \forall i \in [[1,n]] x_{i,j} + x_{i,j+1} \leq 1
\end{aligned}
\tag{2}
$$

L'objective est choisi de façon a avoir le moins de cases noires possibles. Cela augmente nos chance que l'ensemble des cases blanches de la solution trouvée soit connexe. En effet, nous n'avons par réussi à trouver de contrainte pour imposé la connexité et nous avons donc choisis ce compromis.
La première contraintes exprime le fait que pour chaque colonne, si une valeur de la grille se

répète dans cette colonne, on doit obligatoirement en masqué une. La seconde contrainte est la même en permutant ligne et colonne.
La troisième et quatrième contrainte expriment l'interdiction d'avoir deux cases noircis adjacentes.

2.3 Implémentation

Dans cette partie nous décrirons les idées qui ont permis d'élaborée le code mais nous n'explicterons pas le code lui même, pour le consulter voir dans **./jeu1 (rectangles)/src**

2.3.1 Generation de Grilles

C'est clairement cette partie qui a été la plus délicate dans le cas de ce jeu. En effet si on se content de générer des grilles aléatoires de taille (n, n), ces grilles n'ont quasiment jamais de solution et donc elles n'étaient pas pertinente pour tester nos programmes de résolution. Nous avons donc dû un peu travailler pour générer des grilles résolubles.

Notre idée à été la suivante :

— On commence par initialiser des matrices G et X de zeros de tailles (n, n).
— On parcourt les lignes de X et pour chacune d'entre elles on choisis aléatoirement un sixième de ces cases qu'on colore en noir. (en mettant les valeurs de X à 1 pour ces cases). On s'assure également qu'aucune cases noires ne soient choisis de manière adjacentes.
— On parcourt les cases de G et pour chacune d'entre elles, si elle doit être noir (i.e $X[i, j] = 1$) on choisis un nombre au hasard entre 1 et n, sinon on choisis n'importe quel nombre entre 1 et n qui doit être compatible avec les cases déjà choisis sur le début de sa colonne de sa ligne.

Malheuresement, la dernière étape où on doit choisir un nombre aléatoire compatible avec les précédents n'est pas toujours possible et dépend beaucoup de l'aléa des tirage aléatoires. Lorsqu'un blocage se produit on signal en sortie de fonction que la grille générée n'est pas solvable afin de pouvoir généré des grilles en recommançant le processus jusqu'à en avoir une correcte.

2.3.2 Resolution de Grilles par PLNE

On implémente de façon transparente la modélisation effectuée dans la partie 2.2. Pour plus de lisibilité dans le code, les contraintes ont été écrites à l'intérieurs de boucles *for* et des conditions *if* au lieu d'écrire les boucles et conditions à l'intérieur des contraintes.

2.3.3 Resolution de Grilles par méthode heuristique

Afin de résoudre ce problème de façon heuristique, nous avons identifier des stratégies en essayant de compléter des grilles à la main. En voici un résumé :

— Pour commencer (ou relancer une partie au point mort), on essayer d'identifier des cases nous déjà traitées dont la couleur peut être déterminée avec certitude, Pour cela, on essaye de repérer des patternes.
Le premier patterne identifié est par exemple |1|2|1|, dans ce cas on est certain que la case au milieu devra être blanche car si elle était noir on aurait forcément deux cases noires adjacentes ce qui est interdit puisqu'au moins un des deux 1 devra être noir.
Le second patterne est par exemple |1|...|1|1|. Dans ce cas on sait que le 1 qui est tout seul devra forcément être noir car si il était blanc, on forcerait les deux 1 adjacents à être noir ce qui est interdit.

Enfin, lorsqu'on ne trouve pas de patterne, on se contente de coloré en noir la case dont la valeur est la plus répétée sur sa ligne et colonne, cela induit de l'aléa qui explique pourquoi cette méthode ne converge pas toujours vers une solution.

— Lorsque la partie est bien lancée, on a une pile de cases noires à traité et une pile de cases blanches.
— Si la pile de cases noires est non vide, on dépile et on traite les cases noires, pour cela on sait que les 4 cases adjacentes à une case noire vont forcément êtres blanches donc on les ajoute à la pile des cases blanches.
— Si la pile de cases blanches est non vide, on dépile et on traite les cases blanches, pour cela on sait que les cases sur la même ligne et de même valeur doivent être noires donc on les ajoutes à la pile noir, de même pour les cases sur la même colonne et de même valeur.

Avec cette stratégie, avec un peu de chance on repère un patterne qui nous assure de ne pas faire d'erreur, puis l'inertie des empilages et dépilages permet également de diminuer le nombre de cases sans faire d'erreur logique. Malheuresement, lorsque les piles sont vides et qu'on ne repère pas de patterne on doit faire des choix arbitraire et souvent non optimaux qui ne permettent pas toujours de converger vers une solution.

2.4 Résultats

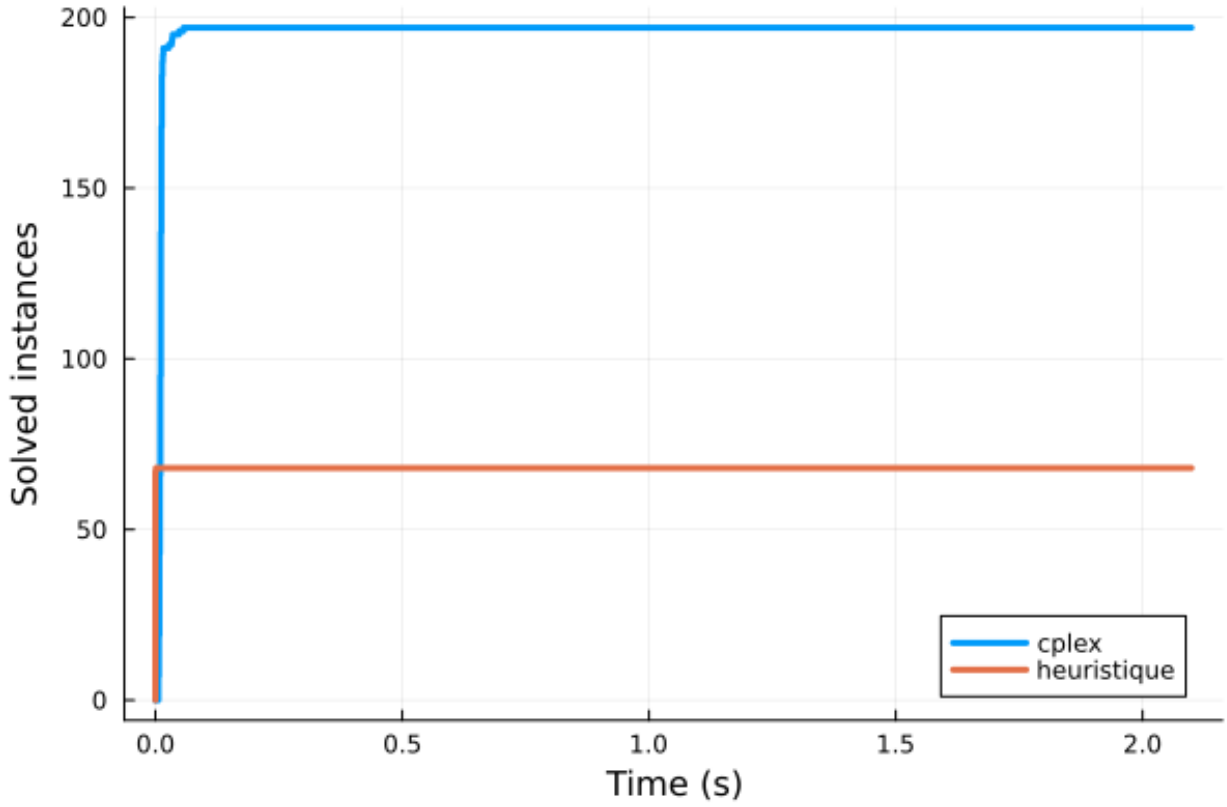

FIGURE 4 – Performances obtenus

Sur les 200 fichiers générés, tous résoluble, le PLNE à trouvé une solution à tous ces problèmes là où la stratégie heuristique n'en a résolu qu'une soixantaine. En regardant les résultats de plus près sur le rapport de test, on constate que la méthode heuristique s'en sort très bien

pour des petites tailles jusqu'à (6,6) mais pour des tailles plus grandes cettes stratégies est beaucoup moins performante.

Concernant l'aspect de ces courbe, celui-ci montre que les problèmes sont résolus en un temps infime (0.01s pour cplex et 0.0s affiché pour heuristique lorsqu'une solution a été trouvée). Cela est dûe au fait que le temps de génération des grilles avec nos algorithmes est bien plus long que le temps de résolution. Avec notre algorithme nous ne pouvons pas générer de grille de taille supérieur à 15 en un temps raisonnable, alors même que ces grilles sont résolus instantanément par CPLEX.

```
solveTime = 0.01100015640258789
isOptimal = true
solution = [0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0; 1.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0
1.0 0.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0; 0.0 0.0 0.0 0.0
0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0]
# solution trouvée
# |------+------+------+------+------+------+------+------+------+------+------+------+
# | 11   |  4   |  6   |  7   |  2   |  5   |      | 12   |  8   | 10   |  3   |  9   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  7   |  6   | 11   |  9   |  1   |  2   | 10   |  8   |  5   |  3   |  4   | 12   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  2   |  8   | 10   |  1   |  3   | 11   | 12   |  6   |  7   |  5   |  9   |  4   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# | 10   | 12   |  9   |  3   |  4   |  8   |  5   |  7   |  6   |      | 11   |  2   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# | 12   | 11   |  3   |  2   |  5   |  6   |  9   | 10   |  1   |  4   |  7   |      |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |      |  1   |  4   | 10   |  9   |  3   |  6   | 11   |  2   |  7   |  5   |  8   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  1   |  5   |  7   | 12   |  8   |  4   | 11   |  9   | 10   |      |  6   |  3   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  9   |  7   |  1   |  6   | 12   | 10   |      |  3   |  4   |  8   |  2   |  5   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  6   |  3   |  5   |  4   | 10   |      |  1   |  2   | 12   | 11   |  8   |  7   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  5   |  2   |  8   | 11   |  6   |  9   |  4   |      |  3   |  1   | 12   | 10   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  3   |  9   | 12   |  8   |  7   |  1   |  2   |  4   |      |  6   | 10   | 11   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
# |  8   | 10   |  2   |  5   | 11   |  7   |  3   |      |  9   | 12   |  1   |  6   |
# |------+------+------+------+------+------+------+------+------+------+------+------+
```

FIGURE 5 – Exemple de grille résolu par CPLEX de taille (12,12)

La figure ci-dessus montre un exmple de grille résolu par CPLEX, on constate que les contraintes sont bien respectés, en particulier la connexité des cases blanches.

2.5 Difficultés rencontrées

Les problèmes liés aux temps de génération de grilles comparé aux temps de résolution ont été abordés dans la partie précédente.

L'autre problème majeur rencontré est lors de la résolution par méthode heuristique, en effet nous somme obligé de faire face à de l'aléa lorsqu'on ne trouve aucun patterne, on aurait pu se dire que lorsqu'on test de relancer le jeu avec un choix arbitraire qui n'est pas sûre on devrait pouvoir revenir en arrière en cas d'échec, cependant nous avons été à court de temps,

de plus cette approche ressemble beaucoup à du branch and bound et cela perd en intérêt dans le cadre d'une méthode heuristique selon nous.

D'un point de vue plus théorique, nous avons échoué à écrire des contraintes mathématique modélisant la connexité malgré une tentative en introduisant des variables binaire $Y(i, j, k, l)$ valant 1 si les cases (i, j) et (k, l) sont dans la même composante connexe et 0 sinon. Cependant, nous avons constaté qu'en pratique, toutes les solutions trouvés respectaient la contrainte de connexité, cela est sans doute dûe à l'objectif d'avoir le moins de cases noires possibles.