RO 203

Rapport de projet

Baptiste MONTAGNES - Julien SEGONNE

27 Avril 2024

Table des matières

Introduction

Ce rapport vise à présenter nos programmes linéaires en nombres entiers ainsi que l'implémentation du modèle adapté que nous avons créé afin de résoudre deux jeux. Nous nous sommes intéressés aux jeux Bridges qui valait 4 points et Singles qui valait 6 points. Nos codes sont joints avec ce rapport mais ils se trouvent également en annexe par précaution.

Remarque : Dans le code du deuxième jeu, nous appelons une fonction déjà implémentée dans le dossier sudoku1.0, si vous souhaitez compiler notre code, cela nécessitera d'avoir le dossier avec les codes sources.

1 Jeu 1 : Bridges

1.1 Présentation du jeu

Dans ce jeu, nous disposons d'un certain nombre de sommet à relier entre eux. Deux sommets ne peuvent être reliés que s'ils sont alignés verticalement ou horizontalement sans aucun autre sommet entre eux. Dans ce cas, il est possible de placer 1 ou 2 ponts entre ces deux sommets. De plus, le nombre de ponts partant d'un sommet est égal au nombre indiqué sur le sommet.

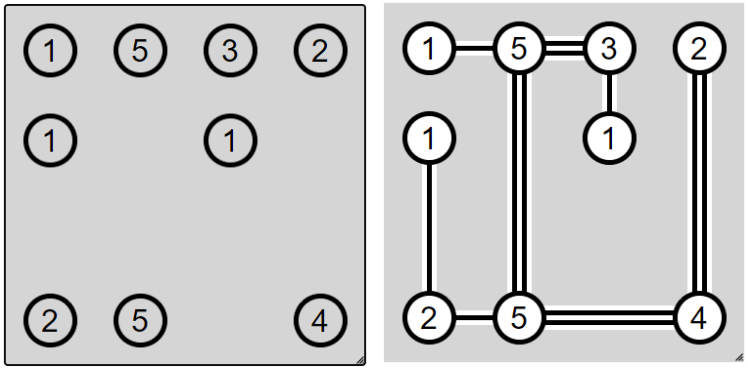

FIGURE 1 – Exemple d'une partie

1.2 Modélisation du problème

On suppose que tous les sommets sont numérotés de 1 à n $\in \mathbf{N}^*$.

Soit $B = 0$, la fonction objectif qui demeurera constante car il n'y a pas réellement de fonction à maximiser ou minimiser.

Soit $V \in M_{1,n}(\mathbb{N}^*)$, le vecteur regroupant le nombre de ponts partant de chaque sommet. Par exemple, V_i représente le nombre de ponts liés au sommet i.

$\forall i, j \in [\![1, n]\!]$, x_{ij} représente le nombre de ponts reliant le sommet i au sommet j. Dans notre cas, x_{ij} peut prendre les valeurs 1 et 2 s'il y a un 1 ou 2 ponts entre i et j ; 0 s'il n'y a aucun pont.

On peut donc modéliser le jeu Bridges par ce PLNE :

$$
(P_1)\begin{cases}
\text{Maximiser } B \\
\text{s.c. :} \\
\quad \forall i \in [\![1,n]\!],\ \sum_{j\neq i} x_{ij} = V_i \\
\quad \forall i,j \in [\![1,n]\!],\ x_{ij} = x_{ji} \\
\quad \forall i,j \in [\![1,n]\!],\ 0 \leq x_{ij} \leq 2 \\
\quad \forall i,j,k,l \in [\![1,n]\!] \text{ tels qu'un pont reliant i à j et un pont reliant k à l se croisent,} \\
\qquad x_{ij} = 0 \text{ ou } x_{lk} = 0
\end{cases}
$$

La première contrainte impose que le nombre de ponts liés aux sommets soit le bon. Ensuite, la deuxième contrainte impose qu'il y ait autant de ponts reliant i à j et j à i. La troisième contrainte impose qu'il n'y ait pas plus de 2 ponts reliant 2 mêmes sommets. Enfin, la quatrième contrainte impose qu'aucun pont ne se croise.

1.3 Modélisation informatique

1.3.1 Création d'une instance

Pour créer une instance, nous commençons par fixer une taille de grille n et une densité. Puis :

— On initialise une matrice t de taille n x n à zéro qui contiendra les sommets.

— On initialise une matrice occupied_cells de taille n x n à zéro qui contiendra les cases de la grille occupées par un pont ou un sommet.

— On place 2 arêtes initiales connectées par un pont simple ou double (aléatoire). Ces arêtes sont placées aléatoirement sur la grille et ajoutées dans t.

— Tant que la densité souhaitée n'est pas atteinte :

- On choisit un sommet non connecté à l'ensemble déjà créé.
- On vérifie qu'il est possible de le relier à l'ensemble de sommets existants.
- On vérifie si le nouveau pont ne chevauche pas ou ne croise pas d'autres ponts déjà existants.

L'algorithme s'arrête soit lorsqu'il atteint la densité souhaitée, soit lorsqu'il atteint le nombre maximal d'itérations (dans le cas où il ne peut pas atteindre la densité souhaitée sans chevaucher les ponts). Si le nombre maximal d'itérations est atteint, on recommence le processus depuis le début. Le nombre maximal d'itérations garantit que le programme ne tourne pas en boucle dans une configuration où il n'est plus possible d'ajouter de nouveaux sommets.

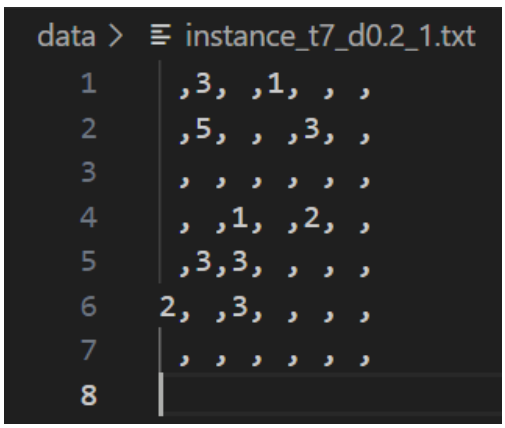

FIGURE 2 – Exemple d'une instance générée avec cet algorithme

1.3.2 Résolution d'une instance

La fonction qui permet de résoudre une instance prend en paramètre une matrice de taille n x n qui contient des zéros là où il n'y a pas de sommet et la valeur du sommet sinon. Elle procède de la manière suivante :

— On crée un modèle d'optimisation avec CPLEX comme solveur

— On défini la variable à optimiser comme étant une liste x de 5 dimensions à valeurs binaires. Ainsi, si x[i, j, i', j', k] = 1 alors, les sommets (i, j) et (i', j') sont reliés par $k-1$ ponts, où $i, j, i', j' \in [\![1, n]\!]$ et $k \in [\![1,3]\!]$ (car Julia ne prend que des indices de liste commençant à 1).

— On ajoute les contraintes pour garantir que :

- un point de la grille sans sommet ne sera relié à aucun autre sommet,
- un sommet ne peut pas être relié à lui-même,
- deux sommets ne peuvent pas être reliés s'ils ne sont pas voisins
- le sommet (a,b) est autant de fois relié à (c,d) que le sommet (c,d) à (a,b)
- la somme des poids des arêtes autour de chaque sommet est égale au poids spécifié.

— On configure l'objectif de maximisation comme étant nul. En effet, la fonction objectif est sans importance dans ce cas, car nous cherchons simplement à vérifier la faisabilité des contraintes.

— On résout grâce au solveur CPLEX

— On retourne un triplet contenant :

- true si une solution réalisable est trouvée, sinon false,
- les valeurs des variables de décision x,
- le temps pris pour résoudre le modèle.

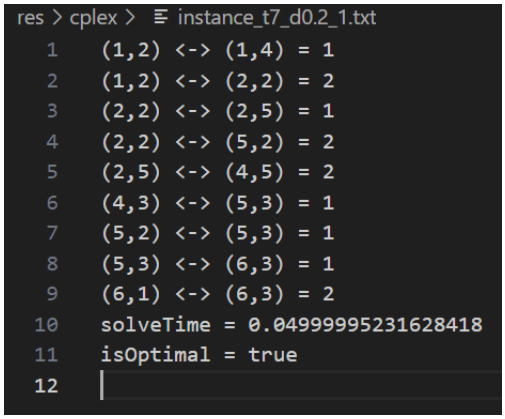

FIGURE 3 – Exemple d'une résolution à partir de l'instance affichée figure 2

L'instance résolue contient la liste des sommets avec le sommet auquel ils sont reliés et la valeur du pont. A cela, on a ajouté le temps de résolution et si on a trouvé une solution.

De plus, lorsque nous générons un grand nombre d'instances, il est possible de récupérer les temps de résolution et les valeurs d'optimalité pour chacune d'entre elles et de générer un rendu sous Latex :

	cplex	
Instance	Temps (s)	Optimal ?
instance_t5_d0.25_1.txt	0.131	×
instance_t5_d0.25_2.txt	0.059	×
instance_t5_d0.2_1.txt	0.053	×
instance_t5_d0.2_2.txt	0.066	×
instance_t5_d0.3_1.txt	0.048	×
instance_t5_d0.3_2.txt	0.05	×
instance_t7_d0.25_1.txt	0.097	×
instance_t7_d0.25_2.txt	0.097	×
instance_t7_d0.2_1.txt	0.084	×
instance_t7_d0.2_2.txt	0.131	×
instance_t7_d0.3_1.txt	0.085	×
instance_t7_d0.3_2.txt	0.083	×
instance_t9_d0.25_1.txt	0.167	×
instance_t9_d0.25_2.txt	0.17	×
instance_t9_d0.2_1.txt	0.557	×
instance_t9_d0.2_2.txt	0.209	×
instance_t9_d0.3_1.txt	0.199	×
instance_t9_d0.3_2.txt	0.16	×

FIGURE 4 – Liste des temps et des valeurs d'optimalité pour des grilles de taille 5, 6, 7 et pour des densités de 0.2, 0.25, 0.3

1.3.3 Affichage graphique dans la console

Deux fonctions ont été codées afin de fournir un affichage graphique dans la console. La première est utilisée pour les instances non résolues tandis que la deuxième est utilisée pour les instances résolues. Le résultat de ces fonctions est disponible ci-dessous pour l'instance utilisée figure 2 et figure 3 :

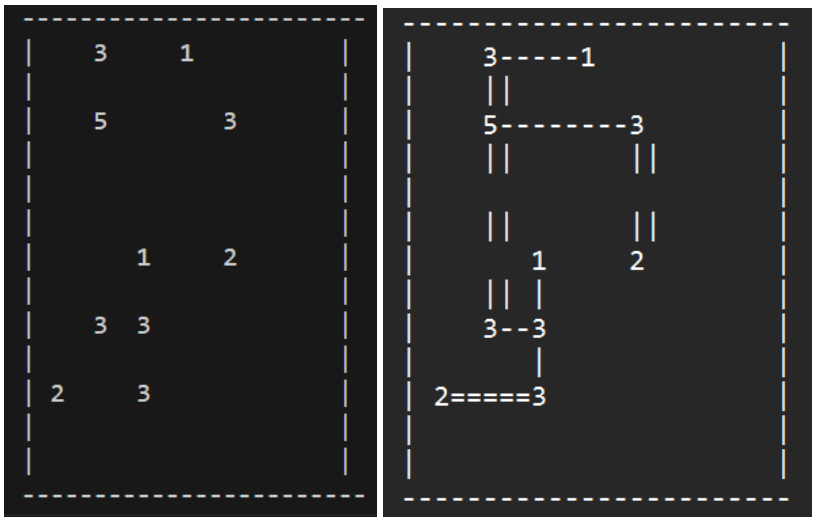

FIGURE 5 – Affichage graphique d'une instance non résolue et d'une instance résolue

1.4 Résultats des méthodes de résolution

1.4.1 Impact de la taille des grilles et de la densité sur le temps de résolution

Nous avons fait plusieurs résolutions pour des densités et des tailles de grille différentes :

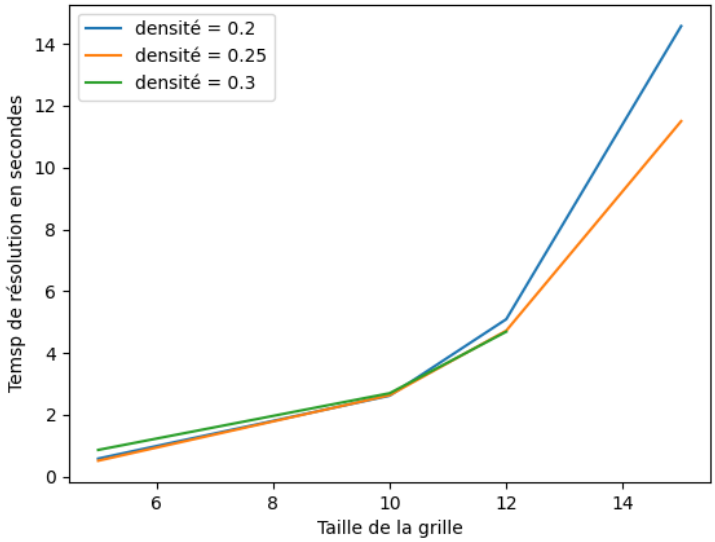

FIGURE 6 – Tracé du temps de résolution pour 10 instances avec une variation de la densité et de la taille de la grille

La valeur pour une densité de 0.3 et une taille de grille de 15 est manquante car le temps de génération des instances était beaucoup trop important.

Sur les résolutions faites précédemment, il semble que la taille de la grille impacte plus le temps de résolution que la densité de la grille. Cependant, il faudrait faire varier de manière plus significative la densité pour s'assurer que c'est vraiment la taille de la grille qui impacte le temps de résolution.

Il serait également intéressant d'étudier les mêmes paramètres sur le temps de génération des instances mais là n'est pas l'intérêt de ce projet.

1.4.2 Existence d'une solution

Pour une densité de 0.2, nous avons testé jusquà une taille de grille de 21. Jusqu'à cette valeur, la résolution était toujours exacte. Le problème à partir de cette valeur est le temps de génération des instances qui devient vraiment très conséquent. Il serait possible de tester une grille plus grande en la générant à partir du site *https://www.chiark.greenend.org.uk/sgtatham/puzzles/js/br* mais ce serait trop fastidueux de rentrer 22 x 22 valeurs.

Nous avons testé des instances contenant des cycles ce qui n'avait pas été fait auparavant (c'est un cas qui ne peut pas être générer par notre processus de génération d'instances). La résolution arrive à trouver un optimal mais le résultat proposé est faux.

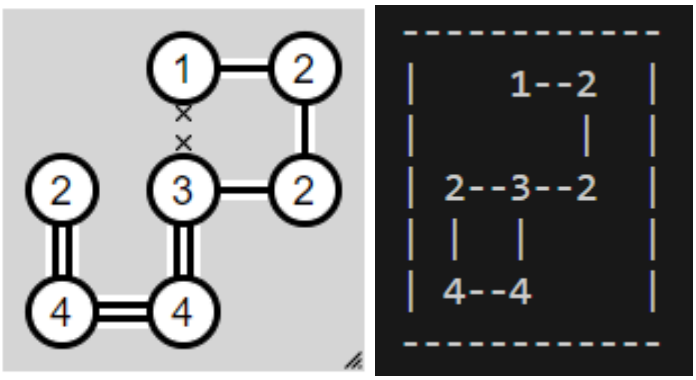

FIGURE 7 – Exemple contenant une boucle pour lequel notre résolution ne fonctionne pas

2 Jeu 2 : Singles

2.1 Présentation du jeu

Dans ce jeu, nous disposons d'une grille remplie de chiffres. Le but est de masquer certains chiffres de manière à ce qu'aucun chiffre ne soit visible plus d'une fois sur chaque ligne et chaque colonne. De plus, il faut que les cases masquées ne soient pas adjacentes (elles peuvent être adjacentes en diagonales) et que l'ensemble des cases visibles soit connexe.

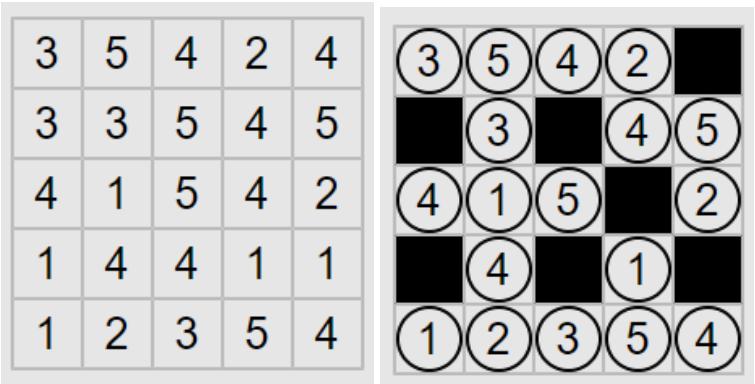

FIGURE 8 – Exemple d'une partie

2.2 Modélisation du problème

Soit $B = 0$, la fonction objectif qui demeurera constante car il n'y a pas réellement de fonction à maximiser ou minimiser.

Soit $A \in M_{n,n}(\llbracket \mathbf{1}, \mathbf{n} \rrbracket)$, la matrice représentant la grille du jeu.

$\forall i, j \in \llbracket 1, n \rrbracket, \forall k \in \llbracket 0, n \rrbracket$, x_{ijk} vaut 1 si la valeur se trouvant (i,j) vaut k et 0 sinon. Le cas k = 0 représente une case noire.

On peut donc modéliser le jeu Singles par ce PLNE :

$$(P_2) \begin{cases} \text{Maximiser } B \\ \text{s.c. :} \\ \quad \forall j, k \in \llbracket 1, n \rrbracket, \ \sum_{i=1}^{n} x_{ijk} \leq 1 \\ \quad \forall i, k \in \llbracket 1, n \rrbracket, \ \sum_{j=1}^{n} x_{ijk} \leq 1 \\ \quad \forall i \in \llbracket 1, n-1 \rrbracket, \forall j \in \llbracket 1, n \rrbracket, x_{i,j,0} + x_{i+1,j,0} \leq 1 \\ \quad \forall i \in \llbracket 1, n \rrbracket, \forall j \in \llbracket 1, n-1 \rrbracket, x_{i,j,0} + x_{i,j+1,0} \leq 1 \\ \quad \text{L'ensemble des cases blanches est connexe} \end{cases}$$

La première et la deuxième contrainte imposent que chaque chiffre soit au plus une fois dans une ligne et dans une colonne.
La troisième et la quatrième contrainte imposent qu'il n'y ait pas 2 cases noires adjacentes.
Enfin, la dernière contrainte est la plus difficile à modéliser, nous allons chercher des contraintes plus simples à exprimer afin de se rapprocher au mieux de cette contrainte.

2.3 Modélisation informatique

2.3.1 Création d'une instance

Pour créer une instance, nous commençons par fixer une taille de grille n et une densité. Puis :

— On crée une grille définit comme ceci :

$$\begin{array}{ccccc} 1 & 2 & 3 & 4 & \dots \\ n & 1 & 2 & 3 & \dots \\ n-1 & n & 1 & 2 & \dots \\ n-2 & n-1 & n & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Cela permet d'avoir une grille de base avec une seule fois chaque nombre par ligne et par colonne.

— On échange aléatoirement des colonnes deux par deux pour introduire de l'aléatoire dans la grille. On fait la même chose pour les lignes.

— On supprime de manière aléatoire des cases de la grille jusqu'à ce que la densité soit inférieure à la valeur fixée. La suppression est effectuée en vérifiant que les cases voisines ne sont pas déjà supprimées et en vérifiant que l'ensemble des nombres restant est toujours connexe.

— On remplit les cases supprimées avec des nombres aléatoires de 1 à n.

Ici, on ne fixe pas de nombre maximal d'itérations pour supprimer une case. En effet, ce procédé est beaucoup plus dépendant de la valeur de la densité fixéé que de la configuration qui a commencé à ête générée. Ainsi, recommencer et générer une autre configuration risquerait trop souvent de ne pas fonctionner.

Alternative

Pour créer les instances, nous aurions aussi pu générer des grilles aléatoirement et leur imposer des conditions restrictives assez fortes afin de tendre vers un modèle réalisable, nous avions pensé aux conditions suivantes :

— chaque chiffre doit apparaître au moins une fois dans la grille (meilleur diversité de chiffres et pas de cas ou une grille contient un grand nombre d'apparitions du même chiffre)

— chaque chiffre apparaît au maximum n+3 fois (nombre observé en regardant beaucoup de grilles sur le site internet) dans la grille

— chaque chiffre n'apparaît pas plus de 2 fois sur une même ligne ou une même colonne (nombre observé également)

2.3.2 Résolution d'une instance

La fonction qui résout une instance prend en paramètre une matrice carrée de taille n x n de nombres tous compris entre 1 et n. Elle procède ainsi :

— On crée le modèle d'optimisation sous CPLEX

— On définit la variable à optimiser comme une liste x à 3 dimensions à valeurs binaires : si $x[i, j, k] = 1$ alors la coordonnée (i, j) vaut k où $(i, j) \in [\![1, n]\!]$ et $k \in [\![0, n]\!]$ (nous pensions que le fait que k puisse prendre la valeur 0 pose problème mais en réalité non, l'algorithme résout sans erreur l'instance, une alternative aurait été de prendre n+1 pour les cases noires)

— On ajoute les différentes contraintes :
  - ► chaque case contient au moins une valeur k (soit la valeur de la même case sur la matrice d'entrée, soit 0 si la case est noire)
  - ► chaque ligne et chaque colonne ne contienne au plus qu'une seule fois la valeur d'un nombre
  - ► il n'existe pas 2 cases noires adjacentes
  - ► il n'existe pas de diagonales de cases noires
  - ► il n'existe pas de cases blanches isolées (pas de losange ou de triangle sur un bord)
— Comme précédemment, on configure l'objectif de maximisation comme étant nul. En effet, la fonction objectif est sans importance dans ce cas, car nous cherchons simplement à vérifier la faisabilité des contraintes.
— On résoût grâce au solveur CPLEX
— On retourne un triplet contenant :
  - ► true si une solution réalisable est trouvée, sinon false,
  - ► les valeurs des variables de décision x,
  - ► le temps pris pour résoudre le modèle.

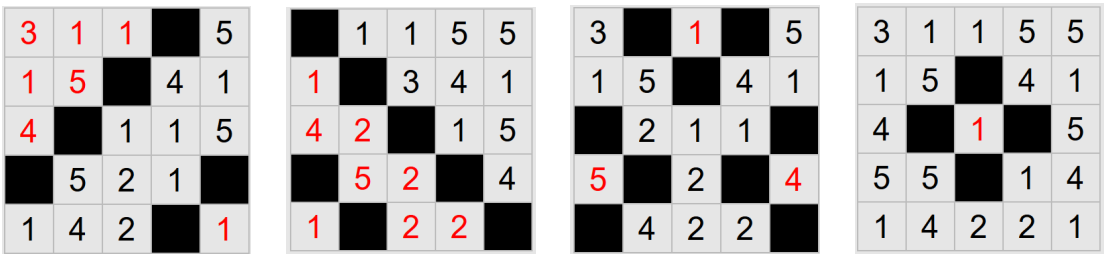

FIGURE 9 – Exemples des figures interdites

Voici les cas que nous avons traités pour les contraintes, il en existe d'autres (et de plus en plus lorsque la taille de la grille grandit) dont nous montrerons quelques exemples en annexe (Singles : Formes à implémenter pour les contraintes.

Conclusion sur la connexité

Nous avons pu tirer différents enseignement de l'étude de cette connexité :

— Si la taille de la grille est petite, l'ensemble des formes interdites est identifiable. En revanche pour de grandes grilles, le nombre devient trop grand.
— En prenant un peu de temps, on peut pas à pas réussir à coder ces contraintes mais elles sont de plus en plus complexes (par exemple les paternes de lignes brisées).
— Pour notre étude, il n'est pas nécessaire de coder l'ensemble des contraintes pour obtenir des résultats, celles déjà codées recouvrent une grande partie des cas où la grille n'est pas connexe après résolution.

2.3.3 Affichage graphique dans la console

Nous avons codé une fonction pour afficher les grilles non résolues (avec toutes les cases remplies) et les grilles résolues. Nous avons opté pour un affichage épuré, sans quadrillage pour ne pas nuire à la lecture, déjà compliquée pour ce genre de grille. Les cases noires sont représentées par des espaces.

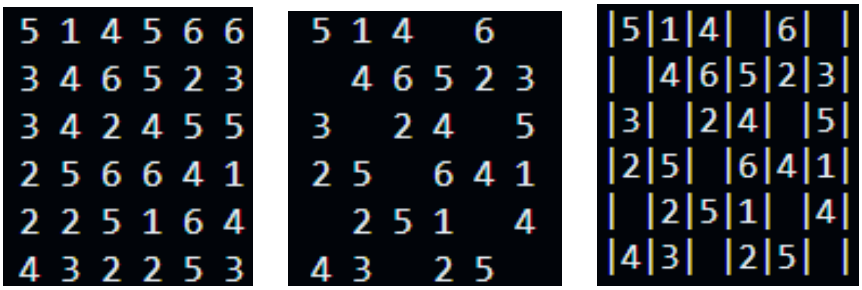

FIGURE 10 – Affichage des grilles (en dernier avec le quadrillage)

2.3.4 Méthode heuristique

Les différentes étapes de la méthode heuristique sont :

— On parcourt chaque élément de la grille et on compte le nombre d'occurrences de chaque chiffre. Les chiffres qui apparaissent le plus dans la grille seront ceux à prioriser lors du masquage.

— On trie cette liste pour obtenir les chiffres les plus utilisés en premier.

— On parcours les coins de la grille. Si un chiffre présent dans un coin est présent sur la même ligne et la même colonne que ce coin, on masque le coin. On commence par les coins pour éviter qu'ils ne se fassent isoler.

— On parcourt le reste des points par ordre de fréquence. Si pour un point, le chiffre est présent sur la même ligne et sur la même colonne, on masque le point s'il respecte les conditions du jeu.

— On parcourt le reste des points par ordre de fréquence. Si pour un point, le chiffre est présent sur la même ligne ou sur la même colonne, on masque le point s'il respecte les conditions du jeu.

— On retourne la grille avec les chiffres masqués.

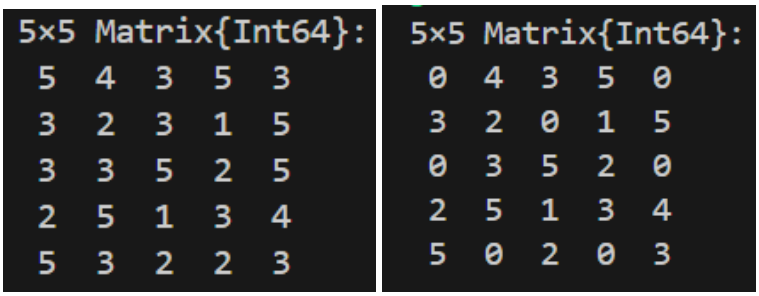

FIGURE 11 – Exemple de matrice en entrée et en sortie de la méthode

2.4 Résultats des méthodes de résolution

Voici les résultats obtenus. La fonction heuristique n'a pas bien fonctionné pour toutes les tailles de grille, nous ne pouvons donc pas exploiter les résultats de cette dernière. En revanche, nous pouvons constater que pour la résolution CPLEX, comme il était attendu, le temps de résolution augmente avec la taille de la grille. De plus, jusqu'à une taille de 10 x 10 nous trouvons systématiquement une solution.

	cplex		heuristique	
Instance	**Temps (s)**	**Optimal ?**	**Temps (s)**	**Optimal ?**
instance_t10_1.txt	0.08	×	0.08	×
instance_t10_10.txt	0.08	×	0.08	×
instance_t10_2.txt	0.08	×	0.08	×
instance_t10_3.txt	0.08	×	-	-
instance_t10_4.txt	0.08	×	-	-
instance_t10_5.txt	0.1	×	-	-
instance_t10_6.txt	0.12	×	-	-
instance_t10_7.txt	0.12	×	-	-
instance_t10_8.txt	0.12	×	-	-
instance_t10_9.txt	0.12	×	-	-
instance_t4_1.txt	0.04	×	-	-
instance_t4_10.txt	0.04	×	-	-
instance_t4_2.txt	0.12	×	-	-
instance_t4_3.txt	0.03	×	-	-
instance_t4_4.txt	0.03	×	-	-
instance_t4_5.txt	0.04	×	-	-
instance_t4_6.txt	0.04	×	-	-
instance_t4_7.txt	0.05	×	-	-
instance_t4_8.txt	0.04	×	-	-
instance_t4_9.txt	0.04	×	-	-
instance_t5_1.txt	0.05	×	-	-
instance_t5_10.txt	0.04	×	-	-
instance_t5_2.txt	0.04	×	-	-
instance_t5_3.txt	0.04	×	-	-
instance_t5_4.txt	0.05	×	-	-
instance_t5_5.txt	0.05	×	-	-
instance_t5_6.txt	0.06	×	-	-
instance_t5_7.txt	0.12	×	-	-
instance_t5_8.txt	0.07	×	-	-

	cplex		heuristique	
Instance	**Temps (s)**	**Optimal ?**	**Temps (s)**	**Optimal ?**
instance_t5_9.txt	0.08	×	-	-
instance_t6_1.txt	0.05	×	-	-
instance_t6_10.txt	0.07	×	-	-
instance_t6_2.txt	0.07	×	-	-
instance_t6_3.txt	0.06	×	-	-
instance_t6_4.txt	0.05	×	-	-
instance_t6_5.txt	0.08	×	-	-
instance_t6_6.txt	0.06	×	-	-
instance_t6_7.txt	0.05	×	-	-
instance_t6_8.txt	0.06	×	-	-
instance_t6_9.txt	0.06	×	-	-
instance_t8_1.txt	0.09	×	-	-
instance_t8_10.txt	0.22	×	-	-
instance_t8_2.txt	0.08	×	-	-
instance_t8_3.txt	0.09	×	-	-
instance_t8_4.txt	0.1	×	-	-
instance_t8_5.txt	0.12	×	-	-
instance_t8_6.txt	0.1	×	-	-
instance_t8_7.txt	0.23	×	-	-
instance_t8_8.txt	0.08	×	-	-
instance_t8_9.txt	0.08	×	-	-
instance_t10_1.txt	0.08	×	0.08	×
instance_t10_10.txt	0.08	×	0.08	×
instance_t10_2.txt	0.08	×	0.08	×

FIGURE 12 – Résultats de la résolution du dataset

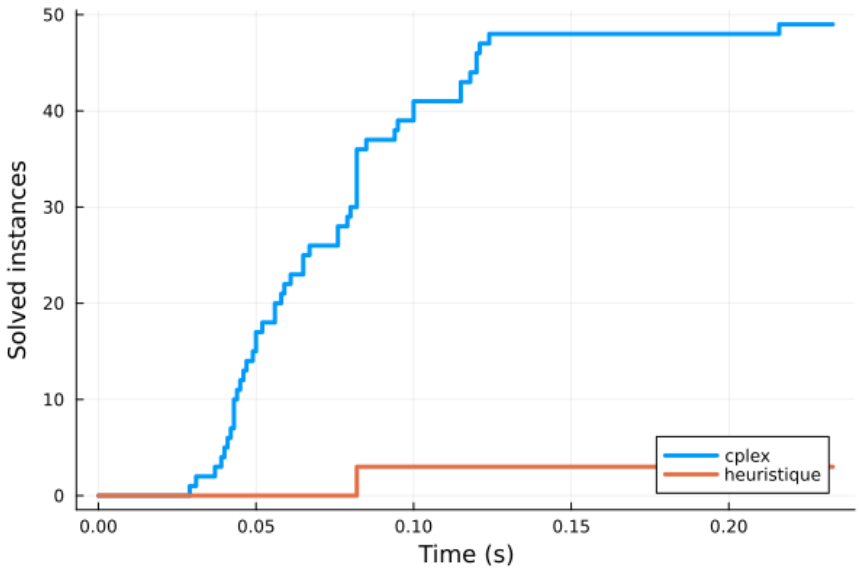

FIGURE 13 – Résultats de la résolution du dataset : graphique

3 Annexes

3.1 Singles : Formes à implémenter pour les contraintes

3.1.1 Losanges

Nous avons constaté que nous aurions pu généraliser le cas de la case isolée (petit losange) pour des plus grands losanges et imposer qu'aucun losange ne soit créé dans toute la grille.

6	12	11	■	7	10	2	2	8	12	5	10
4	2	■	3	■	6	10	8	1	12	2	1
1	■	2	10	6	■	2	7	4	7	1	6
■	10	1	10	4	8	■	9	2	7	3	2
2	■	11	6	12	■	3	11	1	4	12	10
5	4	■	12	■	11	8	4	5	8	6	2
8	4	5	■	10	10	6	4	■	2	12	9
5	6	4	2	3	1	12	■	7	■	8	1
12	2	1	7	7	3	■	1	9	9	■	12
1	12	6	9	5	2	7	■	8	■	9	6
7	1	12	8	11	3	4	5	■	3	10	6
10	10	3	9	3	4	1	1	12	4	11	9

FIGURE 14 – Grands losanges

3.1.2 Triangles

De même, les triangles sur les bords auraient pu être généralisés à des dimensions plus grandes.

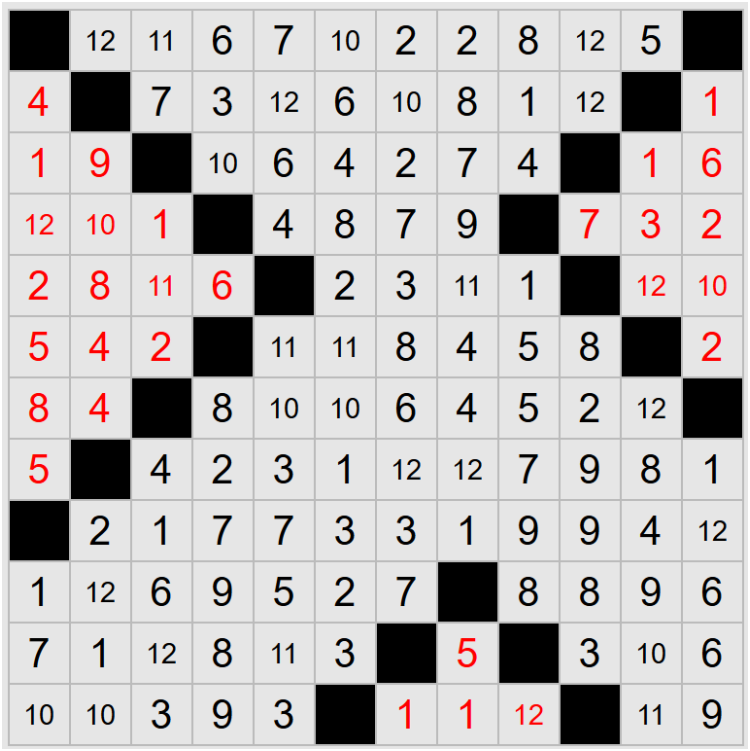

FIGURE 15 – Grands triangles

3.1.3 Rectangles

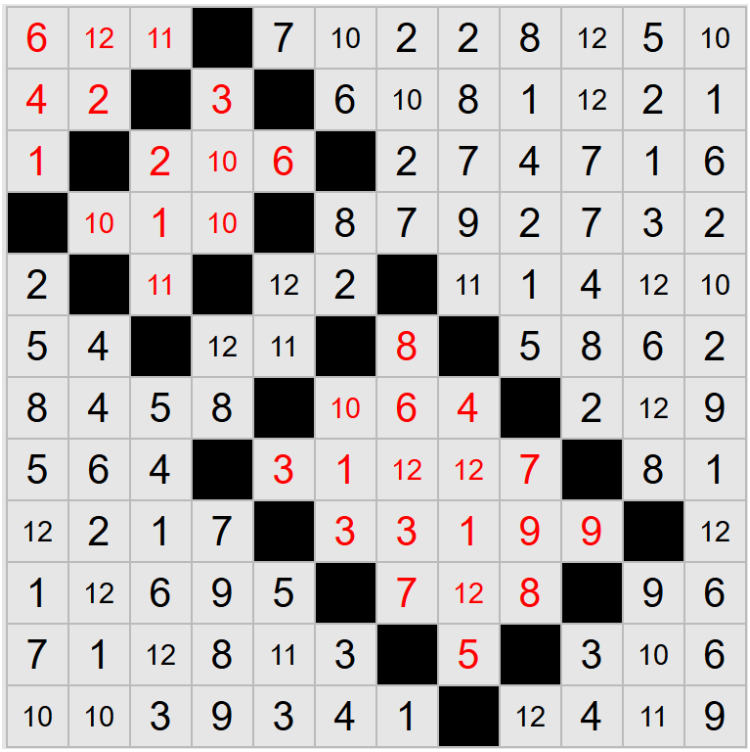

FIGURE 16 – Grands rectangles

3.1.4 Lignes brisées

De manière générale, toute ligne brisée séparant la grille doit être interdite. Certaines sont plus simples à coder que d'autres.

6	12	■	6	7	10	■	2	8	12	5	10
4	■	7	3	12	■	10	8	1	12	2	1
1	9	■	10	6	4	■	7	4	7	1	6
12	■	1	10	4	8	7	■	2	7	3	2
2	8	■	6	12	2	3	11	■	4	12	10
5	■	2	12	11	11	8	4	5	■	6	2
8	4	■	8	10	10	6	4	5	2	■	9
5	■	4	2	3	1	12	12	7	■	8	1
12	2	■	7	7	3	3	1	■	9	4	12
1	■	6	9	5	2	7	12	8	■	9	6
7	1	■	8	11	3	4	5	2	3	■	6
10	■	3	9	3	4	1	1	12	4	11	■

FIGURE 17 – Lignes brisées