

Institut polytechnique de paris  
École nationale supérieure de techniques avancées

---

# Rapport du projet RO203 : Singles

---

*Réalisé par :*

Mahdi Cheikhrouhou

Marwen Bahri

*2ème année Techniques Avancées*

*02 mai 2021*

# Table des matières

<b>1</b>	<b>Description du jeu</b>	<b>3</b>
<b>2</b>	<b>Raisonnement pour la résolution du problème</b>	<b>5</b>
2.1	Premier sous problème . . . . .	5
2.2	Deuxième sous problème . . . . .	5
2.3	Troisième sous problème . . . . .	6
<b>3</b>	<b>Programme linéaire</b>	<b>9</b>
3.1	Premier sous problème . . . . .	9
3.2	Deuxième sous problème . . . . .	10
3.3	Troisième sous problème . . . . .	10
<b>4</b>	<b>Résolution heuristique</b>	<b>13</b>
4.1	Méthode de résolution . . . . .	13
4.1.1	Backtracking with intelligent pruning . . . . .	13
4.1.2	Calcul du nombre de composantes connexes . . . . .	13
<b>5</b>	<b>Les fonctions principales du code :</b>	<b>14</b>
5.1	Fonctions d'entrée et sortie . . . . .	14
5.1.1	Fonctions d'entrée . . . . .	14
5.1.2	Fonctions de sorties . . . . .	15
5.2	Fonction de génération . . . . .	16
5.3	Fonction de Résolution . . . . .	17

# Table des figures

1.1	Exemple de grille donnée de problème . . . . .	3
1.2	Exmepole de grille résolu . . . . .	4
2.1	Mauvaise solution . . . . .	5
2.2	Bonne solution . . . . .	5
2.3	Exemple de grille . . . . .	6
2.4	Exemple de graphe . . . . .	7
2.5	Exemple de graphe avec sink . . . . .	8
3.1	Illustration des variables du flot . . . . .	12
5.1	Exemple de grille initiale dans un fichier txt . . . . .	14
5.2	Sortie de la fonction DisplayGrid . . . . .	15
5.3	Résultat de la fonction displaySolution . . . . .	15
5.4	Diagramme de performance du méthode simplex . . . . .	16

# Chapitre 1

## Description du jeu

Les règles de ce jeu consistent à masquer des cases de la grille de façon à ce qu'aucun chiffre ne soit visible plus d'une fois sur chaque ligne et chaque colonne, les cases masquées ne soient pas adjacentes et l'ensemble des cases visibles est connexes. La plupart de ces règles peuvent être facilement modélisées par des contraintes linéaires, la seule qui pose de difficulté est celle de la connexité mais on a pu surmonter cette difficulté en utilisant l'algorithme de flot maximale. Dans ce qui suit, on va décomposer le problème du jeu en sous-problème et on expliquera le principe d'utilisation de l'algorithme de flow maximale pour vérifier la connexité.

0

1

2

3

4

5

6

7

8

9


10

11



Singles







Game

Type

Help

5	2	2	1	5	8	7	4
2	2	7	5	4	6	6	5
4	6	4	3	2	1	5	2
5	5	1	7	2	1	8	2
3	6	6	7	5	2	1	7
8	7	4	6	4	5	2	7
7	5	2	6	1	6	3	5
3	3	5	2	5	7	4	4

FIGURE 1.1 – Exemple de grille donnée de problème



FIGURE 1.2 – Exmepile de grille résolu

## Chapitre 2

# Raisonnement pour la résolution du problème

Dans ce chapitre on va indiquer les idées. Dans le chapitre suivant on va faire la description des contraintes dans le programme linéaire.

### 2.1 Premier sous problème

Le premier sous-problème est d'assurer qu'un chiffre soit présent au plus une fois dans chaque ligne et chaque colonne.

### 2.2 Deuxième sous problème

Les cases masquées doivent être non adjacentes (elle peuvent néanmoins être placées en diagonale)



FIGURE 2.1 – Mauvaise solution

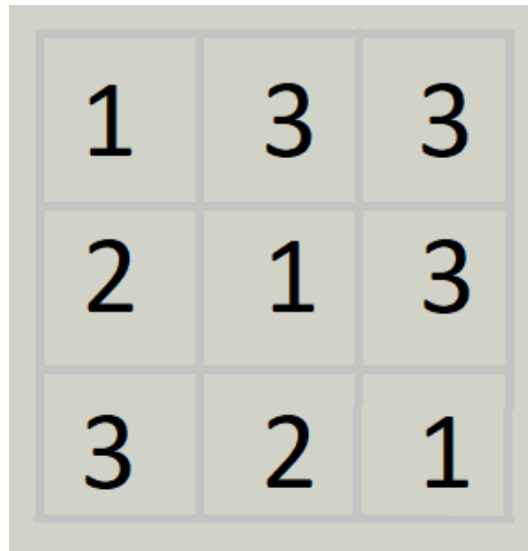


FIGURE 2.2 – Bonne solution

A noter que dans la figure 2.2 la solution ne vérifie pas la règle de connexité, c'est une solution pour ce sous problème uniquement.

## 2.3 Troisième sous problème

Le troisième et dernier sous problème est celui de la connexité. Pour assurer que les cases visibles forment un espace connexe, on commence par les visualiser comme étant des noeuds d'un graphe tel que les cases adjacentes sont liées par des arcs non orientés. Si on avait la possibilité d'utiliser un algorithme de parcours en profondeur ou en largeur il nous sera très facile à vérifier la connexité mais on peut pas modéliser ces algorithmes en programmes linéaires. Cependant, en appliquant l'algorithme de flot maximale sur notre graphe en lui apportant une petite modification et en spécifiant les capacités des arcs de manière précise on est capable de vérifier si le flot atteint tous les noeuds de notre graphe. Supposons que notre grille initiale est la suivante :



1	3	3
2	1	3
3	2	1

FIGURE 2.3 – Exemple de grille

Cette grille correspondra au graphe suivant :

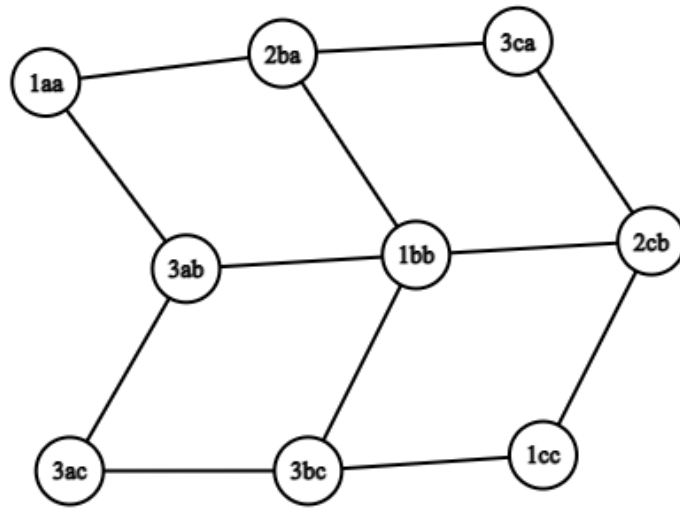


FIGURE 2.4 – Exemple de graphe

Les indices des lignes et des colonnes sont des lettres alphabétiques.  $1_{aa}$  désigne la case du premier ligne et la première colonne et sa valeur qui vaut 1 et ainsi de suite. Chaque arc non orienté est représenté par deux arcs orientés de capacité infinie chacun et on prend le noeud  $1_{aa}$  comme noeud source à titre d'exemple. Maintenant on ajoute un noeud superficiel qui sera notre sink et on relie chaque noeud du graphe à ce sink par un arc de capacité égale à 1 et on relie le sink au noeud source par un arc de capacité infinie, on se retrouve finalement avec ce graphe (on a pas représenté les noeuds de la dernière ligne pour des raisons de clarté) :



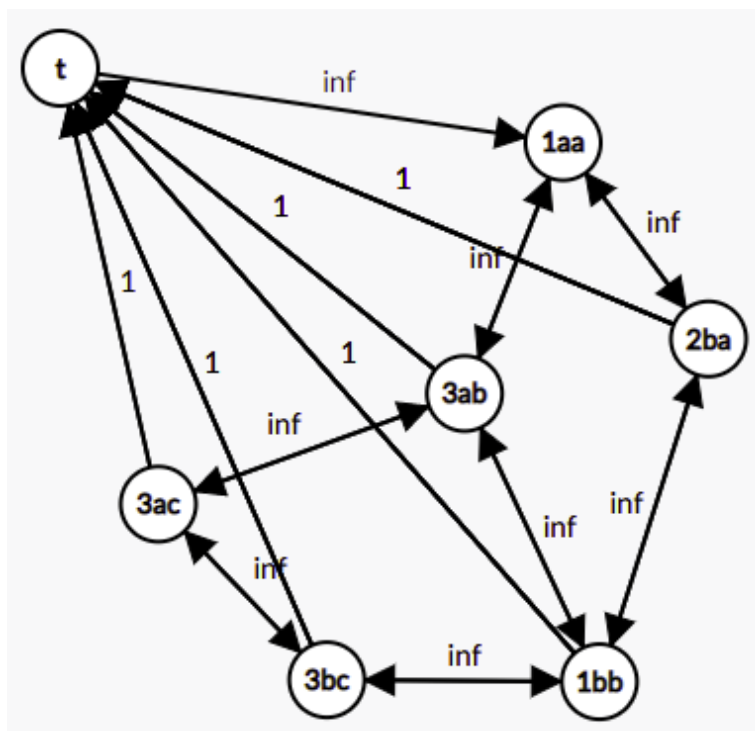


FIGURE 2.5 – Exemple de graphe avec sink

Finalement, il suffit d'appliquer l'algorithme sur ce graphe et vérifier si le flot est égale au nombre des noeuds de notre graphe  $MaxFlow == n$ . Si c'est le cas on sait alors qu'on peut atteindre chaque noeud du graphe à partir du noeud source (qui lui même peut être n'importe quel noeud du graphe puisque ce dernier est non orienté) et on a alors la connexité. En tenant compte du deuxième sous problème, on est sûr que deux cases adjacentes de la grille ne peuvent être toutes les deux masquées et donc nécessairement l'une d'entre elle appartient à notre graphe. Par conséquent, on appliquera l'algorithme de flot maximal en prenant comme noeud source la première case de notre grille puis la deuxième ( $G[1,1]$  et  $G[1,2]$ ) et si dans les deux cas on n'obtient pas un flot égale à  $n$  alors on a pas de connexité.

# Chapitre 3

## Programme linéaire

Dans cette partie on va exprimer les sous problèmes définis précédemment sous forme de contraintes linéaires.

### 3.1 Premier sous problème

On suppose dans la suite que les entiers présents dans la grille sont compris entre 1 et  $n$ , avec  $n$  est la dimension de la grille, et que celle-ci soient carrée. Il est possible de généraliser notre démarche pour des cas quelconques. Soient maintenant les variables binaires  $x_{ij} \in \{0, 1\}$ ,  $i, j \in \{1, \dots, n\}$  telque  $x_{ij} = 0$  si la case d'indices  $(i, j)$  est masquée et  $x_{ij} = 1$  si non. Pour assurer qu'un entier ne soit présent plus d'une fois dans un ligne (resp. colonne), on va itérer sur les lignes (resp. les colonnes) de la grille et pour chaque ligne (resp. colonne) on va itérer sur les entiers  $k$  avec  $k \in \{1, \dots, n\}$  et on prend la somme sur  $j$  des  $x_{ij}$  telque  $grille_{ij} = k$ . Pour assurer la contrainte de sous problème il suffit que chacune de ces sommes soit inférieur ou égale à 1.

*les contraintes sur les lignes :*

$$\forall i \in \{1, \dots, n\}, \forall k \in \{1, \dots, n\}$$

$$\sum_{j \in \gamma_k} (x_{ij}) \leq 1$$

$$\text{avec } \gamma_k = \{j, \forall j \in \{1, \dots, n\} \mid grille_{ij} = k\}$$

*les contraintes sur les colonnes :*

$$\forall j \in \{1, \dots, n\}, \forall k \in \{1, \dots, n\}$$

$$\sum_{i \in \gamma_k} (x_{ij}) \leq 1$$

$$\text{avec } \gamma_k = \{i, \forall i \in \{1, \dots, n\} \mid grille_{ij} = k\}$$

## 3.2 Deuxième sous problème

Les contraintes qui permet d'assurer que deux cases adjacentes ne soient à la fois toutes les deux masquées sont les suivantes :

$$\forall i \in \{1, \dots, n\} \quad x_{ij} + x_{ij+1} \geq 1 \quad \forall j \in \{1, \dots, n-1\}$$

$$\forall j \in \{1, \dots, n\} \quad x_{ij} + x_{i+1j} \geq 1 \quad \forall i \in \{1, \dots, n-1\}$$

$x_{ij}$  étant égale à 0 si la case (i,j) est masquée,  $x_{ij} + x_{ij+1} = 0$  implique que la contrainte de ce sous problème n'est pas vérifiée.

## 3.3 Troisième sous problème

C'est le sous problème le plus complexe dans le programme. Dans cette partie on va développer la partie flot à partir d'une source vers un puits imaginaire. Pour ce faire, on va d'abord définir quelques variables :

$$flowH_{i,j,k} \in \mathcal{N}; 1 \leq i \leq n, 1 \leq j \leq n-1, 1 \leq k \leq 2$$

$$flowV_{i,j,k} \in \mathcal{N}; 1 \leq i \leq n-1, 1 \leq j \leq n, 1 \leq k \leq 2$$

$$flowSink_{i,j} \in \{0, 1\}; 1 \leq i \leq n, 1 \leq j \leq n$$

$$phi \in \mathcal{N}$$

### Description des variables :

- $flowH_{i,j,1}$  correspond au flot qui passe par l'arc  $edgeH_{i,j}$  de gauche à droite c'est à dire : de  $(i, j)$  à  $(i, j+1)$
- $flowH_{i,j,2}$  correspond au flot qui passe par l'arc  $edgeH_{i,j}$  de droite à gauche c'est à dire : de  $(i, j)$  à  $(i, j-1)$
- $flowV_{i,j,1}$  correspond au flot qui passe par l'arc  $edgeV_{i,j}$  de haut vers bas c'est à dire : de  $(i, j)$  à  $(i+1, j)$
- $flowV_{i,j,2}$  correspond au flot qui passe par l'arc  $edgeV_{i,j}$  de bas vers haut c'est à dire : de  $(i, j)$  à  $(i-1, j)$
- $flowSink_{i,j}$  correspond au flot qui passe du noeud  $(i, j)$  vers le puits virtuelle. c'est une variable binaire car le flot maximum qu'on peut envoyer vers le puits égale à 1 depuis un seul noeud.
- $Phi$  correspond au flot dans le puits.

On ne va pas utiliser des variables pour représenter les capacités des arcs mais plutôt on va définir des contraintes en utilisant des variables déjà définies pour représenter les contraintes sur les capacités. Posons maintenant  $x_s, y_s$ , les coordonnées de la source. Notre objectif est de maximiser le flot, c'est-à-dire :

$$obj : \max(phi)$$

On donne les constraints liées au flot , les contraintes seront illustrées dans la figure suivante :

- **Flot entrant égale au flot sortant** : on va aborder seulement le cas général pour lequel le noeud n'est pas sur le bord. les cas particuliers des bords sont traités dans le l'implémentation.

Pour les cases autres que la source, Posons :

$$flotEntrant = flowV_{i-1,j,1} + flowV_{i,j,2} + flowH_{i,j-1,1} + flowH_{i,j,2}$$

$$flotSortant = flowV_{i-1,j,2} + flowV_{i,j,1} + flowH_{i,j-1,2} + flowH_{i,j,1} + flowSink_{i,j}$$

$$flotEntrant - flotSortant == 0$$

$$2 \leq i, j \leq n, (i, j) \neq (x_s, y_s)$$

Pour la source :

$$flotEntrant = phi$$

$$flotSortant = flowV_{x_s-1,y_s,2} + flowV_{x_s,y_s,1} + flowH_{x_s,y_s-1,2} + flowH_{x_s,y_s,1} + flowSink_{x_s,y_s}$$

$$flotEntrant - flotSortant == 0$$

Pour le puits :

$$phi - \sum_{1 \leq i,j \leq n} flowSink_{ij} = 0,$$

Pour assurer que chaque case non masquée transmet une quantité égale à 1 vers le puits.

- **Flot vers le puits** : on veut que le flot d'un noeud soit égale à 1 si la case est visible et 0 si elle est masquée.

$$flowSink_{ij} = x_{ij} \quad \forall i, j \in \{1, \dots, n\}$$

- **Condition sur les capacités** : il faut assurer deux choses, la première est que le flot entrant vers une case masquée et le flot sortant d'une telle case soient nulls. Pour assurer cette condition on a exploité le fait que les variables  $x_{ij}$  sont binaires et on a écrit les contraintes suivantes :

$$flowH_{ij,k} \leq 10000x_{ij}$$

$$flowV_{ij,k} \leq 10000x_{ij}$$

$$\forall i, j \in \{1, \dots, n\}$$

ces contraintes assurent que le flot vers une case masquée soit null et que si la case est visible alors il soit non majoré (la dimension de la grille n est très inférieure à  $10^4$ ).

- **Condition sur la connexité** : il faut que le flot total soit égale au nombre de cases non masquées

$$phi - \sum_{1 \leq i,j \leq n} x_{ij} = 0$$

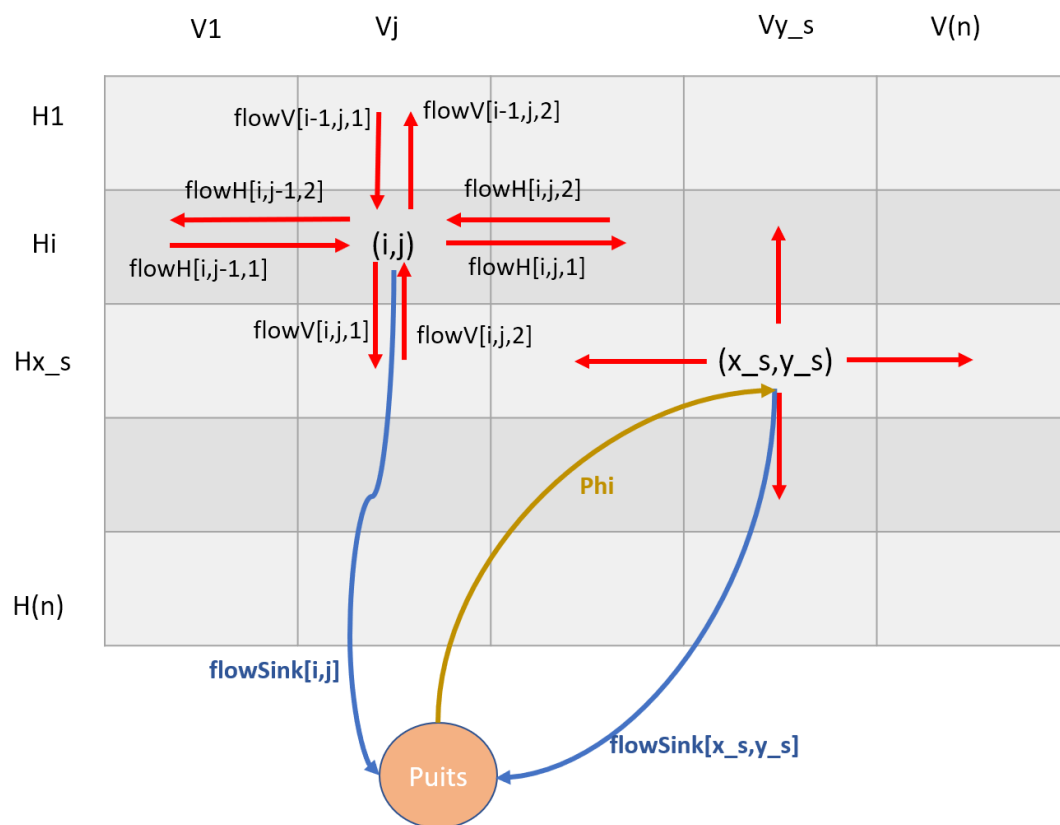


FIGURE 3.1 – Illustration des variables du flot

# Chapitre 4

## Résolution heuristique

### 4.1 Méthode de résolution

Pour résoudre ce problème on a utilisé un algorithme de *Brute Force* qui nous permet de générer tous les possibilités des masque d'une grille donnée et voir si l'une d'entre eux est optimals. On a optimisé l'agorithme en empêchant la recherche d'une solution à partir d'un état qu'on sait déjà qu'il non valide, la méthode qu'on a utilisé s'appelle *Backtracking With Intelligent Pruning*. De plus, pour encore optimiser l'algorithme on a implémenté l'algorithme de *Disjoint Set Union (DSU)* pour calculer le nombre de composantes connexes.

#### 4.1.1 Backtracking with intelligent pruning

à chaque fois qu'on est à l'état  $(x, y, mask)$  on vérifie s'il est valide ou non. Un état est valide s'il vérifie les conditions du jeu que nous avons abordé dans les précédents chapitres.

#### 4.1.2 Calcul du nombre de composantes connexes

Pour un état valide, le nombre de composante connexe doit être égale à 1 (l'ensemble des cases dont la valeur du masque est égale à 0). Pour le calcul du nombre des composantes connexes, on a utilisé l'algorithme de *Disjoint Set Union (DSU)* or *Union Find*.

**Description de l'algorithme :**

- Chaque noeud constitue un ensemble qui est représente par lui même
- On boucle sur tous les arcs  $(a, b)$  et on unissent l'ensemble qui contient  $a$  avec l'ensemble qui contient  $b$ .
- Le nombre de composantes connexes est le nombre des ensembles à la fin

# Chapitre 5

## Les fonctions principales du code :

### 5.1 Fonctions d'entrée et sortie

#### 5.1.1 Fonctions d'entrée

On a seulement une fonction d'entrée.

##### **readInputFile**

Cette fonction prend comme argument une chaîne de caractères qui représente le chemin du fichier où on a généré la grille initiale et retourne un tableau de 2 dimension dont les valeurs sont celles du fichier.

La grille initiale est mémorisé dans un fichier texte (terminaison .txt) dans le répertoire *data* tel est l'exemple de la figure suivante ( cette figure représente le jeu dans la figure 1.1 ).Les cases de la grille sont séparées par des virgules (",") et les cases vides sont représentées par le nombre 5 car c'est impossible de trouver 5 dans la grille et c'est afin de faciliter la récupération des données.

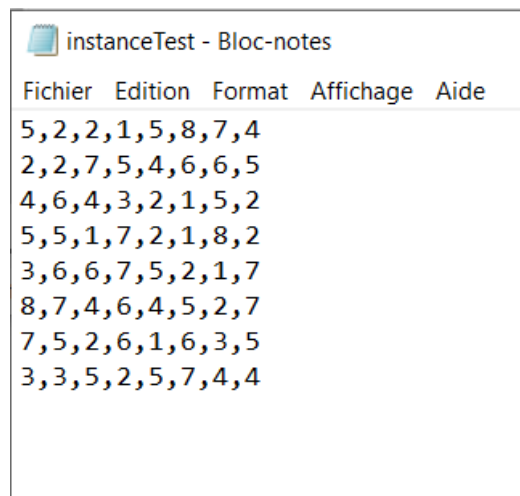


FIGURE 5.1 – Exemple de grille initiale dans un fichier txt

## 5.1.2 Fonctions de sorties

### displayGrid

Cette fonction à pour but d'afficher une grille non résolu dans le console en lui donnant une chaîne qui représente le chemin du fichier. L'affichage est donnée par la figure suivante.

```
julia> displayGrid("../data\\instanceTest.txt")
.-.-.-.-.-
|5|2|2|1|5|8|7|4|
.-.-.-.-.-
|2|2|7|5|4|6|6|5|
.-.-.-.-.-
|4|6|4|3|2|1|5|2|
.-.-.-.-.-
|5|5|1|7|2|1|8|2|
.-.-.-.-.-
|3|6|6|7|5|2|1|7|
.-.-.-.-.-
|8|7|4|6|4|5|2|7|
.-.-.-.-.-
|7|5|2|6|1|6|3|5|
.-.-.-.-.-
|3|3|5|2|5|7|4|4|
```

FIGURE 5.2 – Sortie de la fonction DisplayGrid

### displaySolution

Cette fonction a pour but d'afficher une grille résolue.

```
julia> displaySolution("../res/cplex/instanceTest.txt")
.-.-.-.-.-
|5|2|x|1|x|8|7|4|
.-.-.-.-.-
|2|x|7|5|4|x|6|x|
.-.-.-.-.-
|4|6|x|3|x|1|5|2|
.-.-.-.-.-
|x|5|1|7|2|x|8|x|
.-.-.-.-.-
|3|x|6|x|5|2|1|7|
.-.-.-.-.-
|8|7|4|6|x|5|2|x|
.-.-.-.-.-
|7|x|2|x|1|6|3|5|
.-.-.-.-.-
|x|3|5|2|x|7|4|x|
.-.-.-.-.-
```

FIGURE 5.3 – Résultat de la fonction displaySolution



## performanceDiagram

Le diagramme de performance du méthode simplex est le suivant :

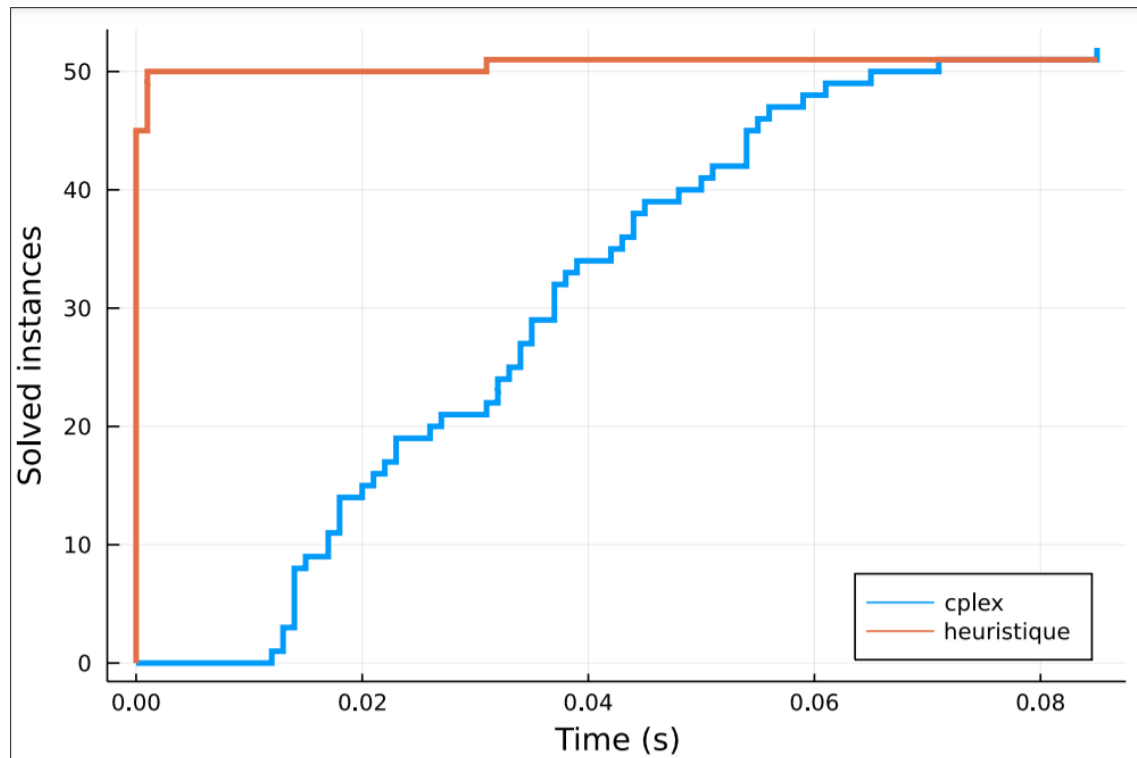


FIGURE 5.4 – Diagramme de performance du méthode simplex

## resultsArray

Cette fonction génère un fichier latex qui contient les informations sur les instances sur lesquelles on a testé les algorithmes cplex et heuristique.

## 5.2 Fonction de génération

### generateDataSet

Cette fonction prend en arguments le nombre des instances qu'on veut générer et fait appel à la fonction **generateInstance** en la passant une dimension pour la grille tirée au hasard entre 3 et 15 et une densité entre 0.2 et 0.4.

### generateInstance

Cette fonction prend en arguments la dimension et la densité de l'instance à générer. Pour générer l'instance, on a pensé à procéder comme suit :

- on commence par déterminer le nombre de cases visibles que la grille finale doit avoir en utilisant les valeurs passées en paramètres de la fonction.
- on tire au hasard une position à partir duquel on commence le parcours de la grille.

- à chaque fois qu'on est sur une case, on augmente le nombre de cases atteintes par 1 puis on cherche le premier nombre qui n'est pas présent sur la ligne et la colonne correspondantes et on affecte cette case par ce nombre. Si on ne trouve pas un nombre disponible, on affecte cette case par 0.
- on tire au hasard un nombre aléatoire des voisins de cette case qu'on va garder visibles et on les ajoute dans un tableau qui contient les prochaines cases possibles pour continuer le parcours.
- tant qu'on a pas atteint le nombre de cases cherché, on tire au hasard une nouvelle case à partir du tableau et on répète ces étapes.

Une fois qu'on a sorti du boucle, on se retrouve par une instance qui contient des nombre entre 0 et  $n$  avec  $n$  est la dimension de la grille. L'espace correspondant aux cases dont les valeurs sont différentes de 0 est connexe. Maintenant, on fait un parcours de la grille générée et à chaque fois qu'on trouve une case affectée par zéro on l'affecte par un nombre aléatoire entre 1 et  $n$ .

## 5.3 Fonction de Résolution

### **cplexSolve**

Cette fonction prend un tableau 2D comme argument est applique tout le raisonnement qu'on a présenté au début du rapport et retourne une grille résolue.

### **heuristicSolve**

Cette fonction reçoit en paramètres une grille 2D et retourne un masque de la grille. Voici la description des variables :

- *mask* : tableau 2D binaire telque  $mask_{ij} = 0$  si la case (i,j) est visible et égale à 1 si non.
- *hidden* : tableau 2D binaire telque  $hidden_{ij} = 0$  si l'entier présent à la case (i,j) est unique sur la ligne i et sur la colonne j. Dans ce cas, il est inutile de masquer cette case (ça sert à réduire le nombre des cas testés).

### **solveDataSet**

Cette fonction permet de résoudre toutes les instances dans le répertoire *data* avec plusieurs méthodes possibles (cplex et heuristique) et mémorise les résultat obtenus dans des sous répertoires du répertoire *res* comme représenté dans la partie displaySolution du rapport.

## Results Array

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest110.txt	0.02	×	0.0	×
instanceTest12.txt	0.04	×	0.0	×
instanceTest161.txt	0.04	×	0.0	×
instanceTest196.txt	0.03	×	0.0	×
instanceTest212.txt	0.03	×	0.0	×
instanceTest224.txt	0.02	×	0.0	×
instanceTest236.txt	0.03	×	0.0	×
instanceTest244.txt	0.01	×	0.0	×
instanceTest25.txt	0.03	×	0.0	×
instanceTest251.txt	0.02	×	0.0	×
instanceTest281.txt	0.02	×	0.0	×
instanceTest291.txt	0.06	×	0.0	×
instanceTest307.txt	0.04	×	0.0	×
instanceTest321.txt	0.03	×	0.0	×
instanceTest326.txt	0.02	×	0.0	×
instanceTest355.txt	0.05	×	0.0	×
instanceTest372.txt	0.02	×	0.0	×
instanceTest381.txt	0.02	×	0.0	×
instanceTest388.txt	0.01	×	0.0	×
instanceTest410.txt	0.04	×	0.0	×
instanceTest415.txt	0.01	×	0.0	×
instanceTest417.txt	0.02	×	0.0	×
instanceTest432.txt	0.02	×	0.0	×
instanceTest458.txt	0.02	×	0.0	×
instanceTest476.txt	0.02	×	0.0	×
instanceTest500.txt	0.02	×	0.0	×
instanceTest506.txt	0.04	×	0.0	×
instanceTest512.txt	0.02	×	0.0	×
instanceTest534.txt	0.04	×	0.0	×

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest566.txt	0.03	×	0.0	×
instanceTest582.txt	0.03	×	0.0	×
instanceTest636.txt	0.04	×	0.0	×
instanceTest639.txt	0.08	×	0.0	×
instanceTest644.txt	0.02	×	0.0	×
instanceTest653.txt	0.02	×	0.0	×
instanceTest763.txt	0.02	×	0.0	×
instanceTest777.txt	0.02	×	0.0	×
instanceTest779.txt	0.02	×	0.0	×
instanceTest790.txt	0.01	×	0.0	×
instanceTest800.txt	0.02	×	0.0	×
instanceTest842.txt	0.03	×	0.0	×
instanceTest847.txt	0.02	×	0.0	×
instanceTest848.txt	0.03	×	0.0	×
instanceTest863.txt	0.02	×	0.0	×
instanceTest871.txt	0.04	×	0.0	×
instanceTest912.txt	0.04	×	0.0	×
instanceTest914.txt	0.08	×	0.0	×
instanceTest953.txt	0.03	×	0.0	×
instanceTest959.txt	0.03	×	0.0	×
instanceTest984.txt	0.03	×	0.0	×
instanceTest989.txt	0.02	×	0.0	×
instanceTest994.txt	0.05	×	0.0	×
instanceTest110.txt	0.02	×	0.0	×
instanceTest12.txt	0.04	×	0.0	×
instanceTest161.txt	0.04	×	0.0	×
instanceTest196.txt	0.03	×	0.0	×
instanceTest212.txt	0.03	×	0.0	×
instanceTest224.txt	0.02	×	0.0	×
instanceTest236.txt	0.03	×	0.0	×

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest244.txt	0.01	×	0.0	×
instanceTest25.txt	0.03	×	0.0	×
instanceTest251.txt	0.02	×	0.0	×
instanceTest281.txt	0.02	×	0.0	×
instanceTest291.txt	0.06	×	0.0	×
instanceTest307.txt	0.04	×	0.0	×
instanceTest321.txt	0.03	×	0.0	×
instanceTest326.txt	0.02	×	0.0	×
instanceTest355.txt	0.05	×	0.0	×
instanceTest372.txt	0.02	×	0.0	×
instanceTest381.txt	0.02	×	0.0	×
instanceTest388.txt	0.01	×	0.0	×
instanceTest410.txt	0.04	×	0.0	×
instanceTest415.txt	0.01	×	0.0	×
instanceTest417.txt	0.02	×	0.0	×
instanceTest432.txt	0.02	×	0.0	×
instanceTest458.txt	0.02	×	0.0	×
instanceTest476.txt	0.02	×	0.0	×
instanceTest500.txt	0.02	×	0.0	×
instanceTest506.txt	0.04	×	0.0	×
instanceTest512.txt	0.02	×	0.0	×
instanceTest534.txt	0.04	×	0.0	×
instanceTest566.txt	0.03	×	0.0	×
instanceTest582.txt	0.03	×	0.0	×
instanceTest636.txt	0.04	×	0.0	×
instanceTest639.txt	0.08	×	0.0	×
instanceTest644.txt	0.02	×	0.0	×
instanceTest653.txt	0.02	×	0.0	×
instanceTest763.txt	0.02	×	0.0	×
instanceTest777.txt	0.02	×	0.0	×

Instance	cplex		heuristique	
	Temps (s)	Optimal ?	Temps (s)	Optimal ?
instanceTest779.txt	0.02	×	0.0	×
instanceTest790.txt	0.01	×	0.0	×
instanceTest800.txt	0.02	×	0.0	×
instanceTest842.txt	0.03	×	0.0	×
instanceTest847.txt	0.02	×	0.0	×
instanceTest848.txt	0.03	×	0.0	×
instanceTest863.txt	0.02	×	0.0	×
instanceTest871.txt	0.04	×	0.0	×
instanceTest912.txt	0.04	×	0.0	×
instanceTest914.txt	0.08	×	0.0	×
instanceTest953.txt	0.03	×	0.0	×
instanceTest959.txt	0.03	×	0.0	×
instanceTest984.txt	0.03	×	0.0	×
instanceTest989.txt	0.02	×	0.0	×
instanceTest994.txt	0.05	×	0.0	×