

RO203-Resolution des jeux Towers et Singles

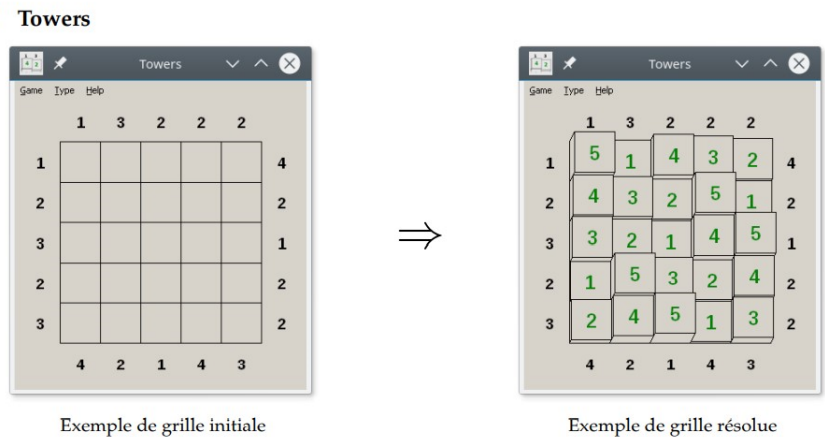
Justine De Sousa et Nina Gregorio

Table des matières

1	Towers	1
1.1	Modélisation	1
1.2	Génération d'instance	2
1.3	Heuristique de résolution	2
1.4	Programme linéaire de résolution	3
1.5	résultats	5
2	Singles	8
2.1	modélisation Singles	8
2.2	génération Singles	8
2.3	Heuristique Singles	9
2.4	Cplex Singles	9
2.5	résultats	10

1 Towers

Towers est un jeu constitué d'une grille de taille n fois n initialement vide. On remplit cette grille de tours, qui doivent être de tailles différentes sur les lignes et les colonnes. En début et bout de chaque est indiqué le nombre de tours visibles.



1.1 Modélisation

Nous avons donc assez intuitivement fait le choix de travailler avec une matrice grille d'entiers de taille n fois n à remplir, et de 4 vecteurs, nord, sud, ouest et est qui contiennent le nombre de tours visibles depuis ces directions. En pratique nos grilles s'affichent comme suit (éventuellement d'intérieur vide)

		1	2	2		

1		3	1	2		2
2		2	3	1		2
3		1	2	3		1

		3	2	1		

1.2 Génération d'instance

Nous avons d'abord rempli l'intérieur de la grille. On place au fur et à mesure des chiffres aléatoires sur les lignes et les colonnes. La génération aléatoire se fait sur les chiffres encore non présents sur les lignes ou les colonnes. Si l'on est bloqué on remplit autrement la dernière case remplie. Si l'on est bloqué à nouveau on recommence tout.

Exemple :

1	3	2	4
3	2		

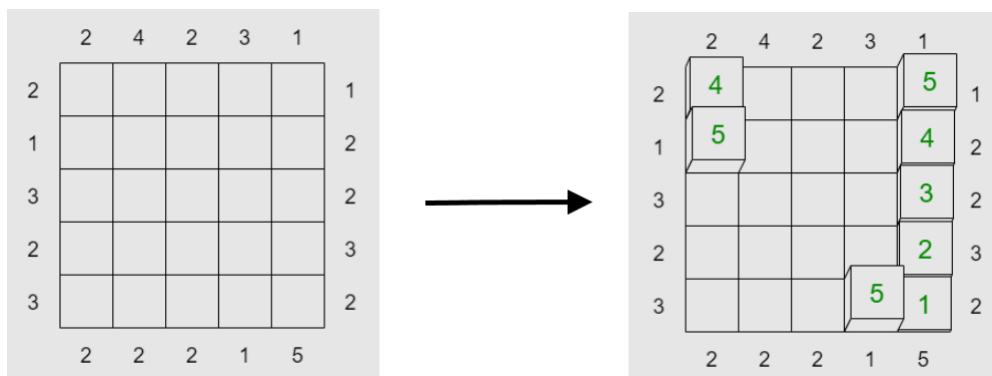
A ce stade de l'exemple on cherche à remplir la case (2,3). on choisit aléatoirement dans la liste [4,1]. Si on choisit 1, il sera impossible de finir de remplir la ligne. On retournera en arrière, choisissant cette fois dans la liste [4].

Une fois la grille remplie on remplit aisément les vecteurs nord, sud, ouest et est.

1.3 Heuristique de résolution

Nous avons d'abord rempli certaines cases faciles à déduire : sur les lignes ou colonnes comportant des n toutes les tours sont placées en ordre croissant. Sur les lignes ou les colonnes comportant des 1, la première case comporte nécessairement un n. Dans ce cas là, si le bord d'en face contient un 2, la première case en face contiendra n-1.

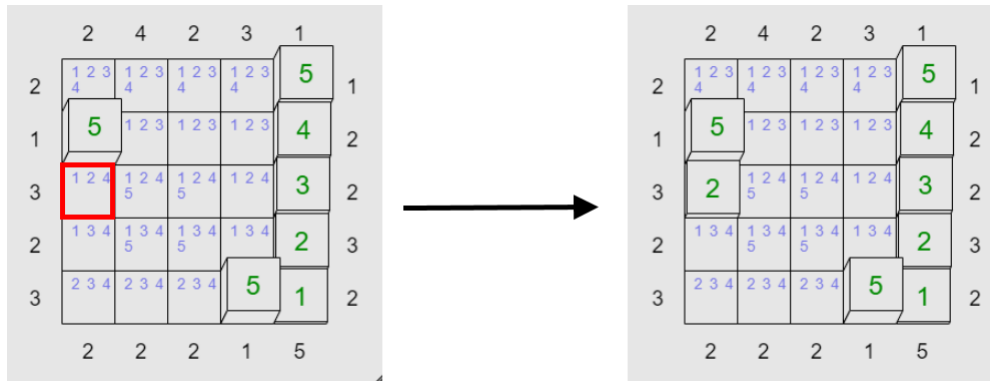
Exemple :



Puis nous avons petit à petit rempli le reste aléatoirement en procédant de la façon suivante :

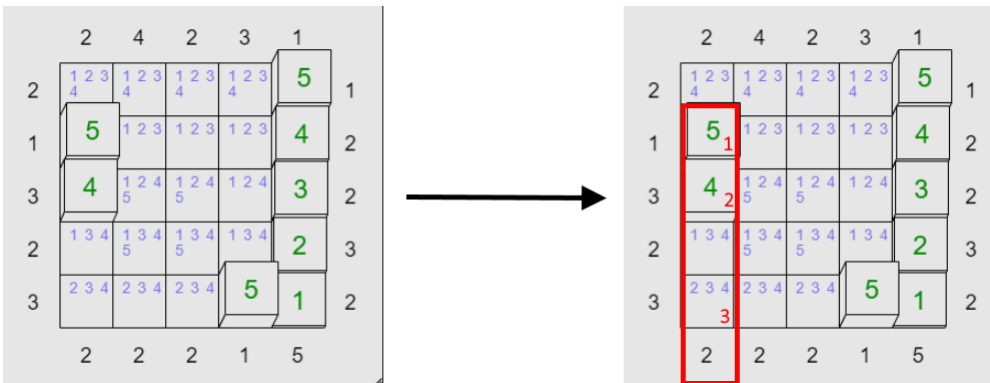
On parcourt toute la grille et on répertorie pour chaque case l'ensemble des valeurs possibles qu'elle peut contenir. On choisit ensuite l'une des cases qui a une plus grande contrainte, c'est-à-dire qui a le moins de valeurs possibles. On va ensuite tester une à une les valeurs possibles.

Exemple :



Si l'ajout d'une des valeurs crée un conflit dans la grille, on retire cette valeur des "valeurs possibles" pour cette case et on teste la prochaine valeur jusqu'à en obtenir une qui matche.

Exemple (de conflit) :



Lorsqu'on a trouvé une valeur, on passe alors à la case suivante dans la grille et on procède de la même façon. Si aucune valeur ne matche, alors on revient en arrière et on remet à zéro la dernière case ayant été modifiée. L'idée de base étant de revenir en arrière et de mémoriser les configurations qui ne sont pas possibles pour ne pas les tester à nouveau mais nous n'avons pas aboutit et en l'état, l'algorithme recommence à partir de la dernière case modifiée sans avoir retenu ses erreurs. C'est pourquoi il y a peu de chance que cela aboutisse à une solution correcte sauf sur de très petites grilles.

1.4 Programme linéaire de résolution

Pour la résolution cplex nous avons utilisé les variables suivantes :

- $\forall i, j \in \{1, \dots, n\}$ yn_{ij} , ys_{ij} , yo_{ij} et ye_{ij} qui valent 1 si la tour à la position (i, j) est visible respectivement depuis le nord, le sud, l'ouest et l'est, 0 sinon
- $\forall i, j, k \in \{1, \dots, n\}$ $x_{ijk} = 1$ si la case à la position (i, j) contient k , 0 sinon

Les premières contraintes sont similaires à celles du sudoku. Elles codent les règles (parfois implicites) suivantes :

- pas de doublons sur les lignes
- pas de doublons sur les colonnes
- un et un seul chiffre par case

```

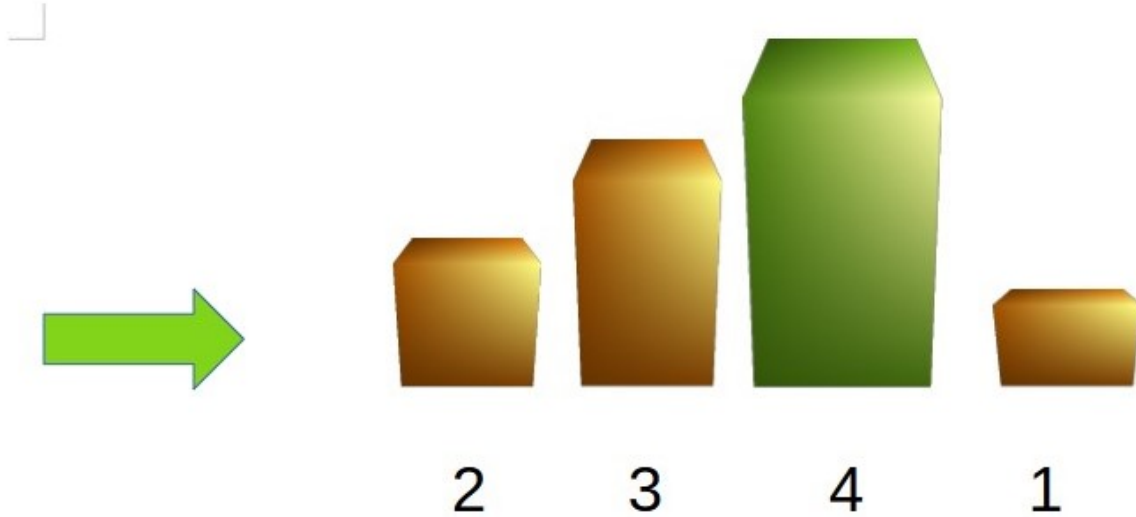
@variable (m,x[1:n,1:n,1:n],Bin) # ==1 ssi (i,j) contient k
@variable (m,yn[1:n,1:n],Bin)   # ==1 ssi (i,j) visible depuis le nord
@variable (m,ys[1:n,1:n],Bin)   # ==1 ssi (i,j) visible depuis le sud
@variable (m,yo[1:n,1:n],Bin)   # ==1 ssi (i,j) visible depuis l'ouest
@variable (m,ye[1:n,1:n],Bin)   # ==1 ssi (i,j) visible depuis l'est

#Une seule valeur par case
@constraint (m, [i in 1:n, j in 1:n], sum(x[i,j,k] for k in 1:n) == 1)
#Chiffres différents sur une ligne
@constraint (m, [i in 1:n, k in 1:n], sum(x[i,j,k] for j in 1:n) == 1)
#Chiffres différents sur une colonne
@constraint (m, [j in 1:n, k in 1:n], sum(x[i,j,k] for i in 1:n) == 1)

```

On s'intéresse maintenant à la contrainte plus difficile qui impose le bon nombre de tours visibles selon les directions nord, sud, ouest et est.

On considère d'abord la direction ouest. On procédera de même pour les trois autres directions.



On considère la somme suivante :

$$\sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-] = \begin{cases} 0 & \text{si la tour en } (i, j) \text{ de taille } k \text{ est visible depuis l'ouest} \\ 1 & \text{sinon} \end{cases}$$

En effet, il n'y a pas de tour plus grande, ie de taille $k_- > k$ aux positions $j_- < j$. Si la tour n'est pas visible cette somme est strictement positive.

On souhaite se ramener à une variable $y = 1$ ssi la tour est visible.

Or, on a toujours

$$0 \leq \frac{\sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-]}{2n} < 1$$

On considère ainsi d'abord

$$1 - \frac{\sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-]}{2n} = 1 \Leftrightarrow \frac{\sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-]}{2n} = 0 \Leftrightarrow \text{la tour est visible}$$

On en déduit alors la contrainte 2 ci dessous qui impose

$$yo[i, j] = 0 \text{ si } \sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-] > 0$$

```
#Ouest
@constraint(m, [i in 1:n], sum(yo[i,j] for j in 1:n)==ouest[i])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yo[i,j]<=1-sum(x[i,j_-,k_-] for j_- in 1:j-1 for k_- in k:n)/(2*n)+1-x[i,j,k])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yo[i,j]>=1-sum(x[i,j_-,k_-] for j_- in 1:j-1 for k_- in k:n)-2*n*(1-x[i,j,k]))
```

Par un raisonnement similaire la 3ème contrainte ci dessus impose

$$y[i, j] = 1 \text{ si } \sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-] = 0$$

On a donc bien grâce à ces deux contraintes

$$yo[i, j] = 1 \text{ ssi } \sum_{j_- < j} \sum_{k_- > k} x[i, j_-, k_-] = 0$$

NB : la fin de la contrainte est présente pour ne pas imposer $y[i, j]$ dans les cas où $x[i, j, k] = 0$, ie où il n'y a pas de tour de taille k en (i, j) .

De même, on obtient les contraintes pour le nord, le sud et l'est. Ce qui donne :

```
#Nord
@constraint(m, [j in 1:n], sum(yn[i,j] for i in 1:n)==nord[j])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yn[i,j]<=1-sum(x[i_-,j,k_-] for i_- in 1:i-1 for k_- in k:n)/(2*n)+1-x[i,j,k])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yn[i,j]>=1-sum(x[i_-,j,k_-] for i_- in 1:i-1 for k_- in k:n)-2*n*(1-x[i,j,k]))

#Sud
@constraint(m, [j in 1:n], sum(ys[i,j] for i in 1:n)==sud[j])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], ys[i,j]<=1-sum(x[i,j_-,k_-] for i_- in i+1:n for k_- in k:n)/(2*n)+1-x[i,j,k])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], ys[i,j]>=1-sum(x[i,j_-,k_-] for i_- in i+1:n for k_- in k:n)-2*n*(1-x[i,j,k]))

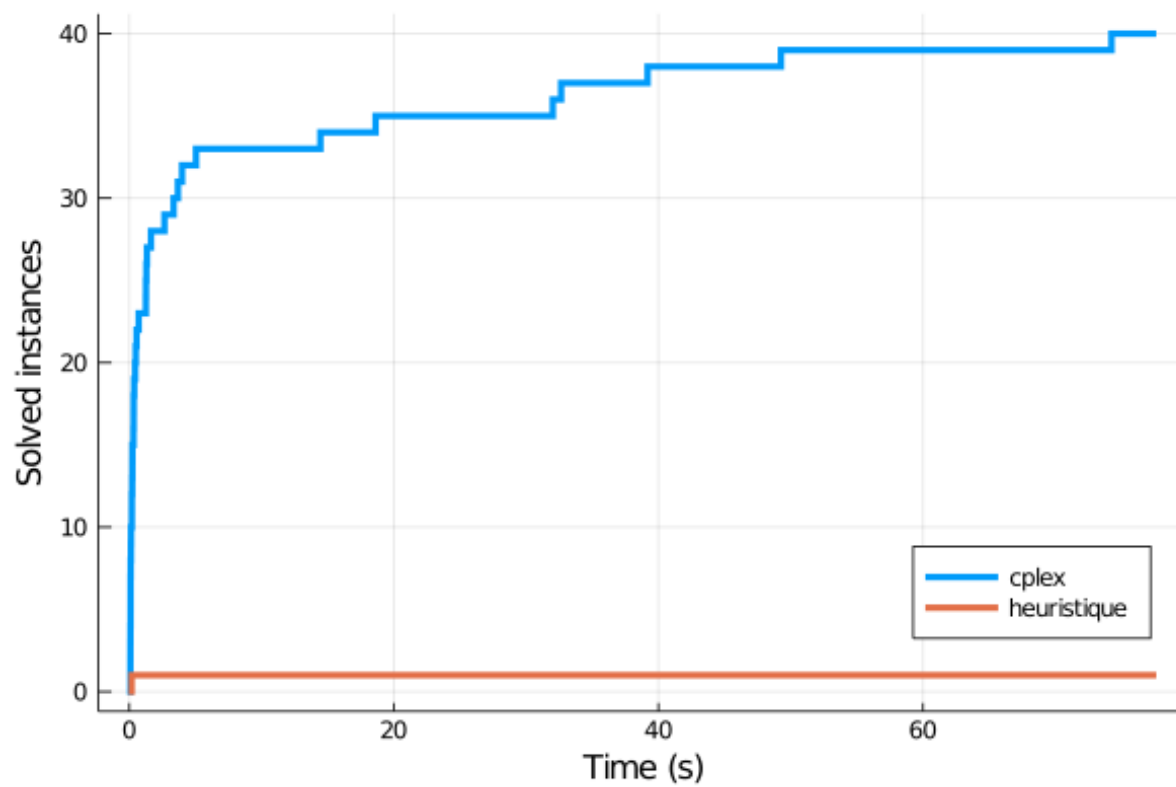
#Est
@constraint(m, [i in 1:n], sum(ye[i,j] for j in 1:n)==est[i])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], ye[i,j]<=1-sum(x[i,j_-,k_-] for j_- in j+1:n for k_- in k:n)/(2*n)+1-x[i,j,k])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], ye[i,j]>=1-sum(x[i,j_-,k_-] for j_- in j+1:n for k_- in k:n)-2*n*(1-x[i,j,k]))

#Ouest
@constraint(m, [i in 1:n], sum(yo[i,j] for j in 1:n)==ouest[i])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yo[i,j]<=1-sum(x[i,j_-,k_-] for j_- in 1:j-1 for k_- in k:n)/(2*n)+1-x[i,j,k])
@constraint(m, [i in 1:n, j in 1:n, k in 1:n], yo[i,j]>=1-sum(x[i,j_-,k_-] for j_- in 1:j-1 for k_- in k:n)-2*n*(1-x[i,j,k]))
```

L'ensemble de ces contraintes permettent d'imposer que le nombres de tours visibles correspond aux indications sur les bords. Ces indications sont stockées dans les vecteurs nord, est , sud et ouest.

1.5 résultats

On a généré des instances de taille 5,6,7,8. Elles sont toutes résolues par cplex en environ 1s sauf pour les instance de taille 8 où il faut quelques dizaines de secondes. L'heuristique quant à elle ne parvient qu'à résoudre une instance de taille 3, au delà cela devient très peu probable. La raison a été mentionnée dans la section 1.3



Instance	cplex		heuristique	
	Temps (s)	Optimal?	Temps (s)	Optimal?
instance_t3_1.txt	0.03	×	0.0	×
instance_t5_1.txt	0.11	×	16.8	
instance_t5_10.txt	0.11	×	15.47	
instance_t5_2.txt	0.09	×	17.65	
instance_t5_3.txt	0.08	×	18.24	
instance_t5_4.txt	0.1	×	16.36	
instance_t5_5.txt	0.08	×	14.97	
instance_t5_6.txt	0.07	×	10.77	
instance_t5_7.txt	0.06	×	13.99	
instance_t5_8.txt	0.09	×	5.24	×
instance_t5_9.txt	0.08	×	0.0	×
instance_t6_1.txt	0.19	×	16.04	
instance_t6_10.txt	0.16	×	10.0	
instance_t6_2.txt	0.26	×	12.7	
instance_t6_3.txt	0.17	×	10.01	
instance_t6_4.txt	0.2	×	10.0	
instance_t6_5.txt	0.83	×	12.36	
instance_t6_6.txt	0.18	×	10.0	
instance_t6_7.txt	0.28	×	21.3	
instance_t6_8.txt	0.17	×	10.0	
instance_t6_9.txt	0.24	×	11.07	
instance_t7_1.txt	1.14	×	10.02	
instance_t7_10.txt	0.62	×	10.0	
instance_t7_2.txt	1.26	×	10.01	
instance_t7_3.txt	4.08	×	10.01	
instance_t7_4.txt	0.42	×	10.01	
instance_t7_5.txt	0.53	×	10.0	
instance_t7_6.txt	0.8	×	10.0	
instance_t7_7.txt	1.92	×	18.33	
instance_t7_8.txt	0.94	×	10.0	
instance_t7_9.txt	0.6	×	10.01	
instance_t8_1.txt	59.01	×	14.32	
instance_t8_10.txt	20.68	×	18.76	
instance_t8_2.txt	24.25	×	10.0	
instance_t8_3.txt	103.69	×	24.73	
instance_t8_4.txt	21.28	×	17.09	
instance_t8_5.txt	3.88	×	10.01	
instance_t8_6.txt	67.91	×	10.0	
instance_t8_7.txt	2.63	×	10.01	
instance_t8_8.txt	9.38	×	28.45	
instance_t8_9.txt	3.58	×	26.28	

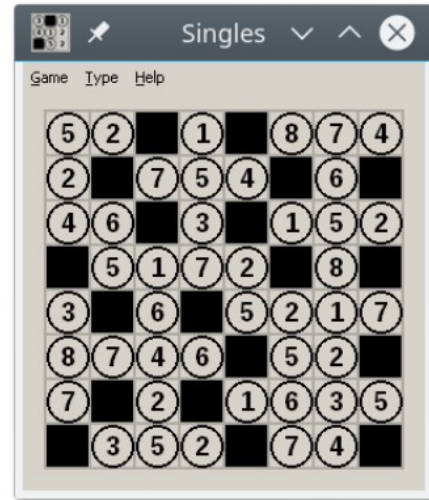
2 Singles

3 Singles



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

Exemple de grille initiale

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

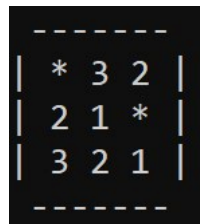
Exemple de grille résolue

Ce jeu est constitué d'une grille de chiffres. La résolution du jeu consiste à noircir certaines cases tel que :

- Il n'y ait pas deux cases noires qui se touchent
- Les cases blanches soient convexes
- Une fois certaines cases noircies, il n'y ait plus deux chiffres identiques sur les lignes et les colonnes

2.1 modélisation Singles

Le jeu sera modélisé par une matrice grille de taille $n \times n$ contenant 0 si la case est noircie (à cet endroit là une étoile sera affichée), et le chiffre contenu dans la case (i,j) sinon.



*	3	2
2	1	*
3	2	1

2.2 génération Singles

La génération de la grille suit les étapes ci-dessous : On place un maximum de cases noires non voisines tel que la grille reste connexe (ie de zéros). Pour cela on a réalisé entre autre deux fonctions annexes qui serviront beaucoup dans la suite.

Une fonction `isConnexe` qui indique si les cases blanches sont connexes. Elle considère que la grille est un graphe dont les noeuds sont les cases blanches, reliées par une arêtes si elles sont voisines. La fonction parcourt le graphe en profondeur en partant d'un sommet tiré au hasard et compte si la taille du graphe qu'il est ainsi possible de parcourir est égale au nombre de cases blanches de la grille. Nous avons également codé pour ce faire une fonction qui liste les cases blanches candidates à être blanches, ie qui n'ont pour l'instant pas de voisines blanches. Ensuite on remplit les chiffres des cases blanches tel qu'il n'y ait aucun doublon sur les lignes et sur les colonnes comme ce qui a été fait dans towers. Finalement on remplit au hasard les cases noires avec des entiers de 1 à n.

2.3 Heuristique Singles

L'heuristique suit les étapes suivantes :

On liste les doublons présents dans les lignes et les colonnes.

Par exemple :

2	3	2
2	1	1
3	2	1

Doublons=[[(1,1),(1,3)] [(2,2),(2,3)] [(1,1),(2,1)] [(2,3),(3,3)]]

On noircit dans ces listes de doublons autant de cases que nécessaire pour qu'il n'y ait plus de doublons, ie que chaque sous liste ne contienne qu'un élément. Pour cela il est nécessaire de s'assurer que les cases que l'on noircit sont admissibles, ie qu'elles ne sont pas voisines d'autres cases noires. Ici l'algorithme peut être bloqué. auquel cas on efface tout et on recommence de zéro.

Tant que c'est connexe et qu'il reste des cases admissibles et des doublons, on continue, sinon on recommence. par exemple :

2	3	2
2	*	1
3	2	*

2	3	*
*	1	1
3	2	*

*	3	2
2	1	*
3	2	1

L'heuristique a généré les deux premières grilles non valides, puis finalement en a généré une dont les cases blanches sont bien connexes.

Il aurait été possible de générer un algorithme plus intelligent qui mémorise les solution essayées, ou qui ne revient qu'un peu en arrière par exemple.

2.4 Cplex Singles

La cplex traite une matrice booléenne y de taille $n*n$ contenant 1 ssi la case est blanche (0 si elle est noire). On impose de façon similaire au sudoku qu'il n'y ait pas de doublons sur les lignes et les colonnes dont les cases sont blanches. De plus, on impose qu'il n'y ait pas de cases noires qui se touchent. Pour cela on s'assure que la somme de deux cases voisines est toujours strictement supérieure à 1, ie qu'il y a au moins une case blanche sur les deux.

```
@variable(singles,y[1:n,1:n],Bin)    # == 1 ssi (i,j) blanche
@objective(singles,Max,y[1,1])

#Chiffres différents sur une ligne, zero mis à part
@constraint(singles, lin[k in 1:n, i in 1:n], sum(y[i,j] for j in 1:n if x[i,j] == k) <= 1 )
#Chiffres différents sur une colonne, zero mis à part
@constraint(singles, col[k in 1:n, j in 1:n], sum(y[i,j] for i in 1:n if x[i,j]==k) <= 1)

#pas deux cases voisines noires
@constraint(singles, black[i in 1:n-1, j in 1:n], y[i,j]+y[i+1,j] >= 1)
@constraint(singles, black_[j in 1:n-1, i in 1:n], y[i,j]+y[i,j+1] >= 1)
```

Puis on s'intéresse à la contrainte de connexité des cases blanches. Pour cela, on mémorise les solutions trouvées par cplex et tant que la solution trouvée n'est pas connexe, on la mémorise et on l'interdit dans la suite. On crée ainsi au fur et à mesure une liste de solutions interdites et on s'assure que la solution nouvellement générée par cplex n'est pas une de celles-ci grâce à la contrainte suivante :

```

y_m = JuMP.value.(y)

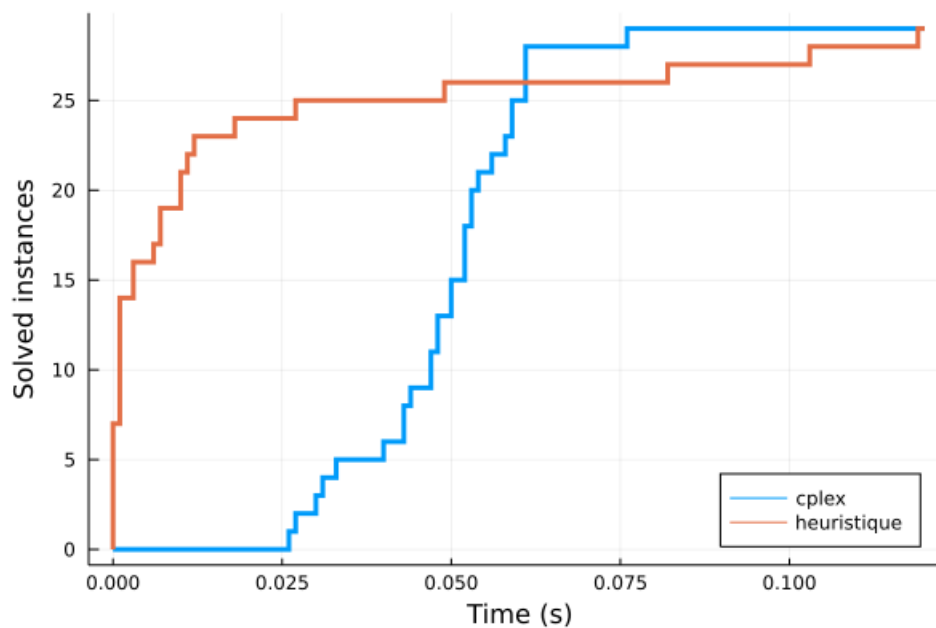
#contraintes de connexité
@constraint(singles, connexite, sum( y[i,j] for i in 1:n for j in 1:n if y_m[i,j] == 0 ) >= 1)

```

NB : il suffit de s'assurer que les cases noires sont différentes et c'est stricto-sensu ce que code cette contrainte.

2.5 résultats

Voici les résultats du jeu singles pour des instances de tailles 5 à 7. On a lancé avec 10 instances de chaque. La résolution dure moins d'une minute.



Instance	cplex		heuristique	
	Temps (s)	Optimal?	Temps (s)	Optimal?
instance_t5_1.txt	0.22	×	0.15	×
instance_t5_10.txt	0.06	×	0.01	×
instance_t5_2.txt	0.08	×	0.0	×
instance_t5_3.txt	0.09	×	0.0	×
instance_t5_4.txt	0.02	×	0.01	×
instance_t5_5.txt	0.02	×	0.0	×
instance_t5_6.txt	0.07	×	0.0	×
instance_t5_7.txt	0.02	×	0.0	×
instance_t5_8.txt	0.02	×	0.0	×
instance_t5_9.txt	0.03	×	0.0	×
instance_t6_1.txt	0.03	×	0.0	×
instance_t6_10.txt	0.03	×	0.0	×
instance_t6_2.txt	0.02	×	0.0	×
instance_t6_3.txt	0.03	×	0.0	×
instance_t6_4.txt	0.02	×	0.0	×
instance_t6_5.txt	0.02		0.0	×
instance_t6_6.txt	0.03	×	0.0	×
instance_t6_7.txt	0.03	×	0.0	×
instance_t6_8.txt	0.03	×	0.01	×
instance_t6_9.txt	0.03	×	0.01	×
instance_t7_1.txt	0.04	×	0.02	×
instance_t7_10.txt	0.04	×	0.01	×
instance_t7_2.txt	0.03	×	0.01	×
instance_t7_3.txt	0.04		0.09	×
instance_t7_4.txt	0.03	×	0.01	×
instance_t7_5.txt	0.04	×	0.09	×
instance_t7_6.txt	0.03	×	0.07	×
instance_t7_7.txt	0.04	×	0.03	×
instance_t7_8.txt	0.04	×	0.04	×

Instance	cplex		heuristique	
	Temps (s)	Optimal?	Temps (s)	Optimal?
instance_t7_9.txt	0.04	×	0.0	×
instance_t5_1.txt	0.22	×	0.15	×
instance_t5_10.txt	0.06	×	0.01	×
instance_t5_2.txt	0.08	×	0.0	×
instance_t5_3.txt	0.09	×	0.0	×
instance_t5_4.txt	0.02	×	0.01	×
instance_t5_5.txt	0.02	×	0.0	×
instance_t5_6.txt	0.07	×	0.0	×
instance_t5_7.txt	0.02	×	0.0	×
instance_t5_8.txt	0.02	×	0.0	×
instance_t5_9.txt	0.03	×	0.0	×
instance_t6_1.txt	0.03	×	0.0	×
instance_t6_10.txt	0.03	×	0.0	×
instance_t6_2.txt	0.02	×	0.0	×
instance_t6_3.txt	0.03	×	0.0	×
instance_t6_4.txt	0.02	×	0.0	×
instance_t6_5.txt	0.02		0.0	×
instance_t6_6.txt	0.03	×	0.0	×
instance_t6_7.txt	0.03	×	0.0	×
instance_t6_8.txt	0.03	×	0.01	×
instance_t6_9.txt	0.03	×	0.01	×
instance_t7_1.txt	0.04	×	0.02	×
instance_t7_10.txt	0.04	×	0.01	×
instance_t7_2.txt	0.03	×	0.01	×
instance_t7_3.txt	0.04		0.09	×
instance_t7_4.txt	0.03	×	0.01	×
instance_t7_5.txt	0.04	×	0.09	×
instance_t7_6.txt	0.03	×	0.07	×
instance_t7_7.txt	0.04	×	0.03	×
instance_t7_8.txt	0.04	×	0.04	×

Instance	cplex		heuristique	
	Temps (s)	Optimal?	Temps (s)	Optimal?
instance_t7_9.txt	0.04	×	0.0	×

Au delà d'une taille de 10 l'heuristique commence à ne pas fournir de résultats sur certaines instances en moins de 100 000 itérations. Pour approfondir davantage ce projet il faudrait créer une heuristique plus intelligente, et notamment une heuristique qui mémorise les chemins déjà testés pour ne pas recommencer les mêmes erreurs, et qui de plus revient en arrière d'abord un peu, puis toujours plus tant qu'elle aboutit à des impasses.

Cplex résoud quant à lui aisément de grosses instances. Par exemple pour une instance de taille 100, il mets 33s. A ce stade c'est la fonction de génération des instances qui est limitante car elle met 5 :40 min.