RAPPORT PROJET MINIZINC

Réalisé par :

- Omar Juba
- Abdellah Bourbia
- Maxence Marot
- Maignan Quentin
- Chafik Akmouche

Sommaire

Table des matières

I - NReines

Réalisé par Abdellah Bourbia et Omar Lemdani.

1 - Description de l'exercice :

1- Le problème à résoudre est celui des « n-reines » qui consiste à placer n reines sur un échiquier (de taille n x n) de sorte qu'aucune reine ne puisse en attaquer une autre, ce qui implique les contraintes suivantes :

2- • Une et une seule reine par ligne

3- • Une et une seule reine par colonne

4- • Une et une seule reine par diagonale

5- Les contraintes ont été mises en place de différentes manières selon les 4 modèles demandées ainsi que leurs résolutions, on fera une comparaison et une analyse sur les résultats obtenus selon plusieurs critères.

2- Les Modèles

M1) Un modèle simple basé sur des variables « domaine finis » entières sans contrainte globale :

Le domaine va de 1 à n (n étant le nombre de reines), pour les variables, on a choisis d'utiliser un tableau d'entiers contenant n cases, les indices vont de 1 à n et représentent les colonnes de l'échiquier, donc la paire (indice, la valeur du tableau à cet indice) représente la position de la reine dans l'échiquier.

• **Remarque** : De part cette définition du problème, on obtient la contrainte d'une et une seule reine par colonne. (Comme c'est un tableau à une seule dimension, on n'a qu'une seule valeur pour chaque indice).

Il ne reste plus que 2 contraintes, pour obtenir la contrainte d'une et une seule reine par ligne, on impose que toutes les valeurs du tableau soient différentes.

Pour la dernière contrainte d'une et une seule reine par diagonale, on l'a divisé en deux parties :

L'addition d'une valeur du tableau et son indice doit être différente à toute autre valeur du tableau additionné à son indice, cela va permettre d'avoir une et une seule reine dans la diagonale « descendante » où elle se situe.

La soustraction d'une valeur du tableau et son indice doit être différente à toute autre valeur du tableau soustrait à son indice, cela va permettre d'avoir une et une seule reine dans la diagonale « ascendante » où elle se situe.

M2) Un modèle basé sur des variables « domaine finis » entières avec contraintes globales et cassant des symétries :

Le domaine et les variables choisis sont identiques au premier modèle. Par contre, pour les contraintes, on a utilisé la contrainte globale : « alldifferent » qui prend un tableau de variables et les contraint à prendre des valeurs différentes.
Comme pour le premier model, la contrainte d'une et une seule reine par colonne est obtenue de par les variables et le domaine choisis.
Pour la contrainte d'une et une seule ligne, on passe le tableau en paramètre à la contrainte globale 'alldifferent'.
Pour la contrainte d'une et une seule par diagonale, on a divisé la tâche en deux :

On passe un tableau qui contient, à chaque case, la valeur du tableau initial additionné à son indice, en paramètre à 'alldifferent', cela va nous permettre d'avoir qu'une et une seule reine dans la diagonale « descendante » où elle se situe.

On passe un tableau qui contient, à chaque case, la valeur du tableau initial soustrait à son indice, en paramètre à 'alldifferent', cela va nous permettre d'avoir qu'une et une seule reine dans la diagonale « ascendante » où elle se situe.

Afin de casser des symétries, il faut imposer un ordre, on a choisi d'imposer un ordre lexicographique et pour cela, on a utilisé la contrainte globale : « lex_lesseq » en lui passant le tableau et son inverse en paramètre.

M3) Un modèle basé sur des variables booléennes et contraintes booléennes :

Comme il s'agit d'un model booléen, le domaine ne contient que les deux valeurs : 0 et 1. Pour les variables, on a choisis d'utiliser un tableau à 2 dimensions contenant n lignes et n colonnes, donc les indices des colonnes comme pour les lignes vont de 1 à n (n étant le nombre de reines). La valeur du tableau sera égal à 1 si une reine se trouve à la ligne i et colonne j et 0 sinon.

Pour avoir la contrainte d'une et une seule reine par ligne, on impose que la somme des valeurs du tableau pour une colonne donnée soit égale à 1.

Pour avoir la contrainte d'une et une seule reine par colonne, on impose que la somme des valeurs du tableau pour une ligne donnée soit égale à 1.

Pour avoir la contrainte d'une et une seule reine par diagonale (ascendante et descendante), on affecte pour chaque case du tableau la valeur 1 si toutes les cases de la diagonale où elle se trouve sont égales à 0, sinon on affecte la valeur 0, si l'une des cases de la diagonale ou elle se trouve est égal à 1.

M4) un modèle basé sur des variables booléennes et contraintes booléennes et cassant des symétries :

Le domaine, les variables ainsi que toutes les contraintes décrites dans le troisième model sont identiques dans ce 4ème model, la différence c'est qu'on a ajouté plusieurs contraintes pour casser des symétries.

On a ajouté 7 contraintes pour casser les symétries en utilisant la contrainte globale « lex_lesseq » qui impose un ordre lexicographique. En lui passant en 1er paramètre le tableau de deux dimensions transformé en un tableau d'une seule dimension et en 2ème paramètre différentes permutations du tableau.

3 - Analyse et comparaison des modèles :

Modeles		Model 1	Model 2	Model 3	Model 4
Analyse et comparaison					
	Nombre de contraintes	3	4	3	10
	Nombre de variables	N	N	N*N	N*N
	Nombre de symétries	Nombre Symétrie Total	Nombre Symétrie Total/2	Nombre Symétrie Total	0

- Pour les modèles basés sur des variables entières (le model 1 et 2), on a le même nombre de variables (N qui représente le nombre de reine) et on a une contrainte supplémentaire pour le model 2 comparé au premier model afin de casser des symétries.
- Pour les modèles basés sur des variables booléennes (le model 3 et 4), on a le même nombre de variables (N*N vu qu'on a des tableaux à 2 dimensions). Cependant, on a 7 contraintes supplémentaire dans le model 4 comparé au troisième model afin de casser des symétries.
- Les modèles basés sur des variables booléenne ont N fois le nombre de variables des modèles basés sur des variables entières, car dans les modèles 3 et 4 on utilise des tableaux à 2 dimensions, par contre, dans les modèles 1 et 2 on utilise des tableaux à une seule dimension.
- Quand on compare les modèles cassant des symétries (le model 2 et 4), on remarque que le 4ème modèle à 6 contraintes supplémentaires que le troisième model mais on constate que le model 3 casse uniquement la moitié des symétries, cependant, le 4ème model casse toutes les symétries.
- Comme a pu le constaté, les modèles basés sur des variables booléenne ont plus de variables que ceux basés sur des variables entières, par contre , quand on compare les domaines, on constate une grande différence, pour les modèles à variables booléennes, on a un domaine qui contient seulement 2 valeurs (0 et 1), et pour les modèles à variables entières, le domaine contient N valeurs (N étant le nombre de reines).

4 - Les Résolutions :

R1) Résolution de M1 avec MiniZinc :

- Domaine et Variables :

```
int: n;

array [1..n] of var 1..n: q;
```

- Une et une seule reine par ligne :

```
constraint forall(i,j in 1..n where i!=j) (
                q[i]!=q[j]
            );
```

- Une et une seule reine par diagonale (ascendante) :

```
constraint forall(i,j in 1..n where i!=j) (
                q[i] + i != q[j] + j
            );
```

- Une et une seule reine par diagonale (descendante) :

```
constraint forall(i,j in 1..n where i!=j) (
                q[i] - i != q[j] - j
            );
```

- **Remarque** : Comme pour le model 1, on obtient la contrainte d'une et une seule reine par colonne de par les variables et domaine choisis (Comme le tableau 'q' n'a qu'une seule dimension, on n'aura qu'une seule valeur pour chaque indice).

R2) Résolution de M2 avec MiniZinc :

- Domaine et Variables :

```
int: n;

array [1..n] of var 1..n: q;
```

- Une et une seule reine par ligne :

```
constraint alldifferent(q);
```

- Une et une seule reine par diagonale (ascendante) :

```
constraint alldifferent([ q[i] + i | i in 1..n]);
```

- Une et une seule reine par diagonale (descendante) :

```
constraint alldifferent([ q[i] - i | i in 1..n]);
```

- Contrainte pour casser des symétries :

```
constraint forall(i,j in 1..n where i!=j) (
                q[i] - i != q[j] - j
            );
```

- **Remarque** : Comme pour la première résolution, on obtient la contrainte d'une et une seule reine par colonne de par les variables et domaine choisis.

R3) Résolution de M3 avec MiniZinc :

- Domaine et Variables :

```
int: n;

array[1..n,1..n] of var bool: qb;
```

- Une et une seule reine par ligne :

```
constraint forall(i in 1..n)
            (
              sum(j in 1..n) (
                bool2int(qb[i,j])
              ) = 1
            )  ;
```

- Une et une seule reine par colonne :

```
constraint forall(j in 1..n)
            (
              sum(i in 1..n) (
                bool2int(qb[i,j])
              ) = 1
            )  ;
```

- Une et une seule reine par diagonale (ascendante et descendante) :

```
constraint forall(i, j in 1..n)(
 qb[i,j] = not (
  exists(j1 in 1..n where j1 != j)(qb[i,j1])
 \/ exists(i1 in 1..n where i1 != i)(qb[i1,j])
 \/ exists(k in 1..n)(
   qb[i+k,j+k] \/ qb[i-k,j+k]
     \/ qb[i+k,j-k] \/ qb[i-k,j-k]
  )
 )
);
```

R4) Résolution de M4 avec MiniZinc :

• Le domaine, les variables ainsi que toutes les contraintes décrites dans la troisième résolution sont identiques dans cette 4ème résolution, la différence c'est l'ajout des 7 contraintes pour casser les symétries :

```
constraint
    lex_lesseq(array1d(qb), [ qb[j,i] | i,j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i in reverse(1..n), j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i in 1..n, j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i in 1..n, j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i in reverse(1..n), j in 1..n ])
/\  lex_lesseq(array1d(qb), [ qb[i,j] | i,j in reverse(1..n) ])
/\  lex_lesseq(array1d(qb), [ qb[j,i] | i,j in reverse(1..n) ])
;
```

5 - Analyse des résolutions :

• Nous avons fait varier le nombre de reines et les stratégies d'énumération pour analyser les résolutions.

Voici les stratégies d'énumération en question :

a) Choix de variables :

1) Input_order : Choisis dans l'ordre du tableau
2) First_fail : Choisis la variable dont la taille du domaine est la plus petite
3) Smallest : Chsois la variable qui a la plus petite valeur dans son domaine

b) Contraindre une variable :

1) indomain_min : Attribue à la variable sa plus petite valeur de son domaine

2) indomain_median : Attribue à la variable la valeur médian de son domaine

3) indomain_random : Attribue à la variable une valeur aléatoire de son domaine

4) indomain_split : Coupe en deux le domaine des variables et exclue la partie supérieur

R1 VS R2

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_1.mzn	n in [1,10] : 170ms -> 230ms	n in [1,10] : 180ms -> 240ms	n in [1,10] : 170ms -> 225ms	n in [1,10] : 170ms -> 230ms
	n = 11 : 380ms	n = 11 : 400ms	n = 11 : 370ms	n = 11 : 360ms
	n = 12 : 1s100ms	n = 12 : 1s200ms	n = 12 : 1s140ms	n = 12 : 1s100ms
	n = 13 : 5s200ms	n = 13 : 5s200ms	n = 13 : 5s250ms	n = 13 : 5s100ms
	n = 14 : 27s140ms	n = 14 : 27s	n = 14 : 27s	n = 14 : 26s250
nqueen_2.mzn	n in [1,10] : 160ms -> 200ms	n in [1,10] : 160ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 330ms	n = 11 : 310ms	n = 11 : 320ms	n = 11 : 320ms
	n = 12 : 930ms	n = 12 : 900ms	n = 12 : 920ms	n = 12 : 920ms
	n = 13 : 4s200ms	n = 13 : 4s80ms	n = 13 : 4s250ms	n = 13 : 4s250ms
	n = 14 : 23s500ms	n = 14 : 22s80ms	n = 14 : 23s800ms	n = 14 : 23s270ms

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_1.mzn	n in [1,10] : 180ms -> 230ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 230ms	n in [1,10] : 170ms -> 220ms
	n = 11 : 350ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 1s5ms	n = 12 : 1s2ms	n = 12 : 1s6ms	n = 12 : 1s18ms
	n = 13 : 4s517ms	n = 13 : 4s500ms	n = 13 : 4s500ms	n = 13 : 4s450ms
	n = 14 : 23s200	n = 14 : 23s46ms	n = 14 : 23s100ms	n = 14 : 23s300ms
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 290ms	n = 11 : 280ms	n = 11 : 290ms	n = 11 : 280ms
	n = 12 : 770ms	n = 12 : 750ms	n = 12 : 770ms	n = 12 : 750ms
	n = 13 : 3s350ms	n = 13 : 3s250ms	n = 13 : 3s350ms	n = 13 : 3s300ms
	n = 14 : 17s600ms	n = 14 : 17s200ms	n = 14 : 18s	n = 14 : 17s250ms

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_1.mzn	n in [1,10] : 160ms -> 1s200ms	n in [1,10] : 160ms -> 250ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 1s200ms
	n = 11 : 8s	n = 11 : 500ms	n = 11 : 1s	n = 11 : 9s
	n = 12 : 1minute	n = 12 : 2s	n = 12 : 4s500ms	n = 12 : 57s
	n = 13 : /	n = 13 : 11s250ms	n = 13 : 30s	n = 13 : //
	n = 14 : /	n = 14 : 1min10secondes	n = 14 : 3min //	n = 14 : //
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 210ms
	n = 11 : 300ms	n = 11 : 320ms	n = 11 : 370ms	n = 11 : 380ms
	n = 12 : 800ms	n = 12 : 980ms	n = 12 : 1s200ms	n = 12 : 1s200ms
	n = 13 : 3s700ms	n = 13 : 4s600ms	n = 13 : 6s300ms	n = 13 : 6s
	n = 14 : 20s200ms	n = 14 : 26s200ms	n = 14 : 38s	n = 14 : 32s

- On remarque que la résolution du deuxième modèle est plus rapide que la résolution du premier modèle (le temps d'exécution est plus court), et on obtient ce résultat qu'importe la stratégie d'énumération choisie.

- On remarque également que la différence de temps s'accroit proportionnellement aux nombres de reines.

R3 VS R4

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 630ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s850ms	n = 13 : 9s350ms
	n = 14 : 50s	n = 14 : 50s200ms	n = 14 : 52s500ms	n = 14 : 50s
nqueen_4.mzn	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms
	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 730ms	n = 12 : 730ms	n = 12 : 740ms	n = 12 : 740ms
	n = 13 : 2s520ms	n = 13 : 2s520ms	n = 13 : 2s650ms	n = 13 : 2s620ms
	n = 14 : 12s300ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s780ms
	n = 15 : 1m10s	n = 15 : 1m10s	n = 15 : 1m13s	n = 15 : 1m13s

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 620ms	n = 11 : 600ms
nqueen_3.mzn	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100ms	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s800ms	n = 13 : 9s350ms
	n = 14 : 50s100ms	n = 14 : 50s200ms	n = 14 : 52s600ms	n = 14 : 50s
	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 290ms
	n = 11 : 340ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 340ms
	n = 12 : 720ms	n = 12 : 720ms	n = 12 : 750ms	n = 12 : 720ms
nqueen_4.mzn	n = 13 : 2s500	n = 13 : 2s500	n = 13 : 2s600	n = 13 : 2s530
	n = 14 : 12s500ms	n = 14 : 12s500ms	n = 14 : 12s700ms	n = 14 : 12s25ms
	n = 15 : 1m9s	n = 15 : 1m10s	n = 15 : 1m12s	n = 15 : 1m9s

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 630ms	n = 11 : 600ms
nqueen_3.mzn	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s850ms	n = 13 : 9s350ms
	n = 14 : 50s	n = 14 : 50s200ms	n = 14 : 52s500ms	n = 14 : 50s
	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms
	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 730ms	n = 12 : 730ms	n = 12 : 740ms	n = 12 : 740ms
nqueen_4.mzn	n = 13 : 2s520ms	n = 13 : 2s520ms	n = 13 : 2s650ms	n = 13 : 2s620ms
	n = 14 : 12s300ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s780ms
	n = 15 : 1m10s	n = 15 : 1m10s	n = 15 : 1m13s	n = 15 : 1m13s

- On remarque des résultats similaires avec R1 et R2, R4 est plus rapide que R3 la différence de temps s'accroit proportionnellement aux nombres de reines.
- On peut conclure que le fait de casser des symétries permet un gain de temps considérable.

R1 VS R3

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_1.mzn	n in [1,10] : 170ms -> 230ms	n in [1,10] : 180ms -> 240ms	n in [1,10] : 170ms -> 225ms	n in [1,10] : 170ms -> 230ms
	n = 11 : 380ms	n = 11 : 400ms	n = 11 : 370ms	n = 11 : 360ms
	n = 12 : 1s100ms	n = 12 : 1s200ms	n = 12 : 1s140ms	n = 12 : 1s100ms
	n = 13 : 5s200ms	n = 13 : 5s200ms	n = 13 : 5s250ms	n = 13 : 5s100ms
	n = 14 : 27s140ms	n = 14 : 27s	n = 14 : 27s	n = 14 : 26s250
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 610ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100ms	n = 12 : 2s
	n = 13 : 9s300ms	n = 13 : 9s300ms	n = 13 : 9s800ms	n = 13 : 9s300ms
	n = 14 : 50s	n = 14 : 51s	n = 14 : 52s600ms	n = 14 : 50s800ms

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	fail/indomain_split
nqueen_1.mzn	n in [1,10] : 180ms -> 230ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 230ms	n in [1,10] : 170ms -> 220ms
	n = 11 : 350ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 1s5ms	n = 12 : 1s2ms	n = 12 : 1s6ms	n = 12 : 1s18ms
	n = 13 : 4s517ms	n = 13 : 4s500ms	n = 13 : 4s500ms	n = 13 : 4s450ms
	n = 14 : 23s200	n = 14 : 23s46ms	n = 14 : 23s100ms	n = 14 : 23s300ms
	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	fail/indomain_split
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 620ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100ms	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s800ms	n = 13 : 9s350ms
	n = 14 : 50s100ms	n = 14 : 50s200ms	n = 14 : 52s600ms	n = 14 : 50s

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_1.mzn	n in [1,10] : 160ms -> 1s200ms	n in [1,10] : 160ms -> 250ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 1s200ms
	n = 11 : 8s	n = 11 : 500ms	n = 11 : 1s	n = 11 : 9s
	n = 12 : 1minute	n = 12 : 2s	n = 12 : 4s500ms	n = 12 : 57s
	n = 13 : /	n = 13 : 11s250ms	n = 13 : 30s	n = 13 : //
	n = 14 : /	n = 14 : 1min10secondes	n = 14 : 3min //	n = 14 : //
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 630ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s850ms	n = 13 : 9s350ms
	n = 14 : 50s	n = 14 : 50s200ms	n = 14 : 52s500ms	n = 14 : 50s

- On compare la résolution du modèle 1 et la résolution du modèle 3 afin de voir la différence entre une résolution d'un modèle à variables booléennes et une résolution d'un modèle à variables entières, on remarque que R1 est plus rapide que R3. (Le temps d'exécution est plus court)
- On peut conclure que les modèles à variable entière sont plus rapides que les modèles à variables booléennes, et cela est dû aux nombres de variables (les modèles booléens ayant N fois le nombre de variables des modèles entiers).

R2 VS R4

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_2.mzn	n in [1,10] : 160ms -> 200ms	n in [1,10] : 160ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 330ms	n = 11 : 310ms	n = 11 : 320ms	n = 11 : 320ms
	n = 12 : 930ms	n = 12 : 900ms	n = 12 : 920ms	n = 12 : 920ms
	n = 13 : 4s200ms	n = 13 : 4s80ms	n = 13 : 4s250ms	n = 13 : 4s250ms
	n = 14 : 23s500ms	n = 14 : 22s80ms	n = 14 : 23s800ms	n = 14 : 23s270ms
nqueen_4.mzn	n in [1, 10] : 160ms -> 290ms	n in [1, 10] : 160ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 165ms -> 300ms
	n = 11 : 340ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 340ms
	n = 12 : 710ms	n = 12 : 710ms	n = 12 : 740ms	n = 12 : 720ms
	n = 13 : 2s500	n = 13 : 2s500	n = 13 : 2s600	n = 13 : 2s500
	n = 14 : 12s320ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s250ms
	n = 15 : 1m9s	n = 15 : 1m9s	n = 15 : 1m13s	n = 15 : 1m9s

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 290ms	n = 11 : 280ms	n = 11 : 290ms	n = 11 : 280ms
	n = 12 : 770ms	n = 12 : 750ms	n = 12 : 770ms	n = 12 : 750ms
	n = 13 : 3s350ms	n = 13 : 3s250ms	n = 13 : 3s350ms	n = 13 : 3s300ms
	n = 14 : 17s600ms	n = 14 : 17s200ms	n = 14 : 18s	n = 14 : 17s250ms
nqueen_4.mzn	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 290ms
	n = 11 : 340ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 340ms
	n = 12 : 720ms	n = 12 : 720ms	n = 12 : 750ms	n = 12 : 720ms
	n = 13 : 2s500	n = 13 : 2s500	n = 13 : 2s600	n = 13 : 2s530
	n = 14 : 12s500ms	n = 14 : 12s500ms	n = 14 : 12s700ms	n = 14 : 12s25ms
	n = 15 : 1m9s	n = 15 : 1m10s	n = 15 : 1m12s	n = 15 : 1m9s

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 210ms
	n = 11 : 300ms	n = 11 : 320ms	n = 11 : 370ms	n = 11 : 380ms
	n = 12 : 800ms	n = 12 : 980ms	n = 12 : 1s200ms	n = 12 : 1s200ms
	n = 13 : 3s700ms	n = 13 : 4s600ms	n = 13 : 6s300ms	n = 13 : 6s
	n = 14 : 20s200ms	n = 14 : 26s200ms	n = 14 : 38s	n = 14 : 32s
nqueen_4.mzn	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms
	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 730ms	n = 12 : 730ms	n = 12 : 740ms	n = 12 : 740ms
	n = 13 : 2s520ms	n = 13 : 2s520ms	n = 13 : 2s650ms	n = 13 : 2s620ms
	n = 14 : 12s300ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s780ms
	n = 15 : 1m10s	n = 15 : 1m10s	n = 15 : 1m13s	n = 15 : 1m13s

- On remarque que R4 est beaucoup plus rapide que R2. (Le temps d'exécution est beaucoup plus court) parce que R4 casse toutes les symétries, alors que R2 casse uniquement la moitié des symétries.
- On conclue que R4 est la meilleure résolution pour ce problème.

• Pour le n maximum atteint, MiniZinc ne nous permet de savoir quand est extrêmement long et quand il est infini, donc on a fixé une limite de temps :
- R1 => n_max = 16
- R2 => n_max = 17
- R3 => n_max = 15
- R4 => n_max = 18

• En ce qui concerne les stratégies d'énumérations :

R1

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_1.mzn	n in [1,10] : 170ms -> 230ms	n in [1,10] : 180ms -> 240ms	n in [1,10] : 170ms -> 225ms	n in [1,10] : 170ms -> 230ms
	n = 11 : 380ms	n = 11 : 400ms	n = 11 : 370ms	n = 11 : 360ms
	n = 12 : 1s100ms	n = 12 : 1s200ms	n = 12 : 1s140ms	n = 12 : 1s100ms
	n = 13 : 5s200ms	n = 13 : 5s200ms	n = 13 : 5s250ms	n = 13 : 5s100ms
	n = 14 : 27s140ms	n = 14 : 27s	n = 14 : 27s	n = 14 : 26s250

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_1.mzn	n in [1,10] : 180ms -> 230ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 230ms	n in [1,10] : 170ms -> 220ms
	n = 11 : 350ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 1s5ms	n = 12 : 1s2ms	n = 12 : 1s6ms	n = 12 : 1s18ms
	n = 13 : 4s517ms	n = 13 : 4s500ms	n = 13 : 4s500ms	n = 13 : 4s450ms
	n = 14 : 23s200	n = 14 : 23s46ms	n = 14 : 23s100ms	n = 14 : 23s300ms

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_1.mzn	n in [1,10] : 160ms -> 1s200ms	n in [1,10] : 160ms -> 250ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 1s200ms
	n = 11 : 8s	n = 11 : 500ms	n = 11 : 1s	n = 11 : 9s
	n = 12 : 1minute	n = 12 : 2s	n = 12 : 4s500ms	n = 12 : 57s
	n = 13 : /	n = 13 : 11s250ms	n = 13 : 30s	n = 13 : //
	n = 14 : /	n = 14 : 1min10secondes	n = 14 : 3min //	n = 14 : //

- Pour R1, la meilleure combinaison est first_fail et indomain_median car il obtient les meilleurs résultats. Par contre, on remarque que smallest obtient les pires résultats, il est donc le moins adapté.

R2

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_2.mzn	n in [1,10] : 160ms -> 200ms	n in [1,10] : 160ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 330ms	n = 11 : 310ms	n = 11 : 320ms	n = 11 : 320ms
	n = 12 : 930ms	n = 12 : 900ms	n = 12 : 920ms	n = 12 : 920ms
	n = 13 : 4s200ms	n = 13 : 4s80ms	n = 13 : 4s250ms	n = 13 : 4s250ms
	n = 14 : 23s500ms	n = 14 : 22s80ms	n = 14 : 23s800ms	n = 14 : 23s270ms

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms
	n = 11 : 290ms	n = 11 : 280ms	n = 11 : 290ms	n = 11 : 280ms
	n = 12 : 770ms	n = 12 : 750ms	n = 12 : 770ms	n = 12 : 750ms
	n = 13 : 3s350ms	n = 13 : 3s250ms	n = 13 : 3s350ms	n = 13 : 3s300ms
	n = 14 : 17s600ms	n = 14 : 17s200ms	n = 14 : 18s	n = 14 : 17s250ms

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_2.mzn	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 200ms	n in [1,10] : 170ms -> 220ms	n in [1,10] : 170ms -> 210ms
	n = 11 : 300ms	n = 11 : 320ms	n = 11 : 370ms	n = 11 : 380ms
	n = 12 : 800ms	n = 12 : 980ms	n = 12 : 1s200ms	n = 12 : 1s200ms
	n = 13 : 3s700ms	n = 13 : 4s600ms	n = 13 : 6s300ms	n = 13 : 6s
	n = 14 : 20s200ms	n = 14 : 26s200ms	n = 14 : 38s	n = 14 : 32s

- Pour R2, la meilleure combinaison est encore une fois first_fail et indomain_median. Cependant, il n'y a pas de très mauvais résultat car les combinaisons avec smallest et input_order sont proche du meilleur résultat.

R3

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 610ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100ms	n = 12 : 2s
	n = 13 : 9s300ms	n = 13 : 9s300ms	n = 13 : 9s800ms	n = 13 : 9s300ms
	n = 14 : 50s	n = 14 : 51s	n = 14 : 52s600ms	n = 14 : 50s800ms

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 620ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100ms	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s800ms	n = 13 : 9s350ms
	n = 14 : 50s100ms	n = 14 : 50s200ms	n = 14 : 52s600ms	n = 14 : 50s

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_3.mzn	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms	n in [1,10] : 160ms -> 300ms
	n = 11 : 600ms	n = 11 : 600ms	n = 11 : 630ms	n = 11 : 600ms
	n = 12 : 2s	n = 12 : 2s	n = 12 : 2s100	n = 12 : 2s
	n = 13 : 9s350ms	n = 13 : 9s350ms	n = 13 : 9s850ms	n = 13 : 9s350ms
	n = 14 : 50s	n = 14 : 50s200ms	n = 14 : 52s500ms	n = 14 : 50s

- Pour R3, il y'a plusieurs combinaisons qui obtiennent le meilleur résultat : input_order/indomain_min, first_fail/indomain_split, smallest/indomain_min et smallest/indomain_split. On peut conclure qu'indomain_random est moins adapté pour cette résolution.

R4

	first_fail/indomain_min	first_fail/indomain_median	first_fail/indomain_random	first_fail/indomain_split
nqueen_4.mzn	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 290ms
	n = 11 : 340ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 340ms
	n = 12 : 720ms	n = 12 : 720ms	n = 12 : 750ms	n = 12 : 720ms
	n = 13 : 2s500	n = 13 : 2s500	n = 13 : 2s600	n = 13 : 2s530
	n = 14 : 12s500ms	n = 14 : 12s500ms	n = 14 : 12s700ms	n = 14 : 12s25ms
	n = 15 : 1m9s	n = 15 : 1m10s	n = 15 : 1m12s	n = 15 : 1m9s

	input_order/indomain_min	input_order/indomain_median	input_order/indomain_random	input_order/indomain_split
nqueen_4.mzn	n in [1, 10] : 160ms -> 290ms	n in [1, 10] : 160ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 165ms -> 300ms
	n = 11 : 340ms	n = 11 : 340ms	n = 11 : 350ms	n = 11 : 340ms
	n = 12 : 710ms	n = 12 : 710ms	n = 12 : 740ms	n = 12 : 720ms
	n = 13 : 2s500	n = 13 : 2s500	n = 13 : 2s600	n = 13 : 2s500
	n = 14 : 12s320ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s250ms
	n = 15 : 1m9s	n = 15 : 1m9s	n = 15 : 1m13s	n = 15 : 1m9s

	smallest/indomain_min	smallest/indomain_median	smallest/indomain_random	smallest/indomain_split
nqueen_4.mzn	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms	n in [1, 10] : 170ms -> 300ms
	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms	n = 11 : 350ms
	n = 12 : 730ms	n = 12 : 730ms	n = 12 : 740ms	n = 12 : 740ms
	n = 13 : 2s520ms	n = 13 : 2s520ms	n = 13 : 2s650ms	n = 13 : 2s620ms
	n = 14 : 12s300ms	n = 14 : 12s300ms	n = 14 : 12s800ms	n = 14 : 12s780ms
	n = 15 : 1m10s	n = 15 : 1m10s	n = 15 : 1m13s	n = 15 : 1m13s

- Pour R4, les combinaisons d'input_order sont légèrement meilleurs que les autres. Mais encore une fois les combinaisons avec indomain_random obtiennent les mauvais résultats.

• La combinaison first_fail/indomain_median est la meilleur pour les résolutions basés sur des variables entières, tandis que les combinaisons avec input_order semble est le plus sûr (celui qui obtient les meilleurs résultats) pour les résolutions basés sur des variables booléennes.

6 - Outils utilisés & Source :

Dans le cadre de la résolution de cet exercice, on a utilisé le langage de modélisation de contraintes : MiniZinc , ainsi que sa documentation officielle : MiniZinc Handbook (*https://www.minizinc.org/doc-2.3.0/en*).

7 - Conclusion :

Pour conclure, les langages de modélisation de contraintes tel que MiniZinc sont très efficace pour résoudre des problèmes complexes très rapidement, comme cela a été le cas pour le problème N-Queens.

II - Garam

Réalisé par Maxence Marot et Quentin Maignan.

Préambule : le but de cet exercice et d'automatiser la résolution du garam jeu de mathématique.

1 - les premières idées

Avant même de passer au code nous avons essayé de réfléchir au modèle qui serait le plus adapter pour modéliser le Garam. Pour ce faire nous avons commencé par jouer plusieurs parties pour bien comprendre le fonctionnement du jeu et comprendre toutes les règles. Une fois ce travail de « découverte » nous nous sommes mis à réaliser des esquisses de modélisation, mais pas sur Minizinc directement, d'abord sur feuille, pour que la syntaxe ne nous pose aucun problème.

Après plusieurs minutes de réflexion nous nous sommes mis d'accord sur un modèle qui représente le jeu comme un tableau a deux dimensions.

Chaque case du tableau représentait une information du Garam : soit un chiffre (1/9), soit une valeur a trouvée (0), soit une opération (10/12), soit un vide (-1). Nous allions donc mettre des contraintes à chaque ligne suivant la forme :

```
constraint if garam[1,3] == 1 then garam[1,4] = garam[1,1] + garam[1,2] endif;
```

Ce qui se traduit en français par : « si la case [1,3] est un '+' alors la case [1,4] est égale à la case [1,1] + la case [1,2]. Et cela pour toutes les opérations du jeu.

Nous avons pris la partie de ne pas faire de boucles pour modéliser ce jeu.

2 - des contraintes qui guident vers un modèle plus propre

Nous avions maintenant une bonne idée de comment modéliser, avec Minizinc, notre problème, mais presque dès le début nous avons compris que le fait d'avoir qu'un seul tableau pour tout représenter serait insuffisant. D'une part mélanger les valeurs et les opérations dans le même tableau n'avait pas trop de sens, en effet pourquoi une case représenterai le nombre 3 et celle voisine un '+' qui est codé avec l'entier 10 et pas le nombre 10 ? Nous avons donc choisi de retirer toutes les opérations de notre matrice. Désormais on a une matrice pour représenter les nombres et les cases vides, et un tableau à une dimension pour les opérations. D'autre part car nous commencions à nous mélanger lors de l'ajout des contraintes et l'entrée des données, car notre matrice était illisible du fait de son grand nombre de ligne et de colonne.

Un peu plus tard dans la réalisation de notre programme nous avons eu de nouveau soucis, en effet avec la ligne de code de la partie ci-dessus, nous étions incapable de récrire la valeur de garam[1,4], en effet c'est contradictoire de mettre un contrainte sur sa valeur alors qu'on a déjà rentrée sa valeur dans notre matrice. Nous avions donc un modèle insatisfiable dans tous les cas, car garam[1,4] ne peut pas être égale à 0 et en même temps a garam[1,1] + garam[1,2]. Une var et un Int sont complètement diffèrent pour Minizinc, de plus dans notre modèle nous réalisons un :

```
solve satisfy;
```

Ce qui dit que toutes les contraintes doivent être validées. De ce fait nous avons encore séparer notre matrice, cette fois ci en deux parties, les entrées d'un côté et les valeurs définitives de l'autre, comme ceci :

```
array[nbLigne, nbColonne] of var -1..9: garamEntry;

array[nbLigne, nbColonne] of var 0..9: garam;
```

Nous avons donc une matrice entièrement constituée de var (garam) ce qui va nous permettre de réaliser toutes nos contraintes sans avoir le problème 'modèle insatisfiable' cité plus haut et une autre pour entrer les données (garamEntry).

Notre première contrainte et donc de remplir notre matrice (garam) grâce aux valeurs de (garamEntry) comme ceci :

```
constraint forall(i, j in nbLigne)(if  j < 8  /\ garamEntry[i,j] > 0 then garam[i,j] = garamEntry[i,j]
                                    endif);
```

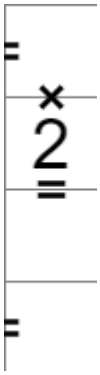
3 - Un modèle pour un garam de taille fixe

Dans notre modélisation, nous avons un problème de taille, en effet le fait d'avoir rentrée toutes nos contraintes en « dur » (comme dans la partie 1), nous as certes apporté un code facile à écrire et rapide à lire, mais nous empêche de rendre dynamique notre modèle. De ce fait une des améliorations possibles pour notre modèle de garam serait de le rendre adaptatif pour toutes les autres formes de garam possible et inimaginable.

Par exemple nous aurions pu mettre nos contraintes dans une boucle qui aurait passé en revue toute la matrice et qui aurait créé à la volée toutes les bonnes contraintes. Nous aurions aussi pu changer notre représentation pour décomposer encore plus notre matrice, et pourquoi pas avoir plusieurs tableaux avec des informations pour lier les lier entres eux et donc facilité le changement de structure du garam

4 – Les dernières contraintes

Après avoir testé notre modèle sur des exemples réels nous avons eu un problème, certaines lignes n'était pas acceptées par le vérificateur du site du garam, notre problème



était vrai facile à régler car c'était juste le fait d'accepter des zéros sur la première case des doubles cases qui posait un problème : comme ci-dessous.

Donc on a juste ajouté quelques contraintes sur ces cases pour les empêcher d'être égale à zéro.

```
constraint garam[3,1] > 0;
```

5 – L'analyse des résultats

Une exécution moyenne du garam nous prends environ 300ms avec une variance très faible entre deux garam, ce qui veut dire que notre programme met autant de temps à résoudre un exemple très difficile qu'une simple vérification de garam, ce qui montre la puissance de Minizinc.

Notre modélisation va toujours trouver la solution avec les plus petites valeurs possible, nous pourrions changer ce comportement en rajoutant des maximise sur les valeurs par exemple.

Nous avons aussi remarqué, que la taille de l'arbre ne correspond pas à la difficulté du garam, au début nous pensions que plus le garam serait dur plus nous aurions à remonter dans l'arbre mais enfaite cela dépends juste du garam, par exemple vous trouverez deux exemple de garam en .dzn, un normal de difficulté facile et un hard de difficulté maximale, et il se trouve que le facile à plus de 'leaves' et de 'braches' que celui très difficile. Ce qui montre encore une fois que notre modèle ne résous pas du tout comme nous les garams. Notre modèle va juste faite la somme de toutes les possibilités et garder la première qui satisfasse toutes les contraintes.

6 – exemple d'utilisation

Nous allons d'abord sur le site du garam pour prendre un exemple de problème à résoudre. Puis nous créons un fichier data qui correspond au problème à résoudre, avec la matrice du garam et le tableau des opérations. Nous lançons contre compilation et en sortie nous avons une matrice avec le garam complet !

Si nous obtenons un modèle UNSATISFIABLE c'est que le garam est impossible à résoudre.

III - Rikudo

Réalisé par Chafik AKMOUCHE

1 - Présentation du Rikudo

Rikudo est un jeu de logique de la famille des jeux de labyrinthes de nombres.

2 - Caractéristiques et règles du jeu (rikudo)

- Rikudo est une grille en forme d'hexagone qui contient des cellules (remplies ou à remplir) voir figure 1.
- La cellule centrale ne peut pas contenir de nombre, par conséquent, aucun chemin ne peut la traverser, voir figure 2.
- Les nombres ne peuvent pas être répétés.
- Une solution correcte doit remplir toutes les cellules ayant chacune un numéro unique.
- L'objectif final est de placer tous les numéros de 1 à 36 pour former un chemin de nombres consécutifs, voir figure 2.
- Deux nombres consécutifs doivent être voisins.

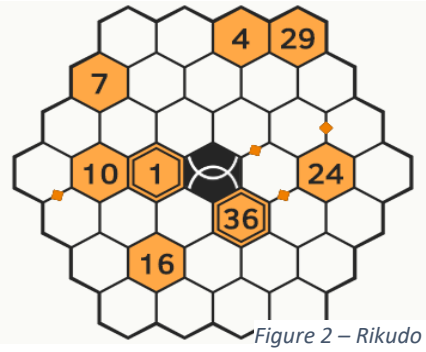


Figure 2 – Rikudo 36

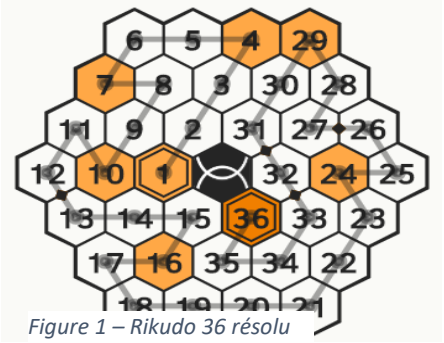


Figure 1 – Rikudo 36 résolu

3 - Réalisation d'un modèle rikudo (MiniZinc)

3.1 - Réalisation de la grille rikudo

À partir d'une matrice 7x7 (49 cellules), on construit la grille rikudo en associant à partir des paramètres d'entrée (fournis par l'utilisateur) des valeurs négatives aux cellules inutilisables (cellules en rouge sur la figure 3 et figure 4).

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

Figure 3

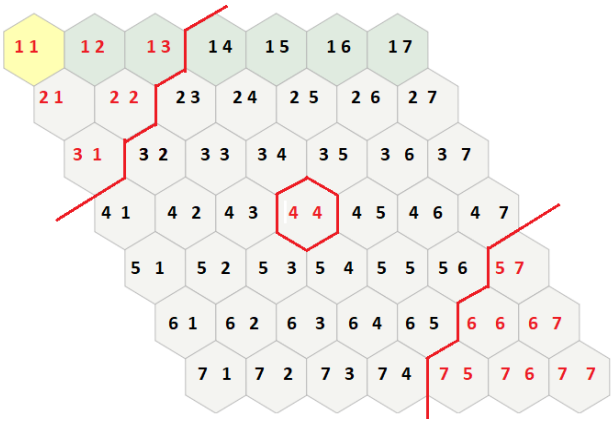


Figure 4

3.2 - Les nombres associés à chaque cellule doivent être différents

Contrainte (MiniZinc) qui satisfait cette règle :

```
constraint forall (i,j in x) (alldifferent (grille));
```

i, j : indices de la matrice (grille)

x : entier (1..7)

3.3 - Règles de voisinage pour construire un chemin de nombres consécutifs

3.3.1 - Règles de voisinage pour les coins de l'hexagone

Chaque cellule située dans l'un des 6 coins de la grille a 3 cellules voisines, donc son successeur doit être dans l'une de ces 3 cellules.

a. Coin supérieur gauche (1,4)

```
constraint forall(i,j in x)(
          if i=1 /\ j=4 /\ grille[i,j]<36 then
          grille[i,j]+1 = grille[i,j+1]   \/
          grille[i,j]+1 = grille[i+1,j]   \/
          grille[i,j]+1 = grille[i+1,j-1]
          endif
          );
```

b. Coin supérieur droit (1,7)

```
constraint forall(i,j in x)(
            if i=1 /\ j=7 /\ grille[i,j]<36 then
            grille[i,j]+1 = grille[i,j-1]   \/
            grille[i,j]+1 = grille[i+1,j-1] \/
            grille[i,j]+1 = grille[i+1,j]
            endif
            );
```

Et ainsi de suite pour les 4 coins restants.

3.3.2 - Règles de voisinage pour les bordures de l'hexagone

Chaque cellule aux bordures de la grille a 4 cellules voisines, donc son successeur doit être dans l'une de ces 4 cellules.

Contraintes (MiniZinc) qui permettent de satisfaire cette règle :

a. Bordure supérieure (à l'exclusion des coins)

```
constraint forall(i,j in x)(
            if i=1 /\ (j>4 /\ j<7) /\ grille[i,j]<36 then
            grille[i,j]+1 = grille[i,j-1]   \/
            grille[i,j]+1 = grille[i+1,j-1] \/
            grille[i,j]+1 = grille[i+1,j]   \/
            grille[i,j]+1 = grille[i,j+1]
            endif
            );
```

b. Bordure supérieure gauche (à l'exclusion des coins)

```
constraint forall(i,j in x)(
            if j=1 /\ (i<7 /\ i>4) /\ grille[i,j]<36 then
            grille[i,j]+1 = grille[i-1,j]   \/
            grille[i,j]+1 = grille[i-1,j+1] \/
            grille[i,j]+1 = grille[i,j+1]   \/
            grille[i,j]+1 = grille[i+1,j] endif );
```

Et ainsi de suite pour les 4 bordures restantes

3.3.3 Reste des cellules de la grille

Chaque cellule qui n'est pas dans l'un des 6 coins ou bordures de la grille a 6 cellules voisines, donc son successeur doit être dans l'une des six cellules voisines.

Contrainte (MiniZinc) qui satisfait cette règle :

```
constraint forall(i,j in x)(
if grille[i,j]>0 /\ grille[i,j]<36 /\ i!=1 /\ i!=7 /\ j!=1 /\ j!=7 then
   grille[i,j]+1 = grille[i,j-1]   \/
   grille[i,j]+1 = grille[i+1,j-1] \/
   grille[i,j]+1 = grille[i+1,j]   \/
   grille[i,j]+1 = grille[i,j+1]   \/
   grille[i,j]+1 = grille[i-1,j+1] \/
   grille[i,j]+1 = grille[i-1,j]
endif);
```

4 - Fonctionnement du modèle réalisé

À partir d'un fichier « .dzn » on initialise les paramètres d'entrée :

Une matrice (7x7) remplie de nombres positifs, négatif ou des 0 (voir figure 5) tel que :

- Nombre négatif : cellule inutilisable
- Nombre positif : valeur définitive de la case
- 0 : case à remplir par le modèle réalisé

```
% Paramètres d'entrée

[|-11,-13,-4,     0, 0,  4, 29
 | -6, -5,      7, 0, 0,  0,  0
 | -7,         0, 0, 0, 0,  0,  0
 |           0, 10, 1, -1, 0, 24, 0
 |            0,  0,  0, 36, 0,  0    , -8
 |              0,  16,  0, 0, 0     , -2, -9
 |                0,  0,  0, 0     , -3,-12,-10|];
```

Figure 5 – Exemple de paramètres d'entrée

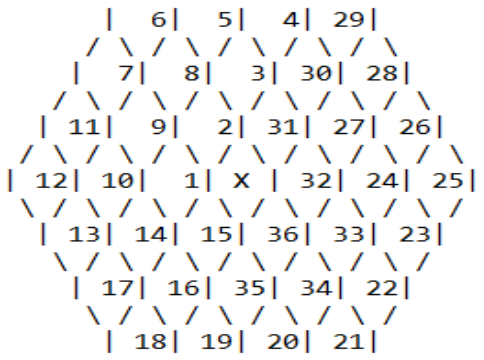


Figure 6 - Résultat

5 - Analyse des résultats

Toutes les règles du jeu, traduites en contraintes sous MiniZinc sont satisfaites (toutes les cellules sont remplies, pas de redondance des nombres, règles de voisinage respectées, etc.)

Temps d'exécution selon les paramètres d'entrée :

- Paramètres d'entrée avec 0 nombre donné : ~ 6 s
- Paramètres d'entrée avec seulement un point du départ (1) et un point d'arrivée (36) : ~ 19 s
- Paramètres d'entrée avec 10 nombres données : ~ 31 s

Pour chaque cellule de la grille, le modèle parcourt tous les chemins possibles jusqu'à ce qu'il trouve le chemin approprié vers la dernière cellule (36).

6 - Proposition d'une amélioration pour le modèle réalisé :

Parcourir tous les chemins de chaque case vers la dernière case (36) est couteux en termes de temps d'exécution. Pour cela d'autres contraintes peuvent être ajoutées afin d'éviter au modèle de parcourir tous les chemins existants pour chaque case.

- Éviter les cases encerclées

Si une case est encerclée lors du remplissage, elle devient bloquée et donc inaccessible. Le modèle doit être en mesure d'identifier ces situations et les éviter, ce qui va diminuer le nombre de chemins à parcourir.