# RAPPORT PROJET MINIZINC


Réalisé par :

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

# *Sommaire*

## Table des matières

# I - NReines

# II - Garam

Réalisé par Maxence Marot et Quentin Maignan.

## 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 réaliser, 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 pou 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);
```

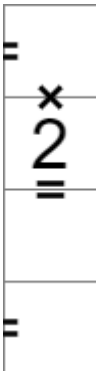
## 3 - Un modèle pour un garam de taille fixe

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 la 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.

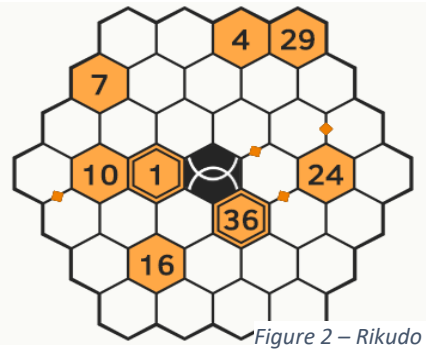



*Figure 2 – Rikudo 36*

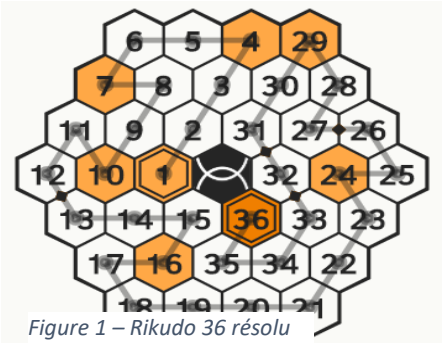



*Figure 1 – Rikudo 36 résolu*

# 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*

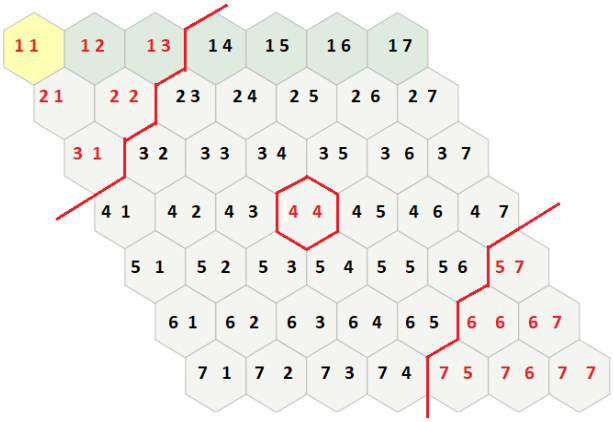



*Figure 4*

## 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*

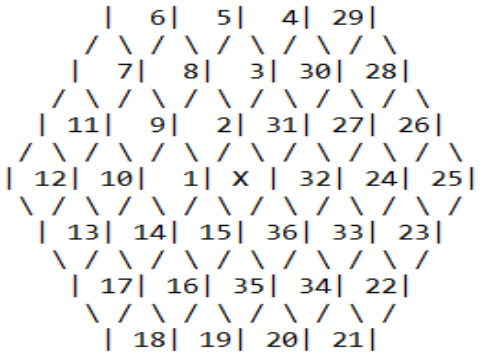



*Figure 6 - Résultat*

# 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.