2020/2021

Master 1 Informatique

MODÉLISATION ET RÉSOLUTION

Réalisé par :

Coyle Matthew

Khadji Marwa

Abou-Bichara Khaoula

El Haimour Amina

Encadré par :

MONFROY Eric

Table des matières

Tables des figures

Introduction Générale

La **Programmation Par Contraintes** (PPC, ou CP pour constraint programming en anglais) est un paradigme de programmation permettant de résoudre des problèmes combinatoires de grande taille tels que les problèmes de planification et d'ordonnancement. En programmation par contraintes, on sépare la partie modélisation à l'aide de problèmes de satisfaction de contraintes (ou CSP pour Constraint Satisfaction Problem), de la partie résolution dont la particularité réside dans l'utilisation active des contraintes du problème pour réduire la taille de l'espace des solutions à parcourir (on parle de propagation de contraintes).

Dans le cadre de la programmation par contraintes, les problèmes sont modélisés à l'aide de variables de décision et de contraintes, où une contrainte est une relation entre une ou plusieurs variables qui limite les valeurs que peuvent prendre simultanément chacune des variables liées par la contrainte.

Dans ce rapport, on va travailler sur 3 problèmes : N-reines, Garam, Rikudo. Pour le premier, on a 4 modèles avec 2 résolutions pour les modèles 3 et 4. Et pour le deuxième et le dernier modèle, on a un modèle pour chacun.

Exercice 1 : Les n-reines

Introduction

Le problème consiste à placer n reines sur un échiquier n × n de telle sorte qu'aucune reine ne puisse en attaquer une autre, N étant un paramètre.

En souhaitant résoudre le problème des N reines. Ils doivent pouvoir choisir la taille du problème, la méthode de résolution, ses paramètres, et le traitement éventuel des résultats obtenus.

I. M1 : Un modèle simple basé sur des variables « domaine fini » entières (FD) sans contrainte globale

Un modèle classique sans contraintes globales est défini dans les formules suivant.

En observant que deux reines ne peuvent pas être placées sur la même colonne, on peut imposer que la reine i soit sur la colonne i. Ainsi, la variable lide le domaine {1, . . ., n} représente la ligne où est placée la reine dans la colonne i.

Un premier modèle consiste à associer deux variables entières L i et C i à chaque reine i.

L i et C i correspondent respectivement à la ligne et à la colonne sur laquelle la reine est positionnée. Le problème s'écrit alors comme suit :

✓ Variables

X = {L 1, ..., L n, C 1, ..., C n} ;

✓ Domaines de définition

D (L 1) = ... = D (L n) = D (C 1) = ... = D (C n) = {1..n} ;

✓ Contraintes

Les reines doivent être sur des lignes différentes :

→ Clig = L i 6 ≠ L j /i ∈ {1..n } , j ∈ { 1..n } , i 6 ≠ j }

Les reines doivent être sur des colonnes différentes :

→ Ccol = C i 6 ≠ C j /i ∈ {1..n } , j ∈ { 1..n } , i 6 ≠j }

Les reines doivent être sur des diagonales ascendantes différentes :

→ Cdm = C i + L i 6 ≠ C j + L j /i ∈ {1..n } , j ∈ { 1..n } , i 6 ≠ j }

Les reines doivent être sur des diagonales descendantes différentes :

→ Cdd = C i − L i 6≠ C j − L j /i ∈ {1..n } , j ∈ { 1..n } , i 6 ≠ j }

✓ Implémentation

Cette modélisation est basique et représente l'aspect naturel et intuitif de la solution. Elle présente un nombre élevé de variables. Une amélioration de ce modèle est possible, en réduisant le nombre de variables et en enlevant les informations superflues.

```
include "globals.mzn";
int:n;
% une seule reine par colonne
% nous pouvons concevoir cette contrainte dans la structure de données
% les reines sont des colonnes
array[1..n] of var 1..n: Q;
% pour chaque reine i, Q [i] nous donne la ligne
% pour les diagonales, on peut imaginer que les reines sont des points
% et l'échiquier est un système de coordonnées cartésien
% on veut éviter les pentes de + -1
constraint forall([
% la methode de la valeur abs
% abs(Q[j]-Q[i])!=abs(j-i)
% Notre proposition est plus optimale que la solution de la valeur absolue
 Q[i] != Q[j] /\ Q[i] + i != Q[j] + j /\ Q[i] - i != Q[j] - j
    |i,j in 1..n where i!=j
]);

solve satisfy;

output  [show(n)]++[" queens :\n"] ++
  [ if fix(Q[i]) = j then "Q " else ". " endif ++
    if j = n then "\n" else "" endif
  | i, j in 1..n
  ];
```

Figure 1:Modèle 1

```
Output
Compiling nQueens_N.mzn, additional arguments n=8;
Running nQueens_N.mzn
8 queens :
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
----------
Finished in 14msec
```

Figure 2:Résultat de compilation Minizinc - Modèle 1

II. M2 : un modèle basé sur des variables « domaine fini" » entières (FD) avec contraintes globales et cassant des symétries

La modélisation des problèmes complexes est facilitée par l'utilisation de contraintes hétérogènes agissant indépendamment sur de petits ensembles de variables. Toutefois, la détection locale des inconsistances affaiblit la réduction des domaines. Les contraintes globales corrigent partiellement ce comportement en utilisant l'information sémantique issue de raisonnements sur des sous-problèmes. Elles permettent généralement d'augmenter l'efficacité du filtrage ou de réduire les temps de calcul.

Par exemple, l'arc-consistance ne détecte pas l'inconsistance globale du CSP présenté.

En effet, l'inconsistance est issue des trois contraintes qui imposent que les trois variables prennent des valeurs distinctes alors que l'union de leurs domaines ne contient que deux valeurs.

La contrainte globale **allDifferent ($x_1, \ldots, x_n$)** qui impose que ses variables prennent des valeurs distinctes détecte cette inconsistance triviale.

Un second modèle classique pour le problème des reines utilise des contraintes globales allDifferent pour représenter les cliques d'inégalités binaires. On introduit généralement les variables auxiliaires x_i et y_i par le biais des contraintes de liaison $\mathbf{x_i = l_i - i}$ $\mathbf{y_i = l_i + i}$ $\mathbf{1 \leq i \leq n}$.

Les variables auxiliaires correspondent aux projections sur la première colonne de la reine i en suivant les diagonales.

La contrainte **allDifferent ($l_1, \ldots, l_n$)** impose que les reines soient sur des colonnes différentes alors que les contraintes **allDifferent ($x_1, \ldots, x_n$)** et **allDifferent ($y_1, \ldots, y_n$)** imposent que deux reines soient placées sur des diagonales différentes.

✓ **Implémentation**

```
include "globals.mzn";

int:n;
array[1..n] of var 1..n: Q;
% pour chaque reine i, Q [i] nous donne la ligne
% les lignes doivent être toutes différentes aussi c'est une contrainte globale

constraint alldifferent(Q);
constraint alldifferent(i in 1..n)(Q[i] + i);
constraint alldifferent(i in 1..n)(Q[i] - i);

% En cassant la symétrie
% cette contrainte supprime la symétrie verticale
constraint Q[1] <= n/2;

solve satisfy;
output  [show(n)]++[" queens :\n"] ++
  [ if fix(Q[i]) = j then "Q " else ". " endif ++
    if j = n then "\n" else "" endif
  | i, j in 1..n
  ];
```

Figure 3:Modèle 2

III. M3 : un modèle basé sur des variables booléennes et contraintes booléennes

Contrairement aux deux premiers modèles où les variables désignent les positions des reines sur l'échiquier, les variables du troisième modèle désignent les états des cases de l'échiquier. Une variable binaire est associée à chacune des n × n cases. X ij désigne la variable associée à la case située à la ligne i et à la colonne j. Elle peut prendre la valeur 0 si la case est vide ou la valeur 1 si la case est occupée par une reine. Le problème s'écrit alors :

✓ **Variables**

X = X ij, i ∈ {1..n } etj ∈ {1..n} ;

✓ **Domaines de définition**

D (X ij) = {0, 1}, i ∈ {1..n} , j ∈ {1..n} ;

✓ **Contraintes**

Il y a une reine par ligne :

→ Clig = ∑ nj = 1 X ij = 1/i ∈ {1..n}

Il y a une reine par colonne :

→ Ccol = ∑ i n = 1 X ij = 1/j ∈ {1..n}

Les reines doivent être sur des diagonales ascendantes différentes :

→ Cdm = i + j = k + l =⇒ X ij + X kl ≤ 1/i, j, k, l ∈ {1..n}

Les reines doivent être sur des diagonales descendantes différentes :
$\rightarrow$ Cdd = i − j = k − l $\Rightarrow$ X ij + X kl ≤ 1/i, j, k, l ∈ {1..n}

Cette modélisation permet de réduire considérablement l'espace de recherche puisque le domaine de définition des variables est binaire (0 ou 1). Cela permet au solveur d'accélérer la recherche de solutions.

Ce modèle énumère les affectations totales et vérifie leur consistance, c'est-à-dire qu'aucune contrainte n'est violée. En général, les affectations sont énumérées grâce à une recherche arborescente qui instancie itérativement les variables. Cet algorithme ne réveille les contraintes que lorsqu'une affectation est totale, mais considère par contre un nombre exponentiel d'affectations le rendant impraticable même pour des problèmes de petite taille. Des résultats de complexité ont montré que prouver la satisfiabilité d'un CSP était un problème NP-complet mais qu'exhiber une solution admissible ou optimale était un problèmeNP- difficile.

L'algorithme simple retour arrière (backtrack) étend progressivement l'affectation vide en instanciant une nouvelle variable à chaque étape. L'algorithme vérifie alors que l'affectation partielle étendue est consistante avec les contraintes du problème. Dans le cas contraire, la dernière instanciation faite est remise en cause et l'on effectue une nouvelle instanciation. De cette manière, l'algorithme construit un arbre de recherche dont les nœuds représentent les affectations partielles testées. Cet algorithme teste l'ensemble des affectations possibles de manière implicite, c'est-à-dire sans les générer toutes. Le nombre d'affectations considéré est ainsi considérablement réduit, mais en revanche, la consistance des contraintes est vérifiée à chaque affectation partielle.

- ✓ Implémentation

  - **Prédicat**

    S = sum (baord) renvoie la somme des éléments de board le long de la première dimension du tableau dont la taille n'est pas égale à 1. board est une matrice, alors sum (board) renvoie un vecteur de ligne contenant la somme de chaque colonne.

- **Contraintes**

1) On commence dans la colonne la plus à gauche

2) Si toutes les reines sont placées, on retourne vrai

3) On Essaye toutes les lignes de la colonne actuelle.

4) On fait ce qui suit pour chaque ligne essayée.

a) Si la reine peut être placée en toute sécurité dans cette rangée, on marque cette [ligne, colonne] comme faisant partie de la solution et on vérifie de manière récursive si placer la reine ici conduit à une solution.

b) Si placer la reine dans [ligne, colonne] mène à une solution, alors renvoie vrai.

c) Si placer la reine ne mène pas à une solution, on décroche cette [ligne, colonne] (retour en arrière) et on passe à l'étape (a) pour essayer d'autres lignes.

5) Si toutes les lignes ont été essayées et que rien n'a fonctionné, on retourne false pour déclencher le retour en arrière.

```
include "globals.mzn";
%-----------------------------------------------------------------------------%
% Parameters
int: n; % size
%-----------------------------------------------------------------------------%
% Variables
array[1..n,1..n] of var bool: board;
%-----------------------------------------------------------------------------%
%   x = bool2int(e) is equivalent to (e /\ x=1) \/ (not e /\ x=0).
predicate mul2dmor(array[int,int] of var bool: b, var int: m )=
  sum(i,j in 1..length(b))(bool2int(b[i,j]))==m;
% (b[i,j] /\ n=1) \/ (not b[i,j] /\ n=0))==m;

% Constraints
constraint forall(i, j in 1..n)(
  board[i,j] =  not (
    exists(j1 in 1..n where j1 != j)(board[i,j1])     %check row/col
  \/  exists(i1 in 1..n where i1 != i)(board[i1,j])   %check col/row
  \/  exists(k in 1..n-1)(                            %check diagonals
      board[i+k,j+k] \/ board[i-k,j+k]
      \/  board[i+k,j-k] \/ board[i-k,j-k]
    )
  )
);
constraint mul2dmor(board,n);                         %makes sure there are n queens

solve satisfy;
```

Figure 4:Modèle 3

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

Les symétries de ce problème sont des symétries géométriques. En effet l'échiquier reste inchangé par des rotations et des symétries axiales et centrales (il y en a 7, sans compter l'identité). Avec notre modélisation la symétrie axiale verticale (V) et la symétrie axiale horizontale (H) correspondent respectivement à une symétrie de valeur (puisqu'elle consiste à permuter les valeurs) et une symétrie de variable (puisqu'elle consiste à permuter les variables). La rotation est (R) est une symétrie plus compliquée puisque le symétrique dépend de la valeur et de la variable.

Alors que le problème de N reines n'est plus considéré comme un défi pour les résolveurs de contrainte, trouver toutes les solutions reste difficile. L'échiquier a 7 symétries de rotation et de miroir, à l'exclusion de l'identité. Pour éliminer les solutions symétriques, il suffit d'écrire 7 fonctions, chacune retournant une contrainte représentant l'équivalent symétrique de vars [i] = j. vars [i] = j correspond Qi = j, et n(Q) est une variable globale. Le fait que certaines symétries puissent encore exister après qu'une ou plusieurs affectations aient été effectuées doit à nouveau être pris en compte. Considérez la symétrie de 180° rotation et supposons que les variables sont affectées dans l'ordre Q1,Q2,Q3,....,Nous pourrions ajouter la contrainte Q1 < n+1- Qn ; pour éviter, dans la plupart des cas, de trouver deux solutions équivalentes, mais cela pourrait encore arriver si, par exemple, Q1=2 et Qn =n-1 ; pour les éliminer, nous pourrions ajouter une nouvelle contrainte conditionnelle: if Q1=n+1-Qn puis Q2<n+1-Qn-1. Nous pourrions avoir besoin de contraintes supplémentaires, si Q1=n+1-Qn et Q2=n+1-Qn-1 alors Q3< n+1-Qn-2, etc. Sûr d'éliminer complètement cette symétrie, il faudrait ajouter jusqu'à n/2 contraintes à la formulation, chacune avec une condition de plus que la précédente.

✓ Implémentation

```
include "lex_lesseq.mzn";

%-----------------------------------------------------------------------------%
% Parameters

int: n; % size

%-----------------------------------------------------------------------------%
% Variables

array[1..n,1..n] of var bool: filled;
array[1..n,1..n] of var 0..1: f =
     array2d(1..n,1..n,[bool2int(filled[i,j]) | i,j in 1..n]);
array[1..n] of var 0..n: q;
var 0..n: objective;

%-----------------------------------------------------------------------------%
% Predicates for symmetry breaking

predicate rot_sqr_sym(array[int,int] of var int: x) =
  let {
    int: n = card(index_set_1of2(x)),
        int: n2 = card(index_set_2of2(x)),
        int: l = n * n,
        array[1..l] of var int: y = [x[i,j] | i in index_set_1of2(x),
                                              j in index_set_2of2(x) ],
        array[1..4,1..l] of 1..l: p = array2d(1..4,1..l,
             [ if k == 1 then i*n + j - n else
               if k == 2 then (n - j)*n + i else
               if k == 3 then (n - i)*n + (n - j)+1 else
                              i*n + (n - j) - n + 1 endif endif endif
             | k in 1..4, i,j in 1..n ])
  } in assert(n == n2, "rot_sqr_sym: rotation symmetry applied to" ++
              " non square matrix",
    var_perm_sym(y,p));

predicate var_perm_sym(array[int] of var int: x, array[int,int] of int: p) =
   let { int: l = min(index_set_1of2(p)),
         int: u = max(index_set_1of2(p)),
         array[1..length(x)] of var int: y = [ x[i] | i in index_set(x)] } in
   forall (i in l..u, j in l..u where i != j) (
     var_perm_sym_pairwise(y, %% forces index 1..length(x)
                           [ p[i,k] | k in index_set_2of2(p)],
                           [ p[j,k] | k in index_set_2of2(p)]) );

predicate var_perm_sym_pairwise(array[int] of var int: x,
          array[int] of int: p1, array[int] of int: p2) =
   let { array[1..length(x)] of 1..length(x): invp1 =
             [ j | i,j in 1..length(x) where p1[j] = i ] } in
   lex_lesseq(x, [ x[p2[invp1[i]]] | i in 1..length(x) ]);

%-----------------------------------------------------------------------------%
% Constraints

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

Figure 5:Modèle 4

V. Résolution : Modèle 3 avec MiniSat (format DIMACS)

Pour cette résolution on a utilisé un programme qui permet de générer un fichier CNF d'un modèle à partir d'un fichier python, le but de ce programme n'est pas la résolution du N Queens lui-même, qui est délégué au SAT-Solver, mais sa formalisation comme un ensemble de clauses en forme normale conjonctive (CNF) utilisant le CNF-SAT Format DIMACS.

De cette façon, les problèmes pour un nombre variable de reines peuvent être commodément représentés pour le résoudre avec minisat ou même avec d'autres solveurs SAT qui gèrent le format CNF-SAT DIMACS.

✓ Implémentation

Ce programme prend un nombre des Reines « n » comme macro, qui représente la dimension du problème (dimension du nombre de reines à mettre). Ensuite, le programme écrit la spécification du problème au format DIMACS CNF et délègue sa résolution à Minisat.

Le programme génère les fichiers :

- « clauses.txt » : avec uniquement les clauses CNF-SAT DIMACS.
- « coding.cnf » : avec la spécification CNF-SAT DIMACS complète.
- « solution.txt » : avec la relecture de la solution.

```python
import sys,string,os,linecache

n = input("Enter n: ")
if n < 4:
  print "UNSATISFIABLE"
  exit()
count=0
filen="qin.cnf"
filem="qout.txt"
file1= open(filen,"w");
file2= open(filem,"w");

#generator rule
for i in range(0,n):
 str0='';
 for j in range(1,n+1):
  number=str(j+n*i);
  str0=str0+number+" "
 str0=str0+" 0\n"
 file1.write(str0);
 count+=1

#constraints on rows
for i in range(0,n):
 for j in range(1,n):
   number=j+n*i;
   for l in range(1,n-j+1):
    str0="-"+str(number)+" -"+str(number+l)+" 0\n"
    file1.write(str0);
    count+=1

#constraints on columns
for j in range(1,n+1):
 for i in range(0,n):
   number=j+n*i;
   for l in range(1,n-i):
    str0="-"+str(number)+" -"+str(number+n*l)+" 0\n"
    file1.write(str0);
    count+=1

#constraints on l->r diagonal
# part 1, upper bound triangle
for i in range(0,n-1):
 for j in range(i,n-1):
  number=j+1+n*i;
  for l in range(1,n-j):
    str0="-"+str(number)+" -"+str(number+l*(n+1))+" 0\n"
    file1.write(str0);
    count+=1

# part 2, lower bound triangle
for i in range(0,n-1):
 for j in range(0,i):
  number=j+1+n*i;
  for l in range(1,n-i):
   str0="-"+str(number)+" -"+str(number+l*(n+1))+" 0\n"
   file1.write(str0);
   count+=1
```

Figure 6:Modèle 3 en python

Figure 8:Compilation avec n = 8

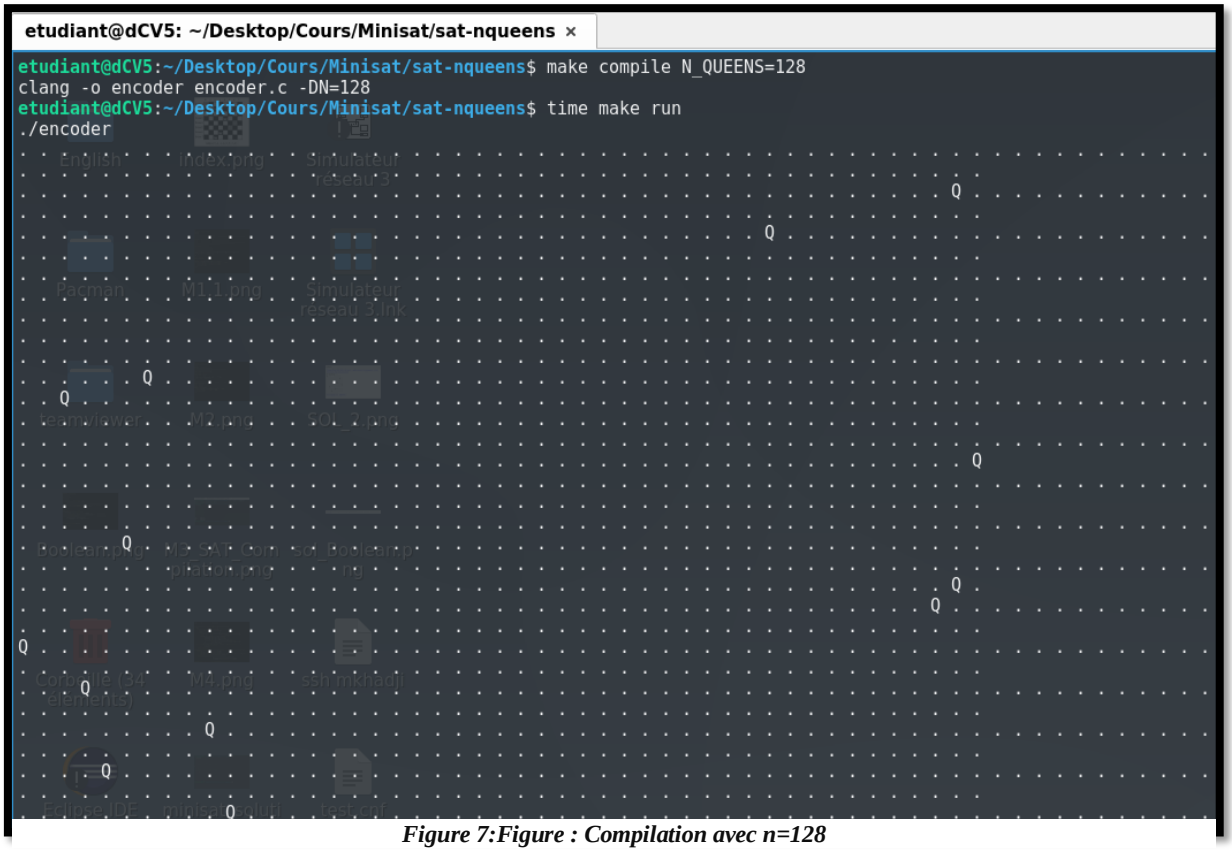


Figure 7:Figure : Compilation avec n=128

Figure 9:Le temps d'exécution n=128 à Minisat

VI. Résultat et Discussion

Solveur Gécode

Gecode est une boîte à outils C ++ open source pour le développement de systèmes et d'applications basés sur des contraintes. Gecode fournit un solveur de contraintes aux performances de pointe tout en étant modulaire et extensible.

Gecode est radicalement ouvert à la programmation : il peut être facilement interfacé avec d'autres systèmes. Il prend en charge la programmation de nouvelles contraintes, stratégies de branchement et moteurs de recherche. De nouveaux domaines variables peuvent être programmés avec le même niveau d'efficacité que les variables qui sont prédéfinies avec Gecode.

Solveur Chuffed

Chuffed est un solveur de contraintes basé sur la génération de clauses paresseuses. Ce type de solveur adapte les techniques de résolution SAT, telles que l'apprentissage des clauses de conflit, la propagation littérale surveillée et les heuristiques de recherche basées sur l'activité, et peut souvent être beaucoup plus rapide que les solveurs CP traditionnels.

SAT Solveur

Le problème SAT consiste à décider si une formule de la logique propositionnelle, présentée en forme normale conjonctive (CNF), est satisfiable ou non. Ce problème est NP-complet. Par conséquent, tous les algorithmes connus sont de complexité exponentielle. C'est le cas, en particulier, de l'algorithme naïf qui consiste à énumérer toutes les valuations des variables.

	Modèle 1	Modèle 2	Modèle 3	Modèle 4
Solveur Gecode	**n=8->13 ms** **n=128->2** m **24** s	**n=8-> 15 ms** **n=128->48 ms**	**n=8-> 21 ms** **n=128->** +∞	**n=8: 32ms** **n=128:** +∞
Solveur Chuffed	**n=8-> 128 ms** **n=128->** +∞	**n=8 -> 158 ms** **n=128-> 1s 128 ms**	**n=8-> 206 ms** **n=128->** +∞	**n=8: 192 ms** **n=128:** +∞
Solveur SAT			**n=8: 6 s** **n=128: 159 s**	**n=8:** **n=128:**

D'après les résultats de comparaison du temps d'exécution ou bien de compilation pour les 3 solveurs Gecode , Chuffed et SAT et en prenant la taille min n=8 reines et Max n=128 , on peut constater que le solveur Chuffed a pris plus de temps à la compilation pour chaque modèle , en ce qui concerne le nombre des Reines, on peut voir que pour n=128 les modèles booléen 3 et 4 avec et sans symétries, et le modèle simple basé sur des variables « domaine fini » entières sans contrainte globale ne génèrent aucune solution et il prend +∞ pour exécuter les modèles ,par contre le modèle basé sur des variables "domaine fini" entières avec contraintes globales et cassant des symétries se compile pour n=128 avec un temps d'exécution 1 sec et 128 ms , donc le modèle le plus optimisé et permet de résoudre une grande taille n élevé est le modèle avec contrainte globale et cassant les symétries .

le Solveur SAT permet aussi d'exécuter une taille n élevé et rendre des solutions boolean et il ne prend pas plus de temps par rapport le solveur Chuffed et GeCode, Alors, dans quelle mesure pouvons-nous améliorer un modèle profondeur pour le problème N-Reines est une bonne question ? Le fait de convertir un modèle boolean d'extension mzn ou bien flatzn à l'extension CNF, Reconsidérons le problème N-Reines comme un problème SAT. Le problème N-Reines peut être converti en un problème de satisfiabilité propositionnel (ou booléen) et résolu de manière très efficace avec un solveur SAT.

Exercice 2 : Garam

I. Présentation

Le Garam est un jeu de logique mathématique à base d'opérations simples.

Il suffit de remplir chaque case avec un seul chiffre de sorte que chaque ligne et chaque colonne forment une opération correcte.

Le résultat d'une opération verticale est un nombre à deux chiffres si deux cases suivent le symbole égal.

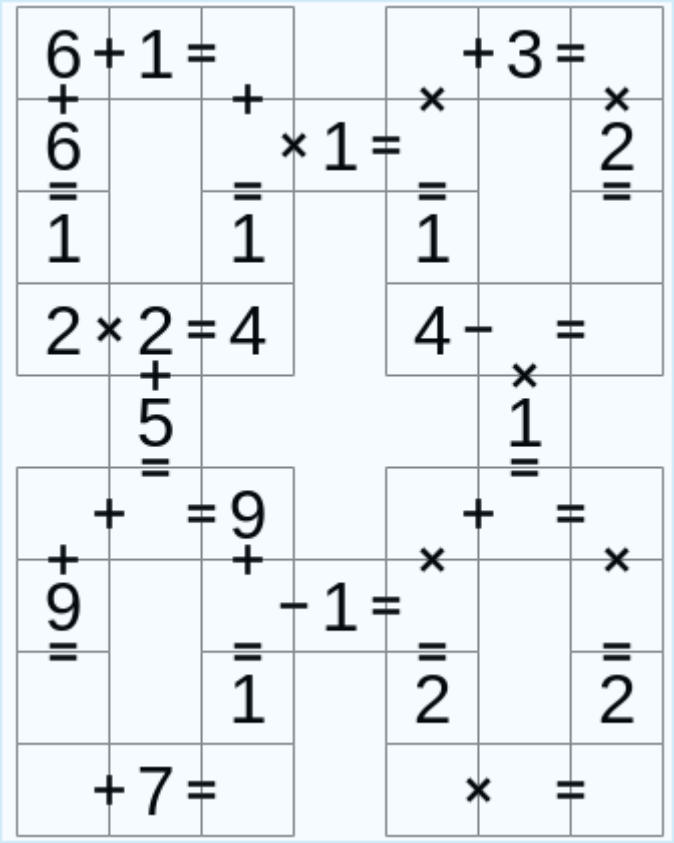


Figure 10: Garam

II. Modélisation

On a choisi de modéliser notre Garam sous forme des tableaux qui contiennent plusieurs cases représentants les chiffres et des tableaux représentants les opérations, donc on aura deux tableaux qui représentent les chiffres et deux tableaux qui représentent les opérations.

Le premier tableau de chiffres va représenter les chiffres des équations à 4 chiffres, et on a 8 équations donc on aura un tableau de taille 4*8, et le deuxième tableau va représenter les chiffres des équations à 3 chiffres on aura alors un tableau de taille 3*12 , et pour les tableaux des opérations on aura un premier tableau de 12 opérations pour les équations à 3 chiffres et un tableau de 8 opérations pour les équations à 4 chiffres, et l'idée est d'assurer que chaque ligne

de tableau avec l'opération soit vrai , et la première case du premier tableau et la première case du deuxième tableau doivent être égaux.

III. Prédicats et Contraintes

Pour cela on a défini un ensemble de contraintes et prédicats, un premier prédicat (op dans la figure 11) qui prend les chiffres et leur opération et qui va permettre de faire en sorte que toutes les valeurs par ligne soient correctes.

```
predicate op (var int:x, var int:y, var int:r, var 1..3 :o)=
  if     o=1 then r=x+y
  elseif o=2 then r=x-y
  elseif o=3 then r=x*y
  else false
  endif;
```

Figure 11:Prédicat 1 - Garam

Un deuxième prédicat (op2 dans la figure 12) qui aussi prends les chiffres et leur opération et qui va permettre de faire en sorte que toutes les valeurs par colonnes soient correctes.

```
predicate op2 (var int:x, var int:y, var int:r, var int:s, var 1..3 :o)=
  if     o=1 then r*10+s=x+y
  elseif o=2 then r*10+s=x-y
  elseif o=3 then r*10+s=x*y
  else false
  endif;
```

Figure 12: Prédicat 2 - Garam

Et on a défini une contrainte « toutes les opérations sont vraies » après avoir défini les prédicats des opérations cette contrainte va vérifier que toutes ces opérations sont vraies (figure 13).

```
constraint forall % . op . = .
  (i in 1..12)
  (op(number[i,1],number[i,2],number[i,3],operation[i]));
constraint forall % . op . = ..
  (i in 1..8)
  (op2(number2[i,1],number2[i,2],number2[i,3],number2[i,4],operation2[i]));
```

Figure 13: Contrainte 1 - Garam

Et puis une contrainte pour repérer les cases verticaux et horizontaux des rectangles, c'est à dire les équations doivent être bien positionner dans l'espace (figure 14).

```
constraint forall
  (i in 1..8)
  (
    if i mod 2 = 1 then
      number[i,1]=number2[i,1] /\ number[i,3]=number2[i+1,1]
    else
      number[i,3]=number2[i,4] /\ number[i,1]=number2[i-1,4]
    endif
  );
```

Figure 14: Contrainte 2 - Garam

Et ensuite une contrainte pour repérer les cases qui relient les rectangles (figure 15).

```
constraint number2[2,2] = number[9,1];
constraint number2[3,2] = number[9,3];

constraint number2[6,2] = number[10,1];
constraint number2[7,2] = number[10,3];

constraint number[2,2] = number[11,1];
constraint number[5,2] = number[11,3];

constraint number[4,2] = number[12,1];
constraint number[7,2] = number[12,3];
```

Figure 15: Contrainte 3 - Garam

Et au final on a défini une contrainte pour vérifier que la valeur de dizaine soit positive et différente de zéro (figure 16).

```
constraint forall(i in 1..8)(number2[i,3]>0);
```

Figure 16: Contrainte 4 - Garam

Exercice 3 : Rikudo

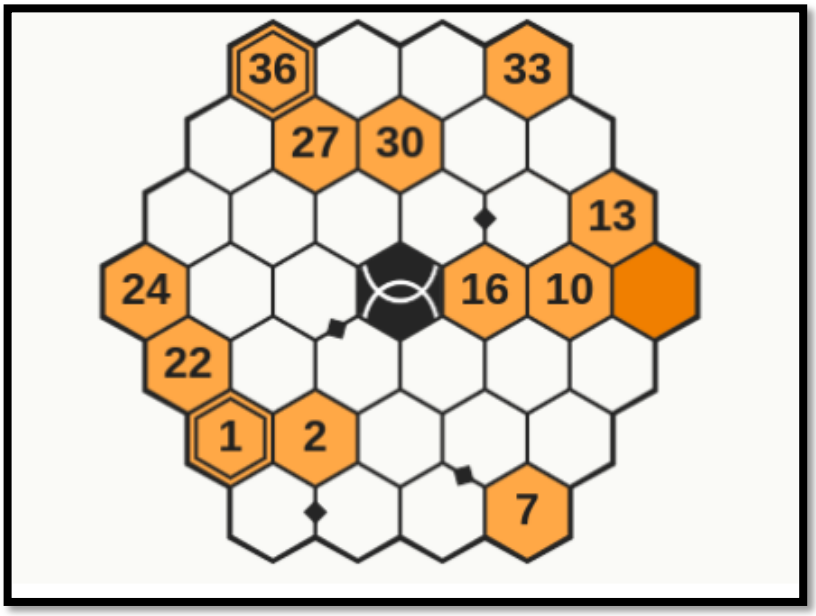


Figure 17: Rikudo

I. The Puzzle

Rikudo is a puzzle game which involves placing numbers in cells similarly to Sudoku or Garam but also has aspects of a labyrinth. The main rule for Rikudo is: you must place all different numbers in all the cells, and two sequential numbers are next to each other on the board.

The board for this puzzle is formed with hexagons, it can be constructed by placing a central hexagon and tiling successive layers of hexagons around it. The easiest problems have 3 layers for a total of 36 cells, while the hardest have 6 layers and a total of 126 cells. One could also consider problems of smaller and larger sizes.

The game's name is inspired by Naruto, a story from Japan. A character of the story is the Rikudo Sennin, or Sage of the Six Paths (SixWay Sage). This is possibly because from one hexagonal cell, you have 6 availiable paths. I am pleased to say, our model has it's own avatar, an antagonist in this story: known as Omochimaru the snake lord.

II. The Cube Coordinate System

Another Prerequesite before we go forward, is the Cube Coordinate system in our hexagonal grid.

2D Hexagons and 3D Cubes share a number of sides, and thanks to this parallel we can imagine our hexagons as cubes in 3D space, and give each hexagon a distinct and coherant set of (x,y,z) coordinates.

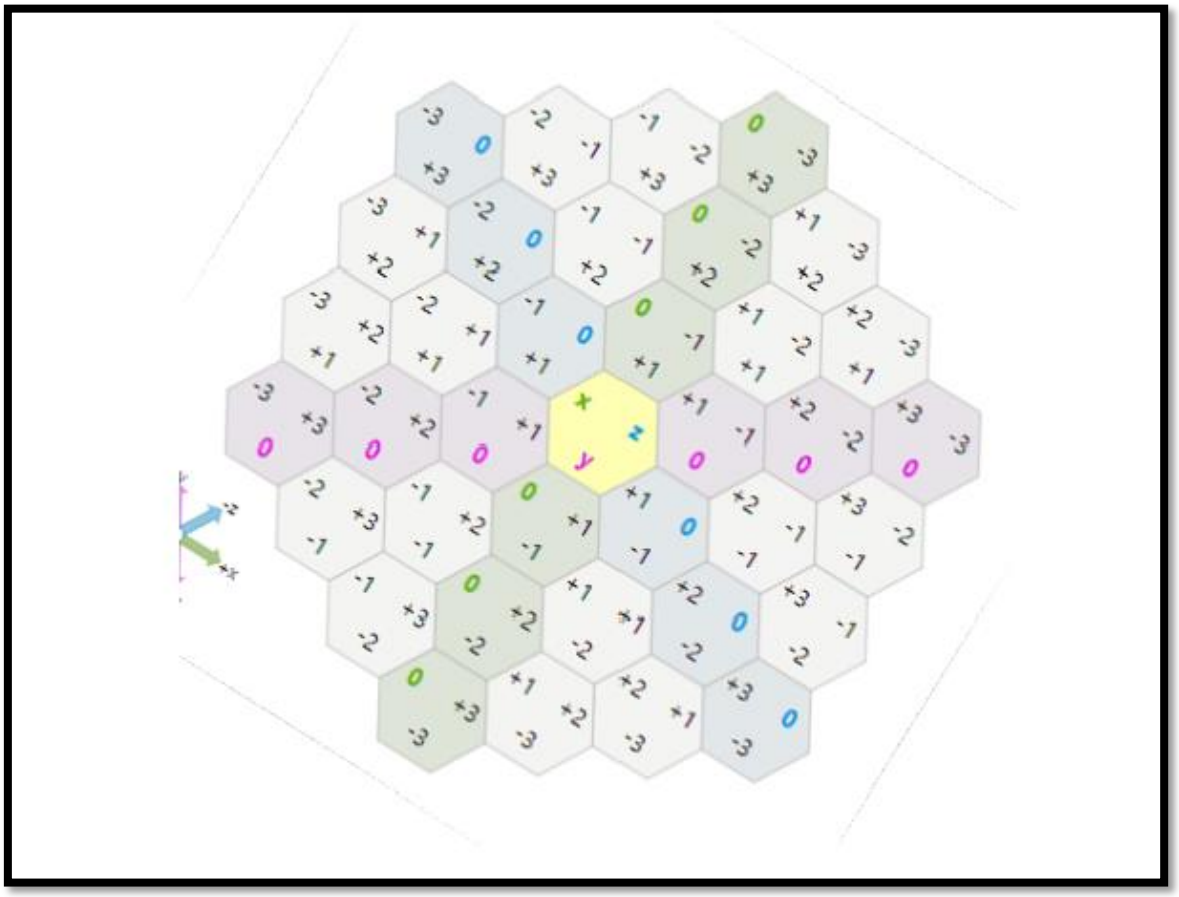


Figure 18: The Cube Coordinate System

For a set of coordinates to be valid the sum of x y z must equal = 0, and there is a simple distance function :

distance(i,j) = abs(ix - jx) + abs(iy - jy) + abs(iz – jz)

III. The Model

There are two main ways to see the Rikudo problem, in my opinion. Associating to each cell a numeber, or associating to each number a cell. Because of my knowledge of the cube coordinate system for Hexagon Grids, and it's distance function, I decided to explore a solution to use those rules as constraints on the cells associated to each number.

For all linear programing, we have variables and constraints. We have decided our variables are the cells associated to each number. Specifically, our variables are the x y z coordinates of each cell associated to a number of the series. This is how our model earns it's snake name: our snake is an array from 1 to n of set of coordinates.

```
6 array[1..n,1..3] of var int: snake; %array of coordinates
7 % successif numbers are NextTo each other on the board
8 % we can choose to make the coordinates the problem variables
9 % we can use cube coordinates
```

The constraints on these coordinates to find a solution for a Rikudo isn't at first obvious but with predicates our constraints are easier to understand.

1. Predicates

In this model we have two vague groups of predicates, those that make sure we're looking in the right direction and two that are more specific to the rules of Rikudo. But all predicates define rules of Cube Coordinates in a Hexgrid, and by using these predicates together we can model the Rikudo.

a. Valid Problem

ValidProblem is a predicate that makes sure the number of cells and the layers of the grid are coherent. All proposed Rikudos comme on pseudo-Hexagonal grids of hexagons, with a Layer to Cell ratio defined by :

```
17 predicate ValidProblem(var int: range, var int: numbers) =
18   sum(i in 1..range)(i*6) = numbers;
```

The number of cells and layers are also limits on our search space, and the size of our constraint loops.

b. In Range

inrange is an example of a predicate that uses one of those crucial numbers defined by ValidProblem: the max absolute value of any coordinate is the number of layers.

```
46 predicate inrange(var int: x) =
47   abs(x)<=r; %
```

c. Valid Coordinate

ValidCoord is simply a predicate that makes sure a set of coordinates is valid in the context of Cube Coordiantes (sum(xyz)=0), are within the given board (all coords in range) and because Rikudo doesn't allow placing in the middle, we don't allow (0,0,0)

```
49 predicate ValidCoord(array[1..3] of var int: xyz)=
50     sum(xyz) = 0
51 /\  forall(i in 1..3)(inrange(xyz[i]))
52 % 0,0,0 isn't valid
53 /\ let { array[1..3] of var int: zero = [0,0,0] } in Different(xyz,zero)
```

d. Next To and Different

Using the distance function of the Cube Coordinate system, we can define both NextTo and Different: nexto(a,b) => distance(a,b)=1 and different(a,b) => distance(a,b)!=0.

However the distance function uses the absolute value, which isn't solved for quickly. The Different predicate can also be expressed as "not all coordinates are equal", which is much faster to compute.

```
% NextTo
predicate NextTo(array[1..3] of var int: I,array[1..3] of var int: J) =
  let{%s use cube coordinates
    var int: ix = I[1], var int: jx = J[1],
    var int: iy = I[2], var int: jy = J[2],
    var int: iz = I[3], var int: jz = J[3]
  } in abs(ix - jx) + abs(iy - jy) + abs(iz - jz) = 2; % Hex Distance = 1

% Different
predicate Different(array[1..3] of var int: I,array[1..3] of var int: J) =
  let{%s use cube coordinates
    var int: ix = I[1], var int: jx = J[1],
    var int: iy = I[2], var int: jy = J[2],
    var int: iz = I[3], var int: jz = J[3]
  } in (ix != jx)  \/ (iy != jy) \/ (iz != jz); % a coords is different

% Different
%predicate Different(array[1..3] of var int: I,array[1..3] of var int: J) =
%  let{%s use cube coordinates
%    var int: ix = I[1], var int: jx = J[1],
%    var int: iy = I[2], var int: jy = J[2],
%    var int: iz = I[3], var int: jz = J[3]
%  } in abs(ix - jx) + abs(iy - jy) + abs(iz - jz) != 0; % Hex Distance != 0 =>
different
```

2. Constraints

We will constrain for some of these predicates to be true in general, and for some to be true for specific cases.

a. Valid Problem

First the overall problem must be valid.

```
% The Problem is Valid
constraint ValidProblem(r,n);
```

b. Valid Coordinates

All the coordinates of the snake must be valid.

```
% All Coordinates Are Valid
constraint forall (i in 1..n)(
  let {
    array[1..3] of var int: xyz = [snake[i,j] | j in 1..3]
  } in ValidCoord(xyz)
);
```

c. Next To

We want out snake to be continuous; If two indexes of the array are "next to" each other, then the associated coordiates must also be next to each other.

```
% All Numbers Are Next To Each Other
constraint forall (i,j in 1..n where j-i=1)( % successif numbers i and j
  let {
    array[1..3] of var int: I = [snake[i,x] | x in 1..3], % coords of i
    array[1..3] of var int: J = [snake[j,x] | x in 1..3], % coords of j
  } in NextTo(I,J) % NextTo each other on the board
);
```

d. All Different

We don't want our snake eating it's self, or going throught the same cell more than once, so all coordinates of the snake must be different.

```
% All Coordinates Are Different
constraint forall (i,j in 1..n where i<j)( %i<j to only check each couple once
           %we could not check neighbours
  let {
    array[1..3] of var int: I = [snake[i,x] | x in 1..3], % coords of i
    array[1..3] of var int: J = [snake[j,x] | x in 1..3], % coords of j
  } in Different(I,J) % Different to each other
);
```

e. Tunnels

Tunnels are another big aspect when it comes to constraints. Checking a tunnel is valid using existing predicates is easy: both extremities but be adjacent valid coordinates.

```
,,
% All Tunnels are Valid
constraint forall(s in 1..t)(
  let {
    array[1..3] of var int: A = [tunnels[s,1,x] | x in 1..3], % coords of A
    array[1..3] of var int: B = [tunnels[s,2,x] | x in 1..3], % coords of B
  } in ValidCoord(A) /\ ValidCoord(B) /\ NextTo(A,B)
);
```

Making sure the snake goes through the tunnels is a bit harder, but our model trys to simply say: if part of a snake is near a tunnel, then it goes through the tunnel.

```
% All Tunnels Are Used
% if in a sequence of 3 numbers the middle one is at the opening of a tunnel
% then either the one before or after is at the other opening of the tunnel

% we loop over the snake in chunks of 3, a "part of a snake"
constraint forall (i,j,k in 1..n where j-i=1 /\ k-j=1)( % successif numbers i, j and k
  let {
    array[1..3] of var int: I = [snake[i,x] | x in 1..3], % coords of i
    array[1..3] of var int: J = [snake[j,x] | x in 1..3], % coords of j
    array[1..3] of var int: K = [snake[k,x] | x in 1..3], % coords of k
  } in forall(s in 1..t)(
      let { %for each part of the snake we see if it is near any tunnels
        array[1..3] of var int: A = [tunnels[s,1,x] | x in 1..3], % coords of A
        array[1..3] of var int: B = [tunnels[s,2,x] | x in 1..3], % coords of B
      } in (
      if (Different(J,A) /\  Different(J,B)) then true else (
      % if a part of the snake is next to a tunnel it must go through
      % we know that if the middle J of the part is at a tunnel extremity
      % thus I or J must be at the other end of the tunnel
         (if ( not Different(J,A) ) then
            (not Different(I,B) \/ not Different(K,B) )else false endif
          ) \/ (
          if ( not Different(J,B) ) then
            (not Different(I,A) \/ not Different(K,A) ) else false endif)
      )endif
      )
  )
);
```

IV. Testing The Model

Now with our model fnding the correct solutions for any given rikudo, we can now ask how the solver fnds solutions to our model, and how difcult the different aspects of the problem are.

1. Search Space

A frst question to consider, is how does our model differenciate from a model which uses the alternative strategy of fnding a value for each cell.

In the broadest terms, the other strategy has n cells with n possible values.

A search space of n squared.

When expressing our search space, is it easier to use the radius, or number of layers, when calculating the search space.

number of cells = n = sum(i in 1..r)(i*6).

Thus the other model's search space can be expressed as :

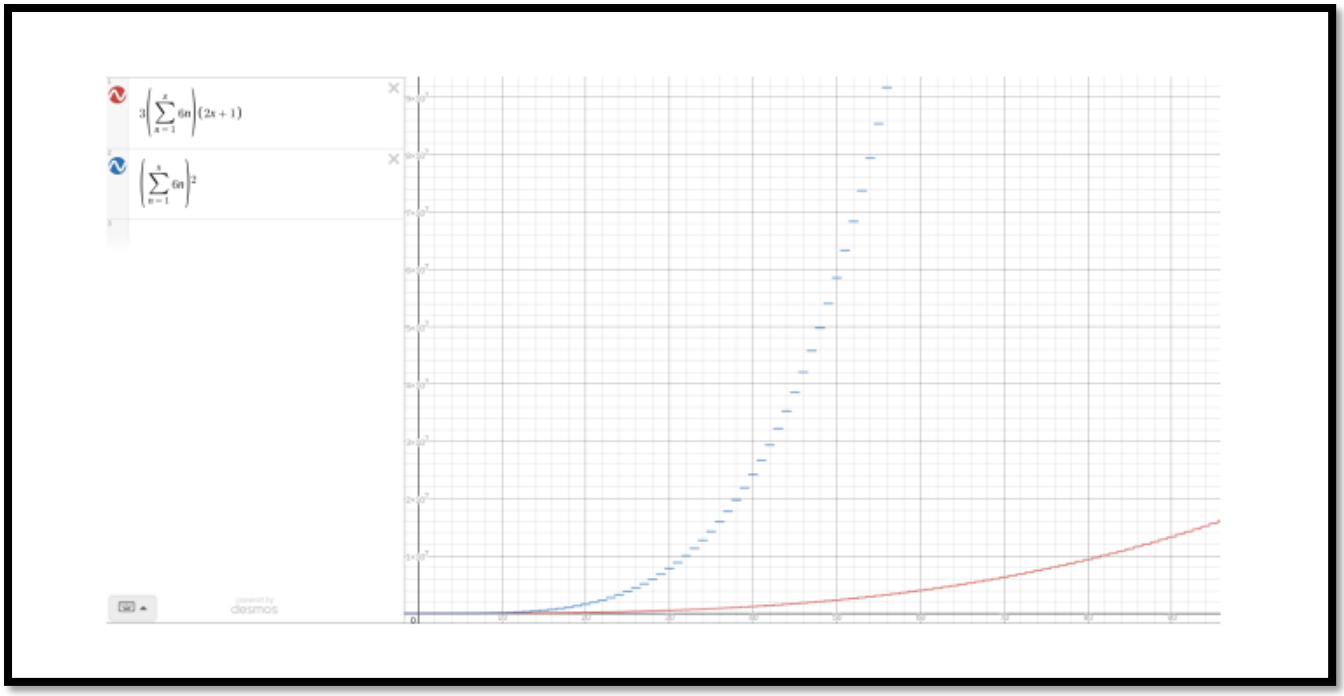
$$\left(\sum_{i=1}^{r} 6i\right)^2$$

The snake is n long, each section of the snake has three variables in -r..r.

Which translates to a search space of size :

(sum(i in 1..r)(i*6)) * 3 * (2r+1)

$$\left(\sum_{i=1}^{r} 6i\right)\left(3(2r+1)\right)$$



This means that on a broad level, the snake strategy starts off with a smaller space to search in. Further constraints will reduce the search space for both types of models, but the snake seems to have an advantage.

2. Solving For Size 3 and 4 : Easy(1/6), Average(2/6) and Hard(3/6) Difficulty

The Rikudo.fr website is the obvious source for instances of problems.

At first, we attempted to test problems of sizes 3 to 6. After initial testing of size 6 and then 5 problems, resulting in no solutions after 9 and 3 hours respectively, sizes above 4 were abandoned for this study.

Easy (size 3)	Average (size 4)	Hard (size 4)
230 ms	24 000 ms	No Solution after 2 Hours

3. Checking Solutions

An interesting aspect of a problem is: how long it takes to check a solution is correct. Sudoku is an example of a hard problem for which possible solutions are fast and easy to check. On the other hand, one can easily write a program that halts, but it is impossible to write a program that can determine whether any program halts or not. Because of this, it is interesting to verify how long the solver checks a valid and false solution in our model.

Easy (size 3)	Average (size 4)	Hard (size 4)
30 ms	40 ms	

4. Filling in the Gaps

The way you normally play Rikudo, is by filling in the gaps in a solution, so one way to test the model is to give it different types of gaps to fill. For easy versions of the problem, the player is given around a dozen values on the grid of 36. We can look for extreme scenarios with similar amounts of information.

The two extremes seem to be: one large gap in the middle and evenly spaced small gaps all along the snake. Real instances of the problem seem to be in between: randomly scattered gaps of varying sizes, separating small islands of solved cell sequences.

Intuitively, the size of the gaps is more important than the number of gaps.

And this is verifed by the two extreme examples.

Remove Multiples			Remove Middle				
2n	3n & 5n	2n & 5n	10 - 19	8 - 19	10 - 22	10 - 29	7 - 29
19/36	20/36	13/36	26/36	24/36	23/36	16/36	13/36
30 ms	40 ms	60 ms	50 ms	60 ms	1 s	39 s	300 s
Solution Differences			{}	{}	{10,11}	{10, 11, 26, 27, 28}	{10, 11, 23, 24, 25, 26, 27, 28}

5. Redundant Variables

Because of a constraint on our coordinates, inherited from cube coordinate system: for each cell, one of it's three coordinates is redundant.

Indeed, for a set of coordinates to be valid, their sum must equal zero.

Cube Coordinates for a hexgrid => x + y + z = 0

Thus, if we know x and y, we know z. The solver can find the missing z for a solved Rikudo, in the same time it checks a solution is valid (times are within margin of error, only milliseconds apart).

This has an implication on the size of our whole search space; So far we've assumed all 3 coordinates define the size, but we only really need to search using two coordinates. The search space is possibly 2/3 the size initially predicted.

V. Conclusions

A major beneft of modeling this way, we can change the shape of the rikudo board mainly be redefning the predicates and subsequently adapting the variables and constraints. If the board was made of titles squares for example: we could use a Cartesian coordinate system and distance function to implement our predicates, and adjust the snake and constraints to use sets of (x,y) as variables.

However there are most certainly optimisations : on the tips page of rikudo.fr they suggest some strategies to avoid running down the wrong path. Such heuristics are yet to be implemented.

Rikudo quickly becomes a very difficult problem. Despite the models ability to take on any size problem, you don’t get an answer in a reasonable amount of time past a certain threshold. The model can however verify a solution quickly.

Another reason this model is interesting, is because one can see the same problem as that of protein folding. A protein starts it's life a long sequence of chemicals, copied from DNA. To take action in an organism, a protein must take on a certain shape, with specific chemicals in specific locations. Similarly our snake must take on the shape of our board, go through the tunnels and some values must be in specific places. The tunnels and the shape of the board are analogous to the shape of the protein and it's active sites, while one can imagine numbering the chemicals in a protein and thus see the equivalence between a given number of a Rikudo board, and constraining the position of a chemical in the final shape of the protein.

Conclusion Générale

La programmation par contraintes permet principalement de résoudre des problèmes combinatoires et NP-complet.

- On déclare et le système le résout automatiquement.
- On se concentre sur le problème plutôt que sa méthode de résolution.
- Il existe de nombreux sous-paradigmes plus ou moins efficaces pourcertaines classes de problèmes (local search, MIP, SMT...).

Pour conclure, Ce projet était une bonne occasion pour découvrir et de s'informer sur la programmation par contraintes, et la modélisation des problèmes.

Références

[1] N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. InCP-AI-OR'05,volume 3524 ofLNCS, pages 64–78, 2005.

[2] N. Beldiceanu, X. Lorca, and P. Flener. Combining tree partitioning, prece-dence, and incomparability constraints.Constraints, 13(4), 2008.

[3] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future.Constraints, 12(1) :21–62,2007.

[4] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering al-gorithms from constraint checkers. InCP'04, volume 3258 ofLNCS, pages107–122, 2004.

[5] Ian P. Gent and Barbara Smith. Symmetry breaking in constraint programming. In W. Horn, editor, Proceedings of European Conference on Artificial Intelligence ECAI-2000, pages 599–603. IOS Press,2000.

[6] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.