Évaluer la difficulté d'une grille de sudoku à l'aide d'un modèle contraintes

François Laburthe, Guillaume Rochart, Narendra Jussien

Évaluer la difficulté d'une grille de sudoku à l'aide d'un modèle contraintes

François Laburthe[1] **Guillaume Rochart**[1] **Narendra Jussien**[2]

[1] e-lab – Bouygues SA – 1 av. Eugène Freyssinet
F-78061 St Quentin en Yvelines, France
[2] École des Mines de Nantes – LINA CNRS 2729 – 4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
{flaburthe, grochart}@bouygues.com narendra.jussien@emn.fr

Résumé

Le *sudoku* est un jeu de logique qui est devenu en quelques mois un phénomène de société en France. Il envahit les métros, les trains, les bus, les salles de cours et même le journal *Le Monde*. Grâce à ce jeu, le grand public est devenu le M. Jourdain de la Programmation Par Contraintes. En effet, l'intérêt de ce jeu pour montrer très rapidement et très simplement les principes premiers de la programmation par contraintes n'est plus à démontrer. De plus, la technologie contraintes est très performante pour modéliser à l'aide de quelques contraintes globales ce problème et le résoudre quasiment simplement par propagation. Par contre, la mesure de la difficulté d'une grille – qui laisse à désirer pour de nombreuses instances publiées actuellement – n'a pas encore été capturée de manière satisfaisante par un modèle *contraintes*. Une raison est qu'une telle mesure est totalement subjective car elle dépend de la façon dont un joueur aborde son instance. Dans cet article, nous montrons qu'il est possible de définir des modèles contraintes permettant de retrouver des combinaisons de règles utilisées par les joueurs. Ces modèles ouvrent la porte à une évaluation de la difficulté d'une instance par une approche purement contraintes et même de fournir des systèmes d'aide eux-aussi basés sur un telle approche.

1 Introduction

Est-il encore besoin de présenter le *sudoku* ? Ce jeu est un jeu de logique : sa grille carrée de 81 cases est divisée en 9 blocs (3×3) de 9 cases chacun (voir la figure 1). Des chiffres sont inscrits dans certaines cases. Il faut remplir les autres en utilisant les chiffres de 1 à 9, mais aucun chiffre ne doit apparaître deux fois dans la même ligne, ni dans la même colonne, ni dans le même bloc. Les instances sont sélectionnées de sorte qu'il n'existe qu'une unique solution. Il a été montré qu'il existait 6 670 903 752 021 072 936 960 grilles (complètes) différentes [2]. Si on tient compte des symétries intrinsèques de la grille (permutation des valeurs, des colonnes, des lignes...), ce nombre descend à 5 472 730 538 [5].

Ce jeu est un cas d'école pour la programmation par contraintes pour deux raisons : la première est que la discipline paraît être la technique idoine pour résoudre automatiquement les instances du *sudoku* (en effet, le modèle intrinsèque du problème est basé tout naturellement sur la contrainte `all-different` [3]); la deuxième, et peut-être finalement la plus importante, tient au fait que les techniques mises en œuvre par les joueurs humains pour résoudre leurs grilles fait appel à des principes de base de la programmation par contraintes : la réduction de domaine (*élimination de candidats* dans le vocabulaire *sudokumaniaque*), le raisonnement local (dans chaque zone – ligne, colonne ou bloc) et la propagation (partage de l'information par les domaines – les listes de candidats).

Une problématique liée au *sudoku* est la mesure de la difficulté des grilles (très souvent précisée dans les instances publiées dans la presse). La difficulté d'une grille (de *très facile* à *expert*) est un concept éminemment subjectif puisqu'il est censé rendre compte de la difficulté pour un joueur à remplir correctement sa grille. Contrairement à quelques idées reçues, cette difficulté ne dépend pas du nombre de cases pré-remplies dans l'instance. Une manière de plus en plus répandue de mesurer cette difficulté consiste à déterminer le niveau minimal de complexité des règles à appli-

5	8				4	2		
	2		6		8			
				7				
9				4				
					7	3	8	6
								4
			8					5
	3	6						9
		5		9			7	2

Figure 1: Une instance *difficile* de sudoku

quer pour résoudre l'instance. En effet, des collections de règles à appliquer (aux noms aussi évocateurs que *X-Wing* ou *Swordfish*) fleurissent sur internet [1]. Celles-ci sont très souvent présentées dans un ordre de difficulté (de compréhension, d'identification et d'application) croissant.

Mais, une telle manière de mesurer la difficulté laisse de côté la programmation par contraintes. Cela est assez dommageable. Nous chercherons donc dans cet article à proposer différents modèles du problème permettant de mesurer la difficulté dans un cadre *contraintes*.

2 Résolution à base de règles

Nous proposons une description semi-formelle des règles de résolution, telles qu'elles sont présentées sur [1]. Dans la suite de cet article, on notera C, L et B les colonnes, lignes et blocs de la matrice ; on notera (i, j, b) leurs indices respectifs. Chaque case $x[i, j]$ de la grille appartient à une ligne $L_i \in L$ (i-ème ligne[1]), une colonne $C_j \in C$ (j-ème colonne[2]), et un carré $B_b \in B$ (b-ème bloc[3]). On note $b_{i,j}$ l'indice du bloc contenant la case (i, j) et $a_{i,j}$ l'indice de cette case (i, j) au sein de ce bloc (c'est-à-dire que $B_{b_{i,j}}[a_{i,j}] = x[i, j]$). Dans l'expression des règles, on écrira $x[i, j] := v$ pour remplir effectivement la case dite avec la valeur v ; et on écrira $x[i, j] \neq v$ quand on est sûr que la valeur v est interdite pour la case.

Tout ensemble de cases de la grille est appelé une **zone**. Pour toute zone Z, on note $quoi(Z)$ l'ensemble des valeurs qui peuvent rentrer dans une des cases de la zone ; formellement :

$quoi(Z) = \{v \in 1..9 | \exists x[i_0, j_0] \in Z$ tel que:
$\forall j_1 \neq j_0, x[i_0, j_1] \neq v, \quad \forall i_1 \neq i_0, x[i_1, j_0] \neq v,$
$\forall (i_1, j_1) \neq (i_0, j_0)$ tel que
$b_{i_1,j_1} = b_{i_0,j_0}, x[i_1, j_1] \neq v\}$

[1]Du haut vers le bas.
[2]De la gauche vers la droite.
[3]De gauche à droite puis de haut en base.

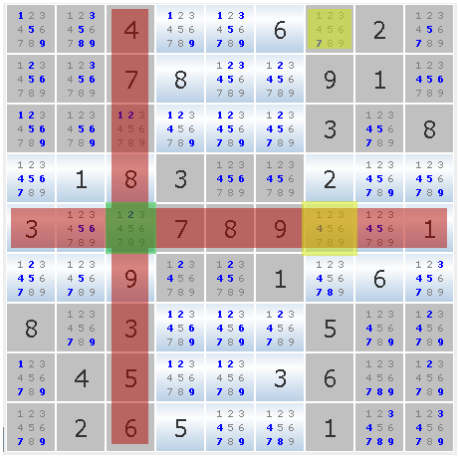


Figure 2: *Single Candidate* : les ligne et colonne rouges permettent de supprimer presque toutes les valeurs du domaine de l'intersection (toutes sauf la valeur 2) ; on peut donc en déduire que l'intersection aura la valeur 2. D'autres raisonnements du même type peuvent être utilisés pour les cases en jaunes.

Par ailleurs, pour tout ensemble de valeurs V, et tout zone Z, on note $ou(V, Z)$, le sous-ensemble des cases de la zone pouvant accueillir une des valeurs de V. Formellement :

$$ou(V, Z) = \{x[i, j] \in Z | quoi(\{x[i, j]\}) \cap V \neq \emptyset\}$$

Dans l'ensemble des figures[4], on représente à l'intérieur de chaque case, en grande police la valeur quand la case est remplie, et dans le cas contraire, en petite police, en bleu, les valeurs possibles, en gris, les valeurs impossibles par examen immédiat de la ligne, la colonne et le bloc. Enfin, on représente en rouge les valeurs qu'un examen plus approfondi permet d'éliminer.

L'ensemble des règles présentées par la suite s'appliquent aussi bien sur une ligne, une colonne ou un bloc (lorsqu'elles ne portent que sur une zone) ; ou bien des blocs vers les lignes, des blocs vers les colonnes, des lignes vers les blocs, ou encore des colonnes vers les blocs (lorsqu'elles portent sur une interaction de zones).

2.1 Single Candidate

Cette règle remplit une case (i_0, j_0) lorsque l'examen de sa ligne, de sa colonne et de son bloc interdisent toutes les valeurs sauf une, v_0. C'est-à-dire pour chacune des valeurs $v \neq v_0$, il existe une des zones C_{i_0}, L_{j_0} ou $B_{(i_0,j_0)}$ ayant une case contenant déjà la valeur v (voir la figure 2).

[4]Les illustrations proviennent de l'applet disponible sur `www.emn.fr/jussien/sudoku/jouer.html`

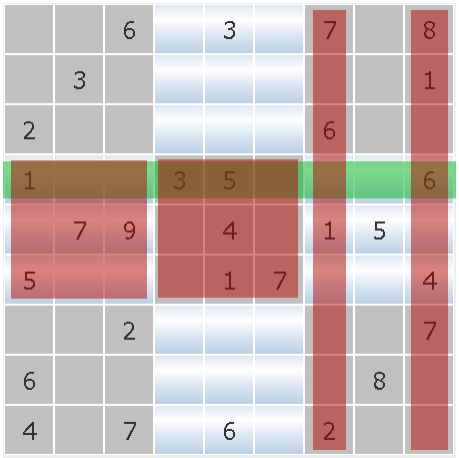


Figure 3: *Single Position* : la ligne verte représente une zone d'étude; les zones rouges représentent des placements interdits ; ne restant plus qu'un emplacement pour la valeur 7, on peut instancier la case à cette valeur.

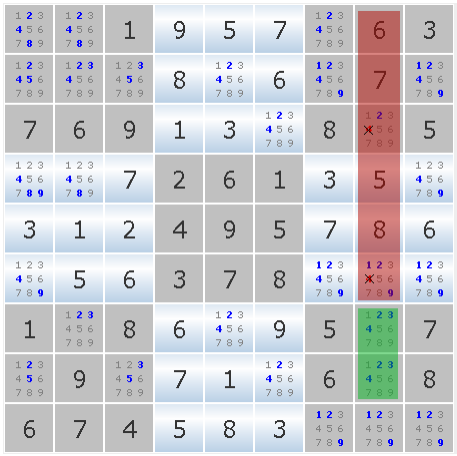


Figure 4: *Candidate Line* : Les cases en vert représentent les seules cases pouvant avoir la valeur 4 dans le bloc correspondant (bloc 9); appartenant toutes à la même colonne, on peut en déduire, que cette valeur n'apparaîtra pas dans cette même colonne dans les autres blocs.

singleCandidate$(x[i,j])$
si $\exists v, quoi(\{x[i,j]\}) = \{v\}$ alors $x[i,j] := v$

2.2 Single Position

Cette règle choisit une zone (disons une ligne L_i), et une valeur v qui n'est pas encore placée dans cette ligne et cherche si une seule place est disponible (voir figure 3).

singlePosition(L_i, v)
si $\exists j, ou(\{v\}, L_i) = \{x[i,j]\}$
alors $x[i,j] := v$

2.3 Candidate Lines

Cette règle examine un bloc B_b et une valeur v n'y ayant pas encore été placée. Lorsque toutes les cases pouvant possiblement accueillir cette valeur sont situées sur une même ligne L_i alors, on peut interdire de toutes les autres cases de la ligne (hors du bloc) cette valeur v.

candidateLine(B_b, v)
si $\exists i$ tel que $ou(\{v\}, B_b) \subseteq (L_i \cap B_b)$
alors $\forall x[i,j] \in L_i \setminus B_b, x[i,j] \neq v$

L'exemple de la figure 4 montre par exemple que la valeur 4 ne peut pas appartenir à la deuxième colonne des deux blocs supérieurs de droite (blocs 3 et 6).

2.4 Double pairs

Cette règle examine deux blocs contigus B_{b_1} et B_{b_2} et repère une valeur v , n'ayant dans chacun de ces blocs que deux places possibles : $x[i_1,j_1], x[i_2,j_2]$ dans B_{b_1}, et $x[i_3,j_3], x[i_4,j_4]$ dans B_{b_2}. Si, par ailleurs, ces cases sont situées dans une même colonne (par exemple si $i_1 = i_3$), alors on peut interdire la valeur v des autres cases de la colonne C_{i_1} (celles qui ne sont ni dans B_{b_1} ni dans B_{b_2}). La figure 5 illustre cette règle.

doublePair(B_{b_1}, B_{b_2}, v)
soit b_3, j_1, j_2, j_3 tels que
$B_{b_1} \cup B_{b_2} \cup B_{b_3} = C_{j_1} \cup C_{j_2} \cup C_{j_3}$
si $\exists i_1, i_2$ tels que
$ou(\{v\}, B_{b_1}) = \{x[i_1,j_1], x[i_1,j_2]\}$
$ou(\{v\}, B_{b_2}) = \{x[i_2,j_1], x[i_2,j_2]\}$
alors, $\forall x[i,j] \in B_{b_3} \setminus C_{j_3}, x[i,j] \neq v$

2.5 Multiple lines

Cette règle, généralisant la précédente, examine deux blocs B_{b_1} et B_{b_2} d'un même tiers vertical, et cherche une valeur v qui n'ait pas encore été placée dans ces deux blocs, mais dont les positions possibles, tant à l'intérieur de B_{b_1} que de B_{b_2} sont réparties sur deux des trois colonnes. Alors on peut affirmer qu'au sein du troisième bloc B_{b_3} de ce tiers vertical, la valeur v_0 sera nécessairement placée sur la troisième colonne. La figure 6 illustre cette règle.

multipleLines(B_{b_1}, B_{b_2}, v)
soit b_3, j_1, j_2, j_3 tels que

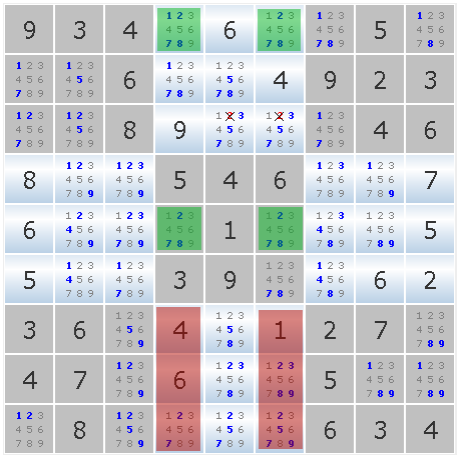


Figure 5: *Double Pairs* : Tout d'abord, en utilisant la règle *Candidate Line*, on remarque que les deux valeurs 2 en rouge dans le bloc central du haut (bloc 2) sont impossibles. Il ne reste donc que deux positions possibles pour la valeur 2 dans ce bloc et deux positions dans le bloc en-dessous (bloc 5) ; s'agissant des même colonnes dans les deux cas, on peut en déduire que les deux blocs devront se partager ces deux colonnes pour cette valeur et qu'elle ne peut donc apparaître dans ces mêmes colonnes dans le bloc central du bas (bloc 8). C'est ce qui est représenté par les zones rouges dans ce bloc qui permettent par exemple d'instancier une case à 7.

$$B_{b_1} \cup B_{b_2} \cup B_{b_3} = C_{j_1} \cup C_{j_2} \cup C_{j_3}$$
si $ou(\{v\}, B_{b_1}) \subseteq C_{j_1} \cup C_{j_2}$
et si $ou(\{v\}, B_{b_2}) \subseteq C_{j_1} \cup C_{j_2}$
alors $\forall x[i,j] \in B_{b_3} \backslash C_{j_3}, x[i,j] \neq v$

2.6 Naked pairs

Cette règle examine une ligne et cherche s'il existe deux cases pour lesquelles seulement deux valeurs v_1 et v_2 sont possibles. Si c'est le cas, v_1, comme v_2 peuvent être interdites pour l'ensemble des autres cases de la ligne (hormis les deux repérées). La figure 7 illustre cette règle.

nakedPair(L_i)
si $\exists j_1, j_2, v_1, v_2$ tels que
$quoi(L_i \cap (C_{j_1} \cup C_{j_2})) = \{v_1, v_2\}$
alors $\forall j, j \neq j_1, j \neq j_2, x[i,j] \notin \{v_1, v_2\}$

Cette règle se généralise aisément à k valeurs, par exemple, pour trois valeurs, on cherchera trois cases, pour lesquelles il n'existe aucune autre valeur possible en dehors de trois valeurs v_1, v_2 et v_3. Ce qui s'exprime par la règle Naked Tuple :

nakedTuple(L_i, k)
si $\exists j_1, ..., j_k, v_1, ..., v_k$ tels que
$quoi(L_i \cap (C_{j_1} \cup ... \cup C_{j_k})) = \{v_1, ..., v_k\}$

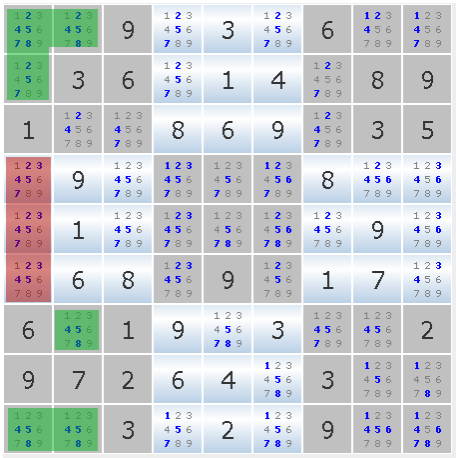


Figure 6: *Multiple Lines* : Considérons les blocs de gauche; celui du haut (bloc 1) ne peut avoir la valeur 5 que dans les deux première colonnes ; de même pour le bloc du bas (bloc 7) ; ce qui signifie que le bloc du milieu (bloc 4) ne doit avoir de 5 que dans la troisième colonne ce qui supprime les 5 de la première colonne (en rouge).

alors $\forall j, j \notin \{j_1, ..., j_k\}, x[i,j] \notin \{v_1, ..., v_k\}$

2.7 Hidden pairs

Cette règle examine une ligne L_i et cherche s'il existe deux valeurs v_1 et v_2 pour lesquelles seulement deux cases sont possibles. Si c'est le cas, toutes les autres valeurs sont interdites pour ces deux cases. La figure 8 illustre cette règle.

hiddenPair(L_i)
si $\exists v_1, v_2, j_1, j_2$ tels que
$ou(\{v_1, v_2\}, L_i) = L_i \cap (C_{j_1} \cup C_{j_2})$
alors $\forall j, j \neq j_1, j \neq j_2, x[i,j] \notin \{v_1, v_2\}$

La généralisation à plus de deux valeurs donne :

hiddenTuple(L_i, k)
si $\exists v_1, ..., v_k, j_1, ..., j_k$ tels que
$ou(\{v_1, ..., v_k\}, L_i) = L_i \cap (C_{j_1} \cup ... \cup c_{j_k})$
alors $\forall j, j \notin \{j_1, ..., j_k\}, x[i,j] \notin \{v_1, ..., v_k\}$

Lemme : Soit L_i une ligne contenant encore p cases non remplies, les règles **nakedTuple**(L_i,n) et **hiddenTuple**$(L_i,p-n)$ effectuent les mêmes inférences.
Preuve : ces deux règles effectuent une partition des cases de la ligne :

- *un ensemble E de $9-p$ cases remplies ;*
- *un ensemble F de n cases qui globalement ne peuvent être remplies que par n valeurs ;*

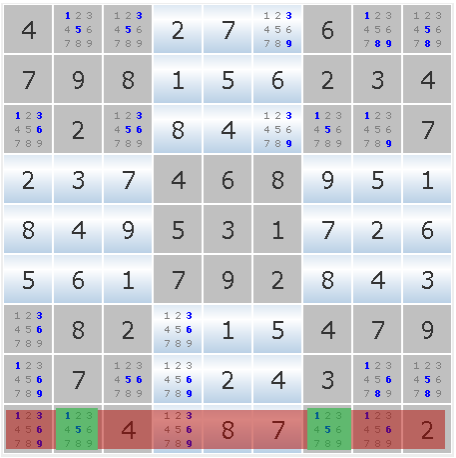


Figure 7: *Naked Pairs* : Deux cases de la ligne du bas ne peuvent être instanciées qu'à 1 et 5 ; ceci signifie que ces valeurs n'apparaîtront que dans ces deux cases ; elles peuvent être donc retirées du reste de la ligne, permettant ainsi d'instancier une case à 6.

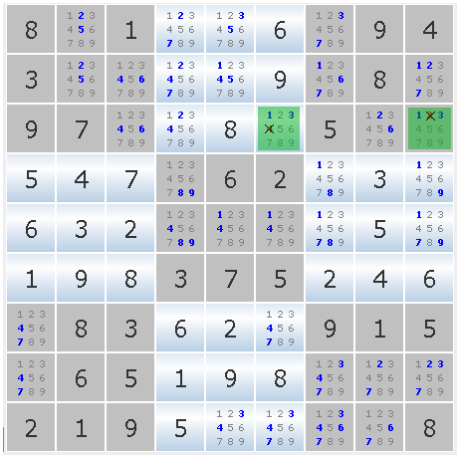


Figure 8: *Hidden Pairs* : Dans la troisième ligne, seule deux cases peuvent être instanciées à 1 et 3; cela signifie que ces deux cases ne peuvent être à instanciées qu'à ces deux seules valeurs; on peut donc retirer les valeurs 4 et 2 des domaines respectifs de ces deux cases.

- *et son complémentaire, un ensemble G de $p-n$ cases qui globalement contiennent $p-n$ valeurs ne pouvant pas être mises ailleurs que dans G.*

Les règles Naked sur F et Hidden sur G interdisent toutes aux cases de G de recevoir une des valeurs considérées par les cases de F.

Par exemple, si on considère la ligne 9 dans la figure 7, le raisonnement HiddenTuple sur les valeurs 1, 3 et 9 (qui n'ont que trois colonnes disponibles : 1, 4 et 8) aboutit aux mêmes conclusions que le raisonnement NakedPair sur les colonnes 2 et 7 (qui n'ont que deux valeurs possibles : 1 et 5). De même, sur la troisième ligne de la figure 8, HiddenPair sur les valeurs 1 et 3 aboutit aux mêmes conclusions que NakedTuple sur les colonnes 3, 4 et 8.

2.8 X-Wing et Swordfish

Ces deux règles sont similaires ; elles consistent à trouver une valeur v telle que, sur plusieurs lignes, seules deux cases peuvent avoir cette valeur et que ces cases ne contiennent à leur tour que deux valeurs. De plus, il est nécessaire que l'on puisse trouver un circuit passant par ces cases en suivant alternativement une ligne et une colonne[5]. Dans le cas simple où on ne considère que deux lignes, il s'agit d'un carré comme le montre la figure 9. Dans le cas où l'on considère trois lignes, on obtiendrait, par exemple un L.

Dès lors, on peut remarquer que dans de telles configurations, on peut en déduire que la valeur v apparaîtra dans chacune des colonnes de ces cases. On peut donc en déduire que la valeur v peut être retirée des autres cases de ces colonnes. La figure 9 illustre cette règle.

X-Wing(v)
si $\exists i_1, i_2, j_1, j_2$ tels que
$ou(\{v\}, L_{i_1}) = L_{i_1} \cap (C_{j_1} \cup C_{j_2})$
et $ou(\{v\}, L_{i_2}) = L_{i_2} \cap (C_{j_1} \cup C_{j_2})$
alors $\forall i, i \neq i_1, i \neq i_2, x[i, j_1] \neq v, x[i, j_2] \neq v$

Swordfish(v, k)
si $\exists i_1, ..., i_k, j_1, ..., j_k$ tels que
$ou(\{v\}, L_{i_1}) = L_{i_1} \cap (C_{j_1} \cup C_{j_2})$
$ou(\{v\}, L_{i_2}) = L_{i_2} \cap (C_{j_2} \cup C_{j_3})$
...
$ou(\{v\}, L_{i_k}) = L_{i_k} \cap (C_{j_k} \cup C_{j_1})$
alors $\forall i, i \notin \{i_1, ..., i_k\}$ et $\forall j \in \{j_1, ..., j_k\}$
$x[i, j] \neq v$

2.9 Forcing chains

La dernière règle que nous évoquerons ici est la règle *Forcing chains*, qui consiste à essayer une valeur dans une case de sorte à vérifier si ce choix amène à une contradiction ou non (si c'est le cas, la valeur n'est pas correcte, sinon on ne peut rien déduire). Il s'agit de la technique utilisée par exemple dans la singleton-cohérence. S'agissant, dans une certaine mesure, d'un début de recherche, nous n'étudierons donc pas les règles qui suivent sur [1] car elles ne semblent pas applicables comme techniques de filtrage.

[5] Le raisonnement est valide si l'on passe aussi par un bloc mais le voir *à la main* est nettement plus ardu.

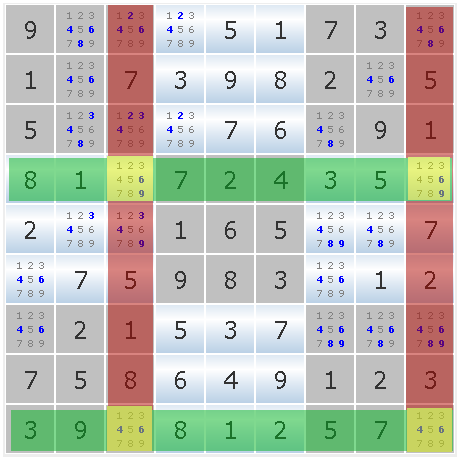


Figure 9: *X-Wing* : On considère ici les occurrences de la valeur 6 dans la quatrième et dans la dernière ligne; ces domaines formant deux paires identiques, on peut remarquer que si l'on choisit la valeur 6 pour un des coins du carré ainsi formé, le coin opposé sera forcément instancié à 6 aussi; on peut donc en déduire que la valeur 6 peut être retirée des autres cases des colonnes.

3 Résolution par des contraintes

Ces règles ont été exprimées dans un langage compréhensible par n'importe qui (enfin presque !) mais ça et là interviennent différent concepts fortement liés à la programmation par contraintes. Nous proposons donc maintenant une série de modèles sous forme de CSP permettant de résoudre une grille de sudoku.

Un CSP $M = (V, D, C)$ est défini par un ensemble de variables V, leurs domaines D et des contraintes C. Une affectation de valeurs aux variables σ est une application associant à chaque variable v_i de V une valeur dans D_i. Une affectation est une solution si pour toute contrainte $c \in C$, la substitution des variables par leurs valeurs se réduit en une contrainte $c[v \leftarrow \sigma(v)]$ vraie. L'ensemble des solutions d'un CSP M est noté $sol(M)$. La projection d'une affectation σ sur un sous-ensemble de variables W est notée $W \downarrow \sigma$. Étant donnés deux modèles M_i et M_j, on notera $M_i \succcurlyeq M_j$ lorsque le modèle M_i est un renforcement de M_j, c'est-à-dire lorsque :

$$var(M_i) \subset var(M_j)$$

$$sol(M_j) = sol(M_i) \downarrow var(M_j)$$

Par ailleurs, soit M un CSP, on fera référence aux niveaux de propagation suivants: AC pour l'arc cohérence, k-AC pour la k-cohérence (cohérence sur des chemins d'au plus k contraintes), et SAC pour la singleton arc-cohérence. Pour un modèle M, on notera ainsi $AC(M)$ l'état des domaines de M après calcul du point fixe associé à l'AC.

Dans la suite, nous présentons différents niveaux de raisonnement pour le sudoku et basés sur la programmation par contraintes. Chaque niveau de difficulté d'un problème sera défini comme l'ensemble des grilles de sudoku qui sont entièrement résolues par calcul d'arc cohérence pour un des modèles donnés.

3.1 Raisonnements locaux

Les premières règles présentées en section 2 sont très simples. Il est ainsi possible de proposer des modèles CSP uniquement basés sur des raisonnements locaux pour les modéliser.

3.1.1 Modèle PRIMAL

Le premier modèle, probablement le plus simple, introduit une variable par case et des contraintes de diségalités binaires entre variables associées à des cases d'une même zone.

Variables On associe à chaque case une variable : $\forall i, j, x[i, j] \in \{1, \ldots, 9\}$.

Contraintes On ajoute les contraintes suivantes au problème :
sur les lignes :
$\forall i, \forall j_1 \neq j_2, x[i, j_1] \neq x[i, j_2]$
sur les colonnes :
$\forall i, \forall i_1 \neq i_2, x[i_1, j] \neq x[i_2, j]$
sur les blocs :
$\forall b, \forall (i_1, j_1), (i_2, j_2) \in B_b, x[i_1, j_1] \neq x[i_2, j_2]$

Lien avec les règles Le calcul de l'arc-cohérence sur ce modèle, que l'on notera AC(PRIMAL), implémente *Single Candidate*. En effet, on peut voir, que les figures proposées en section 2 notent sur chaque case de la grille son domaine de valeurs et que les inférences de *Single Candidate* correspondent à la propagation des contraintes binaires de différence.

3.1.2 Modèle DUAL

Le second modèle que nous proposons prend un point de vue dual cherchant à répartir des valeurs dans des blocs plutôt que d'associer des valeurs à des cases. On associe à chaque valeur sa place au sein d'un bloc ; par exemple $idxL(j, c)$ représente au sein de la j-ème ligne, l'indice de la colonne où l'on trouvera la valeur v.

Variables On associe à chaque zone (ligne, colonne ou bloc), des variables indiquant l'indice correspondant à la case contenant chacune des valeurs possibles (1 à 9) :

$\forall i, j, x[i, j] \in \{1, \ldots, 9\}$
$\forall L_i \in L, idxL(i, c) \in \{1, \ldots, 9\}$
$\forall C_j \in C, idxC(j, c) \in \{1, \ldots, 9\}$
$\forall B_b \in L, idxB(b, c) \in \{1, \ldots, 9\}$

Contraintes On ajoute tout d'abord des contraintes de liaison entre les deux modèles :
- une valeur v se trouvant à l'indice j dans une colonne C_i remplit la case $x[i, j]$;
  $\forall i, j, v, idxC(i, v) = j \Leftrightarrow x[i, j] = v$
- similairement, une valeur v se trouvant à l'indice i dans une ligne L_j remplit la case $x[i, j]$;
  $\forall i, j, v, idxL(j, v) = i \Leftrightarrow x[i, j] = v$
- enfin, une valeur v se trouvant à l'indice $a_{i,j}$ dans un bloc $B_{b_{i,j}}$ remplit la case $x[i, j]$;
  $\forall i, j, v, idxB(b_{i,j}, v) = a_{i,j} \Leftrightarrow x[i, j] = v$

Puis les contraintes propres à ce modèle, indiquant que deux valeurs différentes v_1 et v_2 se trouvent nécessairement à des indices différents au sein
- d'une colonne C_i;
  $\forall C_j \in C, idxC(j, v_1) \neq idxC(j, v_2)$:
- d'une ligne L_j;
  $\forall L_i \in L, idxL(i, v_1) \neq idxL(i, v_2)$:
- d'un bloc B_b.
  $\forall B_b \in B, idxB(b, v_1) \neq idxB(b, v_2)$:

Lien avec les règles Le calcul de l'arc-cohérence sur ce modèle, que l'on notera AC(DUAL) calcule le même résultat (les mêmes domaines) que l'application de la règle *Single Position*. En effet, puisque l'on associe à chaque ligne, la position de chaque valeur au sein de cette ligne, quand une valeur v n'a qu'une position possible dans cette ligne L_j, la variable $idxL(j, v)$ sera instanciée, et par propagation des contraintes de liaison, la variable $x[i, j]$ sera instanciée aussi.

L'union des deux modèles, PRIMAL ∪ DUAL, combine les raisonnements suivant les deux vues. L'arc-cohérence, AC(PRIMAL ∪ DUAL), calcule le même résultat que l'application des deux règles *SinglePosition* et *SingleCandidate*.

Les instances de sudoku pour lesquelles AC(PRIMAL ∪ DUAL) suffit à instancier la grille (rappelons qu'une instance de sudoku comporte toujours une et une seule solution) sont appelées *faciles*.

3.1.3 Modèle ABSTRACT

Ce modèle ajoute de nouvelles variables qui modélisent pour une ligne (resp. une colonne, ou un bloc) et une valeur, non plus l'indice de la case dans laquelle tombe la valeur (9 possibles), mais simplement le bloc (respectivement colonne ou ligne) dans lequel tombe cette case (3 possibles). En quelques sortes il s'agit d'une expression du modèle de départ, avec des domaines hiérarchiques, introduisant un niveau abstrait de valeurs (les séquences de trois cases contigues et d'un même bloc).

Variables $bidxL(i, v) \in \{b \mid B_b \cap L_i \neq \emptyset\}$
$bidxC(j, v) \in \{b \mid B_b \cap C_j \neq \emptyset\}$
$lidxB(b, v) \in \{i \mid B_b \cap L_i \neq \emptyset\}$
$cidxB(b, v) \in \{j \mid B_b \cap C_j \neq \emptyset\}$

Contraintes On pose deux familles de contraintes : des contraintes de liaison entre les variables du modèle DUAL et celles de ABSTRACT :
$idxL(i, v) = j \Rightarrow bidxL(i, v) = b_{i,j}$;
$idxC(j, v) = i \Rightarrow bidxC(j, v) = b_{i,j}$;
$idxB(b_{i,j}, v) = a_{i,j} \Rightarrow lidxB(b_{i,j}, v) = i$;
$idxB(b_{i,j}, v) = a_{i,j} \Rightarrow cidxB(b_{i,j}, v) = j$;
et les contraintes propres du modèle qui sont des contraintes de différence entre ces nouvelles variables :
pour i_1, i_2 deux lignes du même tiers horizontal,
  $bidxL(i_1, v) \neq bidxL(i_2, v)$;
pour j_1, j_2 deux colonnes du même tiers vertical,
  $bidxC(j_1, v) \neq bidxC(j_2, v)$;
pour b_1, b_2 deux blocs du même tiers horizontal,
  $lidxB(b_1, v) \neq lidxB(b_2, v)$;
pour b_1, b_2 deux blocs du même tiers vertical,
  $cidxB(b_1, v) \neq cidxB(b_2, v)$;

Lien avec les règles L'arc-cohérence sur ce modèle AC(PRIMAL ∪ DUAL ∪ ABSTRACT) calcule le même résultat que l'application des règles *SinglePosition*, *SingleCandidate* et *CandidateLine*. Par exemple, dans la figure 4, la propagation de la contrainte de liaison sur le bloc B_9 (en bas à droite) et la valeur 4 amène à instancier la variable $cidxB(9, 4)$ à 8, indiquant que la valeur 4 dans ce bloc est nécessairement située sur la colonne du milieu. La propagation des contraintes de différence amène $cidxB(3, 4) \neq 8$, c'est à dire qu'à l'intérieur du bloc en haut à droite, le 4 ne pourra pas être placé sur cette colonne du milieu. Enfin, la propagation des contraintes de liaison amène $idxB(3, 4) \neq 8$, et $x[3, 8] \neq 4$ pour interdire le 4 dans la case en bas au milieu du bloc en haut à droite.

3.2 Modèles avec all-different

Nous montrons dans cette section comment sur les 3 modèles présentés précédemment, le remplacement des contraintes de différences par des contraintes globales de différence (`all-different`) permet d'effectuer plus d'inférences sur les grilles de sudoku, et de définir ainsi des niveaux de difficulté supérieurs

3.2.1 3-ABSTRACT

Ce modèle est construit à partir du modèle ABSTRACT, en ajoutant des contraintes globales de différence entre les nouvelles variables de ce modèle :
- pour $C_{j_1}, C_{j_2}, C_{j_3}$ trois colonnes du même tiers vertical,
`all-different`$(bidxC(j_1, v), bidxC(j_2, v), bidxC(j_3, v))$
- pour $L_{i_1}, L_{i_2}, L_{i_3}$ trois lignes du même tiers horizontal,
`all-different`$(bidxL(i_1, v), bidxL(i_2, v), bidxL(i_3, v))$
- pour $B_{b_1}, B_{b_2}, B_{b_3}$ trois blocs du même tiers horizontal,
`all-different`$(lidxB(b_1, v), lidxB(b_2, v), lidxB(b_3, v))$
- pour $B_{b_1}, B_{b_2}, B_{b_3}$ trois blocs du même tiers vertical,
`all-different`$(cidxB(b_1, v), cidxB(b_2, v), cidxB(b_3, v))$

Par exemple, sur le premier tiers vertical (les blocs 1, 4, 7), on propage globalement les trois inégalités entre variables $cidxB(1, v)$, $cidxB(4, v)$ et $cidxB(7, v)$, ce pour toute valeur v.

Lien avec les règles L'arc-cohérence sur ce modèle 3-ABSTRACT calcule le résultat de l'application des règles *DoublePair* et *MultipleLines*. Prenons à titre illustratif l'exemple de la figure 5 pour la règle *DoublePair*. Dans le tiers vertical du milieu (blocs 2, 5, 8), on examine les places possibles de la valeur 2. Dans le bloc 2, la valeur 2 peut aller dans les cases $x[1, 4]$ et $x[1, 6]$, soit $idxB(2, 2) \in \{1, 3\}$, et donc, en termes de colonnes $cidxB(2, 2) \in \{1, 3\}$. Dans le bloc 5, la valeur 2 peut aller dans les cases $x[5, 4]$ et $x[5, 6]$, soit $idxB(5, 2) \in \{4, 6\}$, et donc, en termes de colonnes $cidxB(5, 2) \in \{1, 3\}$. La propagation globale des différences entre $cidxB(2, 2), cidxB(5, 2)$ et $cidxB(8, 2)$ amène à déduire $cidxB(8, 2) = 2$

La règle *MultipleLines* est basée sur le même principe, à savoir qu'une contrainte `all-different`(x, y, z) permet de réduire le domaine de z à $\{3\}$ lorsque $domain(x) = domain(y) = \{1, 2\}$.

3.2.2 k-PRIMAL

Nous proposons ici une série de modèles, pour $k = 3, ..., k = 9$, basés sur le modèle PRIMAL, et lui ajoutant les contraintes globales de différence pour toute clique de différences binaires de k variables. Ces modèles sont évidemment emboîtés, au sens où : 9-PRIMAL $\succcurlyeq \ldots \succcurlyeq$ 3-PRIMAL $\succcurlyeq$ PRIMAL et pour $k = 9$, on obtient la méthode capable des inférences les plus performantes, et qui propage globalement les inférences sur chaque zone (ligne, colonne ou bloc).

Lien avec les règles Montrons que AC(3-PRIMAL) calcule les inférences de la règle $NakedPair$. Dans le cas représenté figure 7, on a $x[9, 2] \in \{1, 5\}$, $x[9, 7] \in \{1, 5\}$, $x[9, 8] \in \{1, 6\}$. La propagation globale des diségalités entre $x[9, 2]$, $x[9, 7]$ et $x[9, 8]$ permet immédiatement d'instancier $x[9, 8] = 6$.

On peut aisément généraliser et remarquer que AC(4-PRIMAL) calcule les inférences de la règle Naked Triple et que AC(k-PRIMAL) calcule les inférences de la règle $NakedTuple(k - 1)$.

Par ailleurs, en se référant au lemme liant les raisonnements entre Naked et Hidden, on remarque aussi que k-AC(PRIMAL) calcule les inférences de la règle $HiddenTuple(9 - k)$.

3.2.3 k-DUAL

Nous proposons maintenant une série de modèles, pour $k = 3, ..., k = 9$, basés sur le modèle DUAL, et lui ajoutant les contraintes de différence globale pour toute clique de différences binaires de k variables. Ici encore, ces modèles sont évidemment emboîtés au sens où : 9-DUAL $\succcurlyeq \ldots \succcurlyeq$ 3-DUAL $\succcurlyeq$ DUAL

Lien avec les règles Montrons que AC(3-DUAL) calcule les inférences de la règle *X-Wing*. Dans le cas représenté figure 9, on a $idxL(4, 6) \in \{3, 9\}$ et $idxL(9, 6) \in \{3, 9\}$ (le 6 dans les lignes 4 et 9 ne peut être placé qu'à l'intersection des colonnes 3 ou 9). Prenons une autre ligne, par exemple la ligne 1, où le 6 peut être placé en colonne 2, 3 ou 9 : $idxL(1, 6) \in \{2, 3, 9\}$. La propagation globale des diségalités entre $idxL(1, 6)$, $idxL(4, 6)$ et $idxL(9, 6)$ permet immédiatement d'inférer $idxL(1, 6) \notin \{3, 9\}$.

On peut enfin montrer que AC(4-DUAL) calcule les inférences de la règle *Swordfish*. La règle identifie trois lignes L_{j_1}, L_{j_2} et L_{j_3} et une valeur v telles que $idxL(j_1, v) \in \{a, b\}$, $idxL(j_2, v) \in \{b, c\}$, $idxL(j_3, v) \in \{a, c\}$ et infère que pour toute autre ligne L_{j_4}, $idxL(j_4, v) \notin \{a, b, c\}$. Remarquons qu'il ne s'agit que d'un cas particulier de propagation de la clique de diségalités entre les variables $idxL(j_1, v)$, $idxL(j_2, v)$, $idxL(j_3, v)$ et $idxL(j_4, v)$.

3.2.4 Hiérarchie de modèles

Nous avons proposé différents modèles de programmation par contraintes pour le sudoku. Ces modèles sont tous sémantiquement équivalents au sens où ils admettent les mêmes solutions (pour les variables qui leurs sont à tous communces, les $x[i, j]$). Néanmoins, ces modèles peuvent être classés au sens où le calcul d'AC infère plus ou moins d'information sur les domaines. Ces modèles s'organisent dans la hiérarchie suivante :
PRIMAL $\cup$ DUAL $\cup$ ABSTRACT $\succcurlyeq$ PRIMAL $\cup$ DUAL $\succcurlyeq$ PRIMAL
9-PRIMAL $\succcurlyeq \ldots \succcurlyeq$ 3-PRIMAL $\succcurlyeq$ PRIMAL

niveau	règle minimale
très facile	SinglePosition
facile	SingleCandidate
moyen	CandidateLines ou DoublePairs ou MultipleLines
difficile	NakedPairs ou HiddenPairs ou NakedTriples ou HiddenTriples
très difficile	X-Wing ou SwordFish
expert	autres règles

Table 1: Niveaux de difficulté en fonction des règles à appliquer.

9-DUAL $\succcurlyeq \ldots \succcurlyeq$ 3-DUAL $\succcurlyeq$ DUAL
9-PRIMAL $\equiv$ 9-DUAL
9-PRIMAL $\cup$ 3-ABSTRACT $\succcurlyeq$ 9-PRIMAL $\cup$ ABSTRACT $\succcurlyeq$ 9-PRIMAL

3.3 Branchements

Tout comme en programmation par contraintes où le filtrage n'est pas toujours suffisant pour trouver une solution, les règles de déductions pures ne sont pas toujours suffisantes non plus pour résoudre complètement une instance de sudoku. Pour contourner ces limites, plusieurs règles proposent des techniques permettant plus ou moins explicitement de faire de la recherche.

Par exemple, pour calculer les inférences de la règle *Forcing chains*, il suffit d'utiliser une singleton-cohérence. En effet, cette technique consiste à essayer une valeur d'un domaine pour détecter une contradiction et dans ce cas en déduire que cette valeur est inconsistante. On peut donc en déduire que SAC(PRIMAL) permet d'implémenter la règle *Forcing chains*.

4 Liens entre règles et modèles

À partir de la liste des règles présentées en section 2 nous définissons 5 niveaux de difficultés. Les règles sont ordonnées de sorte que la difficulté de compréhension et de mise en œuvre[6] soit croissante. Le niveau de difficulté est ensuite défini par le niveau minimal de règle à appliquer pour résoudre la grille. On obtient le tableau de définition des niveaux de difficulté de la table 1 (pour le niveau expert, les règles définies dans cet article ne suffisent pas à résoudre la grille).

L'intérêt des modèles présentés dans la section 3 est qu'ils permettent de caractériser chaque niveau de difficulté puisque ceux-ci sont spécifiés par un niveau de règles déterminé. Ainsi, le tableau 2 présente les liens entre les niveaux de difficulté et les modèles présentés.

[6] Une règle simple à comprendre peut être difficile à mettre en œuvre en pratique.

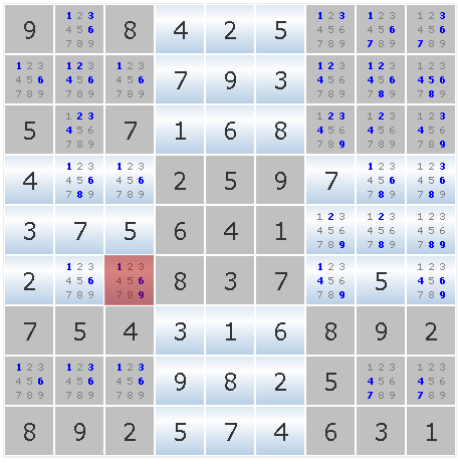
niveau	modèle contraintes
très facile	AC(PRIMAL)
facile	AC(PRIMAL $\cup$ DUAL)
moyen	AC(PRIMAL $\cup$ DUAL $\cup$ 3-ABSTRACT)
difficile	AC(4-PRIMAL $\cup$ DUAL $\cup$ 3-ABSTRACT)
très difficile	AC(4-PRIMAL $\cup$ 4-DUAL $\cup$ 3-ABSTRACT)

Table 2: Niveaux de difficulté et modèles contraintes.

Nous proposons ainsi un lien entre niveaux de difficulté et modèles contraintes. Les niveaux présentés sont bien différents. Pour s'en convaincre, on peut considérer le théorème suivant :
Théorème : Le modèle (PRIMAL $\cup$ DUAL) est un renforcement strict du modèle PRIMAL : (PRIMAL $\cup$ DUAL) $\succcurlyeq$ PRIMAL
Preuve : L'instance suivante est complètement résolue par AC(PRIMAL $\cup$ DUAL) et non par AC(PRIMAL).

9		8	4					
					3			
		7		6	8			
4					9	7		
	7	5	6					
2				3	7		5	
7	5	4			6	8	9	
						5		
8		2		7	4	6	3	

En effet, si l'arc-cohérence sur le modèle PRIMAL calcule le point fixe suivant (seules les variables instantiées sont représentées):



Or, on voit bien, par exemple, que si l'on ajoute le modèle DUAL, la case marquée en rouge doit prendre la valeur 9 puisque c'est la seule case de sa zone à pouvoir prendre la valeur 9, ce qui instancie la variable du modèle DUAL et par propagation entre les modèles instancie la valeur de la case.

Notons que [6] propose aussi différents modèles contraintes pour résoudre des grilles de sudoku. Mais, l'objectif est tout autre, il s'agit de raffiner les différents modèles par des niveaux de cohérence et de propagation (contraintes globales) croissants, de sorte à résoudre une grille de sudoku quelconque par propagation uniquement. Ainsi, le degré de sophistication du modèle nécessaire pour résoudre une instance donnée apparaît *a posteriori* comme une mesure, la plupart du temps mais pas tout le temps, correcte de la difficulté affichée de la grille.

Notre démarche est tout autre puisque nous cherchons plutôt à *coller* au plus près au raisonnement humain en proposant des modèles contraintes modélisant différentes règles largement appliquées.

5 Conclusion et perspectives

Dans cet article, nous avons présenté différents modèles contraintes permettant de modéliser les différentes règles communément appliquées pour résoudre à la main des grilles de sudoku. L'intérêt de ces modèles permet de déterminer trois niveaux globaux de difficultés liés à la puissance du raisonnement mis en œuvre pour résoudre des grilles :

- des modèles basés sur un raisonnement **local** : modèles PRIMAL, DUAL et ABSTRACT implémentant les règles Single Candidate, Single Position et Candidate Line. Ces règles ne nécessitent qu'un raisonnement local (nécessité traduite dans les modèles par une utilisation exclusive de contraintes de diségalité) pour être appliquées.
- des modèles nécessitant un raisonnement **global** : cohérences AC(3-ABSTRACT), AC(k-PRIMAL) et AC(k-DUAL) implémentant les règles Double Pair, Multiple Lines, Naked Pairs, Naked Triple, X-Wing et Swordfish. Ces règles nécessitent un raisonnement global pour être appliquées. Cette nécessité est traduite dans les modèles par l'utilisation de la contrainte globale de type `all-different`.
- des modèles nécessitant une étape de recherche. Il s'agit d'implémenter alors la technique dite de Forcing Chains.

Ces trois niveaux permettent de réconcilier puissance de filtrage mis en œuvre dans les modèles contraintes et impression de difficulté lors d'une résolution manuelle d'une grille de sudoku. Nous avons en effet mis l'accent sur une variétés de modèles plutôt que sur l'utilisation de techniques de filtrage de plus en plus sophistiquées comme dans [6]. On peut ainsi y trouver un intérêt pédagogique.

Une perspective intéressante de ce travail est d'utiliser ces modèles et des mécanismes à base d'explications pour fournir une aide à la résolution orientée utilisateur. On se rend compte d'ailleurs qu'il est aussi nécessaire de repenser les modèles classiques d'explications [4] car il est important ici d'expliquer finement le raisonnement interne à une contrainte (pour faire par exemple apparaître explicitement la notion de doublet ou de triplet dans un all-different). Cette nécessité ne se limite pas au sudoku et peut être nécessaire dans de nombreux problèmes dont le modèle se réduit à l'utilisation d'une ou deux contraintes globales puissantes. Enfin, nous pouvons aussi envisager la génération directe d'instances de sudoku pour un niveau de difficulté donné.

References

[1] `http://www.palmsudoku.com`.

[2] Bertram Felgenhauer and Frazer Jarvis. `http://www.afjarvis.staff.shef.ac.uk/-sudoku/`.

[3] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[4] Guillaume Rochart and Narendra Jussien. Une contrainte `stretch` expliquée. *JEDAI: Journal Électronique d'Intelligence Artificielle*, 3(31), 2004.

[5] Ed Russell and Frazer Jarvis. `http://www.afjarvis.staff.shef.ac.uk/-sudoku/sudgroup.html`.

[6] Helmut Simonis. Sudoku as a constraint problem. In *Fourth CP internation Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, Sitges, Espagne, 2005.