Modélisation et résolution du problème de Sudoku

Projet PSAR

David TOTY et Maxime TRAN

25 Février 2016

Sommaire

Chapitre 1

Introduction

Dans le cadre de l'année de Master 1 du semestre 2, on a été amené à réaliser un projet. Nous avons choisis le sujet sur le problème du Sudoku car c'est un jeu de "puzzle" très connu et qu'il était très intéréssant de voir le fonctionnement de la modélisation et la résolution du problème du Sudoku en détail en utilisant un solveur basé sur des algorithmes tel que la Programmation Par Contraintes.

Chapitre 2

Objectif

2.1 Présentation du Sudoku

Un problème de Sudoku est constitué d'une grille de dimension 9x9. La grille peut être considérée comme une matrice de 3x3 régions carrées, chacune de ces régions étant elle-même constituée de 3x3 cases. Chaque case de la grille contient soit un blanc (vide), soit un nombre compris entre 1 et 9.
Le problème consiste à compléter les cases blanches, en inscrivant dans chacune d'elles un nombre, de telle sorte que chaque ligne, chaque colonne et chaque région carrée contienne exactement une occurrence de nombres.

		3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

FIGURE 2.1 – Grille de Sudoku incomplète

2.2 Programmation par contraintes

En programmation par contraintes, un problème doit être formulé à l'aide des notions suivantes :

- Des *variables* prenant leurs valeurs dans des *domaines*.
- Des *contraintes* restreignant les valeurs possibles des variables dans leurs domaines en fonction des valeurs des autres variables.

La programmation par contraintes va utiliser pour chaque contrainte, une méthode de résolution spécifique afin de supprimer les valeurs des domaines des variables impliquées dans la contrainte qui, compte tenu des valeurs des autres domaines, ne peuvent appartenir à aucune solution de cette contrainte. Ce mécanisme est appelé *filtrage*. En procédant ainsi pour chaque contrainte, les domaines des variables vont se réduire.

Après chaque modification du domaine d'une variable, il est nécessaire d'étudier à nouveau l'ensemble des contraintes impliquant cette variable, car la modification peut conduire à de nouvelles déductions. Autrement dit, la réduction du domaine d'une variable peut permettre de déduire la non-appartenance de certaines valeurs d'autres variables à une solution. Ce mécanisme est appelé *propagation*.

Ensuite, afin de parvenir à une solution, l'espace de recherche va être parcouru en essayant d'affecter successivement une valeur à toutes les variables. Ce parcours suit une stratégie de recherche qui revient à caractériser des critères permettant de choisir la prochaine variable et la prochaine valeur qui sera affectée à cette variable. Les mécanismes de filtrage et de propagation étant bien entendu relancés après chaque essai puisqu'il y a modification de domaines. Parfois, une affectation peut entraîner la disparition de toutes les valeurs d'un domaine : on dit alors qu'un échec se produit ; le dernier choix d'affectation est alors remis en cause, il y a *backtrack* (retour en arrière) et une nouvelle affectation est tentée.

2.3 Présentation d'un solveur

Un solveur va ensuite, à partir des ces informations, affecter des valeurs de leurs domaines aux variables. Une affectation de l'ensemble des variables qui ne brise aucune contrainte est une solution.

Le mécanisme des solveurs repose sur des algorithmes de propagation de contraintes, de filtrage et de vérification de la consistance locale pour réduire l'espace de recherche et élaguer l'arbre des solutions au fur et à mesure qu'il est construit.

Dans le but de résoudre le problème du Sudoku, nous utiliserons un solveur. Dans notre cas, nous utiliserons l'outil OR-Tools.

Qu'est ce que c'est : Un solveur est un programme informatique qui permet de fournir le résultat d'un problème complexe. Le problème est représenté en une formule booléenne.

Synthaxe : $\wedge$ (ET), $\vee$ (OU), $\neg$ (NON), $\Rightarrow$ (IMPLICATION), $\Leftrightarrow$ (EQUIVALENCE)

Exemple : $((A \vee B) \wedge \neg B) \vee C$
Solution : (true, false, false) et (false, false, true)

Pour trouver une solution à une formule, il faut :

- Etablir une hypothèse valide de départ. C'est la manière dont notre solveur va parcourir et appliquer les variables.
- Propager les conséquences. Le fait d'assigner une valeur à une ou plusieurs variables peut impacter l'ensemble des solutions pour d'autres variables.
Pour notre Sudoku, il faut, par exemple, ne pas qu'il y ait la même valeur deux fois dans une même région. Le fait d'assigner une valeur à une variable (case) dans une région va restreindre le choix de cette valeur pour les autres variables.
- Si il y a une absurdité (non-solution) est rencontrée, on procéde au backtrack. Cette technique consiste à revenir en arrière lorsqu'on arrive à un point d'erreur, où il n'existe pas de solution. Dans notre cas, par exemple, si une case n'a plus d'assignation possible dans son ensemble de solution (domaine = 0). Les affectations précédentes peuvent alors être contesté pour procéder à d'autres affectations.

Et pour les nombres, nous pouvons utiliser la Satisfiability Modulo Theories (SMT). Cela nous permet de remplacer les variables booléennes par des prédicats.

Exemple :

$if(x < 5) \qquad \Rightarrow \qquad x < 5 \wedge y = x + 3$
$\quad y = x + 3$

Maintenant que nous avons vu ce que c'était un Sudoku et un solveur, nous pouvons dès à présent modéliser et avoir une résolution de notre problème de Sudoku.

Chapitre 3

Comment résoudre le problème ?

3.1 Modélisation

3.1.1 Modélisation pour un solveur SMT

Soit x une valeur d'une case de Sudoku. x s'identifie par le triplet (x_i, x_j, $Region_x$) avec x_i correspondant au numéro de ligne de x, x_j correspondant au numéro de colonne de x et $Region_x$ correspondant à la région de x.

On obtient une telle modélisation :

$$\forall x, y, x = y \wedge x_i \neg = y_i \wedge x_j \neg = y_j \wedge Region_x \neg = Region_y$$

Cette modélisation signifie : "*Pour 2 cases x et y, si la valeur de leur case sont égaux entre eux, alors ils ne sont pas sur la même ligne, ni sur la même colonne et ni dans la même région*".

La contraposé est vraie :

$$\forall x, \neg \exists y, x = y \wedge x_i = y_i \vee x_j = y_j \vee Region_x = Region_y$$

Qui signifie : "*Pour une case x, il n'existe pas de case y tel que si la valeur de leur case sont égaux entre eux, alors ils sont sur la même ligne, sur la même colonne et dans la même région*".

3.1.2 Modélisation avec la programmation par contraintes

La programmation par contraintes est un modèle de problèmes qui permet de résoudre des problèmes combinatoires de grandes tailles.

Le problème est modélisé sous la forme d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine. De façon plus formelle, on définit le problème par un triplet (X,D,C) tel que :

- $X = X_1, X_2, ..., X_n$ est l'ensemble des variables du problèmes.
- D est la fonction qui associe à chaque variable X_i son domaine D(X_i), c'est-à-dire l'ensemble des valeurs que peut prendre X_i.
- $C = C_1, C_2, ..., C_k$ est l'ensemble des contraintes. Chaque contrainte C_j est une relation entre certaines variables de X, restreignant les valeurs que peuvent prendre simultanément ces variables.

Exemple : Voici un exemple appliqué au problème du Sudoku.

X = Chacune des 81 cases de la grille de Sudoku.
D = 1, 2, 3, 4, 5, 6, 7, 8, 9, l'ensemble des valeurs que peuvent prendre chaque cases.
C = 1 occurrence d'un chiffre par ligne, colonne et région.

3.1.3 Principe du solveur OR-Tools

Nous utiliserons l'outil OR-Tools, un solveur Open-Source developpé par Google qui utilise la programmation par contraintes comme technique de résolution.

OR-Tools prend en entrée un tableau d'entiers correspondant au Sudoku et un fichier MiniZinc (.mzn) qui correspond à la modélisation du problème du Sudoku. L'outil donne ainsi en sortie un type String (Sudoku résolu, nombre de solutions, temps de résolution).

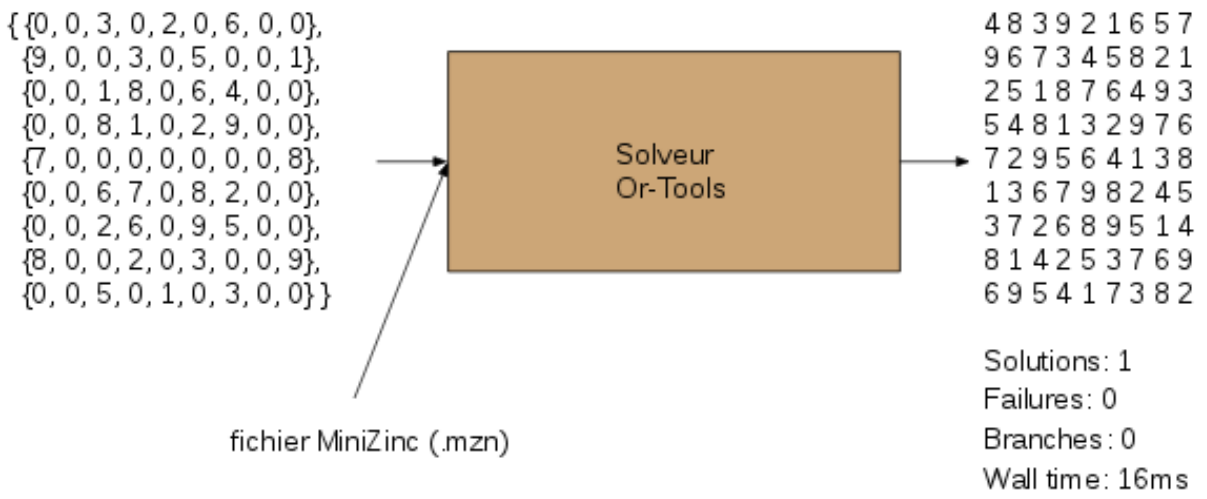

FIGURE 3.1 – Schéma représentatif du solveur

Le format d'entrée : Un tableau d'entier. Une case blanche sera représentée par le caractères $\ll 0 \gg$. Et un fichier MiniZinc .mzn qui correspond à la modélisation du problème. Dans le fichier MiniZinc, on retrouve le input (grille du sudoku), les contraintes et le output.

Le format de sortie : Un affichage de type "String" avec la grille de Sudoku résolue, ainsi que le nombre solution, temps de résolution.

3.2 Résolution

La résolution du problème se base sur la modélisation vu précédemment en utilisant le solveur OR-Tools.

Dans un premier temps, nous devons lui fournir une grille de Sudoku, ainsi que le fichier MiniZinc (cf. Principe du solveur OR-Tools) sans quoi il ne saura comment résoudre le Sudoku.
L'utilisateur pourra soit fournir une grille, soit en générer un. Si l'utilisateur choisit de fournir son propre grille, il devra fournir la grille sous la forme d'un tableau d'entier (cf. Principe du solveur OR-Tools).
Si sa grille est mal construit, le solveur ne saura pas trouver une solution à son problème. Dans le cas, où la grille est correctement construit, le solveur utilisera la programmation par contraintes ainsi qu'un modèle de recherche (recherche en profondeur, recherche en largeur, etc) pour résoudre le Sudoku. Une fois, la résolution est terminé, il affichera le Sudoku complété, ainsi que le nombre de solution trouvé et le temps de résolution.

3.3 Exemple : Problème du N-Queens

Le problème du N-Queens est de placer N reines sur l'échiquier NxN de telle sorte qu'aucunes reines ne puissent s'attaquer entre eux.
On rappelle qu'une reine peut se déplacer horizontalement, verticalement et diagonalement.

Voici un exemple de résolution du solveur sur le problème du N-Queens :

Variables : Q_i où i est la colonne
Domaine : $D_i = \{1, 2, 3, 4\}$ (ligne)
Contraintes : $Q_i \neq Q_j$ (pas être sur la même ligne)
$|Q_i - Q_j| \neq |i - j|$ (pas sur la même diagonale)

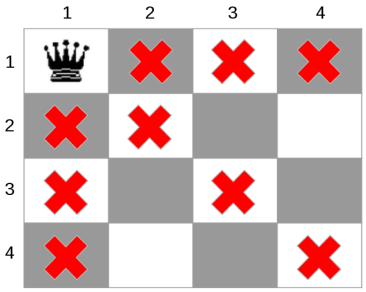

FIGURE 3.2 – Etape 1

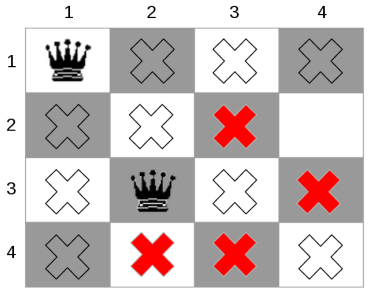

FIGURE 3.3 – Etape 2

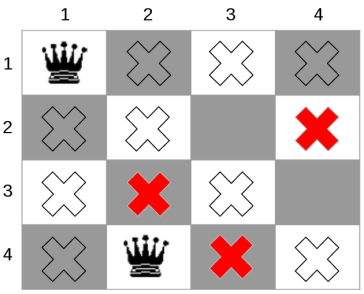

FIGURE 3.4 – Etape 3

A l'étape 1, la reine R1 interdit d'autres reines d'être sur la même ligne, colonne et diagonales qu'elle (cf. Figure 3.5).
Maintenant, les contraintes sont propagées. Nous pouvons alors tester d'autres hypothèses et placer une deuxième reine R2 sur l'une des cases autorisées. Notre solveur peut décider de la placer sur le première case autorisée de la deuxième colonne (cf. Figure 3.6).

Après avoir propagé les contraintes de la reine R2, nous pouvons constater qu'il ne reste qu'une case possible. Or, il reste encore 2 reines à positionner.

Sans solution possible à ce stade, nous devons faire retour en arrière, appelé "*Backtracking*. L'option du solveur est de choisir l'autre case disponible de la deuxième colonne (cf. Figure 3.7).

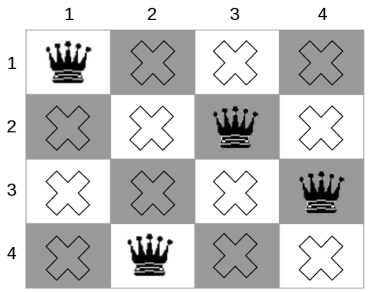

FIGURE 3.5 – Etape 4

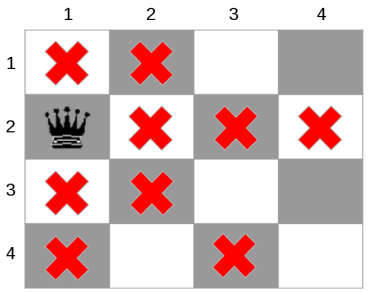

FIGURE 3.6 – Etape 5

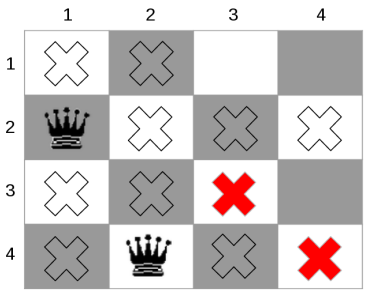

FIGURE 3.7 – Etape 6

Toutefois, la propagation des contraintes force alors, une reine R3 dans la deuxième case de la troisième colonne, ne laissant pas d'emplacement valide pour la quatrième reine R4 (cf. Figure 3.8).

Le solveur doit alors revenir encore en arrière, cette fois jusqu'à l'emplacement de la première reine. Nous avons maintenant montrer qu'il n'y a pas de solution pour notre problème avec l'extrême coin gauche occupé.

Par conséquent, le solveur déplace la reine R1 d'un cran vers le bas et propage les contraintes (cf. Figure 3.9) ne laissant qu'un emplacement possible pour la reine R2 (cf. Figure 3.10).

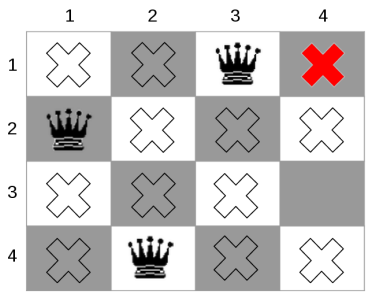

FIGURE 3.8 – Etape 7

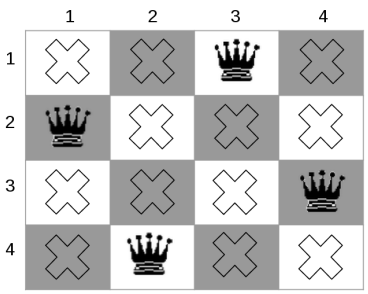

FIGURE 3.9 – Etape 8

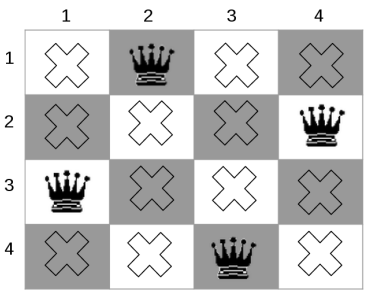

FIGURE 3.10 – Etape 9

Encore une fois, la propagation révèle un emplacement possible pour la reine R3 (cf. Figure 3.11). Ainsi, la dernière reine R4 sera positionnée sur la dernière case disponible (cf. Figure 3.12).

Nous avons obtenus alors notre première solution. On peut remarquer qu'il existe une solution symétrique qui est représenté sur la Figure 3.13.

Chapitre 4

Evaluation et interprétation

4.1 Interprétations

Nous essayerons de rendre notre logiciel final sous la forme suivante :

FIGURE 4.1 – Logiciel final

Le logiciel est l'interface graphique de notre travail. Il servira ainsi de démonstration lors de la présentation.
Comme vous pouvez le voir sur la figure ci-dessus, l'utilisateur pourra choisir de charger un Sudoku, en générer un et lancer la résolution.

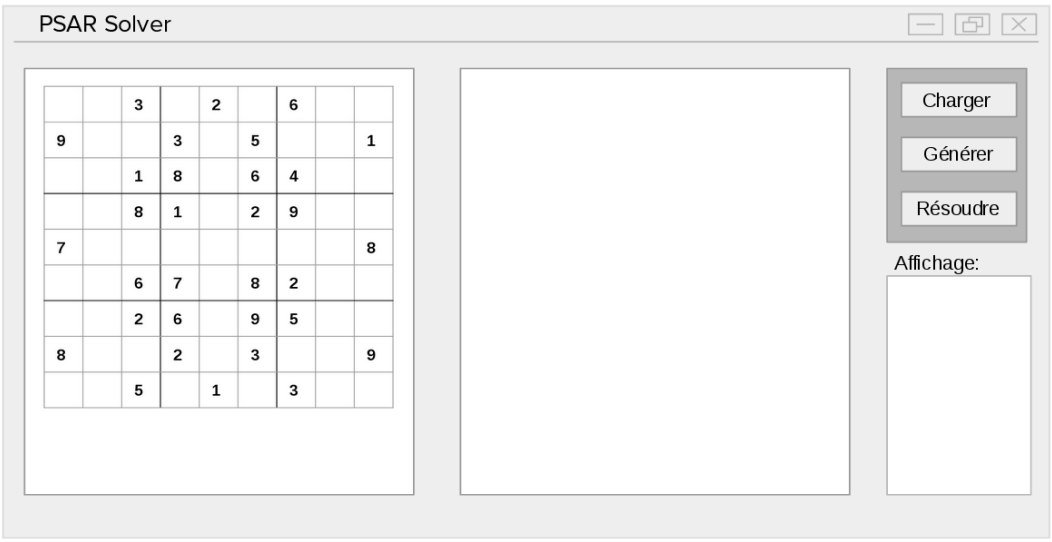

FIGURE 4.2 – Logiciel final

Comme on peut le voir, après avoir charger/générer une grille, celle-ci est affiché alors sur l'encadré gauche du logiciel.

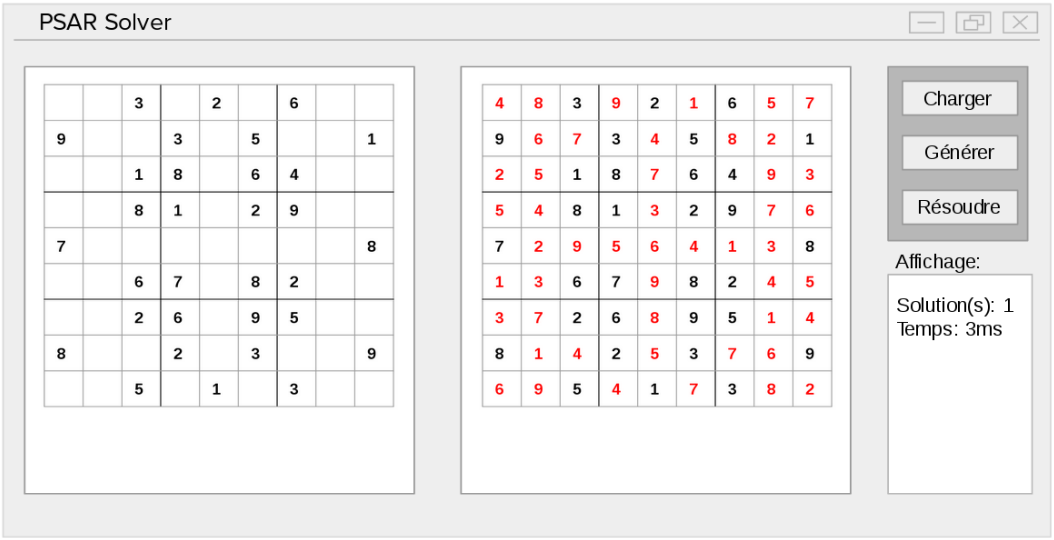

FIGURE 4.3 – Logiciel final

Une fois qu'on a la grille, nous pouvons lancer la résolution du Sudoku. Le Sudoku résolu sera alors affiché sur l'encadré de droite du logiciel avec un affichage du nombre de solution et le temps nécéssaire à la résolution. Si par malheur la grille est mal formé, la solution ne sera pas affiché et l'affichage montrera qu'il y a 0 solutions.

4.2 Efficacité

Dans un premier temps, nous allons utiliser la technique par Force Brute. Dans notre cas, elle consiste à regarder ligne par ligne en essayant chaque valeur commençant par la plus petite.

On commence par placer une valeur sur la première case disponible de la première ligne. Les valeurs possibles étant 1, 4, 5, 7, 8, 9 puisque les valeurs 2, 3 et 6 sont déjà placé sur la première ligne.

1		3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

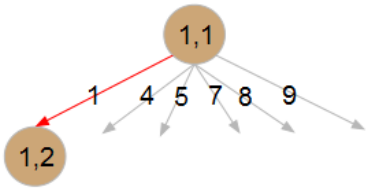

FIGURE 4.4 – Représentation du Sudoku sous forme arborescente

On essaye ensuite la valeur 2 mais étant déjà sur la première ligne, cela l'exclut des choix possibles. De même pour la valeur 3.

1	2	3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

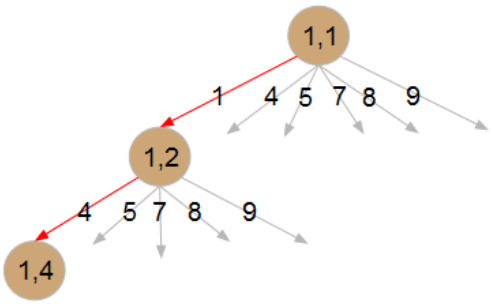

FIGURE 4.5 – Représentation du Sudoku sous forme arborescente

On assigne alors la valeur 4 à la deuxième case de la première ligne. Et on réitère ce procédé jusqu'à la fin de la ligne.

1	4	3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

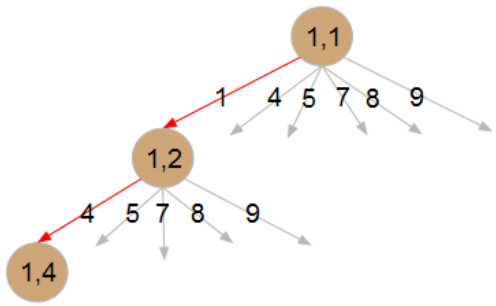

FIGURE 4.6 – Représentation du Sudoku sous forme arborescente

Une fois arrivé en fin de ligne, on constate que la neuvième case qui a pour valeur 9 ne peut être placé à cet emplacement puisque la valeur 9 étant déjà sur cette colonne. On fait alors un retour en arrière.

1	4	3	5	2	7	6	8	9
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7								8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1		3		

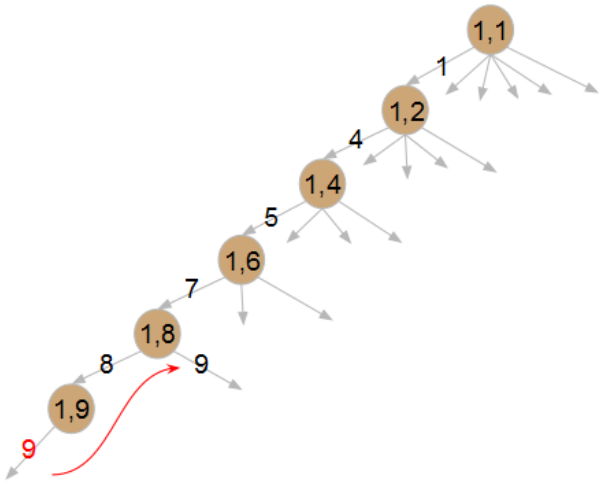

FIGURE 4.7 – Représentation du Sudoku sous forme arborescente

Au lieux d'assigner la valeur 8 à la huitième case, on choisit d'assigner la valeur 9. Mais on remarque aussi qu'il y a non-solution puisque la valeur 8 est déjà présent sur la même colonne. On est amené à revenir en arrière. On va réitéré ce procédé jusqu'à remplir complétement la grille.

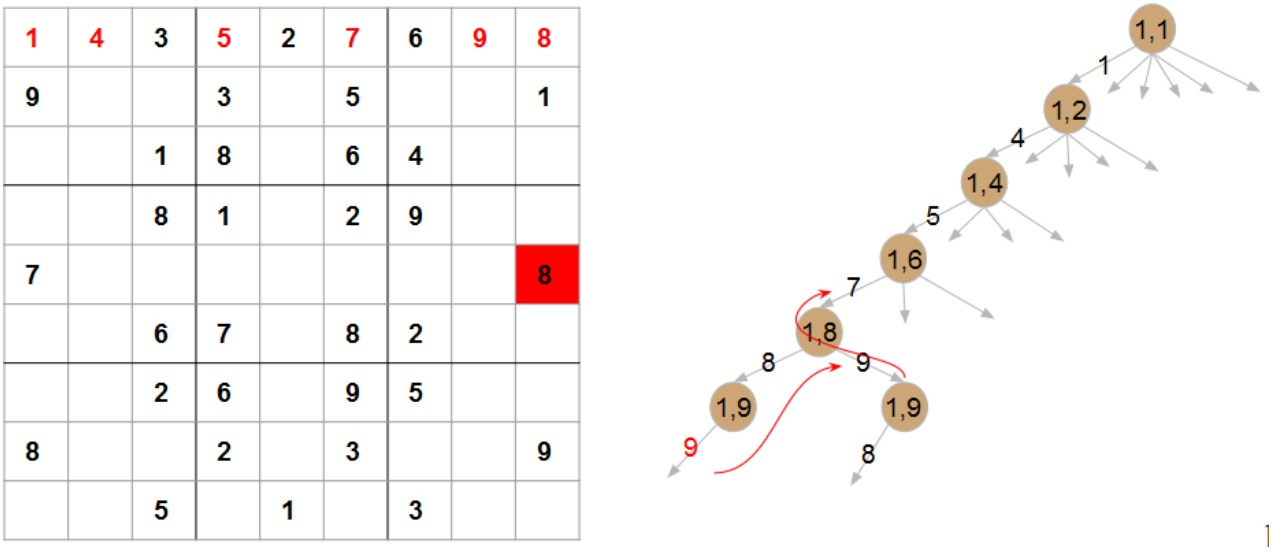

FIGURE 4.8 – Représentation du Sudoku sous forme arborescente

On constate que cette méthode peut coûter cher en terme de temps.
Il y a 9x9 cases où chaque case peut prendre une valeur de 1 à 9. Par conséquent, dans le pire des cas nous avons une compléxité de l'ordre de 9^{81}.

Au cours de notre avancement, nous verrons si la formulation de la modélisation d'un problème peut impacter sur le temps de calcul.

Chapitre 5

Validation du travail