Table des matières

Résolution d'un sudoku par calcul booléen

solution : (Centrale 2014)

1 Présentation du jeu de sudoku

1.1 Règles du jeu

Le sudoku est un jeu de réflexion très répandu depuis quelques années. Il se présente sous l'aspect d'une grille carrée à 81 cases (9 lignes et 9 colonnes), elle-même scindée en 9 sous-grilles de taille 3×3 appelées blocs. Cette grille est initialement partiellement remplie par des chiffres compris entre 1 et 9 : c'est la grille initiale. Le but du jeu consiste à compléter entièrement la grille en remplissant les cases initialement vides par des chiffres de 1 à 9, de telle manière qu'une fois la grille remplie, chacun des chiffres compris entre 1 et 9 apparaisse une et une seule fois dans chacune des 9 lignes, chacune des 9 colonnes et chacun des 9 blocs de la grille : on obtient alors une grille finale associée à la grille initiale.

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

Exemple de grille initiale

	0	1	2	3	4	5	6	7	8
0	0	9	0	2	0	0	6	0	5
1	3	2	0	0	0	7	0	0	0
2	0	7	0	9	0	5	0	0	8
3	0	1	0	0	0	0	0	0	0
4	0	0	7	0	0	0	0	9	4
5	6	0	0	0	0	0	0	0	0
6	0	0	8	0	0	0	0	0	7
7	0	3	0	4	9	1	5	0	0
8	0	0	0	0	0	3	0	0	0

Représentation de la grille initiale ci-contre (où le bloc numéro 3 est grisé)

FIGURE 1

On suppose dans tout le sujet que la grille initiale est ainsi faite qu'elle admet une et une seule grille finale qui lui soit associée. Résoudre une grille de sudoku consiste donc à déterminer la grille finale associée à une grille initiale donnée.

L'objet du problème est d'étudier un algorithme de résolution des grilles de sudoku fondé sur la manipulation de formules logiques. En effet, le principe de la résolution d'une grille de sudoku peut s'énoncer facilement par les cinq règles logiques ci-dessous.

Construire à partir d'une grille initiale I donnée une grille finale F telle que :

- (K) toute case de F contient une et une seule fois l'un des chiffres 1 à 9 ;
- (L) toute ligne de F contient une et une seule fois chacun des chiffres 1 à 9 ;
- (C) toute colonne de F contient une et une seule fois chacun des chiffres 1 à 9 ;
- (B) tout bloc de F contient une et une seule fois chacun des chiffres 1 à 9 ;
- (I) toute case de I remplie conserve la même valeur dans F .

À ces cinq règles, il convient donc d'ajouter implicitement que la grille F existe et est unique. On peut remarquer que les quatre premières conditions (K), (L), (C) et (B) présentent de la redondance d'un point de vue logique, mais cette redondance s'avère utile pour déduire plus facilement de nouveaux faits permettant de remplir la grille.

1.2 Représentation d'une grille de sudoku

Une grille est représentée par un tableau à deux dimensions à 9 lignes et 9 colonnes numérotées chacune de 0 à 8. Elle est découpée en 9 blocs numérotés également de 0 à 8 dans l'ordre de lecture de gauche à droite et de haut en bas et les 9 cases d'un bloc donné sont également numérotées de 0 à 8 selon le même procédé. Ainsi, la même case d'une grille peut être référencée de deux manières différentes : pour $(i, j) \in [\![0, 8]\!]^2$, on appelle case d'indice (i, j) la case de la grille située à l'intersection de la ligne i et de la colonne j ; pour $(b, r) \in [\![0, 8]\!]^2$, on appelle case de bloc (b, r) la case de la grille située dans le bloc numéro b ayant le numéro r. Une case non encore remplie est affectée de la valeur 0.

Une grille de sudoku est représentée par un tableau défini par `Array.make_matrix 9 9 0` ; pour $(i, j) \in [\![0, 8]\!]^2$, l'accès à la case d'indice (i, j) d'un tel tableau t se fait par l'expression `t.(i).(j)`.

1.3 Préliminaires

solution : Dans le fichier `sudoku.ml`, j'ai en plus des fonctions proposées par l'énoncé, rajouté un peu d'affichage pour suivre la résolution de la grille.

1. Écrire une fonction `appartient` d'appartenance d'un élément à une liste.
2. Écrire une fonction `suppression` de suppression de toutes les occurrences d'un élément dans une liste.
3. crire une fonction `ajout` d'ajout d'un élément dans une liste sans redondance (si l'élément appartient déjà à la liste, on renvoie la liste sans la modifier).
   solution :

```
let ajout x l=
    if appartient x l then l
    else x::l
;;
```

4. Écrire une fonction indice qui, étant donné un couple $(b, r) \in [\![0, 8]\!]^2$ correspondant à la case de bloc (b, r) dans une grille, renvoie l'indice (i, j) de cette case dans la grille. Cette fonction pourra utiliser judicieusement les quotients et restes de divisions euclidiennes par 3 et on justifiera les formules utilisées. Par exemple, la case de bloc $(3, 6)$ est la case d'indice $(5, 0)$ (c'est la case grisée en plus foncé dans la figure 1) : indice appliqué au couple $(3, 6)$ doit donc renvoyer le couple $(5, 0)$.
   La fonction aura pour signature `indice : int * int -> int * int = < fun >`.
   solution :

```
let indice (b,r)= (* nb : la fonction doit prendre un couple en entrée, et non deux
    ↪ argument entiers *)
    let x_bloc,y_bloc = b mod 3, b/3 in
        (3*x_bloc + r mod 3, 3*y_bloc +r/3)
;;
```

2 Codage de la formule initiale

2.1 Représentation des formules logiques

Dans tout le problème, $\wedge$, $\vee$ et $\neg$ dénotent respectivement les connecteurs de conjonction, de disjonction et de négation. Étant donné un ensemble V de variables propositionnelles, on considère l'ensemble F des formules logiques construites à partir de V et de l'ensemble de connecteurs $\{\wedge, \vee, \neg\}$.

Une valuation sur V est une application $\sigma : V \to B$ (où $B = \{vrai, faux\}$ désigne l'ensemble des booléens) ; cette application σ s'étend en une application $Ev_\sigma : F \to B$, attribuant une valeur de vérité à toute formule $f \in F$ sous la valuation σ. Une formule logique f est satisfiable s'il existe une valuation σ telle que $Ev_\sigma(f) = vrai$. Deux formules logiques f et g sont équivalentes si $Ev_\sigma(f) = Ev_\sigma(g)$ pour toute valuation σ, et on note alors $f \equiv g$.

Un littéral est une formule logique réduite à p ou à $\neg p$, où p est une variable propositionnelle. Une clause est une formule logique écrite sous la forme d'une disjonction de littéraux. La clause vide est la clause ne contenant aucun littéral, elle est notée $\bot$ et est considérée comme non satisfiable. Une clause unitaire est une clause réduite à un seul littéral ; une clause unitaire positive (respectivement négative) est une formule logique réduite à p (respectivement à $\neg p$), où p est une variable propositionnelle. Une formule en forme normale conjonctive est une formule logique écrite sous la forme d'une conjonction de clauses. La formule vide est la formule ne contenant aucune clause, elle est notée $\square$ et est considérée comme satisfiable.

Dans tout le problème, les variables propositionnelles sont indexées par un triplet d'entiers (i, j, k) et notées $x^k_{(i,j)}$, avec la sémantique suivante : pour un triplet $(i, j, k) \in [\![0, 8]\!] \times [\![0, 8]\!] \times [\![1, 9]\!]$, une valuation σ assigne la valeur vrai à $x \times k_{(i,j)}$ lorsque la case d'indice (i, j) de la grille de sudoku contient la valeur k. L'ensemble V utilisé dans le problème est donc constitué des $9^3 = 729$ variables propositionnelles $x^k_{(i,j)}$, où (i, j, k) parcourt $[\![0, 8]\!] \times [\![0, 8]\!] \times [\![1, 9]\!]$.

À titre d'exemples :

- La clause $x^5_{(0,0)} \vee x^5_{(0,8)} \vee x^5_{(8,0)} \vee x^5_{(8,8)}$ exprime qu'une au moins des quatre cases situées aux coins de la grille est occupée par un 5 ;
- la formule sous forme normale conjonctive $\bigwedge_{i=0}^{8} \left(\bigvee_{k=1}^{9} x^k_{i,7} \right)$ exprime que chacune des valeurs de la colonne 7 est constituée de chiffres compris entre 1 et 9 ;
- La clause $\bigvee_{j=0}^{8} x^6_{\texttt{indice3j}}$ exprime que l'une des cases du bloc numéro 3 de la grille contient le chiffre 6 (où `indice` est la fonction écrite partie 1.3).

Pour la programmation, les clauses sont des listes de littéraux et les formules logiques (qui seront écrites exclusivement en forme normale conjonctive) des listes de clauses.

Ainsi, la formule $x^7_{1,8} \wedge \left(x^3_{0,4} \vee \neg x^5_{2,7} \right)$ sera représentée par la liste $[[(1, 8, 7)]; [(0, 4, 3); \overline{(2, 7, 5)}]]$ (où un triplet surligné désigne un littéral négatif). Plus précisément On déclare le type litteral comme suit :

```
type litteral = X of int * int * int | NonX of int * int * int ;;
```

Dans un triplet d'entiers de type `int * int * int`, le premier élément correspond au numéro de ligne, le second au numéro de colonne et le troisième à la valeur comprise entre 1 et 9. Une clause est de type `litteral list` et une formule logique (en forme normale conjonctive) de type `litteral list list`.

Avant de décrire des stratégies de résolution d'une grille de sudoku, on va coder l'information contenue dans la grille initiale par une formule logique en forme normale conjonctive. Ce codage se fait en deux temps : on code déjà la règle générale du jeu, représentée par les quatre règles logiques (K), (L), (C) et (B) ; puis on code l'information propre à la grille initiale fournie, qui repose sur la cinquième règle (I).

2.2 Formule logique décrivant la règle du jeu

On s'intéresse dans cette partie à coder par une formule logique en forme normale conjonctive l'information fournie par la règle du jeu. Chacune des quatre règles logiques (K), (L), (C) et (B) peut être scindée en deux sous-règles :

- (K1) toute case de F contient au moins une fois l'un des chiffres 1 à 9 ;
- (L1) toute ligne de F contient au moins une fois chacun des chiffres 1 à 9 ;
- (C1) toute colonne de F contient au moins une fois chacun des chiffres 1 à 9 ;
- (B1) tout bloc de F contient au moins une fois chacun des chiffres 1 à 9 ;
- (K2) toute case de F contient au plus une fois l'un des chiffres 1 à 9 ;
- (L2) toute ligne de F contient au plus une fois chacun des chiffres 1 à 9 ;
- (C2) toute colonne de F contient au plus une fois chacun des chiffres 1 à 9 ;
- (B2) tout bloc de F contient au plus une fois chacun des chiffres 1 à 9.

1. (a) Pour $(i, j) \in [\![0, 8]\!]^2$, écrire une clause qui traduit la phrase mathématique : $\exists k \in [\![1, 9]\!], x^k_{(i,j)}$.
   solution : $\bigvee_{k=1}^{9} x^k_{(i,j)}$

(b) Traduire la condition (K1) en une phrase mathématique.
solution : $\forall (i,j) \in [\![0,8]\!]^2$, $\exists k \in [\![1,9]\!]$, $x^k_{(i,j)}$.

(c) En déduire une formule $K1$ en forme normale conjonctive qui exprime la condition (K1).
solution : $\bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} \bigvee_{k=1}^{9} x^k_{(i,j)}$.

(d) Déterminer le nombre de clauses qui contient $K1$.
solution : Il y a une clause pour chaque couple $(i,j) \in [\![0,8]\!]^2$, ce qui fait 81 clauses.

(e) Écrire une fonction `case1` qui ne prend pas d'argument et renvoie la liste qui représente la formule logique $K1$.
solution :

```
1 ≠
```

2. Traduire la condition (L1) par une phrase mathématique et en déduire une formule $L1$ en forme normale conjonctive qui exprime (L1).
solution : $\forall i \in [\![0,8]\!]$, $\forall k \in [\![1,9]\!]$, $\exists j \in [\![0,8]\!]$ tq $x^k_{(i,j)}$.

D'où la formule logique $\bigwedge_{i=0}^{8} \bigwedge_{k=1}^{9} \bigvee_{j=0}^{8} x^k_{(i,j)}$.

3. (a) Obtenir de même des formules logiques $C1$ et $B1$ exprimant les conditions (C1) et (B1).
solution : $C1 = \bigwedge_{j=0}^{8} \bigwedge_{k=1}^{9} \bigvee_{i=0}^{8} x^k_{(i,j)}$

Pour (B1), on écrirait en maths $\forall b \in [\![0,8]\!], \forall k \in [\![1,9]\!], \exists r \in [\![0,8]\!], x^k_{\texttt{indice}(b,r)}$. (Dans chaque bloc, pour chaque chiffre entre 1 et 9, il existe une des cases du bloc qui contient k.)

Ce qui donne en formule logique : $B1 = \bigwedge_{b=0}^{8} \bigwedge_{k=1}^{9} \bigvee_{r=0}^{8} x^k_{\texttt{indice}(b,r)}$

(b) Écrire une fonction `bloc1` qui renvoie la liste qui représente la formule logique B1.
solution :

```
153 let bloc1()=
154   (* Crée la formule (B1) *)
155     let res= ref [] in
156     for b=0 to 8 do
157         for k=1 to 9 do
158             let c=ref[] in
159                 for r=0 to 8 do
160                     let i,j=indice(b,r) in
161                     c:= X(i,j,k):: !c
162                 done;
163                 res:= !c:: !res
164         done
165     done;
166     (!res:formule)
167 ;;
```

4. *Condition (L2)*

(a) Soit $(i,k) \in [\![0,8]\!] \times [\![1,9]\!]$. Exprimer mathématiquement le fait que la ligne i de la grille contient au plus une fois la valeur k, et écrire une formule en forme normale conjonctive qui traduit cette condition.
solution : En mathématiques, on écrit généralement une unicité à l'aide d'une implication : $\forall (j_1,j_2) \in [\![0,8]\!]^2$, $x^k_{(i,j_1)} \wedge x^k_{(i,j_2)} \Rightarrow j_1 = j_2$.
On peut bien sûr transformer l'implication à l'aide de $\wedge$ et $\neg$:

$$\forall (j_1,j_2) \in [\![0,9]\!]^2,\ j_1 = j_2 \vee \neg x^k_{(i,j_1)} \vee \neg x^k_{(i,j_2)}.$$

Ce qui s'écrit sous forme normale conjonctive ainsi :

$$\bigwedge_{j_1=0}^{8} \bigwedge_{j_2=0}^{8} j_1 = j_2 \vee \neg x^k_{(i,j_1)} \vee \neg x^k_{(i,j_2)}.$$

Pour enlever les $j_1 = j_2$, qui ne sont pas traduits par des variables propositionnelles dans l'énoncé, on peut l'écrire ainsi en mathématiques : $\forall (j_1,j_2) \in [\![0,8]\!]^2$, $j_1 < j_2 \Rightarrow \neg x^k_{(i,j_1)} \vee \neg x^k_{(i,j_2)}$. Nous obtenons alors la formule :

$$\bigwedge_{j_1=0}^{8} \bigwedge_{j_2=j1+1}^{8} \neg x^k_{(i,j_1)} \vee \neg x^k_{(i,j_2)}.$$

(b) En déduire une phrase mathématique qui traduit la condition (L2) et une formule $L2$ en forme normale conjonctive qui exprime (L2).
solution : On rajoute un pour tout i et pour tout k devant la formule précédente. Forme normale conjonctive :

$$\bigwedge_{i=0}^{8}\bigwedge_{k=1}^{9}\bigwedge_{j_1=0}^{8}\bigwedge_{j_2=j_1+1}^{8} \neg x^k_{(i,j_1)} \vee \neg x^k_{(i,j_2)}.$$

(c) Déterminer le nombre de clauses que contient $L2$.
solution : La formule $L2$ contient une clause pour chaque $i \in [\![0,8]\!]$, chaque $k \in [\![1,9]\!]$, chaque $j_1 \in [\![0,8]\!]$ et chaque $j_2 \in [\![j_1+1,8]\!]$.

Le nombre de couples $(j_1,j_2) \in [\![0,8]\!]^2$ tel que $j_2 > j_1$ est $\dfrac{9\times 8}{2}$ (moitié supérieure stricte d'un carré), ce qui vaut 36.
Le nombre de clauses final est alors $9 \times 9 \times 36$, qui vaut 2916.

(d) Écrire une fonction `ligne2` qui renvoie la liste qui représente $L2$.
solution :

```
let ligne2()=
     (* Crée la formule (L2) : toute ligne a au plus un chiffre*)
    let res=ref[] in
    for i =0 to 8 do
        for k=1 to 9 do
            for j1=0 to 8 do
                for j2=j1+1 to 8 do
                        res:= [ NonX(i,j1,k); NonX(i,j2,k)]:: !res
                done
            done
        done
    done;
    !res
;;
```

5. Obtenir de même des formules logiques C2, B2 et K2 exprimant les conditions (C2), (B2) et (K2).

On suppose avoir écrit des fonctions lig1, col1, case2, col2 et bloc2 qui renvoient respectivement les listes qui représentent les formules logiques L1, C1, K2, C2 et B2.

On pose alors $F_{\text{règle}} = K1 \wedge L1 \wedge C1 \wedge B1 \wedge K2 \wedge L2 \wedge C2 \wedge B2$: $F_{\text{règle}}$ est la formule logique en forme normale conjonctive décrivant la règle du jeu. Elle contient 11988 clauses.

2.3 Formule logique décrivant la grille initiale

On s'intéresse dans cette partie à coder par une formule logique en forme normale conjonctive l'information fournie par la grille initiale à compléter.

On pourrait pour cela se contenter de considérer une formule logique F construite à partir de la cinquième règle (I) donnée au début de l'énoncé : pour toute case de la grille d'indice (i,j) initialement remplie par la valeur $k \in [\![1,9]\!]$, on considère la clause réduite au littéral positif $x^k_{(i,j)}$ et, si r est le nombre de cases remplies dans la grille initiale, on pose F comme étant la conjonction des r clauses ainsi obtenues.

Mais, pour accélérer la phase d'inférences logiques qui sera menée dans la partie 3, on n'hésite pas à introduire à nouveau de la redondance en déduisant d'emblée un certain nombre de faits que l'on ajoute à la formule F liée à la grille initiale. Ces nouveaux faits sont de deux ordres :

- si dans la grille initiale, la case d'indice (i,j) est déjà remplie par la valeur $k \in [\![1,9]\!]$, on peut considérer d'emblée, en plus de la clause unitaire positive $x^k_{(i,j)}$ (déjà prise en compte par la formule F ci-dessus), les 8 clauses unitaires négatives $\neg x^l_{(i,j)}$ avec $l \in [\![1,9]\!] \setminus \{k\}$; on enrichit ainsi la formule F en une formule F_1 en forme normale conjonctive constituée de r clauses unitaires positives (celles de F) et $8r$ négatives (où r est le nombre de cases remplies dans la grille initiale) ;
- si, dans une grille initiale, une case d'indice (i,j) n'est pas remplie (c'est-à-dire est occupée par la valeur 0), alors on sait que cette case ne peut être remplie par aucune valeur $k \in [\![1,9]\!]$ apparaissant déjà dans la ligne i, ou dans la colonne j, ou dans le bloc auquel appartient la case d'indice (i,j) ; puisqu'une telle valeur k est interdite pour la case d'indice (i,j), on peut donc considérer la clause unitaire négative $\neg x^k_{(i,j)}$; ainsi, pour la case d'indice (1, 3) de l'exemple de la figure 1 à droite, il n'est pas question de remplacer la valeur 0 par aucune des valeurs appartenant à $\{2,3,4,5,7,9\}$: on peut donc ajouter les clauses $\neg x^2_{(1,3)}$, $\neg x^3_{(1,3)}$, $\neg x^4_{(1,3)}$, $\neg x^5_{(1,3)}$, $\neg x^7_{(1,3)}$

et $\neg x^9_{(1,3)}$; on note alors F_2 la formule en forme normale conjonctive constituée de la conjonction de toutes ces clauses unitaires négatives obtenues en parcourant toutes les cases non remplies de la grille initiale.

La formule décrivant la grille initiale est donc $F_{\text{grille}} = F_1 \wedge F_2$.

1. Écrire une fonction `donnees` qui, à partir d'une grille de sudoku initiale (donnée sous la forme d'un tableau), renvoie la formule F_1 , toujours sous la forme d'une liste de listes de littéraux.
solution :

```
let donnees g=
    (* Lit les cases déjà remplies de la grille g.
     Renvoie la liste de clauses isolées contenant pour chaque case (i,j) contenantt une
         ↪ valeur k, [X(i,j,k)]  et [NonX(i,j,l)] pour ≠lk *)
    let res= ref [] in
    for i=0 to 8 do
        for j=0 to 8 do
            if g.(i).(j)<>0 then
                for k =1 to 9 do
                    if g.(i).(j)=k then
                        res:=  [X(i,j,k)] :: !res
                    else
                        res:=   [NonX(i,j,k)] :: !res
                done
        done
    done;
    !res
;;
```

2. (a) Étant donnée une case d'indice $(i, j) \in [\![0, 8]\!]^2$, exprimer en fonction de i et j le numéro de bloc $b \in [\![0, 8]\!]$ auquel cette case appartient.
solution : L'indice de ligne du bloc est $i/3$ (division euclidienne) et son indice de colonne est $j/3$, de sorte que son numéro est $3 \times \left\lfloor \frac{i}{3} \right\rfloor + \frac{j}{3}$.

   (b) Écrire une fonction `interdites_ij` qui, étant données une grille de sudoku initiale et une case d'indice (i, j) non remplie de la grille initiale, renvoie la conjonction des clauses unitaires négatives correspondant aux valeurs interdites pour la case d'indice (i, j).
solution :

```
≠
let interdites_ij g (i,j)=
    (* Renvoie la formule (formée de clause unitaires négatives) indiquant les
        ↪ chiffres impossibles en case (i,j), après utilisation directe des règles
        ↪ B2, L2, et K2. *)

    let res= ref [] in

    (* lecture de la ligne i *)
    for j'=0 to 8 do
         if j' <> j && g.(i).(j') <>0 then  res:= ajout [NonX(i,j',g.(i).(j'))]  !res
    done;

    (* lecture de la colone j*)
    for i'=0 to 8 do
         if i' <> i && g.(i').(j) <>0 then  res:= ajout [NonX(i',j,g.(i').(j))] !res
    done;

    (* lecture du bloc*)
    let b = numeroBloc (i,j) in
    for r=0 to 8 do
        let (i',j') = indice (b,r) in
         if (i,j) <> (i',j') && g.(i').(j') <>0 then  res:= ajout
             ↪ [NonX(i',j',g.(i').(j'))]  !res
    done;

```

```
    !res
;;
```

(c) Écrire une fonction `interdites` qui, à partir d'une grille de sudoku initiale, renvoie la formule F_2 .
solution :

```
≠
let interdites g=
    let res = ref[] in
    for i=0 to 8 do
        for j=0 to 8 do
            if g.(i).(j) =0 then
                res:= interdites_ij g (i,j) @ !res (* il ne peut pas y avoir de
                    ↪ doublons entre ces deux listes *)
        done
    done;
    !res
;;
```

3. Montrer que le nombre de clauses de la formule F_{grille} est majoré par 729.
solution : Pour chaque case de la grille, il y a :
   - 9 clauses (dans F_1, clauses unitaires, une positive et 8 négatives) si la case est déjà remplie ;
   - au plus 8 clauses (dns F_2, unitaires négatives) si elle ne l'est pas.

   Ceci nous fait au plus 9×81 soit 729 clauses.

La formule logique à associer à une grille initiale qui code à la fois les informations qu'elle contient et qui peut permettre de la résoudre est donc $F_{\texttt{initiale}} = F_{\texttt{grille}} \wedge F_{\texttt{règle}}$.

On suppose dans la suite avoir écrit une fonction `formule_initiale` qui, à partir d'une grille initiale, renvoie la formule $F_{\texttt{initiale}}$.

3 Résolution

3.1 Propagation unitaire

Nous supposons désormais que nous disposons d'une formule $F_{initiale}$ en forme normale conjonctive encodant un sudoku. Pour le résoudre on cherche à satisfaire $F_{initiale}$.

1. Combien existe-t-il de valuations satisfaisant $F_{initiale}$?
solution : Si la grille initiale est correcte, il existe une unique manière de la compléter, donc il existe une unique valuation satisfaisant $F_{initiale}$.
2. Déterminer le nombre de lignes de la table de vérité de $F_{initiale}$.
solution : Cette formule contient 9^3 variables (9 dans chaque case). Il y aura donc 2^{9^3} lignes dans la table de vérité.
3. On justifie formellement cette simplification. Soit F une formule en forme normale conjonctive contenant un littéral isolé l (associé à la variable propositionnelle p : on a donc $l = p$ ou $l = \neg p$). F peut s'écrire alors sous la forme $F = l \wedge F_1 \wedge F_2 \wedge F_3$, où :
   - $F1$ est une formule constituée de clauses contenant chacune le littéral l ;
   - $F2$ est une formule constituée de clauses contenant chacune le littéral $\neg l$ et ne contenant pas le littéral l ;
   - $F3$ est une formule constituée de clauses ne contenant chacune ni le littéral l, ni le littéral $\neg l$.

   Remarquons que chacune des formules F_1 , F_2 et F_3 peut être réduite à la formule vide, satisfiable.
   La formule simplifiée de F par le littéral l est alors la formule $F' = F'_2 \wedge F_3$, où F'_2 s'obtient à partir de F_2 en supprimant toutes les occurrences du littéral $\neg l$ dans chacune des clauses de F_2 .
   Soit σ une valuation de $\mathcal{V} \setminus \{p\}$. Montrer que σ satisfait F' si et seulement s'il existe une valuation $\bar{\sigma}$ de $\mathcal{V}$ prenant les mêmes valeurs que σ sur $\mathcal{V} \setminus \{p\}$. On précisera la valeur de $\bar{\sigma}(p)$.
   solution : Je note pour toute valuation σ et toute formule f $\mathcal{E}_\sigma(f)$ le résultat de l'évaluation de f en σ.

   Commençons par remarquer que F' ne contient plus la variable p, puisqu'on a éliminé tous les p et les $\neg p$ de F. C'est pourquoi pour toute valuation θ sur $\mathcal{V} \setminus \{p\}$, $\mathcal{E}_\theta(F')$ est bien défini.

- Supposons que σ satisfait F'. On définit alors σ sur $\mathcal{V}$ comme étant la valuation qui coïncide avec σ sur $\mathcal{V} \setminus \{p\}$ et telle que $\bar{\sigma}(p) = \top$ si $l = p$ et $\bar{\sigma}(p) = \bot$ si $l = \neg p$. De sorte que $\mathcal{E}_{\bar{\sigma}}(l) = \top$.
  Montrons que $\bar{\sigma}$ satisfait F.
  - ◇ Les clauses de F_1 sont des disjonction contenant le littéral l, qui est vérifié. Donc elles sont vérifiées. Donc F_1 est vérifiée.
  - ◇ Pour toute clause c de F_2, il existe une clause c' telle que $c = c' \vee \neg l$, et c' est vérifiée par σ car elle apparaît dans F_2'. Donc c' est également vérifiée par $\bar{\sigma}$, et donc c est vérifiée par $\bar{\sigma}$.
    Ainsi, $\mathcal{E}_{\bar{\sigma}}(F_2) = \top$.
  - ◇ Ensuite, $\mathcal{E}_{\bar{\sigma}}(F_3) = \mathcal{E}_{\sigma}(F_3) = \top$. ($\mathcal{E}_{\sigma}(F_3)$ est bien définie car F_3 n'utilise pas la variable p.)
  - ◇ Enfin, comme dit ci-dessus, $\bar{\sigma}$ satisfait l.

  Ainsi, $\bar{\sigma}$ satisfait F.
- Réciproquement, supposons qu'il existe $\bar{\sigma}$ qui prolonge σ à $\mathcal{V}$ et qui vérifie F.
  Comme F contient la clause l, $\bar{\sigma}$ satisfait l. Donc $\bar{\sigma}(p) = \top$ si $l = p$ et $\bar{\sigma}(p) = \bot$ sinon.
  - ◇ On a encore $\mathcal{E}_{\bar{\sigma}}(F_3) = \mathcal{E}_{\sigma}(F_3) = \top$ car F_3 n'utilise pas la variable p.
  - ◇ Soit c une clause de F_2. Soit c' obtenue en retirant $\neg l$ de c, donc $c = c' \vee \neg l$. Comme $\bar{\sigma}$ ne satisfait pas $\neg l$, elle satisfait c'. Et c' ne contient pas la variable p, donc $\mathcal{E}_{\sigma}(c') = \mathcal{E}_{\bar{\sigma}}(c') = \top$.
    On prouve ainsi que σ satisfait F_2'.

  Ainsi, σ satisfait $F_2' \wedge F_3$.

N.B. Utilité de l'équivalence prouvée ci-dessus : Pour trouver la valuation $\bar{\sigma}$ qui vérifie F, il faut et il suffit de trouver la valuation σ sur $\mathcal{V} \setminus \{p\}$ qui vérifie F' et de la compléter en posant $\bar{\sigma}(p) = \top$ si $l = p$ et $\bot$ sinon.

Et en particulier, s'il existe une unique valuation sur $\mathcal{V}$ qui satisfait F, alors il existe une unique valuation sur $\mathcal{V} \setminus \{p\}$ qui satisfait F'.

4. Appliquer l'algorithme de propagation unitaire à la formule F ci-dessous :

$$x^1_{0,0} \wedge \left(x^4_{2,2} \vee x^6_{3,6} \vee x^7_{7,7}\right) \wedge \left(\neg x^1_{0,0} \vee \neg x^6_{3,6}\right).$$

solution :

$$\begin{array}{l} F \\ \vDash \left(x^4_{2,2} \vee x^6_{3,6} \vee x^7_{7,7}\right) \wedge \neg x^6_{3,6} \quad \big\downarrow \textit{ en utilisant le littéral unitaire } x^1_{0,0} \\ \vDash x^4_{2,2} \vee x^7_{7,7}. \quad \big\downarrow \textit{ en utilisant le littéral unitaire } \neg x^6_{3,6} \end{array}$$

J'emploie la notation $F_1 \vDash F_2$ pour signifier « Toute valuation qui satisfait F_1 satisfait aussi F_2 et toute valuation qui satisfait F_2 peut être étendue à une valuation vérifiant F_1 en fixant la valeur d'un littéral supplémentaire. ».

5. (a) Sur l'exemple de la figure 1, déterminer la valeur de la case $(0, 7)$ en raisonnant à la manière d'un joueur de Sudoku.
   solution : La présence d'un 7 en case $(1, 5)$ empêche qu'il y ait un autre 7 dans la ligne 1.
   De même, il ne peut y avoir de 7 en ligne 2 à cause de la case $(2, 1)$.
   Ainsi, le seul emplacement possible pour le 7 du bloc 2 est à la case $(0, 7)$.

   (b) Retrouver ce résultat en appliquant l'algorithme de propagation unitaire, c'est-à-dire montrer que l'on peut, au terme d'un certain nombre de simplifications à partir de la formule initiale, déduire le littéral isolé $x^7_{0,7}$.
   On ne sélectionnera dans la formule $F_{initiale}$ que des clauses utiles à l'obtention du littéral isolé $x^7_{0,7}$.
   solution : Comme on l'a vu ci-dessus, les clauses utiles au remplissage de la case $(0, 7)$ sont :
   - $L2$ pour les lignes 1 et 2 et le chiffre 7. On va même réduire les clauses utilisées : nous avons besoin, avec les notations utilisées à la question 4b de $j_1 = 5$ pour la ligne 1 et $j_1 = 1$ pour la ligne 2, et $j_2 \in [\![6, 8]\!]$ dans les deux cas.
   - $B1$ pour le bloc 2 et le chiffre 7.
   - Les clauses unitaires $(x^7_{1,5})$ et $(x^7_{2,1})$.
   - Les clauses unitaires $\neg x^7_{0,6}$ et $\neg x^7_{0,8}$ (les deux cases sont déjà occupées, par 6 et 8 respectivement).

   Notons F la formule logique correspondante, donc F est formée de 9 des clauses de $F_{initiale}$ et :

$$F = x^7_{1,5} \wedge x^7_{2,1} \wedge \neg x^7_{0,6} \wedge \neg x^7_{0,8} \bigvee_{r=0}^{8} x^7_{indice(2,r)} \wedge \bigwedge_{j_2=6}^{8} \left(\neg x^7_{1,5} \vee \neg x^7_{1,j2}\right) \wedge \bigwedge_{j_2=6}^{8} \left(\neg x^7_{2,1} \vee \neg x^7_{2,j2}\right).$$

On applique l'algorithme de propagation unitaire :

$$
\begin{aligned}
F \vDash & \neg x^7_{0,6} \wedge \neg x^7_{0,8} \wedge \bigvee_{r=0}^{8} x^7_{indice(2,r)} \wedge \bigwedge_{j_2=6}^{8} \neg x^7_{1,j2} \wedge \bigwedge_{j_2=6}^{8} \vee \neg x^7_{2,j2} \quad \Big) \textit{ En utilisant les deux premières c} \\
\equiv & \neg x^7_{0,6} \wedge \neg x^7_{0,8} \\
& \left(x^7_{0,6} \vee x^7_{0,7} \vee x^7_{0,8} \vee x^7_{1,6} \vee x^7_{1,7} \vee x^7_{1,8} \wedge \vee x^7_{2,6} \vee x^7_{2,7} \vee x^7_{2,8}\right) \\
& \wedge \neg x^7_{1,6} \wedge \neg x^7_{1,7} \wedge \neg x^7_{1,8} \\
& \wedge \neg x^7_{2,6} \wedge \neg x^7_{2,7} \wedge \neg x^7_{2,8} \quad \Big) \textit{ En utilisant les 8 clauses unitair} \\
\vDash & x^7_{0,7}.
\end{aligned}
$$

6. Écrire une fonction `nouveau_lit_isole` qui, à partir d'une formule F , renvoie un littéral isolé de F .
Si F ne contient pas de tel littéral, la fonction renverra le littéral $x^{-1}_{(-1,-1)}$ qui n'est pas utilisé comme variable propositionnelle par ailleurs.
solution :

```
≠
let rec nouveau_lit_isole f=
    (* Renvoie un littéral isolé dans f. Renvoie X(-1,-1,-1) s'il n'y en a pas.*)
    match f with
        |[]         -> X(-1,-1,-1)
        |[l] :: _   -> l
        |_::q       -> nouveau_lit_isole q
;;
(* O(n) *)
```

7. Écrire une fonction `simplification` qui, à partir d'un littéral l et d'une formule F , renvoie la formule simplifiée F' après propagation du littéral l dans F .
solution :

```
≠
let non = function
    | X(i,j,k)      -> NonX(i,j,k)
    |NonX(i,j,k)    -> X(i,j,k)
;;

let rec simplification l f=
    (* l est un litéral isolé. Donc toute valuation qui satisfait f doit satisfaire l.
     À partir de là, elle satisfait toute clause contenant l, qui devient donc inutile.
     Et de plus, toute clause contenant non l ne peut être satisfaite via ce non l, on
         ↪ peut donc enlever ce littéral.
     *)
     match f with
        |[]     ->[]
        |clause :: q  when appartient l clause  -> simplification l q
        |clause :: q        -> (supprime (non l) clause) :: simplification l q
;;
```

8. Écrire une fonction `propagation` qui, à partir d'un tableau `t` représentant le sudoku et d'une formule F représentant les contraintes du sudoku, renvoie la formule obtenue par propagation unitaire. Cette fonction modifiera le tableau t quand de nouveaux littéraux isolés découverts au cours de l'algorithme permettent de déduire la valeur d'une case du sudoku.
solution :

```
≠
let rec propagation g f =
    let l = nouveau_lit_isole f in
        match l with
            |X(-1,-1,-1)    ->  f
            |X(i,j,k)       ->   g.(i).(j) <- k;
                                 propagation g (simplification l f)
            |NonX(i,j,k)    ->  propagation g (simplification l f)
```

```
;;


(* Version avec affichage pour suivre la résolution en direct *)
open Printf;;
let rec propagation ?(dec = "") g f =
  (* dec (argument facultatif, valeur par défaut "") décale les affichages. Utilisé dans
      ↪ la suite pour les essais lors de la résolution. *)
    let l = nouveau_lit_isole f in
        match l with
            |X(-1,-1,-1)    -> printf "%s -- propagation unitaire finie.\n" dec;  f
            |X(i,j,k)       -> printf "%s je mets %i case %i \n%!" dec k (10*i+j);
                               g.(i).(j) <- k;
                               propagation g (simplification l f)
            |NonX(i,j,k)    ->  propagation g (simplification l f)
;;
```

9. Que peut-on dire sur le tableau modifié t et la formule renvoyée par la fonction propagation dans le cas où l'algorithme de propagation unitaire permet de résoudre le sudoku ? Et dans le cas où il ne permet pas de le résoudre ?
solution :
   - Si l'algorithme a résolu le sudoku, alors t est entièrement rempli (plus de 0). Pour chaque case (i, j), on a trouvé un littéral isolé $X^k_{i,j}$, k étant la valeur à mettre dans la case (i, j). Ce littéral a alors été retiré de la formule lors de la propagation. Puis tous les $X^l_{i,j}$, pour $l \neq k$ ont été retirés aussi à cause des clauses $\neg x^k_{ij} \vee \neg x^l_{ij}$ de la condition K_2, clauses qui sont devenues des littéraux isolés après application de la propagation unitaire.
     Au final, la formule renvoyée est vide.
   - Dans le cas contraire, il reste des 0 dans t, et la formule renvoyée n'est pas vide. En effet, certaines cases n'ont pas encore été remplies, donc il existe $(i, j) \in [\![0, 8]\!] \times [\![0, 8]\!]$ tel que pour tout $k \in [\![1, 9]\!]$, $x^k_{i,j}$ n'est jamais apparu comme littéral isolé. La clause $\bigvee_{k=1}^{9} x^k_{i,j}$ (qui vient de $K1$) n'a donc pas été supprimée ni réduite à un singleton, il lui reste au moins deux littéraux.

10. On appelle taille d'une formule F le nombre total de littéraux apparaissant dans ses clauses. Évaluer le nombre d'opérations effectuées par les fonctions `nouveau_lit_isole`, `simplification`, puis par la fonction `propagation` quand elles s'appliquent à une formule de taille n. On donnera une estimation de la forme $O(f(n))$ que l'on justifiera.
solution : Soit F une formale de taille n.
   - La fonction `nouveau_lit_isole` parcourt une fois la liste des clauses, sa complexité est donc en $O(n)$ (pour être plus précis, on pourrait dire « en O(nombre de clauses) ».
   - La fonction `simplification` parcourt chaque clause, puis lance `appartient` sur chacune. Au final, chaque littéral est lu une fois, d'où une complexité en $O(n)$.
   - ◇ Le nombre d'appels récursifs à la fonction `propagation` est au plus de n car à chaque appel un littéral disparaît de la formule.
     ◇ À chacun de ces appels, les fonctions `simplication` et `nouveau_lit_isole` sont appelées, le reste est en $O(1)$. Ainsi chaque appel récursif coûte $O(n)$.

     Ainsi `propagation` appelé sur la formule F a une complexité en $O(n^2)$.

3.2 Règle du littéral infructueux

On décrit maintenant une autre méthode de déduction plus puissante combinant la propagation unitaire et une opération appelée règle du littéral infructueux décrite ci-dessous.

Étant donné une formule F en forme normale conjonctive et une variable propositionnelle x,

- si l'algorithme de propagation unitaire appliqué à la formule $F \wedge \neg x$ permet de déduire la clause vide alors on ajoute la clause x à F;
- si l'algorithme de propagation unitaire appliqué à la formule $F \wedge x$ permet de déduire la clause vide alors on ajoute la clause $\neg x$ à F .

1. Justifier formellement que si l'on peut déduire la clause vide à partir de $F \wedge \neg x$, alors $F \equiv F \wedge x$.
solution : La clause vide n'est pas satisfiable, donc si $F \wedge \neg x$ permet de déduire la clause vide, c'est que $F \wedge \neg x$ n'est pas satisfiable non plus, autrement dit $F \wedge \neg x \equiv \bot$.

On peut alors effectuer le simple calcul suivant :

$$\begin{aligned} F \equiv & F \wedge (x \vee \neg x) \\ \equiv & F \wedge x \vee F \wedge \neg x \\ \equiv & F \wedge x \vee \bot \\ \equiv & F \wedge x. \end{aligned}$$

2. Écrire une fonction `variables` qui, à partir d'une formule, renvoie la liste de ses variables sans doublons.
solution :

```
≠
 let rec variablesDansClause = function
    (* Renvoie la liste des variables utilisées dans une clause, sans doublon.*)
    | []    -> []
    |X(i,j,k)::q       -> ajout (X(i,j,k)) (variablesDansClause q)
    |NonX(i,j,k)::q    -> ajout (X(i,j,k)) (variablesDansClause q)
;;

let variables (f:formule)=
   List.fold_left
     concatSansDoublon
     []
     (List.map variablesDansClause f)
;;
```

3. Écrire une fonction `deduction` qui, à partir d'un tableau, d'une variable x et d'une formule F, renvoie 1 si la règle du littéral infructueux permet d'ajouter la clause x à F, -1 si elle permet d'ajouter la clause $\neg x$ à F et 0 sinon.
solution :

```
≠
let copie_matrice m=
     let n,p=(Array.length m), (Array.length m.(0)) in
     let res = Array.make_matrix n p 0 in
     for i=0 to n-1 do
        for j=0 to p-1 do
            res.(i).(j) <- m.(i).(j)
        done
    done;
    res;;


(* Encore un peu d'affichage... Ne vous préoccupez pas des deux fonctions ci-dessous ni
    ↪ des printf dans la suite.*)
let litt_to_string = function
  | X(i,j,k)  ->  string_of_int k ^" en case "^string_of_int (10*i+j)
  | NonX(i,j,k)  ->"pas "^ string_of_int k ^" en case "^string_of_int (10*i+j)
;;

let affiche_litt _ = function
  | X(i,j,k)  -> printf "%i en case %i" k (10*i+j)
  | NonX(i,j,k)  -> printf "pas %i en case %i" k (10*i+j)
;;

let deduction g x f =
  (* Renvoie 1 s'il est impossible que non x, -1 s'il est impossible que x, et 0 si la
      ↪ méthode utilisée ne permet par de conclure.*)
    let gc = copie_matrice g in
    (* Comme la fonction propagation modifie la grille fournie, nous lui fournissons
       ↪ uniquement une copie.*)
    if (
      printf "J'essaie %a " affiche_litt x;
      appartient [] (propagation ~dec:"   " gc ([x]::f))
```

```
    )then (
      printf "absurde!";
      -1
    )
    else if (
      printf "J'essaie %a " affiche_litt (non x);
      appartient [] (propagation ~dec:"   " gc ([non x]::f))
    )then (
      printf "absurde !";
      1
    )
    else 0
;;
```

4. Écrire une fonction `propagation2` qui, à partir d'un tableau t représentant le sudoku et d'une formule F représentant les contraintes du sudoku, met en œuvre cet algorithme. Cette fonction modifiera le tableau t selon la valeur des cases pouvant être déduites.
solution :

```
≠
let rec propagation2 g f=
    let f1 = propagation g f in

    let rec parcourtVariables=function
        (* Cette fonction auxiliaire est chargée de parcourir les variables de f1 pour
           ↪ voir s'il y a une déduction à faire.*)
        |[]     -> () (*Plus aucune déduction possible, fin de l'algo*)
        |x::q   ->
            begin
            match deduction g x f1 with
                | 0     -> parcourtVariables q
                | 1     ->  printf "déduction : %a \n" affiche_litt x;
                            propagation2 g ([x]::f1)
                | -1    ->  printf "déduction : %a\n " affiche_litt (non x);
                            propagation2 g ([non x]::f1)
                | _     -> failwith "résultat de déduction non valide"
            end
    in

    parcourtVariables (variables f1)
;;

let meth2 grille=
    propagation2 grille (formule_initiale grille);
    affiche grille
    ;;
```

Remarque : La méthode proposée ici consiste à faire *une* hypothèse et à voir si on aboutit à une contradiction. En général, cela suffit à terminer le sudoku. Cependant, il pourrait y avoir des cas de figure où il faudrait faire une deuxième hypothèse au sein de la première pour aboutir à une contradiction. Voire une hypothèse dans l'hypothèse dans l'hypothèse...

Ainsi, pour obtenir un algorithme qui conclut à coup sûr on devrait utiliser une fonction récursive, qui émet une hypothèse puis se ré-appelle elle-même pour voir si elle aboutit à une contradiction. Dans la méthode ci-dessus, la fonction ne se ré-appelle pas elle-même mais appelle la fonction `propagation` plus simple de la partie précédente.

La méthode générale récursive s'appelle une méthode « backtracking ».