TP : Résolution de sudoku

Le but du TP est de résoudre un sudoku, le fameux jeu japonais que tout le monde connaît. Pour ce faire, on va interpréter le problème de la résolution comme la recherche d'une valuation rendant vraie une certaine expression logique. Le TP qui suit est essentiellement une réécriture de l'épreuve 2014 du concours Centrale. Commencez par récupérer le fichier annexe, qui contient les types utilisés, quelques fonctions utiles et deux exemples de sudoku.

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 Préliminaires

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, voir figure 1.

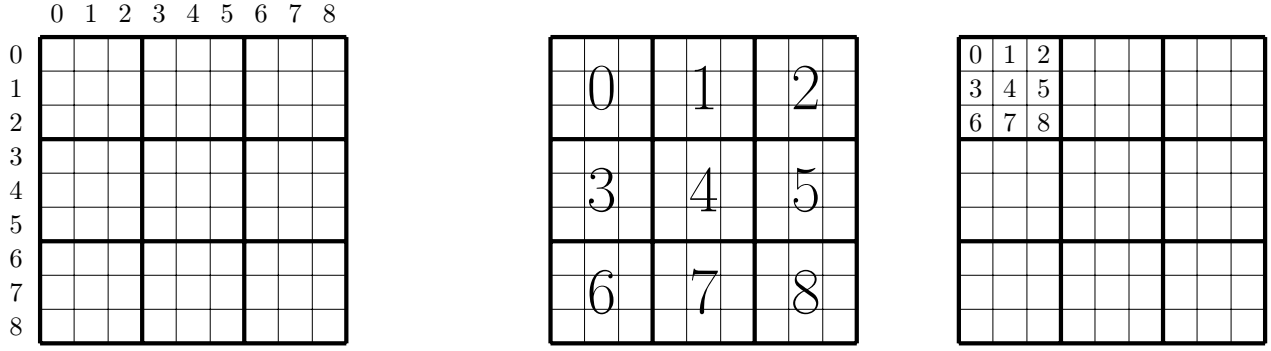

FIGURE 1: Les deux numérotations d'un sudoku : (a) par lignes et colonnes ; (b) numérotation des blocs ; (c) numérotation des régions du bloc 0.

Ainsi, une grille de sudoku est représentée par un tableau défini par `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)`. Une case non encore remplie est affectée de la valeur 0. Initialement, certaines cases sont déja remplies (elles contiennent une valeur différente de 0). Une fois le sudoku résolu, toutes les cases sont remplies, de manière à satisfaire les règles. Les valeurs présentes dans les cases initialement remplies n'ont pas changé, voir par exemple figure 2.

Question 1. Écrire une fonction `appartient x q` testant l'appartenance d'un élément `x` à une liste `q`. (Rappel : cette fonction existe déja, c'est `List.mem`).

Question 2. Écrire une fonction `supprime x q` de suppression de toutes les occurences de `x` dans une liste `q`.

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

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

FIGURE 2: (a) Le sudoku numéro 1 sous forme initiale ; (b) le même sudoku, résolu

Question 3. Écrire une fonction `ajoute x q` d'ajout d'un élément dans une liste *sans redondance* (si l'élément appartient déja à la liste, on la renvoie à l'identique).

Question 4. Écrire une fonction `indice (b,r)` donnant le couple (i, j) d'indices de ligne et de colonne associés à une case repérée par le bloc b et le numéro r dans le bloc. On fera usage de division euclidienne par 3 (simplement `/` en Caml) et de modulo 3 (rappel : `mod` pour modulo). *Remarque : une « recherche exhaustive » fonctionne, mais serait très mal vue dans une copie de concours !*

```
#indice (5,0) ;;
- : int * int = 3, 6
#indice (3,2) ;;
- : int * int = 3, 2
#indice (4,1) ;;
- : int * int = 3, 4
```

2 Codage de la formule initiale

On rappelle que, étant fixé un ensemble V de variables logiques :

— un littéral et un élément de la forme v ou $\neg v$, pour $v \in V$;

— une clause est un disjonction de littéraux ;

— la clause vide sera notée 0 (naturellement non satisfiable) ;

— une clause est dite unitaire si elle est de la forme p ou $\neg p$ avec p une variable logique, elle est dite de plus positive si elle est de la forme p et négative dans le cas contraire ;

— une expression logique est dite sous forme normale conjonctive si c'est une conjonction de clauses ;

— la forme normale conjonctive vide sera notée 1 : elle est tautologique.

Dans tout le problème, les variables propositionnelles seront les $x^k_{(i,j)}$ pour $(i, j, k) \in [\![0, 8]\!]^2 \times [\![1, 9]\!]$ avec la sémantique suivante : une valuation σ assigne la valeur vraie à $x^k_{(i,j)}$ si la case (i, j) de la grille contient la valeur k. L'ensemble V de variables est donc constituée des $9^3 = 729$ variables logiques $x^k_{(i,j)}$. À titre d'exemples :

— $x^5_{(0,0)} \vee x^5_{(8,0)} \vee x^5_{(0,8)} \vee x^5_{(8,8)}$ est une proposition logique exprimant le fait qu'un au moins des coins de la grille contient un 5 ;

— $\bigwedge_{i=0}^{8} \left(\bigvee_{k=1}^{9} x^k_{(i,7)} \right)$ exprime que chacune des valeurs de la colonne 7 est constituée des chiffres de 1 à 9 ;

— $\bigvee_{j=0}^{8} x^6_{\texttt{indice(3,j)}}$ exprime que l'une des cases du bloc numéro 3 contient le chiffre 6.

Pour la programmation, on utilise les types suivants :

```
type litteral = X of int * int * int | NonX of int * int * int ;;
type clause = litteral list ;;
type fnc = clause list ;;
```

— le type littéral permet d'encoder des littéraux : $x^1_{0,1}$ est codé comme `X(0,1,1)`, alors que $\neg x^9_{8,8}$ est codé comme `NonX(8,8,9)`.

— une clause est donnée comme la liste de ses littéraux (on rappelle que `==` définit simplement un synonyme) ;

— une formue logique sous forme normale conjonctive est donnée comme la liste des clauses qui la compose.

Ainsi, $x^7_{1,8} \wedge (x^3_{0,4} \vee \neg x^5_{2,7})$, qui est une formule en FNC, est représentée en Caml comme la liste de deux clauses, la première ayant un seul littéral et la seconde deux : `[[X(1,8,7)]; [X(0,4,3); NonX(2,7,5)]]`.

2.1 Formule logique décrivant la règle du jeu

Les règles (valables pour tout sudoku, indépendamment des chiffres déja présents) peuvent être décrites par les assertions ci-dessous :

— (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.

Question 5. ***1.*** Traduire (K1) sous forme d'une formule en FNC. Elle doit avoir 81 clauses, chacune ayant 9 littéraux.

2. Écrire une fonction `case1 ()` sans argument, renvoyant la formule associée à (K1).

```
#case1 () ;;
- : litteral list list =
 [[X (8, 8, 9); X (8, 8, 8); X (8, 8, 7); X (8, 8, 6); X (8, 8, 5);
   X (8, 8, 4); X (8, 8, 3); X (8, 8, 2); X (8, 8, 1)];
  [X (...); ...]; ...]
#  List.length (case1 ()) ;;
- : int = 81
```

Question 6. Faire de même avec les conditions (L1), (C1) et (B1), et des fonctions `lig1`, `col1` et `bloc1`.

Question 7. Pour une case (i, j) de la grille, et k et k' deux entiers tels que $1 \leq k < k' \leq 9$, le fait que la case (i, j) ne contienne pas à la fois le chiffre k et le chiffre k' s'exprime comme la clause à deux littéraux $\neg x^k_{i,j} \vee \neg x^{k'}_{i,j}$.

1. En déduire une formule en FNC pour (K2), elle doit avoir 2916 clauses.

2. Écrire une fonction `case2()` produisant cette formule logique.

```
#case2 () ;;
- : litteral list list =
 [[NonX (8, 8, 8); NonX (8, 8, 9)]; [NonX (8, 8, 7); NonX (8, 8, 9)];
  [NonX (8, 8, 7); NonX (8, 8, 8)];  [...]; ...]
#List.length (case2 ()) ;;
- : int = 2916
```

Question 8. Faire de même avec conditions (L2), (C2) et (B2), et des fonctions `lig2`, `col2` et `bloc2`.

Question 9. Déduire des fonctions précédentes une fonction `fnc_regles()` produisant la FNC associée aux règles, elle contient donc 11988 clauses. On rappelle que `@` est la concaténation de listes.

2.2 Formule logique décrivant la grille initiale

Le sudoku possède des cases déja remplies : si dans la case (i, j) est déja inscrit le chiffre k, on peut ajouter la clause unitaire positive $x^k_{i,j}$ à notre formule logique. Rajouter toutes les clauses de cette sorte suffit à écrire une formule logique satisfiable avec une unique distribution de vérité, associée à la solution au sudoku. Néanmoins les algorithmes de résolution que l'on va écrire par la suite faisant usage de clauses unitaires, on va ajouter un peu de redondance pour accélérer le processus de résolution.

Question 10. Aux clauses unitaires positives $x_{i,j}^k$ (le chiffre k est marqué en case (i,j)), on peut ajouter les clauses unitaires négatives $\neg x_{i,j}^\ell$ pour tout $\ell \in [\![1,9]\!] \backslash \{k\}$. Écrire une fonction `donnees t` prenant en entrée un sudoku (sous la forme d'une matrice 9×9) et renvoyant une formule logique en FNC constituée des clauses unitaires $x_{i,j}^k$ et $\neg x_{i,j}^\ell$ avec (i,j) une case déja remplie.

```
#donnees sudoku_1 ;;
- : litteral list list =
 [[NonX (8, 6, 9)]; [NonX (8, 6, 8)]; [NonX (8, 6, 7)]; [NonX (8, 6, 6)];
  [NonX (8, 6, 5)]; [NonX (8, 6, 4)]; [X (8, 6, 3)]; [NonX (8, 6, 2)];
 ...]
#List.length (donnees sudoku_1) ;;
- : int = 288
```

La formule obtenue contient un nombre de clauses égal à 9 fois le nombre de cases initialement remplies (ici 9×32).

Question 11. Inversement, si la case (i,j) n'est pas déja remplie (elle contient 0), les règles du sudoku imposent que l'on ne peut y mettre aucun chiffre apparaissant sur la même ligne, la même colonne, ou le même bloc. On peut donc rajouter autant de clauses unitaires négatives qu'il y a de valeurs interdites pour la case (i,j).

1. Écrire une fonction `interdites_ij t (i,j)` prenant en entrée un sudoku et les indices i et j de ligne et colonne d'une case non remplie, et renvoyant la conjonction des clauses unitaires négatives associées aux valeurs interdites en (i,j).

```
#interdites_ij sudoku_1 (0,0) ;;
- : litteral list list =
 [[NonX (0, 0, 1)]; [NonX (0, 0, 8)]; [NonX (0, 0, 7)]; [NonX (0, 0, 9)];
  [NonX (0, 0, 6)]; [NonX (0, 0, 2)]; [NonX (0, 0, 3)]]
#interdites_ij sudoku_1 (2,8) ;;
- : litteral list list =
 [[NonX (2, 8, 9)]; [NonX (2, 8, 4)]; [NonX (2, 8, 6)]; [NonX (2, 8, 8)];
  [NonX (2, 8, 1)]]
```

On pourra utiliser le fait que le numéro de bloc associé à la case (i,j) est $3 \times \left\lfloor \frac{i}{3} \right\rfloor + \left\lfloor \frac{j}{3} \right\rfloor$. La formule produite possède au plus 8 clauses unitaires négatives.

2. En déduire une fonction `interdites t` rassemblant ces clauses unitaires négatives. La formule produite possède au plus 8 clauses unitaires négatives par case non remplie.

Question 12. Déduire des deux questions précédentes une fonction `fnc_grille t` donnant la FNC associée aux valeurs déja remplies dans la grille. La formule produite a moins de $9^3 = 729$ clauses.

```
#List.length (fnc_grille (sudoku_1)) ;;
- : int = 570
```

Question 13. Déduire de toutes les questions précédentes une fonction `fnc_initiale t` donnant la FNC associée aux règles et aux valeurs déja remplies.

3 Résolution d'une grille de sudoku

3.1 Règle de propagation unitaire

La formule logique produite par `fnc_initiale t` est satisfiable (car le sudoku a une solution), et il n'y a qu'une distribution de vérité sur les $x_{i,j}^k$ qui la satisfasse (si le sudoku a bien une unique solution). On souhaite la déterminer. Étant donné qu'il y a $9^3 = 729$ variables logiques, il est absolument exclus d'utiliser l'algorithme du TP 3 pour la trouver : il est impossible d'effectuer de l'ordre de 2^{729} opérations. En effet, à raison de 10^{15} opérations par seconde [1], il faudrait beaucoup plus que l'âge de l'univers pour résoudre notre sudoku. On présente ici une solution polynomiale, qui ne permet pas de conclure à tous les coups, mais qui simplifie la formule en prenant en compte les clauses unitaires.

Soit F une formule logique en FNC qui présente une clause unitaire, c'est-à-dire comportant un unique littéral ℓ, qu'on appel *littéral isolé*). Alors :

— nécessairement une valuation qui satisfait F a la valeur 1 sur ℓ ;

1. ce qui est bien plus rapide que le meilleur PC actuel.

— on peut supprimer toutes les clauses de F qui contiennent ℓ ;

— on peut supprimer le littéral $\neg\ell$ de toutes les clauses qui le contiennent (remarque : si une clause est réduite à $\neg\ell$, on la remplace par la clause vide 0 (liste vide en Caml) non satisfiable. La formule F n'est alors pas satisfiable).

Ce procédé s'appelle la *propagation unitaire*. On peut répéter ce procédé tant qu'il reste des clauses unitaires dans la formule. Par exemple avec $F = x^1_{(0,0)} \wedge (x^4_{(2,2)} \vee x^6_{(3,6)} \vee x^7_{(7,7)}) \wedge (\neg x^1_{(0,0)} \vee \neg x^6_{(3,6)})$:

— le littéral $x^1_{(0,0)}$ est isolé. Sa suppression mène à $F' = (x^4_{(2,2)} \vee x^6_{(3,6)} \vee x^7_{(7,7)}) \wedge \neg x^6_{(3,6)}$;

— le littéral $\neg x^6_{(3,6)}$ est maintenant isolé. Sa suppression mène à $F'' = (x^4_{(2,2)} \vee x^7_{(7,7)})$;

— il n'y a plus de littéral isolé : le procédé s'arrête.

Question 14. Écrire une fonction `nouveau_lit_isole f` renvoyant un littéral isolé d'une formule en FNC. S'il n'y en a pas, on renverra $x^{-1}_{(-1,-1)}$ qui n'est pas utilisé par ailleurs.

```
#nouveau_lit_isole (fnc_initiale (sudoku_1)) ;;
- : litteral = NonX (8, 6, 9)
```

(Remarque : j'obtiens $\neg x^9_{8,6}$ ici, cela dépend de la manière dont vos fonctions sont écrites, puisqu'il y en a 570 !)

Question 15. Écrire une fonction `simplification l f` qui simplifie une formule logique `f` en FNC à partir du littéral isolé `l`.

```
#List.length (fnc_initiale (sudoku_1)) ;;
- : int = 12558
#List.length (simplification (NonX(8,6,9)) (fnc_initiale (sudoku_1))) ;;
- : int = 12525
```

Question 16. Écrire une fonction (récursive) `propagation t f` prenant en entrée un sudoku, la formule logique associée `f`, et appliquant l'algorithme de propagation unitaire jusqu'à ce que la formule passée en paramètre ne possède plus de clause unitaire. On modifiera au passage le sudoku `t` pour prendre en compte les nouveaux chiffres découverts. Tester votre algorithme sur le `sudoku_1`.

```
#propagation (sudoku_1) (fnc_initiale (sudoku_1)) ;;
- : litteral list list = []
#affiche_sudoku sudoku_1 ;;
 _____________________
| 4 8 3 | 9 2 1 | 6 5 7 |
| 9 6 7 | 3 4 5 | 8 2 1 |
| 2 5 1 | 8 7 6 | 4 9 3 |
|---------------------------
| 5 4 8 | 1 3 2 | 9 7 6 |
| 7 2 9 | 5 6 4 | 1 3 8 |
| 1 3 6 | 7 9 8 | 2 4 5 |
|---------------------------
| 3 7 2 | 6 8 9 | 5 1 4 |
| 8 1 4 | 2 5 3 | 7 6 9 |
| 6 9 5 | 4 1 7 | 3 8 2 |
|---------------------------
- : unit = ()
```

3.2 Règle du littéral infructueux

Le sudoku pris en exemple était entièrement résoluble par propagation unitaire, mais ce n'est pas le cas en général, comme on peut le voir sur l'exemple de la figure 3.

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 donnés 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.

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

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

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

FIGURE 3: (a) Le sudoku numéro 2 ; (b) le même sudoku, après application de l'algorithme de propagation unitaire ; (c) le sudoku résolu.

Formellement, cela se justifie ainsi : la clause vide étant une formule antilogique, on a $F \wedge \neg x \Rightarrow 0 \equiv \neg(F \wedge \neg x) \vee 0 \equiv \neg F \vee x$. Ainsi $F \wedge (F \wedge \neg x \Rightarrow 0) \equiv F \wedge (\neg F \vee x) \equiv F \wedge x$, car $F \wedge \neg F \equiv 0$. De même $F \wedge (F \wedge x \Rightarrow 0) \equiv F \wedge \neg x$.

Question 17. Écrire une fonction `variables f` renvoyant toutes les variables présentes dans la formule en FNC `f`, sans doublons. On pourra utiliser la fonction `flatten` fournie, qui concatène une liste de listes en une seule liste. Remarque : on veut les variables, pas les littéraux. Il ne devra pas y avoir de littéraux de la forme `NonX(i,j,k)` dans le résultat.

Question 18. Écrire une fonction `deduction t x f` prenant en entrée un sudoku, une variable `x` et une formule en FNC `f`, et renvoyant :

— 1 si la règle du littéral infructueux permet d'ajouter la clause unitaire x à `f` ;

— -1 si elle permet d'ajouter la clause unitaire négative $\neg x$;

— 0 sinon.

On utilisera la fonction `copie_matrice` pour ne pas modifier le sudoku.

Le deuxième algorithme de résolution consiste à appliquer tant que possible l'algorithme de propagation unitaire, puis à appliquer la règle du littéral infructueux jusqu'à trouver une nouvelle clause unitaire x ou $\neg x$ que l'on peut ajouter à f. On reprend alors l'algorithme de propagation unitaire, etc... Le processus s'arrête lorsque ni la règle du littéral infructueux ni l'algorithme de propagation unitaire ne progressent (en pratique c'est parce qu'on a résolu le sudoku !).

Question 19. Écrire un algorithme (récursif !) `propagation2 t f` mettant en œuvre cette stratégie, et modifiant le sudoku avec les nouvelles valeurs déduites.

Question 20. En déduire une fonction `sudoku t` prenant en entrée un sudoku et modifiant le sudoku au maximum avec la stratégie précédente.

```
#sudoku (sudoku_2) ; affiche_sudoku sudoku_2 ;;
_____________________
| 8 9 1 | 2 3 4 | 6 7 5 |
| 3 2 5 | 6 8 7 | 4 1 9 |
| 4 7 6 | 9 1 5 | 3 2 8 |
---------------------------
| 9 1 4 | 7 5 2 | 8 6 3 |
| 2 5 7 | 3 6 8 | 1 9 4 |
| 6 8 3 | 1 4 9 | 7 5 2 |
---------------------------
| 1 4 8 | 5 2 6 | 9 3 7 |
| 7 3 2 | 4 9 1 | 5 8 6 |
| 5 6 9 | 8 7 3 | 2 4 1 |
---------------------------
- : unit = ()
```

Expérimentalement, la règle de propagation unitaire permet de résoudre les sudokus les plus faciles et environ la moitié des sudokus les plus difficiles. À notre connaissance, il n'existe pas de sudoku ne pouvant être résolu intégralement à l'aide de la règle du littéral infructueux.

Question 21. Résoudre le problème 96 du site Project Euler.