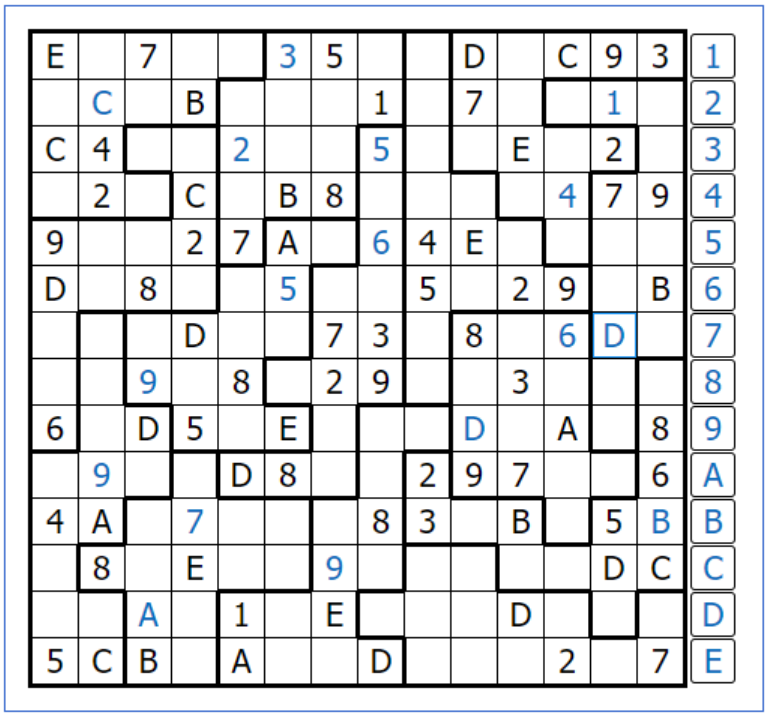
JIGSAW SUDOKU

CHEA Alexandre 19007001 - ORRY Eric 19007639

Mars 2021

Ce devoir est réalisé dans le cadre du cours de
Programmation Déclarative et base de données.



Licence Informatique - Département STN
Première année
Université Paris 8
2020-2021

Table des matières

1 Présentation du Projet

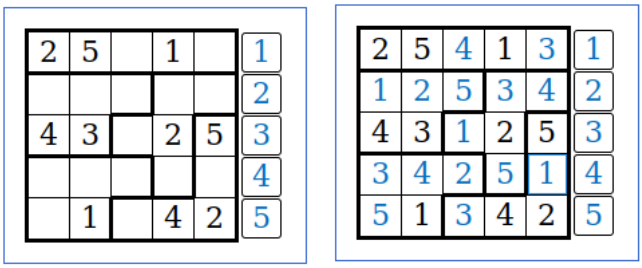
1.1 Introduction

Le projet consiste à créer un programme en prolog permettant de résoudre une grille de Jigsaw Sudoku.

1.2 Règle du Jigsaw Sudoku

Le Jigsaw Sudoku est un casse-tête avec des règles simples, qui fait reflechir.

Les règles de Jigsaw Sudoku sont simples :
1. Nous avons une grille en n * n
2. Cette grille comporte donc n lignes, n colonnes et n blocs de n cases.
Contrairement au sudoku normal, les blocs ont des formes irrégulières et différentes selon les grilles.
3. Les cases peuvent être remplies que par des valeurs allant de 1 a n.
4. Les contraintes sont les suivantes :
- Chaque ligne doit comporter que des valeurs différentes.
- Chaque colonne doit comporter que des valeurs différentes.
- Chaque blocs doit comporter que des valeurs différentes.
Par exemple, voici un Jigsaw Sudoku 5x5 facile :



Nous avions utilisé le site puzzle-jigsaw-sudoku.com pour trouver nos puzzles et essayer les solutions que propose notre programme.

1.3 Réalisation

Pour commencer, nous avons commencé par essayer de résoudre une grille avec la taille la plus petite possible. Sur le site *puzzle-jigsaw*, la taille la plus petite qu'on trouve est 5x5. Nous avons donc décidé d'essayer par cette taille.
Après avoir réussi à résoudre n'importe quel sudoku de taille 5x5, nous nous sommes intéressé à une grille de taille 7x7, puis de façon plus générale, une grille de taille n x n.

2 Un programme qui résout une grille 5x5

2.1 Première version

Afin de résoudre le puzzle jigsaw sudoku, nous nous sommes d'abord intéressé et poser la question suivante : Comment résoudre, en prolog, un sudoku normal ?

Nous avons donc pris les règles basiques d'un sudoku normal et avons créé les prédicats suivants :

nombre(X) : qui permet de valider une valeur dans une case. Dans une grille 5x5, les nombres valides vont de 1 à 5, nous avons donc entrer une liste [1,2,3,4,5].

```
nombre(X) :-member(X,[1,2,3,4,5])
```

```
?- nombre(1).
true.

?- nombre(6).
false.

?- nombre(X).
X = 1 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5.
```

diff([A|B]) : qui permet de verifier que dans une liste, tous les éléments sont différents

```
diff([]).
diff([L|Ls]) :-not(member(L,Ls)), diff(Ls).
```

```
?- diff([1,2,3,4,5]).
true.

?- diff([1,2,2,4,5]).
false.
```

remplir([A,B,C,D,E]) : qui permet de verifier que dans une liste [A,B,C,D,E], les cinq éléments sont bien des nombres valides et sont tous différents.

```
remplir([A,B,C,D,E]) :-nombre(A), nombre(B), nombre(C), nombre(D),
                       nombre(E), diff([A,B,C,D,E]).
```

```
?- remplir([1,2,3,4,5]).
true.

?- remplir([1,3,4,6,1]).
false.

?- remplir([1,1,1,5,3]]).
false.

?- remplir([1,3,A,2,4]).
A = 1.

?- remplir([2,A,B,3,5]).
A = 1,
B = 4 ;
A = 4,
B = 1.
```

Avec ces trois prédicats, nous pouvons créer **sudoku([Grille])** qui vérifie que chaque ligne est différente, chaque colonne est différente et chaque bloc est différent. Pour celà, il faut dire à notre prédicat **remplir** de remplir chacune des lignes et ensuite, de verifier avec **diff** que les colonnes et blocs sont différents.

```
sudoku([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5,
        D1,D2,D3,D4,D5,E1,E2,E3,E4,E5]) :-

    /* Remplissage des lignes */
    remplir([A1,A2,A3,A4,A5]),
    ...
    remplir([E1,E2,E3,E4,E5]),

    /* Vrification des colonnes */
    diff([A1,B1,C1,D1,E1]),
    ...
    diff([A5,B5,C5,D5,E5]),

    /* Bloc : A CHANGER SELON LES PUZZLES */
    diff([A1,A2,A3,A4,A5]),
    diff([B1,C1,C2,D1,D2]),
    diff([B2,B3,B4,B5,C5]),
    diff([E1,E2,E3,D3,C3]),
    diff([C4,D4,D5,E4,E5]),

    /* Affichage de la grille */
    print_ligne([A1,A2,A3,A4,A5]),
    ...
    print_ligne([E1,E2,E3,E4,E5]).
```

Voir code entier en annexe 5.1

Nous remarquons qu'avec ce programme, lorsqu'on veut résoudre plusieurs grilles différentes, il faut modifier les cases des blocs directement dans le programme, dans le prédicat principal **sudoku**.

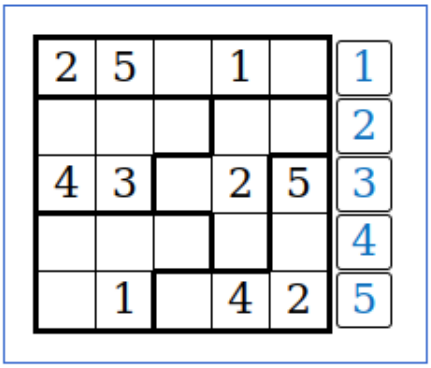
Ce qui nous amène à ajouter quelques prédicats pour améliorer notre programme.

Vous trouverez un exemple d'utilisation de la première version dans les exemples cf. 4.1

2.2 Deuxième version

Voir code entier en annexe 5.2

Pour améliorer la première version de notre programme, il faut entrer un deuxième paramètre à notre prédicat **sudoku** qui indique à notre programme, à quel bloc appartient chaque case.
Nous avons donc opté pour attribuer à chaque case, une information qui donne le bloc auquel il appartient. Voici un exemple, sur une grille de sudoku :



a	a	a	a	a
b	b	b	c	c
b	b	c	c	d
e	e	e	c	d
e	e	d	d	d

En prolog, nous écrirons ceci dans notre prédicat.

```
sudoku(
[_,_,_,4,3,3,_,2,_,_,_,_,_,_,_,_,_,4,_,5,4,2,_,_,_],
[a,a,a,a,a,b,b,b,c,c,b,b,c,c,d,e,e,e,c,d,e,e,d,d,d]).
```

Les deux arguments sont donc deux listes :
- Dans la première liste, les valeurs qui apparaissent dans l'énoncé grille. On met un underscore lorsque la grille est vide.
- Dans la deuxième liste, on donne des noms aux blocs et donne donc pour chaque case dans l'ordre, à quel bloc la case appartient.

Pour celà, nous avons donc créer **infocase** qui permet d'identifier à quel bloc appartient la case.
Nous pouvons donc entrer en paramètre dans le nouveau sudoku, une liste de coordonnées.
Dans ce prédicat, nous entrons la liste de case dans l'ordre et une liste de valeur indiquant dans quel bloc appartient la case, Y nous renvoie une liste de sous-liste avec en premier élément la case, et le deuxième son bloc.

```
infocase([A],[B],[[A,B]]).
infocase([A|As],[B|Bs],[[A,B]|Y]) :-infocase(As,Bs,Y).
```

```
?- listcase([1,2,3,1,3,4,6],[a,b,e,f,g,g,d],Y).
Y = [[1,a],[2,b],[3,e],[1,f],[3,g],[4,g],[6,e]].
true

?- listcase([1,2,3],[a,b,c,d],Y).
false.
```

Il suffit donc maintenant de réunir tous les éléments ayant le même deuxième élément dans une liste. Pour cela, nous avons créé **meme2**. Nous entrons dans ce prédicat, une valeur X et une liste de sous-liste comportant deux éléments. Y va nous renvoyer une liste de premier élément des sous-listes qui ont le même X en deuxième élément.

```
meme2(_,[],[]).
meme2(X,[[A,X]|Ls],[A|Y]) :-meme2(X,Ls,Y).
meme2(X,[[_,B]|Ls],Y) :-X \= B,meme2(X,Ls,Y).
```

```
?- meme2(a,[[1,a],[2,b],[3,a],[4,3]],Y).
Y = [1,3].
```

Grâce aux deux précédents prédicats, nous pouvons créer **listbloc** qui permet d'avoir une liste de sous-liste où chaque sous-liste correspond à la liste des cases d'un bloc.

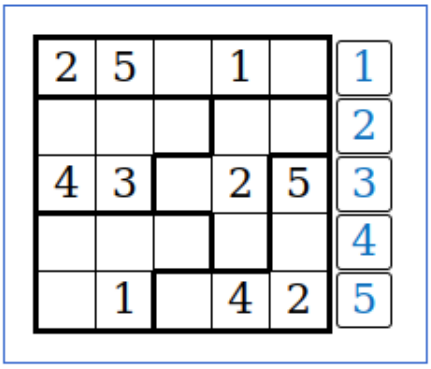
```
listbloc(_,[],[]).
listbloc([],L,[]) :-L \= [].
listbloc(L,[A|Ab],Y) :-meme2(A,L,Y1),
                       delete(Ab,A,Ac),
                       listbloc(L,Ac,Y2),
                       append([Y1],Y2,Y).
```

```
?- listbloc([1,2,3,4,5,6,1,2,3,7],[a,b,b,a,2,2,b,a,a,c],Y).
Y = [[1,4,2,3],[2,3,1],[5,6],[7]].
```

Grâce aux prédicats précédemment créés, nous pouvons donc résoudre notre problème de la première version en supprimant le changement des blocs à la main. Il suffit de récupérer la liste des blocs et de vérifier chaque sous-liste est bien conforme à la règle de l'unicité.

2.3 Essayons..

Nous allons essayer de résoudre le puzzle affiché en 1.2.
Nous entrons donc dans notre terminal sudoku(Grille,Bloc).avec :
Grille, une liste de valeur qu'on entre en ligne, on met la valeur lorsqu'elle est connu ou un underscore si on ne connait pas la valeur.
Bloc, une liste de valeur qui indique le nom du bloc de la case. (Les blocs peuvent avoir une valeur ou une lettre).



Nous avons donc le résultat suivant dans notre terminal :

```
?- sudoku([2,5,_,1,_,_,_,_,_,_,4,3,_,2,5,_,_,_,_,_,_,_,1,_,4,2],[a,a,a,a,a,b,b,b,c,
c,b,b,c,c,d,e,e,e,c,d,e,e,d,d,d]).
2 5 4 1 3
1 2 5 3 4
4 3 1 2 5
3 4 2 5 1
5 1 3 4 2
true
```

3 Un programme généralisé

Voir code en annexe 5.3.

Après avoir réussi à résoudre une grille de taille 5x5, nous allons nous intéresser à la résolution d'une grille de taille n x n..

3.1 La fonction principale en détail

Pour commencer, nous allons initier le prédicat qui va résoudre la grille. Le prédicat **sudoku** est composé de 3 variables.

1. **Dim** : La variable "Dim" pour dimension permet de connaître la taille du sudoku. Par exemple si le sudoku fait 5x5 , alors l'utilisateur devra indiquer 5.
2. **Sudo** : La variable "Sudo" est la grille du sudoku à résoudre. L'utilisateur entrera la grille du sudoku en indiquant les cases sans valeurs avec des "_".
3. **Block** : La variable "Block" pour nommer les bloc des cases permet d'indiquer au prédicat les limites des cases dans la grille du Jigsaw Sudoku.

A l'aide de la dimension de la grille, nous allons vérifier que la liste **Sudo** et que la liste **Block** ont bien le même nombre d'élément que la dimension au carré. Puis à l'aide de la dimension indiqué par l'utilisateur, nous allons créer la liste des valeur possibles dans un sudoku à n dimension.

```
valeur(1,[1]).
valeur(X,[X|Y]) :-X > 0,
                  X1 is X-1,
                  valeur(X1,Y).
```

```
/* Pour une liste de dimension 5 */
?- valeur(5,X).
X = [1, 2, 3, 4, 5]

/* Pour une liste de dimension 9 */
?- valeur(9,Y).
Y = [1, 2, 3, 4, 5, 6, 7, 8, 9] .
```

Comme pour un sudoku de taille 5x5, nous cherchons a récupéré tous les blocs dans une seule liste. Nous allons donc utiliser les prédicats **infocase**, **meme2** et **listblock** afin d'avoir la liste de sous-liste ne comportant que les blocs. *(Explication des prédicats dans la partie 2.2)*

Ensuite, nous avons créé un prédicat **ligne** qui récupère toutes les lignes et les met dans des sous-listes. Nous indiquons que chaque ligne comporte n éléments.
Nous indiquons le n lorsque on entre la dimension de la grille. En effet, pour une grille de taille 7x7, n = 7, les lignes font donc 7 éléments, les colonnes et les cases aussi.
Pour récuperer les lignes, nous recuperons les n premiers éléments que l'on stockons dans une liste, puis nous effaçons les n premiers éléments et recommençons avec la liste modifiée jusqu'a ce qu'on arrive à une liste vide.

```
ligne(X,L,[L]) :-longueur(L,R),
                 X >= R.
ligne(X,L,Y) :-longueur(L,R),
               X < R,
               recup(X,L,L1),
               remove(X,L,L2),
               ligne(X,L2,Y1),
               append([L1],Y1,Y).
```

De même pour les colonnes, nous avons crée un prédicat **colonne** qui récupère toutes les colonnes et les met ans des sous-listes. Pour celà, nous récupérons un élément tous les n éléments, puis on les supprime et on rappelle le prédicat qui récupère tous les n-1 éléménts. On répète cette opération jusqu'a ce qu'on se retrouve à un n = 1, où l'on retourne la liste modifié entièrement.

```
colonne(1,L,[L]).
colonne(X,L,Y) :-X > 1,
                 col(X,L,R1),
                 retire(X,L,R2),
                 X1 is X-1,
                 colonne(X1,R2,Y1),
                 append([R1],Y1,Y).
```

A ce stade, nous avons trois listes de sous-liste qui comportent les lignes, les colonnes et les blocs. Pour finir la résolution, il suffit de remplir une liste de sous-liste avec le prédicat **valo** et ensuite de verifier que les deux autres listes sont différentes !

La fonction majeure sera donc au final constitué comme ceci :

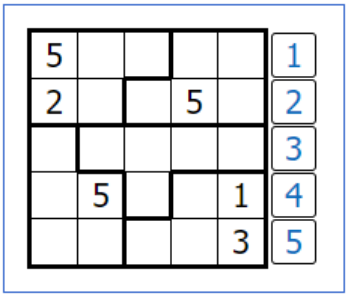
```
sudoku(Dim,Sudo,Block) :-carre(Dim,Dimen1),
                         longueur(Sudo,Dimen1),
                         longueur(Block,Dimen1),
                         valeur(Dim,Valeur),
                         infocase(Sudo,Block,L1),
                         listblock(L1,Block,Block2),
                         ligne(Dim,Sudo,Ligne),
                         colonne(Dim,Sudo,Colonne),
                         valo(Block2,Valeur),
                         diffs(Ligne),
                         diffs(Colonne),
                         aff2(Ligne).
```

3.2 Listing des prédicats utilisés

- Prédicat **valeur(X,Y)** : Y renvoie une liste de X éléments, les éléments sont des valeurs allant 1 à X.
- Prédicat **aff2([L|Ls])** : Soit une liste de sous liste, ce prédicat affiche une sous-liste par ligne
- Prédicat **diffs([L|Ls])** : Vérifie que chacun des sous listes de [L|Ls] comportent bien des éléments différents.
- Prédicat **liste([A|Ab],L)** : Vérifie que les éléments de la liste L appartiennent tous à la liste L.
- Prédicat **valo([A|C],Valeur)** : Vérifie que :
  1. les valeurs de chaque sous liste est dans la liste Valeur.
  2. les valeurs à l'interieur d'une sous liste sont différentes.
- Prédicat **carre(N,Y)** :- Y est le carré de N.
- Prédicat **longueur([_|Ls],Y)** : Y est la longueur de [_|Ls]
- Prédicat **recup(X,[L|Ls],[L|Y])** : Vérifie que Y est bien la liste des X premiers éléments de [L|Ls]
- Prédicat **remove(X,[_|Ls],Y)** : Vérifie que Y est bien la liste [_|Ls] sans les X premiers éléments.
- Prédicat **ligne(X,L,Y)** : Vérifie que Y est bien la liste L divisé en sous liste de X éléments.
- Prédicat **col(X,[L|Ls],[L|Y])** : Vérifie que Y est bien la liste des éléments en sautant à chaque fois X éléments
- Prédicat **retire(X,L,Y)** : Vérifie que Y est bien la liste L en ayant retiré un élément tous les X éléments
- Prédicat **colonne(X,L,Y)** : Récupère la liste de sous listes où cha que sous-liste est une colonne (X étant le nombre d'élément par ligne)
- Prédicat **infocase([A|As],[B|Bs],[[A,B]|Y])** : Vérifie que Y est bien une liste de sous liste avec chaque sous liste de deux éléments comportant A et B qui sont à la même position dans les deux listes.
- Prédicat **meme2(X,[[A|X]|Ls],Y)** : Soit une liste de sous liste de deux éléments A et X, renvoie une liste de Y composé des A qui ont le même X (indiqué au lancement du prédicat)
- Prédicat **listblock(L,[A|Ab],Y)** : Rend Y, une liste de sous-liste où les sous-listes sont les différents blocs du sudoku

3.3 Essayons..

Nous allons essayer de résoudre le sudoku suivant :



Nous entrons donc dans notre terminal sudoku(Dim,Grille,Bloc).avec :
Dim, la dimension d'une ligne ou d'une colonne, dans ce cas, nous mettrons 5.
Grille, une liste de valeur qu'on entre en ligne, on met la valeur lorsqu'elle est connu ou un underscore si on ne connait pas la valeur.
Bloc, une liste de valeur qui indique le nom du bloc de la case. (Les blocs peuvent avoir une valeur ou une lettre).

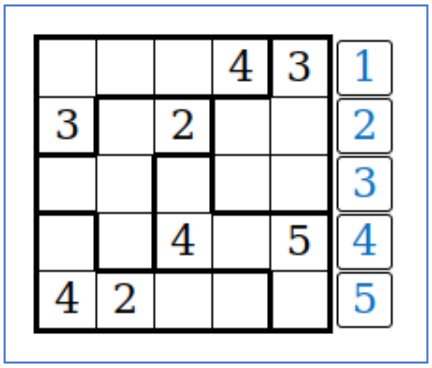
Nous avons donc le résultat suivant dans notre terminal :

```
?- sudoku(5,[5,_,_,_,_,2,_,_,5,_,_,_,_,_,_,_,_,5,_,_,1,_,_,_,_,3],
[a,a,a,b,b,a,a,b,b,b,c,d,d,d,d,c,c,d,e,e,c,c,e,e,e]).
5 1 4 3 2
2 3 1 5 4
3 4 2 1 5
4 5 3 2 1
1 2 5 4 3
true
```

4 Quelques essais ..

4.1 .. avec un sudoku de taille 5x5 (Première version)

Voici un Jigsaw sudoku de taille 5x5 :



Pour différencier les différentes familles de bloc, nous utilisons le prédicat **diff** ce qui permet de ne pas avoir une répétition de même nombre.

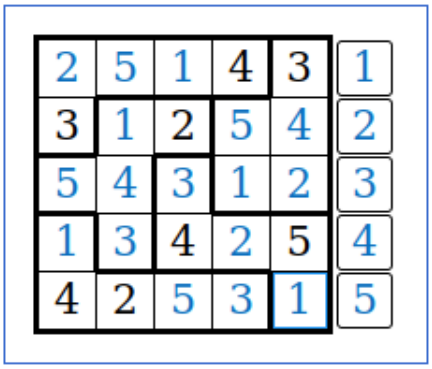
```
/* Case : A CHANGER SELON LES PUZZLES */
    diff([A1,A2,A3,A4,B1]),
    diff([A5,B4,B5,C4,C5]),
    diff([B2,B3,C1,C2,D2]),
    diff([C3,D3,D4,D5,E5]),
    diff([D1,E1,E2,E3,E4]),

solver() :- sudoku(
                  [_,_,_,4,3,
               3,_,2,_,_,
               _,_,_,_,_,
               _,_,4,_,5,
               4,2,_,_,_]).
```

Le terminal nous renvoie donc la réponse suivante :

```
?- solver().
2 5 1 4 3
3 1 2 5 4
5 4 3 1 2
1 3 4 2 5
4 2 5 3 1
true .
```

(Pour information, le temps de calcul est de 3,52 seconde).

Ce qui correspond bien à la réponse attendu au Jigsaw sudoku.



4.2 .. avec un sudoku de taille 5x5 (Deuxième version)

Dans cette nouvelle version du 5x5, nous utilisons les coordonnées des blocs.

```
solve(10) :- sudoku(
     [_,_,_,4,3,
      3,_,2,_,_,
      _,_,_,_,_,
      _,_,4,_,5,
      4,2,_,_,_],
     [a,a,a,a,b,
      a,c,c,b,b,
      c,c,d,b,b,
      e,c,d,d,d,
      e,e,e,e,d]).
```

Ce qui nous donne :

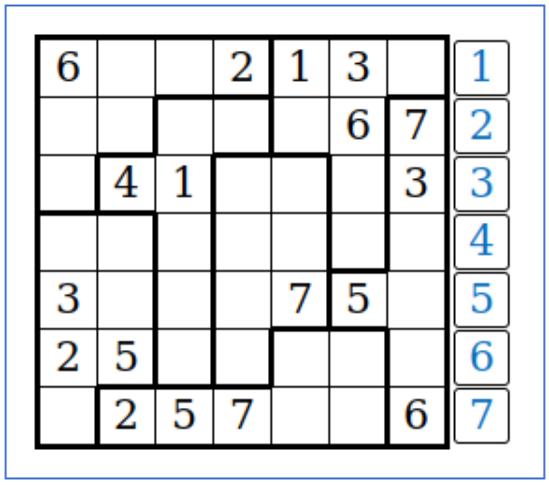
```
?- solve().
2 5 1 4 3
3 1 2 5 4
5 4 3 1 2
1 3 4 2 5
4 2 5 3 1
true .
```

(Ici le temps de calcul est de 3,10 secondes.)

Nous obtenons bien le même résultat que le premier programme.

4.3 .. avec un sudoku de taille 7x7

Voici un Jigsaw sudoku de taille 7x7 :



6			2	1	3		1
					6	7	2
	4	1				3	3
							4
3				7	5		5
2	5						6
	2	5	7			6	7

Pour cette exemple, nous allons utiliser notre programme NxN.

```
solve() :- sudoku(7,
              [6,_,_,2,1,3,_,
              _,_,_,_,_,6,7,
              _,4,1,_,_,_,3,
              _,_,_,_,_,_,_,
              3,_,_,_,7,5,_,
              2,5,_,_,_,_,_,
              _,2,5,7,_,_,6],
          [a,a,a,a,b,b,b,
          a,a,c,c,b,b,d,
          a,c,c,e,e,b,d,
          f,f,c,e,e,b,d,
          f,f,c,e,e,d,d,
          f,f,c,e,g,g,d,
          f,g,g,g,g,g,d]).
```

Le programme est toujours en train de tourner ;)

5 Annexe : les codes

5.1 Première version en 5x5

```
/* Valeur  placer */
nombre(X) :-member(X,[1,2,3,4,5]).

/* Fonction qui vrifie que les valeurs sont diffrentes */
diff([]).
diff([L|Ls]) :-not(member(L,Ls)), diff(Ls).

/* Rempli une liste de 5 valeurs diffrentes */
remplir([A,B,C,D,E]) :-nombre(A), nombre(B), nombre(C), nombre(D),
                       nombre(E), diff([A,B,C,D,E]).

/* Affiche une ligne */
print_ligne([]) :-write("\n").
print_ligne([A|Ab]) :-write(A), write(" "), print_ligne(Ab).

/* Fonction qui solve le sudoku */
sudoku([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5,D1,D2,D3,D4,D5,E1,E2,E3,E4,E5]) :-
    /* Ligne */
    remplir([A1,A2,A3,A4,A5]),
    remplir([B1,B2,B3,B4,B5]),
    remplir([C1,C2,C3,C4,C5]),
    remplir([D1,D2,D3,D4,D5]),
    remplir([E1,E2,E3,E4,E5]),

    /* Colonne */
    diff([A1,B1,C1,D1,E1]),
    diff([A2,B2,C2,D2,E2]),
    diff([A3,B3,C3,D3,E3]),
    diff([A4,B4,C4,D4,E4]),
    diff([A5,B5,C5,D5,E5]),

    /* Case : A CHANGER SELON LES PUZZLES */
    diff([A1,A2,A3,A4,A5]),
    diff([B1,C1,C2,D1,D2]),
    diff([B2,B3,B4,B5,C5]),
    diff([E1,E2,E3,D3,C3]),
    diff([C4,D4,D5,E4,E5]),

    /* Fonction print */
    print_ligne([A1,A2,A3,A4,A5]),
    print_ligne([B1,B2,B3,B4,B5]),
    print_ligne([C1,C2,C3,C4,C5]),
    print_ligne([D1,D2,D3,D4,D5]),
    print_ligne([E1,E2,E3,E4,E5]).
```

5.2 Seconde version 5x5

```
/* Valeur  placer */
nombre(X) :-member(X,[1,2,3,4,5]).

/* Fonction qui vrifie que les valeurs sont diffrentes */
diff([]).
diff([A|Ab]) :-not(member(A,Ab)),
               diff(Ab).
diffs([]).
diffs([A|Ab]) :-diff(A),
                diffs(Ab).

/* Rempli une liste de 5 valeurs diffrentes */
remplir([A,B,C,D,E]) :-nombre(A),
                       nombre(B),
                       nombre(C),
                       nombre(D),
                       nombre(E),
                       diff([A,B,C,D,E]).

/* Affiche une ligne */
print_ligne(A,B,C,D,E) :-write(A),
                         write(" "),
                         write(B),
                         write(" "),
                         write(C),
                         write(" "),
                         write(D),
                         write(" "),
                         write(E),
                         write("\n").

/* Fonction recupere les infos cases */
infocase([A],[B],[[A,B]]).
infocase([A|As],[B|Bs],[[A,B]|Y]) :-infocase(As,Bs,Y).

/* Fonction qui recupere la liste des X qui ont le mme Y */
meme2(_,[],[]).
meme2(X,[[A,X]|Ls],[A|Y]) :-meme2(X,Ls,Y).
meme2(X,[[_,B]|Ls],Y) :-X \= B,
                        meme2(X,Ls,Y).

/* Fonction qui rcupere une liste de sous liste ayant le meme Y */
listbloc(_,[],[]).
listbloc([],L,[]) :-L \= [].
listbloc(L,[A|Ab],Y) :-meme2(A,L,Y1),
                       delete(Ab,A,Ac),
                       listbloc(L,Ac,Y2),
                       append([Y1],Y2,Y).
```

```
/* Fonction qui solve le sudoku */
sudoku([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5,D1,D2,D3,D4,D5,
       E1,E2,E3,E4,E5],Bloc) :-

   infocase([A1,A2,A3,A4,A5,B1,B2,B3,B4,B5,C1,C2,C3,C4,C5,
   D1,D2,D3,D4,D5,E1,E2,E3,E4,E5],Bloc,Case),
   listbloc(Case,Bloc,Bloc1),
   /* Ligne */
   remplir([A1,A2,A3,A4,A5]),
   remplir([B1,B2,B3,B4,B5]),
   remplir([C1,C2,C3,C4,C5]),
   remplir([D1,D2,D3,D4,D5]),
   remplir([E1,E2,E3,E4,E5]),

   /* Colonne */
   diff([A1,B1,C1,D1,E1]),
   diff([A2,B2,C2,D2,E2]),
   diff([A3,B3,C3,D3,E3]),
   diff([A4,B4,C4,D4,E4]),
   diff([A5,B5,C5,D5,E5]),

   /* Block */
   diffs(Bloc1),

   /* Fonction print */
   print_ligne(A1,A2,A3,A4,A5),
   print_ligne(B1,B2,B3,B4,B5),
   print_ligne(C1,C2,C3,C4,C5),
   print_ligne(D1,D2,D3,D4,D5),
   print_ligne(E1,E2,E3,E4,E5).

 solver() :-sudoku(
                [_,_,_,4,3,
             3,_,2,_,_,
             _,_,_,_,_,
             _,_,4,_,5,
             4,2,_,_,_],
             [a,a,a,a,b,
             a,c,c,b,b,
             c,c,d,b,b,
             e,c,d,d,d,
             e,e,e,e,d]).
```

5.3 Programme N*N

```
/* Fonction valeur : Cration d'une liste comportant les X premiers entiers */
valeur(1,[1]).
valeur(X,[X|Y]) :-X > 0,
                  X1 is X-1,
                  valeur(X1,Y).

/* Fonction affichage */
aff([]) :-write("\n").
aff([L|Ls]) :-write(L),
                  write(" "),
                  aff(Ls).

aff2([]).
aff2([L|Ls]) :-aff(L),
                   aff2(Ls).

/* Fonction diff => Vrifie que X n'est pas dans la liste L */
diff([A,B]) :-A \= B.
diff([L|Ls]) :-not(member(L,Ls)),
                     diff(Ls).

/* Fonction different chaque element */
diffs([]).
diffs([L|Ls]) :-diff(L),
                    diffs(Ls).

/* Liste valide */
liste([A],L) :-member(A,L).
liste([A|Ab],L) :-member(A,L),
                      delete(L,A,L1),
                      liste(Ab,L1).

/* Remplir */
valo([],_).
valo([A|C],Valeur) :-liste(A,Valeur),
                         diff(A),
                         valo(C,Valeur).

/* Fonction carre : Renvoie le carre de N */
carre(N,Y) :-Y is N*N.

/* Fonction longueur : Renvoie la longueur de la liste */
longueur([],0).
longueur([_|Ls],Y) :-longueur(Ls,Y2),
                         Y is Y2 + 1.

/* Recupere les X premiers lments de la liste */
recup(0,_,[]).
recup(1,[L|_],[L]).
recup(_,[],[]).
```

```
recup(X,[L|Ls],[L|Y]) :-X > 1,
                    X1 is X-1,
                    recup(X1,Ls,Y).

/* Fonction qui retire les X premiers elements de la liste */
remove(0,L,L).
remove(_,[],[]).
remove(X,[_|Ls],Y) :-X > 0,
                     X1 is X-1,
                     remove(X1,Ls,Y).

/* Fonction qui divise la liste L en sous liste de X elements */
ligne(X,L,[L]) :-longueur(L,R),
                 X >= R.
ligne(X,L,Y) :-longueur(L,R),
               X < R,
               recup(X,L,L1),
               remove(X,L,L2),
               ligne(X,L2,Y1),
               append([L1],Y1,Y).

/* Recupere les lments tous les X */
col(X,[L|Ls],[L]) :-longueur([L|Ls],R),
                    X >= R.
col(X,[L|Ls],[L|Y]) :-longueur([L|Ls],R),
                  X < R,
                  X1 is X-1,
                  remove(X1,Ls,L1),
                  col(X,L1,Y).

/* Retire les elements tous les X */
retire(_,[],[]).
retire(X,[L|Ls],Ls) :-longueur([L|Ls],R1),
                      X >= R1.
retire(X,L,Y) :-longueur(L,R),
                X < R,
                remove(1,L,R1),
                X1 is X-1,
                recup(X1,R1,R2),
                remove(X1,R1,R3),
                retire(X,R3,Y2),
                append(R2,Y2,Y).

/* Recupere les colonnes tous les X */
colonne(1,L,[L]).
colonne(X,L,Y) :-X > 1,
                 col(X,L,R1),
                 retire(X,L,R2),
                 X1 is X-1,
                 colonne(X1,R2,Y1),
                 append([R1],Y1,Y).
```

```
/* Fonction recupere les infos cases */
infocase([A],[B],[[A,B]]).
infocase([A|As],[B|Bs],[[A,B]|Y]):- infocase(As,Bs,Y).

/* Fonction qui recupere la liste des X qui ont le mme Y */
meme2(_,[],[]).
meme2(X,[[A,X]|Ls],[A|Y]) :-meme2(X,Ls,Y).
meme2(X,[[_,B]|Ls],Y) :-X \= B, meme2(X,Ls,Y).

/* Fonction qui rcupere une liste de sous liste ayant le meme Y */
listblock(_,[],[]).
listblock([],L,[]) :-L \= [].
listblock(L,[A|Ab],Y) :-meme2(A,L,Y1),
                        delete(Ab,A,Ac),
                        listblock(L,Ac,Y2),
                        append([Y1],Y2,Y).

/* Fonction solver */
sudoku(Dim,Sudo,Block) :-carre(Dim,Dimen1),
                         longueur(Sudo,Dimen1),
                         longueur(Block,Dimen1),
                         valeur(Dim,Valeur),
                         infocase(Sudo,Block,L1),
                         listblock(L1,Block,Block2),
                         ligne(Dim,Sudo,Ligne),
                         colonne(Dim,Sudo,Colonne),
                         valo(Block2,Valeur),
                         diffs(Ligne),
                         diffs(Colonne),
                         aff2(Ligne).
```