Concours commun Centrale - Supelec 2014

Corrigé de l'épreuve d'informatique

MP - option informatique

Partie I: préliminaires

Il n'y a aucun problème pour écrire les trois premières fonctions:

```
let rec appartient x l = match l with
  | [] -> false
  | a::q -> a=x || appartient x q ;;

let rec supprime x l = match l with
  | [] -> []
  | a::q when a=x -> supprime x q
  | a::q -> a:: supprime x q;;

let rec ajoute x l = match l with
  | [] -> [x]
  | a::q when a=x -> l
  | a::q -> a:: ajoute x q;;
```

ou (à coût presque identique):

```
let rec ajoute x l = match appartient x l with
  | true -> l
  | false -> x :: l;;
```

En notant (i,j) l'indice de la case de bloc (b,r) et $b = 3b_1 + b_2$ et $r = 3r_1 + r_2$ les divisions euclidiennes de b et de r par 3, nous avons $i = 3b_1 + r_1$ et $j = 3b_2 + r_2$, ce qui donne:

```
let indice (b,r) = 3*(b/3)+(r/3)3, 3*(b mod 3)+(r mod 3);;
```

Partie II: codage de la formule initiale

II.A -Formule logique décrivant la règle du jeu

A.1) La clause s'écrit $\bigvee_{k=1}^{9} x_{i,j}^k$. La règle (K1) s'écrit mathématiquement: $\forall (i,j) \in [\![0,8]\!]^2,\ \exists k \in [\![1,9]\!],\ x_{i,j}^k$, ce qui correspond à la forme normale conjonctive $K1 = \bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} \left(\bigvee_{k=1}^{9} x_{i,j}^k \right)$ qui contient 81 clauses.

On construit facilement $K1$ à l'aide d'une triple boucle: chaque boucle sur k construit une clause qui est ajoutée à la formule F (on utilise des référence sur des listes pour faire évoluer dynamiquement la clause c et la formule f):

```
let case1 () =
  let f = ref [] in
  for i = 0 to 8 do for j = 0 to 8 do
    let c = ref [] in
    for k = 1 to 9 do
      c := X(i,j,k) :: !c
    done;
     f := !c :: !f
  done; done;
  !f;;
```

A.2) (L1) s'écrit: $\forall (i,k) \in [\![0,8]\!] \times [\![1,9]\!],\ \exists j \in [\![0,8]\!],\ x_{i,j}^k$, ce qui donne $L1 = \bigwedge_{i=0}^{8} \bigwedge_{k=1}^{9} \left(\bigvee_{j=0}^{8} x_{i,j}^k \right)$.

A.3) On obtient de la même façon $C1 = \bigwedge_{j=0}^{8} \bigwedge_{k=1}^{9} \left(\bigvee_{i=0}^{8} x_{i,j}^k \right)$ et $B1 = \bigwedge_{b=0}^{8} \bigwedge_{k=1}^{9} \left(\bigvee_{r=0}^{8} x_{\texttt{indice}(b,r)}^k \right)$.

On obtient ainsi la fonction `bloc1`, presque identique à la fonction `ligne1`:

```
let bloc1 () =
  let f = ref [] in
  for b = 0 to 8 do for k = 1 to 9 do
    let c = ref [] in
    for r = 0 to 8 do
      let i,j = indice (b,r) in
      c := X(i,j,k) :: !c
    done;
     f := !c :: !f
  done; done;
  !f;;
```

A.4) La ligne i ne contient pas deux fois la valeur k si et seulement:

$$\forall j_1, j_2 \in [\![0,8]\!],\ j_1 < j_2 \Longrightarrow \left(\neg\, x_{i,j_1}^k \vee \neg\, x_{i,j_2}^k\right)$$

ce qui se traduit par la forme normale conjonctive $\bigwedge_{j_1=0}^{7} \bigwedge_{j_2=j_1+1}^{8} \left(\neg\, x_{i,j_1}^k \vee \neg\, x_{i,j_2}^k\right)$.

(L2) s'écrit donc $\forall i \in Ii,\ \forall k \in [\![1,9]\!],\ \forall j_1, j_2 \in [\![0,8]\!],\ j_1 \neq j_2 \Longrightarrow \left(\neg\, x_{i,j_1}^k \vee \neg\, x_{i,j_2}^k\right)$ et la formule:

$$L2 = \bigwedge_{i=0}^{8} \bigwedge_{k=1}^{9} \bigwedge_{j_1=0}^{8} \bigwedge_{\substack{j_2=0 \\ j_2 \neq j_1}}^{8} \left(\neg\, x_{i,j_1}^k \vee \neg\, x_{i,j_2}^k\right).$$

Cette formule contient $9 \times 9 \times \dfrac{9 \times 8}{2}$, soit 2916 clauses et est calculée par la fonction:

```
let lig2 () =
  let f = ref [] in
  for i = 0 to 8 do for k= 1 to 9 do for j1= 0 to 7 do for j2= j1 to 8 do
    f := [NonX(i,j1,k); NonX(i,j2,k)] :: !f
  done; done; done; done;
  !f;;
```

A.5) De la même manière, on obtient:

$$\begin{cases} C2 = \bigwedge_{j=0}^{8} \bigwedge_{k=1}^{9} \bigwedge_{i_1=0}^{7} \bigwedge_{i_2=i_1+1}^{8} \left(\neg\, x^k_{i_1,j} \vee \neg\, x^k_{i_2,j}\right) \\ K2 = \bigwedge_{i=0}^{8} \bigwedge_{j=0}^{8} \bigwedge_{k_1=1}^{8} \bigwedge_{k_2=k_1+1}^{9} \left(\neg\, x^{k_1}_{i,j} \vee \neg\, x^{k_2}_{i,j}\right) \\ B2 = \bigwedge_{b=0}^{8} \bigwedge_{k=1}^{9} \bigwedge_{r_1=0}^{7} \bigwedge_{r_2=r_1+1}^{8} \left(\neg\, x^k_{\texttt{indice}(b,r_1)} \vee \neg\, x^k_{\texttt{indice}(b,r_1)}\right) \end{cases}$$

Les formules $K1$, $L1$, $C1$, $B1$ (resp. $K2$, $L2$, $C2$, $B2$) contiennent chacunes 81 (resp. 2916) clauses, donc $F_{\texttt{règle}}$ en contient bien $4 \times (81 + 2916) = 11988$.

II.B -Formule logique décrivant la grille initiale

B.1) On initialise une pile vide de clause puis on parcourt le tableau. Si l'on trouve une valeur $k \neq 0$ dans la case (i, j), on ajoute à la pile les clauses $x^k_{i,j}$ et $\neg x^q_{i,j}$ pour tout $q \neq k$ (attention: une clause unitaire est une liste de longueur 1):

```
let donnees t =
  let f = ref [] in
  for i = 0 to 8 do for j = 0 to 8 do
  match t.(i).(j) with
    | 0 -> ()
    | k ->
     for q = 1 to 9 do
       if q = k then f := [X(i,j,q)] :: !f else f := [NonX(i,j,q)] :: !f
    done;
  done; done;
  !f;;
```

B.2) La fonction `indice` est égale à son inverse, donc $b = 3(i/3) + j/3$ où / désigne la division entière (comme dans Caml).

Pour éviter d'écrire plusieurs fois la même clause unitaire négative, nous utilisons la fonction `ajoute` de la partie I.

```
let interdites_ij t i j = let f = ref [] in
  let b = 3*(i/3)+(j/3) in
  for l = 0 to 8 do
    if t.(l).(j) <> 0 then
      f := ajoute NonX(i,j,t.(l).(j)) !f;
    if t.(i).(l) <> 0 then
    f := ajoute NonX(i,j,t.(i).(l)) !f;
    let i1,j1 = indice(b,l) in
    if t.(i1).(j1) <> 0 then
      f := ajoute NonX(i,j,t.(i1).(j1)) !f;
  done;
  !f ;;
```

En remarquant que l'utilisation de la concaténation de liste n'est pas trop couteuse, puisque les listes `interdites_ij T i j` est de longueur au plus 8, cela donne la fonction:

```
let interdites t =
  let f = ref [] in
  for i = 0 to 8 do for j = 0 to 8 do
    if t.(i).(j)=0 then
      f := (interdites_ij t i j)  !f;
  done; done;
  !f;;
```

B.3) Chaque case remplie (i, j) donne 9 clauses et chaque case non remplie donne au maximum 8 clauses (comme le problème a une solution, il y a au moins une valeur non interdite pour chaque case). La formule F contient donc au maximum $9N + 8(81 - N)$ clauses, où N est le nombre de cases pleines de la grille initiale. Dans le pire des cas (toutes les cases sont remplies), F contient $9^3 = 729$ clauses.

Partie III: résolution d'une grille de sudoku

III.A - Règle de propagation unitaire

A.1) Comme la grille initiale est associée à une et une seule grille finale, il existe une unique valuation satisfaisant $F_{\texttt{initiale}}$.

A.2) Il y a $9^3 = 729$ variables, donc la table de vérité de F contient 2^{729} lignes (soit plus de10^{219}); il est bien illusoire de tester toutes les lignes à la recherche de l'unique valuation solution du problème.

A.3) Si σ satisfait F', prolongeons σ en une valuation $\overline{\sigma}$ définie sur $\mathcal{V}$ en posant $\overline{\sigma}(p) = \texttt{true}$ si $\ell = p$ et $\overline{\sigma}(p) = \texttt{false}$ si $\ell = \neg p$. Nous avons alors

$$\mathrm{Ev}_{\overline{\sigma}}(F) = \mathrm{Ev}_{\overline{\sigma}}(\ell) \wedge \mathrm{Ev}_{\overline{\sigma}}(F_1) \wedge \mathrm{Ev}_{\overline{\sigma}}(F_2) \wedge \mathrm{Ev}_{\overline{\sigma}}(F_3).$$

- si C est une clause contenant ℓ, elle est satisfaite par $\overline{\sigma}$: on en déduit que $\mathrm{Ev}_{\overline{\sigma}}(\ell) = \mathrm{Ev}_{\overline{\sigma}}(F_1) = \texttt{true}$;
- si C est une clause contenant $\neg\ell$ et ne contenant pas ℓ, $\overline{\sigma}(C) = \overline{\sigma}(C')$ où C' est la clause obtenue en supprimant de C les occurences du littéral $\neg\ell$: on en en déduit que $\mathrm{Ev}_{\overline{\sigma}}(F_2) = \mathrm{Ev}_{\sigma}(F_2')$;
- si C est une clause ne contenant ni ℓ, ni $\neg\ell$, la variable p n'intervient pas dans C et $\mathrm{Ev}_{\overline{\sigma}}(C) = \mathrm{Ev}_{\sigma}(C)$: on en déduit que $\mathrm{Ev}_{\overline{\sigma}}(F_3) = \mathrm{Ev}_{\sigma}(F_3)$.

Nous obtenons donc:

$$\mathrm{Ev}_{\overline{\sigma}}(F) = \mathrm{Ev}_{\sigma}(F_2') \wedge \mathrm{Ev}_{\sigma}(F_3) = \mathrm{Ev}_{\sigma}(F') = \texttt{true}.$$

Supposons réciproquement qu'il existe une valuation $\overline{\sigma}$ qui prolonge σ et qui satisfait F. Comme ℓ est une clause de F, σ satisfait ℓ et on obtient comme ci-dessus $\mathrm{Ev}_{\sigma}(F') = \mathrm{Ev}_{\overline{\sigma}}(F) = \texttt{true}$.

A.4 Une première étape permet de remplacer F par $\left(x^4_{(2,2)} \vee x^6_{(3,6)} \vee x^7_{(7,7)}\right) \wedge \left(\neg x^6_{(3,6)}\right)$, qui devient ensuite $\left(x^4_{(2,2)} \vee x^7_{(7,7)}\right)$. Ces transformations doivent être accompagnées de l'affectation $T.(0).(0) \leftarrow 1$.

A.5 Seules les valeurs 1, 3, 4 et 7 ne sont pas déjà présentes sur la ligne, la colonne ou le bloc de la case $(0,7)$. Les cases vides de la ligne 0 sont les cases $(0,0), (0,2), (0,4), (0,5), (0,7)$. Comme le chiffre 7 apparaît déjà dans les blocs 0 et 1, (dans les cases $(2,1)$ et $(1,5)$), il ne peut être placé ni en $(0,0)$, $(0,2)$, $(0,4)$ ou $(0,5)$: on en déduit que le 7 de la première ligne est nécessairement dans la case $(0,7)$.

En terme de formule, nous pouvons écrire:

$$F_{\texttt{initiale}} = F' \wedge F_1' \wedge F_2' \wedge F_3$$

$$\text{où } \begin{cases} F_1' = \neg x^7_{(0,1)} \wedge \neg x^7_{(0,3)} \wedge \neg x^7_{(0,6)} \wedge \neg\, x^7_{(0,8)} \\ F_2' = \neg x^7_{(0,0)} \wedge \neg x^7_{(0,2)} \wedge \neg x^7_{(0,4)} \wedge \neg\, x^7_{(0,5)} \wedge \neg\, x^7_{(0,5)} \\ F_3' = x^7_{(0,0)} \vee x^7_{(0,1)} \vee x^7_{(0,2)} \vee x^7_{(0,3)} \vee x^7_{(0,4)} \vee x^7_{(0,5)} \vee x^7_{(0,6)} \vee x^7_{(0,7)} \vee x^7_{(0,8)} \vee \end{cases}.$$

En effet, F_1' est une partie de F_1, F_2' une partie de F_2 et F_3' une partie de $L1$. En appliquant l'algorithme de propagation unitaire successivement avec les littéraux isolés $\neg x^7_{(0,0)}$, $\neg x^7_{(0,1)}$, $\neg x^7_{(0,2)}$, $\neg x^7_{(0,3)}$, $\neg x^7_{(0,4)}$, $\neg x^7_{(0,5)}$, $\neg x^7_{(0,6)}$, $\neg x^7_{(0,8)}$, nous obtenons $F_{\texttt{initiale}} \equiv F'' \wedge x^7_{(0,7)}$ qui donne enfin $x^7_{(0,7)} = \texttt{true}$ en choisissant le littéral isolé $x^7_{(0,7)}$.

A.6) Il suffit de parcourir récursivement la formule à la recherche d'une clause de longueur 1 (sans faire appel à la fonction `list_length`!), ce qui donne:

```
let rec nouveau_lit_isole f = match f with
  | [] -> X(-1,-1,-1)
  | [lit] :: f1 -> lit
  | _ :: f1 -> nouveau_lit_isole f1;;
```

A.7) On peut traduire directement la transformation proposée:

```
let rec simplification l f = match f with   | [] -> []
  | C :: f1 when appartient l C -> simplification l f1
  | C :: f1 -> supprime (neg l) C :: simplification l f1;;
```

A.8) Il suffit d'appliquer la méthode proposée:

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

A.9) À la fin du calcul, si la méthode fonctionne, le tableau T contient la solution au problème posé et la formule renvoyée F' est équivalente à $F_{\texttt{initiale}}$ et ne contient aucun littéral isolé. Supposons que $F \neq [\,]$: il existe alors une variable $x^k_{i,j}$ qui est encore présente dans F', ce qui impose que les littéraux isolés `X(i,j,k)` et `NonX(i,j,k)` ne sont jamais apparus dans la formule (sinon, une simplification aurait supprimé tous les $x^k_{(i,j)}$). Quatre cas sont alors possibles:

1. la case (i,j) était remplie au début du calcul et $T.(i).(j) = k$; comme le littéral isolé $x^k_{(i,j)}$ était présent dans la formule initiale, il n'a pu être supprimé que lors d'une simplification faite avec le littéral isolé `X(i,j,k)` et $x^k_{(i,j)}$ n'apparaît plus dans F': c'est absurde;

2. la case (i,j) était remplie au début du calcul et $T.(i).(j) = k' \neq k$; comme la clause $C = \neg x^k_{(i,j)} \vee \neg x^{k'}_{(i,j)}$ était dans la formule initiale et que $x^{k'}_{(i,j)}$ n'apparaît pas dans F' (cas 1.), F' contient le littéral isolé $\neg x^k_{(i,j)}$: c'est absurde;

3. la case (i,j) n'était pas remplie au début du calcul et $T.(i).(j) = k$; il y a donc un moment où, dans la propagation, on a effectué l'affectation `T.(i).(j) <- k`, qui a été suivie de la simplification par $x^k_{(i,j)}$: c'est absurde, puisque cette simplification supprime tous les $x^k_{(i,j)}$;

4. la case (i,j) n'était pas remplie au début du calcul et $T.(i).(j) = k' \neq k$; comme précédemment, F' ne contient pas la variable $x^{k'}_{(i,j)}$ et la clause $C = \neg x^k_{(i,j)} \vee \neg x^{k'}_{(i,j)}$ présente dans F est devenue $\neg x^k_{(i,j)}$ au moment de la simplification par le littéral isolé $X(i,j,k')$: c'est une nouvelle fois absurde.

Si la méthode ne fonctionne pas, le tableau T n'est pas entièrement rempli et la formule renvoyée n'est pas vide et ne contient aucun littéral isolé.

A.10) La fonction `nouveau_lit_isole` parcourt, dans le pire des cas, toute la liste F en effectuant une opération de coût constant sur chaque clause: son coût est donc dans le pire des cas de l'ordre de la longueur de F, ce qui donne (grossièrement) un $O(n)$.

La fonction `simplification` demande, dans le pire des cas, le parcours de toutes les clauses au plus deux fois (une seule fois si le littéral est présent dans la clause et une seconde fois s'il n'est pas présent, lors du calcul `supprime (neg l) C1`): le temps de calcul est une nouvelle fois un $O(n)$.

Enfin, dans le pire des cas (pour le temps de calcul, puisque ce pire des cas est celui où l'algorithme permet de résoudre le sudoku), chaque variable est supprimée de la formule initiale et la simplification est effectuée autant de fois qu'il y a de variables. Comme on demande un résultat ne dépendant que de n, on fait l'hypothèse que toutes les variables apparaissent dans la formule (ce qui est le cas avec $F_{\texttt{initiale}}$): le nombre de variable est donc inférieur à n et le coût total de `propagation` est un $O(n^2)$.

On pourrait faire un calcul un peu plus précis en exprimant le temps de calcul (toujours grossièrement) comme un $O(g(N))$ où N est la taille du sudoku (ici, $N = 9$ et n est de l'ordre de N^4).

III.B - Règle du littéral infructueux

B.1) Si l'on peut déduire la clause vide de la formule $F \wedge \neg x$, $F \wedge \neg x$ est une antilogie, ce qui donne:

$$F \equiv F \wedge (x \vee \neg x) \equiv (F \wedge x) \vee (F \wedge \neg x) \equiv (F \wedge x) \vee \bot \equiv F \wedge x$$

B.2) Je ne vois pas l'intérêt d'utiliser la fonction `flatten` qui va nécessiter de concaténer les clauses. Nous obtenons la fonction `variables` qui utilise les fonctions auxiliaires `ajouter_clause` et `ajouter_formule` qui ajoutent (sans doublons) les variables rencontrées à la pile `Pile`:

```
let variables f = let pile = ref [] in
  let rec ajouter_clause c = match c with
    | [] -> ()
    | X(i,j,k)::c1 -> pile := ajoute (X(i,j,k)) !pile; ajouter_clause c1
    | NonX(i,j,k)::c1 -> pile := ajoute (X(i,j,k)) !pile; ajouter_clause c1 in
  let rec ajouter_formule f = match f with
    | [] -> ()
    | c :: f1 -> ajouter_clause c; ajouter_formule f1 in
  ajouter_formule f;
  !pile;;
```

B.3 Nous allons appliquer la fonction `propagation` à des formules qui ne seront plus satisfiables (on ajoute un littéral quelconque que l'on espère être infructueux): en appliquant la fonction `simplification`, une instruction `supprime (neg l) C` peut donc conduire à la liste vide $\bot$: il est alors maladroit de laisser cette liste vide dans la formule finale, puisqu'une formule contenant la clause $\bot$ n'est pas satisfiable (c'est une antilogie). Autrement-dit, dès que l'on obtient une clause vide, il serait plus intéressant de faire renvoyer par la fonction `simplification` la formule $L = [[]]$ qui représente une antilogie (et non pas la formule $F = []$ qui représente une tautologie), plutôt qu'une longue formule contenant une clause vide (cela évitera de chercher à chaque étape si la formule contient une clause vide, alors que la présence d'une telle clause peut être détectée à l'étape précédente). Nous allons donc modifier la fonction `simplification` de sorte qu'elle renvoie $[[]]$ dès qu'une clause vide a été trouvée:

```
let rec simplification l F = match F with
  | [] -> []
  | C1 :: F1 when appartient l C1 -> simplification l F1
  | C1 :: F1 -> match supprime (neg l) C1 with
    | [] -> [[]]
    | C2 -> match simplification l F1 with
      | [[]] -> [[]]
      | F2 -> C2 :: F2;;
```

La fonction `deduction` s'écrit alors facilement: on applique la fonction `propagation` à la formule $[\neg x] :: F$; si l'on obtient l'antilogie $[[]]$, on renvoie 1; sinon, on applique la fonction `propagation` à la formule $[x] :: F$; si l'on obtient l'antilogie $[[]]$, on renvoie -1; sinon, on renvoie 0. On utilise des copies de la matrice T car la fonction `propagation` modifie la matrice à laquelle elle est appliquée.

```
let deduction t x f = let t1 = copier_matrice t in match x with
  | X(i,j,k) when propagation t1 ([NonX(i,j,k)] :: f) = [[]] -> 1
  | _ -> let t2 = copier_matrice t in
    if propagation t2 ([x] :: f) = [[]] then -1 else 0;;
```

B.4 Nous allons utiliser l'analyse suivante: un booléen `fini` initialisé à la valeur `false` permet de savoir si le calcul est terminé. Il prend la valeur `true` si après une propagation, la formule obtenue est [] (le sudoku est alors résolu), ou bien si on ne trouve pas de littéral infructueux. Cela donne:

```
let deduction2 t f = let fini = ref false and f1 = ref f in
 while not !fini do
  f1 := propagation t !f1;
  if !f1=[] then
   fini:= true
  else
   begin
   match litteral_infructueux t (variables !f1) !f1 with
    | X(-1,-1,-1) -> fini := true
    | x -> f1 := [x] :: !f1
   end
 done;;
```

B.4 Il suffit de construire la formule $F_{\texttt{initiale}}$ en concaténant les résultats des huit fonctions construites dans la partie II (pour accélérer le temps de calcul de `nouveau_lit_isole`, nous mettons en tête de liste les littéraux isolés, puis les clauses de longueur 2). Cela donne:

```
let sudoku t =
 let f = (donnees t) @ (interdites t) @ case2() @ lig2() @ col2() @ bloc2() @
        case1() @ lig1() @ col1() @ bloc1() in
 deduction2 t f;;
```

Il est enfin possible de tester la fonction sur l'exemple donné en début d'énoncé, qui donne la solution du problème en 5 secondes:

```
let t1 = [|[|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|]|];;

sudoku t1;;
```

qui donne la table complétée:

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

On pourra tester cette fonction sur d'autres sudokus, en particulier sur les deux exemples ci-dessous, qualifiés respectivement de "très difficile" et de "diabolique" dans le recueil où je les ai trouvés:

```
let t2 = [|[|8;0;0;0;7;9;4;0;0|]; [|0;7;0;0;0;0;0;0;0|]; [|0;0;6;3;0;0;0;9;0|];
           [|0;0;0;0;0;0;7;0;2|]; [|0;0;0;5;0;0;1;0;0|]; [|2;6;0;0;0;8;0;0;3|];
           [|0;5;7;0;0;0;0;0;0|]; [|0;0;8;0;0;0;5;0;4|]; [|0;0;4;0;1;0;2;0;0|]|];;

let t3 = [|[|0;9;0;0;6;0;3;0;8|]; [|0;3;0;0;0;2;0;0;0|]; [|4;0;0;8;0;0;5;0;0|];
           [|0;0;0;3;0;0;0;6;0|]; [|5;0;0;1;0;6;0;0;9|]; [|0;6;0;0;0;5;0;0;0|];
           [|0;0;2;0;0;8;0;0;1|]; [|0;0;0;4;0;0;0;7;0|]; [|9;0;1;0;5;0;0;2;0|]|];;

sudoku t2; sudoku t3;
```