Résolution du jeu de Sudoku par backtracking

MPI, Lycée Leconte de Lisle

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

Le jeu de *Sudoku* consiste à remplir une grille 9×9 avec des chiffres entre 1 et 9 inclus. Il faut respecter les contraintes suivantes.

a. Un chiffre ne peut pas appparaître deux fois sur une même ligne.

b. Un chiffre ne peut pas appparaître deux fois sur une même colonne.

c. Dans chacune des 9 sous-grilles 3×3 un chiffre ne peut apparaître deux fois.

Nous nous proposons de résoudre algorithmiquement ce jeu à l'aide de la méthode du *backtracking*[1].

1 Codage bit à bit des chiffres possibles

Dans chaque case, pour une grille donnée, les règles du jeu nous autorisent à placer certains chiffres parmi ceux de 1 à 9. On souhaite coder cet ensemble de possibilités. Pour cela l'idée naturelle est d'utiliser une liste d'entiers mais pour des raisons d'efficacité nous allons plutôt utiliser un *code* qui sera un entier entre 0 et 1022 représentant les chiffres possibles.

Le code c d'une case est défini par

[1] *retour sur trace* en français, mais personne ne dit ça...

$$c = \sum_{k=1}^{9} \varphi(k) \times 2^k$$

avec $\varphi(k) = 1$ si le placement du chiffre k est autorisé et $\varphi(k) = 0$ sinon. Ainsi chaque *bit* de l'entier c code la possibilité de placer un chiffre. On remarque que le premier bit de l'entier (bit de poids faible) n'est pas utilisé dans notre représentation. Tous nos codes seront donc des entiers pairs.

Pour manipuler facilement les codes, nous utiliserons les **opérateurs bits à bits** travaillant sur les entiers.

Opérateur	**OCAML**	**Résultat**
Et	`x land y`	Construit un entier en calculant le *ET* entre chaque bit de x et y.
Ou	`x lor y`	Construit un entier en calculant le *OU* entre chaque bit de x et y.
Non	`lnot x`	Construit un entier en inversant $(0 \leftrightarrow 1)$ chaque bit de x.
Décalage gauche	`x lsl k`	Construit un entier en décalant de k bits vers la gauche la réprésentation en base 2 de x.
Décalage droite	`x lsr k`	Construit un entier en décalant de k bits vers la droite la réprésentation en base 2 de x. Les k bits les plus à droite de x sont perdus.

Le tableau suivant décrit un exemple de calcul de Ou bit à bit.

	Représentation en base 2							
`164`	1	0	1	0	0	1	0	0
`202`	1	1	0	0	1	0	1	0
`164 lor 202 = 238`	1	1	1	0	1	1	1	0

L'avantage de ces opérations est qu'elles correspondent à des intructions de base du processeur de l'ordinateur, autrement dit on pourra toujours les considérer de complexité $O(1)$.

On pourra remarquer que `x lsl k` correspond à calculer $x \times 2^k$ et `x lsr k` correspond à calculer la division entière $x//2^k$. Ces opérateurs sont donc très utiles en OCAML où il n'y a pas d'opérateur puissance sur les entiers.

Dans toutes les questions de cette partie, on s'efforcera d'utiliser le plus souvent les opérateurs logiques.

À titre d'illustration, je propose un corrigé de la première question (à ne pas regarder si vous avez bien compris).

Question 1

Écrire une fonction `ajoute : int -> int -> int` telle que `ajoute k c` retourne un code c' obtenu à partir de c en autorisant le chiffre k. Si le chiffre k était déjà présent, c' vaut c. La complexité sera $O(1)$.

```
let ajoute k code =
    code lor (1 lsl k)
;;
```

Question 2

Écrire une fonction `enleve : int -> int -> int` telle que `enleve k c` retourne un code c' obtenu à partir de c en interdisant le chiffre k. Si le chiffre k était déjà absent, c' vaut c. La complexité sera $O(1)$.

Question 3

Ecrire une fonction `encode : int list -> int` calculant le code c correspondant à une liste de chiffres autorisés.

Question 4

Réciproquement écrire une fonction `decode : int -> int list` retournant une liste des chiffres autorisés par un code.

On appellera *taille* d'un code, le nombre de chiffres autorisés par ce code, c'est-à-dire le nombre de bits 1 dans la représentation en base 2 du code.

Question 5

Écrire la fonction `taille : int -> int` calculant la taille d'un code. Pour s'entraîner on pourra coder cette fonction sans utiliser `List.length`.

On pourra vérifier que `encode [1; 2; 3; 4; 5; 6; 7; 8; 9]` donne bien 1022 et que `enleve 4 1022` retourne 1006.

2 Programmation des règles du jeu

Nous représenterons une grille de Sudoku par une matrice 9×9 d'entiers.

```
type grille = int array array;;
```

Les cases non remplies contiendront la valeur -1. Voici un exemple de définition d'une grille de Sudoku correspondant à la grille diabolique donnée en exemple.

```
let diabolique = [|
    [| 5; -1; -1; -1; 8; -1; -1; -1; -1 |];
    [| 1; -1; 9; -1; -1; -1; -1; -1; -1 |];
    [| 2; -1; -1; -1; -1; 4; -1; 8; -1 |];
    [| 9; -1; 4; -1; -1; 6; 8; -1; -1; |];
    [| -1; -1; -1; 9; -1; -1; -1; -1; 4 |];
    [| -1; 2; -1; -1; -1; 7; -1; -1; -1 |];
    [| 8; -1; -1; 7; -1; -1; 9; -1; -1 |];
    [| -1; 6; -1; -1; -1; -1; -1; -1; -1 |];
    [| -1; -1; 7; 1; 5; -1; 6; -1; 2 |]
  |];;
```

Afin d'obtenir un affichage plus joli de la grille on pourra utiliser la fonction suivante.

```
let print_sudoku s =
  let print_ligne () =
    for k = 1 to 37 do
      print_char '-'
    done;
    print_newline ();
  in
  print_ligne ();
  for i = 0 to 8 do
    print_string "| ";
    for j = 0 to 8 do
      if s.(i).(j) = -1 then
        print_char '?'
      else
        print_int s.(i).(j)
      ;
      print_string " | "
    done;
    print_newline ();
    print_ligne ()
  done
;;
```

Question 6

Écrire une fonction `nb_cases_vides : grille -> int` comptant le nombre de cases vides, *i.e.* de valeur -1, d'une grille.

En complément de la grille de jeu, on utilisera également une **grille des possibilités** qui sera également une matrice 9×9 d'entiers mais contenant cette fois-ci les codes de possibilité pour chacune des cases.

Dans cette seconde grille, une case contenant le code 0 signifiera que le problème est insoluble et les cases correspondant à 1 chiffre déjà trouvé (ou de l'énoncé) auront des codes de taille 1. Attention la réciproque n'est pas vraie, une case contenant un code taille 1 n'est pas forcément une case résolue mais une case pour laquelle on sait qu'on n'aura qu'une seule possibilité pour choisir le chiffre.

Question 7

Écrire une fonction `ajoute_grille : grille -> (int * int) -> int`

telle que `ajoute_grille possibilites (i, j) k` ajoute le chiffre k à la grille en position (i, j), c'est-à-dire modifie la grille des possibilités passée en argument ainsi :

1. Les cases de la ligne i, exceptée la case (i, j) ne sont plus autorisées à contenir la valeur k.
2. Les cases de la colonne j, exceptée la case (i, j) ne sont plus autorisées à contenir la valeur k.
3. Les cases de la sous-grille 3×3 concernée, exceptée la case (i, j) ne sont plus autorisées à contenir la valeur k.
4. La case (i, j) a une seule possibilité celle correspondant au chiffre k.

Pour le point 3, on pourra remarquer que `(i / 3 * 3, j / 3 * 3)` correspond à un coin de la sous-grille 3×3 contenant (i, j).

Question 8

En déduire une fonction `make_grille_possibilites : grille -> grille` qui a partir d'une grille énoncé de Sudoku construit la grille des codes de possibilités correspondante. On procèdera en partant d'une grille remplie de code 1022 (toutes les possibilités) et en y ajoutant progressivement les chiffres déjà placés dans la grille énoncé.

Nous savons maintenant à partir d'une grille énoncé, construire la grille des possibilités et modifier cette dernière grille lorsqu'on voudra ajouter un chiffre dans le sudoku.

3 Algorithme de backtracking

Notre algorithme sera très efficace et permettra de résoudre n'importe quel problème de Sudoku en quelques secondes.

Le *backtracking* consiste à construire la solution partiellement (pour nous case par case) et dans le cas où la solution partielle brise une contrainte revenir sur ses pas pour modifier les choix qui ont été précédemment faits.

Nous allons réaliser un *backtracking* intelligent c'est-à-dire utilisant une stratégie pour choisir l'ordre des cases la construction de la solution. On décidera de remplir d'abord la case la plus contrainte c'est-à-dire :

- une case non remplie : correspondant à la valeur -1 dans la grille de sudoku
- une case dont le code des possibilités est de taille minimale, c'est-à-dire pour laquelle on a le moins de choix possibles

Cette stratégie permettra d'éviter l'explosion combinatoire lors de la construction de la solution.

Question 9

Écrire une fonction `case_plus_containte : grille -> grille -> (int * int)` telle que `case_plus_contrainte sudoku possibilites` retourne les coordonnées (i, j) d'une case la plus contrainte parmi les cases non encore remplies. Dans le cas où toutes les cases de la grille sont déjà remplies, on retournera la valeur $(-1, -1)$.

Pour pouvoir réaliser le *backtracking* nous aurons besoin de revenir sur nos pas et donc de conserver les informations antérieures. Pour cela nous écrivons une petite fonction permettant de copier une grille.

Question 10

Écrire une fonction `grille_copy : grille -> grille` construisant une nouvelle grille contantant les mêmes valeurs que la grille initiale.

Question 11

Compléter la fonction ci-dessous

```
bt_resolution : grille -> grille -> int -> grille
```

telle que `bt_resolution sudoku possibilites r` tente par *backtracking* de remplir r cases vides du problème et retourne la nouvelle grille obtenue.

```
exception Impossible;; (* Pour gérer les echecs *)

let rec bt_resolution sudoku possibles r =
    if r = 0 then
        (* COMPLETER *)
    else
        let (i, j) = case_plus_contrainte sudoku possibles in
        (* Si (i, j) = (-1, -1) *)
            (* Erreur : raise Impossible *)
        (* Sinon : *)

        (* La fonction ci-dessous à completer
         * construit une nouvelle instance
         * du problème : (s2, possibles2, r2) correspondant à l'ajout
         * du chiffre k dans la case (i, j) et appelle la résolution
         * de façon récursive
         *)

        let essaye_chiffre k =
            (* COMPLETER *)
            (* Attention a bien creer s2 et possibles2 à partir
             * de copies de sudoku et possibilites *)
        in

        (* La fonction ci-dessous va essayer un a un les chiffres pro-
poses
           dans la liste chiffres passee en arguments *)
        let rec aux chiffres = match chiffres with
        | [] -> raise Impossible (* si tout echoue on signale l'echec *)
        | t::q ->
            begin
                try
                    essaye_chiffre t (* tentative *)
                with
                | Impossible -> aux q (* si on a eu un echec on es-
saie avec un
                                         autre chiffre possible *)
            end
        in
        aux (* COMPLETER *)
;;
```

Question 12

Déduire de tout ce qui précède une petite fonction `solve : grille -> grille` synthétisant toutes les étapes de la résolution du Sudoku.

Vous pouvez maintenant tenter d'utiliser votre programme pour résoudre le Sudoku diabolique proposé. La résolution ne devrait prendre que quelques secondes.

L'utilisation de la statégie de case la plus contrainte d'abord, rend réellement l'algorithme plus efficace. Si on avait choisi de remplir les cases dans l'ordre usuel on aurait pu trouver certaines instances très difficiles à rédoudre, par exemple :

```
let antibt = [|
    [| -1; -1; -1; -1; -1; -1; -1; -1; -1|];
    [| -1; -1; -1; -1; -1; 3; -1; 8; 5 |];
    [| -1; -1; 1; -1; 2; -1; -1; -1; -1|];
    [| -1; -1; -1; 5; -1; 7; -1; -1; -1|];
    [| -1; -1; 4; -1; -1; -1; 1; -1; -1|];
    [| -1; 9; -1; -1; -1; -1; -1; -1; -1|];
    [| 5; -1; -1; -1; -1; -1; -1; 7; 3|];
    [| -1; -1; 2; -1; 1; -1; -1; -1; -1|];
    [| -1; -1; -1; -1; 4; -1; -1; -1; 9|]
  |]
;;
```