Projet MrSudoku : solveur SAT

Projet MrSudoku : encodage du Sudoku en SAT

Dans le cadre du projet MrSudoku, la résolution du jeu de Sudoku par DPLL nécessite l'encodage de la grille du Sudoku sous forme d'une formule propositionnelle.

Les principes d'encodage sont décrits ci-dessous.

```
(ns mrsudoku.sat.encode
  (:require [midje.sweet :refer [fact]]
            [mrsudoku.grid :as g]))
```

Pour les exemples et les tests, nous allons travailler sur la grille suivante :

```
(def ex-grille @#'g/sudoku-grid)
```

```
(println (g/grid->str ex-grille))
```

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


nil
```

```
(fact
  (g/cell ex-grille 1 1) => {:status :init, :value 5}
  (g/cell ex-grille 4 2) => {:status :init, :value 1}
  (g/cell ex-grille 9 9) => {:status :init, :value 9}
  (g/cell ex-grille 4 5) => {:status :init, :value 8}
  (g/cell ex-grille 4 6) => {:status :empty})
```

Codage des cellules remplies

Nous allons commencer par construire la formule qui permet d'encoder les cellules déjà remplies, et qui ont donc un champs `:status` différent de `:empty` et une valeur associée au champ `:value`.

Chaque cellule remplie contient un entier entre 1 et 9. Nous devons coder cet entier sous la forme d'un nombre binaire composé de 4 bits:

— l'entier 1 se code 0001

— l'entier 2 se code 0010
— ...
— l'entier 7 se code 0111
— l'entier 8 se code 1000
— l'entier 9 se code 1001

Pour commencer, on crée une fonction `log-binary` qui convertit en entier en une séquence de 0 et de 1.

```
(declare log-binary)
```

```
(fact
  (log-binary 5) => '(1 0 1)
  (log-binary 9) => '(1 0 0 1)
  (log-binary 42) => '(1 0 1 0 1 0))
```

Comme nous voulons coder les entiers de 1 à 9 sur exactement 4 bits, on utilise la fonction suivante :

```
(declare pad-seq)
```

```
(fact
  (pad-seq (log-binary 1) 0 4) => '(0 0 0 1)
  (pad-seq (log-binary 5) 0 4) => '(0 1 0 1)
  (pad-seq (log-binary 9) 0 4) => '(1 0 0 1)
  (pad-seq '(:a :b :c) :x 7) => '(:x :x :x :x :a :b :c))
```

On obtient la fonction suivante pour l'encodage d'une valeur de cellule déjà remplie :

```
(defn encode-value
  "Encode une valeur de cellule `n` (entre 1 et 9) en une séquence de 4 bits."
  [n]
  (pad-seq (log-binary n) 0 4))
```

Chaque cellule est repérée dans la grille par deux coordonnées :

— `cx` le numéro de la colonne (de 1 à 9)
— `cy` le numéro de la ligne (de 1 à 9)

On nomme aussi chaque bit de la représentation booléenne, dans `(0 1 0 1)` (codage de la valeur 5):

— le bit de poids fort (valeur 0), en premier dans la séquence, se nomme `b3`
— le bit suivant (valeur 1) se nomme `b2`
— le bit suivant (valeur 0) se nomme `b1`
— le bit de poids faible (valeur 1), dernier élément de la séquence, se nomme `b0`.

La fonction suivante permet de créer une variable propositionnelle spécifique à un bit dans la valeur d'une cellule :

```
(defn mkcellbit
  "Créer la variable du `bit` spécifié (entre 0 poids faible
   et 3 poids fort) pour la cellule située en `cx` (colonne)
  et `cy` (ligne)."
  [cx cy bit]
  (symbol (str "x" cx "y" cy "b" bit)))
```

```
(fact
  (mkcellbit 6 2 2) => 'x6y2b2
  (mkcellbit 4 3 0) => 'x4y3b0)
```

On définit maintenant la fonction principale d'encodage.

```
(declare encode-num)
```

```
(fact
  ;; bits '(0 1 0 1) donne x6y2b0 /\ (not x6y2b1) /\ x6y2b2 /\ (not x6y2 b3)
  (encode-num 6 2 5) => '(and x6y2b0 (and (not x6y2b1) (and x6y2b2
                                                          (and (not x6y2b3) true))))
  ;; bits '(0 0 0 1) donne x6y2b0 /\ (not x6y2b1) /\ (not x6y2b2) /\ (not x6y2 b3)
  (encode-num 2 3 1) => '(and x2y3b0 (and (not x2y3b1) (and (not x2y3b2)
                                                          (and (not x2y3b3) true))))
  ;; bits '(1 0 0 1) donne x6y2b0 /\ (not x6y2b1) /\ (not x6y2b2) /\ x6y2 b3)
  (encode-num 2 3 9) => '(and x2y3b0 (and (not x2y3b1) (and (not x2y3b2)
                                                          (and x2y3b3 true)))))
```

Remarque : votre fonction pourra renvoyer une formule équivalente, l'ordre entre les bits importe peu et le `true` terminal n'est pas nécessaire (mais il peut faciliter l'écriture de la fonction).

On peut maintenant encoder la formule qui représente toutes les cellules déjà saisies dans le sudoku :

```
(defn encode-inits
  "Formule d'encodage des cellules déjà remplies dans la `grille`"
  [grille]
  (g/reduce-grid
    (fn [acc cx cy cell]
      (if (= (:status cell) :empty)
        acc
        (list 'and (encode-num cx cy (:value cell)) acc))) true grille))
```

Encodage des cellules vides

Les cellules vides peuvent contenir un entier entre 1 et 9, ce qui se traduit par une disjonction de la forme :

`en cx,cy: <encodage de 1> \/ <encodage de 2> \/ ... \/ <encodage de 9>`

On a donc effectué l'essentiel du travail nécessaire avec la fonction `encode-num` précédente.

```
(declare encode-vide)
```

```
(fact
  (encode-vide 7 3) ;; encodage d'une cellule vide en cx=7 et cy=3
  => '(or (and
            x7y3b0
            (and (not x7y3b1) (and (not x7y3b2) (and x7y3b3 true))))
          (or (and (not x7y3b0) (and (not x7y3b1) (and (not x7y3b2) (and x7y3b3 true))))
              (or (and x7y3b0 (and x7y3b1 (and x7y3b2 (and (not x7y3b3) true))))
                  (or (and (not x7y3b0) (and x7y3b1 (and x7y3b2 (and (not x7y3b3) true))))
                      (or (and x7y3b0 (and (not x7y3b1)
                                           (and x7y3b2 (and (not x7y3b3) true))))
                          (or (and (not x7y3b0)
                                   (and (not x7y3b1)
                                        (and x7y3b2 (and (not x7y3b3) true))))
                              (or (and x7y3b0 (and x7y3b1
                                                   (and (not x7y3b2)
                                                        (and (not x7y3b3) true))))
                                  (or (and (not x7y3b0) (and x7y3b1
                                                              (and (not x7y3b2)
                                                                   (and (not x7y3b3) true))))
                                      (or (and x7y3b0 (and (not x7y3b1)
                                                           (and (not x7y3b2)
```

```
                                                        (and (not x7y3b3) true))))
                                                false))))))))))
```

La fonction principage d'encodage est la suivante :

```
(defn encode-vides
  "Formule d'encodage des cellules vides de la `grille`."
  [grille]
  (g/reduce-grid
    (fn [acc cx cy cell]
      (if (= (:status cell) :empty)
        (list 'and (encode-vide cx cy) acc)
        acc)) true grille))
```

Encodage des contraintes de différence

Les principales contraintes du sudoku concernent les contraintes de différence suivantes:

- toutes les cellules d'une même colonne doivent contenir des valeurs distinctes
- idem pour les lignes
- idem pour chaque bloc : les carrés de (taille 3x3) sur la grille (de 9x9)

On peut établir ces contraintes sur une base commune.

Distinctions 2 à 2

On commence par construire une formule permettant de distinguer deux cellules. Les sous-cas à considérer sont les suivants:

- si les deux cellules sont vides on encode la distinction des 4 bits individuellement.
- si une des deux cellules contient une valeur et l'autre non, on utilise la valeur déjà présente pour contraindre la cellule vide
- si les deux cellules contiennent déjà des nombres, on résout le problème directement (en levant éventuellement une exception si les deux valeurs sont identiques).

Pour le premier cas, on définit la fonction suivante.

```
(declare distinct-empty-empty)

(fact
  ;; les cellules entre cx1=2,cy1=3 et cx2=2,cy=5 doivent être distinctes
  (distinct-empty-empty 2 3 2 5)
  => '(and (<=> l3c2b0 (not l5c2b0)) ; bits b0 distincts
           (and (<=> l3c2b1 (not l5c2b1)) ; b1
                (and (<=> l3c2b2 (not l5c2b2)) ; b2
                     (and (<=> l3c2b3 (not l5c2b3))  ; b3
                          true)))))
```

Pour le deuxième cas, on suppose que la première cellule contient une valeur `cval1` et la deuxième cellule en `cx2,cy2` est vide. L'objectif est de construire la proposition la plus simple possible permettant de distinguer les cellules. Par exemple, si `cval1=5` (en binaire `(0 1 0 1)`) alors on doit code :

```
en cx2,cy2: <le bit b3 est à 1> \/ <le bit b2 est à 0> \/ <le bit b1 est à 1>
\/ <le bit b0 est à 0>
```

On définit la fonction suivante.

```
(declare distinct-filled-empty)
```

```
(fact
  ;; cval1=5  et cx2=3,cy2=6
  (distinct-filled-empty 5 3 6)
  => '(or (not x3y6b0) (or x3y6b1 (or (not x3y6b2) (or x3y6b3 false)))))
```

La formule de comparaison 2 à 2 se définit alors de la façon suivante :

```
(defn distinct-pair [cx1 cy1 cell1 cx2 cy2 cell2]
  (cond
    ;; cas 1 : deux cellules vides
    (and (= (:status cell1) :empty) (= (:status cell2) :empty))
    (distinct-empty-empty cx1 cy1 cx2 cy2)
    ;; cas 2a : la première est vide
    (= (:status cell1) :empty)
    (distinct-filled-empty (:value cell2) cx1 cy1)
    ;; cas 2b : la seconde est vide
    (= (:status cell2) :empty)
    (distinct-filled-empty (:value cell1) cx2 cy2)
    ;; cas 3 : la formule n'est pas satisfiable
    (= (:value cell1) (:value cell2))
    false
    ;; cas par défaut : contrainte satisfaite
    :else true))
```

```
(fact
  (distinct-pair 2 3 {:status :empty} 5 6 {:status :empty})
  => '(and (<=> l3c2b0 (not l6c5b0))
           (and (<=> l3c2b1 (not l6c5b1))
                (and (<=> l3c2b2 (not l6c5b2))
                     (and (<=> l3c2b3 (not l6c5b3)) true))))
  (distinct-pair 2 3 {:status :init :value 5} 5 6 {:status :empty})
  => '(or (not x5y6b0) (or x5y6b1 (or (not x5y6b2) (or x5y6b3 false))))
  (distinct-pair 5 6  {:status :empty} 2 3 {:status :init :value 5})
  => '(or (not x5y6b0) (or x5y6b1 (or (not x5y6b2) (or x5y6b3 false))))
  (distinct-pair 2 3 {:status :init :value 5} 5 6 {:status :init :value 6})
  => true
  (distinct-pair 2 3 {:status :init :value 5} 5 6 {:status :init :value 5})
  => false)
```

Distinctions n-aires

On en déduit une fonction permettant de construire les distinctions pour une séquence de cellules composée de la façon suivante :

```
([cx1 cy1 cell1] [cx2 cy2 cell2] ... [cxN cyN cellN])
```

```
(declare distinc-cells)
```

```
(fact
  (distinct-cells '([1 1 {:status :empty}]
                    [2 1 {:status :init :value 5}]
                    [2 2 {:status :empty}]))
  => '(and true
           (and (and (or (not x2y2b0)
                         (or x2y2b1 (or (not x2y2b2) (or x2y2b3 false)))) true)
                (and (and (and (<=> l1c1b0 (not l2c2b0))
```

```
                                (and (<=> l1c1b1 (not l2c2b1))
                                     (and (<=> l1c1b2 (not l2c2b2))
                                          (and (<=> l1c1b3 (not l2c2b3)) true))))
                           (and (or (not x1y1b0) (or x1y1b1
                                                     (or (not x1y1b2) (or x1y1b3 false))))
                                true))
                      true))))
```

Remarque : on fera attention que deux cellules données ne soit comparées qu'une seule fois (par exemple si on a comparé `[cx1 cy1 cell1]` vs. `[cx2 cy2 cell2]` alors on ne fait pas la comparaison `[cx2 cy2 cell2]` vs `[cx1 cy1 cell1]`)

Contraintes du Sudoku

On peut maintenant terminer l'encodage du sudoku.

Contraintes de lignes

```
(defn fetch-row
  "Récupère les cellules de la ligne `cy` de la `grille`."
  [grille cy]
  (map (fn [cx cell]
         [cx cy cell]) (range 1 10) (g/row grille cy)))

(defn encode-rows
  "Formule d'encodage des contraintes de ligne dans la grille."
  [grille]
  (loop [cy 1, phi true]
    (if (<= cy 9)
      (recur (inc cy) (list 'and (distinct-cells (fetch-row grille cy))
                            phi))
      phi)))
```

Contraintes de colonne

```
(defn fetch-col
  "Récupère les cellules de la colonne `cx` de la `grille`."
  [grille cx]
  (map (fn [cy cell]
         [cx cy cell]) (range 1 10) (g/col grille cx)))

(defn encode-cols
  "Formule d'encodage des contraintes de colonne dans la grille."
  [grille]
  (loop [cx 1, phi true]
    (if (<= cx 9)
      (recur (inc cx) (list 'and (distinct-cells (fetch-col grille cx))
                            phi))
      phi)))
```

Contraintes de bloc

Rappel : le block 1 est le premier en haut à gauche de la grille, le 2 celui à sa droite, . . . , et le 9 est le dernier en bas à droite

```
(defn block-rows
  "Séquence des coordonnées de ligne des cellules du block `b`"
  [b]
  (map #(+ (* (mod (dec b) 3) 3)
           1 (mod % 3)) (range 0 9)))

(defn block-cols
  [b]
  (map #(+ (quot % 3) 1
           (* (quot (dec b) 3) 3)) (range 0 9)))

(defn fetch-block
  "Récupère les cellules du block `b` de la `grille`."
  [grille b]
   (map vector (block-rows b) (block-cols b) (g/block grille b)))

(defn encode-blocks
  "Formule d'encodage des contraintes de bloc dans la grille."
  [grille]
  (loop [b 1, phi true]
    (if (<= b 9)
      (recur (inc b) (list 'and (distinct-cells (fetch-block grille b))
                           phi))
      phi)))
```

Encodage de la grille

On arrive enfin à la formule qui code l'ensemble des contraintes de la grille du sudoku.

```
(defn encode-sudoku
  "Formule d'encodage de la grille du Sudoku."
  [grille]
  (list 'and
        (encode-inits grille)
        (list 'and (encode-vides grille)
              (list 'and (encode-rows grille)
                    (list 'and (encode-cols grille)
                          (list 'and (encode-blocks grille)))))))

;; (encode-sudoku ex-grille)
;; => .... attention : formule énorme ! ....
```

. . . il ne nous reste plus qu'à convertir cette formule en DCNF et lancer DPLL pour la résolution . . .