1 Prise en main de `minisat`

Dans tout le TP, nous allons utiliser le programme `minisat`, qui est un solveur de satisfaisabilité de formules propositionnelles (SAT).

Voilà un exemple d'une formule en CNF :

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2)$$

et la même formule sous le format de base de `minisat` (**dimacs cnf**) :

```
p cnf 3 4
1 2 3 0
-1 -2 -3 0
2 -3 0
1 -2 0
```

- La ligne `p cnf 3 4` indique qu'il s'agit d'une CNF avec 3 variables et 4 clauses.
- Chaque ligne suivante décrit une clause.
- Un nombre positif indique une variable.
- Un nombre négatif indique la négation d'une variable.
- `0` termine la ligne.

Pour utiliser le programme, il suffit de taper la commande

```
> minisat fichier.cnf fichier.out
```

où `fichier.cnf` contient l'exemple ci-dessus. Le résultat est affiché comme ceci :

```
...
Memory used           : 21.00 MB
CPU time              : 0 s

SATISFIABLE
```

Dans le fichier `fichier.out` vous trouvez une affectation satisfaisante :

```
SAT
1 -2 -3 0
```

Ce fichier indique que la formule est saitsfaisable (`SAT`) et qu'une affectation satisfaisante est donnée par $(x_1, x_2, x_3) = (1, 0, 0)$.

Notez que ce n'est pas obligatoire d'entrer les bonnes valeurs pour le nombre de variables et le nombre de clauses. On peut mettre `p cnf 0 0` et dans ce cas `minisat` va donner un avertissement et lui même calculer les bonnes valeurs.

Une petite introduction au format dimacs cnf et l'utilisation de `minisat` se trouve à l'adresse

http://www.dwheeler.com/essays/minisat-user-guide.html

► **Question 1** ◄ Vérifiez que vous trouvez le coupable dans l'exemple "Mystère" du cours en utilisant `minisat`.

2 Sudoku

On formalise le jeu Sudoku en CNF afin de résoudre des grilles à l'aide de `minisat`. Comme langage de programmation on utilise le Python.

2.1 Les règles du jeu

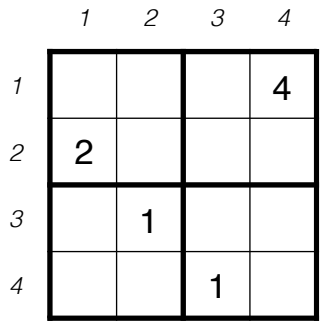

Figure 1: Une grille de Sudoku de 4×4 cases.

Le jeu de Sudoku se joue normalement sur une grille de 9×9 cases. Dans un premier temps, on se contente de joueur une version plus simple sur une grille de 4×4 cases. D'ailleurs, on essaye de faire en sorte que le programme puisse facilement être modifié pour résoudre des

grilles de 9×9 cases en utilisant un paramètre N pour indiquer une grille de $N \times N$ cases où N peut être choisi à $4, 9, 16, ...$

Le but du jeu est de **remplir les cases** avec des chiffres tirés de l'ensemble $\{1, \ldots, 4\}$ ($\{1, \ldots, N\}$ en général) en satisfaisant les contraintes suivantes :

1. toute ligne doit contenir chacun des chiffres $1, \ldots, 4$
2. toute colonne doit contenir chacun des chiffres $1, \ldots, 4$
3. les 2×2 *sous-grilles* de taille 2×2 (indiquées en gras dans la figure) doivent contenir chacun des chiffres $1, \ldots, 4$
4. aucune case ne doit contenir plus d'un chiffre

► **Question 2** ◄ Résoudre la grille de la figure 1 à la main.

2.2 Littéraux, clauses et formules

Introduire une variable x_{ijk} pour $i = 1, \ldots, 4$, $j = 1, \ldots, 4$ et $k = 1, \ldots, 4$ avec l'interprétation

$$x_{ijk} = 1 \text{ si la case } (i, j) \text{ contient le chiffre } k$$

On utilise une *représentation interne* d'un littéral qui est un tuple (s, i, j, k) où i, j, k sont les indices de la variable et s le *signe* du littéral, soit 1 (littéral positif), soit -1 (littéral négatif). Les fonctions suivantes permettent de créer des littéraux.

```
def var(i,j,k):
    """Return the literal Xijk."""
    return (1,i,j,k)

def neg(l):
    """Return the negation of the literal l."""
    (s,i,j,k) = l
    return (-s,i,j,k)
```

On représente une clause par une liste de littéraux et une formule en CNF par une liste de clauses.

Exemple. La liste `cnf = [[var(2,2,2), neg(var(3,3,3))], [neg(var(1,2,3)), neg (var(4,4,4))], [var(1,4,4)]]` représente la formule suivante.

$$(x_{222} \vee \neg x_{333}) \wedge (\neg x_{123} \vee \neg x_{444}) \wedge (x_{144})$$

► **Question 3** ◄ Écrire une fonction `initial_configuration` qui renvoie une formule qui décrit la configuration initiale de la figure 1. C-à-d, une formule qui dit "il y a le chiffre 4 dans la case $(1, 4)$, le chiffre 2 dans la case $(2, 1)$ et le chiffre 1 dans les cases $(3, 2)$ et $(4, 3)$."

2.3 Formaliser les règles – lignes, colonnes et sous-grilles

On va maintenant générer une formule qui formalise les règles du jeu. La fonction suivante génère la conjonction de formules renvoyées par quatre fonctions que vous allez écrire dans les questions qui suivent. Les trois premières exprimeront que chaque chiffre doit apparaitre dans chaque ligne, chaque colonne et chaque sous-grille. La dernière exprime que chaque case contient au plus un chiffre.

```
def generate_rules(N):
    """Return a list of clauses describing the rules of the game."""
    cnf = []
    cnf.extend(row_rules(N))
    cnf.extend(column_rules(N))
    cnf.extend(subgrid_rules(N))
    cnf.extend(one_digit_rules(N))
    return cnf
```

2.3.1 Lignes

Pour toute ligne $i = 1, \ldots, 4$ et tout chiffre $k = 1, \ldots, 4$, le chiffre k doit apparaitre sur la ligne i. Cela est imposé en utilisant une clause par ligne et chiffre :

$$x_{i1k} \vee x_{i2k} \vee x_{i3k} \vee x_{i4k} \qquad \text{pour tout } i = 1, \ldots, 4 \text{ et } k = 1, \ldots, k$$

La fonction suivante prend en argument la taille N de la grille et renvoie une formule (un ensemble de clauses) qui exprime toutes les contraintes sur les lignes.

```
def row_rules(N):
    """Return a list of clauses describing the rules for the rows."""
    cnf = []
    for i in range(1,N+1):
        for k in range(1,N+1):
            # create a clause containing all variables
            # for a fixed row i and digit k
            clause = [var(i,j,k) for j in range(1,N+1)]
            cnf.append(clause)
    return cnf
```

2.3.2 Colonnes

Pour toute colonne $j = 1, \ldots, 4$ et tout chiffre $k = 1, \ldots, 4$, le chiffre k doit apparaitre dans la colonne j. Cela est imposé en utilisant une clause par colonne et chiffre :

$$x_{1jk} \vee x_{2jk} \vee x_{3jk} \vee x_{4jk} \qquad \text{pour tout } j = 1, \ldots, 4 \text{ et } k = 1, \ldots, k$$

► **Question 4** ◄ Écrire une fonction `column_rules` qui renvoie une formule qui exprime toutes les contraintes sur les colonnes.

2.3.3 Sous-grilles

Pour toute sous-grille et tout chiffre $k = 1, \ldots, 4$, le chiffre k doit apparaitre dans la sous-grille. Cela est imposé en utilisant une clause par sous-grille et chiffre.

► **Question 5** ◄ Écrire une fonction `subgrid_rules` qui renvoie une formule qui exprime toutes les contraintes sur les sous-grilles.

2.3.4 Au plus un chiffre dans chaque case

Pour l'instant, rien n'interdit à une case de contenir plusieurs chiffres.

► **Question 6** ◄ Écrire une fonction `at_most_one` qui prend en argument une liste L de littéraux et renvoie la formule en CNF qui dit "au plus un des littéraux de L est vrai".

Exemple.

```
>>> lst = [var(1, 1, 1), var(2, 2, 2), var(3, 3, 3)]
>>> at_most_one(lst)
[[(-1, 1, 1, 1), (-1, 2, 2, 2)], [(-1, 1, 1, 1), (-1, 3, 3, 3)], [(-1, 2,
    2, 2), (-1, 3, 3, 3)]]
```

Conseil : En Python, vous pouvez utiliser `itertools.combinations(lst, 2)` pour itérer sur tout couple d'éléments d'une liste `lst`. Exemple d'utilisation :

```
>>> import itertools
>>> for (x,y) in itertools.combinations(lst, 2):
...     print(neg(x),neg(y))
...
(-1, 1, 1, 1) (-1, 2, 2, 2)
(-1, 1, 1, 1) (-1, 3, 3, 3)
```

```
(-1, 2, 2, 2) (-1, 3, 3, 3)
```

► **Question 7** ◄ Pour toute case (i, j), on veut qu'il existe **au plus un** chiffre k tel que x_{ijk} est vraie. Écrire une fonction `rules_one_digit` qui renvoie une formule qui exprime cette contrainte pour chaque case. Utiliser la fonction `at_most_one`.

2.4 Résolution d'une grille

Pour générer un fichier `.cnf` on doit traduire la représentation interne d'un littéral à sa *représentation externe* (un seul entier).

► **Question 8** ◄ Écrire une fonction qui prend en argument un littéral en représentation interne et la taille de la grille. La fonction renvoie sa représentation externe. Utiliser la correspondance suivante :

$$(s, i, j, k) \to s \times (N^2 \times (i-1) + N \times (j-1) + k)$$

► **Question 9** ◄ Écrire une fonction qui prend une formule en représentation interne et écrit la formule en format `.cnf` sur un fichier.

Exemple. Pour la liste `cnf = [[var(2,2,2), neg(var(3,3,3))], [neg(var(1,2,3)), neg(var(4,4,4))], [var(1,4,4)]]` et avec la taille $N = 4$ de la grille, la fonction génère le fichier suivant.

```
p cnf 64 3
22 -43 0
-7 -64 0
16 0
```

► **Question 10** ◄ Générer la formule qui décrit la configuration initiale de la figure 1 et les règles du jeu. La résoudre avec `minisat`.

2.5 Solveur de Sudoku

► **Question 11** ◄ Écrire un programme qui :

1. lit la représentation d'une grille dans un fichier, cf. les exemples de grilles « ICI » ;
2. génère le fichier `.cnf` correspondant à la grille et les règles du jeu ;
3. lance `minisat` sur le fichier `.cnf` ;
4. récupère la solution de `minisat` et affiche celle-ci dans le terminal.

Exemple d'utilisation :

```
> python3 sudoku.py grid4x4-1.txt
1 3 2 4
2 4 3 1
3 1 4 2
4 2 1 3
```

Pour récupérer la solution de `minisat`, vous pouvez utiliser les fonctions suivantes.

```
def unpack_var(val, N):
    """Return the indices i, j, k corresponding to the variable number
   val."""
    val = val - 1
    k = val % N
    val = val // N
    j = val % N
    val = val // N
    i = val % N
    return i+1, j+1, k+1

def read_output(filename, N):
    """Read an output file from minisat and fill a Sudoku grid."""
    solution = [[0 for j in range(N)] for i in range(N)]
    with open(filename, 'r') as f:
        content = f.readlines()
        l = content[1]
        vals = [int(lit) for lit in l.split() if int(lit) > 0]
        for val in vals:
            i, j, k = unpack_var(val, N)
            solution[i-1][j-1] = k
    return solution
```