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 $(x_1, x_2, x_3) = (1, 0, 0)$.

Noter 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

https://dwheeler.com/essays/minisat-user-guide.html

► **Exercice 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

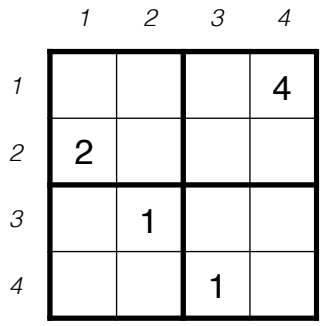

Figure 1: Une grille de Sudoku de 4×4 cases.

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

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

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

Télécharger le fichier `sudoku-incomplete.py` sur elearning. Noter que ce fichier contient des doctests pour certaines fonctions et que vous êtes encouragé à en produire d'autres (dans le fichier `.py` ou dans un fichier `.txt` auxiliaire).

On utilise une *représentation interne* (dans le programme Python) 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.

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

► **Exercice 2** ◄ 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 At least one, at most one

On commence par implanter les "macros" `at_least_one` et `at_most_one`.

► **Exercice 3** ◄ Écrire une fonction `at_least_one(L)` qui prend en argument une liste L de littéraux et renvoie **la formule** en CNF qui dit "au moins un des littéraux de L est vrai". Pour ce faire, la fonction place la liste `L` comme la seule clause à l'intérieure d'une formule CNF (voir l'exemple).

Exemple.

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

► **Exercice 4** ◄ Écrire une fonction `at_most_one(L)` 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(['abc', 42, 'bonjour!'], 2):
...     print('{',x,',',y,'}')
...
{ abc , 42 }
{ abc , bonjour! }
{ 42 , bonjour! }
```

2.4 Formaliser les règles

On va maintenant générer les clauses qui formalisent les règles du jeu. La fonction suivante génère la conjonction de formules renvoyées par quatre fonctions que vous allez écrire dans les exercices qui suivent. La première exprime que chaque case contient exactement un chiffre. Les trois suivantes exprimeront que chaque chiffre doit apparaitre dans chaque ligne, chaque colonne et chaque sous-grille.

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

2.4.1 Formaliser l'application $(i, j) \mapsto k$

La première chose à faire est de s'assurer que les variables x_{ijk} représente une application qui donne à chaque case (i, j) **exactement un** chiffre k.

► **Exercice 5** ◄ Pour chaque case (i, j), on veut qu'il existe **exactement un** chiffre k tel que x_{ijk} est vraie. **Compléter** la fonction `assignment_rules(N)` ci-dessous qui génère et renvoie une formule qui exprime cette contrainte pour toute case. Vous pouvez utiliser les fonctions `at_least_one` et `at_most_one` précédemment implantées.

```
def assignment_rules(N):
    """Return a list of clauses describing the rules for the assignment
    (i,j) -> k."""
    cnf = []
    for i in range(1,N+1):
        for j in range(1,N+1):
            # add clauses to cnf saying that (i,j) contains
            # *exactly* one of the digits k=1..N
            ...
    return cnf
```

Pour tester vos fonctions, vous pouvez simplement afficher les clauses générées et les vérifier.

```
>>> cnf = assignment_rules(4)
>>> len(cnf)
112
>>> for clause in cnf: print(clause)
...
[(1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 1, 3), (1, 1, 1, 4)]
[(-1, 1, 1, 1), (-1, 1, 1, 2)]
[(-1, 1, 1, 1), (-1, 1, 1, 3)]
[(-1, 1, 1, 1), (-1, 1, 1, 4)]
[(-1, 1, 1, 2), (-1, 1, 1, 3)]
[(-1, 1, 1, 2), (-1, 1, 1, 4)]
[(-1, 1, 1, 3), (-1, 1, 1, 4)]
[(1, 1, 2, 1), (1, 1, 2, 2), (1, 1, 2, 3), (1, 1, 2, 4)]
...
```

2.4.2 Lignes

Pour toute ligne $i = 1, \ldots, 4$ et tout chiffre $k = 1, \ldots, 4$, on veut que le chiffre k apparait exactement une fois sur la ligne i.

► **Exercice 6** ◄ Écrire la fonction `row_rules(N)` qui renvoie une formule qui exprime toutes les contraintes sur les lignes.

2.4.3 Colonnes

Pour toute colonne $j = 1, \ldots, 4$ et tout chiffre $k = 1, \ldots, 4$, on veut que le chiffre k apparait exactement une fois dans la colonne j.

► **Exercice 7** ◄ Écrire la fonction `column_rules(N)` qui renvoie une formule qui exprime toutes les contraintes sur les colonnes.

2.4.4 Sous-grilles

Pour toute sous-grille et tout chiffre $k = 1, \ldots, 4$, on veut que le chiffre k apparait exactement une fois dans la sous-grille.

► **Exercice 8** ◄ Écrire la fonction `subgrid_rules(N)` qui renvoie une formule qui exprime toutes les contraintes sur les sous-grilles.

2.5 Résolution d'une grille

Pour générer un fichier `.cnf` on doit traduire la représentation interne d'un littéral à sa *représentation externe* (l'entier qui représente ce littéral dans le fichier `.cnf`).

► **Exercice 9** ◄ Écrire une fonction `literal_to_integer(l, N)` qui prend en argument un littéral en représentation interne et la taille de la grille. La fonction renvoie sa représentation externe (un entier). Utiliser la correspondance suivante :

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

► **Exercice 10** ◄ Écrire une fonction qui prend une formule en représentation interne et la taille de la grille 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
```

► **Exercice 11** ◄ Faire en sorte que le programme génère la formule qui décrit la configuration initiale de la figure 1 ainsi que les règles du jeu. Le programme écrit la formule en format `.cnf` sur un fichier dont le nom est spécifié sur la ligne de commande :

```
$ python3 sudoku.py sudoku.cnf
```

Résoudre le fichier `sudoku.cnf` par `minisat` et **vérifier la solution**. Un exemple d'un fichier `.cnf` et de solution pour la grille de la figure 1 se trouvent sur elearning.

2.6 Solveur de Sudoku

► **Exercice 12** ◄ Écrire un programme qui :

1. lit la représentation d'une grille dans un fichier, cf. les exemples de grilles sur elearning ;
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`, vérifie qu'elle est correcte 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
```