# TD 5. Modélisation et DPLL

**Exercice 1.** DPLL à la main

Nous rappelons que l'algorithme DPLL décide la satisfiabilité d'une formule en CNF par l'application des règles suivantes à l'ensemble de ses clauses, en donnant la priorité aux règles (unit) et (pure) (car elles ne demandent jamais un retour en arrière) avant de brancher avec ($\mathsf{split}_P$) ou ($\mathsf{split}_{\neg P}$).

| | | |
|---|---|---|
| (unit) | $F \cup \{\{\ell\}\} \to_{\mathsf{dpll}} F[\top/\ell]$ | |
| (pure) | $F \to_{\mathsf{dpll}} F[\top/\ell]$ | où $\ell \in \mathrm{Pur}(F)$ |
| ($\mathsf{split}_P$) | $F \to_{\mathsf{dpll}} F[\top/P]$ | où $P \in \mathrm{fp}(F)$ |
| ($\mathsf{split}_{\neg P}$) | $F \to_{\mathsf{dpll}} F[\top/\neg P]$ | où $P \in \mathrm{fp}(F)$ |

Appliquer l'algorithme DPLL à la formule :

$$\begin{aligned}\varphi = \quad & (P \vee Q \vee R) \quad \wedge \quad (P \vee \neg Q \vee \neg R) \quad \wedge \quad (P \vee \neg S) \\ \wedge\, & (\neg Q \vee \neg R \vee \neg S) \quad \wedge \quad (\neg P \vee \neg Q \vee R) \quad \wedge \quad (T \vee U) \\ \wedge \quad & (T \vee \neg U) \quad \wedge \quad (Q \vee \neg T) \quad \wedge \quad (\neg R \vee \neg T)\end{aligned}$$

**Exercice 2.** Le format DIMACS

Le but de cet exercice est de comprendre le format DIMACS qui est utilisé par les solveurs SAT, et par l'algorithme que vous allez implémenter dans le mini-projet 1.

Nous rappelons que le type `formula` peut être défini en OCaml par :

```
type formula =
  | Prop of int
  | Neg of formula
  | And of formula * formula
  | Or of formula * formula
```

On supposera dans cet exercice que les variables propositionnelles sont représentées par des entiers *strictement positifs*. Nous rappelons aussi qu'une formule en forme clausale peut être représentée comme une liste de listes d'entiers : par exemple, la formule en CNF

$$(P_1 \vee P_2 \vee P_3) \wedge (\neg P_1 \vee Q_1) \wedge (\neg Q_2 \vee \neg P_3)$$

peut être représenté par la liste de listes `[[1;2;3];[-1;4];[-5;3]]`, où chaque proposition $P_i$ est représentée par l'entier $i$ et chaque $Q_i$ est représentée par l'entier $i+3$. On remarque que si un entier strictement positif $n$ représente une proposition $Q$, alors $-n$ représente le littéral $\neg Q$. C'est l'essentiel de la représentation au format DIMACS des formules en forme clausale.

Dans cet exercice, nous souhaitons écrire une fonction `dimacs_of_cnf` qui renvoie la liste de listes d'entiers qui correspond à une formule donnée sous forme clausale.

(a) Donnez une liste de listes d'entiers qui représente la formule en CNF $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, où

$$\varphi_1 = P_1 \vee \neg P_2 \vee \neg P_3, \quad \varphi_2 = \neg P_1 \vee P_2 \vee \neg P_3, \quad \varphi_3 = \neg P_1 \vee \neg P_2 \vee P_3\ .$$

(b) Écrivez une fonction (en OCaml) `is_clause : formula -> bool` qui retourne `true` si et seulement si la formule donnée en entrée est une clause, c'est-à-dire une disjonction de littéraux. Par exemple, avec les notations de la question précédente, `is_clause phi1` retourne `true` mais `is_clause phi` retourne `false`.

(c) Écrivez une fonction `is_cnf : formula -> bool` qui retourne `true` si et seulement si la formule donnée en entrée est en CNF, c'est-à-dire une conjonction de clauses. Par exemple, `is_cnf phi` retourne `true` mais `is_cnf (Neg phi1)` retourne `false`.

(d) Écrivez une fonction `list_of_clause : formula -> int list` qui retourne la liste d'entiers correspondante à la clause donnée en argument, ou échoue avec le message `"not a clause"` si la formule donnée en entrée n'est pas une clause. Par exemple, `list_of_clause phi2` retourne la liste `[1; -2; 3]` (ou une permutation).

*Indications.* On peut utiliser `failwith "not a clause"` pour les échecs. Pour concaténer deux listes, on peut utiliser `List.rev_append`.

(e) Écrivez une fonction `clauses_of_cnf : formula -> formula list` qui retourne la liste de clauses d'une formule en CNF, ou échoue avec le message `"not in CNF"`. Par exemple, `clauses_of_cnf phi` retourne la liste `[phi1; phi2; phi3]` (ou une permutation).

(f) Écrivez une fonction `dimacs_of_cnf : formula -> (int * int) list option` qui retourne `Some l` si la formule donnée en entrée est en CNF, où `l` est la liste de listes d'entiers qui correspond à la formule, et qui retourne `None` autrement.

**Exercice 3.** Principe des tiroirs

Le principe des tiroirs affirme que $k$ pigeons peuvent être mis dans $n$ tiroirs, de manière à ce que chaque pigeon soit dans un tiroir distinct, si et seulement si $k \leq n$.

(a) Pour $k, n$ fixés, donner un ensemble de propositions qui permettent de modéliser le placement des $k$ pigeons dans les $n$ tiroirs.

(b) Donner une formule sous CNF modélisant le fait que chaque pigeon est dans au moins l'un des tiroirs.

(c) Donner une formule sous CNF modélisant le fait chaque tiroir contient au plus un pigeon.

(d) Utiliser l'algorithme DPLL pour montrer que 2 pigeons peuvent se répartir entre 2 tiroirs.

(e) (Bonus) Utiliser l'algorithme DPLL pour montrer que 3 pigeons ne peuvent pas se repartir entre 2 tiroirs.

**Exercice 4.** Modélisation par recouvrement exact et résolution par DPLL

Le problème de recouvrement exact peut être décrit abstraitement de la manière suivante : étant donnée une matrice de 0s et de 1s, existe-t-il un ensemble de lignes contenant exactement un 1 dans chaque colonne ?

(a) La matrice suivante a-t-elle un tel ensemble de lignes ?

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

(b) Le premier intérêt du problème de recouvrement exact est qu'il est facile à coder dans SAT-CNF. Étant donné un problème de recouvrement exact, expliquer comment construire une formule en CNF $\varphi$ telle que les interprétations qui satisfont $\varphi$ correspondent aux solutions du problème de recouvrement exact.

(c) Appliquer cette méthode à la matrice ci-dessus et appliquer DPLL à la CNF obtenue.

(d) Le second intérêt du problème de recouvrement exact est qu'il permet d'encoder simplement un grand nombre de problèmes combinatoires naturels. Étant donnée une grille de Sudoku $9 \times 9$ à résoudre, expliquer comment construire un problème de recouvrement exact dont les solutions correspondent à celles de la grille.