

# Sémantique et SAT

OPTION INFORMATIQUE - TP n° 1.2 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ☞ représenter une valuation par un entier codé en binaire
- ☞ expliquer le problème SAT
- ☞ résoudre SAT par la force brute
- ☞ savoir simplifier une expression logique d'après les règles de simplification
- ☞ résoudre SAT par l'algorithme de Quine

## A Valuation d'une formule sous la forme d'un entier

On choisit de représenter les formules logiques comme dans le TD précédent mais en ajoutant le constructeur de l'implication :

```
1 type formule =  
2   | T (* true *)  
3   | F (* false *)  
4   | Var of int (* variable *)  
5   | Not of formule (* negation *)  
6   | And of formule * formule (* conjonction *)  
7   | Or of formule * formule (* disjonction *)  
8   | Imp of formule * formule (* implication *)
```

Soit une formule logique  $\phi$  qui possède  $n$  variables propositionnelles. Chaque variable peut être vraie ou fausse et représentée par un bit à 0 pour F et 1 pour T. Une valuation de la formule logique peut donc être représentée par un nombre entier.

■ **Exemple 1 — Valuation et nombre binaire.** Soit  $\phi = a \wedge b \vee c$ . Cette formule comporte trois variables propositionnelles.  $a, b, c$  peuvent être vraies ou fausses. On attribue (arbitrairement) des numéros aux variables en commençant à zéro et en incrémentant de un : par exemple,  $(a, 0)$ ,  $(b, 1)$  et  $(c, 2)$ . On peut alors représenter une valuation de  $\phi$  par un nombre entier codé sur trois bits. Par exemple :

- $000_2 = 0_{10} \longrightarrow (c, b, a) = (F, F, F)$
- $001_2 = 1_{10} \longrightarrow (c, b, a) = (F, F, T)$
- $010_2 = 2_{10} \longrightarrow (c, b, a) = (F, T, F)$
- $100_2 = 4_{10} \longrightarrow (c, b, a) = (T, F, F)$
- $101_2 = 5_{10} \longrightarrow (c, b, a) = (T, F, T)$

Le bit de poids faible (0) représente la valuation de  $a$ , le second celle de  $b$  et le bit de poids fort

celle de  $c$ . L'ensemble des valuations possibles peut donc être représenté par un ensemble d'entiers :  $\{0, 1, 2, 3, 4, 5, 6, 7\} = \llbracket 0, 2^n - 1 \rrbracket$ .

**Par la suite, on suppose que toutes les variables d'une formule logique sont indexées par un numéro et la numérotation commence à zéro.** On dispose également de la fonction qui permet de calculer le numéro maximal attribué à une variable (`max_var` dans le TD précédent).

## B SAT par la force brute

B1. Écrire une fonction de signature `get_var_k_from_v : int -> int -> bool` qui prend comme paramètre :

1. une valuation  $v$  sous la forme d'un entier
2. un entier  $k$  représentant le numéro d'une variable

et qui renvoie `true` si  $k$  est vraie dans la valuation  $v$ , `false` sinon. Pour cette fonction, on utilisera les fonctions OCaml :

- `Int.logand` : ET bit à bit sur deux entiers. Par exemple, `Int.logand 5 2` renvoie 0 et `Int.logand 5 3` renvoie 1. En effet :  $101_2 \text{ ET } 010_2 = 000_2$  et  $101_2 \text{ ET } 011_2 = 001_2$ .
- `Int.shift_left` : décalage à gauche d'un entier. Elle permet de rapidement calculer une puissance de deux. Par exemple : `Int.shift_left 1 3` vaut 8, car  $1000_2 = 2^3 = 8$ .

et la technique du masquage.

B2. Écrire une fonction de signature `evaluation : int -> formule -> bool` qui évalue une formule logique d'après une valuation donnée par un entier.

B3. Écrire une fonction de signature `brute_force_satisfiability : formule -> bool` qui statue sur la satisfaisabilité d'une formule logique en opérant par la force brute. Cette fonction prend comme paramètre une formule logique et renvoie :

- `true` si une valuation  $v$  satisfait la formule,
- `false` sinon

**Toutes les valuations possibles sont testées, dès qu'une valuation qui satisfait la formule est trouvée, la fonction renvoie `true`.** On procédera par récursivité en commençant par la valuation 0. La condition d'arrêt est qu'une valuation ne peut pas être plus grande que  $2^n - 1$  si la formule possède  $n$  variables propositionnelles.

B4. Dédurre de la fonction précédente une fonction de signature `opt_brute_force_satisfiability : formule -> int option` qui renvoie la valuation trouvée ou `None`, en utilisant un type optionnel.

B5. Tester la validité de la fonction précédente sur les formules :

- $f_1 : a \vee (b \wedge c)$
- $f_2 : (a \wedge \neg b) \vee (b \wedge \neg(c \vee a))$
- $f_3 : (\neg a \wedge b \vee d) \vee (c \wedge \neg(b \vee d))$

```
1   let f4 =
2     let p1 = Var 0
3     and p2 = Or (Var 1, Not(Var 2))
4     and s1 = And(Not(Var 0), Not(Var 1))
5     and s2 = Or (Var 1, And( Not (Var 0), Not (Var 2)))
6     in let p = Or ( And (p1, p2), And(Not p1, Not p2) )
7     and s = Or (And(s1, s2), And(Not s1, Not s2) )
8     in And (p, s);;
```

- B6. Construire une formule logique à trois variables propositionnelles insatisfaisable et le vérifier.
- B7. Quelle est la complexité dans le pire des cas de l'algorithme de résolution de SAT par la force brute en fonction du nombre de variables propositionnelles?

## C Algorithme de Quine

### a Règles de simplification de formules logiques après substitution

L'algorithme de Quine se fonde sur des simplifications de formules : lorsqu'une variable propositionnelle est remplacée par  $\top$  ou  $\perp$ , on peut en déduire des simplifications par équivalence de formules logiques.

Pour les éléments de base  $\top$  et  $\perp$  aucune simplification n'est possible. Pour une variable propositionnelle, on en peut pas non plus simplifier davantage le constructeur `Var`. Par contre, grâce aux règles de simplification énoncées dans le cours, on peut programmer des constructeurs `not`, `and`, `or` et `imp` qui simplifient les expressions auxquels ils s'appliquent lorsque c'est possible. On appelle ces fonctions des constructeurs élégants<sup>1</sup>.

Le point de départ de la programmation est la fonction suivante :

```

1 let rec simplify f =
2   match f with
3   | Var _ | T | F -> f (* pas de simplifications possibles *)
4   | Not f         -> s_not (simplify f)
5   | And(f1, f2)   -> s_and (simplify f1) (simplify f2)
6   | Or(f1, f2)    -> s_or  (simplify f1) (simplify f2)
7   | Imp(f1, f2)   -> s_imp (simplify f1) (simplify f2)

```

On cherche donc à écrire les fonctions `s_not`, `s_and`, `s_or` et `s_imp`.

- C1. Écrire un constructeur élégant pour le constructeur `not` de signature `s_not : formule -> formule` qui construit la négation logique de la formule passée en paramètre en la simplifiant éventuellement.
- C2. Écrire un constructeur élégant pour le constructeur `and` de signature `s_and : formule -> formule -> formule` qui construit la conjonction de deux formules passées en paramètre en simplifiant éventuellement.
- C3. Écrire un constructeur élégant pour le constructeur `or` de signature `s_or : formule -> formule -> formule` qui construit la disjonction de deux formules passées en paramètre en simplifiant éventuellement.
- C4. Écrire un constructeur élégant pour le constructeur `imp` de signature `s_imp : formule -> formule -> formule` qui construit l'implication des deux formules passées en paramètre en simplifiant éventuellement.

### b Programmation de l'algorithme

On se propose d'implémenter l'algorithme de Quine (cf. algorithme 1).

---

1. smart constructors

**Algorithme 1** Algorithme Quine (SAT)

---

```

1: Fonction QUINE_SAT( $f$ ) ▷  $f$  est une formule logique
2:   SIMPLIFIER( $f$ )
3:   si  $f \equiv \top$  alors
4:     renvoyer Vrai
5:   sinon si  $f \equiv \perp$  alors
6:     renvoyer Faux
7:   sinon
8:     Choisir une variable  $x$  parmi les variables propositionnelles restantes de  $f$ 
9:     renvoyer QUINE( $f[x \leftarrow \top]$ ) || QUINE( $f[x \leftarrow \perp]$ )

```

---

- C1. Écrire une fonction de signature `subst : int -> formule -> formule -> formule` qui substitue une variable  $k$  par une formule  $r$  dans une formule  $f$ . On l'utilisera ainsi : `subst 2  $\top$   $f$`  si l'on veut substituer la variable numéro 2 par la formule  $\top$  dans la formule  $f$ .
- C2. Tester la fonction en remplaçant par exemple la variable de numéro 0 par  $\top$  dans  $f_1$ .
- C3. Tester la simplification de la formule  $f_1$  dans le cas où la variable de numéro 0 a été remplacée par  $\top$ .
- C4. Écrire une fonction de signature `quine_sat : formule -> bool` qui statue sur la satisfaisabilité d'une formule logique. Cette fonction prend en paramètre une formule logique et renvoie un booléen, vrai si la formule est satisfaisable, faux sinon.
- C5. Tester la fonction sur  $f_1$  et sur la formule suivante :

$$((p \implies (q \vee r)) \wedge (s \implies \neg r \vee t)) \implies (p \implies s)$$