

Sémantique et SAT

OPTION INFORMATIQUE - TP n° 2.2 - 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 Évaluation de formules logiques

- A1. Construire la table de vérité pour chaque formule ci-dessous. En déduire la nature des propositions.
- (a) $F_1 = P \Rightarrow (Q \Rightarrow P)$
 - (b) $F_2 = \neg P \Rightarrow (P \Rightarrow Q)$.
- A2. Pour chaque formule ci-dessous, trouver par la table de vérité et par le calcul.
- (a) la forme normale disjonctive de $F_2 = (P \vee \neg Q) \Rightarrow R$
 - (b) la forme normale conjonctive de $F_1 = (P \wedge (Q \Rightarrow R) \Rightarrow S)$

B 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 :

```
type formule =  
  | T (* true *)  
  | F (* false *)  
  | Var of int (* variable *)  
  | Not of formule (* negation *)  
  | And of formule * formule (* conjonction *)  
  | Or of formule * formule (* disjonction *)  
  | Imp of formule * formule (* implication *)
```

Soit une formule logique ϕ 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 ϕ 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).

C SAT par la force brute

C3. 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 ET $010_2 = 000_2$ et 101_2 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.

C4. Écrire une fonction de signature `evaluation : int -> formule -> bool` qui évalue une formule logique d'après une valuation donnée par un entier.

C5. 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.

C6. D  duire de la fonction pr  c  dente une fonction de signature `opt_brute_force_satisfiability : formule -> int option` qui renvoie la valuation trouv  e ou `None`, en utilisant un type optionnel.

C7. Tester la validit   de la fonction pr  c  dente sur les formules :

- $f_1 : a \vee (b \wedge c)$
- $f_2 : (a \wedge \neg b) \vee (b \wedge \neg(c \vee a))$
- $f_3 : (\neg a \wedge b \vee d) \vee (c \wedge \neg(b \vee d))$
- ```
let f4 =
 let p1 = Var 0
 and p2 = Or (Var 1, Not(Var 2))
 and s1 = And(Not(Var 0), Not(Var 1))
 and s2 = Or (Var 1, And(Not (Var 0), Not (Var 2)))
 in let p = Or (And (p1, p2), And(Not p1, Not p2))
 and s = Or (And(s1, s2), And(Not s1, Not s2))
 in And (p, s);;
```

C8. Construire une formule logique    trois variables propositionnelles insatisfaisable et le v  rifier.

C9. Quelle est la complexit   dans le pire des cas de l'algorithme de r  solution de SAT par la force brute en fonction du nombre de variables propositionnelles?

## D Algorithme de Quine

### a R  gles de simplification de formules logiques apr  s substitution

L'algorithme de Quine se fonde sur des simplifications de formules : lorsqu'une variable propositionnelle est remplac  e par  $\top$  ou  $\perp$ , on peut en d  duire des simplifications par   quivalence de formules logiques.

Pour les   l  ments 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, gr  ce aux r  gles de simplification   nonc  es 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   l  gants<sup>1</sup>.

Le point de d  part de la programmation est la fonction suivante :

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

On cherche donc      crire les fonctions `s_not`, `s_and`, `s_or` et `s_imp`.

D1.   crire un constructeur   l  gant pour le constructeur `not` de signature `s_not : formule -> formule` qui construit la n  gation logique de la formule pass  e en param  tre en la simplifiant   ventuellement.

1. smart constructors

- D2. Écrire un constructeur élégant pour le constructeur **and** de signature  
 $s\_and : \text{formule} \rightarrow \text{formule} \rightarrow \text{formule}$  qui construit la conjonction de deux formules passées en paramètre en simplifiant éventuellement.
- D3. Écrire un constructeur élégant pour le constructeur **or** de signature  
 $s\_or : \text{formule} \rightarrow \text{formule} \rightarrow \text{formule}$  qui construit la disjonction de deux formules passées en paramètre en simplifiant éventuellement.
- D4. Écrire un constructeur élégant pour le constructeur **imp** de signature  
 $s\_imp : \text{formule} \rightarrow \text{formule} \rightarrow \text{formule}$  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).

---

### 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($f[x \leftarrow \perp]$)

```

---

- D1. Écrire une fonction de signature  $\text{subst} : \text{int} \rightarrow \text{formule} \rightarrow \text{formule} \rightarrow \text{formule}$  qui substitue une variable  $k$  par une formule  $r$  dans une formule  $f$ . On l'utilisera ainsi :  $\text{subst } 2 \ \top \ f$  si l'on veut substituer la variable numéro 2 par la formule  $\top$  dans la formule  $f$ .
- D2. Tester la fonction en remplaçant par exemple la variable de numéro 0 par  $\top$  dans  $f_1$ .
- D3. Tester la simplification de la formule  $f_1$  dans le cas où la variable de numéro 0 a été remplacée par  $\top$ .
- D4. Écrire une fonction de signature  $\text{quine\_sat} : \text{formule} \rightarrow \text{bool}$  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.
- D5. Tester la fonction sur  $f_1$  et sur la formule suivante :

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