

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 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).

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 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.

Solution :

```
let get_var_k_from_v v k = Int.logand v (Int.shift_left 1 k) != 0;;
```

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

Solution :

```
let rec evaluation v f =
  match f with
  | T -> true
  | F -> false
  | Var k -> get_var_k_from_v v k
  | Not p -> not (evaluation v p)
  | And (p, q) -> evaluation v p && evaluation v q
  | Or (p, q) -> evaluation v p || evaluation v q
  | Imp (p, q) -> not (evaluation v p) || (evaluation v q);;
```

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.

Solution :

```
let brute_force_satisfiability f =
  let n = nb_var f in
  (* n : nombre de variables propositionnelles de type int de 0 à n-1 *)
  let v_limit = Int.shift_left 1 n in
  (* v_limit = 2^n est la première valuation impossible *)
  let rec check_val valuation =
    match valuation with
    | v when v < v_limit -> if evaluation v f
                           then true
                           else check_val (v + 1)
    | _ -> false (* condition d'arrêt de la récursivité *)
  in check_val 0;;
```

- 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.

Solution :

```
let opt_brute_force_satisfiability f =
  let n = nb_var f in
  (* n : nombre de variables propositionnelles de type int de 0 à n-1 *)
  let v_limit = Int.shift_left 1 n in
  (* v_limit = 2^n est la première valuation impossible *)
  let rec check_val valuation =
    match valuation with
    | v when v < v_limit -> if evaluation v f
                           then Some v
                           else check_val (v + 1)
    | _ -> None (* condition d'arrêt de la récursivité *)
  in check_val 0;;
```

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

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

**Solution :**

```
let f1 = Or ((Var 0), And ((Var 1),(Var 2)));;
let f2 = Or (And ((Var 0),(Not (Var 1))), And ((Var 1), Not(Or ((Var 2),(Var
0)))));;
let f3 = Or (Or (And (Not (Var 0) , (Var 1)), (Var 3)), And ((Var 2), Not(
Or((Var 1), (Var 3))))) ;;
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);;

brute_force_satisfiability f1;;
brute_force_satisfiability f2;;
brute_force_satisfiability f3;;
brute_force_satisfiability f4;;
```

B6. Construire une formule logique à trois variables propositionnelles insatisfaisable et le vérifier.

**Solution :**  $f_5 : (\neg a \wedge b \wedge c) \wedge (a \wedge \neg(b \vee c))$

```
let f5 = And (And (And (Not (Var 0) , (Var 1)), (Var 2)), And ((Var 0), Not
(Or((Var 1), (Var 2)))));;
brute_force_satisfiability f5;;
```

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?

**Solution :** Hors appels récursifs, la fonction `brute_force_satisfiability` n'exécute qu'un nombre fini d'opérations de complexité constante en  $O(1)$ . Dans le pire des cas, toutes les valuations possibles sont testées et donc  $2^n$  appels récursifs sont effectués. La complexité est donc exponentielle en  $O(2^n)$ .  
Cet algorithme est équivalent à la réalisation d'une table de vérité pour la formule considérée.

## 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 :

```
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 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.

**Solution :**

```
let s_not f =
 match f with
 | F -> T
 | T -> F
 | _ -> Not f
```

- 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.

**Solution :**

```
let s_and f1 f2 =
 match (f1, f2) with
 | F, _ | _, F -> F
 | T, f | f, T -> f
 | _, _ -> And (f1, f2)
```

- 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.

**Solution :**

1. smart constructors

```

let s_or f1 f2 =
 match (f1, f2) with
 | T, _ | _, T -> T
 | F, f | f, F -> f
 | _, _ -> Or (f1, f2)

```

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.

**Solution :**

```

let s_imp f1 f2 =
 match (f1, f2) with
 | F, _ -> T (* ex falso quodlibet *)
 | _, T -> T
 | T, f -> f
 | f, F -> s_not f
 | _, _ -> Imp (f1, f2)

```

## b Programmation de l'algorithme

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

### Algorithme 1 Algorithme Quine (SAT)

|                                                                                                                                                                                                                                                                                                                                                                                                                   |                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|
| <pre> 1: <b>Fonction</b> QUINE_SAT(<i>f</i>) 2:   SIMPLIFIER(<i>f</i>) 3:   <b>si</b> <i>f</i> ≡ ⊤ <b>alors</b> 4:     renvoyer Vrai 5:   <b>sinon si</b> <i>f</i> ≡ ⊥ <b>alors</b> 6:     renvoyer Faux 7:   <b>sinon</b> 8:     Choisir une variable <i>x</i> parmi les variables propositionnelles restantes de <i>f</i> 9:     renvoyer QUINE(<i>f</i>[<i>x</i> ← ⊤])    QUINE(<i>f</i>[<i>x</i> ← ⊥]) </pre> | <p>▷ <i>f</i> est une formule logique</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|

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 ⊤ f` si l'on veut substituer la variable numéro 2 par la formule *⊤* dans la formule *f*.

**Solution :**

```

let rec subst k r f =
 match f with
 | F -> F
 | T -> T
 | Var j when k = j -> r

```

|             |                                    |
|-------------|------------------------------------|
| Var _       | -> f                               |
| Not f       | -> Not (subst k r f)               |
| And(f1, f2) | -> And(subst k r f1, subst k r f2) |
| Or(f1, f2)  | -> Or(subst k r f1, subst k r f2)  |
| Imp(f1, f2) | -> Imp(subst k r f1, subst k r f2) |

---

C2. Tester la fonction en remplaçant par exemple la variable de numéro 0 par T dans  $f_1$ .

**Solution :**

```
let f1 = Or ((Var 0), And ((Var 1),(Var 2)));;
simplify f1;;
```

---

C3. Tester la simplification de la formule  $f_1$  dans le cas où la variable de numéro 0 a été remplacée par T.

**Solution :**

```
simplify (subst 0 T f1);;
```

---

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.

**Solution :**

```
let rec quine_sat f valuation =
 match simplify f with
 | T -> true
 | F -> false
 | f -> let v = max_var f 0 in simple_quine_sat (subst v T f) ||
 simple_quine_sat (subst v F f);;
```

---

C5. Tester la fonction sur  $f_1$  et sur la formule suivante :

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

**Solution :**

```
quine_sat f1;;
let a = Imp(Var 0, Or(Var 1, Var 2));;
let b = Imp(Var 3, Or(Not (Var 2), Var 4));;
let c = Imp(Var 0, Var 3);;
```

```
let d = Imp(And(a,b), c);;
quine_sat d;;
```

---