Logique des propositions

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Logique formelle

Une logique est définie par une syntaxe et une sémantique.
On distingue deux types de logique :

- la logique des propositions formalise le raisonnement ;

  $(a \text{ ou } b) \Rightarrow b$ est une formule de la logique propositionnelle.

- la logique des prédicats formalise le langage des mathématiques.

  $\exists x \mid (x \text{ ou } b) \Rightarrow b$ est une formule de la logique des prédicats.

Logique formelle

Une logique est définie par une syntaxe et une sémantique.
On distingue deux types de logique :

- la logique des propositions formalise le raisonnement ;

  $(a \text{ ou } b) \Rightarrow b$ est une formule de la logique propositionnelle.

- la logique des prédicats formalise le langage des mathématiques.

  $\exists x \mid (x \text{ ou } b) \Rightarrow b$ est une formule de la logique des prédicats.

La sémantique permet d'attribuer aux variables libres la valeur vrai ou faux pour en déduire par des règles de calcul la valeur d'une formule :

- pour $a =$ vrai et $b =$ faux la formule $(a \text{ ou } b) \Rightarrow b$ est fausse ;
- pour $b =$ faux la formule $\exists x \mid (x \text{ ou } b) \Rightarrow b$ est vraie.

Formules logiques

On utilise un alphabet Σ constitué :

- de *constantes* Faux et Vrai représentées par les valeurs 0 et 1 ;
- de *variables* représentées par les lettres romaines $a, b, c \ldots$;
- d'un connecteur unaire $\neg$;
- de quatre connecteurs binaires $\wedge, \vee, \Rightarrow, \Leftrightarrow$.

Formules logiques

On utilise un alphabet Σ constitué :

- de *constantes* Faux et Vrai représentées par les valeurs 0 et 1 ;
- de *variables* représentées par les lettres romaines $a, b, c \ldots$;
- d'un connecteur unaire $\neg$;
- de quatre connecteurs binaires $\wedge, \vee, \Rightarrow, \Leftrightarrow$.

Une formule logique se définit à l'aide des règles suivantes :

- toute constante et toute variable est une formule ;
- si F est une formule, $(\neg F)$ est une formule ;
- si F_1 et F_2 sont des formules et $\triangle$ un opérateur binaire, alors $(F_1 \triangle F_2)$ est une formule.

Formules logiques

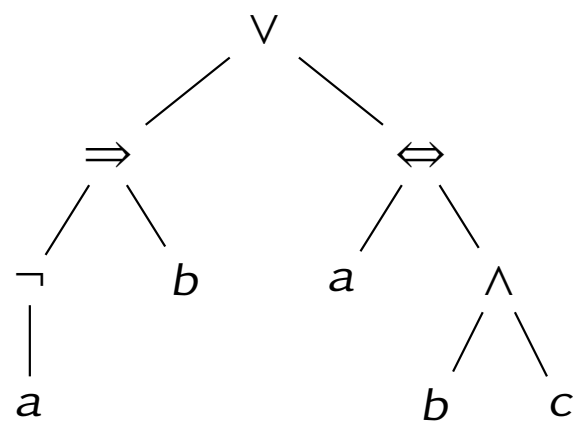
On utilise un alphabet Σ constitué :

- de *constantes* Faux et Vrai représentées par les valeurs 0 et 1 ;
- de *variables* représentées par les lettres romaines a, b, c ... ;
- d'un connecteur unaire $\neg$;
- de quatre connecteurs binaires $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.

Une formule logique se définit à l'aide des règles suivantes :

- toute constante et toute variable est une formule ;
- si F est une formule, $(\neg F)$ est une formule ;
- si F_1 et F_2 sont des formules et $\triangle$ un opérateur binaire, alors $(F_1 \triangle F_2)$ est une formule.

Par exemple, $(\neg a \Rightarrow b) \vee (a \Leftrightarrow b \wedge c)$ est une formule qui possède la représentation arborescente :



Sémantique

Le contexte d'une formule est une application qui à chacune des variables présente associe la valeur 0 ou 1. À tout contexte correspond une évaluation de la formule, définie inductivement par les tables de vérité associées aux opérateurs logiques.

a	$\neg a$
0	1
1	0

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \Rightarrow b$
0	0	1
0	1	1
1	0	0
1	1	1

a	b	$a \Leftrightarrow b$
0	0	1
0	1	0
1	0	0
1	1	1

Sémantique

Le contexte d'une formule est une application qui à chacune des variables présente associe la valeur 0 ou 1. À tout contexte correspond une évaluation de la formule, définie inductivement par les tables de vérité associées aux opérateurs logiques.

Exemple. Une table de vérité permet d'associer à chaque contexte la valeur de la formule logique (ici $\neg b \Rightarrow a \wedge \neg b$) :

a	b	$\neg b$	$a \wedge \neg b$	$\neg b \Rightarrow a \wedge \neg b$
0	0	1	0	0
0	1	0	0	1
1	0	1	1	1
1	1	0	0	1

Sémantique

Le contexte d'une formule est une application qui à chacune des variables présente associe la valeur 0 ou 1. À tout contexte correspond une évaluation de la formule, définie inductivement par les tables de vérité associées aux opérateurs logiques.

On peut enrichir le langage par de nouveaux opérateurs, en précisant leur table de vérité. C'est le cas des opérateurs XOR, NAND et NOR :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

Formules logiques équivalentes

Deux formules logiques F_1 et F_2 sont dites logiquement équivalentes lorsque leurs évaluations coïncident dans tout contexte. On note alors : $F_1 \equiv F_2$.

Exemples.

- $(a \Rightarrow b) \equiv (\neg b \Rightarrow \neg a)$ (raisonnement par contraposée) ;
- $a \equiv (\neg a \Rightarrow 0)$ (raisonnement par l'absurde) ;
- $(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$;
- $(a \Rightarrow b \vee c) \equiv (a \wedge \neg b \Rightarrow c)$.

Formules logiques équivalentes

Deux formules logiques F_1 et F_2 sont dites logiquement équivalentes lorsque leurs évaluations coïncident dans tout contexte. On note alors : $F_1 \equiv F_2$.

Exemples.

- $(a \Rightarrow b) \equiv (\neg b \Rightarrow \neg a)$ (raisonnement par contraposée) ;
- $a \equiv (\neg a \Rightarrow 0)$ (raisonnement par l'absurde) ;
- $(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$;
- $(a \Rightarrow b \vee c) \equiv (a \wedge \neg b \Rightarrow c)$.

Principe de substitution

Si $a_1, a_2, \ldots, a_n$ sont des variables et $F_1(a_1, \ldots, a_n)$ et $F_2(a_1, \ldots, a_n)$ deux formules logiquement équivalentes faisant intervenir ces variables, alors quelles que soient les formules logiques $f_1, \ldots, f_n$, les formules $F_1(f_1, \ldots, f_n)$ et $F_2(f_1, \ldots, f_n)$ restent logiquement équivalentes.

Le principe de substitution associé aux équivalences usuelles permet de simplifier certaines formules logiques.

Satisfiabilité et tautologies

Les formules logiques qui prennent la valeur vrai dans tout contexte sont des tautologies.

Exemples.

- $a \vee \neg a$ (le *tiers exclu*) ;
- $\neg(a \wedge \neg a)$ (la *non contradiction*) ;
- $\Big((a \Rightarrow b) \wedge (b \Rightarrow c)\Big) \Rightarrow (a \Rightarrow c)$ (la transitivité de l'implication).

Remarque. Deux formules logiques F_1 et F_2 sont équivalentes si et seulement si $(F_1 \Leftrightarrow F_2)$ est une tautologie.

Satisfiabilité et tautologies

Les formules logiques qui prennent la valeur vrai dans tout contexte sont des tautologies.

Exemples.

- $a \vee \neg a$ (le *tiers exclu*) ;
- $\neg(a \wedge \neg a)$ (la *non contradiction*) ;
- $\Big((a \Rightarrow b) \wedge (b \Rightarrow c)\Big) \Rightarrow (a \Rightarrow c)$ (la transitivité de l'implication).

Remarque. Deux formules logiques F_1 et F_2 sont équivalentes si et seulement si $(F_1 \Leftrightarrow F_2)$ est une tautologie.

Une formule logique est dite satisfiable lorsqu'il existe un contexte pour lequel la formule logique prend la valeur vrai.

Une formule logique F est satisfiable si et seulement si $\neg F$ n'est pas une tautologie. Reconnaître une tautologie ou une formule satisfiable sont donc deux problèmes équivalents.

Lois de De Morgan

Si a et b sont deux variables propositionnelles, on dispose des équivalences suivantes :

$$\neg(a \wedge b) \equiv \neg a \vee \neg b \qquad \text{et} \qquad \neg(a \vee b) \equiv \neg a \wedge \neg b.$$

Lois de De Morgan

Si a et b sont deux variables propositionnelles, on dispose des équivalences suivantes :

$$\neg(a \wedge b) \equiv \neg a \vee \neg b \qquad \text{et} \qquad \neg(a \vee b) \equiv \neg a \wedge \neg b.$$

Les formules suivantes :

$$a \wedge b \equiv \neg(\neg a \vee \neg b) \qquad (a \Rightarrow b) \equiv (\neg a \vee b)$$
$$a \vee b \equiv \neg(\neg a \wedge \neg b) \qquad (a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$$

associées au principe de substitution prouvent que toute formule logique est équivalente à une formule logique définie à l'aide des seuls connecteurs logiques $\neg$ et $\wedge$ (ou $\neg$ et $\vee$).
On dit que $\{\neg, \wedge\}$ et $\{\neg, \vee\}$ sont des systèmes complets de connecteurs logiques.

Algèbre de Boole

Dans l'algèbre de Boole,

$$\neg a \text{ est noté } \overline{a}, \quad a \wedge b \text{ est noté } ab, \quad a \vee b \text{ est noté } a + b.$$

Ces notations se justifient par les propriétés de commutativité, d'associativité et de distributivité des opérateurs de conjonction et de disjonction :

$$a + b \equiv b + a \qquad a + (b + c) \equiv (a + b) + c \qquad a(b + c) \equiv ab + ac$$
$$ab \equiv ba \qquad a(bc) \equiv (ab)c$$

Algèbre de Boole

Dans l'algèbre de Boole,

$$\neg a \text{ est noté } \overline{a}, \quad a \wedge b \text{ est noté } ab, \quad a \vee b \text{ est noté } a + b.$$

Ces notations se justifient par les propriétés de commutativité, d'associativité et de distributivité des opérateurs de conjonction et de disjonction :

$$a + b \equiv b + a \qquad a + (b + c) \equiv (a + b) + c \qquad a(b + c) \equiv ab + ac$$
$$ab \equiv ba \qquad a(bc) \equiv (ab)c$$

Certaines équivalences demeurent surprenantes : $a + bc \equiv (a + b)(a + c)$.

$$(a + b)(a + c) = a^2 + ab + ac + bc = a + ab + ac + bc = a(1 + b) + ac + bc$$
$$= a + ac + bc = a(1 + c) + bc = a + bc.$$

Algèbre de Boole

Dans l'algèbre de Boole,

$$\neg a \text{ est noté } \overline{a}, \quad a \wedge b \text{ est noté } ab, \quad a \vee b \text{ est noté } a + b.$$

Ces notations se justifient par les propriétés de commutativité, d'associativité et de distributivité des opérateurs de conjonction et de disjonction :

$$a + b \equiv b + a \qquad a + (b + c) \equiv (a + b) + c \qquad a(b + c) \equiv ab + ac$$
$$ab \equiv ba \qquad a(bc) \equiv (ab)c$$

Certaines équivalences demeurent surprenantes : $a + bc \equiv (a + b)(a + c)$.

En effet, ces deux lois ne confèrent pas à $\{0, 1\}$ une structure d'anneau. Pour retrouver la structure d'anneau de $\mathbb{Z}/2\mathbb{Z}$, il ne faut pas utiliser le « ou » logique mais le « ou exclusif », qu'on note dans ce contexte $\oplus$:

$+$	0	1
0	0	1
1	1	1

$\oplus$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

Disjonctions et conjonctions

On appelle littéral toute formule logique de la forme a ou $\overline{a}$.
On appelle disjonction toute formule logique F s'écrivant $F_1 + F_2 + \cdots + F_n$.
On appelle conjonction toute formule logique F s'écrivant $F_1 F_2 \cdots F_n$.

Disjonctions et conjonctions

On appelle littéral toute formule logique de la forme a ou $\overline{a}$.
On appelle disjonction toute formule logique F s'écrivant $F_1 + F_2 + \cdots + F_n$.
On appelle conjonction toute formule logique F s'écrivant $F_1 F_2 \cdots F_n$.

Toute formule est équivalente à une disjonction de conjonctions de littéraux, et à une conjonction de disjonctions de littéraux.

Disjonctions et conjonctions

On appelle littéral toute formule logique de la forme a ou $\overline{a}$.
On appelle disjonction toute formule logique F s'écrivant $F_1 + F_2 + \cdots + F_n$.
On appelle conjonction toute formule logique F s'écrivant $F_1 F_2 \cdots F_n$.

Toute formule est équivalente à une disjonction de conjonctions de littéraux, et à une conjonction de disjonctions de littéraux.

Soit F une formule construite à l'aide des opérateurs de négation et de conjonction. On montre par récurrence sur la longueur de F que F est équivalente à une disjonction de conjonctions de littéraux et à une conjonction de disjonctions de littéraux.

- C'est clair si F est un littéral.

Disjonctions et conjonctions

On appelle littéral toute formule logique de la forme a ou $\overline{a}$.
On appelle disjonction toute formule logique F s'écrivant $F_1 + F_2 + \cdots + F_n$.
On appelle conjonction toute formule logique F s'écrivant $F_1 F_2 \cdots F_n$.

Toute formule est équivalente à une disjonction de conjonctions de littéraux, et à une conjonction de disjonctions de littéraux.

Soit F une formule construite à l'aide des opérateurs de négation et de conjonction. On montre par récurrence sur la longueur de F que F est équivalente à une disjonction de conjonctions de littéraux et à une conjonction de disjonctions de littéraux.

- Si $F = \overline{F_1}$, alors :

$$F_1 \equiv A_1 + \cdots + A_p \Longrightarrow F \equiv \overline{A_1} \cdots \overline{A_p}$$

où A_i est une conjonction de littéraux et $\overline{A_i}$ est une disjonction de littéraux.

$$F_1 \equiv A_1 \cdots A_p \Longrightarrow F \equiv \overline{A_1} + \cdots + \overline{A_p}$$

où A_i est une disjonction de littéraux et $\overline{A_i}$ est une conjonction de littéraux.

Disjonctions et conjonctions

On appelle littéral toute formule logique de la forme a ou $\overline{a}$.
On appelle disjonction toute formule logique F s'écrivant $F_1 + F_2 + \cdots + F_n$.
On appelle conjonction toute formule logique F s'écrivant $F_1 F_2 \cdots F_n$.

Toute formule est équivalente à une disjonction de conjonctions de littéraux, et à une conjonction de disjonctions de littéraux.

Soit F une formule construite à l'aide des opérateurs de négation et de conjonction. On montre par récurrence sur la longueur de F que F est équivalente à une disjonction de conjonctions de littéraux et à une conjonction de disjonctions de littéraux.

- Si $F = F_1 F_2$, alors :

$$F_1 \equiv A_1 \cdots A_p \quad \text{et} \quad F_2 \equiv B_1 \cdots B_q \Longrightarrow F \equiv A_1 \cdots A_p B_1 \cdots B_q$$

où A_i et B_j sont des disjonctions de littéraux.

$$F_1 \equiv A_1 + \cdots + A_p \text{ et } F_2 \equiv B_1 + \cdots + B_q \Longrightarrow F \equiv \sum_{i,j} A_i B_j$$

où A_i et B_j sont des conjonctions de littéraux.

Formes normales

Considérons la formule $F = a \oplus (\neg b \Rightarrow c)$:

a	b	c	$a \oplus (\neg b \Rightarrow c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Formes normales

Considérons la formule $F = a \oplus (\neg b \Rightarrow c)$:

a	b	c	$a \oplus (\neg b \Rightarrow c)$	
0	0	0	0	
0	0	1	1	◀
0	1	0	1	◀
0	1	1	1	◀
1	0	0	1	◀
1	0	1	0	
1	1	0	0	
1	1	1	0	

$F \equiv \overline{a}\overline{b}c + \overline{a}b\overline{c} + \overline{a}bc + a\overline{b}\overline{c}$ (forme normale disjonctive).

Lorsque chaque variable propositionnelle apparaît exactement une fois dans chaque conjonction de littéraux, l'expression disjonctive est unique à permutation près.

Formes normales

Considérons la formule $F = a \oplus (\neg b \Rightarrow c)$:

a	b	c	$a \oplus (\neg b \Rightarrow c)$	
0	0	0	0	◀
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	1	
1	0	1	0	◀
1	1	0	0	◀
1	1	1	0	◀

$\overline{F} \equiv \overline{a}\overline{b}\overline{c}+a\overline{b}c+ab\overline{c}+abc$ donc $F \equiv (a+b+c)(\overline{a}+b+\overline{c})(\overline{a}+\overline{b}+c)(\overline{a}+\overline{b}+\overline{c})$ (forme normale conjonctive).

Lorsque chaque variable propositionnelle apparaît exactement une fois dans chaque disjonction de littéraux, l'expression conjonctive est unique à permutation près.

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

- ce bloc est équivalent à $\overline{a}c$;

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

- ce bloc est équivalent à $\overline{a}c$;
- ce bloc est équivalent à $\overline{a}b$;

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

- ce bloc est équivalent à $\overline{a}c$;
- ce bloc est équivalent à $\overline{a}b$;
- ce bloc est équivalent à $a\overline{b}\overline{c}$;

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

- ce bloc est équivalent à $\overline{a}c$;
- ce bloc est équivalent à $\overline{a}b$;
- ce bloc est équivalent à $a\overline{b}\overline{c}$;
- donc $F \equiv \overline{a}c + \overline{a}b + a\overline{b}\overline{c}$.

Tableaux de Karnaugh

On considère toujours $F = a \oplus (\neg b \Rightarrow c)$:

a \ bc	00	01	11	10
0	0	1	1	1
1	1	0	0	0

Un bloc rectangulaire de 1, 2, 4, 8, ... valeurs voisines égales à 1 peut être simplifié :

- ce bloc est équivalent à $\overline{a}c$;
- ce bloc est équivalent à $\overline{a}b$;
- ce bloc est équivalent à $a\overline{b}\overline{c}$;
- donc $F \equiv \overline{a}c + \overline{a}b + a\overline{b}\overline{c}$.

De même, $\overline{F} \equiv \overline{a}\overline{b}\overline{c} + ac + ab$, donc $F \equiv (a+b+c)(\overline{a}+\overline{c})(\overline{a}+\overline{b})$.

Tableaux de Karnaugh

Un autre exemple à quatre variables :

ab \ cd	00	01	11	10
00	1	0	1	1
01	0	0	1	1
11	0	0	0	1
10	1	0	0	1

Tableaux de Karnaugh

Un autre exemple à quatre variables :

ab \ cd	00	01	11	10
00	1	0	1	1
01	0	0	1	1
11	0	0	0	1
10	1	0	0	1

$\overline{a}c$

Tableaux de Karnaugh

Un autre exemple à quatre variables :

ab \ cd	00	01	11	10
00	1	0	1	1
01	0	0	1	1
11	0	0	0	1
10	1	0	0	1

$c\overline{d}$

Tableaux de Karnaugh

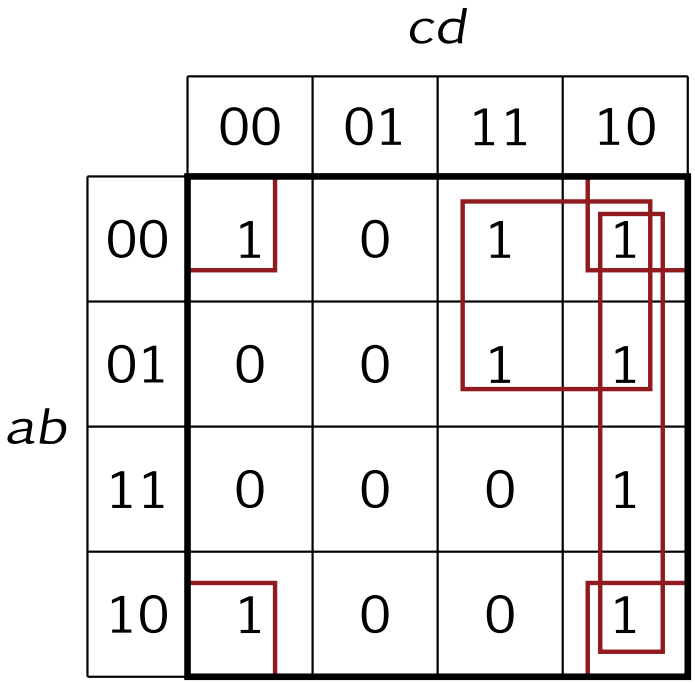
Un autre exemple à quatre variables :

ab \ cd	00	01	11	10
00	1	0	1	1
01	0	0	1	1
11	0	0	0	1
10	1	0	0	1

$\overline{b}\,\overline{d}$

Tableaux de Karnaugh

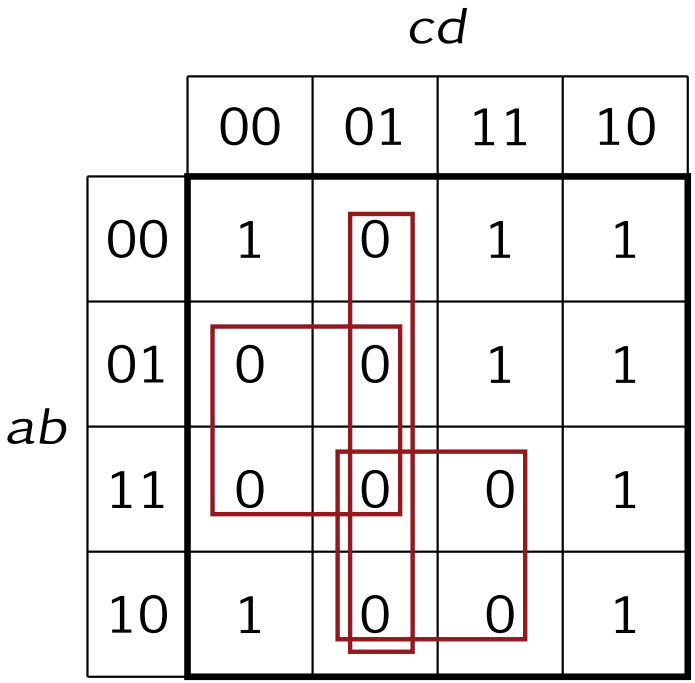
Un autre exemple à quatre variables :



Donc $F \equiv \overline{a}\,c + c\,\overline{d} + \overline{b}\,\overline{d}$.

Tableaux de Karnaugh

Un autre exemple à quatre variables :



Donc $F \equiv \overline{a}c + c\overline{d} + \overline{b}\,\overline{d}$.

De même, $\overline{F} \equiv b\overline{c} + ad + \overline{c}d$, donc $F \equiv (\overline{b} + c)(\overline{a} + \overline{d})(c + \overline{d})$.

Représentation d'une formule logique

On définit le type *formule* :

```
type formule = Const of bool
       | Var of char
       | Op_unaire of (bool -> bool) * formule
       | Op_binaire of (bool -> bool -> bool) * formule * formule ;;
```

et les connecteurs usuels :

```
let neg p = not p ;;
let et p q = p && q ;;
let ou p q = p || q ;;
let impl p q = q || (not p) ;;
let equiv p q = (impl p q) && (impl q p) ;;
let xor p q = not (equiv p q) ;;
```

Représentation d'une formule logique

On définit le type *formule* :

```
type formule = Const of bool
       | Var of char
       | Op_unaire of (bool -> bool) * formule
       | Op_binaire of (bool -> bool -> bool) * formule * formule ;;
```

et les connecteurs usuels :

```
let neg p = not p ;;
let et p q = p && q ;;
let ou p q = p || q ;;
let impl p q = q || (not p) ;;
let equiv p q = (impl p q) && (impl q p) ;;
let xor p q = not (equiv p q) ;;
```

On utilise un analyseur lexical pour définir une formule :

```
# let F = analyseur "(a ou b) <=> (non a et c)" ;;
F: formule = Op_binaire
    (<fun>, Op_binaire (<fun>, Var 'a', Var 'b'),
     Op_binaire (<fun>, Op_unaire (<fun>, Var 'a'), Var 'c'))
```

Évaluation d'une formule logique

On utilise la fonction **assoc** dont on rappelle la définition :

```
let rec assoc k = function
  | []                    -> raise Not_found
  | (x, y)::_ when k = x -> y
  | _::q                  -> assoc k q ;;
```

L'évaluation d'une formule pour une liste de valeurs donnée s'écrit alors :

```
let rec evalue f lst = match f with
  | Const c                -> c
  | Var v                  -> assoc v lst
  | Op_unaire (u, f1)      -> u (evalue f1 lst)
  | Op_binaire (b, f1, f2) -> b (evalue f1 lst) (evalue f2 lst) ;;
```

Par exemple :

```
# let F = analyseur "(a ou b) <=> (non a et c)"
  in evalue F [('a',true) ; ('b',true) ; ('c',false)] ;;
- : bool = false
```

Tautologies

Nous avons besoin d'une fonction qui extrait la liste des variables d'une formule :

```
let liste_des_variables f =
  let rec cherche_var lst = function
    | Const c -> lst
    | Var p when (mem p lst) -> lst
    | Var p -> p::lst
    | Op_unaire (u,f1) -> cherche_var lst f1
    | Op_binaire (b,f1,f2) -> cherche_var (cherche_var lst f1) f2
  in cherche_var [] f ;;
```

et d'une fonction qui remplace toute les occurrences d'une variable **p** par une constante booléenne **a** :

```
let rec subs f p a = match f with
  | Const c                -> Const c
  | Var q when q = p       -> Const a
  | Var q                  -> Var q
  | Op_unaire (u,f1)       -> Op_unaire (u,(subs f1 p a))
  | Op_binaire (b,f1,f2) ->
                      Op_binaire (b,(subs f1 p a),(subs f2 p a)) ;;
```

Tautologies

La fonction de vérification automatique des tautologies peut maintenant être écrite :

```
exception Echec ;;

let est_une_tautologie f =
  let rec teste f = function
    | []   -> if not (evalue f []) then raise Echec
    | t::q -> let g = subs f t true in teste g q ;
              let g = subs f t false in teste g q
  in let lst = liste_des_variables f
  in try teste f lst ; true
     with Echec -> false ;;
```

Tautologies

La fonction de vérification automatique des tautologies peut maintenant être écrite :

```
exception Echec ;;

let est_une_tautologie f =
  let rec teste f = function
    | []   -> if not (evalue f []) then raise Echec
    | t::q -> let g = subs f t true in teste g q ;
              let g = subs f t false in teste g q
  in let lst = liste_des_variables f
  in try teste f lst ; true
     with Echec -> false ;;
```

À titre d'exemple, vérifions la validité du raisonnement par contraposée :

```
# est_une_tautologie (analyseur "(a => b) <=> (non b => non a)") ;;
- : bool = true
# est_une_tautologie (analyseur "(a => b) <=> (non a => non b)") ;;
- : bool = false
```

Satisfiabilité

La satisfiabilité se traite de manière analogue :

```
let affiche lst =
  let affiche_valeur = function
    | (c,true)  -> print_char c ; print_string " = vrai  "
    | (c,false) -> print_char c ; print_string " = faux  "
  in do_list affiche_valeur lst ;
  print_newline () ;;

let satisfiabilite f =
  let rec teste f s = function
    | []   -> if evalue f s then affiche s
    | t::q -> teste f ((t,false)::s) q ;
              teste f ((t,true)::s) q
  in let lst = liste_des_variables f
  in teste f [] lst ;;
```

Satisfiabilité

La satisfiabilité se traite de manière analogue :

```
let affiche lst =
  let affiche_valeur = function
    | (c,true)  -> print_char c ; print_string " = vrai  "
    | (c,false) -> print_char c ; print_string " = faux  "
  in do_list affiche_valeur lst ;
  print_newline () ;;

let satisfiabilite f =
  let rec teste f s = function
    | []   -> if evalue f s then affiche s
    | t::q -> teste f ((t,false)::s) q ;
              teste f ((t,true)::s) q
  in let lst = liste_des_variables f
  in teste f [] lst ;;
```

À quelle condition $a \Rightarrow b$ est-il équivalent à $\neg a \Rightarrow \neg b$?

```
# satisfiabilite (analyseur "(a => b) <=> (non a => non b)") ;;
a = faux  b = faux
a = vrai  b = vrai
- : unit = ()
```

Un exemple de problème de logique

Vous êtes perdus dans le désert et vous suivez une piste depuis de longues heures quand vous débouchez soudain sur une bifurcation. Vous savez que les deux pistes qui s'ouvrent à vous peuvent éventuellement conduire à une oasis, mais aussi vous perdre à tout jamais. Chacune d'elles est gardée par un sphinx qui s'anime à votre arrivée et commence à parler :
Le premier vous dit : « une au moins des deux pistes conduit à une oasis. »
Le second ajoute : « la piste de droite se perd dans le désert. »
Sachant que les deux sphinx disent tous deux la vérité, ou bien mentent tous deux, que faites vous ?

Un exemple de problème de logique

Vous êtes perdus dans le désert et vous suivez une piste depuis de longues heures quand vous débouchez soudain sur une bifurcation. Vous savez que les deux pistes qui s'ouvrent à vous peuvent éventuellement conduire à une oasis, mais aussi vous perdre à tout jamais. Chacune d'elles est gardée par un sphinx qui s'anime à votre arrivée et commence à parler :
Le premier vous dit : « une au moins des deux pistes conduit à une oasis. »
Le second ajoute : « la piste de droite se perd dans le désert. »
Sachant que les deux sphinx disent tous deux la vérité, ou bien mentent tous deux, que faites vous ?

Assertion a : « la piste de droite conduit à une oasis ».

Assertion b : « la piste de gauche conduit à une oasis ».

Le premier sphinx affirme « $a \vee b$ » et le deuxième « $\neg a$ ».

Un exemple de problème de logique

Vous êtes perdus dans le désert et vous suivez une piste depuis de longues heures quand vous débouchez soudain sur une bifurcation. Vous savez que les deux pistes qui s'ouvrent à vous peuvent éventuellement conduire à une oasis, mais aussi vous perdre à tout jamais. Chacune d'elles est gardée par un sphinx qui s'anime à votre arrivée et commence à parler :
Le premier vous dit : « une au moins des deux pistes conduit à une oasis. »
Le second ajoute : « la piste de droite se perd dans le désert. »
Sachant que les deux sphinx disent tous deux la vérité, ou bien mentent tous deux, que faites vous ?

Assertion a : « la piste de droite conduit à une oasis ».

Assertion b : « la piste de gauche conduit à une oasis ».

Le premier sphinx affirme « $a \vee b$ » et le deuxième « $\neg a$ ».

Il nous faut satisfaire la formule $F = \Big((a \vee b) \wedge \neg a\Big) \vee \Big(\neg(a \vee b) \wedge a\Big)$.

```
# let F = analyseur "((a ou b) et non a) ou (non (a ou b) et a)"
  in satisfiabilite F ;;
a = faux  b = vrai
- : unit = ()
```

Une seule possibilité : la piste de droite se perd dans le désert alors que celle de gauche conduit à un oasis.

Le problème n-SAT

On cherche à résoudre le problème de la satisfiabilité en se restreignant aux formules mises sous forme normale conjonctive.

Le problème n-SAT

On cherche à résoudre le problème de la satisfiabilité en se restreignant aux formules mises sous forme normale conjonctive.

On appelle clause d'ordre n toute disjonction d'au plus n littéraux et forme normale conjonctive d'ordre n toute conjonction de clauses d'ordre n.

Déterminer si une formule sous forme normale conjonctive d'ordre n est satisfiable est appelé problème n-SAT.

Le problème n-SAT

On cherche à résoudre le problème de la satisfiabilité en se restreignant aux formules mises sous forme normale conjonctive.

On appelle clause d'ordre n toute disjonction d'au plus n littéraux et forme normale conjonctive d'ordre n toute conjonction de clauses d'ordre n.

Déterminer si une formule sous forme normale conjonctive d'ordre n est satisfiable est appelé problème n-SAT.

Par exemple, une forme normale conjonctive d'ordre 2 est de la forme :

$$(a_1 + a_2)(a_3 + a_4)\cdots(a_{2n-1} + a_{2n})$$

avec $a_i \in \{x_1, \overline{x}_1, \ldots, x_p, \overline{x}_p\}$, les x_i étant des variables propositionnelles.

Le problème 2-SAT

La clause $(a \vee b)$ est équivalente à la formule $(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$.

a	b	$\neg b \Rightarrow a$	$\neg a \Rightarrow b$	$(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	1	1

Le problème 2-SAT

La clause $(a \vee b)$ est équivalente à la formule $(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$.

On construit le graphe orienté $G = (V, E)$ en suivant les règles :

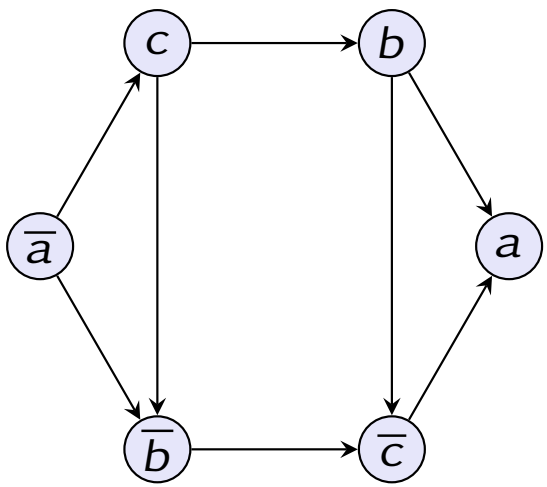
- les sommets V sont les différentes variables propositionnelles a_i présentes dans la formule ainsi que leurs négations $\overline{a}_i$;
- à chaque clause $(a_i + a_j)$ sont associés les arcs $(\overline{a}_i, a_j)$ et $(\overline{a}_j, a_i)$.

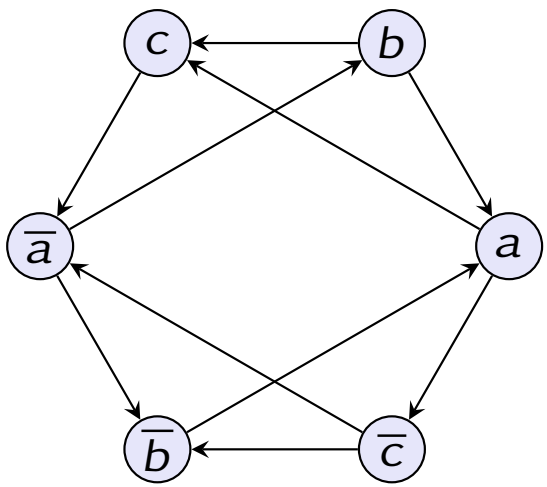
Le problème 2-SAT

La clause $(a \vee b)$ est équivalente à la formule $(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$.

On construit le graphe orienté $G = (V, E)$ en suivant les règles :

- les sommets V sont les différentes variables propositionnelles a_i présentes dans la formule ainsi que leurs négations $\overline{a}_i$;
- à chaque clause $(a_i + a_j)$ sont associés les arcs $(\overline{a}_i, a_j)$ et $(\overline{a}_j, a_i)$.





$f_1 = (a + \overline{b})(a + c)(\overline{b} + \overline{c})(b + \overline{c})$

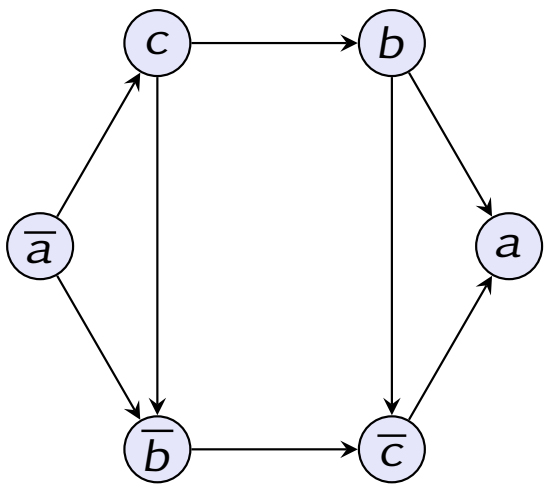
$f_2 = (a + b)(\overline{a} + c)(a + \overline{b})(\overline{b} + c)(\overline{a} + \overline{c})$

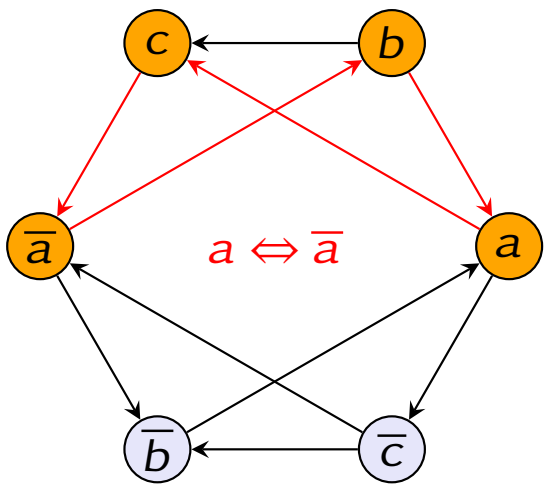
Le problème 2-SAT

La clause $(a \vee b)$ est équivalente à la formule $(\neg a \Rightarrow b) \wedge (\neg b \Rightarrow a)$.

Si la formule f est satisfiable, alors pour toute variable v les sommets v et $\overline{v}$ n'appartiennent pas à la même composante fortement connexe.

Pour que f soit satisfiable il faut que la distribution de vérité soit constante sur chaque composante fortement connexe.





$f_1 = (a + \overline{b})(a + c)(\overline{b} + \overline{c})(b + \overline{c})$

$f_2 = (a + b)(\overline{a} + c)(a + \overline{b})(\overline{b} + c)(\overline{a} + \overline{c})$

L'algorithme de Aspvall, Plass et Tarjan

On suppose que pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H :

- les sommets de H sont les composantes fortement connexes de G ;
- on relie par un arc la composante X à la composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

L'algorithme de Aspvall, Plass et Tarjan

On suppose que pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H :

- les sommets de H sont les composantes fortement connexes de G ;
- on relie par un arc la composante X à la composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

H est un graphe sans circuit.

L'algorithme de Aspvall, Plass et Tarjan

On suppose que pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H :

- les sommets de H sont les composantes fortement connexes de G ;
- on relie par un arc la composante X à la composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

H est un graphe sans circuit.

Soient X et Y deux composantes fortement connexes distinctes.
S'il existe un chemin dans H reliant X à Y, il existe un chemin dans G reliant un sommet $s_1 \in X$ à un sommet $s_2 \in Y$.
S'il existe un chemin dans H reliant Y à X, il existe un chemin dans G reliant un sommet $s_3 \in Y$ à un sommet $s_4 \in X$.

L'algorithme de Aspvall, Plass et Tarjan

On suppose que pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H :

- les sommets de H sont les composantes fortement connexes de G ;
- on relie par un arc la composante X à la composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

H est un graphe sans circuit.

Soient X et Y deux composantes fortement connexes distinctes.
S'il existe un chemin dans H reliant X à Y, il existe un chemin dans G reliant un sommet $s_1 \in X$ à un sommet $s_2 \in Y$.
S'il existe un chemin dans H reliant Y à X, il existe un chemin dans G reliant un sommet $s_3 \in Y$ à un sommet $s_4 \in X$.
s_1 et s_4 appartiennent à X donc il existe un chemin dans G menant de s_4 à s_1.
s_2 et s_3 appartiennent à Y donc il existe un chemin dans G menant de s_2 à s_3.

L'algorithme de Aspvall, Plass et Tarjan

On suppose que pour tout sommet v, le sommet $\overline{v}$ n'appartient pas à la même composante connexe que v. Pour cela, on construit le graphe orienté H :

- les sommets de H sont les composantes fortement connexes de G ;
- on relie par un arc la composante X à la composante Y dans H s'il existe un arc allant d'un sommet de X à un sommet de Y dans G.

H est un graphe sans circuit.

Soient X et Y deux composantes fortement connexes distinctes.
S'il existe un chemin dans H reliant X à Y, il existe un chemin dans G reliant un sommet $s_1 \in X$ à un sommet $s_2 \in Y$.
S'il existe un chemin dans H reliant Y à X, il existe un chemin dans G reliant un sommet $s_3 \in Y$ à un sommet $s_4 \in X$.
s_1 et s_4 appartiennent à X donc il existe un chemin dans G menant de s_4 à s_1.
s_2 et s_3 appartiennent à Y donc il existe un chemin dans G menant de s_2 à s_3.
Ceci montre l'existence d'un cycle $s_1 \to s_2 \to s_3 \to s_4 \to s_1$ qui prouve que ces quatre sommets doivent appartenir à la même composante connexe de G, ce qui est absurde.

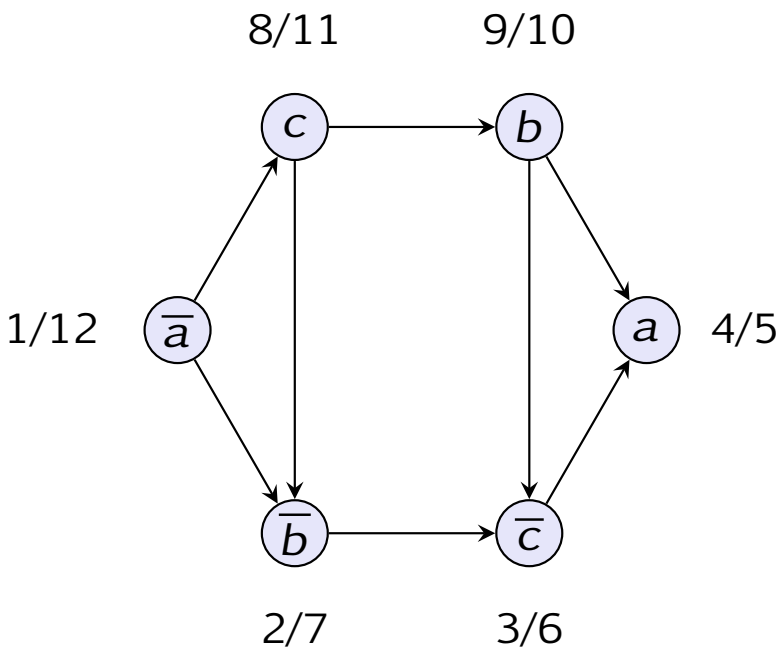
L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

L'algorithme de Aspvall, Plass et Tarjan

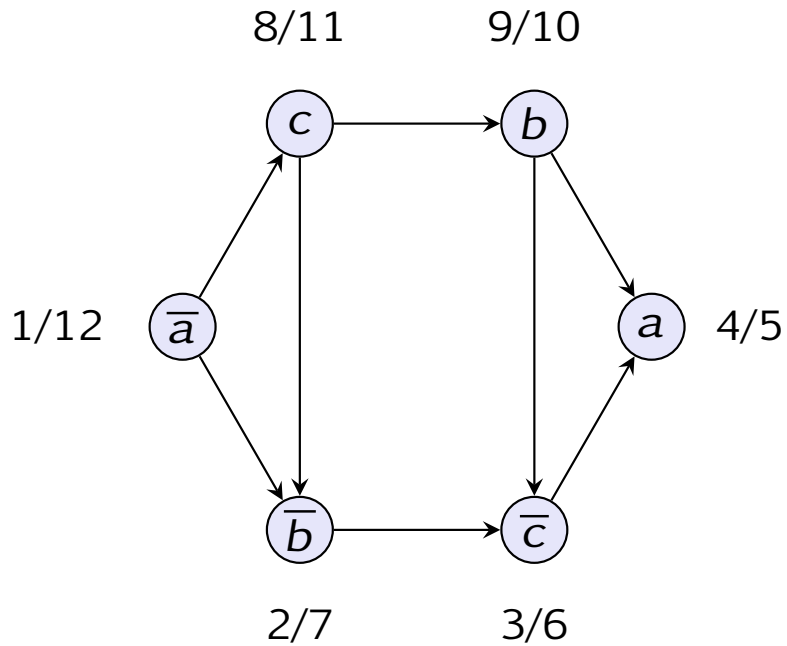
L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

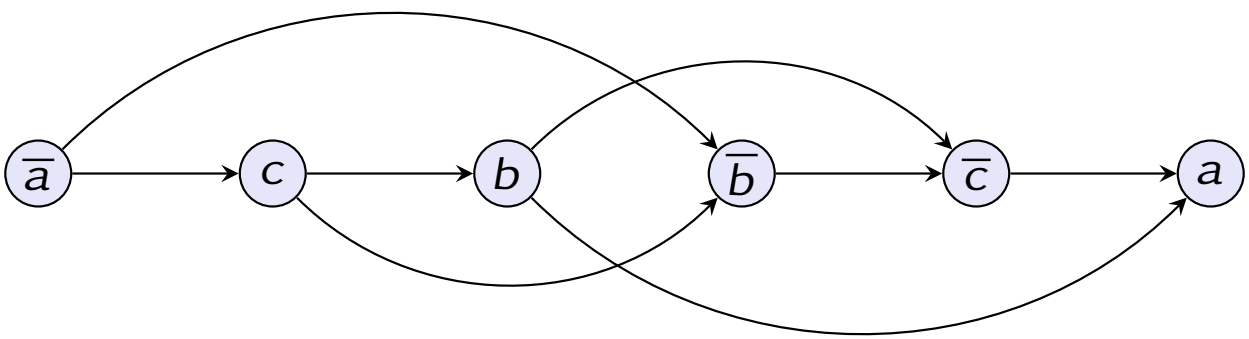
```
procedure TRI_TOPOLOGIQUE(graphe : H)
    ordre ← [ ]
    déjàVus ← ∅
    while |ordre| < |H| do
        H \ ordre → s0
        DFS(s0)
    return ordre

procedure DFS(sommet : s0)
    àTraiter ← s0
    déjàVus ← s0
    while àTraiter ≠ ∅ do
        àTraiter → s
        for t ∈ voisins(s) do
            if t ∉ déjàVus then
                DFS(t)
        ordre ← s :: ordre
```



L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.





L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

Si $X \leqslant Y$ alors X est situé avant Y dans la liste chaînée.

L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

Si $X \leqslant Y$ alors X est situé avant Y dans la liste chaînée.

Considérons le moment où on explore l'arc $X \to Y$.

- Si Y n'a pas encore été vu il rentrera dans la liste chaînée avant X et sera donc situé après X quand X y rentrera à son tour.

L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

Si $X \leqslant Y$ alors X est situé avant Y dans la liste chaînée.

Considérons le moment où on explore l'arc $X \to Y$.

- Si Y n'a pas encore été vu il rentrera dans la liste chaînée avant X et sera donc situé après X quand X y rentrera à son tour.
- Si Y a déjà été vu, soit il est déjà dans la liste chaînée (et le problème est réglé), soit il est encore dans la pile.

L'algorithme de Aspvall, Plass et Tarjan

L'absence de circuit permet d'établir sur l'ensemble des sommets de H un ordre partiel : $X \leqslant Y \iff (X = Y$ ou il existe un arc reliant X à Y dans $H)$.
L'algorithme de tri topologique prolonge cet ordre en un ordre total.

Si $X \leqslant Y$ alors X est situé avant Y dans la liste chaînée.

Considérons le moment où on explore l'arc $X \to Y$.

- Si Y n'a pas encore été vu il rentrera dans la liste chaînée avant X et sera donc situé après X quand X y rentrera à son tour.
- Si Y a déjà été vu, soit il est déjà dans la liste chaînée (et le problème est réglé), soit il est encore dans la pile.

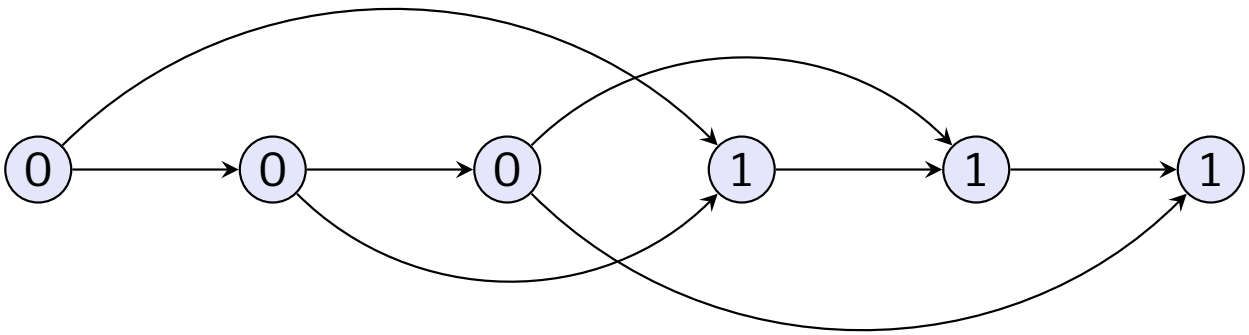
Dans ce dernier cas X est un descendant de Y avec pour conséquence l'existence d'un cycle dans H, ce qui est exclus.

L'algorithme de Aspvall, Plass et Tarjan

On définit la distribution de vérité sur f de la manière suivante :

si $s \in X$ et $\overline{s} \in Y$ on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Dans le cas de la formule f_1 cela revient à poser $a = 1$, $b = 0$, $c = 0$.

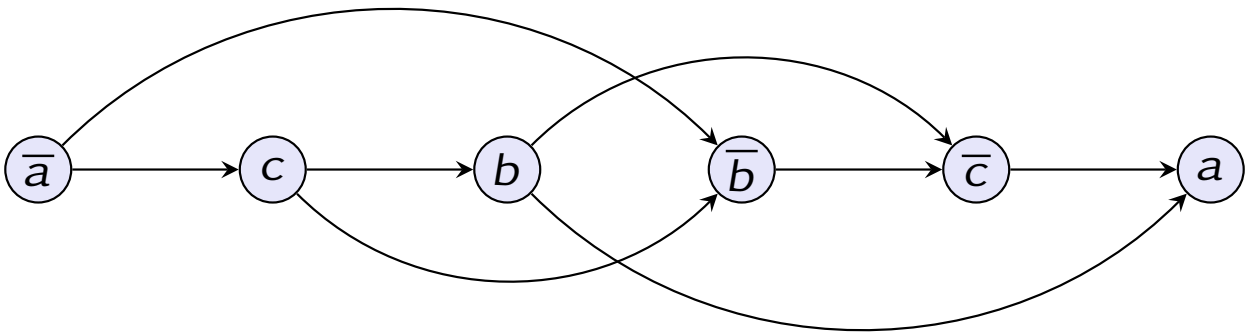


L'algorithme de Aspvall, Plass et Tarjan

On définit la distribution de vérité sur f de la manière suivante :

si $s \in X$ et $\overline{s} \in Y$ on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Ainsi définie, cette distribution de vérité satisfait f.

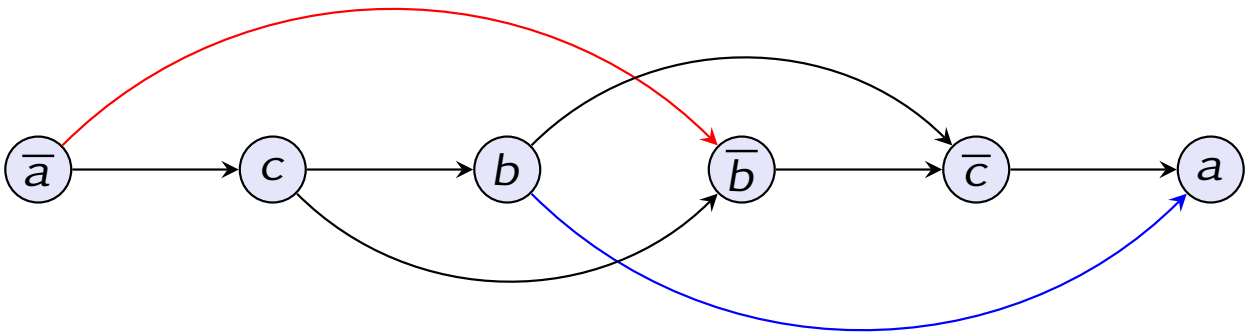


L'algorithme de Aspvall, Plass et Tarjan

On définit la distribution de vérité sur f de la manière suivante :

si $s \in X$ et $\overline{s} \in Y$ on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Ainsi définie, cette distribution de vérité satisfait f.



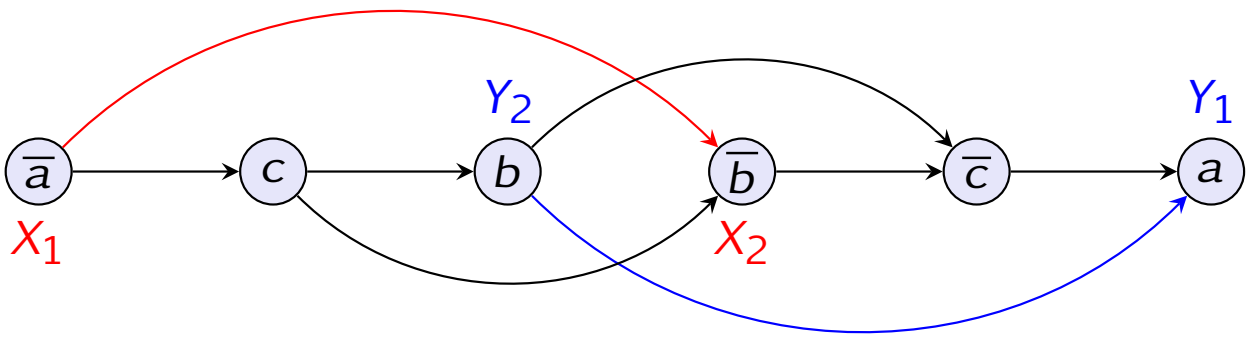
Soit s_1 et s_2 dans G reliés par un arc $s_1 \to s_2$. Il existe aussi un arc $\overline{s}_2 \to \overline{s}_1$.
Il faut montrer que pour la distribution de vérité l'implication $s_1 \Rightarrow s_2$ est vraie.

L'algorithme de Aspvall, Plass et Tarjan

On définit la distribution de vérité sur f de la manière suivante :

si $s \in X$ et $\overline{s} \in Y$ on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Ainsi définie, cette distribution de vérité satisfait f.



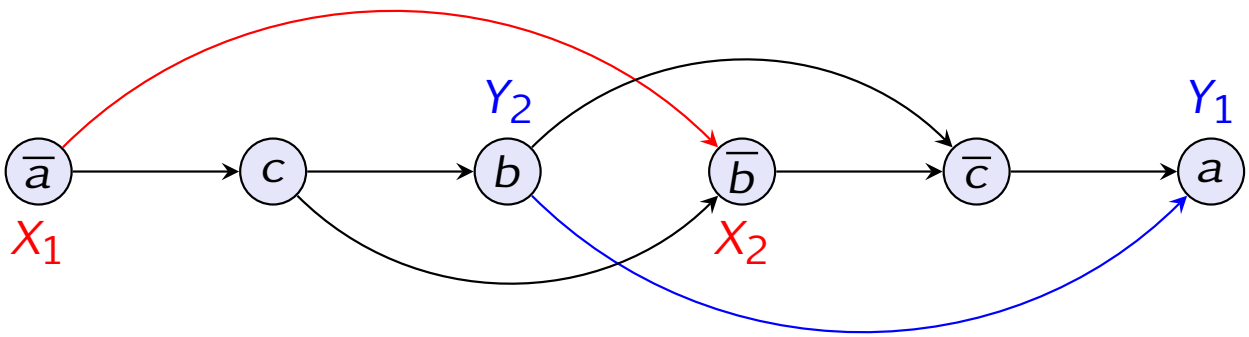
Soit s_1 et s_2 dans G reliés par un arc $s_1 \to s_2$. Il existe aussi un arc $\overline{s}_2 \to \overline{s}_1$.
Il faut montrer que pour la distribution de vérité l'implication $s_1 \Rightarrow s_2$ est vraie.
Notons X_1 la composante fortement connexe de s_1 et X_2 celle de s_2. On a $X_1 \leqslant X_2$.
Notons Y_1 la composante fortement connexe de $\overline{s}_1$ et Y_2 celle de $\overline{s}_2$. On a $Y_2 \leqslant Y_1$.

L'algorithme de Aspvall, Plass et Tarjan

On définit la distribution de vérité sur f de la manière suivante :

si $s \in X$ et $\overline{s} \in Y$ on pose $s = 0$ si $X < Y$ et $s = 1$ si $Y < X$.

Ainsi définie, cette distribution de vérité satisfait f.



Soit s_1 et s_2 dans G reliés par un arc $s_1 \to s_2$. Il existe aussi un arc $\overline{s}_2 \to \overline{s}_1$.
Il faut montrer que pour la distribution de vérité l'implication $s_1 \Rightarrow s_2$ est vraie.
Notons X_1 la composante fortement connexe de s_1 et X_2 celle de s_2. On a $X_1 \leqslant X_2$.
Notons Y_1 la composante fortement connexe de $\overline{s}_1$ et Y_2 celle de $\overline{s}_2$. On a $Y_2 \leqslant Y_1$.
Si on avait $s_1 = 1$ et $s_2 = 0$ on aurait $Y_1 < X_1$ et $X_2 < Y_2$, ce qui n'est pas possible.

L'algorithme de Aspvall, Plass et Tarjan

Complexité du problème 2-SAT

- On calculer les composantes fortement connexes de G pour un coût en $\Theta(|V|+|E|)$ (algorithme de Tarjan).

  Si n désigne le nombre de variables de la formule à satisfaire nous avons $|V| \leqslant 2n$ et $|E| \leqslant |V|(|V|-1)$ donc le calcul des composantes fortement connexes de G est un $O(n^2)$.

L'algorithme de Aspvall, Plass et Tarjan

Complexité du problème 2-SAT

- On calculer les composantes fortement connexes de G pour un coût en $\Theta(|V|+|E|)$ (algorithme de Tarjan).

  Si n désigne le nombre de variables de la formule à satisfaire nous avons $|V| \leqslant 2n$ et $|E| \leqslant |V|(|V|-1)$ donc le calcul des composantes fortement connexes de G est un $O(n^2)$.
- Le tri topologique de H est un parcours en profondeur donc son coût est un $O(|V|+|E|) = O(n^2)$.

L'algorithme de Aspvall, Plass et Tarjan

Complexité du problème 2-SAT

- On calculer les composantes fortement connexes de G pour un coût en $\Theta(|V|+|E|)$ (algorithme de Tarjan).

  Si n désigne le nombre de variables de la formule à satisfaire nous avons $|V| \leqslant 2n$ et $|E| \leqslant |V|(|V|-1)$ donc le calcul des composantes fortement connexes de G est un $O(n^2)$.
- Le tri topologique de H est un parcours en profondeur donc son coût est un $O(|V|+|E|) = O(n^2)$.
- L'attribution des valeurs de vérité a un coût en $O(n^2)$.

L'algorithme de Aspvall, Plass et Tarjan

Complexité du problème 2-SAT

- On calculer les composantes fortement connexes de G pour un coût en $\Theta(|V|+|E|)$ (algorithme de Tarjan).

  Si n désigne le nombre de variables de la formule à satisfaire nous avons $|V| \leqslant 2n$ et $|E| \leqslant |V|(|V|-1)$ donc le calcul des composantes fortement connexes de G est un $O(n^2)$.
- Le tri topologique de H est un parcours en profondeur donc son coût est un $O(|V|+|E|) = O(n^2)$.
- L'attribution des valeurs de vérité a un coût en $O(n^2)$.

Bilan : il existe une solution de coût polynomial au problème 2-SAT.

Le problème 3-SAT

Tous les problèmes n-SAT pour $n \geqslant 3$ découlent du problème 3-SAT : toute clause $(x_1 + x_2 + \cdots + x_n)$ avec $n \geqslant 3$ est équivalent à :

$$(x_1 + x_2 + y_2)(\overline{y}_2 + x_3 + y_3)(\overline{y}_3 + x_4 + y_4) \cdots (\overline{y}_{n-3} + x_{n-2} + y_{n-2})(\overline{y}_{n-2} + x_{n-1} + x_n)$$

ce qui permet de réduire en temps polynomial un problème n-SAT pour $n \geqslant 3$ à un problème 3-SAT. Cependant, le problème 3-SAT est un problème NP-complet.

Le problème 3-SAT

Tous les problèmes n-SAT pour $n \geqslant 3$ découlent du problème 3-SAT : toute clause $(x_1 + x_2 + \cdots + x_n)$ avec $n \geqslant 3$ est équivalent à :

$$(x_1 + x_2 + y_2)(\overline{y}_2 + x_3 + y_3)(\overline{y}_3 + x_4 + y_4) \cdots (\overline{y}_{n-3} + x_{n-2} + y_{n-2})(\overline{y}_{n-2} + x_{n-1} + x_n)$$

ce qui permet de réduire en temps polynomial un problème n-SAT pour $n \geqslant 3$ à un problème 3-SAT. Cependant, le problème 3-SAT est un problème NP-complet.

On classe les problèmes de décision en plusieurs familles :

- la classe P constituée des problèmes de décision qui peuvent être résolus en temps polynomial par rapport à la taille de l'entrée ;
- la classe NP constituée des problèmes de décision pour lesquels on peut vérifier la validité d'une solution en temps polynomial.

Exemple : le problème 2-SAT appartient à la classe P, et les problèmes n-SAT appartiennent à la classe NP.

Le problème 3-SAT

Tous les problèmes n-SAT pour $n \geqslant 3$ découlent du problème 3-SAT : toute clause $(x_1 + x_2 + \cdots + x_n)$ avec $n \geqslant 3$ est équivalent à :

$$(x_1 + x_2 + y_2)(\overline{y}_2 + x_3 + y_3)(\overline{y}_3 + x_4 + y_4) \cdots (\overline{y}_{n-3} + x_{n-2} + y_{n-2})(\overline{y}_{n-2} + x_{n-1} + x_n)$$

ce qui permet de réduire en temps polynomial un problème n-SAT pour $n \geqslant 3$ à un problème 3-SAT. Cependant, le problème 3-SAT est un problème NP-complet.

On classe les problèmes de décision en plusieurs familles :

- la classe P constituée des problèmes de décision qui peuvent être résolus en temps polynomial par rapport à la taille de l'entrée ;
- la classe NP constituée des problèmes de décision pour lesquels on peut vérifier la validité d'une solution en temps polynomial.

On dispose de l'inclusion $P \subset \text{NP}$; Le problème de l'inclusion réciproque est un des plus grands défis de l'informatique théorique.

Le problème 3-SAT

Tous les problèmes n-SAT pour $n \geqslant 3$ découlent du problème 3-SAT : toute clause $(x_1 + x_2 + \cdots + x_n)$ avec $n \geqslant 3$ est équivalent à :

$$(x_1 + x_2 + y_2)(\overline{y}_2 + x_3 + y_3)(\overline{y}_3 + x_4 + y_4) \cdots (\overline{y}_{n-3} + x_{n-2} + y_{n-2})(\overline{y}_{n-2} + x_{n-1} + x_n)$$

ce qui permet de réduire en temps polynomial un problème n-SAT pour $n \geqslant 3$ à un problème 3-SAT. Cependant, le problème 3-SAT est un problème NP-complet.

On classe les problèmes de décision en plusieurs familles :

- la classe P constituée des problèmes de décision qui peuvent être résolus en temps polynomial par rapport à la taille de l'entrée ;
- la classe NP constituée des problèmes de décision pour lesquels on peut vérifier la validité d'une solution en temps polynomial.

On dispose de l'inclusion $P \subset \text{NP}$; Le problème de l'inclusion réciproque est un des plus grands défis de l'informatique théorique.

Problèmes NP-complets : si on connaissait une solution polynomiale pour l'un d'eux, on en connaîtrait une pour tout problème de la classe, et on aurait $P = \text{NP}$.

Le problème SAT a été le premier problème NP-complet trouvé (théorème de Cook).