Cours 9 : Algèbre de Boole

Les valeurs Booléennes qui représentent le vrai et le faux sont True et False. Les opérations logiques utilisées: le ET `&` , OU `|` et NON, `~`

9.1 Représentation d'une fonction Booléenne

Représentons $f = (x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$

In [1]:
```
from sympy import *
init_printing()
x,y,z=symbols('x,y,z')
f=(x&y)|(y&z)|(z&x)
f
```

Out[1]: $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$

Une fois f définie, on peut chercher quelles sont les variables de f par la méthode `free_symbols` :

In [7]:
```
f.free_symbols
```

Out[7]: $\{x, y, z\}$

On rappelle que la *valuation* v d'une fonction Booléenne f est une distribution des valeurs de vérité aux variables de cette fonction. Cette valuation *satisfait* la fonction f si, avec la valuation v des variables de f, f est évaluée à vrai.

On peut chercher une valuation qui satisfait la formule. Il suffit de substituer à une variable le vrai ou le faux: la première valuation ne satisfait pas la formule mais la seconde oui.

In [5]:
```
f.subs({x:True,y:False,z:False})
```

Out[5]: False

In [2]:
```
f.subs({x:True,y:True,z:False})
```

Out[2]: True

On peut aussi mélanger valuation et variables:

In [45]:
```
f.subs({y:False})
```

Out[45]: $x \wedge z$

La fonction peut aussi être définie comme une Lambda-expression `Sympy` :

In [9]:
```
g=Lambda([x,y,z],(x&y)|(y&z)|(z&x))
```

In [10]:
```
g(True,True,False)
```

Out[10]: True

Avec cette définition, les variables de `g` ne sont plus libres (elles sont liées par `Lambda`)

In [11]:
```
g.free_symbols
```

Out[11]: {}

9.2 Construire la table de vérité

La seconde représentation est plus facile si on veut calculer la valeur de vérité de la fonction pour un triplet donné, voire pour l'ensemble des triplets pour avoir la table de vérité.

In [18]:
```
for i in cartes([False,True],repeat=3):
    a,b,c=i
    print(i,'\t',g(a,b,c))
```

```
(False, False, False)    False
(False, False, True)     False
(False, True, False)     False
(False, True, True)      True
(True, False, False)     False
(True, False, True)      True
(True, True, False)      True
(True, True, True)       True
```

la fonction `cartes` de `sympy` nous permet de calculer directement le produit Cartésien qui nous engendre l'ensemble des valuaitons possibles des variables de la fonction

On peut construire une fonction `BooleanTable` qui prend en entrée une fonction Booléenne et qui retourne sa table de vérité:

In [35]:
```
def BooleanTable(chaine_expr):
    expr=sympify(chaine_expr)
    variables=expr.free_symbols
    for valuation in cartes([False,True],repeat=len(variables)):
        valeur=dict(zip(variables,valuation))
        print(tuple(valeur.items()),expr.subs(valeur))
```

In [37]:
```
BooleanTable('a&b')
```

```
((b, False), (a, False)) False
((b, False), (a, True)) False
((b, True), (a, False)) False
((b, True), (a, True)) True
```

In [40]:
```
BooleanTable('~(x|(y>>x))&y')
```

```
((y, False), (x, False)) False
((y, False), (x, True)) False
((y, True), (x, False)) True
((y, True), (x, True)) False
```

Quelle était la fonction Booléenne dont on a cherché la table de vérité?

In [41]:
```
exp=sympify('~(x|(y>>x))&y')
exp
```

Out[41]: $y \wedge \neg(x \vee (y \Rightarrow x))$

source pour BooleanTable (https://stackoverflow.com/questions/12462747/truth-tables-in-python-using-sympy)

9.3 Vérifier les identités remarquables

Au moyen de la fonction `Equivalent(A,B)` qui est vraie si et seulement si `A` et `B` sont soit tous deux vrais soit tous deux faux.

Vérifions par exemple la distributivité du $\vee$ par rapport au $\wedge$: $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$

In [56]:
```
p,q,r=symbols('p,q,r')
mg=p|(q&r)
md=(p|q)&(p|r)
Equivalent(mg,md)
```

Out[56]: $((p \vee q) \wedge (p \vee r)) \equiv (p \vee (q \wedge r))$

Mais `Equivalent` ne nous répond pas ! L'algorithme ne sait pas résoudre l'équivalence Booléenne d'expressions distinctes et dans ce cas `Equivalent` nous renvoie une expression Booléenne. Une manière de s'en sortir est d'utiliser `equals` qui va utiliser la méthode `satisfiable` sur `Equivalent` (on va revenir bientôt sur `satisfiable`)

In [57]:
```
mg.equals(md)
```

Out[57]: `True`

Que se passe-t-il lorsqu'on n'a pas une tautologie? Par exemple, pour $y \leftrightarrow x \vee (x \wedge y)$, on obtient une erreur

In [3]:
```
x,y=symbols('x,y')
mg=y
md=(x|(x&y))
#mg.equals(md)
```

9.4 Satisfiabilité

Si une formule n'est pas une tautologie, cela signifie qu'il est possible de trouver une (ou plusieurs) valuations (ou modèles) qui rendent fausse cette fonction. Commençons par chercher s'il existe une valuation qui rend vraie la fonction par `satisfiable` :

In [4]:
```
satisfiable(Equivalent(mg,md))
```

Out[4]: `{y: False, x: False}`

`satisfiable` nous retourne une valuation qui permet d'évaluer la fonction à vrai. Mais il y a peut-être d'autres valuations qui rendent vraie la fonction. Cherchons les en demandant à `satisfiable` de nous fournir tous les modèles par un itérable:

In [75]:
```
satisfiable(Equivalent(mg,md),all_models=True)
```

Out[75]: `<generator object _all_models at 0x107601990>`

In [76]:
```
valuation=satisfiable(Equivalent(mg,md),all_models=True)
```

In [77]:
```
next(valuation)
```

Out[77]:
```
{y: False, x: False}
```

In [78]:
```
next(valuation)
```

Out[78]:
```
{x: True, y: True}
```

In [79]:
```
next(valuation)
```

```
---------------------------------------------------------------------
---------
StopIteration                             Traceback (most recent c
all last)
<ipython-input-79-9b1d9c612f72> in <module>()
----> 1 next(valuation)

StopIteration:
```

On a donc 2 valuations qui rendent vraie la fonction Booléenne, lorsque x et y ont la même valeur de vérité (et donc 2 autres qui rendent fausse la fonction, lorsque x et y ont des valeurs de vérité différentes.

Si on avait voulu chercher les valeurs de vérité qui rendent fausse la formule, on aurait cherché:

In [6]:
```
val=satisfiable(~Equivalent(mg,md),all_models=True)
```

In [7]:
```
next(val)
```

Out[7]:
```
{x: True, y: False}
```

In [8]:
```
next(val)
```

Out[8]:
```
{y: True, x: False}
```

9.5 Mise sous forme normale

Toute fonction Booléenne peut être mise sous forme normale

- d'une conjonction de disjonctions (FN conjonctive)
- d'une disjonction de conjonctions (FN disjonctive)

Définissons une fonction f

In [12]:
```
f=(x|y)&~(z>>x)
```

Pour trouver la FN conjonctive (resp. disjonctive) de f, on fait appel à la fonction `to_cnf` (resp. `to_dnf`)

In [13]:
```
to_cnf(f)
```

Out[13]: $z \wedge \neg x \wedge (x \vee y)$

In [14]:
```
to_dnf(f)
```

Out[14]: $(x \wedge z \wedge \neg x) \vee (y \wedge z \wedge \neg x)$

Pour simplifier la fonction, on utilise `simplify` :

In [15]:
```
simplify(f)
```

Out[15]: $y \wedge z \wedge \neg x$

9.6 Un problème inverse

Comment faire si on connait la table de vérité et qu'on veut en déduire une fonction Booléenne? Si les algorithmes sont un peu compliqués, ils sont implémentés dans `Sympy` .

Prenons une table de vérité:

x	y	z	f
0	0	0	$*$
0	0	1	$*$
0	1	0	1

On définit d'abord les valuations qui rendent vraies la table de vérité (on les appelle des *mintermes*)

In [21]:
```
x,y,z=symbols('x,y,z')
minterms=[]
dontcares=[]
```

In [22]:
```
minterms=[[0,1,0]]
```

Ensuite, les valuations pour lesquelles les valeurs de vérité ne sont pas précisées (elles sont vraies ou fausses), on les appelle des *dontcares*

In [24]:
```
dontcares=[[0,0,0],[0,0,1]]
```

Toutes les autres valeur sont supposées à faux.

Une fois la table de vérité décrite, on fait appel à la fonction `SOPform` qui cherche une fonction Booléenne qui a la même valeur de vérité:

In [28]:
```
SOPform([x,y,z],minterms)
```

Out[28]: $y \wedge \neg x \wedge \neg z$

On n'est pas obligé de définir les don't care.

Prenons une table de vérité:

x	y	?
0	0	0
0	1	1
1	0	0
1	1	0

In [31]:
```
min=[[0,1]]
f1=SOPform([x,y],min)
f1
```

Out[31]: $y \wedge \neg x$

vérifions que cette solution correspond à la table de vérité de la fonction $\neg(x \vee (y \rightarrow x)) \wedge y$

In [33]:
```
f2=~(x|(y>>x))&y
f2
```

Out[33]: $y \wedge \neg(x \vee (y \Rightarrow x))$

In [34]:
```
f1.equals(f2)
```

Out[34]: True

Calculons la table de vérité de `f2` avec la fonction `BooleanTable` que nous avons introduite:

In [36]:
```
BooleanTable('~(x|(y>>x))&y')
```

```
((y, False), (x, False)) False
((y, False), (x, True)) False
((y, True), (x, False)) True
((y, True), (x, True)) False
```

Les deux tables de vérité correspondent et nous avons pu retrouver la fonction spécifiée par sa table de vérité. L'agorithme nous a donné la forme la plus simple, que nous retrouvons aussi avec `simplify`

In [37]:
```
simplify(~(x|(y>>x))&y)
```

Out[37]: $y \wedge \neg x$