Recherche de contradictions par la méthode des consensus

1. Présentation

Soient $p_1, \ldots, p_n$ des variables booléennes. On appelle :

- *littéraux* les propositions p_i et leurs complémentaires $\overline{p_i}$;
- *clause* toute disjonction de littéraux, par exemple $p_1 + p_2 + \overline{p_3}$;
- *système de clauses* toute famille finie de clauses.

La clause c est dite *triviale* s'il existe une variable p telle que p et $\overline{p}$ apparaissent dans c. Dans ce cas la valeur de vérité de c est vraie quelles que soient les valeurs de vérité des p_i. La clause *Faux* est la clause constituée d'aucun littéral, sa valeur de vérité est fausse quelles que soient les valeurs des variables p_i. Dans ce TP, on ne considèrera que des clauses non triviales. Ces clauses seront représentées par un couple de deux listes : la liste des variables apparaissant *positivement* dans c (p_1 et p_2 dans l'exemple précédent) et la liste des variables apparaissant *négativement* (p_3 dans l'exemple précédent). Pour une clause non triviale, ces listes sont donc disjointes, et elles sont toutes deux vides pour la clause *Faux* et pour elle seulement.

Un système de clauses est dit *contradictoire* si la conjonction de toutes les clauses du système est égale à la clause *Faux*. Par exemple le système :

$$\{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$$

est contradictoire comme on peut s'en rendre compte en développant à la main le produit :

$$(c + \overline{d})(\overline{c} + \overline{m})(d + \overline{m})m = c\,\overline{c}\,d\,m + c\,\overline{c}\,\overline{m}\,m + c\,\overline{m}\,d\,m + c\,\overline{m}\,\overline{m}\,m + \overline{d}\,\overline{c}\,d\,m + \overline{d}\,\overline{c}\,\overline{m}\,m + \overline{d}\,\overline{m}\,d\,m + \overline{d}\,\overline{m}\,\overline{m}\,m.$$

Cette méthode de développement complet n'est pas praticable lorsque n est grand : la longueur d'une clause non triviale peut atteindre n, et il y a 2^n clauses de longueur n ; le produit de toutes ces clauses comporte donc dans sa forme développée n^{2^n} termes… On peut aussi se rendre compte qu'un système est contradictoire en calculant la table de vérité de la conjonction des clauses (voir le TP `bool`), mais pour n variables booléennes, il y a quand même 2^n valeurs de vérité à calculer. L'objectif de ce TP est d'étudier un autre algorithme de reconnaissance de contradictions, appelé *méthode des consensus*, qui permet de déterminer si un système est contradictoire en général plus efficacement qu'en calculant le produit de toutes les clauses du système ou en cherchant la table de vérité de leur conjonction.

Si c et c' sont deux clauses, on dit que c *implique* c' si tout littéral apparaissant dans c apparaît aussi dans c'. Ceci correspond à la notion usuelle d'implication : $c \Longrightarrow c'$ si et seulement si tout choix des variables rendant c vraie rend aussi c' vraie. Par convention la clause *Faux* implique toutes les autres. Dans un système S, une clause $c \in S$ est dite *minimale* s'il n'existe pas de clause $c' \in S \setminus \{c\}$ telle que $c' \Longrightarrow c$.

Si c et c' sont deux clauses telles qu'il existe une variable p apparaissant positivement dans une clause et négativement dans l'autre, par exemple $c = p + c_1$ et $c' = \overline{p} + c'_1$ (c_1 et c'_1 ne contenant ni p ni $\overline{p}$), on appelle *consensus* de c et c' la clause $c'' = c_1 + c'_1$. c'' a la propriété d'être vraie chaque fois que c et c' le sont, et c'est la plus petite clause pour l'ordre d'implication parmi les clauses indépendantes de p ayant cette propriété (c'est-à-dire si d est une clause indépendante de p, on a $cc' \Longrightarrow d$ si et seulement si $c'' \Longrightarrow d$). S'il existe une autre variable q apparaissant positivement dans l'une des clauses c, c' et négativement dans l'autre, alors $c_1 + c'_1$ contient $q + \overline{q}$ donc est la clause triviale. Ceci montre qu'il existe trois cas possibles pour deux clauses c et c' :

- ou bien elle n'ont pas de consensus (aucune variable n'apparaît positivement dans une clause et négativement dans l'autre) ;
- ou bien elles ont un unique consensus ;
- ou bien elles ont plusieurs consesnsus mais alors ils sont tous triviaux.

Soit S un système de clauses. On détermine si S est contradictoire de la manière suivante :

1. Simplifier S en retirant toutes les clauses non minimales ;
2. Former tous les consensus non triviaux entre deux clauses de S et les ajouter à S.
3. Recommencer les étapes 1 et 2 tant que l'on obtient en sortie de 2 un système différent de ceui en entrée de 1.
4. Le système d'origine est contradictoire si et seulement si le système final est réduit à $\{Faux\}$.

La terminaison de cet algorithme est immédiate (il n'y a qu'un nombre fini de systèmes de clauses pour un ensemble de variables donné), sa correction l'est moins, ce pourrait être un très bon sujet de problème.

Exemple :

Quand ils ont un devoir, les élèves apprennent leur cours. S'ils ont appris leur cours, ils n'ont pas de mauvaises notes. S'ils n'ont pas de devoir, ils n'ont pas non plus de mauvaise notes. Montrer que les élèves n'ont jamais de mauvaises notes.

On modélise ce problème avec trois variables booléennes : d : « avoir un devoir » ; m : « avoir une mauvaise note » ; c : « apprendre son cours » ; et on dispose des hypothèses :

$$h_1 \equiv (d \Longrightarrow c) \equiv (\overline{c} + d), \quad h_2 \equiv (c \Longrightarrow \overline{m}) \equiv (\overline{c} + \overline{m}), \quad h_3 \equiv (\overline{d} \Longrightarrow \overline{m}) \equiv (d + \overline{m}).$$

Il s'agit de montrer que $h_1\, h_2\, h_3 \Longrightarrow \overline{m}$ soit que le système $S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$ est contradictoire. En déroulant à la main l'algorithme des consensus on obtient successivement :

1. $S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m\}$
2. $S = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m},\ m,\ \overline{d} + \overline{m},\ c + \overline{m}, \overline{c},\ d\}$
1. $S = \{c + \overline{d},\ m,\ \overline{d} + \overline{m},\ c + \overline{m},\ \overline{c},\ d\}$
2. $S = \{c + \overline{d},\ m,\ \overline{d} + \overline{m},\ c + \overline{m},\ \overline{c}, d,\ \overline{d},\ c,\ \overline{m}\}$
1. $S = \{\ m,\ \overline{c},\ d,\ \overline{d},\ c,\ \overline{m}\}$
2. $S = \{\ m,\ \overline{c},\ d,\ \overline{d},\ c,\ \overline{m},\ \mathit{Faux}\}$
1. $S = \{\mathit{Faux}\}$
2. $S = \{\mathit{Faux}\}$

Dans cet exemple on aurait aussi partir du système $S' = \{c + \overline{d},\ \overline{c} + \overline{m},\ d + \overline{m}\}$ et le « simplifier » par l'algorithme des consensus, on obtient en sortie un système contenant $\overline{m}$ ce qui suffit à prouver $h_1\, h_2\, h_3 \Longrightarrow \overline{m}$.

2. Types de données et fonctions fournies

Les variables booléennes seront représentées par des chaînes de caractères, les clauses par des couples de listes de chaînes de caractères et les systèmes de clauses par des liste de tels couples :

```
type clause  == (string list) * (string list);; (* v. positives, v. négatives *)
type systeme == clause list;;

(* affichage d'une clause sous forme d'implication ou de disjonction *)
let rec concat sep = function [] -> "" | [x] -> x | x::suite -> x^sep^(concat sep suite);;
let string_of_clause(p,n) =
  (if n=[] then "" else (concat "." n)^" => ") ^
  (if p=[] then "Faux" else concat "+" p);;

let print_clause(c) = format__print_string(string_of_clause c);;
install_printer "print_clause";;

(* construction de clauses *)
let prefix =>   a b = ([b],  [a])     (* a => b         *)
and prefix =>~  a b = ([],   [a;b])   (* a => non b     *)
and prefix ~=>  a b = ([a;b],[])      (* non a => b     *)
and prefix ~=>~ a b = ([a],  [b])     (* non a => non b *)
;;
```

La fonction `print_clause` affiche une clause c sous la forme $c_1 \Longrightarrow c_2$ où c_1 est la conjonction des variables négatives de c et c_2 la disjonction des variables positives de c (de fait, $(\overline{p} + \overline{q} + r + s) \equiv (p.q \Longrightarrow r + s)$). Le rôle de l'instruction `install_printer "print_clause"` est d'indiquer au système qu'il doit dorénavant utiliser cette fonction

à chaque fois qu'il doit afficher une clause. Les fonctions `=>`, `~=>`, `=>~` et `~=>~` permettent de saisir des clauses sous une forme presque mathématique, on écrira :

```
let s' = ["d" => "c"; "c" =>~ "m"; "d" ~=>~ "m"];;
```

pour désigner le système $S' = \{h_1, h_2, h_3\}$ présenté en exemple. En réponse, le système interractif affiche :

```
s' : (string list * string list) list = [d => c; c.m => Faux; m => d]
```

conformément au code de `print_clause`.

On pourra par ailleurs utiliser les fonctions « à caractère ensembliste » faisant partie de la bibliothèque standard de Caml :

`union l1 l2`	retourne une liste ℓ constituée des éléments de ℓ_1 et des éléments de ℓ_2 ;
`intersect l1 l2`	retourne une liste ℓ constituée des éléments appartenant à la fois à ℓ_1 et ℓ_2 ;
`subtract l1 l2`	retourne une liste ℓ constituée des éléments de ℓ_1 qui n'appartiennent pas à ℓ_2 ;
`mem x l1`	dit si l'objet x appartient à la liste ℓ_1 ;
`except x l1`	retourne une liste ℓ constituée des éléments ℓ_1 autres que x ($\ell = \ell_1$ si x n'appartient pas à ℓ).

3. Programmation de l'étape 1. Écrire les fonctions suivantes :

`implique : clause -> clause -> bool`
`implique` c c' dit si $c \Longrightarrow c'$.

`present : clause -> systeme -> bool`
`present` c s dit s'il existe une clause c' dans s telle que $c' \Longrightarrow c$.

`ajoute : clause -> systeme -> systeme`
`ajoute` c s retourne un système s' équivalent au système $s \cup \{c\}$ en retirant de $s \cup \{c\}$ toutes les clauses $c' \neq c$ telles que $c \Longrightarrow c'$. De la sorte, si l'on part d'un système s ne contenant que des clauses minimales, le système s' retourné a aussi cette propriété.

`simplifie : clause -> systeme -> systeme`
`simplifie` s retourne un système s' équivalent à s et ne contenant que des clauses minimales. Pour ce faire, on ajoutera une à une les clauses de s à un système initialement vide.

4. Programmation de l'étape 2. Écrire les fonctions suivantes :

`ajoute_cons : clause -> clause -> systeme -> systeme`
`ajoute_cons` c c' s calcule le consensus éventuel entre c et c' et, s'il est non trivial, l'ajoute au système s.

`ajoute_consensus : clause -> systeme -> systeme -> systeme`
`ajoute_consensus` c s s' calcule tous les consensus non triviaux entre c et une des clauses de s et les ajoute au système s'.

`cloture : systeme -> systeme`
`cloture` s calcule la clôture par consensus du système s, c'est à dire le système s' que l'on obtient en sortie de l'étape 3. On utilisera la fonction `subtract` pour détecter si deux systèmes sont égaux indépendamment de l'ordre dans lequel sont rangées les clauses dans chaque système.

5. Applications

1. Vérifier avec votre programme que les élèves ne peuvent jamais avoir de mauvaises notes (sous les hypothèses de l'exemple donné plus haut).

2. Les règles d'admission dans un club écossais sont les suivantes :

- *Tout membre non écossais porte des chaussettes rouges.*
- *Tout membre qui porte des chaussettes rouges porte un kilt.*

– *Les membres mariés ne sortent pas le dimanche.*
– *Un membre sort le dimanche si et seulement s'il est écossais.*
– *Tout membre qui porte un kilt est écossais et marié.*
– *Tout membre écossais porte un kilt.*

Montrer que les règles de ce club sont si contraignantes que personne ne peut en être membre.

5. Complément : affichage des étapes d'une contradiction

Le programme précédent dit que le club écossais ne peut pas avoir de membres, mais on ne sait pas vraiment pourquoi : on sait seulement que la machine a trouvé une contradiction entre toutes les règles… En fait il n'est pas difficile d'obtenir une démonstration en bonne et due forme de la contradiction trouvée, il suffit de mémoriser, à chaque fois qu'on produit un nouveau consensus, à partir de quelles clauses il a été obtenu. Lorsqu'on aboutit à la clause *Faux*, il ne reste plus qu'à afficher ces étapes dans le bon ordre.

Modifier le type `clause` de la manière suivante :

```
type clause == (string list) * (string list) * raison
and  raison  =  H | D of clause * clause;;
```

Une clause est à présent constituée de la liste de ses variables positives, de la liste de ses variables négatives, et d'un composante de type `raison` indiquant son origine : `H` pour une hypothèse, `D(c1,c2)` pour une clause déduite (par consensus) des clauses c_1 et c_2.

Modifier en conséquence votre programme de recherche de contradiction et écrire une fonction d'affichage imprimant, lorsqu'une contradiction a été trouvée, toutes les étapes l'ayant produite. Dans le cas du club écossais on obtiendra par exemple :

```
let syst = cloture ["e" ~=>  "r";
                    "r"  =>  "k";
                    "m"  =>~ "d";
                    "d"  =>  "e"; "e" => "d";
                    "k"  =>  "e"; "k" => "m";
                    "e"  =>  "k"]
          in explique (hd syst);;

J'ai l'hypothèse : e+r
J'ai l'hypothèse : r => k
De r => k et e+r je déduis : k+e
J'ai l'hypothèse : k => e
De k+e et k => e je déduis : e
J'ai l'hypothèse : e => k
De e et e => k je déduis : k
J'ai l'hypothèse : m.d => Faux
J'ai l'hypothèse : k => m
De k => m et m.d => Faux je déduis : k.d => Faux
J'ai l'hypothèse : e => d
De k => e et e => d je déduis : k => d
De k.d => Faux et k => d je déduis : k => Faux
De k et k => Faux je déduis : Faux
- : unit = ()
```