

Méthode des tableaux : Optimisation de la détermination de la satisfaisabilité en logique propositionnelle

Contents

1	Introduction	1
1.1	Définition	1
1.2	Principe	1
2	Formules de forme alternée	3
2.1	Définition	3
2.2	Satisfaisabilité	4
2.3	Etude de complexité	4
2.4	Optimisation du stockage par segmentation de donnée	5
2.5	Preuve sous hypothèse	6
3	Logique du 1er ordre	7

1 Introduction

1.1 Définition

Definition 1: La *logique propositionnelle* est un type de logique où les formules sont obtenus par des variables propositionnelles reliées par des connecteurs.

Definition 2 (Modèle): Un *modèle* d'une formule ϕ est une valuation qui rend vraie cette formule. On note l'ensemble des modèles de ϕ par:

$$Mod(\phi) := \{v \in Val | v \models \phi\}$$

Val étant l'ensemble des valuations de ϕ et $v \models \phi$ signifiant que la valuation v satisfait ϕ

Definition 3 (Conséquence Logique): Une formule ϕ est *conséquence logique* d'une formule, notée ψ si $Mod(\psi) \subseteq Mod(\phi)$. On note cela $\psi \models \phi$

Definition 4 (Arbre de déduction): Arbre dont les sommets sont composés de formule, qui sont soit une hypothèse à la racine de l'arbre, soit une formule obtenue par l'application d'une règle sur une formule présente dans la même branche plus proche de la racine.

Definition 5 (Branche fermée): Une branche est fermée si elle contient ϕ et $\neg\phi$

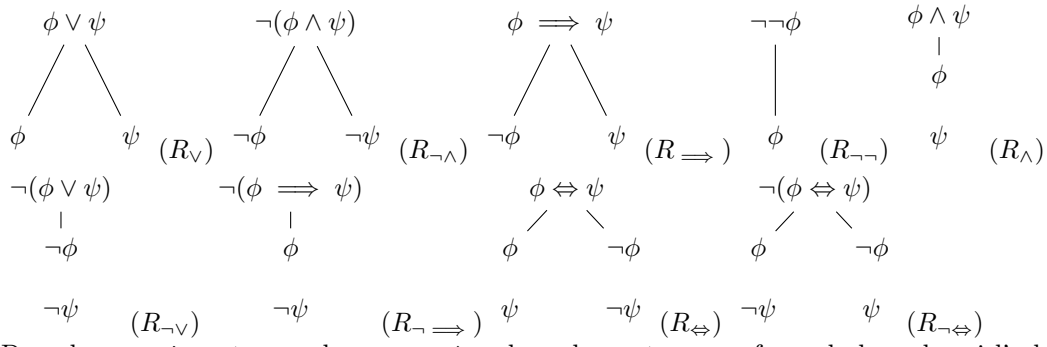
Definition 6 (Arbre fermé): Un arbre de déduction est fermé si toutes les branches le sont.

1.2 Principe

On note $n \in \mathbb{N}^*$, le procédé de la méthode des tableaux consiste donc à séparer les formules logiques complexes en plus petite formule jusqu'à que des paires complémentaires de littéraux (a et $\neg a$) soit extrait ou qu'on ne peut plus simplifier la formule. On en déduit ainsi si une formule ϕ est insatisfaisable ou pas en adoptant des règles qu'on applique sur un arbre de déduction.

- On place $\neg\phi$ dans la racine de l'arbre.
- On applique les règles (R_x) suivantes à chaque branche non fermé de l'arbre
- Si l'arbre est in fine fermé, alors ϕ est vrai

Les règles (R_x) dites Smullyan-style sont:



Dans les cas où on trouve dans une même branche a et $\neg a$, on ferme la branche, si l'arbre devient fermé, alors $\neg\phi$ est forcément faux, donc ϕ est forcément vrai.

Proposition 1: Soit Ψ un ensemble d'hypothèse et ϕ une formule, on note $\Psi \vdash \phi$ l'existence d'un arbre fermé pour $\Psi \cup \{\neg\phi\}$.

$$\Psi \vdash \phi \Leftrightarrow \Psi \models \phi$$

Autrement dit, appliqué la méthode des tableaux à la négation de la formule et ses hypothèses est équivalent à prouver cette formule (*Prouver dans le sens que c'est une tautologie en supprimant Ψ*)

Preuve 1: La preuve de cette proposition permet de prouver la correction de l'implémentation de base. Elle est trouvable en ligne

Cette proposition assure un moyen de prouver une formule.

Proposition 2: Soit Ψ un ensemble d'hypothèse et ϕ une formule, on note $\Psi \vdash \phi$ la non existence d'un arbre fermé pour Ψ .

$$\Psi \vdash \phi \Leftrightarrow \Psi \models \phi$$

Autrement dit, appliqué la méthode des tableaux est équivalent à prouver la satisfaisabilité de cette formule

Preuve 2: En ligne

Cette proposition assure un moyen de prouver la satisfaisabilité d'une formule.

On écrit donc l'algorithme de base de détermination de la satisfaisabilité d'une formule

```

1 type formula =
2   | Atom of string
3   | Not of formula
4   | And of formula * formula
5   | Or of formula * formula
6
7 let rec expand formula =
8   match formula with
9   | Not (Not f) -> [[f]]
10  | Not (And (f1, f2)) -> [[Not f1]; [Not f2]]
11  | Not (Or (f1, f2)) -> [[Not f1; Not f2]]
12  | And (f1, f2) -> [[f1; f2]]
13  | Or (f1, f2) -> [[f1]; [f2]]
14  | _ -> [];
15
16 let rec has_cycle branch =
17   List.exists (fun f -> List.mem (Not f) branch) branch;;
18
19 let rec tableau branches =
20   match branches with
21   | [] -> false
22   | branch :: rest ->
23
24     if has_cycle branch then
25       tableau rest
26     else
27       match branch with
28       | [] -> true
29       | f :: fs ->
30
31         let expansions = expand f in match expansions with
32         | [] -> tableau (fs :: rest)
33         | new_branches ->
34
35           let expanded_branches = List.map (fun b -> b @ fs) new_branches in
36           tableau (expanded_branches @ rest);;
37
38 let is_satisfiable formula =
39   let initial_branch = [formula] in tableau [initial_branch];;

```

2 Formules de forme alternée

2.1 Definition

Definition 7: Soit $n \in \mathbb{N}^*$ et $(\alpha_k)_{k \in [1, n]}$ des littéraux, une formule ϕ est de forme alternée ssi

$$\phi := \alpha_1 \wedge (\alpha_2 \vee (\alpha_3 \wedge (\dots (\alpha_n))))$$

On gardera les parenthèses dans la suite pour garder le côté intuitif de cet écriture et surtout ne pas laisser l'ambiguïté sur la priorité entre les opérateurs logiques

On remarque une CNS pour que ce type de formule soit vrai (pas satisfaisable!):

Proposition 3 (CNS de Vérité): ϕ de forme alternée est vrai ssi:

$$(\exists k \in \mathbb{N}^* \text{ minimal, } \alpha_{2k-2} \text{ OU } n \text{ impair} \implies \alpha_n \text{ avec } k = \frac{n-1}{2}) \text{ ET } \forall i \in [1, k], \alpha_{2i-1}$$

Preuve 3: (\Leftarrow) Immédiat

(\Rightarrow) Supposons ϕ de forme alternée vrai: α_1 est forcément vrai, deux possibilités : soit α_2 est vrai, soit $\alpha_3 \wedge (\dots)$ est vrai. Dans le deuxième cas, on répète le raisonnement sur $\alpha_3 \wedge (\dots)$ qui est bien de forme alternée.

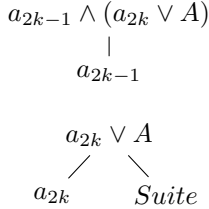
Deux cas de figure:

- On arrête donc le processus dès que un littéral indexé par un nombre pair est vrai. Dans ce cas là, tout les indexés impair précédents sont aussi vrai.
- Si aucun littéral pair est vrai, alors n est impair sans quoi ϕ n'est pas vrai car α_n doit être vrai. On en déduit que tout les littéraux impairs sont vraies, donc la formule est vrai.

Mais ceci n'est pas important, cela aide juste à se donner une idée de la structure de la formule, pour trouver des CNS plus intéressantes.

2.2 Satisfaisabilité

On rappelle que pour prouver la satisfaisabilité d'une formule, on montre qu'il n'existe pas d'arbre fermé. Revenons à la méthode des tableaux, en appliquant les règles à la famille de formule de forme alternée, on observe un pattern sur l'arbre résultant $\forall k \in \mathbb{N}^*$:

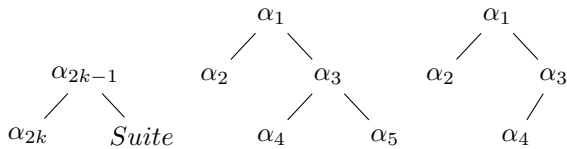


où A est la suite de la forme alternée, qui est aussi de forme alternée ou un littéral.

Proposition 4: Chaque arbre induit par la méthode des tableaux est un arbre binaire, tel qu'on nomme dans les nodes les hypothèses à l'étape associé de l'algorithme

Preuve 4: Soit A_1 et A_2 deux arbres binaires représentant l'arbre induit par la méthode des tableaux T_1 et T_2 respectivement, étiquetés par des littéraux, on suppose qu'ils sont égaux. On peut les reconstruire par une succession d'opération sur une formule par la méthode des tableaux. Etant donné que $A_1 = A_2$, on applique les mêmes opérations, sans quoi on aura pas les mêmes arbres binaires, donc on a la même formule et les mêmes hypothèses, donc $T_1 = T_2$

L'arbre induit ; pour $n = 5$; et pour $n = 4$



Faisons des raisonnements pour des formules de taille $n \in [1, 3]$:

1. Toujours satisfaisable
2. Satisfaisable ssi $\alpha_2 \neq \neg\alpha_1$
3. Satisfaisable ssi $\neg(\alpha_1 = \neg\alpha_2 = \neg\alpha_3)$

C'est à partir du cas $n = 4$ que la satisfaisabilité devient difficile à rétablir. On peut l'établir par 2-SAT, et trouver qu'il y a 4096 formules possibles avec 168 d'entre elles qui sont insatisfaisables, on sent que trouver une CNS peut être compliqué, surtout quand on va prendre des n plus hauts. De plus, ça reviendrait à résoudre une sous instance du problème k-SAT, on sent que trouver une CNS est tout bonnement impossible. Attaquons nous donc à des approches plus réaliste pour le moment.

2.3 Etude de complexité

Etudions les points faibles de notre algorithme en terme de complexité pour pouvoir par la suite étudier des solutions à implémenter. En prenant n le nombre de branches

Premièrement, *expand* et *has_cycle* sont respectivement en $\mathcal{O}(1)$ et $\mathcal{O}(n^2)$. Ensuite, dans le cas où on tombe à chaque fois sur un cycle en analysant une branche, on a une equation de récurrence de type :

$$C(n) = n^2 \times C(n-1)$$

Soit par recurrence immédiate une méthode des tableaux en $\mathcal{O}(n!^2)$

On peut montrer que l'algorithme a une complexité qui est très largement pire que cela, intéressons nous au point critique en terme de complexité. Premièrement, *has_cycle*, qui est appelé à chaque appel récursif, mais aussi et surtout l'expansion.

Rappelons premièrement le code de *has_cycle*:

```
1 let has_cycle branch =
2   List.exists (fun f -> List.mem (Not f) branch) branch;;
```

Il regarde dans la liste représentant la branche et les hypothèses extraites si pour chaque element, il existe sa négation. Une optimisation triviale serait d'éviter les doubles tests, mais cela reste en $\mathcal{O}(n^2)$.

Il faudrait donc utiliser une structure de donnée permettant de vérifier l'existence de la négation d'une formule en un meilleur temps que $\mathcal{O}(n^2)$

Voici les approches que nous allons testé en terme de rapidité:

- Par filtrage de la liste, utilisation de tableau ?
- Utilisation d'un dictionnaire
- Utilisation de serialisation
- Utilisation de SQL
- Par segmentation de donnée par arbre

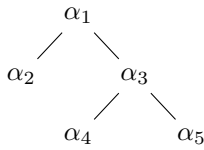
2.4 Optimisation du stockage par segmentation de donnée

On va s'intéresser au dernier étant le plus intéressant pour exploiter la forme particulière de formule qu'on étudie.

Actuellement, nos branches sont stockés sous forme de liste de liste, et nos formules ont une forme générale. Nous allons maintenant stocker nos formules et branches sous forme d'arbre.

```
1 type formula = (* Definition d'un type simple de formule logique *)
2   | Atom of string
3   | Not of string
4   | And of string * formula
5   | Or of string * formula
6
7 type branch = (* Definition de notre stockage des branches*)
8   | Empty
9   | Node of (formula option * formula * branch);;
```

Par exemple, en faisant tourner l'algorithme, sur l'exemple $\alpha_1 \wedge (\alpha_2 \vee (\alpha_3 \wedge (\alpha_4 \vee \alpha_5)))$



Mais on remarque quelque chose, on peut assimiler un tel arbre à une liste chaînée, il faudrait trouver un moyen d'équilibrer les arbres qu'on utilise. Mais pour l'instant, on ignore *gentillement* ce problème d'équilibrage (ce problème implique la recherche d'une bijection qui préserve les propriétés). Mais en réalité, on se rend compte que la méthode des tableaux est absolument parfaite pour ce type de formule:

Proposition 5 (Alternée-SAT est dans P): Il existe un algorithme permettant de résoudre en un temps polynomial le problème Alternée-SAT.

La preuve réside dans l'algorithme qui suit qui est dérivé de la méthode des tableaux. Nous décrirons sa complexité et comment il marche ici.

Avant cela, il faut affirmer la cohérence de ceci, car Alternée SAT est dérivable en 2-FNC-SAT qui est connu comme résolvable polynomialement

```
11 let rec formula2branch (f:formula) : branch = match f with
12   | And(a, Or(b, Atom(c))) -> Node(Some(Atom b), Atom a, Node(None, Atom(c), Empty))
13   | And(a, Or(b, c)) -> Node(Some(Atom b), Atom a, formula2branch c)
14   | And(a, Atom(b)) -> Node(Some (Atom b), Atom a, Empty)
15   | _ -> failwith "Pas alternee"
16
17 let has_cycle (br:branch) : bool =
18   let rec aux (br:branch) (d:(string,bool) Hashtbl.t) : bool = match br with
19   | Node(None, Atom (f, b), Empty) ->
20     if Hashtbl.mem d f then
21       Hashtbl.find d f = b
22     else
23       true
24   | Node(Some(Atom(fg, bg)), Atom (fd, bd), Empty) ->
25     if Hashtbl.mem d fd then
26       if Hashtbl.find d fd = bd then
27         not @@ Hashtbl.mem d fg && Hashtbl.find d fg <> bg
```

```

28         else
29             false
30         else(
31             Hashtbl.add d fd bd;
32             not @@ Hashtbl.mem d fg && Hashtbl.find d fg <> bg)
33 | Node(Some (Atom (fg, bg)), Atom (fd, bd), nb) ->
34     if Hashtbl.mem d fd then
35         if Hashtbl.find d fd <> bd then
36             false
37         else
38             if Hashtbl.mem d fg then
39                 if Hashtbl.find d fg = bg then
40                     true
41                 else
42                     aux nb d
43             else
44                 true
45         else
46             (Hashtbl.add d fd bd;
47             if Hashtbl.mem d fg then
48                 if Hashtbl.find d fg = bg then
49                     true
50                 else
51                     aux nb d
52             else
53                 true)
54 | _ -> failwith "Pas alternee"
55 in aux br (Hashtbl.create 100));
56
57 let is_satisfiable (f:formula) : bool = let b = formula2branch f in has_cycle b;;

```

2.5 Preuve sous hypothèse

Proposition 6: Avec les mêmes définitions, soit $\phi = \alpha_1 \wedge (\alpha_2 \vee (\alpha_3 \wedge (\dots (\alpha_n))))$ une formule de formule alternée:

$$\neg\phi = \neg\alpha_1 \vee (\neg\alpha_2 \wedge (\neg\alpha_3 \vee (\dots (\neg\alpha_n))))$$

Cette forme est cohérente avec les propriétés de \neg .

Preuve 5: On prouve par recurrence forte sur $n \in \mathbb{N}^*$

$$\mathcal{P}(n) : \neg\phi = \neg\alpha_1 \vee (\neg\alpha_2 \wedge (\neg\alpha_3 \vee (\dots (\neg\alpha_n))))$$

$$\mathcal{P}(1) : \neg\phi = \neg\alpha_1 \text{ vrai}$$

$\forall k \in \mathbb{N}^*$, supposons $\mathcal{P}(k)$:

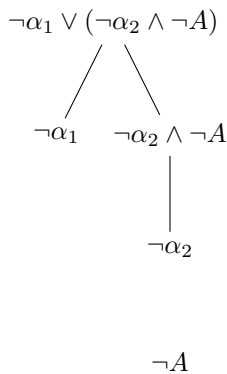
$$\neg\phi = \neg(\alpha_1 \wedge (\alpha_2 \vee (\alpha_3 \wedge (\dots (\alpha_{n+1}))))$$

$$= \neg\alpha_1 \vee (\neg\alpha_2 \wedge \neg(\alpha_3 \wedge (\dots (\alpha_{n+1}))))$$

On trouve le résultat en appliquant l'hypothèse de recurrence sur $\alpha_3 \wedge (\dots (\alpha_{n+1}))$ qui est de forme alternée.

$$\forall n \in \mathbb{N}^*, \mathcal{P}(n)$$

De là, on peut enfin en déduire la forme de l'arbre induit par la méthode des tableaux sur la négation d'une formule de forme alternée sans hypothèse



Proposition 7: $\neg\phi$ de forme alternée est vrai ssi:

$$(\forall k \in \mathbb{N}^*, \neg\alpha_{2k-2} \text{ ET } n \text{ impair ET } \neg\alpha_n \text{ avec } k = \frac{n-1}{2}) \text{ OU } \exists i \in [1, k], \neg\alpha_{2i-1}$$

Preuve 6: On utilise la négation de la proposition 3 ainsi que $\phi \rightarrow F \Leftrightarrow \neg\phi \rightarrow T$

3 Logique du 1er ordre

La logique propositionnelle étant mathématiquement limitée, on se propose l'utilisation de la logique du 1er ordre. Cela nous permettra ainsi d'étudier Zenon, un prouveur d'automatique de theoreme.

Definition 8: La *logique du 1er ordre* est un type de logique qui en plus des éléments de la logique propositionnelle permet l'utilisation de quantificateurs et de *termes*.

Definition 9: Soit un ensemble infini de variables $X = \{x, y, x_1, x_2, \dots\}$ et un ensemble $\mathcal{F} = \{c, f, g, \dots\}$ de symboles de fonction (autrement appelé signature). On rappelle que l'arité d'un symbole est son nombre d'argument. Les termes sont définis par induction:

- $\forall x \in X, x$ est un terme
- Tout symbole d'arité 0 (les constantes) est un terme
- $f(t_1, \dots, t_n)$ est un terme ssi f est un symbole d'arité n et t_1, \dots, t_n sont des termes

On note $\mathcal{T}(\mathcal{F}, X)$ l'ensemble des termes sur \mathcal{F} et X .