

# TIPE 25/26 - Cycles et Boucles

Méthode des tableaux : Optimisation et étude de la satisfiabilité de formule

GIL Dorian

- 1 Introduction Générale et Objectifs
- 2 Etude Préliminaire: Forme Alternée
- 3 Objectifs futurs

# Introduction - Definition

## Definition (Méthode des tableaux)

Algorithme pour prouver qu'une assertion  $\phi$  ayant pour hypothèse  $(H_n)$  soit satisfiable

On supposera que aucune hypothèse n'est faite, on peut facilement adapter l'étude que l'on va faire lors d'ajout d'hypothèse.

# Introduction - Definition

## Definition (Méthode des tableaux)

Algorithme pour prouver qu'une assertion  $\phi$  ayant pour hypothèse  $(H_n)$  soit satisfiable

On supposera que aucune hypothèse n'est faite, on peut facilement adapter l'étude que l'on va faire lors d'ajout d'hypothèse.

- On place  $\phi$  et ses hypothèses dans la racine.
- On applique des règles ( $R_x$ ) à chaque formule en bout d'arbre qui sont developpables
- Si on trouve des contradictions (des *cycles*) dans toutes les branches de l'arbre (branches fermées), l'arbre est fermé donc la formule est insatisfiable.

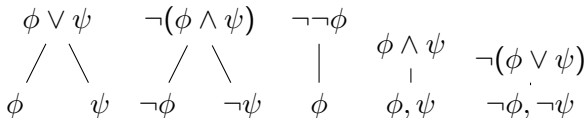
# Introduction - Definition

## Definition (Méthode des tableaux)

Algorithme pour prouver qu'une assertion  $\phi$  ayant pour hypothèse ( $H_n$ ) soit satisfiable

On supposera que aucune hypothèse n'est faite, on peut facilement adapter l'étude que l'on va faire lors d'ajout d'hypothèse.

- On place  $\phi$  et ses hypothèses dans la racine.
- On applique des règles ( $R_x$ ) à chaque formule en bout d'arbre qui sont developpables
- Si on trouve des contradictions (des *cycles*) dans toutes les branches de l'arbre (branches fermées), l'arbre est fermé donc la formule est insatisfiable.



# Introduction - Definition

Les règles définis précédemment sont dites Smullyan-Style

## Definition (Branche fermée)

Une branche est fermée si elle contient  $\phi$  et  $\neg\phi$

Une formule est insatisfiable ssi son arbre associé est dit fermé ssi toutes les branches le sont.

# Introduction - Remarques

- On peut utiliser la méthode des tableaux pour montrer qu'une formule est une tautologie:
  - 1 On place  $\neg\phi$  et ses hypothèses dans la racine.
  - 2 On applique des règles ( $R_x$ ) à chaque formule en bout d'arbre qui sont développables
  - 3 Si on trouve  $a$  et  $\neg a$  dans les branches de l'arbre (des *cycles*), alors  $\phi$  est une tautologie

On pourra donc aussi adapter nos recherches pour la recherche de tautologie.

- Nous allons maintenant introduire un type de formule et étudier la méthode des tableaux sur ce cas particulier, pour en déduire des propriétés intéressantes. Ou même des optimisations de cette méthode.

# Alternée - Definition

## Definition (Forme Alternée)

Soit  $n \in \mathbb{N}^*$ , et  $(a_k)_{k \in [1, n]}$  des littéraux, on dit que  $\varphi$  est de forme alternée ssi

$$\varphi = a_1 \wedge (a_2 \vee (a_3 \wedge (\dots (a_n))))$$

# Alternée - Definition

## Definition (Forme Alternée)

Soit  $n \in \mathbb{N}^*$ , et  $(a_k)_{k \in [1, n]}$  des littéraux, on dit que  $\varphi$  est de forme alternée ssi

$$\varphi = a_1 \wedge (a_2 \vee (a_3 \wedge (\dots (a_n))))$$

On remarquera le parenthesage qui enlève toute ambiguïté sur la priorité entre les opérateurs logiques.

## Theorem (Forme Negative)

$\varphi$  une forme alternée:

$$\neg\varphi = \neg a_1 \vee (\neg a_2 \wedge (\neg a_3 \vee (\dots (\neg a_n))))$$

## Theorem (Forme Negative)

$\varphi$  une forme alternée:

$$\neg\varphi = \neg a_1 \vee (\neg a_2 \wedge (\neg a_3 \vee (\dots (\neg a_n))))$$

## Theorem

*Chaque arbre induit par la méthode des tableaux sur une forme alternée est un arbre binaire tel qu'on nomme dans les nodes les hypothèses.*

# Alternée - Recherche Algorithme

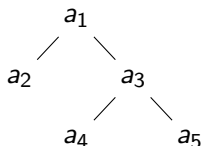
Nous recherchons maintenant un algorithme utilisant la méthode des tableaux pour trouver la satisfiabilité des formes alternées.

Nous allons modéliser conformément à la méthode des tableaux et au dernier théorème présenté comme un arbre binaire, plus précisément en liste chaînée: Par exemple pour  $n = 5$ , on aura le schéma suivant:

# Alternée - Recherche Algorithme

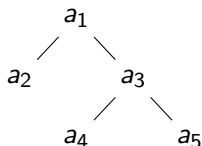
Nous recherchons maintenant un algorithme utilisant la méthode des tableaux pour trouver la satisfiabilité des formes alternées.

Nous allons modéliser conformément à la méthode des tableaux et au dernier théorème présenté comme un arbre binaire, plus précisément en liste chaînée: Par exemple pour  $n = 5$ , on aura le schéma suivant:



# Alternée - Recherche Algorithme

Nous recherchons maintenant un algorithme utilisant la méthode des tableaux pour trouver la satisfiabilité des formes alternées. Nous allons modéliser conformément à la méthode des tableaux et au dernier théorème présenté comme un arbre binaire, plus précisément en liste chaînée: Par exemple pour  $n = 5$ , on aura le schéma suivant:

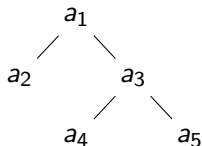


Et ce conformément à la règle de la méthode des tableaux. La construction de cet arbre binaire se fait à priori en  $\mathcal{O}(n)$

# Alternée - Observations

On va faire 3 observations, et on va en déduire un algorithme:

- Si un des littéraux dans une branche gauche (les pairs) ne provoquent aucune contradiction, c'est gagné.
- Dans le cas inverse, on doit rechercher plus profond dans l'arbre (parcours de la branche droite)
- Et ce ainsi de suite, jusqu'à en déduire un arbre fermé, ou pas
- Et ce en prenant aussi en compte les littéraux impairs



# Alternée - Schema Algorithmme

On décrit un appel de l'algorithme qu'on implémente de manière recursive (dans le cas général): Avant cela on crée un dictionnaire. On appelle littéral droit les littéraux impairs et gauche les pairs:

- 1 On analyse le littéral droit, si il y a contradiction, l'arbre est fermé, sinon on ajoute eventuellement dans le dictionnaire le littéral
- 2 On analyse le littéral gauche, si il produit une contradiction, appel recursif plus profond dans l'arbre, sinon la formule est satisfiable

On peut montrer que cette algorithme est en  $\mathcal{O}(n)$

# Alternée - Conséquence

On a donc trouvé un moyen polynomial pour montrer la satisfiabilité d'une forme alternée !!

# Alternée - Conséquence

On a donc trouvé un moyen polynomial pour montrer la satisfiabilité d'une forme alternée !!

- En effet, ce programme est en  $\mathcal{O}(n)$ , c'est un parcours linéaire d'une liste chaînée, utilisant des opérateurs sur les Hashtbl qu'on peut supposé constant.
- On a prouvé la terminaison et la correction du programme:
- La correction (preuve faite) est assuré par l'invariant "Toutes les branches déjà traités sont fermés"
- La terminaison (preuve faite) est assuré simplement.

# Alternée - Conversion

On se demande naturellement quelles sont les formules qu'on peut transformer en formule alternée.

- Quand on traduit une forme alternée, on obtient une FNC dont la taille des clauses est strictement croissante.
- On peut trouver des contres exemples: par exemple  $a_1 \vee a_2$  ne pourra pas s'exprimer sous forme alternée (sauf si on fixe un des littéraux)
- En convertissant toutes les formules en FNC, seules certaines formules ayant des clauses à taille croissante pourront être reconvertit en forme alternée.

## Alternée - Conclusion

On décide de tester la vitesse d'exécution de mon algorithme contre deux algorithmes de satisfiabilités optimisés: Quine et Coq. On utilise un script Python pour générer les mêmes formules pour les différentes structures données demandés par les algorithmes. Sur un dataset de 100 formules genere pseudo-aléatoirement, nous avons les résultats suivants:

- **Alternée:** 0.000493s
- **Quine (avec conversion en CNF):** 0.025874s
- **Quine (sans conversion en CNF):** 0.018691s

Donc notre algorithme est 50 fois plus rapide sur ce dataset, ce qui n'est pas tellement supprenant.

# La logique du 1er ordre

- On ajoute les quantificateurs  $\exists$  et  $\forall$
- Un ensemble de fonctions de symboles  $\mathcal{F}$  qui a des symboles associe un symbole
- Un ensemble de relation  $\mathcal{R}$  qui a des symboles associe un booléen.

On note l'arité le nombre d'argument d'une fonction et  $X$  les variables.

## Definition

On définit les termes  $\mathcal{T}(\mathcal{F}, X)$  par induction:

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

# La méthode des tableaux pour logique du 1er ordre

On ajoute quatres règles:

$$\begin{array}{cccc} \forall x, P(x) & \exists x, P(x) & \neg(\forall x, P(x)) & \neg(\exists x, P(x)) \\ | & | & | & | \\ P(t) & P(c) & \exists x, \neg P(x) & \forall x, \neg P(x) \end{array}$$

où  $t$  et  $c$  est une variable fixe quelquonque.

**Attention:** Pour la règle  $\forall$ , on peut choisir  $t$ , pour la règle  $\exists$ , on prend une variable fraiche  $c$  quelquonque!

# Logique temporelle du temps linéaire: une branche de la logique Modale

# Bibliographie

- 1 Logique: fondements et applications (Dunod) de Pierre Le Barbenchon, Sophie Pinchinat, François Schwarzentruher
- 2 Mathematical Logic: Tableaux Reasoning for Propositional Logic de Chiara Ghidini  
(<https://dit.unitn.it/~ldkr/ml2015/slides/PLtableau.pdf>)
- 3 Tableau Methods for Propositional Logic and Term Logic de Tomasz Jarmużek
- 4 Principles of Model Checking - Christel Baier et Joost-Pieter Katoen

# Code - Méthode des tableaux classique 1

```
type formula =  
  | Atom of string  
  | Not of formula  
  | And of formula * formula  
  | Or of formula * formula  
  
let rec expand formula =  
  match formula with  
  | Not (Not f) -> [[f]]  
  | Not (And (f1, f2)) -> [[Not f1]; [Not f2]]  
  | Not (Or (f1, f2)) -> [[Not f1; Not f2]]  
  | And (f1, f2) -> [[f1; f2]]  
  | Or (f1, f2) -> [[f1]; [f2]]  
  | _ -> [];  
  
let rec has_cycle branch =  
  List.exists (fun f -> List.mem (Not f) branch) branch;;
```

# Code - Méthode des tableaux classique 2

```
let rec tableau branches =  
  match branches with  
  | [] -> false  
  | branch :: rest ->  
  
    if has_cycle branch then  
      tableau rest  
    else  
      match branch with  
      | [] -> true  
      | f :: fs ->  
  
        let expansions = expand f in match expansions with  
        | [] -> tableau (fs :: rest)  
        | new_branches ->  
  
          let expanded_branches = List.map (fun b -> b @ fs) new_branches in  
          tableau (expanded_branches @ rest);;  
  
let is_satisfiable formula =  
let initial_branch = [formula] in tableau [initial_branch];;
```

# Code - Méthode des tableaux classique 1

```
type prop = | Var of string | Not of prop | And of prop * prop | Or of prop *
prop
(* Une branche c'est une liste de formule avec un signe *)
type branch = (bool * prop) list

let is_literal = function
| (true, Var _) -> true
| (false, Var _) -> true
| (true, Not (Var _)) -> true
| (false, Not (Var _)) -> true
| _ -> false

(* Check les contradictions *)
let branch_closed (br : branch) : bool =
  let pos = Hashtbl.create 16 in
  let neg = Hashtbl.create 16 in
  let record = function
    | (true, Var v) -> Hashtbl.replace pos v true
    | (false, Var v) -> Hashtbl.replace neg v true
    | (true, Not (Var v)) -> Hashtbl.replace neg v true
    | (false, Not (Var v)) -> Hashtbl.replace pos v true
    | _ -> ()
  in
  List.iter record br;
  let closed = ref false in
  Hashtbl.iter (fun v _ -> if (Hashtbl.mem pos v) && (Hashtbl.mem neg v) then
    closed := true) pos;
  !closed
```

## Code - Méthode des tableaux classique 2

```
(* La decomposition usuelle faites durant la methode des tableaux *)
let decompose_once (br : branch) : branch list option =
  let rec find_nonlit acc = function
    | [] -> None
    | x :: xs ->
      if is_literal x then find_nonlit (x::acc) xs
      else Some (List.rev acc, x, xs)
  in
  match find_nonlit [] br with
  | None -> None
  | Some (left, (sign, form), right) ->
    let rest = left @ right in
    let mk b p = (b, p) in
    (match sign, form with
    | true, And (a,b) ->
      Some [ (mk true a) :: (mk true b) :: rest ]
    | false, Or (a,b) ->
      Some [ (mk false a) :: (mk false b) :: rest ]
    | true, Or (a,b) ->
      Some [ (mk true a)::rest; (mk true b)::rest ]
    | false, And (a,b) ->
      Some [ (mk false a)::rest; (mk false b)::rest ]
    | true, Not a ->
      Some [ (mk false a) :: rest ]
    | false, Not a ->
      Some [ (mk true a) :: rest ]
    | _, _ -> None)
```

# Code - Méthode des tableaux classique 3

```
(* La Methode des Tableaux en soit *)  
let satisfiable (phi : prop) : bool =  
let initial_branch = [ (true, phi) ] in  
let rec explore_stack stack =  
  match stack with  
  | [] -> false  
  | br :: rest ->  
    if branch_closed br then explore_stack rest else  
    match decompose_once br with  
    | None -> true  
    | Some new_branches -> explore_stack (new_branches @ rest)  
in explore_stack [ initial_branch ]
```

# Code - Alternée 1

```
type formula =
  | Atom of (string* bool)
  | And of (string*bool) * formula
  | Or of (string*bool) * formula

type branch =
  | Empty
  | Node of (formula option * formula * branch);;

let extract (f:formula option) = match f with
  | None -> Atom("none", false)
  | Some t -> t

let rec print_formula (f:formula) = match f with
  | Atom(s, b) -> if b then print_string s else print_string "Not ";
    print_string s;
  | And ((f, b),g) -> if b then print_string f else print_string "Not ";
    print_string f;print_string " And ";print_formula g
  | Or ((f,b),g) -> if b then print_string f else print_string "Not ";
    print_string f;print_string " Or ";print_formula g;;

let rec print_branches (b:branch) =
  print_string "[";
  match b with
  | Empty -> ()
  | Node(a1, a2, b) -> print_formula@@extract a1;print_string ", ";
    print_formula a2;print_branches b;
  print_string "]";;
```

## Code - Alternée 2

```
let rec formula2branch (f:formula) : branch = match f with
| And(a, Or(b, Atom(c))) -> Node(Some(Atom b), Atom a, Node(None, Atom(c),
    Empty))
| And(a, Or(b, c)) -> Node(Some(Atom b), Atom a, formula2branch c)
| And(a, Atom(b)) -> Node(Some (Atom b), Atom a, Empty)
| _ -> failwith "Pas d'alternee"

let has_cycle (br:branch) : bool =
let rec aux (br:branch) (d:(string,bool) Hashtbl.t) : bool = match br with
| Node(None, Atom (f, b), Empty) ->
    if Hashtbl.mem d f then
        Hashtbl.find d f = b
    else
        true
| Node(Some(Atom(fg, bg)), Atom (fd, bd), Empty) ->
    if Hashtbl.mem d fd then
        if Hashtbl.find d fd = bd then
            not @@ Hashtbl.mem d fg && Hashtbl.find d fg <> bg
        else
            false
    else(
        Hashtbl.add d fd bd;
        not @@ Hashtbl.mem d fg && Hashtbl.find d fg <> bg)
| Node(Some (Atom (fg, bg)), Atom (fd, bd), nb) ->
    if Hashtbl.mem d fd then
        if Hashtbl.find d fd <> bd then
```

# Code - Alternée 3

```
false
  else
    if Hashtbl.mem d fg then
      if Hashtbl.find d fg = bg then
        true
      else
        aux nb d
    else
      true
else
  (Hashtbl.add d fd bd;
   if Hashtbl.mem d fg then
     if Hashtbl.find d fg = bg then
       true
     else
       aux nb d
   else
     true)
| _ -> failwith "Pas d'alternee"
in aux br (Hashtbl.create 100);;

let is_satisfiable (f:formula) : bool = let b = formula2branch f in has_cycle b
;;
```