TIPE 25/26 - Cycles et Boucles

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

GIL Dorian

Sommaire

Introduction - Definition

Definition (Méthode des tableaux)

Algorithme pour prouver qu'une assertion ϕ 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 ϕ 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 ϕ 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.

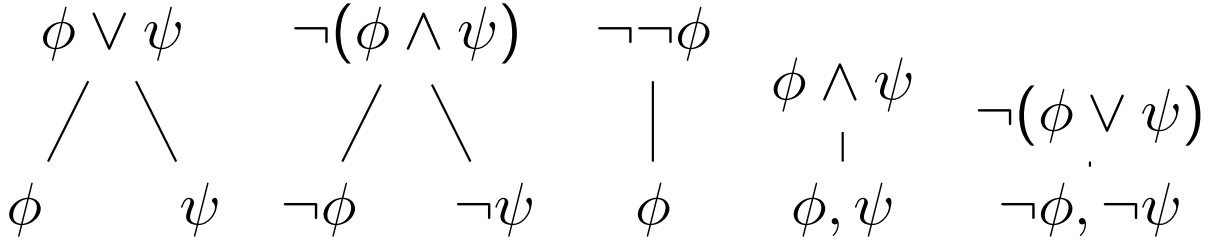
Introduction - Definition

Definition (Méthode des tableaux)

Algorithme pour prouver qu'une assertion ϕ 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 ϕ 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 précedemment sont dites Smullyan-Style

Definition (Branche fermée)

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

Une formule est insatisfisable 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 developpables
  3. Si on trouve a et $\neg a$ dans les branches de l'arbre (des *cycles*), alors ϕ 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 interessantes. 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 litteraux, on dit que φ est de forme alternée ssi

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

Alternée - Definition

Definition (Forme Alternée)

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

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

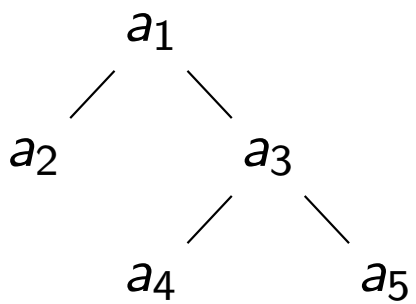
On remarquera le parenthesage qui enlève toute ambiguité sur la prioritée entre les opérateurs logiques.

Alternée - Recherche Algorithme

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

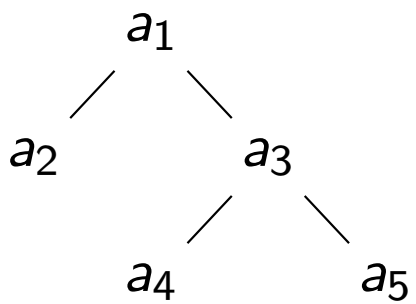
Alternée - Recherche Algorithme

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

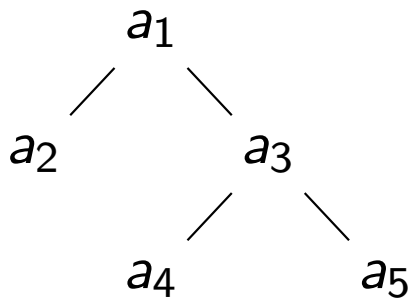


Et ce conformement au règle de la méthode des tableaux. La construction de cette arbre binaire se fait à priori en $\mathcal{O}(n)$

Alternée - Observations

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

- Si un des litteraux 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 litteraux impairs



Alternée - Schema Algorithme

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

1. On analyse le litteral droit, si il y a contradiction, l'arbre est fermé, sinon on ajoute eventuellement dans le dictionnaire le litteral
2. On analyse le litteral 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 chainée, 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'execution 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).
- $f(t_1, \ldots, t_n)$ est un terme si f est un symbole d'arité n et $t_1, \ldots, t_n$ sont des termes.

La méthode des tableaux pour logique du 1er ordre

On ajoute quatres règles:

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

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!

La logique temporelle du temps linéaire

Definition (Formule de la LTL)

On définit F l'ensemble des formules de la LTL inductivement par:

- $p \in AP \implies p \in F$
- si ψ et ϕ sont des formules de LTL alors $\neg\psi, \phi \vee \psi, X\psi, \phi U\psi$ sont des formules de LTL

AP un ensemble fini de variables propositionnelles.

Definition (Opérateurs X et U)

- $X\phi$: ϕ doit être satisfaite dans l'état suivant (neXt)
- $\psi U\phi$: ψ doit être satisfaite dans tous les états jusqu'à un état où ϕ est satisfait (Until)

Etat dans la LTL

Definition (Monde)

On définit un tel objet comme $\omega := w_0, w_1, \ldots$ une suite infinie d'état. On écrira $\omega^i := w_i, w_{i+1}, \ldots$ un suffixe de ω.

Soit $v : \omega \times F \to \{T, F\}$ une fonction de valuation.

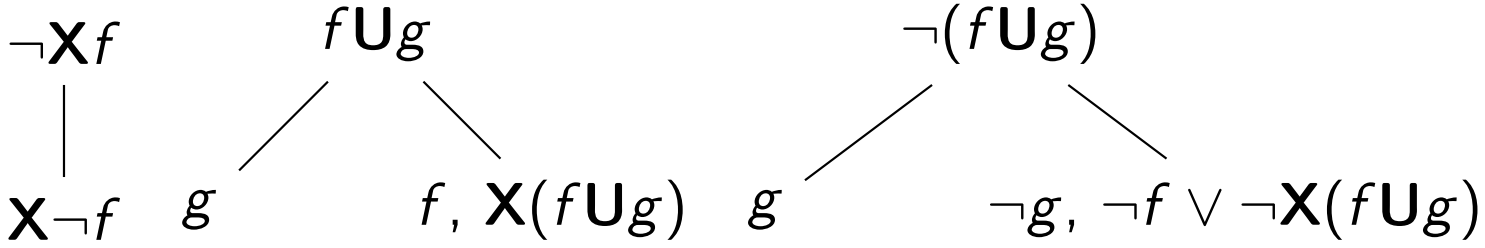
Definition (Satisfaction d'un monde)

En LTL, on définit $\omega \models f$ via:

- $\omega \models a$ si $v(w_0, a) = T$ si a atomique
- $\omega \models \neg f$ si $\neg(\omega \models f)$
- $\omega \models f \vee g$ si $\omega \models f$ ou $\omega \models g$
- $\omega \models \mathbf{X} f$ si $\omega^1 \models f$
- $\omega \models f \mathbf{U} g$ si $\omega \models g$ ou $\omega \models f \wedge \mathbf{X}(f \mathbf{U} g)$

Méthode des tableaux en LTL

On définit deux nouvelles règles pour la méthode des tableaux en LTL:



Definition (Formule élementaire)

f est élementaire ssi il respecte un de ses 2 points:

- C'est une formule atomique (ou la négation d'une formule atomique)
- Il a comme connecteur logique "principale" $\mathbf{X}$ (des X-formules)

Si un noeud contient uniquement des formules elementaires, alors on créé un fils du noeud contenant toutes les $\mathbf{X}$-formules sans leur connecteur logique principal $\mathbf{X}$

Exemples d'utilisation de la méthode des tableaux

On peut trouver plusieurs applications à la LTL et à la méthode des tableaux:

- Preuve de programme concurrentiel
- Raisonnement sur des circuits intégrés
- Raisonnement sur les protocoles de communications

Toutes ces applications s'inscrivent dans le cadre de la vérification de modèles qui est une méthode permettant de montrer la correction de systèmes informatiques complexes.

Definition (Verification de modèle (Model Checking))

Technique de vérification qui explore tout les états possible d'un système de manière force brute.

A propos du model checking

- Ce n'est pas avec la LTL que nous allons faire des gros exemples de model checking (Un model checker de base peut s'occuper de 10^9 états environ, allant jusqu'à 10^{476} pour les meilleurs!).
- Ainsi un model checker typique pourrait utiliser entre autres la LTL pour s'occuper en particulier des deadlocks. C'est ainsi qu'on pourrait utiliser la méthode des tableaux pour prouver une formule qui montre que des algorithmes vont bien s'executer de manière à ne pas avoir d'inter-dépendance.
- Ainsi nous allons examiner un exemple dans lequel nous allons appliquer la méthode des tableaux pour prouver le bon fonctionnement du système choisit!

Automate de Büchi

Un automate de Büchi est un 5-uplets (S, I, T, F, Σ) tel que

- S est un ensemble fini d'état
- $I \subseteq S$ un ensemble d'état initiaux
- $F \subseteq S$ un ensemble d'état finaux
- Σ Un ensemble fini de symboles appelé alphabet
- $T : \{S \times \Sigma\} \mapsto \mathcal{P}(S)$ Une fonction de transition.

Cet automate particulier accepte des séquences infinis (donc des mots infinis) ssi le mot passe par un état acceptant une infinité de fois. C'est un outil utilisé dans le cadre de la vérification de modèle en LTL.

Büchi en LTL

On va utiliser l'algorithme de Gerth pour transformer notre formule LTL en Automate de Büchi Généralisé (GBA)

Definition (GBA)

Un GBA est un automate de Büchi avec la seule particularité que F est un ensemble d'ensemble d'état acceptant appelé **condition d'acceptation**.
Un GBA accepte un mot ssi il est passé au moins une fois par un état dans chaque ensemble d'état acceptant.

Nous nous ramenerons à un Automate de Büchi via un autre algorithme pour simplifier.

GBA to BA

Un ensemble multiple d'état acceptant peut être traduit en un ensemble en construisant un automate par une "counting construction". C'est à dire si $A = (S, I, \{F_1, \ldots, F_n\}, \Sigma, T)$, alors il est equivalent à $A' = (S', I', F, \Sigma, T')$ telle que

- $Q' = Q \times \{1, ..., n\}$
- $I' = I \times \{1\}$
- $F' = F_1 \times \{1\}$
- $\Delta' = \{((q, i), a, (q', j)) | (q, a, q') \in \Delta$ et si $q \in F_i$, $j = i + 1$ mod n sinon $j = i\}$

qui est bien un automate de Büchi.

Résumé

Pour résumé, nous devons:

1. Implémentation de la méthode des tableaux en LTL
2. Implémentation des automates de Büchi
   - Implémentation des automates de Büchi
   - Implémentation de l'algorithme de Gerth
   - Implémentation de la traduction entre GBA et BA
3. Comparaison dans un exemple du model checking en terme de rapidité et compléxité (théorique et pratique)

Contexte

Application, un simple feu rouge ?

Bibliographie

1. Logique: fondements et applications (Dunod) de Pierre Le Barbenchon, Sophie Pinchinat, François Schwarzentruber
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. The tableau method for temporal logic: an overview - Pierre WOLPER
5. 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 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␣alternee"
in aux br (Hashtbl.create 100);;

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