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.

$$\begin{array}{ccc} & \phi \vee \psi & \\ / & & \backslash \\ \phi & & \psi \end{array} \qquad \begin{array}{ccc} & \neg(\phi \wedge \psi) & \\ / & & \backslash \\ \neg\phi & & \neg\psi \end{array} \qquad \begin{array}{c} \neg\neg\phi \\ | \\ \phi \end{array} \qquad \begin{array}{c} \phi \wedge \psi \\ | \\ \phi, \psi \end{array} \qquad \begin{array}{c} \neg(\phi \vee \psi) \\ \\ \neg\phi, \neg\psi \end{array}$$

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

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

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

Alternée - Propriétés

Theorem (Forme Negative)

φ une forme alternée:

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

Alternée - Propriétés

Theorem (Forme Negative)

φ *une forme alternée:*

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

Theorem

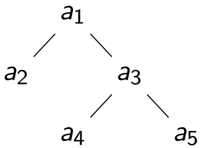
Chaque arbre induit par la méthode des tableaux sur une forme alternée est un arbre binaire tel qu'on nomme dans les nodes les hypothèses.

Alternée - Recherche Algorithme

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

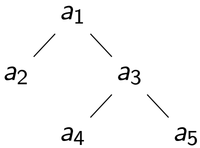
Alternée - Recherche Algorithme

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:

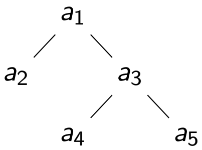


Et ce conformement au règle de la méthode des tableaux. La construction de cette arbre binaire se fait à priori en $\mathcal{O}(n)$

Alternée - Observations

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

- Si un des litteraux dans un arbre 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



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 !! Mais il y a une arnaque:

Theorem

Toute forme alternée peut être écrit en 2-FNC.

Alternée - Conséquence

On a donc trouvé un moyen polynomial pour montrer la satisfiabilité d'une forme alternée !! Mais il y a une arnaque:

Theorem

Toute forme alternée peut être écrit en 2-FNC.

Donc heureusement qu'on arrive à trouver un algorithme polynomial pour résoudre le problème... Néanmoins on gagne théoriquement un tout petit peu en terme de compléxité (même si 2-SAT se résout aussi linéairement)

Alternée - Conclusion

On conclut de cette analyse plusieurs choses:

1. Il faut voir avant toute chose si on ne peut pas trivialement se dériver le problème qu'on pose en un problème déjà résolu
2. Cette étude m'a permit de mieux comprendre la méthode des tableaux.

Nous allons maintenant traiter un cas plus robuste.

Forme ...

Definition (Forme ...)

φ est de forme ... ssi

$$\varphi = a_1 \implies a_2 a_3 \dots a_n$$

Objectifs Futurs et Conclusion

Voici ce que j'aimerai faire pendant les grandes vacances

1. Avancer mon étude de la forme ...
2. M'intéresser à la logique du premier ordre pour faire une ultime étude de la méthode des tableaux

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

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 - 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);;

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