TIPE 25/26 - Cycles et Boucles

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

GIL Dorian

Position du problème

- On cherche à étudier une méthode algorithmique permettant de montrer la satisfiabilité d'une formule: la Méthode des tableaux.

Position du problème

- On cherche à étudier une méthode algorithmique permettant de montrer la satisfiabilité d'une formule: la Méthode des tableaux.
- Cette méthode consiste à construire un arbre avec la formule à la racine, et à utiliser des règles pour développer ou créer des branches.

Position du problème

- On cherche à étudier une méthode algorithmique permettant de montrer la satisfiabilité d'une formule: la Méthode des tableaux.
- Cette méthode consiste à construire un arbre avec la formule à la racine, et à utiliser des règles pour développer ou créer des branches.
- On regarde ensuite si il y a des contradictions dans toutes les branches, si c'est le cas, la formule est insatisfaisable.

Position du problème

- On cherche à étudier une méthode algorithmique permettant de montrer la satisfiabilité d'une formule: la Méthode des tableaux.
- Cette méthode consiste à construire un arbre avec la formule à la racine, et à utiliser des règles pour développer ou créer des branches.
- On regarde ensuite si il y a des contradictions dans toutes les branches, si c'est le cas, la formule est insatisfaisable.
- Cette méthode est utilisé dans diverses logiques, pour l'instant, on se restreint à la logique propositionnelle.

Un exemple graphique

Formule: $\neg(a \Rightarrow (b \Rightarrow a))$

$$\boxed{\neg(a \Rightarrow (b \Rightarrow a))}$$

Un exemple graphique

Formule: $\neg(a \Rightarrow (b \Rightarrow a))$

$$\begin{array}{c} \boxed{\neg(a \Rightarrow (b \Rightarrow a))} \\ \downarrow \\ \boxed{a, \neg(b \Rightarrow a)} \end{array}$$

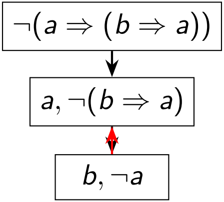
Un exemple graphique

Formule: $\neg(a \Rightarrow (b \Rightarrow a))$

$$\begin{tikzcd}
\boxed{\neg(a \Rightarrow (b \Rightarrow a))} \arrow[d] \\
\boxed{a, \neg(b \Rightarrow a)} \arrow[d] \\
\boxed{b, \neg a}
\end{tikzcd}$$

Un exemple graphique

Formule: $\neg(a \Rightarrow (b \Rightarrow a))$



Première approche

Après l'avoir implémenter, j'ai décidé de me resteindre à une forme particulière de formule logique.

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

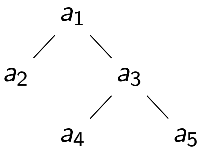
$$\varphi = a_1 \wedge (a_2 \vee (a_3 \wedge (\ldots (a_n))))$$

Notre but en faisant une restriction du problème est:

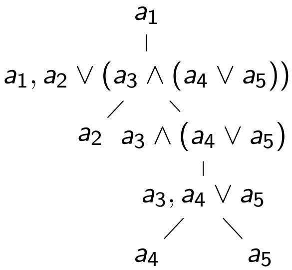
- De mieux comprendre les avantages de cette méthode (dans quelle type de formule la méthode est-il meilleur ?)
- De trouver des algorithmes polynomiales pour nos restrictions (si ce n'est possible, alors on améliorera aux maximum l'algorithme)

Ce que j'ai fait - Algorithme

En utilisant une propriété que j'ai démontré, on va re-écrire l'arbre induit par la méthode des tableaux d'une manière différente:



Au lieu de :

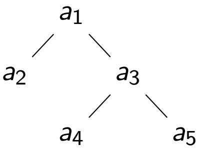


Pour $\varphi = a_1 \wedge (a_2 \vee (a_3 \wedge (a_4 \vee a_5)))$

Résolution du problème

L'algorithme récursif consiste à faire ces analyses (en créant un dictionnaire stockant le "signe" des litteraux):

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



Le cas de base étant l'arriver au bout du peigne.

Preuves et stats

L'algorithme est en $\mathcal{O}(n)$, en supposant les opérations Hashtbl constant.

Preuves et stats

L'algorithme est en $\mathcal{O}(n)$, en supposant les opérations Hashtbl constant.

- La correction (preuve faite) est assuré par l'invariant "Toutes les branches déjà traités sont fermés"
- La terminaison (preuve faite) est assuré simplement.

Preuves et stats

L'algorithme est en $\mathcal{O}(n)$, en supposant les opérations Hashtbl constant.

- La correction (preuve faite) est assuré par l'invariant "Toutes les branches déjà traités sont fermés"
- La terminaison (preuve faite) est assuré simplement.

On créé une base de donnée de 100 formules de forme alternée et on fait tourner Quine et notre algorithme dessus.

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

Objectifs Spé

J'hésite toujours actuellement entres deux dernières approches de mon TIPE:

1. Soit trouver une autre formule de la logique propositionelle à étudier.
2. Soit étudier la méthode des tableaux dans la logique du premier ordre.

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
     ;;
```