TP Satisfiabilité

Préparation à l'agrégation d'informatique, Sorbonne Université, 2024-2025

On s'intéresse au problème SAT de la *satisfiabilité des formules booléennes* : étant donnée une formule sous *forme normale conjonctive*, c'est-à-dire une conjonction de disjonctions de littéraux[1], le problème est de décider s'il existe une *interprétation* des variables propositionnelles, c'est-à-dire une fonction associant à chaque variable une des deux valeurs de vérité V ou F, dans laquelle cette formule est satisfaite.

On dit qu'une clause est un *littéral unitaire* si elle est réduite à un littéral.

Une attention particulière sera accordée à la lisibilité des programmes écrits, ainsi qu'à leur découpage en sous-programmes cohérents. On s'attachera aussi à avoir des types de retour, éventuellement sous forme d'exception, pertinents.

1 Représentation de la logique propositionnelle

On va représenter les formules de la manière suivante :

- les variables propositionnelles comme des entiers strictement positifs. On supposera toujours les variables numérotées entre 1 et n : de ce fait, il faut préciser pour certaines fonctions l'indice maximal des variables propositionnelles ;
- les littéraux comme des entiers relatifs non-nuls ;
- les clauses comme des listes de littéraux ;
- les formules en formes normales conjonctives comme des listes de clauses.

Exercice 1. Donner un type abstrait de litteraux, clauses, et formules de manière à ce que les termes de la Figure 1 soient de type `cnf`. Donner une fonction retournant la variable propositionnelle associée à un littéral.

```
let test1 : cnf = [[1;2;-3];[2;3];[-1;-2;3];[-1;-3];[1;-2]]
let test2 : cnf = [[1;-1;-3];[-2;3];[-2]]
let test3 : cnf = [[1;2;3];[-2;3];[-3]]
let test4 : cnf = [[1;2;3];[-1;2;3];[3];[1;-2;-3];[-1;-2;-3];[-3]]
let test5 : cnf = [[1;-2;3];[1;-3];[2;3];[1;-2]]
let test6 : cnf = [[-1;2];[1;-2];[1;-3];[1;2;3];[-1;-2]]
```

Figure 1 – Un jeu de test

On s'attachera à ce que tous les programmes soient corrects sur les formules de la Figure 1.

2 Approche par simplification

Une première approche du problème va consister à choisir des valeurs pour une variable propositionnelle, simplifier les autres clauses pour en trouver des conséquences ou une absurdité, et se servir des appels récursifs pour faire un retour sur trace.

Exercice 2. Écrire une fonction prenant en entrée un littéral et une formule en forme normale conjonctive et renvoyant la formule simplifiée.

1. Un *littéral* est soit une variable propositionnelle, soit sa négation.

Écrire une fonction prenant en entrée une formule en forme normale conjonctive et renvoyant la première clause unitaire (c'est-à-dire constituée uniquement d'un littéral) ou une erreur.

Exercice 3. Justifiez un type de données `interpretation` pour coder l'interprétation.

Exercice 4. Écrire une fonction prenant en entrée une formule en forme normale conjonctive et renvoyant une interprétation la satisfaisant si elle est satisfaisable, ou une valeur spéciale sinon.

Exercice 5. Donner la complexité en pire cas en temps et en espace du programme de la question précédente. Commenter.

3 Interprétations mutables

La solution précédente n'est pas satisfaisante car l'interprétation partielle est recopiée à chaque appel récursif. Aussi, on va donner une représentation des interprétations partielles permettant la mutation : on définit donc le type

```
1 type pinterpretation = bool option array
```

Exercice 6. Décrire le codage d'une interprétation partielle par un élément de ce type.

Donner une fonction retournant le premier littéral non-défini dans une interprétation partielle. Donner une fonction déterminant si un littéral est satisfait dans une interprétation partielle.

Exercice 7. Écrire deux fonctions, l'une déterminant si un ensemble de clauses est satisfaite dans une interprétation ; l'autre si elle est insatisfaite.

On va stocker, à côté de l'interprétation partielle en cours de construction, une *pile* contenant les choix opérés sur les interprétations. Autrement dit, on va stocker dans un élément du type

```
1 type historique = int Stack.t
```

les variables propositionnelles (codées par des entiers strictement positifs) qui ont été fixées, soit que l'on aie fait un choix, soit que leur valeur a été obtenue par propagation. Avant chaque point de choix, on ajoute un `0` pour savoir jusqu'où effacer dans un retour sur trace.

Exercice 8. Définir une fonction qui met à jour une interprétation partielle pour satisfaire un littéral et modifie l'historique en conséquence.

Exercice 9. Définir une fonction mettant à jour un historique et une interprétation partielle en effaçant l'historique jusqu'au dernier point de choix (inclus).

Exercice 10. Écrire une fonction prenant en entrée une formule, une interprétation partielle et un historique qui cherche tous les littéraux unitaires dans la formule et les ajoute à l'interprétation et à l'historique.

Exercice 11. En déduire un programme résolvant le problème SAT : prenant en entrée une formule dont les variables propositionnelles sont numérotées de 1 à n et le nombre de variables propositionnelles n, elle renvoie une interprétation partielle si la formule est satisfiable, et une valeur particulière sinon.

Exercice 12. Commenter la complexité en temps et en espace.

4 Revue de code

Exercice 13. Corriger, reformater et documenter le code suivant :

```
type 'a elt =
  { vl : 'a
  ; mutable nxt : 'a elt option
  ; mutable prv : 'a elt option
  }
type 'a dlist  = 'a elt option ref
let crdlist ()  = ref None
let edl t = Option.is_none !t
let v e  = e.vl
let f t    = !t
let n e   = e.nxt
let prv e   = e.prv
let ins t vl =
  let ne = { prv = None; nxt = !t; vl } in
  (match !t with
  | Some oldf -> oldf.prv <- Some ne
  | None           -> ());
  t := Some ne

let rem (t:'a dlist) (e:'a elt) =
   let { prv; nxt; _ } = e in
   (match prv with
   | Some p -> p.nxt <- nxt
   | None -> t := nxt);
   (match nxt with
   | Some n -> n.prv <- prv
   | None -> ());
   e.prv <- None;
   e.nxt <- None

let iter (t:'a dlist) ~(f:'a elt -> unit) =
  let rec loop = function
    | None -> ()
    | Some el ->
      f el;
      loop (n el)
  in
  loop !t

let find t ~f =
  let rec loop = function
    | None -> None
    | Some elt -> if f (v elt) then Some elt else loop (n elt)
  in
  loop !t
```

Il est aussi fourni dans le fichier `revue.ml`

5 Littéraux témoins et propagation efficace

On voit que dans les algorithmes précédemment utilisés, il est long d'identifier les littéraux unitaires : on doit reparcourir toute la formule à chaque fois. On se propose donc d'associer à chaque clause non-unitaire deux de ses littéraux, qu'on appelle ses *témoins*. On représente donc dorénavant les clauses par le type

```
type tclause   = {c: clause; mutable tl: litteral list}
```

et on maintiendra, en parallèle, un tableau listant, pour chaque littéral, les clauses dont il est un témoin

```
type temoins   = tclause dlist array
```

ainsi qu'un ensemble de littéraux identifiés comme unitaires

```
type unitaires = litteral Stack.t
```

Exercice 14. Écrire une fonction prenant en entrée une formule, un ensemble de témoins et de littéraux unitaires, et renvoyant une liste de `tclause` représentant cette formule tout en ayant mis-à-jour l'ensemble de témoins et d'unitaires si cela est possible.

Exercice 15. Écrire une fonction déterminant les nouveaux littéraux unitaires quand on met un littéral en argument à faux, et mettant à jour en conséquence les éléments en paramètre.

Exercice 16. En déduire une fonction prenant en entrée une formule et utilisant des littéraux témoins pour en calculer une interprétation partielle la satisfaisant si c'est possible.

6 Référence Ocaml

```
— List.filter_map: ('a -> 'b option) -> 'a list -> 'b list
— List.hd: 'a list -> 'a
— List.find: ('a -> bool) -> 'a list -> 'a
— List.length: 'a list -> int
— List.for_all: ('a -> bool) -> 'a list -> bool
— Stack.push: 'a -> 'a Stack.t -> unit
— Stack.is_empty: 'a Stack.t -> bool
— Stack.top: 'a Stack.t -> 'a
— Stack.pop: 'a Stack.t -> 'a
— Stack.clear: 'a Stack.t -> unit
— Stack.create: unit -> 'a Stack.t
— Stack.to_seq: 'a Stack.t -> 'a Seq.t
— Array.make: int -> 'a -> 'a array
— Array.map: ('a -> 'b) -> 'a array -> 'b array
```