Satisfiabilité

1 Introduction

1.2 Problème SAT

1. Les formules b) et c) sont satisfiables, les formules a) et d) ne le sont pas.

On ne demandait pas de justification, mais pour b), la distribution de vérité qui à tout x_i associe **V** est un modèle (parmi d'autres), et pour c) celle qui à x_0 et x_1 associe **V** et à x_2 associe **F** en est un (le seul). Pour a) et d), quelle que soit la valeur booléenne associée à x_0, la formule devient $x_1 \wedge \neg x_1$, qui n'est pas satisfiable.

2. Il s'agit de trouver la longueur de la plus longue liste dans la liste de listes de littéraux fournie en argument. Cela s'écrit par exemple :

```
let rec dim = function
  | [] -> 0
  | t::q -> max (List.length t) (dim q)
```

Il n'est pas indispensable de redéfinir `List.length` ici car c'est une (des deux!) fonctions OCaml à connaître sans rappel et ce n'est pas le but de la question.

3. On cherche cette fois le plus grand entier i figurant parmi les `V i` et `NV i` dans la liste de lists de littéraux fournie en argument. Le sujet ne précise pas ce qu'il faut faire si la formule est vide (mais c'est une situation toujours satisfiable, sans intérêt concret), on a choisi ici de renvoyer -1 dans ce cas. Par exemple :

```
let rec var_max = function
  | [] -> -1        (* renvoie -1 pour une formule vide *)
  | (V i::q)::r
  | (NV i::q)::r -> max i (var_max (q::r))
  | []::r -> var_max r
```

On aurait aussi pu commencer par déterminer le plus grand entier dans une clause :

```
let rec var_max_clause = function
  | [] -> -1        (* renvoie -1 pour une clause vide *)
  | V i::q
  | NV i::q -> max i (var_max_clause q)

let rec var_max = function
  | [] -> -1        (* renvoie -1 pour une formule vide *)
  | t::q -> var_max (var_max_clause t) (var_max q)
```

4. On peut commencer à déterminer la taille d'une clause. S'il s'agit d'une k-clause, il doit y avoir k feuilles, autant de nœuds « $\neg$ » que de littéraux de la forme $\neg x_i$ et $k-1$ nœuds « $\vee$ ». Cela donne :

```
let rec taille_clause = function
  | [] -> failwith "Erreur: clause vide"
  | [V _] -> 1
  | [NV _] -> 2
  | V _::q -> 2 + taille_clause q   (* 1 + 1 + taille_clause q *)
  | NV _::q -> 3 + taille_clause q  (* 2 + 1 + taille_clause q *)
```

Ou bien :

```
let rec taille_clause c =
  let somme_lst lst = List.fold_left (+) 0 lst in
  List.length c - 1 + somme_lst (List.map (fun V _ -> 1 | NV _ -> 2) c)
```

Puis, s'il y a p clauses, il faut prendre la taille des p clauses et $p-1$ nœuds « $\wedge$ », soit :

```
let rec taille = function
  | [] -> failwith "Erreur: formule vide"
  | [t] -> taille_clause t
  | t::q -> taille_clause t + 1 + taille q
```

Ou bien

```
let rec taille f =
  let somme_lst lst = List.fold_left (+) 0 lst in
  List.length f - 1 + somme_lst (List.map taille_clause f)
```

Si l'on part du principe qu'une formule ne peut pas être vide et qu'une clause ne peut pas l'être non plus, on peut simplifier en :

```
let rec taille_clause = function
  | [] -> -1
  | V _::q -> 2 + taille_clause q
  | NV _::q -> 3 + taille_clause q

let rec taille = function
  | [] -> -1
  | t::q -> taille_clause t + 1 + taille q
```

2 Résolution de 1−SAT

5. On construit un tableau `tab` contenant une distribution de vérité de taille adéquate (attention à ne pas oublier le `+1` !), initialisé à `Indetermine`, puis on parcourt les 1-clauses pour chercher un éventuel conflit qui rendrait la formule insatisfiable. Si on n'en trouve pas, c'est satisfiable.

On a choisi ici, pour les variables n'apparaissant pas dans la formule, de les mettre à `false` lors de la conversion du tableau de triléens en tableau de booléens effectuée par le `Array.map`.

```
let un_sat f =
  let tab = Array.make (var_max f + 1) Indetermine in
  let rec itere = function
    | [] -> Some (Array.map (function tril -> tril = Vrai) tab)
    | [V i]::q -> if tab.(i) = Faux then None
                                    else (tab.(i) <- Vrai; itere q)
    | [NV i]::q -> if tab.(i) = Vrai then None
                                     else (tab.(i) <- Faux; itere q)
    | _ -> failwith "Erreur: pas une 1-clause"
  in itere f
```

3 Résolution de k−SAT pour k arbitraire

6. Cela donne en logique triléenne :

$a \vee b$	a = Vrai	a = Indet.	a = Faux
b = Vrai	Vrai	Vrai	Vrai
b = Indet.	Vrai	Indet.	Indet.
b = Faux	Vrai	Indet.	Faux

7. Rien de bien compliqué ici (attention, on parle de `et`, donc pas de la fonction dont on vient d'écrire le tableau de vérité !) :

```
let et u v = match u, v with
  | Faux, _
  | _, Faux -> Faux
  | Vrai, Vrai -> Vrai
  | _ -> Indetermine
```

8. On commence, comme suggéré par l'énoncé, par écrire une fonction pour une clause :

```
let rec eval_clause t = function
  | V i::q -> ou t.(i) (eval_clause t q)
  | NV i::q -> ou (non t.(i)) (eval_clause t q)
  | [] -> Faux
```

Puis on en déduit une fonction pour une fnc :

```
let rec eval t = function
  | clause::q -> et (eval_clause t clause) (eval t q)
  | [] -> Vrai
```

Notons que de la sorte, on perd le caractère paresseux (on va poursuivre l'évaluation jusqu'au bout de la dernière clause quoi qu'il arrive). Pour le retrouver, on peut écrire :

```
let rec eval_clause t = function
  | V i::q -> if t.(i) = Vrai then Vrai
                              else ou t.(i) (eval_clause t q)
  | NV i::q -> if t.(i) = Faux then Vrai
                               else ou (non t.(i)) (eval_clause t q)
  | [] -> Faux

let rec eval t = function
  | clause::q -> let b = eval_clause t in
                 if b = Faux then Faux
                             else et b (eval t q)
  | [] -> Vrai
```

9. On implémente ici un retour sur trace. On stocke l'état courant des variables propositionnelles dans un tableau `t`, initialisé (1) avec des `Indetermine`. La fonction explorant l'arbre des possibilités (sans le construire) est la fonction `essaie`, qui prend en paramètre le nombre de variables booléennes déjà fixées dans la distribution de vérité (2). Elle renvoie `true` si elle trouve un modèle (qui sera, en logique triléenne, dans `t`), et `false` sinon.

Pour savoir si la branche est intéressante, on effectue (3) une évaluation partielle avec `eval`. `Vrai` signifie que l'on a un modèle (3.a), `Faux` que toutes les branches dans cette direction seront des échecs et qu'il faut donc essayer l'autre branche (3.b), `Indetermine` que la branche peut contenir un modèle (et donc continue à être explorée, (3.c)), mais si ce n'est pas le cas, il faut essayer l'autre branche (3.d).

Si l'exploration détermine que la formule est satisfiable (`essaie` renvoie `true`), alors on fixe toutes les variables encore indéterminées arbitrairement à `false` pour obtenir un modèle (4).

Cela donne donc :

```
let k_sat f =
  let t = Array.make (var_max f + 1) Indetermine in              (* 1 *)
  let rec essaie i =                                             (* 2 *)
    if i = Array.length t then eval t f = Vrai else              (* 5 *)
    begin
      t.(i) <- Vrai;
      match eval t f with                                        (* 3 *)
        | Vrai -> true                                           (* 3.a *)
        | Faux -> t.(i) <- Faux; essaie (i+1)                    (* 3.b *)
        | Indetermine -> essaie (i+1)                            (* 3.c *)
                         || (t.(i) <- Faux; essaie (i+1))        (* 3.d *)
    end
  in if essaie 0
       then Some (Array.map (function tril -> tril = Vrai) t)    (* 4 *)
       else None
```

On notera la condition d'arrêt de l'exploration (5)... Si on y parvient, toutes les variables ont été fixées, donc l'évaluation ne peut plus être indéterminée, mais si la dernière variable a été fixée à `Faux`, on n'a pas encore testé la distribution de vérité courante !

On aurait pu éviter ce cas en évaluant directement la fnc après `t.(i) <- Faux` au (3.d) : `Vrai`, on renvoie `true`, `Faux`, on renvoie `false`, et `Indetermine`, on poursuit l'exploration de l'arbre de possibilités). La complexité est la même, de tout façon, mais cette approche donne une fonction légèrement plus brève.

Cela étant dit, on peut jouer avec une garde et le fonctionnement du filtrage en OCaml pour obtenir une solution un peu plus succinte :

```
let k_sat f =
  let t = Array.make (var_max f + 1) Indetermine in
  let rec essaie i =
      t.(i) <- Vrai;
      match eval t f with
        | Vrai -> true
        | Indetermine when essaie (i+1)-> true
        | _ -> t.(i) <- Faux; i+1 < Array.length t && essaie (i+1)
  in if essaie 0
       then Some (Array.map (function tril -> tril = Vrai) t)
       else None
```

4 Problème SAT

10. On obtient

a) $(x_1 \vee \neg x_0) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_2)$;

b) $(x_0 \vee x_2 \vee x_4) \wedge (x_0 \vee x_2 \vee x_5) \wedge (x_0 \vee x_3 \vee x_4) \wedge (x_0 \vee x_3 \vee x_5) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_5) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee x_5)$.

11. Attention ici à ne pas confondre formules équisatisfiable et équivalentes. Par exemple, si f est une formule ne faisant pas intervenir x_k, f et $f \wedge x_k$ sont équisatisfiables, mais pas équivalentes !

Le point à préciser est celui lié à la disjonction, car toutes les autres transformations remplacent une formule propositionnelle par une formule propositionnelle équivalente (donc nécessairement équisatisfiable). Avec l'introduction d'une nouvelle variable, ce n'est pas trivial pour la disjonction. On va montrer successivement que f^* satisfiable implique f' satisfiable et inversement en utilisant le principe d'induction.

Si f^* est satisfiable, on a nécessairement au moins φ^* satisfiable ou ψ^* est satisfiable. Supposons φ^* satisfiable. Par induction, φ' est satisfiable, et f' est satisfiable avec le même modèle auquel on rajoute $\mu(x) = \mathbf{F}$. Sinon, ψ^* est satisfiable, donc ψ' est satisfiable, et f' est satisfiable avec le même modèle auquel on rajoute $\mu(x) = \mathbf{V}$.

Inversement, si f' est satisfiable, alors si le modèle vérifie $\mu(x) = \mathbf{F}$, on a nécessairement φ' satisfiable, donc φ^* satisfiable, donc f^* satisfiable. Même raisonnement avec ψ si $\mu(x) = \mathbf{F}$.

12. Attention à ne pas oublier de cas !

```
let rec negs_en_bas = function
  | Non (Non f) -> negs_en_bas f
  | Non (Et (f1, f2)) -> Ou (negs_en_bas (Non f1),
                             negs_en_bas (Non f2))
  | Non (Ou (f1, f2)) -> Et (negs_en_bas (Non f1),
                             negs_en_bas (Non f2))
  | Et (f1, f2) -> Et (negs_en_bas f1, negs_en_bas f2)
  | Ou (f1, f2) -> Ou (negs_en_bas f1, negs_en_bas f2)
  | f -> f
```

On fera par ailleurs attention de ne pas mélanger les constructeurs `Et` et `Ou` et les fonctions `et` et `ou` introduites tantôt !

13. On applique les transformations proposés, en utilisant une référence indiquant le plus grand indice actuellement utilisé pour les variables afin de pouvoir en créer de nouvelles lorsque cela est nécessaire.

```
let formule_vers_fnc f =
  let k = ref (var_max_ f) in    (* Dernier indice utilisé *)
  let rec transf = function
    | Var x -> [[V x]]
    | Non (Var x) -> [[NV x]]
    | Et (f1, f2) -> transf f1 @ transf f2
    | Ou (f1, f2) -> incr k;
                     List.map (fun clause -> V !k::clause) (transf f1)
                     @
                     List.map (fun clause -> NV !k::clause) (transf f2)
    | _ -> failwith "Cas impossible"
  in transforme (negs_en_bas f)
```

14. Les complexités ne sont pas triviales. Dans le cas de `negs_en_bas`, on peut voir que la complexité est majorée par la taille de l'arbre *obtenu*. Par ailleurs, l'arbre obtenu est similaire à l'arbre fourni en argument, avec uniquement des changements entre nœuds `Et` et nœuds `Ou` (ce qui ne change pas la taille) et la descente de nœuds `Non`. À cause de ce dernier changement, la taille de l'arbre augmente, mais l'augmentation de la taille est majorée par le nombre final de nœuds `Non`, lui-même majoré par le nombre de feuilles, à son tour majoré par la taille de l'arbre. Donc la complexité reste linéaire en la taille de l'arbre fourni en argument.

Pour la fonction `formule_vers_fnc`, c'est encore plus subtil. La complexité est majorée par la taille de la fnc produite. Mais la fnc est potentiellement bien plus grande que la formule initiale (comme on le voit sur l'exemple b) précédent!) ***

15. Pour déterminer le modèle, il faut mémoriser les disjonctions qui ont produit l'apparition de nouvelles variables x. Si l'algorithme de retour sur trace détermine que $\mu(x) = \mathbf{F}$, il faut trouver un modèle compatible avec le φ^* correspondant (ce qui peut se faire avec le même principe, en exploitant par ailleurs les informations déjà collectées sur la distribution). Sinon, il faut trouver un modèle compatible avec ψ^*.