Corrigé : satisfiabilité des formules booléennes (X-ENS 2016)

Rédigé par Jean-Pierre Becirspahic (jp.becir@info-llg.fr)

Préliminaires

Question 1.

a) La formule $x_1 \wedge (x_0 \vee \neg x_0) \wedge \neg x_1$ n'est pas satisfiable ;

b) la formule $(x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee \neg x_2)$ est satisfaite par les deux valuations $\sigma(x_0) = \sigma(x_1) = \sigma(x_2) =$ **`Vrai`** et $\sigma(x_0) = \sigma(x_1) = \sigma(x_2) =$ **`Faux`** ;

c) la formule $x_0 \wedge \neg(x_0 \wedge \neg(x_1 \wedge \neg(x_1 \wedge \neg x_2)))$ est satisfaite par la valuation $\sigma(x_0) = \sigma(x_1) = \sigma(x_2) =$ **`Vrai`** ;

d) la formule $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$ n'est pas satisfiable.

Question 2. La fonction **`indice`**, de type *litteral –> int*, renvoie l'indice d'un littéral :

```
let indice = function
  | V n  -> n
  | NV n -> n ;;
```

La fonction **`var_max_clause`**, de type *clause –> int*, renvoie l'indice maximal d'une variable présente dans une clause non vide :

```
let var_max_clause (c : clause) =
  it_list (fun a b -> max a (indice b)) 0 c ;;
```

Enfin, la fonction **`var_max`**, de type *fnc –> int*, renvoie l'indice maximal d'une variable présente dans une FNC non vide :

```
let var_max (f : fnc) =
  it_list (fun a b -> max a (var_max_clause b)) 0 f ;;
```

On notera que dans le cas d'une formule vide ou d'une formule ne contenant que des clauses vides, cette fonction renvoie la valeur 0. Dans la suite de ce corrigé nous supposerons les entiers m et n_i qui apparaissent dans la formule (1) *non nuls*, en conséquence de quoi les clauses et les FNC seront représentées en machine par des listes *non vides*.

Partie I. Résolution de 1-SAT

Question 3. On se contente de suivre l'énoncé :

```
let un_sat (f : fnc) =
  let n = var_max f in
  let t = make_vect (n+1) Indetermine in
  let rec aux = function
    | []                                  -> true
    | [V k]::q when t.(k) = Indetermine   -> t.(k) <- Vrai ; aux q
    | [NV k]::q when t.(k) = Indetermine  -> t.(k) <- Faux ; aux q
    | [V k]::q when t.(k) = Faux          -> false
    | [NV k]::q when t.(k) = Vrai         -> false
    | _::q -> aux q
  in aux f ;;
```

Partie II. Résolution de 2-SAT

II.1 Recherche des composantes fortement connexes dans un graphe orienté

Question 4. La complexité d'un parcours en profondeur lorsque le traitement (ici l'insertion en tête de la liste **`resultat`**) est de coût constant est en $O(|V| + |E|)$, autrement dit de coût linéaire en la taille du graphe (c'est un résultat du cours). En

effet, la boucle de la ligne 10 induit un coût en $O(n)$, et le parcours de chacune des listes d'adjacence de la ligne 7 a un coût cumulé en $O(p)$ puisque $\sum_i |g.(i)| = p$.

Question 5.

```
let renverser_graphe (g : graphe) =
  let n = vect_length g in
  let gt = make_vect n [] in
  let rec aux a = function
    | []   -> ()
    | b::q -> gt.(b) <- a::gt.(b) ; aux a q
  in
  for a = 0 to n-1 do aux a g.(a) done ;
  (gt : graphe) ;;
```

Chaque liste d'adjacence est parcourue une fois, la complexité totale de cette fonction est en $O(|V| + |E|)$.

Question 6.

```
let dfs_cfc (g : graphe) l =
  let n = vect_length g in
  let dejavu = make_vect n false in
  let rec dfs lst = function
    | s when dejavu.(s) -> lst
    | s                 -> dejavu.(s) <- true ;
                           it_list dfs (s::lst) g.(s)
  and aux lst = function
    | []                   -> lst
    | s::q when dejavu.(s) -> aux lst q
    | s::q                 -> aux ((dfs [] s)::lst) q
  in aux [] l ;;
```

La fonctionnelle **`it_list`** évite de recourir à une référence de liste comme le fait l'énoncé pour définir la fonction **`dfs_tri`**.

Question 7. Il reste à combiner les différentes fonctions écrites en suivant la description de l'algorithme de Kosaraju-Sharir :

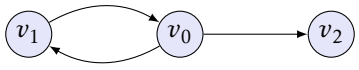
```
let cfc (g: graphe) =
  let l = dfs_tri g and gt = renverser_graphe g
  in dfs_cfc gt l ;;
```

Question 8. Notons $\preccurlyeq$ la relation *être subordonnée à*, et considérons trois composantes fortement connexes C_1, C_2 et C_3. La relation $\preccurlyeq$ est :

- *réflexive*. En effet, si v et v' sont deux sommets de C_1, il existe par définition d'une composante fortement connexe un chemin de v' à v, en conséquence de quoi $C_1 \preccurlyeq C_1$;
- *anti-symétrique*. Considérons deux sommets $v_1 \in C_1$ et $v_2 \in C_2$. Si $C_1 \preccurlyeq C_2$ et $C_2 \preccurlyeq C_1$, il existe un chemin allant de v_2 à v_1 et un chemin allant de v_1 à v_2, donc v_1 et v_2 appartiennent à la même composante fortement connexe et $C_1 = C_2$;
- *transitive*. Considérons trois sommets $v_i \in C_i$ pour $1 \leqslant i \leqslant 3$. Si $C_1 \preccurlyeq C_2$ et $C_2 \preccurlyeq C_3$, il existe un chemin allant de v_2 à v_1 et un chemin allant de v_3 à v_2 donc un chemin allant de v_3 à v_1, ce qui prouve que $C_1 \preccurlyeq C_3$.

La relation $\preccurlyeq$ est donc une relation d'ordre.

Question 9. Le résultat demandé est faux, comme on peut le voir avec l'exemple suivant :



Si le traitement du graphe débute par le sommet v_0, on aura $t_1 < t_2 < t_0$ si v_1 est situé avant v_2 dans la liste d'adjacence de v_0, bien qu'il existe un chemin de v_1 vers v_2 et pas le contraire.

On pourrait reformuler la question en remplaçant t_1 par t_{C_1} où C_1 est la composante fortement connexe contenant v_1, mais je préfère prouver le résultat suivant, qui me semble plus éclairant :

> *Soit* $(v_i, v_j) \in E$ *un arc dont les extrémités appartiennent à deux composantes fortement connexes distinctes* C_i *et* C_j. *Montrer que* $t_{C_i} > t_{C_j}$.

Ce résultat repose sur la constatation suivante : *si* C *est une composante fortement connexe, alors* $t_C = t_i$, *où* v_i *est le premier sommet de* C *à avoir été vu*. En effet, au moment où v_i est vu, tous les sommets de C sont vierges et puisqu'il existe un chemin menant de v_i à chacun de ces sommets, tous font partie de l'arborescence issue de v_i qui sera traitée avant que v_i ne le soit.

Notons maintenant v_k le premier sommet vu de C_i, v_ℓ le premier sommet vu de C_j et traitons deux cas.

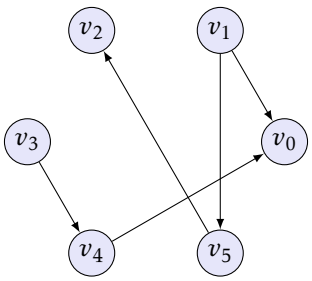
- Si v_k est vu avant v_ℓ, l'exploration de l'arborescence issue de v_k va rencontrer v_i puis v_j et donc aussi v_ℓ. Le traitement de v_ℓ sera donc achevé avant la fin du parcours de cette arborescence et on a donc $t_{C_i} > t_{C_j}$.
- Si v_k est vu après v_ℓ, l'exploration de l'arborescence issue de v_ℓ est nécessairement achevée quand commence celle issue de v_k : en effet dans le cas contraire il existerait un chemin de v_ℓ à v_k et donc de v_j à v_i. On a donc aussi $t_{C_i} > t_{C_j}$.

Question 10. Considérons deux composantes fortement connexes C et C′ telles que $C \preccurlyeq C'$. Si $C = C'$ on a bien entendu $t_C = t_{C'}$; on suppose donc $C \neq C'$ et on considère un chemin reliant un sommet de C′ à un sommet de C. Notons $C_1 = C', \dots, C_p = C$ les différentes composantes fortement connexes rencontrées sur ce parcours.
Pour tout $k \in [\![0, p-1]\!]$ il existe deux sommets consécutifs v_i et v_j sur ce chemin tels que $v_i \in C_k$ et $v_j \in C_{k+1}$. D'après le résultat modifié de la question précédente on a $t_{C_k} > t_{C_{k+1}}$. Ainsi, $t_{C_1} > t_{C_2} > \cdots > t_{C_p}$ et par voie de conséquence $t_{C'} > t_C$.

Question 11. Considérons le sommet v_k en tête de la liste renvoyée par la fonction `dfs_tri`. Il s'agit d'un sommet appartenant à la composante fortement connexe C pour laquelle $t_C = t_k$ est maximale. L'exploration de l'arborescence issue de v_k au sein de G′ traite tous les sommets de C avant que ne s'achève le traitement de v_k. Supposons qu'au cours de cette exploration on ait rencontré un sommet appartenant à une autre composante fortement connexe $C' \neq C$. Il y a donc dans G′ un arc reliant un sommet $v_j \in C$ à un sommet $v_i \in C'$. Mais dans G il y a donc un arc reliant $v_i \in C'$ à $v_j \in C$ et d'après la question 9 modifiée, ceci implique $f(C') > f(C)$, ce qui est absurde.
Ceci montre que l'exploration de l'arborescence à partir de v_k n'a rencontré que les sommets de C, et permet de conclure par récurrence sur le nombre de composantes fortement connexes.

II.2 Des composantes fortement connexes à 2-SAT

Question 12. La formule comporte trois variables distinctes et est associée au graphe d'ordre 6 suivant :



- La clause $x_1 \vee x_2$ crée les deux arêtes $3 \to 4$ et $5 \to 2$;
- la clause x_0 crée l'arête $1 \to 0$;
- la clause $x_2 \vee \neg x_2$ est supprimée ;
- la clause $\neg x_2 \vee x_0$ crée les deux l'arêtes $4 \to 0$ et $1 \to 5$.

Question 13. On passe en revue tous les cas énumérés par l'énoncé :

```
let deux_sat_vers_graphe (f: fnc) =
  let n = var_max f in
  let g = make_vect (2*(n+1)) [] in
  let rec traite_clause = function
    | [V i]                    -> g.(2*i+1) <- (2*i)::g.(2*i+1)
    | [NV i]                   -> g.(2*i) <- (2*i+1)::g.(2*i)
    | [V i; V j] when i <> j   -> g.(2*i+1) <- (2*j)::g.(2*i+1) ;
```

```
                                  g.(2*j+1) <- (2*i)::g.(2*j+1)
    | [V i; NV j] when i <> j  -> g.(2*i+1) <- (2*j+1)::g.(2*i+1) ;
                                  g.(2*j) <- (2*i)::g.(2*j)
    | [NV i; V j] when i <> j  -> g.(2*i) <- (2*j)::g.(2*i) ;
                                  g.(2*j+1) <- (2*i+1)::g.(2*j+1)
    | [NV i; NV j] when i <> j -> g.(2*i) <- (2*j+1)::g.(2*i) ;
                                  g.(2*j) <- (2*i+1)::g.(2*j)
    | [V i; V j] when i = j    -> traite_clause [V i]
    | [NV i; NV j] when i = j  -> traite_clause [NV i]
    | _                        -> ()
  in
  do_list traite_clause f ;
  (g: graphe) ;;
```

Question 14. Posons dans cette question $a = \lfloor i/2 \rfloor$ et $b = \lfloor j/2 \rfloor$.
Considérons tout d'abord le cas d'un arc $(v_i, v_j) \in E$. Il a été créé par la présence dans f d'une clause c de la forme $l_a \vee l_b$ (ou de la forme l_a lorsque $a = b$, clause logiquement équivalente à la précédente). Puisque f est satisfaite par σ, il en est de même de c, et par construction de G l'implication $v_i \Rightarrow v_j$ est aussi satisfaite par σ (en identifiant le sommet v_i et le littéral x_a ou $\neg x_a$ qui le représente).
Par transitivité, ce résultat s'étend lorsqu'il existe un chemin allant de v_i à v_j. Ainsi, lorsque v_i et v_j appartiennent à la même composante fortement connexe, les deux implications $v_i \Rightarrow v_j$ et $v_j \Rightarrow v_i$ sont satisfaites par σ.

- Lorsque $i - j$ est pair, on a $v_i = x_a$ et $v_j = x_b$ ou $v_i = \neg x_a$ et $v_j = \neg x_b$. Puisque l'équivalence $v_i \Leftrightarrow v_j$ est satisfaite par σ on doit avoir $\sigma(x_a) = \sigma(x_b)$.
- Lorsque $i - j$ est impair, on a $v_i = x_a$ et $v_j = \neg x_b$ ou $v_i = \neg x_a$ et $v_j = x_b$. Puisque l'équivalence $v_i \Leftrightarrow v_j$ est satisfaite par σ on doit avoir $\sigma(x_a) = \neg\sigma(x_b)$.

Question 15. Soit x_i une variable présente dans f, et σ une valuation qui satisfait f. Puisque $(2i+1) - (2i)$ est impair les sommets v_{2i} et v_{2i+1} ne peuvent se trouver dans la même composante connexe, faute de quoi la question précédente impliquerait $\sigma(x_i) = \neg\sigma(x_i)$, ce qui est naturellement absurde.

Question 16. Le principe de l'algorithme est simple : on dispose de fonctions déjà écrites pour calculer le graphe associé à f puis les composantes fortement connexes de ce dernier. Il reste à vérifier s'il existe deux sommets v_{2i} et v_{2i+1} dans la même composante.
Pour réaliser ceci en temps linéaire, on crée un tableau associant à chaque sommet le numéro de la composante fortement connexe auquel il appartient. Une fois ce tableau rempli il reste à vérifier par un simple parcours de ce tableau s'il existe i tel que v_{2i} et v_{2i+1} appartiennent à la même composante.

```
let deux_sat (f: fnc) =
  let g = deux_sat_vers_graphe f in
  let cc = cfc g in
  let t = make_vect (vect_length g) 0 in
  let rec attribue_numero k = function
    | []   -> ()
    | v::q -> t.(v) <- k ; attribue_numero k q
  and numerote k = function
    | []   -> ()
    | c::q -> attribue_numero k c ; numerote (k+1) q
  in numerote 1 cc ;
  let rec verifie = function
    | i when i >= vect_length g -> true
    | i when t.(i) = t.(i+1)    -> false
    | i                         -> verifie (i+2)
  in verifie 0 ;;
```

La fonction **`attribue_numero k`**, de type *int list –> unit* attribue aux éléments d'une composante fortement connexe passée en argument le numéro k.
La fonction **`numerote`**, de type *int –> int list list –> unit*, attribue à chaque sommet le numéro de la composante auquel il appartient, la numérotation des composantes étant arbitrairement fixée par l'ordre induit par le retour de la fonction **`cfc`**.
Enfin, la fonction **`verifie`** se charge de déterminer s'il existe i tel que v_{2i} et v_{2i+1} se sont vus attribuer le même numéro.

Partie III. Résolution de k-SAT pour k arbitraire

Question 17.

```
let et = fun
  | Vrai Vrai -> Vrai
  | Faux _    -> Faux
  | _    Faux -> Faux
  | _    _    -> Indetermine ;;
```

```
let ou = fun
  | Faux Faux -> Faux
  | Vrai _    -> Vrai
  | _    Vrai -> Vrai
  | _    _    -> Indetermine ;;
```

```
let non = function
  | Vrai -> Faux
  | Faux -> Vrai
  | _    -> Indetermine ;;
```

Question 18.

```
let eval (f: fnc) t =
  let rec eval_clause b = function
    | []                          -> b
    | (V x)::_ when t.(x) = Vrai  -> Vrai
    | (V x)::q when t.(x) = Faux  -> eval_clause b q
    | (V x)::q                    -> eval_clause Indetermine q
    | (NV x)::_ when t.(x) = Faux -> Vrai
    | (NV x)::q when t.(x) = Vrai -> eval_clause b q
    | (NV x)::q                   -> eval_clause Indetermine q
  in
  let rec eval_fnc b = function
    | []                                  -> b
    | c::q when eval_clause Faux c = Faux -> Faux
    | c::q when eval_clause Faux c = Vrai -> eval_fnc b q
    | c::q                                -> eval_fnc Indetermine q
  in eval_fnc Vrai f ;;
```

La fonction **`eval_clause Faux`**, de type *clause –> trileen*, évalue une clause en fonction de la valuation partielle t.
La fonction **`eval_fnc Vrai`**, de type *fnc –> trileen*, évalue une FNC en fonction de la valuation partielle définie par t.

Question 19. On propose le code suivant :

```
let k_sat (f: fnc) =
  let n = var_max f in
  let t = make_vect (n+1) Indetermine in
  let rec aux = function
    | k when eval f t = Vrai -> raise Exit
    | k when eval f t = Faux -> ()
    | k -> t.(k) <- Vrai ; aux (k+1);
           t.(k) <- Faux ; aux (k+1)
  in try aux 0 ; false
     with Exit -> true ;;
```

La fonction **aux**, de type *int –> unit* a pour objet d'explorer la partie utile de l'arbre.
Si, lors de ce parcours, le résultat d'une évaluation partielle donne Vrai, on déclenche une exception pour interrompre l'exploration.
Lorsque le résultat d'une évaluation donne Faux, il n'y a pas d'appel récursif : l'exploration de la branche courante s'interrompt.
Dans les autres cas, les deux branches sont explorées. Notons qu'il n'y a aucun risque de débordement puisqu'une fois attribuées la valeur Vrai ou Faux à chacune des $n+1$ variables le résultat de l'évaluation ne peut plus être indéterminé.

Partie IV. De k-SAT à SAT

Question 20. On obtient (en détaillant les calculs) :

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

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

$= \neg((\neg x_0 \wedge \neg x_2 \wedge \neg x_4) \vee (\neg x_0 \wedge \neg x_2 \wedge \neg x_5) \vee (\neg x_0 \wedge \neg x_3 \wedge \neg x_4) \vee (\neg x_0 \wedge \neg x_3 \wedge \neg x_5) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_4)$
$\vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_5) \vee (\neg x_1 \wedge \neg x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge \neg x_3 \wedge \neg x_5))$

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

Question 21. Si f est une formule, notons $\mathcal{V}(f)$ l'ensemble des variables présentes dans f. Le résultat que nous allons établir repose sur la constatation suivante :

si $f^* = \phi^* \wedge \psi^*$ ou $f^* = \phi^* \vee \psi^*$, $\mathcal{V}(\phi') \cap \mathcal{V}(\psi') \subset \mathcal{V}(f^*)$;

Autrement dit, les variables qui ont été rajoutées pour former ϕ' et ψ' à partir de ϕ^* et ψ^* sont différentes.
Puisque f et f^* sont logiquement équivalentes elle sont *a fortiori* équisatisfiables ; il suffit donc montrer que f^* et f' sont équisatisfiables. Nous allons raisonner par induction structurelle sur f^*, en prouvant le résultat plus fort suivant :

(i) si f^* est satisfaite par une valuation σ, f' est satisfaite par une valuation qui prolonge σ sur $\mathcal{V}(f')$;
(ii) si f' est satisfaite par une valuation σ', f^* est satisfaite par la restriction de σ' sur $\mathcal{V}(f^*)$.

– Si $f^* = x_i$ est une variable booléenne, $f' = x_i$ et le résultat est évident.
– Si $f^* = \neg\phi^*$, ϕ^* est nécessairement une variable compte tenu de l'étape 1. de l'algorithme donc $f^* = \neg x_i$ et $f' = f^*$.
– Si $f^* = \phi^* \wedge \psi^*$, on suppose le résultat acquis pour ϕ^* et ψ^*. On a $f' = \phi' \wedge \psi'$.
Si f' est satisfaite par une valuation σ', celle-ci satisfait à la fois ϕ' et ψ' ; par hypothèse, la restriction de σ' à $\mathcal{V}(f^*)$ satisfait ϕ^* et ψ^*, et donc f^*.
Si f^* est satisfaite par une valuation σ, celle-ci satisfait à la fois ϕ^* et ψ^*. Compte tenu de la remarque faite en préambule à cette question, σ peut être prolongée pour satisfaire à la fois ϕ' et ψ' et donc f'.
– Enfin, si $f^* = \phi^* \vee \psi^*$:
Supposons f' est satisfaite par une valuation σ'. Si $\sigma'(x) =$ vrai alors σ' satisfait ψ' ; si $\sigma'(x) =$ faux alors σ' satisfait ϕ'. Dans tous les cas, σ' satisfait $\phi' \vee \psi'$, et la restriction de σ' à $\mathcal{V}(f^*)$ satisfait donc f^*.
Si f^* est satisfaite par une valuation σ, celle-ci satisfait ϕ^* ou ψ^* et peut donc être prolongée en une valuation σ' qui satisfait ϕ' ou ψ'. Dans le premier cas, on pose $\sigma'(x) =$ faux ; dans le second on pose $\sigma'(x) =$ vrai. On obtient ainsi une valuation qui satisfait f'.

Question 22.

```
let rec negs_en_bas = function
  | Var n           -> Var n
  | Non (Var n)     -> Non (Var n)
  | Non (Non f)     -> negs_en_bas f
  | Non (Et (f, g)) -> Ou (negs_en_bas (Non f), negs_en_bas (Non g))
  | Non (Ou (f, g)) -> Et (negs_en_bas (Non f), negs_en_bas (Non g))
  | Et (f, g)       -> Et (negs_en_bas f, negs_en_bas g)
  | Ou (f, g)       -> Ou (negs_en_bas f, negs_en_bas g) ;;
```

Question 23. Non demandée, la nouvelle fonction **`var_max`** se définit simplement :

```
let rec var_max = function
  | Var n     -> n
  | Non f     -> var_max f
  | Et (f, g) -> max (var_max f) (var_max g)
  | Ou (f, g) -> max (var_max f) (var_max g) ;;
```

On définit alors la fonction :

```
let formule_vers_fnc f =
  let vm = ref (var_max f) in
  let rec aux = function
  | Var n       -> [[V n]]
  | Non (Var n) -> [[NV n]]
  | Non f       -> failwith "négation non descendue"
  | Et (f1, f2) -> (aux f1) @ (aux f2)
  | Ou (f1, f2) -> let x = !vm + 1 in incr vm ;
                   let phi = aux f1 and psi = aux f2 in
                   (map (function l -> (V x)::l) phi) @ (map (function l -> (NV x)::l) psi)
  in aux f ;;
```

Puisqu'il faut introduire de nouvelles variables, on utilise une référence `vm` pour garder en mémoire le plus grand indice de variable utilisé dans la formule ; cette référence est incrémentée à chaque ajout d'une nouvelle variable.

Question 24. La fonction `negs_en_bas` effectue un parcours en profondeur de l'arbre associé à la formule f donc son coût est linéaire vis-à-vis de la taille de f.

Comme le suggère l'énoncé, montrons maintenant que le nombre de clauses $\mathrm{nc}(f')$ dans f' est égal au nombre de littéraux $\mathrm{nl}(f^*)$ présents dans f^*, en raisonnant par induction structurelle :

- c'est évident dans le cas d'une variable ou d'une négation : on a $\mathrm{nl}(f^*) = \mathrm{nc}(f') = 1$;
- si $f^* = \phi^* \wedge \psi^*$, on ajoute aux clauses de ϕ' les clauses de ψ' donc $\mathrm{nc}(f') = \mathrm{nc}(\phi') + \mathrm{nc}(\psi')$ et $\mathrm{nl}(f^*) = \mathrm{nl}(\phi^*) + \mathrm{nl}(\psi^*)$ donc si le résultat est vrai pour ϕ^* et ψ^* il reste vrai pour f^* ;
- si $f^* = \phi^* \vee \psi^*$, on a toujours $\mathrm{nl}(f^*) = \mathrm{nl}(\phi^*) + \mathrm{nl}(\psi^*)$ ainsi que $\mathrm{nc}(f') = \mathrm{nc}(\phi') + \mathrm{nc}(\psi')$ puisqu'on se contente d'ajouter x ou $\neg x$ dans chacune des clauses de ϕ' et ψ'. Le résultat reste donc vrai pour f^*.

On peut donc majorer le coût de chacune des concaténations effectuées par la fonction `formule_vers_fnc` par un $\mathrm{O}(\mathrm{nl}(f^*))$, elle-même majorée par un $\mathrm{O}(|f^*|)$, où $|f^*|$ désigne la taille de f^*. S'agissant d'un parcours en profondeur de l'arbre associé à la formule f^*, le coût total de la fonction `formule_vers_fnc` est donc un $\mathrm{O}(|f^*|^2)$, polynomial comme annoncé.