Satisfiabilité des formules booléennes (X-ENS 2016)
Un corrigé

Préliminaires

Pour la commodité d'écriture, on utilisera parfois les notations 0 et 1 pour `Faux` et `Vrai`.

Q.1
a. $x_1 \wedge (x_0 \vee \neg x_0) \wedge \neg x_1$ n'est pas satisfiable (vaut 0 que x_1 soit égal à 0 ou 1).
b. $(x_0 \vee \neg x_1) \wedge (\neg x_0 \vee x_2) \wedge (x_1 \vee \neg x_2)$ est satisfiable (donner à tous les x_i la valeur 1).
c. $x_0 \wedge \neg(x_0 \wedge \neg(x_1 \wedge (x_1 \wedge \neg x_2)))$ est satisfiable (prendre $x_0 = 1$ et $x_1 = 0$).
d. $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$ n'est pas satisfiable (les deux premières disjonctions imposent $x_1 = 1$ et les deux dernières $x_1 = 0$).

Q.2 Une première fonction (de type `clause` $\rightarrow$ `int`) donne l'indice maximum d'une variable présente dans une clause. Le cas d'une clause vide est particulier. On pourrait l'exclure en renvoyant une erreur ou exception. On peut aussi convenir que l'indice maximal d'une clause vide est l'entier minimum en Caml. On vérifie que cette convention est cohérente avec le cas d'une clause de longueur 1.

```
let rec var_max_clause c=
    match c with
   |[] -> min_int
   |(V x)::q -> max x (var_max_clause q)
   |(NV x)::q -> max x (var_max_clause q) ;;
```

Le principe est le même pour la fonction demandée en itérant sur les clauses.

```
let rec var_max f =
    match f with
   |[] -> min_int
   |c::q -> max (var_max_clause c) (var_max q) ;;
```

1 Résolution de 1-SAT

Q.3 On commence par initialiser un tableau assez grand de triléens ; sa taille dépend bien sûr du numéro maximal de variable. On itère ensuite sur les clauses supposées de taille 1 avec une fonction `aux_un_sat : clause` $\rightarrow$ `booléen`. Cette fonction indique si la formule (supposée être une $1-FNC$) est satisfiable ET met à jour le tableau des triléens.

```
let un_sat f =
   let sigma=make_vect (1+(var_max f)) Indetermine in
   let rec aux_un_sat f =
      match f with
     |[] -> true
     |[V x]::q -> if sigma.(x)=Faux then false
                  else if sigma.(x)=Vrai then aux_un_sat q
                  else begin
                     sigma.(x) <- Vrai ;
                     aux_un_sat q
                     end
     |[NV x]::q -> if sigma.(x)=Vrai then false
```

```
                    else if sigma.(x)=Faux then aux_un_sat q
                    else begin
                       sigma.(x) <- Faux ;
                       aux_un_sat q
                       end
  in aux_un_sat f;;
```

2 Résolution de 2-SAT

2.1 Recherche de composantes fortement connexes

Q.4 La fonction `dfs_rec` n'induit de parcours de la liste d'adjacence d'un sommet que si celui-ci n'est pas déjà marqué comme vu dans le tableau `deja_vu`. Chaque liste d'adjacence est donc parcourue au plus une fois (puisqu'un sommet désigné comme vu ne change plus ensuite de statut). Du fait dela boucle, chaque liste est même parcourue une unique fois. Lors du traitement d'un sommet, on effectue (hors appels récursifs) un nombre constant d'opérations. Les appels à `dfs_rec` ont donc un coût global majoré par la somme des $1 + l_i$ où l_i est le nombre de voisins de i et donc un coût $O(taille(g))$. Hors de la boucle, on a un coût $O(|V|)$ pour initialiser le tableau. Le coût de `dfs_tri` est donc linéaire en la taille du graphe argument.

Q.5 On crée un tableau `rg` de listes vides que l'on remplit petit à petit. Pour cela, on examine la liste d'adjacence `l` de chaque sommet i. Si j est dans `l` alors il faut placer i dans la liste d'adjacence `rg.(j)` du nouveau graphe. La fonction auxiliaire `renverser : int list` $\rightarrow$ `int` $\rightarrow$ `unit` prend en argument `l` et i et fait les mises à jour dans `rg`.

```
let renverser_graphe g =
   let n=vect_length g in
   let rg=make_vect n [] in
   let rec renverser l i =
      match l with
     |[] -> ()
     |j::q -> rg.(j)<-i::rg.(j) ;renverser q i in
   for i=0 to n-1 do
      renverser g.(i) i
      done;
   rg;;
```

Q.6 J'essaye ici de donner une version qui se rapproche le plus possible de la fonction `dfs_tri` donnée par l'énoncé. Deux choses changent essentiellement.

- On doit itérer sur les éléments d'une liste et non d'un tableau. Il est donc naturel d'introduire une fonction auxiliaire récurrente `parcourt`.
- On doit renvoyer une liste de listes et non plus une liste.

La fonction récurrente de parcours en profondeur du graphe s'appelle maintenant `composantes`. Elle prend en argument un entier i et modifie une liste référence `resultat` en lui ajoutant les nouveaux sommets rencontrés dans un parcours en profondeur à partir de i.
Pour chaque élément i de la liste `ls` de sommets, on effectue un parcours en profondeur si i n'a pas déjà été visité. On réinitialise donc la référence `resultat`, on effectue le parcours depuis i pour obtenir une composante c. Dans tous les cas, on continue avec le reste de la liste. Dans le premier cas, on ajoute c à la liste résultat. On a ainsi le type `parcourt : in list` $\rightarrow$ `int list list`.

```
let dfs_cfc rg ls =
   let deja_vu = make_vect (vect_length rg) false in
   let resultat=ref [] in
   let rec composantes i =
       if not deja_vu.(i) then begin
           deja_vu.(i) <- true ;
           do_list composantes rg.(i) ;
           resultat := i::!resultat
           end in
   let rec parcourt l =
       match l with
      |[] -> []
      |i::q -> if not deja_vu.(i) then begin
                       resultat:=[];
                       composantes i;
                       let c= !resultat in
                       c::(parcourt q) end
                else parcourt q
   in parcourt ls;;
```

Q.7 Il suffit de combiner les différentes étapes.

```
let cfc g =dfs_cfc (renverser_graphe g) (dfs_tri g);;
```

Q.8 Une composante fortement connexe est subordonnée à elle même (on prend un élément x de cette composante, qui est non vide, et le chemin de x vers lui même de longueur nulle). On a ainsi la reflexivité.
Si C_1 est subordonnée à C_2 et C_2 subordonnée à C_3, il existe un chemin de C_3 vers C_2 et un de C_2 vers C_1. La concaténée de ces chemins va de C_3 vers C_1 et C_1 est subordonnée à C_3. Ceci donne la transitivité.
Si C_1 est subordonnée à C_2 et réciproquement alors il y a un chemin de C_1 vers C_2 (de x_1 vers x_2) et un autre de C_2 vers C_1 (de y_2 vers y_1). Soient $x, y \in C_1 \cup C_2$; si x, y sont dans le même C_i, ils sont reliés. Sinon, on a par exemple $x \in C_1$ et $y \in C_2$. On a un chemin de x vers x_1, un autre de x_1 vers y, un autre de y vers y_2 et la concaténée donne un chemin de x vers y. Par maximalité d'un composante fortement connexe, on a $C_1 \cup C_2 = C_1 = C_2$. ceci donne l'antisymétrie.

Q.9 On suppose qu'il y a un chemin de v_i à v_j mais pas de v_j à v_i (et on a donc $v_i \neq v_j$). Deux cas sont possibles.

- Dans le premier parcours en profondeur, on rencontre le sommet v_i avant le sommet v_j (l'instant d_i de début de traitement de v_i est inférieur à celui d_j de v_j). Comme il existe un chemin de v_i vers v_j, on va commencer le traitement de v_j avant de terminer celui de v_i ($d_i < d_j < t_i$). Par définition du parcours en profondeur, on doit alors terminer le traitement de v_j avant celui de v_i (v_j est empilé "au-dessus" de v_i et sera donc dépilé avant v_i).
- Avec les mêmes notations, $d_j < d_i$. Comme il n'existe pas de chemin de v_j vers v_i, v_i n'apparaît pas dans le parcours en profondeur à partir de v_j et donc $t_j < d_i$. A fortiori, $t_j < t_i$ à nouveau.

Q.10 Supposons C subordonnée à C'. Il existe donc un chemin d'un sommet y de C' vers un sommet x de C.

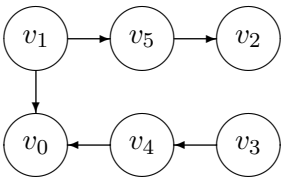
- Si $C = C'$ alors $t_C = t_{C'}$ et donc $t_C \leq t_{C'}$.

- Sinon, considérons un sommet $v_j \in C$. Comme C est une composante connexe, il y a un chemin de x vers v_j. Soit $v_i \in C'$ tel que $t_{C'} = t_i$. C' étant une composante connexe, il y a un chemin de v_i vers y. Il y en a donc aussi un de v_i vers v_j. Comme $C \neq C'$, il n'y en a alors pas de v_j vers v_i. On a ainsi, avec la question précédente, $t_j < t_i = t_{C'}$. Comme ceci est vrai pour tout $v_i \in C$, on en déduit que $t_C \leq t_{C'}$.

Q.11 Notons $C_1, \ldots, C_p$ les composantes fortement connexes en choisissant la numérotation en sorte que $t_{C_1} > \cdots > t_{C_p}$. Dans l'étape b de l'algorithme, on commence donc le parcours à partir d'un élément $v_1 \in C_1$. Pour $i \geq 2$, C_1 n'est pas subordonnée à C_i dans G (avec la question précédente et car $t_{C_1} > t_{C_i}$). Avec la seconde remarque de l'énoncé, C_i n'est donc pas subordonnée à C_1 dans G'. Ainsi, les seuls éléments atteignables dans le parcours à partir de v_1 dans G' sont des éléments de C_1. Lors de ce parcours, on obtient bien tous les éléments de C_1 par définition du parcours en profondeur (qui permet d'atteindre tout sommet v tel qu'il existe un chemin de v_1 à v dans G', c'est à dire de v à v_1 dans G et ce qui est vérifié par tout élément de C_1 par définition d'une composate fortement connexe). On recommence alors avec un sommet de C_2 pour obtenir exactement C_2 et ainsi de suite.

2.2 Des composantes fortement connexes à 2-SAT

Q.12 La clause $(x_2 \vee \neg x_2)$ disparaît. Comme on a deux clauses de longueur 2 et une de longueur 1, on obtient 5 arêtes :



Q.13 Après avoir défini un graphe sans arête de bonne taille, on écrit une fonction auxiliaire `parcourt` qui prendre en argument une clause de longueur ≤ 2 (en éliminant les cas dégénérés évoqués par l'énoncé) et qui ajoute les arêtes associées au graphe. Il suffit de la faire agir sur chaque clause et de renvoyer le graphe.

```
let deux_sat_vers_graphe f=
   let g=make_vect (2*(1+(var_max f))) [] in
   let ajout_clause c=
      match c with
     |[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] -> g.(2*i+1) <- (2*j)::g.(2*i+1);
                  g.(2*j+1) <- (2*i)::g.(2*j+1)
     |[V i;NV j] -> g.(2*i+1) <- (2*j+1)::g.(2*i+1);
                   g.(2*j) <- (2*i)::g.(2*j)
     |[NV i;V j] -> g.(2*i) <- (2*j)::g.(2*i);
                   g.(2*j+1) <- (2*i+1)::g.(2*j+1)
     |[NV i;NV j] -> g.(2*i) <- (2*j+1)::g.(2*i);
                    g.(2*j) <- (2*i+1)::g.(2*j)
   in
   do_list ajout_clause f;
   g;;
```

Q.14 Les sommets de G correspondent à des littéraux. Dans la suite, on utilise sans le préciser cette identification.
Soient v_i et v_j deux sommets situés dans une même composante. Il existe donc un chemin $\ell_0, \ldots, \ell_p$ dans G avec $\ell_0 = v_i$ et $\ell_p = v_j$, les ℓ_k étant des littéraux. Comme le graphe possède les arêtes (ℓ_k, ℓ_{k+1}), la formule contient l'une des clauses $\neg\ell_k \vee \ell_{k+1}$ ou $\neg\ell_{k+1} \vee \ell_k$. L'une de ces clauses est ainsi vérifiée par σ ce qui indique (avec les remarques de l'énoncé) que $\ell_k \Rightarrow \ell_{k+1}$ l'est aussi pour tout k. Ceci étant vrai pour tout k, il en résulte que $\ell_0 \Rightarrow \ell_p$ est aussi vérifiée par σ. Comme il existe aussi un chemin de ℓ_p vers ℓ_0, on en déduit que $\ell_p \Rightarrow \ell_0$ est aussi vérifiée par σ. Finalement, ℓ_p et ℓ_0 ont même valeur de vérité.
Les variables associées à v_i et v_j ont donc même valeur de vérité si $j-i$ est pair (car les littéraux ℓ_0 et ℓ_p ont alors même signe) et des valeurs de vérité opposées sinon. C'est ce qu'il convenait de montrer.

Q.15 Deux littéraux dans la même composante ayant même valeur de vérité par une valuation qui satisfait f, une même composant du graphe ne peut contenir un littéral et son opposé dans le cas satisfiable.

Q.16 Pour tester si la formule 2-SAT f est satisfiable, on forme (en temps linéaire en la taille de f) le graphe associé puis (en temps linéaire en la taille du graphe et donc aussi linéaire en la taille de f) la liste des composantes fortement connexes de g. On doit alors vérifier qu'aucune de ces composantes ne contient deux éléments du type $2i$ et $2i+1$. Pour conserver la complexité linéaire, il faut faire cela efficacement, en n'effectuant qu'un seul passage dans chaque liste représentant une composante. Ces listes n'étant pas triées, cela ne me semble pas immédiat. Je propose alors de commence par introduire un tableau t de taille égale au nombre de sommets. Par un unique parcours de chaque liste, on peut remplir le tableau t en sorte que $t.(i)$ contienne le numéro d'un sommet référence de sa composante (par exemple le premier élément de la liste obtenue).
La fonction `remplir` prend en argument une liste correspondant à une composante et remplit le tableau t comme indiqué. Il reste à l'utiliser sur toutes les composantes.
La boucle finale effectue la vérification sur t expliquée plus haut.

```
let deux_sat f =
   let g=deux_sat_vers_graphe f in
   let cf=cfc g in
   let n=vect_length g in
   let t=make_vect n 0 in
   let remplir_t l = do_list (fun j -> t.(j)<-hd l) l in
   do_list remplir_t cf;
   let i=ref 0 in
   while !i<n/2 && t.(2* !i)<>t.(2* !i+1) do i:= !i+1 done;
   !i=n/2;;
```

3 Résolution de k-SAT

Q.17 Les fonctions sont élémentaires. On essaye de limiter le nombre des tests.

```
let et a b =
    if b=Faux||a=Faux then Faux
    else if a=Vrai&&b=Vrai then Vrai
    else Indetermine;;

let ou a b =
    if a=Vrai||b=Vrai then Vrai
```

```
    else if a=Faux&&b=Faux then Faux
    else Indetermine;;

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

Q.18 Une première fonction `evalclause` effectue l'évaluation d'une clause (la fonction est locale et a donc accès au tableau `t` argument de `eval`). On peut conclure quand on trouve un littéral valant `Vrai` ou `Indetermine` et continuer l'exploration sinon. On vérifie la cohérence du choix de valeur pour une clause vide en regardant ce qui se passe pour une clause de longueur 1. Il s'agit ensuite, dans la fonction `evalformule`, d'évaluer les clauses successives. On peut s'arrêter quand l'une vaut `Faux`. Sinon, tout dépend de la valeur de la clause et de celle du reste de la formule. On vérifie de même la cohérence du choix de valeur pour une formule vide.

```
let eval f t =
   let rec evalclause c =
      match c with
     |[] -> Faux
     |(V i)::q -> if t.(i)=Vrai then Vrai
                  else if t.(i)=Indetermine then Indetermine
                  else evalclause q
     |(NV i)::q -> if t.(i)=Faux then Vrai
                   else if t.(i)=Indetermine then Indetermine
                   else evalclause q
   in
   let rec evalformule f=
      match f with
     |[] -> Vrai
     |c::q -> let t=evalclause c in
              if t=Faux then Faux
              else begin
                  let tt=evalformule q in
                  if tt=Faux then Faux
                  else if tt=Vrai then t
                  else Indetermine
                  end
   in evalformule f;;
```

Q.19 Notre fonction principale va gérer une fonction auxiliaire `explore` prenant en argument un entier $k \in [|0, n|]$ correspondant au nombre de variables ayant reçu une valeur booléenne. Cette fonction doit renvoyer un booléen indiquant si cette valuation triléenne peut être complétée en une valuation booléenne qui satisfait notre formule. Elle est donc de type `int` $\rightarrow$ `bool`. Son schéma est le suivant

```
On calcule l'\'evaluation tril\'eenne de la formule.
Si elle vaut Vrai alors, on renvoie True
Si elle vaut Faux alors, on renvoie False
Sinon
     on pose t.(k)=Vrai et on fait un appel r\'ecursif avec k+1
     Si l'appel renvoie True, on renvoie True
```

```
     Sinon
         on pose t.(k)=Faux et on fait un appel r\'ecursif avec k+1
         Si l'appel renvoie True, on renvoie True
         Sinon on pose t.(k)=Indetermine (*remont\'ee dans l'arbre*) et on renvoie False
```

La fonction `var_max` permet de connaître la taille du tableau à utiliser. La suite est la traduction Caml de l'algorithme de l'énoncé.

```
let k_sat f =
    let n=var_max f in
    let t=make_vect (n+1) Indetermine in
    let rec explore k =
        let b=eval f t in
        if b=Vrai then true
        else if b=Faux then false
        else begin
             t.(k) <- Vrai ;
             if explore (k+1) then true
             else begin
                  t.(k) <- Faux ;
                  if explore (k+1) then true
                  else begin
                       t.(k) <- Indetermine ;
                       false
                       end
                  end
             end
     in explore 0;;
```

On pourrait plus efficacement utiliser une exception afin d'arrêter le processus complet quand on trouve une évaluation (partielle) convenable (rendant vraie la formule). La fonction récurrente auxiliaire est alors `explore : int → unit`. Dans l'appel `explore k`, on suppose avoir déjà donné une valeur aux variables de numéro $\leq k-1$ et on teste toutes les façons de compléter cette évaluation partielle en suivant le processus décrit par l'énoncé. On lève une exception `succes` dès que l'on trouve une évaluation partielle convenable. Il suffit donc de lancer le processus avec $k = 0$; si l'exception est levée, c'est que l'on a trouvé une valuation (partielle) convenable. Sinon, toutes les valuations ont échoué.

```
exception succes;;

let k_sat f =
    let n=var_max f in
    let t=make_vect (n+1) Indetermine in
    let rec explore k =
       let b=eval f t in
       if b=Vrai then raise succes
       else if b=Indetermine then begin
             t.(k) <- Vrai ;
             explore (k+1) ;
             t.(k) <- Faux ;
             explore (k+1) ;
             t.(k) <- Indetermine
```

```
        end
  in try explore 0 ; false
     with _ -> true;;
```

4 De k-SAT à SAT

Q.20 a. $(x_1 \vee \neg x_0) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_2)$.

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

Q.21 La première étape conserve la satisfiabilité puisquon n'utilise que les lois de De Morgan. La seconde étape travaille avec des formules de l'ensemble $\mathcal{F}'$ défini récursivement à partir des littéraux (cas de base) en utilisant uniquement les connecteurs $\vee$ et $\wedge$. Il s'agit de montrer que pour toute formule f de $\mathcal{F}'$, l'étape b conserve la satisfiabilité (c'est à dire que f et f' sont équisatisfiables). Plus précisément, on montre que pour $f \in \mathcal{F}'$,

- si f est satisfaite par une valuation σ alors il existe une valuation σ' satisfaisant f' et telle que σ et σ' ont même action sur les variables présentes dans f ;
- si f' est satisfaite par une valuation σ' alors σ' satisfait aussi f.

On procède par induction structurelle.

- Le résultat est immédiat dans le cas de base car on ne modifie alors pas la formule.
- Soient ϕ et ψ deux éléments de $\mathcal{F}'$ vérifiant l'hypothèse.
  Si $f = \phi \wedge \psi$ alors $f' = \phi' \wedge \psi'$.
  - Supposons f satisfiable par σ ; ϕ et ψ le sont alors aussi. On peut alors trouver des valuations σ_1 et σ_2 satisfaisant ϕ' et ψ'. De plus, σ_1 et σ_2 ont la même action que σ sur les variables de f. Les variables supplémentaires dans ϕ' et dans ψ' ne sont pas les mêmes (ce n'est pas clair dans l'énoncé de la méthode mais c'est le cas si on en juge par l'exemple donné dans l'énoncé). On peut donc poser σ' qui agit comme σ sur le svariables de f, comme σ_1 sur les variables supplémentaires de ϕ' et comme σ_2 sur les variables supplémentaires de ψ'. σ' satisfait f' et a la même action que σ sur les variables de f.
  - Supposons que f' est satisfiable. Il existe alors une valuation σ' qui satisfait ϕ' et ψ'. Elle satisfait alors ϕ et ψ et donc aussi f.

  Si $f = \phi \vee \psi$, on note x la variable supplémentaire introduite et qui n'est présente ni dans ϕ' ni dans ψ'.
  - Si f est satisfaite par σ alors σ satisfait ϕ ou ψ. Les cas étant symétriques, supposons qu'elle satisfait ϕ. En considérant σ' qui satisfait ϕ' et en imposant $\sigma'(x) = 0$ (ce qui est possible car pour satisfaire ϕ', la valeur de x n'importe pas), on trouve une valuation qui satisfait f'.
  - Si f' est satisfaite par une valuation σ'. Si $\sigma(x) = 0$, alors σ' satisfait ϕ' et sinon elle satisfait ψ'. Dans le premier cas, elle satisfait ϕ et donc f. Dans le second, elle satisfait ψ et donc aussi f.

Q.22 C'est un processus récurrent simple. Il faut juste être attentif à ne pas oublier de cas.

```
let rec negs_en_bas f =
   match f with
  |Var i -> Var i
  |Et(a,b) -> Et(negs_en_bas a,negs_en_bas b)
```

```
|Ou(a,b) -> Ou(negs_en_bas a,negs_en_bas b)
|Non(Var i) -> Non (Var i)
|Non(Et(a,b)) -> Ou(negs_en_bas(Non a),negs_en_bas(Non b))
|Non(Ou(a,b)) -> Et(negs_en_bas(Non a),negs_en_bas(Non b))
|Non(Non a) -> negs_en_bas a;;
```

Q.23 On applique l'algorithme assez naïvement sans essayer de réduire le nombre de concaténations. On gère une référence k indiquant le numéro de la dernière variable utilisée (afin de savoir quel numéro donner à une nouvelle variable à introduire).

```
let formule_vers_fnc f=
    let n=var_max f in
    let k=ref n in
    let rec construit_fnc f=
       match f with
      |Var i -> [[V i]]
      |Non(Var i) -> [[NV i]]
      |Et(a,b) -> (construit_fnc a)@(construit_fnc b)
      |Ou(a,b) -> incr k;
                  let v= !k in
                  let fcb=map (fun l -> (NV v)::l) (construit_fnc b) in
                  let fca=map (fun l -> (V v)::l) (construit_fnc a) in
                  fca@fcb
    in construit_fnc f;;
```

Q.24 La taille d'une formule logique est par définition le nombre de noeuds qui composent l'arbre d'expression associé. On montre par récurrence que le nombre total C_p d'appels à la fonction `negs_en_bas` lors de l'appel initial `negs_en_bas f` quand f est de taille p est plus petit $2p-1$.

- Il y a un unique appel dans le cas d'un littéral et l'arbre est alors de taille 1 ou 2. Le résultat est donc vrai dans ce cas.
- Supposons le résultat vrai pour des formules a et b de tailles p et q.
  $f = a \wedge b$ est de taille $p+q+1$ et l'appel avec f induit un nombre d'appels égal à $1 + C_p + C_q \leq 2(p+q) - 1 \leq 2(p+q+1) - 1$.
  C'est la même chose pour $f = a \vee b$.
  $f = \neg(a \wedge b)$ est de taille $p+q+2$ et l'appel avec f induit un nombre d'appels égal à $1 + C_{p+1} + C_{q+1} \leq 2(p+q) + 3 = 2(p+q+2) - 1$.
  C'est la même chose pour $f = \neg(a \vee b)$.
  $f = \neg(\neg a)$ est de taille $p+2$ et l'appel avec f induit un nombre d'appels égal à $C_p + 1 \leq 2p \leq 2(p+2) - 1$.

Comme il y a un nombre constant d'opération à chaque appel (en dehors des appels récursifs), `negs_en_bas` est donc de complexité linéaire en la taille de f.
Montrons maintenant que le nombre de clauses formées dans l'appel `formule_vers_fnc f` est égal au nombre de littéraux de f (f étant supposée obtenue à partir de l'étape 1 et donc avec des $\neg$ intériorisés).

- C'est immédiat dans le cas où la formule est un littéral.
- Supposons le résultat vrai pour des formules ϕ et ψ. Cela reste immédiatement vrai pour $\phi \wedge \psi$ et $\phi \vee \psi$.

Dans l'algorithme utilisé, les clause construites le sont à partir des littéraux de la formule initiale en ajoutant éventuellement des littéraux associés à de nouvelles variables. Si on note n le nombre

de connecteurs $\vee$ présents dans la formule, il y aura exactement n nouvelles variables ajoutées. Le nombre de littéraux étant égal au nombre de connecteurs $\vee$ ou $\wedge$ plus 1 (récurrence simple), on a $2n-1$ qui est inférieur à la taille de f. Les clause ont donc une longueur inférieure ou égale à la taille de f (et même à 1 plus la moitié de la taille de f moins 1).
En notant p la taille de f, on construit donc un nombre de clauses inférieur à p et chaque clause à un longueur plus petite que p.
Pour construire une clause, on procède par consage ou concaténation. Le cas le pire est celui où on concatène une grosse liste sur une petite (le coût est celui de la clause de gauche de la concaténation). Le coû maximal d'une construction de clause est alors $1 + 2 + \cdots + p$ (et correspond au cas où on ajoute des éléments par la droite, c'est à dire où on concatène à droite sur une liste de taille 1). Construire une clause a donc un coût $O(p)$.
La complexité de `formule_vers_fnc` est donc au pire cubique en la taille de la formule argument.