Satisfiabilité des formules booléennes

X-ENS 2016

24 avril 2016

Préliminaires

Question 1

1. $x_1 \wedge (x_0 \vee \neg x_0) \wedge \neg x_1$ n'est pas satisfiable à cause de $x_1 \wedge \neg x_1$.
2. $(x_0 \vee \neg x_1) \wedge (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_0)$ est satisfiable par $\{\texttt{Vrai}, \texttt{Vrai}, \texttt{Vrai}\}$
3. $x_0 \vee \neg(x_0 \vee \neg(x_1 \vee \neg(x_1 \vee \neg x_2)))$ est satisfiable par $\{\texttt{Vrai}, \texttt{Vrai}, \texttt{Vrai}\}$
4. $(x_0 \vee x_1) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee \neg x_1) \wedge (\neg x_0 \vee \neg x_1)$ est équivalente à $(x_0 \wedge \neg x_0) \vee (x_1 \wedge \neg x_1)$ donc n'est pas satisfiable.

Question 2

On décompose en étapes : littéral, clause puis formule.

```
let var = function V k -> k| NV k -> k;;

let rec var_max_c c =
  match c with
  |[] -> failwith "clause vide"
  |[x] -> var x
  |t::q -> max (var t) (var_max_c q);;

let rec var_max fn =
  match fn with
  |[] -> failwith "forme vide"
  |[x] -> var_max_clause x
  |t::q -> max (var_max_c t) (var_max q);;
```

En utilisant un maximum prédéfini pour les clauses vides on peut utiliser les fonctionnelles rappelées dans l'énoncé.

```
let var = function V k -> k| NV k -> k;;

let var_max_c c =
  list_it (fun k -> max (var k)) c min_int;;

let var_max fn =
  list_it (fun c -> max (var_max_c c)) fn min_int;;
```

Partie I. Résolution de 1-SAT

Question 3

```
let un_sat fn =
  let n = var_max fn in
  let tab = make_vect (n+1) Indetermine in
  let rec aux liste =
    match liste with
    |[] -> true
    |[V k]::q -> if tab.(k) = Faux
                 then false
                 else (tab.(k) <- Vrai; aux q)
    |[NV k]::q -> if tab.(k) = Vrai
                  then false
                  else (tab.(k) <- Faux; aux q)
    |_ -> failwith "Ce n'est pas une 1-formule" in
  aux fn;;
```

Partie II. Résolution de 2-SAT

II.1 Composantes fortement connexes

Question 4

On appelle la fonction `dfs_rec` pour chaque sommet puis pour toutes les extrémités des arêtes depuis un sommet non encore vu. Comme le nombre d'instructions en dehors de l'appel récursif est borné on obtient bien une complexité linéaire en $|V| + |E|$.

Question 5

```
let renverser_graphe g =
  let n = vect_length g in
  let g_renv = make_vect n [] in
  for i = 0 to (n-1) do
    do_list (fun k -> g_renv.(k) <- i :: g_renv.(k)) g.(i) done;
  g_renv;;
```

Question 6

On commence par une fonction qui crée la composante depuis un sommet. Ce sommet de doit pas avoir déjà été vu.

```
let composante g deja_vu sommet =
  let rec aux_comp liste composante =
    match liste with
    |[] -> composante
    |t::q when deja_vu.(t) -> aux_comp q composante
    |t::q -> deja_vu.(t) <- true;
             let comp = aux_comp g.(t) (t::composante) in
             aux_comp q comp in
  deja_vu.(sommet) <- true;
  aux_comp g.(sommet) [sommet];;
```

On parcourt ensuite la liste des sommets en ajoutant les composantes à la liste.

```
1  let dfs_cfc g liste =
2    let n = vect_length g in
3    let deja_vu = make_vect n false in
4    let rec aux_cc a_voir =
5      match a_voir with
6      |[] -> []
7      |t::q when deja_vu.(t) -> aux_cc q
8      |t::q -> let cc = composante g deja_vu t
9               in cc::aux_cc q in
10   aux_cc liste;;
```

Aux lignes 8 et 9 on doit calculer la composante avant de poursuivre en raison de la modification sur la liste `a_voir` (effet de bord).

Question 7

Il suffit de tout assembler.

```
let cfc g =
  let gr = renverser_graphe g in
  let liste = dfs_tri g in
  dfs_cfc gr liste;;
```

Question 8

C, C' et C'' désignent des composantes fortement connexes.

- La relation est réflexive car, pour $v \in C$, $s = (v)$ est un chemin de v à v.
- On suppose que C est subordonnée à C' et C' est subordonnée à C.
  v et v' sont des sommets appartenant respectivement à C et C'.
  D'après la remarque qui suit la définition il existe un chemin de v vers v' car C est subordonnée à C' et un chemin de v' vers v car C' est subordonnée à C.
  Ainsi v et v' appartiennent à la même composante fortement connexe : $v' \in C$ et $v \in C'$.
  On a donc $C' \subset C$ et $C \subset C'$ d'où $C = C'$: la relation est antisymétrique.
- On suppose que C est subordonnée à C' et C' est subordonnée à C''.
  Il existe des sommets $v \in C$ et $v' \in C'$ et un chemin de v à v'
  et des sommets $w' \in C'$ et $w'' \in C''$ et un chemin de w' à w''.
  Comme C' est fortement connexe il existe un chemin de v' à w'.
  En composant ces chemins on obtient un chemin de v vers w'' donc C est subordonnée à C'' : la relation est transitive.

Question 9

La question telle que posée est fausse. Dans l'exemple il existe un chemin de v_4 vers v_3 et il n'en existe pas de v_3 vers v_4 mais on a $t_4 < t_3$.
On va transformer la question en prouvant que s'il existe une arête de v_i vers v_j mais s'il n'existe pas de chemin de v_j vers v_i alors $t_i > t_j$.

- Si v_i est marqué comme vu avant v_j alors `dfs_rec i` invoque `do_list dfs_rec g.(i)` qui appelle, entre autres, `dfs_rec j`. Ainsi `dfs_rec j` est terminé avant `dfs_rec i` donc $t_j < t_i$.
- Si v_j est marqué comme vu avant v_i alors `dfs_rec j` est invoqué avant `dfs_rec i`. Comme v_i n'est pas visible depuis v_j alors `dfs_rec j` se termine avant que l'on puisse invoquer `dfs_rec i` donc, ici encore, $t_j < t_i$.

Le problème dans la question initiale est dans le premier des cas ci-dessus. Si le chemin de v_i vers v_j passe par un sommet v qui est en attente alors, lorsqu'on arrive à v celui-ci est déjà vu donc on ne peut pas poursuivre pour arriver à v_j.

Question 10

On procède en plusieurs étapes.

1. C est une composante fortement connexe : on commence par prouver que si v_i est le premier sommet de C visité lors de `dfs_tri` alors $t_C = t_i$.
   En effet supposons, par l'absurde, qu'un sommet v_j de C n'est pas visité lors de l'appel de `dfs_rec i`. Il existe un chemin de v_i vers v_j dans C :

   $$v_{i_0} \to v_{i_1} \to v_{i_2} \to \cdots \to v_{i_{p-2}} \to v_{i_{p-1}} \to v_{i_p} \text{ avec } i_0 = i \text{ et } i_p = j$$

   Aucun de ces sommet n'a été visité au début de l'appel de `dfs_rec i`.
   Si on considère le premier sommet, v_{i_k}, dans ce chemin qui ne soit pas visité : on aboutit à une contradiction car $v_{i_{k-1}}$ est visité et l'appel de `dfs_rec` $v_{i_{k-1}}$ visitera v_{i_k}.
   Ainsi tous les sommets de C sont visités donc leur placement dans la liste `resultat` se fera avant la fin de `dfs_tri i` d'où $t_j < t_i$ pour tout $t_j \in C$.
2. C et C' sont deux composantes fortement connexes distinctes.
   On suppose qu'il existe une arête entre un sommet de C, v_i, et un sommet de C', v_j.
   On note v_k le premier sommet de C' visité lors de `dfs_tri`.
   - Si v_i est visité avant v_k lors de `dfs_tri` alors, comme v_j est visité lors de `dfs_rec i`, v_k est visité aussi lors de ce parcours donc $t_k < t_i$.
     Comme on a $t_{C'} = t_k$ et $t_C \geq t_i$ on en déduit que $t_{C'} < t_C$.
   - Si v_k est visité avant v_i alors, lors de l'appel de `dfs_rec k`, v_i n'est pas visité donc $t_{C'} = t_k < t_i \leq t_C$.

   Dans les deux cas on a $t_{C'} < t_C$.
3. On suppose C subordonnée à C' avec $C \neq C'$.
   Il existe un chemin de longueur au moins 1 entre un sommet de C' et un sommet de C.
   On va prouver par récurrence sur la longueur du chemin que $t_C < t_{C'}$.
   - Pour un longueur de longueur 1 c'est la propriété que l'on vient de prouver.
   - On suppose que la propriété est vraie lorsque les chemins sont de longueur $n \geq 1$.
     Si le chemin $v' \to v_1 \to v_2 \to \cdots \to v_n \to v$ est de longueur $n+1$ on considère la composante fortement connexe contenant v_n : C''.
     C'' est subordonnée à C' avec un chemin de longueur n donc, d 'après l'hypothèse de récurrence, $t_{C''} < t_{C'}$.
     Si $C'' = C$ on a immédiatement $t_C = t_{C''} < t_{C'}$.
     Si $C'' \neq C$ on a une arête entre un sommet de C'' et un sommet de C donc on peut appliquer le résultat ci-dessus et $t_C < t_{C''} < t_{C'}$.
   - On a bien $t_C < t_{C'}$ dans tous les cas de chemin.

Question 11

On montre qu'à chaque étape on construit une composante fortement connexe.
On suppose qu'on construit k composantes fortement connexes et qu'il reste des sommets ; on note

1. A l'ensemble des sommets non vus,
2. v l'élément de A d'instant maximum t,
3. C la composante fortement connexe contenant v.

L'algorithme va parcourir les sommets de A à partir de v pour donner un ensemble A'.
On veut montrer que $A' = C$.

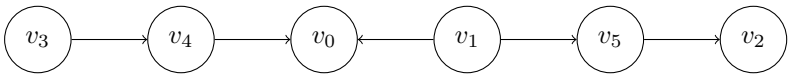
1. Si $v' \in C$ alors v' n'appartient pas aux autres composantes fortement connexes donc $v' \in A$.
   On en déduit que $t_C = t$.
2. Si $v' \in C$ alors il existe un chemin dans C de v' vers v donc un chemin de sommets non vus de v vers v' dans le graphe inverse dont les sommets sont dans A.
   On en déduit que v' est visité, $v' \in A'$, donc $C \subset A'$.

3. Si $v' \in A \setminus C$ alors v' appartient à une autre composante fortement connexe C'. On a $C' \subset A$ et v n'appartient pas à C' dont $T_{C'} < t = t_C$.
   Si v' appartenait à A' on aurait un chemin de v' vers v donc C serait subordonnée à C'.
   La question 10 impliquerait $t_C < t_{C'}$ ce qui est contradictoire.
   Ainsi les éléments de A' appartiennent à C : $A' \subset C$.

On a bien $C = A'$ donc l'algorithme produit bien une composante fortement connexe à chaque étape.

II.2 Des composantes fortement connexes à 2-SAT

Question 12



Question 13

On traduit l'énoncé

```
let deux_sat_vers_graphe fn =
  let n = var_max fn in
  let g = make_vect (2*n+2) [] in
  let rec aux liste =
    match liste with
    |[] -> ()
    |[V i]::q -> g.(2*i+1) <- (2*i) :: g.(2*i+1);
                 aux q
    |[NV i]::q -> g.(2*i) <- (2*i+1) :: g.(2*i);
                  aux q
    |[V i; V j]::q -> g.(2*i+1) <- (2*j) :: g.(2*i+1);
                      g.(2*j+1) <- (2*i) :: g.(2*j+1);
                      aux q
    |[V i; NV j]::q -> g.(2*i+1) <- (2*j+1) :: g.(2*i+1);
                       g.(2*j) <- (2*i) :: g.(2*j);
                       aux q
    |[NV i; V j]::q -> g.(2*i) <- (2*j) :: g.(2*i);
                       g.(2*j+1) <- (2*i+1) :: g.(2*j+1);
                       aux q
    |[NV i; NV j]::q -> g.(2*i) <- (2*j+1) :: g.(2*i);
                        g.(2*j) <- (2*i+1) :: g.(2*j);
                        aux q
    |_ -> failwith "Ce n'est pas une 2-formule" in
  aux fn;
  g;;
```

Question 14

On considère que les noms des sommets, v_i, sont des littéraux : $l_{2i} = x_i$ et $l_{2i+1} = \neg x_i$.
Par construction l'arête $v_i \to v_j$ signifie $\neg v_i \vee v_j$.
Si f est satisfiable par σ et si le graphe contient $v_i \to v_j$ alors $\neg\sigma(v_i) \vee \sigma(v_j) =$ `Vrai`.
Si on a un chemin $v_i \to v_j \to v_k$ alors $\neg\sigma(v_i) \vee \sigma(v_j)$ et $\neg\sigma(v_j) \vee \sigma(v_k)$ sont vraies.
Dans ce cas, si $\sigma(v_j)$ est vrai alors $\sigma(v_k)$ est vrai et si $\sigma(v_j)$ est faux alors $\neg\sigma(v_i)$ est vrai : on a donc forcément $\neg\sigma(v_i) \vee \sigma(v_k)$ qui est vraie.
En assemblant ainsi les arêtes on voit que s'il existe un chemin de v_i à v_j,

$$v_i \to v_{i_1} \to v_{i_2} \to \cdots \to v_{i_{p-2}} \to v_{i_{p-1}} \to v_j$$

dans le graphe associé à une formule satisfiable alors $\neg\sigma(v_i) \vee \sigma(v_j) =$ `Vrai`.

Si v_i et v_j appartiennent à la même composante fortement connexe alors il existe un chemin de v_i à v_j et un chemin de v_j à v_i donc $\neg\sigma(v_i) \vee \sigma(v_j)$ et $\neg\sigma(v_j) \vee \sigma(v_i)$ sont vraies.
Cela signifie que $\sigma(v_i) \Leftrightarrow \sigma(v_j) =$ `Vrai` donc $\sigma(v_i) = \sigma(v_j)$.
Si $i - j$ est pair alors i et j ont la même parité donc v_i et v_j représente tous deux une variable ou la négation d'une variable : les deux variables doivent donc avoir la même valuation.
De même si $i - j$ est impair les deux variables représentées doivent donc avoir des valuations opposées.

Question 15

En particulier si v_{2i} et v_{2i+1} appartenaient à la même composante fortement connexe alors une éventuelle valuation qui satisferait la formule devrait avoir des valeurs opposées pour x_i et x_i ce qui est impossible.
Si une formule est satisfiable il n'existe pas de variable x_i telle que v_{2i} et v_{2i+1} appartiennent à la même composante fortement connexe.

Question 16

Après avoir calculé les composantes fortement connexes on va construire un tableau qui attribue à chaque sommet le numéro de sa composante (le numéro est arbitraire). La complexité est linéaire.

```
let tableau_composantes liste_cfc taille =
  let tab = make_vect taille 0 in
  let rec aux liste ind_cfc =
    match liste with
    |[] -> ()
    |t::q -> do_list (fun k -> tab.(k) <- ind_cfc) t; aux q (
       ind_cfc + 1) in
  aux liste_cfc 1;
  tab;;
```

On passe la taille en paramètre car elle se calcule facilement à partir du graphe.

Il suffit alors de teste l'égalité des termes d'indices $2i$ et $2i + 1$.

```
let deux_sat f =
  let g = deux_sat_vers_graphe f in
  let liste_cfc = cfc g in
  let n = vect_length g in
  let tab = tableau_composantes liste_cfc n in
  let resultat = ref true in
  for i = 0 to (n/2 - 1) do
    if tab.(2*i) = tab.(2*i+1) then resultat := false done;
  !resultat;;
```

Partie III. Résolution de k-SAT

Question 17

```
let et p q =
  match p,q with
  |Faux, _ -> Faux
  |_, Faux -> Faux
  |Indetermine, _ -> Indetermine
  |_, Indetermine -> Indetermine
  |Vrai, Vrai -> Vrai;;

let ou p q =
  match p,q with
  |Vrai, _ -> Vrai
  |_, Vrai -> Vrai
  |Indetermine, _ -> Indetermine
  |_, Indetermine -> Indetermine
  |Faux, Faux -> Faux;;

let non p =
  match p with
  |Vrai -> Faux
  |Indetermine -> Indetermine
  |Faux -> Vrai;;
```

Question 18

On travaille pas-à-pas.

```
let eval_litt l valuation =
  match l with
  |V k -> valuation.(k)
  |NV k -> non valuation.(k);;

let rec eval_clause c valuation =
  match c with
  |[] -> Faux
  |t::q when eval_litt t valuation = Vrai -> Vrai
  |t::q -> ou (eval_litt t valuation) (eval_clause q valuation);;

let rec eval f valuation =
  match f with
  |[] -> Vrai
  |t::q when eval_clause t valuation = Faux -> Faux
  |t::q -> et (eval_clause t valuation) (eval q valuation);;
```

Question 19

On a besoin de créer une fonction récursive qui teste en changeant les valeurs de x_i.
Mais on veut sortir en cas de vérité, pour cela on peut utiliser une exception qui sera rattrapée.

```
1  exception CaMarche;;
2
3  let k_sat f =
4    let n = var_max fn in
5    let val = make_vect (n+1) Indetermine in
6    let rec explore i =
7      match eval f val with
8      |Vrai -> raise CaMarche
9      |Faux -> ()
10     |Indetermine -> if i = (n+1)
11                     then ()
12                     else (val.(i) <- Vrai;
13                           explore (i+1);
14                           val.(i) <- Faux;
15                           explore (i+1);
16                           val.(i) <- Indetermine) in
17   try (explore 0; false) with CaMarche -> true;;
```

La fonction auxiliaire de la ligne 6 correspond à une valuation où i valeurs sont données.
On commence par évaluer la formule, si elle est vraie on sort des boucles par une exception (ligne 8).
Si elle est fausse on arête l'évaluation (ligne 9).
Si elle est indéterminée et qu'il reste des valeurs à donner à la valuation (ligne 12), on ajoute les valeurs `Vrai` puis `Faux` à l'indice i et on descend dans l'évaluation (lignes 13 et 15).
On re-neutralise la valuation en i (ligne 16) avant de finir l'exploration à ce niveau.

Partie IV. De k-SAT à SAT

Question 20

- $(x_1 \vee \neg x_0) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_2)$.
- $(x_0 \vee x_2 \vee x_4) \wedge (x_0 \vee x_2 \vee x_5) \wedge (x_0 \vee x_3 \vee x_4) \wedge (x_0 \vee x_3 \vee x_5) \wedge \cdots$
  $\cdots \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee x_5) \wedge (x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_3 \vee x_5)$.

Question 21

Les lois de Morgan conservent les valuations donc f et f^* sont équi-satisfiables.

La deuxième étape doit être précisée : à chaque application de la seconde règle on introduit une variable qui doit être distincte à chaque étape du procédé et distincte de toutes les variables de la formule initiale.
Pour formaliser on note $\mathrm{var}(f)$ l'ensemble des variables intervenant dans f.
Chaque fois que l'on transforme deux formules ϕ^* et ψ^* on aboutit à deux formules ϕ' et ψ' telles que $\mathrm{var}(\phi') = \mathrm{var}(\phi^*) \cup A$ et $\mathrm{var}(\psi') = \mathrm{var}(\psi^*) \cup B$ avec

$$\big(\mathrm{var}(\phi^*) \cup \mathrm{var}(\psi^*)\big) \cap A = \emptyset \quad \big(\mathrm{var}(\phi^*) \cup \mathrm{var}(\psi^*)\big) \cap B = \emptyset \quad A \cap B = \emptyset$$

On va aussi préciser la notion d'équi-satisfiabilité : si f^* et f' sont satisfiables elles le sont par des valuations σ^* et σ' telles que σ' prolonge σ^* et seulement sur les nouvelles variables de f'.

On peut alors procéder par induction structurelle.

- Si $f^* = \phi^* \wedge \psi^*$ est satisfiable alors il existe une valuation σ^* telle que $\sigma^*(\phi^* \wedge \psi^*) = \texttt{Vrai}$ donc $\sigma^*(\phi^*) = \sigma^*(\psi^*) = \texttt{Vrai}$.
  L'équi-satisfiabilité de ϕ^* et ϕ' implique qu'il existe une valuation σ'_1 qui prolonge σ^* telle que $\sigma'_1(\phi')$ soit vraie. De même il existe une valuation σ'_2 qui prolonge σ^* telle que $\sigma'_1(\psi')$ soit vraie.
  σ'_1 et σ'_2 sont égales sur l'intersection de leurs domaines de définitions donc elles permettent de définir une nouvelle valuation σ' telle que $\sigma'(\phi') = \sigma'(\psi') = \texttt{Vrai}$.
  On a alors $\sigma'(\phi' \wedge \psi') = \texttt{Vrai}$ avec σ' qui prolonge σ^* sur les nouvelles variables.
- Si $f' = \phi' \wedge \psi'$ est satisfiable alors il existe une valuation σ' telle que $\sigma'(\phi' \wedge \psi') = \texttt{Vrai}$ donc $\sigma'(\phi') = \sigma'(\psi') = \texttt{Vrai}$.
  L'équi-satisfiabilité de ϕ^* et ϕ' implique que, si σ_1^* est la restriction de σ' en enlevant les nouvelles variables de ϕ', alors $\sigma_1^*(\phi^*) = \texttt{Vrai}$. Comme les nouvelles variables de ψ' on peut restreindre σ_1^* en "oubliant" ces variables en une valuation σ^* telle que $\sigma^*(\phi^*) = \texttt{Vrai}$.
  De même $\sigma^*(\psi^*) = \texttt{Vrai}$ donc $\sigma^*(f^*) = \sigma^*(\phi^* \wedge \psi^*) = \texttt{Vrai}$.
- On a donc prouvé que f^* et f' sont équi-satisfiables dans le cas du $\wedge$.
- Si $f^* = \phi^* \vee \psi^*$ est satisfiable il existe une valuation σ^* telle que $\sigma^*(\phi^* \vee \psi^*) = \texttt{Vrai}$ donc $\sigma^*(\phi^*) = \texttt{Vrai}$ ou $\sigma^*(\psi^*) = \texttt{Vrai}$. On suppose, par exemple, que $\sigma^*(\phi^*) = \texttt{Vrai}$.
  Il existe un prolongement σ'_1 tel que $\sigma'_1(\phi') = \texttt{Vrai}$. On prolonge σ'_1 en σ' en posant $\sigma'(x) = \texttt{Faux}$ et en prenant des valeurs quelconques pour les nouvelles variables de ψ'.
  Si $\phi' = \bigwedge_{i=1}^{p} \phi'_i$ et $\psi' = \bigwedge_{i=1}^{q} \psi'_i$ on a $\sigma'(\phi'_i) = \sigma^*(\phi'_i) = \texttt{Vrai}$ donc $\sigma'(\phi'_i \vee x) = \texttt{Vrai}$.
  On a aussi $\sigma'(\neg x) = \texttt{Vrai}$ donc $\sigma'(\psi'_i \vee \neg x) = \texttt{Vrai}$ d'où
  $\sigma'(f') = \sigma'\left(\bigwedge_{i=1}^{p}(\phi'_i \vee x) \wedge \bigwedge_{i=1}^{q}(\psi'_i \vee \neg x)\right) = \texttt{Vrai}$.
- Si $f' = \bigwedge_{i=1}^{p}(\phi'_i \vee x) \wedge \bigwedge_{i=1}^{q}(\psi'_i \vee \neg x)$ est satisfiable alors il existe une valuation σ' telle que $\sigma'(\phi'_i \vee x) = \texttt{Vrai}$ pour tout $i \in \{1, 2, \ldots, p\}$ et $\sigma'(\psi'_i \vee \neg x) = \texttt{Vrai}$ pour tout $i \in \{1, 2, \ldots, q\}$.
  On suppose, par exemple, $\sigma'(x) = \texttt{Vrai}$, on doit alors avoir $\sigma'(\psi'_i) = \texttt{Vrai}$ pour tout i.
  On en déduit que $\sigma'(\psi') = \sigma'\left(\bigwedge_{i=1}^{q} \psi'_i\right) = \texttt{Vrai}$. Par équi-satisfiabilité on peut restreindre σ' en σ^* telle que $\sigma^*(\psi^*) = \texttt{Vrai}$ donc $\sigma^*(f^*) = \sigma^*(\phi^* \vee \psi^*) = \texttt{Vrai}$.
- On a donc prouvé que f^* et f' sont équi-satisfiables dans le cas du $\vee$.

Question 22

```
let rec negs_en_bas f =
  match f with
  |Var i -> f
  |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;;
```

Question 23

```
let formule_vers_fnc f =
  let n = var_max f in
  let ind = ref (n+1) in
  let rec aux f =
    match f with
    |Var i -> [[V i]]
    |Non(Var i) -> [[NV i]]
    |Non a -> failwith "Ceci ne doit pas arriver"
    |Et(a,b) -> (aux a)@(aux b)
    |Ou(a,b) -> let aa = map (fun l -> (V !ind)::l) (aux a) in
                let bb = map (fun l -> (NV !ind)::l) (aux b) in
                ind := !ind + 1;
                aa@bb in
  aux f;;
```

Question 24

Si on note $C(f)$ la complexité du calcul de `negs_en_bas f` on a
$C(x) = 0$, $C(a \wedge b) = 1 + C(a) + C(b)$, $C(a \vee b) = 1 + C(a) + C(b)$, $C(\neg x) = 2$,
$C(\neg(a \wedge b)) = 1 + C(\neg a) + C(\neg b)$, $C(\neg(a \vee b)) = 1 + C(\neg a) + C(\neg b)$ et $C(\neg\neg a) = C(a)$.
On prouve par induction structurelle que $C(a) \leq 2T(a) - 1$ où $T(a)$ est la taille de a.
Par exemple le cas $\neg(a \wedge b)$ donne
$C(\neg(a \wedge b)) = 1 + C(\neg a) + C(\neg b) \leq 1 + 2(T(a) + 1) - 1 + 2(T(b) + 1) - 1 = 2(T(a) + T(b)) + 3$
Avec $2T(\neg(a \wedge b)) - 1 = 2(T(a) + T(b) + 2) - 1 = 2(T(a) + T(b)) + 3$.

Si on note $C'(f)$ la complexité du calcul de `formule_vers_fnc f` en nombre d'adjonctions à une liste.
On pose $|u|$ la longueur d'une liste et $\varphi(f)$ la liste `formule_vers_fnc f`.
On a $C'(x) = C'(\neg x) = 2$, $|\varphi(x)| = |\varphi(\neg x)| = 1$,
$C'(a \wedge b) = C'(a) + C'(b) + |\varphi(a)|$, $|\varphi(a \wedge b)| = |\varphi(a)| + |\varphi(b)|$
et $C'(a \vee b) = C'(a) + C'(b) + 2(|\varphi(a)| + |\varphi(b)|) + |\varphi(a)|$, $|\varphi(a \vee b)| = |\varphi(a)| + |\varphi(b)|$.
Le 2 provient du fait que la fonction `map` modifie chaque terme de la liste de listes par un ajout et adjoint cette nouvelle liste donc on a 2 fois la longueur.
On en déduit d'abord que $|\varphi(f)| \leq T(f)$ puis $C'f) \leq \frac{3}{2}\big(T(f)\big)^2$.
La complexité est polynomiale.