Mines-Ponts 2010. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Première partie : introduction.

1. $a) \equiv x \vee y \vee z \qquad b) \equiv \overline{x} \vee \overline{y} \vee \overline{z} \qquad c) \equiv x \Longrightarrow (\overline{y} \wedge \overline{z}) \equiv \overline{x} \vee (\overline{y} \wedge \overline{z}) \equiv (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z})$
$d) \equiv y \Longrightarrow (x \vee z) \equiv x \vee \overline{y} \vee z$

2 $F0(x,y,y) \underset{\text{DEF}}{=} a) \wedge b) \wedge c) \wedge d) \equiv (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee \overline{z}) \wedge (x \vee \overline{y} \vee z)$
Considérons l'ensemble des valuations de (x,y,z) classées par ordre lexicographique :
$(0,0,0)$ ne satisfait pas a); $(0,0,1)$ satisfait les quatre; $(0,1,0)$ ne satisfait pas d); $(0,1,1)$ satisfait les quatre;
$(1,0,0)$ satisfait les quatre; $(1,0,1)$ ne satisfait pas c); $(1,1,0)$ ne satisfait pas c) et enfin $(1,1,1)$ ne satisfait pas b)

En conclusion la formule F_0 est satisfiable par les valuations $(0,0,1)$, $(0,1,1)$ et $(1,0,0)$ de (x,y,z) □

3. $F1(x,y,z,1)$ fait apparaître la conjonction $(\overline{x} \vee z) \wedge (\overline{x} \vee \overline{z})$ donc la clause $\overline{x}$. Ainsi $F1(x,y,z,1)$ n'est éventuellement satisfiable que si $x = 0$. Or $F1(0,y,z,1) \equiv (y \vee z) \wedge (\overline{y} \vee \overline{z}) \wedge y \wedge (\overline{y} \vee z)$ insatisfiable.
$F1(x,y,z,0)$ fait apparaître directement la clause $\overline{x}$ et $F1(0,y,z,0) \equiv (y \vee z) \wedge (\overline{y} \vee \overline{z}) \wedge (\overline{y} \vee z)$ satisfiable uniquement avec $(y,z) = (0,1)$
En conclusion $F1(x,y,z,t)$ est satisfiable et sa seule solution est la valuation $(0,0,1,0)$ □

4. Soit $F2(x,y,z)$ la formule obtenue par la conjonction des 8 clauses deux à deux distinctes de longueur 3 que l'on peut former avec trois variables x, y et z. Elle est évidemment insatisfiable car pour toute valuation (a,b,c) la formule comprend une clause alors non satisfaite et d'ailleurs une seule : par exemple si $(a,b,c) = (0,1,1)$ la seule clause non satisfaite est $(x \vee \overline{y} \vee \overline{z})$. Ainsi $\max(F2) = 7$. □

5. Une clause de trois variables (parmi n) est insatisfaite pour une seule valuation des trois variables intervenant donc au total pour 2^{n-3} valuations sur les n variables. Donc $\sum\limits_{val \in V} \varphi(C, val) = 2^n - 2^{n-3} = 7 \times 2^{n-3}$ □

6. On a (Fubini discret sur un rectangle) : $7m2^{n-3} = \sum\limits_{C}\sum\limits_{val} \varphi(C,val) = \sum\limits_{val}\sum\limits_{C} \varphi(C,val) = \sum\limits_{val} \psi(F,val) \leqslant 2^n \max(F)$
Ainsi $\max(F) \geqslant \dfrac{7m}{8} = m - \dfrac{m}{8}$. Donc $\max(F) \geqslant m - \left\lceil \dfrac{m}{8} \right\rceil$ puisque $\max(F)$ est un entier. □

7. La question 4) fournit un exemple de telle formule non satisfiable avec $m = 8$.
Soit désormais une telle formule avec $m \leqslant 7$. Supposons la non satisfiable. On aurait alors $\max(F) < m$ donc a fortiori par la question précédente $m - \left\lceil \dfrac{m}{8} \right\rceil < m$ donc $\left\lceil \dfrac{m}{8} \right\rceil \geqslant 1$ donc $m \geqslant 8$. Contradiction.
En conclusion toute instance 3-SAT avec $m \leqslant 7$ est satisfiable et il existe une instance 3-SAT non satisfiable avec $m = 8$. □

Seconde partie : satisfiabilité, méthode exacte.

8. On commence pour simplifier par écrire une fonction auxiliaire qui renvoie la valeur d'un littéral pour une valuation donnée des variables :

```
let valeur_litteral x val =
  if (x > 0) then val.(x)
  else 1 - val.(-x)
;;
valeur_litteral : int -> int vect -> int = <fun>
```

On parcourt ensuite la clause à évaluer en s'arrêtant dès qu'on rencontre un littéral qui vaut vrai :

```
let valeur_clause C val =
  let p = C.(0) and k = ref 1 and valeur = ref 0 in
    while (!valeur = 0) && ( !k <= p) do
      if (valeur_litteral C.(!k) val = 1) then valeur := 1
      else incr k
    done;
    !valeur
;;
valeur_clause : int vect -> int vect -> int = <fun>
```

La fonction `valeur_litteral` est à temps constant donc la fonction `valeur_clause` est au mieux à temps constant (le premier littéral examiné est vrai) et au pire à complexité proportionnelle à p (les $p-1$ premiers littéraux sont faux). On retiendra donc que la complexité est un $O(p)$ en désignant par p le nombre de littéraux de la clause. □

9. Même principe : on parcourt les clauses de la formule en s'arrêtant dès qu'une clause est fausse.

```
let satisfait_formule F val =
  let m = F.(0).(0) and k = ref 1 and valeur = ref true in
    while (!valeur = true) && ( !k <= m) do
      if (valeur_clause F.(!k) val = 0) then valeur := false
      else incr k
    done;
    !valeur
;;
satisfait_formule : int vect vect -> int vect -> bool = <fun>
```

On effectue au plus m tours de boucles dont chacun est en $O(p)$ où p est le nombre de littéraux de la clause envisagée. On a donc une complexité en $O(\ell)$ où ℓ désigne la longueur de la formule c'est à dire la somme des longueurs des différentes clauses. □

10. Écriture immédiate par récursion sur $n-k$:

```
let rec resoudre_rec F val k =
  let n = F.(0).(1) in match (n -k) with
    | 0 -> satisfait_formule F val
    | _ -> (resoudre_rec F val (k+1)) ||
           (val.(k+1) <- 1 - val.(k+1); resoudre_rec F val (k+1))
;;
resoudre_rec : int vect vect -> int vect -> int -> bool = <fun>
```

11. On obtient immédiatement :

```
let resoudre F =
  let n = F.(0).(1) in let val = make_vect (n+1) 0 in
    if resoudre_rec F val 0 then val.(0) <- 1;
    val
;;
resoudre : int vect vect -> int vect = <fun>
```

12. Notons $p = n-k$ et $T(p)$ la complexité de `resoudre_rec F val k`.
Il vient $T(0) = O(\ell)$ (question 9) et pour $p \geqslant 1$:
$T(p) = T(p-1)$ si le premier `resoudre_rec` vaut vrai et $T(p) = 2T(p-1)+\alpha$ sinon où α correspond au temps de changement de la valeur de la case $k+1$ de val.
Au pire lorsque la formule est non satisfiable on a $T(p)-\alpha = 2(T(p-1)-\alpha)$ tout au long de la récursion.
Donc $T(n)-\alpha = 2^n(T(0)-\alpha)$ soit $T(n) = O(2^n\ell)$ □

Troisième partie : MAX-SAT.

13. $diff(F1,x) = 1$; $diff(F1,y) = 0$; $diff(F1,z) = 1$; $diff(F1,t) = -2$;
On a donc $\alpha_0 = t$ et on est conduit à poser $t = 0$ ce qui conduit à la formule $F' = (x \vee y \vee z) \wedge (x \vee \overline{y} \vee \overline{z}) \wedge \overline{x} \wedge (x \vee \overline{y} \vee z)$ en supprimant 3 clauses.
Il vient alors $diff(F',x) = 2$; $diff(F',y) = -1$; $diff(F',z) = 1$;
Ce qui conduit à poser $x = 1$ et à supprimer 3 clauses et à considérer $F" = \overline{x} = 0$

Ainsi l'heuristique à conduit à satisfaire 6 clauses sur 7 avec la valuation $(1,-,-,0)$ (alors que $F1$ est satisfiable par (0,0,1,0) d'après la question 3).

14. On parcourt les littéraux de la clause en s'arrêtant dès qu'on rencontre le littéral recherché. D'où une complexité en $O(p)$ en désignat par p la longueur de la clause.

```
let place C litt =
  let p = C.(0) and k = ref 1 in
    while (!k <= p) && (C.(!k) <> litt) do incr k done;
    if (!k <= p) then p else 0
;;
place : int vect -> int -> int = <fun>
```

15. Il suffit, compte tenu du codage d'une clause de remplacer le contenu de la case d'indice i par celle d'indice p et de décrémenter p c'est à dire la case d'indice 0. Ce qui est bien à temps constant.

```
let supprimer_variable C i =
  let p = C.(0) in
    C.(i) <- C.(p);
    C.(0) <- pred C.(0)
;;
supprimer_variable : int vect -> int -> unit = <fun>
```

16. Même méthode.

```
let supprimer_clause F i =
  let m = F.(0).(0) in
    F.(i) <- F.(m);
    F.(0).(0) <- pred m
;;
supprimer_clause : int vect vect -> int -> unit = <fun>
```

17. On parcourt les clauses successives de F et pour chaque clause on parcourt successivement les différents littéraux d'où une complexité en $\Theta(\ell)$ en désignant par ℓ la longueur de la formule.

```
let calculer_diff F =
  let m = F.(0).(0) and n = F.(0).(1) in let diff = make_vect (n+1) 0 in
    for i = 1 to m do
      for j = 1 to F.(i).(0) do let litt = F.(i).(j) in
        diff.(abs(litt)) <- diff.(abs(litt)) + (litt / abs(litt))
      done
    done;
    diff
;;
calculer_diff : int vect vect -> int vect = <fun>
```

18. On initialise une variable i à 1 qui désignera le numéro de la clause envisagée de l'état de la formule à cet instant.
À l'examen de la clause numéro C de numéro i, il y a naturellement trois cas :
1) si C contient le littéral *litt* : alors on modifie la formule en supprimant la clause (notons que m est ainsi décrémenté) puis on incrémente *compteur* MAIS PAS i compte tenu de la manière dont on supprime une clause.
2) sinon si C ne contient pas le littéral $-litt$ c'est à dire le complémenté de *litt* : alors on ne change pas la formule et on incrémente i.
3) sinon (C contient alors le littéral $-litt$) : il y a deux cas : si C est réduite à $-litt$ on supprime C (sans modifier ni i ni *compteur*) et sinon on supprime $-litt$ de la clause considérée et on incrémente i

On poursuit tant que $i \leqslant m$ où m désigne le nombre de clauses de l'état actuel dela formule .

La complexité de la suppression d'une variable dans une clause ou d'une clause dans une formule étant à temps constant, les trois cas possibles de chaque boucle sont à temps constant.
MAIS la détermination du cas à considérer (c'est à dire celui de la fonction `place`) est un $O(p)$ où p est la longueur de la clause envisagée.
À chaque tour de boucle soit i est incrémenté et m inchangé soit m est décrémenté et i inchangé. Donc la valeur de $m - i$ diminue d'une unité. Il y a donc m tours de boucle en désignant par m le nombre de clause de la formule de départ : normal toutes les clauses sont balayées.

En conclusion on a une complexité en $O(\ell)$ où ℓ désigne la longueur de la formule de départ. □

```
let simplifier F alpha v =
  let i = ref 1 and compteur = ref 0
  and litt = if (v = 1) then alpha else -alpha in
    while (!i <= F.(0).(0)) do match 0 with
      | _ when place F.(!i) litt >0     -> supprimer_clause F !i; incr compteur
      | _ when place F.(!i) (-litt) = 0 -> incr i
      | _ -> begin match F.(!i).(0) with
          | 1 -> supprimer_clause F !i
          | _ -> let k = place F.(!i) (-litt) in supprimer_variable F.(!i) k;
                 incr i
        end
    done;
    !compteur
;;
simplifier : int vect vect -> int -> int -> int = <fun>
```

19. On commence pour simplifier le code de la fonction `heuristique` par écrire une fonction `litt_max` de sorte que `litt_max V` où `V` est le tableau codant `calculer_diff F` renvoie le littéral à supprimer par l'heuristique dans la formule `F`.
Par exemple si -4 est renvoyé, l'heuristique consistera à appliquer `simplifier F 4 0`.

```
let litt_max V =
  let n = vect_length V - 1 and maxi = ref V.(1)
  and litt = if (V.(1) > 0) then ref 1 else ref (-1) in
    for i = 2 to n do
      if (abs V.(i) > abs !maxi) then begin
        maxi := V.(i);
        litt := if (V.(i) > 0) then i else -i
      end
    done;
    !litt
;;
litt_max : int vect -> int = <fun>
```

La complexité de cette fonction est clairement en $O(n)$ où n est le nombre de variables.

Le code de la fonction `heuristique` est alors immédiat :
On initialise le tableau `valuation` avec des -1 (sauf la case d'indice 0 initialisée à 0) de sorte qu'à la fin les cases contenant -1 désigneront des variables qui n'ont pas été affectées par l'heuristique. Il suffit ensuite d'appliquer l'heuristique de base c'est à dire la fonction `simplifier` tant que la formule contient au moins une clause et d'ajouter ce qu'elle renvoie à la case d'indice 0 du tableau `valuation`.

```
let heuristique F =
  let n = F.(0).(1) in let valuation = make_vect (n+1) 2 in
    valuation.(0) <- 0;
    while (F.(0).(0) > 0) do
      let i = litt_max (calculer_diff F) in
        let v = if (i > 0) then 1 else 0 in
        valuation.(abs i) <-  v;
        valuation.(0) <- valuation.(0) + (simplifier F (abs i) v)
    done;
    valuation
;;
heuristique : int vect vect -> int vect = <fun>
```

À chaque tour de boucle, le nombre de clauses de la formule diminue d'au moins une unité. Il y a donc au plus m tours de boucle en désignant par m le nombre de clauses de la formule initiale.
Le coût de chaque tour de boucle est celui de la fonction `calculer_diff` soit $O(\ell')$ où ℓ' est la longueur actuelle de la formule, de la fonction `litt_max` à savoir $O(n)$ et de la fonction `simplifier` soit $O(\ell')$ soit au total $O(n)+O(\ell')$.

On a bien sûr $\ell' \leqslant \ell$ et $n \leqslant \ell$ puisque toutes les variables figurent dans la formule de départ. Ainsi le coût de chaque étape est un $O(\ell)$ de sorte que le coût total est un $O(m\ell)$. □

Quatrième partie : étude d'un cas paticulier.

20.

a) Quitte à renuméroter les littéraux et à remplacer la variable y par $\overline{y}$, on ne restreint pas la généralité du problème en supposant que $F = (x \vee y \vee \ldots) \wedge (\overline{x} \vee \overline{y} \ldots) \wedge G$
Naturellement si F est satisfiable, G l'est.
Supposons G satisfiable. Comme F est une formule 1-occ, G ne contient ni x, ni y, ni leurs complémentés. Soit alors une valuation (ne contenant ni x ni y) satisfaisant G. Alors en lui ajoutant $x = 1$ et $y = 0$ on obtient une valuation satisfaisant F. □

b) Notons $F'' = (l_1 \vee l_2 \vee \ldots \vee l_k \vee l_{k+1} \vee \ldots \vee l_{k+h})$ qui est bien une clause puisque F" ne contient aucune variable et son complémenté et notons $F' = F$" $\wedge\, G$ qui est bien une formule 1-occ puisque F en est une.
Comme F est une formule 1-occ, G ne continet ni x ni $\overline{x}$ donc F' non plus.
Ainsi F' est bien une formule 1-occ ne contenant ni x ni $\overline{x}$.

Supposons F satisfaite par une valuation v. Alors naturellement v satisfait G. Si $v(x) = 1$ alors comme v satisfait la clause $(\overline{x} \vee l_{k+1} \vee \ldots \vee l_{k+h})$ alors v satisfait la clause $(l_{k+1} \vee \ldots \vee l_{k+h})$ donc la clause F''. Même raisonnement si $v(x) = 0$. Ainsi si F est satisfiable, il en va de même de F'.

Supposons désormais F' satisfaite par une valuation v. Alors v à laquelle on rajoute n'importe quelle valeur de x satisfait G puisque G ne contient ni x ni $\overline{x}$. Comme F'' est satisfaite par v, l'un au moins de ses littéraux l_{i_0} est satisfait par v. Si $i_0 \leqslant k$ alors la valuation $(\overline{x}, v)$ satisfait $(x \vee l_1 \vee \ldots \vee l_k) \wedge (\overline{x} \vee l_{k+1} \vee \ldots \vee l_{k+k})$ donc F et si $i_0 \geqslant k+1$ alors la valuation (x, v) satisfait de même F.

En conclusion F est satisfiable si et seulement si la formule 1-occ F' l'est. □

21.

a) La formule réduite par rapport à x est $G = (\overline{y} \vee \overline{t})$ d'après la question 20.a).

b) La formule réduite par rapport à t est $F' = (\overline{x} \vee \overline{z} \vee \overline{u}) \wedge (z \vee u)$ d'après la question 20.b).

22. On raisonne par récurrence sur le nombre n de variables.
Pour $n = 0$ on a une formule vide donc satisfiable. Supposons le résultat établi pour $n \leqslant n_0$ et considérons une formule 1-occ ayant $n_0 + 1$ variable.
On envisage le premier littéral de la première clause. Deux cas :
Premier cas : le complémenté de ce littéral figure dans une autre clause. Soient alors x la variable correspondant à ce littéral et F' la formule réduite de F par rapport à x (suivant 20.a) ou 20.b) suivant le cas). Alors F' est une formule 1-occ ayant au plus n_0 variables donc est satisfiable par hypothèse de récurrence. Donc F aussi par la question 20.
Deuxième cas : le complémenté de ce littéral ne figure pas dans les autres clauses. On donne à la variable correspondant au littéral la valeur qui satisfait ce littéral donc la première clause. Il reste à appliquer l'hypothèse de récurrence sur la formule 1-occ ayant n_0 variables obtenue à partir de F en supprimant la première clause.

Ainsi toute formule 1-occ est satisfiable.

23. L'algorithme est basée sur la démonstration ci-dessus.
On considère le premier littéral *litt* de la première clause C_1 (correspondant à la variable x c'est à dire $litt = x$ ou $litt = \overline{x}$) de la formule F puis on parcourt par une boucle *while* les clauses suivantes tant qu'on n'a pas rencontré le complémenté $-litt$.
Premier cas : on le rencontre dans la clause C_i :

On examine alors la clause F'' formée par la réunion des littéraux autres que *litt* et $-litt$ se trouvant dans les clauses C_1 et C_i. Deux cas à nouveau :
Premier sous-cas : on y trouve à la fois une variable y et son complémenté :

On remplit alors les cases du tableau *val* correspondant aux variables x et y par les valeurs adéquates comme expliqué dans la question 20.a) puis on appelle récursivement *calculer_solution* sur la formule G (réduite de F par rapports à x) et le tableau *val*

Deuxième sous-cas : on n'y trouve aucune variable et son complémenté :

On appelle récursivement *calculer_solution* sur la formule réduite F' de F par rapport à x (ce qui remplit les cases autres que x du tableau *val*) puis on cherche un littéral de F'' satisfait par *val* (il y en a forcément au moins un) et suivant que ce littéral provient de C_1 ou de C_i on remplit comme expliqué dans la question 20) la case x du tableau *val*.

Deuxième cas : on ne le rencontre pas :

On affecte la valeur 1 à la variable x si $litt = x$ et 0 si $litt = \overline{x}$ dans le tableau *val* et on appelle récursivement *calculer_solution* sur la formule F' déduite de F en retirant la première clause.

24.
Premier appel sur $F4$: premier cas avec x et premier sous-cas. Ce qui conduit à $x = 0$ et $z = 1$ par exemple (ou $x = 1$ et $z = 0$) et à appeler la fonction sur $F'4 = (\overline{y} \vee \overline{v}) \wedge (u \vee v) \wedge (\overline{t} \vee \overline{u})$
Deuxième appel sur $F'4$: premier cas avec v et deuxième sous-cas avec $F''4 = (\overline{y} \vee u) \wedge (\overline{t} \vee \overline{u})$ sur laquelle on appelle récursivement la fonction.
Troisième appel sur $F''4$: Premier cas avec u et deuxième sous-cas avec $F'''4 = (\overline{y} \vee \overline{t})$ sur laquelle on appelle la fonction.
Quatrième appel sur $F'''4$: Deuxième cas : ce qui conduit à $y = 0$ et $t = 0$.
Retour au troisième appel : comme $\overline{y}$ est satisfait, l'algorithme conduit à poser $u = 0$ (notons que $u = 1$ conviendrait également).
Retour au deuxième appel : de même comme $\overline{y}$ est satisfait l'algorithme conduit à poser $v = 1$.
Fin de l'algorithme car le premier appel était terminal.

Cela conduit à la solution $(x, y, z, t, u, v) = (0, 0, 1, 0, 0, 1)$

FIN