Un corrigé : X-ENS INFO C 2024
Egalités et différences

Pour toutes les remarques ou corrections, vous pouvez m'envoyer un mail à
galatee.hemery@gmail.com

1 Partie I. Équivalences de littéraux

Question 1. On remarque que $\ell_1 \leftrightarrow \ell_2$ est équivalent à $(\ell_1 \vee \neg\ell_2) \wedge (\neg\ell_1 \vee \ell_2)$ d'après les remarques dans les notations.
De plus $\ell_1 \leftrightarrow \ell_2$ et $\ell_2 \leftrightarrow \ell_1$ sont redondants. F_0 est équivalent à :

$$(X_3 \vee X_2) \wedge (\neg X_3 \vee \neg X_2) \wedge (X_4 \vee \neg X_3) \wedge (\neg X_4 \vee X_3) \wedge (X_1 \vee X_5) \wedge (\neg X_1 \vee \neg X_5)$$

Question 2. Pour $F \in \mathcal{F}(L_1)$, on peut construire une formule 2-CNF en temps linéaire en associant à une contrainte $c = \ell_1 \leftrightarrow \ell_2$ la formule $f_c = (\ell_1 \vee \neg\ell_2) \wedge (\neg\ell_1 \vee \ell_2)$ et en simplifiant les éventuelles double-négation.
Ainsi, $F_2 = \bigwedge_{c \in F} f_c$ est une formule 2-CNF équivalente à F et la construction se fait en temps linéaire.
F est une instance positive de P_{L_1} si et seulement si F_2 est une instance positive du problème de satisfiabilité des 2-CNF.
Ainsi, $P_{L_1} \leq_m^{lin} 2-SAT$, on a bien une réduction linéaire.

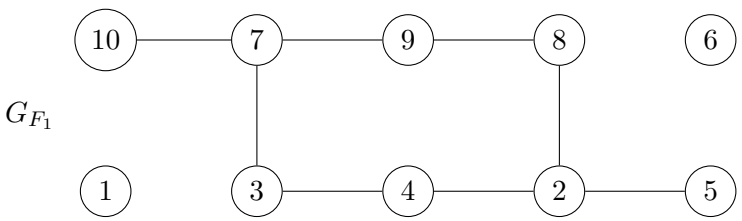
Question 3. n est une variable globale.

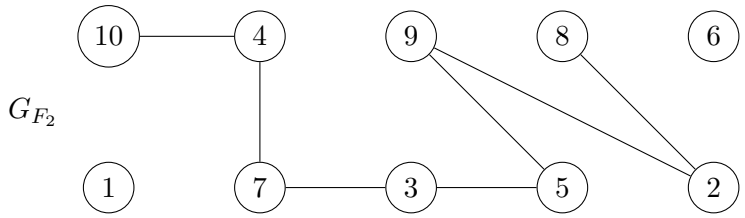
```
let var_of_lit i =
    if i>0 && i<= n then i
    else if i> n && i<= 2*n then i-n
    else raise (Bad_literal i)

let neg_lit i =
    if i>0 && i<= n then i+n
    else if i> n && i<= 2*n then i-n
    else raise (Bad_literal i)

let rec eform1_is_wf e = match e with
    | [] -> true
    | (i1,i2)::q ->
        try
            let _ = var_of_lit i1 in
            let _ = var_of_lit i2 in
            eform1_is_wf q
        with Bad_literal i -> false
```

Question 4.





Question 5. n est toujours une variable globale donc il n'est pas nécessaire de le chercher.

```
let eform1_to_graph f =
    let g = g_create (2*n) in
    let rec parcours l = match l with
        | [] -> g
        | (i,j)::q when i = neg_lit j -> raise Exn_unsat
        | (i,j)::q when i = j -> parcours q
        | (i,j)::q -> g_add_edge g i j ;
                      g_add_edge g (neg_lit i) (neg_lit j) ;
                      parcours q in
    parcours f
```

Question 6. L'appel à `g_create` est en $O(n)$ ($2n$ sommets).
On parcours ensuite les p contraintes et pour chaque contrainte non contradictoire ou triviale, on a trois appels à `neg_lit` en $O(1)$ et deux à `g_add_edge` en $O(n)$.
Au total, on a une complexité temporelle en $O(n \times (p+1))$.

Question 7.

```
let g_cc_array g =
    let v = g_nb_vertex g in
    let c = Array.make (v+1) 0 in
    let rec explore cc s=
        if s = v+1 then c, (cc-1)
        else if c.(s) <> -1 then explore cc (s+1)
        else begin
            c.(s) <- cc;
            List.iter (explore cc) (g_succ g s) ;
            explore (s+1) (cc+1)
        end in
    explore 1 1
```

Question 8. Soit i et j deux sommets distincts d'une même composante connexe de G_F, alors il existe un chemin de i à j, notons le : $i = i_0, i_1, \ldots, i_p, i_{p+1} = j$.
On note i'_k pour `neg_lit`(i_k) comme dans la définition du graphe.
En particulier, pour tout $0 \leq k \leq p$, on a une arête entre i_k et i_{k+1} correspondant à une contrainte $litt(i_k) \leftrightarrow litt(i_{k+1})$ ou bien $litt(i'_k) \leftrightarrow litt(i'_{k+1})$ dans F.
Soit v une valuation satisfaisant F, alors pour chaque contrainte $litt(i_k) \leftrightarrow litt(i_{k+1})$ ou $litt(i'_k) \leftrightarrow litt(i'_{k+1})$ dans F, v satisfait les deux contraintes qui sont équivalentes car l'une au moins apparaît dans F, conjonction de contraintes.
Ainsi, on a $v(litt(i_k)) = v(litt(i_{k+1}))$ nécessairement pour tout k.
L'égalité est vraie pour tout le chemin donc $v(litt(i)) = v(litt(j))$ et v satisfait $litt(i) \leftrightarrow litt(j)$.
D'où $F \vDash litt(i) \leftrightarrow litt(j)$.

Question 9. Montrons dans un premier temps qu'il existe un chemin entre toute paire de j_k.
Soit $j_k, j_{k'}$ dans $\{j_1, ..., j_q\}$.

Il existe un chemin de i_k à $i_{k'}$ car ils sont dans la même composante connexe, notons le : $i_k = s_0, s_1, \ldots, s_p, s_{p+1} = i_{k'}$ où les s_i sont dans $\{i_1, ..., i_q\}$
Pour toute arête (s_r, s_{r+1}), il existe une contrainte qui a permis d'ajouter deux arêtes au graphe (s_r, s_{r+1}) et (s'_r, s'_{r+1}) ou s'_r et s'_{r+1} correspondent à `neg_lit`(s_r) et `neg_lit`(s_{r+1}).
Ainsi, il existe un chemin $j_k = s'_0, s'_1, \ldots, s'_p, s'_{p+1} = j_{k'}$.
Ainsi tous les j_k sont dans la même composante connexe.
Montrons maintenant qu'il n'y en a pas d'autres.
Soit un sommet s dans la composante connexe contenant $\{j_1, ..., j_q\}$.
On note alors s' sa négation et par le même raisonnement que précédemment, on a un chemin de i_1 à s' et $s' \in C = \{i_1, ..., i_q\}$ donc $s \in C' = \{j_1, ..., j_q\}$.
Ainsi, on a bien $C' = \{j_1, ..., j_q\} \in CC(G_F)$.

Question 10. Raisonnons par double implication.
$\Leftarrow$: Soit F tel qu'il existe $C \in CC(G_F)$ et i un sommet de G_F tel que `neg_lit`(i) est dans C.
Par la question 8, on a $F \vDash lit(i) \leftrightarrow lit($`neg_lit`$(i))$
Or $lit(i) \leftrightarrow lit($`neg_lit`$(i))$ est une contradiction donc n'est satisfaite par aucune valuation.
Nécessairement F est une contradiction.
$\Rightarrow$: On raisonne par contraposée, on suppose qu'il n'existe pas de telle composante connexe, montrons qu'il existe une valuation satisfaisant F.
On construit une valuation comme suit :

- on affecte 1 à X_1 et on affecte 1 (Vrai) à tous les littéraux des sommets de la composante connexe de 1, $\{1 = i_1, \ldots, i_q\}$ et 0 à toute la composante connexe de $n+1$, $\{n+1 = j_1, \ldots, j_q\}$ avec les notations de la question 9 ;
- tant qu'il en existe des non traitées, on choisit une composante connexe et on réalise la même opération que pour la composante de 1.

Comme une composante ne contient i et `neg_lit`(i) pour tout i, on a aucune contradiction. De plus, par la question précédente, on a la garantie que les négation sont dans la composante "compagnon" traitée. Ainsi, on ne rencontre plus les variables déjà traitées dans les composantes suivantes.
Pour chaque arête du graphe, on a la même valeur de vérité pour les littéraux correspondant aux deux extrémités donc la valuation ainsi construite satisfait toutes les contraintes, donc F.

Question 11. On vérifie la condition précédente.

```
let cc_is_sat c =
    let n = ((Array.length c)-1)/2 in
    let rec test i =
        if i = n+1 then true
        else if c.(i) = c.(neg_lit i) then false
        else test (i+1) in
    test 1
```

Question 12. On applique l'algorithme décrit dans les questions précédentes.
La construction du graphe `g` se fait bien en $O(n \times p)$ *(dans le cas où $p \neq 0$)* d'après la queston 6.
La recherche des composantes connexes se fait en $O(n+p)$ puis qu'on réalise un parcours du graphe à $2n$ sommets et au plus $2p$ arêtes et chaque sommet et arêtes est traité une unique fois (la ligne 9 de la question 7 correspond au traitement des arêtes et la ligne 8 au traitement d'un sommet et la ligne 6 est une vérification pour ne pas traiter le sommet une fois de plus).
`cc_is_sat` se fait en $O(n)$ car on vérifie chaque variable.
La construction de la valuation dans le cas où elle existe se fait en $O(n)$.
On parcourt le tableau des composantes connexes pour créer la valuation en associant 1 pour

une composante non traité et 0 à sa négation.
On construit ensuite la valuation en combinant la valuation sur les composantes connexes et le tableau des composantes connexes.
Au total, c'est la création du graphe qui prend le plus de temps et la complexité est bien en $O(n \times p)$.

```
let eform1_sat f =
    let g = eform1_to_graphe f in
    let c,m = g_cc_array g in
    if cc_is_sat c then
        let val_cc = Array.make (m+1) (-1) in
        val_cc.(0) <- 0 :
        val_cc.(1) <- 1 ;
        for i = 1 to n do
            if val_cc.(c.(i)) = -1 then (
                val_cc.(c.(i)) <- 1 ;
                val_cc.(c.(neg_lit i)) <- 0
            )
        done;
        let v = Array.make (n+1) 0 in
        for i = 1 to n do
            v.(i) <- val_cc.(c.(i))
        done;
        Some v
    else None
```

2 Partie II. Équivalences avec une conjonction de deux littéraux

Question 13. Une contrainte $\ell_1 \leftrightarrow (\ell_2 \wedge \ell_3)$ est équivalente à :

$$(\neg\ell_1 \vee (\ell_2 \wedge \ell_3)) \wedge (\ell_1 \vee \neg(\ell_2 \wedge \ell_3)),$$

puis à (distribution et loi de Morgan)

$$(\neg\ell_1 \vee \ell_2) \wedge (\neg\ell_1 \vee \ell_3)) \wedge (\ell_1 \vee \neg\ell_2 \vee \neg\ell_3)).$$

Pour $F \in \mathcal{F}(L_2)$, on peut construire une formule 3-CNF en temps linéaire en associant à une contrainte $c = \ell_1 \leftrightarrow (\ell_2 \wedge \ell_3)$ la formule $f_c = (\neg\ell_1 \vee \ell_2) \wedge (\neg\ell_1 \vee \ell_3)) \wedge (\ell_1 \vee \neg\ell_2 \vee \neg\ell_3))$ et en simplifiant les éventuelles double-négation.
Ainsi, $F_3 = \bigwedge_{c \in F} f_c$ est une formule 3-CNF équivalente à F et la construction se fait en temps linéaire.
F est une instance positive de P_{L_2} si et seulement si F_3 est une instance positive du problème de satisfiabilité des 3-CNF.
Ainsi, $P_{L_2} \leq_m^P 3-SAT$, on a bien une réduction linéaire donc polynomiale.

Question 14. On pose $\Gamma = \{X_2 \to (\neg X_2 \wedge X_3), (\neg X_2 \wedge X_3) \to X_2, X_2\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash X_2 \to (\neg X_2 \wedge X_3)}\,\text{ax} \quad \dfrac{}{\Gamma \vdash X_2}\,\text{ax}}{\Gamma \vdash \neg X_2 \wedge X_3}\,{\to_E}}{\Gamma \vdash \neg X_2}\,{\wedge_E^1} \quad \dfrac{}{\Gamma \vdash X_2}\,\text{ax}}{\Gamma \vdash \bot}\,{\neg_E}}{X_2 \to (\neg X_2 \wedge X_3), (\neg X_2 \wedge X_3) \to X_2 \vdash \neg X_2}\,{\neg_I}$$

Question 15.

```
1  let contr2_simplifiy_eq5 e2 =
2      if e2.l1 <> 1 then None
3      else if e2.lr1 = neg_lit e2.lr2 then raise Exn_unsat
4      else Some [(1,e2.lr1); (1,e2.lr2)]
```

Question 16.

```
1  let eform1_of_valp v = (* renvoie une representation de F *)
2      let n = (Array.length v)-1 in
3      let rec construit i =
4          if i = n+1 then []
5          else if v.(i) = 1 then (i,1)::(construit (i+1))
6          else if v.(i) = 0 then (i,n+1)::(construit (i+1))
7          else construit (i+1) in
8      construit 1
```

Soit v' une valuation.
Si $v' \vDash F$, alors v' satisfait tous les contraintes de F donc pour tout i tel que $v(X_i) = 1$, v' satisfait $X_1 \leftrightarrow X_i$ et par convention $v'(X_1) = v(X_1) = 1$ donc $v'(X_i) = v'(X_1) = v(X_1) = v(X_i)$, de même pour tout i tel que $v(X_i) = 0$ v' satisfait $\neg X_1 \leftrightarrow X_i$ et par convention $v'(X_1) = v(X_1) = 1$ donc $v'(X_i) = 1 - v'(X_1) = 1 - v(X_1) = v(X_i)$.
Ainsi, v est compatible avec v'.
Réciproquement, si v compatible avec v', alors pour tout i tel que $v(X_i) = 1$, alors $v'(X_i) = 1$ et v' satisfait $X_1 \leftrightarrow X_i$ et pour tout i tel que $v(X_i) = 0$, alors $v'(X_i) = 0$ et v' satisfait $\neg X_1 \leftrightarrow X_i$ et au total v' satisfait toutes les contraintes de F donc $v' \vDash F$.

Question 17. On note qu'il n'y a en fait pas d'appels récursifs à `eform2_sat` et uniquement un appel à `eform2_sat_aux` qui est elle effectivement récursive.
Montrons la terminaison de `eform2_sat_aux lst0 [] vinit` où `vinit` est un tableau de taille $(n+1)$ rempli du -1 sauf en position 0, 0 et en position 1, 1, représentant la valuation partielle où aucune variable n'est affecté sauf X_1 à 1 par convention.
Notons que la somme du nombre de contraintes et du nombre de variables non affectées est strictement décroissante pour les appels récursifs : en effet, l'appel récursif ligne 17 se fait avec une contrainte de moins et l'appel récursif des lignes 24 et 28 se font avec une variable non affectée en moins car on affecte 1 ou 0 à `xvlr1`.
En effet la variable `xvlr1` n'est pas déjà affectée sinon on pourrait appliqué la simplification (1) ou (2) car sa variable est considéré comme X_1 ou $\neg X_1$.
Ainsi, on a un **variant** strictement décroissant positif et en particulier si on a aucune variable non affectée, on connaît la valeur de vérité de chaque contrainte donc les contraintes sont consommés via les appels de la ligne 17 ou bien l'exception `Exn_unsat` est levée.
Enfin, quand il n'y a plus de contrainte, on appelle `eform1_of_valp` et `eform1_sat` qui termine bien.
Ainsi, tout appel `eform2_sat lst0` termine.

Question 18. On suppose que `eform2_sat_aux l2 l1 vp` lève l'exception `Exn_unsat` si la formule formée par la conjonction des contraintes des listes `l2` de classe L_2 et `l1` de classe L_1 n'admet pas de valuation v compatible avec la valuation partielle représentée par `vp` la satisfaisant et `v` une représentation d'une valuation v compatible avec la valuation partielle représentée par `vp` la satisfaisant s'il en existe une.
En particulier, `eform2_sat` fait un appel à cette fonction dans la ligne 5 et renvoie donc `Some v`

avec `v` une valuation compatible avec la valuation partielle où seul $v(X_1) = 1$ est fixé et satisfaisant les contraintes de `lst` s'il en existe une.
L'appel lève l'exception `Exn_unsat` s'il n'en existe pas, qui est rattrapé pour renvoyer `None`.
Cela justifie la correction de la fonction en admettant la correction de la première.

Question 19. Un appel `eform2_sat_aux [] l1 vp` fait appels aux deux fonctions `eform1_of_valp` et `eform1_sat` dont on sait la correction.
`l1'` représente donc une formule F qui encode les assignations de la valuation partielle `vp`.
`eform1_sat` renvoie `None` si la formule représenté par `l1'@l1` n'est pas satisfiable, c'est à dire s'il n'existe pas de valuation satisfaisant `l1'` et `l1`, donc par la question 16 s'il n'existe pas de valuation compatible avec `vp` satisfaisant `l1`.
`eform1_sat` renvoie `Some v` si la formule représenté par `l1'@l1` est satisfiable par une valuation `v`, c'est à dire `v` satisfait `l1'` et `l1`, donc par la question 16 `v` compatible avec `vp` et satisfait `l1`.
Le `match` permet de lever `Exn_unsat` dans le premier cas et renvoie `v` dans le deuxième, ce qui permet de conclure quant à la correction.

Question 20. La formule `l1'` est équivalente à la contrainte `e2` dans le cadre de la valuation partielle `vp` donc satisfaire `e2::l2'` et `l1` en étant compatible à `vp` est équivalent à satisfaire `l2'` et `l1'@l1` en étant compatible à `vp`.
Ainsi si l'appel récursif est correct, l'appel `eform2_sat_aux l2 l1 vp` est correct.

Question 21. Les lignes 22-23 et les lignes 26-27 copient la valuation partielle et affecte la variable `xlr1` à `1` ou `0` dans les deux cas. On a donc une valuation partielle avec une variable de plus affecté.
On a déjà justifié dans la preuve de terminaison que dans le cas où `contr2_simplify_vp` renvoie `None` cette variable est telle que `vp(xlr1) = -1`.
`eform2_sat_aux l2 l1 vp1` renvoie `v` une valuation compatible avec `vp1` satisfaisant `l2` et `l1` ou lève une exception `Exn_unsat` :

- dans le premier cas, on renvoie bien une valuation compatible avec `vp1` donc `vp` en particulier satisfaisant `l2` et `l1` ;
- dans le second cas l'exception est rattrapé et on renvoie le résultat de `eform2_sat_aux l2 l1 vp0` qui fonctionne comme le premier appel récursif :
  - dans le cas où l'on obtient une valuation, elle convient par les même arguments ;
  - dans le cas où l'exception `Exn_unsat` est à nouveau levée, cela signifie qu'il n'existe aucune valuation compatible avec `vp1` ou `vp0` satisfaisant `l2` et `l1`. Or toute valuation compatible avec `vp` est compatible avec `vp1` ou `vp0` selon la valeur de vérité affectée à `xlr1`. Dans il n'existe pas de valuation compatible avec `vp` satisfaisant `l2` et `l1`

Au total, dans tous les cas le résultats renvoyé est correct.

Question 22. L'appel à `eform1_sat` est en $O(n \times p)$ avec p le nombre de contraintes dans `l1'@l1` or `l1'` contient au plus $n-1$ contraintes (une par variable sauf X_1) et `l1` contient $O(k)$ contraintes issues des k simplifications (elles se font en $O(1)$ donc produisent une formule de taille $O(1)$). Donc l'appel à `eform1_sat` est en $O(n \times (n+k))$.
`eform1_of_valp` est en $O(n)$.
On note $T(n, i, k)$ la complexité dans le pire cas pour n variables, dont i non affectées et k contraintes :

$$T(n, i, k) = \max(2T(n, i-1, k) + O(n), T(n, i, k) + O(1))$$

la copie des valuations est coûteuse.
Donc sans justification supplémentaire (donner dans l'énoncé) $T(n, i, k) = O((k+n)2^i)$.
La complexité est en $O((k+n)2^n)$

3 Partie III. Egalités et différences entre entiers

Question 23.

```
struct eform3_s {
    bool is_eq;
    int xi;
    int xj;
    struct eforme3_s * suite; //suite de la liste chaînée
};
typedef struct eform3_s eform3_t;
```

Question 24.

```
void eform3_print (eform3_t *lst) {
    if (lst == NULL) {
        printf("true\n");
    }
    else if (lst->is_eq) {
        printf("X\_%i = X\_%i \n", lst->xi, lst->xj);
        if (lst->suite != NULL) {
            eform3_print(lst->suite);
        }
    }
    else {
        printf("X\_%i != X\_%i \n", lst->xi, lst->xj);
        if (lst->suite != NULL) {
            eform3_print(lst->suite);
        }
    }
}
```

Question 25. n est une variable globale entière.

```
uf_t *eform3_to_uf (eform_t *lst) {
    cs = uf_create(n+1);
    eform_t * maillon = lst ;
    while (maillon != NULL) {
        if (maillon->is_eq) {
            uf_union(cs,maillon->xi,maillon->xj);
        }
        maillon = maillon->suite;
    }
    return cs;
}
```

Question 26. On utilise la fonction précédente et on vérifie que pour chaque contrainte de différence, les deux variables ne sont pas dans la même classe d'équivalence.
Si cette condition est bien vérifiée, alors il suffit d'affecter des valeurs différentes pour toutes les classes d'équivalences :

— toutes les contraintes d'égalité sont vérifiées car les deux membres sont dans une même classe donc affectés au même entier ;

— toutes les contraintes de différences sont vérifiées car les deux membres sont dans des classes distinctes dont affectés à des entiers distincts.

Si la condition n'est pas vérifiée, alors on a une suite d'égalité qui relie deux variables $X_i = X_{i_1}$, $X_{i_1} = X_{i_2}$, ..., $X_{i_k} = X_j$, celles qui ont permis de mettre i et j dans la même classe et $X_i \neq X_j$ une contrainte de différence.
Ainsi, par transitivité de l'égalité, on doit avoir $X_i = X_j$ et $X_i \neq X_j$ qui est impossible.
Cela justifie la correction de la fonction suivante :

```
bool eform3_sat_int(eform3_t *lst) {
    if (lst == NULL) {
        return true;
    }
    cs = eform3_to_uf(lst);
    eform_t * maillon = lst ;
    while (maillon != NULL) {
        if (!maillon->is_eq &&
            uf_find(cs,maillons->xi)==uf_find(cs,maillon->xj)) {
                return false;
        }
        maillon = maillon->suite;
    }
    return true;
}
```

Question 27. `eform3_to_uf` fait un appel à `uf_create` en $\Theta(n)$, puis e appels à `uf_union` en $O(\log(n))$ et la boucle `while` parcourt l'ensemble des contrainte en $O(e+d)$.
Au total, `eform3_to_uf` a une complexité en $O(n + e(\log(n)+1) + d)$
Dans `eform3_sat_int`, on ajoute $2d$ appels à `uf_find` en $O(\log(n))$ et la boucle `while` parcourt l'ensemble des contrainte en $O(e+d)$.
Au total, `eform3_sat_int` a une complexité en $O(n + (e+d)\log(n))$ (on admet $n > 1$).

Question 28. On compte la classe de l'élément 0 ici car ce n'est pas préciser que l'on se restreint au cas où 0 ne sert qu'à décaler.
On compte les représentants des classes qui sont tels que `cs->parent[i] == i`.

```
int uf_classes(uf_t *cs) {
    if (cs == NULL) { return 0; }
    int res = 0;
    for (int i = 0; i<cs->nelem; i++) {
        if (cs->parent[i] == i) {
            res = res +1;
        }
    }
    return res;
}
```

Question 29. On effecte d'abord des valeurs aux classes via les représentants puis aux éléments.

```
int *uf_valuation(uf_t *cs) {
    int n = cs->nelem -1;
    int * v = malloc(sizeof(int)*(n+1));
    assert(v!=NULL);
```

```
5       int classes = 0;
6       v[0] = 0;
7       for (int i = 1; i<n+1; i++) {
8           // affectation des valeurs aux représentants
9           if (cs->parent[i] == i) {
10              classes = classes +1;
11              v[i] = classes;
12          }
13      }
14      for (int i = 1; i<n+1; i++) {
15          // affectation des valeurs aux autres
16          v[i] = v[uf_find(cs,i)];
17      }
18      return v;
19  }
```

Question 30. Cela a déjà été justifié dans la question 26 où la construction d'une telle valuation justifie l'existence d'une valuation satisfaisant les contraintes.

Question 31. Si F ne contient pas de contraintes de différence, on peut prendre une valuation constante et on a une unique valeur dans l'image de v. On ne peut pas faire moins car X_1 doit avoir une valeur (si $n \geq 1$).
Si F contient une unique différence et F satisfiable, on peut affecter une même valeur à toutes les classes d'équivalence sauf une contenant un des deux membres de la différence. On a alors un nombre minimal de deux valeurs. Avec une unique valeur, la contrainte de différence n'est pas satisfaite.

Question 32. On construit un graphe dont les sommets sont les classes d'équivalences obtenues grâce aux contraintes d'égalité et dont les arêtes sont obtenues via les contraintes de différences. Pour tout contrainte $X_i \neq X_j$, on ajoute une arête entre les classes de i et j.
Une valuation satisfaisant F utilise nécessairement une unique valeur pour une classe d'équivalence, ainsi cela correspond à un coloriage du graphe $G_{F,3} = (S_{F,3}, A_{F,3})$ où l'on associe une valeur (couleur) à chaque sommet de telle sorte que les deux extrémités d'une arête doivent avoir des couleurs différentes.
Déterminer s'il existe v une valuation dont l'image est de taille 2 et satisfaisant F revient à déterminer si le graphe a au moins 2 sommets et est coloriable avec 2 couleurs.
Pour vérifier cela, on réalise un parcours du graphe en $O(|S_3| + |A_3|)$ qui colorie en deux couleurs (1 et 2). En effet, on part d'un sommet arbitraire, on lui associe la couleur 1, les voisins ont nécessairement la couleur 2, etc.
Si le parcours réussit à construire un tel coloriage, on renvoie la valuation correspondante qui convient, sinon ce n'est pas possible (lorsque l'on doit colorier un sommet avec 1 et qu'il est déjà colorié avec 2 ou inversement).
On a $A_{F,3}$ de cardinal au plus d et $|S_{F,3}|$ de cardinal au plus n donc le parcours est en $O(d+n)$.
La construction d'un tel graphe demande la construction des classe d'équivalence et le parcours des contraintes de différences donc a la même complexité que `eform3_sat_int` donc $O(n + (e + d)\log(n))$.
Au total, on a une complexité en $O(n + (e + d)\log(n))$.

4 Partie IV. Égalités et différence d'expressions

Question 33.
$\mathcal{T}(t_1) = \{f_1(f_2(X_1, X_2), X_2), f_2(X_1, X_2), X_1, X_2\}$
$\mathcal{T}(t_2) = \{f_3(X_1, f_2(X_1, X_2)), f_2(X_1, X_2), X_1, X_2\}$

Question 34. Il n'est pas clair que la structure unir et trouver suffit. On suppose que toutes les contraintes d'égalité sont dans la structure (sinon cela revient à faire une fonction similaire à la question suivante). De plus chaque terme est représenté par une unique position, donc tester l'égalité ou l'égalité contrainte par `g->uf` suffit.

```
bool term_is_diff(egraph *g, int tid1, int tid2) {
    return !(tid1 == tid2||uf_find(g->uf,tid1) == uf_find(g->uf,tid2));
}
```

Question 35. `term_is_congruent` est une fonction récursive qui réalise deux appels récursifs lorsque les labels sont égaux. Elle en réalise au plus autant qu'il y a de noeuds dans le terme t_1 ou dans le terme t_2 car on descend en parallèle dans les deux arbres des termes.
Ainsi, on a $O(\texttt{g.tset\_sz})$ appels récursifs au plus, les deux appels à `uf_find` sont en $O(\log(\texttt{g.tset\_sz}))$ et les autres opérations, en particulier la comparaison des labels, sont en $O(1)$.
Au total on a une complexité en $O(\texttt{g.tset\_sz}\log(\texttt{g.tset\_sz}))$.

```
bool egalite_label(char * l1, char * l2) {
    int i = 0;
    while (l1[i]!='\0' && l2[i]!= '\0') {
        if (l1[i] != l2[i]) {
            return false;
        }
        i = i+1;
    }
    return true;
}
bool term_is_congruent(egraph *g, int tid1, int tid2) {
    if (tid1 == tid2||uf_find(g->uf,tid1) == uf_find(g->uf,tid2)) {
        //même classe dans g.uf
        return true;
    }
    else if (egalite_label(g->tset[tid1].label,g->tset[tid2].label)) {
        return term_is_congruent(g, g->tset[tid1].li, g->tset[tid2].li)
            && term_is_congruent(g, g->tset[tid1].ri, g->tset[tid2].ri)
    }
    else { // pas le même label
        return false;
    }
}
```

Question 36. On fusionne récursivement en utilisant le tableau des prédécesseurs.

```
void merge(egraph_t *g, int tid1, int tid2) {
    if (uf_find(g->uf,tid1) == uf_find(g->uf,tid2)) {
        // termine immédiatement
        return ;
    }
    uf_union(g->uf, tid1, tid2);
    for (int t1 = 1; t1<g->tset_sz ; t1++) {
        if (g->pre[t1*g->tset_sz+tid1]) {
        // t1 admet t1' comme sous-terme direct
            for (int t2 = 1; t2<g->tset_sz ; t2++) {
```

```
11                  if (g->pre[t2*g->tset_sz+tid2]) {
12                  // et t2 admet t1' comme sous-terme direct
13                      if (uf_find(g->uf,t1) != uf_find(g->uf,t2))
14                         && term_is_congruent(g,t1,t2) {
15                          merge(g,t1,t2);
16                      }
17                  }
18              }
19          }
20      }
21  }
```

Question 37. Dans le pire des cas, on peut avoir autant d'appels que de sommets dans le graphe car chaque appel ajoute une arête à la forêt représentant la structure unir et trouver. Ainsi, on ne peut pas dépasser `g.tset_sz-1` arêtes (arbre), donc on ne peut pas dépasser `g.tset_sz-1` appels.
On pose $t_0 = X_1$ et $t_n = f_1(X_1, f_(X_1, f(X_1, \ldots, f(X_1, X_1))))$ avec n applications de f_1 pour tout $n \geq 0$. On considère le graphe formé à partir de t_0 et t_n avec n fixé.
On a alors `g.tset_sz` $= n + 1$ car on a tous les t_i avec $i \leq n$.
On se donne la contrainte $X_1 = f(X_1, X_1)$.
On a alors les appels avec $t_2 = f_1(X_1, f_1(X_1, X_1))$ et $t_1 = f(X_1, X_1)$, puis avec t_3, t_2, etc jusqu'à t_n, t_{n-1}.
Cela montre que le maximum annoncé est atteint.

Question 38. Pour chaque appel, on a un appel à `uf_union` et deux appels à `uf_find` pour chaque couple d'arêtes entrant dans les deux sommets.
Ainsi, chaque appel est en $O(a^2 \log(\texttt{g.tset\_sz}))$ sans les appels récursif et à l'aide de la majoration de la question précédente, on obtient une complexité en $O(a^2 \texttt{g.tset\_sz} \log(\texttt{g.tset\_sz}))$