OPTION INFORMATIQUE
Devoir surveillé 3
Corrigé

1 Coloration de graphe et logique propositionnelle

1. G_1 contient un cycle de longueur impaire (par exemple 0-1-2), il n'est donc pas 2-coloriable. G_1 admet un 3-coloriage, par exemple (sous forme de tableau de couleurs indicé par les sommets)

```
[|0;1;2;1;0;2|]
```

2. Il y a nk variables distinctes, chacune pouvant recevoir deux valeurs booléennes, il y a donc 2^{nk} lignes dans une telle table de vérité.

3. (a) Cette formule affirme qu'une couleur n'est attribuée que si les couleurs précédentes ont aussi été attribuées : il n'y a pas de trou dans les couleurs effectivements attribuées à un sommet du graphe. Autrement dit, l'ensemble des couleurs attribuées à un sommet est de la forme $[\![0, k']\!]$, avec $k' < k$.

   (b) Cette formule est satisfiable, par exemple par la valuation booléeenne rendant vraie toute variable.

   (c) Cette formule n'est pas valide, il suffit pour le démontrer d'exhiber une valuation ne la satisfaisant pas. Comme $k > 1$, l'indice $j = 1$ existe. Considérons la valuation rendant $C_{0,1}$ vraie et toutes les autres variables fausses. La formule ϕ est une conjonction dont un des termes est

$$C_{0,1} \to \bigvee_{i'=0}^{n-1} C_{i',0}.$$

   Sa prémisse est vraie, mais toutes les variables $C_{i',0}$ sont fausses, donc sa conclusion est fausse : ce terme et donc ϕ sont donc faux sous cette valuation.

4.
$$\psi_1 = \bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{k-1} C_{i,j}.$$

5.
$$\psi_2 = \bigwedge_{i=0}^{n-1} \bigwedge_{0 \leq j < j' \leq k-1} (\overline{C_{i,j}} \vee \overline{C_{i,j'}}).$$

6. En notant E l'ensemble des arêtes (paires $\{i, i'\}$ de sommets voisins), pour chaque arête et chaque couleur j les deux extrémités ne peuvent porter simultanément la couleur j :

$$\psi_3 = \bigwedge_{\{i,i'\} \in E} \bigwedge_{j=0}^{k-1} (\overline{C_{i,j}} \vee \overline{C_{i',j}}).$$

7. Montrons que *G est k-coloriable* $\iff$ *$\psi_1 \wedge \psi_3$ est satisfiable.*

   ($\Rightarrow$) Supposons G k-coloriable et soit $c : [\![0, n-1]\!] \to [\![0, k-1]\!]$ un coloriage. Définissons la valuation v par

$$v(C_{i,j}) = V \iff c(i) = j.$$

   - $v \models \psi_1$: pour chaque sommet i, la variable $C_{i,c(i)}$ est vraie, donc $\bigvee_j C_{i,j}$ est vraie.
   - $v \models \psi_3$: soit $\{i, i'\} \in E$. Comme c est un coloriage, $c(i) \neq c(i')$. Pour toute couleur j, on ne peut avoir simultanément $c(i) = j$ et $c(i') = j$; donc $C_{i,j}$ et $C_{i',j}$ ne sont pas toutes deux vraies, et la clause $\overline{C_{i,j}} \vee \overline{C_{i',j}}$ est satisfaite.

Ainsi $v \models \psi_1 \wedge \psi_3$, qui est donc satisfiable.

($\Leftarrow$) Supposons $\psi_1 \wedge \psi_3$ satisfiable et soit v une valuation la satisfaisant. Pour chaque sommet i, ψ_1 garantit qu'au moins une variable $C_{i,j}$ est vraie ; posons

$$c(i) = \min\{\, j \;\mid\; v(C_{i,j}) = V \,\}.$$

On définit ainsi une fonction $c : [\![0, n-1]\!] \to [\![0, k-1]\!]$. Montrons qu'elle est un coloriage propre : soit $\{i, i'\} \in E$, et supposons par l'absurde $c(i) = c(i') = j$. Alors $v(C_{i,j}) = v(C_{i',j}) = V$, ce qui contredit la clause $\overline{C_{i,j}} \vee \overline{C_{i',j}}$ de ψ_3. Donc $c(i) \neq c(i')$: c est un k-coloriage, et G est k-coloriable.

2 Pile avec oubli

On représente une pile de capacité c par un tableau de taille $c+2$. Les c premières cases contiennent les éléments de la pile, dans l'ordre mais avec possiblement de la circularité. La case d'indice c indique la taille de la pile, et la case d'indice $c+1$ indique la position du sommet.
Par exemple, la pile $(5, 2, 1, 3)$ (où 5 est le sommet) de capacité 10 peut être représentée par le tableau `[|3;1;2;5;0;0;0;0;0;0;4;3|]`, ou également par le tableau `[|2;5;8;2;0;6;4;6;3;1;4;1|]`

```
let c = 30;;

let creer_pile () = Array.make (c+2) 0;;

let est_vide p =
    let c = Array.length p - 2 in p.(c) = 0;;

let empiler p v =
    let c = Array.length p - 2 in
    let i = p.(c+1) in
    let i' = (i + 1) mod c in
    p.(i') <- v;
    p.(c+1) <- i';
    if p.(c) < c then p.(c) <- p.(c) + 1;;

let depiler p =
    if est_vide p then failwith "depilement de pile vide" else
    let c = Array.length p - 2 in
    let i =p.(c+1) in
    let v = p.(i) in
    let i' = (i - 1) mod c in
    p.(c+1) <- i';
    p.(c) <- p.(c) - 1;
    v;;
```

3 Arbres binaires de recherche

1. V est un arbre binaire de recherche.
   $N(g, c, x, d)$ est un arbre binaire de recherche si et seulement si g est un arbre binaire de recherche, d est un arbre binaire de recherche, et $g < c < d$.
2. (a)
```
let rec valeur_associee a c =
        match a with
            V -> failwith "clé non présente"
            |N(g,c',v,d) -> if c = c' then v
                            else if c < c' then valeur_associee g c
                            else valeur associee d c;;
```

(b) Pour chaque noeud, il y a un appel récursif sur un seul fils. La complexité admet donc en fonction de la hauteur la relation de récurrence $C(h) = C(h-1) + O(1)$, d'où $C(h) = O(h)$.

3. (a)
```
let rec maptree f a =
    match a with
        V -> V
        |N(g,c,x,d) -> N(maptree f g, f c ,x , maptree f d);;
```

(b) Il faut et il suffit que f conserve l'ordre strict, autrement dit soit strictement croissante.

(c)
- Supposons f strictement croissante. Montrons par induction structurelle que pour tout arbre binaire a, si a est un arbre binaire de recherche, alors `maptree f a` est un arbre binaire de recherche :
  - Si $a = V$, alors `maptree f a` renvoie V, qui est un arbre binaire de recherche.
  - Si $a = N(g, c, x, d)$ avec g et d vérifiant la propriété, alors g' et d' images de g et d par `maptree f` sont des arbres binaires de recherche. Pour toute clé n de g, on a $n < c$,donc la clé correspondante $f(n)$ de g' vérifie $f(n) < f(c)$ par stricte croissante de f, ie $g' < f(c)$. Similairement, $f(c) < d'$. Le résultat de `maptree f a`, qui est $N(g', f(c), x, d')$, est donc bien un arbre binaire de recherche.
- Supposons que `maptree f` transforme tout arbre binaire de recherche en arbre binaire de recherche. Soient $n, m \in \mathbb{Z}$ vérifiant $n < m$. L'arbre binaire de recherche $N(V, n, 0, N(V, m, 0, V))$ a pour image par `maptree f` $N(V, f(n), 0, N(V, f(m), 0, V))$, qui doit être un arbre binaire de recherche. On en déduit donc $f(n) < f(m)$. En conclusion, f est donc bien strictement croissante

4. (a)
```
let rec scission a n =
    match a with
        V -> V,V
        |N(g,c,x,d) ->
            if c<=n then
                let d1,d2 = scission d n in
                N(g,c,x,d1),d2
            else
                let g1,g2 = scission g n in
                g1,N(g2,c,x,d)
;;
```

(b) Pour chaque noeud, il y a un appel récursif sur un seul fils. La complexité admet donc en fonction de la hauteur la relation de récurrence $C(h) = C(h-1) + O(1)$, d'où $C(h) = O(h)$.

5.
```
let rec fusion a b =
    match a with
        V -> b
        |N(g,c,x,d) ->
            let bg,bd = scission b c in
            N(fusion g bg, c,x,fusion d bd);;
```

4 Minimum exclus

1. 1

2. (a)
```
let appartient x t =
    let l = Array.length t in
    let i = ref 0 in
    while !i < l && t.(!i) <> x do
        incr i
    done;
    !i <> l;;
```

(b)
```
let mex1 t =
    let k = ref 0 in
    while appartient !k t do incr k done;
    !k;;
```

(c) `appartient` est en $O(n)$. Chaque passage dans le while est donc en $O(n)$.
Dans le meilleur cas (quand 0 n'est pas dans t), on sort du while la première fois que la condition est évaluée, d'où une complexité en $O(n)$.
Dans le pire cas (atteint quand t contient tous les éléments de 0 à son maximum), on passe $n(= m+1)$ fois dans le while, d'où une complexité en $O(n^2)$ (pas besoin de l'exprimer en fonction de m, le sujet est trompeur).

3. (a)
```
let maximum t =
    let m = ref t.(0) in
    for i = 1 to Array.length t - 1 do m := max !m t.(i) done;
    !m;;

let presences t =
    let m = maximum t in
    let est_present = Array.make (m+1) false in
    for i = 0 to Array.length t - 1 do est_present.(t.(i)) <- true done;
    est_present;;
```

(b)
```
let mex2 t =
    if Array.length t = 0 then 0 else begin
    let est_present = presences t in
    let m = Array.length est_present in
    let k = ref 0 in
    while !k <= m && est_present.(!k) do incr k done;
    !k
    end;;
```

(c) `maximum` est en $O(n)$, `presences` est en $O(n+m)$ (à cause du `Array.make (m+1) false`, initialiser un tableau a un coût linéaire), donc `mex2` est en $O(n+m)$, dans le pire comme dans le meilleur cas.

(d) D
On note A l'ensemble des entiers naturels qui ne sont pas dans t, et on pose

$$I : \forall x \in A, k \leq x$$

On montre que I est un invariant du `while` de `mex2` :

- Initialement, $k = 0$, donc I est vérifié.
- Supposons que I soit vrai au début d'une itération.
  A la fin de l'itération, en notant k' la nouvelle valeur de la variable, on a $k' = k+1$. Puisque l'itération a lieu, la condition est vérifiée, donc k est présent dans t, donc $k \notin A$. On en déduit (d'après I) $\forall x \in A, k < x$, d'où $\forall x \in A, k+1 \leq x$.
  I est donc toujours vrai en fin d'itération.
- En conclusion, I est vrai en sortie de boucle, mais on a alors $k > m$ ou `est_present(k)` faux. Dans les deux cas, $k \in A$, et d'après I, k est un minorant de A, c'est donc bien le minimum de A.

4. On n'a en fait pas besoin des valeurs (toujours égales à 0 dans la suite), on utilise ici la structure de dictionnaire comme une façon efficace de manipuler l'ensemble formé des clés.

```
let mex3 t =
    let h = Hashtbl.create 0 in
    for i = 0 to Array.length t - 1 do
        Hashtbl.add h t.(i) 0;
    done;
```

```
let k = ref 0 in
while Hashtbl.mem h !k do incr k done;
!k;;
```

La boucle `for` est en $O(n)$.

Chaque passage dans le `while` est en $O(1)$. On fait au moins un passage dans le while (quand 0 n'est pas dans t) et au plus n (quand t contient les éléments de 0 à son maximum).

Dans les deux cas, la complexité totale est en $O(n)$.