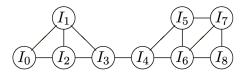
Correction du DM2: Graphes d'intervalles et coloration

1. (a) Voici la fonction demandée :

```
let conflit (a,b) (c,d) = not ((b<c)||(d<a));;
```

- (b) /
- (c) i. Voici le graphe d'intervalles pour le problème B de la figure 1 :



ii. On initialise un tableau g de bonne taille avec des listes vides. A l'aide d'une double boucle, on considère tous les couples (I_i, I_j) d'intervalles avec i < j. Quand on détecte un conflit, on construit une arête, ce qui revient à ajouter i à la liste numéro j et j à la liste numéro i (ce qui est possible puisque la structure de tableau est mutable). Voici la fonction demandée :

```
let construire_graphe t =
  let n = Array.length t in
  let g = Array.make n [] in
    for i = 0 to n-2 do
       for j = i+1 to n-1 do
       if conflit t.(i) t.(j) then
        begin
            g.(i) <- j::g.(i)
            g.(j) <- i::g.(j)
        end;
       done;
       done;
       done;
       g;;</pre>
```

(d) i. Le graphe associé au problème A ne peut être colorié avec moins de trois couleurs du fait de la présence du triangle reliant I_0, I_1, I_2 . Une 3-coloration sera donc optimale si on en trouve une. On peut choisir

De même, on a besoin d'au moins 3 couleurs pour le graphe du problème B (triangle I_0, I_1, I_2). Une coloration optimale convenable est

ii. Pour la fonction appartient :

```
let rec appartient l x = match l with
    | [] -> false
    | t::q -> (t=x) || (appartient q x)
;;
```

On remarque que cette fonction est récursive terminale grâce à l'évaluation paresseuse (si le test t=x vaut true, l'appel récursif n'est pas effectué).

iii. Pour la fonction plus_petit_absent :

On teste successivement tous les entiers en incrémentant une référence tant que sa valeur est présente.

```
let plus_petit_absent 1 =
  let i = ref 0 in
   while (appartient 1 !i) do
        i := !i+1
        done;
!i;;
```

iv. Pour la fonction couleurs_voisins :

La fonction auxiliaire locale construit : int list -> int list prend en argument une liste de sommets et renvoie la liste des couleurs de ceux-ci (ceux qui sont coloriés). Il suffit de l'appeler avec la liste des voisins de i. Cette fonction étant locale, elle connaît le tableau couleurs. L'énoncé ne précise pas si la liste résultat peut comporter des doublons (rien n'empêche un sommet d'avoir plusieurs voisins de la même couleur). La fonction suivante impose une liste résultat sans doublon (d'où le test d'appartenance).

v. Pour la fonction couleur_disponible :

Il suffit de combiner les deux fonctions précédentes.

```
let couleur_disponible aretes couleurs i =
  plus_petit_absent (couleurs_voisins aretes couleurs i);;
```

(e) i. Si G ne possède pas d'arêtes, alors (de façon immédiate)

$$\chi(G) = 1$$
 et $\omega(G) = 1$.

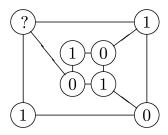
Si G est complet à n sommets alors (de façon immédiate)

$$\chi(G) = n$$
 et $\omega(G) = n$.

ii. Une bonne coloration d'un graphe G induit une bonne coloration de tout sous-graphe de G. Avec la question précédente, on en déduit que :

$$\omega(G) \leq \chi(G)$$
.

L'égalité est fausse en général. Dans le graphe suivant, il n'y a pas de triangles et donc pas de clique de taille 3. Cependant, on vérifie aisément que trois couleurs sont indispensables pour le colorier correctement. Pour le voir, on donne la couleur 0 à un sommet et on complète de manière obligatoire en n'utilisant que 0 ou 1 pour tomber sur une contradiction.



- iii. Il s'agit de voir si, pour tout élément i de xs, la liste aretes.(i) contient bien tous les autres éléments de xs. On utilise pour cela deux fonctions auxiliaires locales (qui connaîtront donc aretes):
 - tester : int list -> int -> bool : dans l'appel tester 1 i on regarde si tous les éléments de 1 différents de i sont voisins de i.
 - parcourir : int list -> bool : dans l'appel parcourir xs on teste si chaque élément de xs est relié à tous les autres.

2. (a) Dans le problème B, certaines extrémités gauches de segments sont égales. On voit sur cet exemple que l'ordre des intervalles n'est donc pas parfaitement défini. En choisissant la numérotation proposée par l'énoncé, on obtient la coloration

(b) Notons que dans la fonction demandée, l'argument segments ne sert à rien puisque tous les renseignements sont contenus dans le graphe (alternativement, on pourrait se contenter de segments à partir de qui on sait construire le graphe).

Il nous suffit de créer un tableau de couleurs de bonne taille (initialisé avec l'absence de couleur -1) et de le remplir petit à petit grâce aux fonctions de 1.(d).

```
let coloration segments aretes =
  let n = Array.lentgh aretes in
```

```
let couleur = Array.make n (-1) in
  couleurs.(0) <- 0;
  for i = 1 to n-1 do
    let c = couleur_disponible aretes couleurs i in
      couleurs.(i) <- c
    done;
couleurs ;;</pre>
```

- (c) i. Puisque I_k s'est vu attribuer la couleur c, il est en conflit avec des intervalles ayant reçu les couleurs $0, \ldots, c-1$. Avec l'ordre choisi sur les extrémités gauches des segments, ceci signifie que a_k est inférieur ou égal à au moins c des entiers b_0, \ldots, b_{k-1} . a_k appartient donc à coup sûr à au moins c des segments $I_0, \ldots I_{k-1}$.
 - ii. Considérons l'ensemble C formé des intervalles I_i tels que $i \leq k$ et $a_k \in I_i$. On vient de voir que cet ensemble est au moins de cardinal c+1 (il y a I_k et c autres des précédents intervalles). Il forme une clique. En effet, considérons i < j des éléments tels que $I_i, I_j \in C$. On a $a_i \leq a_j \leq a_k$ puisque les intervalles sont ordonnés selon la borne inférieure. De plus, $a_k \leq b_i$ (par définition de C) et donc $a_i \leq a_j \leq b_i$. Ainsi, I_i et I_j sont en conflit et donc reliés dans le graphe.
 - iii. La question 1.(e).(ii) indique alors que le nombre chromatique est plus grand que c+1.
 - iv. Quand on introduit une couleur, son numéro est toujours inférieur au nombre chromatique. Par ailleurs, la coloration partielle est toujours une bonne coloration (c'est un invariant d'itération évident puisqu'une couleur ajouté ne contredit pas la bonne coloration). On obtient donc finalement une coloration avec un nombre de couleurs inférieur au nombre chromatique. Il est égal à $\chi(G)$ par minimalité de $\chi(G)$ et c'est une coloration optimale.
- (d) La fonction appartient est de complexité O(p) où p est la longueur de la liste argument.

La fonction plus_petit_absent met en oeuvre une boucle effectuée au plus p+1 fois où p est la longueur de la liste argument (en effet, par le lemme des tiroirs, un des entiers $0, \ldots, p$ est absent de la liste). Chaque itération est de complexité O(p). Le coût total est donc $O(p^2)$.

La fonction couleurs_voisins parcourt la liste des voisins d'un sommet. A chaque élément rencontré, on effectue O(n) opérations où n est un majorant du nombre de couleurs qu'on utilisera, par exemple le nombre de sommets (c'est l'appel à appartient qui induit ce coût). On a donc une complexité $O(m_i n)$ où m_i est le nombre de voisins de i et n le nombre total de sommets.

La fonction couleur_disponible a donc un coût $O(m_i n) + O(n)$ (comme on a formé une liste de couleurs sans doublon, on appelle plus_petit_absent avec une liste d'au plus n éléments) et donc $O((m_i + 1)n)$.

La fonction coloration initialise un tableau pour un coût O(n) où n est le nombre de sommets. On fait un appel à couleur_disponible pour un coût

global
$$\sum_{i=0}^{n-1} O((m_i+1)n) = O(mn) + O(n)$$
. Le coût global est donc $O(mn) + O(n)$.