Coloration d'un graphe et polynôme chromatique : Corrigé

Partie I. Détermination des voisins des sommets

Question 1. Par exemple :

```
let rec insere q x=match q with
  | [] -> [x]
  | y::_ when y>x -> x::q
  | y::_ when y=x -> q
  | y::r -> y::insere r x
;;
```

Remarque : pas mal d'erreurs sur cette question, qu'il faut savoir faire sans hésiter. Erreurs fréquentes :

- oubli du cas de base `[]` (basiquement, une fonction sur les listes doit présenter les deux motifs `[]` et `y::r`) ;
- filtrage du motif `x::r`, avec `x` l'élément passé en paramètre de la fonction, en pensant que le `x` du motif est la valeur du paramètre : le motif de filtrage `x::r` absorbe toute liste non vide, si c'est le cas alors `x` prend localement la valeur de la tête et `r` la queue (à droite du filtrage). Le `when` permet de faire un test de valeur.
- proposition d'une solution trop compliquée à base de filtrage sur deux crans `y::z::r` ou utilisation d'un accumulateur qu'il faut retourner au bon moment (et parfois pas fait au bon moment).

Question 2. La complexité est linéaire en la longueur de `q` dans le pire cas.

Question 3. Par exemple.

```
let voisins g s =
  let rec aux ar q = match ar with
    | [] -> q
    | x::p when x.a = s -> aux p (insere q x.b)
    | x::p when x.b = s -> aux p (insere q x.a)
    | _::p -> aux p q
  in aux g.ar []
;;
```

Erreurs fréquentes :

- la non écriture d'une fonction auxiliaire : c'est possible, mais il faut alors reconstruire un graphe car `voisins` a le type `graphe -> int -> int list`. J'ai souvent vu un appel récursif avec la liste des sommets au lieu d'un graphe.
- quelque chose comme `insere q x.b ; aux p`, en espérant peut-être que `insere` « modifie » la liste `q` à la manière de Python. Une liste chaînée est *immuable*, la fonction `insere` renvoie une nouvelle liste.
- l'oubli des appels récursifs dans les cas de filtrage où `s` est présent dans l'arête.
- la difficulté d'accès aux composantes d'une arête : `fst` et `snd` ne fonctionne que sur des couples, revoir le cas échéant le paragraphe du chapitre 0 sur les types enregistrements (d'autant que la méthode d'accès était rappelée dans l'énoncé !). Si vous voulez filtrer une liste d'arêtes en accédant aux composantes, utiliser `{a=x;b=y}::r` et accès aux extrémités via `x` et `y` (mais je déconseille, le filtrage d'un type enregistrement n'est pas très naturel).

Partie II. Un algorithme de bonne coloration du graphe

Question 4. La coloration fournie par l'algorithme sur G_{ex2} est la suivante : $c(0) = c(1) = 0$, $c(2) = c(3) = 1$, $c(4) = c(5) = 2$. La coloration $c'(0) = c'(2) = c'(4) = 0$ et $c'(1) = c'(3) = c'(5) = 1$ est une autre bonne coloration de G_{ex2}, avec moins de couleurs.

Question 5. Par exemple. La fonction `coul_vois` extrait de la liste des voisins du sommet i (défini dans la boucle principale) les couleurs déja attribuées. Il suffit via une boucle `while` de regarder la plus petite couleur non attribuée.

```
let coloration g =
  let n=g.n in
  let c=Array.make n (-1) in
  for i=0 to n-1 do
    let rec coul_vois q = match q with
      | [] -> []
      | x::p -> if c.(x) > -1 then c.(x)::coul_vois p else coul_vois p
    in let cvi = coul_vois (voisins g i) in
       let x=ref 0 in
       while List.mem !x cvi do
         incr x
       done ;
       c.(i) <- !x
  done ;
  c
;;
```

Remarque : dans le pire cas, la complexité est $O(n^3)$ avec n le nombre de sommets de `g`. On peut faire un peu mieux, la fonction suivante possède une complexité réduite à $O(n^2)$:

```
let coloration g =
  let n=g.n in
  let c = Array.make n (-1) in
  for i=0 to n-1 do
    let prises = Array.make n false in
    List.iter (function x -> if c.(x) <> -1 then prises.(c.(x)) <- true) (voisins g i) ;
    let j=ref 0 in
    while prises.( !j) do incr j done ;
    c.(i) <- !j
  done ;
  c
;;
```

(`List.iter f t` applique une fonction de type `'a -> unit` sur un `'a array`).
Remarque : cette question était la plus longue du sujet (en écriture du code). Des solutions consistants à chercher la plus grande couleur des voisins et lui ajouter/retrancher 1 ne fonctionnaient pas.

Partie III. Définition du nombre chromatique de G

Question 6. L'ensemble $\mathrm{BC}(G, n(G))$ est non vide, car on peut colorer les sommets avec les $n(G)$ couleurs distinctes $[\![0, n(G)-1]\!]$ pour obtenir une bonne $n(G)$-coloration. De plus, si un graphe G possède une bonne p-coloration, c'est en particulier une bonne $(p+1)$-coloration. Ainsi, si $\mathrm{fc}(G,p)$ non nul, alors $\mathrm{fc}(G,p+1)$ l'est également. Par suite on a l'implication $p \in \mathrm{EC}(G) \Rightarrow p+1 \in \mathrm{EC}(G)$. D'où l'existence d'un unique entier $\mathrm{nbc}(G)$ tel que :

$$\mathrm{EC}(G) = \{p \in \mathbb{N}^* \mid p \geq \mathrm{nbc}(G)\}$$

Question 7. Si G n'a pas d'arête, toute coloration de G est une bonne coloration, en particulier la fonction constante $c(s) = 0$ pour tout $s \in G$. Ainsi, $\mathrm{nbc}(G) = 1$. De plus, $\mathrm{BC}(G,p) = \{f \text{ fonction de } [\![0, n(G)-1]\!] \text{ dans } [\![0, p-1]\!]\}$, de cardinal $\mathrm{fc}(G,p) = p^{n(G)}$.

Remarque : les deux questions suivantes vous ont posé des difficultés pour dénombrer, c'est normal vous n'avez pas encore fait beaucoup de dénombrement en mathématiques.

Question 8. Au contraire, si G possède toute paire de sommets comme arête, toute coloration du graphe doit posséder au moins $n(G)$ couleurs, car les couleurs attribuées aux sommets doivent être distinctes. Ainsi, $\mathrm{nbc}(G) = n(G)$. Pour choisir une bonne p-coloration (pour $p \geq n(G)$), il suffit de choisir les n différentes couleurs attribuées (il y a $\binom{p}{n(G)}$ choix), et attribuer à chaque sommet une couleur ($n(G)!$ possibilités). D'où

$$\mathrm{fc}(G,p) = \binom{p}{n(G)} n(G)! = \frac{p!}{(p-n(G))!} = p(p-1)\cdots(p-n(G)+1)$$

(formule compatible pour $p < n(G)$).

Question 9. Puisque G_{ex1} possède 3 sommets deux à deux connectés, $\text{nbc}(G_{ex1}) \geq 3$. De plus, la 3-coloration $c(0) = c(1) = c(2) = 0$, $c(3) = 1$, $c(4) = 2$ est une bonne 3-coloration, donc $\text{nbc}(G_{ex1}) = 3$. Dénombrons les bonnes p-colorations pour $p \geq 3$. Pour choisir une telle coloration, il suffit :

— de choisir 3 couleurs parmi les p, et de les attribuer à 0, 3, 4 ($\binom{p}{3}3!$ possibilités) ;

— choisir pour 1 une couleur différente de celle attribuée à 3 ($p-1$ possibilités) ;

— choisir une couleur quelconque pour 2 (p possibilités).

D'où

$$\text{fc}(G_{ex1}, p) = \binom{p}{3}3! \times (p-1) \times p = p^2(p-1)^2(p-2)$$

(formule également compatible pour $p < 3$).

Partie IV. Les applications H et K

Question 10. Simplement :

```
let prem_voisin g s =
  List.hd (voisins g s)
;;
```

Question 11. Par exemple en utilisant `voisins` :

```
let prem_ni g=
  let i=ref 0 in
  while voisins g !i =  [] do incr i done ;
  !i
;;
```

Ou en parcourant une seule fois la liste des arêtes, ce qui est meilleur en complexité :

```
let prem_ni g =
  let rec aux mini q = match q with
    | [] -> mini
    | x::r -> aux (min mini (min x.a x.b)) r
  in aux g.n g.ar
;;
```

(`mini` stocke le plus petit sommet à l'extrémité d'une arête vu pour le moment, il suffit de prendre `g.n` pour l'initialisation).
Remarque : beaucoup d'entre vous m'ont proposé une solution semblable à la deuxième, mais se contentait d'une arête plus petite que le minimum temporaire, sans regarder l'autre, ce qui n'allait pas.

Question 12. Il suffit de supprimer de la liste d'arêtes celles de la forme $\{s_1, s_2\}$:

```
let h g=
  let nb=g.n and s1=prem_ni g in
  let s2=prem_voisin g s1 in
  let rec aux acc q=match q with
    | [] -> {n = nb; ar = acc}
    | x::r when x.a = s1 && x.b = s2 || x.b = s1 && x.a = s2 -> aux acc r
    | x::r -> aux (x::acc) r
  in aux [] g.ar
;;
```

Remarque : beaucoup n'ont pas sûr reconstruire un graphe une fois la liste d'arêtes construites, d'autres ont cru que l'on pouvait modifier la liste d'arête d'un graphe (ce qui n'est pas possible, le champ `ar` n'étant pas mutable).

Question 13. On introduit au préalable une fonction `change_extremite` prenant en entrée l'extrémité d'une arête et lui appliquant la discussion de l'énoncé.

```
let change_extremite x s1 s2=
  if x<s2 then x
  else if x=s2 then s1
  else x-1
;;

let k g =
  let nb=g.n and s1=prem_ni g in
  let s2=prem_voisin g s1 and hg = h g in
  let rec aux acc q = match q with
    | [] -> {n=nb-1; ar = acc}
    | x::r -> aux ({a=change_extremite x.a s1 s2; b=change_extremite x.b s1 s2}::acc) r
  in aux [] hg.ar
;;
```

Partie V. Fonction fc(G, p) et polynôme chromatique

Question 14. $H(G)$ est obtenu à partir de G en supprimant des arêtes. Par suite, une bonne coloration de G en est une de $H(G)$. En particulier : $\mathrm{BC}(G, p) \subset \mathrm{BC}(H(G), p)$.

Question 15. Les deux cardinaux sont égaux, comme on peut l'imaginer vu l'égalité à démontrer ! Exhibons une bijection entre les colorations de $K(G)$ et celles de $H(G)$, qui ne sont pas des colorations de G. Pour simplifier la démonstration, on suppose simplement que $K(G)$ possède les mêmes sommets que G, à l'exception de x', sommet de G qui a été supprimé, ainsi que toutes ses arêtes le liant au sommet x de G.

— Si c est une bonne coloration de $K(G)$, la coloration obtenue en attribuant à x' la même couleur que x (disons qu'on obtient c') est une coloration qui n'est pas une bonne coloration de G (car x et x' sont reliés dans G), mais c'est une bonne coloration de $H(G)$.

— Réciproquement, une bonne coloration de $H(G)$ qui n'est pas une bonne coloration de G associe nécessairement les mêmes couleurs à x et x' : on en déduit par restriction à $S(K(G))$ une bonne coloration de $K(G)$.

Les deux applications ainsi construites sont clairement réciproques l'une de l'autre, et les colorations associées ont même nombre de couleurs. On a donc établi une bijection entre $\mathrm{BC}(K(G), p)$ et $\mathrm{BC}(H(G), p) \backslash \mathrm{BC}(G, p)$, d'où l'égalité

$$\mathrm{fc}(G, p) = \mathrm{fc}(H(G), p) - \mathrm{fc}(K(G), p)$$

Question 16. On peut appliquer l'algorithme suivant pour calculer $\mathrm{fc}(G, p)$:

— si G ne possède aucune arête, alors $\mathrm{fc}(G, p) = p^{n(G)}$ d'après la question 7.

— sinon, calculer $K(G)$ et $H(G)$, calculer récursivement $\mathrm{fc}(K(G), p)$ et $\mathrm{fc}(H(G), p)$, et en déduire $\mathrm{fc}(G, p) = \mathrm{fc}(H(G), p) - \mathrm{fc}(K(G), p)$.

Pour justifier la terminaison, remarquons que la fonction effectue au plus deux appels récursifs, sur des graphes G' tels que $n(G') + |A(G')| < n(G) + |A(G)|$. Elle termine donc.

Question 17. C'est immédiat par récurrence forte sur la quantité précédente :

— si G ne possède pas d'arête, alors $\mathrm{fc}(G, p) = p^{n(G)}$ est bien polynomiale en p ;

— sinon, par hypothèse de récurrence, $\mathrm{fc}(K(G), p)$ et $\mathrm{fc}(H(G), p)$ sont polynomiales en p, et par différence $\mathrm{fc}(G, p)$ également.

Partie VI. Calcul du polynôme $P_C(G)$ et de nbc(G)

Question 18.

```
let difference p q =
  let diff = Array.copy p in
  for i=0 to Array.length q - 1 do
    diff.(i) <- diff.(i) - q.(i)
  done ;
  diff
;;
```

Question 19. On applique simplement l'algorithme évoqué en question 16.

```
let rec pc g=
  let n=g.n in
  if g.ar = [] then
    let p=Array.make (n+1) 0 in
    p.(n) <- 1 ;
    p
  else
    difference (pc (h g)) (pc (k g))
;;
```

Question 20. C'est classique.

```
let eval p x=
  let s=ref 0 in
  let puiss = ref 1 in
  for i=0 to Array.length p - 1 do
    s:= !s + p.(i) * !puiss ;
    puiss := !puiss * x
  done ;
  !s
;;
```

Question 21. Il suffit de chercher la première valeur p telle que $P_c(p) \neq 0$.

```
let nbc g =
  let pol = pc g in
  let x=ref 1 in
  while eval pol !x  = 0 do incr x done ;
  !x
;;
```