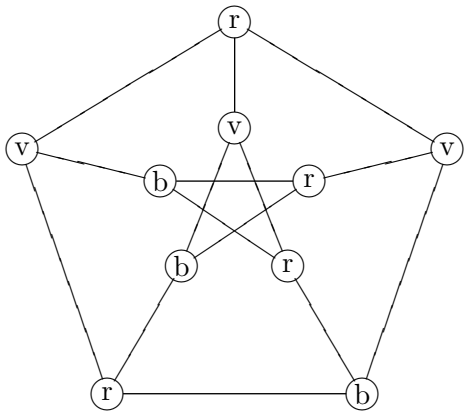
X-ENS 2018 : option informatique
Un corrigé

Dans tout le sujet, on supposera qu'un arête est composée de deux sommets distincts, c'est à dire que le graphe ne comporte pas de "bouclette" (ce qui n'est pas supposé a priori par l'énoncé mais semble raisonnable si on veut qu'il existe un coloriage).

1 Coloriage

1. *Telle que cette question est posée, on doit répondre oui pour les deux graphes. En fait, tout graphe fini est colorié puisqu'il admet un n-coloriage où n est le nombre de ses sommets (ceci sous l'hypothèse d'absence de bouclette).*

   L'étiquetage proposé du graphe de gauche n'est pas un coloriage car deux sommets adjacents sont rouges (et donc de même couleur).
   L'étiquetage proposé du graphe de droite est un coloriage.
2. Le graphe de Petersen possédant un cycle de longueur 5 $(0, 4, 3, 9, 6, 0)$ et donc impaire, il ne peut être 2-colorié. On peut exhiber une 3-coloration (on note r,v,b les trois couleurs)



   On en concut que le graphe de Petersen est de nombre chromatique 3.
3. On va tester successivement chaque sommet en comparant sa couleur à celle de ses voisins. Comme le graphe est non orienté, on peut même se contenter de comparer un sommet et ses voisins de numéro plus grand.
   La première fonction `test : graphe → coloriage → int → bool` effectue le test pour un sommet du graphe (on suppose que la taille de l'étiquetage est assez grande et que l'entier fourni est bien un numéro de sommet). On utilise une boucle conditionnelle pour arrêter de tester dès qu'un problème est détecté.

```
let test gphe etiq i =
   let n=Array.length gphe in
   let j=ref (i+1) in.  (*prochain voisin envisageable*)
   while !j<n && (not gphe.(i).(!j) || etiq.(i)<>etiq.(!j)) do
      incr j
      done;
   !j=n;;  (*tous les voisins ont \'et\'e test\'es avec succ\`es*)
```

   Il suffit alors de tester successivement tous les sommets (après avoir vérifié les tailles des arguments).

```
let est_col gphe etiq =
   let n=Array.length gphe in
   if Array.length etiq<>n then false
   else begin
      let i=ref 0 in
      while !i<n && test gphe etiq !i do incr i done;
      !i=n
      end;;
```

La première fonction est immédiatement de complexité $O(|S|)$. La seconde l'appelle $|S|$ fois au pire et on a bien une complexité $O(|S|^2)$.

4. On note $n = |S|$. Un algorithme naïf consiste à tester tous les k-coloriages pour $k = 1, 2, \ldots, n-1$ en s'arrêtant quand on trouve un coloriage adéquat. On peut s'arrêter à $n-1$ car à coup sûr un n-coloriage convient.
   Le nombre de k-coloriages est égal à k^n. On effectue donc au plus

$$\sum_{k=1}^{n-1} k^n \leq n^{n+1}$$

   tests de bon coloriage. Chaque coloriage est testé en $O(n^2)$ pour un coût total majoré par $O(n^{n+3}) = O(2^{n^2})$. Ainsi, on peut trouver le nombre chromatique en temps exponentiel.

2 2-coloriage

5. Supposons G biparti et (U, T) une partition de S associée. En donnant au sommets de U la couleur 0 et à ceux de T la couleur 1, on obtient une bonne 2-coloration.
   Réciproquement, supposons que l'on dispose d'une bonne 2-coloration. Si cette coloration utilise une unique couleur, le graphe ne possède aucune arête et toute partition de S montre que G est biparti (une telle partition existe dès que le graphe possède au moins deux sommets). Sinon, on note U l'ensemble des sommets de la première couleur et T celui des sommets de l'autre couleur. U et T sont vides, disjoints, et recouvrent S. Ils forment une partition de S. Par construction, les arêtes relient un élément de T et un autre de U. Le graphe est ainsi biparti.

6. Il s'agit en fait d'effectuer un parcours en profondeur à partir de chaque sommet successivement quand ils ne sont pas déjà colorié. On va donc avoir une boucle condition (celle d'indice i ci-dessous) qui, quand i n'est pas colorié, affecte la couleur 0 à i et lance le pracours depuis i.
   C'est la fonction auxiliaire `appel : int → unit` qui effectue le parcours en modifiant le tableau de coloriage : pour chaque voisin j de i, soit il est colorié de la mauvaise couleur et on s'arrête (levée d'exception), soit il n'est pas colorié et on le colorie à l'inverse de i et on lance récursivement un parcours depuis j.
   On rappelle que l'on a choisi de lever une exception dans le cas d'un graphe non biparti (on aurait pu utiliser `failwith`).

```
exception Col2;;

let deux_col gphe =
   let n=Array.length gphe in
   let etiq=Array.make n (-1) in

   let rec appel i=
      for j=0 to n-1 do
         if gphe.(i).(j) && etiq.(j)=etiq.(i) then raise Col2
         else if gphe.(i).(j) && etiq.(j)=(-1) then
             begin
             etiq.(j) <- 1-etiq.(i) ;
             appel j
             end;
         done

in for i=0 to n-1 do
     if etiq.(i)=(-1) then begin
         etiq.(i) <- 0 ;
         appel i
         end;
      done;
   etiq;;
```

3 Algorithmes gloutons

7. Pour la première numérotation on obtient le 3-coloriage $[|0;0;0;0;1;1;1;2;2;2|]$.
   Pour la seconde numérotation on obtient le 4-coloriage $[|0;3;0;2;1;1;1;0;2;3|]$.

8. Pour respecter la contrainte de complexité, j'adopte une stratégie de marquage. Le tableau `m` est tel que `m.(c)` vaut `true` si et seulement si on a rencontré un voisin de `s` de couleur `c`. Quand ce tableau est rempli, il reste à trouver le numéro de la première case valant `false` (il y en a forcément une car un sommet a au plus $n-1$ voisins).

```
let min_couleur_possible gphe etiq s =
    let n=Array.length gphe in
    let m=Array.make n false in
    for t=0 to n-1 do
       if gphe.(s).(t) && etiq.(t)<>(-1) then
            m.(etiq.(t)) <- true
       done ;
    let c=ref 0 in
    while m.(!c) do incr c done;
    !c;;
```

   La complexité est bien $O(n)$ (deux boucles effectuées au plus n fois avec chaque itération en temps constant). Bien sûr, on a aussi une complexité spatiale $O(n)$.

9. On applique scrupuleusement l'algorithme sans se préoccuper d'une éventuelle fausse numérotation. En utilisant un rattrapage d'exception, on renvoie le tableau de taille n avec uniquement des -1 si un problème est survenu.

```
let glouton gphe num =
   let n=Array.length gphe in
   let etiq=Array.make n (-1) in
   try
     for i=0 to n-1 do
        etiq.(num.(i)) <- min_couleur_possible gphe etiq num.(i)
        done ;
     etiq
   with
     _ -> Array.make n (-1);;
```

On fait n appels à la fonction de la question précédente pour un coût $O(n^2)$. Les autres opérations (en particulier, la création et la modification de `etiq`) a un coût $O(n)$ qui est négligeable.

10. On montre que la proposition "Au début de l'étape i, `etiq` est un coloriage du sous-graphe induit par les i premiers sommets du graphe dans l'ordre donné" est un invariant de boucle (la numérotation commençant à 0).
    - C'est vrai initialement puisqu'on a alors un sous-graphe vide.
    - Supposons le résultat vrai à une étape $i \leq n-1$. L'étape numéro i revient à prendre en compte le sommet de numéro i et on lui donne une couleur compatible avec les précédents. La propriété est donc conservée.

    En fin de boucle, l'étiquetage obtenu est ainsi une coloration du graphe entier.
    Par ailleurs, quand on attribue une couleur à un sommet, on exclut un nombre de couleurs égal au plus au degré du sommet étudié. Ce sommet se voit donc attribuer au pire la couleur d (on numéroté les couleurs à partir de 0). Ainsi, le coloriage obtenu comporte au plus $d+1$ couleurs (numéros entre 0 et d).
11. Soit L un coloriage du graphe utilisant k couleurs $c_0 < \cdots < c_{k-1}$. On choisit une numérotation de G en commençant par les sommets de couleur c_0, puis ceux de couleurs c_1 etc. jusqu'à numéroter les sommets de couleur c_{k-1}.
    Quand on applique l'algorithme glouton avec cette numérotation,
    - les sommets de couleur initiale c_0 se voient affecter la couleur 0 (aucune arête entre deux tels sommets)
    - les sommets de couleur initiale c_1 se voient affecter la couleur 0 ou 1 (il ne seront pas reliés à un sommet ayant déjà la couleur 1 car deux sommets de couleur initiale c_1 ne sont pas reliés).
    - plus généralement, les sommets de couleur initiale c_i se voient affecter une couleur de numéro $\leq i$.

    Ainsi, si $s \in S$ et $L(s) = c_i$ alors $L'(s) \leq i \leq c_i$ ($i \leq c_i$ se prouve par récurrence immédiate en partant de $c_0 \geq 0$).
12. L'algorithme glouton ayant une complexité $O(n^2)$, il ne sert à priori à rien d'écrire un tri optimal de complexité $O(n\ln(n))$. Je propose donc une stratégie par insertion. On commence par écrire une fonction `graphe` $\rightarrow$ `int array` qui nous donnera le tableau des degrés des sommets.

```
let degre gphe =
   let n=Array.length gphe in
   let d=Array.make n 0 in
   for i=0 to n-1 do
      for j=0 to n-1 do
         if gphe.(i).(j) then d.(i) <- d.(i) + 1
         done ;
      done;
   d;;
```

Dans la stratégie par insertion, on insère successivement les sommets $1, \ldots, n-1$ dans la table `num` triée par degré décroissant et contenant $0, \ldots, s-1$. Une première fonction `int array` $\rightarrow$ `int` $\rightarrow$ `int array` $\rightarrow$ `int` prend en argument le tableau d'ordre en construction, le sommet à insérer, le tableau des degré et renvoie la position où l'insertion doit se faire.

```
let indice num s deg =
   let j=ref 0 in
   while !j<s && deg.(num.(!j))>deg.(s) do incr j done;
   !j;;
```

Pour les sommets succesifs, on calcule la position d'insertion et on décale les éléments pour pouvoir mettre en place le sommet étudié.

```
let tri_degre gphe =
   let deg=degre gphe in
   let n=Array.length gphe in
   let num=Array.make n 0 in
   for s=1 to n-1 do
      let pos=indice num s deg in
      for t=s downto pos+1 do
         num.(t) <- num.(t-1)
         done;
      num.(pos) <- s
      done ;
   num;;
```

La fonction principale s'en déduit immédiatement

```
let welsh_powell gphe =
   glouton gphe (tri_degre gphe) ;;
```

4 Algorithme de Widgerson

13. On suppose que G est $(k+1)$-coloriable avec $k \geq 1$ et on se donne s un sommet de G et $G' = (V(s), A')$ le sous graphe induit par $V(s)$.
Notons L une $(k+1)$-coloration de G. L'ensemble $\{L(t)/\ t \in V(s)\}$ des ouleurs des voisins de s est alors de cardinal $\leq k$. En effet, il ne contient pas $L(s)$ (par définition d'une coloration) et est inclus dans un ensemble de cardinal $\leq k+1$ qui contient $L(s)$ (définition d'une $(k+1)$ coloration). Ainsi, la restriction de L à $V(s)$ définit une k-coloration de G'.

14. Montrons que pour la boucle de l'étape (2), on a l'invariant suivant : "au début de l'étape numéro i, le sous-graphe de G induit par les sommets déjà coloriés est bien colorié avec des couleurs $\leq c-1$ et $c \leq 2i$".
    - La propriété est vraie pour $i = 0$ (aucun sommet colorié et $c = 0$).
    - Supposons le résultat à une certaine étape i et supposons que la boucle s'effectue une fois de plus. Les couleurs que l'on va utiliser sont c et $c+1$ et ne sont pas déjà utilisées. On va donc garder une bonne coloration. De plus, les couleurs qui seront utilisées auront un numéro plus petit que l'ancien c plus le nombre de nouvelles couleurs et restera plus petit que le nouveau c moins 1. Et comme c augmente au plus de 2, le nouveau c sera inférieur à $2(i+1)$.

    Dans l'étape 3, on va utiliser de nouvelles couleurs ($\geq c$) et on gardera une bonne coloration (principe valable de l'algorithme glouton pour les nouveaux sommets coloriés et pas de contradiction avec les anciens sommets coloriés puisque l'on utilise de nouvelles couleurs).

On a montré que $\boxed{\text{l'agorithme renvoie une bonne coloration}}$.
L'étape 2 s'effectue au plus de l'ordre de $\sqrt{n}$ fois puisque chaque étape colorie $\sqrt{n}$ sommets et chaque étape utilise deux couleurs en plus au maximum. Cette étape crée donc $O(\sqrt{n})$ couleurs.
Dans l'étape 3, on opère sur un graphe de degré $\leq \sqrt{n}$ et on utilise encore au plus de l'ordre de $\sqrt{n}$ couleurs. On utilise donc bien au total $O(\sqrt{n})$ couleurs.
On a montré que $\boxed{\text{l'agorithme crée } O(\sqrt{n}) \text{ couleurs}}$.

15. Cette fois, on ne préoccupe pas des errreurs possibles et on SUPPOSE que la précondition sur `sg` évoquée par l'énoncé est vérifiée.
L'énoncé ne permet pas d'utiliser `Array.make_matrix` et on doit commencer par une boucle pour initialiser la matrice à renvoyer (problème des identités physiques si on écrit `Array.make nsg (Array.make nsg false)`).
A ceci près, le code est immédiat.

```
let sous_graphe gphe sg =
   let nsg=Array.length sg in
   let m=Array.make nsg [|false|] in
   for i=0 to nsg-1 do
      m.(i) <- Array.make nsg false
      done;
   for s=0 to nsg-1 do
      for t=0 to nsg-1 do
         if gphe.(sg.(s)).(sg.(t)) then m.(s).(t) <- true
         done;
      done ;
   m;;
```

16. On gère une référence de liste que l'on fait grossir quand on rencontre un voisin non colorié.

```
let voisins_non_colories gphe etiq s =
   let l=ref [] in
   for t=0 to (Array.length gphe)-1 do
      if gphe.(s).(t) && etiq.(t)=(-1) then l:=t::(!l)
      done;
   !l;;
```

Il suffit de calculer la longueur de la liste obtenue.

```
let degre_non_colorie gphe etiq s =
    List.length (voisins_non_colories gphe etiq s);;
```

17. On gère à nouveau une référence de liste.

```
let non_colories gphe etiq =
   let l=ref [] in
   for s=0 to (Array.length gphe)-1 do
      if etiq.(s)=(-1) then l:=s::(!l)
      done;
   !l;;
```

18. Une première fonction servira pour l'étape (2). Elle donne le plus petit entier plus grand que $\sqrt{n}$.

```
let racine n =
   let p=ref 1 in
   while (!p)*(!p)<n do incr p done;
   !p;;
```

On va procéder exactement comme le stipule l'algorithme, ce qui nous assure que les propriétés de la question 14 seront vérifiées.
Après les initialisations évidentes, l'étape (2) correspond à la première boucle inconditionnelle. Il n'est pas gênant de traiter les sommets dans l'ordre car le nombre de sommets coloriés ne fait qu'augmenter. Ainsi, si à un moment un sommet n'a pas assez de voisins coloriés, il en sera toujours de même.
Dans cette boucle, on commence par construire, pour un `s` donné le tableau `sg` de ses voisins non coloriés. Si celui-ci est assez gros, on appelle la fonction de 2-coloration avec le sous-graphe induit. On obtient un tableau `col`. Chaque sommet `i` coloré par $\alpha \in \{0, 1\}$ dans le sous-graphe correspond au sommet `sg.(i)` du graphe et doit être colorié avec `c`$+\alpha$. La réfence `plus` sert à savoircombien de couleurs on ajoute à l'étape 2(a).

Pour l'étape (3), c'est similaire mais on fait appel à l'algorithme glouton. Il donne un coloriage de sous-graphe et il faut décaler les couleurs pour colorier le graphe.

```
let widgerson gphe =
   let n= Array.length gphe in
   let rn=racine n in
   let c=ref 0 in
   let etiq=Array.make n (-1) in

   for s=0 to n-1 do
      let sg=Array.of_list (voisins_non_colories gphe etiq s) in
      let nsg=Array.length sg in
      if nsg>=rn then
         begin
         let col=deux_col (sous_graphe gphe sg) in
         let plus=ref 1 in
         for i=0 to nsg-1 do
            etiq.(sg.(i)) <- !c+col.(i) ;
            if col.(i)=1 then plus:=2;
            done;
         c:= !c + !plus
         end ;
      done;

    let sg=Array.of_list (non_colories gphe etiq) in
    let nsg=Array.length sg in
    let col=glouton (sous_graphe gphe sg) (range nsg) in
    for i=0 to nsg-1 do etiq.(sg.(i)) <- !c + col.(i) done;
    etiq;;
```

Toutes les fonctions utilisées dans la boucle sont de complexité polynomiale et comme la boucle est effectuée n fois, la première boucle est de complexité polynomiale. La seconde partie a aussi, comme `glouton` une complexité polynomiale.

19. L'idée est de se ramener à des sous-graphes coloriables avec une couleur de moins. Un processus récurrent permettrait d'adapter l'algorithme de Widgerson (? ?).