INF 431, COMPOSITION D'INFORMATIQUE

Luc Maranget

25 avril 2006

⋆⋆⋆

La duchesse de Valombreuse est chargée de dresser le plan des tables du dîner annuel de l'amicale du club de polo de Lyons-la-forêt. La duchesse souhaite faire du dîner le succès majeur de la saison, mais les membres de l'amicale ne s'entendent pas forcément très bien entre eux. La duchesse a donc demandé discrètement à chaque membre de l'amicale la liste de ceux qu'il ne souhaite pas voir à sa table. Elle est dépassée par le reste de sa tâche. Elle vous demande (car vous vous êtes imprudemment promené à Lyons en grand-uniforme !) de dresser le plan du dîner.

Partie I. Fabriquer un graphe

Afin de préserver la quiétude de la vie mondaine de Lyons-la-forêt, la duchesse a identifié chacun des membres de l'amicale par un entier entre 0 et $n-1$. Elle vous donne ensuite l'état des relations sociales sous la forme d'un fichier texte.

- La première ligne du fichier donne le nombre n de membres de l'amicale. Elle se termine par le caractère retour à la ligne « `'\n'` ».
- Les lignes suivantes sont organisées ainsi :
  - Chacune de ces lignes commence par un entier i, suivi du caractère « `':'` »,
  - ensuite vient la liste des membres avec qui le membre i refuse de dîner, sous la forme d'une suite d'entiers séparés par des virgules « `','` » et terminée par le caractère retour à la ligne « `'\n'` ». Cette suite peut être vide.

Par exemple, voici une liste possible.

```
4
0:1,2
3:
1:0,3
2:3
```

Question 1. Cette question demande diverses expressions régulières. On utilisera *impérativement* la syntaxe suivante : les motifs caractères sont donnés en syntaxe Java (`'0'`, `'1'`, ..., `':'`, `'\n'`, etc.), le motif vide est noté ϵ, la séquence par la juxtaposition E_1E_2, l'alternative par l'opérateur binaire $E_1 \mid E_2$, la répétition par l'étoile $E^\star$, et on utilise systématiquement les parenthèses pour lever les ambiguïtés. Une fois nommée une expression régulière, on pourra utiliser son nom à la place de celle-ci.

a) Donner l'expression régulière D qui décrit un chiffre, puis l'expression régulière I qui décrit l'écriture en base 10 d'un entier positif ou nul.

Réponse : Le plus simple :

$$D = \texttt{'0'} \mid \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'}, \quad I = D(D)^\star$$

Si on souhaite éviter les zéro initiaux inutiles, ce qui n'est pas l'usage en informatique, on écrira plutôt :

$$D_1 = \texttt{'1'} \mid \texttt{'2'} \mid \texttt{'3'} \mid \texttt{'4'} \mid \texttt{'5'} \mid \texttt{'6'} \mid \texttt{'7'} \mid \texttt{'8'} \mid \texttt{'9'}, \quad I = (D_1(D)^\star) \mid \texttt{'0'}$$

Note : Une erreur fréquente a été de poser $I = (D)^\star$, qui ne convient pas, puisque le mot vide n'est pas l'écriture d'un entier. □

b) Donner l'expression régulière L qui décrit les lignes du fichier à partir de la seconde.

Réponse :

$$L = I\texttt{':'}(((I\texttt{','})^\star I)|\epsilon)\texttt{'\textbackslash n'}$$

Note : Le nombre de copies où j'ai trouvé D à la place de I dans les sous-questions b) et c) est proprement incroyable (non-sanctionné). □

c) Donner l'expression régulière F qui décrit le fichier de la duchesse.

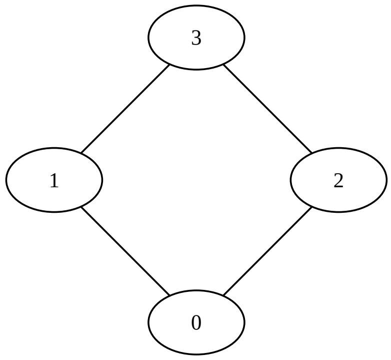
Réponse :

$$F = I\texttt{'\textbackslash n'}(L)^\star$$

□

Que le membre i refuse de dîner avec j ou que le membre j refuse de dîner avec i conduit au même résultat : les membres i et j ne peuvent pas dîner à la même table. Pour organiser le dîner, on introduit un graphe (non-orienté) (S, A), dont l'ensemble des sommets S est $0, 1, \ldots, n-1$ et l'ensemble des arêtes est l'ensemble des ensembles de la forme $\{i, j\}$, tels que i et j ne peuvent pas dîner à la même table. Par exemple, pour la liste de la duchesse déjà donnée, le graphe est donné par la figure 1. Il est important de remarquer que les arêtes sont ici des *ensembles* de deux

Figure 1: Un premier graphe



sommets. C'est-à-dire qu'il existe au plus une arête entre deux sommets distincts et qu'il n'y a pas d'arête reliant un sommet à lui même — ce qui est logique un membre de l'amicale qui refuse de s'asseoir à la même table que lui-même ne viendra pas au dîner. On notera une arête entre deux sommets s_0 et s_1, indifféremment comme $s_0 \leftrightarrow s_1$ ou $s_1 \leftrightarrow s_0$.

Les graphes sont représentés par des objets de la classe **Graphe**, partiellement spécifiée ci-dessous :

```
class Graphe {
  int n ; // Nombre de sommets du graphes
  Graphe (int n) ; // Fabriquer un graphe à n sommets et sans arêtes
  void ajouterArc(int i, int j) ; // Créer l'arête i ↔ j
}
```

Question 2. La liste de la duchesse est maintenant disponible en machine sous la forme d'un tableau `t` de tableaux d'entiers. Le tableau `t` est de taille n et chacun de ses éléments `t[`i`]` est un tableau qui regroupe ceux avec qui i refuse de dîner. Ainsi, dans le cas de notre exemple, `t[0]` est le tableau `{1, 2}`, `t[1]` est le tableau `{0, 3}`, `t[2]` est le tableau `{3}`, et `t[3]` est le tableau vide `{}`. Écrire la méthode `static Graphe fabriquer(int [] [] t)` qui prend le tableau `t` en argument et renvoie le graphe correspondant.

Attention : Il faut créer une arête donnée $i \leftrightarrow j$ au plus une fois, sachant bien que les notations $i \leftrightarrow j$ et $j \leftrightarrow i$ représentent la même arête.

Réponse :

```
static Graphe fabriquer(int [] [] t) {
  int n = t.length ;
  Graphe g = new Graphe(n) ;
  boolean [][] mat = new boolean [n][n] ;
  // NB: les cases de mat sont initialisées à false

  for (int i = 0 ; i < n ; i++) {
    int [] tt = t[i] ;
    for (int k = 0 ; k < tt.length ; k++) {
      int j = tt[k] ;
      if (!mat[i][j]) {
        mat[i][j] = mat[j][i] = true ;
        // Ou bien mat[i][j] = true ; mat[j][i] = true ;
        g.ajouterArc(i,j) ;
      }
    }
  }
  return g ;
}
```

On notera que la solution construit la représentation matricielle `mat` du graphe. Comme le graphe est non-orienté, cette matrice est symétrique. Dans cette représentation matricielle, le fameux `g.existeArc(i,j)` manquant correspond exactement au test `mat[i][j]`.

Note : Cette question a souvent été ratée. Il fallait bien voir que...

- On a parfaitement le droit de considérer que les cases de **new boolean**`[n][n]` valent **false** , car la norme de Java le spécifie. Je dirais même que c'est une bonne pratique que de s'appuyer sur la norme.
- Il n'y a aucun piège dans l'idée d'un tableau de tableaux, `t[`i`]` est un tableau, dont la taille est `t[`i`].length` et dont un élément est `t[`i`][`k`]` — à lire comme `(t[`i`])[`k`]`.

  Et d'ailleurs, en Java, une « matrice » est toujours un tableau de tableaux. Écrire par exemple **new boolean**`[n][n]` le cache un peu. Mais en fait, ce constructeur

  - alloue un premier tableau (de tableaux), de taille n,
  - et initialise chaque case du tableau ci-dessus à un tableau à chaque fois nouveau et dont les n cases valent initialement **false** .
- Le tableau vide `{}` n'est pas **null**, mais un tableau dont la taille est zéro.
- Très souvent l'indice dans `t[i]` (`k` ici) a remplacé la valeur (`t[i][k]` ici). Le code, en posant `j = t[i][k]` évite de répéter quatre fois `t[i][k]`, et surtout, il rend l'erreur moins probable.
- Il n'y a pas de méthode `existeArc`. Si on en voulait absolument une, on pouvait la programmer (avec la méthode `voisins` présentée plus tard, il est vrai).
- L'appel `g.ajouterArc(`s_0`,`s_1`)` construit une arête dans un graphe non-orienté[1]. La matrice `mat`, qui reflète le graphe partiellement construit doit être symétrique.
- Il faut traiter tous les cas (i refuse de dîner avec j, j avec i, ou les deux), ce qui exclut toutes les solutions qui se contentent de comparer i et j.

[1] Si on implémente **Graphe** avec des listes de voisins, alors un appel à `ajouterArc` modifie deux listes de façon interne. Mais il n'est pas utile de le savoir.

Certaines copies font usage de la boucle **for** améliorée, les risques d'erreurs entre `k` et `t[i][k]` sont alors réduits à néant.

```
static Graphe fabriquer(int [] [] t) {
  int n = t.length ;
  Graphe g = new Graphe(n) ;
  boolean [][] mat = new boolean [n][n] ;

  for (int i = 0 ; i < n ; i++) {
    for (int j : t[i]) {
      if (!mat[i][j]) {
        mat[i][j] = mat[j][i] = true ;
        g.ajouterArc(i,j) ;
      }
    }
  }
  return g ;
}
```

□

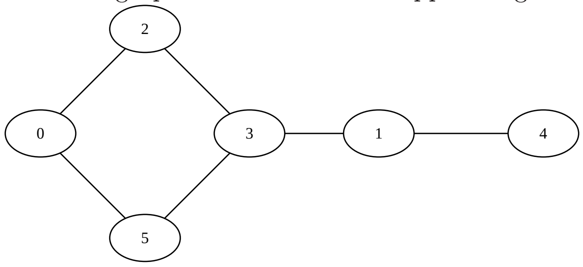
Partie II. Approche gloutonne

L'organisation du dîner revient à affecter chaque membre de l'amicale à une table, de sorte que deux convives qui refusent de dîner ensemble sont assis à des tables différentes. Il s'agit exactement du problème de *coloriage de graphe*, associer une couleur à chaque sommet du graphe, de sorte que, pour toute arête $s_0 \leftrightarrow s_1$, les couleurs associées à s_0 et s_1 sont distinctes.

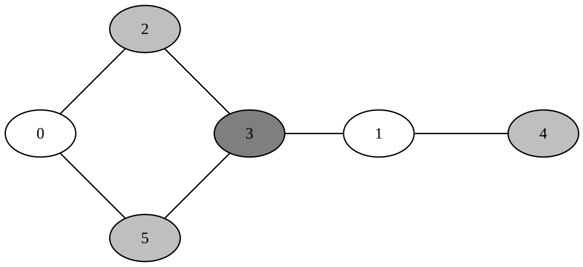
Évidement, organiser le dîner est possible : chacun peut manger seul. Mais la convivialité risque d'en souffrir. On cherche à faire mieux avec l'algorithme glouton qui colorie les sommets du graphe dans l'ordre $0, 1, \ldots, n-1$. Les couleurs sont représentées par des entiers, $0, 1$, etc. Pour colorier le sommet i, on choisit la plus petite couleur possible qui est compatible avec les couleurs des voisins de i qui sont déjà coloriés.

Question 3. Colorier le graphe de la figure 2 à la main selon l'approche gloutonne. Le plus simple est de redessiner le graphe colorié (avec des hachures ou des crayons de couleurs).

Figure 2: Un graphe à colorier selon l'approche gloutonne



Réponse :



Note : Il n'est peut-être pas inutile de préciser le déroulement du coloriage :

sommet	voisins[couleur]	couleur allouée
0	$2[], 5[]$	0
1	$3[], 4[]$	0
2	$0[0], 3[]$	1
3	$1[0], 2[1], 5[]$	2
4	$1[0]$	1
5	$0[0], 3[2]$	1

□

La classe **Graphe** possède une méthode `voisins` spécifiée ainsi :

```
Iterable<Integer> voisins(int s) ; // Récupérer les voisins du sommet s
```

L'appel `voisins(`s`)` renvoie donc les sommets v du graphe tels qu'il existe une arête $s \leftrightarrow v$, sous la forme d'un objet qui implémente l'interface `Iterable<Integer>`. Ce qui veut simplement dire que l'on peut parcourir les voisins[2] d'un sommet s par la construction **for**. Par exemple, voici comment on peut afficher les voisins d'un sommet donné s :

```
System.out.print("voisins de " + s + ":") ;
for (int v: this.voisins(s)) {
  System.out.print(" " + v) ;
}
System.out.println() ;
```

Question 4. Compléter la classe **Graphe** par une méthode `int [] glouton()` qui réalise l'algorithme glouton. Le tableau d'entiers retourné par la méthode `glouton` indique les couleurs affectées aux n sommets.

Réponse :

```
static int premierFaux(boolean t[]) {
  for(int i=0; i<t.length; i++) {
    if (t[i]==false) return i;
  }
  /* Code non atteint, un sommet possède au plus n-1 voisins
     et t est de taille n */
  return -1;
}

int[] glouton() {
  int[] couleurs = new int[n];
  for(int i=0; i<n; i++) {
    boolean impossible[] = new boolean[n]; // NB: cases initialisées à false
```

[2] Ne vous laissez pas troubler, les voisins de i dans le graphe sont justement ceux qui ne peuvent pas être les voisins de i à table

```
    for(int j : voisins(i)) {
      if (j < i) impossible[couleurs[j]] = true;
    }
    couleurs[i] = premierFaux(impossible);
  }
  return couleurs;
}
```

Note : Il y a deux difficultés :

1. Bien identifier les voisins déjà coloriés. Le test `j < i` donne une solution élégante.
2. Trouver la plus petite couleur qui n'est pas une couleur de voisin. Le tableau de booléens `impossible` donne une solution simple, mais qui conduit à une complexité quadratique. En effet, allouer un tableau de taille n coûte de l'ordre de n opérations (à cause de l'initialisation).

Une autre solution sans tableau, avec un usage relativement sûr et élégant des boucles :

```
int[] glouton() {
  int[] couleurs = new int[n];

  for(int i=0; i<n; i++) {
    boolean found = false ;
    int c = 0 ; // Il faut déclarer c ici, car il sert après la boucle
    for ( ; !found ; c++) {
      found = true ; // Soyons optimistes
      for(int j : voisins(i)) {
        if (j < i && c == couleurs[j]) found = false ;
      }
    }
    couleurs[i] = c ;
  }
  return couleurs;
}
```

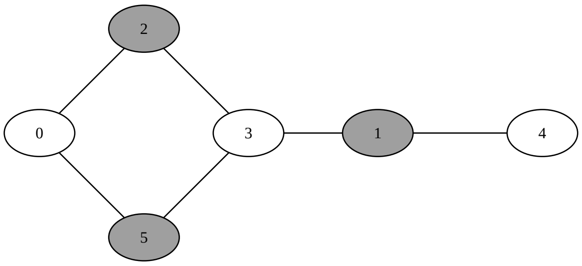
Je trouve ces manips de variables booléennes assez casse-cou. Utiliser **return** dans une méthode auxiliaire est souvent plus sûr.

```
private int trouverLeTrou(int c, int i, int [] couleurs) {
  for (int j : voisins(i)) {
    if (j < i && c == couleurs[j])
      return trouverLeTrou(c+1, i, couleurs) ;
  }
  // Ici on sait que c n'est pas une couleur de voisin
  return c ;
}

int[] glouton() {
  int[] couleurs = new int[n];

  for(int i=0; i<n; i++) couleurs[i] = trouverLeTrou(0,i, couleurs) ;
  return couleurs;
}
```

□

Question 5. La duchesse souhaite maximiser la convivialité, c'est-à-dire que vous souhaitez minimiser le nombre de tables.
a) Confirmer par un exemple que l'algorithme glouton ne colorie pas les graphes avec le nombre minimal de couleurs possible.

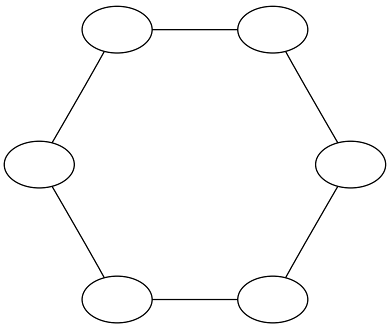
Réponse : Le graphe de la question 3 (pour lequel l'algorithme glouton utilise trois couleurs) est coloriable avec deux couleurs.



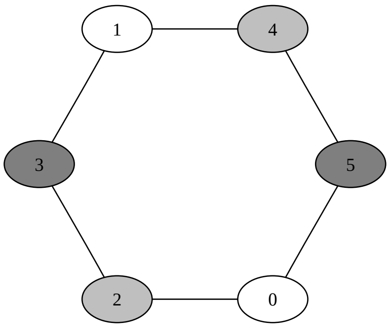
□

b) Un camarade vous suggère de commencer par placer les membres les plus exigeants, autrement dit de colorier les sommets du graphe dans l'ordre décroissant des degrés (le degré d'un sommet est le nombre d'arêtes issues du sommet). Quand vous lui demandez : mais comment départager deux sommets de même degré ? Il repond : dans ce cas, on fait ce que l'on veut, ça marchera toujours.

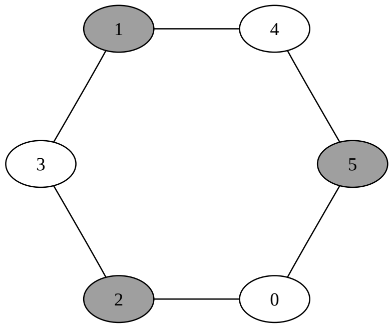
Montrer lui que son astuce ne permet pas de colorier un graphe arbitraire avec un nombre minimal de couleurs.
Indication : On pourra montrer que la suggestion du camarade ne conduit pas au coloriage avec le moins de couleurs possibles d'un graphe particulier, par exemple celui-ci.



Réponse : On considère le circuit à six sommets. Tous les sommets sont de degré 2, par conséquent tout ordre de coloriage est compatible avec l'astuce. On considère alors l'algorithme glouton appliqué selon l'ordre indiqué par les numéros de sommets :



On utilise donc trois couleurs. Or, le graphe est coloriable avec deux couleurs seulement :



□

Partie III. Dîner à deux tables.

La duchesse vient de vous téléphoner pour vous signaler que le club-house ne possède que deux tables. La question posée devient donc : colorier un graphe avec deux couleurs.

Un *chemin* (élémentaire) de taille p ($p \geq 0$), noté $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_p$, est défini comme une suite ordonnée de sommets *deux à deux distincts* s_0, s_1, ..., s_p tels que les arêtes $s_0 \leftrightarrow s_1$, ..., $s_{p-1} \leftrightarrow s_p$ existent.

Un *circuit* (élémentaire) de taille p ($p > 2$), noté $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{p-1} \leftrightarrow s_0$, est défini ainsi :

1. $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{p-1}$ est un chemin de taille $p-1$,
2. et l'arête $s_{p-1} \leftrightarrow s_0$ existe.

On note qu'une simple arête $s_0 \leftrightarrow s_1$ n'engendre pas de circuit $s_0 \leftrightarrow s_1 \leftrightarrow s_0$, en vertu de la condition $p > 2$.

Question 6. Montrer qu'un graphe qui contient un circuit de taille impaire ne peut pas être colorié avec deux couleurs.

Réponse : Si un graphe est coloriable avec k couleurs, tous ses sous-graphes le sont également. Or un circuit de taille impaire n'est pas coloriable avec deux couleurs. En effet, soit le circuit impair $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{2k} \leftrightarrow s_0$. Supposons par exemple que s_0 est blanc, alors l'existence du chemin $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_{2k}$ entraîne que s_{2k} est blanc, tandis que l'arc $s_{2k} \leftrightarrow s_0$ entraîne que s_{2k} est noir. Contradiction.

Note : C'est une fausse « preuve par l'absurde ». Il est plus élégant de formaliser différemment : un circuit 2-coloriable est nécessairement de taille paire. Soit donc un circuit 2-coloriable, et soit 0 la couleur de s_0 (choix arbitraire). Pour tout k ($0 \leq k < p$), en considérant le chemin $s_0 \leftrightarrow s_1 \leftrightarrow \cdots \leftrightarrow s_k$ on voit que la couleur de s_k est 0, si et seulement si k est pair, et 1 autrement. Par ailleurs, en raison de l'arête $s_{p-1} \leftrightarrow s_0$, la couleur de s_{p-1} est 1. L'entier $p-1$ est donc nécessairement impair, c'est à dire que le circuit est de taille paire. □

On peut voir un arbre comme un graphe particulier : il existe un sommet particulier (la racine), et pour tout sommet s il existe un unique chemin issu de la racine et aboutissant en s.

Question 7. Colorier un arbre avec deux couleurs.

Réponse : On colorie l'arbre ainsi : Soit r la racine, et soit s un sommet quelconque de l'arbre, on appelle profondeur de s, noté $P(s)$, la taille du chemin (unique) qui va de r à s. Si $P(s)$ est pair, s est colorié en blanc. Sinon, s est colorié en noir. Les voisins d'un sommet s sont alors nécessairement de la couleur opposée à s, car les profondeurs de s et d'un de ses voisins diffèrent nécessairement de un.

Note : Le fait ci-dessus est « évident » : en gros deux sommets voisins dans un arbre sont en relation père–fils. On peut vouloir le montrer à partir de la définition des arbres de l'énoncé (cela n'a pas été jugé nécessaire lors de la correction). Soit donc une arête $s \leftrightarrow v$. Soient S et V, les chemins (uniques) de la racine à s et v respectivement. On va montrer que l'on a soit $S = V \leftrightarrow s$, soit $V = S \leftrightarrow v$. Supposons d'abord que $V \leftrightarrow s$ est un chemin (c'est-à-dire supposons que s n'est pas un sommet de V), alors on peut conclure par l'unicité du chemin S. Sinon, s apparaît dans V, qui est donc de la forme $V = V_1 \leftrightarrow s \leftrightarrow V_2$. Par unicité de S, il vient $V = S \leftrightarrow V_2$, ce qui entraîne que v n'apparaît pas dans dans S (sinon V contiendrait deux fois v). On peut alors conclure comme précédemment en inversant les rôles de s et de v.

Re-Note : Il me semble que dire « colorier les sommets de l'arbre avec la parité de leur profondeur » est suffisamment explicite. Par sécurité (mais on perd alors du temps) on peut vouloir donner un algorithme en pseudo-code ou en Java. Il convient alors d'être précis. Si on réutilise la structure de graphe de l'énoncé, il faut faire attention à ne pas boucler, car le père d'un sommet est bien un voisin de ses fils :

```
// On suppose que this est un arbre, racine est le sommet racine
int [] colorierArbre(int racine) {
  int [] couleur = new int [n] ;
  couleur[racine] = 0 ;
  colorierArbre(-1, racine, couleur) ;
  return couleur ;
}

// p est le père de s
void colorierArbre(int p, int s, int [] couleur) {
  for (int f : voisins(s)) {
    if (f != p) {
      couleur[f] = 1-couleur[s] ;
      colorierArbre(s, f, couleur) ;
    }
  }
}
```

Si on utilise sa propre structure, il faut la définir, ou au moins faire comprendre au correcteur ce que l'on a voulu dire :

```
static int [] colorierArbre(Arbre a) {
  int [] couleur = new int [n] ;
  couleur[a.racine] = 0 ;
  colorierArbre(a.racine, a, couleur) ;
  return couleur ;
}

// NB: a.fils(s) renvoie les fils (ds a) du sommet s, selon le modèle de l'énoncé
static colorierArbre(int s, Arbre a, int [] couleur) {
  for (int f : a.fils(s)) {
    couleur[f] = 1 - couleur[s] ;
    colorierArbre(f, a, couleur) ;
  }
}
```

□

Question 8. En utilisant la notion d'arbre couvrant du cours, colorier un graphe connexe dont tous les circuits sont de taille paire, en utilisant deux couleurs.

Réponse : Soit donc $\mathcal{G} = (S, A)$ graphe connexe, et soit (S, T) arbre couvrant de $\mathcal{G}$. En effet, par la connexité de $\mathcal{G}$, tous les sommets de $\mathcal{G}$ sont aussi des sommets de l'arbre couvrant. En coloriant l'arbre comme à la question précédente, on parvient à colorier tous les sommets.

Il reste à montrer que les arêtes de $A \setminus T$ relient des sommets de couleurs différentes. Soit donc $s_0 \leftrightarrow s_1$ une telle arête. Il existe deux chemins $C_0 = r \leftrightarrow \cdots \leftrightarrow s_0$ et $C_1 = r \leftrightarrow \cdots \leftrightarrow s_1$ dont toutes les arêtes sont dans T. Les chemins C_0 et C_1, qui commencent par le même sommet admettent un plus grand préfixe commun $P = r \leftrightarrow \cdots \leftrightarrow a$.

1. Si a n'est ni s_0 ni s_1, alors on écrit $C_0 = P \leftrightarrow S_0$ et $C_1 = P \leftrightarrow S_1$. Les sommets des chemins S_0 et S_1 sont deux à deux distincts (autrement il existerait deux chemins de la racine vers un sommet de l'arbre), on peut donc construire le circuit : $a \leftrightarrow S_0 \leftrightarrow \bar{S}_1 \leftrightarrow a$ ($\bar{S}_1$ est S_1 écrit à l'envers). Ce circuit est de taille paire, et donc la taille du chemin (de l'arbre) $\bar{S}_1 \leftrightarrow a \leftrightarrow S_0$ est impaire, ce qui entraîne que s_0 et s_1 portent des couleurs distinctes.

2. Sinon, si a est s_0 (par exemple), on a directement un circuit élémentaire $s_0 \leftrightarrow S_1 \leftrightarrow s_0$. Ce circuit est pair, ce qui entraîne que le chemin (de l'arbre) $s_0 \leftrightarrow S_1$ est impair et donc que s_0 et s_1 portent des couleurs distinctes.

Note : Ici une preuve a été requise lors de la correction, car la 2-coloriabilité du graphe n'est pas évidente. En abusant des notations on peut se passer du deuxième cas ci-dessus.

Il y a une preuve plus élégante, ou plus exactement une organisation plus élégante des arguments de la preuve ci-dessus. Tout d'abord, l'existence d'un chemin élémentaire entre deux sommets quelconques d'un graphe connexe est facile à démontrer (par ex. en itérant le remplacement des circuits $c \leftrightarrow \cdots \leftrightarrow c$ par c dans un chemin quelconque). Par connexité du graphe, colorier l'un quelconque de ses arbres couvrants affecte une couleur à tous les sommets. Soit maintenant $s_0 \leftrightarrow s_1$ arête qui n'est pas une arête de l'arbre choisi. L'arbre couvrant est connexe (comme tous les arbres) il existe donc un chemin élémentaire de s_0 à s_1 dans l'arbre, qui devient un circuit élémentaire si on y ajoute l'arc $s_0 \leftrightarrow s_1$. Le circuit est par hypothèse de taille paire, le chemin (de l'arbre) est donc de taille impaire, ce qui entraîne que les couleurs de s_0 et s_1 sont distinctes.

En tout état de cause l'exercice de rédiger une preuve est délicat. Il convient de bien identifier ce qui doit être justifié. Par exemple à la question 6, une récurrence est exagérée. En revanche, dans cette question, il faut identifier le circuit élémentaire considéré. □

Question 9. Dans cette question on suppose que le graphe **this** est connexe. Ajouter une méthode **int** `[] colorierDeuxConnexe ()` à la classe des graphes, spécifiée ainsi :

- Si le graphe **this** n'est pas coloriable avec deux couleurs, alors la méthode renvoie **null**.
- Sinon, la méthode renvoie un tableau d'entiers qui réalise une affectation des deux couleurs 0 et 1 aux sommets $0, \ldots, n-1$.

Indication : Un parcours de graphe construit un arbre couvrant.

Réponse : On a choisi un parcours DFS pour sa concision, mais n'importe quel parcours fait l'affaire.

```
int [] dfs(int s, int coul, int [] couleur, boolean [] vu) {
  if (vu[s]) {
    if (couleur[s] == coul) return couleur ;
    else return null ;
  } else {
    vu[s] = true ; couleur[s] = coul ;
    for (int v : voisins(s)) {
      if (dfs(v, 1-coul, couleur, vu) == null) return null ;
    }
    return couleur ;
  }
}
```

```
int [] colorierDeuxConnexe() {
  int [] couleurs = new int[n] ;
  boolean [] vu = new boolean[n] ;
  return dfs(0, 0, couleurs, vu) ;
}
```

On peut se passer du tableau `vu`, mais alors il faut initialiser les cases de `couleur` à par exemple `-1` et remplacer le test de `vu[s]` par `couleur[s] >= 0`.

Note : La solution exploite la question précédente. On colorie l'arbre couvrant du parcours. La correction du programme (preuve non-exigée) repose sur les observations suivantes :

1. Si le coloriage réussit, il est correct. En effet, on aura vu chaque arête deux fois, dont au moins une fois avec les deux sommets coloriés, ce qui entraîne que toutes les arêtes auront été vérifiées. (Le double parcours des arêtes est une propriété du parcours de graphe non-orienté.)
2. Si le coloriage échoue, il était impossible. En effet, on montre alors que le graphe contient un circuit de taille impaire. L'argument clé est ici que deux sommets de couleurs égales sont nécessairement reliés par un chemin élémentaire de taille paire.

Le parcours DFS peut aussi se programmer avec le test de `vu[s]` avant les appels à `dfs`, ce qui est je crois la technique du cours. Cela revient bien entendu au même. On peut d'ailleurs dans ce cas économiser l'argument **int** `coul`.

```
int [] dfs2(int s, int [] couleur) {
  for (int v : voisins(s)) {
    if (couleur[v] == -1) {
      couleur[v] = 1-couleur[s] ;
      if (dfs2(v, couleur) == null) return null ;
    } else if (couleur[v] == couleur[s]) return null ;
  }
  return couleur ;
}

int [] colorierDeuxConnexe2() {
  int [] coul = new int[n] ;
  Arrays.fill(coul,-1) ;
  // Ou bien : for (int k = 0 ; k < n ; k++) coul[k] = -1 ;
  coul[0] = 0 ;
  return dfs2(0, coul) ;
}
```

Certains prenent au pied de la lettre la structure de l'énoncé, et procèdent ainsi :

1. Fabriquer l'arbre couvrant,
2. le colorier,
3. vérifier le coloriage.

Il faut alors écrire pas mal de code. . .

```
int [] colorierDeuxConnexe() {
  Graphe a = arbreCouvrant(0) ;
  int [] couleur = a.colorierArbre(0) ; // cf. Question 7
  if (verifier(couleur))
```

```
    return couleur ;
  else
    return null ;
}

Graphe arbreCouvrant(int s) {
  Graphe a = new Graphe(n) ;
  boolean [] vu = new boolean [n] ;
  vu[s] = true ;
  arbreCouvrant(s, vu, a) ;
  return a ;
}

void arbreCouvrant(int s, boolean [] vu, Graphe a) {
  for (int v : voisins(s)) {
    if (!vu[v]) {
      vu[v] = true ;
      a.ajouterArc(s, v) ;
      arbreCouvrant(v, vu, a) ;
    }
  }
}

boolean verifier(int [] couleur) {
  for (int i = 0 ; i < n ; i++) {
    for (int v : voisins(i)) {
      if (couleur[i] == couleur[v]) return false ;
    }
  }
  return true ;
}
```

□

Question 10. Même question que la précédente en ne supposant pas le graphe **this** connexe.

Réponse : L'écriture de la solution de la question précédente en deux méthodes facilite la réponse.

```
  int [] colorierDeux() {
    int [] couleurs = new int[n] ;
    boolean [] vu = new boolean[n] ;
    for (int s = 0 ; s < n ; s++) {
      if (!vu[s])
        if (dfs(s, 0, couleurs, vu) == null) return null ;
    }
    return couleurs ;
}
```

Note : Quelques copies utilisent une exception pour signaler les graphes non deux-coloriables. Voici donc un nouveau DFS, écrit dans la classe **Graphe**.

```
  // Oui, on peut déclarer une classe dans une classe
  class CircuitImpair extends Exception {} ;

  /* Noter la signature de dfs3 ci-dessous :
```

```
    1. Le type de retour est void
    2. L'exception qui peut être renvoyée est déclarée
  */
  void dfs3(int s, int [] couleur) throws CircuitImpair {
    for (int v : voisins(s)) {
      if (couleur[v] == -1) {
        couleur[v] = 1-couleur[s] ;
        dfs3(v, couleur) ;
      } else if (couleur[v] == couleur[s]) {
        throw new CircuitImpair () ;
      }
    }
  }
```

Cela peut conduire à une solution particulièrement simple pour cette question, car dégagée de la nécessité de vérifier la valeur rendue par le parcours des composantes connexes.

```
1  int [] colorierDeuxConnexe() {
2    int [] coul = new int[n] ; Arrays.fill(coul,-1) ;
3    try {
4      coul[0] = 0 ;
5      dfs3(0, coul) ;
6    } catch (CircuitImpair e) { return null ; }
7    return coul ;
8  }
```

Et pour la question 10, remplacer les deux lignes 4 et 5 par :

```
for (int s = 0 ; s < n ; s++) {
  if (coul[s] == -1) {
    coul[s] = 0 ;
    dfs3(s, coul) ;
   }
}
```

□