Devoir surveillé #5 (avec solutions)

Mardi 2026-04-28 ; durée : quatre heures

Nombre chromatique et coloriage de graphe

Ce sujet est largement repris de l'épreuve de composition d'informatique A du concours X-ENS 2018. Les différences principales sont le changement du langage de programmation de *Caml Light* à *C*, la suppression d'une question de programmation un peu redondante en fin de sujet, ainsi que de la dernière question qui demandait de généraliser l'algorithme « de Wigderson » à un k quelconque.
Merci à Jean-Baptiste Bianquis pour ses reproductions TikZ des figures du sujet original.

Préliminaires

Complexité. Par **complexité en temps** d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de A dans le pire cas.
Lorsque la complexité en temps dépend d'un ou plusieurs paramètres $\kappa_1, \dots, \kappa_r$, on dit qu'elle est en $O(f(\kappa_1, \dots, \kappa_r))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_1, \dots, \kappa_r$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), la complexité est au plus $C \times f(\kappa_1, \dots, \kappa_r)$.
On dit que la complexité en temps est **linéaire** (resp. **quadratique**) quand f est une fonction linéaire (resp. quadratique) des paramètres $\kappa_1, \dots, \kappa_r$, **polynomiale** quand f est une fonction polynomiale des paramètres $\kappa_1, \dots, \kappa_r$, et enfin **exponentielle** quand $f = 2^g$, où g est une fonction polynomiale des paramètres $\kappa_1, \dots, \kappa_r$.
Les complexités (en temps) des algorithmes **devront être justifiées**.

Graphes. Rappelons qu'un graphe non-orienté est la donnée (S, A) de deux ensembles finis :

- un ensemble S de **sommets**, et
- un ensemble $A \subseteq S \times S$ d'**arêtes**, tel que pour tout couple de sommets (s, t), on a $(s, t) \in A$ si et seulement si $(t, s) \in A$. On considérera uniquement des graphes sans boucles, et l'on aura donc toujours $s \neq t$ si $(s, t) \in A$.

On rappelle également les définitions suivantes :

- Étant donné un graphe $G = (S, A)$, le **sous-graphe induit** par un ensemble de sommets $T \subseteq S$ est $(T, A \cap (T \times T))$.
- Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet de G. Un **voisin** de s est un sommet t de G qui est relié à s par une arête, c'est-à-dire tel que $(s, t) \in A$. On note $V(s)$ l'ensemble des voisins de s.
- Le **degré** $d(s)$ de s est le cardinal de $V(s)$.
- Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.
- Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non-vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple être des entiers, des listes ou des chaînes de caractères.
- On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est-à-dire lorsque L vérifie la condition (1) suivante :
$$\forall s, t \in S, (s, t) \in A \Rightarrow L(s) \neq L(t). \tag{1}$$
- Un graphe G est dit ***k*-coloriable** s'il admet un coloriage avec au plus k couleurs. Un graphe étiqueté est dit colorié si sa fonction d'étiquetage est un coloriage. Un exemple de graphe colorié est donné sur la Figure 1.
- Le **nombre chromatique** d'un graphe non-orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul des nombres chromatiques et de coloriages.

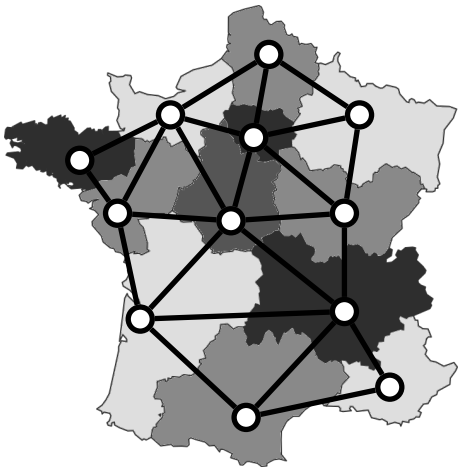


FIGURE 1 – Exemple de graphe colorié : le graphe des régions métropolitaines françaises (hors Corse). Deux régions sont reliées par une arête dans le graphe si et seulement si elles sont voisines. Deux régions voisines sont de couleurs différentes.

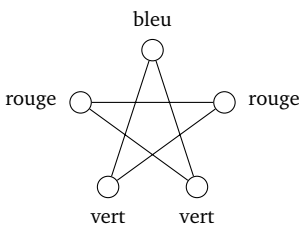
Programmation. Toutes les questions de programmation de ce sujet doivent être traitées en **langage C**.
On pourra supposer que les résultats des opérations arithmétiques sont toujours définis. En particulier, on pourra ignorer tout éventuel problème lié au dépassement de capacité des types entier signés.
Lors d'une utilisation de la fonction `malloc`, on pourra supposer que celle-ci n'échoue jamais (ne renvoie jamais de valeur de pointeur nul).
Toute mémoire allouée doit être libérée. Les fuites mémoire sont considérées comme des erreurs de programmation à part entière.

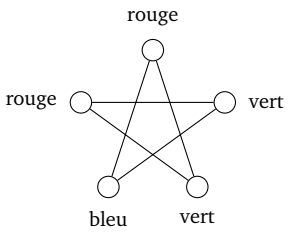
Représentation des graphes étiquetés. On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence. On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par un tableau d'entiers de type `int`.
Un graphe non-orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par un objet `g` de type `bool**` tel que pour tous sommets $i, j \in S$, on ait `g[i][j] == true` si et seulement si $(i, j) \in A$. Le graphe G étant supposé non-orienté, on a alors également par symétrie `g[j][i] == true`. Puisqu'il est supposé sans boucle, on a également `g[i][i] == false` pour tout `i`.
Pour un étiquetage `e` de `g`, l'étiquette du sommet `i` de `g` est donnée par `e[i]`.
Pour toutes les fonctions calculant un étiquetage, on pourra supposer (sauf mention du contraire) comme précondition que l'argument `e` correspondant à l'étiquetage est tel que `e[i] == -1` pour tout `i` au moment de l'appel.

Le sujet comporte seulement 17 questions, dont certaines situées à la fin sont tout à fait abordables. Prenez le temps de le chercher en entier.

Coloriage

1. Indiquer, pour chacun des graphes suivants, s'il est colorié :





Le graphe de gauche n'est pas colorié car les deux sommets étiquetés en rouge sont voisins.
Le graphe de droite est colorié.

2. Donner le nombre chromatique, ainsi qu'un exemple de coloriage correspondant, pour le graphe de Petersen représenté à la Figure 2.

Le sous-graphe induit par les sommets de numéros 0, 6, 9, 3, 4 est le cycle C_5. Celui-ci ne peut pas être colorié avec moins de 3 couleurs (on doit alterner les couleurs entre sommets consécutifs, et l'un des sommets aura des voisins de deux couleurs différentes), donc le nombre chromatique de ce graphe est au moins 3. C'est en fait exactement 3, comme le montre le coloriage suivant :
0 : bleu 1 : vert 2 : vert 3 : rouge 4 : vert 5 : rouge 6 : rouge 7 : bleu 8 : rouge 9 : bleu

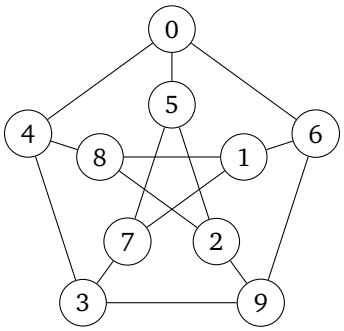


FIGURE 2 – Le graphe de Petersen, de sommets 0,...,9.

3. La vérification de la propriété de coloriage est le problème suivant.

— *Entrée* : un graphe G, et un étiquetage L de G.
— *Question* : L est-il un coloriage de G ?

Écrire une fonction C de signature `bool est_col(int n, bool **g, int e[n])` telle que `est_col(n, g, e)` renvoie `true` si et seulement si `e` est un coloriage de `g`, comportant `n` sommets. On demande une complexité quadratique en le nombre de sommets du graphe.

On propose :
```
bool est_col(int n, bool **g, int e[n])
{
    for (int s = 0; s < n; s++) // O(n) itérations
    {
        for (int t = s + 1; t < n; t++) // O(n) itérations
        {
            if (g[s][t] && (e[s] == e[t])) // O(1)
            {
                return false;
            }
        }
    }
    return true;
}
```
On itère au plus $\mathtt{n}^2$ fois un bloc de coût $O(1)$: la complexité en temps est bien quadratique en `n`.

4. Démontrer que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets.

On note G le graphe (quelconque) et n sont nombre de sommets. On a la majoration évidente $\chi(G) \leq n$: attribuer des couleurs deux-à-deux distinctes aux n sommets de G est trivialement un n-coloriage.
Il suffit donc de vérifier pour tout $k \in [\![1, n-1]\!]$ si G est k-coloriable. Il existe k^n étiquetages utilisant k couleurs, et il suffit donc de vérifier au plus $\sum_{k=1}^{n-1} k^n \leq n^{n+1}$ étiquetages, ce qui se fait en temps $O(n^2 \times n^{n+1})$ par la question

précédente. On peut réécrire cela en un $O(n^2 \times 2^{(n+1)\log n})$, qui est aussi (brutalement) un $O(2^{n+(n+1)\log n})$ et donc (par exemple) un $O(2^{n^2})$. Cet algorithme calcule donc bien $\chi(G)$ en temps exponentiel en n (pour la définition du sujet).

2-coloriage

Nous avons vu à la question 4 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-problèmes. On considère dans cette partie le cas du 2-coloriage.

Graphe biparti. Un graphe G est biparti lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U, tels que chaque arête a une extrémité dans T et l'autre dans U.

5. Démontrer qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

Biparti $\Rightarrow$ *2-coloriable* : Soient T, U les sous-ensembles de S donnés par le fait que G est biparti, on peut étiqueter chaque sommet de T en une couleur (disons noir) et chaque sommet de U en une autre (disons rouge). Par le fait que G est biparti, aucun sommet de T n'est adjacent à un sommet de T et de même pour U, et cet étiquetage est un 2-coloriage.
2-coloriable $\Rightarrow$ *biparti* : G est 1-coloriable ssi. $A = \emptyset$; dans ce cas la bipartition donnée par $T = S$ et $U = \emptyset$ convient. Sinon soient B, R les ensembles de sommets de S coloriés respectivement en noir et en rouge par un 2-coloriage, ils sont non vides et forment une partition de S. De plus, toute arête $(s,t) \in A$ connecte nécessairement un sommet de B à un sommet de R puisque sinon l'étiquetage ne serait pas un coloriage. On peut donc prendre $T = B$ et $U = R$ comme bipartition de G.

On se propose de programmer la vérification de la 2-coloriabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons `-1`, `0` et `1`. L'étiquetage est initialisé à `-1` pour tous les sommets, et on teste la 2-colorabilité avec `0` et `1`. Le principe de l'algorithme est le suivant :

— 1/ On choisit un sommet s d'étiquette `-1`
— 2/ On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs `0` et `1` à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.
— 3/ Enfin, s'il reste des sommets d'étiquette `-1`, alors on revient au point 1/.

6. Écrire une fonction C de signature `void deux_col(int n, bool **g, int e[n])` telle que `deux_col(n, g, e)` modifie `e` en un 2-coloriage de `g` s'il est 2-coloriable. Le coloriage utilisera les couleurs `0` et `1`. On demande une complexité quadratique en le nombre de sommets `n` du graphe. Le comportement de la fonction est laissé au choix du ou de la candidate lorsque `g` n'est pas 2-coloriable.
Indication : on pourra introduire une fonction auxiliaire récursive qui effectue un parcours en profondeur depuis un sommet donné en argument.

On propose :

```
void deux_col_rec(int n, bool **g, int e[n], int s, int c)
{
    if (e[s] == -1) // exécuté une fois par sommet
    {
        e[s] = c;
        for (int t = 0; t < n; t++) // n itérations
        {
            if (g[s][t]) // coût marginal O(1)
            {
                deux_col_rec(n, g, e, t, 1 - c);
            }
        }
    }
```

```
}

void deux_col(int n, bool **g, int e[n])
{
    for (int s = 0; s < n; s++) // n itérations
    {
        if (e[s] == -1) // coût marginal O(1)
        {
            deux_col_rec(n, g, e, s, 0);
        }
    }
}
```

Le coût marginal de `deux_col` est un $O(n)$, et chacun des `n` sommets occasionne un coût $O(n)$ une unique fois lors de sa visite. La complexité temporelle totale est donc bien quadratique en `n`.

Algorithmes gloutons

Dans cette partie, nous allons étudier deux algorithmes permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.
Ces deux algorithmes prennent en paramètre un ordre sur les sommets du graphe, que l'on appellera **ordre de numérotation** Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et $0 < 7 < 2 < 5 < 4 < 6 < 8 < 1 < 3 < 9$ sont deux ordres de numérotation des sommets du graphe de Petersen (Figure 2).
Pour un graphe `g` à `n` sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de `n` valeurs entières de type `int`, tel que `num[k] == j` si et seulement si le sommet `j` apparaît en `(k + 1)`$^{\text{ième}}$ position dans l'ordre.
Nous commençons par l'**algorithme glouton** de coloriage. Cet algorithme construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :
On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné. Pour chaque sommet s parcouru :

— 1/ On calcule l'ensemble $C(s) = \{L(t) \mid t \in V(s)\}$ des couleurs déjà données aux voisins de s.
— 2/ On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.
— 3/ On pose $L(s) = c$.

7. Considérons le graphe de Petersen (Figure 2) et les deux ordres de numérotation :

```
int num1[10] = {1, 3, 4, 0, 2, 6, 5, 9, 8, 7}
int num2[10] = {0, 7, 2, 5, 4, 6, 8, 1, 3, 9}
```

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

Pour l'ordre `num1` on obtient un coloriage tel que donné par :

```
int col1[10] = {0, 0, 0, 0, 1, 1, 1, 2, 2, 2}
```

qui nécessite trois couleurs.
Pour l'ordre `num2` on obtient un coloriage tel que donné par :

```
int col2[10] = {0, 3, 0, 2, 1, 1, 1, 0, 2, 3}
```

qui nécessite quatre couleurs.

8. Écrire une fonction C de signature :

```
int min_couleur_possible(int n, bool **g, int s, int e[n])
```

telle que pour un graphe `g` à `n` sommets, un étiquetage `e` à valeurs dans $\{-1, \ldots, n-1\}$, et pour un sommet `s` de `g`, `min_couleur_possible(n, g, s, e)` renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{e[t] \mid t \in V(s)\}$. On demande une complexité linéaire en `n`.

On propose :

```c
int min_couleur_possible(int n, bool **g, int s, int e[n])
{
    bool *available = malloc(n * sizeof(bool));
    for (int t = 0; t < n; t++)
    {
        available[t] = true;
    }
    for (int t = 0; t < n; t++)
    {
        if (g[s][t] && e[t] > -1)
        {
            available[e[t]] = false;
        }
    }
    for (int t = 0; t < n; t++)
    {
        if (available[t])
        {
            free(available);
            return t;
        }
    }
    assert(false);
}
```

Cette fonction est une succession de trois boucles `for` effectuant au plus `n` itérations de coût constant. La complexité temporelle totale est donc bien linéaire en `n`.

9. Écrire une fonction C de signature :

`void glouton(int n, bool **g, int num[n], int e[n])`

telle que pour un graphe `g` et un ordre de numérotation `num` de ses sommets, `glouton(n, g, num, e)` modifie `e` en le coloriage glouton de `g`, avec au plus $d+1$ couleurs où d est le degré de `g`. On demande une complexité quadratique en `n`, le nombre de sommets de `g`.
Dans le cas où le tableau `num` contient autre chose qu'un ordre de numérotation des sommets de `g`, le résultat de la fonction est laissé au choix du ou de la candidate.

On propose :

```c
void glouton(int n, bool **g, int num[n], int e[n])
{
    for (int i = 0; i < n; i++) // n itérations
    {
        int c = min_couleur_possible(n, g, num[i], e); // coût O(n)
        e[num[i]] = c;
    }
}
```

10. Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.

La fonction de la question précédente implémente l'algorithme glouton avec un coût quadratique en le nombre de sommets du graphe. *A fortiori*, cet algorithme termine.

La boucle principale de l'algorithme glouton (tel qu'implémenté à la question précédente) satisfait l'invariant qu'à la fin de la $i$ème itération i sommets ont été coloriés et aucun sommets voisins n'ont la même couleur :
Initialisation. Triviale par le fait qu'avant l'entrée dans la boucle aucun sommet n'est colorié.
Conservation. On suppose la propriété vraie à la fin de la $i$ème itération. À l'itération suivante, on affecte une couleur à un unique sommet non encore colorié (le $i+1$ème dans l'ordre de numérotation), qui est distincte de celle de ses voisins. La propriété est donc conservée.

L'algorithme effectuant autant d'itérations de cette boucle qu'il y a de sommet dans le graphe, on a qu'à la fin de celle-ci (et de l'algorithme) tous les sommets sont coloriés d'une couleur distincte de leurs voisins, ce qui constitue bien un coloriage du graphe.

On montre maintenant que ce coloriage utilise au plus $d+1$ couleurs. Par définition l'algorithme colorie un sommet s en $c := \min \mathbb{N} \setminus C(s)$. Puisque $\#C(s) \leq \#V(s) = d(s) \leq d$, l'on a $c \leq d$. Cette majoration étant valable pour tout sommet, au plus $d+1$ couleurs $0, \ldots, d$ sont utilisées dans le coloriage.

11. Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G.
En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

Quitte à renuméroter, on suppose que les couleurs de L sont $[\![k]\!]$ pour un certain $k \in \mathbb{N}$.
On va montrer que le coloriage glouton L' pour un ordre de numérotation où les sommets sont de couleurs croissante pour L convient.
Les sommets d'une même couleur pour L étant nécessairement non voisins, ont traite leur coloriage glouton pour L' comme une unique étape, et l'on va montrer la propriété voulue par récurrence forte sur la valeur de la couleur de L de l'étape en cours. On pose $\mathcal{P}(k) : \forall s, L(s) = k.\ L'(s) \leq k$.
Initialisation. ($k = 0$.) Aucun sommet n'est encore colorié pour L', qui affecte donc $L'(s) = 0$ à tous les sommets de L coloriés en 0.
Conservation. On suppose $\mathcal{P}(i)$ pour tout $0 \leq i \leq k$, et l'on considère l'ensemble des sommets s coloriés en $k+1$ par L (ils sont donc non voisins entre eux). Chacun de ces sommets est colorié par l'algorithme glouton en considérant ses voisins déjà coloriés. Par hypothèse de récurrence forte ceux-ci sont de couleurs $\leq k$, d'où $\#C(s) \leq k+1$ et $\min \mathbb{N} \setminus C(s) \leq k+1$. On a donc $L'(s) \leq k+1$ et la propriété est préservée.

La déduction demandée est immédiate : appliquer comme ci-dessus l'algorithme glouton pour l'ordre donné par L utilise au plus autant de couleurs que L, et il suffit alors de considérer un coloriage optimal (qui existe toujours) pour ce dernier.

Les questions 7 et 11 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas les indices de la matrice d'adjacence, c'est-à-dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 11, mais cela n'apporterait aucun bénéfice vis-à-vis de la complexité temporelle du problème.
Une alternative est donnée par l'heuristique « de Welsh-Powell ». L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

12. Écrire une fonction C de signature `void tri_degre(int n, bool **g, int num[n])` qui modifie `num` en l'ordre des `n` sommets de `g` trié par ordre décroissant de leurs degrés.
En déduire une fonction C de signature `void welsh_powell(int n, bool **g, int e[n])` qui implémente l'heuristique « de Welsh-Powell », et justifier le choix de votre algorithme de tri pour la fonction `tri_degre`.

Pour `tri_degre`, on propose :

```
void tri_degre(int n, bool **g, int num[n])
{
    int *deg = malloc(n * sizeof(int));
    for (int s = 0; s < n; s++)
    {
        deg[s] = 0;
        for (int t = 0; t < n; t++)
        {
            if (g[s][t])
            {
                deg[s] += 1;
            }
        }
    }

    for (int i = 0; i < n; i++)
    {
```

```
        int max = deg[i];
        int maxi = i;
        for (int j = i + 1; j < n; j++)
        {
            if (deg[j] > max)
            {
                max = deg[j];
                maxi = j;
            }
        }
        deg[maxi] = deg[i]; // pas d'échange nécessaire
        num[i] = maxi;
    }
    free(deg);
}
```

Le calcul des degrés étant déjà quadratique en le nombre de sommet, un tri quadratique (ici par sélection) est suffisant.
On a ensuite :

```
void welsh_powell(int n, bool **g, int e[n])
{
    int *num = malloc(n * sizeof(int));
    tri_degre(n, g, num);
    glouton(n, g, num, e);
    free(num);
}
```

Algorithme « de Wigderson »

Considérons un graphe G avec n sommets. Supposons que G soit 3-coloriable, mais que l'on ait cette information sans pour autant effectivement disposer d'un 3-coloriage de G. Trouver un 3-coloriage de G pourrait prendre un temps exponentiel en n.
L'algorithme « de Wigderson » permet, pour un graphe G supposé 3-coloriable, de trouver en temps polynomial en n un coloriage de G avec $O(\sqrt{n})$ couleurs (au sens où il existe $C > 0$ tel que pour tout n suffisamment grand, ce coloriage ait au plus $C\sqrt{n}$ couleurs).
Cet algorithme repose sur la propriété établie dans la question qui suit.

13. Soit $k > 0$. Montrer que si G est $(k+1)$-coloriable, alors pour tout sommet s de G le sous-graphe induit par $V(s)$ est k-coloriable.

Soit L un $(k+1)$-coloriage de G, c'est aussi un coloriage de tous ses sous-graphes induits (le contraire serait absurde, puisque les sommets et arêtes de ceux-ci sont des sous-ensembles de ceux de G). Ensuite, pour tout $t \in V(s)$ la présence d'une arête (s,t) implique $L(s) \neq L(t)$: L utilise donc au plus k couleurs pour colorier $V(s)$, et la conclusion suit.

Voici le principe de l'algorithme « de Wigderson ». Soit G un graphe à n sommets, et tel que G est 3-coloriable.

- 1/ On se donne comme couleur initiale $c = 0$.
- 2/ Pour chaque sommet s de G pas encore colorié et ayant au moins $\sqrt{n}$ voisins pas encore coloriés :
  - a/ On 2-colorie, avec les couleurs c et $c+1$, le sous-graphe induit par l'ensemble des voisins de s pas encore coloriés.
  - b/ On incrémente c de 2.
- 3/ Enfin, on utilise l'algorithme glouton (avec un ordre de numérotation quelconque) pour colorier, avec des couleurs supérieures ou égales à c, le sous-graphe induit par l'ensemble des sommets pas encore coloriés.

14. Montrer que l'algorithme de Wigderson appliqué à un graphe G 3-coloriable construit toujours un coloriage, et que ce coloriage utilise un nombre de couleur en $O(\sqrt{n})$, où n est le nombre du sommets du graphe.

Coloriage.
Chaque application de l'étape 2.a colorie au moins un sommet, et le nombre de sommets non coloriés (minoré par zéro) est donc un variant de la boucle de l'étape 2, qui termine. L'étape 3 termine également par la question 10. L'algorithme termine.

Soit L, L' deux coloriages de sous-graphes induits par des sous-ensembles disjoints U, U' de G, on dira qu'ils sont compatibles si l'étiquetage qu'ils définissent ensemble sur le sous-graphe induit par $U \cup U'$ est un coloriage. Il est clair que si L et L' utilisent des ensembles de couleurs disjoints alors ils sont compatibles.
Toutes les étapes (chaque itération de l'étape 2.a et l'étape 3) colorient des sous-graphes induits de G par des sous-ensembles de sommets disjoints deux à deux. L'étape 3 colorie tout sommet de G qui ne serait pas encore colorié, et donc l'union de ces sous-ensembles est l'ensemble des sommets de G. Il suffit donc de montrer que tous ces coloriages sont compatibles.
L'étape 2.a n'échoue jamais car par la question précédente les sous-graphes induits considérés sont $3-1=2$-coloriables (ce sont des sous-graphes induits par les sommets non coloriés du sous-graphe induit par les voisins d'un sommet).
L'étape 2.b garantit que chaque application de l'étape 2.a se fera avec de nouvelles couleurs. Les coloriages produits par l'étape 2 sont donc compatibles deux à deux.
Par la question 10, l'étape 3 construit un coloriage valide du sous-graphe induit qu'elle considère. De plus, celui-ci utilise des couleurs distinctes de toutes celles utilisées à l'étape 2, et il est donc compatible avec les coloriages de cette étape.
L'étiquetage obtenu en considérant l'union de tous les coloriages produits par l'étape 2 et l'étape 3 est donc bien un coloriage.

$O(\sqrt{n})$ couleurs.
Chaque application de l'étape 2.a utilise au plus deux couleurs et colorie au moins $\sqrt{n}$ sommets pas encore coloriés. Cette étape est donc effectuée au plus $n/\sqrt{n} = \sqrt{n}$ fois, et l'ensemble de l'étape 2 utilise au plus $O(\sqrt{n})$ couleurs.
Par la condition de la boucle de l'étape 2, le sous-graphe induit de G (éventuellement) considéré par l'étape 3 est de degré $< \sqrt{n}$. Par la question 10 le coloriage effectué à cette étape utilise donc au plus $O(\sqrt{n})$ couleurs.
Le nombre de couleurs utilisé par l'algorithme entier est donné par la somme de ceux utilisés à chaque étape, et est donc bien un $O(\sqrt{n})$.

Nous allons maintenant implémenter cet algorithme. Commençons par programmer quelques fonctions auxiliaires simples.

15. Écrire une fonction C de signature `bool** sous_graphe(bool **g, int m, int sg[m])` telle que pour un graphe `g` d'au moins `m` sommets et `sg` à valeurs dans $\{0,\dots,m-1\}$ et sans répétition, alors `sous_graphe(g, m, sg)` renvoie la matrice d'adjacence du graphe de sommets $\{0,\dots,m-1\}$, et qui a une arête entre les sommets `s` et `t` si et seulement si `g[sg[s]][sg[t]] == true`.
Dans le cas où `sg` a des valeurs hors de $\{0,\dots,m-1\}$, ou a des répétitions, le comportement de la fonction `sous_graphe` est laissé au choix du candidat.

On propose :
```
bool** sous_graphe(bool **g, int m, int sg[m])
{
    bool **ng = malloc(m * sizeof(bool*));
    for (int i = 0; i < m; i++)
    {
        ng[i] = malloc(m * sizeof(bool));
        for (int j = 0; j < m; j++)
        {
            ng[i][j] = g[sg[i]][sg[j]];
        }
    }
    return ng;
}
```
De complexité temporelle quadratique en `m`.

Nous nous proposons d'utiliser pour les étiquetages la même convention que précédemment : on se donnera un étiquetage `e` de longueur `n` le nombre de sommets de `g`, initialisé à `-1`, et que l'on mettra à jour au fur et à mesure de l'algorithme.

16. Écrire une fonction C de signature :

`int degre_non_colories(int n, bool **g, int e[n], int s)`
telle que `degre_non_colories(n, g, e, s)` renvoie le nombre de voisins `e` de `s` dans `g` de `n` sommets tels que `e[t] == -1`.
Écrire également une fonction C de signature :
`int *voisins_non_colories(int n, bool **g, int e[n], int s)`
telle que `voisins_non_colories(n, g, e, s)` renvoie un tableau représentant l'ensemble des voisins `t` de `s` tels que `e[t] == -1`.

On propose :
```
int degre_non_colories(int n, bool **g, int e[n], int s)
{
    int dnc = 0;
    for (int t = 0; t < n; t++)
    {
        if (g[s][t] && e[t] == -1)
        {
            dnc += 1;
        }
    }
    return dnc;
}

int *voisins_non_colories(int n, bool **g, int e[n], int s)
{
    int dnc = degre_non_colories(n, g, e, s);
    int *vnc = malloc(dnc * sizeof(int));
    int vnci = 0;
    for (int t = 0; t < n; t++)
    {
        if (g[s][t] && e[t] == -1)
        {
            vnc[vnci] = t;
            vnci += 1;
        }
    }
    return vnc;
}
```
Ces deux fonctions sont de complexité temporelle linéaire en `n`.

Pour la prochaine question, on pourra utiliser des fonctions C de signatures :
`int num_non_colories(int n, int e[n])`
`int *non_colories(int n, int e[n])`
`int *inite(int n)`
telles que `non_colories(n, e)` renvoie un tableau représentant l'ensemble des indices `i` de `e` tels que `e[i] == -1` ; `num_non_colories(n, e)` renvoie le cardinal de cet ensemble ; `inite(n)` renvoie l'adresse d'un objet `e` alloué avec `malloc` tel que pour tout `i` dans $\{0,\dots,n-1\}$, `e[i] == -1`.

17. Écrire une fonction C de signature `void wigderson(int n, bool **g, int e[n])` telle que si `g` est 3-coloriable, alors `wigderson(n, g, e)` modifie `e` en un coloriage de `g` obtenu par l'algorithme « de Wigderson » décrit plus haut. On demande une complexité polynomiale en le nombre de sommets `n` de `g`. Le comportement de la fonction est laissé au choix du ou de la candidate lorsque `g` n'est pas 3-coloriable. *Indication : on pourra introduire des fonctions intermédiaires, dont on précisera le rôle.*

On commence par écrire deux fonctions auxiliaires. La première relève le coloriage d'un sous-graphe à celui du graphe dont il est issu, en décalant les couleurs d'un *offset* `c`. La seconde libère la mémoire occupée par un graphe :
```
void releve(int n, int e[n], int m, int sg[m], int es[m], int c)
{
    for (int s = 0; s < m; s++)
    {
```

```
        e[sg[s]] = es[s] + c;
    }
}

void freeg(int n, bool **g)
{
    for (int i = 0; i < n; i++)
    {
        free(g[i]);
    }
    free(g);
}
```

On suit ensuite la description de l'algorithme :

```
void wigderson(int n, bool **g, int e[n])
{
    int c = 0;

    for (int s = 0; s < n; s++) // n itérations en O(n^2)
    {
        if (e[s] == -1)
        {
            int dnc = degre_non_colories(n, g, e, s); // O(n)
            if (n / dnc <= dnc)
            {
                int *vnc = voisins_non_colories(n, g, e, s); // O(n)
                bool **sivnc = sous_graphe(g, dnc, vnc); // O(n^2)
                int *es = inite(dnc); // O(n)
                deux_col(dnc, sivnc, es); // O(n^2)
                releve(n, e, dnc, vnc, es, c); // O(n)
                c += 2;
                free(vnc);
                freeg(dnc, sivnc);
                free(es);
            }
        }
    }
    int nnc = num_non_colories(n, e); // O(n)
    int *nc = non_colories(n, e); // O(n)
    bool **sinc = sous_graphe(g, nnc, nc); // O(n^2)
    int *en = inite(nnc); // O(n)
    welsh_powell(nnc, sinc, en); // O(n^2)
    releve(n, e, nnc, nc, en, c);
    free(nc);
    freeg(nnc, sinc);
    free(en);
}
```

La complexité temporelle totale de cette fonction est un $O(n^3)$, et même un $O(n^{2.5})$ puisqu'au plus $\sqrt{n}$ itérations de la première boucle auront un coût non constant.