Concours CPGE EPITA-IPSA-ESME
2024

Corrigé de l'épreuve de Sciences du numérique MPI

Le sujet est composé d'un problème et d'un questionnaire à choix multiples. Le questionnaire à choix multiples devra être inséré dans votre copie. Les fonctions à produire dans ce sujet devront être rédigées dans le langage spécifié dans l'intitulé des parties et se conformer aux restrictions décrites ci-après. Il est possible d'écrire des fonctions auxiliaires non explicitement demandées à condition de les documenter et de les définir avant d'en faire usage.

Restrictions des langages C et OCaml

Conformément au programme de l'enseignement d'informatique en classe de MPI, sont autorisés les traits et éléments techniques à connaître des langages C et OCaml. En particulier, l'usage des fonctions et modules OCaml ci-après sera considéré comme nul :

```
Array.exists        List.exists     module Hashtbl
Array.for_all       List.filter     module Queue
Array.init          List.fold_left  module Stack
Array.iter          List.for_all
Array.make_matrix   List.iter
Array.map           List.map
Array.mem           List.mem
```

Problème – Coloration de graphes

On considère dans ce problème des **graphes non orientés** $G = (V, E)$, à $n \in \mathbb{N}^\star$ sommets, où $V = [\![0, n-1]\!]$ est l'ensemble des sommets de G et E l'ensemble de ses arêtes. E est un ensemble de parties à 2 éléments de V.

On dit que $c : V \to \mathbb{N}$ est une **coloration** des sommets de G lorsque deux sommets voisins v_1 et v_2 de V ont toujours des **couleurs** $c(v_1)$ et $c(v_2)$ distinctes. Formellement : $\forall \{v_1, v_2\} \in E,\ c(v_1) \neq c(v_2)$.
Dans le cas où une coloration de G existe, on dira que G est **colorable**.
De plus lorsqu'il existe $k \in \mathbb{N}^\star$ tel que $c(V) \subseteq [\![0, k-1]\!]$, on dira que c est une ***k*-coloration** de G et que le graphe est ***k*-colorable**.
Une coloration d'une graphe G sera dite **optimale** ou **minimale** lorsqu'il n'existera pas de coloration de G avec moins de couleurs.

On appelle **graphe induit** de $G = (V, E)$ par un sous-ensemble V' de ses sommets, le graphe $G' = (V', E')$ où E' est l'ensemble des arêtes de E ayant leurs extrémités dans V'. Formellement $E' = \{\{v_1, v_2\} \in E \,|\, (v_1, v_2) \in V' \times V'\}$.

On appelle **stable** d'un graphe $G = (V, E)$, tout sous-ensemble non vide V' de sommets de G vérifiant :

$$\forall (v_1, v_2) \in V' \times V',\ \{v_1, v_2\} \notin E$$

Autrement dit, V' est un stable de G si et seulement si le graphe induit par l'ensemble non vide de sommets V' ne contient pas d'arêtes. On dira qu'un stable S de G est **maximal** lorsque le seul stable de G contenant S est S lui-même.

1 Un algorithme glouton – Langage C

Allocation mémoire

Avec les notations précédentes, un graphe G sera représenté par la donnée de son nombre de sommets n et par sa matrice d'adjacence M. Ainsi $M[i][j]$ vaudra 1 lorsque le graphe G contiendra l'arête $\{i, j\}$ et 0 sinon.

On adoptera la structure suivante pour le représenter :

```
struct graph
{
    int   n;
    int **M;
};
typedef struct graph graph;
```

Question 1 Écrire une fonction de signature `graph graph_make(int n)` retournant un graphe à n sommets et sans arêtes.

```
graph graph_make(int n):
{
    graph g;
    g.n = n;
    g.M = malloc(n*sizeof(int*));
    for (int i=0; i<n; i++)
    {
        g.M[i] = malloc(n*sizeof(int));
        for (int j=0; j<n; j++)
        {
            g.M[i][j]=0;
        }
    }
    return g;
}
```

Question 2 Écrire une fonction de signature `void graph_free(graph g)` libérant la mémoire allouée à un graphe.

```
void graph_free(graph g)
{
    for (int i=0; i<g.n; i++)
    {
        free(g.M[i]);
    }
    free(g.M);
}
```

Itérateur de permutations

On représente une permutation σ de $[\![0, n-1]\!]$ par le tableau d'entiers $[\sigma(0), \cdots, \sigma(n-1)]$.

On considère l'ordre les permutations de $[\![0, n-1]\!]$ comme étant l'ordre lexicographique des tableaux qui les représentent :

$$[a_1] < [b_1] \Leftrightarrow a_1 < b_1$$

$$\forall p \geqslant 2,\ [a_1, \cdots, a_p] < [b_1, \cdots, b_p] \Leftrightarrow a_1 < b_1 \text{ ou } \begin{cases} a_1 = b_1 \\ [a_2, \cdots, a_p] < [b_2, \cdots, b_p] \end{cases}$$

Ainsi $[2,3,1,5,4] < [2,3,4,1,5] < [2,3,4,5,1] < [2,3,5,1,4]$.

Question 3 Donner les 3 permutations qui suivent $[2,3,5,1,4]$ dans l'ordre lexicographique.

$[2,3,5,1,4] < [2,3,5,4,1] < [2,4,1,3,5] < [2,4,1,5,3]$

Question 4 Écrire une fonction de signature `int* identite(int n)` retournant la permutation identité de $[\![0, n-1]\!]$.

```
int* identite(int n)
{
    int* p = malloc(n*sizeof(int));
    for (int i=0; i<n; i++)
    {
        p[i] = i;
    }
    return p;
}
```

Question 5 Écrire une fonction de signature `void inverse(int* p, int i, int j, int n)` modifiant la permutation de $[\![0, n-1]\!]$ passée en paramètre. Cette fonction inversera l'ordre des éléments des cases dont l'indice appartient à $[\![i, j]\!]$. Cette fonction ne fera rien dans le cas où $i \geqslant j$ et on supposera que cette fonction n'est appelée que lorsque i et j sont dans $[\![0, n-1]\!]$.

```
void inverse(int* p, int i, int j, int n)
{
    int tmp;
    while (i<j)
    {
        tmp  = t[i];
        t[i] = t[j];
        t[j] = tmp;
        i++;
        j--;
    }
}
```

Question 6 Écrire une fonction de signature `int indice_pps(int* p, int i, int n)` prenant en paramètre une permutation de $[\![0, n-1]\!]$, un entier i de $[\![0, n-1]\!]$ et n. Cette fonction retournera, parmi les cases dont l'indice est strictement supérieur à i et dont la valeur est supérieure à $p[i]$, l'indice de la case de valeur minimale. On suppose que cette fonction ne sera appelée que lorsque cet indice existera.

```
int indice_pps(int* p, int i, int n)
{
    int j=i+1;
    while (j<n && p[j]<=p[i])
    {
        j += 1;
    }
    for(k=j+1; k<n; k++)
    {
        if (p[k]>p[i] && p[k]<p[j])
        {
            j = k;
        }
    }
    return j;
}
```

Question 7 Écrire une fonction de signature `bool suivante(int* p, int n)` qui modifie la permutation de l'ensemble $[\![0, n-1]\!]$ passée en paramètre par la suivante dans l'ordre lexicographique. Lorsque la permutation suivante n'existe pas, la fonction renverra la valeur `true` ; dans le cas contraire, elle renverra la valeur `false` sans modifier la permutation.

```
bool suivante(int* p, int n)
{
    int i = n-1;
    int j;
    int tmp;
    bool continuer = true;
    bool derniere;
    while(i>0 && continuer)
    {
        i--;
        continuer = ( p[i] > p[i+1] );
    }
    derniere = (i==-1);
    if (!derniere)
    {
        inverse(p,i+1,n-1,n);
        j = indice_pps(p,i,n);
        tmp = p[i];
        p[i] = p[j];
        p[j] = tmp;
    }
    return derniere;
}
```

Implémentation de l'algorithme glouton

On considère le graphe $G_t = (V_t, E_t)$ représenté par la figure 1.

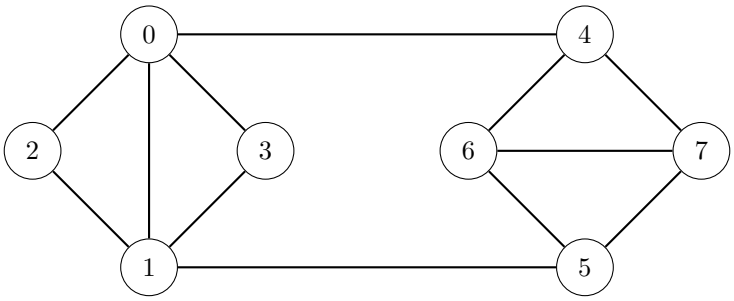
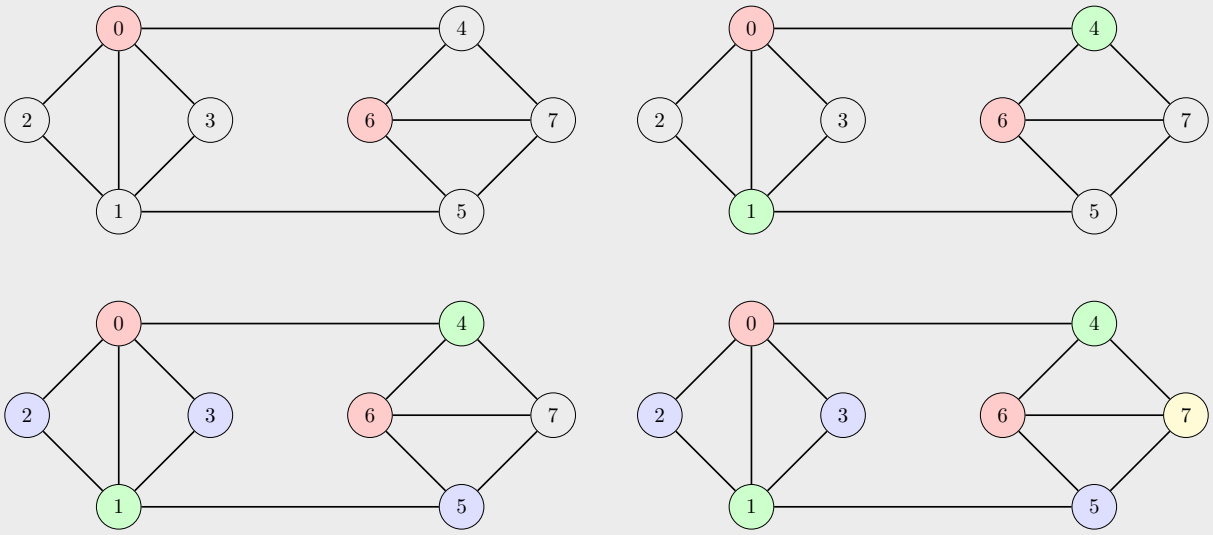


FIGURE 1 – Le graphe G_t

On considère un algorithme glouton de coloration prenant en paramètre une permutation σ de $[\![0, n-1]\!]$ et un graphe G à n sommets à colorer. L'algorithme parcourt le graphe dans l'ordre des sommets $\sigma(0), \cdots, \sigma(n-1)$ et colore chaque sommet non coloré par la couleur 0 si aucun de ses voisins n'est déjà coloré avec cette couleur. L'algorithme recommence alors avec la couleur suivante jusqu'à ce que tous les sommets soient colorés.

Question 8 Appliquer l'algorithme glouton au graphe G_t.



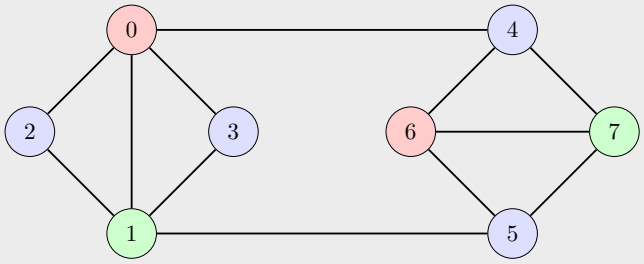
On obtient finalement la coloration suivante :

s	0	1	2	3	4	5	6	7
$c(s)$	0	1	2	2	1	2	0	4

Question 9 Montrer que le graphe G_t est 3-colorable.

La coloration suivante permet de garantir que G_t est 3-colorable :

s	0	1	2	3	4	5	6	7
$c(s)$	0	1	2	2	2	2	0	1



Question 10 Écrire la fonction de signature `int* glouton(graph g, int* p)` prenant en paramètre un graphe non orienté g et une permutation de l'ensemble de ses sommets. Cette fonction retournera la coloration produite par l'algorithme glouton.

```
bool est_couleur_possible(graph g, c, i, couleur)
{
    possible = true;
    int j=0;
    while (possible && j<n)
    {
        if (g.M[i][j]==1 && c[j] == couleur)
        {
            possible = false;
        }
        j++;
    }
    return possible;
}
```

```
int* glouton(graph g, int* p)
{
    int* c = malloc(g.n*sizeof(int));
    int sommets = g.n;
    int couleur = 0;
    for (int i=0; i<g.n; i++)
    {
        c[i] = -1;
    }
    while (sommets>0)
    {
        for (int i=0; i<g.n; i++)
        {
            if (c[i]==-1 && est_couleur_possible(g, c, i, couleur))
            {
                c[i] = couleur;
                sommets--;
            }
        }
        couleur++;
    }
    return c;
}
```

Algorithme de coloration optimale

On fixe dans cette partie un graphe $G = (V, E)$ à $n \in \mathbb{N}^\star$ sommets.

Pour chaque entier $k \in \mathbb{N}^\star$ et toute k-coloration, on définit $(\star_k)$ la proposition suivante :
$\forall i \in [\![0, k-1]\!]$ « L'ensemble C_i des sommets de couleur i est un stable maximal du sous-graphe de $G = (V, E)$ induit par les sommets $V \backslash (C_0 \cup \cdots \cup C_{i-1})$ »

Question 11 En notant k le nombre de couleurs retourné par l'algorithme glouton sur une permutation des sommets de G, montrer que la coloration obtenue vérifie $(\star_k)$.

L'algorithme glouton commence par colorer des sommets de $G = (V, E)$ avec comme première couleur 0 jusqu'à ce que cela ne soit plus possible.
Notons C_0 l'ensemble de ces sommets et G_0 le sous-graphe de G induit par ces sommets. C_0 est un stable maximal de G car :

- les sommets de G_0 ont été colorés de la même couleur donc C_0 est un stable de G ;
- ce stable est maximal sinon on aurait un stable C contenant C_0 et de cardinal strictement supérieur. Ainsi il existerait un sommet $s \in C \backslash C_0$ qui n'a pas été coloré par l'algorithme glouton avec la valeur 0 et qui admettait donc un voisin v coloré par l'algorithme avec la valeur 0. Dès lors $v \in C_0 \subseteq C$ donc v et s sont deux sommets de C voisins et C ne serait donc pas un stable.

En continuant récursivement le procédé sur le sous graphe de G induit par les sommets $V \backslash C_0$, on obtient la propriété demandée.

Question 12 On considère une coloration à k couleurs satisfaisant $(\star_k)$. Montrer qu'elle est produite par l'exécution de l'algorithme glouton sur au moins une permutation des sommets de G.

Considérons une k-coloration de G vérifiant $(\star_k)$ et notons $s_1, \cdots, s_n$ les n sommets de G numérotés de façon à ce que les sommets de s_1 à s_{p_0} soient les sommets de C_0, $\cdots$, les sommets de $s_{p_0+\cdots+p_{k-2}+1}, \cdots, s_n$ soient les sommets de C_{k-1}.

On appliquant l'algorithme glouton à cette numérotation des sommets, l'algorithme colore $s_1, \cdots, s_{p_0}$ avec la couleur 0 puisque ces sommets sont ceux de C_0 qui est un stable (maximal) de G d'après $(\star_k)$. L'algorithme poursuit en es-

sayant de colorer les autres sommets avec la couleur 0 mais sans y parvenir car sinon C_0 ne serait pas maximal dans G.

En procédant récursivement avec les sommets restant et une nouvelle couleur, on obtient que l'algorithme glouton redonne la coloration satisfaisant $(\star_k)$ considérée au début de la preuve. Pour toute coloration satisfaisant $(\star_k)$ il existe donc bien une permutation des sommets de G pour laquelle l'algorithme glouton produit cette coloration.

Question 13 Soit $k \in \mathbb{N}^\star$ le nombre minimum de couleurs des colorations de G. Montrer qu'il existe au moins une k-coloration vérifiant $(\star_k)$.

On se donne une coloration minimale de G. On note k son nombre de couleurs et pour chaque entier $i \in [\![0, k-1]\!]$, on désigne par C_i l'ensemble des sommets de couleur i.

Puisque l'on a une coloration, nécessairement C_i est un stable de G. On prolonge alors C_i en stable maximal C_i' pour chaque entier $i \in [\![0, k-1]\!]$. On applique à tous les sommets de $C_0' \backslash C_0$ la couleur 0. La nouvelle coloration est toujours minimale car elle ne modifie pas le nombre de couleurs.

On applique récursivement ce procédé sur le sous graphe de $G = (V, E)$ induit par les sommets $V \backslash C_0$ et on obtient donc une coloration minimale vérifiant $(\star_k)$.

Question 14 Écrire une fonction `int* coloration(graph g)` qui retourne une coloration optimale du graphe passé en paramètre.

```
int nb_couleurs(int* c, int n)
{
    int max = 0;
    for (int i=0; i<n; i++)
    {
        if (c[i]>max)
        {
            max = c[i];
        }
    }
    return (max+1)
}

int* coloration(graph g)
{
    int* c;
    int* tmp_c;
    int  min_couleurs = g.n + 1;
    int* p = identite(g.n);
    bool continuer = true;
    while (continuer)
    {
        tmp_c = glouton(g, p);
        if (nb_couleurs(tmp_c, g.n) < min_couleurs)
        {
            c = tmp_c;
        }
        continuer = suivante(p, g.n);
    }
    return c
}
```

2 Complexité

On considère le problème de décision **3-COLOR** suivant :

> « Étant donné un graphe non orienté G à au moins 3 sommets, existe-t-il une 3-coloration de ses sommets ? »

On considère également le problème de décision **3-DOMINATION** suivant :

> « Étant donné un graphe non orienté G à au moins 3 sommets, existe-t-il un ensemble de 3 sommets de G tel que chaque sommet de G soit à une distance au plus un de l'un d'eux ? »

Dans le cas où un tel ensemble existe, on dira qu'il s'agit d'un **ensemble 3-dominant** du graphe G.

Question 15 Montrer que 3-DOMINATION est dans P.

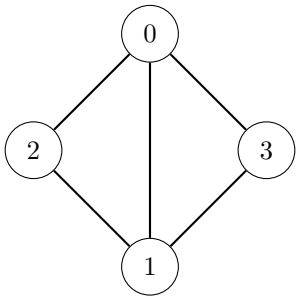
Soit G un graphe à $n \geqslant 3$ sommets, alors il existe $\binom{n}{3} = \frac{n!}{3!(n-3)!} = \frac{n(n-1)(n-2)}{6}$ ensembles de 3 sommets du graphe. Il suffit alors de déterminer pour chaque ensemble de 3 sommets du graphe leur distance aux autres sommets du graphe ; par exemple à l'aide d'un parcours en largeur. On a une 3-DOMINATION si pour au moins un ensemble de 3 sommets du graphe, tous les sommets sont à distance au plus 1 de l'un d'entre eux (il suffit de regarder le minimum des 3 distances).
La complexité finale reste donc bien polynomiale et 3-DOMINATION est dans P.

On considère à présent la construction suivante :

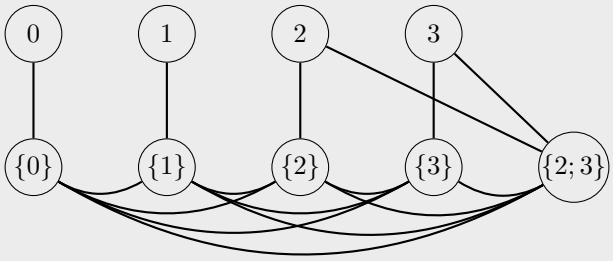
À partir d'un graphe $G = (V, E)$, on construit le graphe $G' = (V', E')$ tel que :

1. $V' = V \cup V_s$ où V_s est l'ensemble des stables de G ;
2. E' est l'union de l'ensemble de toutes les arêtes possibles entre les sommets de V_s et de l'ensemble des arêtes reliant les éléments de $v \in V$ aux éléments de $v_s \in V_s$ pour lesquels v est un sommet du stable v_s.

Question 16 Représenter le graphe G' associé au graphe suivant :



Soit $V_s = \{\{0\}; \{1\}; \{2\}; \{3\}; \{2; 3\}\}$ l'ensemble des stables de G, on en déduit G' :



Question 17 Montrer qu'un graphe G admet une 3-coloration si et seulement si G' admet un ensemble dominant de taille 3.

$\Rightarrow$ Supposons que G admette une 3-coloration, et notons C_i l'ensemble des sommets colorés avec la couleur $i \in [\![1,3]\!]$.
Les sommets de chaque ensemble C_i ont la même couleur donc il n'existe aucune arête entre eux. Les C_i forment dont des stables du graphe G et sont donc des sommets du graphe $G' = (V \cup V_s, E)$.
De plus :

- tout sommet de V_s est à distance au plus 1 de ces 3 sommets C_1, C_2 et C_3 par construction de G' ;
- tout sommet $v \in V$ a une couleur $i \in [\![1,3]\!]$ et est alors à distance 1 du sommet C_i car $v \in C_i$.

Ainsi tout sommet de G' est à distance au plus 1 d'un des 3 sommets C_1, C_2 et C_3.
Finalement, si G admet une 3-coloration, G' admet un ensemble 3-dominant.

$\Leftarrow$ Supposons que G' dispose d'une 3-domination s_1, s_2, s_3 de sommets de G'.
Si l'un des s_i est également un sommet de G, on peut le remplacer par le sommet étiqueté par le stable $\{s_i\}$ de façon à conserver une 3-domination (dans le cas où le sommet $\{s_i\}$ était lui aussi dans la 3-domination, il suffit alors de placer dans la domination n'importe quel autre stable non retenu et il y en a nécessairement car on a au moins 3 sommets donc 3 stables). En effet dans ce cas le sommet s_i devient donc à distance 1 du sommet $\{s_i\}$ retenu pour la domination, les sommets représentant des stables sont à distance 1 de $\{s_i\}$ et les sommets de G étaient déjà, soit dans la domination, soit à distance 1 d'un stable de la domination.

On suppose donc à présent, sans perte de généralité, que s_1, s_2 et s_3 sont des sommets de G' étiquetés par des stables de G. Tous les sommets de G sont à distance 1 d'un de ces 3 stables donc ces 3 stables forment une partition de l'ensemble des sommets de G. Les sommets d'un même stable pouvant être colorés d'une même couleur, on en déduit qu'il existe bien une 3-coloration du graphe G.

Ainsi G admet une 3-coloration si et seulement si G' admet un ensemble dominant de taille 3.

Question 18 Justifier que 3-COLOR est dans NP. Peut-on en déduire que $P = NP$? On justifiera sa réponse.

La vérification qu'une supposée 3-coloration est ou non valide s'effectue bien en temps polynomial ; en effet, il suffit de parcourir la coloration pour s'assurer qu'elle contient au plus 3 couleurs et de vérifier que les extrémités de chaque arête ont toujours une couleur différente. On obtient donc une complexité temporelle en $\mathcal{O}(n^2)$ où n est le nombre de sommets du graphe. Donc 3-COLOR est bien dans NP.

De plus d'après les question 15 et 17, on a bien une réduction du problème 3-COLOR vers le problème 3-DOMINATION qui est dans P mais cette réduction ne s'effectue pas en temps polynomial car le nombre de sommets du graphe G' peut-être exponentiel en le nombre de sommets de G ; en effet, le nombre de parties (et donc de stables possibles) d'un ensemble à $|V|$ éléments est $2^{|V|}$ et cela se produit notamment pour les graphes sans arêtes.

Nous n'avons donc malheureusement pas prouvé que $P = NP$.

3 Coloration distribuée – Langage OCaml

Dans cette partie on représente les graphes par liste d'adjacence au moyen du type `int list array`.

L'objectif à présent est de simuler l'exécution d'un algorithme de coloration de graphe distribué. On identifie dans ce cadre chaque sommet du graphe à une machine reliée à d'autres machines conformément aux arêtes du graphe.

Chaque sommet du graphe peut :

- exécuter des instructions ;
- connaître la liste de ses voisins ;
- envoyer des messages (informations) à un ou plusieurs de ses voisins ;
- lire les informations qu'il a reçues ;
- choisir sa propre couleur.

Les sommets ne peuvent donc pas accéder directement à l'information globale de coloration des autres sommets ; pour cela ils sont contraints de s'envoyer des messages.

Dans la mesure où nous effectuons une simulation dont seul le résultat nous intéresse, nous pourrons dans cette partie écrire des algorithmes sans ces limitations mais dont le résultat devra être le même que si ces contraintes étaient effectives. De plus, dans ce sujet lorsque l'on présentera un algorithme distribué, on considèrera que les sommets exécutent les instructions de façon synchrone : tous les sommets concernés par l'étape 1 exécutent les instructions de l'étape 1, puis tous les sommets concernés par l'étape 2 exécutent les instructions de l'étape 2, et ainsi de suite.

Un algorithme rapide de coloration

On considère à présent que les couleurs sont des entiers représentés par des mots binaires et on présente l'algorithme distribué `fast_color` suivant :

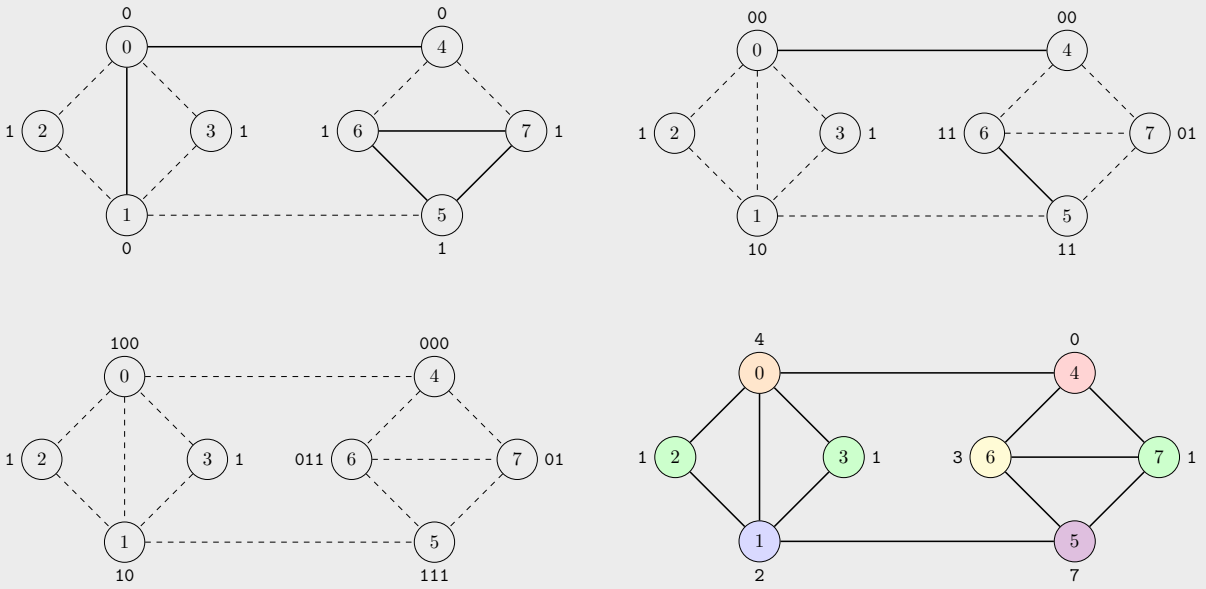
1. au départ aucun sommet n'est coloré et les sommets connaissent la liste de leurs voisins qu'ils considèrent comme actifs. Initialement, chaque sommet considère comme actifs chacun de ses voisins ;
2. chaque sommet non coloré génère aléatoirement et de façon uniforme un bit ;
3. chaque sommet non coloré envoie le bit généré à ses voisins encore considérés comme actifs ;
4. chaque sommet non coloré lit les bits reçus de ses voisins et ne considère plus comme actifs les voisins ayant généré un bit différent du sien ;
5. chaque sommet ne considérant plus aucun de ses voisins comme actif prend pour couleur l'entier représenté par le mot binaire obtenu en concaténant les bits qu'il a généré (du dernier bit généré au premier – de gauche à droite).
6. si il reste des sommets non colorés, on recommence à l'étape 2, sinon l'algorithme se termine.

On notera $c(s)$ le nombre entier représentant la couleur produite par l'algorithme `fast_color` pour le sommet s.

Question 19 Expliquer pourquoi, lorsque l'algorithme `fast_colour` termine, deux sommets voisins ont nécessairement une couleur différente ?

Dans cet algorithme, deux sommets se considèrent comme étant des voisins actifs tant qu'ils génèrent la même suite de bits. Lorsque l'algorithme termine, il n'existe plus de sommets considérant avoir un voisin actif donc deux voisins ont toujours généré, à un moment où à un autre, un bit différent. La suite de bits ainsi construite par ces sommets est donc distincte, ils n'ont donc pas la même coloration.

Question 20 Exécuter l'algorithme `fast_colour` sur le graphe G_t (défini en figure 1) en considérant que l'on exécute les actions des sommets dans l'ordre croissant de leurs étiquettes et que les bits aléatoirement générés sont (de gauche à droite), les suivants : `001101110101101010`. On pourra représenter par des pointillets les arêtes du graphe entre tout couple de sommets se considérant mutuellement inactifs.



On obtient finalement la coloration suivante :

s	0	1	2	3	4	5	6	7
$c(s)$	4	2	1	1	0	7	3	1

Question 21 Écrire une fonction de signature `bin_to_int : int list -> int` prenant un mot binaire représenté par une liste d'entiers de $\{0,1\}$ et retournant le nombre entier qu'il représente (on considèrera le bit de poids le plus fort comme étant le premier élément de la liste).

```
let bin_to_int l =
  let rec calcule l n = match l with
  | []   -> n
  | t::q -> calcule q (2*n+t)
  in
    calcule l 0
;;
```

Question 22 Écrire une fonction prenant en paramètre un graphe G représenté par la liste d'adjacence de ses sommets et une fonction générant un nombre entier aléatoire de $\{0,1\}$. Cette fonction renverra un tableau indexé par les numéros des sommets du graphe G et dont les cases contiennent l'entier représentant sa couleur. La fonction aura pour signature `fast_color : int list array -> (unit -> int) -> int array` et on sera libre de créer des fonctions auxiliaires utiles.

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

let rec remplir_tableau tab l = match l with
| []   -> ()
| t::q -> tab.(t) <- true;
          remplir_tableau tab q
;;

let voisin_encore_actif a =
  let res = ref false and i=ref 0 and n=Array.length a in
    while not !res && !i<n do
      res := a.(!i);
      incr i
    done;
    !res
;;

let fast_color g random_bit =
  let n = Array.length g in
    let cb = Array.make n [] and c = Array.make n (-1) in
      let actifs = Array.make_matrix n n false and nb_sommets_colores = ref 0 in
          for i=0 to n-1 do
            remplir_tableau actifs.(i) g.(i)
          done;
          while !nb_sommets_colores < n do
            for i=0 to n-1 do
              if (c.(i) == -1) then
                cb.(i) <- (random_bit ())::(cb.(i));
            done;
            for i=0 to n-1 do
              for j=i+1 to n-1 do
                if actifs.(i).(j) && List.hd cb.(i) != List.hd cb.(j) then
                (
                  actifs.(i).(j) <- false;
```

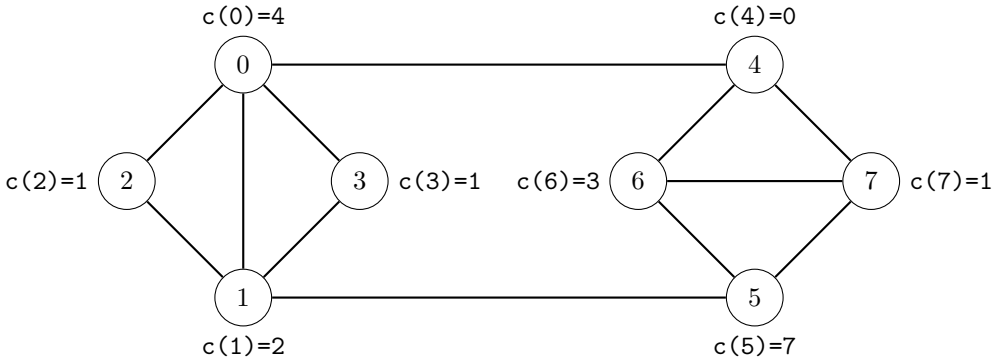
```
                actifs.(j).(i) <- false
              )
            done;
          done;
          for i=0 to n-1 do
            if (c.(i) == -1 && not (voisin_encore_actif actifs.(i))) then
            (
              c.(i) <- bin_to_int cb.(i);
              incr nb_sommets_colores
            )
          done;
        done;
        c
;;
```

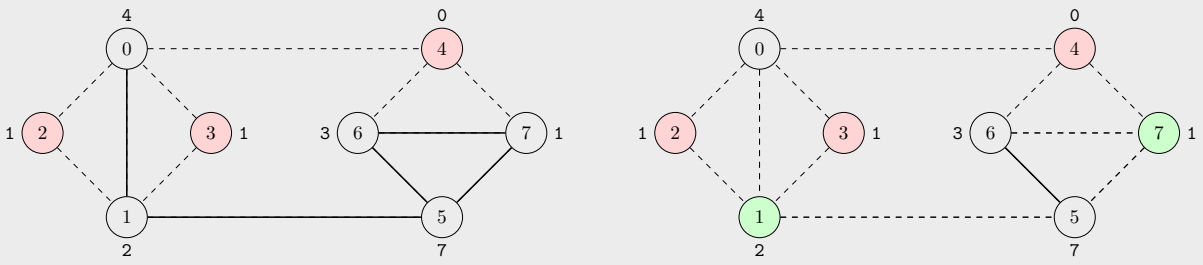
Un algorithme pour améliorer la coloration

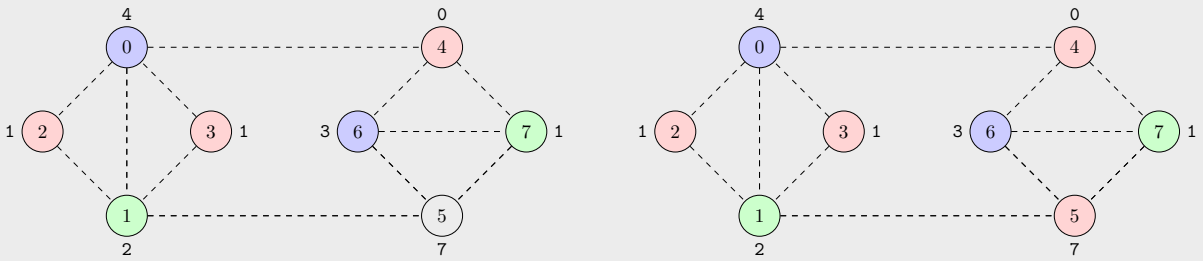
Afin de réduire le nombre de couleurs utilisées par la coloration produite, on envisage un second algorithme distribué, appelé `reduce`, opérant après le précédent et visant à attribuer une nouvelle couleur aux sommets :

1. Chaque sommet stocke une liste ℓ, initialement vide, des couleurs utilisées par ses voisins ;
2. Chaque sommet s sans nouvelle couleur envoie à ses voisins la valeur $c(s)$ produite à l'issue de `fast_colour` ;
3. Chaque sommet s sans nouvelle couleur et dont la valeur $c(s)$ est inférieure à celle de chacun de ses voisins se voit attribuer comme nouvelle couleur la plus petite couleur absente de sa liste ℓ. Il indique ensuite à tous ses voisins la couleur retenue ;
4. Chaque sommet sans nouvelle couleur ajoute à sa liste ℓ la couleur de chacun de ses voisins s'étant vu attribué une nouvelle couleur à l'étape 3 ;
5. Tant qu'il reste des sommets sans nouvelle couleur, l'algorithme recommence à l'étape 2.

Question 23 Éxécuter l'algorithme `reduce` dans le cas où la coloration obtenue initialement était la suivante :

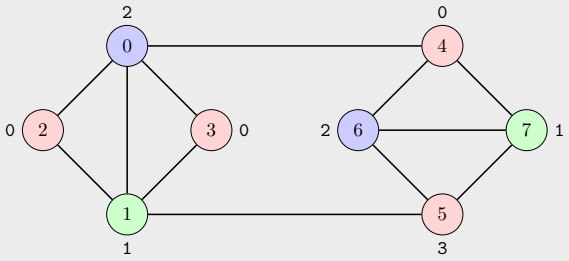






On aboutit donc à la 3-coloration suivante :

s	0	1	2	3	4	5	6	7
$c(s)$	2	1	0	0	0	3	2	7



Question 24 Écrire une fonction de signature `reduce : int list array -> int array -> int array` qui étant donné un graphe G représenté par sa liste d'adjacence et une coloration retournée par `fast_color` pour G, renvoie la coloration obtenue après exécution de l'algorithme distribué `reduce`. On sera libre de concevoir des fonctions auxiliaires aidant à l'élaboration de la fonction.

```
let rec insere x l = match l with
| []   -> [x]
| t::q -> if x<t then x::l
          else if x==t then l
          else t::(insere x q)
;;

let premiere_couleur l =
  let rec tester l i = match l with
  | []   -> i
  | t::q -> if i<t then i
            else if i=t then tester q (i+1)
            else failwith "Impossible"
  in
    tester l 0
;;

let reduce g c =
  let n=Array.length g in
    let c2 = Array.make n (-1) and a_colorer = ref [] and nb_sommets = ref 0 in
      let rec est_a_colorer s = (c2.(s) == -1) && sommets_sans_c2_plus_grands g.(s) c.(s)
      and sommets_sans_c2_plus_grands l coul = match l with
      | []   -> true
      | t::q -> (c2.(t) != -1 || coul < c.(t)) && (sommets_sans_c2_plus_grands q coul)
      and colorer_sommets l = match l with
      | []   -> ()
      | t::q -> colorer_sommet t;
                colorer_sommets q
      and colorer_sommet s =
        c2.(s) <- premiere_couleur (couleurs_sommets g.(s));
        incr nb_sommets
```

```
    and couleurs_sommets l = match l with
    | []   -> []
    | t::q -> if c2.(t) != -1 then insere c2.(t) (couleurs_sommets q)
              else couleurs_sommets q
    in
      while !nb_sommets < n do
        for s=0 to n-1 do
          if est_a_colorer s then
            a_colorer := insere s !a_colorer
        done;
        colorer_sommets !a_colorer;
        a_colorer := []
      done;
      c2
```

Question 25 Montrer que l'algorithme distribué `reduce` termine.

A chaque fois que l'on passe à l'étape 4, un au moins des sommets du graphe va se colorer. En effet, il existe encore au moins un sommet sans nouvelle couleur et l'ensemble des couleurs $c(s)$ de ces sommets admet, en tant qu'ensemble non vide de $\mathbb{N}$, un plus petit élément. Ainsi le sommet ayant obtenu cette couleur lors de l'exécution de `fast_colour` va obtenir sa nouvelle coloration. Finalement à chaque passage à l'étape 4, au moins un sommet obtient sa nouvelle coloration, il y a donc un nombre strictement décroissant de sommets restant à colorer donc l'algorithme termine.

Question 26 Montrer que la coloration ainsi obtenue contient au plus $\Delta+1$ couleurs différentes où Δ est le maximum des degrés des sommets du graphe.

Lorsqu'un sommet s se voit attribuer une couleur, il prend la plus petite valeur disponible.
Si s est de degré d, il existe forcément une couleur disponible dans $\{0, \cdots, d\} \subseteq \{0, \cdots, \Delta\}$.
Ainsi chaque sommet se voit attribuer une couleur dans $\{0, \cdots, \Delta\}$ et on obtient bien un $(\Delta+1)$-coloration.

QCM – MPI

Numéro de candidat :
Document à insérer dans votre copie d'examen

Dans ce questionnaire à choix multiples, chaque question comporte une ou plusieurs bonnes réponses.
Chaque réponse correcte fait gagner des points, mais chaque réponse fausse annule tous les points de la question.
Les questions sont majoritairement formulées au pluriel par commodité d'expression. Cela n'implique pas nécessairement qu'elles admettent plusieurs réponses correctes.

---

1. Le nombre de langages sur l'alphabet $\{a, b\}$ dont l'étoile est un langage fini est :
   - ☐ 0
   - ☐ 1
   - ■ 2
   - ☐ infini

2. Parmi les langages suivants, indiquer ceux qui sont réguliers :
   - ■ $\{a^n \mid n \in \mathbb{N}\}$
   - ☐ $\{a^n b^n \mid n \in \mathbb{N}\}$
   - ■ $\{(ab)^n \mid n \in \mathbb{N}\}$
   - ■ $\{a^n b^p \mid n \in \mathbb{N},\ p \in \mathbb{N},\ n + p = 2024\}$

3. Parmi les langages suivants, décrits de façon ensembliste ou dénotés à l'aide d'une expression régulière, sont engendrés par la grammaire : $S \to bSS \mid SbS \mid SSb \mid a$:
   - ☐ $\{\omega \in \{a, b\}^\star \mid\ |\omega|_a > |\omega|_b\}$
   - ■ $\{\omega \in \{a, b\}^\star \mid\ |\omega|_a = |\omega|_b + 1\}$
   - ☐ $a(ba)^\star \mid b(ab)^+$
   - ☐ $a(ba)^\star \mid b(ab)^\star$

4. Quelles grammaires, parmi les suivantes, permettent de produire le mot $3p1t4$ sur l'alphabet $\{d, 3, z, 1, p, 4, t\}$:
   - ■ $S \to 3S \mid S4 \mid pS \mid St \mid 1$
   - ■ $S \to SSS \mid SS \mid 1 \mid 3 \mid 4 \mid p \mid t \mid z$
   - ☐ $S \to S1S \mid SpS \mid StS \mid 3 \mid 4$
   - ■ $S \to SpS \mid StS \mid 1 \mid 3 \mid 4$

5. Quelles expressions régulières parmi les suivantes, dénotent un langage contenant le mot 314159 :
   - ☐ $(((1|9)(3|4)^\star)\ ((1|3)(5|9)^\star)^\star)^\star$
   - ☐ $(((1|9)(3|4)^\star)^\star((1|3)(5|9)\ )^\star)^\star$
   - ☐ $(((1|9)(3|4)^\star)\ ((1|3)(5|9)^\star)^\star)^\star$
   - ■ $(((1|9)(3|4)^\star)^\star((1|3)(5|9)^\star)\ )^\star$

6. On considère un alphabet contenant au moins deux caractères. Indiquer parmi les propriétés suivantes celles qui sont vraies :
   - ■ Il y a une infinité de langages réguliers.
   - ☐ Tout langage inclus dans un langage régulier est régulier.
   - ■ Tout automate fini a pour langage reconnu un langage régulier.
   - ■ Une intersection finie de langages réguliers est un langage régulier.

Fin du corrigé