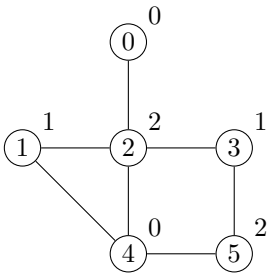
COMPOSITION D'INFORMATIQUE n°4

Corrigé

⋆ ⋆ ⋆

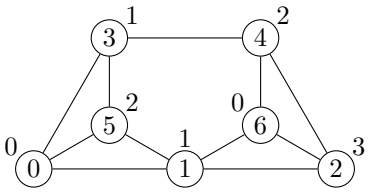
1 Préliminaires

Question 1 La numérotation suivante est une 3-coloration :

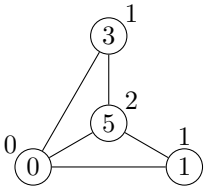


Il n'existe pas de 2-coloration, car il existe trois sommets qui sont deux à deux adjacents, c'est-à-dire qu'il est nécessaire d'utiliser au moins 3 couleurs pour ces trois sommets. On en déduit que $\chi(G_0) = 3$.

Question 2 La numérotation suivante est une 4-coloration



Les sommets 0 et 5 sont adjacents entre eux et tous deux adjacents aux sommets 1 et 3 donc, quitte à renuméroter, si G_1 possède une 3-coloration, on doit colorer ces 4 sommets de la manière suivante :



Dès lors, les sommets 2, 4 et 6 sont adjacents à des sommets dont la couleur est 1. Comme ces trois sommets sont deux à deux adjacents, ils ne peuvent pas recevoir la même couleur, ni la couleur 1. On en déduit qu'il est nécessaire d'utiliser une quatrième couleur pour colorer le graphe G_1.

Question 3 On distingue :

– si n est pair, alors $\chi(\mathcal{C}_n) = 2$. En effet, si on indexe les sommets dans l'ordre du cycle $0, 1, \ldots, n-1$, alors deux sommets de même parité ne seront jamais adjacents, donc il est possible d'attribuer la couleur 0 à tous les sommets d'indice pairs et la couleur 1 à tous les sommets d'indice impair. Le graphe est donc 2-colorable. Le graphe n'est pas 1-colorable, car $n \geqslant 3$ (par définition de $\mathcal{C}_n$), donc il existe au moins deux sommets adjacents.

– Si n est impair, alors $\chi(\mathcal{C}_n) = 3$. En effet, si on indexe les sommets dans l'ordre du cycle $0, 1, \ldots, n-1$, alors on peut attribuer la couleur 0 aux sommets pairs de $0, \ldots, n-3$, la couleur 1 aux sommets impairs de $1, \ldots, n-2$ et la couleur 2 au sommet $n-1$. Le graphe est donc 3-colorable. Le graphe n'est pas 2-colorable, car si on suppose que c'est le cas, en attribuant sans perte de généralité la couleur 0 au sommet 0, il est nécessaire d'attribuer la couleur 1 au sommet 1, puis la couleur 0 au sommet 2, et on montre par récurrence que les sommets $0, \ldots, n-2$ se voit attribuer la même couleur que décrit ci-dessus. Dès lors, le sommet $n-1$ est adjacent au sommet 0 (coloré par 0) et au sommet $n-2$ (coloré par 1). Il est donc nécessaire d'utiliser une troisième couleur.

Question 4 Le nombre chromatique d'un graphe complet à n sommets est n, car il faut attribuer une couleur différente à chaque sommet du graphe (car ils sont deux à deux adjacents).

Question 5 On a $\chi(G) \geqslant \omega(G)$. En effet, dans un graphe G, s'il existe une clique de taille k, alors il sera nécessaire d'utiliser k couleurs différentes pour colorer ces k sommets qui sont deux à deux adjacents. Le nombre chromatique de G sera donc au moins égal à k.

Cette inégalité est une égalité pour les graphes complets à n sommets d'après la question 4.

D'après la question 3, c'est une inégalité stricte pour $\mathcal{C}_n$ avec n impair (car $\omega(\mathcal{C}_n) = 2$). Dans le cas n pair, on peut rajouter un sommet adjacent à un seul sommet de $\mathcal{C}_{n-1}$ pour avoir une inégalité stricte.

Question 6 Soit c une k-coloration de G. Alors pour tout $i \in [\![0, k-1]\!]$, $C(i) = \{s \in S \mid c(s) = i\}$ forme un stable, par définition d'une coloration. On en déduit que $|C(i)| \leqslant \alpha(G)$. Comme l'ensemble des $C(i)$ forme une partition de S, on en déduit que $n = \sum_{i=0}^{k-1} |C(i)| \leqslant k\alpha(G)$, soit $\frac{n}{\alpha(G)} \leqslant k$. Ceci étant vrai pour toute k-coloration, c'est en particulier vrai pour une $\chi(G)$-coloration, d'où la minoration voulue.

Les mêmes exemples de la question précédente conviennent pour le cas d'égalité et d'inégalité stricte (pour les graphes de l'inégalité stricte, on a $\alpha(G) = \left\lfloor \frac{n}{2} \right\rfloor$ et $\chi(G) = 3$).

2 Calcul du nombre chromatique

Question 7 On libère la queue de la liste si elle est non vide, puis on libère le pointeur.

```
void liberer_liste(liste* lst){
    if (lst != NULL){
        liberer_liste(lst->suivant);
        free(lst);
    }
}
```

Question 8 On commence par une allocation mémoire, puis on attribue les bonnes valeurs aux deux champs avant de renvoyer la nouvelle liste.

```
liste* cons(int x, liste* lst){
    liste* nouv_lst = malloc(sizeof(liste));
    nouv_lst->val = x;
    nouv_lst->suivant = lst;
    return nouv_lst;
}
```

Question 9 Pour chaque sommet s du graphe, on parcourt sa liste d'adjacence, et on vérifie que chaque voisin a une couleur différente du sommet s :

```
bool coloration(liste** G, int* c, int n){
    for (int s=0; s<n; s++){
        liste* lst = G[s];
        while (lst != NULL){
            if (c[s] == c[lst->val]){
                return false;
            }
            lst = lst->suivant;
        }
    }
    return true;
}
```

Question 10 On parcourt le tableau `c` en partant de la fin en mettant à 0 toutes les valeurs qui contiennent $k-1$. Si on arrive au début du tableau, c'est qu'on était arrivés à la dernière numérotation, et on renvoie `true`. Sinon on incrémente la première valeur trouvée qui ne valait pas $k-1$ et on renvoie `false`.

```
bool incrementer(int* c, int n, int k){
    for (int i=n-1; i>=0; i--){
        if (c[i] == k-1){
            c[i] = 0;
        } else {
            c[i]++;
            return false;
        }
    }
    return true;
}
```

Question 11 On montre que pour k^n incrémentations, la complexité totale est $\mathcal{O}(k^n)$ (ce qui montrera que la complexité amortie est $\mathcal{O}(1)$). Pour cela, on remarque que la fonction `incrementer` sortira au i-ème passage dans la boucle si les $i-1$ dernières cases de `c` valent $k-1$ et la i-ème dernière ne vaut pas $k-1$. Pour $i \in [\![0, n]\!]$, notons $E(i)$ le nombre de numérotations qui terminent par i valeurs $k-1$ ou plus. On remarque que le nombre de numérotations qui terminent par exactement i valeurs $k-1$ vaut $E(i) - E(i+1)$. En supposant $E(n+1) = 0$, Le nombre total de passages est alors de l'ordre de :

$$\sum_{i=0}^{n} i(E(i) - E(i+1)) = \sum_{i=1}^{n} E(i) = \sum_{i=1}^{n} k^{n-i} = k^n - 1$$

Question 12 On commence par créer une numérotation contenant des 0. Tant qu'on n'a pas trouvé une coloration compatible, on incrémente la coloration. On pense à libérer la mémoire avant de renvoyer le booléen.

```
bool k_colorable(liste** G, int n, int k){
    int* c = malloc(n * sizeof(*c));
    for (int i=0; i<n; i++) {c[i] = 0;}
    bool fini = false;
    while (!fini){
        if (coloration(G, c, n)){
            free(c);
            return true;
        }
        fini = incrementer(c, n, k);
    }
    free(c);
    return false;
}
```

Question 13 On teste les valeurs de k par ordre croissant. On remarque que $\chi(G) \leqslant n$, donc on peut renvoyer n sans tester si G est n-colorable.

```
int chi(liste** G, int n){
    for (int k=1; k<n; k++){
        if (k_colorable(G, n, k)){
            return k;
        }
    }
    return n;
}
```

Question 14 On distingue :

- pour la complexité temporelle, la fonction `coloration` parcourt chaque arête du graphe une seule fois, et chaque sommet au moins une fois. La complexité est donc en $\mathcal{O}(|S| + |A|)$. Dans le pire des cas, le graphe est complet et $|A| = |S|^2$. La complexité totale est donc en $\mathcal{O}(n^2)$. La fonction `incrementer` a une complexité amortie en $\mathcal{O}(1)$. Dans le pire cas, la fonction `k_colorable` parcourt toutes les numérotations possibles (il y en a k^n) et applique les deux fonctions précédentes. Elle est donc en $\mathcal{O}(k^n(n^2+1)) = \mathcal{O}(n^2k^n)$. La fonction `chi` applique `k_colorable` pour chaque valeur de k entre 1 et $n-1$. On trouve alors (par un petit calcul) une complexité dans le pire cas en $\mathcal{O}(n^2n^n) = \mathcal{O}(n^{n+2})$.
- pour la complexité spatiale, on constate que la seule structure de données créée non constante en espace est la numérotation dans la fonction `k_colorable`. Comme on modifie cette liste à chaque incrémentation, aucun espace supplémentaire ne sera requis. La complexité spatiale est donc en $\mathcal{O}(n)$.

Pour $n = 20$, on réalisera de l'ordre de 20^{22} opérations. Or $20^{22} = 2^{22} \times 10^{22} \simeq 4 \times 10^{28}$. Sur un ordinateur qui réalise 10^9 opérations par seconde (1 GHz), il faudrait de l'ordre de 4×10^{19} secondes, soit 1000 milliards d'années, ce qui est évidemment inutilisable. Bien évidemment, c'est encore moins réalisable pour $n = 50$.

3 Problème de décision et NP-complétude

Question 15 À partir d'une instance G du problème k-coloration, on peut créer une instance $f(G) = (G, k)$ qui est une entrée du problème Coloration. On a bien $G \in$ k-coloration $\Leftrightarrow f(G) \in$ Coloration et f est bien constructible en temps polynomial.

Question 16 À partir d'une instance $G = (S, A)$ du problème h-coloration, on construit une instance $f(G) = G'$ de la manière suivante : on rajoute $k - h$ nouveaux sommets deux à deux adjacents, et adjacents à tous les sommets de S. Dès lors, montrons que $G \in$ h-coloration $\Leftrightarrow f(G) \in$ k-coloration :

– si $G \in$ h-coloration, alors il existe une h-coloration de G. En attribuant à chacun des $k-h$ nouveaux sommets une nouvelle couleur, on obtient bien une k-coloration de G', donc $f(G) \in$ k-coloration ;

– réciproquement, si $f(G) \in$ k-coloration, alors il existe une k-coloration de G'. Dans cette coloration, nécessairement, les $k-h$ nouveaux sommets ont chacun une couleur différente qui n'apparait pas parmi les sommets de S. Cela signifie que ces sommets sont colorés en utilisant $k-(k-h)=h$ couleurs, donc $G \in$ h-coloration.

f étant bien constructible en temps polynomial, on en déduit le résultat.

Question 17 On peut résoudre ce problème par un parcours de graphe. On décrit l'algorithme pour un graphe connexe. Dans le cas général, on appliquera l'algorithme à chaque composante connexe.

– on attribue une couleur $c(s_0)=0$ à un premier sommet choisi arbitrairement ;

– pour chaque sommet s à traiter, on parcourt chacun de ses voisins t :

  * si t n'a pas encore de couleur, on pose $c(t)=1-c(s)$ et on relance un traitement depuis t ;
  * si t a déjà une couleur et $c(t)=1-c(s)$, on ne fait rien ;
  * si t a déjà une couleur et $c(s)=c(t)$, on conclut que le graphe n'est pas 2-colorable et on renvoie Faux.

Un tel algorithme a une complexité linéaire en $|S|+|A|$, donc 2-coloration $\in$ P.

Question 18 On commence par écrire une fonction de parcours en profondeur qui fait les vérifications listées plus haut. Cette fonction renverra un booléen indiquant si une contradiction a été trouvée ou non.

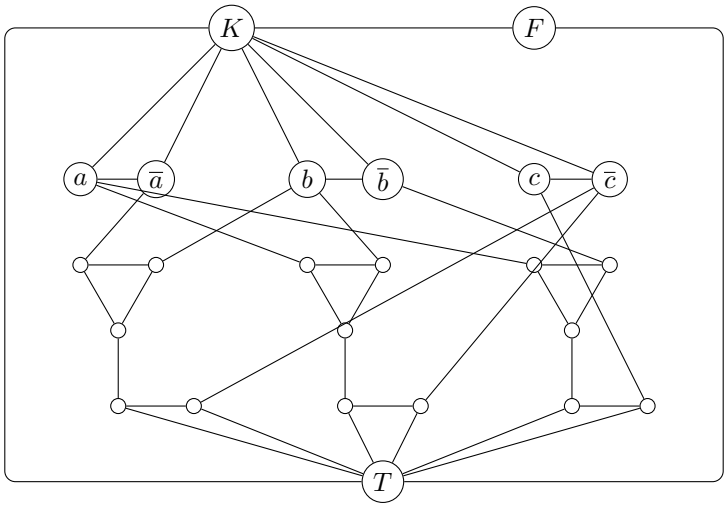
```
bool dfs(liste** G, int* c, int s){
    liste* lst = G[s];
    while (lst != NULL){
        int t = lst->val;
        if (c[t] == c[s]) {return false;}
        if (c[t] == -1){
            c[t] = 1 - c[s];
            if (!dfs(G, c, t)) {return false;}
        }
    }
    return true;
}
```

Dès lors, on peut créer un tableau c correspondant à la 2-coloration en cours de construction et lancer un parcours depuis chaque composante connexe.

```
bool deux_colorable(liste** G, int n){
    int* c = malloc(n * sizeof(*c));
    for (int s=0; s<n; s++) {c[s] = -1;}
    for (int s=0; s<n; s++){
        if (c[s] == -1){
            c[s] = 0;
            if (!dfs(G, c, s)) {
                free(c);
                return false;
            }
        }
    }
    free(c);
    return true;
}
```

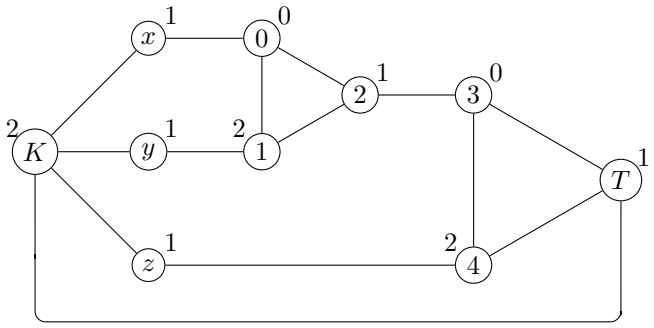
Question 19 La donnée d'une k-coloration est un certificat de taille polynomiale prouvant qu'un graphe est k-colorable. Vérifier que c'est bien une k-coloration consiste à déterminer la valeur maximale d'une couleur et que c'est bien une coloration. Cette vérification se fait bien en temps polynomial.

Question 20 On obtient le graphe suivant :



Question 21 Le nouveau graphe possède $3 + 2n + 5m$ sommets, où n est le nombre de variables et m le nombre de clauses. Il possède par ailleurs $3 + 3n + 10m$ arêtes. Sachant que le nombre de variables qui apparaissent dans φ est au plus $3m$, on en déduit que la taille du graphe G_φ (en nombre de sommets est d'arêtes) est bien polynomial en m, le nombre de clauses.

Question 22 Le graphe est 3-colorable comme en atteste la coloration suivante (on a numéroté les 5 sommets intermédiaires pour la preuve suivante) :



Dès lors, supposons que dans une 3-coloration, chacun des sommets x, y et z ont une couleur différente de T. Sachant que K est adjacent à ces 4 sommets, cela signifie (sans perte de généralité) que T a la couleur 1, K a la couleur 2, et x, y et z ont la couleur 0. On en déduit que nécessairement les sommets 0 et 1 se voient attribuer les couleurs 1 et 2, puis que le sommet 2 est également coloré en 0. Dès lors les sommets 3 et 4 sont tous deux adjacents à un sommet coloré par 0 et un sommet coloré par 1. Comme ils sont eux-mêmes adjacents, il faudrait une quatrième couleur pour terminer la coloration. On conclut par l'absurde que dans une 3-coloration, l'un des sommets x, y ou z possède la même couleur que T.

Question 23 Supposons que φ est satisfiable et soit μ une valuation qui satisfait φ. Construisons par étapes une 3-coloration c de G_φ :

- on pose $c(F) = 0$, $c(T) = 1$ et $c(K) = 2$;
- pour chaque variable v, on pose $c(v) = 0$ si $\mu(v) = \bot$ et $c(v) = 1$ si $\mu(v) = \top$;
- pour chaque variable v, on pose $c(\overline{v}) = 1 - c(v)$.

À cette étape, deux sommets adjacents ont bien une couleur différente. Montrons qu'on peut alors compléter la coloration pour chaque sous-graphe correspondant à une clause $(x \vee y \vee z)$. En réutilisant l'indexation de la

réponse précédente, on pose $c(1) = c(4) = 2$, $c(0) = c(3) = 1 - c(x)$ et $c(2) = c(x)$. À nouveau, deux sommets adjacents ont une couleur différentes.

La numérotation c est bien une 3-coloration de G_φ.

Question 24 Supposons que G_φ est 3-colorable. Sans perte de généralité, il existe une 3-coloration c telle que $c(F) = 0$, $C(T) = 1$ et $c(K) = 2$. Dès lors, pour chaque littéral ℓ, on a $c(\ell) \in \{0, 1\}$ (car les littéraux sont adjacents à K).

Définissons alors la valuation μ telle que pour chaque variable $v \in V, \mu(v) = \top \Leftrightarrow c(v) = 1$. Sachant que chaque variable v est adjacente à $\overline{v}$, on en déduit que pour tout littéral ℓ, $\mu(\ell) = \top \Leftrightarrow c(\ell) = 1$.

Par la question précédente, on en déduit également que dans chaque clause de φ, il existe un littéral ℓ qui a la couleur 1, donc tel que $\mu(\ell) = \top$. On en déduit que μ satisfait φ.

Question 25 Les questions précédentes montrent que $\varphi \in$ 3-SAT $\Leftrightarrow G_\varphi \in$ 3-coloration. On a montré que G_φ est constructible en temps polynomial. On en déduit que 3-SAT est réductible en temps polynomial à 3-coloration. Sachant que 3-SAT est NP-complet (donc NP-difficile), on en déduit que 3-coloration est NP-difficile. Sachant que 3-coloration $\in$ NP, ce problème est également NP-complet.

4 Algorithmes gloutons

Question 26 Pour garantir une complexité linéaire dans tous les cas, il suffit de remarquer que si `lst` est de taille n, alors le plus petit absent de `lst` vaut au plus n (si les éléments de `lst` sont exactement $0, 1, \ldots, n-1$). L'idée est alors d'utiliser un tableau `present` de taille n tel que `present[i]` vaut `true` si `i` apparait dans `lst` et `false` sinon. Il suffit alors de trouver le plus petit `i` tel que `present[i]` vaut `false` (et de renvoyer n s'il n'en existe pas).

La fonction suivante calcule la taille d'une liste :

```
int taille(liste* lst){
    if (lst == NULL){
        return 0;
    } else {
        return 1 + taille (lst->suivant);
    }
}
```

On obtient alors le code suivant :

```
int plus_petit_absent(liste* lst){
    int n = taille(lst);
    bool* present = malloc(n * sizeof(*present));
    for (int i=0; i<n; i++) {present[i] = false;}
    liste* tmp = lst;
    while (tmp != NULL){
        present[lst->val] = true;
        tmp = tmp->suivant;
    }
    for (int i=0; i<n; i++){
        if (!present[i]) {
            free(present);
            return i;
        }
    }
    free(present);
    return n;
}
```

Question 27 On parcourt la liste des voisins et on construit la liste des couleurs en parallèle.

```
liste* couleurs_voisins(liste** G, int* c, int s){
    liste* lst = NULL;
    liste* voisins = G[s];
    while (voisins != NULL){
        lst = cons(c[voisins->val], lst);
        voisins = voisins->suivant;
    }
    return lst;
}
```

Question 28 On obtient la coloration : $[0, 1, 0, 1, 2, 2, 3]$.

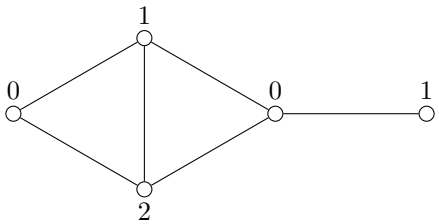
Question 29 On se contente de créer une coloration contenant initialement des -1, puis de suivre l'algorithme qui est décrit. Il est à noter que la présence de nombres négatifs parmi les couleurs des voisins n'est pas problématique pour l'application de la fonction `plus_petit_absent`.

```
int* colo_glouton(liste** G, int n){
    int* c = malloc(n * sizeof(*c));
    for (int s=0; s<n; s++) {c[s] = -1;}
    for (int s=0; s<n; s++){
        liste* lst = couleurs_voisins(G, c, s);
        c[s] = plus_petit_absent(lst);
        free(lst);
    }
    return c;
}
```

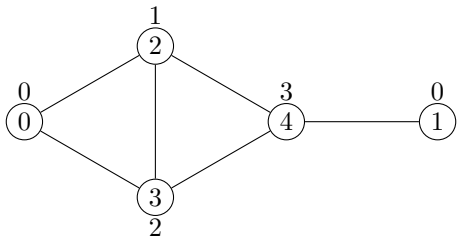
Question 30 On remarque que pour colorer un sommet s, on crée la liste des couleurs de ses voisins, puis on cherche son plus petit absent. Ces deux opérations ont une complexité linéaire en le degré de s. La complexité totale est donc en $\mathcal{O}\left(\sum_{s\in S} \deg(s)\right) = \mathcal{O}(|S| + |A|)$ pour un graphe $G = (S, A)$.

Question 31 On remarque que la couleur attribuée à chaque sommet s vaut au plus son degré (car le plus petit absent d'une liste de taille n vaut au plus n). On en déduit que $\chi(G) \leqslant \Delta(G) + 1$ (le nombre de couleurs vaut 1 de plus que la couleur maximale). Un graphe complet d'ordre n convient pour le cas d'égalité. Un graphe en « étoile » (un sommet adjacent à tous les autres) convient pour le cas d'inégalité stricte.

Question 32 On commence par remarquer que $\chi(G_2) = 3$, car il existe une clique de taille 3, et que la coloration suivante est une 3-coloration :



Or, si on indexe les sommets de la manière suivante, l'algorithme glouton renverra la coloration suivante, qui est une 4-coloration :



Question 33 Soit G un graphe et c une coloration optimale, c'est-à-dire utilisant les $\chi(G) = k$ couleurs $\{0, 1, \ldots, k-1\}$. Supposons qu'il existe un sommet s et une valeur $0 \leqslant x < c(s)$ tels qu'aucun voisin de s n'a la couleur x. En posant c' la numérotation telle que $c'(t) = c(t)$ si $t \neq s$ et $c'(s) = x$, alors c' reste une coloration, et elle utilise au plus k couleurs. Sachant que c était optimale, on en déduit que c' est également optimale.

En répétant cette opération tant qu'il existe de tels sommets (ce processus termine nécessairement, car la somme des couleurs diminue strictement, tout en restant positive), on obtient une nouvelle coloration optimale $\tilde{c}$. Construisons l'indexation cherchée à partir de $\tilde{c}$: indexons les sommets par ordre croissant de couleurs, les cas d'égalité étant tranchés arbitrairement. Montrons que l'application de l'algorithme glouton à un tel graphe renvoie une coloration optimale c_g :

- supposons qu'il existe p sommets de couleur 0 dans $\tilde{c}$. Ces sommets ne sont donc pas adjacents. En appliquant l'algorithme glouton, les sommets $0, 1, \ldots, p-1$ seront les premiers à être colorés, et comme aucun de leur voisin n'est coloré, on posera $c_g(s) = 0 = \tilde{c}(s)$, pour $s \in \{0, \ldots, p-1\}$;
- soit $0 \leqslant h < k$. Supposons que seuls les sommets dont la couleur est $\leqslant h$ dans $\tilde{c}$ ont été colorés par l'algorithme glouton, et que $\tilde{c}(s) = c_g(s)$ pour chacun de ces sommets. Par construction de l'indexation, les sommets qui vont ensuite être colorés par l'algorithme sont ceux de couleur $h+1$ dans $\tilde{c}$. Par construction de $\tilde{c}$, on sait que chaque sommet de couleur $h+1$ a au moins un voisin de couleur 0, 1, …, h dans $\tilde{c}$, et donc dans c_g. On en déduit que ces sommets seront colorés par $h+1$ par l'algorithme glouton.

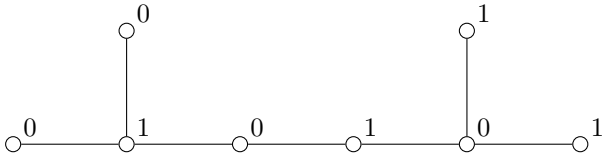
On conclut par récurrence que la coloration c_g renvoyée par l'algorithme glouton est $\tilde{c}$, qui est optimale.

Question 34 L'algorithme de Welsh-Powell commencera par colorer en premier les 3 sommets de degré 3, dans un ordre arbitraire. Ces sommets étant adjacents, ils se verront attribuer les couleurs $0, 1, 2$. Ensuite, le sommet de degré 2 recevra la même couleur que le sommet de degré 3 qui ne lui est pas adjacent. Enfin, le sommet de degré 1 recevra la plus petite des deux couleurs des sommets de degré 3 qui ne lui sont pas adjacents. Dans tous les cas, seules 3 couleurs seront utilisées, ce qui garantit l'optimalité.

Question 35 Avec les notations de l'énoncé, l'énumération $(x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{n-1})$ renvoie un 2-coloriage (le plus petit absent pour colorier les x_i est toujours 0, le plus petit absent pour colorier les y_i est toujours 1), et l'énumération $(x_0, y_0, x_1, y_1, \ldots, x_{n-1}, y_{n-1})$ renvoie un n-coloriage (on colorie toujours x_i et y_i avec la couleur i, car x_i est relié à tous les y_j tels que $j < i$, et symétriquement pour les y_i).

Question 36 L'algorithme de Welsh-Powell commencera par colorer par 0 les deux sommets de degré 3. Ensuite, il faudra colorer les deux sommets de degré 2. Comme ils sont adjacents à un sommet de degré 3, ils ne pourront pas recevoir la couleur 0. Comme ils sont adjacents entre eux, il faudra au moins deux couleurs supplémentaires pour les colorer (l'un avec 1, l'autre avec 2). Finalement, les sommets de degré 1 seront colorés soit par 0, soit par 1. La coloration renvoyée sera donc toujours une 3-coloration.

Or, il s'avère que le graphe est 2-colorable :



On peut en fait montrer que tout graphe sans cycle est 2-colorable.

⋆ ⋆ ⋆