TD #16 — Graphes #1 (avec solutions)

Exercice 1. *Graphes (adapté d'un roman de Georges Perec)*

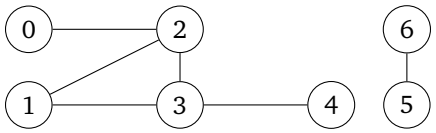
Ces échanges, qui sont suscités aussi bien par des occasions propices de vente ou d'achat (il s'agit alors de faire de la place) que par des inspirations subites, des lubies, des caprices ou des dégoûts, ne se font pas au hasard, et n'épuisent pas les douze possibilités de permutations qui pourraient se faire entre ces quatre lieux et que la figure 1 *met bien en évidence ; ils obéissent strictement au schéma de la* figure 2 *: quand Madame Marcia achète quelque chose, elle le met chez elle, dans son appartement, ou dans sa cave ; de là, ledit objet peut passer dans l'arrière-boutique, et de l'arrière-boutique dans le magasin ; du magasin enfin il peut revenir — ou parvenir s'il venait de la cave — dans l'appartement. Ce qui est exclu, c'est qu'un objet revienne dans la cave, ou arrive au magasin sans être passé par l'arrière-boutique, ou repasse du magasin dans l'arrière-boutique, ou de l'arrière-boutique dans l'appartement, ou enfin passe directement de la cave à l'appartement.*

1. Dessinez les graphes des figures 1 & 2.
2. Comment appelle-t'on les graphes semblables à celui de la figure 1.

Le graphe de la figure 1 est un graphe *complet* (ici orienté).

Exercice 2. *Graphes (adapté d'un exercice de Marie Durand)*

On donne le graphe non orienté suivant :



1. Donnez le degré du sommet 2.

Le sommet 2 a un degré de 3 : il possède trois arêtes le reliant aux sommets 0, 1 et 3.

2. Ce graphe est-il acyclique ? Si non, donnez un exemple de cycle.

Ce graphe n'est pas acyclique : il possède un cycle $2 \to 1 \to 3 \to 2$.

3. Ce graphe est-il connexe ? Si non, donnez les ensembles de sommets formant ses composantes connexes.

Ce graphe n'est pas connexe : il est par exemple impossible d'atteindre le sommet 6 à partir du sommet 0. Il possède deux composantes connexes : $\{0, 1, 2, 3, 4\}$ et $\{5, 6\}$.

4. Donnez deux chemins allant du sommet 0 au sommet 4. Combien y a-t'il de chemins possibles entre ces sommets ? Combien de ces chemins sont élémentaires ?

Il existe exactement deux chemins élémentaires (qui ne passent pas plusieurs fois par le même sommet) : $0 \to 2 \to 3 \to 4$ et $0 \to 2 \to 1 \to 3 \to 4$. Il existe en revanche une infinité de chemins quelconques puisque l'on peut répéter le cycle $2 \to 1 \to 3 \to 2$ un nombre arbitraire de fois.

5. Donnez une représentation en OCaml de ce graphe par un tableau de listes d'adjacence, sous forme d'un `int list array`.

On peut prendre la convention que chaque sommet est représenté par l'`int` lui correspondant et que la case `g.(i)` du tableau représentant le graphe contient la liste des sommets adjacents à `i`. On a alors par exemple :

```
val g : int list array = [|[2]; [2; 3]; [1; 0; 3]; [1; 2; 4]; [3]; [6]; [5]|]
```

Cette représentation n'est pas unique, même à convention unique pour représenter les sommets : l'ordre d'énumération des sommets dans les listes d'adjacence n'est pas fixé.

6. De quelle autre façon pourrait-on représenter ce graphe informatiquement ?

   Par exemple avec une matrice d'adjacence.

7. Proposez un *coloriage* de ce graphe qui associe une couleur à chaque sommet de façon à ce que deux sommets adjacents n'aient pas la même couleur. Votre coloriage devra utiliser le nombre minimum de couleurs distinctes possible pour ce graphe (et vous devez montrer que c'est le cas).

   Le sous-graphe induit par les sommets 1, 2, 3 est le graphe cyclique C_3. Comme tous les graphes cycliques de cardinalité impaire, celui-ci peut se colorier avec 3 couleurs **et pas moins**. Au moins trois couleurs sont donc aussi nécessaires pour colorier le graphe tout entier. On constate alors que colorier 0, 1, 4, 6 en rouge, 2 en vert et 3 et 5 en bleu est un coloriage valide qui atteint ce minorant, et est donc optimal.

Exercice 3. *Parcours complet*

On considère des graphes représentés par tableau de liste d'adjacence en OCaml, *via* le `type graph = int list array`.

1. Écrivez une fonction OCaml :

   `wfs : graph -> int -> int option array`

   telle que `wfs g v` effectue un parcours de graphe d'origine `v` et s'évalue en `int option array` dont la case d'indice `i` contient `None` si `i` n'est pas accessible depuis `v`, et sinon `Some t` avec `t` son ordre de visite par le parcours.
   Votre fonction devra avoir un coût linéaire en la taille de ses arguments.

   Il y a plein de possibilités… On propose un parcours en profondeur implémenté récursivement :

```
let wfs g v =
    let vis = Array.make (Array.length g) None in
    let clock = ref 0 in
    let rec _dfs
        = function
        | [] -> ()
        | v'::vs -> if vis.(v') = None then begin
                        incr clock ;
                        vis.(v') <- Some !clock ;
                        _dfs g.(v')
                    end ;
                    _dfs vs
    in
    _dfs [v] ; vis
```

2. Donnez une caractérisation des graphes pour lesquels un résultat d'appel à `wfs` peut contenir des cases à `None`. (Soyez précis, notamment dans les distinctions que vous pouvez faire en fonction de si les graphes sont orientés ou non. En particulier, n'oubliez pas de considérer les trois notions de connexité distinctes pour les graphes orientés !)

   Dans le cas d'un graphe non orienté la présence de `None` équivaut au fait que le graphe ne soit pas connexe, peu importe le point de départ du parcours.
   Dans le cas d'un graphe orienté, s'il est fortement connexe il n'y aura jamais de `None` peu importe le point de départ ; s'il n'est pas faiblement connexe il y aura des `None` peu importe le point de départ, de même que s'il est faiblement connexe mais non connexe (car dans ce cas il existe une paire de sommets qui ne sont reliés par aucun chemin ; *cf.* par exemple le graphe `> [|[1]; []; [1]|]`).
   S'il est connexe mais non fortement connexe, certains points de départ donneront nécessairement un résultat avec des `None` (sinon le graphe est fortement connexe), et il existe au moins un point de départ produisant un résultat sans `None`. Ce dernier peut être calculé de la façon suivante : on effectue un parcours depuis un sommet v_1, et l'on termine si tous les sommets ont été visités ; sinon il existe un sommet v_2 non visité dans ce parcours (donc un sommet v_2 tel qu'il n'existe pas de chemin entre v_1 et v_2), et par l'hypothèse que le graphe est connexe il existe un chemin $v_2 \rightsquigarrow v_1$; on recommence alors récursivement le même procédé avec v_2. La terminaison est garantie en prenant comme variant la taille de l'ensemble des sommets accessibles depuis le point de départ, qui est majorée par le nombre de sommets et strictement croissante.

3. Modifiez votre fonction `wfs` en une fonction OCaml :

`wfsf : graph -> int * int array`
qui visite *tous* les sommets du graphe (désormais non orienté, mais peu importe sa connexité) et s'évalue en une paire donnant le nombre de composantes connexes et un tableau contenant en case `i` un entier indiquant sa composante connexe (on numérotera par exemple les $\#C$ composantes connexes par des entiers 0, 1, …, $\#C-1$). Cette fonction devra être de **coût linéaire** en la taille de son argument.

On ne peut pas se permettre de passer trop de temps à chercher un éventuel sommet encore non visité si l'on veut bien avoir un coût linéaire. Une solution simple est de partager le tableau de visite pour tous les parcours et de maintenir un pointeur vers son plus petit indice où l'on peut trouver une case encore non visitée (ce pointeur sera donc incrémenté exactement $\#S$ fois pour la *totalité* de l'exécution).

```
let wfsf g =
    let n = Array.length g in
    let vis = Array.make n (-1) in
    let cnum = ref (-1) in
    let rec _dfs
        = function
        | [] -> ()
        | v'::vs -> if vis.(v') = (-1) then begin
                        vis.(v') <- !cnum ;
                        _dfs g.(v')
                    end ;
                    _dfs vs
    in
    let p = ref 0 in
    while !p < n do
        if vis.(!p) = (-1) then
        (incr cnum ; _dfs [!p]) else
        incr p
    done ;
    !cnum + 1, vis
```

Exercice 4. *Bicoloriage*

Pour le besoin de cet exercice, on dit d'un graphe non orienté $G=(S,A)$ qu'il est *biparti* si $A\neq\emptyset$ et S est l'union disjointe de L, R non vides tels que $(u,v)\in A \Rightarrow ((u\in L\wedge v\in R)\vee(u\in R\wedge v\in L))$. (Autrement dit, toute arête de G connecte nécessairement un sommet de L à un sommet de R.) On définit également le *nombre chromatique* d'un graphe G (noté $\chi(G)$) comme le nombre minimal n de couleurs pour lequel G est n-coloriable (c'est à dire que n couleurs sont suffisantes pour que chaque sommet soit colorié d'une couleur différente de celle de tous ses voisins).

1. Montrez l'équivalence (G est biparti) $\iff \chi(G)=2$.

   $\Rightarrow$: Soient L, R les sous-ensembles de S donnés par le fait que G est biparti, on peut colorier chaque sommet de L en une couleur (disons noir) et chaque sommet de R en une autre (disons rouge). Par le fait que G est biparti, aucun sommet de L n'est adjacent à un sommet de L et de même pour R, et ce coloriage est valide. De plus par le fait que $A\neq\emptyset$, G n'est pas 1-coloriable, et donc $\chi(G)=2$.
   $\Leftarrow$: Si $\chi(G)=2$ alors A et S sont tous-deux non vides. Soient $\mathcal{B}$, $\mathcal{R}$ les ensembles de sommets de S coloriés respectivement en noir et en rouge par un 2-coloriage valide, ils sont non vides et forment une partition de S. De plus, toute arête $(u,v)\in A$ connecte nécessairement un sommet de $\mathcal{B}$ à un sommet de $\mathcal{R}$ puisque sinon le coloriage ne serait pas valide. On peut donc prendre $L=\mathcal{B}$ et $R=\mathcal{R}$ comme bipartition de G.

2. Déduisez de cela qu'un arbre de taille >1 est biparti.

   Un arbre non trivial est 2-coloriable : par exemple, on distingue un sommet arbitraire comme racine que l'on colorie en noir, puis l'on colorie ses enfants (non vides) de profondeur impaire (resp. paire) en rouge (resp. en noir). Il est donc biparti par la question précédente.

3. Montrez que $\chi(C_n)=2$ ssi. $n\geq 3$ est pair.

   On considère des représentations canoniques de C_n, avec $S=[\![n]\!]$ et $A=\{(0,1),(1,2),\ldots,(n-1,0)\}$.
   Par $A\neq\emptyset$, C_n n'est jamais 1-coloriable.

Si n est pair, les sommets voisins d'un sommet pair sont impairs, et vice-versa : c'est évident pour tout sommet autre que 0 et $n-1$, et c'est aussi vrai dans ce dernier cas puisque $n-1$ est impair. Un 2-coloriage coloriant les sommets pairs en noir et les sommets impairs en rouge est donc valide, et minimal par ce qui précède.

Si n est impair, on va montrer (par un parcours de l'arbre des possibilités) que tout 2-coloriage de G est invalide. On commence par colorier 0 en noir sans perte de généralité. On a deux choix potentiels pour le sommet 1, mais tout coloriage où il serait colorié en noir serait invalide ; il suffit donc de considérer les coloriages où il est colorié en rouge. De même pour le sommet 2 : tout coloriage où il serait colorié en rouge serait invalide, et il suffit de considérer les coloriages où il serait colorié en noir. On procède de même pour les sommets restants jusqu'à $n-2$ inclus : les sommets pairs sont nécessairement coloriés en noir et les sommets impairs en rouge. C'est notamment le cas de $n-2$, ce qui fait que $n-1$ ne peut pas être colorié en rouge dans un coloriage valide. Puisqu'il est adjacent à 0, il ne peut pas non plus être colorié en noir, et il s'ensuit que G n'est pas 2-coloriable (il est par contre 3-coloriable, puisque le coloriage partiel précédent peut être trivialement complété en un coloriage valide en coloriant $n-1$ en bleu (disons)).

Remarque : si G' est un sous-graphe de G, il est immédiat que $\chi(G) \geq \chi(G')$ (un coloriage valide de G restant valide pour ses sous-graphes puisque ceux-ci sont obtenus en *retirant* des sommets et des arêtes (et donc des contraintes)). On a donc notamment que tout graphe contenant des cycles de longueur impaire n'est pas 2-coloriable.

4. Décrivez un algorithme qui décide si un graphe est 2-coloriable.

On réalise un parcours (complet) du graphe comme à l'exercice précédent, en coloriant chaque point de départ du parcours en noir. Tout sommet visité est alors colorié de l'autre couleur que celle de son parent dans l'arborescence du parcours, et l'on détecte l'invalidité du coloriage en considérant les éventuelles couleurs déjà connues de ses voisins.

5. Proposez-en une implémentation OCaml pour un graphe représenté comme à l'exercice précédent.

On propose :

```
let bicolor g =
    let n = Array.length g in
    let vis = Array.make n None in
    let rec _dfs
        = function
        | [] -> true
        | (v,c)::vs -> if vis.(v) = None then
                           let c' = 1 - c in
                           let () = vis.(v) <- Some c' in
                           (* two-pass meh *)
                           if List.exists (fun i -> vis.(i) = Some c') g.(v) then
                           false else
                           _dfs (List.map (fun x -> (x, c')) g.(v))
                       else _dfs vs
    in
    let p = ref 0 in
    let r = ref true in
    while !p < n && !r do
        if vis.(!p) = None then
        r := _dfs [!p, 1] else
        incr p
    done ;
    !r
```