TP5 - Graphes (1)

Le langage de programmation est OCaml.

Soit $G = (V, E)$ un *graphe non-orienté* d'ensemble de sommets V et d'ensemble d'arêtes E. Il est dit *étiqueté* lorsqu'on dispose d'une *fonction d'étiquetage* de l'ensemble de ses sommets vers un ensemble non vide appelé *ensemble des étiquettes*. Les étiquettes peuvent être des entiers, des listes ou des chaînes de caractères. Une fonction d'étiquetage C est un *coloriage* des sommets de G lorsque deux sommets voisins ont toujours deux étiquettes distinctes, alors appelées *couleurs*, c'est à dire lorsque C vérifie la condition :

$$\forall s, t \in V,\ (s, t) \in E \Rightarrow C(s) \neq C(t)$$

Un graphe est dit *k-coloriable* s'il admet un coloriage avec au plus k couleurs. Un graphe est dit *colorié* s'il est k-coloriable pour un $k > 0$. Le *nombre chromatique* d'un graphe non orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable.

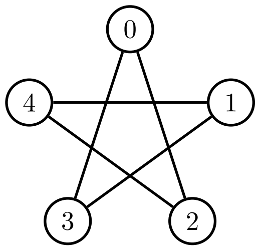
Dans cet énoncé, les graphes sont représentés par des matrices d'adjacence. L'étiquetage d'un graphe est défini par un tableau d'entiers. On donne les types suivants.

```
type graph = bool array array
type label = int array
```

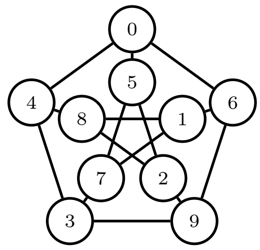
Un graphe non orienté $G = (V, E)$ avec $V = \{0, \dots, n-1\}$ est représenté par un identificateur `g` de type `graph` telle que pour $i, j \in V$, `g.(i).(j)= true` si et seulement si $(i, j) \in E$. Pour un étiquetage `lbl` de `g`, l'étiquette du sommet `i` est `lbl.(i)`.

Question 1.

□ **1.1.** Définir les identificateurs `g1` et `g2` associés aux graphes suivants.



Graphe `g1`.



Graphe `g2`.

□ **1.2.** Déterminer un étiquetage de `g1` qui soit un coloriage. Déterminer son nombre chromatique.

□ **1.3.** Déterminer le nombre chromatique et proposer un exemple de coloriage de `g2`.

□ **1.4.** Ecrire une fonction `is_col : graph -> label -> bool` telle que `is_col g lbl` renvoie `true` si et seulement si `lbl` est un coloriage de `g`. Dans le cas où la taille de l'étiquetage diffère du nombre de sommets du graphe, la fonction renvoie `false`. On demande une *complexité quadratique* en le nombre de sommets du graphe.

Question 2. Dans le cas général, le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets. En se limitant à des sous-probèmes, il est possible de faire mieux. Cette question étudie le cas du 2-coloriage.

□ **2.1.** Un graphe G est *biparti* lorsque l'ensemble de ses sommets V peut être divisé en deux sous-ensembles disjoints T et U (non vides), tels que chaque arête a une extrémité dans T et l'autre dans U. Démontrer qu'un graphe G est biparti si et seulement s'il est 2-coloriable.

□ **2.2.** Pour programmer la vérification de la 2-colorabilité d'un graphe, on effectue un *parcours en profondeur*. Pour ce faire, on se donne trois étiquettes : -1, 0, 1. L'étiquetage est initialisé à -1 pour tous les sommets, et on teste la 2-colorabilité avec 0 et 1 avec l'algorithme suivant.

(1) On choisit un sommet v d'étiquette -1.

(2) On colorie les sommets rencontrés lors du parcours en profondeur à partir de v, 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) S'il reste des sommets d'étiquette -1 alors on revient au point (1).

Ecrire une fonction `two_col : graph -> label` telle que `two_col g` renvoie un 2-coloriage de `g` si `g` est 2-coloriable. Le coloriage utilisera les couleurs 0 et 1. On demande une *complexité quadratique* en le nombre de sommets du graphe. Le comportement de la fonction est laissé au choix du candidat lorsque `g` n'est pas 2-coloriable.

Indication : on pourra se donner un étiquetage `lbl` de longueur `Array.length g`, dont toutes les cases sont initialisées à -1, et que l'on met à jour au fur et à mesure du parcours de `g`.

Question 3. Un *algorithme glouton* permet de colorier un graphe en temps polynomial en donnant en général un coloriage sous-optimal : le coloriage obtenu peut utiliser plus de couleurs que le coloriage optimal. Cet algorithme prend en paramètre un ordre sur les sommets du graphe appelé *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 `g2`. Pour un graphe `g` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num.(k)= j` si et seulement si le sommet j apparaît en $(k+1)$-ième position dans l'ordre. L'*algorithme glouton* de coloriage construit un coloriage C d'un graphe G en utilisant au plus $d(G)+1$ couleurs. On parcourt la liste des sommets du graphe dans l'ordre de numérotation des sommets donné et pour chaque sommet s parcouru :

(1) on calcule l'ensemble $\{C(t) \mid t \in \text{voisins de } s\}$ des couleurs déjà données aux voisins de s ;

(2) on cherche le plus petit entier naturel c qui n'appartient pas à cet ensemble ;

(3) on pose $C(s) = c$.

□ **3.1.** Considérons le graphe `g2` et les deux ordres de numérotation.

```
num1 = [|1;3;4;0;2;6;5;9;8;7|]
num2 = [|0;7;2;5;4;6;8;1;3;9|]
```

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

□ **3.2.** Écrire une fonction `min_col : graph -> label -> int -> int` telle que pour un graphe `g` à n sommets, un étiquetage `lbl` à valeurs dans $\{-1, \dots, n-1\}$ et pour un sommet `s` de `g`, `min_col g etiq s` renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{\text{lbl}.(t) \mid t \in \text{voisins de } s)\}$. On demande une *complexité* $O(n)$.

□ **3.3.** Ecrire une fonction `greedy : graph -> int array -> label`, telle que, pour un graphe `g` et un ordre de numérotation `num` de ses sommets, `greedy g num` renvoie le coloriage glouton de `g`, avec au plus $d+1$ sommets, où d est le degré de `g`. On demande une complexité $O(n^2)$, où n est 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 candidat.

Question 4. L'efficacité de l'algorithme glouton dépend de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe 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 peut prouver l'existence, mais cela n'apporte aucun bénéfice vis-à-vis de la complexité temporelle du problème.
Une alternative est donnée par l'optimisation de Welsh-Powell qui parcourt 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.

□ **4.1.** Ecrire une fonction `deg_sort : graph -> int array` qui calcule le tableau des sommets d'un graphe trié par ordre décroissant de leurs degrés.

□ **4.2.** En déduire une fonction `welsh_powell : graph -> label` qui implémente l'optimisation de Welsh-Powell. Justifier le choix de votre algorithme de tri pour la fonction `deg_sort`.