Applications du parcours en profondeur

Rappelons le principe du parcours en profondeur avec les calculs des "dates de découverte" et des "dates de fin de traitement" :

```
visiter_pp(g,s) :
  si dec[s] = -1 :
   dec[s]<-k; k++;
   pour tous les voisin x de s :
      visiter_pp(g,x);
   fin[s]<-k; k++;

visiter(g) :
   k<-0;
   pour chaque sommet s :
      dec[s]<- -1; fin[s]<- -1;
   pour chaque sommet s :
      si dec[s]= -1 :
         visiter_pp(g,s);
```

1. Ecrire un programme en Ocaml qui utilise le parcours en profondeur pour déterminer les composantes connexes d'un graphe non orienté représenté par ses listes d'adjacence. On pourra renvoyer un tableau indexé par les sommets qui contient dans la case i le numéro de la composante connexe dans laquelle se trouve le sommet i.
2. Réécrire l'algorithme de parcours en profondeur du cours de manière à renvoyer en sortie la forêt de parcours. On représentera une forêt par la donnée des pères des sommets et donc par un type `foret = int array`. Si `pere : int array` représente une forêt, `pere.(i)` pointe vers l'unique entier qui est le père de `i` ou vaut `-1` si le sommet `i` est une racine.
3. **Application 1 : le tri topologique.** Un graphe orienté acyclique induit un ordre partiel sur ses sommets en considérant la cloture refléxive transitive de la relation aRb ssi il existe un arc de a vers b. Trouver un ordre topologique pour un graphe orienté c'est déterminer un ordre total sur les sommets de ce graphe qui soit compatible avec lórdre partiel défini par le graphe. Autrement dit, on cherche à classer les sommets de telle sorte qu si le graphe contient un arc de a vers b alors a apparaitra avant b dans le classement.
   (a) Montrer que si un graphe possède un arce de a vers b alors la date de fin de traitement de b est forcément inférieure à celle de a.
   (b) En déduire une stratégie pour obtenir un ordre topologique sur les sommets.
   (c) Ecrire une fonction Ocaml exploitant un parcours en profondeur qui renvoie une liste contenant les sommets suivant l'ordre topologique. Plutôt que de calculer explicitement les dates de fin de traitement, on pourra insérer les sommets dans une pile au moment adéquat.
4. **Application 2 : 2-coloriage** On dit qu'un graphe $G = (S, A)$ est 2-coloriable lorsqu'il existe une fontion $c : S \to \{0, 1\}$ telle que $\forall (x, y) \in A, c(x) \neq c(y)$.
   (a) Expliquer comment tirer parti d'un parcours en profondeur pour trouver un 2-coloriage s'il en existe un et détecter qu'il n'en existe pas (indication : si on a déterminé la couleur d'un sommet alors on connait nécessairement celle de ses voisins et ainsi de suite).
   (b) Ecrire un programme Ocaml qui met en oeuvre cette stratégie.
5. **Application 3 : détection de cycle dans une graphe non orienté** Lorsque l'on fait un parcours en profondeur d'un graphe non orienté acyclique alors pour tout sommet `s`, dans la fonction `visiter_pp(s)`, le seul voisin de `s` qui est déjà découvert est son père. Ainsi, si on identifie un sommet ayant un voisin différent de son père déjà découvert alors on a mis en évidence l'existence d'un cycle.
   Mettre en oeuvre ce résultat en écrivant une fonction `detect_cycle` qui renvoie vrai ssi le graphe passé en argument est acyclique.
6. **Application 4 : détection de cycle dans une graphe orienté** Dans un graphe orienté, la situation est un peu plus subtile (mais plus facile à mettre en oeuvre) : si on identifie un sommet ayant un voisin déjà vu mais qui n'a pas encore été complètement traité (càd dont la date de fin de traitement n'est pas passée) alors on a mis en évidence l'existence d'un cycle.
   Mettre en oeuvre ce résultat en écrivant une fonction `detect_cycle_oreinte` qui renvoie vrai ssi le graphe passé en argument est acyclique.