TD : Exercices sur les graphes

1 Exercices sur les graphes

Exercice 1 : Exercice

1. Dans chacun des cas suivants, la quantité indiquée est-elle la même pour tous les arbres de parcours en largeur à partir d'un sommet x_0, ou peut-elle dépendre de l'ordre dans lequel les voisins d'un nœud sont examinés ?
   (a) Nombre de nœuds d'arité 1, 2...
   (b) Nombre de feuilles de l'arbre.
   (c) Profondeur du nœud correspondant à un sommet donné.
   (d) Hauteur de l'arbre.
   (e) Nombre de nœuds à chaque niveau de l'arbre.
2. Que peut-on dire des arêtes du graphe qui ne font pas partie de l'arbre ?

Dans tous les exercices qui suivent, on supposera que l'on dispose d'un type `graphe` défini ainsi :

```
type sommet = int
type graphe =
 {nb_sommets : int;
  voisins : sommet -> sommet list;
  adjacents : sommet -> sommet -> bool}
```

On supposera de plus que la fonction `voisins` s'exécute en temps constant, ce qui est le cas si le graphe est stocké sous forme d'un tableau de listes d'adjacence :

```
(* of_listes : sommet list array -> graphe *)
let of_listes t =
  let n = Array.length t in
  let v i = t.(i) in
  let adj i j = List.mem j (voisins i) in
  {nb_sommets = n; adjacents = adj; voisins = v}

(* Si g est défini par let g = of_listes t pour un certain t, alors g.voisins
   est bien en O(1) (mais pas g.adjacents). *)
```

Exercice 2 : Graphes 2-coloriables

Une *k-coloration* d'un graphe non orienté $G = (V, E)$ est une application $\varphi : V \to [0 \dots k-1]$ telle que $xy \in E \Longrightarrow \varphi(x) \neq \varphi(y)$. Un graphe est dit *$k$-coloriable* s'il admet une k-coloration. Déterminer si un graphe est k-coloriable est difficile (problème NP-complet) dès que $k \geqslant 3$. En revanche, le problème est très simple pour $k = 2$.

1. Montrer que si $G = (V, E)$ est connexe et $x_0 \in V$, alors il existe au plus une 2-coloration φ de G tel que $\varphi(x_0) = 0$.
2. Écrire une fonction `deux_coloration : graphe -> int array option` qui renvoie `Some t`, où `t` code une 2-coloration du graphe passé en argument, s'il en existe une, `None` sinon. On exige une complexité *linéaire* en la taille $|E| + |V|$ du graphe.
   Indication : on pourra initialiser `t` à -1 et utiliser la question précédente.

Exercice 3 : Graphe miroir

Si $G = (V, E)$ est un graphe orienté, son *graphe miroir* $G^{\leftarrow}$ est obtenu en gardant le même ensemble de sommets et en inversant le sens de tous les arcs :

$$G^{\leftarrow} := \Big(V, \big\{(v, u) \mid (u, v) \in E\big\}\Big).$$

Écrire une fonction `miroir` (dont vous devriez pouvoir deviner la spécification). On exige une complexité en $\mathrm{O}(n+p)$.

```
miroir : graphe -> graphe
```

```
# miroir g0;;
- : sommet list array =
[| [3]; [6; 4; 0]; [4; 1; 0]; [1];
   [1]; [11; 3]; [8]; [9]; [7]; [6];
   [9; 5]; [10]|]
```

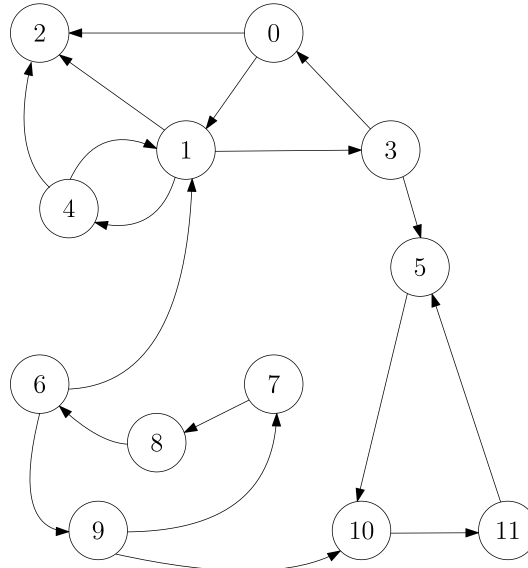

FIGURE 1 – Le graphe `g0`

Exercice 4 : Test de forte connexité

On considère un graphe orienté $G = (V, E)$, et l'on souhaite déterminer de manière efficace s'il est fortement connexe.

1. Proposer un algorithme naïf pour répondre à la question, et déterminer sa complexité.
2. On note $G^{\leftarrow}(V, E^{\leftarrow})$, le graphe obtenu en changeant le sens de toutes les arêtes de G ; autrement dit, on pose $E^{\leftarrow} := \{(v, u) | (u, v) \in E\}$. Proposer un algorithme répondant au problème posé à l'aide d'un parcours de G et d'un parcours de $G^{\leftarrow}$. On justifiera soigneusement sa correction.
3. Déterminer la complexité de l'algorithme.
4. Écrire la fonction `est_fortement_connexe : graphe -> bool`.

Exercice 5 : Théorème de Robbins et détection des ponts

Un graphe non orienté est dit *fortement orientable* s'il existe une orientation de ses arêtes qui le rende fortement connexe. D'autre part, un graphe non orienté connexe est dit 2-arête connexe, ou *sans pont*, s'il n'existe pas d'arête dont la suppression déconnecterait le graphe (une telle arête est appelée *pont*).

1. Montrer qu'une arête est un pont si et seulement si elle ne fait partie d'aucun cycle élémentaire.
2. On considère un graphe connexe G et T un arbre de parcours en profondeur pour G à partir d'un sommet (quelconque) r. On considère l'orientation suivante de G :
   — les arêtes de T sont orientées de la racine vers les feuilles ;
   — les autres arêtes, qui relient forcément un nœud et l'un de ses ancêtres dans T (*cf.* théorème **??**), sont orientées des feuilles vers la racine.
   Montrer que si G est sans pont, cette orientation rend G fortement connexe.
3. En déduire le *théorème de Robbins (1939)* : un graphe est fortement orientable si et seulement si il est sans pont.

   Autrement dit, il est possible de mettre toutes les rues d'une ville à sens unique (en gardant la possibilité d'aller de n'importe quel point A à n'importe quel point B) si et seulement si on ne peut séparer la ville en deux parties qui ne sont reliées que par une seule rue.

4. On définit :

```
(* Soit un graphe, soit une arête *)
type t = G of graphe | A of int * int
```

Écrire une fonction `orientation_forte : graphe -> t` qui prend en entrée un graphe supposé non orienté, et renvoie :
— `G g'`, où `g'` est une orientation forte de `g` s'il en existe une ;
— `A (i, j)`, où ij est un pont dans `g`, sinon.
On demande une complexité linéaire en la taille du graphe (c'est-à-dire en $O(|V| + |E|)$).

Exercice 6 : Couverture par des tests

On considère la fonction suivante :

```c
int count_occurrences(int arr[], int len, int x){
  int count = 0;
  for (int i = 0; i < len; i++) {
    if (arr[i] == x) count++;
  }
  return count;
}
```

1. Dessiner le graphe de contrôle de flot (CFG) associé à cette fonction. On rappelle que ce graphe possède un sommet pour chaque instruction (ou chaque bloc de base, mais ici cela ne changera rien) et un arc du sommet x au sommet y s'il est (syntaxiquement) possible d'exécuter y immédiatement après x.
2. Donner un test ou un jeu de tests permettant d'assurer le critère de *couverture des sommets* : l'exécution du jeu de tests doit passer par tous les sommets du CFG.
3. Proposer une version manifestement fausse de la fonction pour laquelle ce jeu de tests ne détecterait pas d'erreur.
4. Donner un test ou un jeu de tests permettant d'assurer le critère de *couverture des arcs* : l'exécution du jeu de tests doit emprunter tous les arcs du CFG.
5. Proposer une version fausse de la fonction pour laquelle ce nouveau jeu de tests ne détecterait pas d'erreurs.

Exercice 7 : Parcours en profondeur itératif efficace

On a vu en TD que la manière la plus simple d'écrire un « vrai » parcours en profondeur itératif avait une complexité spatiale en $O(|E|)$. Écrire une fonction purement itérative (ou récursive terminale) réalisant un parcours en profondeur, en utilisant l'idée suivante :
— la pile ne contient pas des sommets, mais des *listes de sommets* ;
— un élément de la pile correspond précisément à une *stack frame* du parcours en profondeur récursif : la liste contient les appels récursifs qu'il reste à faire.
En parcourant à partir du sommet 5 dans le graphe de la figure du cours, les premières étapes d'évolution de la pile doivent être :

$$(5) \to (4,6,8) \to \begin{array}{c}(2,3,5,7)\\(6,8)\end{array} \to \begin{array}{c}(0,1,4)\\(3,5,7)\\(6,8)\end{array} \to \begin{array}{c}()\\(1,4)\\(3,5,7)\\(6,8)\end{array} \to \begin{array}{c}(1,4)\\(3,5,7)\\(6,8)\end{array} \to \begin{array}{c}(2,6)\\(4)\\(3,5,7)\\(6,8)\end{array} \to \begin{array}{c}(6)\\(4)\\(3,5,7)\\(6,8)\end{array} \to \begin{array}{c}(1,2,5,8)\\()\\(4)\\(3,5,7)\\(6,8)\end{array}$$

On justifiera bien que, si le graphe est stocké sous forme de listes d'adjacence, la complexité spatiale est en $O(|V|)$ (un schéma mémoire peut être utile).