

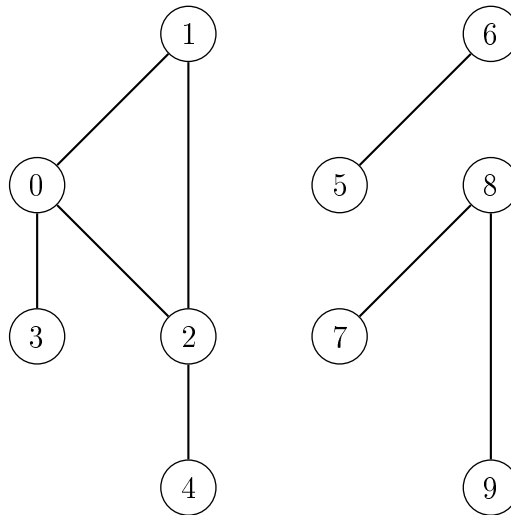
TD 7

Composante connexe

Correction

Une composante connexe d'un graphe non-orienté est un sous graphe connexe maximal du graphe.

On considère le graphe `g_ex` suivant, qui admet 3 composante connexe :



Son implémentation en Ocaml est donné par la liste d'adjacence. Nous utiliserons alors le type suivant :

```
type graphe = int list array ;;
```

Ce qui donne :

```
let (g_ex : graphe) = [| [1;2;3]; [0;2]; [0;1;4]; [0]; [2] ; [6]; [5]; [8];  
[7;9]; [8] |];;
```

1. Écrire la fonction `parcours` qui prend en argument un graphe, une liste de sommets déjà vu, une liste de sommets à voir, et renvoie la liste des sommets accessible depuis la liste des sommets à voir.

(On peut procéder au parcours en profondeur du graphe.)

```
parcours : graphe -> int list -> int list -> int list
```

Exemple :

```
parcours g_ex [] g_ex.(0);;  
- : int list = [3; 4; 2; 0; 1]
```

Correction

Listing 1 – parcours |

```
let rec parcours (g : graphe) listevu listeavoir = match listeavoir with  
| [] -> listevu  
| t :: q -> if List.mem t listevu then parcours g listevu q  
else parcours g (t :: listevu) (g.(t) @ q);;
```

2. Écrire la fonction `est_connexe` qui prend en argument un graphe et renvoie le booléen associé au caractère connexe du graphe.

`est_connexe : graphe -> bool`

Exemple :

```
est_connexe g_ex ;;  
- : bool = false
```

Correction

Listing 2 – `est_connexe` |

```
let est_connexe (g:graphe) =  
  let longueur_parcours = List.length (parcours g_ex [] g_ex.(0)) in  
  let taille = Array.length g in  
  longueur_parcours = taille ;;
```

3. Pour chercher les composantes connexes, l'idée est de parcourir le graphe à partir du premier sommet, puis de le parcourir à partir du premier sommet non vu, et ainsi tant qu'il reste des sommets non vus.

- (a) Écrire la fonction `premier_libre` qui prend en argument une liste d'entier, et renvoie le plus petit entier non présent dans la liste.

`premier_libre : int list -> int`

Exemple :

```
premier_libre [ 5;0;1;3;7;2];;  
- : int = 4
```

Correction

Listing 3 – `premier_libre` |

```
let premier_libre liste =  
  let rec aux k =  
    if List.mem k liste then aux (k+1)  
    else k  
  in aux 0;;
```

- (b) Écrire la fonction `composante_connexes` qui prend en argument un graphe et renvoie la liste des composantes connexes.

On pourra utiliser la fonction `List.concat` pour pouvoir utiliser la fonction `premier_libre` sur la liste des composantes connexes déjà trouvées.

`composante_connexes : graphe -> int list list`

Exemple :

```
composante_connexes g_ex;;  
- : int list list = [[9; 8; 7]; [6; 5]; [3; 4; 2; 1; 0]]
```

Correction

Listing 4 – composante_connexes |

```
let composante_connexes g =  
  let n = Array.length g in  
  let rec aux acc =  
    let liste_vu = List.concat acc in  
    let k = premierLibre liste_vu in  
    if k < n then  
      begin  
        let liste = parcours g [] [k] in  
        aux (liste :: acc)  
      end  
    else acc  
  in aux [];;
```