Table of Contents

TP 5 - Programmation pour la préparation à l'agrégation maths option info

TP 5 : Graphes.

- En OCaml.

```
In [2]: let print = Printf.printf;;
        Sys.command "ocaml -version";;

Out[2]: val print : ('a, out_channel, unit) format → 'a = <fun>

        The OCaml toplevel, version 4.04.2

Out[2]: - : int = 0
```

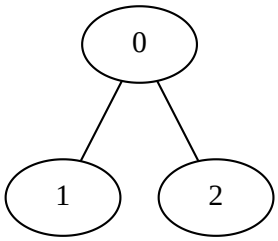
```
In [3]: print_endline

Out[3]: - : string → unit = <fun>
```

Représentation

On prend un petit exemple de graphe avec lequel on va travailler, pour vérifier que chaque représentation permet bien de le représenter.

Graphe :

```
0 - 1
0 - 2
```



Trois représentations

In [4]:
```
type sommet = int;;
```

Out[4]:
```
type sommet = int
```

On supposera que les sommets sont toujours numérotés de 0 à $n-1$.

Pour des graphes non orientés, on doit stocker deux fois chaque arête : $a \to b$ et $b \to a$.

On suppose aussi que des arêtes de la forme $a - a$ ne sont pas considérées : pas de boucle sur soi-même ! Ca simplifie les codes...

Matrice d'adjacence

Plutôt que d'utiliser des `bool`, on utilise `0` et `1` pour facilement compter le nombre d'arêtes en sommant le nombre de `1`. (en plus, ça s'écrit plus vite !)

In [5]:
```
type graphe_mat = int array array;;
```

Out[5]:
```
type graphe_mat = int array array
```

In [6]:
```
let g1__mat : graphe_mat = [|
      [| 0; 1; 1 |];  (* 0 -- 1 et 0 -- 2 *)
      [| 1; 0; 0 |];  (* 1 -- 0 *)
      [| 1; 0; 0 |]   (* 2 -- 0 *)
   |]
;;
```

Out[6]:
```
val g1__mat : graphe_mat = [|[|0; 1; 1|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

Listes d'adjancence

In [7]:
```
type graphe_adj = (sommet list) array;;
```

Out[7]:
```
type graphe_adj = sommet list array
```

In [8]:
```
let g1__adj : graphe_adj = [|
    [1; 2]; (* 0 -- 1 et 0 -- 2 *)
    [0]; (* 1 -- 0 *)
    [0]  (* 2 -- 0 *)
|];;
```

Out[8]:
```
val g1__adj : graphe_adj = [|[1; 2]; [0]; [0]|]
```

Listes d'arêtes

In [9]:
```
type arete = sommet * sommet;;
type graphe_art = arete list;;
```

Out[9]:
```
type arete = sommet * sommet
```

Out[9]:
```
type graphe_art = arete list
```

In [10]:
```
let g1__art : graphe_art = [
    (0, 1); (0, 2); (* 0 -- 1 et 0 -- 2 *)
    (1, 0); (* 1 -- 0 *)
    (2, 0)  (* 2 -- 0 *)
];;
```

Out[10]:
```
val g1__art : graphe_art = [(0, 1); (0, 2); (1, 0); (2, 0)]
```

Nombres de sommets et d'arcs

Matrice d'adjacence

Pour `graphe_mat`, `nb_sommets` est en $\mathcal{O}(1)$ et `nb_arcs` est en $\mathcal{O}(n^2)$.

In [11]:
```
let somme_tableau = Array.fold_left (+) 0;;
let somme_matrice = Array.fold_left (fun x a -> x + (somme_tableau a)) 0;;
```

Out[11]:
```
val somme_tableau : int array → int = <fun>
```

Out[11]:
```
val somme_matrice : int array array → int = <fun>
```

In [12]:
```
let nb_sommets__mat (g : graphe_mat) : int = Array.length g ;;
nb_sommets__mat g1__mat;;

let nb_arcs__mat (g : graphe_mat) : int = (somme_matrice g) / 2 ;;
nb_arcs__mat g1__mat;;
```

Out[12]:
```
val nb_sommets__mat : graphe_mat → int = <fun>
```

Out[12]:
```
- : int = 3
```

Out[12]:
```
val nb_arcs__mat : graphe_mat → int = <fun>
```

Out[12]:
```
- : int = 2
```

Listes d'adjacence

Pour `graphe_adj`, `nb_sommets` est en $\mathcal{O}(1)$ et `nb_arcs` est en $\mathcal{O}(n)$.

In [13]:
```
let somme_list = List.fold_left (+) 0;;
```

Out[13]:
```
val somme_list : int list → int = <fun>
```

In [14]:
```
let nb_sommets__adj (g : graphe_adj) : int = Array.length g ;;
nb_sommets__adj g1__adj;;


let nb_arcs__adj (g : graphe_adj) : int = (somme_list (Array.to_list (Array.map List.length g))) / 2 ;;
nb_arcs__adj g1__adj;;
```

Out[14]:
```
val nb_sommets__adj : graphe_adj → int = <fun>
```

Out[14]:
```
- : int = 3
```

Out[14]:
```
val nb_arcs__adj : graphe_adj → int = <fun>
```

Out[14]:
```
- : int = 2
```

Listes d'arêtes

Pour `graphe_art`, `nb_sommets` est en $\mathcal{O}(n)$ et `nb_arcs` est en $\mathcal{O}(1)$.

In [15]:
```
let max_list = List.fold_left max min_int;;
max_list [1; 3; 4; 19];;
```

Out[15]:
```
val max_list : int list → int = <fun>
```

Out[15]:
```
- : int = 19
```

In [16]:
```
let max_list_couple l =
    let g, d = List.split l in
    max (max_list g) (max_list d)
;;
```

Out[16]:
```
val max_list_couple : (int * int) list → int = <fun>
```

In [17]:
```
let nb_sommets__art (g : graphe_art) : int = 1 + (max_list_couple g);;
nb_sommets__art g1__art;;

let nb_arcs__art (g : graphe_art) : int = (List.length g) / 2 ;;
nb_arcs__art g1__art;;
```

Out[17]:
```
val nb_sommets__art : graphe_art → int = <fun>
```

Out[17]:
```
- : int = 3
```

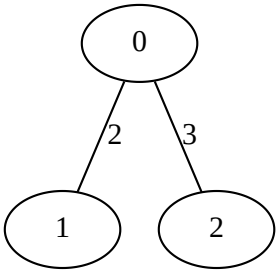
Out[17]:
```
val nb_arcs__art : graphe_art → int = <fun>
```

Out[17]:
```
- : int = 2
```

Graphes pondérés

La définition des types est assez explicite. On utilise le même exemple de graphe :

```
0 -[2]- 1
0 -[3]- 2
```



In [18]:
```
type poids = int;;
```

Out[18]:
```
type poids = int
```

Matrice d'adjacence

`None` indique une absence d'arête, `Some x` une arête pondérée par `x`. Aucune raison qu'on ne puisse pas pondérer par `0`, donc utiliser seulement `0` pour indiquer une absence d'arête ne marchera pas.

In [19]:
```
type graphe_mat_pond = (poids option) array array;;
```

Out[19]:
```
type graphe_mat_pond = poids option array array
```

In [20]:
```
let g1__mat_pond : graphe_mat_pond = [|
      [| None; Some 2; Some 3 |];  (* 0 -[2]- 1 et 0 -[3]- 2 *)
      [| Some 2; None; None |];  (* 1 -[2]- 0 *)
      [| Some 3; None; None |]   (* 2 -[3]- 0 *)
   |]
;;
```

Out[20]:
```
val g1__mat_pond : graphe_mat_pond =
  [|[|None; Some 2; Some 3|]; [|Some 2; None; None|];
    [|Some 3; None; None|]|]
```

Listes d'adjacence

C'est plus facile :

In [21]:
```
type graphe_adj_pond = ((sommet * poids) list) array;;
```

Out[21]:
```
type graphe_adj_pond = (sommet * poids) list array
```

In [22]:
```
let g1__adj_pond : graphe_adj_pond = [|
   [(1, 2); (2, 3)]; (* 0 -[2]- 1 et 0 -[3]- 2 *)
   [(0, 2)]; (* 1 -[2]- 0 *)
   [(0, 3)]  (* 2 -[3]- 0 *)
|];;
```

Out[22]:
```
val g1__adj_pond : graphe_adj_pond = [|[(1, 2); (2, 3)]; [(0, 2)]; [(0, 3)]|]
```

Listes d'arêtes

C'est très facile :

In [23]:
```
type arete_pond = sommet * poids * sommet;;
type graphe_art_pond = arete_pond list;;
```

Out[23]:
```
type arete_pond = sommet * poids * sommet
```

Out[23]:
```
type graphe_art_pond = arete_pond list
```

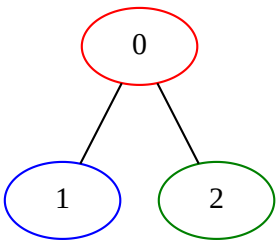
In [24]:
```
let g1__art_pond : graphe_art_pond = [
   (0, 2, 1); (0, 3, 2); (* 0 -[2]- 1 et 0 -[3]- 2 *)
   (1, 2, 0); (* 1 -[2]- 0 *)
   (2, 3, 0)  (* 2 -[3]- 0 *)
];;
```

Out[24]:
```
val g1__art_pond : graphe_art_pond =
  [(0, 2, 1); (0, 3, 2); (1, 2, 0); (2, 3, 0)]
```

Graphes colorés

La définition des types est assez explicite. On utilise le même exemple de graphe :

```
0 [rouge] -- 1 [bleu]
0 [rouge] -- 2 [vert]
```



In [25]:
```
type couleur = int;;
let rouge : couleur = 1 and bleu : couleur = 2 and vert : couleur = 3;;
```

Out[25]:
```
type couleur = int
```

Out[25]:
```
val rouge : couleur = 1
val bleu : couleur = 2
val vert : couleur = 3
```

Matrice d'adjacence

C'est moins facile ! On est obligé d'ajouter une structure qui contient la liste des couleurs séparément... Et donc ce n'est pas très intéressant...

In [26]:
```
type graphe_mat_color = { mat : int array array; couleurs : couleur array } ;;
```

Out[26]:
```
type graphe_mat_color = { mat : int array array; couleurs : couleur array; }
```

In [27]:
```
let g1__mat_color : graphe_mat_color = { mat = [|
      [| 0; 1; 1 |];  (* 0 -- 1 et 0 -- 2 *)
      [| 1; 0; 0 |];  (* 1 -- 0 *)
      [| 1; 0; 0 |]   (* 2 -- 0 *)
   |];
   couleurs = [| rouge; bleu; vert |]
};;
```

Out[27]:
```
val g1__mat_color : graphe_mat_color =
  {mat = [|[|0; 1; 1|]; [|1; 0; 0|]; [|1; 0; 0|]|]; couleurs = [|1; 2; 3|]}
```

Listes d'adjacence

In [28]:
```
type graphe_adj_color = { adj : (sommet list) array; couleurs : couleur array } ;;
```

Out[28]:
```
type graphe_adj_color = {
  adj : sommet list array;
  couleurs : couleur array;
}
```

In [29]:
```
let g1__adj_color : graphe_adj_color = { adj = [|
      [1; 2]; (* 0 -- 1 et 0 -- 2 *)
      [0]; (* 1 -- 0 *)
      [0]  (* 2 -- 0 *)
   |];
   couleurs = [| rouge; bleu; vert |]
}
```

Out[29]:
```
val g1__adj_color : graphe_adj_color =
  {adj = [|[1; 2]; [0]; [0]|]; couleurs = [|1; 2; 3|]}
```

Listes d'arêtes

In [30]:
```
type graphe_art_color = { art : arete list; couleurs : couleur array } ;;
```

Out[30]:
```
type graphe_art_color = { art : arete list; couleurs : couleur array; }
```

In [31]:
```
let g1__art_color : graphe_art_color = { art = [
      (0, 1); (0, 2); (* 0 -- 1 et 0 -- 2 *)
      (1, 0); (* 1 -- 0 *)
      (2, 0)  (* 2 -- 0 *)
   ];
   couleurs = [| rouge; bleu; vert |]
}
```

Out[31]:
```
val g1__art_color : graphe_art_color =
  {art = [(0, 1); (0, 2); (1, 0); (2, 0)]; couleurs = [|1; 2; 3|]}
```

Degrés

Matrice d'adjacence

Pour `graphe_mat`, `degres` est en $\mathcal{O}(n^2)$.

In [32]:
```
let degres__mat (g : graphe_mat) : int array = Array.map somme_tableau g ;;
degres__mat g1__mat;;
```

```
Out[32]: val degres__mat : graphe_mat → int array = <fun>
Out[32]: - : int array = [|2; 1; 1|]
```

Listes d'adjacence

Pour `graphe_adj`, `degres` est en $\mathcal{O}(n)$.

In [33]:
```
let degres__adj (g : graphe_adj) : int array = Array.map List.length g ;;
degres__adj g1__adj;;
```

```
Out[33]: val degres__adj : graphe_adj → int array = <fun>
Out[33]: - : int array = [|2; 1; 1|]
```

Listes d'arêtes

Pour `graphe_art`, `degres` est en $\mathcal{O}(n^2)$.

In [34]:
```
g1__art
```

```
Out[34]: - : graphe_art = [(0, 1); (0, 2); (1, 0); (2, 0)]
```

In [35]:
```
let degres__art (g : graphe_art) : int array =
  let n = nb_sommets__art g in
  Array.init n (fun i ->
    List.length (List.filter (fun (a, _) -> a = i) g)
  )
;;
degres__art g1__art;;
```

```
Out[35]: val degres__art : graphe_art → int array = <fun>
Out[35]: - : int array = [|2; 1; 1|]
```

Parcours de graphes

Pour la suite, on choisit les représentations qui sont les plus adaptées aux algorithmes qu'on doit écrire.

Parcours en profondeur et largeur

Pour les deux parcours, l'implémentation sous forme de listes d'adjacence fonctionne très bien.

Les deux algorithmes sont très similaires, et sont en $\mathcal{O}(|A|)$ ($|A|$ étant le nombre d'arêtes, si $G = (S, A)$), si on utilise une structure de pile/file qui est efficace (insertion, suppression en $\mathcal{O}(1)$).

On va être un peu fainéant, et ces deux parcours *ne renverront rien*, ils vont juste afficher les sommets dans l'ordre dans lesquels on les voit. On pourrait utiliser une référence d'une liste (`list ref`) pour ajouter les sommets un à un.

En profondeur : avec une pile (`Stack`)

In [36]:
```
let profondeur_iter (g : graphe_adj) (debut : sommet) : unit =
  let vu = Array.make (nb_sommets__adj g) false in
  let pile = Stack.create () in
  Stack.push debut pile;
  vu.(debut) <- true;
  while not (Stack.is_empty pile) do
    let i = Stack.pop pile in
    Printf.printf "visite(%d)\n" i;
    flush_all();
    (* Complexité O(deg(i)) pour le sommet i *)
    List.iter (fun j -> if not vu.(j) then begin
      Stack.push j pile;
      vu.(j) <- true
      end)
    g.(i)
  done
  (* donc en tout, complexité en Sigma_i O(deg(i)) = |E| *)
;;
```

Out[36]:
```
val profondeur_iter : graphe_adj → sommet → unit = <fun>
```

On remarque qu'on parcourt les sommets de "la droite vers la gauche" dans cet exemple.

In [37]:
```
g1__adj;;
```

Out[37]:
```
- : graphe_adj = [|[1; 2]; [0]; [0]|]
```

In [38]:
```
profondeur_iter g1__adj 0;;
```

```
visite(0)
visite(2)
visite(1)
```

Out[38]:
```
- : unit = ()
```

En largeur : avec une file (`Queue`)

C'est magique, le code est *exactement* le même, avec `Queue` en lieu et place de `Stack`.

> On a déjà vu tout ça, vous devriez être capable de le réécrire rapidement !

In [39]:
```
let largeur_iter (g : graphe_adj) (debut : sommet) : unit =
  let vu = Array.make (nb_sommets__adj g) false in
  let file = Queue.create () in
  Queue.push debut file;
  vu.(debut) <- true;
  while not (Queue.is_empty file) do
    let i = Queue.pop file in
    Printf.printf "visite(%d)\n" i;
    flush_all();
    List.iter (fun j -> if not vu.(j) then begin
      Queue.push j file;
      vu.(j) <- true
      end)
    g.(i)
  done
;;
```

Out[39]:
```
val largeur_iter : graphe_adj → sommet → unit = <fun>
```

On remarque qu'on parcourt les sommets de "la gauche vers la droite" dans cet exemple.

In [40]:
```
g1__adj;;
```

Out[40]:
```
- : graphe_adj = [|[1; 2]; [0]; [0]|]
```

In [41]:
```
largeur_iter g1__adj 0;;
```

```
visite(0)
visite(1)
visite(2)
```

Out[41]:
```
- : unit = ()
```

> Vous pouvez aussi faire des versions récursives de ces parcours.

`est_connexe`

Un graphe est connexe *si et seulement si* chaque sommet est relié à tout autre sommet (par un chemin de longueur un ou plus). On écrit d'abord une fonction qui vérifie que tous les sommets sont accessibles depuis un sommet, puis on vérifiera que ce prédicat est vrai pour tous les sommets.

In [42]:
```
let tous_vrais = Array.fold_left (&&) true;;
```

Out[42]:
```
val tous_vrais : bool array → bool = <fun>
```

In [43]:
```
let tous_accessibles (g : graphe_adj) (debut : sommet) : bool =
  let vu = Array.make (nb_sommets__adj g) false in
  let file = Queue.create () in
  Queue.push debut file;
  vu.(debut) <- true;  (* on ne peut pas se passer du tableau vu *)
  while not (Queue.is_empty file) do
    let i = Queue.pop file in
    List.iter (fun j -> if not vu.(j) then begin (* car utile ici *)
      Queue.push j file;
      vu.(j) <- true
      end)
    g.(i)
  done;
  (* mais a la fin on s'en sert juste pour ce test *)
  tous_vrais vu
;;
```

Out[43]:
```
val tous_accessibles : graphe_adj → sommet → bool = <fun>
```

In [44]:
```
tous_accessibles g1__adj 0;;
tous_accessibles g1__adj 1;;
tous_accessibles g1__adj 2;;
```

Out[44]:
```
- : bool = true
```

Out[44]:
```
- : bool = true
```

Out[44]:
```
- : bool = true
```

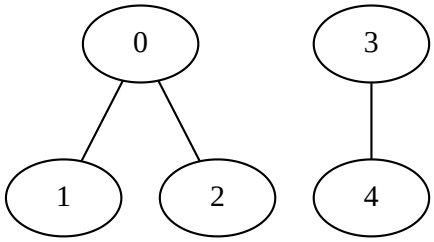
In [45]:
```
let est_connexe (g : graphe_adj) : bool =
  let n = nb_sommets__adj g in
  tous_vrais (Array.init n (fun i -> tous_accessibles g i));
;;
```

Out[45]:
```
val est_connexe : graphe_adj → bool = <fun>
```

In [46]:
```
est_connexe g1__adj;;
```

Out[46]:
```
- : bool = true
```

Et avec un exemple de graphe non connexe :



In [47]:
```
let g2__adj : graphe_adj = [|
   [1; 2]; (* 0 -- 1 et 0 -- 2 *)
   [0]; (* 1 -- 0 *)
   [0]; (* 2 -- 0 *)
   [4]; (* 3 -- 4 *)
   [3]; (* 4 -- 3 *)
|];;
```

Out[47]:
```
val g2__adj : graphe_adj = [|[1; 2]; [0]; [0]; [4]; [3]|]
```

In [48]:
```
largeur_iter g2__adj 0;;

largeur_iter g2__adj 3;;
```

```
visite(0)
visite(1)
visite(2)
```

Out[48]:
```
- : unit = ()
```

```
visite(3)
visite(4)
```

Out[48]:
```
- : unit = ()
```

In [49]:
```
est_connexe g2__adj;;
```

Out[49]:
```
- : bool = false
```

est_arbre

Un arbre est un graphe connexe acyclique.

1. on sait déjà vérifier la connexité.
2. on doit vérifier l'absence de cycle.

Je vous laisse réfléchir par vous-même pour le second point. (Exemple (http://www.geeksforgeeks.org/detect-cycle-undirected-graph/))

Si besoin, voici une correction.

In [64]:
```
let rec est_cyclique_aux (g : graphe_adj) (v : sommet) (vu : bool array) (parent : sommet) =
    vu.(v) <- true; (* v est vu ! *)
    let res = ref false in
    let indice = ref 0 in
    let gv = Array.of_list g.(v) in (* la conversion prend un temps O(deg(v)) une fois *)
    while (not !res) && (!indice < Array.length gv) do
        let i = gv.(!indice) in (* comme ca cette lecture prend O(1) chaque fois *)
        incr indice;
        if not vu.(i) then begin
            if (est_cyclique_aux g i vu v) then
                res := true;
        end
        else begin
            if parent != i then
                res := true
        end
    done;
    !res
;;
```

Out[64]:
```
val est_cyclique_aux : graphe_adj → sommet → bool array → sommet → bool =
  <fun>
```

In [65]:
```
let est_cyclique (g : graphe_adj) : bool =
    let n = nb_sommets__adj g in
    let vu = Array.make n false in
    let res = ref false in
    for i = 0 to n - 1 do
        if not vu.(i) then begin
            if est_cyclique_aux g i vu (-1) then
                res := true;
        end
    done;
    !res
;;
```

Out[65]:
```
val est_cyclique : graphe_adj → bool = <fun>
```

In [66]:
```
let absence_cycle (g : graphe_adj) : bool =
    not (est_cyclique g)
;;
```

Out[66]:
```
val absence_cycle : graphe_adj → bool = <fun>
```

In [67]:
```
let est_arbre (g : graphe_adj) : bool =
    (est_connexe g) && (absence_cycle g)
;;
```

Out[67]:
```
val est_arbre : graphe_adj → bool = <fun>
```

In [68]:
```
est_cyclique g1__adj;;
est_arbre g1__adj;;
```

Out[68]:
```
- : bool = false
```

Out[68]:
```
- : bool = true
```

In [69]:
```
est_cyclique g2__adj;;
est_arbre g2__adj;;
```

Out[69]:
```
- : bool = false
```

Out[69]:
```
- : bool = false
```

Et avec un exemple de graphe connexe mais avec un cycle :

In [70]:
```
let g3__adj : graphe_adj = [|
   [1; 2]; (* 0 -- 1 et 0 -- 2 *)
   [0; 2]; (* 1 -- 0 et 1 -- 2 *)
   [0; 1]; (* 2 -- 0 et 2 -- 1 *)
|];;
```

Out[70]:
```
val g3__adj : graphe_adj = [|[1; 2]; [0; 2]; [0; 1]|]
```

In [71]:
```
est_connexe g3__adj;;
est_arbre g3__adj;;
```

Out[71]:
```
- : bool = true
```

Out[71]:
```
- : bool = false
```

composantes_connexes

Pour chaque sommet, on fait un parcours en largeur, et on ajoute tous les sommets visités dans la même composante connexe. Dès qu'un nouveau sommet n'a pas encore été visité, on commence une nouvelle composante connexe.

Cet algorithme est en $\mathcal{O}(n)$, au pire chaque sommet est visité exactement une fois.

In [72]:
```
let composantes_connexes (g : graphe_adj) : sommet list list =
   let n = nb_sommets__adj g in
   let vu = Array.make n false in
   let cc_courante = ref [] in
   let rec visite (i : sommet) : unit =
      Printf.printf "visite(%d)\n" i; (* permet de vérifier que chaque sommet n'est visité qu'une seule fois ! *)
      flush_all();
      vu.(i) <- true;
      cc_courante := i :: !cc_courante;
      (* cette opération est linéaire en deg(i) le degré de i *)
      List.iter (fun j -> if not vu.(j) then visite j) g.(i)
   in
   let cc = ref [] in
   for i = 0 to n - 1 do
      (* au pire, on est en O(somme deg(i)) = O(n^2) *)
      (* mais en fait un sommet deja vu ne sera pas considere par la suite *)
      (* donc on est en O(n) en fait ! *)
      if not vu.(i) then begin
         visite i; (* au pire, chaque visite est en O(deg(i)) *)
         cc := !cc_courante :: !cc;
         cc_courante := []
      end
   done;
   !cc
;;
```

Out[72]:
```
val composantes_connexes : graphe_adj → sommet list list = <fun>
```

In [73]:
```
composantes_connexes g1__adj;;
```

```
visite(0)
visite(1)
visite(2)
```

Out[73]:
```
- : sommet list list = [[2; 1; 0]]
```

In [74]:
```
composantes_connexes g2__adj;;
```

```
visite(0)
visite(1)
visite(2)
visite(3)
visite(4)
```

Out[74]:
```
- : sommet list list = [[4; 3]; [2; 1; 0]]
```

2-coloriage

Le 2-coloriage est très facile : si un seul sommet a un degré ≥ 3, ce n'est pas possible. Si tous les sommets ont un degrés ≤ 2, on part d'un sommet (pour chaque composante connexe) et on alterne entre deux couleurs en parcourant la composante connexe...

Cet algorithme est aussi en $\mathcal{O}(n)$.

In [77]:
```
type deuxcouleur = Blanc | Noir;; (* on pourrait utiliser bool *)
```

Out[77]:
```
type deuxcouleur = Blanc | Noir
```

In [78]:
```
let alterne_couleur = function  (* avec bool, cette fonction serait... juste not *)
    | Blanc -> Noir
    | Noir -> Blanc
;;
```

Out[78]:
```
val alterne_couleur : deuxcouleur → deuxcouleur = <fun>
```

In [63]:
```
let max_array = Array.fold_left max min_int;;
```

Out[63]:
```
val max_array : int array → int = <fun>
```

In [81]:
```
let deuxcoloriage (g : graphe_adj) : deuxcouleur array =
    let n = nb_sommets__adj g in
    let vu = Array.make n false in
    let couleurs = Array.make n Blanc in
    let cc = composantes_connexes g in
    let rec visite_et_colorie_en_alternance (c : deuxcouleur) (i : sommet) : unit =
        Printf.printf "visite(%d)\n" i;
        flush_all();
        vu.(i) <- true;
        couleurs.(i) <- c;
        List.iter (fun j ->
            if not vu.(j) then
                visite_et_colorie_en_alternance (alterne_couleur c) j
            else begin
                if couleurs.(j) = c then failwith "2-coloriage impossible."
            end
        ) g.(i)
    in
    List.iter (visite_et_colorie_en_alternance Blanc) (List.map List.hd cc);
    couleurs
;;
```

Out[81]:
```
val deuxcoloriage : graphe_adj → deuxcouleur array = <fun>
```

In [82]:
```
deuxcoloriage g1__adj;;
```

```
visite(0)
visite(1)
visite(2)
visite(2)
visite(0)
visite(1)
```

Out[82]:
```
- : deuxcouleur array = [|Noir; Blanc; Blanc|]
```

Pour le deuxième exemple, on voit que la seconde composante connexe $\{3, 4\}$ est coloriée avec deux couleurs aussi.

In [83]:
```
deuxcoloriage g2__adj;;
```

```
visite(0)
visite(1)
visite(2)
visite(3)
visite(4)
visite(4)
visite(3)
visite(2)
visite(0)
visite(1)
```

Out[83]:
```
- : deuxcouleur array = [|Noir; Blanc; Blanc; Noir; Blanc|]
```

In [84]:
```
let g3__adj : graphe_adj = [|
   [1; 2; 3]; (* 0 -- 1 et 0 -- 2 et 0 -- 3 *)
   [0]; (* 1 -- 0 *)
   [0]; (* 2 -- 0 *)
   [0]; (* 3 -- 0 *)
|];;
```

Out[84]:
```
val g3__adj : graphe_adj = [|[1; 2; 3]; [0]; [0]; [0]|]
```

In [85]:
```
deuxcoloriage g3__adj;;
```

```
visite(0)
visite(1)
visite(2)
visite(3)
visite(3)
visite(0)
visite(1)
visite(2)
```

Out[85]:
```
- : deuxcouleur array = [|Noir; Blanc; Blanc; Blanc|]
```

Et un graphe non coloriable, avec une 3-clique $0 - 1 - 2$:

In [86]:
```
let g4__adj : graphe_adj = [|
   [1; 2; 3]; (* 0 -- 1 et 0 -- 2 et 0 -- 3 *)
   [0; 2]; (* 1 -- 0 et 1 -- 2 *)
   [0; 1]; (* 2 -- 0 et 2 -- 1 *)
   [0]; (* 3 -- 0 *)
|];;
```

Out[86]:
```
val g4__adj : graphe_adj = [|[1; 2; 3]; [0; 2]; [0; 1]; [0]|]
```

In [87]:
```
deuxcoloriage g4__adj;;
```

```
visite(0)
visite(1)
visite(2)
visite(3)
visite(3)
visite(0)
visite(1)
visite(2)

Exception: Failure "2-coloriage impossible.".
Raised at file "pervasives.ml", line 32, characters 22-33
Called from file "list.ml", line 77, characters 12-15
Called from file "list.ml", line 77, characters 12-15
Called from file "list.ml", line 77, characters 12-15
Called from file "list.ml", line 77, characters 12-15
Called from file "list.ml", line 77, characters 12-15
Called from file "[81]", line 19, characters 4-75
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Cycles eulériens

Je vous laisse lire cette page (http://perso.crans.org/besson/agreg/modelisation/projet_2/) (`perso.crans.org/besson/agreg/modelisation/projet_2/`). Cherchez en ligne (https://duckduckgo.com/?q=parcours+eul%C3%A9rien+algorithme+de+rosenstielh&t=canonical&ia=web) pour plus d'informations.

`existe_cycle_eulerien`

On compte le nombre de sommets de degrés impairs, et un chemin eulérien existe si et seulement s'il y en a zéros ou deux.

In [106]:
```
let existe_cycle_eulerien (g : graphe_adj) : bool =
    let deg = Array.to_list (degres__adj g) in
    let nb_deg_impair = List.length (List.filter (fun i -> i mod 2 = 0) deg) in
    nb_deg_impair = 0 || nb_deg_impair = 2
;;
```

Out[106]:
```
val existe_cycle_eulerien : graphe_adj → bool = <fun>
```

In [107]:
```
existe_cycle_eulerien g1__adj;;
```

Out[107]:
```
- : bool = false
```

In [108]:
```
existe_cycle_eulerien g2__adj;;
```

Out[108]:
```
- : bool = false
```

In [109]:
```
existe_cycle_eulerien g3__adj;;
```

Out[109]:
```
- : bool = true
```

La suite

Pour trouver un chemin eulérien, on applique l'algorithme suivant (dû à Rosentielh, aussi attribué à Hierholzer (1873) (https://en.wikipedia.org/wiki/Eulerian_path#Hierholzer.27s_algorithm)).

- Compter le nombre de sommets de degré impair,
- Si :
  - c'est 0 : construire un cycle partant du premier sommet (choix arbitraire),
  - c'est 2 : construire un chemin entre les deux sommets en question,
  - sinon, il n'y a pas de chemin eulérien (on répond non).
- parcourir le chemin déjà construit : à chaque sommet :
  - tant qu'il reste des arêtes partant de ce sommet dans le graphe :
    - construire un cycle partant de ce sommet,
    - puis l'insérer dans le chemin.

Étant donné les conditions sur le graphe, construire un chemin ou un cycle n'est pas compliqué : il suffit de partir de l'origine, de prendre la première arête rencontrée et de recommencer récursivement.

Conclusion

Fin. À la séance prochaine. Le TP6 traitera de Lambda calcul (en février).