# Cycles - Correction

Samy Jaziri

## 1 Mise en jambe

```
(* Les graphes seront représentés sous forme de listes d'adjacence *)
type sommet = int ;;
type graphe = ( sommet list ) array ;;
type chemin = ( sommet list ) ;;
type cycle = ( sommet list ) ;;
```

Pour chercher les cycles d'un graphe, on réalise un parcours en profondeur. Si on rejoint dans le parcours un nœud déjà visité, qui n'est pas le parent du dernier nœud visité, l'algorithme vient donc de parcourir un cycle. En conservant l'historique du parcours en profondeur, on peut extraire le cycle qui a été parcouru.

```
let recherche_cycle ( g : graphe ) : cycle =
  let visite = Array.make ( Array.length g ) false in
  let rec extraire_cycle
            ( historique : chemin )
            ( depart : sommet )
            ( cycle : cycle ): cycle =
    match historique with
    | [] -> failwith "Aucun cycle trouvé."
    | i :: hist when i = depart -> depart :: cycle
    | i :: hist -> extraire_cycle hist depart ( i :: cycle ) in
  let rec parcours_composante
            ( noeud : sommet ) ( historique : chemin  ) : cycle =
    if visite.(noeud) then extraire_cycle historique noeud [noeud]
    else
      let rec parcours_voisins ( voisins : sommet list ) : cycle =
        match voisins, historique with
          | [], _ -> []
          | v :: voisins, parent :: _ when v = parent ->
            parcours_voisins voisins
          | v :: voisins, _ ->
             let cycle = parcours_composante v ( noeud :: historique ) in
             if cycle = [] then parcours_voisins voisins
             else cycle in
      begin
        visite.(noeud) <- true;
        parcours_voisins g.(noeud)
      end in
  let rec parcours ( i : sommet ) : cycle =
    if i = Array.length g then []
    else if visite.(i) then parcours ( i + 1 )
    else let cycle = parcours_composante i [] in
         if cycle = [] then parcours ( i + 1 ) else cycle in
  parcours 0
;;
```

## 2 Cycles Eulériens

**Question** Supposons que $G$ possède un cycle Eulérien :

$$C = s_0 \cdot s_1 \cdot \cdots \cdot s_{n-1} \cdot s_n$$

avec $s_n = s_0$.
Intuitivement chaque passage du cycle $C$ par $s$ implique l'utilisation d'une arête pour arriver sur $s$ et d'une arête pour sortir de $s$, ces deux arêtes étant différentes. A chaque passage le cycle passe donc par une paire d'arête différence, et comme le cycle passe par chaque arête du graphe, le degré de $s$ est paire.

Plus formellement, soit $s$ un sommet de $G$ et $E_s$ l'ensemble des arêtes dont $s$ est une extrémité. Soit $I_s = \{0 < i \leq n | s_i = s\}$. En posant $s_{n+1} = s_1$ on a, comme $C$ est Eulérien, $\forall i \neq j \in I_s, (s_{i-1}, s_i) \neq (s_{j-1}, s_j)$ et $(s_i, s_{j+1}) \neq (s_j, s_{j+1})$. Donc $E'_s = \{(s_{i-1}, s_i), i \in I_s\} \cup \{(s_i, s_{i+1})\}$ est un ensemble de cardinal $2\,|I_s|$. Or, comme $c$ est Eulérien, $E'_s = E_s$. De degré de $s$ est donc pair.
**Question** Pour vérifier qu'un cycle est Eulérien on vérifie que toutes ses arêtes sont différentes et qu'il en possède $|E|$.

```
let nombre_arete ( g : graphe ) : int =
  let nb = ref 0 in
  for i = 0 to ( Array.length g ) - 1 do
    nb := !nb + ( List.length g.(i) )
  done ;
  !nb / 2
;;
```

```
(* Prérequis : c est un cycle de g *)
let est_eulerien ( g : graphe ) ( c : cycle ) : bool =
  let n = Array.length g in
  let memo = Array.make_matrix n n false in
  let rec parcours_arete ( c : chemin ) : bool =
    match c with
    | [_] | [] -> true
    | s :: s' :: chemin when memo.(s).(s') -> false
    | s :: s' :: chemin -> begin
        memo.(s).(s') <- true;
        memo.(s').(s) <- true;
        parcours_arete ( s' :: chemin ) end in
  ( nombre_arete g ) = ( List.length c) - 1 && parcours_arete c
;;
```

**Question** Soit $C_1 = s_0 \cdot s_1 \cdot \dots \cdot s_{n-1} \cdot s_n$ et $C_2 = t_0 \cdot t_1 \cdot \dots \cdot t_{m-1} \cdot t_m$ disjoint tels que $s_i = t_j$. Le cycle $C = s_0 \cdot s_1 \cdot \dots \cdot s_i \cdot t_{j+1} \cdot \dots \cdot t_m \cdot \dots t_j \cdot s_{j+1} \cdot \dots s_n$ est un cycle regroupant les arêtes de $C_1$ et $C_2$. (*c.f schéma*)
**Question** Supposons que nous avons supprimé un certain nombre de cycles de $G$ et que $G$ possède toujours des arêtes.
Intuitivement, la suppression d'un cycle implique pour chaque passage dans un sommet du cycle la suppression de l'arête entrante dans le sommet et de l'arête sortante. Chaque arête étant unique dans le cycle, chaque sommet voit un nombre pair d'arêtes incidentes supprimée, sont degré reste pair.
Formellement, reprenons les notations ci-dessus, après suppression du cycle, $E_s \leftarrow E_s \setminus E'_s$. Comme $|E_s|$ est pair et que $|E'_s|$ est pair, le degré de $s$ reste pair.
On considère alors le sous-graphe $G'$ de $G$ obtenu en éliminant tous les sommets de $G$ qui sont isolés après la suppression des cycles. Les sommets de $G'$ sont de degré supérieur ou égal à 2, puisqu'ils sont de degré pair et qu'ils ne sont pas isolés. Donc (*c.f. cours*), $G'$ possède un cycle. Donc $G$ possède un cycle.

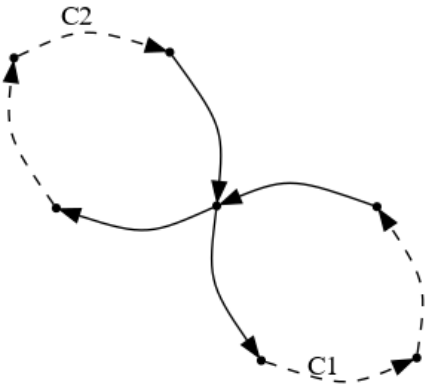


Figure 1: Fusion de cycles

**Question** On utilise la fonction de la partie 1 pour récupérer un cycle. Puis on le supprime en supprimant chaque arête.

```
let supprime_cycle ( g : graphe ) : cycle =
  let cycle = recherche_cycle g in
  let rec remove a = function
    | [] -> []
    | i :: l when i = a -> l
    | i :: l -> i :: (remove a l) in
  let elimine_arete ( s : sommet ) ( s' : sommet ) : unit =
    g.(s) <- remove s' g.(s);
    g.(s') <- remove s g.(s') in
  let rec elimine_chaque_arete ( c : chemin ) : unit =
    match c with
    | [_] | [] -> ()
    | s :: s' :: chemin -> ( elimine_arete s s';
                             elimine_chaque_arete ( s' :: chemin ) ) in
  elimine_chaque_arete cycle;
  cycle;
;;
```

**Question**

**Algorithm 1** Création de la liste de cycle distincts

1: $G' = (V', E') \leftarrow$ Copie de $G$
2: $L \leftarrow \emptyset$
3: **while** $E' \neq \emptyset$ **do**
4: $\quad C \leftarrow$ supprime_cycle$(G')$
5: $\quad$ Ajoute $C$ dans $L$
6: **end while**
7: **return** $L$

L'algorithme termine, car le nombre d'arêtes de $G'$ diminue à chaque itération de la boucle. Grâce à la preuve précédente, on sait qu'il est toujours possible de supprimer un cycle de $G'$ tant qu'il possède des arêtes. Chaque cycle est disjoint comme les arêtes du cycle précédent ont été éliminée du graphe. La suppression d'un cycle se fait en $\mathcal{O}(|E|+|S|)$. Le parcours est en $\mathcal{O}(|E|,+|S|)$ et la suppression du cycle se fait en $\mathcal{O}(\sum_{s\in V} 1+\deg(s)) = \mathcal{O}(|S|+|E|)$. Comme $G$ est connexe et que ses sommets sont de degrés pairs (donc supérieurs ou égaux à 2), on a $|E| \geq |S|$. L'algorithme final tourne donc en $\mathcal{O}(|E|^2)$.
Notons qu'avec une représentation par matrice d'adjacence, la suppression d'un cycle devient linéaire en sa taille. La recherche de cycle peut alors être faite linéairement en la taille du cycle trouvée, en pensant à supprimer les sommets isolés lors de la suppression de la dernière arête reliée au sommet. L'algorithme devient alors linéaire en le nombre d'arêtes de $G$.
**Question** On produit un algorithme qui étant donné un graphe $G$ dont chaque sommer est de degré pair, construit un cycle euclidien. On récupère une liste de cycle de $G$ disjoint contenant toutes les arêtes de $G$ par l'algorithme précédent.
Comme $G$ est connexe on peut trouver deux cycles ayant un sommet commun. En effet, sont deux sommets $s_1$ et $s_2$ appartenant à deux cycles $C_1$ et $C_2$. Il existe un chemin dans $G$ de $s_1$ à $s_2$. Si toutes les arêtes du chemin sont dans $C_1$ alors $s_2$ est un sommet commun à $C_1$ et $C_2$. Sinon il existe une arête du chemin qui n'appartient pas à $C_1$. Soit $C_3$ le cycle qui contient la première arête du chemin qui n'appartient pas à $C_1$ et $s$ le sommet de départ de cette arête. $s$ est dans $C_1$, car l'arête menant à $s$ est dans $C_1$ et $s$ est dans $C_3$ car l'arête quittant $s$ est dans $C_3$. Donc $s$ est commun à $C_1$ et $C_3$.
On peut donc joindre les deux cycles ayant un sommet commun, comme vu précédemment, pour créer un seul cycle groupant toutes leurs arêtes. La liste obtenue après fusion de ces deux cycles est toujours une liste de cycles disjoints regroupant toutes les arêtes de $G$, on peut donc itérer le processus jusqu'à obtenir un seul cycle passant par toutes les arêtes de $G$.
Nous avons donc un algorithme correct et qui termine permettant due construire pour tout graphe connexe non-orienté dont chaque sommet est de degré pair, un cycle Eulérien. Cet algorithme produit une preuve constructive de l'implication réciproque à la propriété montrée au début de cette partie :
*Un graphe connexe possède un cycle eulérien si et seulement chacun de ses sommets est de degré pair.*

```
let a_cycle_eulerien ( g : graphe ) : bool =
  let degre ( g : graphe ) ( s : sommet ) : int = List.length g.(s) in
  let rec parcours ( i : sommet ) : bool =
    if i = Array.length g then true
    else if ( degre g i ) mod 2 = 0 then parcours ( i + 1 )
    else false in
  parcours 0
;;
```

```
let cycle_eulerien ( g : graphe ) : cycle =
  if not ( a_cycle_eulerien g ) then []
  else begin
    let g' = Array.make ( Array.length g ) [] in
    for i = 0 to ( Array.length g ) - 1 do
        g'.(i) <- g.(i)
    done ;
    let cycles = ref [] in
    while nombre_arete g' != 0 do
        cycles := ( supprime_cycle g' ) :: !cycles
    done ;
```

```
let rotation_cycle ( s : sommet ) ( c : cycle ) : cycle =
    let rec apres_s = function
    | [] -> failwith "Sommet hors du cycle"
    | i :: chemin when i = s -> s :: chemin
    | _ :: chemin -> apres_s chemin in
    let rec avant_s = function
    | [] -> failwith "Sommet hors du cycle"
    | i :: chemin when i = s -> [s]
    | i :: chemin -> i :: ( avant_s chemin ) in
    match c with
    | [] -> failwith "Cycle Invalide"
    | s' :: _ when s' = s -> c
    | s' :: chemin -> ( apres_s chemin ) @ ( avant_s chemin ) in
```

```
let fusion ( c1 : cycle ) ( c2 : cycle ) ( s : sommet ) : cycle =
    let rot_c1 = rotation_cycle s c1 in
    let rot_c2 = rotation_cycle s c2 in
    match rot_c2 with
    | [] -> failwith "Cycle invalide"
    | s2 :: chemin -> rot_c1 @ chemin in
```

```
let recherche_sommet_commun ( c1 : cycle ) ( c2 : cycle ) : sommet =
    let rec parcours_cycle ( c : cycle ) ( c' : cycle ) : sommet =
    match c, c' with
    | [], _ -> -1
    | s1 :: c, [] -> parcours_cycle c c2
    | s1 :: _, s2 :: _ when s1 = s2 -> s1
    | _, _ :: c' -> parcours_cycle c c' in
    parcours_cycle c1 c2 in
```

```
let fusion_liste ( cycles : cycle list ) : cycle =
    let rec iterations
            ( c_euler : cycle )
            ( cycles1 : cycle list )
            ( cycles2 : cycle list ) : cycle =
    match cycles1, cycles2 with
    | [], [] -> c_euler
    | [], _ -> iterations c_euler cycles2 []
    | c :: cycles1, _ ->
        let s = recherche_sommet_commun c c_euler in
        if s = ( -1  ) then iterations c_euler cycles1 ( c :: cycles2 )
        else iterations ( fusion c_euler c s ) cycles1 cycles2 in
    match cycles with
    | [] -> []
    | c :: cycles -> iterations c cycles [] in
```

```
    fusion_liste !cycles
  end
;;
```

## 3 Cycles Hamiltoniens

On teste toutes les possibilités de cycles.

```
let cycle_hamiltonien ( g : graphe ) : cycle =
  let rec parcours_exhaustif
      ( s : sommet )
      ( visite : bool array )
      ( nb_visite : int )
      ( cycle : cycle ) : cycle =
    if nb_visite = ( Array.length g )
    then ( if List.mem 0 g.(s) then ( 0 :: cycle ) else [] )
    else begin
      let rec parcours_voisins = function
        | [] -> []
        | v :: voisins ->
          if visite.(v) then parcours_voisins voisins
          else begin
            let visite_v = Array.copy visite in
            visite_v.(v) <- true ;
            let c =
              parcours_exhaustif v visite_v ( nb_visite + 1 ) ( v :: cycle ) in
            if c = [] then parcours_voisins voisins else c
          end in
      parcours_voisins g.(s)
    end in
  let visite = Array.make ( Array.length g ) false in
  visite.(0) <- true;
  parcours_exhaustif 0 visite 1 [0]
;;
```

## 4 Affichage des résultats de la fiche réponse

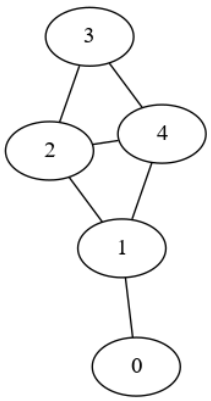


Figure 2: g1

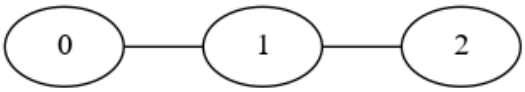


Figure 3: g2

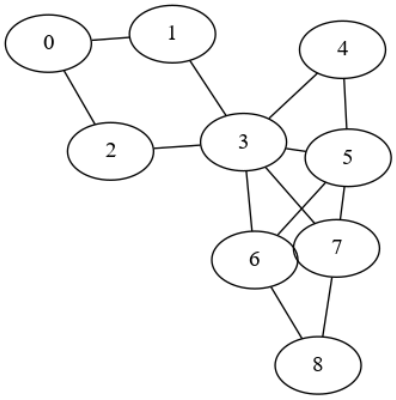


Figure 4: g3

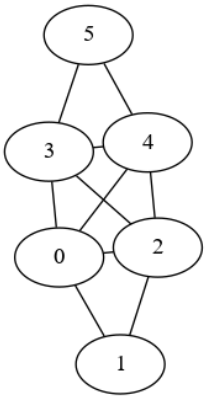


Figure 5: g4

```
let g1 : graphe =
  [|
    (* 0 *) [1 ] ;
    (* 1 *) [0 ; 2 ; 4] ;
    (* 2 *) [1 ; 3 ; 4] ;
    (* 3 *) [2 ; 4] ;
    (* 4 *) [2 ; 3 ; 1] ;
  |];;

let g2 : graphe =
  [|
    (* 0 *) [1 ] ;
    (* 1 *) [0 ; 2 ] ;
    (* 2 *) [1] ;
  |];;

let g3 : graphe =
  [|
    (* 0 *) [1 ; 2] ;
    (* 1 *) [0 ; 3] ;
    (* 2 *) [0 ; 3] ;
    (* 3 *) [1 ; 2 ; 4 ; 5 ; 6 ; 7] ;
    (* 4 *) [3 ; 5] ;
    (* 5 *) [3 ; 4 ; 6 ; 7] ;
    (* 6 *) [3 ; 5 ; 7 ; 8] ;
    (* 7 *) [3 ; 5 ; 6 ; 8] ;
    (* 8 *) [6 ; 7]
  |];;

let g4 : graphe =
  [|
    (* 0 *) [1 ; 2 ; 3 ; 4] ;
    (* 1 *) [0 ; 2] ;
    (* 2 *) [0 ; 1 ; 3 ; 4] ;
    (* 3 *) [0 ; 2 ; 3 ; 5] ;
    (* 4 *) [0 ; 2 ; 3 ; 5] ;
    (* 5 *) [3 ; 4]
  |];;
```

```
let s_cycle ( c : cycle ) : string =
  String.concat "--" ( List.map string_of_int c ) ;;

Printf.printf "Question 1 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> s_cycle ( recherche_cycle g ) )
          [g1 ; g2 ; g3 ; g4 ] ) ) ;;

Printf.printf "Question 2 : \n" ;;
List.iter
  (fun (g,c) -> Printf.printf "%s : %s\n"
                  ( s_cycle c )
                  ( if est_eulerien g c then "Oui" else "Non") )
[ (g1, [1;2;3;4;1]) ; (* Non *)
  (g3, [1;0;2;3;4;5;7;8;6;5;3;7;6;3;1]) ; (* Oui *)
  (g3, [6;5;4;3;6;5;4;3;5;7;6;5;7;8;6]) ; (* Non *)
  (g3, [6;5;7;3;6;5;4;3;5;7;6]) ; (* Non *)
  (g4, [0;1;2;3;5;4;0]) ; (* Non *)
  (g4, [0;1;2;0;4;3;2;4;5;3;0]) ; (* Oui *)
];;

Printf.printf "Question 4 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> s_cycle ( cycle_eulerien g ) )
          [g1 ; g2 ; g3 ; g4 ] ) ) ;;

Printf.printf "Question 5 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> s_cycle ( cycle_hamiltonien g ) )
          [g1 ; g2 ; g3 ; g4 ] ) ) ;;
```