exos-parcours

March 17, 2020

1 Exercices supplémentaires

```
In [1]: type sommet = int
        type graphe =
            {nb_sommets : int;
             voisins : sommet -> sommet list;
             adjacents : sommet -> sommet -> bool}

        let of_listes t =
            let n = Array.length t in
            let v i = t.(i) in
            let adj i j = List.mem j t.(i) in
            {nb_sommets = n; voisins = v; adjacents = adj}
```

```
Out[1]: type sommet = int
```

```
Out[1]: type graphe = {
          nb_sommets : int;
          voisins : sommet -> sommet list;
          adjacents : sommet -> sommet -> bool;
        }
```

```
Out[1]: val of_listes : sommet list array -> graphe = <fun>
```

On ajoute une fonction permettant d'afficher un graphe de manière plus ou moins lisible.

```
In [45]: open Printf

         let affiche ?(oriente = false) g =
             for i = 0 to g.nb_sommets - 1 do
                 let non_vide = ref false in
                 let f j =
                     if oriente then (printf "%d->%d " i j; non_vide := true)
                     else if i < j then (printf "%d-%d " i j; non_vide := true) in
```

```
            List.iter f (g.voisins i);
            if !non_vide then print_newline ()
        done;
        print_newline ()
```

```
Out[45]: val affiche : ?oriente:bool -> graphe -> unit = <fun>
```

On définit quelques graphes que l'on utilisera pour tester nos fonctions.

```
In [46]: let g51 = of_listes
             [| [1; 2];
                [];
                [1];
                [1; 5];
                [0];
                [2; 3; 4] |]

         let g52 = of_listes
             [| [2];
                [2; 6];
                [0; 1; 4; 6];
                [4; 7];
                [2; 3; 5; 7];
                [4; 6; 8];
                [1; 2; 5; 8];
                [3; 4; 9];
                [5; 6];
                [7] |]

         let g53 = of_listes
             [| [];
                [6];
                [0; 1];
                [7];
                [2; 3];
                [4];
                [2; 5; 8];
                [4; 9];
                [5];
                [] |]

         (* Le graphe g52 auquel on a ajouté les arêtes 0-3 et 3-9 *)
         let g52' = of_listes
             [| [2; 3];
                [2; 6];
                [0; 1; 4; 6];
                [0; 4; 7; 9];
```

```
            [2; 3; 5; 7];
            [4; 6; 8];
            [1; 2; 5; 8];
            [3; 4; 9];
            [5; 6];
            [3; 7] |]

        (* g52 + 0-1 + 3-9 - 4-5 *)
        let g52'' = of_listes
            [| [1; 2];
               [0; 2; 6];
               [0; 1; 4; 6];
               [4; 7; 9];
               [2; 3; 7];
               [6; 8];
               [1; 2; 5; 8];
               [3; 4; 9];
               [5; 6];
               [3; 7] |]
```

```
Out[46]: val g51 : graphe = {nb_sommets = 6; voisins = <fun>; adjacents = <fun>}
```

```
Out[46]: val g52 : graphe = {nb_sommets = 10; voisins = <fun>; adjacents = <fun>}
```

```
Out[46]: val g53 : graphe = {nb_sommets = 10; voisins = <fun>; adjacents = <fun>}
```

```
Out[46]: val g52' : graphe = {nb_sommets = 10; voisins = <fun>; adjacents = <fun>}
```

```
Out[46]: val g52'' : graphe = {nb_sommets = 10; voisins = <fun>; adjacents = <fun>}
```

Exercice 5.4

Question 1 Supposons que ϕ soit une coloration de G connexe, et fixons un $x \in G$. Pour tout sommet $y \in G$, il existe un chemin $x = x_1 \dots x_n = y$; mais il est alors immédiat que $\phi(y) = \phi(x)$ si et seulement si $n \equiv 1 \pmod 2$, et $\phi(y)$ est donc entièrement déterminé par $\phi(x)$.

Question 2 Ce que montre la question 1, c'est qu'il n'est jamais nécessaire de revenir sur un choix. Autrement dit, on peut procéder comme suit : - pour chaque sommet du graphe, on regarde s'il est déjà colorié - si c'est le cas, on l'ignore (cela signifie que sa composante connexe a déjà été traitée) ; - sinon, on lui attribue arbitrairement la couleur 0 et l'on colorie toute sa composante connexe de la seule manière possible ; - si à un moment quelconque on échoue dans la coloration de cette composante connexe, c'est que le graphe n'est pas 2-coloriable ; - si l'on arrive à traiter tous les sommets sans échec, on a obtenu une 2-coloration valable.

```
In [47]: exception Impossible

         let deux_coloration g =
             let phi = Array.make g.nb_sommets (-1) in
             let rec colore c v =
                 (* On ne fait rien si phi.(v) vaut déjà c. *)
                 if phi.(v) = -1 then (phi.(v) <- c; List.iter (colore (1 - c)) (g.voisins v))
                 else if phi.(v) <> c then raise Impossible in
             try
                 for v = 0 to g.nb_sommets - 1 do
                     if phi.(v) = -1 then colore 0 v
                 done;
                 Some phi
             with
                 | Impossible -> None

Out[47]: exception Impossible


Out[47]: val deux_coloration : graphe -> int array option = <fun>
```

On définit deux familles de graphes pour tester (sommairement) : - `k n m` représente (efficacement !) le graphe biparti complet $K_{n,m}$ (deux coloriable, bien sûr, puisque deux coloriable et biparti sont en fait synonymes) ; - `c n` (pour $n \geq 3$ représente le graphe cycle à n sommets C_n (deux coloriable si et seulement si n est pair).

```
In [48]: let k n m =
             let blancs = List.init n (fun i -> i) in
             let noirs = List.init m (fun i -> n + i) in
             {nb_sommets = n + m;
              voisins = (fun i -> if i < n then noirs else blancs);
              adjacents = (fun i j -> max i j >= n && min i j < n)}

         let c n =
             (* en Caml, (-7) mod 5 vaut -2, ce qui ne nous convient pas ici *)
             let (%) a b =
                 let m = a mod b in
                 if m >= 0 then m else m + b in
             {nb_sommets = n;
              voisins = (fun i -> [(i + 1) % n; (i - 1) % n]);
              adjacents = (fun i j -> j = (i + 1) % n || j = (i - 1) % n)}

         let () =
             printf "K_{3, 5}\n";
             affiche (k 3 5);
             printf "C_4\n";
             affiche (c 4);
```

```
            printf "C_5\n";
            affiche (c 5)
```

```
Out[48]: val k : sommet -> int -> graphe = <fun>
```

```
Out[48]: val c : int -> graphe = <fun>
```

```
K_{3, 5}
0-3 0-4 0-5 0-6 0-7
1-3 1-4 1-5 1-6 1-7
2-3 2-4 2-5 2-6 2-7

C_4
0-1 0-3
1-2
2-3

C_5
0-1 0-4
1-2
2-3
3-4
```

```
In [5]: deux_coloration (k 3 5)
```

```
Out[5]: - : int array option = Some [|0; 0; 0; 1; 1; 1; 1; 1|]
```

```
In [6]: deux_coloration (c 10)
```

```
Out[6]: - : int array option = Some [|0; 1; 0; 1; 0; 1; 0; 1; 0; 1|]
```

```
In [7]: deux_coloration (c 9)
```

```
Out[7]: - : int array option = None
```

Si l'on ne souhaite pas utiliser d'exception, on peut procéder joliment :

```
In [8]: let deux_coloration_bis g =
            let phi = Array.make g.nb_sommets (-1) in
            let rec colore_sommet c v =
                if phi.(v) = -1 then (phi.(v) <- c; colore_voisins (1 - c) (g.voisins v))
                else phi.(v) = c
            and colore_voisins c = function
```

```
            | [] -> true
            | v :: vs -> colore_sommet c v && colore_voisins c vs in
        let rec loop i =
            if i = g.nb_sommets then Some phi
            else if phi.(i) <> -1 || colore_sommet 0 i then loop (i + 1)
            else None in
        loop 0
```

```
Out[8]: val deux_coloration_bis : graphe -> int array option = <fun>
```

ou plus simplement :

```
In [9]: let deux_coloration_ter g =
            let n = g.nb_sommets in
            let phi = Array.make n (-1) in
            let ok = ref true in
            let rec colore c v =
                if phi.(v) = 1 - c then ok := false
                else if phi.(v) = -1 then (phi.(v) <- c; List.iter (colore (1 - c)) (g.voisins
            let v = ref 0 in
            while !v < n && !ok do
                if phi.(!v) = -1 then colore 0 !v;
                incr v
            done;
            if !v = n then Some phi else None
```

```
Out[9]: val deux_coloration_ter : graphe -> int array option = <fun>
```

Remarque : dans le sujet X-ENS 2018, cette question était posée mais l'on pouvait supposer que le graphe donné était deux coloriable (et renvoyer n'importe quoi sinon). Cela enlevait la majorité des difficultés de programmation rencontrées ici.

1.1 Exercice 5.5

Question 1 On peut effectuer $|V|$ parcours en profondeur indépendants, à partir de chaque sommet, et vérifier que l'on atteint bien tous les sommets à chaque fois. Chaque parcours se fait en temps $\mathcal{O}(|E| + |V|)$, d'où une complexité en $\mathcal{O}(|E| \cdot |V| + |V|^2)$.

Question 2 On choisit un sommet quelconque $x \in G$, puis : - on effectue un parcours en profondeur de G depuis x. Si l'on n'atteint pas tous les sommets, G n'est pas fortement connexe, sinon on passe à l'étape suivante ; - on effectue un parcours en profondeur de $G^{\leftarrow}$, toujours depuis x. Si l'on n'atteint pas tous les sommets, G n'est pas fortement connexe. Si l'on atteint tous les sommets, on conclut que G est fortement connexe. En effet, pour tous sommets $(y, z) \in V^2$: - il existe un chemin $y \to \cdots \to x$ d'après la deuxième étape ; - il existe un chemin $x \to \cdots \to z$ d'après la première étape ; - en les concaténant, on obtient un chemin $y \to \cdots \to z$.

Remarque : on a admis l'équivalence, évidente, entre ń x est accesible depuis y dans G ż et ń y est accessible depuis x dans $G^{\leftarrow}$ ż. Elle se prouve au besoin par récurrence sur la longueur du chemin.

Question 3 Le calcul de $G^{\leftarrow}$ se fait facilement en temps linéaire en la taille du graphe, tout comme chacun des deux parcours en profondeur. On a donc une complexité temporelle en $\mathcal{O}(|V| + |E|)$.

Question 4 On est obligé de construire explicitement les listes d'adjacence du graphe miroir pour avoir la complexité souhaitée.

```
In [10]: let miroir g =
             let n = g.nb_sommets in
             let t = Array.make n [] in
             (* traite_arc i j ajoute l'arc (j, i) dans le graphe miroir *)
             let traite_arc i j = t.(j) <- i :: t.(j) in
             for i = 0 to n - 1 do
                 List.iter (traite_arc i) (g.voisins i)
             done;
             of_listes t

Out[10]: val miroir : graphe -> graphe = <fun>


In [11]: let fortement_connexe g =
             let g' = miroir g in
             (* renvoie true ssi tous les sommets de g sont accessibles depuis 0 *)
             let tous_accessibles g =
                 let vus = Array.make g.nb_sommets false in
                 let rec explore v =
                     if not vus.(v) then begin
                         vus.(v) <- true;
                         List.iter explore (g.voisins v)
                     end in
                 explore 0;
                 Array.fold_left (&&) true vus in
             tous_accessibles g && tous_accessibles g'

Out[11]: val fortement_connexe : graphe -> bool = <fun>


In [12]: fortement_connexe g51

Out[12]: - : bool = false
```

1.2 Exercice 5.6

Question 1 Remarquons d'abord que s'il existe un cycle passant par une arête e, alors tout cycle de longueur minimale passant par e est élémentaire (je vous laisse le démontrer). Ainsi, il existe un cycle passant par e si et seulement si il existe un cycle élémentaire passant par e. On conclut alors à l'aide de la proposition **4.15**.

Question 2 Pour clarifier, on note G le graphe non orienté initial et G' sa version orientée comme dans l'énoncé. Comme G est connexe, T est un arbre (enraciné en r) contenant tous les sommets de G'. Donc tout sommet de G' est accessible depuis r. Supposons à présent que G' ne soit pas fortement connexe, et soit alors s de profondeur minimale dans T tel que r ne soit pas accessible depuis s. s ne peut être égal à r et possède donc un père v : notons e l'arête vs de G. Cette arête est un pont : en effet, si elle faisait partie d'un cycle dans G, il y aurait nécessairement dans G' un arc arrière reliant s ou l'un de ses descendants à v ou l'un de ses ancêtres. Mais v serait alors accessible depuis s, et donc r aussi par minimalité de la profondeur de v.

Question 3 On vient de montrer un sens. Inversement, si xy est un pont dans G, alors tout chemin

Question 4 Identifier un pont demande un peu de travail : dans un premier temps, on va se contenter de déterminer si le graphe est fortement orientable, et de ne renvoyer une orientation forte s'il en existe. Pour cela, il faut effectuer un parcours en profondeur en orientant les arêtes correctement. Quand on suit une arête d'un sommet u vers un sommet v, trois cas sont possibles : - v est vierge : dans ce cas, l'arête se transforme en arc *arbre* $u \to v$; - v a été vu, mais on n'a pas suivi l'arête vu (ce qui implique que l'exploration de v n'est pas terminé, et donc que u est un descendant de v) : dans ce cas, l'arête se transforme en arc *arrière* $u \to v$; - v a été vu,

```
In [13]: let fortement_orientable g =
             let open Hashtbl in
             let suivis = create g.nb_sommets in
             let vus = Array.make g.nb_sommets false in
             let g' = Array.make g.nb_sommets [] in
             let rec explore u =
                 vus.(u) <- true;
                 let f v =
                     if not vus.(v) then begin
                         add suivis (u, v) ();
                         explore v
                     end else if not (mem suivis (v, u)) then
                         add suivis (u, v) ()
                     in
                 List.iter f (g.voisins u) in
             explore 0;
             iter (fun (u, v) () -> g'.(u) <- v :: g'.(u)) suivis;
             g'
             (* fortement_connexe (of_listes g') *)

         let f g =
             let vus = Array.make g.nb_sommets false in
             let fermes = Array.make g.nb_sommets false in
             let g' = Array.make g.nb_sommets [] in
             let ajoute u v = g'.(u) <- v :: g'.(u) in
             let rec explore o d =
                 vus.(d) <- true;
```

```
            let f v =
                if not vus.(v) then
                    (ajoute d v; explore d v)
                else if not fermes.(v) && v <> o then
                    ajoute d v in
            List.iter f (g.voisins d);
            fermes.(d) <- true in
        explore (-1) 0;
        g'
```

```
Out[13]: val fortement_orientable : graphe -> sommet list array = <fun>
```

```
Out[13]: val f : graphe -> sommet list array = <fun>
```

```
In [14]: fortement_orientable g52'
```

```
Out[14]: - : sommet list array =
         [|[2]; [6]; [1]; [7; 0]; [2; 3]; [4; 8]; [5; 2]; [4; 9]; [6]; [3]|]
```

```
In [15]: f g52'
```

```
Out[15]: - : sommet list array =
         [|[2]; [6]; [1]; [7; 0]; [3; 2]; [8; 4]; [5; 2]; [9; 4]; [6]; [3]|]
```

```
In [27]: type t = Orientation of sommet list array | Pont of sommet * sommet

         let orientation_forte g =
             let prefixe = Array.make g.nb_sommets (-1) in
             let pere = Array.make g.nb_sommets (-1) in
             let fermes = Array.make g.nb_sommets false in
             let g' = Array.make g.nb_sommets [] in
             let ajoute u v = g'.(u) <- v :: g'.(u) in
             let i = ref 0 in
             let marque v = prefixe.(v) <- !i; incr i in
             let rec explore d =
                 marque d;
                 let f v =
                     if prefixe.(v) = -1 then
                         (pere.(v) <- d; ajoute d v; explore v)
                     else if not fermes.(v) && pere.(d) <> v then
                         ajoute d v in
                 List.iter f (g.voisins d);
                 fermes.(d) <- true in
             explore 0;
             let gmiroir = miroir (of_listes g') in
```

```
            let vus = Array.make g.nb_sommets false in
            let rec dfs u =
                if not vus.(u) then begin
                    vus.(u) <- true;
                    List.iter dfs (gmiroir.voisins u)
                end in
            dfs 0;
            let pref_min = ref max_int in
            let i_min = ref None in
            for i = 0 to g.nb_sommets - 1 do
                if not vus.(i) && prefixe.(i) < !pref_min then
                    (pref_min := prefixe.(i); i_min := Some i)
            done;
            match !i_min with
            | None -> Orientation g'
            | Some i -> Pont (i, pere.(i))
```

```
Out[27]: type t = Orientation of sommet list array | Pont of sommet * sommet


Out[27]: val orientation_forte : graphe -> t = <fun>
```

```
In [29]: orientation_forte g52

Out[29]: - : t = Pont (2, 0)
```

```
In [30]: orientation_forte g52'

Out[30]: - : t =
         Orientation
          [|[2]; [6]; [1]; [7; 0]; [3; 2]; [8; 4]; [5; 2]; [9; 4]; [6]; [3]|]
```

```
In [31]: orientation_forte g52''

Out[31]: - : t = Pont (4, 2)
```