Feuille de révisions n°6 - Programmation en OCaml : manipulation de graphes

Dans toute cette feuille de révision on suppose définis les types OCaml suivants, permettant respectivement la représentation de graphes (orientés ou non) par table de listes d'adjacences, par matrice d'adjacence.

```
type graphe_adj =  (* graphes par table de listes d'adjacence *)
  int list array

type graphe_mat =  (* graphes par matrice d'adjacence *)
  bool array array
```

1 Pied à l'étrier

R. 6-1 Définir une fonction OCaml prenant en arguments un graphe g (représenté par matrice d'adjacence), deux sommets entiers i et j et retournant si l'arête $\{i, j\}$ est une arête du graphe g.

Solution

```
let est_arete_mat (g: graphe_mat) (i: int) (j: int): bool =
  g.(i).(j)
```

R. 6-2 Définir une fonction OCaml prenant en arguments un graphe g (représenté par table de listes d'adjacence), deux sommets entiers i et j et retournant si l'arête $\{i, j\}$ est une arête du graphe g.

Solution

```
let est_arete (g: graphe_adj) (i: int) (j: int): bool =
  List.mem j g.(i)
```

R. 6-3 Définir une fonction OCaml prenant en arguments deux graphes (représentés par matrice d'adjacence) et testant s'ils sont égaux.

Solution

```
let sont_egaux_mat (g1: graphe_mat)(g2: graphe_mat) : bool =
  g1 = g2
```

R. 6-4 Définir une fonction OCaml prenant en arguments deux graphes (représentés par table de listes d'adjacence)et testant s'ils sont égaux.

Solution

```
exception NotEqual

let sont_egaux_adj (g1: graphe_adj)(g2: graphe_adj) : bool =
```

```
  let n1 = Array.length g1 in
  let n2 = Array.length g2 in
  if (n1 <> n2) then false
  else try
    for i = 0 to n1 - 1 do
      for j = 0 to n1 - 1 do
        if est_arete g1 i j <> est_arete g2 i j
        then raise NotEqual
        else ()
      done
    done; true
    with NotEqual -> false
```

R. 6-5 Définir une fonction OCaml prenant en arguments un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arcs sont les éléments de `l`. On peut supposer que `l` est sans doublons. On déclenchera une erreur si `l` contient un couple qui ne peut être un arc possible de ce graphe.

Solution

```
let fabrique_graphe_o (n: int) (l: (int * int) list): graphe_adj =
  let rep = Array.make n [] in
  List.iter (fun (i, j) ->
    if ( i < 0 or i >= n or j < 0 or j >= n or i = j )
    then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
      in failwith (cij ^ " n'est pas un arc possible")
    else rep.(i) <- j :: rep.(i)
  ) l;
  rep
```

R. 6-6 Définir une fonction OCaml prenant en arguments un entier `n` et une liste `l` de couples d'entiers et retournant le graphe non orienté (représenté par table de listes d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arêtes sont données par les couples de `l`. On peut supposer que les couples de `l` représentent deux à deux des arêtes différentes. On déclenchera une erreur si `l` contient un couple qui ne correspond pas à une arête possible de ce graphe.

Solution

```
let fabrique_graphe_no (n: int) (l: (int * int) list): graphe_adj =
  let rep = Array.make n [] in
  List.iter (fun (i, j) ->
    if ( i < 0 or i >= n or j < 0 or j >= n or i = j )
    then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
      in failwith (cij ^ " n'est pas une arête possible")
    else (rep.(i) <- j :: rep.(i) ; rep.(j) <- i :: rep.(j))
  ) l;
  rep
```

R. 6-7 Définir une fonction OCaml prenant en arguments un entier `n` et une liste `l` de couples d'entiers et retournant le graphe orienté (représenté par matrice d'adjacence) dont les sommets sont les entiers de l'intervalle $[\![0, n-1]\!]$ et les arcs sont données par les éléments de `l`.

Solution

```
let fabrique_graphe_o_mat (n: int) (l: (int * int) list): graphe_mat =
  let mat = Array.make_matrix n n false in
  List.iter (fun (i, j) ->
    if ( i < 0 or i >= n or j < 0 or j >= n or i = j )
      then let cij= "("^(string_of_int i)^", "^(string_of_int j)^")"
        in failwith (cij ^ " n'est pas un arc possible")
      else mat.(i).(j) <- true
    ) l;
  mat
```

R. 6-8 Définir une fonction OCaml prenant en argument un graphe orienté (représenté par table de listes d'adjacence) et retournant la liste de ses arcs.

Solution

```
let liste_arcs (g: graphe_adj): (int * int) list =
  let x, _ = Array.fold_left (fun (accu, i) vois_i ->
      let new_acc = List.fold_left (fun acc j ->
          (i, j) :: acc
        ) accu vois_i in
      (new_acc, i+1)
    ) ([], 0) g in
  x
```

R. 6-9 Définir une fonction OCaml prenant en argument un graphe non orienté (représenté par table de listes d'adjacence) et retournant la liste de ses arêtes représentées par des couples (i, j) avec $i < j$.

Solution

```
let liste_aretes (g: graphe_adj): (int * int) list =
  let x, _ = Array.fold_left (fun (accu, i) vois_i ->
      let new_acc = List.fold_left (fun acc j ->
          if i< j then (i, j) :: acc else acc
        ) accu vois_i in
      (new_acc, i+1)
    ) ([], 0) g in
  x
```

R. 6-10 Définir une fonction OCaml prenant en argument un graphe orienté (représenté par matrice d'adjacence) et retournant la liste de ses arcs.

Solution

```
let liste_arcs_mat (g: graphe_mat): (int * int) list =
  let x, _ = Array.fold_left (fun (acc, i) t ->
```

```
      let acc3, _ = Array.fold_left (fun (acc, j) b ->
          let acc2 = if b then (i, j) :: acc else acc in
          (acc2, j+1)
        ) (acc, 0) t in
      (acc3, i+1)
    ) ([], 0) g
```

R. 6-11 Définir une fonction OCaml prenant en argument un graphe orienté g (représenté par table de listes d'adjacence) et retournant le graphe non orienté (représenté par table de listes d'adjacence) ayant même ensemble de sommets et contenant une arrête $\{x, y\}$ si et seulement si le graphe g contient l'arc (x, y) ou l'arc (y, x).

Solution

```
let gno_from_go (g : graphe_adj) : graphe_adj =
  let n = Array.length g in
  let liste_a = liste_arcs g in
  fabrique_graphe_no n liste_a
```

R. 6-12 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en arguments un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^4)$.

Solution

```
let produit_matriciel (t1: bool array array) (t2: bool array array): bool
↪  array array =
  let n = Array.length t1 in
  let res = Array.make_matrix n n false in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      for k = 0 to (n-1) do
        if t1.(i).(k) && t2.(k).(j) then res.(i).(j) <- true
      done
    done
  done; res

let somme_matricielle (t1: bool array array) (t2: bool array array): bool
↪  array array =
  let n = Array.length t1 in
  let res = Array.make_matrix n n false in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      res.(i).(j) <- t1.(i).(j) || t2.(i).(j)
    done
  done; res

let accessibilite (g: graphe_mat): bool array array =
```

```
  let n = Array.length g in
  let res = ref g in
  for i = 1 to n do
    res := somme_matricielle !res (produit_matriciel !res g)
  done;
  !res
```

R. 6-13 La matrice d'accessibilité d'un graphe g non orienté à n sommets est une matrice de booléens de dimension $n \times n$ contenant `true` dans la case d'indice (i, j) si et seulement s'il existe un chemin du sommet i au sommet j dans le graphe. Définir une fonction prenant en arguments un graphe (représenté par matrice d'adjacence) et retournant sa matrice d'accessibilité. On s'autorise une complexité en $\mathcal{O}(n^3)$, on pourra pour cela s'aider de l'algorithme de Roy-Warshall.

Solution

```
let roy_warshall (g: graphe_mat): bool array array =
  let n = Array.length g in
  let old = ref g in
  (*inv : pr ts i,j old.(i).(j) indique s'il existe une chaîne entre
    i et j ds g dont les sommets intermédiaires sont dans [0..k[ *)
  let cur = ref (Array.make_matrix n n false) in
  for k = 1 to n do
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        !cur.(i).(j) <- !old.(i).(j) || ( !old.(i).(k) && !old.(k).(j) )
      done
    done;
    old := !cur
  done;
  !old
```

2 Parcours de graphes

Dans cette partie les graphes seront tous représentés par table de listes d'adjacence (*i.e.* avec le type `graphe_adj`).

R. 6-14 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser la pile des appels récursifs, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n + m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

version complètement récursive avec un générateur de sommet non visités

```
(* On crée un générateur de sommet non envore visité afin que la
   complexité cumulée des recherches de points de régénération soit
   en O(n) et pas en O(n^2)*)
let cree_generateur_sommets (n: int) : (int ref * (bool array -> unit)) =
  let k = ref 0 in              (* joue le rôle de curseur *)
  let next (visited: bool array) : unit =
```

```
    if (!k < n) then incr k;    (* forcer l'incrémentation *)
    while (!k < n && visited.(!k)) do
      incr k
    done;
  in (k, next)

let parcours_full_rec (g: graphe_adj): int list =
  let n       = Array.length g in
  let visites = Array.make n false in
  let rec explore_descendants (acc: int list) (s: int): int list =
    if not (visites.(s)) then
      let () = visites.(s) <- true in
      List.fold_left explore_descendants (s :: acc) g.(s)
    else acc
  in
  let s, next = cree_generateur_sommets n in
  let rec aux (acc: int list): int list =
    if !s >= n then List.rev acc
    else
      let acc = explore_descendants acc !s in
      next visites; aux acc
  in aux []

```

version avec pile d'appels récursifs mais exploration qui agit par effet de bord

```
let parcours_prof_rec (g: graphe_adj): int list =
  let n       = Array.length g in
  let visites = Array.make n false in
  let res = ref [] in
  let rec explore_descendants (s: int): unit =
  (* explore les descendants de s dans g non encore visités
     met à jour le tableau visités et complete res de sorte que
     List.rev !res soit un parcours en prof du sous-graphe de g
     induit par les sommets visités *)
    if not (visites.(s)) then
      begin
        visites.(s) <- true;
        res := s::!res;
        List.iter explore_descendants g.(s)
      end
    else ()
  in
  for s = 0 to  n - 1 do
    explore_descendants s
  done;
  List.rev !res
```

R. 6-15 Définir une fonction OCaml calculant un parcours en profondeur (une permutation des

sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Stack`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

```
let parcours_prof_avec_todo (g: graphe_adj): int list =
  (* nombre de sommets dans le graphe *)
  let n       = Array.length g in
  (* parcours que nous sommes en train de fabriquer *)
  let l       = ref [] in
  (* ensemble des visités, représenté par un tableau indicateur :
     i est visité, ssi visité.(i) = true *)
  let visites = Array.make n false in
  let explore_descendants (s: int) =
    (* todo pour stocker les sommets à traiter *)
    let todo = Stack.create () in
    (* on ajoute s dans la pile des choses à traiter *)
    Stack.push s todo;
    while (not (Stack.is_empty todo)) do
      (* on extrait un élément dans la pile des choses à traiter *)
      let v = Stack.pop todo in
      (* on teste si v a déjà été visité *)
      if not (visites.(v)) then
        (* si ce n'est pas le cas on le visite *)
        begin
          (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
          visites.(v) <- true;
          (* on ajoute les successeurs de v dans todo *)
          List.iter (fun y -> Stack.push y todo) g.(v);
          (* on ajoute v au parcours *)
          l := v :: !l
        end
    done
  in
  for s = 0 to n - 1 do
    if not visites.(s) then explore_descendants s else ()
  done;
  List.rev !l
```

R. 6-16 Définir une fonction OCaml calculant un parcours en largeur (une permutation des sommets, de type `int list`) du graphe passé en argument. On s'efforcera d'utiliser le module `Queue`, pour stocker les éléments encore à traiter. La complexité de l'implémentation devra être en $\mathcal{O}(n+m)$ où n est le nombre de sommets du graphe et m est le nombre d'arêtes.

Solution

```
let parcours_largeur_avec_todo (g: graphe_adj): int list =
  (* nombre de sommets dans le graphe *)
  let n       = Array.length g in
  (* parcours que nous sommes en train de fabriquer *)
```

```
  let l       = ref [] in
  (* ensemble des visités, représenté par un tableau indicateur :
     i est visité, ssi visité.(i) = true *)
  let visites = Array.make n false in
  let explore_descendants (s: int) =
    (* todo pour stocker les sommets à traiter *)
    let todo = Queue.create () in
    (* on ajoute s dans la pile des choses à traiter *)
    Queue.push s todo;
    while (not (Queue.is_empty todo)) do
      (* on extrait un élément dans la pile des choses à traiter *)
      let v = Queue.pop todo in
      (* on teste si v a déjà été visité *)
      if not (visites.(v)) then
        (* si ce n'est pas le cas on le visite *)
        begin
          (* on l'ajoute aux visités pour ne pas le revisiter plus tard *)
          visites.(v) <- true;
          (* on ajoute les successeurs de v dans todo *)
          List.iter (fun y -> Queue.push y todo) g.(v);
          (* on ajoute v au parcours *)
          l := v :: !l
        end
    done
  in
  for s = 0 to n - 1 do
    if not visites.(s) then explore_descendants s else ()
  done;
  List.rev !l
```