PARCOURS DE GRAPHES

1 Graphes (rappels)

Un graphe est un couple (X, E) où X est un ensemble fini quelconque dit ensembles de *sommets* et E un ensmble de couples d'éléments de X ou une ensemble de paires[1] d'élément de X suivant que le graphe est *orienté* ou non.

Pour tout $x \in X$, on définit l'ensemble, noté $Adj(x, G)$, des sommets *adjacents à x dans G*

orienté $Adj(x, G) = \{y \in X | (x, y) \in E\}$

non orienté $Adj(x, G) = \{y \in X | \{x, y\} \in E\}$

2 Parcours en profondeur d'abord

On considère, dans un premier temps un parcours en profondeur à partir d'un certain sommet $s \in X$.

On suppose disposer sur les éléments de X d'une fonction booléenne Vu. On peut modifier, en un point x la valeur de cette fonction par $Voir(x)$, et alors $Vu(x)$ est vrai, ou par $Cacher(x)$, et alors $Vu(x)$ est faux.

Soient P une pile et L une liste.

```
PROC PP(s,G):
   POUR TOUT u ∈ X FAIRE Cacher(u) FPOUR;
   P ← (s);
   Voir(s);
   L ← ∅;
   TANT QUE P ≠ ∅ FAIRE
      u ← Tete(P);
      SI IL EXISTE v ∈ Adj(u) TEL QUE ¬Vu(v) ALORS
         Empiler(v,P);  Voir(v);
      SINON
         Depiler(P);  Ajouter(u,L);
      FSI
   FTANTQUE
   Retourner(L)
FPROC
```

Nous proposons ci-dessous une implémentation en camllight de cet algorithme :

La fonction Vu

```
let vu_list = ref ([]:int list);;

let init_vu() = vus:=[];;

let voir s = vus := s::!vus;;

let vu s = List.mem s !vus;;
```

[1] dans un couple l'ordre des éléments importe mais pas dans une paire. De plus, une paire peut ne contenir qu'un seul élément. Les couples sont notés (x, y) et si $x \neq y$ alors $(x, y) \neq (y, x)$ Les paires sont notées $\{x, y\}$ et $\{x, x\} = \{x\}$ et $\{x, y\} = \{y, x\}$

Les piles

```
type 'a stack = ('a list) ref;;

let new_stack () = ((ref []):'a stack);;

let push a (s:'a stack) = s:=a::!s;;

exception Empty_stack;;

let top (s:'a stack) =
 try List.hd !s with _ -> raise Empty_stack;;

let pop (s:'a stack) =
 try s := List.tl !s with _ -> raise Empty_stack;;

let is_empty (s:'a stack) = (!s=[]) ;;
```

Le programme

```
(* -- find : selectionne le 'x' element d'une liste qui
             satisfait le critere 'c'                    *)
let rec find c = function
 [] -> raise Not_found
|x::xs -> if (c x) then x else find c xs
;;

(* -- adjs : fournit la liste des sommets adjacents a
             sont premier argumant                       *)
(* NB : son implementation depend de la representation
   du graphe; nous ne la precisons pas ici              *)

(* -- Le programme *)
let pp s g =
 let l = ref ([]:int list) in
 let sp = new_stack() in
   init_vu();
   push s sp;
   voir s;
   while not(is_empty sp) do
    let u = top sp in
     (try
      let v = find (fun v -> not (vu v)) (adjs u g) in
       begin push v sp; voir v end
     with Not_found -> pop sp; l := u::!l)
   done;
   !l
;;
```

L'algorithme PP ne donne un parcours du graphe G que si G est (fortement) connexe. Pour obtenir un parcours dans le cas général, il faut itérer PP sur l'ensemble des sommets. On parcours ainsi toutes les composantes connexes. On ne peut plus alors réeelemnt parler d'un parcours à partir d'un sommet donné.

PROC $GPP(G)$:

```
POUR TOUT u ∈ X FAIRE Cacher(u) FPOUR;
P ← ∅;
L ← ∅;
POUR TOUT s ∈ X FAIRE
   Voir(s);
   Empiler(s, P);
   TANT QUE P ≠ ∅ FAIRE
      u ← Tete(P);
      SI IL EXISTE v ∈ Adj(u) TEL QUE ¬Vu(v) ALORS
         Empiler(v, P);  Voir(v);
      SINON
         Depiler(P);  Ajouter(u, L);
      FSI
   FTANTQUE;
FPOUR;
Retourner(L);
FPROC
```

On peut, trés facilement donner une version récursive d'un algorithme de parcours en profondeur. La pile explicitement gérée dans PP devient la pile (implicite) des appels récursifs.

```
PROC RPP(s, G, L) :
Voir(s);
POUR TOUT u ∈ Adj(s) FAIRE
   SI ¬Vu(u) ALORS RPP(u, G, L) FSI;
FPOUR;
Ajouter(s, L);
FPROC
```

On obtient alors l'algorithme général

```
PROC GRPP(s, G, L) :
L ← ∅;
POUR TOUT s ∈ X FAIRE Cacher(s) FPOUR;
POUR TOUT s ∈ X FAIRE GRPP(s, G, L) FPOUR;
Retourner(L);
FPROC
```

3 Parcours en largeur

Soit F une file.

```
PROC PL(s, G) :
POUR TOUT u ∈ X FAIRE Cacher(u) FPOUR;
F ← (s);
Voir(s);
L ← (s);
TANT QUE F ≠ ∅ FAIRE
   u ← Tete(F);
   POUR TOUT v ∈ Adj(u) TEL QUE ¬Vu(v) FAIRE
      Enfiler(v, F);  Voir(v);  Ajouter(v, L);
   FPOUR;
   Defiler(F);
FTANTQUE
```

12. *Retourner*(L)
FPROC

Les files

```
type 'a fifo = ('a list) ref;;

let new_fifo () = ((ref []):'a fifo);;

let is_empty (f:'a fifo) = (!f=[]);;

let in_fifo a f = f:=a::!f;;

exception Empty_fifo;;

let hd_fifo (f:'a fifo) =
 try List.hd (List.rev !f) with _ -> raise Empty_fifo
;;

let out_fifo (f:'a fifo) =
 try f := List.rev(List.tl(List.rev !f)) with _ -> raise Empty_fifo
;;
```

Le programme

```
let pl s g =
 let l = ref ([]:int list) in
 let f = new_fifo() in
  init_vu();
  in_fifo s f;
  voir s;
  l := s::!l;
  while not(is_empty f) do
   let u = hd_fifo f in
   List.iter
    (fun v
     -> if not(vu v) then (voir v; l := v::!l; in_fifo v f))
    (adjs u g);
   out_fifo f
  done;
  !l
;;
```