(Option informatique-2ème année)

Corrigé Colle d'info n°4

Quelques algorithmes classiques sur les graphes

```
type graphe == (int * int list) list ;;
let (g : graphe) = [ 1,[2;3] ; 2,[1;4;5] ; 3,[1] ; 4,[2] ;
5,[2] ; 6,[7] ; 7,[6] ];;
```

Exercice n°1 : Parcours en profondeur

On étudie chacun des sommets du graphe et pour chaque sommet, on commence par explorer ses fils.

```
2       (* Parcours en profondeur :
3       (* g : graphe à explorer *)
4       let profondeur (g : graphe) =
5       (* g : graphe *)
6       (* vu = liste des sommets déjà visité *)
7       (* troisième argument : liste des sommets à visiter *)
8       let rec parcours g vu = function
9           [] -> vu
10        | x::xs when mem x vu -> parcours g vu xs
11        | x::xs -> parcours g (suivant g (vu@[x]) x) xs
12      and
13        suivant g vu x =
14           (* assoc x g retourne la liste des sommets reliés au sommet x *)
15           parcours g vu (assoc x g)
16        in
17      let gg = ref g and liste = ref []in
18      (* on parcours tout le graphe pour être sûr de ne pas oublier un sommet *)
19      (* liste = liste des sommets parcouru en profondeur *)
20      while !gg <> [] do
21         let (x,l) = hd !gg in
22           if not(mem x !liste) then liste := parcours g (!liste@[x]) l;
23           gg := tl !gg;
24    done;
25    !liste;;
```

Voici une deuxième version plus sobre utilisant la fonction «`it_list`» de Caml

```
26      let (profondeur' : graphe -> int list) = function g ->
27        let rec traite_sommet a l =
28                              it_list (fun acc s' -> if mem s' acc then acc else
29                    traite_sommet (s'::acc) (assoc s' g)) a l
30        and traite_graphe a l =
31                              it_list (fun acc (s,al) -> if mem s acc then acc else
32    traite_sommet (s::acc) al) a l
33        in rev (traite_graphe [] g)
34      ;;
```

Exercice n°2 : Parcours en largeur

On utilise une file d'attente dans laquelle on stocke les sommets à visiter.

```
35      (* g : graphe à explorer *)
36      (* accu : liste référencée des sommets à visiter dans l'ordre de visite *)
37      (* vu : liste référencée des sommets visités dans leur ordre de visite *)
38      let rec largeur (g : graphe) =
39        let gg = ref g and vu = ref [] and accu = ref [] in
40        let rec ajoute accu vu = function
41          [] -> ()
42          | x::xs when mem x vu -> ajoute accu vu xs
43          | x::xs  -> accu := !accu@[x]; ajoute accu vu xs
44        in
45        let rec parcours g vu accu =
46         match !accu with
47            [] -> vu
48          | x::xs when mem x vu -> accu := xs; parcours g vu accu;
49          | x::xs -> ajoute accu vu (assoc x g) ;
50                              parcours g (vu@[x]) accu;
51          in
52       while !gg <> [] do
53            let (x,l) = hd !gg in
54                      if not(mem x !vu) then (ajoute accu !vu l; vu := parcours g (!vu@[x]) accu;);
55              gg := tl !gg;
56       done;
57       !vu;;
```

Si le sommet «`x`» n'a pas encore été visité, on ajoute dans «`accu`» les sommets qui lui sont reliés, on ajoute «`x`» aux sommets visités et on continue notre parcours en largeur.

Exercice n°3 : Composantes connexes.

On étudie chacun des sommets du graphe et pour chaque sommet, on commence par explorer ses fils.

```
58     (* g : graphe à explorer *)
59     let composante (g : graphe) =
60        (* g : graphe *)
61        (* vu = liste des sommets déjà visité *)
62        (* troisième argument : liste des sommets à visiter *)
63        let rec parcours g vu = function
64            [] -> vu,[]
65          | x::xs when mem x vu -> parcours g vu xs
66          | x::xs -> let (v,l) = suivant g (vu@[x]) x in
67                     let (v',l') = parcours g v xs in (v',x::l@l')
68        and
69          suivant g vu x =
70             (* assoc x g retourne la liste des sommets reliés au sommet x *)
71             parcours g vu (assoc x g)
72          in
73        let gg = ref g and composantes = ref [] and vu = ref[] in
74        (* on parcours tout le graphe et pour chaque sommet non encore visité, *)
75        (* on calcule sa composante connexe et on l'ajoute *)
76        (* vu = liste des sommets déjà vus *)
77        while !gg <> [] do
78           let (x,l) = hd !gg in
79             if not(mem x !vu)
80                 then
81                   (let (vu',composante) = parcours g (!vu) (x::l) in
82                         vu := vu';
83                         composantes := composante::!composantes;);
84             gg := tl !gg;
85       done;
86       !composantes ;;
```

Voici une deuxième version plus sobre utilisant la fonction «`it_list`» de Caml

```
87       let (composantes' : graphe -> int list list) = function g ->
88         let rec traite_sommet a l =
89                                it_list (fun acc s' -> if mem s' acc then acc else
90                     traite_sommet (s'::acc) (assoc s' g)) a l
91         and traite_graphe a l =
92                                it_list (fun (acc,res) (s,al) ->
93               if mem s acc then acc,res else
94               let comp = traite_sommet [s] al in
95               comp@acc,comp::res) a l
96         in snd (traite_graphe ([],[]) g)
97       ;;
```

Exercice n°4: Recherche de cycles

```
98
99      let (cyclique : graphe -> bool) = function g ->
100       let rec traite_sommet acc orb = function
101                               []        -> acc
102       | s'::reste -> let acc' = if mem s' orb then
103                       failwith "cycle"
104                     else if mem s' acc then
105                       acc
106                     else
107                       traite_sommet (s'::acc) (s'::orb) (assoc s' g) in
108                      traite_sommet acc' orb reste
109
110       and traite_graphe acc = function
111                               []            -> false
112       | (s,al)::reste -> let acc' = if mem s acc then
113       acc
114     else
115       traite_sommet acc [s] al in
116             traite_graphe acc' reste
117
118       in
119       try traite_graphe [] g with Failure "cycle" -> true ;;
```