Graphes en OCaml

OPTION INFORMATIQUE - TP nº 2.9 - Olivier Reynet

A Représentation

A1. Quel est le type OCaml du graphe suivant ? `let g = [| [1;2] ; [0;3;4] ; [0;5;6] ; [1] ; [1] ; [2] ; [2] |]`. Le dessiner.

> **Solution :** C'est un `int list array`, c'est-à-dire un tableau de listes d'entiers, qui est une liste d'adjacence représentant un graphe.

A2. Écrire une fonction de signature `liste_vers_matrice : int list array -> bool array array` qui tranforme une liste d'adjacence en matrice d'adjacence.

Solution :

```
let liste_vers_matrice g =
  let n = Array.length g in
  let matrice = Array.make_matrix n n false in
  let rec lire_ligne voisins row =
        match voisins with
            | [] -> ()
            | u::t -> matrice.(row).(u) <- true; lire_ligne t row in
  for v = 0 to n - 1 do
    lire_ligne g.(v) v;
  done;
  matrice ;;
```

A3. Écrire une fonction de signature `matrice_vers_liste : bool array array -> int list array` qui effectue l'opération inverse.

Solution :

```
let matrice_vers_liste matrice =
    let n = Array.length matrice in
    let g = Array.make n [] in
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        if matrice.(i).(j) then g.(i) <- j::g.(i)
      done;
    done;
    g ;;
```

B Parcours de graphe en mode récursif

B4. Écrire une fonction de signature `parcours_largeur : int list array -> int -> int list` qui parcours en largeur un graphe donné sous la forme d'une liste d'adjacence. Cette fonction opère à l'aide d'une fonction récursive auxiliaire qui explore les sommets dans le bon ordre. Un type file d'attente n'est **pas** nécessaire, une liste et l'opérateur `@` suffisent. Pour mémoriser les sommets découverts on utilisera un tableau de booléens ou une liste de sommets découverts. Comparer la complexité des deux approches.

Solution : Le test `List.mem` est de complexité linéaire en la taille de la liste, alors que l'accès à `decouverts.(v)` s'effectue en temps constant. La version avec le tableau est donc plus efficace et atteint $O(n+m)$.

```
(* version liste de sommets découverts *)
 let parcours_largeur g s0 =
    let rec explorer file decouverts =
        match file with
              | [] -> []
              | v::t when List.mem v decouverts -> explorer t decouverts
              | v::t -> v::(explorer (t @ g.(v)) (v::decouverts))
    in explorer [s0] [] ;;

(* verison tableaux de booléens *)
let parcours_largeur g s0 =
  let n = Array.length g in
  let decouverts = Array.make n false in
  let rec explorer file =
      match file with
            | [] -> []
            | v::t when decouverts.(v) -> explorer t
            | v::t -> decouverts.(v) <- true; v::(explorer (t @ g.(v)))
  in explorer [s0];;
```

B5. Écrire une fonction de signature `parcours_profondeur : int list array -> int -> int list` qui parcours en profondeur un graphe donné sous la forme d'une liste d'adjacence. Cette fonction opère à l'aide d'une fonction récursive auxiliaire qui explore les sommets dans le bon ordre. La structure de pile n'est pas nécessaire, une liste suffit. Pour mémoriser les sommets découverts on utilisera un tableau de booléens. On pourra utiliser la fonction `List.fold_left`

Solution : Le test `List.mem` est de complexité linéaire en la taille de la liste, alors que l'accès à `decouverts.(v)` s'effectue en temps constant. La version avec le tableau est donc plus efficace et atteint $O(n+m)$.

```
let parcours_profondeur g s0 =
  let n = Array.length g in
  let decouverts = Array.make n false; in
  let rec explorer parcours s =
    if decouverts.(s)
    then parcours
    else (let suivants = g.(s) in decouverts.(s) <- true; List.fold_left
       explorer (parcours @ [s]) suivants)
```

```
        in explorer [] s0;;
```