Graphes : modélisation et parcours

OPTION INFORMATIQUE - TP n° 3.3 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ modéliser un graphe par liste d'adjacence
- ☞ modéliser un graphe par matrice d'adjacence
- ☞ passer d'une modélisation à une autre
- ☞ parcourir un graphe en largeur et en profondeur
- ☞ implémenter l'algorithme de Dijkstra

A Modélisation d'un graphe

Dans ce qui suit on peut considérer le graphe :

```
let g = [| [1;2] ; [0;3;4] ; [0;5;6] ; [1] ; [1] ; [2] ; [2] |] ;;
```

A1. Sous quelle forme le graphe `g` est-il donné ?

A2. Dessiner le graphe `g`. Comment peut-on qualifier ce graphe ?

A3. On dispose d'un graphe sous la forme d'une liste d'adjacence. Écrire une fonction de signature `liste_vers_matrice : int list array -> bool array array` qui transforme cette représentation en une matrice d'adjacence. Les valeurs de la matrice sont des booléens : il y a une arête ou il n'y en a pas.

A4. On dispose d'un graphe sous la forme d'une matrice d'adjacence. Écrire une fonction de signature `matrice_vers_liste : bool array array -> int list array` qui transforme cette représentation en une liste d'adjacence.

A5. On dispose d'un graphe orienté sous la forme d'une liste d'adjacence. Écrire une fonction de signature `desorienter_liste : int list array -> unit` qui transforme ce graphe en un graphe non orienté. Cette fonction travaille en place.

A6. On dispose d'un graphe orienté sous la forme d'une matrice d'adjacence. Écrire une fonction de signature `desorienter_matrice : bool array array -> unit` qui transforme ce graphe en un graphe non orienté. Cette fonction travaille en place.

B Parcourir un graphe

Le parcours d'un graphe (cf. algorithme 1) est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. Les sommets passent dans une **pile** de type Last In First Out.
3. L'algorithme de **Dijkstra** passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'est un parcours en largeur qui utilise une **file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance. La plus petite distance en tête donc.

Dans cette section, on suppose qu'on manipule un graphe sous la forme d'une liste d'adjacence.

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                    ▷ s est un sommet de G
2 :     F ← une file d'attente vide                       ▷ F comme file
3 :     D ← ∅                                             ▷ D ensemble des sommets découverts
4 :     P ← une liste vide                                ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)                  ▷ Pour la preuve de la correction, on précise que d[s] = 0
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                                 ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ D alors                            ▷ x n'a pas encore été découvert
12 :                AJOUTER(D, x)
13 :                ENFILER(F, x)     ▷ Pour la preuve de la correction, on ajoute ici d[x] = d[u] + 1
14 :    renvoyer P                    ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

B1. Démontrer la terminaison du parcours en largeur.

B2. Démontrer la correction du parcours en largeur.

B3. Écrire une fonction de signature `parcours_largeur : int list array -> int -> int list` qui renvoie un parcours en largeur du graphe à partir du sommet de départ passé en paramètre. En mélangeant les approches impératives et récursives, on peut proposer des versions à la fois fluides et efficaces. On s'appuie sur une structure file d'attente programmée comme suit :

```
type 'a file = {
  mutable tete : 'a list;
  mutable queue : 'a list;
}

let filevide () = { tete = []; queue = [] };;

let est_vide file = (file.tete = [] && file.queue = []);;

let enfiler x file = file.queue <-  x :: file.queue;;

let defiler file=
  let rec rev rlst lst =
```

```
    match lst with
    | [] -> rlst
    | h :: t -> rev (h::rlst) t in
  match file.tete with
    | [] ->  ( match rev [] file.queue with
                 | [] -> failwith "Erreur : file vide !"
                 | e :: t -> file.tete <- t; file.queue <- []; e )
    | e :: t -> file.tete <- t; e;;

let tete file=
  match file.tete with
  | [] ->
      (match List.rev file.queue with
       | [] -> None
       | x :: _ -> Some x)
  | x :: _ -> Some x;;
```

B4. Que vaut la complexité de `parcourir_largeur` ?

B5. Écrire une fonction de signature `parcours_profondeur : int list array -> int -> int list` qui parcours en profondeur un graphe et qui renvoie la liste des sommets parcourus.

C Plus courts chemins : algorithme de Dijkstra

Algorithme 2 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(G = (S, A, w), s_0)        ▷ Trouver les plus courts chemins à partir de s_0 ∈ V
2 :     Δ ← s_0                                  ▷ Δ est le dictionnaire des sommets dont on connaît la distance à s_0
3 :     Π ← ∅                                    ▷ Π[s] est le parent de s dans le plus court chemin de s_0 à s
4 :     d ← ∅                                    ▷ l'ensemble des distances au sommet s_0
5 :     ∀ s ∈ V, d[s] ← w(s_0, s)                ▷ w(s_0, s) = +∞ si s n'est pas voisin de s_0, 0 si s = s_0
6 :     tant que Δ̄ n'est pas vide répéter        ▷ Δ̄ : sommets dont la distance n'est pas connue
7 :         Choisir u dans Δ̄ tel que d[u] = min(d[v], v ∈ Δ̄)   ▷ Choix glouton !
8 :         Δ = Δ ∪ {u}                          ▷ On prend la plus courte distance à s_0 dans Δ̄
9 :         pour x ∈ V_G(u) répéter              ▷ pour tous les voisins de u
10 :            si d[x] > d[u] + w(u, x) alors
11 :                d[x] ← d[u] + w(u, x)        ▷ Mises à jour des distances des voisins
12 :                Π[x] ← u                     ▷ Pour garder la tracer du chemin le plus court
13 :    renvoyer d, Π
```

C1. Démontrer la terminaison de l'algorithme de Dijkstra.

C2. Démontrer la correction de l'algorithme de Dijkstra.

C3. Quelle est la complexité de l'algorithme de Dikjstra ?

C4. Exécuter à la main l'algorithme de Dijsktra sur le graphe orienté suivant en complétant à la fois le tableau des distances et le tableau des parents qui permet de reconstruire le chemin a posteriori. Le tableau parent à la case `i` contient le sommet précédent sur le chemin.

```
let g = [| [(1,7);(2,1)] ;
           [(0,7);(2,5);(3,4);(4,2);(5,1)] ;
```

```
[(0,1);(1,5);(4,2);(5,7)];
[(1,4);(4,5)];
[(1,2);(2,2);(3,5);(5,3)];
[(1,2);(2,7);(4,3)] |] ;;
```

Algorithme 3 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

```
1 : Fonction DIJKSTRA(g, s_0)                          ▷ Trouver les plus courts chemins à partir de s_0
2 :     n ← nombre de sommets de G
3 :     fp ← une file de priorités                      ▷ contient les tuples (distance, sommet)
4 :     ENFILER(fp,(0,s_0))                             ▷ initialisation de la file de priorités
5 :     d ← un tableau de taille n                      ▷ recense les distances de chaque sommet à s_0
6 :     ∀s ∈ S, d[s] ← w(s_0, s)                        ▷ w(s_0, s) = +∞, 0 si s = s_0
7 :     parents ← un tableau de taille n                ▷ Recense le père d'un sommet sur le chemin le plus court
8 :     parents[s_0] ← s_0                              ▷ Le sommet de départ est son propre père sur le chemin le plus court
9 :     Δ ← ∅                                           ▷ Les sommets découverts
10 :    pour _ de 1 à n répéter                         ▷ Répéter n fois
11 :        δ, u ← DÉFILER(fp)                          ▷ Choix glouton! On prend le minimum.
12 :        AJOUTER(Δ, u)
13 :        pour v ∈ g[u] répéter                       ▷ Pour chaque voisin de u
14 :            si v ∉ Δ et d[u] + δ < d[v] alors       ▷ Si la distance est meilleure en passant par u
15 :                d[v] ← d[u] + δ                     ▷ Mettre à jour la distance au voisin
16 :                ENFILER(fp, (d[v],v))               ▷ Enfiler dans la file de priorité la nouvelle priorité
17 :                parents[v] ← u                      ▷ Pour garder la trace du chemin le plus court
18 :    renvoyer d, parents
```

C5. Compléter la fonction `dijkstra : (int * int)list array -> int array * int array` en suivant l'algorithme de Dijkstra 3. Cette fonction renvoie les plus courtes distances à partir d'un sommet d'un graphe ainsi que les directions à prendre. Cette fonction s'appuie sur une file de priorités implémentée par un tas binaire.

```
type 'a qdata = {valeur: 'a; priorite: int};;
type 'a file_priorites = {mutable taille: int; tas: 'a qdata array};;

let echanger t i j = let tmp = t.(i) in t.(i) <- t.(j); t.(j) <- tmp;;

let rec faire_monter tas k = match k with
    | 0 -> ()
    | _ -> let p = (k - 1)/2 in
           if tas.(k).priorite < tas.(p).priorite
           then (echanger tas k p ; faire_monter tas p);;

let rec faire_descendre tas taille k = match k with
    | n when 2*n + 1 >= taille -> () (* pas d'enfants *)
    | n when 2*n + 1 = (taille - 1) -> (* un seul enfant *)
       if tas.(n).priorite > tas.(2*n + 1).priorite
       then echanger tas (2*n + 1) n
    | n -> begin (* deux enfants *)
            let f = if tas.(2*n + 1).priorite < tas.(2*n + 2).priorite then  2*n
                 +  1 else 2*n + 2 in
            if tas.(n).priorite > tas.(f).priorite then (echanger tas n f;
                faire_descendre tas taille f;)
```

```
          end;;

let creer_file_priorites n (v,p) = {taille = 0; tas = Array.init n (fun _ -> {
    valeur = v; priorite=p})};;

let inserer filep (v,p) =
  let size = Array.length filep.tas in
  if filep.taille + 1 > size then failwith "FULL_PRIORITY_QUEUE";
  filep.tas.(filep.taille) <- {valeur=v; priorite=p}; (* Placer l'élément à la
      fin *)
  faire_monter filep.tas filep.taille; (* recréer la structure de tas *)
  filep.taille <- filep.taille + 1;;

let retirer_minimum filep =
  if filep.taille = 0 then failwith "EMPTY_PRIORITY_QUEUE";
  let premier = filep.tas.(0) in (* sauvegader l'élément le plus prioritaire *)
    filep.taille <- filep.taille - 1;
    filep.tas.(0) <- filep.tas.(filep.taille);  (* Placer le dernier élément au
        sommet *)
    faire_descendre filep.tas filep.taille 0;   (* Recréer la structure de tas
        *)
    premier;;   (* renvoyer le résultat *)

let show_path start stop d parents =
    print_string "Cost -> "; print_int d.(stop); print_newline ();
    print_string "Path -> ";
    let rec aux current path =
        if current = start
        then List.iter (fun e -> print_int e; print_string " ") path
        else let father = parents.(current) in aux father ( father :: path )
    in aux stop [ stop ];;

let fp_dijkstra g start stop =
    let fp = creer_file_priorites 10 (max_int,max_int) in
    let n = Array.length g in
    let d = Array.make n max_int in
    d.(start) <- 0;
    let parents = Array.make n start in
    let delta = Array.make n false in
    inserer fp (start, d.(start));
    for _ = 1 to n do
        (* transfert de Delta_bar à Delta *)
        (* mise à jour des distances et de la file de priorités *)
    done;
    show_path start stop d parents;;
```

C6. Quelle la complexité de l'algorithme de Dijkstra ainsi implémenté ?