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 `list_to_matrix` qui transforme cette représentation en une matrice d'adjacence.

A4. On dispose d'un graphe sous la forme d'une matrice d'adjacence. Écrire une fonction `matrix_to_list` 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 `desoriented_list` qui transforme ce graphe en un graphe non orienté.

A6. On dispose d'un graphe orienté sous la forme d'une matrice d'adjacence. Écrire une fonction `desoriented_matrix` qui transforme ce graphe en un graphe non orienté.

B Parcourir un graphe

Le parcours d'un graphe 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.

B1. Écrire une fonction de signature `bfs : int list array -> int -> int list` qui parcours en largeur un graphe et qui renvoie la liste des sommets parcourus. On pourra s'inspirer du code récursif suivant :

```
let bfs g v0 =
  let rec explore queue  visited =
  match queue with
      | ..
  in explore [v0] [];;
```

Tester l'algorithme sur le graphe suivant :

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

B2. Écrire une fonction récursive de signature `dfs : int list array -> int -> int list` qui parcours en profondeur un graphe et qui renvoie la liste des sommets parcourus.

B3. Que valent les complexités de `bfs` et `dfs` ?

C Plus courts chemins : algorithme de Dijkstra

Algorithme 1 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

Fonction DIJKSTRA($G = (S, A, w)$, s_0) ▷ Trouver les plus courts chemins à partir de $s_0 \in V$
    $\Delta \leftarrow s_0$ ▷ Δ est le dictionnaire des sommets dont on connaît la distance à s_0
    $\Pi \leftarrow \emptyset$ ▷ $\Pi[s]$ est le parent de s dans le plus court chemin de s_0 à s
    $d \leftarrow \emptyset$ ▷ l'ensemble des distances au sommet s_0
    $\forall s \in V, d[s] \leftarrow w(s_0, s)$ ▷ $w(s_0, s) = +\infty$ si s n'est pas voisin de s_0, 0 si $s = s_0$
    tant que $\bar{\Delta}$ n'est pas vide **répéter** ▷ $\bar{\Delta}$: sommets dont la distance n'est pas connue
        Choisir u dans $\bar{\Delta}$ tel que $d[u] = \min\left(d[v], v \in \bar{\Delta}\right)$ ▷ Choix glouton !
        $\Delta = \Delta \cup \{u\}$ ▷ On prend la plus courte distance à s_0 dans $\bar{\Delta}$
        pour $x \in \mathcal{V}_G(u)$ **répéter** ▷ pour tous les voisins de u
            si $d[x] > d[u] + w(u, x)$ **alors**
                $d[x] \leftarrow d[u] + w(u, x)$ ▷ Mises à jour des distances des voisins
                $\Pi[x] \leftarrow u$ ▷ Pour garder la tracer du chemin le plus court
    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 2 Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

Fonction DIJKSTRA(g, s_0) ▷ Trouver les plus courts chemins à partir de s_0
  $n \leftarrow$ nombre de sommets de G
  pq $\leftarrow$ une file de priorités ▷ contient les tuples (distance, sommet)
  ENFILER(pq,(0,s_0)) ▷ initialisation de la file de priorités
  d $\leftarrow$ un dictionnaire ▷ recense les distances à s_0
  $\forall s \in S, d[s] \leftarrow w(s_0, s)$ ▷ $w(s_0, s) = +\infty$ si s n'est pas voisin de s_0, 0 si $s = s_0$
  parents $\leftarrow \{s_0 : s_0\}$ ▷ parents[s] : le parent de s dans le plus court chemin de s_0 à s (dictionnaire)
  visités $\leftarrow \emptyset$
  tant que pq n'est pas vide **répéter**
    $\delta, u \leftarrow$ DÉFILER(pq) ▷ Choix glouton !
    AJOUTER(visités, u)
    pour $v \in g[u]$ **répéter** ▷ Pour chaque voisin de u
      si v $\notin$visités et $d[u] + \delta < d[v]$ **alors** ▷ si la distance est meilleure en passant par u
        $d[v] \leftarrow d[u] + \delta$ ▷ Mises à jour des distances des voisins
        ENFILER(pq, (d[v],v))
        parents[v] $\leftarrow u$ ▷ Pour garder la tracer du chemin le plus court
  renvoyer d, parents

C5. Compléter la fonction `dijkstra : (int * int)list array -> int array * int array` en suivant l'algorithme de Dijkstra 2. 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 = {value: 'a; priority: int};;
type 'a priority_queue = {mutable first_free: int; heap: 'a qdata array};;

let swap t i j = let tmp = t.(i) in t.(i) <- t.(j); t.(j) <- tmp;;
let rec up heap k = match k with
    | 0 -> ()
    | _ -> let p = (k - 1)/2 in
                if heap.(k).priority < heap.(p).priority then (swap heap k p ; up
                     heap p);;

let rec down heap first_not_used k = match k with
    | n when 2*n + 1 >= first_not_used -> () (* Leave done *)
    | n when 2*n + 1 = (first_not_used - 1) -> if heap.(n).priority > heap.(2*n
      + 1).priority then swap heap (2*n + 1) n (* Leave done *) (* Leave done
      *)
    | n -> begin
```

```
              let f = if heap.(2*n + 1).priority < heap.(2*n + 2).priority then
                 2*n +  1 else 2*n + 2 in
              if heap.(n).priority > heap.(f).priority then (swap heap n f; down
                 heap first_not_used f;)
            end;;

let make_priority_queue n (v,p) = {first_free = 0; heap = Array.init n (fun i ->
     {value = v; priority=p})};;

let insert pq (v,p) =
  let size = Array.length pq.heap in
  if pq.first_free + 1 > size then failwith "FULL_PRIORITY_QUEUE";
  pq.heap.(pq.first_free) <- {value=v; priority=p};
  up pq.heap pq.first_free;
  pq.first_free <- pq.first_free + 1;;

let get_min pq =
  if pq.first_free = 0 then failwith "EMPTY_PRIORITY_QUEUE";
  let first = pq.heap.(0).value in
    pq.first_free <- pq.first_free - 1;
    pq.heap.(0) <- pq.heap.(pq.first_free);
    down pq.heap pq.first_free 0;
    first;;


let pq_dijkstra g start stop =
     let pq = make_priority_queue 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 = Hashtbl.create n in
            let computed = Array.make n false in
                insert pq (start, d.(start));
                for _ = 1 to n do
                (* TODO *)
                done;
        (d, parents);;
```

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