

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 parcourt 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] |] ;;
```

Solution :

```
let bfs g v0 =
  let rec explore queue visited =
    match queue with
    | [] -> []
    | v::t when List.mem v visited -> explore t visited
    | v::t -> v::(explore (t @ g.(v)) (v::visited))
  in explore [v0] [] ;;
```

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

Solution :

```
let dfs g v0 =
  let rec explore stack visited =
    match stack with
    | [] -> []
    | v::t when List.mem v visited -> explore t visited
    | v::t -> v::(explore (g.(v) @ t) (v::visited))
  in explore [v0] [] ;;
```

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

Solution : Les complexités valent $\mathcal{O}(|S| + |A|) = \mathcal{O}(n + m)$. En effet, on défile `queue` jusqu'à ce qu'elle soit vide. Or, on enfile systématiquement un sommet (n) et ses sommets voisins (m) même s'ils ont été découverts. D'où le résultat.

C Plus courts chemins : algorithme de Dijkstra

Algorithme 1 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  $s_0 \in V$ 
2:    $\Delta \leftarrow s_0$                                ▷  $\Delta$  est le dictionnaire des sommets dont on connaît la distance à  $s_0$ 
3:    $\Pi \leftarrow \emptyset$                              ▷  $\Pi[s]$  est le parent de  $s$  dans le plus court chemin de  $s_0$  à  $s$ 
4:    $d \leftarrow \emptyset$                                ▷ l'ensemble des distances au sommet  $s_0$ 
5:    $\forall s \in V, d[s] \leftarrow w(s_0, s)$              ▷  $w(s_0, s) = +\infty$  si  $s$  n'est pas voisin de  $s_0$ , 0 si  $s = s_0$ 
6:   tant que  $\bar{\Delta}$  n'est pas vide répéter           ▷  $\bar{\Delta}$  : sommets dont la distance n'est pas connue
7:     Choisir  $u$  dans  $\bar{\Delta}$  tel que  $d[u] = \min(d[v], v \in \bar{\Delta})$    ▷ Choix glouton!
8:      $\Delta = \Delta \cup \{u\}$                        ▷ On prend la plus courte distance à  $s_0$  dans  $\bar{\Delta}$ 
9:     pour  $x \in \mathcal{V}_G(u)$  répéter                 ▷ pour tous les voisins de  $u$ 
10:      si  $d[x] > d[u] + w(u, x)$  alors
11:         $d[x] \leftarrow d[u] + w(u, x)$            ▷ Mises à jour des distances des voisins
12:         $\Pi[x] \leftarrow u$                        ▷ Pour garder la tracer du chemin le plus court
13:   renvoyer  $d, \Pi$ 

```

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

Solution : Terminaison de l'algorithme : avant la boucle *tant que*, $\bar{\Delta}$ possède $n - 1$ éléments, si $n \in \mathbb{N}^*$ est l'ordre du graphe. À chaque tour de boucle *tant que*, l'ensemble $\bar{\Delta}$ décroît strictement d'un élément et atteint donc nécessairement zéro. Le cardinal de $\bar{\Delta}$ est donc un variant de boucle. L'algorithme se termine lorsque le cardinal de $\bar{\Delta}$ atteint zéro.

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

Solution : On note :

- δ_u la distance la plus courte du sommet s_0 au sommet u .
- $d[u]$ la distance trouvée par l'algorithme entre le sommet s_0 le sommet u .

On souhaite démontrer la correction en montrant que $\mathcal{J} : \forall x \in \Delta, d[x] = \delta_x$ est un invariant de boucle.

1. Avant la boucle : Δ ne contient que le sommet s_0 . Or, $d[s_0] = 0$ et $\delta_{s_0} = 0$. Donc, l'invariant est vérifié à l'entrée de la boucle.
2. Pour une itération quelconque, on suppose que l'invariant \mathcal{J} est vérifié à l'entrée de la boucle. Un sommet u est sélectionné dans $\bar{\Delta}$. Pour ce sommet u , qui n'appartient pas encore à Δ , on souhaite montrer qu'à la fin de l'itération $d[u] = \delta_u$.

On procède par l'absurde en supposant que $d[u] \neq \delta_u$ et qu'il existe un plus court chemin P de s_0 à u tel que la longueur de ce chemin $\lambda(P)$ soit **strictement** plus petite que $d[u]$:

$$\lambda(P) < d[u]$$

Ce chemin P démarre d'un sommet de Δ et le quitte au bout d'un certain temps pour atteindre u . Soit xy la première arête quittant Δ de ce chemin P : $x \in \Delta$ et $y \in \bar{\Delta}$. Soit P_x

le chemin de s_0 à x . Ce chemin est un plus court chemin, par hypothèse d'induction et $d[x] = \delta_x$. On a donc :

$$\lambda(P_x) + w(x, y) = d[x] + w(x, y) \leq \lambda(P) < d[u] \quad (1)$$

Comme y est un sommet adjacent à x , la distance à w a été mise à jour par l'algorithme précédemment. On a donc :

$$d[y] \leq d[x] + w(x, y) \quad (2)$$

De plus, comme on a sélectionné u dans $\bar{\Delta}$ tel que la distance soit minimale au sommet de départ, et comme $y \in \bar{\Delta}$, on a également :

$$d[u] \leq d[y] \quad (3)$$

En combinant ces équations, on aboutit à la contradiction suivante : $d[u] < d[u]$. Un chemin tel que P n'existe donc pas et $d[u] = \delta_u$. L'invariant \mathcal{J} est donc vérifié à la fin de l'itération.

3. Comme l'invariant \mathcal{J} est vérifié à l'entrée de la boucle et qu'il n'est pas modifié par les instructions de la boucle, on en déduit qu'il est vrai à la fin de la boucle. L'algorithme de Dijkstra est donc correct.

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

Solution :

La complexité de l'algorithme de Dijkstra dépend de l'ordre n du graphe considéré et de sa taille m . La boucle *tant que* effectue exactement $n - 1$ tours. La boucle *pour* effectue à chaque fois un nombre de tour égal au nombre d'arêtes non découvertes qui partent du sommet u considéré et vont vers un sommet voisin de $\bar{\Delta}$. On ne découvre une arête qu'une seule fois, puisque le sommet u est transféré dans Δ au début de la boucle. Au final, on exécute donc la mise à jour des distances un nombre de fois égal à $2m$ (si le graphe n'est pas orienté), donc proportionnelle à son nombre d'arêtes.

En notant le coût du transfert c_t , le coût de la mise à jour des distances c_d et en déroulant la boucle *tant que*, on peut écrire :

$$C(n, m) = (n - 1)c_t + mc_d = \quad (4)$$

Les complexités c_d et c_t dépendent naturellement des structures de données utilisées pour implémenter l'algorithme.

Si on choisit une implémentation de d par un tableau, alors on a besoin de rechercher le minimum des distances pour effectuer le transfert : cela s'effectue au prix d'un tri du tableau au minimum en $c_t = O(n \log n)$. Un accès aux éléments du tableau pour la mise à jour est en $c_d = O(1)$. On a donc $C(n) = (n - 1)O(n \log n) + mO(1) = O(n^2 \log n)$.

C4. Exécuter à la main l'algorithme de Dijkstra 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)] |] ;;

```

Solution :

```

val d : int array = [|0; 5; 1; 8; 3; 6|]
val p : int array = [|0; 4; 0; 1; 2; 1|]

```

Algorithme 2 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:    $n \leftarrow$  nombre de sommets de G
3:   pq  $\leftarrow$  une file de priorités                          ▷ contient les tuples (distance, sommet)
4:   ENFILER(pq,(0, $s_0$ ))                                       ▷ initialisation de la file de priorités
5:   d  $\leftarrow$  un dictionnaire                                  ▷ recense les distances à  $s_0$ 
6:    $\forall s \in S, d[s] \leftarrow w(s_0, s)$                        ▷  $w(s_0, s) = +\infty$  si  $s$  n'est pas voisin de  $s_0$ , 0 si  $s = s_0$ 
7:   parents  $\leftarrow \{s_0 : s_0\}$     ▷ parents[s] : le parent de s dans le plus court chemin de  $s_0$  à s (dictionnaire)
8:   visités  $\leftarrow \emptyset$ 
9:   tant que pq n'est pas vide répéter
10:     $\delta, u \leftarrow$  DÉFILER(pq)                            ▷ Choix glouton!
11:    AJOUTER(visités, u)
12:    pour  $v \in g[u]$  répéter                                  ▷ Pour chaque voisin de  $u$ 
13:      si  $v \notin$  visités et  $d[u] + \delta < d[v]$  alors       ▷ si la distance est meilleure en passant par u
14:         $d[v] \leftarrow d[u] + \delta$                           ▷ Mises à jour des distances des voisins
15:        ENFILER(pq, (d[v],v))
16:        parents[v]  $\leftarrow u$                                 ▷ Pour garder la tracer du chemin le plus court
17:   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?

Solution : Si la file de priorité est implémentée par un tas, alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. La complexité est alors en $C(n) = (n + m) \log n$. Cependant, pour que le tas soit une implémentation pertinente, il est nécessaire que $m = O(\frac{n^2}{\log n})$, c'est à dire que le graphe ne soit pas complet, voire un peu creux!