Tas, files et plus courts chemins

MPSI/MP Option informatique - TP nº 3.3 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ définir un tas-min et un tas-max
- ☞ trier un tableau pas tas
- ☞ utiliser un tas pour créer une file de priorité
- ☞ appliquer les tas à l'algorithme de Dijsktra

A Implémentation d'un tas max et tri par tas

A1. Créer une variable `heap_test` de type `Array` contenant les entiers de 1 à 10 dans l'ordre décroissant. On génèrera automatiquement cette variable (pas in extenso). Est-ce un tas? Si oui, de quel type?

Solution :

```
let heap_test = Array.init 10 (fun i -> 10 - i);;
val heap_test : int array = [|10; 9; 8; 7; 6; 5; 4; 3; 2; 1|]
```

C'est un tas-max.

A2. Écrire une fonction de signature `swap : 'a array -> int -> int -> unit` qui échange la place de deux éléments dans un type `Array`. Tester cette fonction sur le tableau `heap_test`.

Solution :

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

Comme en OCaml les `Array` sont indexés à partir de 0, on choisit de placer la racine du tas en 0. Puis, si le père est k alors les fils seront en $2k+1$ et $2k+2$. Si un fils est en k, alors le père est en $\lfloor \frac{k-1}{2} \rfloor$. Il faut noter qu'un tas peut ne pas être rempli. Dans ce cas, on prendra soin de relever l'indice de la première case non remplie.

A3. Écrire une fonction de signature `up : 'a array -> int -> unit` qui, si cela est possible, fait monter un élément d'après son indice dans le tas tout en préservant la structure du tas.

Solution :

```
let up heap k =
  let rec aux i = match i with
    | 0 -> ()
    | _ -> let f = (i - 1)/2 in
           if heap.(i) > heap.(f) then (swap heap i f ; aux f)
  in  aux k;;
```

A4. Écrire une fonction de signature `down : 'a array -> int -> int -> unit` qui, si cela est possible, fait descendre un élément dans un tas max tout en préservant la structure du tas. Le premier entier de la signature est l'indice de la première case non remplie du tas, le second l'indice de l'élément à faire descendre.

Solution :

```
let down heap first_not_used k =
  let rec aux i = match i with
    | i when 2*i + 1 > first_not_used -> () (* Leave *)
    | i when 2*i + 2 = first_not_used  ->  let f = 2*i + 1 in
                   if heap.(i) < heap.(f)
                   then (swap heap i f; aux f;)
    | i -> begin
           let f = if heap.(2*i + 1) > heap.(2*i + 2)
                    then  2*i +  1 else 2*i + 2 in
           if heap.(i) < heap.(f)
           then (swap heap i f; aux f;)
           end
  in aux k;;
```

On cherche à créer un tas de deux manière différente :

- soit en considérant que le premier élément du tableau est un tas que l'on fait grossir avec les éléments de droite du tableau,
- soit en considérant que les feuilles sont des tas à un élément que l'on fait grossir au fur avec les éléments de gauche du tableau. On remarque dans ce cas que quelle que soit la taille n du tas, l'élément d'indice $n/2$ est toujours la première feuille. On va donc insérer les $n/2$ premiers éléments dans leurs feuilles et faire apparaître la structure de tas.

A5. Effectuer à la main la transformation du tableau `[|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]` en tas-max par les deux méthodes.

A6. Écrire une fonction de signature `heap_make_up : 'a array -> unit` qui transforme un tableau en un tas en faisant monter les éléments. Quelle la complexité de cette fonction ?

Solution :

```
let heap_make_up t =  for k = 1 to Array.length t - 1 do up t k done;;
```

La complexité au pire de cette fonction est linéaire en $O(n \log n)$ si n est la taille du tas. En effet, dans le pire des cas, on fait remonter les n nœuds jusqu'à la racine, c'est à dire toute la hauteur de l'arbre qui vaut $\log n$.

A7. Écrire une fonction de signature `heap_make_down : 'a array -> unit` qui transforme un tableau en un tas en faisant descendre les éléments. Quelle est la complexité de cette fonction [1] ?

Solution :

```
let heap_make_down t = let size = Array.length t in
    for k = size/2 - 1 downto 0 do down t size k done;;
```

En considérant que le nombre de nœuds ayant la hauteur h est majoré par $\lceil \frac{n}{2^{h+1}} \rceil$ on montre que :

$$\sum_{h=1}^{p} \lceil \frac{n}{2^{h+1}} \rceil h \leqslant \frac{p(p+1)}{2} + n \sum_{h=1}^{p} \frac{h}{2^{h+1}} = O(n) \tag{1}$$

A8. Écrire une fonction de signature `heap_sort : 'a array -> unit` qui effectue le tri par tas d'un tableau. Quelle la complexité de cette fonction ?

Solution :

```
let heap_sort t =
  heap_make_down t;
  let size = Array.length t in
    for k = size - 1  downto 0 do (swap t 0 k; down t k 0) done;;
```

A9. Tester le tri par tas sur un tableau de 10000 valeurs entières choisies aléatoirement entre 0 et 1000000.

Solution :

```
Random.init 42;;
let test = Array.init 100000 (fun i -> Random.full_int 1000000);;
heap_sort test;;
test;;
```

B Implémentation d'une file de priorités

Pour implémenter une file de priorités à partir d'un tas, il est nécessaire de définir le type de données que contient la file, puis de programmer les opérations de base de la file de priorités. Les opérations sur une file de priorités sont :

1. créer une file vide,
2. insérer dans la file une valeur associée à une priorité (ENFILER),
3. sortir de la file la valeur associée la priorité maximale (ou minimale) (DÉFILER).

1. On suppose que le nombre de nœuds ayant la hauteur h est majoré par $\lceil \frac{n}{2^{h+1}} \rceil$

On considère une file de priorités dont les éléments sont des couples (v, p) et p est un entier naturel. **Plus p est faible, plus la priorité est élevée**.

Par ailleurs, on se dote des types et des exceptions suivantes :

```
type 'a qdata = {value: 'a; priority: int};;
type 'a priority_queue = {mutable first_free: int; heap: 'a qdata array};;
exception EMPTY_PRIORITY_QUEUE;;
exception FULL_PRIORITY_QUEUE;;
```

B1. De quel type est le tas nécessaire à la construction de cette file de priorités?

Solution : Il s'agit d'un tas-min.

B2. Écrire une fonction de signature `up : 'a array -> int -> unit` qui, si cela est possible, fait monter un élément d'après son indice dans le tas tout en préservant la structure du tas. On veillera à bien faire monter en fonction de la priorité.

Solution :

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

B3. Écrire une fonction de signature `down : 'a array -> int -> int -> unit` qui, si cela est possible, fait descendre un élément dans un tas max tout en préservant la structure du tas. Le premier entier de la signature est l'indice de la première case non remplie du tas, le second l'indice de l'élément à faire descendre. On veillera à bien faire descendre en fonction de la priorité.

Solution :

```
let down heap first_not_used k =
  let rec aux i = match i with
    | i when 2*i + 1 > first_not_used -> () (* Leave *)
    | i when 2*i + 2 = first_not_used  ->  let f = 2*i + 1 in
          if heap.(i).priority > heap.(f).priority then (swap heap i f;
              aux f;)
    | i -> begin
          let f = if heap.(2*i + 1).priority < heap.(2*i + 2).priority
                then  2*i +  1 else 2*i + 2 in
          if heap.(i).priority > heap.(f).priority
          then (swap heap i f; aux f;)
        end
  in aux k;;
```

B4. Écrire une fonction de signature `make_priority_queue : int -> 'a * int -> 'a priority_queue` qui crée une file de priorité à n éléments initialisés à l'aide d'un type `qdata` donné en paramètre et dont le premier élément libre est le premier du tas.

Solution :

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

B5. Écrire une fonction de signature `insert : 'a priority_queue -> 'a * int -> unit` qui enfile un élément dans la file de priorités. On veillera à préserver la structure du tas et à renvoyer une exception si l'opération n'est pas possible.

Solution :

```
let insert pq (v,p) =
  let size = Array.length pq.heap in
  if pq.first_free + 1 > size then raise 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;;
```

B6. Écrire une fonction de signature `get_min : 'a priority_queue -> 'a` qui renvoie l'élément `value` de priorité maximale. On veillera à préserver la structure du tas et à renvoyer une exception si l'opération n'est pas possible.

Solution :

```
let get_min pq =
  if pq.first_free = 0 then raise 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;;
```

C Dijkstra et files de priorités

L'algorithme de Dijkstra (cf. figure 1) se décompose en deux opérations répétées à chaque itération :

1. transférer de l'élément de distance minimale dans l'ensemble des éléments explorés,
2. mettre à jour les distances en fonction de ce nouvel élément sur le chemin.

L'utilisation d'une file de priorité afin d'extraire l'élément de distance minimale permet d'améliorer la complexité de l'algorithme et d'atteindre $O((n+m)\log n)$, si n est l'ordre du graphe et m le nombre d'arêtes. On cherche donc à implémenter l'algorithme de Dijkstra sur un graphe pondéré en utilisant une file de priorité dont les priorités sont les distances au sommet de départ de l'algorithme.

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

Fonction DIJKSTRA($G = (V, E, w)$, a) ▷ Trouver les plus courts chemins à partir de $a \in V$
  $\Delta \leftarrow a$ ▷ Δ est l'ensemble des sommets dont on connaît la distance à a
  $\Pi \leftarrow$ un dictionnaire vide ▷ $\Pi[s]$ est le parent de s dans le plus court chemin de a à s
  $d \leftarrow$ l'ensemble des distances au sommet a
  $\forall s \in V, d[s] \leftarrow w(a,s)$ ▷ $w(a,s) = +\infty$ si s n'est pas voisin de a, 0 si $s = a$
  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 $d[u] = \min\left(d[v], v \in \bar{\Delta}\right)$ ▷ On prend la plus courte distance à a dans $\bar{\Delta}$
    $\Delta = \Delta \cup \{u\}$ ▷ Transfert
    pour $x \in \bar{\Delta}$ **répéter** ▷ Ou bien $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$, pour tous les voisins de u dans $\bar{\Delta}$
      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

On se munit d'un type de graphe permettant de modéliser les graphes pondérés statiques :

```
let n = 6;;
type graph = {size: int; adj: int list array; w: int array array};;
```

Les graphes qu'on manipule de taille fixe n. Les sommet sont représentés par des entiers de 0 à $n-1$ et on utilise des listes d'adjacence. Le choix d'un type enregistrement, d'un type `int array` et d'un type `int array array` permet d'accéder directement à un élément du graphe :

- `g.size` est l'ordre du graphe,
- `g.adj.(a)` est la liste des voisins d'un sommet `a`,
- `g.w.(a).(b)` est le poids de l'arête (a,b).

C1. Créer un graphe vide d'ordre 6. Les poids seront initialisés à `max_int`, c'est à dire l'entier le plus élevé représentable en machine.

Solution :

```
let g = {
     size = n;
     adj = Array.make n [];
     w = Array.make_matrix n n max_int
   };;
```

C2. Écrire une fonction de signature `add_edge : graph -> int -> int -> int -> unit` qui permet

Solution :

```
let add_edge g a b wab =
  g.adj.(a) <- b::g.adj.(a);
```

```
    g.adj.(b) <- a::g.adj.(b);
    g.w.(a).(b) <- wab;
    g.w.(b).(a) <- wab;;
```

C3. Compléter le graphe vide précédemment créé graphe à la fonction `add_edge` pour représenter le graphe de la figure 1.

Solution :

```
add_edge g 0 1 7;;
add_edge g 0 2 1;;
add_edge g 1 2 5;;
add_edge g 2 4 2;;
add_edge g 2 5 7;;
add_edge g 1 3 4;;
add_edge g 1 4 2;;
add_edge g 1 5 1;;
add_edge g 3 4 5;;
add_edge g 4 5 3;;
```

C4. Appliquer à la main l'algorithme de Dijkstra sur le graphe de la figure 1.

Solution :

Δ	a	b	c	d	e	f	$\bar{\Delta}$
$\{\}$	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$
$\{a\}$	.	7	1	$+\infty$	$+\infty$	$+\infty$	$\{b,c,d,e,f\}$
$\{a,c\}$	.	6	.	$+\infty$	3	8	$\{b,d,e,f\}$
$\{a,c,e\}$	.	5	.	8	.	6	$\{b,d,f\}$
$\{a,c,e,b\}$	.	.	.	8	.	6	$\{d,f\}$
$\{a,c,e,b,f\}$	.	.	.	8	.	.	$\{d\}$
$\{a,c,e,b,f,d\}$	.	.	.	.	.	.	$\{\}$

C5. Écrire une fonction de signature `pq_dijkstra : graph -> int -> int -> unit` qui affiche sur la console le chemin le plus court d'un sommet a à un sommet b. On affichera également la distance entre les deux sommets.

Solution :

```
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 = Hashtbl.find parents current in
                aux father ( father :: path )
  in aux stop [ stop ];;
```

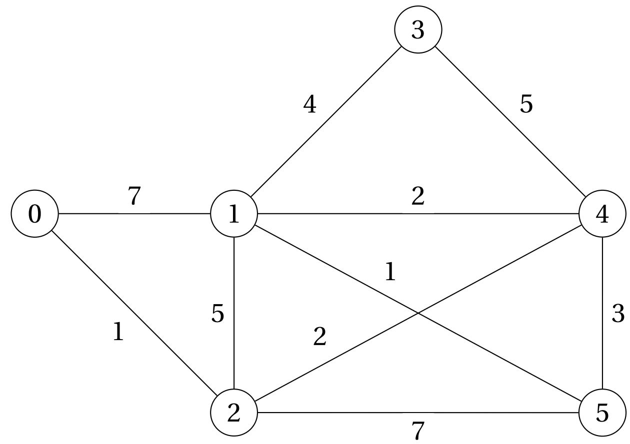

FIGURE 1 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

```
let pq_dijkstra g start stop =
  let pq = make_priority_queue 10 (-1,max_int) in
  let d = Array.make n max_int in
  d.(start) <- 0;
  let parents = Hashtbl.create n in
  insert pq (start, d.(start));
  while pq.first_free > 0 do
    let current = get_min pq in
    begin
      if current = stop then show_path start stop d parents;
      let update x =  if d.(x) > d.(current) + g.w.(current).(x)
      then  begin
              d.(x) <- d.(current) + g.w.(current).(x);
              insert pq (x, d.(x));
              Hashtbl.add parents x current;
              end
      in List.iter update g.adj.(current)
    end
  done;;

pq_dijkstra g 0 3;;
```