Files de priorités : implémentation et usage

OPTION INFORMATIQUE - Devoir n° 3.2m - Olivier Reynet

A 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).

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 suivants :

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

A1. De quel type est le tas nécessaire à la construction de cette file de priorités ?

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

A2. Créer une variable `heap_test` de type `Array` contenant les entiers de 1 à 10 dans l'ordre croissant. On génèrera automatiquement cette variable (pas in extenso). Est-ce un tas ? Si oui, de quel type ?

> **Solution :** C'est un tas-min, ça tombe drôlement bien...
>
> ```
> let heap_test = Array.init 10 (fun i -> 10 - i);;
> ```
>
> ---
>
> C'est un tas-min.

A3. 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;;
> ```
>
> ---

A4. Écrire une fonction de signature `up : 'a qdata 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 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);;
```

A5. Écrire une fonction de signature `down : 'a qdata array -> int -> int -> unit` qui, si cela est possible, fait descendre un élément dans le tas 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 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 *)
    | 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;;
```

A6. 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})
  };;
```

A7. 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 échouer avec le message `"FULL_PRIORITY_QUEUE"` 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 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;;
```

A8. Écrire une fonction de signature `get_min : 'a priority_queue -> 'a` qui renvoie l'élément `value` de priorité maximale et le retire de la file. On veillera à préserver la structure du tas et à échouer avec le message `"EMPTY_PRIORITY_QUEUE"` si l'opération n'est pas possible.

Solution :

```
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;;
```

B Dijkstra et files de priorités

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

1. transférer 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. 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. La distance la plus courte représente la priorité maximale. On pourra donc utiliser la file implémentée à la section précédente.

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

```
1 : Fonction DIJKSTRA(G = (V, E, w), a)        ▷ Trouver les plus courts chemins à partir de a ∈ V
2 :     Δ ← a                                  ▷ Δ est l'ensemble des sommets dont on connaît la distance à a
3 :     Π ← un dictionnaire vide               ▷ Π[s] est le parent de s dans le plus court chemin de a à s
4 :     d ← l'ensemble des distances au sommet a
5 :     ∀s ∈ V, d[s] ← w(a, s)                 ▷ w(a, s) = +∞ si s n'est pas voisin de a, 0 si s = a
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)   ▷ On prend la plus courte distance à a dans Δ̄
8 :         Δ = Δ ∪ {u}                        ▷ Transfert
9 :         pour x ∈ Δ̄ répéter                 ▷ Ou bien x ∈ N_G(u) ∩ Δ̄, pour tous les voisins de u dans Δ̄
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
```

On se munit d'un type (enregistrement) 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 sommets 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.order` 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).

B1. 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 = {
  order = n;
  adj = Array.make n [];
  w = Array.make_matrix n n max_int
};;
```

B2. Écrire une fonction de signature `add_edge : graph -> int -> int -> int -> unit` qui permet d'ajouter une arête au graphe. On manipule des graphes non orientés.

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;;
```

B3. Compléter le graphe vide précédemment créé grâce 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;;
```

B4. Appliquer à la main l'algorithme de Dijkstra sur le graphe de la figure 1. Créer le tableau de résolution comme indiqué ci-dessous : sommets en colonne, distances trouvées en ligne en précisant les sommets découverts). Expliquer le passage d'une ligne à une autre sur le tableau.

Δ	a	b	c	d	e	f	$\bar{\Delta}$
{}	0	7	1	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$

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	.	.	$\{d\}$
$\{a,c,e,b,f,d\}$	.	.	.	.	.	.	{}

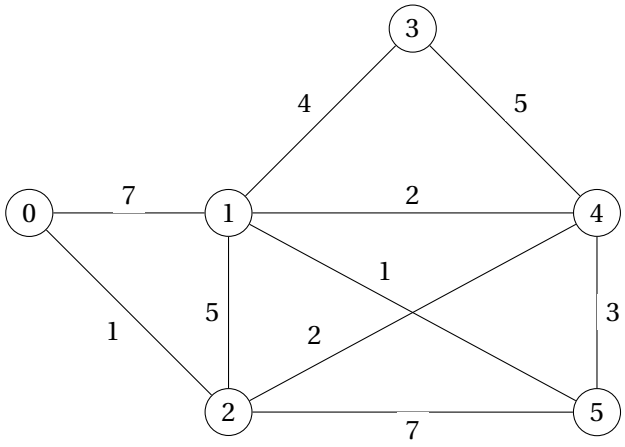

FIGURE 1 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

B5. Écrire une fonction de signature `pq_dijkstra : graph -> int -> int -> int array * (int, int)Hashtbl.t` dont les paramètres sont :

- `start` le sommet de départ,
- `stop` le sommet d'arrivée.

Cette fonction renvoie un tuple : le tableau des distances et la table de hachage des parents. Cet algorithme utilisera une file de priorités pour trier selon les distances et une table de hachage (Hashtbl) pour gérer les parents des sommets.

Solution : À la différence du TP sur les parcours des graphes au cours duquel vous avez déjà programmé Dijstra, cette implémentation est impérative et suit l'expression formelle de l'algorithme.

```
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 ];;

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;
  (d, parents);;


let (dd,hh) = pq_dijkstra g 0 3;;
show_path 0 3 dd hh;;
```

B6. Quelle est la complexité de l'algorithme ? La file de priorité a-t-elle amélioré cette complexité ? Pourquoi ?

Solution : La complexité de l'algorithme de Dijsktra 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 la taille m du graphe, c'est à dire 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 :

$$C(n,m) = (n-1)c_t + mc_d \tag{1}$$

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 $C(n) = (n-1)O(n \log n) + mO(1) = O(n^2 \log n)$, si le graphe n'est pas trop dense...

Si d est implémentée par un tas, alors on a $c_t = O(\log n)$ et $c_d = O(\log n)$. Il ne faut pas oublier le coût (non répété) de la création du tas qui est linéaire. La complexité est alors en $C(n) = O(n + (n+m)\log n) = O((n+m)\log n)$. 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!