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 ?

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 ?

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`.

A4. 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.

A5. Écrire une fonction de signature `down : 'a 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.

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.

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.

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.

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.

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.

B3. Compléter le graphe vide précédemment créé grâce à la fonction `add_edge` pour représenter le graphe de la figure 1.

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, distance 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	$+\infty$	$+\infty$	$+\infty$	$\{a,b,c,d,e,f\}$

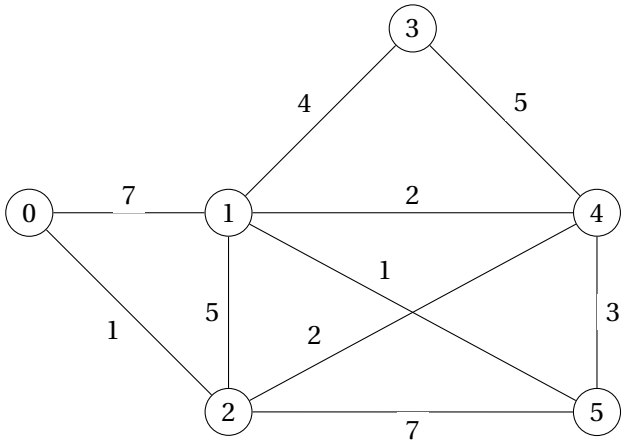

FIGURE 1 – Graphe pondéré à valeurs positives pour l'application de l'algorithme de Dijsktra.

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.

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