Option informatique MP : L'essentiel des arbres et des graphes

Louis Alvin

2 juin 2021

1 Arbres binaires

Définition Soit E un ensemble d'éléments appelés *étiquettes*. L'ensemble des arbres binaires étiquetés sur E, noté $\mathcal{A}_E$, est tel que :

- Vide $\in \mathcal{A}_E$
- $\forall x \in E,\ \forall A_g, A_d \in \mathcal{A}_E, \quad (A_g, x, A_d) \in \mathcal{A}_E$
- $\mathcal{A}_E$ est minimal pour l'inclusion.

```
type 'a arbre = Vide | Noeud of 'a arbre * 'a * 'a arbre
```

Propriété $h(A) + 1 \leq |A| \leq 2^{h(A)+1} - 1$

Arbre binaire de recherche Il s'agit d'un arbre :

- soit Vide
- soit de la forme (A_g, x, A_d) avec :
  - tous les nœuds de A_g sont $< x$
  - tous les nœuds de A_d sont $> x$
  - A_g et A_d sont également des arbres binaires de recherche.

Tas binaires Un tas max (resp. tas min) est un arbre binaire qui vérifie les deux contraintes :

- tout nœud de l'arbre est supérieur (resp. inférieur) à ses fils
- l'arbre est **parfait** : tous les niveaux sont complets sauf éventuellement le dernier, rempli de gauche à droite.

N.B. : on numérote les nœuds en partant de la racine (1), puis de gauche à droite. Si un nœud est numéroté i, alors son fils gauche est numéroté $2i$, son fils droit $2i+1$, son père $\lfloor \frac{i}{2} \rfloor$. On représente le tas par un tableau.

Tri par tas Dans un tas max, le plus grand élément du tas est à la racine, son extraction est donc en $\mathcal{O}(1)$. On l'envoie à la fin du tas ; on utilise `t.(0)` pour conserver l'indice du premier élément hors du tas (à la création, `t.(0) = Array.length t`). Pour retrouver un tas, on effectue des échanges avec le plus grand des fils pour descendre le conflit jusqu'à résolution totale.

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

let plus_grand_fils t i =
        let i_max = ref i in (* si noeud > fils on renvoie i *)
        if 2*i < t.(0) && t.(2*i) > t.(!i_max) then i_max := 2*i; (* test du fils gauche *)
        if 2*i + 1 < t.(0) && t.(2*i + 1) > t.(!i_max) then i_max := 2*i + 1; (* fils droit *)
        !i_max;;

let rec descend t i =
        let i_max = plus_grand_fils t i in
        if i_max <> i then
                (echange t i i_max;
                descend t i_max);;
```

Tant qu'il reste un élément dans le tas, on extrait le max. du tas et on descend la racine pour retrouver une structure de tas.

```
let tri_par_tas t =
        t.(0) <- Array.length t;
        while t.(0) > 1 do
                echange t 1 (t.(0) - 1);
                t.(0) <- t.(0) - 1;
                descend t 1
        done;;
```

2 Graphes

Lemme des poignées de main Le degré entrant (resp. sortant) d'un sommet u, noté $\deg^-(u)$ (resp. $\deg^+(u)$), est défini par $\deg^-(u) = \text{Card}(v \in V,\ (v,u) \in E)$. On énonce alors le lemme :

$$\sum_{v \in V} \deg^+(v) = \sum_{v \in V} \deg^-(v) = |E|$$

Modélisation avec listes d'adjacences

```
type graphe = int list array
```

Modélisation avec matrice d'adjacences N.B. : Si G est non orienté, alors la matrice A est symétrique.

```
type graphe = bool array array
```

2.1 Parcours de graphes

Parcours en profondeur On utilise un tableau de booléens pour mémoriser les sommets visités.

```
let parcours_profondeur g u0 =
        let visite = Array.make (Array.length g) false
        in let rec visiter u =
                if visite.(u) = false then
                        begin
                        visite.(u) <- true;
                        let rec visite_voisins l = match l with
                                | [] -> ()
                                | v::q -> visiter v;
                                          visite_voisins q
                        in visite_voisins g.(u)
                        end
        in visiter u0;;
```

Parcours en largeur On a besoin d'une file pour stocker les sommets selon l'ordre croissant de leur distance à u_0.

```
let parcours_largeur g u0 =
        let n = Array.length g in
        let q = Queue.create () in
        let visites = Array.make n false in
        Queue.add u0 q;
        visites.(u0) <- true
        while not (Queue.is_empty q) do
                let u = Queue.pop q in
                for v = 0 to (n-1) do
                        if visites.(v) = false then
                                (Queue.add v q;
                                visites.(v) <- true)
                done;
        done;;
```

2.2 Algorithmes de plus courts chemins

Pour les graphes non pondérés, à partir d'un sommet donné, on utilise un parcours en largeur.
Pour les graphes pondérés, 3 situations :

- D'un sommet vers tous les autres
- D'un sommet vers un autre

} Dijkstra

- De tous les sommets vers tous les autres } Floyd-Warshall

2.2.1 Algorithme de Floyd-Warshall

On suppose que le graphe pondéré n'admet aucun cycle de poids < 0.

Principe Il s'agit d'un algorithme de programmation dynamique.
On cherche les $d(i,j)$ pour tout $i,j \in V$. Hypothèse : $V = \{0,...,n-1\}$.
Sous-problème : contrainte supplémentaire sur les sommets autorisés dans les chemins.
On note : $d_{ij}^{(k)}$ la longueur d'un plus court chemin de i à j passant uniquement par des sommets dans $\{0,...,k\}$.
Problème à résoudre : $k = n-1$.

On procède par récurrence forte sur k.
Cas de base : $k = -1 \longrightarrow$ aucun sommet intermédiaire dans les chemins :

$$d_{ij}^{(-1)} = \begin{cases} 0 \text{ si } i = j \\ \omega(i,j) \text{ si } (i,j) \in E \\ +\infty \text{ sinon (pas de chemin)} \end{cases}$$

Relation de récurrence : Supposons qu'on ait calculé les $d_{ij}^{(k)}$ pour $k \in [\![-1; n-2]\!]$ fixé, et $\forall (i,j) \in V^2$. On souhaite calculer $d_{ij}^{(k+1)}$ pour $(i,j) \in V^2$.
Au final, on obtient : $d_{i,j}^{(k+1)} = \min\left(d_{i,j}^{(k)};\ d_{i,k+1}^{(k)} + d_{k+1,j}^{(k)}\right)$.
Le résultat recherché correspond à $k = n-1$.

Pseudo-code

Pour tout $(\mathtt{i},\mathtt{j}) \in [\![0; \mathtt{n}-1]\!]$ faire :
  $\mathtt{d}_{\mathtt{ij}}^{(-1)} \longleftarrow \begin{cases} \mathtt{0 \text{ si } i = j} \ ; \ \omega(\mathtt{i},\mathtt{j}) \ \mathtt{si\ i \neq j\ et\ } (\mathtt{i},\mathtt{j}) \in \mathtt{E} \\ +\infty \ \mathtt{sinon} \end{cases}$
  Pour $\mathtt{k} = 0$ à $\mathtt{n}-1$ faire :
    Pour $(\mathtt{i},\mathtt{j}) \in [\![0; \mathtt{n}-1]\!]$ faire :
      $\mathtt{d}_{\mathtt{ij}}^{(\mathtt{k})} \longleftarrow \min\left(\mathtt{d}_{\mathtt{ij}}^{(\mathtt{k}-1)};\ \mathtt{d}_{\mathtt{ik}}^{(\mathtt{k}-1)} + \mathtt{d}_{\mathtt{kj}}^{(\mathtt{k}-1)}\right)$
  Renvoyer $\left(\mathtt{d}_{\mathtt{ij}}^{(\mathtt{n}-1)}\right)_{\mathtt{i,j}}$

2.2.2 Algorithme de Dijkstra

On suppose que $G = (V, E, \omega)$ est un graphe pondéré avec $\omega : E \to \mathbb{R}^+$.
On cherche les longueurs des plus courts chemins d'un sommet s appelé source vers tous les autres sommets de G.

Principe On adapte l'algorithme de parcours en largeur permettant de trouver les plus courts chemins dans un graphe *non pondéré* $\rightarrow$ l'utilisation d'une file nous permettait d'explorer les sommets dans l'ordre de leur distance à s croissante.
Pour Dijkstra, on parcourt également les sommets dans l'ordre de leur distance à s.
À chaque étape, on dispose de surestimations des longueurs des plus courts chemins depuis s, que l'on note $d(u)$ (pour $u \in V$). On notera $\delta(s,u)$ la longueur d'un plus court chemin de s à u.
On manipule un ensemble $S \subseteq V$ de sommets qu'il reste à explorer et dont on ne connaît pas encore la distance à s.

Pseudo-code

```
d(s) ← 0 ; d(u) ← +∞
S ← V
Pour k = 0 à |S| − 1 faire :
        u ← Argmin{d(v), v ∈ S}
        Enlever u de S
        Pour tout v voisin de u faire :
                Si d(v) > d(u) + ω(u, v) :
                        d(v) ← d(u) + ω(u, v)
Renvoyer d
```