Lycée Louis-le-Grand

TD 2 février 2005

Problème dit *union-find*

option informatique

1 Position du problème

On considère un ensemble fini X, de cardinal n. On supposera dans la suite que $X = \{0, 1, 2, \ldots, n-1\}$.
On considère une partition de X, qui va évoluer au fil du temps. Au départ, il s'agit de la partition la plus fine, constituée des singletons $\{i\}$, pour $0 \leqslant i \leqslant n-1$.
On se propose d'effectuer les opérations suivantes sur la structure :

- compter le nombre de parties qui constituent la partition ;
- dire si deux éléments x et y de X sont ou pas dans une même partie ;
- regrouper en une seule partie les parties qui contiennent deux éléments x et y.

2 Solution naïve

Donner une implémentation en mettant simplement à jour un vecteur `v` tel que `v.(x)` contient l'élément minimal de la partie qui contient x.
On écrira les fonctions suivantes :

```
initialise : int -> int vect
compte : int vect -> int
teste : int vect -> int  -> int -> bool
regroupe : int vect -> int -> int -> unit
```

On évaluera le coût de chacune de ces opérations.

```
let initialise n =
  let v = make_vect n 0 in
    for i = 0 to n-1 do v.(i) <- i done ;
    v ;;

let compte v =
  let n = vect_length v
  and compteur = ref 0
  in
    for i = 0 to n - 1 do if v.(i) = i then incr compteur done ;
    !compteur ;;

let teste v x y =
  v.(x) = v.(y) ;;

let regroupe v x y =
  let j = v.(y) and k = v.(x) in
  let j' = min j k and k' = max j k in
    for i = 0 to vect_length v - 1 do
      if v.(i) = k' then v.(i) <- j' done ;;
```

Le test coûte $O(1)$, le regroupement $O(n)$, et le comptage $O(n)$ également.

3 Utilisation d'arbres

On peut représenter une partition par une forêt d'arbres. Par exemple la partition $\{0, 1, 5\}, \{2, 3\}, \{4\}$ pourra être représentée par l'une des forêts ci-dessous :

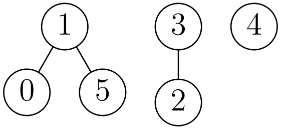

Figure 1 Une première forêt possible

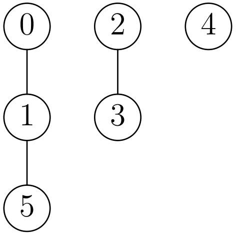

Figure 2 Une deuxième forêt possible

On représentera ce genre de forêt par un couple `(taille,père)`, où `taille` conservera le nombre de parties de la partition, c'est-à-dire le nombre d'arbres de la forêt, et `père.(x)` renverra `x` si `x` est lui-même la racine de son arbre, et sinon la valeur de son père dans son arbre.
Ainsi la structure utilisée est du type

```
type structure = { mutable taille : int ; pere : int vect } ;;
```

Écrire pour cette nouvelle implantation les fonctions habituelles :

```
initialise : int -> structure
compte : structure -> int
teste : structure -> int  -> int -> bool
regroupe : structure -> int -> int -> unit
```

```
let initialise n =
  let f = { taille = n ; pere = make_vect n 0 } in
    for i = 0 to n-1 do f.pere.(i) <- i done ;
    f ;;

let rec racine f x =
  if f.pere.(x) = x then x else racine f f.pere.(x) ;;

let compte f = f.taille ;;

let test f x y =
  (racine f x) = (racine f y) ;;

let regroupe f x y =
  let px = racine f x and py = racine f y in
    if px <> py then
      begin
       f.taille <- f.taille - 1 ;
       f.pere.(py) <- px
      end ;;
```

4 Une première amélioration : surveiller son poids !

On cherche évidemment à limiter la profondeur des arbres de la forêt. Pour cela, on se propose de maintenir à jour un nouveau vecteur, `poids`, tel que `poids.(x)` contienne la taille du sous-arbre de racine `x`. Au départ ce vecteur est plein de 1.

Lors d'un regroupement, on fusionnera les deux arbres de telle sorte que l'on place à la racine du nouvel arbre la racine de poids le plus élevé des deux arbres.

Cela conduira à utiliser une structure du type

```
type structure =
   { mutable taille : int ; pere : int vect ; poids : int vect } ;;
```

Réécrire les fonctions habituelles avec cette nouvelle approche.

```
let initialise n =
  let f = { taille = n ; pere = make_vect n 0 ;
                         poids = make_vect n 1 } in
    for i = 0 to n-1 do f.pere.(i) <- i done ;
    f ;;

let rec racine f x =
  if f.pere.(x) = x then x else racine f f.pere.(x) ;;

let compte f = f.taille ;;

let teste f x y =
  (racine f x) = (racine f y) ;;
```

```
let regroupe f x y =
  let px = racine f x and py = racine f y in
    if px <> py then
      let wx = f.poids.(px) and wy = f.poids.(py) in
       f.taille <- f.taille - 1 ;
       if wy < wx then
         begin
           f.pere.(py) <- px ;
           f.poids.(px) <- wx + wy
         end
       else
         begin
           f.pere.(px) <- py ;
           f.poids.(py) <- wx + wy
         end ;;
```

La compression des chemins On modifie enfin notre approche en remarquant que l'on n'a aucun besoin de connaître le vrai père de chaque élément : il serait bien plus efficace de diriger `père.(x)` sur la **racine** de l'arbre auquel appartient `x`. Ainsi, à chaque recherche de la racine, on mettra à jour les valeurs du tableau `père` des éléments qu'on passe en revue en remontant d'un élément `x` à la racine.

Schématiquement, on peut représenter cela par une compression des arbres. Supposant qu'on ait dans la forêt l'arbre de gauche de la figure suivante. La recherche de l'élément 20 transforme le tableau `père` de telle sorte que le dessin de l'arbre est maintenant celui de droite.

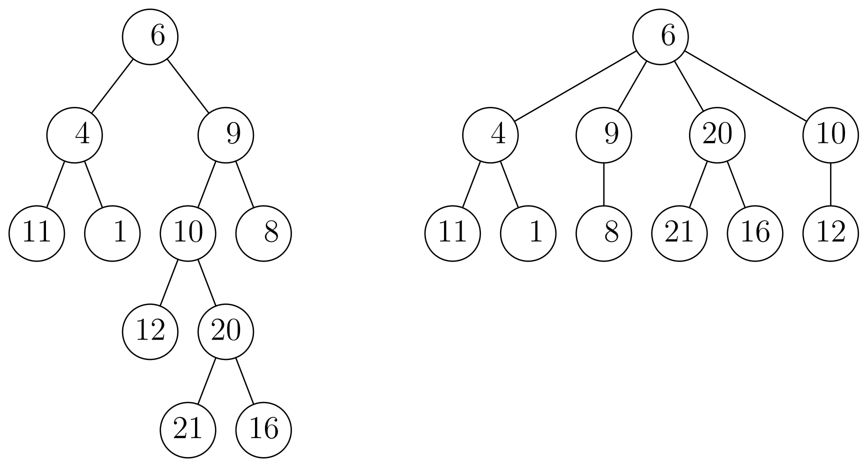

Figure 3 Les arbres avant et après compression des chemins

Réécrire les fonctions habituelles en opérant cette *compression des chemins*.

La seule fonction qui nécessite d'être réécrite est le calcul de la racine.

```
let racine f x =
  let rec ancetre x =
    if x = f.pere.(x) then x else ancetre f.pere.(x)
  in
  let p = ancetre x in
  let rec compression x =
    let y = f.pere.(x) in
      if x <> y then (f.pere.(x) <- p ; compression y)
  in
    compression x ;
    f.pere.(x) ;;
```

5 Un mot de conclusion

On montre que le coût de n opérations constituées d'un test d'égalité et d'une fusion est $O(n\alpha(n))$ où α est une fonction de n qui croît extrêmement lentement : $\alpha(n) \leqslant 2$ si $n < 2^{65536}$.