TD 5 : Un peu d'algorithmique

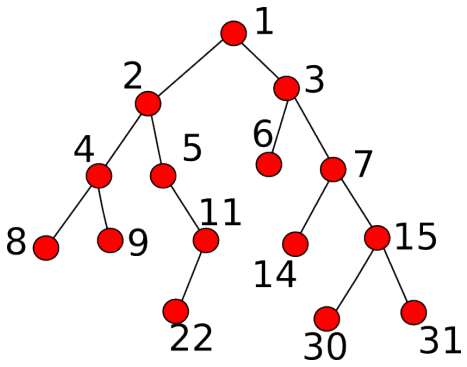
Décembre 2012

Des arbres un peu différent

On a vu dans le TD sur la récursion que l'on pouvait représenter des arbres uniquement avec des listes. Cette fois ci on va utiliser une autre présentation des arbres à partir de tableaux. Pour rappel, un tableau est une structure de donnée indexée par des entiers et représentant des cases contigües dans la mémoire. On s'attend donc à pouvoir accéder à n'importe quelle case en $\mathcal{O}(1)$ mais à devoir recopier tout le tableau lorsque l'on veut modifier sa taille.

Cas des arbres binaires

Un arbre binaire est un arbre qui comporte au plus deux fils à chaque nœuds. En fait, si on regarde un arbre binaire...

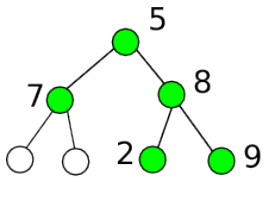


...on remarque que l'on peut numéroter chaque nœud u par un entier n_u tel que si n_p, n_g et n_d sont les entiers respectivement associés au parent, au fils gauche et au fils droit (quand ceux-ci existent) alors on a :

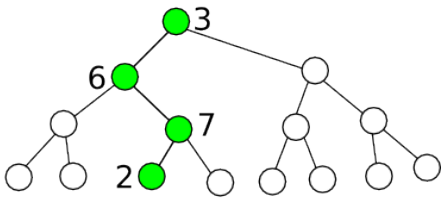
$$\begin{cases} n_p = \left\lfloor \frac{n_u}{2} \right\rfloor \\ n_g = 2 \times n_u \\ n_d = 2 \times n_u + 1 \end{cases}$$

avec $n_u = 1$ si u est la racine.

On peut alors représenter un tel arbre dans un tableau de taille 2^l où l est la profondeur de l'arbre, c'est à dire le maximum de la distance entre un nœud et la racine. Chaque case du tableau contient la clé du nœud qu'elle représente ou le symbole `null` si le nœud n'est pas dans l'arbre. Voici deux exemples d'arbres où les nœuds non-existant sont représenté en blanc.



Arbre A



Arbre B

En plaçant, chaque nœud u de l'arbre à sa position n_u définit plus haut, on obtient les tableaux suivant :

```
arbre_a := Array([5, 7, 8, null, null, 2, 9]);
arbre_b := Array([3, 6, null, null, 7, null, null, null, null, 2]);
```

On va donc représenter un nœud dans un arbre par une paire `(t, n)` où `t` est un tableau contenant un arbre binaire et `n` un indice dans ce tableau indiquant la position du nœud considéré.

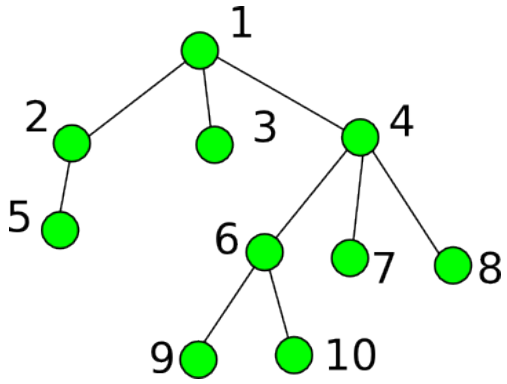
Exercice 1

1. *Implémenter les fonctions* `parent := proc(t, n)`, `left := proc(t, n)` *et* `right := proc(t,n)` *qui renvoient respectivement le parent, le fils gauche et le fils droit du nœud* `(t,n)`*. On ne cherchera pas à vérifier l'existence de ces nœuds.*
2. *Implémenter les fonctions* `is_root := proc(t,n)`, `has_left_child := proc(t,n)` *et* `has_right_child := proc(t, n)` *qui vérifient respectivement si le nœud* `(t,n)` *est la racine, a un fils gauche ou a un fils droit.*

Ce qu'on vient de faire pour les arbres binaires se généralise pour des arbres d'arité k quelconque en posant que le $i^{\text{ème}}$ fils du nœud u d'indice n_u est à la position $k \times n_u + i$ et le parent de u à la position $\left\lfloor \frac{n_u}{k} \right\rfloor$. Il faut cependant garder en tête, que la place en mémoire nécessaire pour représenter un arbre d'arité k et de profondeur l sera en $\mathcal{O}(k^l)$.

Des arbres à l'envers

Dans la prochaine section, on va utiliser des arbres sur tableau un peu particuliers : chaque nœud ne s'intéressera qu'à son parent et pas du tout à ses fils. Dans ce cas, on a plus besoin de borner l'arité des nœuds. En effet, chaque nœud va seulement conserver son parent et sa clé comme données. Ainsi l'arbre suivant



sera représenté comme un tableau de paire `(p, k)` où `p` est l'indice du parent et `k` la valeur de la clé, ce qui donne :

```
invert_tree := Array([ [1,1], [1,2], [1,3], [1,4],
                       [2,5], [4,6], [4,7], [4,8],
                       [6,9], [6,10] ]);
```

La racine qui n'a pas de parent garde son propre indice comme parent.

Exercice 2

Implémentez les fonctions `is_root` *et* `parent` *comme précédemment.*

Remarquez aussi que rien ne nous empêche de loger deux arbres distinct dans le même tableau avec cette représentation. On a donc plutôt une structure de forêt (collection d'arbres).

Union-find

Maintenant qu'on a pas mal bidouillé avec des arbres, on va se lancer dans notre premier algorithme digne de ce nom.

On considère une collection de n objets que l'on veut rassembler en ensembles disjoints. On veut fabriquer une structure de donnée qui gère ces ensembles que l'on peut voir comme les classes d'équivalence d'une certaine relation (d'équivalence) sur nos n objets. On veut ensuite pouvoir faire les opérations suivantes sur la structure de donnée :

- ▷ Créer une nouvelle classe
- ▷ Fusionner deux classes
- ▷ Obtenir un représentant canonique de la classe d'un élèment (par canonique, on entend que deux élèments de la même classe ont le même représentant tant que celle-ci n'est pas modifiée)

La représentation que l'on choisit pour cette structure est d'utiliser un arbre pour chaque classe. Nos objets seront les nœuds de ces arbres et l'ensemble des classes formera une forêt. La structure utilisée pour l'union-find sera donc implémentée par un tableau comme dans la section précédente.

Version naïve

On commence par la gestion de la taille de la structure d'union-find. Pour simplifier la suite, on va considérer que la structure que l'on crée retient le nombre d'élèment qu'elle contient et la taille du tableau, il s'agira donc d'une liste à 3 élèments `[t,k,n]` où `t` est un tableau de taille `n` contenant `k` élèment.

Exercice 3

1. *Implémmentez une fonction* `create_union_find := proc(n)` *qui crée une structure d'union-find de taille n vide.*
2. *Implémentez une fonction* `resize := proc(uf)` *qui renvoie une structure d'union-find contenant les mêmes objets mais de taille $2n$ où* `n = uf[3]` *est la taille de la structure originale.*

Pour ajouter un objet dans la structure, il faut qu'il y ait la place nécessaire (c'est à dire $k < n$ avec les notations ci-dessus) et créer une classe pour cet objet qui sera donc un arbre à lui tout seul.

Exercice 4

Implémentez la fonction `create_set := proc(uf,x)` *qui crée une classe pour* `x` *dans la structure d'union find* `uf` *puis renvoie la pair* `(uf,i)` *où* `i` *est l'indice de* `x` *dans la structure d'union-find.*

Le représentant canonique que l'on va choisir pour une classe est la racine de l'arbre implémentant la classe.

Exercice 5

Implémentez la fonction `get_witness := proc(uf,i)` *qui cherche le représentant canonique de l'élèment d'indice* `i`*.*

Ensuite, il faut pouvoir fusionner deux classes. Cela se fait en ajoutant l'arbre d'une des classes comme fils d'un nœud de l'autre classe.

Exercice 6

Implémentez la fonction `merge := proc(uf, i, j)` *qui fusionne les classes des éléments d'indices* `i` *et* `j`*.*

Il faut maintenant faire le point sur ce qu'on vient de programmer. On a bien obtenu une structure remplissant les contraintes du cahier des charges mais quelle est son efficacité ?

Exercice 7

Estimez la complexité des fonctions `create_set`*,* `get_witness` *et* `merge` *que vous venez d'écrire. En particulier, quelle est la complexité dans le pire des cas d'une série de n* `create_set`*, suivie de $n-1$* `merge` *pour fusionner toutes les classes nouvellement créées ?*

Deux heuristiques étonnantes

Dans cette partie, on va prendre deux idées qui permettent d'améliorer grandement la complexité de nos opérations. Dans notre implémentation précédente, la source des problèmes provenait du fait que nos arbres pouvait être de profondeur linéaire en le nombre de données qu'il comportent.

L'union par rang Notre première amélioration va consister à faire "le bon choix" lorsque l'on fait l'union de deux classes. Pour cela, on pourrait conserver explicitement la profondeur de l'arbre correspondant à chaque classe, puis lorsque l'on veut fusionner deux arbres, on "branche" la racine de l'arbre le plus petit sur la racine de l'arbre le plus grand. Cependant, le calcul et le maintien de la profondeur réelle de l'arbre risque de coûter trop cher en terme de calcul. On va donc plutôt maintenir une approximation, le *rang* d'un arbre qui va majorer sa profondeur.

Les régles pour manipuler le rang sont les suivantes : une nouvelle classe composée d'un seul élèment a pour rang 1, puis lorsque l'on fusionne deux classes `x` et `y` on suit le pseudo-code suivant :

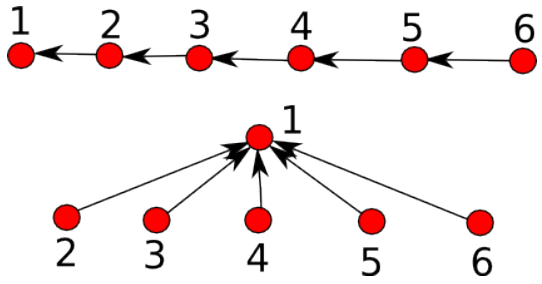
```
Si rang[x] > rang[y] Alors brancher y sur x
Sinon si rang[y] > rang[x] Alors brancher x sur y
Sinon
   rang[x] <- rang[x] + 1
   brancher y sur x
```

Le rang n'est utilisé que pour un nœud qui est racine d'un arbre. En pratique, on va conserver le rang comme une clé, c'est à dire comme un champ supplémentaire à coté du parent et de la valeur du nœud.

Exercice 8

Rajoutez un champ correspondant au rang d'un nœud dans votre implémentation de l'union-find et adaptez votre code pour qu'il implémente l'union des chemins.

La compression des chemins La seconde heuristique se résume par le schéma suivant :



L'idée de cette heuristique est de rattacher tous les nœuds d'une branche à la racine lorsque l'on parcourt la branche et uniquement lorsqu'on la parcourt ! Cela se fait au niveau de la fonction `get_witness` ce qui permet de compresser les chemins d'accés d'un nœud jusqu'à la racine uniquement quand on le demande.

Exercice 9

Implémentez (ça nécessite pas plus de 2-3 lignes) l'idée présentée ci-dessus.

Analyse de complexité L'analyse de l'algorithme une fois modifié avec les deux heuristiques ci-dessus devient extrèmement complexe. On peut montrer que sa complexité amortie (c'est à dire sur n opérations successives) est bornée par du $\mathcal{O}(n\alpha(n))$ où $\alpha = Ack^{-1}$ est la fonction inverse d'Ackermann (cf. TD 4). La fonction αn diverge lorsque n tend vers l'infini mais est plus petite que 5 dans tous les emplois humainement envisageable (c'est à dire que $\alpha(10^{100}) \leq 5$). On a donc une complexité amortie principalement linéaire, à comparer avec notre complexité quadratique précédemment.

Bonus : Tas

Une structure de tas est une structure de donnée permettant d'extraire le maximum (tas-max) ou le minimum (tas-min) d'un ensemble de n objets (n nombres par exemples). Les fonctions attendues sur un tas sont :

- ▷ Créer un tas
- ▷ Insérer un élèment
- ▷ Obtenir le maximum
- ▷ Supprimer le maximum
- ▷ Faire la fusion de deux tas

L'implémentation proposée ici est celle des tas binaire : un tas est représenté par un arbre binaire complet sauf éventuellement à la profondeur maximale. Il doit vérifier la propriété de tas-min (respectivement de tas-max) qui est que les clés de tous les fils d'un nœud sont plus grandes (resp. petites) que sa propre clé.

Exercice 10

Implémentez des tas-min et analysez la complexité de vos fonctions. Si c'est trop facile, utilisez vos tas-min pour implémenter des file de priorité.