TD 05 – Analyse Amortie

Exercice 1. *Pile*

On considère une pile munie des opérations suivantes :
- $PUSH(S, x)$: empile un objet x sur la pile S
- $POP(S)$: dépile le sommet de la pile S et retourne l'objet dépilé
- $MULTIPOP(S, k)$: dépile au plus k objets de la pile S

Algorithm 1: $MULTIPOP(S, k)$

```
début
    tant que S ≠ ∅ et k ≠ 0 faire
        POP(S);
        k ← k − 1;
fin
```

1. Quelle est la complexité de chacune des 3 opérations ? En déduire avec la méthode globale (méthode de l'agrégat) le coût amorti pour une suite de n opérations $PUSH$, POP et $MULTIPOP$ sur une pile initialement vide.
2. Même question avec la méthode des acomptes.
3. Même question avec la méthode des potentiels.
4. On souhaite implémenter une file à l'aide de deux piles, de telle façon que le coût amorti des opérations $ENQUEUE$ et $DEQUEUE$ soit $O(1)$. Comment peut-on faire ?

Exercice 2. *Union-Find*

On considère le problème des ensembles disjoints : soit V un ensemble de n éléments muni d'une certaine partition P (pensez par exemple à V comme étant les sommets d'un graphe partitionné en composantes connexes). Nous allons nous intéresser à la méthode des Union-Find, qui, intuitivement, permet de reconstruire la partition en répondant uniquement à des questions du type "est-ce que x et y sont dans le même sous-ensemble de la partition ?". Plus formellement, on veut une structure de données permettant gérer dynamiquement des ensembles disjoints $V_1, \ldots, V_i, \ldots$, chacun ayant un élément distingué appelé *représentant* de V_i. Elle permettra les opérations suivantes :
- Creer-Ensemble(x) : Crée un ensemble ne contenant que x.
- Union(x, y) : Fusionne l'ensemble contenant x avec celui contenant y (et détruit ces deux-là)
- Trouver(x) : Renvoie le représentant de l'ensemble contenant x.

Le but est de réussir à implémenter une telle structure de données de telle sorte qu'une suite de n opérations Creer-Ensemble et $m - n$ opérations Union ou Trouver coûte un temps quasi-linéaire en m (le nombre total d'opérations).

Pour obtenir cela, chaque ensemble V_i sera représenté par un arbre [1] : chaque noeud contient un élément x de V_i, et possède un pointeur vers son père (Noter que père$(x) = x$ ssi x est la racine de l'arbre), et on définit le représentant de V_i comme étant la racine. Il est maintenant facile de voir comment effectuer simplement les trois opérations :
- Creer-Ensemble(x) crée un arbre constitué du seul noeud x. On mémorisera en même temps que le *rang* de x, qui représente un majorant de la hauteur de l'arbre enraciné en x, vaut 0.
- Trouver(x) parcourt l'arbre depuis le noeud x et remonte jusqu'à trouver la racine, et renvoie son étiquette. Les noeuds rencontrés lors de cette opération constitue le *chemin de découverte* de x, et la complexité de cette opération vaut la longueur de ce chemin.
- Union(x,y) : on effectue un appel à Trouver(x) et un appel à Trouver(y) pour obtenir les représentants x' et y' des ensembles contenant x et y. Puis on effectue l'opération Lier(x', y') qui va "accrocher" l'arbre enraciné en x' comme fils de y', ou le contraire. Pour réduire le coût des opérations Trouver, on essaye d'équilibrer les arbres.

1. pas forcément binaire

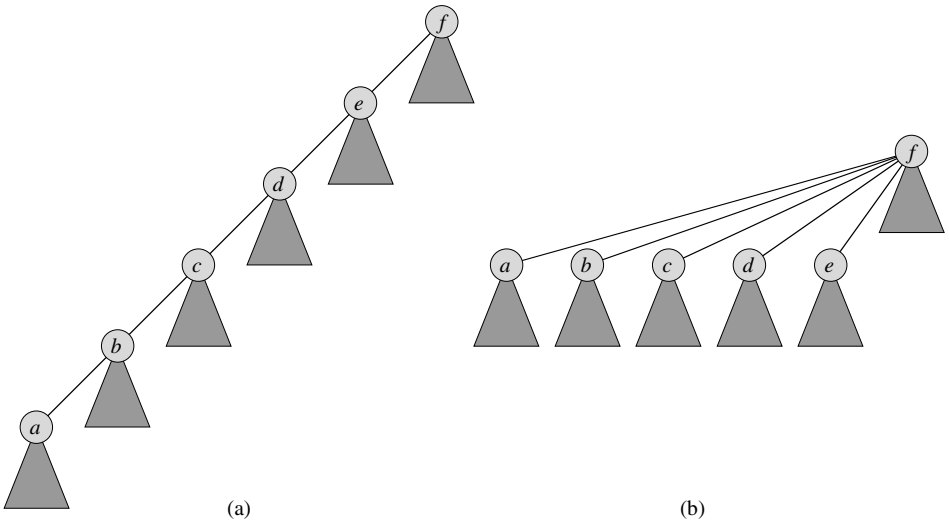

FIGURE 1 – Illustration de la compression de chemin : (a) Avant, (b) après

1. Écrire le pseudo-code de l'opération LIER.

2. Montrer qu'ainsi le coût d'une suite de n opérations CRÉER-ENSEMBLE puis $m - n$ opérations UNION ou TROUVER coûte $\mathcal{O}(m \log n)$.
Indice : On commencera par borner les rangs des éléments par $\log(k)$ où k est le nombre d'éléments dans l'arbre considéré.

Pour améliorer cette complexité, on va utiliser une heuristique de *compression de chemin* : à chaque appel de TROUVER(x), qui parcourt le chemin $x = x_0, x_1, \ldots, x_k$ de x à la racine x_k, on va transformer l'arbre de manière à ce que tous les x_i deviennent fils directs de la racine (pour $0 \leq i < k$), voir Figure 1. Ainsi, le coût de TROUVER est multiplié par une constante s, mais le coût du prochain appel de TROUVER sur un de ces noeuds x_i ou leurs descendants sera considérablement réduit. Il nous faut donc effectuer une analyse amortie pour prouver l'efficacité de cette méthode. Nous utiliserons la méthode du potentiel.

Définissons tout d'abord la *fonction d'Ackermann*, connue pour croître extrêment rapidement :

$$\text{Pour } k \geq 0, j \geq 1, \qquad A_k(j) = \begin{cases} j+1 & \text{si } k = 0\ , \\ A_{k-1}^{(j+1)}(j) & \text{si } k \geq 1\ , \end{cases}$$

où $A_{k-1}^{(j+1)}(j)$ indique la fonction A_{k-1} itérée $j+1$ fois sur j. Cette fonction croît strictement par rapport à j et k, et très rapidement : $A_3(1) = 2047$ et $A_4(1) \gg 10^{80}$!!
On peut alors définir l'*inverse* de la fonction d'Ackermann :

$$\alpha(n) = \min\{k : A_k(1) \geq n\}\ .$$

En particulier pour $n \leq 10^{80} \ll A_4(1)$, on a $\alpha(n) \leq 4$. L'inverse de la fonction d'Ackermann est donc une fonction qui croît *très très* lentement, et que l'on peut presque considérer comme constante pour des valeurs de n raisonnables.

But : Prouvons qu'une suite de n opérations CREER-ENSEMBLE et $m - n$ opérations LIER ou TROUVER a un coût $\mathcal{O}(\alpha(n)m)$, c'est-à-dire quasi-linéaire.

On commencera par remarquer que la valeur de $rang(x)$ commence à o et augmente au cours du temps jusqu'à ce que x ne soit plus une racine. Ensuite, $rang(x)$ ne change plus et $rang(x) < rang(\text{père}(x))$.
On peut alors définir deux fonctions auxiliaires $niveau(x)$ et $iter(x)$ définies pour chaque noeud x qui n'est ni la racine, ni une feuille :

$$niveau(x) = \max\{k : rang(\text{père}(x)) \geq A_k(rang(x))\} .$$

$$iter(x) = \max\{i : rang(\text{père}(x)) \geq A^{(i)}_{niveau(x)}(rang(x))\} .$$

3. Montrer que $0 \leq niveau(x) < \alpha(n)$ et $1 \leq iter(x) \leq rang(x)$. Comment peuvent évoluer $niveau(x)$ et $iter(x)$ au cours du temps ?
On peut maintenant définir la fonction de potentiel après q itérations par : $\Phi_q = \Sigma_{x \in V} \phi_q(x)$, où :

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot rang(x) & \text{si } x \text{ est une racine ou si } rang(x) = 0, \\ (\alpha(n) - niveau(x)) \cdot rang(x) - iter(x) & \text{si } x \text{ n'est pas une racine et si } rang(x) \geq 1 . \end{cases}$$

4. Prouver que pour tout noeud x et toute valeur de q, on a

$$0 \leq \phi_q(x) \leq \alpha(n) \cdot rang(x)$$

5. Considérons la q-ième opération, et supposons qu'il s'agit soit de LIER, soit de TROUVER. Soit x un noeud qui n'est pas racine. Prouvez que :

1. $\phi_q(x) \leq \phi_{q-1}(x)$.
2. Si $rang(x) \geq 1$ et si $niveau(x)$ ou $iter(x)$ est modifié, alors $\phi_q(x) \leq \phi_{q-1}(x) - 1$.

6. Montrez que le coût amorti [2] de chaque opération CRÉER-ENSEMBLE est $\mathcal{O}(1)$.

7. Montrez que le coût amorti de chaque opération LIER(x, y) est $\mathcal{O}(\alpha(n))$. Sans perte de généralité, on supposera que y devient le père de x.
Indice : Commencez par montrer que seul y peut voir son potentiel augmenter.

8. Montrez que le coût de chaque opération TROUVER-ENSEMBLE est $\mathcal{O}(\alpha(n))$.

9. Conclure.

Note : Cet exercice est très fortement inspiré de *Introduction to Algorithms*, T. Cormen, C. Leiserson, R. Rivest, C. Stein (section *21.3 : Forêts d'ensembles disjoints)*.

2. Rappel : il s'agit de (coût réél $+\Phi_q - \Phi_{q-1}$).