Union-Find

I Structure de données abstraite d'ensembles disjoints

I.1 Objectif et définition

On cherche à réaliser une structure permettant de manipuler des ensembles disjoints de manière dynamique, c'est-à-dire en fusionnant des ensembles à la volée.

La structure abstraite union-find est ainsi définie par :

- un type `'a uf` pour manipuler la collection d'ensembles dont les élements sont de type `'a`.
- une opération `makeset` qui étant donné un élèment x va rajouter $\{x\}$ à la collection.
- une opération `find` qui associe à chaque élément x, l'ensemble qui le contient dans la collection.
- une opération `union` qui fusionne deux ensembles disjoints, le plus souvent donnés par des éléments qu'ils contiennent, donc implicitement en utilisant `find`.

I.2 Exemple emblématique d'utilisation

On considère le problème déjà vu du calcul des composantes connexes d'un graphe, mais avec le graphe donné à la volée, arête après arête, par exemple lors d'un téléchargement.

La solution qu'on connait consiste à itérer des parcours quelconques d'un sommet non découvert, à chaque parcours on calcule une nouvelle composante connexe. Cet algorithme est ainsi en $O(|S| + |A|)$, donc linéaire, ce qui est a priori très satisfaisant.

Cependant, si on rajoute une arrête, on change la nature des composantes et il est nécessaire de refaire un parcours.

Avec une structure union-find, on peut procéder ainsi :

- on appelle `makeset` pour chaque sommet
- pour chaque arête (x, y) où $find(x) \neq find(y)$, on appelle $union(x, y)$.

Si jamais une nouvelle arête est à considérer, il suffit d'appeler `union`.

Pour n éléments dans tous les ensembles, si on suppose que la complexité est est $m(n)$ pour `makeset`, $u(n)$ pour `union` et $f(n)$ pour `find`, on a alors une complexité du calcul en $O(m(|S|)|S| + (f(|S|) + u(|S|))|A|)$.

Pour que cet algorithme soit intéressant,il faut que cette complexité soit proche de la précédente. Cela signifie que les trois fonctions devraient être en $O(1)$ ou presque.

I.3 Classe d'équivalence et représentant

On peut définir une relation d'équivalence sur les éléments en posant :

$$x \sim y \iff find(x) = find(y)$$

Les ensembles sont alors des classes d'équivalence dont il suffit de donner un représentant pour les retrouver. Ainsi, `find` pourra se contenter de renvoyer un représentant **sous réserve** que `find(x)` et `find(y)` renvoie le même quand $x \sim y$.

> **Remarque** Comme on l'a vu en mathématiques, on peut toujours représenter une classe de congruence modulo p avec un représentant dans $[\![0, p-1]\!]$.

II Implémentation naïve : listes chaînées

!susbsection(Première implémentation)

On peut représenter chaque élément par un maillon et chaque ensemble par une liste chaînée. Le dernier maillon de chaque liste correspondra alors à l'élément distingué qui représente l'ensemble.

Plus précisément :

- `makeset` crée un maillon $O(1)$
  - ⋆ pour pouvoir manipuler directement des maillons depuis leurs valeurs, il peut être nécessaire de maintenir un dictionnaire permettant d'associer à une valeur son maillon, on peut le faire en $O(1)$ avec des tables de hâchages.
- `find` va descendre le long de la liste correspondant à l'ensemble pour déterminer le maillon final. En pire cas, il faut parcourir toute la liste, c'est donc en $O(n)$ où n est le nombre total d'éléments.
- `union` doit réaliser une concaténation de listes, ce qui est en $O(n)$ en pire cas.

II.1 Quick-find

On peut améliorer les performances de `find` en rajoutant un pointeur direct depuis chaque maillon vers le maillon final de la liste qui les contient. L'opération `find` est alors en $O(1)$. Cependant `union` sera en $O(n)$, même en faisant en sorte d'avoir une concaténation rapide comme expliqué dans le paragraphe suivant, car il faut maintenir les pointeurs vers le maillon final.

Cette implémentation est en général dénommée `Quick-find` car `find` est rapide.

II.2 Quick-union

On peut faire en sorte que l'union soit rapide en faisant en sorte que le maillon distingué pointe sur le premier et dernier maillon de chaque liste. Ainsi, pour l'union, il suffit de faire un nombre constant d'opérations.

Attention, cela suppose ici que `union` ne va pas réaliser des `find`, qui eux, resteront coûteux.

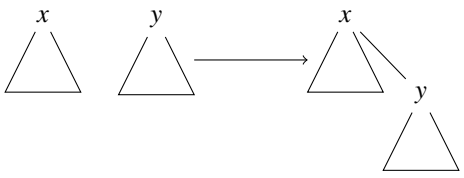
III Implémentation efficace

III.1 Forêt disjointe

On va représenter une structure union-find par une forêt d'arbres dont les étiquettes sont toutes distinctes. Chaque arbre représentera alors l'ensemble des valeurs de ses étiquettes et la forêt sera naturellement une représentation d'une collection d'ensembles disjoints.

On peut alors identifier une valeur au nœud de l'arbre dont c'est l'étiquette et naïvement effectuer les opérations :

- `makeset` crée un arbre avec une feuille et le rajoute à la forêt, ce qui fait en temps constant en supposant que la forêt est une liste d'arbres
- `find` va remonter du nœud vers sa racine et renvoyer sa racine, ce qui est en $O(h)$ où h est la hauteur de la forêt, c'est-à-dire la plus grande hauteur des arbres qu'elle contient.
- `union` va supprimer deux arbres et les remplacer par un unique arbre :



Ainsi, `union` est naturellement en $O(1)$ sous réserve de partir des racines. On retrouve donc une complexité meilleure que Quick-union avec les listes chaînées.

Cependant, il est possible d'avoir des arbres très profonds et en pire cas, `find` est en $O(n)$ où n est le nombre d'éléments.

Remarque Si on considère les entiers de 0 à n, la suite d'union $(1,0),(2,0),\dots,(n,0)$ va créer un peigne avec n à la racine et une unique branche $n--(n-1)---\cdots---2--1--0$ donc 0 est à profondeur n et ainsi $find(0)$ devra remonter les n arêtes.

III.2 Union pondérée

Quand on réalise une union, on a deux choix sur qui sera la racine de l'arbre fusionné. Si on s'assure de toujours faire le choix qui minimise la hauteur, on doit permettre à la forêt d'être de hauteur en $O(log(n))$.

Calculer les tailles ou les hauteurs de chaque arbre n'est pas utile, on peut se contenter de définir le rang d'un arbre noté $rg(a)$:

- au départ, chaque feuille est de rang 0
- quand on réalise une union entre deux arbres a et a'
  - ⋆ si $rg(a) > rg(a')$ on n'a pas intérêt à placer a plus profondément donc on rajoute a' comme enfant de a et on préserve les rangs
  - ⋆ sinon, on fait la fusion dans l'autre sens et dans le cas où les deux rangs étaient égaux, on augmente le rang de a' de 1.

Ainsi, les rangs indiquent le nombre de fusions successifs qui ont augmenté la profondeur et permettent ainsi de calculer sans peine la hauteur de l'arbre.

En procédant ainsi, on s'assure effectivement que la fusion soit en $O(\log n)$ en moyenne.

III.3 Compression de chemins

La structure de l'arbre est souvent inintéressante, on peut donc se permettre de la supprimer et d'aplatir les arbres. Pour cela, si la parenté est déterminé par un tableau `parent` avec `parent[x] = x` à la racine, on peut procéder récursivement pour `find` :

Calcul de `find(x)`

- Si x $\neq$ `parent[x]`
  - ⋆ `parent[x] = find(parent[x])`
- Renvoyer `parent[x]`

Ainsi, la valeur de `parent[x]` est toujours mise à jour vers la racine directement. Cela à revient à aplatir l'arbre pour que chaque nœud soit enfant de la racine. On parle de *compression de chemins* puisque les chemins sont tous triviaux.

On peut prouver ainsi que `find` est en $O(\alpha(n))$ en moyenne où $\alpha(n)$ est une fonction qui croit tellement lentement que $\alpha(n) \leq 4$ pour $n \leq 10^{80}$.

IV Implémentation en C avec un tableau parent

Remarque Cette implémentation peut également se faire facilement avec `OCaml`.

On va considérer que les élèments sont les entiers de 0 à $n-1$ et on va représenter la forêt par un tableau `parent` où `parent[x]`indique le père de x si x n'est pas une racine, et `parent[x] = x` pour les racines.

Remarque On peut donc tester si un élèment est la racine d'un arbre en $O(1)$.

On va représenter la forêt par le type suivant :

```c
struct union_find {
    int *parent;
    int *rang;
    int nelements;
};
typedef struct union_find union_find;
```

Pour commencer, on crée une forêt de feuilles qui sont toutes de rang 0.

```c
union_find uf_makesets(int n)
{
    union_find uf;
    uf.parent = malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++)
```

```
    {
        uf.parent[i] = i;
        uf.rang[i] = 0;
    }
    uf.nelements = n;
    return uf;
}
```

On implémente alors directement `find` et `union`, en prenant garde au fait que `union` est un mot-clé en C :

```c
int uf_find(union_find uf, int x)
{
    if (uf.parent[x] != x)
        uf.parent[x] = uf_find(uf, uf.parent[x]);
    return uf.parent[x];
}
```

```c
void uf_union(union_find uf, int x, int y)
{
    int rx = uf_find(uf, x);
    int ry = uf_find(uf, y);
    assert(rx != ry);

    if (uf.rang[rx] > uf.rang[ry])
        uf.parent[ry] = rx;
    else {
        uf.parent[rx] = ry;
        if (uf.rang[rx] == uf.rang[ry])
            uf.rang[ry] = uf.rang[ry] + 1;
    }
}
```