

STRUCTURE UNIR & TROUVER

(UNION-FIND)

L'objectif de ce chapitre est d'étudier une structure de données spécifiquement adaptée à la gestion de partitions, i.e. de parties disjointes d'un ensemble : la structure *union-find* (que le programme traduit par "unir & trouver", mais cette formulation est très peu usitée même en Français).

I - Introduction au problème

On va définir une structure de données pour le **problème des classes disjointes**. Ce problème consiste à maintenir dans une structure de données une partition d'un ensemble fini, c'est à dire un découpage en sous-ensembles disjoints que l'on appelle des **classes**. On souhaite :

- pouvoir déterminer si deux éléments appartiennent à la même classe
- réunir deux classes en une seule

d'où le nom de *union-find*.

Formalisons un peu l'objet à représenter : Soit E un ensemble (fini) d'éléments de type quelconque, de cardinal n . Sans perte de généralité, on suppose que l'ensemble E à partitionner est celui des n entiers $\llbracket 0, n - 1 \rrbracket$ (quitte à les énumérer). Pour une partition $X = X_1 \cup X_2 \cup \dots \cup X_k$ de l'ensemble E et pour un élément $x \in E$, on note $X(x)$ la partie X_i contenant x .

Autrement dit, de manière équivalente, $X(x)$ est la **classe d'équivalence** de x pour la relation d'équivalence \sim définie par : $x \sim y$ si $X(x) = X(y)$.

Une partie $X(x)$ sera généralement donnée sous la forme d'un **représentant canonique** de la classe.

On souhaite maintenant créer une structure de données pour représenter une partition de E (i.e. ses classes d'équivalence), et plus particulièrement une partition destinée à devenir de plus en plus grossières.

On définit le **type abstrait Union-find** de la manière suivante :

type Union-Find (UF) :

utilise : Entier



Opérations	Effets
CREATE : Entier \rightarrow UF	
FIND : UF \times Entier \rightarrow Entier	
UNION : UF \times Entier \times Entier \rightarrow Rien	

Voyons maintenant plusieurs structures concrètes pour implémenter ce type abstrait. On cherche la structure la plus efficace possible.

II - Structures naïves et complexités

II.1 - Composantes connexes d'un graphe

Une première façon naïve de faire est de représenter une partition de E sous la forme d'un graphe.

 **Attention!** Cette section est à copier sur feuille depuis le tableau. Ceci n'est qu'une annexe de la section. 

Dans cette section, on représente un graphe $G = (S, A)$ par matrice d'adjacence.

Exercice :

Proposer des algorithmes pour effectuer les opérations de l'union-find. On pourra les décrire (succinctement) en Français ou en pseudo-code.

1. CREATE(n)
2. UNION(G, i, j)
3. FIND(G, i)

À faire sur feuille!

Complexités : Analysons la complexité des opérations avec cette implémentation de l'union-find. On rappelle que $n = |E|$.

À faire sur feuille!

On remarque donc que cette implémentation favorise l'opération union, au détriment de l'opération find.

Remarques :

- Ici, on peut créer des cycles dans nos composantes connexes si on appelle union sur deux éléments x et y d'une même composante connexe. Pour l'éviter, on peut commencer l'union en vérifiant que x et y sont dans des classes différentes avec deux find. Mais on perd son exécution en temps constant.
- Si l'on s'assure qu'on n'unit que des composantes distinctes, ou que l'on fait l'hypothèse que union n'est utilisée que sur des classes différentes, alors on maintient l'invariant : « le graphe est acyclique » tout au long des opérations union et find.

II.2 - Tableau de représentants

Principe :

Un représentant canonique d'une classe est alors n'importe quel élément de la classe, au choix (mais c'est le **même pour toute la classe**).

Exemple : $E = \{3\} \cup \{0, 2, 5, 8\} \cup \{1, 6\} \cup \{4, 7, 9\}$.

Un tableau représentant possible pour représenter E est : (entourer en rouge les représentants canoniques choisis dans E)

Exercice : Algorithmes et complexités

Proposer des algorithmes pour effectuer les opérations create, find et union, et analyser leur complexité. On pourra les décrire (succinctement) en Français ou en pseudo-code.

À faire sur feuille!

On remarque donc que cette implémentation favorise l'opération find, au détriment de l'opération union.

III - Forêt d'arbres

On peut faire bien mieux que du $O(n)$ pire cas sur les opérations `union` et `find` en utilisant des arbres.

III.1 - Sans optimisation

Principe général :

Exemple : $E = \{3\} \cup \{0, 2, 5, 8\} \cup \{1, 6\} \cup \{4, 7, 9\}$.

Forêt d'arbres correspondante (possible) :

Rappels : Vous avez vu en première année deux façons principales de représenter les arbres sous la forme d'un tableau¹. Rappelons-les.

- Si l'arbre est **binaire complet**, on peut remplir le tableau avec les noeuds de l'arbre lus **dans l'ordre d'un parcours en largeur**.

Les enfants du noeud d'indice i se trouvent alors aux indices :

La racine se trouve à l'indice 0 et est son propre parent.

Sinon, le parent du noeud d'indice i se trouve à l'indice :

Exemple :

Cette implémentation est communément utilisée pour représenter :

- Si les noeuds ont **des étiquettes dans** $\llbracket 0; n - 1 \rrbracket$, alors la case d'indice i représente le noeud étiqueté par i et on y stocke **le parent du noeud** i (le parent de la racine est elle-même).

Exemple :

Remarques :

- C'est la deuxième implémentation par tableaux que nous utiliserons dans tout le reste de ce chapitre pour nos structures ***union-find***.
- Cette implémentation est particulièrement adaptée lorsque que l'on souhaite pouvoir accéder facilement au parent d'un noeud, sans vouloir facilement accéder à ses enfants. C'est le cas ici, on cherche à remonter aux représentants d'une classe, donc on veut seulement pouvoir facilement accéder à la racine.

1. Ce ne sont bien sûr pas les seules façons de représenter des arbres en informatique. On peut penser par exemple aux types sommes récursifs en OCaml, ou aux cellules avec pointeurs en C.

On adapte facilement cette structure à une forêt d'arbres plutôt qu'un arbre seul. Une forêt d'arbres est implémentée par un tableau `parents` tel que :

- `parents[i] = i` si et seulement si i est l'une des racines
- sinon, `parents[i]` donne le père de i

Exemple :

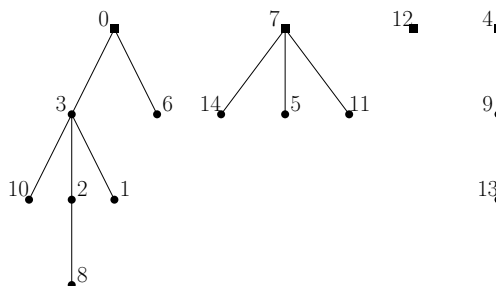


FIGURE 1 – La partition $\{\{0, 1, 2, 3, 6, 8, 10\}, \{5, 7, 11, 14\}, \{12\}, \{4, 9, 13\}\}$ de $[0 \dots 14]$, et une forêt associée.

Exercice : Écrire le tableau `parents` correspondant à cette forêt d'arbres :

Remarque :

- La structure de données précédente (tableau de représentants) est un cas particulier de forêt d'arbres. Dessiner la forêt d'arbres correspondant au tableau de représentants donné plus haut pour $E = \{3\} \cup \{0, 2, 5, 8\} \cup \{1, 6\} \cup \{4, 7, 9\}$:

Exercice :

Proposer des algorithmes pour effectuer les opérations `create`, `find` et `union`. On pourra les décrire (succinctement) en Français ou en pseudo-code.

À faire sur feuille !

Complexité amortie

Pour l'étude de complexité, on va s'intéresser à la complexité amortie : à partir d'une partition en n singletons, on effectue une série de N opérations (qui peuvent être des `FIND` ou des `UNION`). Ce qui nous intéresse n'est pas le coût maximal d'une opération prise individuellement, mais le coût maximal total S de ces N opérations : on appellera **complexité amortie d'une opération** la valeur maximale de S/N .

Rappels : ⚠ Nous avons déjà rencontré cette notion de complexité amortie, mais il n'est pas forcément inutile d'insister à nouveau sur la différence entre complexité amortie et complexité en moyenne.

- Pour la complexité en moyenne, **la moyenne porte sur les entrées possibles** (et il faut donc disposer d'une distribution de probabilités sur ces entrées). On ne dit rien de la complexité dans le pire cas.
- Pour la complexité amortie, la moyenne est une **moyenne temporelle, dans le pire cas**. Indépendamment des entrées, on *garantit* une borne supérieure sur **le coût moyen d'une opération lors d'une série de N opérations**.

Analyse de la complexité amortie avec une forêt d'arbre :

Montrons que la complexité amortie d'une opération peut être de l'ordre de n (pire cas) avec la structure proposée.

À faire sur feuille !

Pour l'instant, on n'a pas réussi à faire mieux que les structures naïves précédentes.

III.2 - Union par hauteur

On voit facilement que :

- le coût d'un appel à UNION est le même (à un facteur deux près) que celui d'un appel à FIND ;
- le coût d'un appel à FIND est proportionnel à la profondeur du nœud sur lequel on fait l'appel.

Il est donc naturel de chercher à obtenir les arbres les plus « plats » possibles. Lors d'une opération UNION, la racine de l'un des arbres devient un fils de la racine de l'autre arbre. On peut remarquer qu'on est parfaitement libre de choisir quel racine joue quel rôle. Une heuristique naturelle est alors de procéder à une **union par hauteur** :

Principe général :

On utilise à présent le type OCaml suivant :

OCaml

```
1 type partition = {  
2   parents : int array;  
3   heights : int array  
4 }
```

On maintient l'invariant suivant : `heights.(i)` contient la hauteur du sous-arbre enraciné en i .

Exercice : Écrire les fonctions `create`, `find` et `merge` en OCaml, en utilisant le type ci-dessus et avec l'heuristique de l'union par hauteur.

À faire sur feuille !

Proposition 1

En utilisant la méthode de l'union par hauteur, la complexité dans le pire cas d'une opération FIND (et donc UNION) est en $O(\log n)$.

Démonstration. Sur papier! □

Remarque :

- Si c'est vrai de la complexité dans le pire cas, c'est *a fortiori* vrai de la complexité amortie.

III.3 - Compression de chemin

Comme nous l'avons déjà dit, on gagne à avoir les arbres les plus plats possibles : l'arbre idéal pour nous serait celui dans lequel tous les nœuds sont directement attachés à la racine. On va profiter des appels à FIND pour optimiser nos arbres.

Principe général :

Cette optimisation est appelée **compression de chemin**.

Exemple :

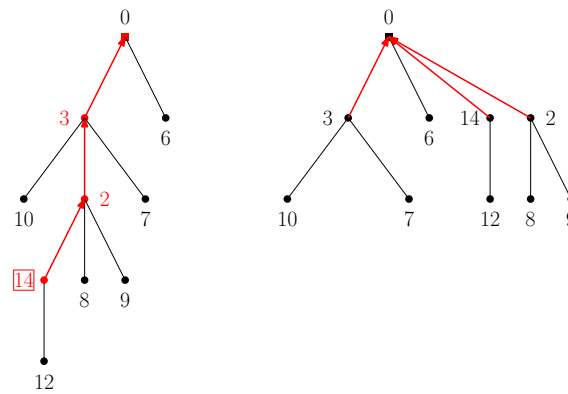


FIGURE 2 – Effet secondaire de l'opération $\text{FIND}(p, 14)$ avec compression de chemin.

Exercice :

Écrire une nouvelle version de la fonction `find` utilisant la compression de chemin, en OCaml.

Indication : on utilisera une version récursive de `find`.

À faire sur feuille!

On peut légitimement se demander si la compression de chemin est compatible avec l'union par hauteur : en effet, chaque appel à `find` peut à présent modifier la hauteur de l'arbre concerné². Rien n'empêche cependant de continuer à utiliser cette heuristique : on parlera simplement d'**union par rang** plutôt que d'union par hauteur. Le rang est défini comme la hauteur qu'aurait le nœud si l'on n'avait fait aucune compression de chemin. On continue donc à le calculer exactement comme on le faisait pour la hauteur, et il s'agit à présent d'un **majorant** de la « vraie » hauteur. Il n'y a donc rien à changer à la fonction d'union (mais il est plus logique de renommer le champ `heights` en `ranks`) :

OCaml

```
1 type partition = {
2   parents : int array;
3   ranks : int array
4 }
```

Complexité amortie avec union par rang et compression de chemin

Définition 2

Logarithme itéré :

Le **logarithme itéré**, noté \log^* , est défini pour $x \in \mathbb{R}$ par :

$$\log^* x = \begin{cases} 0 & \text{si } x \leq 1 \\ 1 + \log^*(\log_2 x) & \text{si } x > 1 \end{cases}$$

Remarques :

- On utilise ici le logarithme en base 2, ce qui est le plus courant puisque \log^* ne se rencontre quasiment qu'en informatique. Rien n'empêche cependant de prendre une autre base (*e* par exemple).
- Le principe du logarithme itéré, c'est qu'il tend vers $+\infty$, mais *sans se presser*.

Exercice :

1. On définit ${}^k x = \underbrace{x^{x^{\cdot^{\cdot^{\cdot}}}}}_k$ (avec ${}^0 x = 1$). Montrer que pour $k \in \mathbb{N}^*$, on a $\log^* n = k$ si et seulement si $n \in]{}^{k-1} 2, {}^k 2]$.
2. Quelle est la plus grande valeur de x pour laquelle on a $\log^* x \leq 3$? $\log^* x \leq 4$? $\log^* x \leq 5$?

À faire sur feuille!

On constate donc que \log^* peut en pratique être considéré comme une (petite) constante : $\log^* n$ ne dépasse 5 que pour des valeurs *ridiculement* grandes de n .

Proposition 2

Une série de m opérations sur une structure *union-find* correspondant à un ensemble de n éléments se fait en temps $O(m \log^* n)$. Ainsi, la complexité amortie d'une opération est en $O(\log^* n)$, et peut être considérée

2. Seule la hauteur de la racine de l'arbre a une importance.

comme constante en pratique.

Démonstration. En TD (exercice). □

Remarques :

- Ce résultat n'est pas à connaître précisément, mais il faut savoir que : la complexité amortie est « constante en pratique ».
- (Culture générale) La complexité énoncée ici est en réalité une majoration grossière : on peut montrer qu'on a en fait du $O(\alpha(n))$, où α est une forme d'inverse de la fonction d'Ackermann, qui croît *incomparablement plus lentement* que \log^* . La fonction d'Ackermann est définie récursivement par :

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

et la fonction α est essentiellement définie comme $\alpha(n) = \min\{k \mid \text{Ack}(k, k) > n\}$ (il en existe plusieurs versions selon les sources, qui restent fondamentalement les mêmes). On n'étudiera pas cette complexité amortie optimale des opérations `union` et `find`, qui est admise par le programme.

- On peut définir des fonctions qui croissent plus vite encore. Une d'entre elles est la fonction *busy beaver*, définie à l'aide des machines de Turing, qui a la propriété de grandir asymptotiquement strictement plus vite que n'importe quelle fonction calculable. En revanche, elle n'est donc pas elle-même calculable.

IV - Applications

Cette année, nous verrons principalement deux problèmes qui nécessitent une structure *union-find* pour être implémentés correctement :

- **Calcul d'un arbre couvrant minimal par l'algorithme de Kruskal.**³
- Calcul d'une coupe minimal (d'un graphe) par l'algorithme de Karger⁴.

3. Rendez-vous au prochain chapitre!

4. Probablement votre prochain DM.