Union Find : des structures de données

Julie Parreaux

2018-2019

Référence du développement : Cormen [1, p.519]

Leçons où on présente le développement : 901 (Structures de données).
Leçons où on peut l'évoquer : 921 (Recherche) ; 925 (Graphes) ; 926 (Analyse en complexité).

1 Introduction

On souhaite une structure de donnée qui permet de faire une partition dans l'ensemble des données et de connaître rapidement à quelle partie de la partition se trouve un élément donnée. Les structures de données Union Find en sont capable. À partir du type abstrait contenant les opérations create, union et find, nous pouvons construire deux implémentations distinctes afin de faire deux structures de données Union Find. Une première consiste à représenter la partition de l'ensemble de départ à l'aide d'une liste chaînée. La seconde quant-à elle représente la partition à l'aide d'une forêt d'arbre.

Dans ce développement nous allons étudier le type abstrait muni de ces deux implémentations possibles. Puis pour chacune d'entre elles nous donnerons quelques heuristiques permettant d'améliorer leur efficacité (nous ne montrons aucun résultats sur la complexité). Enfin, nous donnerons une application de cette structure de données : l'algorithme de Kruskal permettant de calculer un arbre couvrant minimal.

Remarques sur le développement

Ce développement présente une structure de données : il faut donc appuyer sur son implémentation "réelle". Dans ce développement, il n'y a pas de preuves à proprement parler, cependant, il faut garder à l'esprits les preuves de complexités qui n'ont rien de triviales.

1. Présentation des enjeux de cette structure de données.
2. Présentation d'un premier type concret : les listes chaînées.
3. Présentation d'un deuxième type concret : la forêt d'arbres.
4. Présentation de l'algorithme de Kruskal.

2 Présentation de la structure de donnée Union Find

Objectif : Regrouper n éléments dans une collection de d'ensembles disjoints sur lesquels on souhaite trouver l'unique ensemble disjoint contenant un élément (find) ou réunir deux ensembles (union).
Opérations :

— create(x) : créer une nouvelle partie dont x est le seul membre unique si x déjà dans la structure : échec ;

— find(x) : donne un pointeur vers l'unique élément de x ;

— union(x, y) : associe les deux parties de x et de y (avec un choix de nouveau représentant)

Hypothèse : Cette structure est construite en parallèle des données (elle ne contient que des copies). Les données et leurs copies sont relier par un pointeur dans les deux sens. Donc, trouver un élément dans la structure de données a une complexité constante.

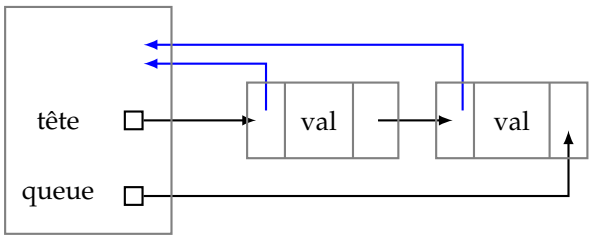

FIGURE 1 – Liste chaînée représentant une partie de la partition dans union find.

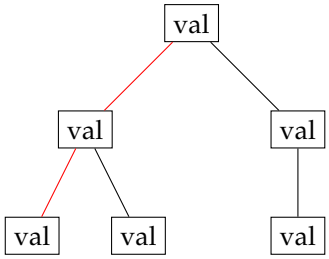

FIGURE 2 – Forêt d'arbre (contenant un seul arbre) représentant la partition d'union find (contenant une seule partie).

Type concret 1 : les listes chaînées Le premier type concret qui va nous donner la première structure de données est un ensemble de listes chaînées. La liste possède deux pointeurs : un qui pointe sur le premier élément et un autre qui pointe sur le dernier. De plus, tous les éléments de la listes ont un pointeur qui va jusqu'à la structure (faire un dessin au tableau, figure 1).

Complexités pour cette structure :

create	$O(1)$	(créer un nombre fixe de pointeur)
find	$O(1)$	(suivre (montrer) les pointeurs bleus de la figure 1)
union	$O(l)$	où l est la longueur de la liste (recopier toute la liste)

Heuristique : Union pondérée : on ajoute un attribut longueur dans la liste et on colle la plus petite liste à la plus grande liste (moins de pointeurs à recopier).

Type concret 2 : la forêt d'arbre Une deuxième implémentation de ce type abstrait est une forêt d'arbre un peu particulière. En effet, dans cette forêt, les arc des arbres sont orientés vers la racine qui porte une boucle (faire un dessin au tableau, figure 2).

Complexités pour cette structure :

create	$O(1)$	(créer un nombre fixe de pointeur)
find	$O(l)$	où l est la longueur de la liste (remonter le chemin de découverte en rouge figure 2)
union	$O(1)$	(modifier un pointeur)

Cette implémentation n'est pas plus efficace que les listes chaînées (on échange les difficultés de union et celle de find).

Heuristiques : Les heuristiques améliorent grandement la complexité.

- Union par rang : on fait pointer la racine contenant le moins de nœuds vers celle en contenant le plus (de profondeur la plus importante). On gère un rang qui majore la hauteur d'un nœud.
- Compression de chemin : on fait pointer les nœuds du chemin de découverte vers la racine (dessin).

Application à l'algorithme de Kruskal La structure Union Find est importante dans l'algorithme de Kruskal (elle garantie la correction et influe sur sa complexité). L'algorithme de Kruskal permet de calculer un arbre couvrant minimal pour un graphe connexe pondéré

3 Étude de la complexité de cette structure

On souhaite une structure de donnée qui permet de faire une partition dans l'ensemble des données et de connaître rapidement à quelle partie de la partition se trouve un élément donnée.

Objectif : Regrouper n éléments dans une collection de d'ensembles disjoints sur lesquels on souhaite trouver l'unique ensemble disjoint contenant un élément (find) ou réunir deux ensembles (union).

Opérations :

- create(x) : créer une nouvelle partie dont x est le seul membre unique si x déjà dans la structure : échec ;
- find(x) : donne un pointeur vers l'unique élément de x ;
- union(x, y) : associe les deux parties de x et de y (avec un choix de nouveau représentant)

Hypothèse : Cette structure est construite en parallèle des données (elle ne contient que des copies). Les données et leurs copies sont relier par un pointeur dans les deux sens. Donc, trouver un élément dans la structure de données a une complexité constante.

Type concret 1 : les listes chaînées Le premier type concret qui va nous donner la première structure de données est un ensemble de listes chaînées. La liste possède deux pointeurs : un qui pointe sur le premier élément et un autre qui pointe sur le dernier. De plus, tous les éléments de la listes ont un pointeur qui va jusqu'à la structure (faire un dessin).
Complexités pour cette structure :

create	$O(1)$	(créer un nombre fixe de pointeur)
find	$O(1)$	(suivre les pointeurs / montrer sur un dessin)
union	$O(l)$	où l est la longueur de la liste recopiée (recopier toute la liste)

Proposition (Complexité amortie sans heuristique)**.** *Avec ce type concret, la complexité amortie des opérations* create, find *et* union *dont n* create *on a donc n objets dans la structure de données est en $O(n)$.*

Démonstration. On applique la méthode de l'agrégat sur cette séquence d'opérations. On va alors exhiber une séquence de m opérations sur n objets (donc n opérations create). Comme, on a n, on peut faire au plus $(n-1)$ union. On pose alors $m = 2n-1$, la séquence est une suite de n opération create puis $n-1$ opérations union.

On dépense alors $O(n)$ pour faire les n create puis $\sum_{i=1}^{n-1} i = \Theta(n^2)$ pour faire les $(n-1)$ union. Comme le nombre d'opérations est $m = 2n-1$, la complexité amortie de chaque opération est $\Theta(n)$. □

Heuristique : Union pondérée : on ajoute un attribut longueur dans la liste et on colle la plus petite liste à la plus grande liste (moins de pointeurs à recopier).

Proposition (Complexité amortie avec l'heuristique)**.** *Avec ce type abstrait et cet heuristique, une séquence de m opérations* create, find *et* union *parmi lesquelles il y a n opérations* create *consomme un temps $O(m + n \log n)$.*

Démonstration. On applique la méthode de l'agrégat sur cette séquence de m opérations.

Étape 1 : on compte le nombre d'union. On effectue au plus $n-1$ unions car il y a n objets et qu'on réunit toujours deux ensembles.

Étape 2 : on majore la nombre de mise-à-jour de pointeurs vers l'objet liste, pour tout objet. Lors d'une première mise-à-jour, on obtient une liste d'au moins deux objets. Lors d'une deuxième mise-à-jour, on obtient une liste d'au moins quatre objets. On montre, par récurrence, pour tout $k \neq n$, après $\lceil \log k \rceil$ mise-à-jours de x contient au moins k membres. Donc un élément x est mise-à-jour au plus $\lceil \log n \rceil$ fois si la collection comporte n objets.

On en déduit que le temps d'exécution total de la fonction union est $O(n \log n)$ (car $O(n \log n)$ manipulations de pointeurs, $O(1)$ pour la mise-à-jour de l'attribut longueur et $O(1)$ pour la mise-à-jour du pointeur sur le dernier élément).

Étape 3 : on évalue la complexité des m opérations. Le temps total pour m opérations est $O(m)$ car il y a au plus m create et find. D'où le résultat.

□

Type concret 2 : la forêt d'arbre Une deuxième implémentation de ce type abstrait est une forêt d'arbre un peu particulière. En effet, dans cette forêt, les arc des arbres sont orientés vers la racine qui porte une boucle (faire un dessin).
Complexités pour cette structure :

create	$O(1)$	(créer un nombre fixe de pointeur)
find	$O(l)$	où l est la longueur de la liste (remonter le chemin de la découverte / dessin)
union	$O(1)$	(modifier un pointeur)

Cette implémentation n'est pas plus efficace que les listes chaînées (on échange les difficultés de union et celle de find).
Heuristiques : Les heuristiques améliorent grandement la complexité.

— Union par rang : on fait pointer la racine contenant le moins de nœuds vers celle en contenant le plus (de profondeur la plus importante). On gère un rang qui majore la hauteur d'un nœud.

— Compression de chemin : on fait pointer les nœuds du chemin de découverte vers la racine (sans modifier le rang) (dessin).

Implémentation de la gestion du rang :

create	$rang = 0$
find	$rang$ inchangé
union	si $rang$ égaux on l'incrémente ; sinon, il reste inchangé

Lemme 1. *La fonction rang est une fonction monotone croissante.*

Arguments de la preuve. Par récurrence directe sur le nombre d'opérations. □

Lemme 2. *Chaque nœud à un rang au plus égal à $n-1$ pour une collection à n objets.*

Démonstration. Le rang de chaque nœud commence à 0 (lors de l'opération create) puis ne peut augmenter que lors d'une union. Comme il y a au plus $n-1$ union (car il y a n objets), tous les rangs valent au plus $n-1$. □

Implémentation de la gestion de la compression de chemin : méthode en deux passes (elle est récursive) :

— récurse : passe dans le chemin de découverte pour trouver le représentant.

— revient en arrière : remet à jour les pointeurs en allant en arrière.

Pour définir la complexité amortie de cette structure de données, on commence par définir la fonction α qui est l'inverse de la fonction d'Ackermann. Pour cela, on rappelle l'expression de la fonction $A_k(j)$ qui pour tout entiers k, j vaut

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

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

Théorème. *Avec ce type abstrait et cet heuristique, une séquence de m opérations* create, find *et* union *parmi lesquelles il y a n opérations* create *consomme un temps $O(m\alpha(n))$ dans le cas défavorable.*

Démonstration. On applique la méthode du potentiel.

Lemme 3. *L'union par rang ne change pas sa complexité.*

Démonstration. Si on convertit une séquence S' de m' opérations de textsfcreate, find et union en une séquence S de m opérations de textsfcreate, find et union-rang en transformant chaque opérations union en deux opérations find et union-rang. Alors si S est en $O(m\alpha(n))$, alors S' est en $O(m'\alpha(n))$ et $m = O(m')$. □

Fonction potentielle : $\phi_q(x)$ pour chaque nœud x après q opérations. Le cumul des potentiel des noeuds donne alors le potentiel de la forêt complète : $\Phi_q = \sum_x \phi(x)$ où Φ_q désigne le potentiel de la forêt après q opérations. Elle est définie comme suit :

$$\Phi_q(x) = \begin{cases} \alpha(n) * x.rang & \text{si } x \text{ est une racine ou si } x.rang = 0 \\ (\alpha(n) - \text{niveau}(x))x.rang - \text{iter}(x) & \text{si } x \text{ n'est pas une racine et si } x.rang > 0 \end{cases}$$

On définie alors $\text{niveau}(x) = \max\{k : x.p.rang \geq A_k(x.rang)\}$ où $x.p$ est le père de x. Elle représente alors le plus grand k tel que A_k appliqué au rang de x est inférieur au rang du père de x. On définie également $\text{iter}(x) = \max\{i : x.p.rang \geq A^{(i)}_{\text{niveau}(x)}(x.rang)\}$. Elle définie le plus grand nombre de fois que l'on peut appliquer $A_{\text{niveau}(x)}$ au rang de x tel qu'il soit inférieur au rang du père de x.

Lemme 4. *Pour tout nœud x et pour toute valeur de q, on a : $0 \leq \Phi_q(x) \leq \alpha(n) * x.rang$.*

Arguments de la preuve. — Si x est racine ou si $x.rang = 0 : 0 \leq \Phi_q(x) = \alpha(n) * x.rang$ (déf).

— Sinon on obtient des majorant pour niveau et iter.
  ⇒ Montrons que $0 \leq \text{niveau}(x) < \alpha(n)$.
    → $0 \leq \text{niveau}(x) : x.p.rang \geq x.rang + 1 = A_0(x.rang)$ lemme 1
    → $\text{niveau}(x) < \alpha(n) : A_{\alpha(n)}(x.rang) \geq A_{\alpha(n)}(1) \geq n > x.p.rang$ (lemme 2)
    → *Remarque* : on a la croissance de niveau par celle de *rang*.
  ⇒ Montrons que $1 \leq \text{iter}(x) \leq x.rang$
    → $1 \leq \text{iter}(x) : x.p.rang \geq A_{\text{niveau}(x)}(x.rang) = A^{(1)}{}_{\text{niveau}(x)}(x.rang)$
    → $\text{iter}(x) \leq x; rang : A^{(}x.rang + 1)_{\text{niveau}(x)}(x.rang) = A_{\text{niveau}(x)+1}(x.rang) > x.p.rang$
  ⇒ Montrons que $0 \leq \Phi_q(x) \leq \alpha(n) * x.rang$.
    → $0 \leq \Phi_q(x) : \Phi_q(x) \geq (\alpha(n) - (\alpha(n) - 1) * x.rang - x.rang = 0$ (majoration)
    → $\Phi_q(x) \leq \alpha(n)x.rang : \Phi_q(x) \leq (\alpha(n) - 0)x.rang - 1 < \alpha(n)x.rang$ (minoration)

On remarque de plus que si x n'est pas une racine et que $x.rang > 0$, alors $\Phi_q(x) < \alpha(n)x.rang$. □

Pour appliquer la méthode su potentiel, il nous faut connaître l'impacte des différentes opérations sur le potentiel. Cela nous donnera le coût amortie de chacune des opérations.

Lemme 5. *Soit x un nœud qui n'est pas une racine et supposons que la $q^{ième}$ opération est Union ou Find. Alors le potentiel de x ne peut pas croître, et si $x.rang \geq 0$ avec iter(x) ou niveau(x) qui change, alors le potentiel de x diminue d'au moins* 1.

Arguments de la preuve.
— x n'est pas racine donc son *rang* ne change pas ; et n ne change pas également. Donc si $x.rang = 0$, alors $\Psi_q(x) = \Psi_{q-1}(x) = 0$.
— Supposons que $x.rang \geq 1$ et on rappelle que niveau(x) est monotone croissante .
  ⇒ Supposons que niveau(x) ne change pas alors iter(x) croît ou ne change pas : deux cas.
    → iter(x) ne change pas : $\Phi_q(x) = \Phi_{q-1}(x)$
    → iter(x) croît (elle croît d'au moins 1) : $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$
  ⇒ Supposons que niveau(x) croît d'au moins 1 et iter(x) pourrait décroître.
    → Le terme $(\alpha(n) - \text{niveau}(x))x.rang$ diminue d'au moins $x.rang$.
    → La décroissance de iter(x) est d'au plus $x.rang - 1$ (borne du lemme 4).
    → Augmentation par iter est inférieure à la diminution par niveau : $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$

□

On s'occupe maintenant des coûts amorties à proprement parler.

Lemme 6. *Le coût amortie de chaque opération Créer est $O(1)$.*

Arguments de la preuve. Supposons que la $q^{\text{ième}}$ opération soit Créer.
— On crée un nœud de $rang = 0$, tel que $\phi_q(x) = 0 : \Phi_q = \Phi_{q-1}$
— Coût réel : $O(1)$

□

Lemme 7. *Le coût amortie de chaque opération Union est $O(\alpha(n))$.*

Arguments de la preuve. Supposons que la $q^{\text{ième}}$ opération soit Union(x, y) où y devient le parent de x.
— Coût réel : $O(1)$
— Calcul du changement de potentiel :
  ⇒ Seul le potentiel de x, de y et des enfants de y peut changer.
  ⇒ Montrons que seul le potentiel de y peut croître.
    → Enfants de y : potentiel ne peut augmenter (lemme 5)
    → Nœud x : racine en $q - 1$ donc $\phi_{q-1}(x) = \alpha(n)x.rang$

- Si $x.rang = 0$, alors $\phi_{q-1}(x) = \phi_q(x) = 0$.
- Sinon $\phi_q(x) < \alpha(n)x.rang = \phi_{q-1}(x)$ (lemme 4) .

⇒ Montrons que cette augmentation est majorer par $\alpha(n)$

→ y racine en $q-1$: $\phi_{q-1}(y) = \alpha(n)y.rang$

→ y racine en q et $x.rang$ accroît d'au plus 1 : $\phi_q(y) = \phi_{q-1}(y)$ ou $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$

— On en déduit le coût amortie : $O(1) + O(\alpha(n)) = O(\alpha(n))$

□

Lemme 8. *Le coût amortie de chaque opération Find est $O(\alpha(n))$.*

Arguments de la preuve. Supposons que la $q^{\text{ième}}$ opération soit Find avec un chemin de découverte contenant s nœuds.

— Coût réel : $O(s)$

— Calcul du changement de potentiel :

⇒ Montrons qu'aucun nœud voit son potentiel croître.

→ x est une racine : $\phi_q(x) = \alpha(n)x.rang$ ne change pas

→ x n'est pas une racine : lemme 5

⇒ Montrons qu'au moins $\max(0, s - (\alpha(n) + 2))$ nœuds voit leur potentiel décroître d'au moins 1.

→ $x.rang > 0$ et $\exists y$ plus loin tel que $\text{niveau}(x) = \text{niveau}(y)$ avant Union.$x.rang > 0$ et $\exists y$ plus loin tel que $\text{niveau}(x) = \text{niveau}(y)$ avant Union.

- $x.rang > 0$ et $\exists y$ plus loin tel que $\text{niveau}(x) = \text{niveau}(y)$ avant Find.
- Pas la racine ni le dernier ni tous les derniers nœuds tels que $\text{niveau}(x) = k$ pour tout $k \in \{1, \dots \alpha(n) - 1\}$.

→ Nœud x qui satisfait la contrainte. Montrons que son potentiel décroît d'au moins 1.

- $k = \text{niveau}(x) = \text{niveau}(y)$
- Juste avant Find :

$$\begin{gathered} x.p.rang \geq A_k^{\text{iter}(x)}(x.rang) \\ y.p.rang \geq A_k(y.rang) \\ y.rang \geq x.p.rang \end{gathered}$$

  On obtient $y.p.rang \geq A_k^{i+1}(x.rang)$.
- Compression de chemin : $x.p.rang = y.p.rang$ sans rien changer : $\text{iter}(x)$ ou $\text{niveau}(x)$ augmente.

→ On conclut par le lemme 5

— On en déduit le coût amortie : $O(S) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$

□

On conclut la preuve en appliquant : le lemme 3, le lemme 6, le lemme 7 et le lemme 8. □

Application à l'algorithme de Kruskal L'algorithme de Kruskal permet de calculer un arbre couvrant minimal pour un graphe connexe pondéré

La structure Union Find est importante dans l'algorithme de Kruskal (elle garantie la correction et influe sur sa complexité, algorithme 1).
Complexité ; $O(|E|\log|E|)$ car le tri des arêtes s'effectue en $O(|E|\log|E|)$ et nous manipulons une structure d'union find sur $|E| + |V|$ opérations dont $|V|$ opérations create. On a donc une complexité en $O\left((|E| + |V|)\log(|E| + |V|)\right)$.
Or, comme $|E| \leq |V|^2$, on a $\log|E| = O(\log|V|)$. On obtient la complexité annoncée.

Algorithm 1 Algorithme de Kruskal calculant un arbre couvrant minimal pour un graphe connexe pondéré.

Entrée : $G = (V, E)$: graphe connexe ; w : fonction de poids
Sortie : E est la liste des arêtes de l'arbre

```
1:  function KRUSKAL(G, w)
2:      E ← ∅
3:      for tout v ∈ V do
4:          create(v)
5:      end for
6:      Trier E par ordre croissant sur w
7:      for tout (u, v) ∈ E do
8:          if find(u) ≠ find(v) then
9:              E ← E ∪ {(u, v)}
10:             union(u, v)
11:         end if
12:     end for
13:     Renvoie E
14: end function
```

Références

[1] Rivest R. Stein C. Cormen T., Leiserson C. *Algorithmique, 3ème édition*. Dunod, 2010.