Chapître XII.

Structures de données d'ensembles disjoints

(UNION-FIND data structures)

1. Structures de données avancées

-) tas binomial

-) tas de Fibonacci

-) structure de données d'ensembles disjoints

-) B-arbres

2. Opérations sur les ensembles disjoints

Une **structure de données d'ensembles disjoints (UNION-FIND data structure)** est une structure de données qui maintient à jour une collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ d'ensemble dynamique disjoints. Chaque ensemble est identifié par un **représentant**, qui est un certain membre de l'ensemble.

Une structure de données d'ensembles disjoints supporte les opérations suivantes.

-) $CREER - ENSEMBLE(x)$

-) $UNION(x, y)$

-) $TROUVER - ENSEMBLE(x)$

$CREER - ENSEMBLE(x)$ est une opération qui crée un nouvel ensemble dont le seul membre est donc le représentant est x. (Il faut que x ne soit pas déjà membre d'un autre ensemble.)

$UNION(x, y)$ réunit les ensemble dynamiques qui contiennent x et y dans un nouvel ensemble qui est l'union de ces deux ensemble. Le représentant de l'ensemble est un membre quelconque.

$TROUVER{-}ENSEMBLE(x)$ retourne un pointeur sur le représentant de l'ensemble contenant x.

3. Algorithme de KRUSKAL

Dans Chapître VII appelé “Arbres couvrants minimaux” nous avons regardé l’algorithme de KRUSKAL qui utilise une structure de données d’ensembles disjoints pour maintenir les composantes connexe de la fôret $G = (V, A)$.

On a considéré en TD quelques implémentations simples, en particulier une implémentation par listes chaînées.

4. Représentation des ensembles disjoints par listes chaînées

Une façon simple d'implémenter une structure de données d'ensembles disjoints utilise des listes chaînées:

-) Chaque ensemble est répresenté par une liste chaînée.

-) Le premier objet de chaque liste chaînée sert de représentant.

-) Chaque objet de la liste chaînée contient un élément de l'ensemble, un pointeur sur l'objet contenant l'élément suivant, et **un pointeur sur le représentant de l'ensemble**.

Analyse du temps:

Avec cette représentation en liste chaînée, $CREER-ENSEMBLE$, et $TROUVER-ENSEMBLE$ sont simples à implémenter, et consomment $O(1)$ au pire des cas.
L'implémentation la plus simple de l'opération $UNION(x,y)$, exécutée par concaténation de la liste de x à la fin de la liste de y, consomme beaucoup plus de temps, même $\Theta(n)$ au pire des cas.

Analyse amortie:

Pour l'analyse amortie d'une implementation nous considérons une séquence des opérations $CREER-ENSEMBLE$, $TROUVER-ENSEMBLE$, et $UNION$.

Nous ferons dépendre l'analyse de deux parametre:

-) n, le nombre d'opérations $CREER-ENSEMBLE$

-) m, le nombre total d'opérations $CREER-ENSEMBLE$**,** $TROUVER-ENSEMBLE$**, et** $UNION$**.**

Constatons que le nombre d'opérations $UNION$ est au plus égal à $n-1$.

Puisque les opérations $CREER-ENSEMBLE$ sont inclues dans le nombre total d'opérations m, on a $m \geq n$.

5. Une heuristique d'union pondé-rée

La représentation par listes chaînées demande $\Theta(m)$ en moyen par appel, puisqu'on pourrait concaténer la liste la plus longue à la plus courte; dans ce cas, il faut mettre à jour le pointeur sur le représentant pour chaque membre de la liste la plus longue.

heuristique d'union pondérée:

Supposons que chaque représentant contienne également la longueur de la liste, (qui peut se mettre à jour facilement) **on concaténe toujours la plus petite liste à la plus longue** - en cas d'égalité entre les longueurs, le choix est arbitraire.

Avec cette simple heuristique une seule opération $UNION$ peut prendre encore $\Theta(m)$ temps si les deux ensembles ont $\Theta(m)$ éléments.

Théorème. *Quand on utilise la représentation en liste chaînée pour les ensembles disjoints associée à l'heuristique de l'union pondérée, une séquence de* m *opérations* $CREER-ENSEMBLE$, $TROUVER-ENSEMBLE$, *et* $UNION$, *dont* n *opérations* $CREER-ENSEMBLE$, *s'exécute en* $O(m + n \log n)$.

Démonstration: On commence par calculer, pour chaque objet d'un ensemble de taille n, une borne supérieure sur le nombre de fois que son pointeur sur le représentant est mis à jour.

Considérons un objet x particulier. **On sait qu'à chaque fois que le pointeur de x sur son représentant est mis à jour, x se trouvait dans le plus petit ensemble.**

$\Rightarrow$

à la premiere mis à jour, l'ensemble resultant devait donc contenir au moins 2 éléments

à la deuxieme mis à jour, l'ensemble resultant devait donc contenir au moins 4 éléments

....

à la $i$eme mis à jour, l'ensemble resultant devait donc contenir au moins 2^i éléments

....

$\Rightarrow$

Pour un $k \leq n$ quelconque, après $\lceil \log k \rceil$ mises à jour du pointeur de x, l'ensemble résultant doit posséder au moins k éléments.

Comme l'ensemble le plus grand contient au plus n éléments, le pointeur de chaque objet sur son représentant a été mis à jour au plus $\lceil \log n \rceil$ fois au cours de toutes les opérations $UNION$.

Le temps total utilisé pour la mise à jour des n objets est donc $(n \log n)$.

Chaque opération $CREER-ENSEMBLE$ ou $TROUVER-ENSEMBLE$ s'exécute en $O(1)$, et il en existe $O(m)$.

$\Rightarrow$

La séquence entière s'exécute donc en $O(m + n \log n)$ au pire des cas. $\square$

6. Fôrets d'ensembles disjoints

Pour une implémentation plus efficace des ensembles disjoints, on représente les ensembles par des **arbres enracinés**.

fôret d'ensembles disjoints:

-) Chaque ensemble est répresenté par un arbre enraciné.

-) La racine de chaque arbre contient le représentant.

-) Chaque nœud contient un élément de l'ensemble et un **pointeur sur son père**.

-) Pour chaque racine le pointeur sur le père pointe sur la racine soi-même.

Implémentation des opérations:

$CREER\!-\!ENSEMBLE$ crée un arbre à un seul nœud.

$TROUVER - ENSEMBLE$ suit les pointeurs pères jusqu'à rencontrer la racine de l'arbre. Les nœuds visités sur ce chemin constituent la **route directe**.

$UNION$ fait pointer la racine d'un arbre sur la racine de l'autre.

Jusqu'à maintenant, nous n'avons pas fait mieux qu'avec la représentation en liste chaînée. Une séquence de $n - 1$ opérations $UNION$ peut créer un arbre qui se réduit à une chaîne libéaire de n nœuds.

Utilisant deux heuristiques on peut atteindre un temps d'exécution qui est presque linéaire par rapport au nombre total d'opérations m.

7. L'union par rang (union by rank)

Bien que l'opération $UNION$ soit fait par un changement d'un seul pointeur, on veut minimiser la hauteur de l'arbre pour obtenir un petit longueur de la route direct pour un nœud quelconque de l'arbre.

Au lieu de conserver explicitement la hauteur de chaque arbre (ou le profondeur de chaque nœud ou la taille du sous-arbre enraciné à chaque nœud), on utilise une approche qui facilite l'analyse.

Pour chaque nœud, on maintient à jour un rang(x) qui est une approximation du logarithme de la taille du sous-arbre enraciné à x, et qui est aussi une borne supérieure pour la hauteur du sous-arbre enraciné à x.

Heureusement il n'est pas necéssaire de mettre à jour la hauteur du sous-arbre enraciné à x, pour chaque nœud. **(stratégie paressée)**

$UNION$

1. La racine de moindre rang pointe sur celle de rang supérieur.

2. Si les deux racines ont le même rang, alors une de deux devient la nouvelle racine et leur rang est incrementée.

8. La compression de chemin (path compression)

C'est une vraie heuristique:
également **très simple** et **très efficace**.

Pendant chaque opération $TROUVER\text{–}ENSEMBLE$ **on fait le pointeur de chaque nœud de la route directe pointe sur la racine.**

9. L'implémentation d'opérations

$CREER - ENSEMBLE(x)$

$p[x] \leftarrow x$
$rang[x] \leftarrow 0$

$UNION(x, y)$

$LIER(TROUVER - ENSEMBLE(x),$
$\quad TROUVER - ENSEMBLE(y))$

$LIER(x, y)$

si $rang[x] > rang[y]$
  alors $p[y] \leftarrow x$
  sinon $p[x] \leftarrow y$
    si $rang[x] = rang[y]$
      alors $rang[y] \leftarrow rang[y] + 1$

$TROUVER-ENSEMBLE(x)$

si $x \neq p[x]$
  alors $p[x] \leftarrow TROUVER{-}ENSEMBLE(p[x])$
retourner $p[x]$

Deux phases (Two-pass method):

1. remonte le long de la route directe jusqu'à la racine

2. redescende la route directe pour mettre à jour chaque nœud de manière qu'il pointe directement sur la racine

Théorème. *Une séquence de m opérations $CREER{-}ENSEMBLE$, $UNION$ et $TROUVER{-}ENSEMBLE$, dont n opérations $CREER{-}ENSEMBLE$, peut être exécutée sur une fôret d'ensembles disjoints grâce à l'union par rang et la compression de chemin en temps $O(m \cdot \alpha(m,n))$ dans le pire des cas.*

Ce théorème de R. TARJAN est un résultat bien connu de l'Algorithmique bien que la démonstration soit très difficile.

Remarque. On appelle la fonction $\alpha(n,m)$ **l'inverse de la fonction d'Ackermann**.

$\alpha(n,m) = \min\{i \geq 1 : \ A(i, \lfloor m/n \rfloor) > \log n\}$

$\alpha(m,n) \leq 4$ pour tous les cas pratiques, par exemple $n \leq m \leq 10^{80}$.