Data Structures IV: Disjoint Set Union

Contenido

1. Definición de la estructura	⇨
2. Simple implementación	⇨
3. Small-to-large technique	⇨
4. Path Compression	⇨
5. Implementación óptima	⇨

Contenido

1. Definición de la estructura	⇨
2. Simple implementación	⇨
3. Small-to-large technique	⇨
4. Path Compression	⇨
5. Implementación óptima	⇨

Disjoint Set Union (DSU)

También llamada "small-to-large". Es una estructura de datos que permite mantener **grupos disjuntos** de cierto conjunto de elementos, manteniendo la información del grupo al que pertenece cada elemento y permitiendo **unir** grupos. Cada grupo será identificado por **un elemento representativo** del grupo.

En otras palabras, es una estructura que, dado n elementos inicialmente en grupos distintos, permite realizar las siguientes operaciones:

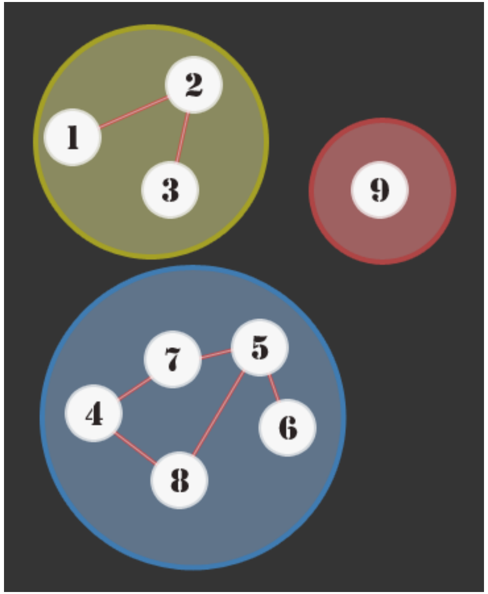
- $make\_set(x)$: Crea un nuevo grupo de un solo elemento
- $find(x)$: Retorna el **elemento representativo** del grupo al que pertenece x
- $union(x, y)$: Une el grupo de x con el grupo de y

Generalmente primero se usa el operador $make\_set$ para cada uno de los n elementos, de forma que se inicia con n grupos disjuntos de un solo elemento.

Nota: En C++ la palabra $union$ es una palabra reservada, por lo que convenientemente la reemplazaremos por $join$ para representar lo mismo.

Disjoint Set Union (DSU)

Una de las aplicaciones de esta estructura es en **grafos**, para saber si dos nodos pertenecen a la misma componente conexa, con la posibilidad de seguir agregando aristas al grafo.



Fuente: https://www.mathblog.dk/disjoint-set-data-structure/

Contenido

1. Definición de la estructura	➡
2. Simple implementación	➡
3. Small-to-large technique	➡
4. Path Compression	➡
5. Implementación óptima	➡

Implementación

Definimos un arreglo $p[u]$ que represente al elemento representativo del grupo en donde está u.

- **Implementación de $make\_set(u)$**

```
int p[MX];

void make_set(int u) { //O(1)
    p[u] = u;
}
```

- **Implementación de $find(u)$**

```
int find(int u) { //O(1)
    return p[u];
}
```

- **Implementación de $join(u, v)$:** Pasamos todos los elementos del grupo de u al grupo de v

```
void join(int u, int v, int n) { //O(n)
    u = p[u];
    v = p[v];
    for (int i = 0; i < n; i++) {
        if (p[i] == u) {
            p[i] = v;
        }
    }
}
```

⚠ ADVERTENCIA

Esta implementación es muy lenta. No trate de usarlas en un problema

Pequeña mejora

¿Y si por cada grupo, aparte de guardar el elemento más representativo, **guardamos la lista de elementos** que pertenece a ese grupo?

De esa forma en el $join$ ya no tendría que recorrer todos los n elementos, sino solo recorrería los elementos del grupo de u y se los pasaría al grupo de v

```
struct DSU{
    int p[MX];
    vector<int> elements[MX];

    void make_set(int u) {
        p[u] = u;
        elements[u] = {u};
    }

    int find(int u) {
        return p[u];
    }

    void join(int u, int v) {
        u = p[u];
        v = p[v];
        if (u != v) {
            for (int x : elements[u]) {
                elements[v].push_back(x);
                p[x] = v;
            }
        }
    }
}dsu;
```

Análisis de Complejidad

En cada uso del $join$ la complejidad ahora es $O(|grupo\ de\ u|)$. Sin embargo, algunas operaciones $join$ pueden ser muy costosas, mientras que otras no tanto. Para poder hacer un mejor análisis, haremos el denominado **análisis amortizado**, en donde se tomará el promedio de los tiempos consumidos en todas las llamadas hechas a una determinada estructura de datos.

En el caso del dsu, considere que se tienen n elementos y m operaciones. Asumamos que las n primeras operaciones son de $make\_set$ entonces tendríamos $m = n + \#op\ find + \#op\ join$.

Note que cada vez que se usa una operación $join$ entre dos grupos distintos, el número de grupos disminuye en 1, por lo tanto como máximo solo $n - 1$ operaciones $join$ tendrán efecto: $\#op\ join \leq n - 1$.

Como las operaciones $find$ son con complejidad constante, por un momento ignorémosla, y asumamos que $\#op\ join = \mathrm{n} - 1$, entonces tendríamos $m = 2n\ - 1$.

Veremos que podemos construir un caso que nos lleve a una complejidad total de $O(n^2)$

Análisis de Complejidad

Peor caso:

Operación	Número de movimientos
$make\_set(x_1)$	1
$make\_set(x_2)$	1
...	...
$make\_set(x_n)$	1
$join(x_1, x_2)$	1
$join(x_2, x_3)$	2
$join(x_3, x_4)$	3
...	...
$join(x_{n-1}, x_n)$	$n-1$
Total	$\boldsymbol{n + \frac{n(n-1)}{2} = O(n^2)}$
Promedio	$\boldsymbol{O(n)}$ **por operación**

Contenido

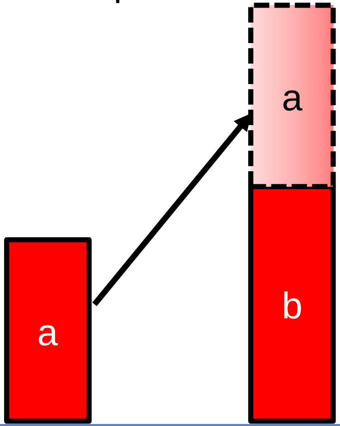
1. Definición de la estructura	⇨
2. Simple implementación	⇨
3. Small-to-large technique	⇨
4. Path Compression	⇨
5. Implementación óptima	⇨

Small-to-large

Para poder mejorar la complejidad del dsu haremos uso de algunas heurísticas.

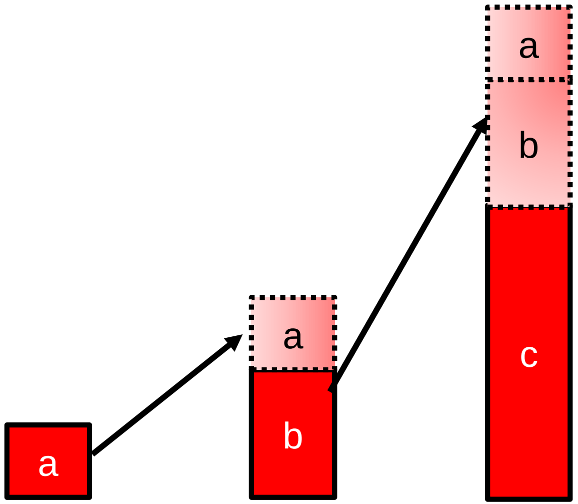
En la sección anterior vimos que nuestra implementación anterior obtenía un $O(n)$ por operación en el peor caso. Pero podemos observar que el peor caso se formaba cuando pasabas todos los elementos de un grupo de tamaño x a un grupo de tamaño 1, haciendo x movimientos. Pero, ¿qué pasaría si moviéramos los elementos del grupo del tamaño 1 al grupo de tamaño x? O, en general, si moviéramos los elementos del **grupo más pequeño hacia el más grande**.

Justamente eso es lo que nos dice la técnica **small-to-large**.



Small-to-large

¿Cómo cambiará la complejidad?
Pensemos en ¿cuántas veces cómo máximo, un elemento se cambia a otro grupo?
Supongamos que tenemos un elemento x que está en un grupo de tamaño a y se traslada a un grupo de tamaño b $(a \leq b)$. Es decir, pasó de un grupo de tamaño a a un grupo de tamaño $a + b$
Como $a \leq b \implies a + a \leq a + b \implies 2a \leq a + b$. Es decir, pasó a un grupo que tendrá el doble o más del tamaño del grupo anterior.



Observación clave: Un grupo solo puede duplicar su tamaño $\log n$ veces.

Por lo tanto, cada elemento solo se trasladará de grupo $\log n$ veces.

Complejidad total de todos los $\boldsymbol{join}$: $O(n \log n)$
Complejidad $\mathbf{join}$ amortizada: $O(\log n)$

Small-to-large

Solo necesitamos hacer un pequeño cambio en nuestro código anterior

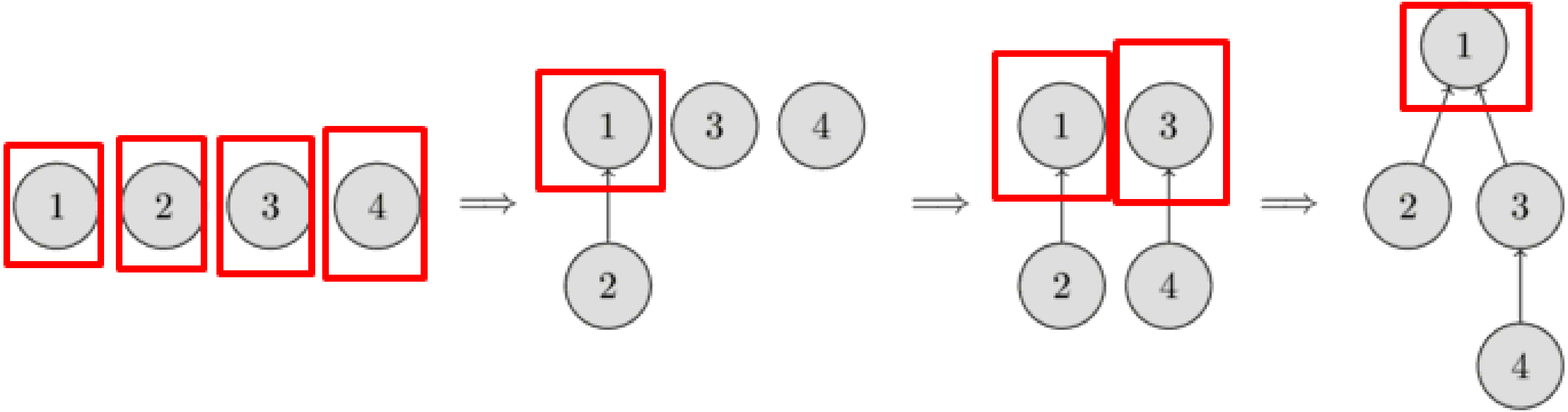
```
void join(int u, int v) { //O(log n) amortized
    u = p[u];
    v = p[v];
    if (u != v) {
        if (elements[u].size() > elements[v].size()) {
            swap(u , v);
        }
        for (int x : elements[u]) {
            elements[v].push_back(x);
            p[x] = v;
        }
    }
}
```

Contenido

1. Definición de la estructura	⇨
2. Simple implementación	⇨
3. Small-to-large technique	⇨
4. Path Compression	⇨
5. Implementación óptima	⇨

Representación

Utilicemos otro enfoque ahora. ¿Sería posible disminuir la complejidad del $join$, sacrificando un poco la complejidad del $find$?
Para eso tendremos que cambiar un poco la idea: representemos a los grupos como un **árbol** en donde el elemento representativo estaría en la raíz.



Fuente: cp-algorithms

Implementación

Definimos un arreglo $parent[u]$ que represente al padre de u.

- **Implementación de $\boldsymbol{make\_set(u)}$**

```
int parent[MX];

void make_set(int u) {
    parent[u] = u;
}
```

- **Implementación de $\boldsymbol{join(u, v)}$**

```
void join(int u, int v) {
    u = find(u);
    v = find(v);
    if(u != v){
        parent[v] = u;
    }
}
```

- **Implementación de $\boldsymbol{find(u)}$:** Recorremos todos los ancestros de u hasta llegar a la raíz

```
int find(int u) {
    if (parent[u] == u) {
        return u;
    } else {
        return find(parent[u]);
    }
}
```

⚠ ADVERTENCIA

Esta implementación es muy lenta. No trate de usarlas en un problema

Análisis de complejidad

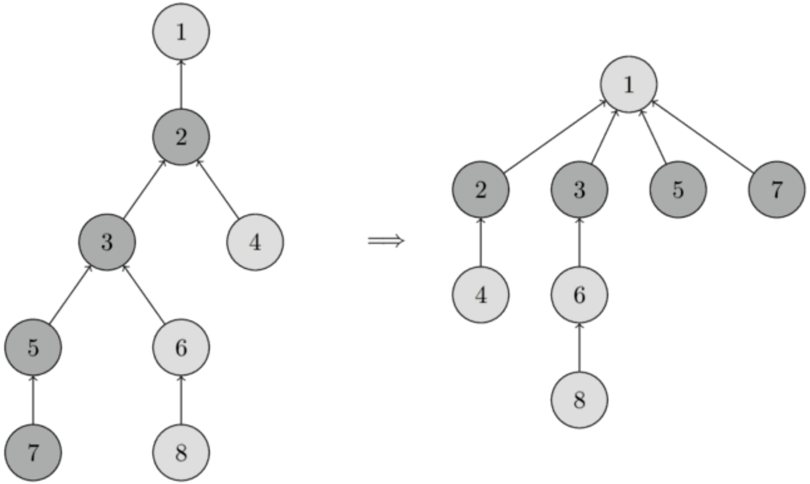
$make\_set$ sigue siendo $O(1)$, pero ahora el $find$ puede ser $O(n)$ cuando el árbol degenera en una cadena. Y el $join$ depende del $find$.

Si tratamos de poner el mismo caso de la diapo 10 que malograba nuestra primera implementación, veremos que también degenera en un $O(n^2)$ en total.

Path Compression

Podemos optimizar nuestra función $find$ con la técnica de path compression. Cada vez que hagamos una operación $find(u)$ podemos directamente actualizar el $parent$ de todos los ancestros de u y ponerles a la raíz como padre.

Ejemplo: Supongamos que en la siguiente imagen se hace un $find(7)$



Fuente: cp-algorithms

Path Compression

El código del $find$ solo sufriría una sutil modificación.

Sorprendentemente, este cambio hace que la complejidad amortizada por operación sea $O(\log n)$. Siendo más exactos, Seidel and Sharir demostraron que la complejidad total era aproximadamente $O((n+m)\log n)$

```
int find(int u) { //O(log n) amortized
    if (parent[u] == u) {
        return u;
    } else {
        return parent[u] = find(parent[u]);
    }
}
```

Contenido

1. Definición de la estructura	⇨
2. Simple implementación	⇨
3. Small-to-large technique	⇨
4. Path Compression	⇨
5. Implementación óptima	⇨

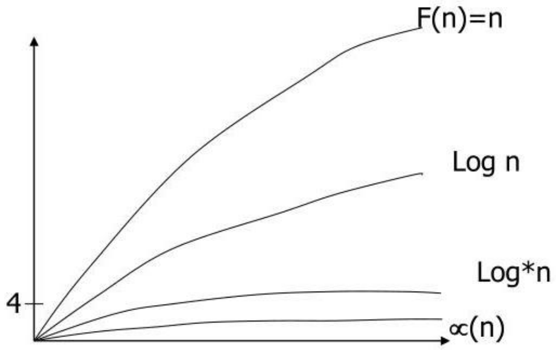
Combinación de Heurísticas

Podemos lograr una implementación óptima de DSU si combinamos las heurísticas de small-to-large y de path compression.

Se puede demostrar (pero es complicado) que la complejidad amortizada al combinar ambas heurísticas es $O\left(\alpha(n)\right)$, donde $\alpha(n)$ es la función inversa de Ackermann.

La función inversa de Ackermann crece muy lento, tan lento que $\alpha(n) \leq 4$, para un $n < 10^{600}$.

De forma práctica podemos considerarlo como una constante.



Fuente: https://www.slideserve.com/Ava/union-by-rank-ackermann-s-function-graph-algorithms

Combinación de Heurísticas

```
int parent[MX];
int size[MX];

void make_set(int u) {
    parent[u] = u;
    size[u] = 1;
}

void build(int n) {
    for (int i = 0; i < n; i++) {
        make_set(i);
    }
}
```

```
int find(int u) { //O(1) amortized
    //path compression
    if (parent[u] == u) {
        return u;
    } else {
        return parent[u] = find(parent[u]);
    }
}

void join(Long u, Long v){ //O(1) amortized
    //small-to-large
    u = find(u);
    v = find(v);
    if(u != v){
        if(size[u] > size[v]){
            swap(u, v);
        }
        parent[u] = v;
        size[v] += size[u];
    }
}
```

Información adicional en el dsu

Así como creamos un arreglo $size[u]$ para almacenar el tamaño del grupo de u, también podemos acumular cualquier otra función **commutativa y asociativa** sobre el grupo.

```
int parent[MX];
int size[MX];
int minimum[MX];

void make_set(int u) {
    parent[u] = u;
    size[u] = 1;
    minimum[u] = u;
}

void build(int n) {
    for (int i = 0; i < n; i++) {
        make_set(i);
    }
}

int find(int u) { //O(1) amortized
    //path compression
    if (parent[u] == u) {
        return u;
    } else {
        return parent[u] = find(parent[u]);
    }
}
```

```
void join(Long u, Long v){ //O(1) amortized
    //small-to-large
    u = find(u);
    v = find(v);
    if(u != v){
        if(size[u] > size[v]){
            swap(u, v);
        }
        parent[u] = v;
        size[v] += size[u];
        minimum[v] = min(minimum[v], minimum[u]);
    }
}

int getMinimum(int u) {
    return minimum[find(u)];
}
```

Problemas

- Codeforces EDU: https://codeforces.com/edu/course/2/lesson/7/1/practice
- Xranda and Tree - IEEE Xtreme: https://csacademy.com/ieeextreme-practice/task/xranda-and-tree/
- Codeforces - Path Queries: https://codeforces.com/problemset/problem/1213/G

Referencias

- CP-algorithms: https://cp-algorithms.com/data_structures/disjoint_set_union.html
- Introduction to Algorithms. CLRS: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf
- Codeforces EDU: https://codeforces.com/edu/course/2/lesson/7/1