Yadran Eterovic

Conjuntos Disjuntos

2 - 2017

Estructuras de Datos y Algoritmos - IIC2133

Propiedades

Una **estructura de datos para conjuntos disjuntos** mantiene una colección *S* de conjuntos disjuntos dinámicos:

- cada conjunto es identificado por un **representante** —algún elemento del conjunto
- si preguntamos por el representante de un conjunto dos veces, sin modificar el conjunto entre las consultas, debemos obtener la misma respuesta ambas veces

makeSet(*x*):

- construye un nuevo conjunto, cuyo único elemento y representante es *x*

findSet(*x*):

- devuelve el representante del conjunto que contiene a *x*

union(*x*,*y*):

- une los conjuntos que contienen a *x* e *y*, *Sx* y *Sy*, en un nuevo conjunto que es la unión de *Sx* y *Sy*
- el representante del conjunto resultante es cualquier elemento de la unión
- como los conjuntos en la colección *S* deben ser disjuntos, eliminamos *Sx* y *Sy* de *S*

Medimos el desempeño de una estructura en función de dos parámetros

n, el número de operaciones *makeSet*( ) —las primeras operaciones ejecutadas

m, el número total de operaciones *makeSet*( ), *union*( ), y *find*( )

$m \geq n$

Cada *union* reduce el número de conjuntos en uno —los conjuntos son disjuntos

Después de $n - 1$ operaciones *union*( ), sólo queda un conjunto

A lo más hay $n - 1$ operaciones *union*( )

Podemos representar cada conjunto mediante una *lista ligada* de los objetos

El primer objeto de cada lista es el representante del conjunto

Hay una referencia, *head*, al representante

Hay una referencia, *tail*, al último objeto

Cada objeto contiene tres campos

Un elemento del conjunto

Una referencia al objeto con el siguiente elemento en el conjunto

Una referencia al representante

¿Cuál es el desempeño de esta estructura?

makeSet( ) y *findSet*( ) toman tiempo O(1)

union(*x*, *y*) significa agregar la lista de *x* a la lista de *y*

Si usamos la heurística de agregar la lista más corta a la más larga —para lo cual, cada conjunto debe saber cuántos elementos tiene—

… entonces *m* operaciones, de las cuales *n* son *makeSet*( ),

… toman en total tiempo O(*m* + *n*log*n*)

Ejemplo de aplicación: Construcción de laberintos

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0,1} {2} {3} {4,6,7,8,9,13,14} {5} {10,11,15} {12} {16,17,18,22} {19} {20} {21} {23} {24}

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0,1} {2} {3} {4,6,7,8,9,13,14, 16,17,18,22} {5} {10,11,15} {12} {19} {20} {21} {23} {24}

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24}

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Podemos hacer eficientes los *findSet*( ) …

En la representación mediante listas ligadas, *findSet*( ) toma tiempo O(1) en el peor caso

… pero *union*( ) toma tiempo proporcional al tamaño del conjunto más pequeño

… o podemos hacer eficientes los *union*( )

También es posible conseguir que *union*( ) tome tiempo O(1) en el peor caso

… pero entonces *findSet*( ) tomará más que tiempo constante

No podemos hacer eficientes los *findSet*( ) y los *union*( ) simultáneamente

Está demostrado que no es posible conseguir que ambas operaciones puedan ser ejecutadas en tiempo O(1) en el peor caso simultáneamente

Podemos representar los conjuntos mediante árboles …

Representamos cada conjunto mediante un árbol,

… en que cada nodo contiene un elemento:

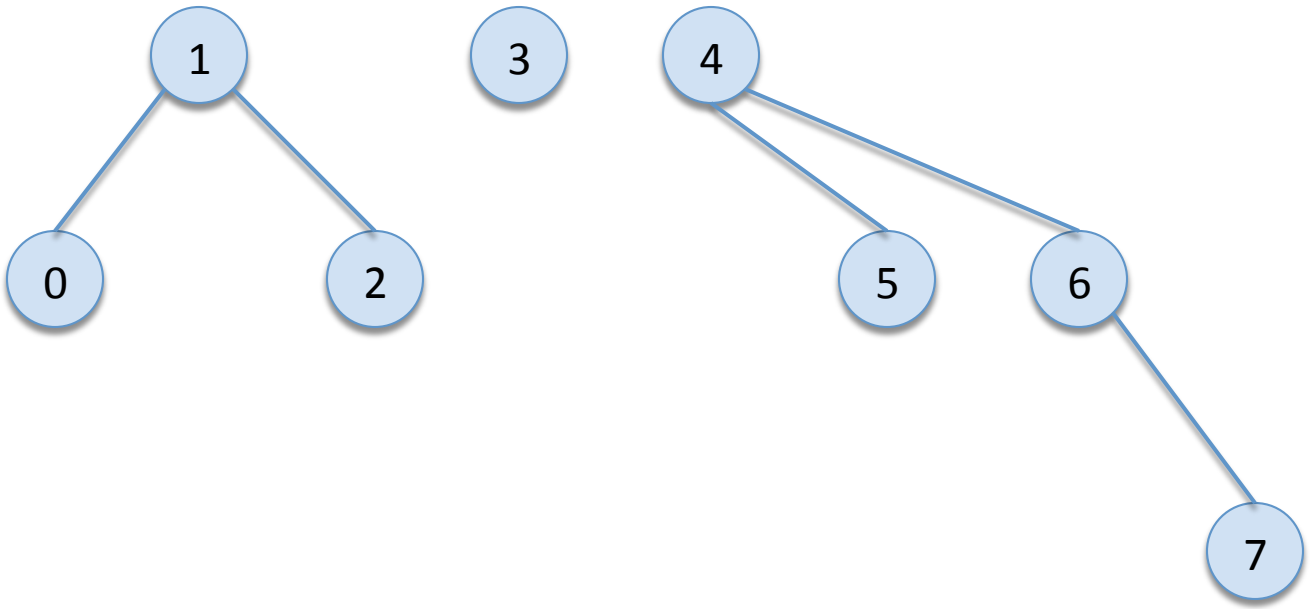
- cada nodo tiene un puntero a su padre
- la raíz contiene al representante y es su propio padre

… que implementamos simplemente en un arreglo

Para cada elemento 0, 1, …, n–1,

… sólo necesitamos el puntero a su padre en el árbol:

- mantenemos los *n* elementos en un arreglo *p*, tal que *p*[*k*] es el padre de *k* en su árbol



0	1
1	–1
2	1
3	–1
4	–1
5	4
6	4
7	6

makeSet(*x*):

- construye un árbol con un único nodo *x*

findSet(*x*):

- sigue los punteros a los padres hasta llegar a la raíz
- los nodos visitados en este recorrido forman la *ruta de búsqueda*

union(*x*, *y*):

- la raíz de uno de los árboles apunta a la raíz del otro

Esta representación podría desempeñarse mal

Una secuencia de *n* – 1 operaciones *union*( ) podría crear un árbol que fuera una lista ligada de *n* elementos

Pero hay una solución, basada en dos técnicas

1) *Unión por rango*:

- hacemos que la raíz del árbol con menos nodos apunte a la raíz del árbol con más nodos
- en lugar de contar exactamente el número de nodos de un árbol, mantenemos un **rango**

2) *Compresión de ruta*:

- durante las operaciones *findSet*( ), hacemos que cada nodo en la ruta de búsqueda apunte directamente a la raíz —no cambia los rangos

```
void makeSet(T x)
    x.p = x
    x.rank = 0

void union(T x, T y)
    link(findSet(x), findSet(y))

void link(T x, T y)
    if (x.rank() > y.rank())
        y.p = x
    else
        x.p = y
        if (x.rank() == y.rank())
            y.incrementRank()

T findSet(T x)
    if (x != x.p)
        x.p = findSet(x.p)
    return x.p
```

Usando sólo *unión por rango*

El tiempo de ejecución es O(*m* log*n*)

Usando sólo *compresión de ruta*

El tiempo de ejecución es O(n+*finds*•(1+log$n$)),

… en que *finds* es el número de operaciones *findSet*

Y si usamos ambas técnicas simultáneamente

Usando tanto unión por rango como compresión de ruta,

… el tiempo de ejecución es O($m\ \alpha(n)$), en que $\alpha(n)$ es una función de muy lento crecimiento:

- para cualquier aplicación práctica, $\alpha(n) \leq 4$
- el tiempo de ejecución es, prácticamente, lineal

$A(1, j) = 2^j$

$A(k, 1) = A(k\text{-}1, 2)$

$A(\mathrm{k}, j) = A(k\text{-}1, \mathrm{A}(k, j\text{-}1))$

$A(2, j) = 2^{2^{2^{2\ldots}}}$

$\log^* 65{,}536 = 4$

$\log^* 2^{65536} = 5$

Definimos $\alpha(n)$ como $\min\{k : A(k,1) \geq n\}$

Podemos comprobar que $\alpha(n) =$

- 0 para $0 \leq n \leq 2$
- 1 para $n = 3$
- 2 para $4 \leq n \leq 7$
- 3 para $8 \leq n \leq 2{,}047$
- 4 para $2{,}048 \leq n \leq A(4, 1) = 16^{512}$