

La estructura union-búsqueda

Union – Find
Disjoint sets

Descripción del problema

- Se tienen parejas de objetos p, q . Se desea determinar si estas parejas están “conectadas” o no.
- Se pueden conectar parejas p, q para formar una componente conexas.
- Inicialmente hay N objetos desconectados, es decir N componentes. Por cada pareja p, q leída se conectan (unen) sus respectivas componentes.

“Conectividad” es una relación de equivalencia

- Reflexiva: Un nodo siempre está conectado a si mismo.
- Simétrica: Si A está conectado a B, entonces B esta conectado a A.
- Transitiva: Si A está conectado a B y B está conectado a C, entonces A está conectado a C.

Aplicaciones

- En redes: Interesa determinar conectividad entre nodos de una red.
- En el grafo social: Un grupo de amigos es una componente conexa
- En lenguajes de programación: Todas las referencias a una misma instancia en memoria son una componente conectada.
- En teoría de conjuntos: Cada partición de un conjunto es una componente. Las componentes son disjuntas y su unión es el conjunto original.

Formulación del problema

- Se asumen N nodos, inicialmente desconectados y numerados $0..n-1$.
- El API a implementar es

public class UF		
	UF(N)	// Crear estructura
boolean	connected(p,q)	// Están conectados p y q?
int	find(p)	// Encontrar componente de p
void	union(p,q)	// Unir la componentes p,q
int	count()	// Número de componentes

Solución del problema

- Una representación de la estructura de datos
- Una implementación de las operaciones

Como se verá, hay varias posibles soluciones con importantes diferencias en términos de su desempeño.

1ª solución: QuickFind

- Representación:
 - vector de enteros $id[0..n-1]$
 - $id[i]$ es el número de la componente de i
- Implementación de las operaciones
 - $find(p)$: Retornar $id[p]$
 - $connected(p, q)$: Retornar $find(p) == find(q)$
 - $union(p, q)$: Reemplazar las referencias al $id[p]$ por el $id[q]$.

Implementación de QuickFind

- Ver código fuente: [QuickFindUF](#)

Análisis de QuickFind

Modelo de costo: Accesos al arreglo

- `find`: Un acceso al arreglo
- `connected`: Dos accesos al arreglo
- `union`: Entre $2+(N+1)$ y $2+(2N+1)$ accesos al arreglo

2ª solución: QuickUnion

- Representación:
 - vector de enteros $id[0..n-1]$
 - $id[i]$ es el número del antecesor de i . El nodo raíz tiene $id[i]=i$.
 - Cada componente queda descrito por un “árbol”
- Implementación de las operaciones
 - $find(p)$: Retorna el id de la raíz del árbol al que pertenece p .
 - $connected(p, q)$: Retornar $find(p) == find(q)$
 - $union(p, q)$: Asignar raíz del árbol q como antecesor de la raíz del árbol p .

Implementación de QuickUnion

- Ver código fuente: [QuickUnionUF](#)

Atributos de un árbol

Se define:

- Tamaño del árbol: Su número de nodos
- Profundidad de un nodo: Número de aristas del nodo a la raíz
- Altura del árbol: Mayor profundidad de todos los nodos

Análisis de QuickUnion

- `find`: 2 x profundidad del nodo más 1.
- `connected`: 2 llamados a `find`.
- `union`: 2 llamados a `find`, más un acceso para la asignación.

Peor caso: Qué el árbol sea una cadena de N nodos, en cuyo caso `find` requiere $\sim 2N$ accesos.

3ª solución:

WeightedQuickUnion

- Idea general: Evitar árboles de mucha profundidad
- Representación: Se maneja el vector `id[0..N-1]` igual que en `QuickUnion` y un segundo vector `sz[0..N-1]` para indicar el tamaño del árbol en la raíz.
- Operaciones `find` y `connected` no cambian.
- Operación `union`: Se conecta el árbol más pequeño como hijo del árbol mayor. Arbitrariamente si son de igual tamaño.

Implementación de `WeightedQuickUnion`

- Ver código fuente: [WeightedQuickUnionUF](#)

Desempeño de WeightedQuickUnion

- **Proposición:** La altura del árbol obtenido por WeightedQuickUnion tiene como máximo altura $\lg(N)$ para un conjunto de N nodos.
- **Corolario:** Las operaciones `find`, `connected` y `union` de WeightedQuickUnion son de orden $\log(N)$.

Demostración

Caso base: $N = 1, H = 0, 0 \leq \lg 1$

Hipotesis: $N = k$, se asume $H_k \leq \lg(k)$

Paso inductivo: Sea $N = i + j$, con $i, j \leq k$.

Asumimos $i \leq j$, el arbol i como hijo del arbol j

Se pueden presentar 2 casos

a. El arbol resultante tiene la misma altura de j

$$h_N = h_j \leq \lg j \leq \lg N$$

b. El arbol resultante tiene altura $h_i + 1$

$$i + j = N$$

$$i + i \leq N$$

$$\lg 2i \leq \lg N$$

$$\lg 2 + \lg i \leq \lg N$$

$$1 + h_i \leq \lg N$$

$$h_N \leq \lg N$$

Extensiones

- Es posible mejorar aún más el desempeño, por ejemplo con la técnica de *compresión de caminos*.
- El análisis amortizado utilizando esta técnica lleva a que el desempeño se acerca a tiempo constante.