Estructura Unión-Búsqueda (Conjuntos Disjuntos)

Notas de clase

Estructuras de datos y algoritmos

Facultad TIC - UPB

Spring 2025

Jorge Mario Londoño Peláez & Varias AI

July 7, 2025

Estructura Unión-Búsqueda (Conjuntos Disjuntos)

La estructura Union-Búsqueda y sus aplicaciones para representar relaciones de conectividad y conjuntos disyuntos.

Contents

1 El problema de componentes conexas / Conjuntos disjuntos

El problema de componentes conexas (o conjuntos disjuntos) se centra en determinar la conectividad entre diferentes nodos en un grafo o red. Dos nodos se consideran conectados si existe un camino entre ellos. Esta relación de conectividad es una relación de equivalencia, lo que significa que cumple con las siguientes propiedades:

- **Reflexiva:** Todo nodo está conectado consigo mismo.
- **Simétrica:** Si el nodo A está conectado al nodo B, entonces el nodo B está conectado al nodo A.
- **Transitiva:** Si el nodo A está conectado al nodo B, y el nodo B está conectado al nodo C, entonces el nodo A está conectado al nodo C.

El objetivo principal es, dado un conjunto de nodos y un conjunto de conexiones entre ellos, poder determinar eficientemente si dos nodos dados están conectados y, en caso de que no lo estén, unirlos.

2 API y representación

La estructura Union-Find representa un conjunto de n elementos distintos, cada uno representando un conjunto disjunto. Convencionalmente, los elementos se representan con números naturales de 0 a $n-1$.

El API básico de la estructura Union-Find consta de las siguientes operaciones:

- **find(p):** Determina el conjunto al que pertenece el elemento p. Devuelve un identificador (un número) que representa el conjunto. Dos elementos están en el mismo conjunto si y solo si `find(p) == find(q)`.
- **connected(p, q):** Verifica si los elementos p y q están en el mismo conjunto. Es equivalente a comparar los resultados de `find(p)` y `find(q)`.
- **union(p, q):** Une los conjuntos que contienen los elementos p y q. Después de esta operación, `find(p) == find(q)`.
- **count():** Devuelve el número de conjuntos disjuntos.

3 Implementación de la estructura Union-Find

3.1 Búsqueda rápida: QuickFind

La implementación **QuickFind** utiliza un arreglo `id[]` de tamaño n, donde `id[i]` representa el identificador del componente al que pertenece el elemento i. Inicialmente, cada elemento está en su propio conjunto, por lo que `id[i] = i` para todo i.

Operaciones:

- **find(p):** Simplemente devuelve `id[p]`. Esta operación toma tiempo constante, $O(1)$.
- **connected(p, q):** Compara `id[p]` y `id[q]`. Si son iguales, los elementos están conectados. Esta operación también toma tiempo constante, $O(1)$.

- **union(p, q):** Para unir los conjuntos que contienen p y q, se recorre todo el arreglo `id[]` y se cambia el identificador de todos los elementos que pertenecen al mismo conjunto que p para que coincida con el identificador del conjunto de q. Esta operación toma tiempo lineal, $O(n)$.

- **count():** Para obtener el número de conjuntos disjuntos, se mantiene una variable `count` que se decrementa cada que se realiza una union. Esta operación toma tiempo constante, $O(1)$.

QuickFind ofrece operaciones `find()` y `connected()` muy rápidas ($O(1)$), pero la operación `union()` es lenta ($O(n)$). Esto hace que QuickFind sea ineficiente para problemas donde se realizan muchas operaciones de unión.

QuickFind - Implementación del texto guía

3.2 Representación por árboles: QuickUnion

La implementación **QuickUnion** utiliza un arreglo `parent[]` de tamaño n, donde `parent[i]` representa el nodo padre del elemento i. En esta representación, cada conjunto disjunto se representa como un árbol, y la raíz del árbol es el representante del conjunto. Inicialmente, cada elemento es la raíz de su propio árbol, por lo que `parent[i] = i` para todo i.

Operaciones:

- **find(p):** Sigue los enlaces de los padres desde el nodo p hasta llegar a la raíz del árbol. La raíz es el representante del conjunto. La eficiencia de esta operación depende de la altura del árbol. En el peor caso, puede ser $O(n)$.

- **connected(p, q):** Encuentra las raíces de los árboles que contienen p y q usando la operación `find()`. Si las raíces son iguales, los elementos están conectados. La eficiencia de esta operación depende de la altura de los árboles, en el peor caso $O(n)$.

- **union(p, q):** Encuentra las raíces de los árboles que contienen p y q. Si las raíces son diferentes, une los dos árboles haciendo que la raíz de uno de los árboles apunte a la raíz del otro árbol. La eficiencia de esta operación depende de la altura de los árboles, en el peor caso $O(n)$.

- **count():** Para obtener el número de conjuntos disjuntos, se mantiene una variable `count` que se decrementa cada que se realiza una union. Esta operación toma tiempo constante, $O(1)$.

Eficiencia:

QuickUnion realmente no mejora la eficiencia de ninguna de las operaciones. Todas tienen un tiempo de peor caso $O(n)$. Sin embargo, la eficiencia se puede describir en forma general en función de la altura del árbol, en cuyo caso los tiempos son $O(h)$ y esta es la base para obtener la estructura siguiente, la cual es mucho más eficiente.

QuickUnion - Implementación del texto guía

3.3 Representación por árboles con peso: WeightedQuickUnion

La implementación **WeightedQuickUnion** es una mejora de QuickUnion que busca evitar que los árboles se vuelvan demasiado altos. Además del arreglo `parent[]` (donde `parent[i]` representa el nodo padre del elemento i), se utiliza un segundo arreglo `size[]` de tamaño n, donde `size[i]` representa el número de nodos en el subárbol cuya raíz es i.

Operaciones:

- **find(p):** Igual que en QuickUnion, sigue los enlaces de los padres desde el nodo p hasta llegar a la raíz del árbol. La eficiencia de esta operación depende de la altura del árbol.
- **connected(p, q):** Igual que en QuickUnion, encuentra las raíces de los árboles que contienen p y q usando la operación `find()`. Si las raíces son iguales, los elementos están conectados.
- **union(p, q):** Encuentra las raíces de los árboles que contienen p y q. Si las raíces son diferentes, une los dos árboles haciendo que la raíz del árbol más pequeño (en términos de número de nodos) apunte a la raíz del árbol más grande. Esto ayuda a mantener los árboles balanceados y reduce la altura de los árboles. Se actualiza el arreglo `size[]` para reflejar el nuevo tamaño del árbol resultante.
- **count():** Para obtener el número de conjuntos disjuntos, se mantiene una variable `count` que se decrementa cada que se realiza una union. Esta operación toma tiempo constante, $O(1)$.

Eficiencia:

WeightedQuickUnion reduce significativamente la altura de los árboles, lo que mejora la eficiencia de las operaciones `find()` y `union()`. En el peor caso, la altura de los árboles crece logarítmicamente con el número de elementos, por lo que las operaciones `find()` y `union()` toman tiempo $O(\log n)$.

WeightedQuickUnion - Implementación del texto guía

3.4 Otras consideraciones

3.4.1 Compresión de caminos

La **compresión de caminos** es una técnica que se puede aplicar a las implementaciones de QuickUnion y WeightedQuickUnion para mejorar aún más su eficiencia. La idea principal es que, al realizar una operación `find(p)`, se recorre el camino desde el nodo p hasta la raíz del árbol. Durante este recorrido, se puede hacer que cada nodo en el camino apunte directamente a la raíz. Esto ”aplana” el árbol y reduce la longitud de los caminos para futuras operaciones `find()` en esos nodos y sus descendientes.

Existen dos variantes comunes de compresión de caminos:

- **Compresión de caminos simple:** Durante la operación `find(p)`, después de encontrar la raíz, se recorre nuevamente el camino desde p hasta la raíz, haciendo que cada nodo en el camino apunte directamente a la raíz.
- **División de caminos:** Similar a la compresión de caminos simple, pero en lugar de hacer que cada nodo apunte directamente a la raíz, se hace que cada nodo apunte a su abuelo en el árbol.

La compresión de caminos puede mejorar significativamente la eficiencia de las operaciones `find()` y `union()`. Aunque el análisis preciso de la eficiencia de Union-Find con compresión de caminos es complejo, se ha demostrado que, en la práctica, el tiempo amortizado por operación es casi constante.

3.4.2 Análisis amortizado

El **análisis amortizado** es una técnica utilizada para analizar la eficiencia de algoritmos que realizan una secuencia de operaciones. En lugar de analizar el costo de cada operación individualmente, el análisis amortizado considera el costo promedio de una operación en una secuencia de operaciones.

En el caso de Union-Find con compresión de caminos, el análisis amortizado muestra que, aunque algunas operaciones individuales pueden ser costosas, el costo promedio por operación en una secuencia de m operaciones es muy bajo, casi constante. Más precisamente, el tiempo amortizado por operación es $O(\alpha(n))$, donde $\alpha(n)$ es la función inversa de Ackermann, que crece extremadamente lento. En la práctica, para valores realistas de n, $\alpha(n)$ es menor o igual a 4.

Esto significa que Union-Find con compresión de caminos es una estructura de datos muy eficiente para resolver problemas de conectividad en la práctica.

Bibliografía

[1] Aditya Bhargava. *Grokking Algorithms*. Manning Publications, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Narasimha Karumanchi. *Data Structures and Algorithms Made Easy*. CareerMonk Publications, 2011.

[4] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 2005.

[5] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.