GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: UNIÓN DE CONJUNTOS DISJUNTOS *(DISJOINT SET UNION)*

---

1. Introducción

En esta guía se va abordar la estructura de datos Unión de Conjuntos Disjuntos (*Disjoint Set Union o DSU*) . A menudo también se le llama *Union Find* debido a sus dos operaciones principales.

2. Conocimientos previos

2.1. Conjunto

Un conjunto es el modelo matemático para una colección de diferentes cosas;un conjunto contiene elementos o miembros , que pueden ser objetos matemáticos de cualquier tipo: números, símbolos, puntos en el espacio, líneas, otras formas geométricas, variables o incluso otros conjuntos. El conjunto sin elemento es el conjunto vacío ; un conjunto con un solo elemento es un singleton . Un conjunto puede tener un número finito de elementos o ser un conjunto infinito . Dos conjuntos son iguales si tienen exactamente los mismos elementos.

2.2. Conjuntos disjuntos

En matemáticas , se dice que dos conjuntos son conjuntos disjuntos si no tienen ningún elemento en común. De manera equivalente, dos conjuntos disjuntos son conjuntos cuya intersección es el conjunto vacío . Por ejemplo, {1, 2, 3} y {4, 5, 6} son conjuntos disjuntos, mientras que {1, 2, 3} y {3, 4, 5} no lo son. Una colección de dos o más conjuntos se llama disjunta si dos conjuntos distintos pueden de la colección son disjuntos.

2.3. Arreglos

Un arreglo es una serie de elementos del mismo tipo ubicados en zonas de memoria continuas que pueden ser referenciados por un índice y un único identificador, esto quiere decir que podemos almacenar 10 valores enteros en un arreglo sin tener que declarar 10 variables diferentes.

3. Desarrollo

Esta estructura de datos proporciona las siguientes capacidades. Se nos dan varios elementos, cada uno de los cuales es un conjunto separado. Una DSU tendrá una operación para combinar dos conjuntos y podrá decir en qué conjunto se encuentra un elemento específico. La versión clásica también introduce una tercera operación, puede crear un conjunto a partir de un nuevo elemento.

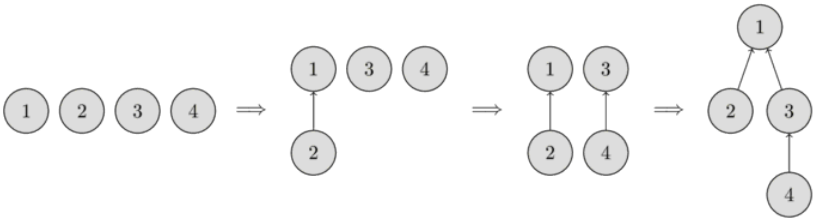
Por lo tanto, la interfaz básica de esta estructura de datos consta de solo tres operaciones:

- **make_set(v):** crea un nuevo conjunto que consiste en el nuevo elemento v
- **union_sets(a, b):** fusiona los dos conjuntos especificados (el conjunto en que se encuentra el elemento a y el conjunto en que se encuentra el elemento b)

- **find_set(v):** devuelve el representante (también llamado líder) del conjunto que contiene el elemento v. Este representante es un elemento de su conjunto correspondiente. Se selecciona en cada conjunto por la propia estructura de datos (y puede cambiar con el tiempo, es decir, después de $union\_sets$ las llamadas). Este representante se puede utilizar para comprobar si dos elementos forman parte del mismo conjunto o no. a y b están exactamente en el mismo conjunto, si $find\_set(a) == find\_set(b)$. De lo contrario, están en conjuntos diferentes.

Almacenaremos los conjuntos en forma de árboles : cada árbol corresponderá a un conjunto. Y la raíz del árbol será la representante/líder del conjunto.

En la siguiente imagen puedes ver la representación de dichos árboles.



Al principio, cada elemento comienza como un solo conjunto, por lo tanto, cada vértice es su propio árbol. Luego combinamos el conjunto que contiene el elemento 1 y el conjunto que contiene el elemento 2. Luego combinamos el conjunto que contiene el elemento 3 y el conjunto que contiene el elemento 4. Y en el último paso, combinamos el conjunto que contiene el elemento 1 y el conjunto que contiene el elemento 3.

Para la implementación, esto significa que tendremos que mantener un arreglo $parent$ que almacene una referencia a su ancestro inmediato en el árbol.

3.1. Implementación sencilla

La primera implementación de la estructura de datos DSU. Será bastante ineficiente al principio, pero luego podemos mejorarlo usando dos optimizaciones, de modo que tomará un tiempo casi constante para cada llamada de función.

Como decíamos, toda la información sobre los conjuntos de elementos se guardará en un arreglo $parent$.

Para crear un nuevo conjunto (operación $make\_set(v)$), simplemente creamos un árbol con raíz en el vértice v, lo que significa que es su propio ancestro.

Para combinar dos conjuntos (operación $union\_sets(a, b)$), primero encontramos el representante del conjunto en el que a se encuentra y el representante del conjunto en el que b se encuentra. Si los representantes son idénticos, que no tenemos nada que hacer, los conjuntos ya están fusionados. De lo contrario, podemos simplemente especificar que uno de los representantes es el padre del otro representante, combinando así los dos árboles.

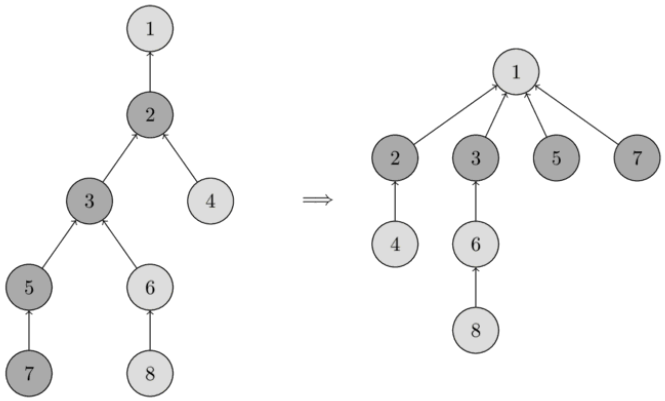
Finalmente la implementación de la función encontrar representante (operación $find\_set(v)$): simplemente subimos los ancestros del vértice vhasta llegar a la raíz, es decir, un vértice tal que la referencia al ancestro lleva a sí mismo. Esta operación se implementa fácilmente de forma recursiva.

3.2. Optimización de la compresión de rutas

Esta optimización está diseñada para acelerar $find\_set$.

Si llamamos $find\_set(v)$ a algún vértice v, en realidad encontramos el representante p de todos los vértices que visitamos en el camino entre v y el representante real p. El truco consiste en acortar las rutas de todos esos nodos, configurando el padre de cada vértice visitado directamente en p.

Puedes ver el funcionamiento en la siguiente imagen. A la izquierda hay un árbol, y a la derecha está el árbol comprimido después de llamar $find\_set(7)$, que acorta los caminos para los nodos visitados 7, 5, 3 y 2.



3.3. Unión por tamaño / rango

En esta optimización cambiaremos la $union\_set$ operación. Para ser precisos, cambiaremos qué árbol se une al otro. En la implementación ingenua, el segundo árbol siempre se adjuntaba al primero. En la práctica, eso puede llevar a que los árboles contengan cadenas de longitud$O(n)$. Con esta optimización evitaremos esto eligiendo con mucho cuidado qué árbol se adjunta.

Hay muchas heurísticas posibles que se pueden utilizar. Los más populares son los siguientes dos enfoques: en el primer enfoque usamos el tamaño de los árboles como rango, y en el segundo usamos la profundidad del árbol (más precisamente, el límite superior de la profundidad del árbol, porque la profundidad se hacen más pequeños al aplicar la compresión de ruta).

En ambos enfoques, la esencia de la optimización es la misma: adjuntamos el árbol con el rango más bajo al que tiene el rango más alto.

4. Implementación

4.1. C++

4.1.1. Implementación sencilla

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

int find_set(int v) {
   if (v == parent[v]) return v;
   return find_set(parent[v]);
}

void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) parent[b] = a;
}
```

4.1.2. Optimización de la compresión de rutas

```
int find_set(int v) {
   if (v == parent[v]) return v;
   return parent[v] = find_set(parent[v]);
}
```

4.1.3. Unión por tamaño / rango

Aquí está la implementación de la unión por tamaño:

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

void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) {
     if (size[a] < size[b]) swap(a, b);
     parent[b] = a;
     size[a] += size[b];
   }
}
```

Y aquí está la implementación de la unión por rango basada en la profundidad de los árboles:

```
void make_set(int v) {
   parent[v] = v; rank[v] = 0;
}

void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) {
     if (rank[a] < rank[b]) swap(a, b);
     parent[b] = a;
     if (rank[a] == rank[b]) rank[a]++;
   }
}
```

Aqui una implementación completa con estructura

```
struct DisjoinSet{
   vector<int> parent,sizes;
   int ncomponents; // cantidad de conjuntos disjuntos

   DisjoinSet(int n) : parent(n),sizes(n){
      for(int i = 0; i < n; i++) sizes[parent[i] = i] = 1;
      ncomponents = n;
   }

   int root(int x){
      if( x == parent[x] ) return x;
      else{
         parent[x] = root(parent[x])
         return parent[x]
      }
   }

   void join(int a,int b){
      int x = root(a); int y = root(b);
      if(x == y) return;
      if(sizes[x] < sizes[y]) swap(x,y);
      parent[y] = x;
      sizes[x] += sizes[y];
      ncomponents--;
   }
};
```

4.2. Java

4.2.1. Implementación sencilla

```
public void make_set(int v) {
    parent[v] = v;
```

```
}

public int find_set(int v) {
   if (v == parent[v]) return v;
   return find_set(parent[v]);
}

public void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) parent[b] = a;
}
```

4.2.2. Optimización de la compresión de rutas

```
public int find_set(int v) {
   if (v == parent[v]) return v;
   return parent[v] = find_set(parent[v]);
}
```

4.2.3. Unión por tamaño / rango

Aquí está la implementación de la unión por tamaño:

```
public void make_set(int v) {
   parent[v] = v; size[v] = 1;
}

public void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) {
     if (size[a] < size[b]) swap(a, b);
     parent[b] = a;
     size[a] += size[b];
   }
}
```

Y aquí está la implementación de la unión por rango basada en la profundidad de los árboles:

```
public void make_set(int v) {
   parent[v] = v; rank[v] = 0;
}

public void union_sets(int a, int b) {
   a = find_set(a); b = find_set(b);
   if (a != b) {
     if (rank[a] < rank[b]) swap(a, b);
```

```
      parent[b] = a;
      if (rank[a] == rank[b]) rank[a]++;
   }
}
```

Aqui una implementación completa con clases

```
private class DisJoinSet{
   public int [] parent;
   public int [] sizes ;
   public int ncomponents;

   public DisJoinSet(int n) {
      this.ncomponents = n;
      parent = new int [this.ncomponents+1];
      sizes = new int [this.ncomponents+1];
      for(int i=0;i<this.ncomponents+1;i++) sizes[parent[i]=i]=1;
   }

   public int root(int x){
      if (x == parent[x]) return x;
      else{
         parent[x] = root(parent[x]);
         return parent[x];
      }
   }

   void join(int a,int b){
      int x = root(a); int y = root(b);
      if(x != y) {
         if(sizes[x] < sizes[y]) {
            int tmp = x; x = y; y =tmp;
         }
         parent[y] = x;
         sizes[x] += sizes[y];
         this.ncomponents--;
      }
   }
}
```

5. Complejidad

La estructura de datos le permite realizar cada una de estas operaciones en casi $O(1)$ tiempo en promedio.

También en una de las subsecciones se explica una estructura alternativa de un DSU, que logra una complejidad promedio más lenta de $O(\log n)$, pero puede ser más potente que la estructura

DSU normal.

5.1. Implementación sencilla

Sin embargo, esta implementación es ineficiente. Es fácil construir un ejemplo, de modo que los árboles degeneren en largas cadenas. En ese caso, cada llamada $find\_set(v)$ puede tomar $O(n)$ tiempo. Esto está muy lejos de la complejidad que queremos tener (tiempo casi constante).

5.2. Optimización de la compresión de rutas

Esta simple modificación de la operación ya logra la complejidad del tiempo. $O(\log n)$ por llamada en promedio (aquí sin prueba). Hay una segunda modificación, que lo hará aún más rápido.

Si combinamos ambas optimizaciones (compresión de ruta con unión por tamaño / rango), alcanzaremos consultas de tiempo casi constante. Resulta que la complejidad del tiempo amortizado final es $O(f(x))$, dónde $f(x)$ es la función de Ackerman, que crece muy lentamente. De hecho crece tan lentamente, que no excede 4 por todo lo razonable n (aproximadamente $n < 10^{600}$)

La complejidad amortizada es el tiempo total por operación, evaluado sobre una secuencia de múltiples operaciones. La idea es garantizar el tiempo total de toda la secuencia, permitiendo que las operaciones individuales sean mucho más lentas que el tiempo amortizado. Por ejemplo, en nuestro caso, una sola llamada podría tomar $O(\log n)$ en el peor de los casos, pero si lo hacemos m tales llamadas consecutivas terminaremos con un tiempo promedio de $O(f(n))$

Tampoco presentaremos una prueba para esta complejidad de tiempo, ya que es bastante larga y complicada.

Además, vale la pena mencionar que DSU con unión por tamaño / rango, pero sin compresión de ruta funciona en $O(\log n)$ tiempo por consulta.

6. Aplicaciones

Veremos varias aplicaciones de la estructura de datos, tanto los usos triviales como algunas mejoras a la estructura de datos.

- **Componentes conectados en un grafo:** Esta es una de las aplicaciones obvias de DSU. Formalmente el problema se define de la siguiente forma: Inicialmente tenemos un grafo vacío. Tenemos que agregar vértices y aristas no dirigidas, y responder consultas de la forma (a, b) son los vértices a y b en la misma componente conexa del grafo?. Aquí podemos aplicar directamente la estructura de datos y obtener una solución que maneje la adición de un vértice o una arista y una consulta en un tiempo promedio casi constante. Esta aplicación es bastante importante, porque casi el mismo problema aparece en el algoritmo de Kruskal para encontrar un árbol de expansión mínimo. Usando DSU podemos mejorar la $O(m \log n + n^2)$ complejidad a $O(m \log n)$.

- **Buscar componentes conectados en una imagen:** Una de las aplicaciones de DSU es la siguiente tarea: hay una imagen de nxm píxeles. Originalmente todos son blancos, pero luego se dibujan algunos píxeles negros. Desea determinar el tamaño de cada componente conectado blanco en la imagen final. Para la solución, simplemente iteramos sobre todos los píxeles blancos de la imagen, para cada celda iteramos sobre sus cuatro vecinos, y si el vecino es blanco, llamamos $union\_sets$. Así tendremos un DSU con nm nodos correspondientes a los píxeles de la imagen. Los árboles resultantes en la DSU son los componentes conectados deseados. El problema también se puede resolver mediante DFS o BFS , pero el método descrito aquí tiene una ventaja: puede procesar la matriz fila por fila (es decir, para procesar una fila solo necesitamos la fila anterior y la actual, y solo necesitamos una DSU construida para los elementos de una fila) en $O(\min(n, m))$ memoria.

- **Almacenar información adicional para cada conjunto:** DSU le permite almacenar fácilmente información adicional en los conjuntos. Un ejemplo sencillo es el tamaño de los conjuntos: el almacenamiento de los tamaños ya se describió en la sección Unión por tamaño (la información fue almacenada por el representante actual del conjunto). De la misma manera, al almacenarlo en los nodos representativos, también puede almacenar cualquier otra información sobre los conjuntos.

- **Comprimir saltos a lo largo de un segmento / Pintar subarreglos fuera de línea:** Una aplicación común de la DSU es la siguiente: hay un conjunto de vértices y cada vértice tiene una arista saliente hacia otro vértice. Con DSU puede encontrar el punto final, al que llegamos después de seguir todos las aristas desde un vertice de inicio dado, en un tiempo casi constante.

  Un buen ejemplo de esta aplicación es el problema de **pintar subarreglos** . Tenemos un segmento de longitud L, cada elemento tiene inicialmente el color 0. Tenemos que volver a pintar el subarreglo $[l, r]$ con el color c para cada consulta (l, r, c). Al final queremos encontrar el color final de cada celda. Suponemos que conocemos todas las consultas de antemano, es decir, la tarea está fuera de línea.

  Para la solución podemos hacer una DSU, que para cada celda almacene un enlace a la siguiente celda sin pintar. Así inicialmente cada celda apunta a sí misma. Después de pintar un repintado solicitado de un segmento, todas las celdas de ese segmento apuntarán a la celda después del segmento.

  Ahora, para resolver este problema, consideramos las consultas en el **orden inverso** : del último al primero. De esta manera, cuando ejecutamos una consulta, solo tenemos que pintar exactamente las celdas sin pintar en el subarreglo $[l, r]$. Todas las demás celdas ya contienen su color final. Para iterar rápidamente sobre todas las celdas sin pintar, usamos la DSU. Encontramos la celda sin pintar más a la izquierda dentro de un segmento, la volvemos a pintar y con el puntero nos movemos a la siguiente celda vacía a la derecha. Aquí podemos usar la DSU con compresión de ruta, pero no podemos usar la unión por rango/tamaño (porque es importante quién se convierte en el líder después de la fusión). Por lo tanto la complejidad será $O(\log n)$ por unión (que también es bastante rápido).

  Implementación:

```
for (int i = 0; i <= L; i++) { make_set(i); }

for (int i = m-1; i >= 0; i--) {
   int l = query[i].l; int r = query[i].r; int c = query[i].c;
   for (int v = find_set(l); v <= r; v = find_set(v)) {
        answer[v] = c; parent[v] = v + 1;
   }
}
```

Hay una optimización: podemos usar la **unión por rango** , si almacenamos la siguiente celda sin pintar en un arreglo adicional $end[]$. Entonces podemos fusionar dos conjuntos en uno clasificado de acuerdo con sus heurísticas, y obtenemos la solución en $O(f(n))$.

- **Distancias de soporte hasta representativas:**

  A veces, en aplicaciones específicas de la DSU, es necesario mantener la distancia entre un vértice y el representante de su conjunto (es decir, la longitud del camino en el árbol desde el nodo actual hasta la raíz del árbol).

  Si no usamos compresión de ruta, la distancia es solo el número de llamadas recursivas. Pero esto será ineficiente.

  Sin embargo, es posible comprimir la ruta si almacenamos la distancia al padre como información adicional para cada nodo.

  En la implementación es conveniente usar un arreglos de pares para $parent[]$ y la función $find\_set$ ahora devuelve dos números: el representante del conjunto y la distancia a él.

```
void make_set(int v) {
   parent[v] = make_pair(v, 0);
   rank[v] = 0;
}

pair<int, int> find_set(int v) {
   if (v != parent[v].first) {
      int len = parent[v].second;
      parent[v] = find_set(parent[v].first);
      parent[v].second += len;
        }
    return parent[v];
}

void union_sets(int a, int b) {
   a = find_set(a).first;
   b = find_set(b).first;
   if (a != b) {
     if (rank[a] < rank[b])
        swap(a, b);
        parent[b] = make_pair(a, 1);
```

```
        if (rank[a] == rank[b])
            rank[a]++;
        }
}
```

- **Admite la paridad de la longitud de la ruta / Comprobación de bipartididad en línea:**

  De la misma manera que se calcula la longitud del camino hacia el líder, es posible mantener la paridad de la longitud del camino ante él. ¿Por qué está esta aplicación en un párrafo aparte?

  El requisito inusual de almacenar la paridad del camino surge en la siguiente tarea: inicialmente se nos da un gráfico vacío, se le pueden agregar aristas y tenemos que responder consultas del tipo ¿es bipartito el componente conexo que contiene este vértice ?.

  Para resolver este problema, creamos una DSU para almacenar los componentes y almacenamos la paridad del camino hasta el representante de cada vértice. Por lo tanto, podemos verificar rápidamente si agregar una arista conduce a una violación de la bipartición o no: es decir, si los extremos de la arista se encuentran en el mismo componente conectado y tienen la misma longitud de paridad que el líder, entonces agregar esta arista producirá un ciclo de longitud impar, y el componente perderá la propiedad de bipartididad.

  La única dificultad que enfrentamos es calcular la paridad en el $union_find$ método.

  Si añadimos una arista (a, b) que conecta dos componentes conectados en uno, luego, cuando adjunta un árbol a otro, debemos ajustar la paridad. Derivemos una fórmula que calcule la paridad emitida al lider del conjunto que se adjuntará a otro conjunto. Dejar x sea la paridad de la longitud del camino desde el vértice a hasta su líder A ,y y como la paridad de la longitud del camino desde el vértice b hasta su líder B , y t la paridad deseada que tenemos que asignar a B después de la fusión. El camino contiene la de las tres partes: desde B a b, de b a a, que está conectado por una arista y por lo tanto tiene paridad 1, y de a a A . Por lo tanto recibimos la fórmula ($\oplus$ denota la operación XOR):

  $$t = x \oplus y \oplus 1$$

  Por lo tanto, independientemente de cuántas uniones realicemos, la paridad de las arista se lleva de un líder a otro.

  Damos la implementación del ESD que apoya la paridad. Como en la sección anterior, usamos un par para almacenar el antepasado y la paridad. Además, para cada conjunto, almacenamos en la matriz bipartite[]si todavía es bipartito o no.

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}
```

```
pair<int, int> find_set(int v) {
   if(v!=parent[v].first){
      int parity = parent[v].second;
      parent[v] = find_set(parent[v].first);
      parent[v].second ^= parity;
   }
   return parent[v];
}

void add_edge(int a, int b) {
   pair<int, int> pa = find_set(a);
   a = pa.first;
   int x = pa.second;

   pair<int, int> pb = find_set(b);
   b = pb.first;
   int y = pb.second;

   if(a == b) {
      if(x == y)
         bipartite[a] = false;
   } else {
      if (rank[a] < rank[b]) swap (a, b);
      parent[b] = make_pair(a, x^y^1);
      bipartite[a] &= bipartite[b];
      if (rank[a] == rank[b]) ++rank[a];
   }
}

bool is_bipartite(int v) {
   return bipartite[find_set(v).first];
}
```

- **RMQ sin conexión (consulta de rango mínimo) en $O(f(n))$ en promedio / El truco de Arpa:**

  Nos dan un arreglo a[] y tenemos que calcular algunos mínimos en segmentos dados de del arreglo. La idea para resolver este problema con DSU es la siguiente: iteraremos sobre la matriz y cuando estemos en el ielemento th responderemos todas las consultas (L, R)con R == i. Para hacer esto de manera eficiente, mantendremos una DSU usando los primeros ielementos con la siguiente estructura: el padre de un elemento es el siguiente elemento más pequeño a la derecha. Luego, utilizando esta estructura, la respuesta a una consulta será $a[find\_set(L)]$, el número más pequeño a la derecha de L.

  Obviamente, este enfoque solo funciona sin conexión, es decir, si conocemos todas las consultas de antemano.

  Es fácil ver que podemos aplicar compresión de ruta. Y también podemos usar Unión por rango, si almacenamos el líder real en una matriz separada.

```
struct Query {
   int L, R, idx;
};

vector<int> answer;
vector< vector<Query> > container;

//container[i]contiene todas las consultas con R == i.

stack<int> s;
for (int i = 0; i < n; i++) {
   while (!s.empty() && a[s.top()] > a[i]) {
      parent[s.top()] = i;
      s.pop();
   }
   s.push(i);
   for (Query q : container[i]) {
      answer[q.idx] = a[find_set(q.L)];
   }
}
```

Hoy en día este algoritmo se conoce como el truco de Arpa. Lleva el nombre de AmirReza Poorakhavan, quien descubrió y popularizó esta técnica de forma independiente. Aunque este algoritmo ya existía antes de su descubrimiento. Offline LCA (ancestro común más bajo en un árbol) en $O(f(n))$ de media

- **Almacenar el DSU explícitamente en una lista establecida / Aplicaciones de esta idea al fusionar varias estructuras de datos:**

  Una de las formas alternativas de almacenar el DSU es la preservación de cada conjunto en forma de una lista explícitamente almacenada de sus elementos . Al mismo tiempo, cada elemento también almacena la referencia al representante de su conjunto.

  A primera vista esto parece una estructura de datos ineficiente: al combinar dos conjuntos tendremos que agregar una lista al final de otra y actualizar el liderazgo en todos los elementos de una de las listas.

  Sin embargo, resulta que el uso de una heurística de ponderación (similar a Unión por tamaño) puede reducir significativamente la complejidad asintótica: $O(m + n \log n)$ actuar m consultas sobre el n elementos.

  Bajo heurística de ponderación queremos decir que siempre agregaremos el más pequeño de los dos conjuntos al conjunto más grande . Agregar un conjunto a otro es fácil de implementar $union_sets$ y llevará un tiempo proporcional al tamaño del conjunto agregado. Y la búsqueda del líder en $find\_set$ tomará $O(1)$ con este método de almacenamiento.

  Demostremos la complejidad del tiempo. $O(m + n \log n)$ para la ejecución de m consultas Arreglaremos un elemento arbitrario. x y cuente con qué frecuencia se tocó en la operación

de combinación $union_sets$. Cuando el elemento x se toca la primera vez, el tamaño del nuevo conjunto será al menos 2 . Cuando se toca por segunda vez, el conjunto resultante tendrá un tamaño de al menos 4 , porque el conjunto más pequeño se suma al más grande. Esto significa que x solo se puede mudar como máximo $\log n$ fusionar operaciones. Así la suma de todos los vértices da $O(n \log n)$ más $O(1)$ para cada solicitud.

Aquí hay una implementación:

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set(int v) {
   lst[v] = vector<int>(1, v);
   parent[v] = v;
}

int find_set(int v) {
   return parent[v];
}

void union_sets(int a, int b) {
   a = find_set(a);b = find_set(b);
   if (a != b) {
      if (lst[a].size() < lst[b].size())
         swap(a, b);
         while (!lst[b].empty()) {
            int v = lst[b].back();
            lst[b].pop_back();
            parent[v] = a;
            lst[a].push_back (v);
         }
      }
   }
}
```

Esta idea de agregar la parte más pequeña a una parte más grande también se puede usar en muchas soluciones que no tienen nada que ver con DSU.

Por ejemplo, considere el siguiente problema : nos dan un árbol, cada hoja tiene un número asignado (el mismo número puede aparecer varias veces en diferentes hojas). Queremos calcular la cantidad de números diferentes en el subárbol para cada nodo del árbol.

Aplicando a esta tarea la misma idea, es posible obtener esta solución: podemos implementar un DFS , que devolverá un puntero a un conjunto de enteros: la lista de números en ese subárbol. Luego, para obtener la respuesta para el nodo actual (a menos, por supuesto, que sea una hoja), llamamos a DFS para todos los elementos secundarios de ese nodo y fusionamos todos los conjuntos recibidos. El tamaño del conjunto resultante será la respuesta para el nodo actual. Para combinar conjuntos múltiples de manera eficiente, solo aplicamos

la receta descrita anteriormente: fusionamos los conjuntos simplemente agregando los más pequeños a los más grandes. Al final obtenemos un $O(n \log^2 n)$ solución, porque un número solo se agregará a un conjunto como máximo $O(\log n)$ veces.

- **Almacenar el DSU manteniendo una estructura de árbol clara / Búsqueda de puentes en línea en $O(f(n))$de media:**

  Una de las aplicaciones más poderosas de DSU es que le permite almacenar árboles comprimidos y sin comprimir . La forma comprimida se puede usar para fusionar árboles y para verificar si dos vértices están en el mismo árbol, y la forma sin comprimir se puede usar, por ejemplo, para buscar caminos entre dos vértices dados u otros recorridos de la estructura del árbol. .

  En la implementación, esto significa que, además de la matriz de ancestros comprimida, $parent[]$ necesitaremos mantener la matriz de ancestros sin comprimir $real_parent[]$. Es trivial que mantener esta matriz adicional no empeore la complejidad: los cambios solo ocurren cuando fusionamos dos árboles, y solo en un elemento.

  Por otro lado, cuando se aplica en la práctica, a menudo necesitamos conectar árboles usando una arista específica que no sea usando los dos nodos raíz. Esto significa que no tenemos más remedio que volver a enraizar uno de los árboles (hacer que los extremos de la arista sean la nueva raíz del árbol).

  A primera vista parece que este rerooteo es muy costoso y empeorará mucho la complejidad del tiempo. De hecho, para enraizar un árbol en el vértice v debemos ir desde el vértice a la raíz anterior y cambiar de dirección en parent[]y $real_parent[]$ para todos los nodos en ese camino.

  Sin embargo, en realidad no es tan malo, podemos volver a enraizar el más pequeño de los dos árboles de forma similar a las ideas de las secciones anteriores, y obtener $O(\log n)$ de media.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando esta estructura:

- DMOJ - Closing the Farm
- DMOJ - Cerrando la Granja
- Codeforces - 1620A - Equal or Not Equal
- Codeforces - 277A - A. Learning Languages