Ficha 09

Árbol de Expansión Mínimo: El Algoritmo de Kruskal

1.) El Algoritmo de Kruskal para el Árbol de Expansión Mínimo.

Hemos visto que el *algoritmo de Prim* permitía calcular un árbol de expansión mínimo (*AEM*) para un grafo de *n* vértices y *m* arcos, no dirigido, ponderado y conexo, en un tiempo *O(m * log(n))*, aplicando una estrategia ávida: tomar siempre el arco de menor peso que permita unir dos vértices en subgrafos separados. Hemos demostrado que si el grafo es efectivamente conexo, el algoritmo de Prim siempre obtiene un *AEM* para el grafo de partida

Existe al menos otro conocido algoritmo que permite calcular *AEM*, designado como *algoritmo de Kruskal*. Se trata de otro *algoritmo ávido*, en principio simple plantear, cuyo tiempo de ejecución es del mismo orden que el de Prim (*O(m * log(n)*), pero que también puede aplicarse sobre un grafo no conexo. En este caso finalmente obtiene un *bosque de árboles de expansión mínimo*, esto es: para cada componente conexa del grafo de partida, obtiene un AEM. Obviamente, si el grafo contiene una sola componente conexa, entonces se obtiene un *AEM* para esa componente, como el algoritmo de Prim.

La idea básica del *algoritmo de Kruskal* es tomar el conjunto de arcos del grafo original, ordenarlo de menor a mayor por pesos, y luego ir incorporando estos arcos al *AEM* parcial uno a uno, en orden de menor a mayor peso, *pero sólo si el arco que se incorpora no produce un ciclo en el AEM actual*. Si el grafo tiene *n* vértices, el proceso puede detenerse en cuanto se hayan incorporado efectivamente *n-1* arcos al *AEM*. La regla ávida de incluir el próximo arco de menor peso que no produzca un ciclo es correcta, y se deduce del propio proceso de construcción: La regla ávida ignora los arcos que producen ciclos y sólo se detiene cuando todos los vértices han sido incorporados, con lo cual se logra siempre un árbol de expansión. Y como los arcos vienen ordenados de menor a mayor, la suma de los arcos que se incorporen al árbol de expansión será siempre mínima, logrando entonces un *AEM*.

Un esquema de pseudocódigo muy general para el algoritmo de Kruskal sería el siguiente:

Entrada: un grafo *G(V, A)*, no dirigido, ponderado, posiblemente conexo y posiblemente denso (*V* es el conjunto de *n* vértices, *A* es el conjunto de *m* arcos).

Salida: un *árbol de expansión mínimo* para *G* (si *G* es conexo) o bien, un *bosque de árboles de expansión mínimo* para *G* (si *G* no es conexo).

Pseudocódigo:

Kruskal(G):
- Sea AO = A, pero ordenado de menor a mayor por pesos.
- Sea T una lista de arcos vacía para el AEM parcial.
- Para k en [1..m] y (cantidad de vértices en T) != n:

- Tome el arco a_k de AO.
- Si a_k no produce un ciclo en T:
  - agregue a_k a T.

- retorne T.

El ordenamiento pedido en el primer paso tendrá un tiempo de ejecución *O(m * log(m)) = O(m * log(n))*. Luego se lanza un ciclo que en el peor caso hará *m* repeticiones y en cada una de ellas hay que hacer una comprobación de existencia de camino cíclico en *T*. Está claro que ese es el paso crítico: si esa comprobación de camino cíclico se hace en un tiempo no mayor a *O(log(m))*, entonces el ciclo completo se ejecuta en *O(m * log(m)) = O(m * log(n))* nuevamente, y ese sería el tiempo de ejecución total para el *algoritmo de Kruskal*.

¿Cómo hacer la comprobación de caminos cíclicos en un tiempo no mayor a *log(m)*? Si simplemente se comienza por uno de los vértices *u* a comprobar, y se recorre el grafo en profundidad para saber si el arco *(u, v)* forma un ciclo, ese recorrido insumiría un tiempo *O(n)* (lineal en el número de vértices) ya que tendría que comprobarlos a todos en el peor caso, haciendo que el ciclo completo del *algoritmo de Kruskal* termine siendo *O(m*n)* y eso es mayor que *O(m * log(n))*.

Para resolver este problema existen estructuras de datos diseñadas específicamente para comprobar en forma rápida si dos elementos pertenecen a un mismo conjunto, y unir dos conjuntos en forma también rápida si fuese necesario. Estas estructuras se designan como *estructuras de Union-Find* (o *estructuras de Unión-Pertenencia*, o también *estructuras de Partición*, según la fuente que se consulte). El uso una estructura *Union-Find* permite que el proceso de comprobar la existencia de un ciclo en el *algoritmo de Kruskal*, así como el rearmado de la estructura para la siguiente comprobación, lleve un *tiempo amortizado*[1] *O(log(n))*, garantizando así que el tiempo completo será efectivamente *O(m * log(n))*.

2.) Estructuras de Union-Find[2].

En muchos problemas se requiere poder comprobar en forma rápida si dos elementos u objetos *u*, *v* pertenecen al mismo conjunto. Los conjuntos o grupos con los que se está trabajando se designan también como *clases de equivalencia*, y debido a que en este tipo de problemas la pertenencia o no a una misma clase de equivalencia suele depender de algún tipo de relación entre los objetos *u* y *v*, es que al trabajar con grafos se presente con cierta frecuencia el problema. Por ejemplo, los objetos *u* y *v* podrían representar vértices de un grafo y el problema de saber si existe un camino que los conecta podría transformarse en una consulta de la forma: *¿pertenecen u y v a la misma componente conexa?*

Desde el punto de vista del *algoritmo de Kruskal* que presentamos en la sección anterior, el hecho de poder determinar con rapidez si dos vértices *u* y *v* están ya conectados entre sí (o sea: pertenecen a la misma componente conexa, lo que en el contexto de *algoritmos Union-Find* es lo mismo que decir que *u*

[1] El *análisis amortizado* toma en cuenta el *tiempo promedio general* de realizar *n* operaciones, y luego acepta que algunas operaciones en particular (siempre que sean pocas) puedan ser más costosas que el tiempo promedio, asumiendo que otras de las *n* operaciones serán a su vez mucho más rápidas. Un tiempo amortizado *O(log(n))*, entonces, significa que en promedio se espera un tiempo logarítmico, pero podrán ocurrir puntualmente algunas pocas operaciones con tiempo mayor, como *O(n)*.

[2] La base para toda la explicación que sigue en cuanto a la implementación y análisis de las estructuras *Union-Find*, está inspirada en los siguientes libros de texto:

- Weiss, M. A. (2000). "*Estructuras de Datos en Java – Compatible con Java 2*" (capítulo 23) . Madrid: Addison Wesley.
- Sedgewick., R. (1995). "*Algoritmos en C++*" (capítulo 30). Reading: Addison Wesley – Díaz de Santos.

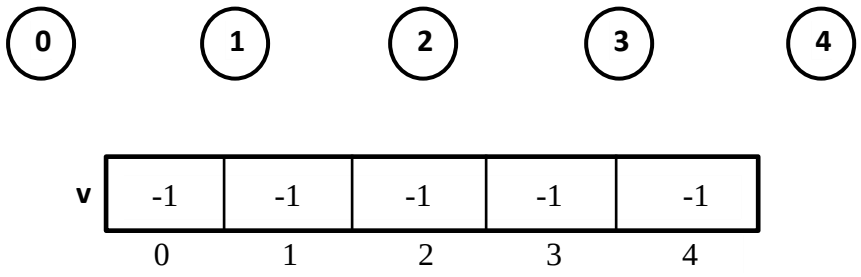
y *v* pertenecen a la misma clase de equivalencia) es importante, ya que si *u* y *v* están ya conectados en el *AEM* parcial y se agrega un nuevo arco que una a *u* con *v*, entonces inmediatamente se tendría un ciclo y el arco debería rechazarse.

El problema de la verificación de pertenencia de dos elementos al mismo conjunto ha sido estudiado con detenimiento y se plantearon diversas estrategias para implementar las estructuras de datos asociadas que se han propuesto, como las ya citadas *Union-Find*. Todas las estrategias e implementaciones sugeridas a lo largo del tiempo son sorprendentemente compactas y simples de implementar, aunque en contrapartida el análisis en cuanto al tiempo de ejecución es intrincado y bastante complejo.

En esencia, se trata de implementar una estructura de datos que partiendo de un *bosque* de *n* objetos separados, permita formular consultas para saber a qué conjunto o clase de equivalencia pertenece un objeto *u* (operación *find(u)*) y tal que admita también acciones como la unión de los conjuntos a los que pertenecen *u* y *v* si fuese el caso que ambos están en conjuntos distintos (operación *union(u, v)*). Sucesivas llamadas a *union(u, v)* para distintos objetos *u* y *v*, hacen que la estructura vaya creciendo y abarcando cada vez más elementos, haciendo a su vez que la operación *find(u)* requiera eventualmente cada vez más trabajo y tiempo. Las distintas estrategias existentes para la implementación de la estructura *Union-Find* se basan justamente en el problema de cómo optimizar una de las dos operaciones sin que ello implique pérdida de eficiencia en la otra. El resultado general al que se ha arribado, y que constituye prácticamente un punto de equilibrio, es que combinando adecuadamente diversas técnicas, se logra que un conjunto de *m* operaciones aleatorias de *find()* y *union()* entremezcladas, alcance un tiempo en el peor caso de *O(m * log(m))*.

La idea básica para implementar una estructura de *Union-Find* consiste en suponer que los *n* objetos de entrada conforman *n* conjuntos o clases de equivalencia distintos (o lo que es lo mismo, se parte de un *bosque* de *n* árboles, que sólo contienen a su raíz). Este bosque de *n* raíces puede implementarse simplemente con un arreglo de *n* componentes en el que el índice de cada casillero coincida con el número de orden de cada objeto (si los objetos están numerados de *1* a *n*, puede declararse en Java un arreglo de *n+1* componentes y simplemente ignorar el casillero *0*). Además, por razones que luego explicaremos, será conveniente inicializar a cada casillero con el valor *-1* (por ahora, podemos asumir que un *-1* en la casilla *v[k]* indica que el objeto *k* es la raíz de un árbol).

A modo de ejemplo, supongamos que tenemos *n = 5* objetos. Podemos verlos como en la gráfica siguiente:

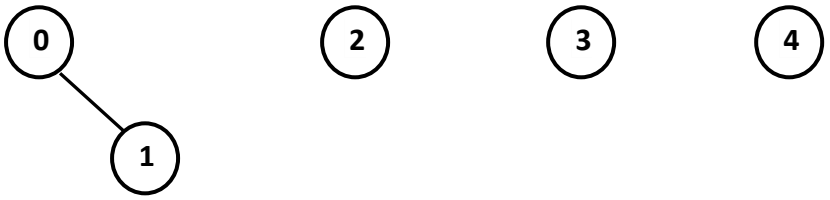


El esquema de partida para representar lo expuesto en Java, se ve en la clase *UnionFind* que sigue:

```
public class UnionFind
{
   private int items[]; // el arreglo que contiene a los elementos…
   private int groups;  // la cantidad de grupos distintos…

   public UnionFind()
   {
      this(100);
   }

   public UnionFind(int n)
   {
      int t = n;
      if(t <= 0) t = 100;

      items = new int[t];
      for(int i = 0; i < t; i++) items[i] = -1;

      groups = t;
   }

   public int countGroups()
   {
      return groups;
   }

   public int find(int idx)
   {
         // proceso de búsqueda del lider de idx…
   }

   public boolean union(int idx1, int idx2)
   {
         // proceso de unión de los grupos de idx1 e idx2…
   }

   public String toString()
   {
      StringBuilder r = new StringBuilder("Grupos: "+groups+ " [");

      for(int i = 0; i < items.length; i++)
      {
         r.append( "(" + i + " : " + items[i] + ")");
         if(i != items.length - 1) r.append(" ");
      }

      r.append("]");
      return r.toString();
   }
}
```

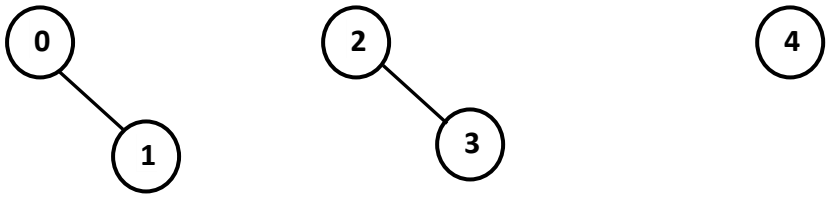
De aquí en adelante, cada clase de equivalencia o grupo de elementos tendrá uno que actuará como *lider* de ese grupo y en cada casillero del arreglo debe almacenarse el índice del elemento lider de ese grupo. Dos elementos *u*, *v* pasan a formar parte del mismo grupo como resultado de una invocación a la operación *union(u, v)*, la cual primero verificará que *u* y *v* pertenezcan a grupos distintos, y sólo en ese caso procederá a unir las dos clases en una sola.

Por ejemplo, si partiendo del estado mostrado en la gráfica anterior, si se hace algo como *union(0, 1)* el resultado será que los elementos 0 y 1 pasarán a formar parte del mismo grupo (y ahora formarán un árbol con dos elementos). Si decidimos que el 1 pase a formar parte del grupo liderado por el 0, entonces *v[1]* debe quedar valiendo 0, y *v[0]* (la casilla del lider) debería seguir valiendo -1 (justamente para indicar que el 0 sigue siendo el lider). Sin embargo, por razones que favorecen una implementación más eficiente de la operación *find()*, lo que haremos será un pequeño ajuste: en cada casilla que represente a un lider de clase, almacenaremos el *tamaño de esa clase* (la cantidad de elementos) pero expresado como un *número negativo*, para evitar ambigüedad:



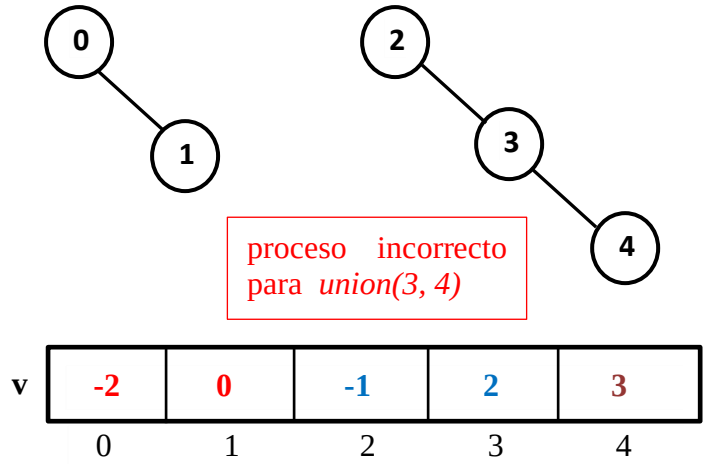
v	-2	0	-1	-1	-1
	0	1	2	3	4

Si queremos saber a qué grupo pertenece un elemento *u*, invocamos a la operación *find(u)*, la cual simplemente retorna el índice del lider del grupo al que pertenece *u*, o un valor negativo si *u* no existe (no es un índice válido). Así, *find(1)* retornará un 0, y *find(0)* retornará también un 0, ya que el 0 es parte del grupo del cual el mismo 0 es lider. Una invocación a *find(2)* retornará un 2 por la misma razón. De esta forma, para saber si dos elementos están en el mismo grupo, sólo se requiere acceder a la casilla de cada uno y comprobar el valor retornado, lo cual en principio puede hacerse en *tiempo constante*. Siguiendo con el ejemplo gráfico anterior, supongamos que ahora hacemos *union(2, 3)* y decidimos dejar al 2 como lider. Todo quedaría así:



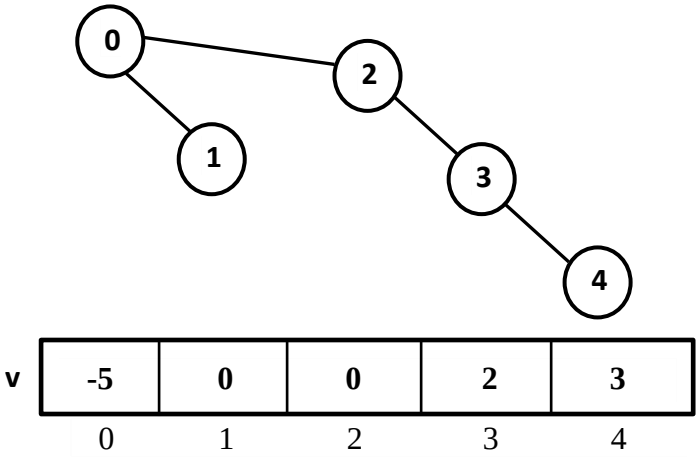
v	-2	0	-2	2	-1
	0	1	2	3	4

Si ahora hacemos *union(3,v4)* comienzan a verse los potenciales problemas: supongamos que se decide poner al 4 en el grupo del 3. Si bien la gráfica del árbol puede quedar clara, lo que ocurre en el arreglo de soporte es que el 4 tendrá como lider al 3 (y no directamente al 2, que es el verdadero lider del grupo):

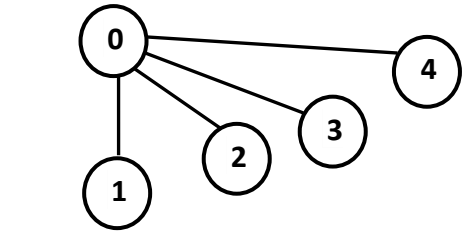


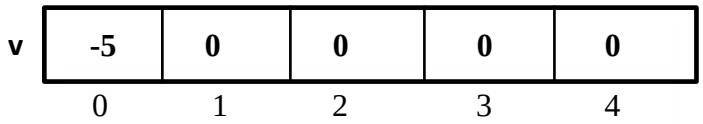
Si en estas condiciones invocamos a *find(4)* para saber cuál es su lider, el método retornaría un 3, cuando debería retornar un 2. Frente a esta situación, podríamos replantear la operación *union()* para que una vez que haga la unión modifique los líderes de todos los elementos de un grupo para que apunten al lider correcto, lo que implica recorrer *todo ese árbol desde el elemento que se acaba de unir hacia arriba y hacia abajo*, haciendo que la operación completa lleve demasiado tiempo: en la figura anterior, si el 4 tuviese hijos que hasta ese momento tenían al 4 como lider, debería cambiarse el lider de todos esos hijos para que deje de ser el 4 y pase a ser el 2…

Para evitar estos problemas sin cargar con todo el trabajo en la operación *union()*, otra salida sería *relajar* la actualización de líderes y hacerla en *forma parcial* cada vez que se hace un *find(u)*: este método ahora deberá entrar en la casilla *v[u]*, *subir* (y sólo subir) por su árbol hasta el lider, recordar cuál es ese lider, y luego *volver a recorrer el árbol* desde *u* hacia arriba, cambiando el lider de todos los elementos en el camino desde el lider hasta *u*. Esta estrategia gráficamente "aplana" el árbol y disminuye la longitud de los caminos desde el lider hacia el resto de los elementos, y se designa como estrategia de *búsqueda con compresión de caminos*. Así, si en algún momento se tuviese un árbol como el que se ve en la gráfica siguiente:



entonces una invocación a *find(4)* provocaría un ajuste de ese árbol para dejarlo como se ve en la figura a continuación:





Con estas ideas, el método *find()* de la clase *UnionFind* puede implementarse así:

```
public int find(int idx)
{
   if(idx < 0) return -1;

   int i = idx;
   while(items[i] >= 0) i = items[i];

   int ix = idx;
   while(items[ix] >= 0)
   {
      int aux = ix;
      ix = items[ix];
      items[aux] = i;
   }

   return i;
}
```

Lo anterior fuerza a *find()* a realizar trabajo extra y su tiempo de respuesta ya no será constante, lo cual lleva a pensar con más detalle lo que debe hacer *union(u, v)* al fusionar las clases de equivalencia de *u* y de *v*: una idea es que cuando se haga la fusión siempre se decida que *el grupo con menos elementos quede como hijo del que tiene más elementos*: de esta forma, se puede esperar que en promedio haya menos actualizaciones pendientes que hacer cada vez que se invoca a *find()*. Por esta razón es que en cada casilla que sea de un lider se almacenan los negativos de los tamaños de cada grupo: esto evidentemente ayuda a decidir qué grupo tiene menos elementos, y por lo tanto cuál debe ir como hijo. Esta estrategia se conoce como *unión por equilibrado de pesos*, y la combinación entre ésta para la operación *union()* y la compresión de caminos para hacer *find()* es la que hemos elegido para la implementación que se muestra como ejemplo en el modelo adjunto.

La operación *union(u, v)* entonces, debe invocar a *find(u)* y a *find(v)* para saber si ambos están en clases diferentes, y en ese caso tomar el grupo de tamaño menor, sumar ese tamaño al lider del grupo mayor, y ajustar el lider del menor para que ahora sea igual al lider del mayor (sin preocuparse de ajustar los líderes de los descendientes del menor). El método *union()* de la clase *UnionFind* puede implementarse entonces así:

```
public boolean union(int idx1, int idx2)
{
   int lx1 = find(idx1);
   int lx2 = find(idx2);

   if(lx1 == -1 || lx2 == -1 || lx1 == lx2) return false;

   if(items[lx2] < items[lx1])
   {

      items[lx2] += items[lx1];
      items[lx1] = lx2;
   }
   else
   {
      items[lx1] += items[lx2];
      items[lx2] = lx1;
   }

   groups--;
   return true;
}
```

Existen otras estrategias para resolver estos problemas, pero las dejamos para ser exploradas por los alumnos. Como ya indicamos, con algo de trabajo se puede probar que el tiempo promedio de ejecución de *m* operaciones aleatorias (entremezclando invocaciones a *find()* con invocaciones a *union()*) para crear una estructura completa, estará en *O(m * log(m))*, lo cual es lo que se esperaba para el *algoritmo de Kruskal también*, y es muy favorable para su uso en otros problemas que requieran la aplicación sistemática de tests de pertenencia.