Árbol Generador Mínimo

Jonás Levy Alfie[1]

[1] Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Algoritmos y Estructuras de Datos III

Clase de hoy!

1. Árbol Generador Mínimo
   - Un problemita
   - El algoritmo de Kruskal
   - Union-Find
     - Representación con listas
     - Representación con bosques
2. Taller
3. Y un problemita más

AGM para los amigos - MST para los *friends*

- **Problema.** Dado un grafo G conexo y con pesos, queremos encontrar un árbol generador mínimo de G.
- A esta altura, muy probablemente sepan cómo resolver este problema, al menos en papel.
- (Tal vez hasta intentaron programar alguna de esas soluciones.)
- Ya en la teórica vieron dos algoritmos para resolver este problema: **Prim** y **Kruskal**, y demostraron que efectivamente construyen un AGM del grafo.

Contenidos

Un problemita

Oreon

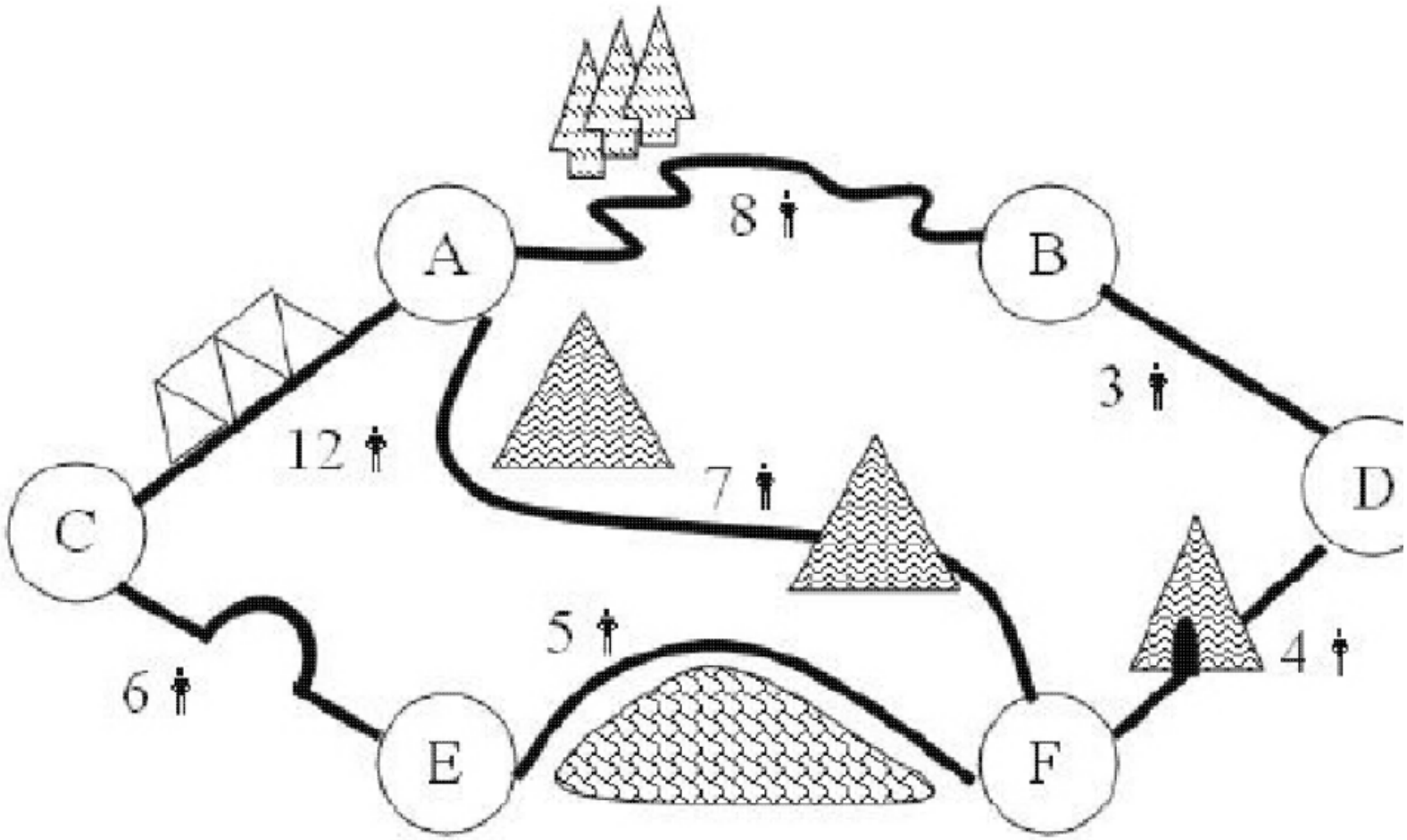
En un futuro distante la civilización se vio obligada a construir ciudades amuralladas, conectadas entre sí por distintos túneles para facilitar el transporte.
Cada ciudad posee un único mineral, el cual se usa para construir y reparar estructuras (los túneles entre ellas). Combinando los minerales de todas las ciudades, forman un material casi indestructible llamado *Oreon* (vital para la supervivencia de los habitantes).
Fuera de las ciudades, hay bárbaros salvajes, que frecuentemente tratarán de atacar. Los túneles, aunque muy resistentes, pueden ser dañados (y la reparación requeriría mucha cantidad de *Oreon*).
Si muchos túneles son dañados, ciudades podrían quedar aisladas, y sería imposible poder formar más *Oreon* (y la civilización colapsaría).

Un problemita

...

Cada túnel requiere una cierta cantidad de soldados para ser protegido.
Su tarea es determinar qué túneles deberán ser protegidos para poder mantener todas las ciudades conectadas (es decir, que no queden ciudades aisladas). Dado que los tiempos son difíciles y los soldados escasean, se debe usar la menor cantidad de soldados posible.

Un problemita (se vería algo así)



Contenidos

Kruskal

En la clase de hoy nos concetraremos en los detalles de implementación para el algoritmo de **Kruskal**, y resolveremos cuestiones sobre la misma.
Recordemos la idea básica del algoritmo:

Kruskal

En la clase de hoy nos concetraremos en los detalles de implementación para el algoritmo de **Kruskal**, y resolveremos cuestiones sobre la misma.
Recordemos la idea básica del algoritmo:

Kruskal (idea)

Comenzar con un conjunto vacío de aristas.
En cada iteración, tomar la arista de menor peso que no forme un ciclo con las que ya tenemos.
Cuando ya formamos un árbol, terminar. El conjunto de aristas logrado será nuestro árbol generador mínimo.

Kruskal (un pseudocódigo)

```
Kruskal(G = (V, E))
    T = {}
    while |T| < |V| - 1 do
        Tomar un eje e de peso mínimo entre los ejes
            que no forman un ciclo con los ejes de T
        T = T + e
    end
    return T
end
```

El algoritmo (como lo vieron en la Teórica)

- Esta versión del algoritmo está muy buena para entenderlo y estudiar su correctitud. Es simple y oculta detalles innecesarios para comprenderlo.
- No está tan buena para analizar la implementación y complejidad.
- Primero necesitamos alguna formulación equivalente que sea más detallada, donde cada operación esté más cerca de ser *atómica*.
- La operación que está muy *cargada* es “Tomar un eje...”.
- **¿Cómo buscamos el mínimo? ¿Cómo sabemos si un eje forma un circuito simple con los ejes de T?**

¿Cómo buscamos el mínimo?

- Podemos tener todos los ejes en alguna estructura, ordenados crecientemente por peso.
- Podemos ir pidiéndole, en orden, los ejes a esa estructura, y luego hacer el chequeo por la existencia del circuito.

¿Cómo buscamos el mínimo?

- Podemos tener todos los ejes en alguna estructura, ordenados crecientemente por peso.
- Podemos ir pidiéndole, en orden, los ejes a esa estructura, y luego hacer el chequeo por la existencia del circuito.
- **Si algún momento agregamos un eje a T, nunca más lo vamos a querer considerar.**
- **Si en algún momento revisamos un eje que forma un circuito simple con los ejes de T, no sólo no lo queremos agregar a T, si no que tampoco nos interesa considerarlo en el futuro.**
- Por lo tanto podemos ir *descartando* los ejes que vamos revisando.

¿Cómo sabemos si un eje forma un circuito simple... ?

Lema

$(u, v) \in E - T$ forma un circuito simple con los ejes de $T \Leftrightarrow u$ y v están en la misma componente conexa de (V, T).

Kruskal (empezando a implementarlo...)

```
Kruskal(G = (V, E))
    A = E
    T = {}
    while |T| < |V| - 1 do
        Tomar un eje (u, v) de A de peso mínimo
        A = A - (u, v)
        if u y v no están en la misma
           componente conexa de (V, T) then
            T = T + (u, v)
        end
    end
    return T
end
```

Análisis de complejidad

- El ciclo principal hace, a lo sumo, $|E|$ iteraciones.
- Para el ordenamiento de A y la extracción de ejes, basta con usar un arreglo y un índice. Pagamos un costo $O(|E| \lg |E|)$ inicial y después cada extracción es $O(1)$.
- **¿Cómo evaluamos si los nodos pertenecen a la misma componente conexa?**

Análisis de complejidad

- El ciclo principal hace, a lo sumo, $|E|$ iteraciones.
- Para el ordenamiento de A y la extracción de ejes, basta con usar un arreglo y un índice. Pagamos un costo $O(|E| \lg |E|)$ inicial y después cada extracción es $O(1)$.
- **¿Cómo evaluamos si los nodos pertenecen a la misma componente conexa?**
- Una forma es con **DFS** desde u o v. En el peor caso, esto cuesta $O(|V|)$. En total, el algoritmo costaría $O(|E| \lg |E| + |E||V|) = O(|E||V|)$.
- ¿Se podrá hacer algo mejor?

Análisis de complejidad (...)

- En cada iteración hacemos dos tipos de operaciones:
  1. Determinar si u y v forman parte de la misma componente de (V, T).
  2. Agregar (u, v) a T y, por ende, unir dos componentes de (V, T).

Análisis de complejidad (...)

- En cada iteración hacemos dos tipos de operaciones:
  1. Determinar si u y v forman parte de la misma componente de (V, T).
  2. Agregar (u, v) a T y, por ende, unir dos componentes de (V, T).
- Con esta estrategia, usando DFS, simplemente agregamos el eje a T (no unimos explícitamente las componentes conexas en otra estructura), con lo cual la segunda operación es muy barata.

Análisis de complejidad (...)

- En cada iteración hacemos dos tipos de operaciones:
  1. Determinar si u y v forman parte de la misma componente de (V, T).
  2. Agregar (u, v) a T y, por ende, unir dos componentes de (V, T).
- Con esta estrategia, usando DFS, simplemente agregamos el eje a T (no unimos explícitamente las componentes conexas en otra estructura), con lo cual la segunda operación es muy barata.
- Como contraparte, nunca estamos *al tanto* de la estructura del grafo, y cada vez que queremos ver si dos vértices están en la misma componente, no nos queda otra que recorrer nuevamente el grafo. Ergo, la primer operación siempre es cara.

Análisis de complejidad (...)

- En cada iteración hacemos dos tipos de operaciones:
  1. Determinar si u y v forman parte de la misma componente de (V, T).
  2. Agregar (u, v) a T y, por ende, unir dos componentes de (V, T).
- Con esta estrategia, usando DFS, simplemente agregamos el eje a T (no unimos explícitamente las componentes conexas en otra estructura), con lo cual la segunda operación es muy barata.
- Como contraparte, nunca estamos *al tanto* de la estructura del grafo, y cada vez que queremos ver si dos vértices están en la misma componente, no nos queda otra que recorrer nuevamente el grafo. Ergo, la primer operación siempre es cara.
- ¡Costos desbalanceados! Hay que encontrar una alternativa más equilibrada...

Visto de otra forma

- Pensemos a las componentes conexas de (V, T) como **conjuntos disjuntos de vértices**.
- Las operaciones anteriores en estos términos serían:
  1. Determinar si u y v forman parte del mismo conjunto.
  2. Unir el conjunto de u con el conjunto de v.

Visto de otra forma

- Pensemos a las componentes conexas de (V, T) como **conjuntos disjuntos de vértices**.
- Las operaciones anteriores en estos términos serían:
  1. Determinar si u y v forman parte del mismo conjunto.
  2. Unir el conjunto de u con el conjunto de v.
- En estos términos es un **problema completamente independiente de Kruskal**.
- Tiene pinta de problema conocido...

Contenidos

Estructuras de datos *Union-Find*

- Son estructuras de datos que resuelven exactamente ese problema.
- Manejan conjuntos *disjuntos*.
- Son capaces de unir dos conjuntos (*union*), y determinar a qué conjunto pertenece cierto elemento (*find*).
- Las dos implementaciones más populares: representación de conjuntos **con listas** y **con bosques**.

Estructuras de datos *Union-Find*

- Para identificar a cada conjunto, se definen **un elemento *representante*** de cada uno.
  Dos conjuntos son iguales $\Leftrightarrow$ tienen el mismo representante.
  Dos elementos están en el mismo conjunto $\Leftrightarrow$ los representantes de sus conjuntos son el mismo.
- Las estructuras de este tipo tienen la siguiente interfaz:
  - MAKE-SET(x). Crea el conjunto $\{x\}$.
  - FIND-SET(x). Devuelve el representante del conjunto de x.
  - UNION(x, y). Une el conjunto de x con el conjunto de y.
- En general no interesa cuál elemento del conjunto es el representante.
  Pero mientras un conjunto no cambie, preguntar varias veces por el representante del conjunto deberá dar siempre el mismo elemento.

Representación con listas - Introducción

- Cada conjunto se representa con una lista. Cada nodo de la lista representa un elemento del conjunto.
- El primer elemento de la lista es el representante.
- Cada nodo x conoce en $next[x]$ al siguiente en la lista, y en $set[x]$ al objeto lista.
- Una lista S conoce en $head[S]$ al primer elemento de la lista, y en $tail[S]$ al último elemento.

Representación con listas - Implementación

```
Make-Set(x)
    Crear una lista S
    head[S] = x
    tail[S] = x
    set[x] = S
end

Find-Set(x)
    return head[set[x]]
end

Union(x, y)
    next[tail[set[x]]] = head[set[y]]
    tail[set[x]] = tail[set[y]]
    node = head[set[y]]
    while node != nil do
        set[node] = set[x]
        node = next[node]
```

Representación con listas - Un dibujito

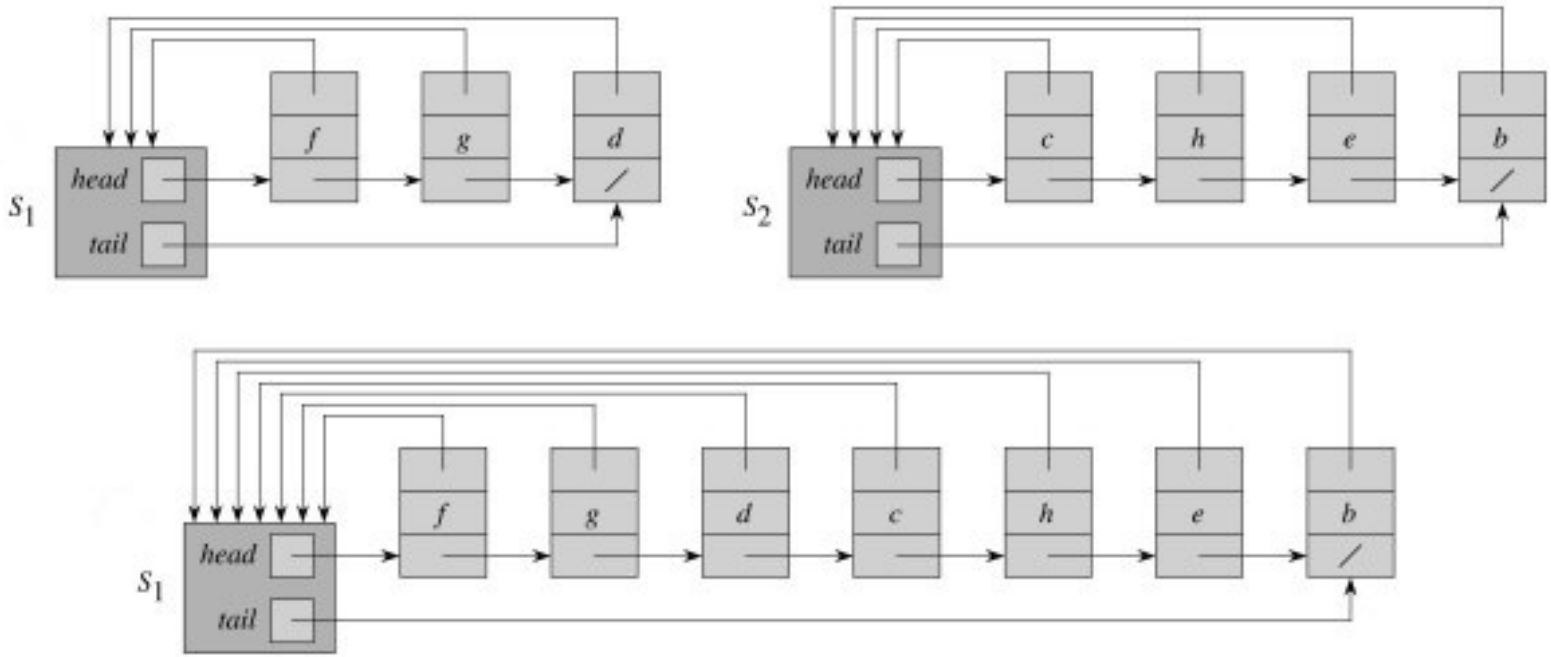


Figura: Dos sets $S1$ y $S2$, y el resultado de hacer *union*.

Representación con listas - Análisis

- MAKE-SET y FIND-SET son $O(1)$.
- UNION(x, y) es $O(size(y))$, donde $size(y)$ es el cardinal del conjunto que contiene a y.

Representación con listas - Análisis

- MAKE-SET y FIND-SET son $O(1)$.
- UNION(x, y) es $O(size(y))$, donde $size(y)$ es el cardinal del conjunto que contiene a y.
- Y si $size(y)$ es mucho más grande que $size(x)$, ¿por qué no cambiamos los punteros del conjunto de x?
- Convendría cambiar el atributo *set* de los elementos del conjunto más chico de los dos...

Representación con listas - Heurística de unión balanceada

- **Idea**: en cada unión, *appendear* el conjunto más pequeño al más grande.
- Para cada lista S, mantenemos un atributo $size[S]$ con la cantidad de elementos que contiene.
- La complejidad de UNION(x, y) ahora es $O(min\{size(x), size(y)\})$.

Representación con listas - Heurística de unión balanceada

- **Idea**: en cada unión, *appendear* el conjunto más pequeño al más grande.
- Para cada lista S, mantenemos un atributo $size[S]$ con la cantidad de elementos que contiene.
- La complejidad de UNION(x, y) ahora es $O(min\{size(x), size(y)\})$.
- No parece una gran mejora...
- Pero la verdadera mejora no es en términos del costo de una operación individual, sino en el **costo amortizado**, es decir, el costo de una secuencia de operaciones.

Representación con listas - Heurística de unión balanceada

Teorema. Usando una representación con listas, junto con la heurística de unión balanceada, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m + n \lg n)$.

Representación con listas - Heurística de unión balanceada

Teorema. Usando una representación con listas, junto con la heurística de unión balanceada, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m + n \lg n)$.

Dem.

- Como hay n MAKE-SET, entonces un conjunto tiene, como mucho, n elementos.
- El costo de todas las operaciones MAKE-SET y FIND-SET es $O(m)$.
- El costo de todas las operaciones UNION lo podemos calcular contando la cantidad de veces que se actualiza el atributo *set* de cada elemento x.
- Cada vez que se actualiza $set[x]$, es porque formaba parte del conjunto más chico. Por lo tanto su tamaño al menos se duplica. En total, sólo se puede duplicar $(\lg n)$ veces.

Representación con bosques - Introducción

- Cada conjunto se representa con un árbol. Cada nodo del árbol representa un elemento del conjunto.
- La raíz del árbol es el representante.
- Cada nodo x tiene un puntero $parent[x]$ a su padre, excepto la raíz que apunta a si misma.

Representación con bosques - Implementación

```
Make-Set(x)
    parent[x] = x
end

Find-Set(x)
    if parent[x] = x then
        return x
     end
     return Find-Set(parent[x])
end

Union(x, y)
    rx = Find-Set(x)
    ry = Find-Set(y)
    parent[ry] = rx
end
```

Representación con bosques - Un dibujito

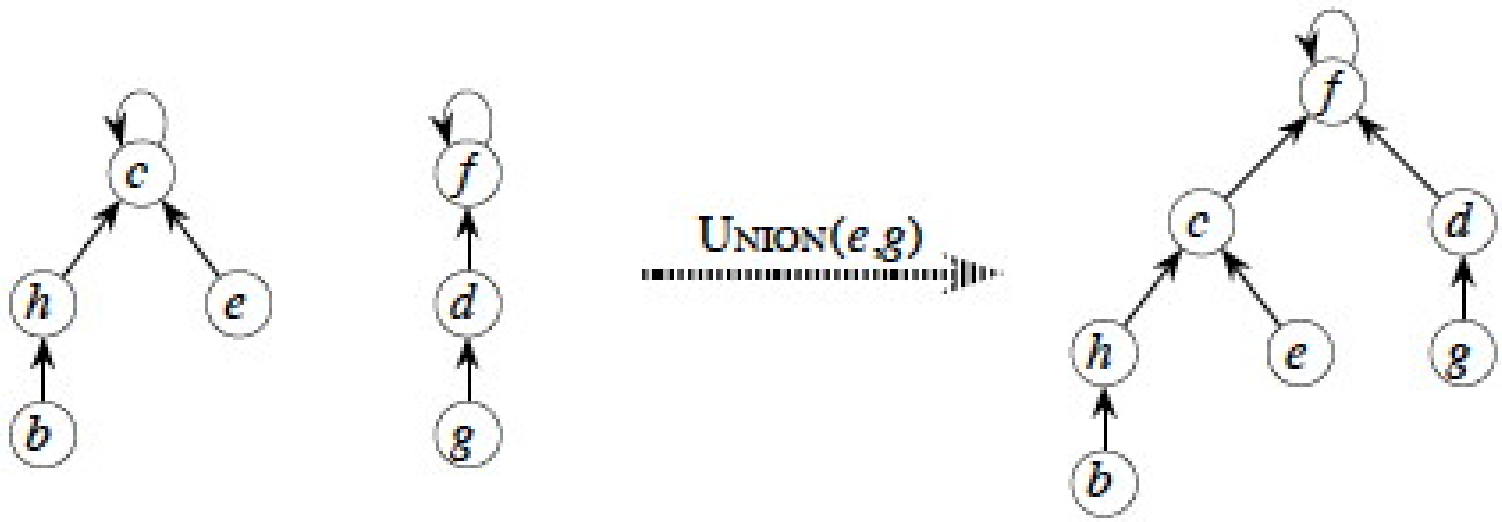


Figura: Dos sets, y el resultado de hacer *union(e, g)*.

Representación con bosques - Análisis

- MAKE-SET es $O(1)$.
- FIND-SET(x) es $O(depth(x))$, donde $depth(x)$ es la profundidad de x en el árbol del que forma parte.
- UNION(x, y) es $O(depth(y))$.

Representación con bosques - Análisis

- MAKE-SET es $O(1)$.
- FIND-SET(x) es $O(depth(x))$, donde $depth(x)$ es la profundidad de x en el árbol del que forma parte.
- UNION(x, y) es $O(depth(y))$.
- **Problema**: los árboles se pueden desbalancear completamente, haciendo que FIND-SET y UNION sean costosas.
- **Solución**: introducir alguna heurística que una adecuadamente los conjuntos para mantener el balance de los árboles.

Representación con bosques - Heurística de unión por altura

- **Idea**: en cada unión, el árbol más pequeño pasa a ser hijo del más grande.
- Para cada elemento x, mantenemos un atributo $height[x]$ con la altura del subárbol del cual es raíz.
- Al unir dos conjuntos de representantes x e y, pueden pasar dos cosas:
  - Si $height[x] \neq height[y]$, el más chico pasa a ser hijo del más grande, y las alturas no cambian.
  - Si $height[x] = height[y]$, elegimos cualquiera de los dos y pasa a ser padre del otro nodo. La altura del padre se incrementa en uno.
- Esta estrategia efectivamente mantiene todos los árboles balanceados.

Representación con bosques - Heurística de unión por altura

Teorema. Usando una representación con bosques, junto con la heurística de unión por altura, a lo largo de una secuencia de operaciones MAKE-SET, FIND-SET y UNION, vale que $2^{height[x]} \leq size(x)$ para toda raíz x.

Representación con bosques - Heurística de unión por altura

Teorema. Usando una representación con bosques, junto con la heurística de unión por altura, a lo largo de una secuencia de operaciones MAKE-SET, FIND-SET y UNION, vale que $2^{height[x]} \leq size(x)$ para toda raíz x.

Dem.

- Inductiva. Queremos ver que al unir dos árboles balanceados, el que se obtiene sigue estando balanceado.
- Si las alturas de los árboles que se unen son distintas, el mayor *absorbe* al menor, y el primero, que ya estaba balanceado, lo sigue estando.
- Si las alturas son iguales, entonces el árbol que resulta tiene casi la misma altura, y está balanceado porque los árboles originales lo estaban.

Representación con bosques - Heurística de unión por altura

Corolario. Usando una representación con bosques, junto con la heurística de unión por altura, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m \lg n)$.

Dem. Por el teorema anterior, las alturas siempre son $O(\lg n)$. Luego, cada una de las $O(m)$ operaciones FIND-SET y UNION tiene costo $O(\lg n)$.

Representación con bosques - Heurística *path compression*

- **Idea**: en cada FIND-SET(x) recorremos el camino desde x hasta la raíz. Podríamos ir linkeando directamente cada uno de los nodos que recorremos con la raíz.
- Las futuras llamadas a FIND-SET para esos ancestros de x son $O(1)$.

```
Find-Set(x)
    if parent[x] != x then
        parent[x] = Find-Set(parent[x])
    end
    return parent[x]
end
```

Representación con bosques - Combinando heurísticas

- Podríamos intentar combinar ambas heurísticas...

Representación con bosques - Combinando heurísticas

- Podríamos intentar combinar ambas heurísticas...
- **Problema**: *path compression* puede romper el atributo *height* de los ancestros del nodo sobre el que hacemos find. Actualizar *height* puede ser caro.

Representación con bosques - Combinando heurísticas

- Podríamos intentar combinar ambas heurísticas...
- **Problema**: *path compression* puede romper el atributo *height* de los ancestros del nodo sobre el que hacemos find. Actualizar *height* puede ser caro.
- **Solución**: relajar la definición de *height*. No necesitamos saber exactamente la altura, nos alcanza con tener alguna cota superior.
- A cada nodo x lo dotamos de un atributo $rank[x]$ que cumple $rank[x] \geq height(x)$.
- Lo inicializamos en 0. Lo utilizamos y actualizamos igual que como hacíamos *height*.

Representación con bosques - Heurística *union by rank*

```
Make-Set(x)
    parent[x] = x
    rank[x] = 0
end

Union(x, y)
    rx = Find-Set(x)
    ry = Find-Set(y)
    if rank[rx] < rank[ry] then
        parent[rx] = ry
    else
        parent[ry] = rx
        if rank[rx] = rank[ry] then
            rank[rx] = rank[rx] + 1
        end
```

Representación con bosques - Heurística *union by rank*

- Notemos que cuando en UNION hay empate de *rank*, sólo le sumamos 1 a la nueva raíz. Podríamos sumarle 71 también. Pero incrementándolo en 1 aseguramos que se mantenga la propiedad del balance $2^{rank[x]} \leq size(x)$.
- Gracias a que *rank* es sólo una cota superior, ahora sí podemos combinar esto con *path compression*.
- **Esta combinación de bosques con *path compression* & *union by rank*, es la más utilizada en estructuras Union-Find.** Es simple y eficiente.
- **The Ultimate Union-Find!**

Representación con bosques - *path compression* & *union by rank* - Un dibujito

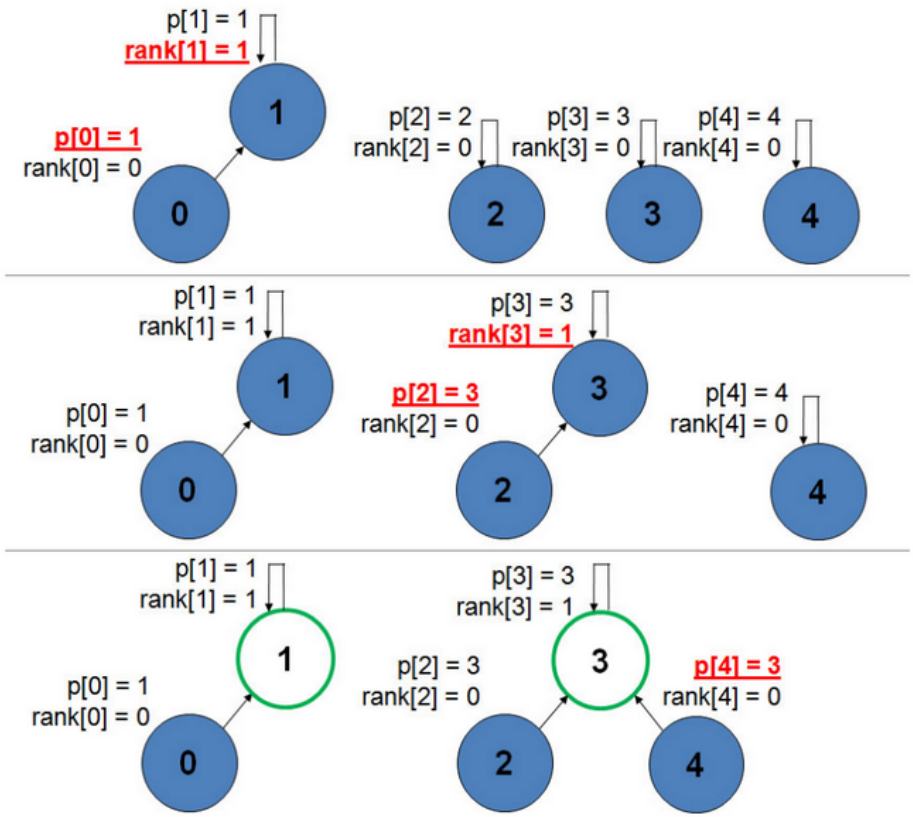


Figura: *union(0, 1)* $\rightarrow$ *union(2, 3)* $\rightarrow$ *union(4, 3)*.

Representación con bosques - *path compression* & *union by rank* - Y otro dibujito

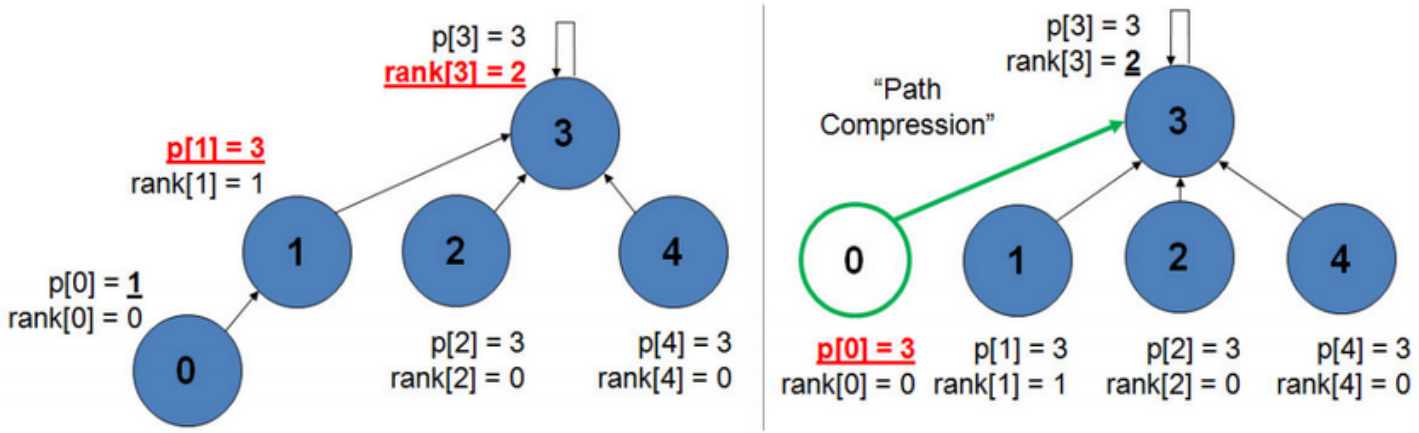


Figura: *union(0, 3)* $\rightarrow$ *find(0)*.

Representación con bosques - *path compression* & *union by rank* - Análisis

Teorema. Usando una representación con bosques, junto con las heurísticas *path compression* y *union by rank*, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m\alpha(n))$.

Representación con bosques - *path compression* & *union by rank* - Análisis

Teorema. Usando una representación con bosques, junto con las heurísticas *path compression* y *union by rank*, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m\alpha(n))$.

- α es la inversa de la función de Ackermann, definida como

$$A(m,n) = (2 \uparrow^{m-2} (n+3)) - 3$$

- $$\alpha(n) = min\{k \in \mathbb{N}_0 : A(k,1) \geq n\}$$

- Algunos valores: $A(0,1) = 2$, $A(1,1) = 3$, $A(2,1) = 7$, $A(3,1) = 2047$, $A(4,1) \gg 10^{80}$.
- $10^{80} \approx$ número de átomos en el universo observable.
- $\alpha(n) \leq 4$ para cualquier valor práctico de n.

Representación con bosques - *path compression* & *union by rank* - Análisis

Teorema. Usando una representación con bosques, junto con las heurísticas *path compression* y *union by rank*, una secuencia de m operaciones MAKE-SET, FIND-SET y UNION, de las cuales n son MAKE-SET, ejecutan en tiempo $O(m\alpha(n))$.

- α es la inversa de la función de Ackermann, definida como

$$A(m,n) = (2 \uparrow^{m-2} (n+3)) - 3$$

-

$$\alpha(n) = min\{k \in \mathbb{N}_0 : A(k,1) \geq n\}$$

- Algunos valores: $A(0,1) = 2$, $A(1,1) = 3$, $A(2,1) = 7$, $A(3,1) = 2047$, $A(4,1) \gg 10^{80}$.
- $10^{80} \approx$ número de átomos en el universo observable.
- $\alpha(n) \leq 4$ para cualquier valor práctico de n.

Dem. Bastante larga y técnica. No la vamos a ver acá.

Representación con bosques - *path compression* & *union by rank*

- Con esta combinación de estructuras y heurísticas, para cualquier valor de n en la práctica, las operaciones del Union-Find se hacen $\approx O(1)$!

Representación con bosques - *path compression & union by rank*

- Con esta combinación de estructuras y heurísticas, para cualquier valor de n en la práctica, las operaciones del Union-Find se hacen $\approx O(1)$!
- **Y volviendo a nuestro Kruskal**, el cual utilizará las operaciones de nuestra estructura Union-Find:
  Lo más caro que terminamos pagando con esta implementación es el costo de ordenar las aristas. De modo que la complejidad termina siendo $O(|E| \lg |E|)$ (bastante mejor de cuando empezamos!)

Y LISTO!

(Uff...)

Tenemos todo para implementar nuestro Kruskal!

(Antes respirar 5 segundos...)

Taller!

- Completar la implementación del Union-Find (con alguna de las versiones que vimos)
- Completar la implementación de Kruskal (y que pase los casos de tests provistos!)
- Completar la implementación del programa principal para que resuelva el problema que vimos al principio

Otras variantes más para pensar:

- Modificar la implementación de Kruskal para que además devuelva las aristas del AGM
- Modificar la implementación de Kruskal para que genere un *Bosque Generador Mínimo* de exactamente C componentes conexas

(Fin!)

- El problema presentado es una versión simplificada del problema original *Oreon*, el cual pueden encontrar siguiendo el link **aquí**. Pueden tratar de resolverlo y subirlo :) La principal diferencia es que se pide indicar qué túneles deben ser protegidos, y las ciudades son representadas con letras.
- Para leer con más detalle sobre Union-Find, Kruskal, Prim y mucho más en el **Cormen**.
  Cormen, T.,Leiserson, C.,Rivest, R.,Stein, C., *"Introduction to Algorithms"*

Un problemita más para pensar y modelar

Anti Brute Force Lock (Versión resumida, link al original)

Se tiene un candado de combinación numérica, con 4 ruedas con los números que van del 0 al 9. Inicialmente el candado se encuentra en la posición 0000.
Se ha modificado este sistema para hacerlo más seguro:

- El candado tiene *N* claves. Pueden destrabarse en cualquier orden.
- El candado posee un botón especial de JUMP el cual "salta" a cualquier clave ya destrabada **sin girar ninguna rueda**.
- El candado se abrirá si y solo si **todas las claves son destrabadas en la menor cantidad de giros de ruedas posible** (sin contar los usos del botón de JUMP). De lo contrario, el candado se resetea.

Dadas *N* claves, calcular la cantidad de giros de ruedas para abrirlo.