

UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

Árboles de expansión mínima y DFS

Propiedad del corte, Kruskal, Prim y componentes fuertemente conexas

Gerson Garrido

Semana 6

Hoy

Sesión A

1. Conectar todo al menor costo
2. La propiedad del corte, y los empates
3. Kruskal y Prim, como corolarios
4. Prim frente a Dijkstra
5. Agrupar con máxima separación

Sesión B

6. DFS con tiempos y tipos de arista
7. Orden topológico
8. Componentes fuertemente conexas
9. Recursión profunda

> **Lectura**
>
> Apuntes de la semana 6. Código en `code/week-06`.

SECCIÓN 01

Conectar todo al menor costo

01

El problema

Un grafo **no dirigido** y conexo, con un peso en cada arista: el precio de tender ese tramo.

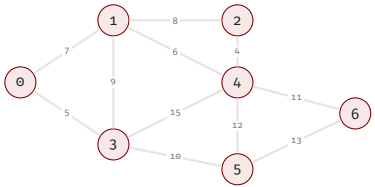
> **Árbol de expansión mínima** ***(minimum spanning tree)***
>
> Un árbol que toca todos los nodos y tiene el menor peso total.

¿Por qué un árbol? Con pesos positivos, si hay un ciclo, quitar una de sus aristas deja todo conectado y sale más barato.

Tres hechos de un árbol T

1. Tiene $V - 1$ aristas. Y $V - 1$ aristas que conectan V nodos son un árbol.
2. Entre dos nodos hay un solo camino.
3. Quitar una arista lo parte en dos pedazos, con un extremo en cada uno.

Tres ideas parecidas que fallan



Por fuerza bruta: 141 árboles de expansión, y el mínimo pesa **43**.

```
terminal

Las n - 1 aristas más baratas:
  aristas: 2-4:4 0-3:5 1-4:6 0-1:7 1-2:8 1-3:9
  costo 39, grupos 3, no es árbol de expansión
Cada nodo elige su arista más barata:
  aristas: 0-3:5 1-4:6 2-4:4 3-5:10 4-6:11
  costo 36, grupos 2, no es árbol de expansión
Árbol de caminos mínimos desde 0:
  aristas: 0-3:5 0-1:7 1-4:6 1-2:8 3-5:10 4-6:11
  costo 47, grupos 1, es árbol de expansión
```

- Menos de 43 es señal de que no es válido.
- Dijkstra da un árbol válido, y no el más barato.

SECCIÓN 02

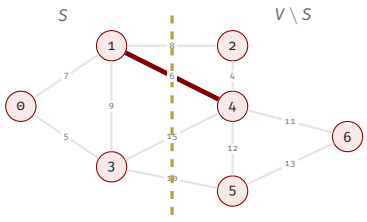
La propiedad del corte

02

Cortes

Corte

Una partición de los nodos en dos conjuntos no vacíos, S y $V \setminus S$. Una arista **cruza** el corte si tiene un extremo en cada lado.



Cruzan 1-2:8, 1-4:6, 3-4:15 y 3-5:10. La más barata es **1-4**.

Lo que vamos a demostrar: la más barata que cruza un corte siempre sirve. Primero, con pesos distintos.

Propiedad del corte, con pesos distintos

Teorema

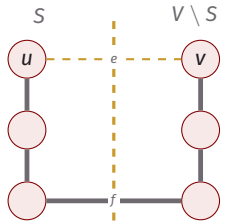
Sea G conexo con pesos distintos. La arista más barata que cruza un corte pertenece a **todo** árbol de expansión mínima.

Técnica. Intercambio: si un árbol T no contiene esa arista, se cambia una de las suyas por ella y sale un árbol más barato.

Sea $e = (u, v)$ la más barata que cruza, y sea T un árbol de expansión sin e.

1. **Hay una arista de T que cruza,** y es del camino de u a v.
2. $T' = T - f + e$ **es un árbol de expansión.**
3. T' **pesa menos.** Entonces T no era mínimo.

La demostración, línea por línea



Las aristas grises son el camino de T entre u y v.

1. T tiene un camino P de u a v. Empieza en S y termina fuera, así que alguna arista f de P cruza. Como $e \notin T$, $f \neq e$.
2. Quitar f parte a T en dos pedazos. P era el **único** camino de u a v, y usaba f: ahora u y v quedan en pedazos distintos.
3. e une esos dos pedazos. T' está conectado y tiene $V - 1$ aristas: es un árbol de expansión.
4. Las dos cruzan, y e es la más barata: $w(e) < w(f)$.
5. $w(T') = w(T) + w(e) - w(f) < w(T)$. ■

Dos consecuencias

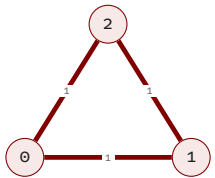
Error común

Sacar cualquier arista que cruce. Si f no está en el camino P, $T - f + e$ tiene el ciclo formado por P y e, y deja dos partes desconectadas. El Paso 2 es lo que justifica la elección: f es la que separa a u de v.

Con pesos distintos hay un solo árbol mínimo

Prim, que veremos enseguida, elige $V - 1$ aristas, cada una la más barata de un corte. Cada una está en **todo** árbol mínimo. Un árbol mínimo tiene exactamente $V - 1$ aristas, así que todos son el mismo.

Empates: el enunciado deja de valer



- Hay tres árboles mínimos. **Ninguna arista está en los tres.**
- ¿Basta con decir "está en **algún** árbol mínimo"? Eso es cierto, pero no alcanza:
- 0-1 es la más barata de $\{0\} \mid \{1, 2\}$. 0-2 lo es de $\{2\} \mid \{0, 1\}$. 1-2 lo es de $\{1\} \mid \{0, 2\}$.
- Las tres juntas forman un ciclo.

Lo que hace falta

Un greedy elige varias aristas, y necesita que **todas juntas** quepan en un mismo árbol mínimo.

Propiedad del corte, general

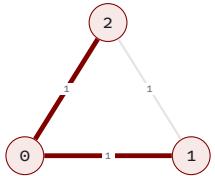
Teorema

Sea F un conjunto de aristas contenido en algún árbol mínimo. Sea un corte que **respeta** a F: ninguna arista de F lo cruza. Sea e de peso mínimo entre las que lo cruzan. Entonces $F \cup \{e\}$ está contenido en algún árbol mínimo.

Sea T un árbol mínimo que contiene a F.

- Si $e \in T$, terminamos.
- Si no, el camino de T entre los extremos de e tiene una f que cruza. $T' = T - f + e$ es un árbol de expansión, por lo mismo de antes.
- $f \notin F$, porque el corte respeta a F. Entonces $F \cup \{e\} \subseteq T'$.
- $w(e) \leq w(f)$, así que $w(T') \leq w(T)$. Como T es mínimo, T' también. ■

El ciclo del triángulo queda descartado



- $F = \{0\text{-}1\}$. El corte $\{2\} \mid \{0, 1\}$ lo respeta.
- Se puede agregar 0-2. Ahora $F = \{0\text{-}1,\ 0\text{-}2\}$.
- Para agregar 1-2 haría falta el corte $\{1\} \mid \{0, 2\}$, y 0-1 lo cruza: **no respeta a F.**

La invariante

Lo elegido cabe en **algún** árbol mínimo, y ese árbol puede ir cambiando en la demostración.

SECCIÓN 03

Kruskal

03

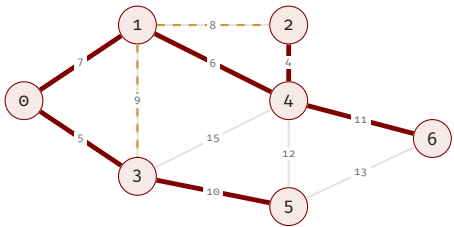
Kruskal: de la más barata a la más cara

pseudocódigo

```
algoritmo kruskal(n, aristas)
    ordenar aristas por peso, de menor a mayor
    crear un union-find con n grupos
    elegidas ← lista vacía
    para cada arista (u, v) en aristas hacer
        si find(u) ≠ find(v) entonces
            union(u, v)
            agregar (u, v) a elegidas
        fin si
    fin para
    devolver elegidas
fin algoritmo
```

- Lo tomado es un **bosque**: árboles sueltos que se van uniendo.
- Una arista forma un ciclo si y solo si sus extremos ya están en un mismo grupo.
- Costo: ordenar $O(E \log E)$, más E operaciones de union-find, casi constantes.
- **Total:** $O(E \log E)$**.** En un grafo simple, $\log E < 2 \log V$.

Kruskal, paso a paso



4-5, 5-6 y 3-4: no. El costo es **43**, con 6 aristas. Coincide con la fuerza bruta.

Por qué Kruskal es correcto

Invariante. F, lo elegido, está contenido en algún árbol mínimo. Vale para cualquier orden entre empates.

Kruskal acepta $e = (u, v)$. Tomemos como S el **grupo de** u en F.

- **El corte respeta a F:** por definición de grupo, ninguna arista de F sale de S.
- e **es de peso mínimo entre las que cruzan.** Sea g una que cruza y se miró antes:
  - si se aceptó, está en F, y nada de F cruza;
  - si se rechazó, sus extremos ya estaban juntos, y los grupos nunca se dividen.
- Por el teorema, $F \cup \{e\}$ cabe en algún árbol mínimo.

Al terminar. Si quedaran dos grupos, como G es conexo hay una arista entre ellos. Al mirarla, sus extremos estaban separados, y se habría aceptado. Luego F tiene $V - 1$ aristas y cabe en un árbol mínimo: es ese árbol. ■

Kruskal en C++

mst.h

```cpp
inline std::vector<Edge> kruskal(int n, const std::vector<Edge>& edges) {
    UnionFind sets(n);
    std::vector<Edge> forest;
    for (const Edge& e : sorted_by_weight(edges)) {
        // Invariante: forest está contenido en algún árbol de expansión mínima.
        if (sets.unite(e.u, e.v)) forest.push_back(e);
    }
    return forest;
}
```

- `unite` devuelve falso si los dos ya estaban en un mismo grupo.
- Devuelve un **bosque**. Si el grafo no es conexo, tiene menos de $n - 1$ aristas, y revisarlo le toca a quien la llama.

SECCIÓN 04

Prim

04

Prim: crecer desde un nodo

pseudocódigo

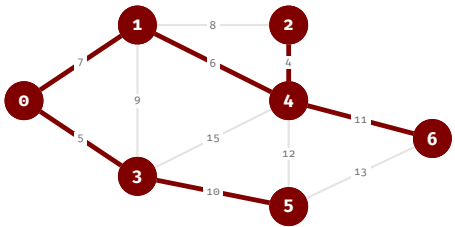
```
algoritmo prim(n, grafo, origen)
    agregar_nodo(origen)
    mientras cola no esté vacía hacer
        sacar la de menor peso: (peso, v, desde)
        si no en_arbol[v] entonces
            agregar (desde, v) a elegidas
            agregar_nodo(v)
        fin si
    fin mientras
fin algoritmo
```

pseudocódigo

```
algoritmo agregar_nodo(u)
    en_arbol[u] ← verdadero
    para cada vecino v de u, con peso w,
     hacer
        si no en_arbol[v] entonces
            meter en cola (w, v, u)
        fin si
    fin para
fin algoritmo
```

Borrado perezoso *(lazy deletion)*. `std::priority_queue` no cambia prioridades: las entradas viejas se quedan, y se descartan al salir.

Prim desde 0, paso a paso



Salen 4-5:12, 5-6:13 y 3-4:15, y se descartan. El costo es **43**. Hubo 11 entradas y 5 descartes.

Por qué Prim es correcto

Propiedad de la cola. Toda arista que cruza $(S, V \setminus S)$ está en la cola, con S los nodos del árbol.

- Sea (u, v) con $u \in S$ y $v \notin S$. Cuando u entró, se metió (w, v, u), porque v estaba afuera.
- No ha salido: si hubiera salido con v afuera, v habría entrado.
- Lo que se descarta tiene los dos extremos en S y no cruza. Luego **la primera que no se descarta es de peso mínimo entre las que cruzan.**

Invariante. F cabe en algún árbol mínimo y es un árbol sobre S. El corte respeta a F, porque F queda adentro de S. Por el teorema, se puede agregar la arista.

Al terminar. Cola vacía: nada cruza. Como G es conexo, S son todos los nodos. ■

El costo de Prim

- Una arista entra a la cola cuando entra al árbol el primero de sus extremos. Cuando entra el segundo, ya no se mete.
- Entonces hay a lo sumo E entradas y E salidas: $O(E \log E)$.
- En un grafo simple, $\log E < 2 \log V$. **Total:** $O(E \log V)$**.**
- Con aristas paralelas, E puede pasar de V^2: se guarda la más barata de cada par, o se deja $O(E \log E)$.
- En un grafo conexo se aceptan $V - 1$ entradas, así que se descartan $E - (V - 1)$.

En el ejemplo

11 aristas, 11 entradas y 5 descartes: $11 - 6 = 5$.

Kruskal y Prim, medidos

terminal

```
$ make bench
build/mst_bench
       V         E  kruskal ms   prim ms    entradas  descartes costo igual
   50000    200000         14.5      35.4      200000     150001        sí
  200000    800000         65.2     186.4      800000     600001        sí
  800000   3200000        288.4    1244.5     3200000    2400001        sí
  100000   3200000        277.2     693.8     3200000    3100001        sí
    3000   3200000        271.0     609.5     3200000    3197001        sí
```

Con `-O2`, el menor de 3 intentos, en el contenedor del curso, en una máquina compartida. Los tiempos dependen de la máquina.

- Entradas y descartes son exactos: E y $E - (V - 1)$.
- En estas corridas Kruskal fue más rápido. Es una medición de esta implementación, no una ley.

SECCIÓN 05

Prim frente a Dijkstra

05

Casi el mismo código, otro problema

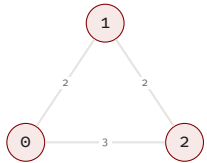
Cambia una sola cosa: la clave con que una arista entra a la cola.

	Prim	**Dijkstra**
Clave	w	$\mathrm{dist}[u] + w$
Minimiza	la suma de pesos del árbol	cada distancia al origen
El origen	no importa para el peso final	es parte del problema
Pesos negativos	funciona	falla

Por qué Prim no ve el signo

Su demostración solo compara pesos entre sí. La de Dijkstra sí usa que no sean negativos.

Error común: el árbol mínimo no da caminos mínimos



terminal

```
$ build/mst_vs_paths
Triángulo: 0-1:2 1-2:2 0-2:3
  árbol de expansión mínima: 0-1:2 1-2:2 costo 4
  de 0 a 2 dentro de ese árbol: 4
  de 0 a 2 en el grafo:         3
  árbol de caminos desde 0:  0-1:2 0-2:3 costo 5
```

Son dos preguntas distintas

Cuánto cuesta construir la red, y cuánto cuesta un viaje dentro de ella. En este grafo, ningún árbol responde bien las dos.

Sumar una constante a todas las aristas

terminal

```
Sumar una constante c a todas las aristas de 0-1:1 1-2:1 0-2:3 2-3:2 1-3:4
 c  árbol de expansión mínima    costo  distancia 0 a 2  camino
 0  0-1 1-2 2-3                  4      2                0-1-2
 1  0-1 1-2 2-3                  7      4                empate
 2  0-1 1-2 2-3                  10     5                0-2
10  0-1 1-2 2-3                  34     13               0-2
```

- **El árbol mínimo no cambia:** todo árbol de expansión tiene $V - 1$ aristas, así que todos suben $(V - 1)\,c$.
- **El camino mínimo sí cambia:** un camino de k aristas sube $k\,c$.
- Por eso sumar una constante no arregla una arista negativa en Dijkstra, y en Prim ni hace falta.

SECCIÓN 06

Agrupar

06

Agrupar con la máxima separación

Separación *(spacing)*

De una partición en $k \geq 2$ grupos: el menor peso entre las aristas con extremos en grupos distintos.

Algoritmo. Para la mayor separación: calcular un árbol mínimo y quitarle las $k - 1$ aristas más caras. Es lo mismo que parar Kruskal cuando quedan k grupos.

Idea que falla

Quitar las $k - 1$ aristas más caras del **grafo**, y no las del árbol. En un grafo con muchas aristas casi nunca separa nada.

Ocho puntos en el plano

terminal

```
$ build/clustering_demo
Puntos: 0=(0,0) 1=(0,2) 2=(2,1) 3=(10,0) 4=(11,2) 5=(12,0) 6=(5,9) 7=(6,10)

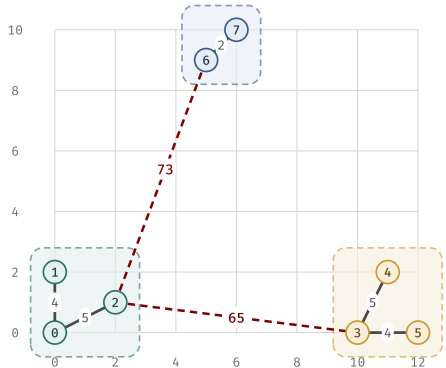
Árbol de expansión mínima, de la arista más barata a la más cara:
  6-7:2 0-1:4 3-5:4 0-2:5 3-4:5 2-3:65 2-6:73

k = 2: {0 1 2 3 4 5} {6 7}
  separación 73, fuerza bruta 73 entre 127 particiones
k = 3: {0 1 2} {3 4 5} {6 7}
  separación 65, fuerza bruta 65 entre 966 particiones
k = 4: {0 1 2} {3 5} {4} {6 7}
  separación 5, fuerza bruta 5 entre 1701 particiones

Idea que falla, k = 3: quitar las 2 aristas más caras del grafo completo
  quedan 1 grupo
```

Los pesos son distancias al cuadrado, que ordenan igual que la distancia. La fuerza bruta prueba todas las particiones.

Con $k = 3$ se quitan las dos más caras



Árbol mínimo, de la más barata a la más cara

6-7	2
0-1	4
3-5	4
0-2	5
3-4	5
2-3	65
2-6	73

k = 3: se quitan las k − 1 = 2 más caras

Quedan tres grupos

{0 1 2} {3 4 5} {6 7}

Separación 65: la menor de las quitadas.

La fuerza bruta da lo mismo: 65, entre 966 particiones.

La demostración

Sea T un árbol mínimo, C la partición que queda, y d^* el menor peso entre las quitadas. Lo que se queda pesa a lo sumo d^*.

Paso 1: la separación de C es exactamente d^*.

- La quitada de peso d^* une dos grupos distintos: es a lo sumo d^*.
- Si g une grupos distintos y $w(g) < d^*$, el camino de T entre sus extremos usa una quitada f, y $T - f + g$ es un árbol más barato. Contradicción.

Paso 2: otra partición C' tiene separación a lo sumo d^*.

- Si no son la misma, hay a, b juntos en C y separados en C'.
- En C hay un camino de a a b con aristas de T que se quedaron: pesan a lo sumo d^*.
- Alguna de ellas une dos grupos de C'. Listo. ■

No hizo falta suponer pesos distintos. Vale para cualquier árbol mínimo y cualquier elección entre empates.

SECCIÓN 07

DFS con tiempos

07

DFS con un reloj

pseudocódigo

```
algoritmo visitar(u)
    reloj ← reloj + 1
    inicio[u] ← reloj
    para cada vecino v de u hacer
        si inicio[v] = 0 entonces
            visitar(v)
        fin si
    fin para
    reloj ← reloj + 1
    fin[u] ← reloj
fin algoritmo
```

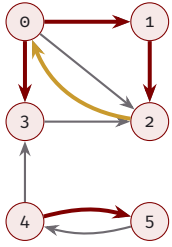
Grafos dirigidos. DFS **completo**: si queda un nodo sin visitar, se sigue desde él. Costo $\Theta(V + E)$.

Estado	Condición
Sin visitar	sin `inicio`
Abierto	con `inicio`, sin `fin`
Cerrado	con `fin`

Los abiertos son la pila de llamadas: **forman un camino** hasta el nodo actual.

Dos intervalos $[\text{inicio}, \text{fin}]$ son disjuntos, o uno está dentro del otro. Las llamadas abren y cierran como paréntesis.

Un ejemplo, con el reloj



5 termina en 11, y 4 en 12.

```
(0 (1 (2 2) 1) (3 3) 0) (4 (5 5) 4)
```

nodo	0	1	2	3	4	5
inicio	1	2	3	6	9	10
fin	8	5	4	7	12	11

El lema del camino libre

Lema

Si al empezar u hay un camino de u a x con todos sus nodos sin visitar, salvo u, entonces x empieza y termina dentro del intervalo de u.

Técnica. Inducción sobre la posición en el camino $u = x_0, x_1, \ldots, x_k = x$.

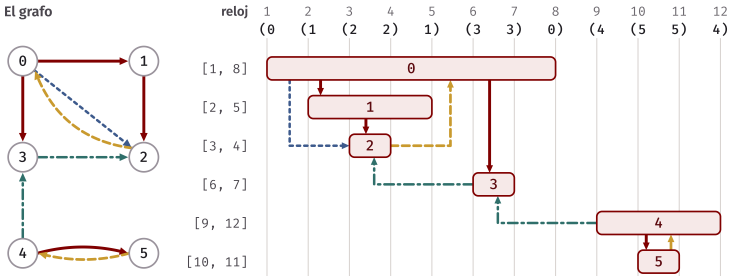
- x_0 empieza durante la llamada de u. Supongamos que x_i también.
- Cuando se visita x_i, se mira su arista a x_{i+1}.
- Si x_{i+1} está sin visitar, se visita ahí mismo.
- Si ya está visitado, empezó antes de este momento, y después de u, porque estaba sin visitar cuando u empezó.
- En los dos casos, x_{i+1} empieza durante la llamada de u. ■

Tipos de arista: el estado de v al mirar $u \to v$

Tipo	**Estado de v**	**Con tiempos**
De árbol *(tree)*	sin visitar	$\text{ini}_u < \text{ini}_v < \text{fin}_v < \text{fin}_u$
Hacia atrás *(back)*	abierto	$\text{ini}_v \leq \text{ini}_u < \text{fin}_u \leq \text{fin}_v$
Hacia adelante *(forward)*	cerrado, empezó después que u	$\text{ini}_u < \text{ini}_v < \text{fin}_v < \text{fin}_u$
De cruce *(cross)*	cerrado, empezó antes que u	$\text{ini}_v < \text{fin}_v < \text{ini}_u < \text{fin}_u$

- Un **lazo** $u \to u$ es hacia atrás: al mirarlo, u está abierto.
- En el ejemplo, las hacia atrás son $2 \to 0$ y $5 \to 4$, y cada una cierra un ciclo.
- En un grafo no dirigido solo aparecen de árbol y hacia atrás.

Los tipos de arista, sobre el reloj



En una arista u → v:

- De árbol: v estaba sin visitar, y su intervalo queda dentro del de u.
- Hacia atrás: v está abierto. Su intervalo contiene al de u.
- Hacia adelante: v ya cerró. Su intervalo está dentro del de u, como en una de árbol.
- De cruce: v ya cerró. Su intervalo es anterior y disjunto.

Arista hacia atrás si y solo si hay ciclo

Teorema

Un grafo dirigido tiene un ciclo si y solo si un DFS completo encuentra una arista hacia atrás. Con cualquier orden de nodos y de vecinos.

(Arista $\Rightarrow$ ciclo) Sea $u \to v$ hacia atrás. v está abierto, y los abiertos forman un camino que termina en u. Ese camino de v a u, más $u \to v$, es un ciclo. Si $u = v$, es un lazo.

(Ciclo $\Rightarrow$ arista) Sea C un ciclo.

- Sea x el primero de C que empieza, e y su antecesor en C: $y \to x$ está en C.
- Al empezar x, los demás de C están sin visitar: hay un camino libre de x a y.
- Por el lema, y empieza y termina dentro del intervalo de x.
- Al visitar y se mira $y \to x$, con x abierto. Es hacia atrás. ■

SECCIÓN 08

Orden topológico

08

Lo que termina último va primero

Orden topológico *(topological order)*

Un orden de los nodos en el que toda arista $u \to v$ va de un nodo a otro posterior. Existe solo si no hay ciclos: en un **DAG**.

Algoritmo. DFS completo, con cada nodo agregado a una lista al terminar, y esa lista al revés.

Afirmación. En un DAG, si hay arista $u \to v$, entonces $\mathrm{fin}[v] < \mathrm{fin}[u]$. Casos, al mirarla desde u:

- v **sin visitar:** se visita dentro de la llamada de u y termina antes.
- v **abierto:** sería hacia atrás, y habría un ciclo. En un DAG no pasa.
- v **cerrado:** ya terminó, y u todavía no. ■

Un ejemplo, y qué pasa con un ciclo

terminal

```
$ build/dfs_traces
DAG: 0->1 0->2 1->3 2->3 2->4 3->5 4->5
  nodo  inicio  fin
     0       1   12
     1       2    7
     2       8   11
     3       3    6
     4       9   10
     5       4    5
  terminan en este orden: 5 3 1 4 2 0
  orden decreciente de fin: 0 2 4 1 3 5  es topológico

Con un ciclo: 0->1 1->2 2->0 2->3
  orden decreciente de fin: 0 1 2 3  no es topológico
```

Error común

La lista sale siempre, sin aviso. Si el problema no garantiza un DAG, hay que buscar aristas hacia atrás o revisar la lista.

SECCIÓN 09

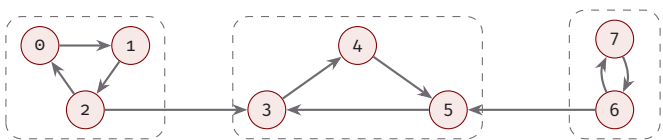
Componentes fuertemente conexas

09

Todo grafo dirigido es un DAG de componentes

Componente fuertemente conexa *(strongly connected component)*

Una clase de nodos mutuamente alcanzables: hay camino de ida y de vuelta entre cada par.



El grafo de componentes es un DAG. Si tuviera un ciclo por C y D, todos sus nodos serían mutuamente alcanzables, y C y D serían una sola.

Un DAG tiene una **fuente** y un **sumidero** *(sink)*. Aquí, $\{3, 4, 5\}$ es el sumidero.

La idea: empezar en un sumidero

- Un recorrido desde un nodo de una componente **sumidero** alcanza exactamente su componente: de ahí no se sale.
- Se marca, se hace como si no estuviera, y ahora hay otro sumidero entre lo que queda.
- **Problema:** ¿cómo se consigue un nodo de un sumidero, sin conocer las componentes?

Grafo invertido $\mathrm{rev}(G)$

Todas las aristas al revés. Tiene las **mismas componentes** que G, y lo que en G es sumidero, en $\mathrm{rev}(G)$ es fuente.

El mayor fin de un DFS cae en una fuente del grafo recorrido. Sobre $\mathrm{rev}(G)$, es un sumidero de G.

Kosaraju-Sharir

pseudocódigo

```
algoritmo componentes(grafo, n)
    dfs_completo(rev(grafo), n), anotando el fin
    comp ← arreglo de n valores -1
    actual ← 0
    para cada r, en orden decreciente de fin, hacer
        si comp[r] = -1 entonces
            recorrer desde r, por nodos sin grupo,
                poniéndoles comp ← actual
            actual ← actual + 1
        fin si
    fin para
    devolver comp
fin algoritmo
```

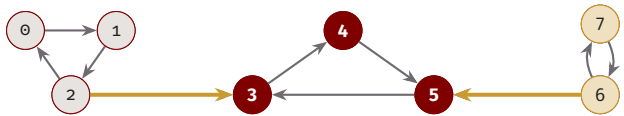
Primero el **invertido**. Costo: $\Theta(V + E)$.

terminal

```
$ build/scc_traces
Paso 1, DFS sobre el grafo invertido:
  nodo  inicio  fin
     0       1    6
     1       3    4
     2       2    5
     3       7   16
     4       9   10
     5       8   15
     6      11   14
     7      12   13
  terminan: 1 2 0 4 7 6 5 3
```

Orden decreciente de fin: 3 5 6 7 4 0 2 1.

Paso 2: recorrer el original con 3 5 6 7 4 0 2 1



Resultado: `{3 4 5} {6 7} {0 1 2}`. Salen de sumidero a fuente: leídas al revés, están en orden topológico.

El lema: siguiendo una arista de G, el fin aumenta

Lema

$\mathrm{fin}(C)$ es el mayor fin de C en el DFS sobre $\mathrm{rev}(G)$. Si G tiene una arista de C a D, con $C \neq D$, entonces $\mathrm{fin}(D) > \mathrm{fin}(C)$.

En $\mathrm{rev}(G)$ la arista va de D a C. Sea x el primero de $C \cup D$ que empieza.

Caso 1: $x \in D$. Desde x se llega a todo D y, por esa arista, a todo C, por nodos sin visitar. Por el lema del camino libre, todos terminan antes que x: $\mathrm{fin}(D) > \mathrm{fin}(C)$.

Caso 2: $x \in C$. Desde x se llega a todo C por nodos sin visitar: $\mathrm{fin}(C) = \mathrm{fin}[x]$.

- Desde x no se llega a D: si se pudiera, C y D serían una sola componente.
- Lo que empieza dentro del intervalo de x es alcanzable desde x. Ningún nodo de D empieza ahí, ni antes.
- Todo D empieza después de que x termina: $\mathrm{fin}(D) > \mathrm{fin}[x] = \mathrm{fin}(C)$. ■

Cada recorrido marca exactamente una componente

Técnica. Inducción. Los primeros t recorridos marcaron t componentes completas. Sea r el siguiente sin grupo, y C su componente.

- Lo marcado es una unión de componentes, y r no está: **nada de C está marcado**.
- r tiene el mayor fin entre los no marcados. Entonces $\mathrm{fin}(C) = \mathrm{fin}[r]$, y cualquier otra componente sin marcar tiene fin menor.
- **Alcanza todo C:** desde r se llega a todo C dentro de C.
- **No sale de C:** sea una arista de C a otra componente D. Por el lema, $\mathrm{fin}(D) > \mathrm{fin}(C)$. Luego D ya está marcada. ■

Gratis

El Paso 2 no necesita un DFS: el argumento solo usa qué se alcanza. Y las componentes salen de sumidero a fuente.

Error común: no invertir

terminal

```
Sin invertir:
  orden decreciente de fin en el original: 6 7 0 1 2 3 4 5
  {3 4 5 6 7} {0 1 2}
  no son las componentes
```

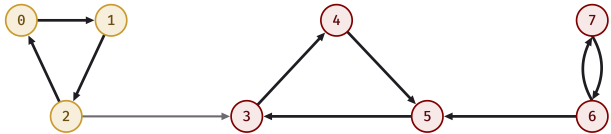
- El mayor fin en G cae en una **fuente** de G: aquí, el 6.
- Un recorrido desde una fuente se lleva todo lo que alcanza: 6 y 7, con 3, 4 y 5.
- No hay ningún aviso. Salen grupos que parecen razonables.

La versión de CLRS

El orden sale del original, y el recorrido es sobre el invertido. Lo que nunca funciona es usar el mismo grafo dos veces.

Sin invertir, el primer recorrido se lleva dos componentes

Sin invertir: el mismo grafo G en los dos pasos



Orden decreciente de fin en G	6 7 0 1 2 3 4 5
Invirtiendo	{3 4 5} {6 7} {0 1 2}
Sin invertir	{3 4 5 6 7} {0 1 2}

r = 7: ya tiene grupo. r = 0: 0, 1 y 2. Los demás ya tienen grupo. No son las componentes.

SECCIÓN 10

Recursión profunda

10



La pila no alcanza

terminal

```
$ bash -c 'build/deep_recursion 100000; echo "código de salida: $?"' 2>&1
con pila explícita: terminaron 100000 nodos
con recursión: terminaron 100000 nodos
código de salida: 0
$ bash -c 'build/deep_recursion 1000000; echo "código de salida: $?"' 2>&1
con pila explícita: terminaron 1000000 nodos
bash: line 1: 69076 Segmentation fault      build/deep_recursion 1000000
código de salida: 139
```

- Un camino de *n* nodos, compilado con `-O2`. `ulimit -s` da 8192: son 8 MB de pila.
- Cada nodo del camino actual gasta un marco de llamada. El juez lo reporta como `RE`.
- La solución: guardar en una pila propia lo mismo que guarda un marco.

Una pila que conoce los tiempos de fin

digraph.h

```
while (!stack.empty()) {  // pares (nodo, posición)
    auto [u, next] = stack.back();
    if (next == g[u].size()) {
        t.finish[u] = ++clock;
        t.finish_order.push_back(u);
        stack.pop_back();
        continue;
    }
    stack.back().second++;
    int v = g[u][next];
    if (t.start[v] == 0) {
        t.start[v] = ++clock;
        stack.push_back({v, 0});
    }
}
```

- Cada elemento: **qué nodo, y por cuál arista iba**.
- Un nodo termina cuando se le acaban las aristas.
- `auto [u, next]` copia, porque `push_back` puede mover el `std::vector`.
- Da los mismos tiempos que la recursiva: probado en 1500 grafos al azar.

Error común: el atajo de meter todos los vecinos

terminal

```
El atajo de la pila, en 0->1 0->2 1->2:
  pila con (nodo, siguiente), terminan: 2 1 0  al revés: 0 1 2  topológico
  atajo, orden al sacar: 0 2 1  no topológico
  atajo, al revés:       1 2 0  no topológico
```

- Saca el 0, mete 1 y 2, y saca el 2 antes de haber visto el 1.
- Un nodo se saca **antes** que todo lo que cuelga de él. Ese momento no es un tiempo de fin.
- Para saber qué se alcanza, sirve: se puede usar en el Paso 2. En el Paso 1, no.

Para la próxima clase

- Leer los apuntes de la semana 6, y hacer sus ejercicios.
- Laboratorio: **Diseñar una red**, en `lab/network_design`, y **Los prerrequisitos de una malla**, en `lab/curriculum`.
- La próxima semana empieza la programación dinámica, y hay **Práctica 2** en el laboratorio.

Preguntas

¿En qué paso de la propiedad del corte se usa que f esté en el camino entre u y v?
Si se suma una constante a todos los pesos, ¿cambia el árbol mínimo? ¿Y los caminos mínimos?
En un DAG, ¿por qué una arista de u a v implica que v termina antes que u?
¿Dónde se pierde la demostración de Kosaraju-Sharir si no se invierte el grafo?