

Estructuras de Datos

Grafos y union-find

Listas de adyacencia, BFS, DFS y conjuntos que se van uniendo

Gerson Garrido

Semana 14

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Grafos, y matriz o listas de adyacencia
2. BFS: por niveles, con una cola
3. Marcar al meter, no al sacar
4. DFS: componentes y ciclos
5. El costo: $\Theta(V + E)$
6. Union-find: rango y compresión, y qué afirma su costo
7. Union-find frente a DFS
8. Laberintos: el laboratorio

> **Lectura**
>
> Apuntes de la semana 14. Open Data Structures, secs. 12.1, 12.2 y 12.3. CLRS, *Elementary Graph Algorithms* y *Data Structures for Disjoint Sets*. Código en `code/week-14`.

SECCIÓN 01

Grafos

01



Grafos en todas partes

Problema	Nodos	Aristas	¿Dirección? ¿Pesos?
Rutas en un mapa	cruces	tramos de calle	a veces; metros o minutos
Enlaces de la web	páginas	enlaces	sí; no
Dependencias de un Makefile	archivos	“se construye con”	sí; no

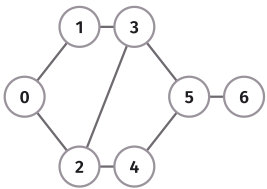
Grafo *(graph)*

Un conjunto V de **nodos** *(nodes)* y un conjunto E de **aristas** *(edges)*, cada una entre dos nodos. V y E también son sus cantidades: $\Theta(V + E)$.

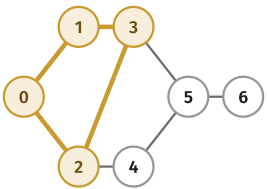
Problemas distintos, misma estructura: un algoritmo de hoy sirve para los tres.

Vocabulario

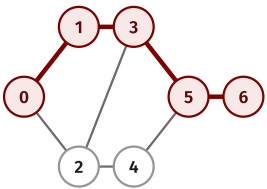
7 nodos, 8 aristas



un ciclo: 0-1-3-2-0



un camino: 0-1-3-5-6, de largo 4



- **Dirigido:** u $\rightarrow$ v solo va de u a v. **No dirigido:** va en los dos sentidos.
- **Grado:** cuántas aristas tocan a un nodo.
- **Camino:** sin repetir nodos, cada uno vecino del anterior. Su **largo** son sus aristas.
- **Ciclo:** vuelve al primer nodo, sin repetir aristas.
- **Componente conexa:** grupo máximo donde cada par tiene un camino.
- Un **árbol** es un grafo conexo sin ciclos.

SECCIÓN 02

Dos representaciones

02

Matriz de adyacencia

	0	1	2	3	4	5	6
0	.	1	1	.	.	.	.
1	1	.	.	1	.	.	.
2	1	.	.	1	1	.	.
3	.	1	1	.	.	1	.
4	.	.	1	.	.	1	.
5	.	.	.	1	1	.	1
6	.	.	.	.	.	1	.

Matriz de adyacencia *(adjacency matrix)*

Tabla de $V \times V$: la celda [u][v] dice si existe la arista.

- `has_edge`: una lectura. $\Theta(1)$.
- Memoria: $\Theta(V^2)$, siempre.
- Vecinos de u: leer toda su fila, $\Theta(V)$, aunque tenga uno solo.

Listas de adyacencia

listas

```
0: 1 2
1: 0 3
2: 0 3 4
3: 1 2 5
4: 2 5
5: 3 4 6
6: 5
```

graph.h

```cpp
void add_edge(int u, int v, int weight = 1) {
    adjacency_[u].push_back({v, weight});
    if (!directed_) {
        adjacency_[v].push_back({u, weight});
    }
    edge_count_++;
}
```

- Cada arista aparece **dos veces**: 2E entradas. Aquí, 16 para 8 aristas.
- Cada entrada es un `Edge`, con destino y peso. `std::vector` es owner de todo.
- La semana 15 reutiliza esta clase, con metros en `weight`.

El mismo grafo, las dos veces

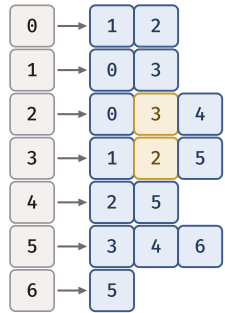
MATRIZ DE ADYACENCIA

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	0	1	0	0	0
2	1	0	0	1	1	0	0
3	0	1	1	0	0	1	0
4	0	0	1	0	0	1	0
5	0	0	0	1	1	0	1
6	0	0	0	0	0	1	0

7 × 7 = 49 celdas, con o sin aristas

la celda [2][3] es la posición 2 · 7 + 3 = 17

LISTAS DE ADYACENCIA



16 entradas: la suma de los grados, 2E

la arista 2-3 sale dos veces en cada una

Qué cuesta cada una

	Matriz	**Listas**
Memoria	$\Theta(V^2)$	$\Theta(V + E)$
¿Existe u-v?	$\Theta(1)$	Θ(grado de u), peor caso
Recorrer vecinos de u	$\Theta(V)$	Θ(grado de u)
Recorrer todas las aristas	$\Theta(V^2)$	$\Theta(V + E)$
Agregar una arista	$\Theta(1)$	$O(1)$ amortizado

- **Disperso** *(sparse)*: muchas menos aristas que V^2. **Denso** *(dense)*: cerca de V^2. No hay frontera exacta.
- Casi todos los grafos reales son dispersos. **Se elige por la operación que más se hace:** si casi todo es recorrer vecinos, listas.

1000 nodos y 3000 aristas

A mano:

- **Matriz:** 1 000 000 de celdas. Un byte por celda: 1 MB.
- **Listas:** 6000 entradas de 8 bytes, más una cabecera por nodo: 72 000 bytes.
- Medido: 87 456. Hay capacidad sobrante, como en la semana 4.
- Unas 11 veces menos, pero 6 lecturas por `has_edge`.

terminal

```
$ build/representations
nodos: 1000, aristas: 3000
sizeof(std::vector<Edge>): 24, sizeof(Edge): 8
matriz, bytes: 1000032
listas, entradas: 6000, capacidad reservada: 7927
listas, bytes: 87456
lista mas larga: 16
preguntas has_edge: 1000000, respuestas si: 6000
lecturas por pregunta, matriz: 1.00, listas: 5.98199
```

Sin contar lo que agrega el administrador de memoria. `sizeof` = 24 es de libstdc++, no del estándar.

Un grafo real: las calles de Miraflores

terminal

```
$ build/map_tour ../../data/mapas/miraflores.txt 1839 1883
nodos: 2242, aristas: 3474
grado medio: 3.09902, grado maximo: 6
aristas sobre las posibles: 0.138287 %
matriz de un byte por celda: 5026564 bytes
listas: 118256 bytes
componentes: 1
```

- Un cruce toca unas 3 calles. Existe el **0.14 %** de las aristas posibles.
- La matriz ocuparía 42 veces más. En `lima.txt`, con 79 050 nodos, más de 6 GB.
- Es una sola componente: el archivo guarda solo la mayor de la zona.

Datos: © colaboradores de OpenStreetMap, bajo ODbL. Ver `data/README.md`.

SECCIÓN 03

BFS

03



BFS: por niveles, con una cola

pseudocódigo

```
algoritmo bfs(grafo, origen)
    distancia[v] ← -1 para todo nodo v
    distancia[origen] ← 0
    padre[origen] ← origen
    meter origen a la cola
    mientras la cola no esté vacía hacer
        u ← sacar de la cola
        para cada vecino v de u hacer
            si distancia[v] = -1 entonces
                distancia[v] ← distancia[u] + 1
                padre[v] ← u
                meter v a la cola
            fin si
        fin para
    fin mientras
fin algoritmo
```

- **BFS** *(breadth-first search)*: el origen, luego sus vecinos, luego los vecinos de esos… Como una onda.
- La **cola** atiende por orden de llegada.
- **Marcado:** ya sabemos que lo atenderemos. Aquí, `distancia[v]` $\neq -1$.
- Se marca al meter a la cola.

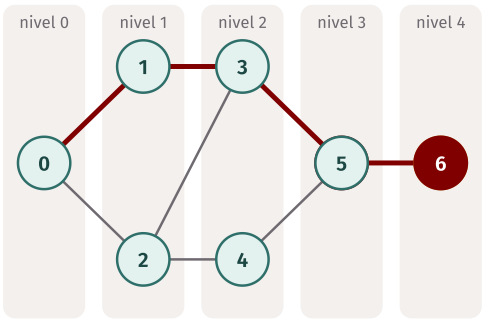
En C++

traversals.h

```
std::queue<int> pending;
result.distance[source] = 0;
result.parent[source] = source;
pending.push(source);
while (!pending.empty()) {
    int u = pending.front();
    pending.pop();
    result.order.push_back(u);
    for (const Edge& edge : graph.neighbors(u)) {
        if (result.distance[edge.to] == -1) {  // se marca al meterlo a la cola
            result.distance[edge.to] = result.distance[u] + 1;
            result.parent[edge.to] = u;
            pending.push(edge.to);
        }
    }
}
```

`std::queue` usa por debajo un `std::deque`, el de la semana 5.

BFS, paso a paso



SALE

6

COLA

sale por la izquierda, entra por la derecha

DISTANCIA

0	1	2	3	4	5	6
0	1	1	2	2	3	4

PADRE

0	1	2	3	4	5	6
0	0	0	1	2	3	5

acaba de salir · marcado, en la cola · atendido · sin marca

Sale 6 y la cola queda vacía. Subiendo por los padres: 0 1 3 5 6, 4 aristas

Lo mismo, corriendo

terminal

```
$ build/bfs_trace
sale  marca     cola al terminar el paso
   -  0         0
   0  1 2       1 2
   1  3         2 3
   2  4         3 4
   3  5         4 5
   4            5
   5  6         6
   6
distancia: 0 1 1 2 2 3 4
padre: 0 0 0 1 2 3 5
camino a 6: 0 1 3 5 6
```

La cola tiene como máximo dos niveles seguidos. **El camino sale de los padres**, al revés.

Por qué da el camino con menos aristas

1. **Nunca es menor que la verdadera.** Siguiendo los padres desde v hay un camino real con `distancia[v]` aristas, y ningún camino real es más corto que el mínimo.
2. **Las distancias salen en orden.** La cola atiende por llegada, y cada nodo entra con la distancia de quien lo marcó más 1. Así, si un vecino v de u ya estaba marcado al salir u, tiene distancia $\leq$ `distancia[u]` + 1.
3. **Nunca es mayor.** Recorran un camino más corto: origen, v_1, v_2… t. v_1 queda con ≤ 1, v_2 con ≤ 2… y t con $\leq$ su largo.

> **Idea clave**
>
> El punto 2 es de la cola: una pila no lo da. Y es **menos aristas**, no menos metros. Para metros: Dijkstra, la próxima semana.

Reconstruir el camino

traversals.h

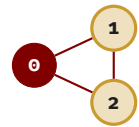
```cpp
std::vector<int> path;
if (parent[target] == -1) {
    return path;
}
int node = target;
while (parent[node] != node) {
    path.push_back(node);
    node = parent[node];
}
path.push_back(node);
std::reverse(path.begin(), path.end());
```

- BFS no guarda caminos: sería demasiada memoria. Guarda **un entero por nodo**: de dónde se llegó.
- Esos padres forman el **árbol de BFS**.
- Convenio: -1 si no se llega, y el origen es su propio padre.

Error común: marcar al sacar

errors/bfs_mark_on_pop.cpp

```cpp
int u = pending.front();
pending.pop();
marked[u] = true;  // demasiado tarde
for (const Edge& edge : graph.neighbors(u)) {
    if (!marked[edge.to]) {
        distance[edge.to] = distance[u] + 1;
        pending.push(edge.to);
```



- Sale el 0: entran 1 y 2, con distancia 1.
- Sale el 1: el 2 no está marcado. Su distancia se sobrescribe con 2.

Idea clave

La marca dice “alguien ya se encargó de este nodo”. Se pone al meterlo.

Qué se rompe, medido

terminal

```
$ build/bfs_mark_on_pop
triangulo, distancia de 0 a 2 (la verdadera es 1):
  marcando al sacar: 2
  e ignorando los ya marcados: 2
rombos  nodos  aristas  entradas a la cola  ignorando marcados
     2      7        8                  13                   9
     4     13       16                  61                  17
     8     25       32                1021                  33
    16     49       64              262141                  65
```

- **La cola explota:** en estas filas, $2^{k+2} - 3$ con k rombos. Cada cruce entra el doble que el anterior.
- Ignorar al sacar evita la explosión, con 2E + 1 entradas como máximo, pero **la distancia sigue mal**.
- Sin ninguna marca, en un ciclo la búsqueda ni termina: `errors/no_marks.cpp`.

SECCIÓN 04

DFS

04

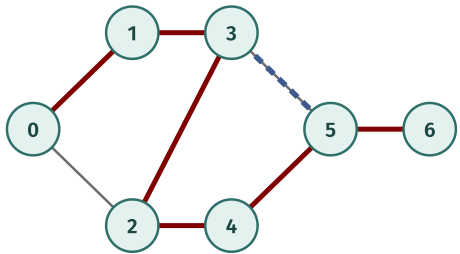
DFS: bajar todo lo posible

traversals.h

```
1  marked[u] = true;
2  order.push_back(u);
3  for (const Edge& edge : graph.neighbors(u)) {
4      if (!marked[edge.to]) {
5          dfs_visit(graph, edge.to, marked, order);
6      }
7  }
```

- **DFS** *(depth-first search)*: al primer vecino sin marcar, y de ahí al suyo…
- Sin vecinos nuevos, **retrocede**.
- El nodo anterior es el marco de llamada anterior.

DFS, paso a paso



DFS 0 1 3 2 4 5 6 6 aristas

BFS 0 1 3 5 6 4 aristas

Los dos empiezan con 0-1-3 y terminan con 5-6.
El tramo 3-5, punteado, es solo de BFS.

estoy aquí · en la pila · terminado · sin marca

DFS no intenta acercarse: da un camino de 6 aristas hasta el 6, y BFS encontró uno de 4

Por qué DFS no da el camino más corto

terminal

```
$ build/map_tour ../../data/mapas/miraflores.txt 1839 1883
de 1839 a 1883:
BFS: 18 aristas, 1871 metros, nodos marcados: 492
DFS: 393 aristas, 32805 metros, nodos marcados: 543
```

- DFS marca un nodo apenas lo ve, por el camino por el que iba. Si después hay algo más directo, ya está marcado.
- En el grafo de la clase llega al 2 **desde el 3**, con 3 aristas, aunque es vecino del origen.
- A veces da el más corto: en un árbol no hay otro. Lo que no da es la garantía.
- Ese 393 depende del orden del archivo. Lo que no depende es que DFS no intenta acercarse.

La recursión puede agotar la pila

terminal

```
$ build/recursive_dfs_overflow 1000000
con pila propia: 1000000 nodos
AddressSanitizer:DEADLYSIGNAL
=================================================================
==42104==ERROR: AddressSanitizer: stack-overflow on address 0xfffffc4f3ff0 (pc 0xaaaadee1f794 bp 0xfffffc4f4040 sp 0xfffffc4f4040 T0)
...
```

- Un marco de llamada por cada nodo del camino actual. Lo mismo que el destructor recursivo de la semana 10.
- La primera línea es de la versión con pila propia, que sí terminó.

Con una pila de marcos

traversals.h

```
while (!stack.empty()) {
    auto& [u, next] = stack.back();
    const std::vector<Edge>& list = graph.neighbors(u);
    if (next == list.size()) {
        stack.pop_back();
        continue;
    }
    int v = list[next].to;
    next++;
    if (!marked[v]) {
        marked[v] = true;
        order.push_back(v);
        stack.push_back({v, 0});  // invalida u y next,
  que no se vuelven a usar en esta vuelta
    }
}
```

- Se guarda lo que guarda un marco: **qué nodo, y por qué posición de su lista iba**.
- Mismo orden que la recursiva. Probado en 200 grafos al azar.
- La pila, en todo momento, es un camino desde el origen.

Error común: la cola de BFS con una pila

terminal

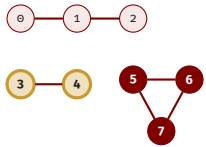
```
$ build/dfs_demo
BFS desde 0:              0 1 2 3 4 5 6
DFS recursivo desde 0:    0 1 3 2 4 5 6
DFS con pila de marcos:   0 1 3 2 4 5 6
cola de BFS con una pila: 0 2 4 5 6 3 1
```

- Cambiar la cola por una pila visita los mismos nodos, en otro orden.
- Basta para saber qué se alcanza. Pero un nodo puede estar varias veces: la pila llega a $\Theta(E)$ entradas.
- Y esa pila **no es un camino**. Eso importa dentro de un momento.

Componentes conexas

traversals.h

```
for (int start = 0; start < graph.node_count();
     start++) {
    if (label[start] != -1) {
        continue;
    }
...
    while (!stack.empty()) {
        int u = stack.back();
        stack.pop_back();
        for (const Edge& edge : graph.neighbors(u))
        {
            if (label[edge.to] == -1) {
                label[edge.to] = count;
                stack.push_back(edge.to);
```



- Una búsqueda marca una componente y nada más.
- Etiquetas: `0 0 0 1 1 2 2 2`.
- Después, “¿conectados?” es comparar dos etiquetas, en $\Theta(1)$. Recuerden esto.

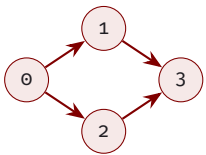
Ciclos en un grafo no dirigido

traversals.h

```
marked[u] = true;
for (const Edge& edge : graph.neighbors(u)) {
    if (!marked[edge.to]) {
        if (undirected_cycle_from(graph, edge.to, u, marked)) {
            return true;
        }
    } else if (edge.to != parent) {
        return true;  // ya marcado y no es de donde vengo: hay otro camino hasta él
    }
}
return false;
```

- La excepción del padre: cada arista está en las dos listas, y desde v siempre se ve de vuelta a su padre.
- Supone que no hay aristas repetidas. Los mapas del curso no las tienen: lo revisamos.

En un grafo dirigido, esa revisión miente



terminal

```
$ build/directed_false_cycle
revision no dirigida: hay ciclo
revision con tres colores: sin ciclo
```

- Un nodo marcado no significa ciclo: dos caminos pueden llegar al mismo sitio.
- Es de lo más normal: dos archivos que usan el mismo encabezado. De 3 no se vuelve.
- Lo que importa es si el nodo está **en el camino actual**.

Tres colores

traversals.h

```
color[u] = Color::Gray;
for (const Edge& edge : graph.neighbors(u)) {
    if (color[edge.to] == Color::Gray) {
        return true;  // volvió a un nodo del camino actual
    }
    if (color[edge.to] == Color::White && directed_cycle_from(graph, edge.to, color))
      {
        return true;
    }
}
color[u] = Color::Black;
```

- **Blanco:** sin ver. **Gris:** en la pila. **Negro:** terminado.
- Por eso no sirve la pila simple: esa pila no es un camino.
- Es la dependencia circular de la que se queja `make`.

Qué responde cada uno

Pregunta	BFS	DFS
¿Qué se alcanza desde aquí?	sí	sí
Componentes conexas	sí	sí
Camino con menos aristas	**sí**	no lo garantiza
Ciclo en grafo no dirigido	también se puede	sí
Ciclo en grafo dirigido	no de esta forma	**sí**, con tres colores
Memoria auxiliar, peor caso	$\Theta(V)$: la cola	$\Theta(V)$: la pila

La última fila tiene trampa. En un camino largo, DFS guarda muchos nodos y BFS casi ninguno. En una estrella, al revés. Ninguno gana siempre.

SECCIÓN 05

El costo

05

$\Theta(V + E)$

Crear los arreglos cuesta $\Theta(V)$. Después:

- Cada nodo alcanzado **entra a la cola una sola vez**, porque se marca al entrar.
- Al salir, se lee su lista completa. Cada lista, una vez: la suma de los grados.
- En uno no dirigido y conexo, esa suma es 2E. Total: $\Theta(V + E)$.

Con precisión

Si el grafo no es conexo, una búsqueda desde un nodo solo paga las aristas de su componente, más el $\Theta(V)$ de los arreglos. Recorriendo todas las componentes, sí es $\Theta(V + E)$ siempre.

Con una matriz, el mismo BFS cuesta $\Theta(V^2)$ si alcanza a todos, por disperso que sea el grafo.

En los tres mapas

terminal

```
nodos   aristas  sacados  entradas leidas   ms lista  ns/(V+E)
     2242      3474     2242            6948      0.013       2.3
    19560     29117    19560           58234      0.290       6.0
    79050    116803    79050          233606      1.841       9.4
matriz en el primer mapa: 5026564 celdas leidas, 3.718 ms
listas en el primer mapa: 0.013 ms
```

Mediana de 11 corridas, con `-O2` y sin sanitizers. Los tiempos dependen de la máquina.

- **Exacto:** `sacados` es V y `entradas leidas` es 2E, en los tres.
- **No se mantiene:** el tiempo por unidad subió unas 4 veces en las 3 corridas. $\Theta(V + E)$ cuenta trabajo, no cuánto tarda cada lectura.
- Probablemente es la cache. Sin `perf` no se puede confirmar: queda como explicación probable.
- La matriz lee 723 veces más, y tardó unas 280 veces más.

SECCIÓN 06

Union-find

06



Las aristas van llegando

¿Están conectados a y b?, mientras se instalan cables, o se tiran paredes.

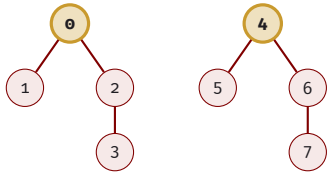
- Buscar en cada pregunta: $\Theta(V + E)$ cada una, en el peor caso.
- Etiquetas: preguntar es $\Theta(1)$, unir recorre todo, $\Theta(n)$.

Union-find ***(disjoint sets)***

Una partición de n elementos. `find(x)`: el representante del conjunto de x. `unite(a, b)`: une los dos conjuntos. `connected(a, b)`: ¿mismo representante?

No hay: separar un conjunto, ni dar un camino. Hace poco, y por eso lo hace muy bien.

Un bosque en un arreglo



parent: 0 0 0 2 4 4 4 6

Cada nodo solo conoce a su padre. La raíz es su propio padre, y es el representante. Cuesta la altura: uniendo 0-1, 1-2, 2-3… sale una cadena de altura $n - 1$.

pseudocódigo

```
algoritmo find(x)
    mientras parent[x] ≠ x hacer
        x ← parent[x]
    fin mientras
    devolver x
fin algoritmo

algoritmo unite(a, b)
    raiz_a ← find(a)
    raiz_b ← find(b)
    si raiz_a = raiz_b entonces
        devolver falso
    fin si
    parent[raiz_a] ← raiz_b
    devolver verdadero
fin algoritmo
```

Error común: enlazar sin buscar la raíz

terminal

```
$ build/unite_without_find
0 y 2: juntos
0 y 1: separados
```

errors/unite_without_find.cpp

```cpp
parent[0] = 1;  // unir 0 con 1
parent[0] = 2;  // unir 0 con 2: debió enlazar las raíces, y 0 ya no es raíz
```

- Si a ya no era raíz, se corta de su árbol.
- Su conjunto se parte en dos, sin que nadie lo pida.
- No se cae nada: solo da una respuesta equivocada.

Primera mejora: unión por rango

pseudocódigo

```
si rango[raiz_a] < rango[raiz_b] entonces
        parent[raiz_a] ⟵ raiz_b
    sino si rango[raiz_b] < rango[raiz_a] entonces
        parent[raiz_b] ⟵ raiz_a
    sino
        parent[raiz_b] ⟵ raiz_a
        rango[raiz_a] ⟵ rango[raiz_a] + 1
    fin si
```

- Colgar el más bajo del más alto: la altura no cambia, salvo en un empate, y ahí sube 1.
- **Rango** *(rank)*: una cota de la altura, guardada en cada raíz.

¿Cuánto vale la altura?

- Una raíz solo llega a rango r si se unen dos árboles de rango $r - 1$.
- Cada uno de esos subió uniendo dos de rango $r - 2$. Dibujadas, esas uniones son un árbol binario perfecto de altura r, con hojas distintas.
- Un árbol de rango r tiene al menos $\mathbf{2^r}$ **nodos**. Con n elementos, $\mathbf{r \leq \log_2 n}$.

Consecuencia

Sin comprimir, el rango es la altura. Entonces `find` hace como máximo $\lfloor \log_2 n \rfloor$ saltos, en el peor caso. Sin esta mejora, podían ser $n - 1$.

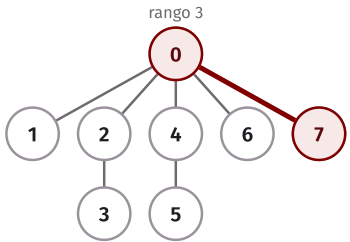
Segunda mejora: compresión de caminos

pseudocódigo

```
algoritmo find(x)
    raiz ← x
    mientras parent[raiz] ≠ raiz hacer
        raiz ← parent[raiz]
    fin mientras
    mientras x ≠ raiz hacer
        siguiente ← parent[x]
        parent[x] ← raiz
        x ← siguiente
    fin mientras
    devolver raiz
fin algoritmo
```

- Dos pasadas: la primera encuentra la raíz, y la segunda cuelga de ella cada nodo del camino.
- Quien no estaba en el camino no cambia.

Uniones y compresión, paso a paso



PARENT

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

RANGO

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

altura: 2 **rango de la raíz: 3**

Otro find(7): ahora cuesta 1 salto. La compresión paga una vez y ahorra después

Lo mismo, corriendo

terminal

```
$ build/uf_demo
rangos:  3 0 1 0 2 0 1 0
antes:   0(1,2(3),4(5,6(7)))
altura:  3
find(7) = 0, saltos: 3
despues: 0(1,2(3),4(5),6,7)
parent:  0 0 0 2 0 4 0 0
rangos:  3 0 1 0 2 0 1 0
altura:  2, rango de la raiz: 3
otro find(7), saltos: 1
```

- **Los rangos no cambiaron.** La altura bajó a 2, y el rango de la raíz sigue en 3.
- Con compresión, el rango es una cota, no la altura. Actualizarla obligaría a revisar otras ramas.
- Se mantiene: si x no es raíz, rango[x] $<$ rango[parent[x]].

Cuánto cuesta de verdad

El resultado, enunciado sin demostración

Con las dos mejoras, cualquier secuencia de **m operaciones** sobre **n elementos**, contando su creación, cuesta **O(m · α(n)) en total**, en el peor caso. α es la inversa de la función de Ackermann.

- **Casi constante:** CLRS muestra que $\alpha(n) \leq 4$ para cualquier n de una computadora real.
- **No es O(1):** α(n) crece sin tope, aunque muy lento.
- **Es amortizado, no "en promedio":** no hay azar. Vale para *toda* secuencia, y habla de la suma. Como `push_back` en la semana 4.
- **Una operación sola puede ser cara:** un `find` puede costar Θ(log n), también con las dos mejoras.

La demostración está en CLRS, *Data Structures for Disjoint Sets*. Queda fuera de este curso.

Contando saltos

terminal

```
Al azar, 4n operaciones
variante                        n  altura       saltos  saltos/op
solo rango                   1024       5         8757       2.14
rango y compresion           1024       2         5070       1.24
solo rango                1048576       8      9529179       2.27
rango y compresion        1048576       3      5210516       1.24
```

- **En una cadena sin mejoras:** $n(n-1)/2$ saltos exactos, 255.87 por operación con n = 1024. Con cualquier mejora, 1.00.
- **Al azar:** casi no cambia al multiplicar n por 1024.
- Esto no demuestra la cota de $\alpha(n)$. Al azar no es el peor caso, y es imposible ver un $\alpha(n)$ en una medición.

Un solo find puede ser caro

terminal

```
$ build/uf_costs
Un solo find en el peor arbol, con las dos mejoras
        n  log2 n  primer find  segundo find
     1024      10           10             1
    16384      14           14             1
   262144      18           18             1
  1048576      20           20             1
```

- Uniones de árboles iguales: altura $\log_2 n$. El primer `find` del más hondo cuesta $\log_2 n$.
- El segundo cuesta 1. El bajo costo es de la secuencia, no de cada operación.

SECCIÓN 07

Union-find frente a DFS

07



Las calles de Miraflores llegan una a una

terminal

```
$ build/online_connectivity ../../data/mapas/miraflores.txt
nodos: 2242, aristas que llegan: 3474, preguntas: 3474
respuestas si: 1143 (todo unido tras la arista 3417)
DFS en cada pregunta, entradas leidas: 3769130
union-find, saltos por parent: 11270
por pregunta: DFS 1084.95, union-find 3.2441
```

- **Unas 330 veces menos:** DFS repite lo que descubrió el anterior, y union-find lo guarda.
- **Con todas las aristas de antemano**, una pasada lee 6948 entradas, y comparar etiquetas es mejor: nada que subir.
- Union-find gana porque no le molesta que el grafo cambie.

Cuándo usar cada uno

	DFS o BFS	**Union-find**
¿Conectados?, aristas de antemano	una pasada, $\Theta(1)$ por pregunta	sirve, pero sobra
¿Conectados?, aristas que llegan	repetir la búsqueda	**$O(\alpha(n))$ amortizado**
Dar el camino	**sí**, con padres	no: solo sí o no
Quitar una arista	sí, y buscar otra vez	**no se puede**
Memoria	las aristas, más marcas	dos arreglos de n
Distancias	BFS, sí	no

Idea clave

Si las conexiones van llegando y solo importa si hay conexión: union-find. Si necesitan un camino, o quitar aristas: el grafo.

SECCIÓN 08

Laberintos

08

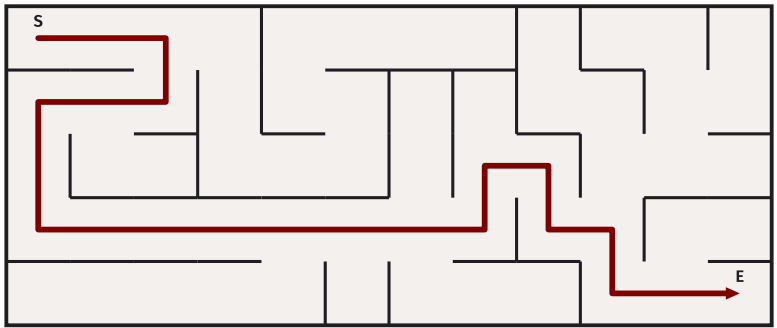
Generar un laberinto

pseudocódigo

```
algoritmo generar(laberinto)
    paredes ← todas las paredes, en orden al azar
    regiones ← un union-find con una región por celda
    para cada pared (a, b) de paredes hacer
        si regiones.unite(a, b) entonces
            abrir la pared (a, b)
        fin si
    fin para
fin algoritmo
```

- Celda: un nodo. Pared derribada: una arista.
- **Laberinto perfecto:** exactamente un camino entre dos celdas. Es un árbol.
- Si ya estaban conectadas, derribar crearía un ciclo.

Un laberinto, pared por pared



`paredes miradas: 103 de 103` `pasajes: 59` `regiones: 1`

Hay un solo camino de S a E. BFS y DFS encuentran el mismo, de 21 aristas

¿Por qué exactamente un camino?

Al menos uno

Al mirar una pared, sus dos celdas quedan en la misma región, y las regiones solo crecen. Toda pareja de vecinas termina junta, así que la cuadrícula entera es una región.

Como máximo uno

Solo se derriba entre regiones sin conexión, así que nunca se cierra un ciclo. Y dos caminos distintos contendrían un ciclo.

Se empieza con N regiones y se termina con 1: exactamente **N − 1 pasajes**.

Una consecuencia fácil de pasar por alto

terminal

```
$ build/solution/maze 8 4 1 6
BFS: camino de 10 aristas, 30 nodos visitados
+--+--+--+--+--+--+--+--+
|SS .. ..|.. .. .. ..   |
+  +--+  +  +  +  +  +  +
|** ** ..|..|.. ..|.. ..|
+--+  +  +  +--+--+  +  +
|.. ** ** ** ** ** ** **|
+  +  +--+--+  +--+--+  +
|..|.. ..|.. .. ..|   EE|
+--+--+--+--+--+--+--+--+
DFS: camino de 14 aristas, 26 nodos visitados
```

- Con un solo camino, BFS y DFS encuentran **el mismo**. Comparar su largo daría siempre empate.
- Lo que sí cambia es cuántas celdas visitó cada uno.
- Por eso `open_extra_walls` derriba algunas más: hay ciclos, y caminos distintos.
- Aquí, con 6 paredes extra: 10 aristas frente a 14.

El laboratorio

Una sola sesión: la otra es la **Práctica 4**. Por eso viene mucho hecho.

1. `UnionFind`, contra una versión ingenua. 30 min
2. `generate_maze`. 15 min
3. `maze_graph`, `bfs_search` y `dfs_search`, con pila propia. 50 min
4. Dibujar, comparar y responder `lab/report.md`. 25 min

Vienen hechos: `Graph`, `NaiveUnionFind`, `Maze`, el dibujo, `shuffle_walls` y `open_extra_walls`. Las dos búsquedas se detienen apenas marcan el destino: en BFS es seguro, y no lo será la próxima semana con Dijkstra.

terminal

```
make lab
make test-uf
make test-maze
make test-search
make test-lab
make compare
```

Para la próxima clase

- Leer los apuntes de la semana 15: caminos mínimos.
- Laboratorio: laberintos. Terminen lo que falte antes de la siguiente clase.
- Guarden su `bfs_search`: la semana 15 compara rutas de BFS con las de Dijkstra.

Pregunta

BFS encontró una ruta de 18 aristas y 1871 metros. ¿Puede haber otra con más aristas y menos metros? ¿Qué tendría que cambiar en BFS para encontrarla?