

Estructuras de Datos

Caminos mínimos y cierre

Dijkstra con su heap, rutas sobre un mapa real y cómo elegir una estructura

Gerson Garrido

Semana 15

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Con pesos, BFS ya no alcanza
2. Relajar una arista, y Dijkstra con un heap
3. Entradas viejas: ignorarlas al sacarlas
4. Dónde detenerse: al sacar, no al meter
5. Pesos negativos, con un contraejemplo
6. El costo, y el mapa de Lima
7. Opcional: tries y filtros de Bloom
8. Repaso: elegir una estructura

Lectura

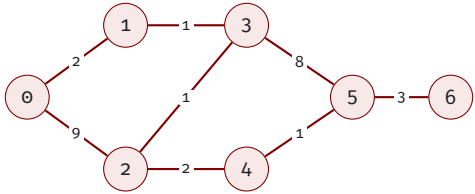
Apuntes de la semana 15. Open Data Structures, cap. 12 para la representación de grafos. CLRS, capítulo *Single-Source Shortest Paths*, la sección de Dijkstra. Código en `code/week-15`.

SECCIÓN 01

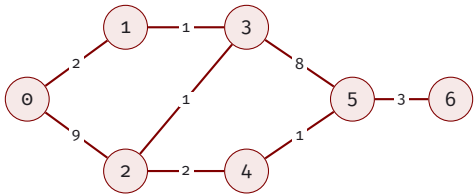
Con pesos, BFS ya no alcanza

01

La pregunta de la clase pasada

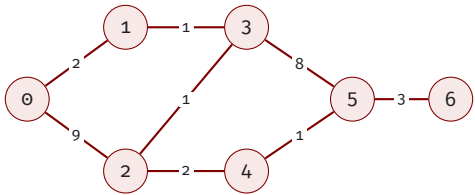


La pregunta de la clase pasada



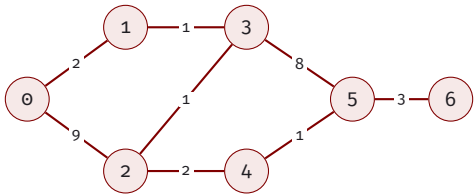
- El grafo de la semana 14, con los mismos nodos y aristas, ahora con metros.

La pregunta de la clase pasada



- El grafo de la semana 14, con los mismos nodos y aristas, ahora con metros.
- BFS, de 0 a 6: 0-1-3-5-6. **4 aristas, 14 metros.**

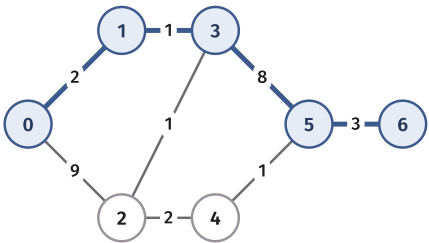
La pregunta de la clase pasada



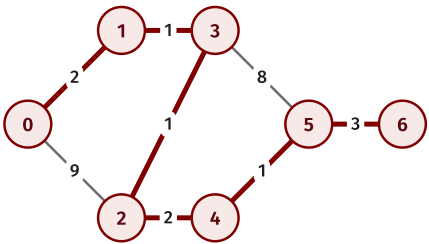
- El grafo de la semana 14, con los mismos nodos y aristas, ahora con metros.
- BFS, de 0 a 6: 0-1-3-5-6. **4 aristas, 14 metros.**
- ¿Puede haber otra ruta, con más aristas y menos metros?

Sí la hay

BFS: 4 aristas, 14 metros



Dijkstra: 6 aristas, 10 metros



El camino con menos aristas no es el más corto. BFS no mira los pesos: responde otra pregunta.

Lo dice el programa

terminal

```
$ build/dijkstra_trace | tail -4
ruta de Dijkstra a 6: 0 1 3 2 4 5 6
  aristas: 6, metros: 10
ruta de BFS a 6: 0 1 3 5 6
  aristas: 4, metros: 14
```

No es una rareza: Miraflores

route_demo, de Malecón de la Reserva a Avenida Arequipa

```
de Malecón de la Reserva (nodo 1839) a Avenida Arequipa (nodo 1883)
linea recta: 1215 metros
BFS:      18 aristas, 1871 metros
Dijkstra: 25 aristas, 1512 metros
ruta sobre linea recta: 1.24
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

No es una rareza: Miraflores

route_demo, de Malecón de la Reserva a Avenida Arequipa

```
de Malecón de la Reserva (nodo 1839) a Avenida Arequipa (nodo 1883)
linea recta: 1215 metros
BFS:      18 aristas, 1871 metros
Dijkstra: 25 aristas, 1512 metros
ruta sobre linea recta: 1.24
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **7 aristas más, y 359 metros menos.**

No es una rareza: Miraflores

route_demo, de Malecón de la Reserva a Avenida Arequipa

```
de Malecón de la Reserva (nodo 1839) a Avenida Arequipa (nodo 1883)
linea recta: 1215 metros
BFS:      18 aristas, 1871 metros
Dijkstra: 25 aristas, 1512 metros
ruta sobre linea recta: 1.24
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **7 aristas más, y 359 metros menos.**
- La línea recta es una revisión barata: ninguna ruta puede ser más corta.

No es una rareza: Miraflores

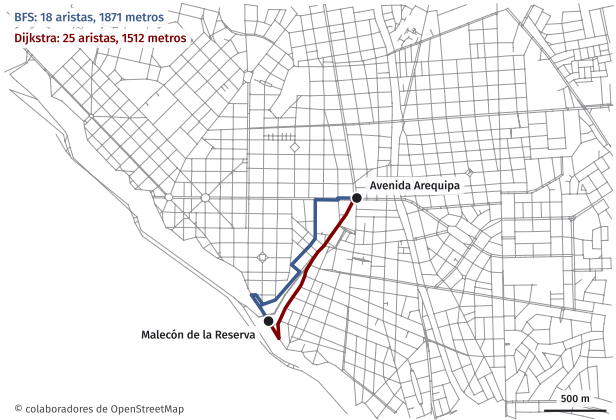
route_demo, de Malecón de la Reserva a Avenida Arequipa

```
de Malecón de la Reserva (nodo 1839) a Avenida Arequipa (nodo 1883)
linea recta: 1215 metros
BFS:      18 aristas, 1871 metros
Dijkstra: 25 aristas, 1512 metros
ruta sobre linea recta: 1.24
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **7 aristas más, y 359 metros menos.**
- La línea recta es una revisión barata: ninguna ruta puede ser más corta.
- Detecta una ruta absurda. No prueba que esta sea la mejor.

Las dos rutas, sobre el mapa



Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Camino mínimo

Camino mínimo ***(shortest path)***

El **peso** de un camino es la suma de los pesos de sus aristas. Un **camino mínimo** de s a t es uno cuyo peso es el menor posible. La **distancia** de s a t es ese peso.

Camino mínimo

Camino mínimo ***(shortest path)***

El **peso** de un camino es la suma de los pesos de sus aristas. Un **camino mínimo** de s a t es uno cuyo peso es el menor posible. La **distancia** de s a t es ese peso.

- Puede haber varios caminos mínimos, todos con el mismo peso.

Camino mínimo

Camino mínimo ***(shortest path)***

El **peso** de un camino es la suma de los pesos de sus aristas. Un **camino mínimo** de s a t es uno cuyo peso es el menor posible. La **distancia** de s a t es ese peso.

- Puede haber varios caminos mínimos, todos con el mismo peso.
- Hasta hoy, “distancia” era cantidad de aristas. En un grafo con pesos, es la suma de los pesos.

Camino mínimo

Camino mínimo *(shortest path)*

El **peso** de un camino es la suma de los pesos de sus aristas. Un **camino mínimo** de s a t es uno cuyo peso es el menor posible. La **distancia** de s a t es ese peso.

- Puede haber varios caminos mínimos, todos con el mismo peso.
- Hasta hoy, “distancia” era cantidad de aristas. En un grafo con pesos, es la suma de los pesos.
- Lo que veremos, desde un origen, responde a **todos** los destinos.

SECCIÓN 02

Relajar una arista

02

Relajar

Conocemos una ruta de s a u. Si hay arista u $\to$ v, conocemos también una ruta a v. ¿Es mejor?

pseudocódigo

```
algoritmo relajar(u, v, w)
    si distancia[u] + w < distancia[v] entonces
        distancia[v] ⟵ distancia[u] + w
        padre[v] ⟵ u
    fin si
fin algoritmo
```

Relajar

Conocemos una ruta de s a u. Si hay arista u $\to$ v, conocemos también una ruta a v. ¿Es mejor?

pseudocódigo

```
algoritmo relajar(u, v, w)
    si distancia[u] + w < distancia[v] entonces
        distancia[v] ← distancia[u] + w
        padre[v] ← u
    fin si
fin algoritmo
```

- Al principio, `distancia[s]` = 0, y las demás valen **infinito**.

Relajar

Conocemos una ruta de s a u. Si hay arista u $\to$ v, conocemos también una ruta a v. ¿Es mejor?

pseudocódigo

```
algoritmo relajar(u, v, w)
    si distancia[u] + w < distancia[v] entonces
        distancia[v] ← distancia[u] + w
        padre[v] ← u
    fin si
fin algoritmo
```

- Al principio, `distancia[s]` = 0, y las demás valen **infinito**.
- Toda distancia finita es el peso de **alguna ruta real**: la de los padres.

Relajar

Conocemos una ruta de s a u. Si hay arista u $\to$ v, conocemos también una ruta a v. ¿Es mejor?

pseudocódigo

```
algoritmo relajar(u, v, w)
    si distancia[u] + w < distancia[v] entonces
        distancia[v] ← distancia[u] + w
        padre[v] ← u
    fin si
fin algoritmo
```

- Al principio, `distancia[s]` = 0, y las demás valen **infinito**.
- Toda distancia finita es el peso de **alguna ruta real**: la de los padres.
- Por eso, una distancia nunca queda por debajo de la verdadera. Relajar solo la baja.

Relajar

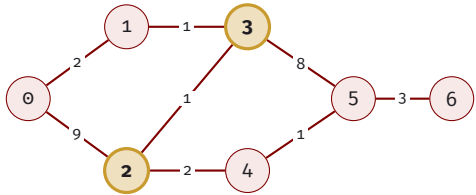
Conocemos una ruta de s a u. Si hay arista u → v, conocemos también una ruta a v. ¿Es mejor?

pseudocódigo

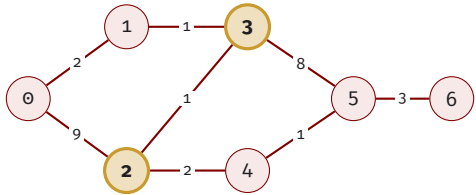
```
algoritmo relajar(u, v, w)
    si distancia[u] + w < distancia[v] entonces
        distancia[v] ← distancia[u] + w
        padre[v] ← u
    fin si
fin algoritmo
```

- Al principio, `distancia[s]` = 0, y las demás valen **infinito**.
- Toda distancia finita es el peso de **alguna ruta real**: la de los padres.
- Por eso, una distancia nunca queda por debajo de la verdadera. Relajar solo la baja.
- Falta decidir **en qué orden** se relaja. Ese es Dijkstra.

Un ejemplo

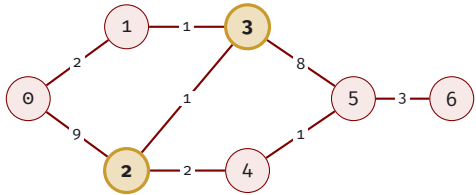


Un ejemplo



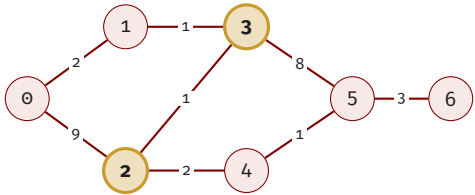
- `distancia[3]` = 3, y `distancia[2]` = 9.

Un ejemplo



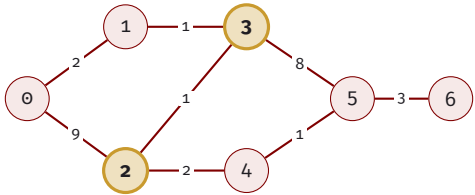
- `distancia[3]` = 3, y `distancia[2]` = 9.
- Relajar $3 \to 2$, de peso 1: 3 + 1 = 4, menor que 9. **Se actualiza.**

Un ejemplo



- `distancia[3]` = 3, y `distancia[2]` = 9.
- Relajar $3 \to 2$, de peso 1: 3 + 1 = 4, menor que 9. **Se actualiza.**
- `distancia[2]` = 4, y `padre[2]` = 3.

Un ejemplo



- `distancia[3]` = 3, y `distancia[2]` = 9.
- Relajar 3 $\rightarrow$ 2, de peso 1: 3 + 1 = 4, menor que 9. **Se actualiza.**
- `distancia[2]` = 4, y `padre[2]` = 3.
- Relajar 2 $\rightarrow$ 3: 4 + 1 = 5, no es menor que 3. **No cambia nada.**

SECCIÓN 03

Dijkstra

03

La idea

Dijkstra

De todos los nodos sin atender, se atiende el de **menor distancia conocida**, y se relajan sus aristas.

La idea

Dijkstra

De todos los nodos sin atender, se atiende el de **menor distancia conocida**, y se relajan sus aristas.

- Es la onda de BFS, solo que avanza en metros y no en aristas.

La idea

Dijkstra

De todos los nodos sin atender, se atiende el de **menor distancia conocida**, y se relajan sus aristas.

- Es la onda de BFS, solo que avanza en metros y no en aristas.
- Para encontrar rápido el menor está el **heap** de la semana 13.

La idea

Dijkstra

De todos los nodos sin atender, se atiende el de **menor distancia conocida**, y se relajan sus aristas.

- Es la onda de BFS, solo que avanza en metros y no en aristas.
- Para encontrar rápido el menor está el **heap** de la semana 13.
- En el heap van pares **(distancia, nodo)**, y manda la distancia.

La idea

Dijkstra

De todos los nodos sin atender, se atiende el de **menor distancia conocida**, y se relajan sus aristas.

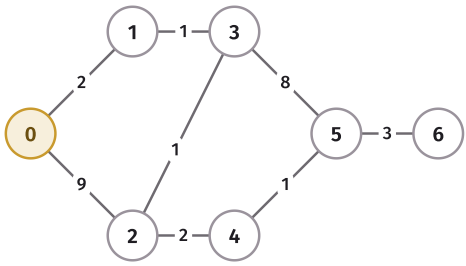
- Es la onda de BFS, solo que avanza en metros y no en aristas.
- Para encontrar rápido el menor está el **heap** de la semana 13.
- En el heap van pares **(distancia, nodo)**, y manda la distancia.
- Un nodo está **sacado** cuando sale por primera vez. Ahí su distancia ya es definitiva.

En pseudocódigo

pseudocódigo

```
algoritmo dijkstra(grafo, origen)
    distancia[v] ← infinito, padre[v] ← -1 y sacado[v] ← falso para todo nodo v
    distancia[origen] ← 0
    padre[origen] ← origen
    meter (0, origen) al heap
    mientras el heap no esté vacío hacer
        (d, u) ← sacar el menor del heap
        si sacado[u] entonces
            volver al inicio del ciclo
        fin si
        sacado[u] ← verdadero
        para cada arista (u, v, w) de la lista de u hacer
            si distancia[u] + w < distancia[v] entonces
                distancia[v] ← distancia[u] + w
                padre[v] ← u
                meter (distancia[v], v) al heap
            fin si
        fin para
    fin mientras
fin algoritmo
```

Paso a paso, desde el 0



DISTANCIA

0	∞	∞	∞	∞	∞	∞
0	1	2	3	4	5	6

PADRE

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

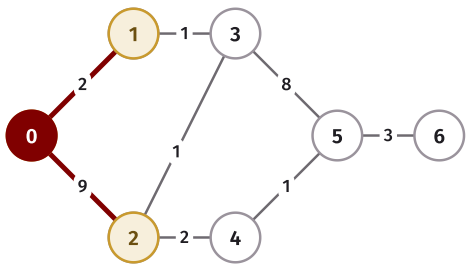
SALE | HEAP, EN ORDEN DE SALIDA

	(0,0)			

acaba de salir | sacado: distancia definitiva | con una ruta conocida | entrada vieja

Al inicio: la distancia del 0 es 0, las demás son infinito, y el heap tiene (0,0)

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	9	∞	∞	∞	∞

PADRE

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

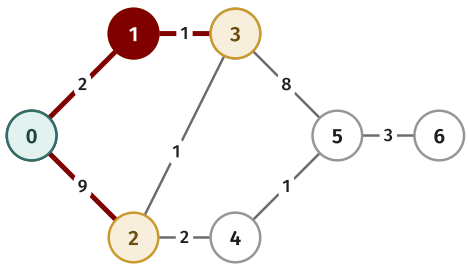
SALE: (0,0)

HEAP, EN ORDEN DE SALIDA: (2,1) (9,2)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (0,0). Mejoran el 1, a 2, y el 2, a 9: entran (2,1) y (9,2)

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	9	3	∞	∞	∞

PADRE

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

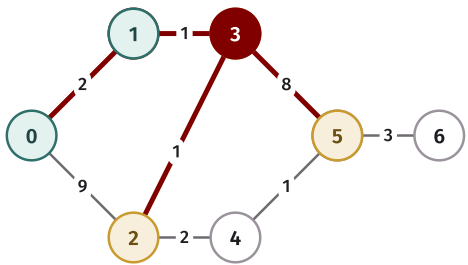
SALE: (2,1)

HEAP, EN ORDEN DE SALIDA: (3,3) (9,2)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (2,1). Mejora el 3, a 3: entra (3,3)

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	4	3	∞	11	∞

PADRE

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

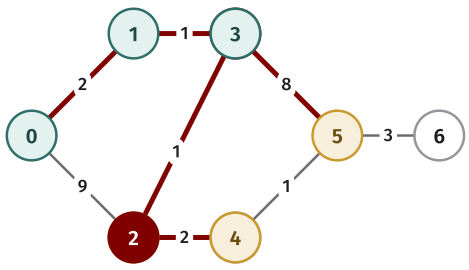
SALE: (3,3)

HEAP, EN ORDEN DE SALIDA: (4,2) (9,2) (11,5)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (3,3). El 2 baja de 9 a 4: entra (4,2), y (9,2) se queda como vieja. El 5 llega a 11

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	4	3	6	11	∞

PADRE

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

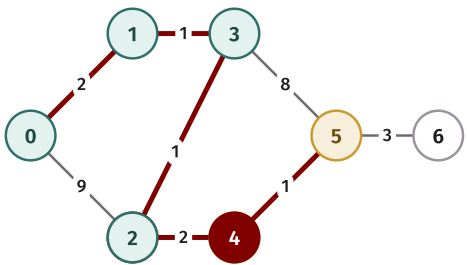
SALE: (4,2)

HEAP, EN ORDEN DE SALIDA: (6,4) (9,2) (11,5)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (4,2). Mejora el 4, a 6

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	4	3	6	7	∞

PADRE

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

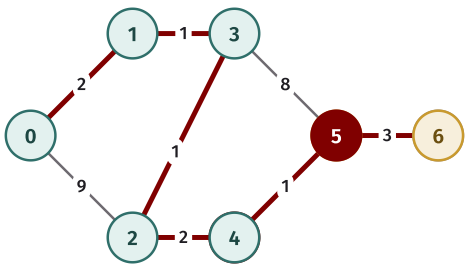
SALE: (6,4)

HEAP, EN ORDEN DE SALIDA: (7,5) (9,2) (11,5)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (6,4). El 5 baja de 11 a 7: entra (7,5), y (11,5) se queda como vieja

Paso a paso, desde el 0



DISTANCIA

0	1	2	3	4	5	6
0	2	4	3	6	7	10

PADRE

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

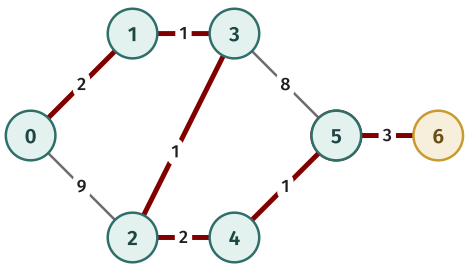
SALE: (7,5)

HEAP, EN ORDEN DE SALIDA: (9,2) (10,6) (11,5)

acaba de salir · sacado: distancia definitiva · con una ruta conocida · entrada vieja

Sale (7,5). Mejora el 6, a 10

Paso a paso, desde el 0



DISTANCIA

0	2	4	3	6	7	10
0	1	2	3	4	5	6

PADRE

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

SALE | HEAP, EN ORDEN DE SALIDA

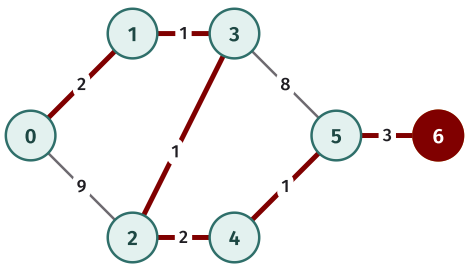
(9,2)	(10,6)	(11,5)		

vieja: se ignora

acaba de salir | sacado: distancia definitiva | con una ruta conocida | entrada vieja

Sale (9,2), pero el 2 ya salió, con 4. Es una entrada vieja: se ignora, sin leer su lista

Paso a paso, desde el 0



DISTANCIA

0	2	4	3	6	7	10
0	1	2	3	4	5	6

PADRE

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

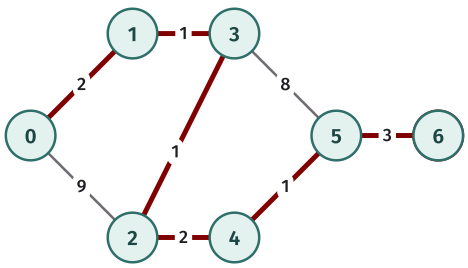
SALE | HEAP, EN ORDEN DE SALIDA

(10,6) | (11,5)

acaba de salir | sacado: distancia definitiva | con una ruta conocida | entrada vieja

Sale (10,6). Su único vecino, el 5, ya tiene 7: no relaja nada

Paso a paso, desde el 0



DISTANCIA

0	2	4	3	6	7	10
0	1	2	3	4	5	6

PADRE

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

SALE | HEAP, EN ORDEN DE SALIDA

(11,5) | | | | |

vieja: se ignora

acaba de salir | sacado: distancia definitiva | con una ruta conocida | entrada vieja

Sale (11,5): vieja, se ignora. Heap vacío. Ruta al 6: 0 1 3 2 4 5 6, con 10 metros

Lo mismo, en el programa

build/dijkstra_trace

```
sale     relaja     distancias      heap
-        -          0 - - - - - -   (0,0)
(0,0)    1 2        0 2 9 - - - -   (2,1) (9,2)
(2,1)    3          0 2 9 3 - - -   (3,3) (9,2)
(3,3)    2 5        0 2 4 3 - 11 -  (4,2) (9,2) (11,5)
(4,2)    4          0 2 4 3 6 11 -  (6,4) (9,2) (11,5)
(6,4)    5          0 2 4 3 6 7 -   (7,5) (9,2) (11,5)
(7,5)    6          0 2 4 3 6 7 10  (9,2) (10,6) (11,5)
(9,2)    vieja      0 2 4 3 6 7 10  (10,6) (11,5)
(10,6)              0 2 4 3 6 7 10  (11,5)
(11,5)   vieja      0 2 4 3 6 7 10
ruta de Dijkstra a 6: 0 1 3 2 4 5 6
  aristas: 6, metros: 10
ruta de BFS a 6: 0 1 3 5 6
  aristas: 4, metros: 14
```

9 entradas, 7 nodos sacados y 2 viejas: las pruebas lo comprueban.

En C++, sin las líneas que cuentan

shortest_paths.h

```cpp
while (!heap.empty()) {
    int u = heap.top().second;
    heap.pop();
    if (settled[u]) {  // entrada vieja: ya salió con una distancia menor
        continue;
    }
    settled[u] = true;
    for (const Edge& edge : graph.neighbors(u)) {
        Meters through_u = result.distance[u] + edge.weight;
        if (through_u < result.distance[edge.to]) {  // relajar la arista
            result.distance[edge.to] = through_u;
            result.parent[edge.to] = u;
            heap.push({through_u, edge.to});
        }
    }
}
```

Aquí, con `std::priority_queue`. En el laboratorio, **con su `MinHeap`**.

Los tipos, y las dos colas

shortest_paths.h

```cpp
using Meters = std::int64_t;
const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
using HeapEntry = std::pair<Meters, int>;
using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
    HeapEntry>>;
```

Los tipos, y las dos colas

shortest_paths.h

```cpp
using Meters = std::int64_t;
const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
using HeapEntry = std::pair<Meters, int>;
using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
    HeapEntry>>;
```

- `std::pair` compara por `first`, y luego por `second`: manda la distancia.

Los tipos, y las dos colas

shortest_paths.h

```
1  using Meters = std::int64_t;
2  const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
3  using HeapEntry = std::pair<Meters, int>;
4  using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
       HeapEntry>>;
```

- `std::pair` compara por `first`, y luego por `second`: manda la distancia.
- `MinHeap<HeapEntry>`, con `std::less`, deja arriba al **menor**: `less(a, b)` dice que a va más cerca de la raíz.

Los tipos, y las dos colas

shortest_paths.h

```cpp
using Meters = std::int64_t;
const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
using HeapEntry = std::pair<Meters, int>;
using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
    HeapEntry>>;
```

- `std::pair` compara por `first`, y luego por `second`: manda la distancia.
- `MinHeap<HeapEntry>`, con `std::less`, deja arriba al **menor**: `less(a, b)` dice que a va más cerca de la raíz.
- `std::priority_queue`, con ese mismo comparador, deja arriba al **mayor**. Por eso necesita `std::greater`.

Los tipos, y las dos colas

shortest_paths.h

```cpp
using Meters = std::int64_t;
const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
using HeapEntry = std::pair<Meters, int>;
using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
    HeapEntry>>;
```

- `std::pair` compara por `first`, y luego por `second`: manda la distancia.
- `MinHeap<HeapEntry>`, con `std::less`, deja arriba al **menor**: `less(a, b)` dice que a va más cerca de la raíz.
- `std::priority_queue`, con ese mismo comparador, deja arriba al **mayor**. Por eso necesita `std::greater`.
- En los dos, `pop` no devuelve nada: primero se lee `top`.

Los tipos, y las dos colas

shortest_paths.h

```
1  using Meters = std::int64_t;
2  const Meters UNREACHABLE = std::numeric_limits<Meters>::max();
3  using HeapEntry = std::pair<Meters, int>;
4  using MinFirstQueue = std::priority_queue<HeapEntry, std::vector<HeapEntry>, std::greater<
       HeapEntry>>;
```

- `std::pair` compara por `first`, y luego por `second`: manda la distancia.
- `MinHeap<HeapEntry>`, con `std::less`, deja arriba al **menor**: `less(a, b)` dice que a va más cerca de la raíz.
- `std::priority_queue`, con ese mismo comparador, deja arriba al **mayor**. Por eso necesita `std::greater`.
- En los dos, `pop` no devuelve nada: primero se lee `top`.
- `UNREACHABLE` hace de infinito. Nunca se le suma nada: solo se suma desde un nodo sacado.

Por qué funciona

Esto es intuición, no una demostración

La demostración y otros algoritmos quedan para Algoritmos.

Por qué funciona

Esto es intuición, no una demostración

La demostración y otros algoritmos quedan para Algoritmos.

- Cuando sale u con la menor distancia, todas las demás entradas pesan lo mismo o más.

Por qué funciona

Esto es intuición, no una demostración

La demostración y otros algoritmos quedan para Algoritmos.

- Cuando sale u con la menor distancia, todas las demás entradas pesan lo mismo o más.
- Otra forma de llegar a u tendría que pasar por uno de esos pendientes, y seguir desde ahí.

Por qué funciona

Esto es intuición, no una demostración

La demostración y otros algoritmos quedan para Algoritmos.

- Cuando sale u con la menor distancia, todas las demás entradas pesan lo mismo o más.
- Otra forma de llegar a u tendría que pasar por uno de esos pendientes, y seguir desde ahí.
- Si ninguna arista resta, seguir no baja el total. No puede salir mejor.

Por qué funciona

Esto es intuición, no una demostración

La demostración y otros algoritmos quedan para Algoritmos.

- Cuando sale u con la menor distancia, todas las demás entradas pesan lo mismo o más.
- Otra forma de llegar a u tendría que pasar por uno de esos pendientes, y seguir desde ahí.
- Si ninguna arista resta, seguir no baja el total. No puede salir mejor.
- Esa frase **supone** algo: que ninguna arista reste.

La ruta

- Como en BFS: `padre[v]` dice de dónde se llegó, y se sube hasta el origen.

La ruta

- Como en BFS: `padre[v]` dice de dónde se llegó, y se sube hasta el origen.
- `path_to` de la semana 14 sirve sin cambiar nada.

La ruta

- Como en BFS: `padre[v]` dice de dónde se llegó, y se sube hasta el origen.
- `path_to` de la semana 14 sirve sin cambiar nada.
- Lo distinto: en BFS, el padre se escribe una vez. Aquí se **sobrescribe** con cada mejora. `padre[2]` empieza en 0, y termina en 3.

La ruta

- Como en BFS: `padre[v]` dice de dónde se llegó, y se sube hasta el origen.
- `path_to` de la semana 14 sirve sin cambiar nada.
- Lo distinto: en BFS, el padre se escribe una vez. Aquí se **sobrescribe** con cada mejora. `padre[2]` empieza en 0, y termina en 3.
- Los padres forman el **árbol de caminos mínimos**: una corrida responde a todos los destinos.

La ruta

- Como en BFS: `padre[v]` dice de dónde se llegó, y se sube hasta el origen.
- `path_to` de la semana 14 sirve sin cambiar nada.
- Lo distinto: en BFS, el padre se escribe una vez. Aquí se **sobrescribe** con cada mejora. `padre[2]` empieza en 0, y termina en 3.
- Los padres forman el **árbol de caminos mínimos**: una corrida responde a todos los destinos.
- Si dos rutas empatan, cualquiera sirve. No esperen una ruta exacta, sino una **de ese peso**.

SECCIÓN 04

Entradas viejas

04

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- Subirla es el `sift_up` de siempre.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.
- Habría que llevar otro arreglo, con su posición, y actualizarlo en cada intercambio.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.
- Habría que llevar otro arreglo, con su posición, y actualizarlo en cada intercambio.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- La nueva tiene menor distancia: sale antes.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.
- Habría que llevar otro arreglo, con su posición, y actualizarlo en cada intercambio.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- La nueva tiene menor distancia: sale antes.
- Cuando sale la vieja, el nodo ya está sacado. Se descarta con una línea.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.
- Habría que llevar otro arreglo, con su posición, y actualizarlo en cada intercambio.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- La nueva tiene menor distancia: sale antes.
- Cuando sale la vieja, el nodo ya está sacado. Se descarta con una línea.
- Su `MinHeap` funciona como quedó.

Cuando la distancia mejora, ¿qué se hace con la entrada?

Actualizarla

Buscar (9,2), cambiarla por (4,2), y subirla.

- Subirla es el `sift_up` de siempre.
- Encontrarla es lo difícil: un heap no sabe buscar.
- Habría que llevar otro arreglo, con su posición, y actualizarlo en cada intercambio.

Dejarla, e ignorarla al sacar

Se mete una entrada nueva, y la vieja se queda.

- La nueva tiene menor distancia: sale antes.
- Cuando sale la vieja, el nodo ya está sacado. Se descarta con una línea.
- Su `MinHeap` funciona como quedó.

Olvidar esa línea no daña las distancias, y cada entrada vieja relee su lista. El laboratorio cuenta las aristas leídas: tienen que ser 2E.

Cada estructura en su sitio

Estructura	Para qué	Lo que más se usa
Listas de adyacencia	Los vecinos de u, con sus pesos	Recorrer una lista
Arreglo de distancias	La mejor distancia conocida	Leer y escribir, $\Theta(1)$
Arreglo de padres	De dónde se llegó, para la ruta	Escribir, $\Theta(1)$
Arreglo de sacados	Saber si una entrada es vieja	Leer, $\Theta(1)$
Heap	El siguiente nodo que se atiende	`push`, `top`, `pop`

Cada una hace lo que hace bien

El heap no sabe en qué posición está un nodo: los arreglos sí. Los arreglos no saben cuál es el menor: el heap sí.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- Una ruta sin nodos repetidos no pasa de la suma de todas las aristas: 8 540 931.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- Una ruta sin nodos repetidos no pasa de la suma de todas las aristas: 8 540 931.
- Unas 250 veces menos que el límite: **en estos mapas, un `int` no hace overflow**.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- Una ruta sin nodos repetidos no pasa de la suma de todas las aristas: 8 540 931.
- Unas 250 veces menos que el límite: **en estos mapas, un `int` no hace overflow**.
- 40 466 es solo una ruta larga que se encontró, no la más larga de todas. Basta con la suma.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- Una ruta sin nodos repetidos no pasa de la suma de todas las aristas: 8 540 931.
- Unas 250 veces menos que el límite: **en estos mapas, un `int` no hace overflow**.
- 40 466 es solo una ruta larga que se encontró, no la más larga de todas. Basta con la suma.
- Aun así, 64 bits: `dijkstra` recibe cualquier `Graph`, y dos aristas de 2 000 000 000 ya no caben en un `int`.

¿Alcanza un int?

distance_range

```
int llega hasta 2147483647
sizeof(HeapEntry): 16
../../data/mapas/lima.txt: arista mas larga 2574, suma de todas 8540931, ruta larga encontrada 40466
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- Una ruta sin nodos repetidos no pasa de la suma de todas las aristas: 8 540 931.
- Unas 250 veces menos que el límite: **en estos mapas, un `int` no hace overflow**.
- 40 466 es solo una ruta larga que se encontró, no la más larga de todas. Basta con la suma.
- Aun así, 64 bits: `dijkstra` recibe cualquier `Graph`, y dos aristas de 2 000 000 000 ya no caben en un `int`.
- El precio, en esta plataforma: 16 bytes por entrada, en vez de 8.

SECCIÓN 05

Dónde detenerse

05

BFS y Dijkstra, lado a lado

	BFS	Dijkstra
Pregunta	Menos aristas	Menos peso
Lo pendiente	Cola, $\Theta(1)$ por operación	Heap, $O(\log n)$ por operación
Veces que entra un nodo	Una, porque se marca al entrar	Una por cada mejora
Distancia definitiva	Al marcar, al meter	Al sacar por primera vez
Dónde se puede parar	Al marcar el destino	Al sacar el destino
Costo	$\Theta(V + E)$ si se alcanza todo	$O((V + E) \log V)$
Requiere	Nada de los pesos	Pesos no negativos

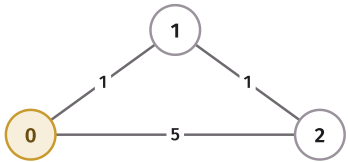
BFS y Dijkstra, lado a lado

	BFS	Dijkstra
Pregunta	Menos aristas	Menos peso
Lo pendiente	Cola, $\Theta(1)$ por operación	Heap, $O(\log n)$ por operación
Veces que entra un nodo	Una, porque se marca al entrar	Una por cada mejora
Distancia definitiva	Al marcar, al meter	Al sacar por primera vez
Dónde se puede parar	Al marcar el destino	Al sacar el destino
Costo	$\Theta(V + E)$ si se alcanza todo	$O((V + E) \log V)$
Requiere	Nada de los pesos	Pesos no negativos

Con todos los pesos iguales, Dijkstra da las mismas distancias que BFS: la primera vez que se descubre un nodo ya es la mejor. BFS es lo que queda cuando los pesos no importan.

Parar al meter, o al sacar

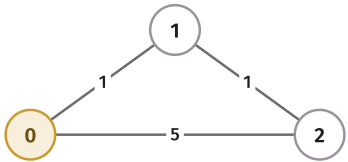
para al meter el destino



distancia: 0 ∞ ∞

HEAP (0,0)

para al sacar el destino



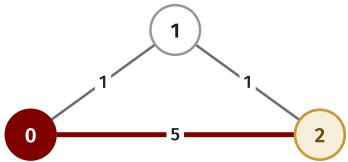
distancia: 0 ∞ ∞

HEAP (0,0)

Las dos versiones empiezan igual, con (0,0) en el heap. Quieren llegar al 2

Parar al meter, o al sacar

para al meter el destino

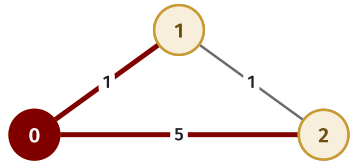


distancia: 0 ∞ 5

HEAP

responde 5

para al sacar el destino



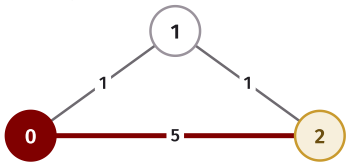
distancia: 0 1 5

HEAP (1,1) (5,2)

Sale (0,0). Su lista empieza por el 2, que se descubre con 5: solo es una ruta, y la mala para ahí

Parar al meter, o al sacar

para al meter el destino

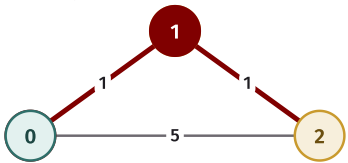


distancia: 0 ∞ 5

HEAP

responde 5

para al sacar el destino



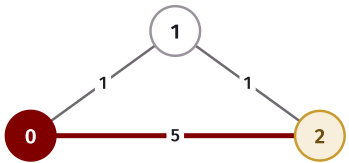
distancia: 0 1 2

HEAP (2,2) (5,2)

La buena sigue: sale (1,1), y el 2 baja de 5 a 2. La primera distancia de un nodo puede mejorar

Parar al meter, o al sacar

para al meter el destino

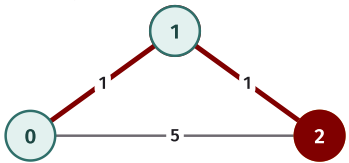


distancia: 0 ∞ 5

HEAP

responde 5

para al sacar el destino



distancia: 0 1 2

HEAP (5,2)

responde 2

Sale (2,2): al sacarlo, su distancia ya no puede mejorar, y la buena para. Responde 2

Detenerse al meter

errors/stop_at_push.cpp

```
de 0 a 2, al meter: 5, al sacar: 2
en el mapa, 158 de 1000 pares dan una distancia equivocada
la peor sobra 334 metros
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Detenerse al meter

- En la semana 14, BFS paraba al marcar el destino. Allí era seguro.

errors/stop_at_push.cpp

```
de 0 a 2, al meter: 5, al sacar: 2
en el mapa, 158 de 1000 pares dan una distancia equivocada
la peor sobra 334 metros
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Detenerse al meter

- En la semana 14, BFS paraba al marcar el destino. Allí era seguro.
- **Al meter, solo se conoce una ruta. Al sacar, ya no puede mejorar.**

errors/stop_at_push.cpp

```
de 0 a 2, al meter: 5, al sacar: 2
en el mapa, 158 de 1000 pares dan una distancia equivocada
la peor sobra 334 metros
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Detenerse al meter

- En la semana 14, BFS paraba al marcar el destino. Allí era seguro.
- **Al meter, solo se conoce una ruta. Al sacar, ya no puede mejorar.**

errors/stop_at_push.cpp

```
de 0 a 2, al meter: 5, al sacar: 2
en el mapa, 158 de 1000 pares dan una distancia equivocada
la peor sobra 334 metros
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Parar **al sacar** ahorra todo lo que está más lejos que el destino: casi todo si está cerca, y nada si es el más lejano. Lo medirán.

La onda se detiene en el destino



Sale el origen, con distancia 0. Los nodos salen de menor a mayor distancia: la onda avanza en metros

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

La onda se detiene en el destino



Ya salieron todos los nodos a 300 metros o menos: 7 de 2242

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

La onda se detiene en el destino



Ya salieron todos los nodos a 600 metros o menos: 52 de 2242

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

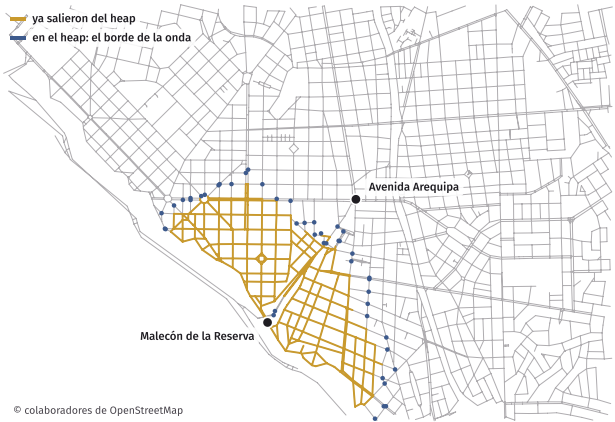
La onda se detiene en el destino



Ya salieron todos los nodos a 900 metros o menos: 126 de 2242

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

La onda se detiene en el destino



Ya salieron todos los nodos a 1200 metros o menos: 250 de 2242

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

La onda se detiene en el destino



Sale Avenida Arequipa, con 1512 metros: es definitiva, y aquí se puede parar. Salieron 441 de 2242

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

La onda se detiene en el destino



La ruta se lee subiendo por los padres. Los nodos más lejos que el destino no se atendieron

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Dos errores que no rompen nada

Olvidar `std::greater`. Sale primero lo más lejano:

errors/forgot_greater.cpp, en Miraflores

```
distancias equivocadas: 2230 de 2242
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL.

errors/reference_to_top.cpp, con ASan

```
$ build/reference_to_top
antes de pop: (3,2)
despues de pop: (5,1)
```

Dos errores que no rompen nada

Olvidar `std::greater`. Sale primero lo más lejano:

errors/forgot_greater.cpp, en Miraflores

```
distancias equivocadas: 2230 de 2242
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL.

Una referencia a `top()`. Apunta al primer lugar, y `pop` pone ahí otra entrada:

errors/reference_to_top.cpp, con ASan

```
$ build/reference_to_top
antes de pop: (3,2)
despues de pop: (5,1)
```

Dos errores que no rompen nada

Olvidar `std::greater`. Sale primero lo más lejano:

errors/forgot_greater.cpp, en Miraflores

```
distancias equivocadas: 2230 de 2242
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL.

Una referencia a `top()`. Apunta al primer lugar, y `pop` pone ahí otra entrada:

errors/reference_to_top.cpp, con ASan

```
$ build/reference_to_top
antes de pop: (3,2)
despues de pop: (5,1)
```

ASan no dice nada: esa memoria sigue siendo del vector. Copien antes de `pop`: `auto [d, u] = heap.top();`, sin `&`.

El peor de los errores: un heap de nodos

Una lambda que captura necesita el constructor `MinHeap(Less)`.

errors/comparator_reads_distance.cpp

```cpp
auto farther = [&distance](int a, int b) { return distance[a] > distance[b]; };
std::priority_queue<int, std::vector<int>, decltype(farther)> heap(farther);
```

en Miraflores, con esta biblioteca

```
origenes con alguna distancia equivocada: 2 de 200
distancias equivocadas en total: 3
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL.

Las claves de un heap no pueden cambiar mientras estén dentro

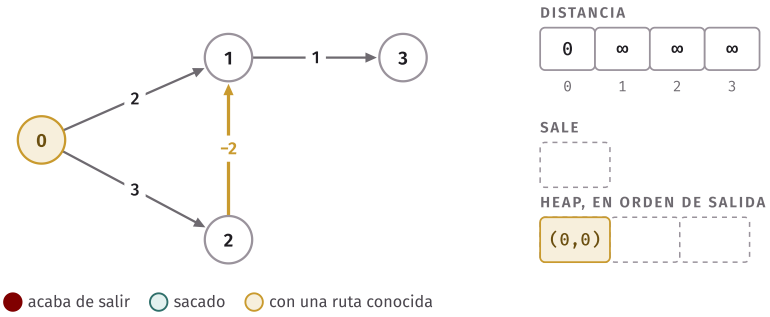
Relajar cambia la clave de un nodo que ya está dentro. Casi siempre da bien: una prueba con un solo origen no alcanza.

SECCIÓN 06

Pesos negativos

06

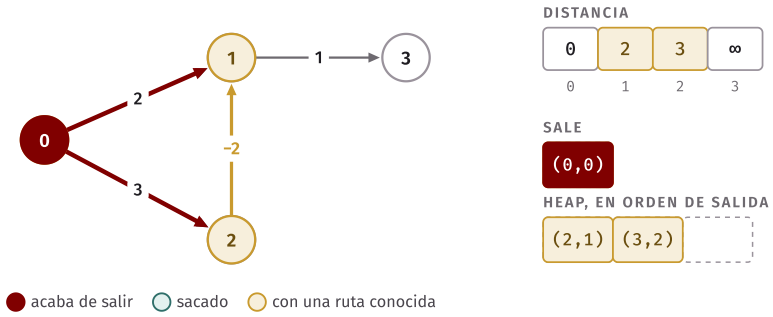
Dijkstra requiere pesos no negativos



Al inicio, el heap tiene (0,0). El grafo es dirigido, y la arista 2 → 1 resta 2

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

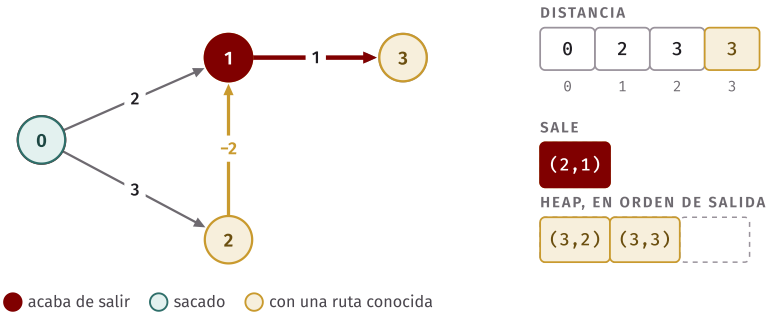
Dijkstra requiere pesos no negativos



Sale (0,0): el 1 queda en 2, y el 2 queda en 3

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

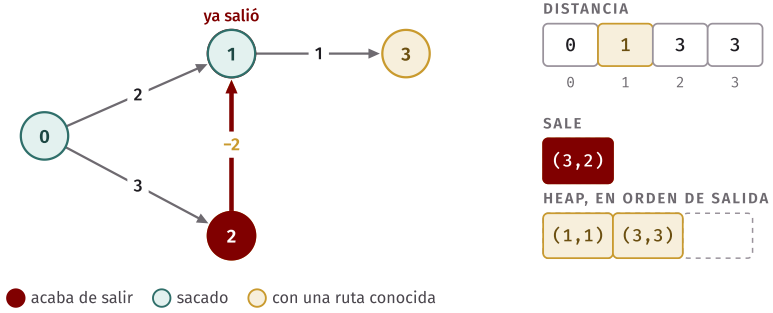
Dijkstra requiere pesos no negativos



Sale (2,1): el 1 queda sacado, y su distancia se da por definitiva. Relaja 1 → 3: el 3 queda en 3

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

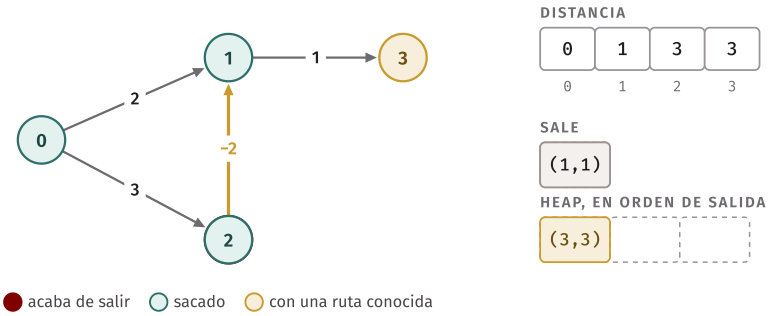
Dijkstra requiere pesos no negativos



Sale (3,2): 3 − 2 = 1 < 2. La distancia del 1 baja a 1 y entra (1,1). Pero el 1 ya salió

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

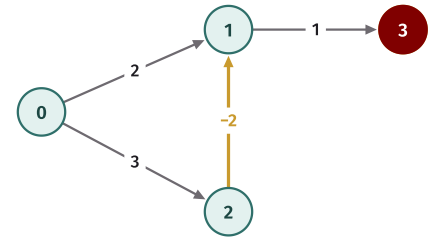
Dijkstra requiere pesos no negativos



Sale (1,1): es una entrada vieja, y se ignora. Nadie vuelve a leer la lista del 1, así que el 3 no mejora

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

Dijkstra requiere pesos no negativos



DISTANCIA

0	1	2	3
0	1	3	3

SALE

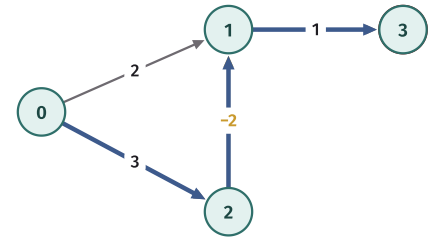
(3,3)

HEAP, EN ORDEN DE SALIDA

Sale (3,3), y termina. Dijkstra responde 0 1 3 3

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

Dijkstra requiere pesos no negativos



DISTANCIA

0	1	2	3
0	1	3	3

el 3 debía dar 2

SALE

HEAP, EN ORDEN DE SALIDA

acaba de salir · sacado · con una ruta conocida

La verdad es 0 → 2 → 1 → 3, con 3 + (−2) + 1 = 2. Con un peso negativo no hay garantía

Es **dirigido** por un motivo: en uno no dirigido, una arista negativa u-v ya es un ciclo, porque se puede ir y volver bajando el total.

Ni aunque en algún ejemplo acierte

errors/negative_edge.cpp

```
$ build/negative_edge
dijkstra:           0 1 3 3
ruta real a 3: 0 2 1 3, con 3 + (-2) + 1 = 2
sin arreglo:        0 1 3 2
entradas sacadas: 6
sin arreglo, parando al sacar el 1: 2 (la verdad es 1)
con un ciclo negativo, cortado tras 100 entradas sacadas
```

Ni aunque en algún ejemplo acierte

errors/negative_edge.cpp

```
$ build/negative_edge
dijkstra:           0 1 3 3
ruta real a 3: 0 2 1 3, con 3 + (-2) + 1 = 2
sin arreglo:        0 1 3 2
entradas sacadas: 6
sin arreglo, parando al sacar el 1: 2 (la verdad es 1)
con un ciclo negativo, cortado tras 100 entradas sacadas
```

- Sin el arreglo de sacados (entrada vieja si $d \neq$ `distancia[u]`), el 1 se procesa otra vez. Aquí acierta.

Ni aunque en algún ejemplo acierte

errors/negative_edge.cpp

```
$ build/negative_edge
dijkstra:           0 1 3 3
ruta real a 3: 0 2 1 3, con 3 + (-2) + 1 = 2
sin arreglo:        0 1 3 2
entradas sacadas: 6
sin arreglo, parando al sacar el 1: 2 (la verdad es 1)
con un ciclo negativo, cortado tras 100 entradas sacadas
```

- Sin el arreglo de sacados (entrada vieja si $d \neq$ `distancia[u]`), el 1 se procesa otra vez. Aquí acierta.
- Si para en el destino, ya no. Un nodo puede salir varias veces, y el costo deja de valer. Con un ciclo negativo, no termina.

Ni aunque en algún ejemplo acierte

errors/negative_edge.cpp

```
$ build/negative_edge
dijkstra:           0 1 3 3
ruta real a 3: 0 2 1 3, con 3 + (-2) + 1 = 2
sin arreglo:        0 1 3 2
entradas sacadas: 6
sin arreglo, parando al sacar el 1: 2 (la verdad es 1)
con un ciclo negativo, cortado tras 100 entradas sacadas
```

- Sin el arreglo de sacados (entrada vieja si $d \neq$ `distancia[u]`), el 1 se procesa otra vez. Aquí acierta.
- Si para en el destino, ya no. Un nodo puede salir varias veces, y el costo deja de valer. Con un ciclo negativo, no termina.
- **Antes de usarlo, revisen sus pesos.** Para pesos negativos hay otros algoritmos, que verán en Algoritmos.

SECCIÓN 07

El costo

07

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.
- **E:** cada lista se lee una sola vez. Son hasta 2E relajaciones, y cada una exitosa hace un `push`.

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.
- **E:** cada lista se lee una sola vez. Son hasta 2E relajaciones, y cada una exitosa hace un `push`.
- Entradas: como máximo **2E + 1**. Cada una sale una vez.

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.
- **E:** cada lista se lee una sola vez. Son hasta 2E relajaciones, y cada una exitosa hace un `push`.
- Entradas: como máximo **2E + 1**. Cada una sale una vez.
- **log V:** cada `push` o `pop` cuesta O(log n), con $n \leq 2E + 1$.

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.
- **E:** cada lista se lee una sola vez. Son hasta 2E relajaciones, y cada una exitosa hace un `push`.
- Entradas: como máximo **2E + 1**. Cada una sale una vez.
- **log V:** cada `push` o `pop` cuesta O(log n), con $n \leq 2E + 1$.
- Como $E < V^2$, $\log n \leq 2 \log V + 1$. Las entradas viejas no cambian el orden.

De dónde sale O((V + E) log V)

Con V nodos y E aristas, sin aristas repetidas:

- **V:** cada nodo sale una sola vez como sacado, gracias a la marca.
- **E:** cada lista se lee una sola vez. Son hasta 2E relajaciones, y cada una exitosa hace un `push`.
- Entradas: como máximo **2E + 1**. Cada una sale una vez.
- **log V:** cada `push` o `pop` cuesta O(log n), con $n \leq 2E + 1$.
- Como $E < V^2$, $\log n \leq 2 \log V + 1$. Las entradas viejas no cambian el orden.

Es una O, una cota superior

Las relajaciones exitosas pueden ser muchas menos que 2E. Y si no se alcanza todo el grafo, solo se paga su componente, más el $\Theta(V)$ de los arreglos.

Sobre los tres mapas

dijkstra_counts, con -O2. Los tiempos dependen de la máquina: mediana de 11 corridas, una de 3 series

nodos	aristas	sacados	aristas leidas	entradas	viejas	heap maximo	ms	ns/((V+E)log V)
2242	3474	2242	6948	2562	320	71	0.045	0.71
19560	29117	19560	58234	22572	3012	201	1.759	2.54
79050	116803	79050	233606	91604	12554	594	9.061	2.84

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Sobre los tres mapas

dijkstra_counts, con -O2. Los tiempos dependen de la máquina: mediana de 11 corridas, una de 3 series

nodos	aristas	sacados	aristas leidas	entradas	viejas	heap maximo	ms	ns/((V+E)log V)
2242	3474	2242	6948	2562	320	71	0.045	0.71
19560	29117	19560	58234	22572	3012	201	1.759	2.54
79050	116803	79050	233606	91604	12554	594	9.061	2.84

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Exacto:** `sacados` es V, y `aristas leidas` es 2E. Lo que predice el análisis.

Sobre los tres mapas

dijkstra_counts, con -O2. Los tiempos dependen de la máquina: mediana de 11 corridas, una de 3 series

nodos	aristas	sacados	aristas leidas	entradas	viejas	heap maximo	ms	ns/((V+E)log V)
2242	3474	2242	6948	2562	320	71	0.045	0.71
19560	29117	19560	58234	22572	3012	201	1.759	2.54
79050	116803	79050	233606	91604	12554	594	9.061	2.84

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Exacto:** `sacados` es V, y `aristas leidas` es 2E. Lo que predice el análisis.
- **Las viejas son pocas:** en `lima.txt`, el 14 % de las entradas. Ese era el precio.

Sobre los tres mapas

dijkstra_counts, con -O2. Los tiempos dependen de la máquina: mediana de 11 corridas, una de 3 series

nodos	aristas	sacados	aristas leidas	entradas	viejas	heap maximo	ms	ns/((V+E)log V)
2242	3474	2242	6948	2562	320	71	0.045	0.71
19560	29117	19560	58234	22572	3012	201	1.759	2.54
79050	116803	79050	233606	91604	12554	594	9.061	2.84

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Exacto:** `sacados` es V, y `aristas leidas` es 2E. Lo que predice el análisis.
- **Las viejas son pocas:** en `lima.txt`, el 14 % de las entradas. Ese era el precio.
- `entradas` queda muy por debajo de 2E + 1. El heap nunca pasó de 594 entradas. Es propiedad de los mapas, no del algoritmo.

Sobre los tres mapas

dijkstra_counts, con -O2. Los tiempos dependen de la máquina: mediana de 11 corridas, una de 3 series

nodos	aristas	sacados	aristas leidas	entradas	viejas	heap maximo	ms	ns/((V+E)log V)
2242	3474	2242	6948	2562	320	71	0.045	0.71
19560	29117	19560	58234	22572	3012	201	1.759	2.54
79050	116803	79050	233606	91604	12554	594	9.061	2.84

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Exacto:** `sacados` es V, y `aristas leidas` es 2E. Lo que predice el análisis.
- **Las viejas son pocas:** en `lima.txt`, el 14 % de las entradas. Ese era el precio.
- `entradas` queda muy por debajo de 2E + 1. El heap nunca pasó de 594 entradas. Es propiedad de los mapas, no del algoritmo.
- Los ns por unidad de (V + E) log V **suben** en las tres series. La fórmula es una cota, y no predice eso. Probablemente sea la cache: no tenemos `perf` para confirmarlo.

Sin heap

En cada vuelta, recorrer todo el arreglo buscando el menor sin sacar. V vueltas, V lecturas: $\Theta(V^2)$.

Mapa	V^2	2E + 1, máximo	Entradas de verdad
`miraflores.txt`	5 026 564	6 949	2 562
`lima-sur.txt`	382 593 600	58 235	22 572
`lima.txt`	6 248 902 500	233 607	91 604

Sin heap

En cada vuelta, recorrer todo el arreglo buscando el menor sin sacar. V vueltas, V lecturas: $\Theta(V^2)$.

Mapa	V^2	2E + 1, máximo	Entradas de verdad
`miraflores.txt`	5 026 564	6 949	2 562
`lima-sur.txt`	382 593 600	58 235	22 572
`lima.txt`	6 248 902 500	233 607	91 604

- Grafo disperso: el heap gana de lejos.

Sin heap

En cada vuelta, recorrer todo el arreglo buscando el menor sin sacar. V vueltas, V lecturas: $\Theta(V^2)$.

Mapa	V^2	2E + 1, máximo	Entradas de verdad
`miraflores.txt`	5 026 564	6 949	2 562
`lima-sur.txt`	382 593 600	58 235	22 572
`lima.txt`	6 248 902 500	233 607	91 604

- Grafo disperso: el heap gana de lejos.
- Grafo denso, E cerca de V^2: (V + E) log V se acerca a V^2 log V, mayor que V^2. Ahí el arreglo puede ser la buena elección.

Sin heap

En cada vuelta, recorrer todo el arreglo buscando el menor sin sacar. V vueltas, V lecturas: $\Theta(V^2)$.

Mapa	V^2	2E + 1, máximo	Entradas de verdad
`miraflores.txt`	5 026 564	6 949	2 562
`lima-sur.txt`	382 593 600	58 235	22 572
`lima.txt`	6 248 902 500	233 607	91 604

- Grafo disperso: el heap gana de lejos.
- Grafo denso, E cerca de V^2: (V + E) log V se acerca a V^2 log V, mayor que V^2. Ahí el arreglo puede ser la buena elección.
- Una vez más, manda la relación entre E y V^2. Lo medirán en el laboratorio.

Del centro a Miraflores

route_demo, sobre lima.txt

```
de Jirón de la Unión (nodo 52639) a Avenida José Larco (nodo 401)
linea recta: 8970 metros
BFS:       109 aristas, 9970 metros
Dijkstra: 121 aristas, 9253 metros
ruta sobre linea recta: 1.03
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Del centro a Miraflores

route_demo, sobre lima.txt

```
de Jirón de la Unión (nodo 52639) a Avenida José Larco (nodo 401)
linea recta: 8970 metros
BFS:       109 aristas, 9970 metros
Dijkstra: 121 aristas, 9253 metros
ruta sobre linea recta: 1.03
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Es el mínimo del grafo, no de la ciudad.** Si una calle falta en OpenStreetMap, no existe para Dijkstra.

Del centro a Miraflores

route_demo, sobre lima.txt

```
de Jirón de la Unión (nodo 52639) a Avenida José Larco (nodo 401)
linea recta: 8970 metros
BFS:       109 aristas, 9970 metros
Dijkstra: 121 aristas, 9253 metros
ruta sobre linea recta: 1.03
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Es el mínimo del grafo, no de la ciudad.** Si una calle falta en OpenStreetMap, no existe para Dijkstra.
- **Mide metros, no tiempo.** Con minutos, sería otra ruta, y el mismo algoritmo.

Del centro a Miraflores

route_demo, sobre lima.txt

```
de Jirón de la Unión (nodo 52639) a Avenida José Larco (nodo 401)
linea recta: 8970 metros
BFS:       109 aristas, 9970 metros
Dijkstra: 121 aristas, 9253 metros
ruta sobre linea recta: 1.03
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Es el mínimo del grafo, no de la ciudad.** Si una calle falta en OpenStreetMap, no existe para Dijkstra.
- **Mide metros, no tiempo.** Con minutos, sería otra ruta, y el mismo algoritmo.
- **Sentido de circulación.** Bastaría un grafo dirigido: `Graph(n, true)` ya lo permite. Falta ese dato en los archivos.

Del centro a Miraflores

route_demo, sobre lima.txt

```
de Jirón de la Unión (nodo 52639) a Avenida José Larco (nodo 401)
linea recta: 8970 metros
BFS:       109 aristas, 9970 metros
Dijkstra: 121 aristas, 9253 metros
ruta sobre linea recta: 1.03
```

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

- **Es el mínimo del grafo, no de la ciudad.** Si una calle falta en OpenStreetMap, no existe para Dijkstra.
- **Mide metros, no tiempo.** Con minutos, sería otra ruta, y el mismo algoritmo.
- **Sentido de circulación.** Bastaría un grafo dirigido: `Graph(n, true)` ya lo permite. Falta ese dato en los archivos.
- `*-calles.tsv` da **un** nodo por nombre, así que es la distancia entre esos dos nodos.

Cómo se prueba un Dijkstra

“Se ve bien” no es una prueba: ya vimos un error que falla en 2 de 200 orígenes.

Cómo se prueba un Dijkstra

“Se ve bien” no es una prueba: ya vimos un error que falla en 2 de 200 orígenes.

1. **Fuerza bruta.** Relajar todo hasta que nada cambie. Es lento, y se revisa leyéndolo. Se corre sobre cientos de grafos chicos al azar, con semilla fija.

Cómo se prueba un Dijkstra

“Se ve bien” no es una prueba: ya vimos un error que falla en 2 de 200 orígenes.

1. **Fuerza bruta.** Relajar todo hasta que nada cambie. Es lento, y se revisa leyéndolo. Se corre sobre cientos de grafos chicos al azar, con semilla fija.
2. **Que cada padre explique a su hijo.** Tiene que existir $p \to v$, con `distancia[p]` + w = `distancia[v]`. No depende de los empates.

Cómo se prueba un Dijkstra

"Se ve bien" no es una prueba: ya vimos un error que falla en 2 de 200 orígenes.

1. **Fuerza bruta.** Relajar todo hasta que nada cambie. Es lento, y se revisa leyéndolo. Se corre sobre cientos de grafos chicos al azar, con semilla fija.
2. **Que cada padre explique a su hijo.** Tiene que existir $p \to v$, con `distancia[p]` + w = `distancia[v]`. No depende de los empates.
3. **Contar.** En un grafo no dirigido conexo: V sacados, $2E$ entradas leídas, y `entradas` = `sacados` + `viejas`.

Cómo se prueba un Dijkstra

"Se ve bien" no es una prueba: ya vimos un error que falla en 2 de 200 orígenes.

1. **Fuerza bruta.** Relajar todo hasta que nada cambie. Es lento, y se revisa leyéndolo. Se corre sobre cientos de grafos chicos al azar, con semilla fija.
2. **Que cada padre explique a su hijo.** Tiene que existir $p \to v$, con `distancia[p]` + w = `distancia[v]`. No depende de los empates.
3. **Contar.** En un grafo no dirigido conexo: V sacados, 2E entradas leídas, y `entradas` = `sacados` + `viejas`.

Y contra BFS: Dijkstra nunca da más metros que BFS, y BFS nunca da más aristas que Dijkstra.

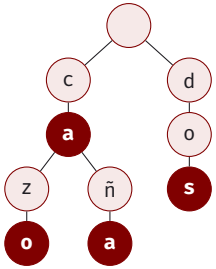
SECCIÓN 08

Opcional: tries

08



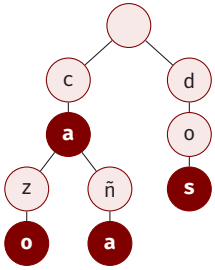
Un trie *(trie)*



Trie

Un árbol en el que cada nodo es un prefijo, y cada arista agrega un carácter. Las claves con un prefijo común comparten ese camino.

Un trie *(trie)*

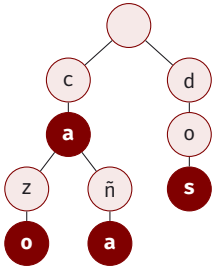


Trie

Un árbol en el que cada nodo es un prefijo, y cada arista agrega un carácter. Las claves con un prefijo común comparten ese camino.

- Claves: `ca`, `cazo`, `caña` y `dos`. En granate, donde termina una.

Un trie *(trie)*

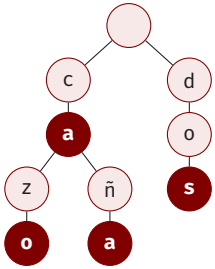


Trie

Un árbol en el que cada nodo es un prefijo, y cada arista agrega un carácter. Las claves con un prefijo común comparten ese camino.

- Claves: `ca`, `cazo`, `caña` y `dos`. En granate, donde termina una.
- `ca` es una clave y el prefijo de otras dos.

Un trie *(trie)*

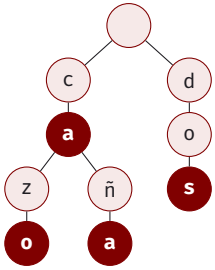


Trie

Un árbol en el que cada nodo es un prefijo, y cada arista agrega un carácter. Las claves con un prefijo común comparten ese camino.

- Claves: `ca`, `cazo`, `caña` y `dos`. En granate, donde termina una.
- `ca` es una clave y el prefijo de otras dos.
- Otra forma de hacer el autocompletado de la semana 6.

Un trie *(trie)*



Trie

Un árbol en el que cada nodo es un prefijo, y cada arista agrega un carácter. Las claves con un prefijo común comparten ese camino.

- Claves: `ca`, `cazo`, `caña` y `dos`. En granate, donde termina una.
- `ca` es una clave y el prefijo de otras dos.
- Otra forma de hacer el autocompletado de la semana 6.
- **Trabaja con bytes:** `ñ` son dos niveles (aquí la dibujamos en uno), y `ca` no ofrece `cámara`.

Rápido, y con mucha memoria

	Arreglo ordenado	Trie
Buscar m bytes	O(m log n)	O(m), sin importar n
Insertar	Θ(n) en el peor caso	O(m)
Memoria	Las claves, seguidas	Un nodo por prefijo distinto

calles de Lima, términos de los libros y todas las palabras

claves	bytes de claves	nodos	nodos/byte	con 256 hijos, MB
11197	197117	88288	0.45	90.41
28039	229806	76324	0.33	78.16
1045757	11897295	1884236	0.16	1929.46

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

Rápido, y con mucha memoria

	Arreglo ordenado	Trie
Buscar m bytes	O(m log n)	O(m), sin importar n
Insertar	Θ(n) en el peor caso	O(m)
Memoria	Las claves, seguidas	Un nodo por prefijo distinto

calles de Lima, términos de los libros y todas las palabras

claves	bytes de claves	nodos	nodos/byte	con 256 hijos, MB
11197	197117	88288	0.45	90.41
28039	229806	76324	0.33	78.16
1045757	11897295	1884236	0.16	1929.46

Datos: © colaboradores de OpenStreetMap, bajo la licencia ODbL. El mapa ignora el sentido de circulación: una ruta puede no servir para un carro.

1.9 GB para 12 MB de texto: un trie real usa algo más compacto por nodo, como el `std::map` del laboratorio, a costa de más tiempo por byte.

SECCIÓN 09

Opcional: filtros de Bloom

09

Un filtro de Bloom *(Bloom filter)*

Filtro de Bloom

Un arreglo de m bits y k funciones hash. **Agregar** enciende k bits. **Preguntar** mira esos k bits: si alguno está apagado, **no está**. Si todos están encendidos, **quizás**.

	0	1	2	3	4	5	6	7	8	9	10	11
al inicio	0	0	0	0	0	0	0	0	0	0	0	0

Un filtro de Bloom *(Bloom filter)*

Filtro de Bloom

Un arreglo de m bits y k funciones hash. **Agregar** enciende k bits. **Preguntar** mira esos k bits: si alguno está apagado, **no está**. Si todos están encendidos, **quizás**.

	0	1	2	3	4	5	6	7	8	9	10	11
al inicio	0	0	0	0	0	0	0	0	0	0	0	0
lima (2, 9)	0	0	1	0	0	0	0	0	0	1	0	0

Las posiciones de este ejemplo están hechas a mano, no calculadas.

Un filtro de Bloom *(Bloom filter)*

Filtro de Bloom

Un arreglo de m bits y k funciones hash. **Agregar** enciende k bits. **Preguntar** mira esos k bits: si alguno está apagado, **no está**. Si todos están encendidos, **quizás**.

	0	1	2	3	4	5	6	7	8	9	10	11
al inicio	0	0	0	0	0	0	0	0	0	0	0	0
lima (2, 9)	0	0	1	0	0	0	0	0	0	1	0	0
puno (5, 9)	0	0	1	0	0	1	0	0	0	1	0	0

Las posiciones de este ejemplo están hechas a mano, no calculadas.

Un filtro de Bloom *(Bloom filter)*

Filtro de Bloom

Un arreglo de m bits y k funciones hash. **Agregar** enciende k bits. **Preguntar** mira esos k bits: si alguno está apagado, **no está**. Si todos están encendidos, **quizás**.

	0	1	2	3	4	5	6	7	8	9	10	11
al inicio	0	0	0	0	0	0	0	0	0	0	0	0
lima (2, 9)	0	0	1	0	0	0	0	0	0	1	0	0
puno (5, 9)	0	0	1	0	0	1	0	0	0	1	0	0

- ¿cusco (3, 9)? El bit 3 está apagado: **no está, seguro**.

Las posiciones de este ejemplo están hechas a mano, no calculadas.

Un filtro de Bloom *(Bloom filter)*

Filtro de Bloom

Un arreglo de m bits y k funciones hash. **Agregar** enciende k bits. **Preguntar** mira esos k bits: si alguno está apagado, **no está**. Si todos están encendidos, **quizás**.

	0	1	2	3	4	5	6	7	8	9	10	11
al inicio	0	0	0	0	0	0	0	0	0	0	0	0
lima (2, 9)	0	0	1	0	0	0	0	0	0	1	0	0
puno (5, 9)	0	0	1	0	0	1	0	0	0	1	0	0

- ¿cusco (3, 9)? El bit 3 está apagado: **no está, seguro**.
- ¿tacna (2, 5)? Los dos encendidos: quizás está, y no se agregó. Un **falso positivo**.

Las posiciones de este ejemplo están hechas a mano, no calculadas.

¿Por qué nunca hay falsos negativos?

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.
2. Al agregarla, se encendieron esas k posiciones.

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.
2. Al agregarla, se encendieron esas k posiciones.
3. **Ningún bit se apaga jamás.**

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.
2. Al agregarla, se encendieron esas k posiciones.
3. **Ningún bit se apaga jamás.**
4. Al preguntar, esas k siguen encendidas. La respuesta tiene que ser “quizás”.

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.
2. Al agregarla, se encendieron esas k posiciones.
3. **Ningún bit se apaga jamás.**
4. Al preguntar, esas k siguen encendidas. La respuesta tiene que ser “quizás”.

Un sí es solo un quizás

Lo seguro es el **no**. Por lo mismo, no se puede borrar: apagar un bit podría dañar a otra clave.

¿Por qué nunca hay falsos negativos?

1. Las funciones hash dan siempre lo mismo para la misma clave.
2. Al agregarla, se encendieron esas k posiciones.
3. **Ningún bit se apaga jamás.**
4. Al preguntar, esas k siguen encendidas. La respuesta tiene que ser “quizás”.

Un sí es solo un quizás

Lo seguro es el **no**. Por lo mismo, no se puede borrar: apagar un bit podría dañar a otra clave.

No guarda las claves: no se puede listarlas, y su tasa de error depende de bits por clave y de k. Para analizarla hace falta probabilidad, así que la **medimos**.

Medirlo con palabras.txt

Se agrega una línea de cada dos. Las otras no están

```
$ build/solution/bloom_measure ../../data/palabras.txt
agregadas: 522879, preguntadas: 522878
bits por clave   k      MB  positivos     tasa  bits encendidos
             4   4    0.26      83442   15.958 %            63.2 %
            10   4    0.65       6148    1.176 %            33.0 %
            16   4    1.05       1300    0.249 %            22.1 %
con 10 bits por clave, cambiando k:
            10   1    0.65      49683    9.502 %             9.5 %
            10   7    0.65       4325    0.827 %            50.3 %
            10  25    0.65      61328   11.729 %            91.8 %
```

Medirlo con palabras.txt

Se agrega una línea de cada dos. Las otras no están

```
$ build/solution/bloom_measure ../../data/palabras.txt
agregadas: 522879, preguntadas: 522878
bits por clave   k      MB  positivos     tasa  bits encendidos
             4   4    0.26      83442   15.958 %            63.2 %
            10   4    0.65       6148    1.176 %            33.0 %
            16   4    1.05       1300    0.249 %            22.1 %
con 10 bits por clave, cambiando k:
            10   1    0.65      49683    9.502 %             9.5 %
            10   7    0.65       4325    0.827 %            50.3 %
            10  25    0.65      61328   11.729 %            91.8 %
```

- **Más bits, menos falsos positivos. k tiene un punto medio:** con muchos hash, se enciende casi todo.

Medirlo con palabras.txt

Se agrega una línea de cada dos. Las otras no están

```
$ build/solution/bloom_measure ../../data/palabras.txt
agregadas: 522879, preguntadas: 522878
bits por clave   k      MB  positivos     tasa  bits encendidos
             4   4    0.26      83442   15.958 %            63.2 %
            10   4    0.65       6148    1.176 %            33.0 %
            16   4    1.05       1300    0.249 %            22.1 %
con 10 bits por clave, cambiando k:
            10   1    0.65      49683    9.502 %             9.5 %
            10   7    0.65       4325    0.827 %            50.3 %
            10  25    0.65      61328   11.729 %            91.8 %
```

- **Más bits, menos falsos positivos. k tiene un punto medio:** con muchos hash, se enciende casi todo.
- **Sale barato:** 0.65 MB, con 0.83 % de error. Solo el texto de esas palabras ocupa unos 6 MB.

Medirlo con palabras.txt

Se agrega una línea de cada dos. Las otras no están

```
$ build/solution/bloom_measure ../../data/palabras.txt
agregadas: 522879, preguntadas: 522878
bits por clave   k      MB  positivos     tasa  bits encendidos
             4   4    0.26      83442   15.958 %            63.2 %
            10   4    0.65       6148    1.176 %            33.0 %
            16   4    1.05       1300    0.249 %            22.1 %
con 10 bits por clave, cambiando k:
            10   1    0.65      49683    9.502 %             9.5 %
            10   7    0.65       4325    0.827 %            50.3 %
            10  25    0.65      61328   11.729 %            91.8 %
```

- **Más bits, menos falsos positivos. k tiene un punto medio:** con muchos hash, se enciende casi todo.
- **Sale barato:** 0.65 MB, con 0.83 % de error. Solo el texto de esas palabras ocupa unos 6 MB.
- Se usa **delante de una búsqueda cara**: un no se la ahorra, y un quizás busca de verdad.

SECCIÓN 10

Repaso: elegir una estructura

10

Primero las operaciones

Si necesitan…	Usen…	Costo
Agregar al final y leer por índice	Arreglo dinámico	O(1) amortizado; Θ(1)
Último en entrar, primero en salir	Pila	O(1) amortizado
Primero en entrar, primero en salir	Cola o deque, buffer circular	O(1) amortizado
Buscar en datos fijos, ordenados	Arreglo ordenado, búsqueda binaria	O(log n), poca memoria
Diccionario, sin orden	Tabla hash	O(1) esperado
Diccionario **con orden**: rangos, sucesor	Árbol balanceado, `std::map`	O(log n) en el peor caso
El menor o el mayor, una y otra vez	Heap	O(log n) por operación
Conexiones y caminos	Grafo, listas de adyacencia	Θ(V + E) un recorrido
¿Están conectados?, con aristas que llegan	Union-find	O(α(n)) amortizado
Rutas con pesos no negativos	Dijkstra	O((V + E) log V)

La regla de toda la materia

Primero escriban qué operaciones necesita el problema, y cuáles son las más frecuentes. Después elijan.

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.**

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.**

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.
- **Saber si una palabra está en un diccionario fijo.**

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.
- **Saber si una palabra está en un diccionario fijo.** Una tabla hash, o un arreglo ordenado. Un filtro de Bloom solo no sirve, porque su sí no es seguro. Si se usa, va delante.

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.
- **Saber si una palabra está en un diccionario fijo.** Una tabla hash, o un arreglo ordenado. Un filtro de Bloom solo no sirve, porque su sí no es seguro. Si se usa, va delante.
- **Un historial con deshacer.**

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.
- **Saber si una palabra está en un diccionario fijo.** Una tabla hash, o un arreglo ordenado. Un filtro de Bloom solo no sirve, porque su sí no es seguro. Si se usa, va delante.
- **Un historial con deshacer.** Una pila: lo último que se hizo es lo primero que se deshace. Con "rehacer", dos pilas.

Cuatro casos, como en un examen

- **Las 10 palabras más frecuentes.** Contar con una tabla hash, y quedarse con las 10 mayores con un heap de tamaño 10: O(d log 10), en vez de ordenar las d palabras.
- **Todas las claves entre dos valores.** Importa el orden: un `std::map`, con `lower_bound` y avanzando. Si son fijas, un arreglo ordenado, con menos memoria.
- **Saber si una palabra está en un diccionario fijo.** Una tabla hash, o un arreglo ordenado. Un filtro de Bloom solo no sirve, porque su sí no es seguro. Si se usa, va delante.
- **Un historial con deshacer.** Una pila: lo último que se hizo es lo primero que se deshace. Con "rehacer", dos pilas.

Justifiquen con la operación que decide

"Porque es O(log n)" no justifica nada. "Porque necesito rangos, y una tabla hash no los da" sí.

Qué promete cada costo

Promesa	Qué dice	Ejemplo
Peor caso	Vale para **cada** operación	`push` de un heap, $O(\log n)$
Amortizado	Vale para la **suma** de una secuencia. Una sola puede ser cara. No tiene azar	`push_back`: $O(1)$ amortizado, $\Theta(n)$ una sola vez
Esperado	Depende de que el hash reparta bien, o del azar. El peor caso es peor	Tabla hash: $O(1)$ esperado, $\Theta(n)$ en el peor caso

Qué promete cada costo

Promesa	Qué dice	Ejemplo
Peor caso	Vale para **cada** operación	`push` de un heap, $O(\log n)$
Amortizado	Vale para la **suma** de una secuencia. Una sola puede ser cara. No tiene azar	`push_back`: $O(1)$ amortizado, $\Theta(n)$ una sola vez
Esperado	Depende de que el hash reparta bien, o del azar. El peor caso es peor	Tabla hash: $O(1)$ esperado, $\Theta(n)$ en el peor caso

- Decir “en promedio” para el `push_back` es un error.

Qué promete cada costo

Promesa	Qué dice	Ejemplo
Peor caso	Vale para **cada** operación	`push` de un heap, $O(\log n)$
Amortizado	Vale para la **suma** de una secuencia. Una sola puede ser cara. No tiene azar	`push_back`: $O(1)$ amortizado, $\Theta(n)$ una sola vez
Esperado	Depende de que el hash reparta bien, o del azar. El peor caso es peor	Tabla hash: $O(1)$ esperado, $\Theta(n)$ en el peor caso

- Decir “en promedio” para el `push_back` es un error.
- Decir “$O(1)$” para union-find, a secas, también.

SECCIÓN 11

El laboratorio

11

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

Sesión 2

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min

Sesión 2

```
terminal
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min
2. `dijkstra` con su `MinHeap`, y `route_to`. 60 min

Sesión 2

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min
2. `dijkstra` con su `MinHeap`, y `route_to`. 60 min
3. `route_meters`, y comparar con BFS. 35 min

Sesión 2

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min
2. `dijkstra` con su `MinHeap`, y `route_to`. 60 min
3. `route_meters`, y comparar con BFS. 35 min

Sesión 2

4. `dijkstra_to`: parar al sacar, y medir el ahorro. 40 min

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min
2. `dijkstra` con su `MinHeap`, y `route_to`. 60 min
3. `route_meters`, y comparar con BFS. 35 min

Sesión 2

4. `dijkstra_to`: parar al sacar, y medir el ahorro. 40 min
5. `dijkstra_priority_queue` y `dijkstra_array`: medir. 80 min

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Rutas sobre un mapa real

Empiezan copiando su heap: `cp ../week-13/lab/min_heap.h lab/min_heap.h`

Sesión 1

1. `build_map_graph`: de los tramos a las listas. 15 min
2. `dijkstra` con su `MinHeap`, y `route_to`. 60 min
3. `route_meters`, y comparar con BFS. 35 min

Sesión 2

4. `dijkstra_to`: parar al sacar, y medir el ahorro. 40 min
5. `dijkstra_priority_queue` y `dijkstra_array`: medir. 80 min
6. Opcional: `PlaceTrie` o `BloomFilter`.

terminal

```
make lab
make test-heap
make test-map
make test-dijkstra
make test-routes
make compare
make test-stop
make savings
make test-queues
make measure
make test-lab
```

Para la próxima clase

- **La semana 16 es el examen final:** todo el curso, con énfasis en las semanas 9 a 15. Lean `exams/README.md`.
- Terminen el laboratorio: rutas sobre el mapa, y las tablas de su informe.
- Para estudiar: las preguntas de repaso de cada guía, y simular cada estructura en papel antes de correr el código.
- El curso de Algoritmos demuestra por qué Dijkstra es correcto, y sigue desde aquí.

Pregunta

Corran Dijkstra a mano sobre un grafo de 6 nodos elegido por ustedes, con el heap y las distancias en cada paso. Luego, cambien una arista por un peso negativo. ¿Da mal? ¿Por qué?