

UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

Huffman y Dijkstra, demostrados

Retraso máximo, intercambio con inducción y grafos de estados

Gerson Garrido

Semana 5

Hoy

Sesión A

1. Minimizar el retraso máximo
2. Inversiones entre vecinas
3. Huffman: un intercambio *y* una inducción

Sesión B

4. Dijkstra: el invariante
5. Dónde se usan los pesos no negativos
6. El grafo de estados

> **Lectura**
>
> Apuntes de la semana 5. Código en `code/week-05`. Laboratorio en `lab/toll_routes`.

SECCIÓN 01

Minimizar el retraso máximo

01

Una máquina, muchos plazos

El problema

n tareas, cada una con duración $t_i > 0$ y **plazo** d_i. Una sola máquina, sin interrumpir.

Una máquina, muchos plazos

El problema

n tareas, cada una con duración $t_i > 0$ y **plazo** d_i. Una sola máquina, sin interrumpir.

- **Horario:** un orden de las tareas, sin tiempo muerto. La primera empieza en 0.

Una máquina, muchos plazos

El problema

n tareas, cada una con duración $t_i > 0$ y **plazo** d_i. Una sola máquina, sin interrumpir.

- **Horario:** un orden de las tareas, sin tiempo muerto. La primera empieza en 0.
- **Retraso:** $L_i = f_i - d_i$, donde f_i es el fin. Negativo si llega con holgura.

Una máquina, muchos plazos

El problema

n tareas, cada una con duración $t_i > 0$ y **plazo** d_i. Una sola máquina, sin interrumpir.

- **Horario:** un orden de las tareas, sin tiempo muerto. La primera empieza en 0.
- **Retraso:** $L_i = f_i - d_i$, donde f_i es el fin. Negativo si llega con holgura.
- **Objetivo:** minimizar el mayor L_i.

Una máquina, muchos plazos

El problema

n tareas, cada una con duración $t_i > 0$ y **plazo** d_i. Una sola máquina, sin interrumpir.

- **Horario:** un orden de las tareas, sin tiempo muerto. La primera empieza en 0.
- **Retraso:** $L_i = f_i - d_i$, donde f_i es el fin. Negativo si llega con holgura.
- **Objetivo:** minimizar el mayor L_i.

No se pierde nada

Si hay un hueco, se adelanta lo que viene después, y ningún fin sube. Y con $\max(0, f_i - d_i)$, el plazo más cercano sigue siendo óptimo.

Dos reglas que fallan

terminal

```
$ build/lateness_rules
Contra dura menos:
  tareas (duración, plazo): A(1, 100) B(10, 10)
  óptimo por fuerza bruta: 0
  dura menos   : 1  orden AB
  menos holgura: 0  orden BA
  plazo antes  : 0  orden BA
Contra menos holgura:
  tareas (duración, plazo): A(1, 2) B(10, 10)
  óptimo por fuerza bruta: 1
  dura menos   : 1  orden AB
  menos holgura: 9  orden BA
  plazo antes  : 1  orden AB
```

Dos reglas que fallan

terminal

```
$ build/lateness_rules
Contra dura menos:
  tareas (duración, plazo): A(1, 100) B(10, 10)
  óptimo por fuerza bruta: 0
  dura menos   : 1  orden AB
  menos holgura: 0  orden BA
  plazo antes  : 0  orden BA
Contra menos holgura:
  tareas (duración, plazo): A(1, 2) B(10, 10)
  óptimo por fuerza bruta: 1
  dura menos   : 1  orden AB
  menos holgura: 9  orden BA
  plazo antes  : 1  orden AB
```

- **Dura menos:** la duración no dice nada sobre la urgencia.

Dos reglas que fallan

terminal

```
$ build/lateness_rules
Contra dura menos:
  tareas (duración, plazo): A(1, 100) B(10, 10)
  óptimo por fuerza bruta: 0
  dura menos   : 1  orden AB
  menos holgura: 0  orden BA
  plazo antes  : 0  orden BA
Contra menos holgura:
  tareas (duración, plazo): A(1, 2) B(10, 10)
  óptimo por fuerza bruta: 1
  dura menos   : 1  orden AB
  menos holgura: 9  orden BA
  plazo antes  : 1  orden AB
```

- **Dura menos:** la duración no dice nada sobre la urgencia.
- **Menos holgura** ($d_i - t_i$): pone a B primero, y A termina en 11 con plazo 2.

La regla correcta: el plazo más cercano

pseudocódigo

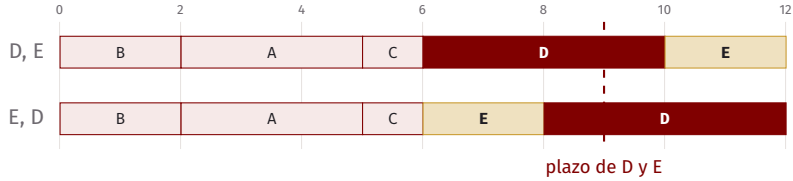
```
algoritmo plazo_mas_cercano(tareas, n)
    ordenar tareas por plazo
    ahora ← 0
    maximo ← -∞
    para i desde 0 hasta n - 1 hacer
        ahora ← ahora + tareas[i].duracion
        maximo ← max(maximo,
                     ahora - tareas[i].plazo
      )
    fin para
    devolver maximo
fin algoritmo
```

terminal

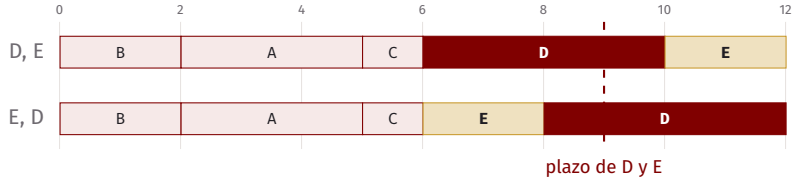
```
$ build/lateness_trace
tarea  duración  plazo  fin  retraso
  B          2      4    2      -2
  A          3      5    5       0
  C          1      7    6      -1
  D          4      9   10       1
  E          2      9   12       3
retraso máximo: 3
```

Costo: $O(n \log n)$. No mira las duraciones.

Con empates, el máximo no se mueve

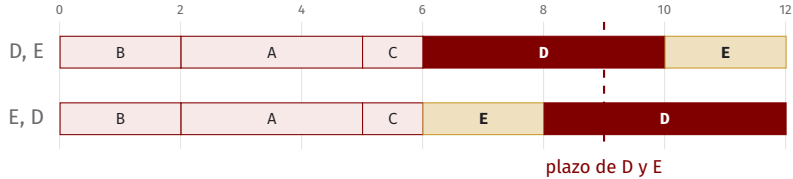


Con empates, el máximo no se mueve



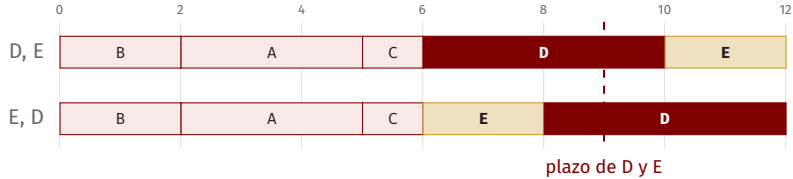
- Los retrasos individuales cambian: D pasa de 1 a 3, E de 3 a -1.

Con empates, el máximo no se mueve



- Los retrasos individuales cambian: D pasa de 1 a 3, E de 3 a -1.
- Pero las dos ocupan juntas el tramo de 6 a 12, y la última termina en 12 con plazo 9.

Con empates, el máximo no se mueve



- Los retrasos individuales cambian: D pasa de 1 a 3, E de 3 a -1.
- Pero las dos ocupan juntas el tramo de 6 a 12, y la última termina en 12 con plazo 9.
- El retraso máximo es 3 en ambos. Esto va a ser el Lema 3.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

- Dos tareas con el mismo plazo no forman una inversión.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

- Dos tareas con el mismo plazo no forman una inversión.
- El plazo más cercano no tiene inversiones, sea cual sea el desempate.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

- Dos tareas con el mismo plazo no forman una inversión.
- El plazo más cercano no tiene inversiones, sea cual sea el desempate.

Teorema. El horario del plazo más cercano tiene el menor retraso máximo posible.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

- Dos tareas con el mismo plazo no forman una inversión.
- El plazo más cercano no tiene inversiones, sea cual sea el desempate.

Teorema. El horario del plazo más cercano tiene el menor retraso máximo posible.

Técnica. Intercambio: quitar inversiones de un horario óptimo sin empeorarlo.

Inversiones, otra vez

Inversión de un horario

Un par (i, j) con i antes que j, en cualquier posición, y $d_i > d_j$, **estrictamente**.

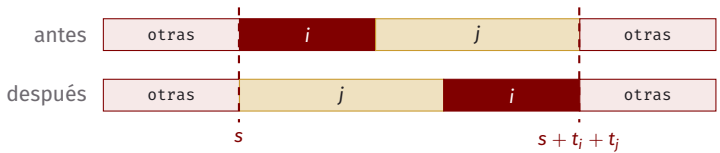
- Dos tareas con el mismo plazo no forman una inversión.
- El plazo más cercano no tiene inversiones, sea cual sea el desempate.

Teorema. El horario del plazo más cercano tiene el menor retraso máximo posible.

Técnica. Intercambio: quitar inversiones de un horario óptimo sin empeorarlo.

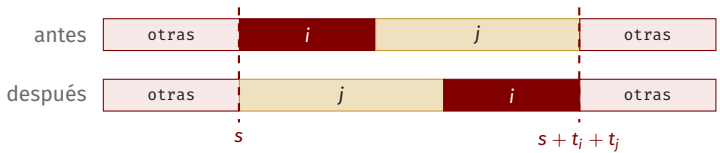
Lema 1. Si hay una inversión, hay una entre vecinas. Si ningún par de vecinas está invertido, los plazos no bajan nunca, y entonces ningún par está invertido. ■

Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

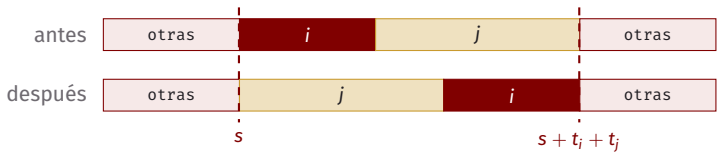
Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

- **Las demás no cambian:** las dos ocupan el mismo tramo. Aquí se usa que son vecinas.

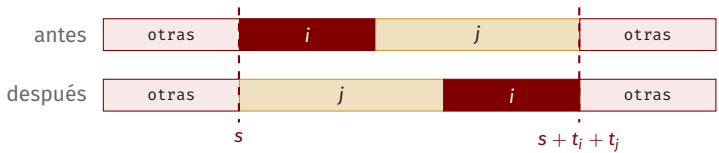
Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

- **Las demás no cambian:** las dos ocupan el mismo tramo. Aquí se usa que son vecinas.
- j **no empeora:** termina antes.

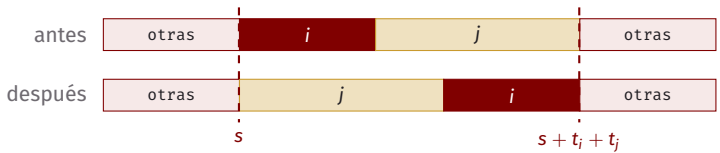
Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

- **Las demás no cambian:** las dos ocupan el mismo tramo. Aquí se usa que son vecinas.
- j **no empeora:** termina antes.
- i **empeora, pero poco:** $L'_i = f_j - d_i < f_j - d_j = L_j$.

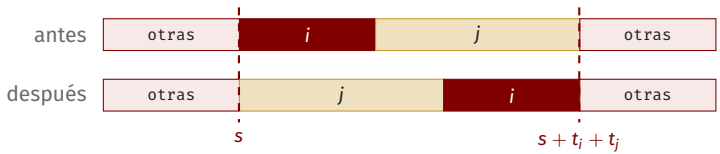
Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

- **Las demás no cambian:** las dos ocupan el mismo tramo. Aquí se usa que son vecinas.
- j **no empeora:** termina antes.
- i **empeora, pero poco:** $L_i' = f_j - d_i < f_j - d_j = L_j$.
- Los dos nuevos retrasos son a lo sumo L_j: el máximo no sube.

Lema 2: cambiar dos vecinas



Sean i, j vecinas, con i antes y $d_i > d_j$.

- **Las demás no cambian:** las dos ocupan el mismo tramo. Aquí se usa que son vecinas.
- j **no empeora:** termina antes.
- i **empeora, pero poco:** $L_i' = f_j - d_i < f_j - d_j = L_j$.
- Los dos nuevos retrasos son a lo sumo L_j: el máximo no sube.
- Solo cambia de orden el par (i, j): hay exactamente una inversión menos. ■

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.
- Cada bloque ocupa siempre el mismo tramo, de a a b. Su mayor retraso es el de la última: $b - d$, en cualquier orden. ■

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.
- Cada bloque ocupa siempre el mismo tramo, de a a b. Su mayor retraso es el de la última: $b - d$, en cualquier orden. ■

Cierre. Sea O óptimo, sin tiempo muerto.

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.
- Cada bloque ocupa siempre el mismo tramo, de a a b. Su mayor retraso es el de la última: $b - d$, en cualquier orden. ■

Cierre. Sea O óptimo, sin tiempo muerto.

- Mientras tenga inversiones, se cambia un par de vecinas invertidas (Lemas 1 y 2).

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.
- Cada bloque ocupa siempre el mismo tramo, de a a b. Su mayor retraso es el de la última: $b - d$, en cualquier orden. ■

Cierre. Sea O óptimo, sin tiempo muerto.

- Mientras tenga inversiones, se cambia un par de vecinas invertidas (Lemas 1 y 2).
- Hay a lo sumo $n(n-1)/2$ inversiones: termina en un O^* sin inversiones, no peor que O.

Lema 3 y cierre

Lema 3. Todos los horarios sin inversiones tienen el mismo retraso máximo.

- Las tareas de un mismo plazo están seguidas: una de otro plazo c en el medio formaría una inversión con la primera ($c < d$) o con la segunda ($c > d$).
- Entonces solo difieren en el orden *dentro* de cada bloque.
- Cada bloque ocupa siempre el mismo tramo, de a a b. Su mayor retraso es el de la última: $b - d$, en cualquier orden. ■

Cierre. Sea O óptimo, sin tiempo muerto.

- Mientras tenga inversiones, se cambia un par de vecinas invertidas (Lemas 1 y 2).
- Hay a lo sumo $n(n-1)/2$ inversiones: termina en un O^* sin inversiones, no peor que O.
- Por el Lema 3, O^* vale lo mismo que el plazo más cercano. ■

Los cambios, en un horario malo

terminal

orden	inversiones	retraso máximo	se cambia
DCEAB	8	8	DC
CDEAB	7	8	EA
CDAEB	6	8	DA
CADEB	5	8	CA
ACDEB	4	8	EB
ACDBE	3	6	DB
ACBDE	2	3	CB
ABCDE	1	3	AB
BACDE	0	3	ninguna

Los cambios, en un horario malo

terminal

orden	inversiones	retraso máximo	se cambia
DCEAB	8	8	DC
CDEAB	7	8	EA
CDAEB	6	8	DA
CADEB	5	8	CA
ACDEB	4	8	EB
ACDBE	3	6	DB
ACBDE	2	3	CB
ABCDE	1	3	AB
BACDE	0	3	ninguna

- Las inversiones bajan de a una, como en la semana 2.

Los cambios, en un horario malo

terminal

orden	inversiones	retraso máximo	se cambia
DCEAB	8	8	DC
CDEAB	7	8	EA
CDAEB	6	8	DA
CADEB	5	8	CA
ACDEB	4	8	EB
ACDBE	3	6	DB
ACBDE	2	3	CB
ABCDE	1	3	AB
BACDE	0	3	ninguna

- Las inversiones bajan de a una, como en la semana 2.
- El máximo nunca sube, y a veces se queda igual.

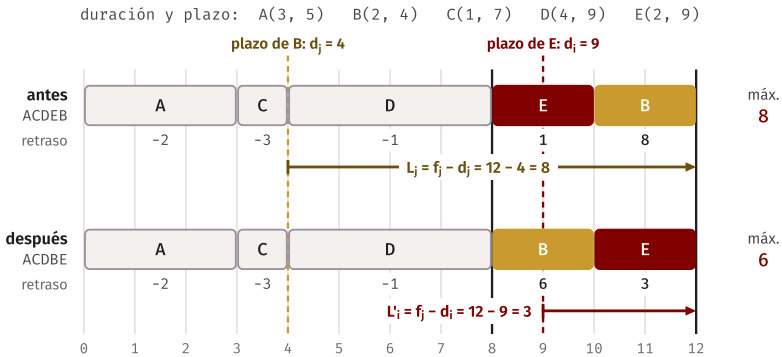
Los cambios, en un horario malo

terminal

orden	inversiones	retraso máximo	se cambia
DCEAB	8	8	DC
CDEAB	7	8	EA
CDAEB	6	8	DA
CADEB	5	8	CA
ACDEB	4	8	EB
ACDBE	3	6	DB
ACBDE	2	3	CB
ABCDE	1	3	AB
BACDE	0	3	ninguna

- Las inversiones bajan de a una, como en la semana 2.
- El máximo nunca sube, y a veces se queda igual.
- D y E nunca se cambian: no forman inversión.

Lema 2, en una fila de la tabla



Se cambian i = E y j = B, vecinas, con $d_i > d_j$: una fila de la tabla de cambios.
Juntas ocupan el mismo tramo, de 8 a 12, así que A, C y D no se enteran.
B termina antes. E termina donde terminaba B, con un plazo mayor: $L'_i = 3 < 8 = L_j$.

Error común: cambiar dos que no son vecinas

terminal

```
X(1, 5) M(1, 2) Y(5, 4)
orden XMY: retraso máximo 3
orden YMX: retraso máximo 4
```

Las del medio se corren

(X, Y) está invertido, y cambiarlo sube el máximo: M se fue más tarde. Por eso el Lema 1 busca una inversión entre vecinas. Las lejanas se quitan con varios cambios de vecinas.

SECCIÓN 02

Huffman: qué hay que demostrar

02

El enunciado, con cuidado

Ya lo programaron en Estructuras de Datos. Falta demostrarlo.

Costo de un árbol

Símbolos en las hojas: $\text{costo}(T) = \sum_x f(x) \cdot d(x)$, con $d(x)$ la profundidad.

El enunciado, con cuidado

Ya lo programaron en Estructuras de Datos. Falta demostrarlo.

Costo de un árbol

Símbolos en las hojas: $\text{costo}(T) = \sum_x f(x) \cdot d(x)$, con $d(x)$ la profundidad.

- $n \geq 2$ símbolos, con **frecuencia positiva**.

El enunciado, con cuidado

Ya lo programaron en Estructuras de Datos. Falta demostrarlo.

Costo de un árbol

Símbolos en las hojas: $\text{costo}(T) = \sum_x f(x) \cdot d(x)$, con $d(x)$ la profundidad.

- $n \geq 2$ símbolos, con **frecuencia positiva**.
- Se compara contra **todos** los árboles, también los que tienen nodos de un solo hijo.

El enunciado, con cuidado

Ya lo programaron en Estructuras de Datos. Falta demostrarlo.

Costo de un árbol

Símbolos en las hojas: $\text{costo}(T) = \sum_x f(x) \cdot d(x)$, con $d(x)$ la profundidad.

- $n \geq 2$ símbolos, con **frecuencia positiva**.
- Se compara contra **todos** los árboles, también los que tienen nodos de un solo hijo.
- Los **empates** pueden dar árboles distintos.

El enunciado, con cuidado

Ya lo programaron en Estructuras de Datos. Falta demostrarlo.

Costo de un árbol

Símbolos en las hojas: $\text{costo}(T) = \sum_x f(x) \cdot d(x)$, con $d(x)$ la profundidad.

- $n \geq 2$ símbolos, con **frecuencia positiva**.
- Se compara contra **todos** los árboles, también los que tienen nodos de un solo hijo.
- Los **empates** pueden dar árboles distintos.

Teorema

Con cualquier forma de romper los empates, unir los dos árboles de menor peso da un árbol de costo mínimo.

Ideas parecidas que fallan

terminal

```
$ build/huffman_rules
frecuencias: 15 7 6 6 5
  mejor árbol, por búsqueda exhaustiva: 87
  dividir parejo: 89  profundidades 2 2 2 3 3
  en cadena     : 91  profundidades 1 2 3 4 4
  Huffman       : 87  profundidades 1 3 3 3 3
```

Ideas parecidas que fallan

terminal

```
$ build/huffman_rules
frecuencias: 15 7 6 6 5
  mejor árbol, por búsqueda exhaustiva: 87
  dividir parejo: 89  profundidades 2 2 2 3 3
  en cadena     : 91  profundidades 1 2 3 4 4
  Huffman       : 87  profundidades 1 3 3 3 3
```

- **Dividir parejo:** ordenar de mayor a menor y partir la lista donde las sumas queden más parejas.

Ideas parecidas que fallan

terminal

```
$ build/huffman_rules
frecuencias: 15 7 6 6 5
  mejor árbol, por búsqueda exhaustiva: 87
  dividir parejo: 89  profundidades 2 2 2 3 3
  en cadena     : 91  profundidades 1 2 3 4 4
  Huffman       : 87  profundidades 1 3 3 3 3
```

- **Dividir parejo:** ordenar de mayor a menor y partir la lista donde las sumas queden más parejas.
- **En cadena:** 0, 10, 110, …, con los más frecuentes arriba.

Ideas parecidas que fallan

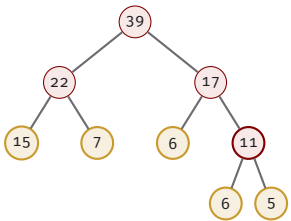
terminal

```
$ build/huffman_rules
frecuencias: 15 7 6 6 5
  mejor árbol, por búsqueda exhaustiva: 87
  dividir parejo: 89  profundidades 2 2 2 3 3
  en cadena     : 91  profundidades 1 2 3 4 4
  Huffman       : 87  profundidades 1 3 3 3 3
```

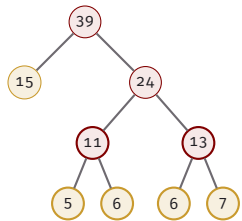
- **Dividir parejo:** ordenar de mayor a menor y partir la lista donde las sumas queden más parejas.
- **En cadena:** 0, 10, 110, …, con los más frecuentes arriba.
- Dividir parejo pierde por poco, y por eso cuesta detectarlo con ejemplos.

Decidir arriba o decidir abajo

dividir parejo: 89



Huffman: 87



Con 15, 7, 6, 6, 5, el primer corte es {15, 7} contra {6, 6, 5}, y baja el 15 a profundidad 2. El mejor árbol lo deja a profundidad 1, y paga con los demás. Dividir parejo decide primero lo de arriba, y Huffman decide primero lo de abajo.

Los empates: árboles distintos, mismo costo

terminal

```
Todas las formas de romper los empates:
frecuencias: 1 1 2 2
  profundidades 2 2 2 2  costo 12
  profundidades 3 3 1 2  costo 12
  profundidades 3 3 2 1  costo 12
frecuencias: 1 1 2 3 5 8
  profundidades 5 5 4 3 2 1  costo 45
```

Los empates: árboles distintos, mismo costo

terminal

```
Todas las formas de romper los empates:
frecuencias: 1 1 2 2
  profundidades 2 2 2 2  costo 12
  profundidades 3 3 1 2  costo 12
  profundidades 3 3 2 1  costo 12
frecuencias: 1 1 2 3 5 8
  profundidades 5 5 4 3 2 1  costo 45
```

- Con 1, 1, 2, 2: tras unir los 1 quedan tres árboles de peso 2, y se pueden unir dos cualesquiera.

Los empates: árboles distintos, mismo costo

terminal

```
Todas las formas de romper los empates:
frecuencias: 1 1 2 2
  profundidades 2 2 2 2  costo 12
  profundidades 3 3 1 2  costo 12
  profundidades 3 3 2 1  costo 12
frecuencias: 1 1 2 3 5 8
  profundidades 5 5 4 3 2 1  costo 45
```

- Con 1, 1, 2, 2: tras unir los 1 quedan tres árboles de peso 2, y se pueden unir dos cualesquiera.
- Con 1, 1, 2, 3, 5, 8 los pesos se repiten, pero solo cambia cuál queda a la izquierda.

Los empates: árboles distintos, mismo costo

terminal

```
Todas las formas de romper los empates:
frecuencias: 1 1 2 2
  profundidades 2 2 2 2  costo 12
  profundidades 3 3 1 2  costo 12
  profundidades 3 3 2 1  costo 12
frecuencias: 1 1 2 3 5 8
  profundidades 5 5 4 3 2 1  costo 45
```

- Con 1, 1, 2, 2: tras unir los 1 quedan tres árboles de peso 2, y se pueden unir dos cualesquiera.
- Con 1, 1, 2, 3, 5, 8 los pesos se repiten, pero solo cambia cuál queda a la izquierda.
- Por eso se dice “*un* árbol óptimo”.

SECCIÓN 03

Huffman: la demostración

03

El plan

x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

El plan

x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

1. **Lema A.** Un árbol óptimo es *lleno*: todo nodo interno tiene dos hijos.

El plan

x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

1. **Lema A.** Un árbol óptimo es *lleno*: todo nodo interno tiene dos hijos.
2. **Lema B.** Existe un árbol óptimo con x, y hermanos en el nivel más profundo. Intercambio.

El plan

x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

1. **Lema A.** Un árbol óptimo es *lleno*: todo nodo interno tiene dos hijos.
2. **Lema B.** Existe un árbol óptimo con x, y hermanos en el nivel más profundo. Intercambio.
3. **Lema C.** Reemplazar x, y por un símbolo z cambia el costo en exactamente $f(x) + f(y)$.

El plan

x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

1. **Lema A.** Un árbol óptimo es *lleno*: todo nodo interno tiene dos hijos.
2. **Lema B.** Existe un árbol óptimo con x, y hermanos en el nivel más profundo. Intercambio.
3. **Lema C.** Reemplazar x, y por un símbolo z cambia el costo en exactamente $f(x) + f(y)$.
4. **Teorema.** Inducción sobre el número de símbolos.

El plan

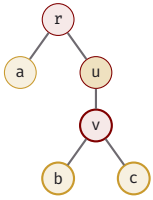
x, y: un par que Huffman podría unir primero. Es decir, $f(x) \leq f(y) \leq f(z)$ para todo otro z.

1. **Lema A.** Un árbol óptimo es *lleno*: todo nodo interno tiene dos hijos.
2. **Lema B.** Existe un árbol óptimo con x, y hermanos en el nivel más profundo. Intercambio.
3. **Lema C.** Reemplazar x, y por un símbolo z cambia el costo en exactamente $f(x) + f(y)$.
4. **Teorema.** Inducción sobre el número de símbolos.

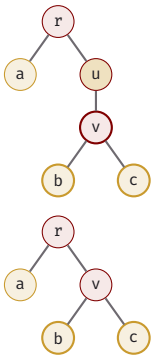
Error común

Creer que el intercambio ya termina la demostración. El Lema B habla solo de la primera unión.

Lema A: un óptimo no tiene nodos de un solo hijo

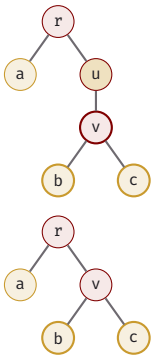


Lema A: un óptimo no tiene nodos de un solo hijo



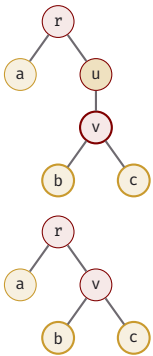
1. Si u tiene un solo hijo v, se quita u y se pone v en su lugar.

Lema A: un óptimo no tiene nodos de un solo hijo



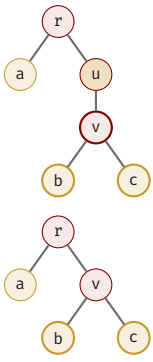
1. Si *u* tiene un solo hijo *v*, se quita *u* y se pone *v* en su lugar.
2. Las hojas debajo de *v* suben un nivel, y las demás no cambian.

Lema A: un óptimo no tiene nodos de un solo hijo



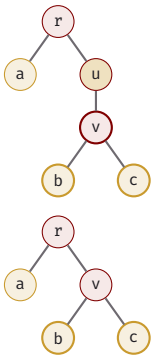
1. Si u tiene un solo hijo v, se quita u y se pone v en su lugar.
2. Las hojas debajo de v suben un nivel, y las demás no cambian.
3. Debajo de v hay una hoja con $f > 0$: el costo baja estrictamente.

Lema A: un óptimo no tiene nodos de un solo hijo



1. Si u tiene un solo hijo v, se quita u y se pone v en su lugar.
2. Las hojas debajo de v suben un nivel, y las demás no cambian.
3. Debajo de v hay una hoja con $f > 0$: el costo baja estrictamente.
4. Contradicción. ■

Lema A: un óptimo no tiene nodos de un solo hijo



1. Si u tiene un solo hijo v, se quita u y se pone v en su lugar.
2. Las hojas debajo de v suben un nivel, y las demás no cambian.
3. Debajo de v hay una hoja con $f > 0$: el costo baja estrictamente.
4. Contradicción. ■

Consecuencia. En un árbol lleno, una hoja de profundidad máxima tiene un hermano, y ese hermano también es hoja: si tuviera hijos, habría algo más profundo.

La cuenta del intercambio

Cambiar de lugar dos símbolos p y q altera solo dos términos de la suma:

La cuenta del intercambio

Cambiar de lugar dos símbolos p y q altera solo dos términos de la suma:

$$\text{nuevo} - \text{viejo} = f(p)d(q) + f(q)d(p) - f(p)d(p) - f(q)d(q)$$

La cuenta del intercambio

Cambiar de lugar dos símbolos p y q altera solo dos términos de la suma:

$$\begin{aligned}\text{nuevo} - \text{viejo} &= f(p)d(q) + f(q)d(p) - f(p)d(p) - f(q)d(q) \\ &= (f(p) - f(q)) \cdot (d(q) - d(p))\end{aligned}$$

La cuenta del intercambio

Cambiar de lugar dos símbolos p y q altera solo dos términos de la suma:

$$\begin{aligned} \text{nuevo} - \text{viejo} &= f(p)d(q) + f(q)d(p) - f(p)d(p) - f(q)d(q) \\ &= (f(p) - f(q)) \cdot (d(q) - d(p)) \end{aligned}$$

Si $f(p) \leq f(q)$ y $d(q) \geq d(p)$, el primer factor es ≤ 0 y el segundo es ≥ 0.

La cuenta del intercambio

Cambiar de lugar dos símbolos p y q altera solo dos términos de la suma:

$$\begin{aligned}\text{nuevo} - \text{viejo} &= f(p)d(q) + f(q)d(p) - f(p)d(p) - f(q)d(q) \\ &= (f(p) - f(q)) \cdot (d(q) - d(p))\end{aligned}$$

Si $f(p) \leq f(q)$ y $d(q) \geq d(p)$, el primer factor es ≤ 0 y el segundo es ≥ 0.

Mandar abajo lo menos frecuente nunca empeora

Esta es la única cuenta del Lema B. Lo demás es cuidar los casos.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.
- $f(x) \leq f(a)$ y $D \geq d(x)$: el costo no sube. y no se movió.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.
- $f(x) \leq f(a)$ y $D \geq d(x)$: el costo no sube. y no se movió.

Poner a y. Sea b el otro símbolo del par.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.
- $f(x) \leq f(a)$ y $D \geq d(x)$: el costo no sube. y no se movió.

Poner a y. Sea b el otro símbolo del par.

- Si $b = y$, listo.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.
- $f(x) \leq f(a)$ y $D \geq d(x)$: el costo no sube. y no se movió.

Poner a y. Sea b el otro símbolo del par.

- Si $b = y$, listo.
- Si no, $b \neq x$, así que $f(y) \leq f(b)$. Y $D \geq d(y)$: el costo no sube. x no se movió.

Lema B: x, y hermanos abajo

Sea T óptimo, lleno, con profundidad máxima D. Hay dos hojas hermanas a profundidad D.

Poner a x.

- Si x ya está en el par, nada.
- Si no, alguna hoja a del par *no es* y. Se cambia x con a.
- $f(x) \leq f(a)$ y $D \geq d(x)$: el costo no sube. y no se movió.

Poner a y. Sea b el otro símbolo del par.

- Si $b = y$, listo.
- Si no, $b \neq x$, así que $f(y) \leq f(b)$. Y $D \geq d(y)$: el costo no sube. x no se movió.

La forma del árbol no cambió, y T era óptimo: el resultado también es óptimo. ■

El intercambio no construye el óptimo

terminal

```
$ build/huffman_trace
Intercambio, con frecuencias 1 2 3 4:
  árbol inicial:    profundidades 1 2 3 3  costo 26
  cambia 1 con 3:   profundidades 3 2 1 3  costo 22
  cambia 2 con 4:   profundidades 3 3 1 2  costo 20
  Huffman:          profundidades 3 3 2 1  costo 19
```

El intercambio no construye el óptimo

terminal

```
$ build/huffman_trace
Intercambio, con frecuencias 1 2 3 4:
  árbol inicial:    profundidades 1 2 3 3  costo 26
  cambia 1 con 3:   profundidades 3 2 1 3  costo 22
  cambia 2 con 4:   profundidades 3 3 1 2  costo 20
  Huffman:          profundidades 3 3 2 1  costo 19
```

- Los cambios no subieron el costo, y el 1 y el 2 quedaron abajo.

El intercambio no construye el óptimo

terminal

```
$ build/huffman_trace
Intercambio, con frecuencias 1 2 3 4:
  árbol inicial:    profundidades 1 2 3 3  costo 26
  cambia 1 con 3:   profundidades 3 2 1 3  costo 22
  cambia 2 con 4:   profundidades 3 3 1 2  costo 20
  Huffman:          profundidades 3 3 2 1  costo 19
```

- Los cambios no subieron el costo, y el 1 y el 2 quedaron abajo.
- Pero quedó en 20, y el mejor es 19: el 3 y el 4 también estaban mal puestos.

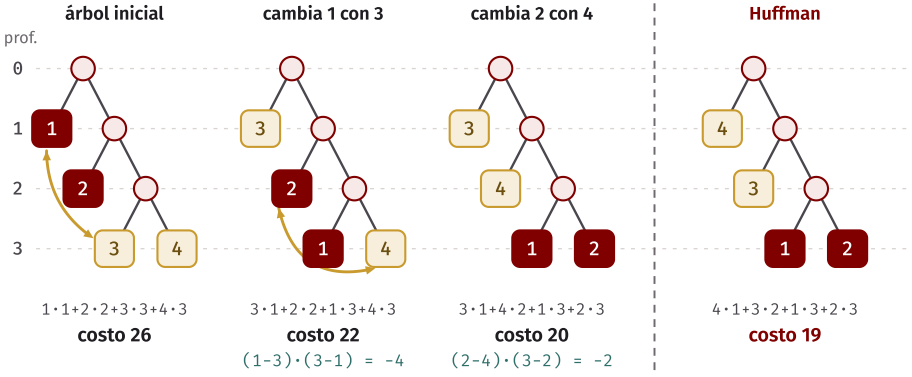
El intercambio no construye el óptimo

terminal

```
$ build/huffman_trace
Intercambio, con frecuencias 1 2 3 4:
  árbol inicial:    profundidades 1 2 3 3  costo 26
  cambia 1 con 3:   profundidades 3 2 1 3  costo 22
  cambia 2 con 4:   profundidades 3 3 1 2  costo 20
  Huffman:          profundidades 3 3 2 1  costo 19
```

- Los cambios no subieron el costo, y el 1 y el 2 quedaron abajo.
- Pero quedó en 20, y el mejor es 19: el 3 y el 4 también estaban mal puestos.
- El Lema B solo dice dónde pueden ir x e y en un árbol que ya era óptimo.

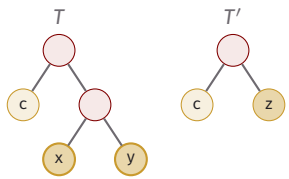
Los mismos cambios, en el árbol



En granate, x = 1 e y = 2: cada cambio las manda más abajo, y el costo nunca sube. La forma no cambia.

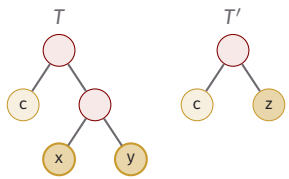
Pero el árbol inicial no era óptimo, y quedar en 20 no es llegar a 19: el 3 y el 4 también estaban mal puestos.

Lema C: reemplazar dos símbolos por uno



C': se quitan x, y y se agrega z, con $f(z) = f(x) + f(y)$.

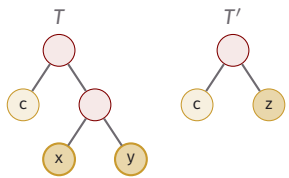
Lema C: reemplazar dos símbolos por uno



C': se quitan x, y y se agrega z, con $f(z) = f(x) + f(y)$.

Si z está a profundidad h en T', entonces x, y están a $h + 1$ en T. Los demás no cambian.

Lema C: reemplazar dos símbolos por uno

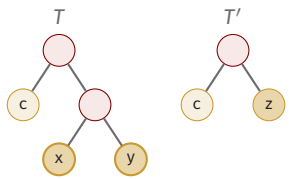


C': se quitan x, y y se agrega z, con $f(z) = f(x) + f(y)$.

Si z está a profundidad h en T', entonces x, y están a $h+1$ en T. Los demás no cambian.

$$f(x)(h+1) + f(y)(h+1) = f(z)\,h + f(x) + f(y)$$

Lema C: reemplazar dos símbolos por uno



C': se quitan x, y y se agrega z, con $f(z) = f(x) + f(y)$.

Si z está a profundidad h en T', entonces x, y están a $h+1$ en T. Los demás no cambian.

$$f(x)(h+1) + f(y)(h+1) = f(z)\,h + f(x) + f(y)$$

La relación

$$\text{costo}(T) = \text{costo}(T') + f(x) + f(y)$$

Vale **en los dos sentidos**: al expandir z en un árbol de C', y al contraer x, y hermanos en un árbol de C.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n - 1$ símbolos y toda forma de romper empates.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n=2$. Costo $f(x)+f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n=2$**.** Costo $f(x)+f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n = 2$. Costo $f(x) + f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n = 2$**.** Costo $f(x) + f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.
2. $\text{costo}(H) = \text{costo}(H') + f(x) + f(y)$. Lema C, expandiendo.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n = 2$**.** Costo $f(x) + f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.
2. $\text{costo}(H) = \text{costo}(H') + f(x) + f(y)$. Lema C, expandiendo.
3. Por el Lema B, hay un óptimo T^* con x, y hermanos.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n = 2$**.** Costo $f(x) + f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.
2. $\text{costo}(H) = \text{costo}(H') + f(x) + f(y)$. Lema C, expandiendo.
3. Por el Lema B, hay un óptimo T^* con x, y hermanos.
4. $\text{costo}(T^*) = \text{costo}(T^{*\prime}) + f(x) + f(y)$. Lema C, contrayendo.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n=2$**.** Costo $f(x)+f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.
2. $\text{costo}(H) = \text{costo}(H') + f(x) + f(y)$. Lema C, expandiendo.
3. Por el Lema B, hay un óptimo T^* con x, y hermanos.
4. $\text{costo}(T^*) = \text{costo}(T^{*\prime}) + f(x) + f(y)$. Lema C, contrayendo.
5. H' es óptimo: $\text{costo}(H') \leq \text{costo}(T^{*\prime})$.

El teorema: la inducción

Hipótesis. Vale para todo conjunto de $n-1$ símbolos y toda forma de romper empates.

Base, $n=2$**.** Costo $f(x)+f(y)$. Ningún símbolo puede estar en la raíz, así que los dos están a profundidad ≥ 1.

Paso. Sea x, y la primera unión, y H el árbol final.

1. Lo que sigue es una corrida de Huffman sobre C': da H', óptimo por hipótesis.
2. $\text{costo}(H) = \text{costo}(H') + f(x) + f(y)$. Lema C, expandiendo.
3. Por el Lema B, hay un óptimo T^* con x, y hermanos.
4. $\text{costo}(T^*) = \text{costo}(T^{*\prime}) + f(x) + f(y)$. Lema C, contrayendo.
5. H' es óptimo: $\text{costo}(H') \leq \text{costo}(T^{*\prime})$.
6. Se suma a los dos lados: $\text{costo}(H) \leq \text{costo}(T^*)$. ■

La inducción, con números

terminal

```
alfabeto                x   y   costo = costo del siguiente + x + y
1 1 2 3 5 8             1   1     45 = 43 + 1 + 1
2 2 3 5 8               2   2     43 = 39 + 2 + 2
3 4 5 8                 3   4     39 = 32 + 3 + 4
5 7 8                   5   7     32 = 20 + 5 + 7
8 12                    8  12     20 = 0 + 8 + 12
```

La inducción, con números

terminal

```
alfabeto                x   y   costo = costo del siguiente + x + y
1 1 2 3 5 8             1   1     45 = 43 + 1 + 1
2 2 3 5 8               2   2     43 = 39 + 2 + 2
3 4 5 8                 3   4     39 = 32 + 3 + 4
5 7 8                   5   7     32 = 20 + 5 + 7
8 12                    8  12     20 = 0 + 8 + 12
```

- La última fila es el caso base.

La inducción, con números

terminal

```
alfabeto                x   y   costo = costo del siguiente + x + y
1 1 2 3 5 8             1   1     45 = 43 + 1 + 1
2 2 3 5 8               2   2     43 = 39 + 2 + 2
3 4 5 8                 3   4     39 = 32 + 3 + 4
5 7 8                   5   7     32 = 20 + 5 + 7
8 12                    8  12     20 = 0 + 8 + 12
```

- La última fila es el caso base.
- El costo es la suma de las uniones: $2 + 4 + 7 + 12 + 20 = 45$.

La inducción, con números

terminal

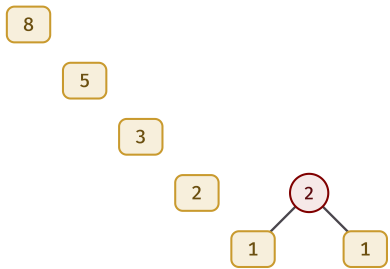
```
alfabeto                x   y   costo = costo del siguiente + x + y
1 1 2 3 5 8             1   1     45 = 43 + 1 + 1
2 2 3 5 8               2   2     43 = 39 + 2 + 2
3 4 5 8                 3   4     39 = 32 + 3 + 4
5 7 8                   5   7     32 = 20 + 5 + 7
8 12                    8  12     20 = 0 + 8 + 12
```

- La última fila es el caso base.
- El costo es la suma de las uniones: $2 + 4 + 7 + 12 + 20 = 45$.
- Tan desbalanceado, y es el óptimo: la búsqueda exhaustiva también da 45.

Las uniones, una por una

alfabeto **costo = siguiente + x + y**

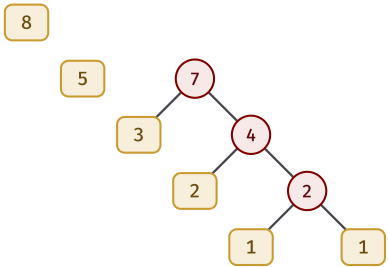
1 1 2 3 5 8 45 = 43 + 1 + 1



suma de las uniones:
2

Se unen 1 y 1, los dos menores: un árbol de peso 2.

Las uniones, una por una



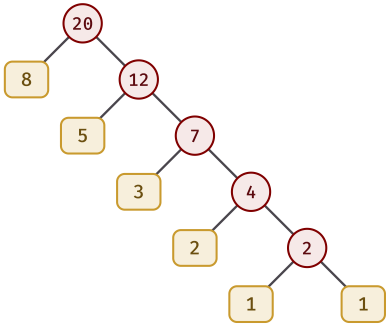
alfabeto	costo = siguiente + x + y
1 1 2 3 5 8	45 = 43 + 1 + 1
2 2 3 5 8	43 = 39 + 2 + 2
3 4 5 8	39 = 32 + 3 + 4

suma de las uniones:

2 + 4 + 7

Se unen 3 y 4: peso 7. El 4 ya es un árbol, y se trata como un símbolo.

Las uniones, una por una



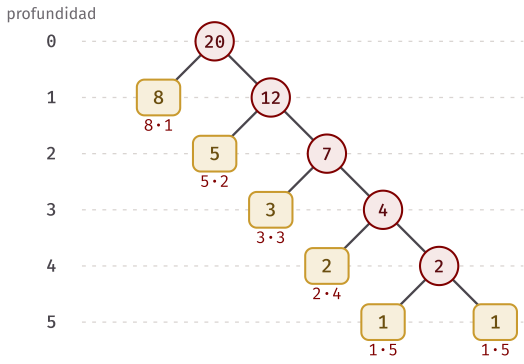
alfabeto	costo = siguiente + x + y
1 1 2 3 5 8	45 = 43 + 1 + 1
2 2 3 5 8	43 = 39 + 2 + 2
3 4 5 8	39 = 32 + 3 + 4
5 7 8	32 = 20 + 5 + 7
8 12	20 = 0 + 8 + 12

suma de las uniones:

2 + 4 + 7 + 12 + 20 = 45

Se unen 8 y 12: peso 20, y queda un solo árbol. Esta fila es el caso base.

Las uniones, una por una



alfabeto	costo = siguiente + x + y
1 1 2 3 5 8	45 = 43 + 1 + 1
2 2 3 5 8	43 = 39 + 2 + 2
3 4 5 8	39 = 32 + 3 + 4
5 7 8	32 = 20 + 5 + 7
8 12	20 = 0 + 8 + 12

suma de las uniones:

$2 + 4 + 7 + 12 + 20 = 45$

frecuencia · profundidad:

$1 \cdot 5 + 1 \cdot 5 + 2 \cdot 4 + 3 \cdot 3 + 5 \cdot 2 + 8 \cdot 1 = 45$

Las dos cuentas dan 45: cada unión baja un nivel a todas sus hojas, y eso agrega su peso.

SECCIÓN 04

Dijkstra: el invariante

04

Dijkstra como greedy

pseudocódigo

```
algoritmo dijkstra(grafo, s)
    d[v] ← ∞ para todo nodo v
    fijado[v] ← falso para todo nodo v
    d[s] ← 0
    meter (0, s) al heap
    mientras el heap no esté vacío hacer
        (x, u) ← sacar el menor del heap
        si no fijado[u] entonces
            fijado[u] ← verdadero
            relajar los arcos de u
        fin si
    fin mientras
    devolver d
fin algoritmo
```

relajar los arcos de u

```
para cada arco (u, v, w) de u hacer
    si d[u] + w < d[v] entonces
        d[v] ← d[u] + w
        meter (d[v], v) al heap
    fin si
fin para
```

Dijkstra como greedy

pseudocódigo

```
algoritmo dijkstra(grafo, s)
    d[v] ← ∞ para todo nodo v
    fijado[v] ← falso para todo nodo v
    d[s] ← 0
    meter (0, s) al heap
    mientras el heap no esté vacío hacer
        (x, u) ← sacar el menor del heap
        si no fijado[u] entonces
            fijado[u] ← verdadero
            relajar los arcos de u
        fin si
    fin mientras
    devolver d
fin algoritmo
```

relajar los arcos de u

```
para cada arco (u, v, w) de u hacer
    si d[u] + w < d[v] entonces
        d[v] ← d[u] + w
        meter (d[v], v) al heap
    fin si
fin para
```

- Decisión irrevocable: fijar para siempre el de menor d.

Dijkstra como greedy

pseudocódigo

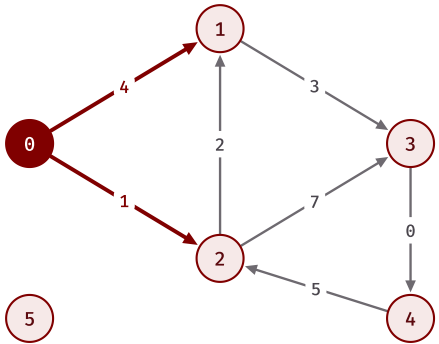
```
algoritmo dijkstra(grafo, s)
    d[v] ← ∞ para todo nodo v
    fijado[v] ← falso para todo nodo v
    d[s] ← 0
    meter (0, s) al heap
    mientras el heap no esté vacío hacer
        (x, u) ← sacar el menor del heap
        si no fijado[u] entonces
            fijado[u] ← verdadero
            relajar los arcos de u
        fin si
    fin mientras
    devolver d
fin algoritmo
```

relajar los arcos de u

```
para cada arco (u, v, w) de u hacer
    si d[u] + w < d[v] entonces
        d[v] ← d[u] + w
        meter (d[v], v) al heap
    fin si
fin para
```

- Decisión irrevocable: fijar para siempre el de menor d.
- Las entradas viejas se ignoran al salir: borrado perezoso.

Un ejemplo



d, después de relajar

0	1	2	3	4	5
0	4	1	∞	∞	∞

S = {0}

heap, de menor a mayor

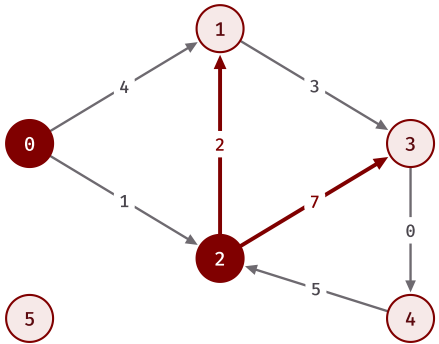
(1,2) (4,1)

sale (0,0)

fija al 0, con d[0] = 0 = δ(0)

Sale (0,0): fija al 0. Sus arcos dan d[1] = 4 y d[2] = 1.

Un ejemplo



d, después de relajar

0	1	2	3	4	5
0	3	1	8	∞	∞

S = {0, 2}

heap, de menor a mayor

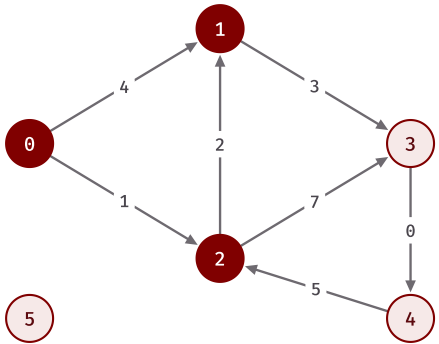
(3,1) (4,1) (8,3)

sale (1,2)

fija al 2, con d[2] = 1 = δ(2)

Sale (1,2): fija al 2. 1 + 2 < 4, y d[1] baja a 3. El arco de peso 7 da d[3] = 8.

Un ejemplo



d, después de relajar

0	1	2	3	4	5
0	3	1	6	∞	∞

S = {0, 2, 1}

heap, de menor a mayor

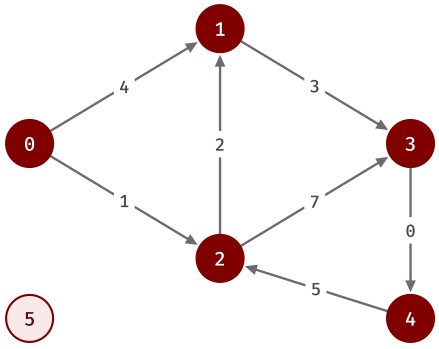
(6,3) (8,3)

`sale (4,1)`

vieja: su nodo ya está fijado

Sale (4,1): el 1 ya está fijado. Es una entrada vieja, y se ignora.

Un ejemplo



d, después de relajar

0	1	2	3	4	5
0	3	1	6	6	∞

`S = {0, 2, 1, 3, 4}`

heap, de menor a mayor

`vacío`

`sale (8,3)`

vieja: su nodo ya está fijado

Sale (8,3): vieja. El heap queda vacío, y el 5 sigue en ∞: no se alcanza.

La misma corrida, en la salida

terminal

```
$ build/dijkstra_trace
sale    qué pasa      d[0] d[1] d[2] d[3] d[4] d[5]  verdad
(0,0)   fija al 0       0    4    1  inf  inf  inf     0
(1,2)   fija al 2       0    3    1    8  inf  inf     1
(3,1)   fija al 1       0    3    1    6  inf  inf     3
(4,1)   vieja           0    3    1    6  inf  inf
(6,3)   fija al 3       0    3    1    6    6  inf     6
(6,4)   fija al 4       0    3    1    6    6  inf     6
(8,3)   vieja           0    3    1    6    6  inf
```

La misma corrida, en la salida

terminal

```
$ build/dijkstra_trace
sale    qué pasa      d[0] d[1] d[2] d[3] d[4] d[5]  verdad
(0,0)   fija al 0       0    4    1  inf  inf  inf     0
(1,2)   fija al 2       0    3    1    8  inf  inf     1
(3,1)   fija al 1       0    3    1    6  inf  inf     3
(4,1)   vieja           0    3    1    6  inf  inf
(6,3)   fija al 3       0    3    1    6    6  inf     6
(6,4)   fija al 4       0    3    1    6    6  inf     6
(8,3)   vieja           0    3    1    6    6  inf
```

- Al fijarse, cada nodo ya tiene la verdad.

La misma corrida, en la salida

terminal

```
$ build/dijkstra_trace
sale    qué pasa      d[0] d[1] d[2] d[3] d[4] d[5]  verdad
(0,0)   fija al 0       0    4    1  inf  inf  inf     0
(1,2)   fija al 2       0    3    1    8  inf  inf     1
(3,1)   fija al 1       0    3    1    6  inf  inf     3
(4,1)   vieja           0    3    1    6  inf  inf
(6,3)   fija al 3       0    3    1    6    6  inf     6
(6,4)   fija al 4       0    3    1    6    6  inf     6
(8,3)   vieja           0    3    1    6    6  inf
```

- Al fijarse, cada nodo ya tiene la verdad.
- El peso 0 no molesta.

La misma corrida, en la salida

terminal

```
$ build/dijkstra_trace
sale    qué pasa      d[0] d[1] d[2] d[3] d[4] d[5]  verdad
(0,0)   fija al 0       0    4    1  inf  inf  inf     0
(1,2)   fija al 2       0    3    1    8  inf  inf     1
(3,1)   fija al 1       0    3    1    6  inf  inf     3
(4,1)   vieja           0    3    1    6  inf  inf
(6,3)   fija al 3       0    3    1    6    6  inf     6
(6,4)   fija al 4       0    3    1    6    6  inf     6
(8,3)   vieja           0    3    1    6    6  inf
```

- Al fijarse, cada nodo ya tiene la verdad.
- El peso 0 no molesta.
- El 5 no se alcanza: se queda en ∞.

Ideas parecidas que fallan

terminal

```
$ build/dijkstra_wrong
0 -> 1 (4), 0 -> 2 (1), 2 -> 1 (2). Distancia a 1:
  fijar al descubrir: 4
  Dijkstra:           3
  árbitro:            3
0 -> 1 (1), 1 -> 3 (10), 0 -> 2 (2), 2 -> 3 (2). Distancia a 3:
  arco más barato:    11
  Dijkstra:           4
  árbitro:            4
```

Ideas parecidas que fallan

terminal

```
$ build/dijkstra_wrong
0 -> 1 (4), 0 -> 2 (1), 2 -> 1 (2). Distancia a 1:
  fijar al descubrir: 4
  Dijkstra:           3
  árbitro:            3
0 -> 1 (1), 1 -> 3 (10), 0 -> 2 (2), 2 -> 3 (2). Distancia a 3:
  arco más barato:    11
  Dijkstra:           4
  árbitro:            4
```

- **Fijar al descubrir**, como en BFS: el 1 se descubre por el arco de peso 4.

Ideas parecidas que fallan

terminal

```
$ build/dijkstra_wrong
0 -> 1 (4), 0 -> 2 (1), 2 -> 1 (2). Distancia a 1:
  fijar al descubrir: 4
  Dijkstra:           3
  árbitro:            3
0 -> 1 (1), 1 -> 3 (10), 0 -> 2 (2), 2 -> 3 (2). Distancia a 3:
  arco más barato:    11
  Dijkstra:           4
  árbitro:            4
```

- **Fijar al descubrir**, como en BFS: el 1 se descubre por el arco de peso 4.
- **Caminar por el arco más barato:** mira el arco siguiente, y no el total.

Ideas parecidas que fallan

terminal

```
$ build/dijkstra_wrong
0 -> 1 (4), 0 -> 2 (1), 2 -> 1 (2). Distancia a 1:
  fijar al descubrir: 4
  Dijkstra:           3
  árbitro:            3
0 -> 1 (1), 1 -> 3 (10), 0 -> 2 (2), 2 -> 3 (2). Distancia a 3:
  arco más barato:    11
  Dijkstra:           4
  árbitro:            4
```

- **Fijar al descubrir**, como en BFS: el 1 se descubre por el arco de peso 4.
- **Caminar por el arco más barato:** mira el arco siguiente, y no el total.
- Dijkstra compara los totales desde el origen de todo lo que tiene a la vista.

Dos hechos previos

Hecho 1: $d[v]$ es el largo de un camino real, o ∞

Cada asignación extiende un camino real por un arco. Entonces $d[v] \geq \delta(v)$ siempre, y $d[v]$ solo baja.

Dos hechos previos

Hecho 1: $d[v]$ es el largo de un camino real, o ∞

Cada asignación extiende un camino real por un arco. Entonces $d[v] \geq \delta(v)$ siempre, y $d[v]$ solo baja.

Hecho 2: la entrada que fija a u trae el menor d entre los no fijados

Todo no fijado v con $d[v]$ finito tiene en el heap su entrada $(d[v], v)$, y sus otras entradas traen valores mayores. Por eso, una entrada de un no fijado sale con su d actual, y es la menor del heap.

Dos hechos previos

Hecho 1: $d[v]$ es el largo de un camino real, o ∞

Cada asignación extiende un camino real por un arco. Entonces $d[v] \geq \delta(v)$ siempre, y $d[v]$ solo baja.

Hecho 2: la entrada que fija a u trae el menor d entre los no fijados

Todo no fijado v con $d[v]$ finito tiene en el heap su entrada $(d[v], v)$, y sus otras entradas traen valores mayores. Por eso, una entrada de un no fijado sale con su d actual, y es la menor del heap.

Las entradas viejas no estorban: son de nodos fijados, o salen después de la actual de su nodo.

El invariante

Teorema

Con pesos no negativos, cada vez que se fija un nodo u, $d[u] = \delta(u)$. Al final, los fijados son exactamente los alcanzables.

Técnica. Inducción sobre el número de nodos fijados. S: el conjunto de fijados.

El invariante

Teorema

Con pesos no negativos, cada vez que se fija un nodo u, $d[u] = \delta(u)$. Al final, los fijados son exactamente los alcanzables.

Técnica. Inducción sobre el número de nodos fijados. S: el conjunto de fijados.

Base. El primero es s, con $d[s] = 0$. Y $\delta(s) = 0$: ningún camino mide menos que 0.

El invariante

Teorema

Con pesos no negativos, cada vez que se fija un nodo u, $d[u] = \delta(u)$. Al final, los fijados son exactamente los alcanzables.

Técnica. Inducción sobre el número de nodos fijados. S: el conjunto de fijados.

Base. El primero es s, con $d[s] = 0$. Y $\delta(s) = 0$: ningún camino mide menos que 0.

Paso. El invariante vale para S, y el siguiente en fijarse es u. Tomamos cualquier camino P de s a u, y demostramos que mide al menos $d[u]$.

El invariante

> **Teorema**
> Con pesos no negativos, cada vez que se fija un nodo u, $d[u] = \delta(u)$. Al final, los fijados son exactamente los alcanzables.

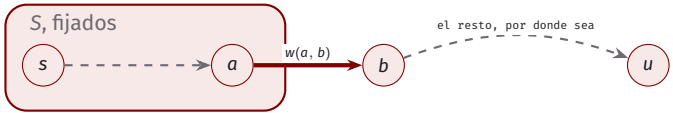
Técnica. Inducción sobre el número de nodos fijados. S: el conjunto de fijados.

Base. El primero es s, con $d[s] = 0$. Y $\delta(s) = 0$: ningún camino mide menos que 0.

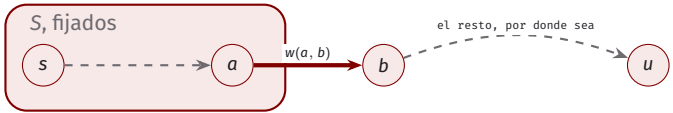
Paso. El invariante vale para S, y el siguiente en fijarse es u. Tomamos cualquier camino P de s a u, y demostramos que mide al menos $d[u]$.

P empieza en S y termina fuera: tiene un **primer arco** (a, b) que sale de S.

El paso, línea por línea

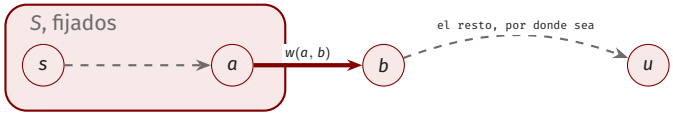


El paso, línea por línea



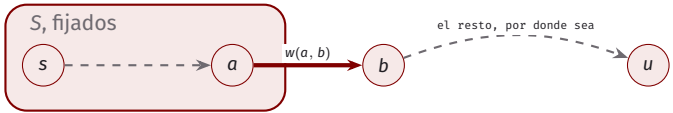
$\text{largo}(P) \quad \geq \text{largo}(P_1)$ el resto pesa ≥ 0 (1)

El paso, línea por línea



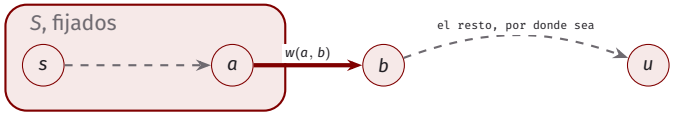
$$\begin{aligned} \text{largo}(P) &\geq \text{largo}(P_1) && \text{el resto pesa} \geq 0 \\ &\geq \delta(a) + w(a,b) && P_1 \text{ llega a } a \text{ y usa } (a,b) \end{aligned} \tag{1}$$

El paso, línea por línea



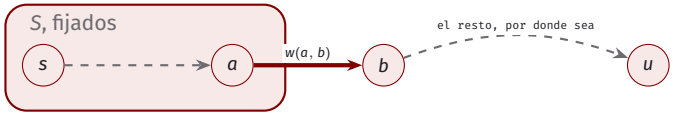
$$
\begin{aligned}
\operatorname{largo}(P) &\geq \operatorname{largo}(P_1) && \text{el resto pesa} \geq 0 && (1)\\
&\geq \delta(a) + w(a,b) && P_1 \text{ llega a } a \text{ y usa } (a,b) &&\\
&= d[a] + w(a,b) && a \in S\text{: invariante} && (2)
\end{aligned}
$$

El paso, línea por línea



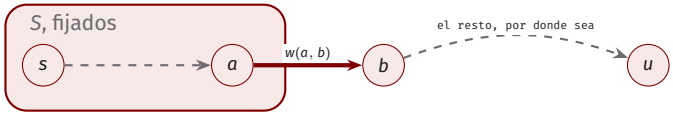
$$
\begin{aligned}
\operatorname{largo}(P) &\geq \operatorname{largo}(P_1) && \text{el resto pesa} \geq 0 && (1)\\
&\geq \delta(a) + w(a,b) && P_1 \text{ llega a } a \text{ y usa } (a,b) &&\\
&= d[a] + w(a,b) && a \in S\text{: invariante} && (2)\\
&\geq d[b] && a \text{ relajó } (a,b) \text{ al fijarse} && (3)
\end{aligned}
$$

El paso, línea por línea



$$
\begin{aligned}
\text{largo}(P) &\geq \text{largo}(P_1) && \text{el resto pesa} \geq 0 && (1)\\
&\geq \delta(a) + w(a,b) && P_1 \text{ llega a } a \text{ y usa } (a,b)\\
&= d[a] + w(a,b) && a \in S\text{: invariante} && (2)\\
&\geq d[b] && a \text{ relajó } (a,b) \text{ al fijarse} && (3)\\
&\geq d[u] && \text{el heap eligió a } u\text{: Hecho 2} && (4)
\end{aligned}
$$

El paso, línea por línea



$$
\begin{aligned}
\text{largo}(P) &\geq \text{largo}(P_1) && \text{el resto pesa} \geq 0 && (1)\\
&\geq \delta(a) + w(a,b) && P_1 \text{ llega a } a \text{ y usa } (a,b)\\
&= d[a] + w(a,b) && a \in S\text{: invariante} && (2)\\
&\geq d[b] && a \text{ relajó } (a,b) \text{ al fijarse} && (3)\\
&\geq d[u] && \text{el heap eligió a } u\text{: Hecho 2} && (4)
\end{aligned}
$$

Todo camino mide al menos $d[u]$: $\delta(u) \geq d[u]$. Con el Hecho 1, $d[u] = \delta(u)$. ■

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.
- Un camino de s a v tiene un primer arco (a, b) que sale de los fijados.

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.
- Un camino de s a v tiene un primer arco (a, b) que sale de los fijados.
- a relajó ese arco, así que $d[b]$ es finito.

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.
- Un camino de s a v tiene un primer arco (a, b) que sale de los fijados.
- a relajó ese arco, así que $d[b]$ es finito.
- Por el Hecho 2, b tiene una entrada en el heap. Pero el heap está vacío. Contradicción.

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.
- Un camino de s a v tiene un primer arco (a, b) que sale de los fijados.
- a relajó ese arco, así que $d[b]$ es finito.
- Por el Hecho 2, b tiene una entrada en el heap. Pero el heap está vacío. Contradicción.
- Los no alcanzables no se fijan: un d finito sería un camino (Hecho 1). ■

Quiénes quedan fijados

- Supongamos que al terminar hay un v alcanzable y no fijado.
- Un camino de s a v tiene un primer arco (a, b) que sale de los fijados.
- a relajó ese arco, así que $d[b]$ es finito.
- Por el Hecho 2, b tiene una entrada en el heap. Pero el heap está vacío. Contradicción.
- Los no alcanzables no se fijan: un d finito sería un camino (Hecho 1). ■

Un argumento sobre la frontera

Cualquier camino a u cruza de los fijados a los no fijados por algún lado, y en ese cruce ya mide tanto como lo que eligió el heap.

Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

- (2) es la hipótesis de inducción.

Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

- (2) es la hipótesis de inducción.
- (3) es lo que hace el código.

Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

- (2) es la hipótesis de inducción.
- (3) es lo que hace el código.
- (4) es la regla del heap.

Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

- (2) es la hipótesis de inducción.
- (3) es lo que hace el código.
- (4) es la regla del heap.
- Ninguna de ellas mira el signo de los pesos.

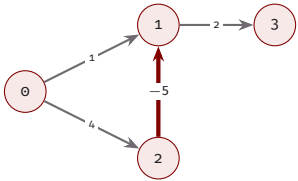
Dónde se usan los pesos no negativos

En la desigualdad (1)

$\text{largo}(P) \geq \text{largo}(P_1)$: lo que falta del camino, de b a u, no resta.

- (2) es la hipótesis de inducción.
- (3) es lo que hace el código.
- (4) es la regla del heap.
- Ninguna de ellas mira el signo de los pesos.
- La misma idea aparece en el caso base: volver a s no da un largo negativo.

El contraejemplo

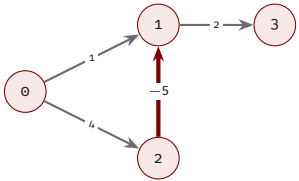


Dirigido: una arista negativa no dirigida ya sería un ciclo negativo.

terminal

```
$ build/negative_edge
se fija  con distancia  verdad
   0          0          0
   1          1          -1   se rompió el invariante
   3          3          1   se rompió el invariante
   2          4          4
al final, Dijkstra: 0 -1 4 3
al final, árbitro:  0 -1 4 1
```

El contraejemplo



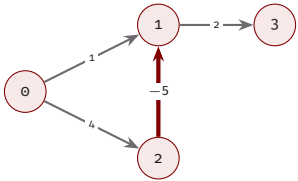
Dirigido: una arista negativa no dirigida ya sería un ciclo negativo.

```
terminal

$ build/negative_edge
se fija  con distancia  verdad
   0          0          0
   1          1          -1   se rompió el invariante
   3          3          1   se rompió el invariante
   2          4          4
al final, Dijkstra: 0 -1 4 3
al final, árbitro:  0 -1 4 1
```

$S = \{0\}$ y $u = 1$, con $d[1] = 1$. Con $P = 0 \to 2 \to 1$: $(a, b) = (0, 2)$ y $P_1 = 0 \to 2$.

El contraejemplo



Dirigido: una arista negativa no dirigida ya sería un ciclo negativo.

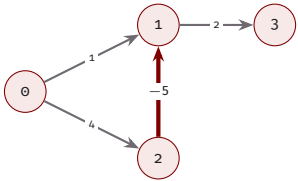
terminal

```
$ build/negative_edge
se fija  con distancia  verdad
   0          0          0
   1          1          -1   se rompió el invariante
   3          3          1   se rompió el invariante
   2          4          4
al final, Dijkstra: 0 -1 4 3
al final, árbitro:  0 -1 4 1
```

$S = \{0\}$ y $u = 1$, con $d[1] = 1$. Con $P = 0 \to 2 \to 1$: $(a, b) = (0, 2)$ y $P_1 = 0 \to 2$.

(1) pide $-1 \geq 4$. Falso. Las otras líneas siguen en pie: $4 = d[0] + 4 = d[2] \geq d[1] = 1$.

El contraejemplo



Dirigido: una arista negativa no dirigida ya sería un ciclo negativo.

```
terminal

$ build/negative_edge
se fija  con distancia  verdad
   0          0          0
   1          1          -1   se rompió el invariante
   3          3          1   se rompió el invariante
   2          4          4
al final, Dijkstra: 0 -1 4 3
al final, árbitro:  0 -1 4 1
```

$S = \{0\}$ y $u = 1$, con $d[1] = 1$. Con $P = 0 \to 2 \to 1$: $(a, b) = (0, 2)$ y $P_1 = 0 \to 2$.

(1) pide $-1 \geq 4$. Falso. Las otras líneas siguen en pie: $4 = d[0] + 4 = d[2] \geq d[1] = 1$.

Y el daño se propaga: el 1 ya relajó con el valor 1, y nadie vuelve a leer su lista. El 3 queda en 3.

¿Sirve sumar una constante?

terminal

```
$ build/shift_weights
se suma 5 a cada peso
camino que gana tras sumar: 0 1 3  pesa 13 con la suma, 3 sin ella
camino mínimo original:     0 2 1 3  pesa 16 con la suma, 1 sin ella
árbitro, grafo original, distancia a 3: 1
```

¿Sirve sumar una constante?

terminal

```
$ build/shift_weights
se suma 5 a cada peso
camino que gana tras sumar: 0 1 3  pesa 13 con la suma, 3 sin ella
camino mínimo original:     0 2 1 3  pesa 16 con la suma, 1 sin ella
árbitro, grafo original, distancia a 3: 1
```

- **Un camino de h arcos sube $h \cdot c$:** los de más arcos suben más.

¿Sirve sumar una constante?

terminal

```
$ build/shift_weights
se suma 5 a cada peso
camino que gana tras sumar: 0 1 3  pesa 13 con la suma, 3 sin ella
camino mínimo original:     0 2 1 3  pesa 16 con la suma, 1 sin ella
árbitro, grafo original, distancia a 3: 1
```

- **Un camino de h arcos sube $h \cdot c$:** los de más arcos suben más.
- Dijkstra responde bien sobre el grafo nuevo, que es otro problema.

¿Sirve sumar una constante?

terminal

```
$ build/shift_weights
se suma 5 a cada peso
camino que gana tras sumar: 0 1 3  pesa 13 con la suma, 3 sin ella
camino mínimo original:     0 2 1 3  pesa 16 con la suma, 1 sin ella
árbitro, grafo original, distancia a 3: 1
```

- **Un camino de h arcos sube $h \cdot c$:** los de más arcos suben más.
- Dijkstra responde bien sobre el grafo nuevo, que es otro problema.
- Multiplicar por una constante positiva sí conserva cuál camino gana.

¿Sirve sumar una constante?

terminal

```
$ build/shift_weights
se suma 5 a cada peso
camino que gana tras sumar: 0 1 3  pesa 13 con la suma, 3 sin ella
camino mínimo original:     0 2 1 3  pesa 16 con la suma, 1 sin ella
árbitro, grafo original, distancia a 3: 1
```

- **Un camino de h arcos sube $h \cdot c$:** los de más arcos suben más.
- Dijkstra responde bien sobre el grafo nuevo, que es otro problema.
- Multiplicar por una constante positiva sí conserva cuál camino gana.

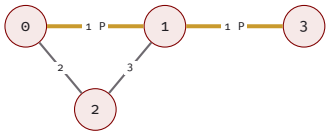
Costo: cada arco se mira una vez y mete a lo sumo una entrada. $O((V + E) \log E)$, y $\log E \leq 2 \log V$ si no hay arcos repetidos, de modo que queda $O((V + E) \log V)$.

SECCIÓN 05

El grafo de estados

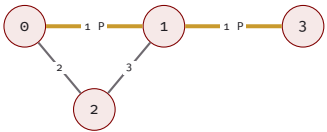
05

Cuando importa algo más que el nodo



Doble sentido. P: con peaje. De 0 a 3, con $k = 1$.

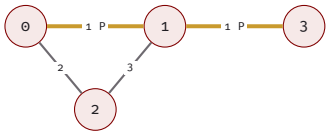
Cuando importa algo más que el nodo



Doble sentido. P: con peaje. De 0 a 3, con $k = 1$.

- $0 \to 1 \to 3$ mide 2, pero usa dos peajes.

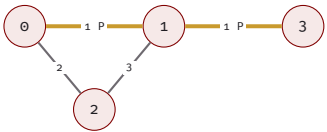
Cuando importa algo más que el nodo



Doble sentido. P: con peaje. De 0 a 3, con $k = 1$.

- $0 \to 1 \to 3$ mide 2, pero usa dos peajes.
- Hay que llegar a 1 sin haber pagado:
  $0 \to 2 \to 1 \to 3$ mide 6.

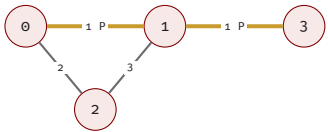
Cuando importa algo más que el nodo



Doble sentido. P: con peaje. De 0 a 3, con $k = 1$.

- $0 \to 1 \to 3$ mide 2, pero usa dos peajes.
- Hay que llegar a 1 sin haber pagado: $0 \to 2 \to 1 \to 3$ mide 6.
- Dijkstra sobre las ciudades fija al 1 con distancia 1, y con el peaje ya gastado.

Cuando importa algo más que el nodo



Doble sentido. P: con peaje. De 0 a 3, con $k = 1$.

- $0 \to 1 \to 3$ mide 2, pero usa dos peajes.
- Hay que llegar a 1 sin haber pagado: $0 \to 2 \to 1 \to 3$ mide 6.
- Dijkstra sobre las ciudades fija al 1 con distancia 1, y con el peaje ya gastado.

Lo que falla

El teorema sigue siendo cierto. Lo que pasa es que “lo mejor para llegar a 1” depende de cuántos peajes quedan. Dos llegadas a una ciudad ya no son comparables.

Dos formas de equivocarse

- **Contador global:** el peaje de una ruta se cuenta en contra de las demás.

terminal

```
$ build/toll_layers | tail -n 2
contador global: -1
una distancia por ciudad: -1
```

Dos formas de equivocarse

- **Contador global:** el peaje de una ruta se cuenta en contra de las demás.
- **Una distancia por ciudad:** una llegada más corta pisa a una más larga con menos peajes.

terminal

```
$ build/toll_layers | tail -n 2
contador global: -1
una distancia por ciudad: -1
```

Dos formas de equivocarse

- **Contador global:** el peaje de una ruta se cuenta en contra de las demás.
- **Una distancia por ciudad:** una llegada más corta pisa a una más larga con menos peajes.

terminal

```
$ build/toll_layers | tail -n 2
contador global: -1
una distancia por ciudad: -1
```

Las dos dicen que no hay ruta, y sí hay una de largo 6.

Dos formas de equivocarse

- **Contador global:** el peaje de una ruta se cuenta en contra de las demás.
- **Una distancia por ciudad:** una llegada más corta pisa a una más larga con menos peajes.

terminal

```
$ build/toll_layers | tail -n 2
contador global: -1
una distancia por ciudad: -1
```

Las dos dicen que no hay ruta, y sí hay una de largo 6.

Error común

Un contador global. Lo que hay que recordar *por ruta* tiene que vivir en el nodo.

La idea: lo que importa entra en el nodo

Grafo de estados *(state graph)*

Un nodo (c, j) por cada ciudad c y cada j de 0 a k: “estoy en c y llevo j peajes”. Son $k + 1$ **capas**.

La idea: lo que importa entra en el nodo

Grafo de estados *(state graph)*

Un nodo (c, j) por cada ciudad c y cada j de 0 a k: "estoy en c y llevo j peajes". Son $k+1$ **capas**.

- Carretera **sin** peaje, de u a v: un arco $(u, j) \to (v, j)$ en cada capa.

La idea: lo que importa entra en el nodo

Grafo de estados ***(state graph)***

Un nodo (c, j) por cada ciudad c y cada j de 0 a k: "estoy en c y llevo j peajes". Son $k+1$ **capas**.

- Carretera **sin** peaje, de u a v: un arco $(u, j) \to (v, j)$ en cada capa.
- Carretera **con** peaje, de u a v: un arco $(u, j) \to (v, j+1)$, para j de 0 a $k-1$.

La idea: lo que importa entra en el nodo

Grafo de estados *(state graph)*

Un nodo (c, j) por cada ciudad c y cada j de 0 a k: "estoy en c y llevo j peajes". Son $k+1$ **capas**.

- Carretera **sin** peaje, de u a v: un arco $(u, j) \to (v, j)$ en cada capa.
- Carretera **con** peaje, de u a v: un arco $(u, j) \to (v, j+1)$, para j de 0 a $k-1$.
- Cada carretera de doble sentido da sus arcos en los dos sentidos.

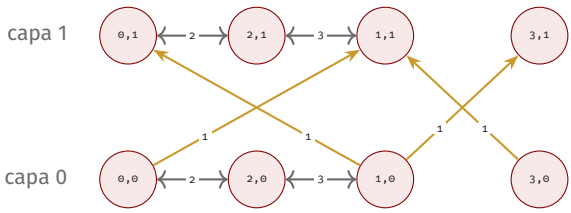
La idea: lo que importa entra en el nodo

Grafo de estados *(state graph)*

Un nodo (c, j) por cada ciudad c y cada j de 0 a k: "estoy en c y llevo j peajes". Son $k+1$ **capas**.

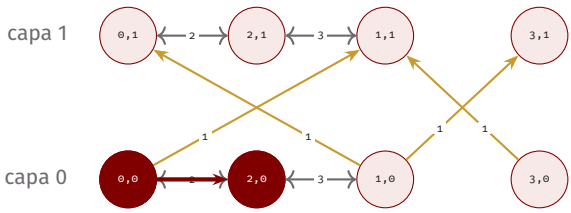
- Carretera **sin** peaje, de u a v: un arco $(u, j) \to (v, j)$ en cada capa.
- Carretera **con** peaje, de u a v: un arco $(u, j) \to (v, j+1)$, para j de 0 a $k-1$.
- Cada carretera de doble sentido da sus arcos en los dos sentidos.
- **Respuesta:** el menor $d[(t, j)]$ entre todas las capas. Se pueden usar menos de k peajes.

El ejemplo, con dos capas



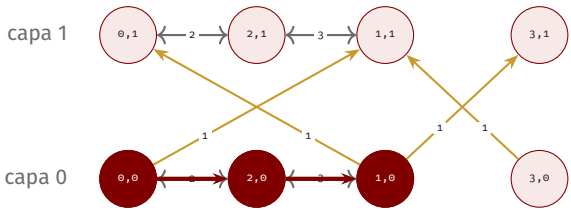
Doce arcos: 8 sin peaje, en dos capas, y 4 con peaje, que suben.

El ejemplo, con dos capas



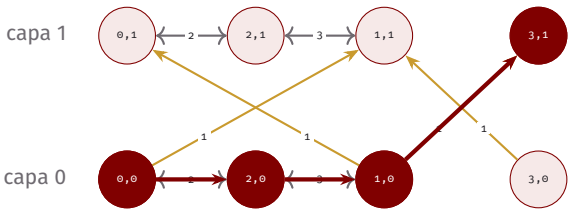
De (0,0) a (2,0), largo 2, sin gastar el peaje.

El ejemplo, con dos capas



De (2,0) a (1,0), total 5. La ciudad 1 también existe como (1,1), con distancia 1, y no molesta.

El ejemplo, con dos capas



De (1,0) sube a (3,1), total 6. Es la respuesta.

Dijkstra corre sin cambios

terminal

```
$ build/toll_layers
estado  distancia
(0,0)    0
(1,1)    1
(2,0)    2
(2,1)    4
(1,0)    5
(0,1)    6
(3,1)    6
a (3,0): no se llega
a (3,1): 6 por (0,0) (2,0) (1,0) (3,1)
```

Dijkstra corre sin cambios

terminal

```
$ build/toll_layers
estado  distancia
(0,0)    0
(1,1)    1
(2,0)    2
(2,1)    4
(1,0)    5
(0,1)    6
(3,1)    6
a (3,0): no se llega
a (3,1): 6 por (0,0) (2,0) (1,0) (3,1)
```

- Los doce arcos están escritos a mano, y `dijkstra` no sabe qué es un peaje.

Dijkstra corre sin cambios

terminal

```
$ build/toll_layers
estado  distancia
(0,0)    0
(1,1)    1
(2,0)    2
(2,1)    4
(1,0)    5
(0,1)    6
(3,1)    6
a (3,0): no se llega
a (3,1): 6 por (0,0) (2,0) (1,0) (3,1)
```

- Los doce arcos están escritos a mano, y `dijkstra` no sabe qué es un peaje.
- La ciudad 1 son dos nodos: (1,1) con 1, y (1,0) con 5. Ya no hay conflicto.

Dijkstra corre sin cambios

terminal

```
$ build/toll_layers
estado  distancia
(0,0)    0
(1,1)    1
(2,0)    2
(2,1)    4
(1,0)    5
(0,1)    6
(3,1)    6
a (3,0): no se llega
a (3,1): 6 por (0,0) (2,0) (1,0) (3,1)
```

- Los doce arcos están escritos a mano, y `dijkstra` no sabe qué es un peaje.
- La ciudad 1 son dos nodos: (1,1) con 1, y (1,0) con 5. Ya no hay conflicto.
- Los pesos siguen sin ser negativos, así que el teorema aplica tal cual.

Por qué es correcto

Afirmación. El menor $d[(t, j)]$ es la distancia mínima entre las rutas con a lo sumo k peajes.

Técnica. Una correspondencia entre rutas y caminos, que conserva el largo, en los dos sentidos.

Por qué es correcto

Afirmación. El menor $d[(t, j)]$ es la distancia mínima entre las rutas con a lo sumo k peajes.

Técnica. Una correspondencia entre rutas y caminos, que conserva el largo, en los dos sentidos.

De ruta a camino. Se recorre llevando la cuenta j de peajes. Sin peaje: un arco que no cambia j. Con peaje: un arco que sube j, y existe porque $j + 1 \leq k$.

Por qué es correcto

Afirmación. El menor $d[(t, j)]$ es la distancia mínima entre las rutas con a lo sumo k peajes.

Técnica. Una correspondencia entre rutas y caminos, que conserva el largo, en los dos sentidos.

De ruta a camino. Se recorre llevando la cuenta j de peajes. Sin peaje: un arco que no cambia j. Con peaje: un arco que sube j, y existe porque $j + 1 \leq k$.

De camino a ruta. Se toman solo las ciudades. Cada subida de capa es un peaje, y las capas solo suben: exactamente $j \leq k$ peajes.

Por qué es correcto

Afirmación. El menor $d[(t, j)]$ es la distancia mínima entre las rutas con a lo sumo k peajes.

Técnica. Una correspondencia entre rutas y caminos, que conserva el largo, en los dos sentidos.

De ruta a camino. Se recorre llevando la cuenta j de peajes. Sin peaje: un arco que no cambia j. Con peaje: un arco que sube j, y existe porque $j + 1 \leq k$.

De camino a ruta. Se toman solo las ciudades. Cada subida de capa es un peaje, y las capas solo suben: exactamente $j \leq k$ peajes.

Los mínimos coinciden. Sin ruta válida no hay camino, y todos dan ∞. ■

Por qué es correcto

Afirmación. El menor $d[(t, j)]$ es la distancia mínima entre las rutas con a lo sumo k peajes.

Técnica. Una correspondencia entre rutas y caminos, que conserva el largo, en los dos sentidos.

De ruta a camino. Se recorre llevando la cuenta j de peajes. Sin peaje: un arco que no cambia j. Con peaje: un arco que sube j, y existe porque $j + 1 \leq k$.

De camino a ruta. Se toman solo las ciudades. Cada subida de capa es un peaje, y las capas solo suben: exactamente $j \leq k$ peajes.

Los mínimos coinciden. Sin ruta válida no hay camino, y todos dan ∞. ■

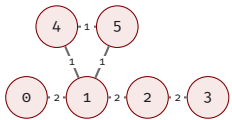
¿Repetir ciudades? Con peajes, nunca hace falta: quitar el tramo entre dos visitas no sube el largo ni los peajes.

Otra restricción: número par de tramos

Dos choferes se turnan en cada tramo, y debe llegar manejando el que salió.

state_graph.h

```cpp
// Estado: ciudad * c + resto. Arcos al vuelo.
std::size_t city = state / layers;
std::size_t next_rest = (state % layers + 1) % layers;
for (const Arc& arc : graph[city]) {
    std::size_t next = static_cast<std::size_t>(arc.to)
                     * layers + next_rest;
    if (d + arc.weight < distance[next]) {
        distance[next] = d + arc.weight;
        parent[next] = state;
        heap.push({distance[next], next});
    }
}
```



Aquí sí hay que repetir una ciudad

terminal

```
$ build/driver_turns
sin la regla: 6 por 0 1 2 3
con número par de tramos, solo la línea: no hay recorrido
con número par de tramos, con el triángulo: 9 por 0 1 5 4 1 2 3
```

Aquí sí hay que repetir una ciudad

terminal

```
$ build/driver_turns
sin la regla: 6 por 0 1 2 3
con número par de tramos, solo la línea: no hay recorrido
con número par de tramos, con el triángulo: 9 por 0 1 5 4 1 2 3
```

- Solo con la línea: de 0 a 3 siempre son tramos impares.

Aquí sí hay que repetir una ciudad

terminal

```
$ build/driver_turns
sin la regla: 6 por 0 1 2 3
con número par de tramos, solo la línea: no hay recorrido
con número par de tramos, con el triángulo: 9 por 0 1 5 4 1 2 3
```

- Solo con la línea: de 0 a 3 siempre son tramos impares.
- Con el triángulo: una vuelta de 3 tramos cambia la paridad, y pasa dos veces por el 1.

Aquí sí hay que repetir una ciudad

terminal

```
$ build/driver_turns
sin la regla: 6 por 0 1 2 3
con número par de tramos, solo la línea: no hay recorrido
con número par de tramos, con el triángulo: 9 por 0 1 5 4 1 2 3
```

- Solo con la línea: de 0 a 3 siempre son tramos impares.
- Con el triángulo: una vuelta de 3 tramos cambia la paridad, y pasa dos veces por el 1.
- Son dos estados distintos: $(1, \text{impar})$ y $(1, \text{par})$.

Aquí sí hay que repetir una ciudad

terminal

```
$ build/driver_turns
sin la regla: 6 por 0 1 2 3
con número par de tramos, solo la línea: no hay recorrido
con número par de tramos, con el triángulo: 9 por 0 1 5 4 1 2 3
```

- Solo con la línea: de 0 a 3 siempre son tramos impares.
- Con el triángulo: una vuelta de 3 tramos cambia la paridad, y pasa dos veces por el 1.
- Son dos estados distintos: $(1, \text{impar})$ y $(1, \text{par})$.

La pregunta

¿Qué tengo que saber aquí para decidir lo que sigue? Si es algo más que el nodo, va en el nodo.

El costo

Con E contando **arcos dirigidos** (una carretera de doble sentido cuenta dos):

terminal

```
$ make bench
capas    estados    sacadas  arcos mirados      ms
    1      20000      28963          99260     3.7
    2      40000      57886         198520     8.1
    4      80000     115685         397040    17.9
    8     160000     230835         794080    36.5
   16     320000     466512        1588160    80.5
   32     640000     957475        3176320   172.9
```

Grafo de resto módulo c, 20 000 ciudades al azar. Conteos exactos. Tiempos: una corrida en el contenedor con `-O2`, menor de 3 intentos, máquina compartida; dependen de la máquina.

El costo

Con E contando **arcos dirigidos** (una carretera de doble sentido cuenta dos):

- **Nodos:** $V(k+1)$. **Arcos:** a lo sumo $E(k+1)$. En el ejemplo: 8 nodos, 12 arcos, cota 16.

terminal

```
$ make bench
capas    estados    sacadas  arcos mirados      ms
    1      20000      28963          99260     3.7
    2      40000      57886         198520     8.1
    4      80000     115685         397040    17.9
    8     160000     230835         794080    36.5
   16     320000     466512        1588160    80.5
   32     640000     957475        3176320   172.9
```

Grafo de resto módulo c, 20 000 ciudades al azar. Conteos exactos. Tiempos: una corrida en el contenedor con `-O2`, menor de 3 intentos, máquina compartida; dependen de la máquina.

El costo

Con E contando **arcos dirigidos** (una carretera de doble sentido cuenta dos):

- **Nodos:** $V(k+1)$. **Arcos:** a lo sumo $E(k+1)$. En el ejemplo: 8 nodos, 12 arcos, cota 16.
- **Dijkstra:** $O((V+E)(k+1)\log(V(k+1)))$.

terminal

```
$ make bench
capas    estados    sacadas  arcos mirados      ms
    1      20000      28963          99260     3.7
    2      40000      57886         198520     8.1
    4      80000     115685         397040    17.9
    8     160000     230835         794080    36.5
   16     320000     466512        1588160    80.5
   32     640000     957475        3176320   172.9
```

Grafo de resto módulo c, 20 000 ciudades al azar. Conteos exactos. Tiempos: una corrida en el contenedor con `-O2`, menor de 3 intentos, máquina compartida; dependen de la máquina.

SECCIÓN 06

Cierre

06

Tres demostraciones, comparadas

	Qué se trabaja	Por qué no empeora	Qué completa

Tres demostraciones, comparadas

	Qué se trabaja	Por qué no empeora	Qué completa
Retraso máximo	Un horario óptimo, cambiando vecinas	La que empeora no pasa del retraso viejo de la otra	Los horarios sin inversiones valen lo mismo

Tres demostraciones, comparadas

	Qué se trabaja	Por qué no empeora	Qué completa
Retraso máximo	Un horario óptimo, cambiando vecinas	La que empeora no pasa del retraso viejo de la otra	Los horarios sin inversiones valen lo mismo
Huffman	Un árbol óptimo, cambiando hojas	Mandar abajo lo menos frecuente no sube el costo	Inducción, con un símbolo menos

Tres demostraciones, comparadas

	Qué se trabaja	Por qué no empeora	Qué completa
Retraso máximo	Un horario óptimo, cambiando vecinas	La que empeora no pasa del retraso viejo de la otra	Los horarios sin inversiones valen lo mismo
Huffman	Un árbol óptimo, cambiando hojas	Mandar abajo lo menos frecuente no sube el costo	Inducción, con un símbolo menos
Dijkstra	Un invariante sobre los fijados	Lo que falta de un camino no pesa negativo	Todo camino cruza la frontera

Tres demostraciones, comparadas

	Qué se trabaja	Por qué no empeora	Qué completa
Retraso máximo	Un horario óptimo, cambiando vecinas	La que empeora no pasa del retraso viejo de la otra	Los horarios sin inversiones valen lo mismo
Huffman	Un árbol óptimo, cambiando hojas	Mandar abajo lo menos frecuente no sube el costo	Inducción, con un símbolo menos
Dijkstra	Un invariante sobre los fijados	Lo que falta de un camino no pesa negativo	Todo camino cruza la frontera

En los tres hubo que hablar de empates

Plazos iguales, frecuencias iguales, distancias iguales. Las demostraciones que los ignoran suelen tener un agujero justo ahí.

Para la próxima clase

- Leer los apuntes de la semana 5, y hacer sus ejercicios.
- Laboratorio: `lab/toll_routes`. Antes de programar, calculen estados, arcos y la mayor distancia posible.
- La próxima semana: árboles de expansión mínima y DFS.

Preguntas

Con frecuencias 1, 1, 2, 3, 5, 8, ¿qué árbol da Huffman y cuál es su costo?
¿En qué paso exacto falla la demostración de Dijkstra si hay un arco negativo?
Con k peajes, ¿cuántos nodos y arcos hay en el grafo de estados, y cuál es el costo de Dijkstra?