

UNIVERSIDAD NACIONAL DE INGENIERÍA

Teoría de la Computación

Espacio

Savitch, PSPACE, TQBF y juegos, L y NL

Gerson Garrido

Semana 14

Hoy

Sesión A

1. El espacio se reutiliza
2. Medir el espacio. SAT en espacio lineal
3. Espacio y tiempo: contar configuraciones
4. PSPACE y la cadena de clases
5. El teorema de Savitch

Sesión B

6. Fórmulas cuantificadas y TQBF
7. TQBF es PSPACE-completo
8. Un juego es una fórmula
9. Geografía generalizada
10. Menos espacio que la entrada: L y NL

> **Lectura**
>
> Apuntes de la semana 14. Apoyo: Sipser, sec. 8.1 a 8.6; Barak, cap. 17. Código en `code/week-14`.

SECCIÓN 01

El espacio se reutiliza

01

De dónde venimos

- Semanas 12 y 13: un solo recurso, el **tiempo**. P, NP, Cook-Levin.
- Quedó abierto: ¿adivinar ayuda? ¿P $=$ NP?

De dónde venimos

- Semanas 12 y 13: un solo recurso, el **tiempo**. P, NP, Cook-Levin.
- Quedó abierto: ¿adivinar ayuda? ¿$\mathrm{P} = \mathrm{NP}$?
- Hoy: el otro recurso, la cantidad de **cinta**. Las mismas preguntas, con otras respuestas.

De dónde venimos

- Semanas 12 y 13: un solo recurso, el **tiempo**. P, NP, Cook-Levin.
- Quedó abierto: ¿adivinar ayuda? ¿$\mathrm{P} = \mathrm{NP}$?
- Hoy: el otro recurso, la cantidad de **cinta**. Las mismas preguntas, con otras respuestas.

Toda la diferencia

El **tiempo** se gasta: un paso que ya se dio no se recupera.
El **espacio** se reutiliza: una celda que ya no hace falta se vuelve a escribir.

Medir el espacio

Complejidad en espacio ***(space complexity)***

M determinista, se detiene siempre. $f(n)$ es el máximo número de **celdas** que visita con entradas de longitud n.
Si M es no determinista y todas sus ramas se detienen: el máximo sobre todas las ramas.

Medir el espacio

Complejidad en espacio ***(space complexity)***

M determinista, se detiene siempre. $f(n)$ es el máximo número de **celdas** que visita con entradas de longitud n.
Si M es no determinista y todas sus ramas se detienen: el máximo sobre todas las ramas.

- SPACE($f(n)$): lenguajes decididos por una máquina determinista con espacio $O(f(n))$.
- NSPACE($f(n)$): lo mismo, con una máquina no determinista.

Medir el espacio

Complejidad en espacio ***(space complexity)***

M determinista, se detiene siempre. $f(n)$ es el máximo número de **celdas** que visita con entradas de longitud n.
Si M es no determinista y todas sus ramas se detienen: el máximo sobre todas las ramas.

- SPACE($f(n)$): lenguajes decididos por una máquina determinista con espacio $O(f(n))$.
- NSPACE($f(n)$): lo mismo, con una máquina no determinista.
- La entrada está en la cinta: por ahora, $f(n) \geq n$.
- Varias cintas o una: el espacio cambia solo en una constante.

SAT se decide con espacio lineal

Con la entrada $\langle \varphi \rangle$, de variables $x_1, \ldots, x_m$:

1. Para cada asignación de $x_1, \ldots, x_m$:
2. evaluar φ. Si es verdadera, *aceptar*.
3. Si ninguna sirvió, *rechazar*.

SAT se decide con espacio lineal

Con la entrada $\langle \varphi \rangle$, de variables $x_1, \ldots, x_m$:

1. Para cada asignación de $x_1, \ldots, x_m$:
2. evaluar φ. Si es verdadera, *aceptar*.
3. Si ninguna sirvió, *rechazar*.

- Tiempo: 2^m asignaciones.

SAT se decide con espacio lineal

Con la entrada $\langle \varphi \rangle$, de variables $x_1, \ldots, x_m$:

1. Para cada asignación de $x_1, \ldots, x_m$:
2. evaluar φ. Si es verdadera, *aceptar*.
3. Si ninguna sirvió, *rechazar*.

- Tiempo: 2^m asignaciones.
- Espacio: **una** asignación, en m celdas. La siguiente se escribe encima, como un contador binario.

Resultado

$\mathrm{SAT} \in \mathsf{SPACE}(n)$

SAT se decide con espacio lineal

Con la entrada $\langle\varphi\rangle$, de variables $x_1, \ldots, x_m$:

1. Para cada asignación de $x_1, \ldots, x_m$:
2. evaluar φ. Si es verdadera, *aceptar*.
3. Si ninguna sirvió, *rechazar*.

- Tiempo: 2^m asignaciones.
- Espacio: **una** asignación, en m celdas. La siguiente se escribe encima, como un contador binario.

Resultado

$\mathrm{SAT} \in \mathsf{SPACE}(n)$

Idea clave

Probar 2^m candidatos cuesta 2^m veces el tiempo de probar uno, y el espacio de probar uno más un contador.

SAT, ejecutado: tiempo y espacio

terminal

```
$ build/eval_qbf machines/sat_example.qbf machines/unsat_example.qbf
machines/sat_example.qbf: ∃x ∃y ∃z ((x ∨ y) ∧ (¬x ∨ ¬y) ∧ (¬x ∨ z) ∧ (y ∨ z))
valor: verdadero   llamadas: 7   llamadas activas a la vez: 4

machines/unsat_example.qbf: ∃x ∃y ((x ∨ y) ∧ (x ∨ ¬y) ∧ (¬x ∨ y) ∧ (¬x ∨ ¬y))
valor: falso   llamadas: 7   llamadas activas a la vez: 3
```

- **Llamadas** = tiempo. Con la que no es satisfacible, el árbol completo: $1 + 2 + 4 = 7$.
- **Activas a la vez** = espacio. Una por variable, más la que evalúa.
- Con m variables: hasta $2^{m+1} - 1$ llamadas, y nunca más de $m + 1$ activas.

SECCIÓN 02

Espacio y tiempo

02

Contar configuraciones

Poco tiempo obliga a poco espacio: en t pasos se visitan a lo sumo $t + 1$ celdas.
$\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$.

Contar configuraciones

Poco tiempo obliga a poco espacio: en t pasos se visitan a lo sumo $t + 1$ celdas. $\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$.

¿Y al revés? Una máquina con $|Q|$ estados y alfabeto de cinta Γ que usa f celdas:

Componente	**Opciones**
El estado	$\lvert Q\rvert$
La posición del cabezal	f
El contenido de las f celdas	$\lvert\Gamma\rvert^f$

Contar configuraciones

Poco tiempo obliga a poco espacio: en t pasos se visitan a lo sumo $t + 1$ celdas. $\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$.

¿Y al revés? Una máquina con $|Q|$ estados y alfabeto de cinta Γ que usa f celdas:

Componente	Opciones
El estado	$\lvert Q\rvert$
La posición del cabezal	f
El contenido de las f celdas	$\lvert\Gamma\rvert^f$

Total

$|Q| \cdot f \cdot |\Gamma|^f = 2^{O(f)}$ configuraciones

Contar configuraciones

Poco tiempo obliga a poco espacio: en t pasos se visitan a lo sumo $t + 1$ celdas. $\mathrm{TIME}(f(n)) \subseteq \mathrm{SPACE}(f(n))$.

¿Y al revés? Una máquina con $|Q|$ estados y alfabeto de cinta Γ que usa f celdas:

Componente	Opciones
El estado	$\lvert Q\rvert$
La posición del cabezal	f
El contenido de las f celdas	$\lvert\Gamma\rvert^f$

Total

$|Q| \cdot f \cdot |\Gamma|^f = 2^{O(f)}$ configuraciones

Ejemplo: 6 estados, 5 símbolos, 5 celdas: $6 \cdot 5 \cdot 5^5 = 93\,750$. Un tope holgado, que vale para cualquier máquina de ese tamaño.

Poco espacio pone un tope al tiempo

Teorema

Si una máquina determinista usa espacio $f(n) \geq n$ y se detiene siempre, corre en tiempo $2^{O(f(n))}$.

Poco espacio pone un tope al tiempo

Teorema

Si una máquina determinista usa espacio $f(n) \geq n$ y se detiene siempre, corre en tiempo $2^{O(f(n))}$.

1. Hay a lo sumo $2^{cf(n)}$ configuraciones, con $c = \log_2 |Q| + 1 + \log_2 |\Gamma|$.

Poco espacio pone un tope al tiempo

Teorema

Si una máquina determinista usa espacio $f(n) \geq n$ y se detiene siempre, corre en tiempo $2^{O(f(n))}$.

1. Hay a lo sumo $2^{cf(n)}$ configuraciones, con $c = \log_2 |Q| + 1 + \log_2 |\Gamma|$.
2. La máquina **no repite** ninguna. Si repitiera C: es determinista, así que desde C haría otra vez lo mismo, volvería a C, y no se detendría.

Poco espacio pone un tope al tiempo

Teorema

Si una máquina determinista usa espacio $f(n) \geq n$ y se detiene siempre, corre en tiempo $2^{O(f(n))}$.

1. Hay a lo sumo $2^{cf(n)}$ configuraciones, con $c = \log_2 |Q| + 1 + \log_2 |\Gamma|$.
2. La máquina **no repite** ninguna. Si repitiera C: es determinista, así que desde C haría otra vez lo mismo, volvería a C, y no se detendría.
3. Cada paso da una configuración nueva: menos de $2^{cf(n)}$ pasos. ■

Poco espacio pone un tope al tiempo

Teorema

Si una máquina determinista usa espacio $f(n) \geq n$ y se detiene siempre, corre en tiempo $2^{O(f(n))}$.

1. Hay a lo sumo $2^{cf(n)}$ configuraciones, con $c = \log_2 |Q| + 1 + \log_2 |\Gamma|$.
2. La máquina **no repite** ninguna. Si repitiera C: es determinista, así que desde C haría otra vez lo mismo, volvería a C, y no se detendría.
3. Cada paso da una configuración nueva: menos de $2^{cf(n)}$ pasos. ■

Error común

Leerlo como "usa espacio $f(n)$, entonces tarda $2^{f(n)}$". Es una cota superior.

PSPACE y la cadena de clases

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \qquad \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

PSPACE y la cadena de clases

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \qquad \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

- P $\subseteq$ NP: una máquina determinista es una no determinista que nunca elige.

PSPACE y la cadena de clases

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \qquad \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

- $\text{P} \subseteq \text{NP}$: una máquina determinista es una no determinista que nunca elige.
- $\text{NP} \subseteq \text{PSPACE}$: probar **todos** los certificados, uno por uno, en el mismo espacio. Es el argumento de SAT.

PSPACE y la cadena de clases

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \qquad \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

- P $\subseteq$ NP: una máquina determinista es una no determinista que nunca elige.
- NP $\subseteq$ PSPACE: probar **todos** los certificados, uno por uno, en el mismo espacio. Es el argumento de SAT.
- PSPACE $\subseteq$ EXPTIME: espacio n^k obliga a tiempo $2^{O(n^k)}$. Es el teorema anterior.

PSPACE y la cadena de clases

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k) \qquad \text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$$

- P $\subseteq$ NP: una máquina determinista es una no determinista que nunca elige.
- NP $\subseteq$ PSPACE: probar **todos** los certificados, uno por uno, en el mismo espacio. Es el argumento de SAT.
- PSPACE $\subseteq$ EXPTIME: espacio n^k obliga a tiempo $2^{O(n^k)}$. Es el teorema anterior.

Qué se sabe

Solo que P $\neq$ EXPTIME (próxima semana). Alguna de las tres inclusiones es estricta, y **no se sabe cuál**.

SECCIÓN 03

El teorema de Savitch

03

El teorema de Savitch

Teorema (Savitch)

Para toda f con $f(n) \geq n$: $\quad \mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}\left(f(n)^2\right)$.

- Para el **tiempo**, quitar el no determinismo cuesta, hasta donde se sabe, una exponencial.

El teorema de Savitch

Teorema (Savitch)

Para toda f con $f(n) \geq n$: $\quad \mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}\big(f(n)^2\big)$.

- Para el **tiempo**, quitar el no determinismo cuesta, hasta donde se sabe, una exponencial.
- Para el **espacio**, cuesta elevar al cuadrado.

El teorema de Savitch

Teorema (Savitch)

Para toda f con $f(n) \geq n$: $\quad \mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}\big(f(n)^2\big)$.

- Para el **tiempo**, quitar el no determinismo cuesta, hasta donde se sabe, una exponencial.
- Para el **espacio**, cuesta elevar al cuadrado.
- El cuadrado de un polinomio es un polinomio: $\mathrm{PSPACE} = \mathrm{NPSPACE}$.

El teorema de Savitch

Teorema (Savitch)

Para toda f con $f(n) \geq n$: $\quad \mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}\big(f(n)^2\big)$.

- Para el **tiempo**, quitar el no determinismo cuesta, hasta donde se sabe, una exponencial.
- Para el **espacio**, cuesta elevar al cuadrado.
- El cuadrado de un polinomio es un polinomio: $\mathrm{PSPACE} = \mathrm{NPSPACE}$.

Lo que hay que construir

De una máquina no determinista N con espacio $f(n)$, una determinista M con espacio $O(f(n)^2)$.

Un intento que no alcanza

Idea. Recorrer las ramas de N una por una, reutilizando el espacio. Funcionó con los certificados de NP.

Un intento que no alcanza

Idea. Recorrer las ramas de N una por una, reutilizando el espacio. Funcionó con los certificados de NP.

- Para recorrer las ramas en orden hay que recordar **en cuál vamos**: qué eligió N en cada paso.

Un intento que no alcanza

Idea. Recorrer las ramas de N una por una, reutilizando el espacio. Funcionó con los certificados de NP.

- Para recorrer las ramas en orden hay que recordar **en cuál vamos**: qué eligió N en cada paso.
- Una rama de una máquina de espacio $f(n)$ puede tener $2^{O(f(n))}$ pasos.

Un intento que no alcanza

Idea. Recorrer las ramas de N una por una, reutilizando el espacio. Funcionó con los certificados de NP.

- Para recorrer las ramas en orden hay que recordar **en cuál vamos**: qué eligió N en cada paso.
- Una rama de una máquina de espacio $f(n)$ puede tener $2^{O(f(n))}$ pasos.
- Anotar una elección por paso ocupa $2^{O(f(n))}$ celdas. Muy lejos de $f(n)^2$.

Un intento que no alcanza

Idea. Recorrer las ramas de N una por una, reutilizando el espacio. Funcionó con los certificados de NP.

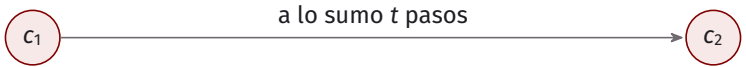
- Para recorrer las ramas en orden hay que recordar **en cuál vamos**: qué eligió N en cada paso.
- Una rama de una máquina de espacio $f(n)$ puede tener $2^{O(f(n))}$ pasos.
- Anotar una elección por paso ocupa $2^{O(f(n))}$ celdas. Muy lejos de $f(n)^2$.

Dónde falla

En NP las ramas eran cortas. Aquí son largas. Hace falta una idea que no recorra ramas.

La idea: partir el camino por la mitad

Alcance. ¿Puede N pasar de la configuración c_1 a c_2 en a lo sumo t pasos?



La idea: partir el camino por la mitad

Alcance. ¿Puede N pasar de la configuración c_1 a c_2 en a lo sumo t pasos?



- Si hay camino, a mitad de camino hay alguna configuración c_m.

La idea: partir el camino por la mitad

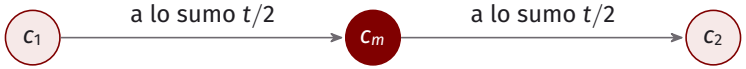
Alcance. ¿Puede N pasar de la configuración c_1 a c_2 en a lo sumo t pasos?



- Si hay camino, a mitad de camino hay alguna configuración c_m.
- No sabemos cuál es. Una máquina determinista las prueba **todas**, una por una.

La idea: partir el camino por la mitad

Alcance. ¿Puede N pasar de la configuración c_1 a c_2 en a lo sumo t pasos?



- Si hay camino, a mitad de camino hay alguna configuración c_m.
- No sabemos cuál es. Una máquina determinista las prueba **todas**, una por una.
- Las dos mitades se resuelven una **después** de la otra: la segunda usa las celdas de la primera.

CANYIELD

$\mathrm{CANYIELD}(c_1, c_2, t)$, con t una potencia de 2:

1. Si $t = 1$: *aceptar* si $c_1 = c_2$ o si c_1 produce c_2 en un paso de N. Si no, *rechazar*.
2. Si $t > 1$: para cada configuración c_m que usa a lo sumo $f(n)$ celdas:
3. correr $\mathrm{CANYIELD}(c_1, c_m, t/2)$;
4. correr $\mathrm{CANYIELD}(c_m, c_2, t/2)$;
5. si las dos aceptaron, *aceptar*.
6. Si ninguna c_m sirvió, *rechazar*.

CANYIELD

$\mathrm{CANYIELD}(c_1, c_2, t)$, con t una potencia de 2:

1. Si $t = 1$: *aceptar* si $c_1 = c_2$ o si c_1 produce c_2 en un paso de N. Si no, *rechazar*.
2. Si $t > 1$: para cada configuración c_m que usa a lo sumo $f(n)$ celdas:
3. correr $\mathrm{CANYIELD}(c_1, c_m, t/2)$;
4. correr $\mathrm{CANYIELD}(c_m, c_2, t/2)$;
5. si las dos aceptaron, *aceptar*.
6. Si ninguna c_m sirvió, *rechazar*.

La máquina M

Con la entrada w: correr $\mathrm{CANYIELD}(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)})$ y responder lo mismo. $2^{df(n)}$: una cota del número de configuraciones de N. c_{accept} es única: N borra la cinta antes de aceptar.

Por qué responde bien

Afirmación. $\mathrm{CANYIELD}(c_1, c_2, t)$ acepta si y solo si N pasa de c_1 a c_2 en a lo sumo t pasos.

Por inducción sobre $t = 2^j$.

Por qué responde bien

Afirmación. $\mathrm{CANYIELD}(c_1, c_2, t)$ acepta si y solo si N pasa de c_1 a c_2 en a lo sumo t pasos.

Por inducción sobre $t = 2^i$.

- **Base**, $t = 1$: cero pasos ($c_1 = c_2$) o un paso. Es la línea 1.

Por qué responde bien

Afirmación. $\mathrm{CANYIELD}(c_1, c_2, t)$ acepta si y solo si N pasa de c_1 a c_2 en a lo sumo t pasos.

Por inducción sobre $t = 2^j$.

- **Base**, $t = 1$: cero pasos ($c_1 = c_2$) o un paso. Es la línea 1.
- **Paso**, si acepta: halló c_m con las dos mitades. Juntando los tramos, a lo sumo t pasos.

Por qué responde bien

Afirmación. $\mathrm{CANYIELD}(c_1, c_2, t)$ acepta si y solo si N pasa de c_1 a c_2 en a lo sumo t pasos.

Por inducción sobre $t = 2^j$.

- **Base**, $t = 1$: cero pasos ($c_1 = c_2$) o un paso. Es la línea 1.
- **Paso**, si acepta: halló c_m con las dos mitades. Juntando los tramos, a lo sumo t pasos.
- **Paso**, si hay un camino de $\ell \leq t$ pasos: sea c_m la configuración después de $\lceil \ell/2 \rceil$ pasos. Los tramos tienen $\lceil \ell/2 \rceil$ y $\lfloor \ell/2 \rfloor$ pasos, los dos $\leq t/2$. El ciclo llega a esa c_m.

Por qué responde bien

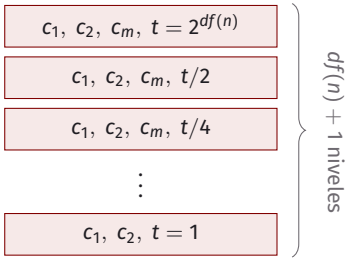
Afirmación. $\mathrm{CANYIELD}(c_1, c_2, t)$ acepta si y solo si N pasa de c_1 a c_2 en a lo sumo t pasos.

Por inducción sobre $t = 2^j$.

- **Base**, $t = 1$: cero pasos ($c_1 = c_2$) o un paso. Es la línea 1.
- **Paso**, si acepta: halló c_m con las dos mitades. Juntando los tramos, a lo sumo t pasos.
- **Paso**, si hay un camino de $\ell \leq t$ pasos: sea c_m la configuración después de $\lceil \ell/2 \rceil$ pasos. Los tramos tienen $\lceil \ell/2 \rceil$ y $\lfloor \ell/2 \rfloor$ pasos, los dos $\leq t/2$. El ciclo llega a esa c_m.

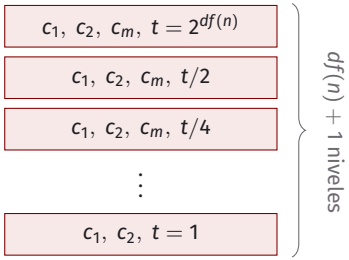
Y M decide $L(N)$: una rama que acepta no repite configuraciones, así que tiene menos de $2^{df(n)}$ pasos.

De dónde sale el cuadrado



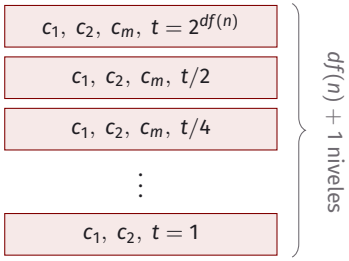
- Cada llamada guarda c_1, c_2, c_m y el exponente de t: $O(f(n))$ celdas.

De dónde sale el cuadrado



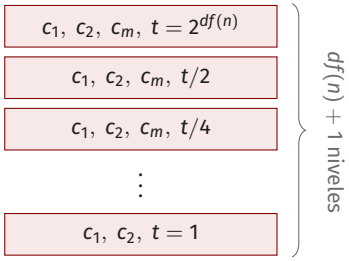
- Cada llamada guarda c_1, c_2, c_m y el exponente de t: $O(f(n))$ celdas.
- Las llamadas de las líneas 3 y 4 **no** están activas a la vez.
- Las activas forman una sola cadena: una por nivel.

De dónde sale el cuadrado



- Cada llamada guarda c_1, c_2, c_m y el exponente de t: $O(f(n))$ celdas.
- Las llamadas de las líneas 3 y 4 **no** están activas a la vez.
- Las activas forman una sola cadena: una por nivel.
- Niveles: $t = 2^{df(n)}, \ldots, 2, 1$.

De dónde sale el cuadrado



- Cada llamada guarda c_1, c_2, c_m y el exponente de t: $O(f(n))$ celdas.
- Las llamadas de las líneas 3 y 4 **no** están activas a la vez.
- Las activas forman una sola cadena: una por nivel.
- Niveles: $t = 2^{df(n)}, \ldots, 2, 1$.

Idea clave

$O(f(n))$ niveles $\times$ $O(f(n))$ celdas por nivel $= O(f(n)^2)$.

Si la máquina no conoce $f(n)$

M necesita $f(n)$: para saber qué configuraciones recorrer y con qué t empezar.

- Si f se calcula con espacio $O(f(n))$, como n^2: M la calcula primero. Son las funciones **construibles**.

Si la máquina no conoce $f(n)$

M necesita $f(n)$: para saber qué configuraciones recorrer y con qué t empezar.

- Si f se calcula con espacio $O(f(n))$, como n^2: M la calcula primero. Son las funciones **construibles**.
- Si no: probar con $i = n, n+1, n+2, \ldots$

1. ¿Se llega a c_{accept} usando solo i celdas? Si sí, *aceptar*.
2. ¿Se llega a alguna configuración con el cabezal en la celda $i+1$?
3. Si no, N nunca usa más de i celdas: *rechazar*. Si sí, seguir con $i+1$.

Si la máquina no conoce $f(n)$

M necesita $f(n)$: para saber qué configuraciones recorrer y con qué t empezar.

- Si f se calcula con espacio $O(f(n))$, como n^2: M la calcula primero. Son las funciones **construibles**.
- Si no: probar con $i = n, n+1, n+2, \ldots$

1. ¿Se llega a c_{accept} usando solo i celdas? Si sí, *aceptar*.
2. ¿Se llega a alguna configuración con el cabezal en la celda $i+1$?
3. Si no, N nunca usa más de i celdas: *rechazar*. Si sí, seguir con $i+1$.

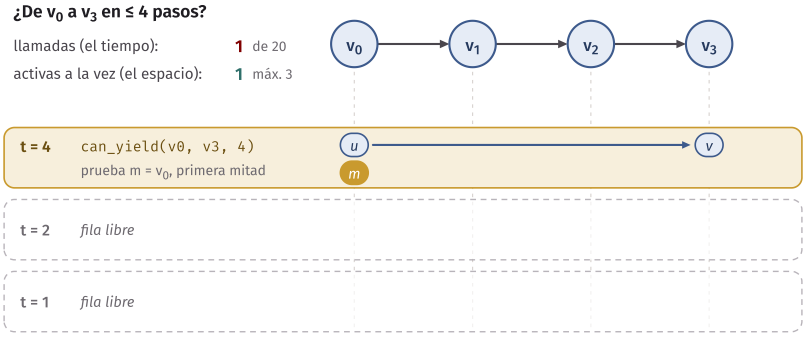
Las dos preguntas se responden con CANYIELD. i nunca pasa de $f(n)$, y cada vuelta reutiliza el espacio de la anterior: $O(f(n)^2)$.
El teorema vale también para $f(n) \geq \log n$, con el modelo de dos cintas de la sesión B.

La recursión, en un grafo: sin marcas ni cola

savitch.h

```cpp
inline bool can_yield(const Graph& g, int u, int v, long long t, int depth,
                      SavitchStats& stats) {
    stats.calls++;
    stats.max_depth = std::max(stats.max_depth, depth);
    if (t == 1) {
        return u == v || has_edge(g, u, v);
    }
    int n = static_cast<int>(g.nodes.size());
    for (int middle = 0; middle < n; middle++) {
        if (can_yield(g, u, middle, t / 2, depth + 1, stats) &&
            can_yield(g, middle, v, t / 2, depth + 1, stats)) {
            return true;
        }
    }
    return false;
}
```

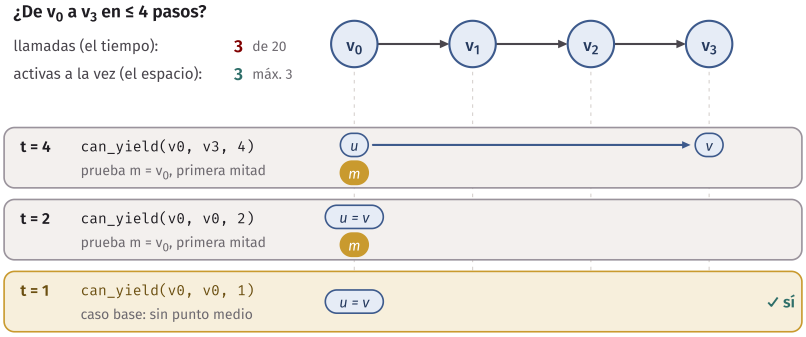
La recursión, llamada por llamada



Llamada 1: ¿de v_0 a v_3 en ≤ 4 pasos? Prueba cada nodo como punto medio, empezando por v_0.

Cada fila es una llamada activa: guarda u, v, el punto medio m que está probando y t.

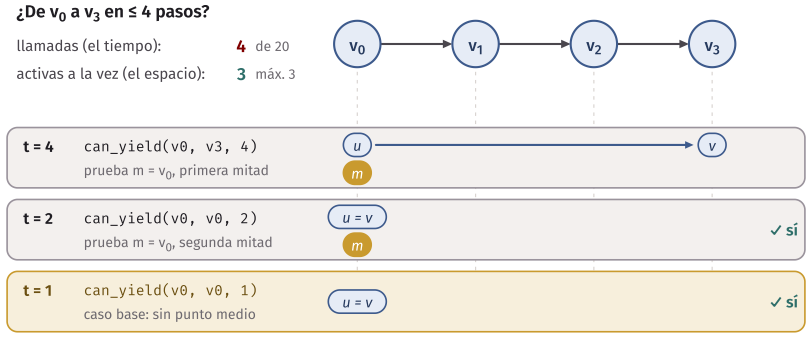
La recursión, llamada por llamada



Llamada 3: ¿de v_0 a v_0 en ≤ 1 paso? Sí: es el mismo nodo.

La primera mitad con punto medio v_0 acepta. Sigue la segunda, en la misma fila.

La recursión, llamada por llamada

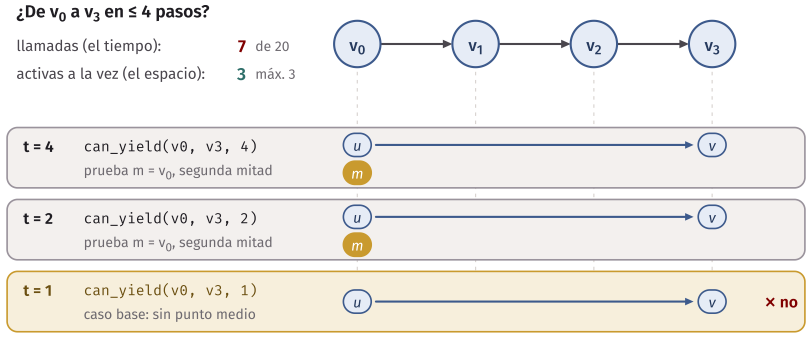


Llamada 4: ¿de v_0 a v_0 en ≤ 1 paso? Sí: es el mismo nodo.

Con punto medio v_0 aceptan las dos mitades: de v_0 a v_0 en ≤ 2 pasos, sí.

Era la primera mitad del nivel 1 con punto medio v_0. Las dos filas de abajo se desocupan.

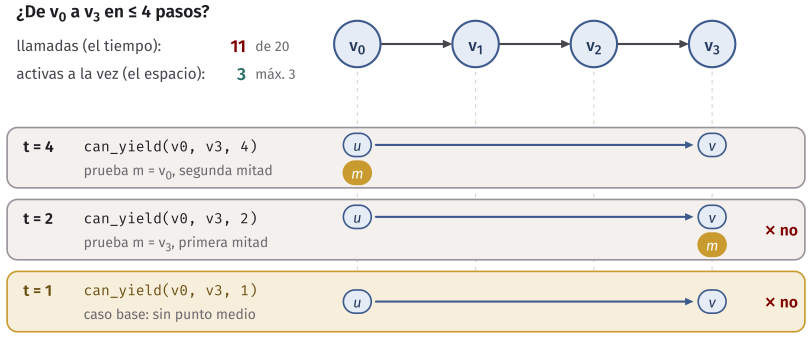
La recursión, llamada por llamada



Llamada 7: ¿de v_0 a v_3 en ≤ 1 paso? No: no hay arista.

Con punto medio v_0 falla la segunda mitad: se prueba el siguiente nodo.

La recursión, llamada por llamada

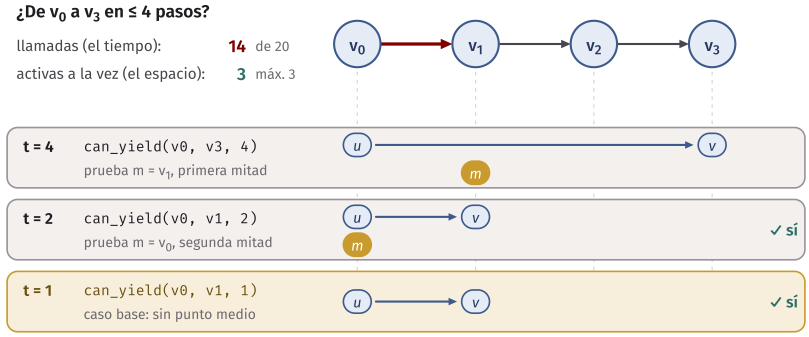


Llamada 11: ¿de v_0 a v_3 en ≤ 1 paso? No: no hay arista.

Ningún punto medio sirvió: de v_0 a v_3 en ≤ 2 pasos, no.

En el nivel 1, v_0 queda descartado como punto medio. Las dos filas de abajo se desocupan.

La recursión, llamada por llamada

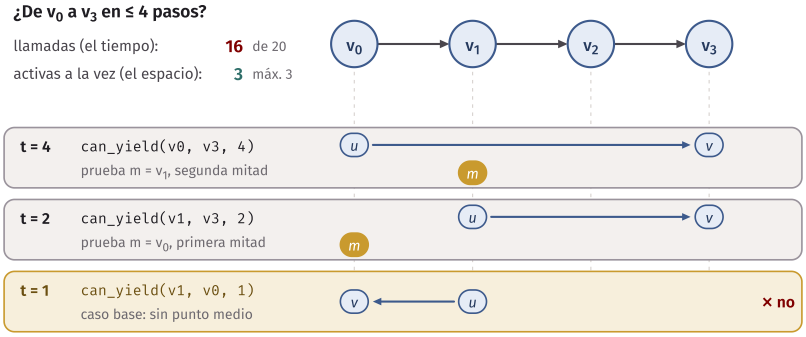


Llamada 14: ¿de v_0 a v_1 en ≤ 1 paso? Sí: hay una arista.

Con punto medio v_0 aceptan las dos mitades: de v_0 a v_1 en ≤ 2 pasos, sí.

Era la primera mitad del nivel 1 con punto medio v_1. Las dos filas de abajo se desocupan.

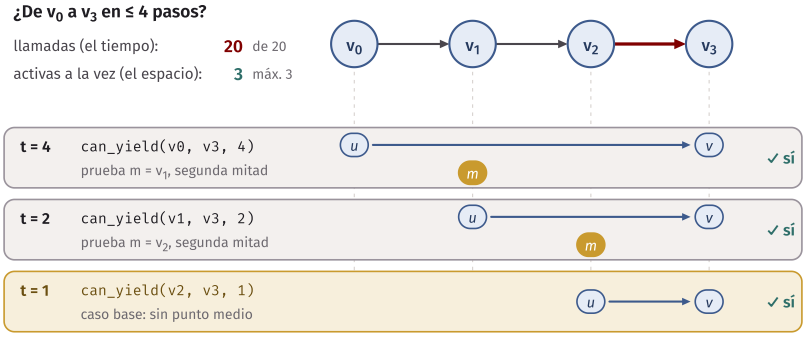
La recursión, llamada por llamada



Llamada 16: ¿de v_1 a v_0 en ≤ 1 paso? No: no hay arista.

Con punto medio v_0 falla la primera mitad: se prueba el siguiente nodo.

La recursión, llamada por llamada

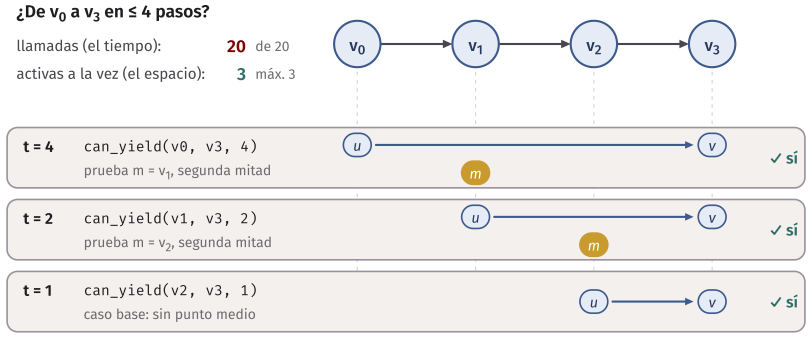


Llamada 20: ¿de v_2 a v_3 en ≤ 1 paso? Sí: hay una arista.

Con punto medio v_2 aceptan las dos mitades: de v_1 a v_3 en ≤ 2 pasos, sí.

Y con punto medio v_1 el nivel 1 tiene sus dos mitades: acepta.

La recursión, llamada por llamada



Acepta: de v_0 se llega a v_3. Fueron 20 llamadas, y nunca hubo más de 3 activas a la vez.

Cada llamada guarda tres nodos de 2 bits y el exponente de t, otros 2: 3 llamadas × 8 bits = 24 bits.

Son los números de la fila n = 4 de `savitch_demo`.

Savitch frente a BFS, con números

El camino $v_0 \to v_1 \to \cdots \to v_{n-1}$: ¿se llega de v_0 a v_{n-1}?

terminal

```
$ build/savitch_demo
   n     t   llega   Savitch: llamadas  activas  bits   BFS: aristas  cola  bits
   4     4   sí                     20        3    24              3     1     6
   8     8   sí                    443        4    44              7     1    11
  16    16   sí                  47038        5    75             15     1    20
  32    32   sí               22290333        6   108             31     1    37
```

- **Activas**: $\log_2 t + 1$. **Bits**: $\lceil \log_2 n \rceil$ por nodo; BFS, además, un bit de visitado por nodo.

Savitch frente a BFS, con números

El camino $v_0 \to v_1 \to \cdots \to v_{n-1}$: ¿se llega de v_0 a v_{n-1}?

terminal

```
$ build/savitch_demo
   n     t   llega   Savitch: llamadas  activas  bits   BFS: aristas  cola  bits
   4     4   sí                     20        3    24              3     1     6
   8     8   sí                    443        4    44              7     1    11
  16    16   sí                  47038        5    75             15     1    20
  32    32   sí               22290333        6   108             31     1    37
```

- **Activas**: $\log_2 t + 1$. **Bits**: $\lceil \log_2 n \rceil$ por nodo; BFS, además, un bit de visitado por nodo.
- Con estos tamaños BFS gana en todo. El teorema habla de cómo **crecen**: n contra $(\log_2 n)^2$.

Savitch frente a BFS, con números

El camino $v_0 \to v_1 \to \cdots \to v_{n-1}$: ¿se llega de v_0 a v_{n-1}?

terminal

```
$ build/savitch_demo
   n     t   llega   Savitch: llamadas  activas  bits   BFS: aristas  cola  bits
   4     4   sí                     20        3    24              3     1     6
   8     8   sí                    443        4    44              7     1    11
  16    16   sí                  47038        5    75             15     1    20
  32    32   sí               22290333        6   108             31     1    37
```

- **Activas**: $\log_2 t + 1$. **Bits**: $\lceil \log_2 n \rceil$ por nodo; BFS, además, un bit de visitado por nodo.
- Con estos tamaños BFS gana en todo. El teorema habla de cómo **crecen**: n contra $(\log_2 n)^2$.
- Con $n = 2^{20}$ nodos: $21 \cdot (3 \cdot 20 + 5) = 1365$ bits, contra más de un millón. Calculado, no medido.
- Nadie va a esperar a que termine: miren la columna de llamadas.

Por qué no sirve para el tiempo

Si K es el número de configuraciones, las llamadas de CANYIELD en el peor caso:

$$T(t) = 2K \cdot T(t/2) + 1 \quad \Longrightarrow \quad T(t) \geq (2K)^{\log_2 t} = 2^{O(f(n)^2)}$$

Por qué no sirve para el tiempo

Si K es el número de configuraciones, las llamadas de CANYIELD en el peor caso:

$$T(t) = 2K \cdot T(t/2) + 1 \quad \Longrightarrow \quad T(t) \geq (2K)^{\log_2 t} = 2^{O(f(n)^2)}$$

- En **espacio**, dos mitades cuestan lo que una: la segunda usa las celdas de la primera.

Por qué no sirve para el tiempo

Si K es el número de configuraciones, las llamadas de CANYIELD en el peor caso:

$$T(t) = 2K \cdot T(t/2) + 1 \quad \Longrightarrow \quad T(t) \geq (2K)^{\log_2 t} = 2^{O(f(n)^2)}$$

- En **espacio**, dos mitades cuestan lo que una: la segunda usa las celdas de la primera.
- En **tiempo**, dos mitades cuestan el doble, y se paga una vez por cada punto medio.

Por qué no sirve para el tiempo

Si K es el número de configuraciones, las llamadas de CANYIELD en el peor caso:

$$T(t) = 2K \cdot T(t/2) + 1 \quad \Longrightarrow \quad T(t) \geq (2K)^{\log_2 t} = 2^{O(f(n)^2)}$$

- En **espacio**, dos mitades cuestan lo que una: la segunda usa las celdas de la primera.
- En **tiempo**, dos mitades cuestan el doble, y se paga una vez por cada punto medio.

Error común

Aplicar el argumento al tiempo y concluir $\mathrm{NTIME}(f(n)) \subseteq \mathrm{TIME}(f(n)^2)$.
El teorema de Savitch no dice nada sobre P y NP.

PSPACE $=$ NPSPACE

Si N usa espacio n^k, la máquina M del teorema usa $O(n^{2k})$: sigue siendo polinomial.

	Tiempo polinomial	**Espacio polinomial**
¿El no determinismo agrega poder?	No se sabe: P frente a NP	No: PSPACE $=$ NPSPACE
¿Cerrada bajo complemento?	P sí. NP: no se sabe	Sí: es una clase determinista

PSPACE $=$ NPSPACE

Si N usa espacio n^k, la máquina M del teorema usa $O(n^{2k})$: sigue siendo polinomial.

	Tiempo polinomial	**Espacio polinomial**
¿El no determinismo agrega poder?	No se sabe: P frente a NP	No: PSPACE $=$ NPSPACE
¿Cerrada bajo complemento?	P sí. NP: no se sabe	Sí: es una clase determinista

PSPACE = NPSPACE

Si N usa espacio n^k, la máquina M del teorema usa $O(n^{2k})$: sigue siendo polinomial.

	Tiempo polinomial	**Espacio polinomial**
¿El no determinismo agrega poder?	No se sabe: P frente a NP	No: PSPACE = NPSPACE
¿Cerrada bajo complemento?	P sí. NP: no se sabe	Sí: es una clase determinista

Para lo que sigue

Para mostrar que algo está en PSPACE se puede dar un algoritmo no determinista de espacio polinomial.

SECCIÓN 04

Fórmulas booleanas cuantificadas

04

Cuantificadores sobre variables booleanas

- $\exists x\, \varphi$ es verdadera si φ lo es con $x = 0$ **o** con $x = 1$.
- $\forall x\, \varphi$ es verdadera si φ lo es con $x = 0$ **y** con $x = 1$.

Cuantificadores sobre variables booleanas

- $\exists x\, \varphi$ es verdadera si φ lo es con $x = 0$ **o** con $x = 1$.
- $\forall x\, \varphi$ es verdadera si φ lo es con $x = 0$ **y** con $x = 1$.

Fórmula booleana cuantificada *(quantified Boolean formula)*

$\varphi = Q_1 x_1\, Q_2 x_2 \cdots Q_m x_m\, \psi$, con cada $Q_i \in \{\exists, \forall\}$ y ψ sin cuantificadores.
Totalmente cuantificada: toda variable tiene su cuantificador. **Prenex:** todos van al principio.

Cuantificadores sobre variables booleanas

- $\exists x\, \varphi$ es verdadera si φ lo es con $x = 0$ **o** con $x = 1$.
- $\forall x\, \varphi$ es verdadera si φ lo es con $x = 0$ **y** con $x = 1$.

Fórmula booleana cuantificada *(quantified Boolean formula)*

$\varphi = Q_1 x_1\, Q_2 x_2 \cdots Q_m x_m\, \psi$, con cada $Q_i \in \{\exists, \forall\}$ y ψ sin cuantificadores.
Totalmente cuantificada: toda variable tiene su cuantificador. **Prenex:** todos van al principio.

El problema

$\mathrm{TQBF} = \{\, \langle \varphi \rangle : \varphi$ es una fórmula totalmente cuantificada y verdadera $\}$

Cuantificadores sobre variables booleanas

- $\exists x\, \varphi$ es verdadera si φ lo es con $x = 0$ **o** con $x = 1$.
- $\forall x\, \varphi$ es verdadera si φ lo es con $x = 0$ **y** con $x = 1$.

Fórmula booleana cuantificada *(quantified Boolean formula)*

$\varphi = Q_1 x_1\, Q_2 x_2 \cdots Q_m x_m\, \psi$, con cada $Q_i \in \{\exists, \forall\}$ y ψ sin cuantificadores.
Totalmente cuantificada: toda variable tiene su cuantificador. **Prenex:** todos van al principio.

El problema

$\mathrm{TQBF} = \{\, \langle \varphi \rangle : \varphi$ es una fórmula totalmente cuantificada y verdadera $\}$

SAT es el caso con solo $\exists$. Las tautologías, el caso con solo $\forall$. Lo nuevo: se **alternan**.

El orden importa

$\psi = (x \vee y) \wedge (\neg x \vee \neg y)$ dice que x e y son **distintas**.

El orden importa

$\psi = (x \vee y) \wedge (\neg x \vee \neg y)$ dice que x e y son **distintas**.

$\varphi_1 = \forall x \, \exists y \; \psi$ **verdadera**

Para cada x hay una y distinta.
$x = 0$: sirve $y = 1$. $x = 1$: sirve $y = 0$.

El orden importa

$\psi = (x \vee y) \wedge (\neg x \vee \neg y)$ dice que x e y son **distintas**.

$\varphi_1 = \forall x \exists y\ \psi$ **verdadera**

Para cada x hay una y distinta.
$x = 0$: sirve $y = 1$. $x = 1$: sirve $y = 0$.

$\varphi_2 = \exists y \forall x\ \psi$ **falsa**

Hay una y distinta de *todas* las x.
$y = 0$: falla $x = 0$. $y = 1$: falla $x = 1$.

El orden importa

$\psi = (x \vee y) \wedge (\neg x \vee \neg y)$ dice que x e y son **distintas**.

$\varphi_1 = \forall x \exists y\ \psi$ **verdadera**

Para cada x hay una y distinta.
$x = 0$: sirve $y = 1$. $x = 1$: sirve $y = 0$.

$\varphi_2 = \exists y \forall x\ \psi$ **falsa**

Hay una y distinta de *todas* las x.
$y = 0$: falla $x = 0$. $y = 1$: falla $x = 1$.

En φ_1, la y se elige **después** de conocer x. En φ_2, **antes**. “Toda persona tiene una madre” frente a “hay una madre de todas las personas”.

El orden importa

$\psi = (x \vee y) \wedge (\neg x \vee \neg y)$ dice que x e y son **distintas**.

$\varphi_1 = \forall x \exists y\ \psi$ **verdadera**

Para cada x hay una y distinta.
$x = 0$: sirve $y = 1$. $x = 1$: sirve $y = 0$.

$\varphi_2 = \exists y \forall x\ \psi$ **falsa**

Hay una y distinta de *todas* las x.
$y = 0$: falla $x = 0$. $y = 1$: falla $x = 1$.

En φ_1, la y se elige **después** de conocer x. En φ_2, **antes**. "Toda persona tiene una madre" frente a "hay una madre de todas las personas".

Error común: intercambiar un $\forall$ con un $\exists$ vecino

Dos cuantificadores iguales sí se intercambian; dos distintos, no.

El formato de los ejemplos

forall_exists.qbf

```
prefix: A x E y
clauses:
x y
-x -y
```

- `prefix:` el prefijo. `A` es $\forall$ y `E` es $\exists$.
- `clauses:` la matriz, en CNF. Una cláusula por línea; `-x` es $\neg x$.

Sin la línea de comentario inicial.

terminal

```
$ build/eval_qbf errors/free_variable.qbf
error: errors/free_variable.qbf: la variable z no tiene cuantificador: la fórmula no está totalmente cuantificada
```

Sin cuantificador para z, la fórmula no es ni verdadera ni falsa: no es una entrada de TQBF.

TQBF está en PSPACE

T, con la entrada $\langle \varphi \rangle$:

1. Si φ no tiene cuantificadores: evaluarla.
2. Si $\varphi = \exists x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **alguna** aceptó.
3. Si $\varphi = \forall x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **las dos** aceptaron.

TQBF está en PSPACE

T, con la entrada $\langle \varphi \rangle$:

1. Si φ no tiene cuantificadores: evaluarla.
2. Si $\varphi = \exists x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **alguna** aceptó.
3. Si $\varphi = \forall x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **las dos** aceptaron.

- Profundidad de la recursión: el número de variables, $m \leq n$.
- Cada nivel guarda un bit: el valor de su variable.

TQBF está en PSPACE

T, con la entrada $\langle \varphi \rangle$:

1. Si φ no tiene cuantificadores: evaluarla.
2. Si $\varphi = \exists x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **alguna** aceptó.
3. Si $\varphi = \forall x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **las dos** aceptaron.

- Profundidad de la recursión: el número de variables, $m \leq n$.
- Cada nivel guarda un bit: el valor de su variable.
- Las dos llamadas de un nivel van una después de la otra, en las mismas celdas.

TQBF está en PSPACE

T, con la entrada $\langle \varphi \rangle$:

1. Si φ no tiene cuantificadores: evaluarla.
2. Si $\varphi = \exists x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **alguna** aceptó.
3. Si $\varphi = \forall x\, \psi$: correr T con $x = 0$ y después con $x = 1$. Aceptar si **las dos** aceptaron.

- Profundidad de la recursión: el número de variables, $m \leq n$.
- Cada nivel guarda un bit: el valor de su variable.
- Las dos llamadas de un nivel van una después de la otra, en las mismas celdas.

Resultado

Espacio $O(n)$: $\mathrm{TQBF} \in$ PSPACE. El tiempo puede ser 2^m.

La recursión, ejecutada

terminal

```
$ build/eval_qbf --tree machines/forall_exists.qbf
machines/forall_exists.qbf: ∀x ∃y ((x ∨ y) ∧ (¬x ∨ ¬y))
x = 0
  y = 0   ψ = falso
  y = 1   ψ = verdadero
  ∃y: verdadero
x = 1
  y = 0   ψ = verdadero
  ∃y: verdadero
∀x: verdadero
```

terminal

```
$ build/eval_qbf --tree machines/exists_forall.qbf
machines/exists_forall.qbf: ∃y ∀x ((x ∨ y) ∧ (¬x ∨ ¬y))
y = 0
  x = 0   ψ = falso
  ∀x: falso
y = 1
  x = 0   ψ = verdadero
  x = 1   ψ = falso
  ∀x: falso
∃y: falso
```

- Un $\exists$ deja de probar cuando encuentra un valor que sirve; un $\forall$, cuando encuentra uno que falla.
- Las dos ejecuciones: 6 llamadas, 3 activas a la vez.

El tiempo crece; el espacio, casi no

terminal

```
$ build/qbf_growth
∀x1 ∃y1 ∀x2 ∃y2 ((x1 ∨ y1) ∧ (¬x1 ∨ ¬y1) ∧ (x2 ∨ y2) ∧ (¬x2 ∨ ¬y2))

 k   variables   valor       llamadas   activas a la vez
 1           2   verdadero          6                  3
 2           4   verdadero         19                  5
 3           6   verdadero         51                  7
 4           8   verdadero        127                  9
 5          10   verdadero        303                 11
 6          12   verdadero        703                 13
 7          14   verdadero       1599                 15
 8          16   verdadero       3583                 17
 9          18   verdadero       7935                 19
10          20   verdadero      17407                 21
11          22   verdadero      37887                 23
12          24   verdadero      81919                 25
```

- k pares de variables, cada y_i distinta de su x_i.
- Un par más: poco más del **doble** de llamadas.
- Un par más: **dos** llamadas activas más.

SECCIÓN 05

TQBF es PSPACE-completo

05

PSPACE-completo

PSPACE-completo

B es PSPACE-completo si (1) $B \in$ PSPACE y (2) todo $A \in$ PSPACE cumple $A \leq_p B$.
Si solo cumple (2), es **PSPACE-difícil**.

PSPACE-completo

PSPACE-completo

B es PSPACE-completo si (1) $B \in$ PSPACE y (2) todo $A \in$ PSPACE cumple $A \leq_p B$.
Si solo cumple (2), es **PSPACE-difícil**.

- La reducción es en **tiempo** polinomial: tiene que ser más débil que la clase. Con reducciones de espacio polinomial, la reducción resolvería el problema.

PSPACE-completo

PSPACE-completo

B es PSPACE-completo si (1) $B \in$ PSPACE y (2) todo $A \in$ PSPACE cumple $A \leq_p B$.
Si solo cumple (2), es **PSPACE-difícil**.

- La reducción es en **tiempo** polinomial: tiene que ser más débil que la clase. Con reducciones de espacio polinomial, la reducción resolvería el problema.
- Si un problema PSPACE-completo estuviera en P, todo PSPACE estaría en P. Lo mismo con NP.

PSPACE-completo

PSPACE-completo

B es PSPACE-completo si (1) $B \in$ PSPACE y (2) todo $A \in$ PSPACE cumple $A \leq_p B$.
Si solo cumple (2), es **PSPACE-difícil**.

- La reducción es en **tiempo** polinomial: tiene que ser más débil que la clase. Con reducciones de espacio polinomial, la reducción resolvería el problema.
- Si un problema PSPACE-completo estuviera en P, todo PSPACE estaría en P. Lo mismo con NP.

Teorema

TQBF es PSPACE-completo.

Por qué no basta repetir Cook-Levin

Hay que reducir **todo** lenguaje de PSPACE. De A solo se sabe que lo decide una máquina M con espacio n^k.

- Semana 13: la fórmula describía el **tableau**, una fila por paso.

Por qué no basta repetir Cook-Levin

Hay que reducir **todo** lenguaje de PSPACE. De A solo se sabe que lo decide una máquina M con espacio n^k.

- Semana 13: la fórmula describía el **tableau**, una fila por paso.
- Funcionó porque la máquina daba un número polinomial de pasos.

Por qué no basta repetir Cook-Levin

Hay que reducir **todo** lenguaje de PSPACE. De A solo se sabe que lo decide una máquina M con espacio n^k.

- Semana 13: la fórmula describía el **tableau**, una fila por paso.
- Funcionó porque la máquina daba un número polinomial de pasos.
- Una máquina de espacio n^k puede dar $2^{O(n^k)}$ pasos: su tableau tiene un número **exponencial** de filas.

Por qué no basta repetir Cook-Levin

Hay que reducir **todo** lenguaje de PSPACE. De A solo se sabe que lo decide una máquina M con espacio n^k.

- Semana 13: la fórmula describía el **tableau**, una fila por paso.
- Funcionó porque la máquina daba un número polinomial de pasos.
- Una máquina de espacio n^k puede dar $2^{O(n^k)}$ pasos: su tableau tiene un número **exponencial** de filas.

Otra idea

No describir el cómputo paso a paso. Escribir como fórmula la pregunta de alcance de Savitch:
$\varphi_{c_1,c_2,t}$ dice "M pasa de c_1 a c_2 en a lo sumo t pasos".

Primer intento: copiar la recursión

$$\varphi_{c_1,c_2,t} \;=\; \exists m \left[\varphi_{c_1,m,t/2} \;\wedge\; \varphi_{m,c_2,t/2} \right]$$

- Dice lo correcto. El $\exists m$ reemplaza al ciclo que probaba todos los puntos medios.

Primer intento: copiar la recursión

$$\varphi_{c_1,c_2,t} \;=\; \exists m \left[\varphi_{c_1,m,t/2} \;\wedge\; \varphi_{m,c_2,t/2} \right]$$

- Dice lo correcto. El $\exists m$ reemplaza al ciclo que probaba todos los puntos medios.
- Pero la fórmula para t contiene **dos** copias de la fórmula para $t/2$.

Primer intento: copiar la recursión

$$\varphi_{c_1,c_2,t} \;=\; \exists m \left[\varphi_{c_1,m,t/2} \;\wedge\; \varphi_{m,c_2,t/2} \right]$$

- Dice lo correcto. El $\exists m$ reemplaza al ciclo que probaba todos los puntos medios.
- Pero la fórmula para t contiene **dos** copias de la fórmula para $t/2$.
- Cada nivel duplica el tamaño. Con $t = 2^{dn^k}$, la fórmula mide al menos t: exponencial.

Primer intento: copiar la recursión

$$\varphi_{c_1,c_2,t} \;=\; \exists m \left[\varphi_{c_1,m,t/2} \;\wedge\; \varphi_{m,c_2,t/2} \right]$$

- Dice lo correcto. El $\exists m$ reemplaza al ciclo que probaba todos los puntos medios.
- Pero la fórmula para t contiene **dos** copias de la fórmula para $t/2$.
- Cada nivel duplica el tamaño. Con $t = 2^{dn^k}$, la fórmula mide al menos t: exponencial.

Dónde falla

El algoritmo de Savitch reutilizaba el espacio. Una fórmula no reutiliza: lo escrito dos veces ocupa dos veces.

El truco del $\forall$

Escribir la subfórmula **una sola vez**, y exigir que valga para los dos pares:

$$\varphi_{c_1,c_2,t} \;=\; \exists m \;\; \forall (c_3, c_4) \in \{(c_1, m),\, (m, c_2)\} \;\; \left[\, \varphi_{c_3,c_4,t/2} \,\right]$$

El truco del $\forall$

Escribir la subfórmula **una sola vez**, y exigir que valga para los dos pares:

$$\varphi_{c_1,c_2,t} \;=\; \exists m \;\; \forall (c_3, c_4) \in \{(c_1, m),\, (m, c_2)\} \;\; \left[\, \varphi_{c_3,c_4,t/2} \,\right]$$

Con cuantificadores de verdad:

$$\exists m \; \forall c_3 \; \forall c_4 \left[\big((c_3 = c_1 \wedge c_4 = m) \vee (c_3 = m \wedge c_4 = c_2) \big) \rightarrow \varphi_{c_3,c_4,t/2} \right]$$

El truco del $\forall$

Escribir la subfórmula **una sola vez**, y exigir que valga para los dos pares:

$$\varphi_{c_1,c_2,t} \;=\; \exists m \;\; \forall (c_3, c_4) \in \{(c_1, m),\, (m, c_2)\} \;\; \left[\, \varphi_{c_3,c_4,t/2} \,\right]$$

Con cuantificadores de verdad:

$$\exists m \; \forall c_3 \; \forall c_4 \left[\big((c_3 = c_1 \wedge c_4 = m) \vee (c_3 = m \wedge c_4 = c_2) \big) \rightarrow \varphi_{c_3,c_4,t/2} \right]$$

- Cada configuración es un grupo de $O(n^k)$ variables booleanas, como una fila del tableau.
- El caso base, $t = 1$, se escribe con las ventanas de Cook-Levin, para dos filas.

El truco del $\forall$

Escribir la subfórmula **una sola vez**, y exigir que valga para los dos pares:

$$\varphi_{c_1,c_2,t} \;=\; \exists m \;\; \forall (c_3, c_4) \in \{(c_1, m),\, (m, c_2)\} \;\; \left[\, \varphi_{c_3,c_4,t/2} \,\right]$$

Con cuantificadores de verdad:

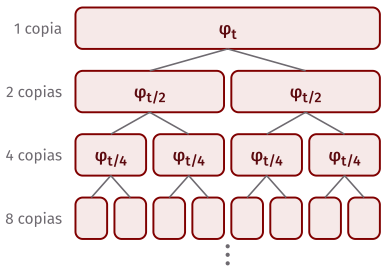
$$\exists m \; \forall c_3 \; \forall c_4 \left[\big((c_3 = c_1 \wedge c_4 = m) \vee (c_3 = m \wedge c_4 = c_2) \big) \rightarrow \varphi_{c_3,c_4,t/2} \right]$$

- Cada configuración es un grupo de $O(n^k)$ variables booleanas, como una fila del tableau.
- El caso base, $t = 1$, se escribe con las ventanas de Cook-Levin, para dos filas.
- Fórmula final: $\varphi_{c_{\text{start}},\, c_{\text{accept}},\, 2^{dn^k}}$. Es verdadera si y solo si M acepta w.

El tamaño: sumar en lugar de duplicar

Primer intento: dos copias

$\varphi_{c_1,\, c_2,\, t} = \exists m\, [\, \varphi_{c_1,\, m,\, t/2} \wedge \varphi_{m,\, c_2,\, t/2}\,]$



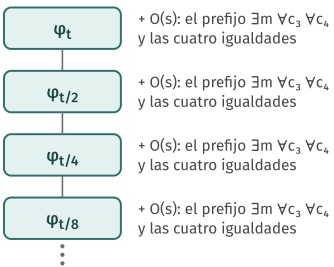
Cada nivel duplica lo escrito.

$S(t) = 2 \cdot S(t/2) + O(s)$

tamaño $\geq t = 2^{ds}$: exponencial

Con el $\forall$: una sola copia

$\varphi_{c_1,\, c_2,\, t} = \exists m\, \forall (c_3, c_4) \in \{(c_1, m), (m, c_2)\}\, [\, \varphi_{c_3,\, c_4,\, t/2}\,]$



Cada nivel suma su prefijo.

$S(t) = S(t/2) + O(s)$

$(ds + 1)$ niveles $\cdot\, O(s) = O(s^2)$: polinomial

Qué hace cada cuantificador

Con $s = n^k$ y $S(t)$ el tamaño de $\varphi_{c_1,c_2,t}$:

- El $\exists m$ reemplaza al ciclo de CANYIELD que prueba todos los puntos medios.

Qué hace cada cuantificador

Con $s = n^k$ y $S(t)$ el tamaño de $\varphi_{c_1,c_2,t}$:

- El $\exists m$ reemplaza al ciclo de CANYIELD que prueba todos los puntos medios.
- El $\forall(c_3, c_4)$ dice las dos mitades con **una** copia: $S(t) = S(t/2) + O(s)$.

Qué hace cada cuantificador

Con $s = n^k$ y $S(t)$ el tamaño de $\varphi_{c_1,c_2,t}$:

- El $\exists m$ reemplaza al ciclo de CANYIELD que prueba todos los puntos medios.
- El $\forall(c_3, c_4)$ dice las dos mitades con **una** copia: $S(t) = S(t/2) + O(s)$.
- Hay $ds + 1$ niveles, y cada uno suma $O(s)$: tamaño $O(s^2) = O(n^{2k})$, polinomial.

Qué hace cada cuantificador

Con $s = n^k$ y $S(t)$ el tamaño de $\varphi_{c_1,c_2,t}$:

- El $\exists m$ reemplaza al ciclo de CANYIELD que prueba todos los puntos medios.
- El $\forall(c_3, c_4)$ dice las dos mitades con **una** copia: $S(t) = S(t/2) + O(s)$.
- Hay $ds + 1$ niveles, y cada uno suma $O(s)$: tamaño $O(s^2) = O(n^{2k})$, polinomial.

Idea clave

Es la recursión de Savitch escrita como fórmula. El $\exists$ prueba todos los puntos medios. El $\forall$ resuelve las dos mitades "en el mismo espacio": una sola copia de la subfórmula.

La demostración completa, con la inducción y el paso a forma prenex, está en los apuntes.

SECCIÓN 06

Juegos y geografía generalizada

06

Un juego es una fórmula cuantificada

El **juego de la fórmula** $\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \psi$:

- El jugador **E** elige el valor de las variables con $\exists$; el jugador **A**, el de las variables con $\forall$.
- Se elige en el orden del prefijo. Al final, si ψ es verdadera gana E; si no, gana A.

Un juego es una fórmula cuantificada

El **juego de la fórmula** $\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \psi$:

- El jugador **E** elige el valor de las variables con $\exists$; el jugador **A**, el de las variables con $\forall$.
- Se elige en el orden del prefijo. Al final, si ψ es verdadera gana E; si no, gana A.

Teorema

E tiene una **estrategia ganadora** *(winning strategy)* si y solo si φ es verdadera.

Un juego es una fórmula cuantificada

El **juego de la fórmula** $\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \psi$:

- El jugador **E** elige el valor de las variables con $\exists$; el jugador **A**, el de las variables con $\forall$.
- Se elige en el orden del prefijo. Al final, si ψ es verdadera gana E; si no, gana A.

Teorema

E tiene una **estrategia ganadora** *(winning strategy)* si y solo si φ es verdadera.

Por inducción sobre el número de cuantificadores: es la definición de $\exists$ y de $\forall$.

Un juego es una fórmula cuantificada

El **juego de la fórmula** $\varphi = \exists x_1 \forall x_2 \exists x_3 \cdots \psi$:

- El jugador **E** elige el valor de las variables con $\exists$; el jugador **A**, el de las variables con $\forall$.
- Se elige en el orden del prefijo. Al final, si ψ es verdadera gana E; si no, gana A.

Teorema

E tiene una **estrategia ganadora** *(winning strategy)* si y solo si φ es verdadera.

Por inducción sobre el número de cuantificadores: es la definición de $\exists$ y de $\forall$.

Y al revés: "el primer jugador gana en a lo sumo tres jugadas"

$\exists j_1 \, \forall j_2 \, \exists j_3 \, \left[\, j_1 \text{ gana } \vee \ (j_2 \text{ es legal} \rightarrow j_3 \text{ es legal y gana}) \, \right]$
$\exists$ es "mi jugada". $\forall$ es "cualquier jugada tuya".

La estrategia se lee en el árbol

terminal

```
$ build/eval_qbf --tree machines/three_moves.qbf
machines/three_moves.qbf: ∃x ∀y ∃z ((x ∨ y) ∧ (¬y ∨ z) ∧ (¬x ∨ ¬z ∨ y))
x = 0
  y = 0
    z = 0   ψ = falso
    z = 1   ψ = falso
    ∃z: falso
  ∀y: falso
x = 1
  y = 0
    z = 0   ψ = verdadero
    ∃z: verdadero
  y = 1
    z = 0   ψ = falso
    z = 1   ψ = verdadero
    ∃z: verdadero
  ∀y: verdadero
∃x: verdadero
valor: verdadero   llamadas: 11   llamadas activas a la vez: 4
```

La estrategia de E:

- Empezar con $x = 1$.
- Si A elige $y = 0$: responder $z = 0$.
- Si A elige $y = 1$: responder $z = 1$.

La estrategia se lee en el árbol

terminal

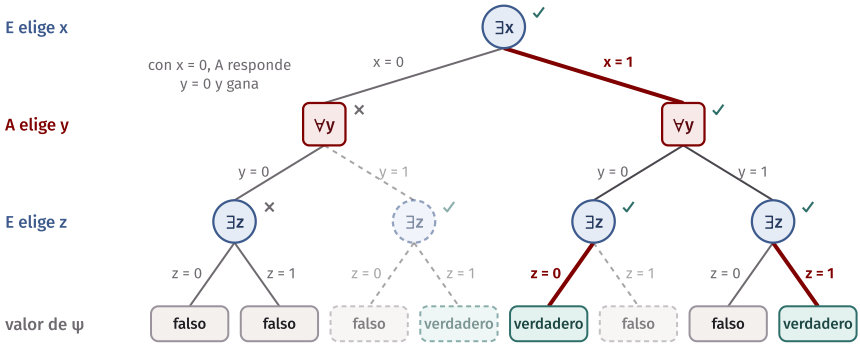
```
$ build/eval_qbf --tree machines/three_moves.qbf
machines/three_moves.qbf: ∃x ∀y ∃z ((x ∨ y) ∧ (¬y ∨ z) ∧ (¬x ∨ ¬z ∨ y))
x = 0
  y = 0
    z = 0   ψ = falso
    z = 1   ψ = falso
    ∃z: falso
  ∀y: falso
x = 1
  y = 0
    z = 0   ψ = verdadero
    ∃z: verdadero
  y = 1
    z = 0   ψ = falso
    z = 1   ψ = verdadero
    ∃z: verdadero
  ∀y: verdadero
∃x: verdadero
valor: verdadero   llamadas: 11   llamadas activas a la vez: 4
```

La estrategia de E:

- Empezar con $x = 1$.
- Si A elige $y = 0$: responder $z = 0$.
- Si A elige $y = 1$: responder $z = 1$.

Con $x = 0$ pierde: A responde $y = 0$ y ninguna z arregla $(x \vee y)$.

El árbol del juego

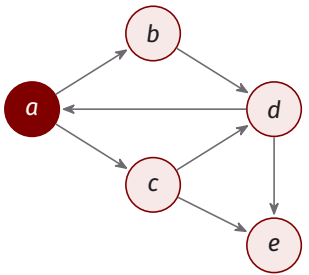


Geografía generalizada

Geografía: decir ciudades; cada una empieza con la letra en que terminó la anterior; sin repetir.

Generalizada *(generalized geography)*: cualquier grafo dirigido.

- La ficha empieza en el nodo de partida, que cuenta como **visitado**.
- Por turnos, se mueve por una arista a un nodo **no visitado**.
- Quien no puede mover, **pierde**.

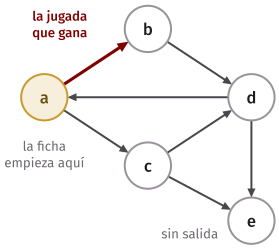


`machines/small_game.graph`, partida en *a*

$$\mathrm{GG} = \{\, \langle G, b\rangle : \text{el primer jugador tiene una estrategia ganadora en } G \text{ partiendo de } b \,\}$$

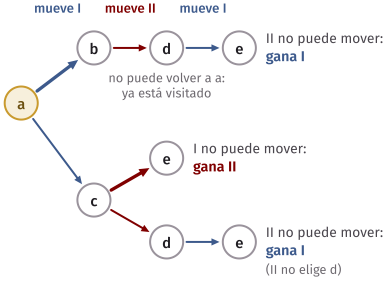
Un ejemplo a mano

El grafo



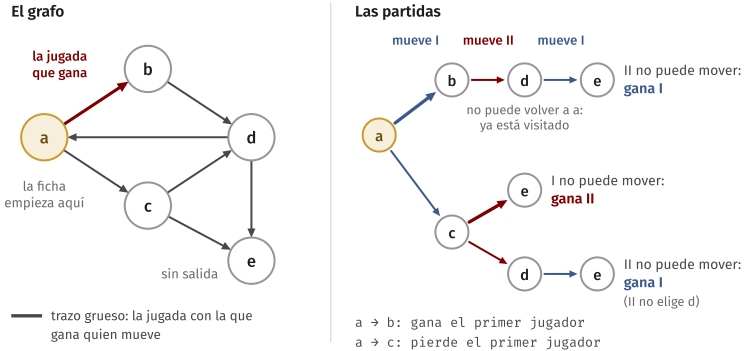
trazo grueso: la jugada con la que gana quien mueve

Las partidas




```
a → b: gana el primer jugador
a → c: pierde el primer jugador
```

Un ejemplo a mano



`build/geography --moves machines/small_game.graph` responde lo mismo: una primera jugada ganadora es `b`. Examina 4 posiciones y nunca guarda un camino de más de 4 nodos.

Con ciudades de verdad

Nueve ciudades; una arista cuando la segunda empieza con la letra en que termina la primera.

`lima` → `arequipa`, `abancay`, `ayacucho`, `andahuaylas`. `abancay` → `yurimaguas`. …

terminal

```
$ build/geography --moves machines/cities.graph
9 nodos, 22 aristas, empieza en lima
  lima → arequipa: pierde el primer jugador
  lima → abancay: gana el primer jugador
  lima → ayacucho: gana el primer jugador
  lima → andahuaylas: pierde el primer jugador
el primer jugador gana; una primera jugada ganadora: abancay
posiciones examinadas: 30   camino más largo guardado: 9 nodos
```

Quien empieza en Lima gana con Abancay o Ayacucho, y pierde con Arequipa o Andahuaylas, si el otro juega bien.

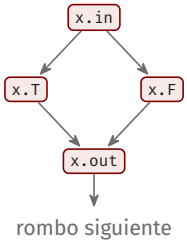
GG está en PSPACE: se guarda solo el camino actual

geography.h

```cpp
inline bool mover_wins(const Graph& g, int current, std::vector<bool>& visited,
                       int path_length, GameStats& stats) {
    stats.calls++;
    stats.max_path = std::max(stats.max_path, path_length);
    for (int next : g.edges[current]) {
        if (visited[next]) {
            continue;
        }
        visited[next] = true;
        bool rival_wins = mover_wins(g, next, visited, path_length + 1, stats);
        visited[next] = false;
        if (!rival_wins) {
            return true;
        }
    }
    return false;
}
```

PSPACE-difícil: de una fórmula a un grafo

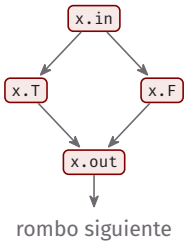
Reducción de TQBF a GG. De $\varphi = \exists x_1 \forall x_2 \cdots \exists x_k \, \psi$, con ψ en CNF, se construye un grafo.



1. **Un rombo por variable**, encadenados. Pasar por `x.T` es elegir x verdadera; por `x.F`, falsa.

PSPACE-difícil: de una fórmula a un grafo

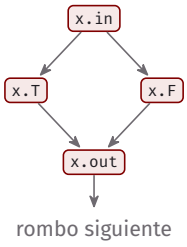
Reducción de TQBF a GG. De $\varphi = \exists x_1 \forall x_2 \cdots \exists x_k \, \psi$, con ψ en CNF, se construye un grafo.



1. **Un rombo por variable**, encadenados. Pasar por `x.T` es elegir x verdadera; por `x.F`, falsa.
2. **La matriz.** Del último rombo, al nodo `matrix`; de ahí, a un nodo por cláusula; de cada cláusula, a un nodo por literal.

PSPACE-difícil: de una fórmula a un grafo

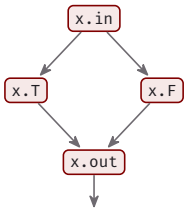
Reducción de TQBF a GG. De $\varphi = \exists x_1 \forall x_2 \cdots \exists x_k \, \psi$, con ψ en CNF, se construye un grafo.



1. **Un rombo por variable**, encadenados. Pasar por `x.T` es elegir x verdadera; por `x.F`, falsa.
2. **La matriz.** Del último rombo, al nodo `matrix`; de ahí, a un nodo por cláusula; de cada cláusula, a un nodo por literal.
3. **Las aristas de regreso.** De cada literal, al nodo de su rombo que lo hace **verdadero**: `x.T` para x, `x.F` para $\neg x$.

PSPACE-difícil: de una fórmula a un grafo

Reducción de TQBF a GG. De $\varphi = \exists x_1 \forall x_2 \cdots \exists x_k \, \psi$, con ψ en CNF, se construye un grafo.



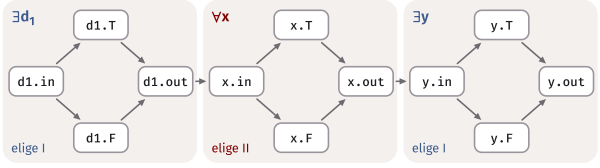
1. **Un rombo por variable**, encadenados. Pasar por `x.T` es elegir x verdadera; por `x.F`, falsa.
2. **La matriz.** Del último rombo, al nodo `matrix`; de ahí, a un nodo por cláusula; de cada cláusula, a un nodo por literal.
3. **Las aristas de regreso.** De cada literal, al nodo de su rombo que lo hace **verdadero**: `x.T` para x, `x.F` para $\neg x$.

Si el prefijo no alterna, o no empieza y termina con $\exists$, se intercalan variables que no aparecen en ψ. El grafo tiene $4k + 1$ nodos, más uno por cláusula y uno por literal.

El grafo de $\exists d_1 \, \forall x \, \exists y \; ((x \vee y) \wedge (\neg x \vee \neg y))$

$\exists d_1 \, \forall x \, \exists y \; ((x \vee y) \wedge (\neg x \vee \neg y))$

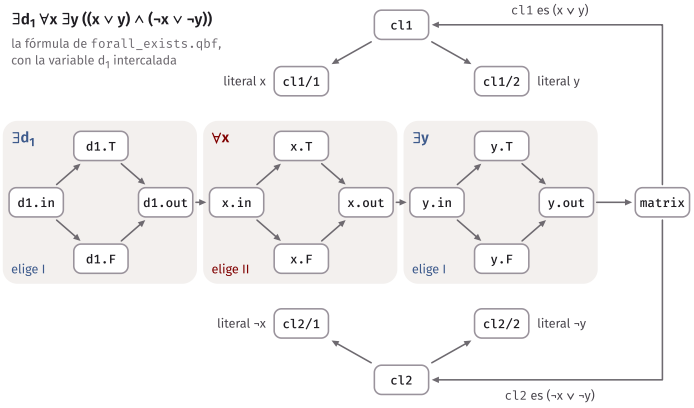
la fórmula de `forall_exists.qbf`,
con la variable d_1 intercalada



Un rombo por variable, encadenados en el orden del prefijo.

Pasar por `x.T` es elegir x verdadera; por `x.F`, falsa. La partida empieza en `d1.in`.

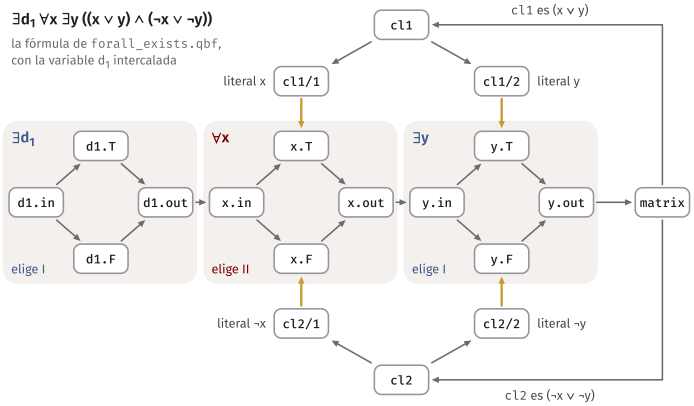
El grafo de $\exists d_1\, \forall x\, \exists y\ ((x \vee y) \wedge (\neg x \vee \neg y))$



La matriz: del último rombo se va a `matrix`; de ahí, a un nodo por cláusula,

y de cada cláusula, a un nodo por cada uno de sus literales.

El grafo de $\exists d_1 \, \forall x \, \exists y \; ((x \vee y) \wedge (\neg x \vee \neg y))$



Las aristas de regreso: de cada literal, al nodo del rombo que lo hace verdadero.

El literal x va a a `x.T` y el literal ¬x, a `x.F`. En total, 19 nodos y 25 aristas.

Quién elige en cada rombo

Jugada	Mueve	De dónde a dónde	¿Elige?
1	I	`x1.in` $\longrightarrow$ `x1.T` o `x1.F`	Sí: el valor de x_1
2	II	$\longrightarrow$ `x1.out`	No
3	I	$\longrightarrow$ `x2.in`	No
4	II	`x2.in` $\longrightarrow$ `x2.T` o `x2.F`	Sí: el valor de x_2
5	I	$\longrightarrow$ `x2.out`	No
6	II	$\longrightarrow$ `x3.in`	No
7	I	`x3.in` $\longrightarrow$ `x3.T` o `x3.F`	Sí: el valor de x_3

Quién elige en cada rombo

Jugada	Mueve	De dónde a dónde	¿Elige?
1	I	$\texttt{x1.in} \to \texttt{x1.T}$ o $\texttt{x1.F}$	Sí: el valor de x_1
2	II	$\to \texttt{x1.out}$	No
3	I	$\to \texttt{x2.in}$	No
4	II	$\texttt{x2.in} \to \texttt{x2.T}$ o $\texttt{x2.F}$	Sí: el valor de x_2
5	I	$\to \texttt{x2.out}$	No
6	II	$\to \texttt{x3.in}$	No
7	I	$\texttt{x3.in} \to \texttt{x3.T}$ o $\texttt{x3.F}$	Sí: el valor de x_3

- Cada rombo consume **tres** jugadas: quien elige se alterna.
- I elige las variables con $\exists$ y II las variables con $\forall$: I hace de E, II hace de A.

Quién elige en cada rombo

Jugada	Mueve	De dónde a dónde	¿Elige?
1	I	`x1.in` $\to$ `x1.T` o `x1.F`	Sí: el valor de x_1
2	II	$\to$ `x1.out`	No
3	I	$\to$ `x2.in`	No
4	II	`x2.in` $\to$ `x2.T` o `x2.F`	Sí: el valor de x_2
5	I	$\to$ `x2.out`	No
6	II	$\to$ `x3.in`	No
7	I	`x3.in` $\to$ `x3.T` o `x3.F`	Sí: el valor de x_3

- Cada rombo consume **tres** jugadas: quien elige se alterna.
- I elige las variables con $\exists$ y II las variables con $\forall$: I hace de E, II hace de A.
- Al salir de los rombos, los nodos visitados dicen qué asignación se eligió.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.
2. **I elige un literal de esa cláusula:** “este literal es verdadero”.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.
2. **I elige un literal de esa cláusula:** “este literal es verdadero”.
3. **II debe mover** al nodo del rombo que hace verdadero al literal.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.
2. **I elige un literal de esa cláusula:** “este literal es verdadero”.
3. **II debe mover** al nodo del rombo que hace verdadero al literal.

El literal es verdadero

Ese nodo ya está visitado. II no puede mover: **pierde II**.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.
2. **I elige un literal de esa cláusula:** “este literal es verdadero”.
3. **II debe mover** al nodo del rombo que hace verdadero al literal.

El literal es verdadero

Ese nodo ya está visitado. II no puede mover: **pierde II**.

El literal es falso

II mueve ahí. La única arista va a un `x.out` ya visitado: **pierde I**.

El final de la partida

Después del último rombo, I mueve a `matrix`. Siguen tres jugadas:

1. **II elige una cláusula:** “esta cláusula es falsa”.
2. **I elige un literal de esa cláusula:** “este literal es verdadero”.
3. **II debe mover** al nodo del rombo que hace verdadero al literal.

El literal es verdadero

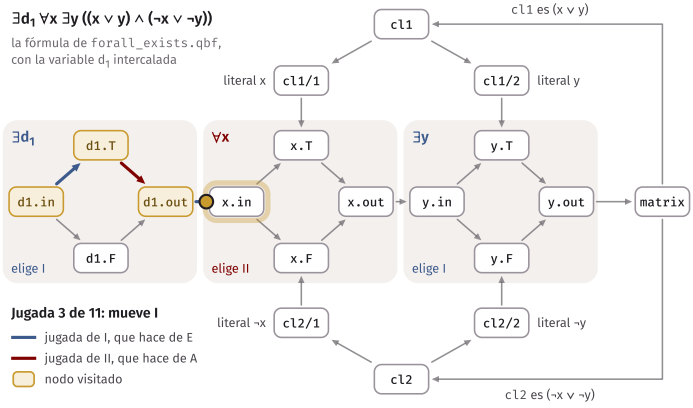
Ese nodo ya está visitado. II no puede mover: **pierde II**.

El literal es falso

II mueve ahí. La única arista va a un `x.out` ya visitado: **pierde I**.

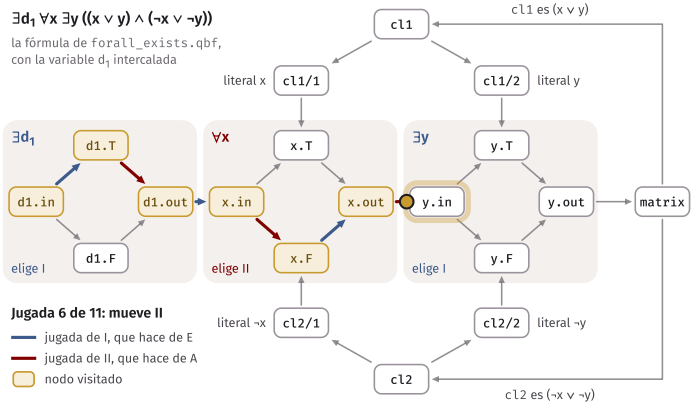
φ verdadera: E lleva a ψ verdadera, toda cláusula tiene un literal verdadero, y I gana.
φ falsa: A lleva a ψ falsa, II elige una cláusula con todos sus literales falsos, y I pierde.

Una partida completa



Jugadas 1 a 3. I elige `d1.T`, que da igual. Las dos jugadas siguientes son obligadas,
y en `x.in` le toca a II: el rombo de la variable con ∀ lo decide II.

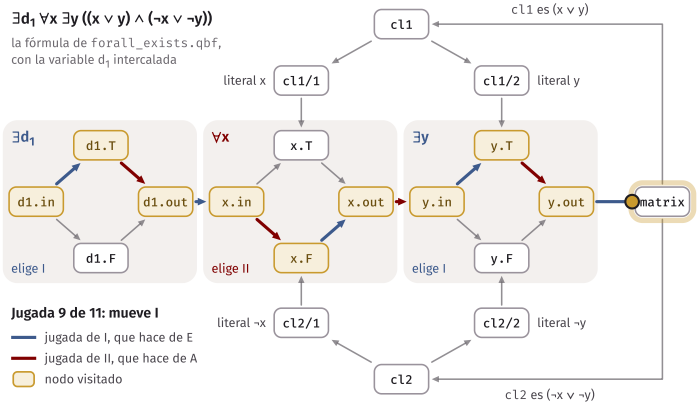
Una partida completa



Jugadas 4 a 6. II elige `x.F`: x falsa.

Otra vez dos jugadas obligadas, y en `y.in` le toca a I.

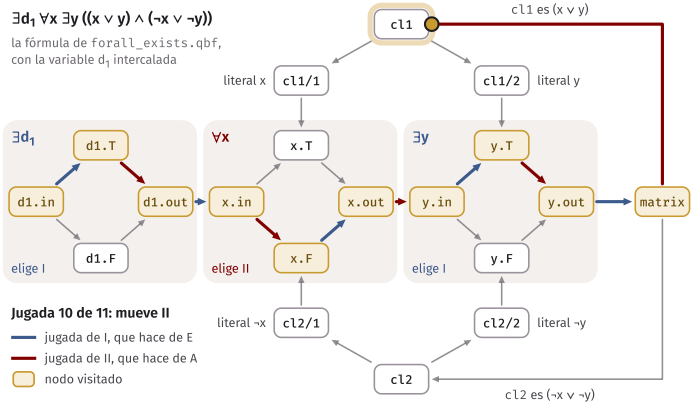
Una partida completa



Jugadas 7 a 9. I responde `y.T`: y verdadera, la contraria de x.

Después II va a `y.out` e I, a `matrix`. Los nodos visitados dicen la asignación: x = 0, y = 1.

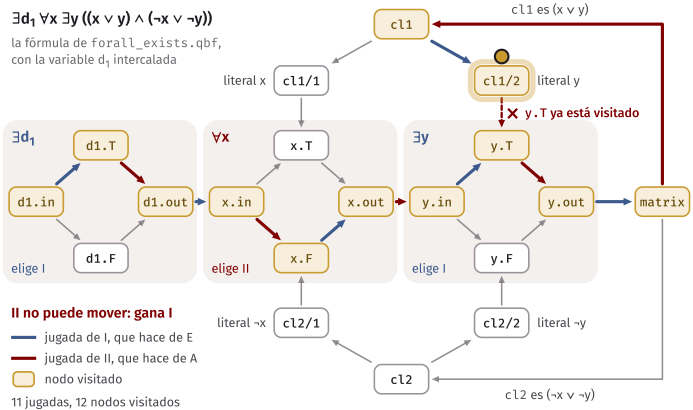
Una partida completa



Jugada 10. II elige una cláusula: `cl1`, que es (x ∨ y).

Es su manera de decir «esta cláusula es falsa».

Una partida completa



Jugada 11. I elige `cl1/2`, el literal y: «este literal es verdadero».

La única arista de `cl1/2` va a a `y.T`, que ya está visitado: II no puede mover y pierde.

La reducción, ejecutada

terminal

```
$ build/formula_to_geography machines/forall_exists.qbf | build/geography -
19 nodos, 25 aristas, empieza en d1.in
el primer jugador gana; una primera jugada ganadora: d1.T
posiciones examinadas: 38   camino más largo guardado: 13 nodos
$ build/formula_to_geography machines/exists_forall.qbf | build/geography -
19 nodos, 25 aristas, empieza en y.in
el primer jugador no tiene estrategia ganadora
posiciones examinadas: 61   camino más largo guardado: 13 nodos
```

La fórmula verdadera da un grafo que gana el primer jugador; la falsa, uno que pierde.

Error común: leer la reducción al revés

Para mostrar que GG es difícil se **parte** de TQBF y se construye una instancia de GG.

Juegos y rompecabezas

Rompecabezas: un solo $\exists$

"Tiene solución" se demuestra mostrándola: una asignación, un camino. Corta, y se revisa rápido. Eso es NP.

Juegos y rompecabezas

Rompecabezas: un solo $\exists$

"Tiene solución" se demuestra mostrándola: una asignación, un camino. Corta, y se revisa rápido. Eso es NP.

Juego: $\exists$ y $\forall$ alternados

"Gano yo" no se demuestra con una lista de jugadas: las mías dependen de las tuyas. Hace falta una **estrategia**: un árbol entero.

Juegos y rompecabezas

Rompecabezas: un solo $\exists$

"Tiene solución" se demuestra mostrándola: una asignación, un camino. Corta, y se revisa rápido. Eso es NP.

Juego: $\exists$ y $\forall$ alternados

"Gano yo" no se demuestra con una lista de jugadas: las mías dependen de las tuyas. Hace falta una **estrategia**: un árbol entero.

- El árbol crece exponencialmente con la duración de la partida.
- Nadie sabe demostrar que no haya certificados cortos: sería demostrar NP $\neq$ PSPACE.

Juegos y rompecabezas

Rompecabezas: un solo $\exists$

"Tiene solución" se demuestra mostrándola: una asignación, un camino. Corta, y se revisa rápido. Eso es NP.

Juego: $\exists$ y $\forall$ alternados

"Gano yo" no se demuestra con una lista de jugadas: las mías dependen de las tuyas. Hace falta una **estrategia**: un árbol entero.

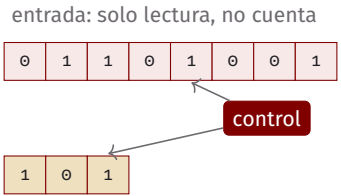
- El árbol crece exponencialmente con la duración de la partida.
- Nadie sabe demostrar que no haya certificados cortos: sería demostrar NP $\neq$ PSPACE.
- Hex, damas, go y ajedrez, en tableros de $n \times n$, son PSPACE-difíciles.

SECCIÓN 07

L y NL

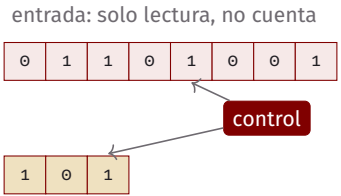
07

Menos espacio que la entrada



- Dos cintas: la de **entrada**, de solo lectura, y la de **trabajo**.
- El espacio es lo que se usa de la cinta de **trabajo**.

Menos espacio que la entrada

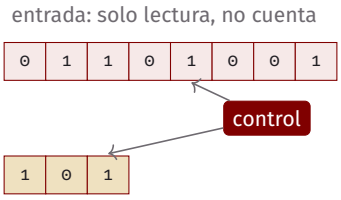


- Dos cintas: la de **entrada**, de solo lectura, y la de **trabajo**.
- El espacio es lo que se usa de la cinta de **trabajo**.

L y NL

$\mathsf{L} = \mathsf{SPACE}(\log n)$
$\mathsf{NL} = \mathsf{NSPACE}(\log n)$

Menos espacio que la entrada



- Dos cintas: la de **entrada**, de solo lectura, y la de **trabajo**.
- El espacio es lo que se usa de la cinta de **trabajo**.

L y NL

$\mathrm{L} = \mathrm{SPACE}(\log n)$
$\mathrm{NL} = \mathrm{NSPACE}(\log n)$

Espacio logarítmico *(logarithmic space)*: una cantidad **constante de contadores y de posiciones** de la entrada. Unas cuantas variables enteras, y ningún arreglo.

Qué cabe en espacio logarítmico

- $\{0^k 1^k\}$ está en L: no se puede tachar, porque la entrada no se escribe. Se **cuenta**: dos contadores binarios, uno para los ceros y otro para los unos.

Qué cabe en espacio logarítmico

- $\{0^k 1^k\}$ está en L: no se puede tachar, porque la entrada no se escribe. Se **cuenta**: dos contadores binarios, uno para los ceros y otro para los unos.
- Los palíndromos están en L: dos posiciones, i y $n + 1 - i$.

Qué cabe en espacio logarítmico

- $\{0^k 1^k\}$ está en L: no se puede tachar, porque la entrada no se escribe. Se **cuenta**: dos contadores binarios, uno para los ceros y otro para los unos.
- Los palíndromos están en L: dos posiciones, i y $n + 1 - i$.
- Configuraciones: estado, cinta de trabajo y dos cabezales. Con espacio $O(\log n)$ son $n \cdot 2^{O(\log n)}$: una cantidad **polinomial**. Por eso $\mathsf{L} \subseteq \mathsf{P}$.

Qué cabe en espacio logarítmico

- $\{0^k 1^k\}$ está en L: no se puede tachar, porque la entrada no se escribe. Se **cuenta**: dos contadores binarios, uno para los ceros y otro para los unos.
- Los palíndromos están en L: dos posiciones, i y $n + 1 - i$.
- Configuraciones: estado, cinta de trabajo y dos cabezales. Con espacio $O(\log n)$ son $n \cdot 2^{O(\log n)}$: una cantidad **polinomial**. Por eso L $\subseteq$ P.

Error común

Contar la entrada como espacio usado. En L y NL solo cuenta la cinta de trabajo. Y el error contrario: creer que la máquina no puede leer toda su entrada. La lee completa, las veces que quiera. Lo que no puede es *recordarla*.

PATH está en NL

$\mathrm{PATH} = \{ \langle G, s, t \rangle : G$ es un grafo dirigido con un camino de s a $t \}$. Está en P. ¿Está en L?

- BFS marca los nodos visitados: un bit por nodo. Espacio lineal, no logarítmico.

PATH está en NL

$\mathrm{PATH} = \{ \langle G, s, t \rangle : G$ es un grafo dirigido con un camino de s a $t \}$. Está en P. ¿Está en L?

- BFS marca los nodos visitados: un bit por nodo. Espacio lineal, no logarítmico.

No determinista: adivinar el camino, nodo por nodo. G tiene m nodos.

1. $actual \leftarrow s$, $pasos \leftarrow 0$.
2. Mientras $actual \neq t$: si $pasos = m - 1$, *rechazar*.
3. Elegir un nodo v. Si no hay arista de $actual$ a v, *rechazar*.
4. $actual \leftarrow v$, $pasos \leftarrow pasos + 1$.
5. *Aceptar*.

PATH está en NL

$\mathrm{PATH} = \{ \langle G, s, t \rangle : G$ es un grafo dirigido con un camino de s a $t \}$. Está en P. ¿Está en L?

- BFS marca los nodos visitados: un bit por nodo. Espacio lineal, no logarítmico.

No determinista: adivinar el camino, nodo por nodo. G tiene m nodos.

1. $actual \leftarrow s$, $pasos \leftarrow 0$.
2. Mientras $actual \neq t$: si $pasos = m - 1$, *rechazar*.
3. Elegir un nodo v. Si no hay arista de $actual$ a v, *rechazar*.
4. $actual \leftarrow v$, $pasos \leftarrow pasos + 1$.
5. *Aceptar*.

Qué guarda

El nodo actual y un contador: $O(\log n)$ celdas. No recuerda el camino. El contador reemplaza a las marcas: si hay un camino, hay uno de a lo sumo $m - 1$ aristas.

Una rama de esa máquina

Las elecciones van en la línea de comandos. Grafo: $s \to a$, $a \to b$, $b \to a$, $b \to c$, $c \to t$, $u \to s$, $u \to t$.

terminal

```
$ build/guess_path machines/two_routes.graph t a b c t
paso 0: en s
paso 1: en a
paso 2: en b
paso 3: en c
paso 4: en t
acepta: llegó a t
memoria de trabajo: 2 números de 3 bits, para un grafo de 6 nodos
$ build/guess_path machines/two_routes.graph t a b a b a b
paso 0: en s
paso 1: en a
paso 2: en b
paso 3: en a
paso 4: en b
paso 5: en a
rechaza: ya dio 5 pasos, y un camino sin repetir nodos no tiene más
```

- Una rama que acierta, y una que da vueltas en el ciclo: la corta el contador.

PATH es NL-completo

Reducción en espacio logarítmico, $A \leq_L B$

Entrada de solo lectura, $O(\log n)$ celdas de trabajo y una cinta de salida en la que solo se escribe. Con $\leq_p$ no serviría: la reducción resolvería el problema.

PATH es NL-completo

Reducción en espacio logarítmico, $A \leq_L B$

Entrada de solo lectura, $O(\log n)$ celdas de trabajo y una cinta de salida en la que solo se escribe. Con $\leq_p$ no serviría: la reducción resolvería el problema.

Idea. Sea N una máquina de NL. Con la entrada w, su **grafo de configuraciones**:

- nodos: las configuraciones de N con w, una cantidad polinomial;
- aristas: de c_1 a c_2 si N pasa de una a otra en un paso;

PATH es NL-completo

Reducción en espacio logarítmico, $A \leq_L B$

Entrada de solo lectura, $O(\log n)$ celdas de trabajo y una cinta de salida en la que solo se escribe. Con $\leq_p$ no serviría: la reducción resolvería el problema.

Idea. Sea N una máquina de NL. Con la entrada w, su **grafo de configuraciones**:

- nodos: las configuraciones de N con w, una cantidad polinomial;
- aristas: de c_1 a c_2 si N pasa de una a otra en un paso;
- N acepta w si y solo si hay un camino de c_{start} a c_{accept}.

PATH es NL-completo

Reducción en espacio logarítmico, $A \leq_L B$

Entrada de solo lectura, $O(\log n)$ celdas de trabajo y una cinta de salida en la que solo se escribe. Con $\leq_p$ no serviría: la reducción resolvería el problema.

Idea. Sea N una máquina de NL. Con la entrada w, su **grafo de configuraciones**:

- nodos: las configuraciones de N con w, una cantidad polinomial;
- aristas: de c_1 a c_2 si N pasa de una a otra en un paso;
- N acepta w si y solo si hay un camino de c_{start} a c_{accept}.

Idea clave

Una máquina de NL con una entrada *es* un grafo de tamaño polinomial. De ahí también NL $\subseteq$ P: BFS.

Dónde quedan L y NL

- **¿PATH está en L?** No se sabe. Es la pregunta $\mathrm{L} = \mathrm{NL}$.

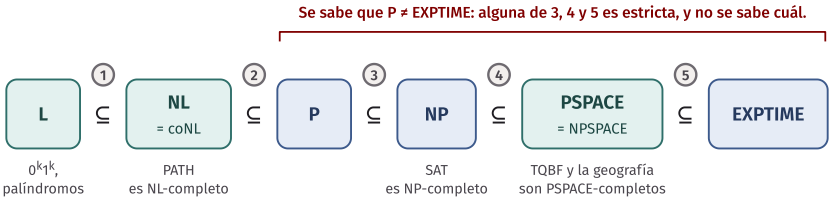
Dónde quedan L y NL

- **¿PATH está en L?** No se sabe. Es la pregunta $\mathrm{L} = \mathrm{NL}$.
- **Savitch**, para $f(n) \geq \log n$: $\mathrm{NL} \subseteq \mathrm{SPACE}((\log n)^2)$. Para PATH, la máquina es `can_yield`.

Dónde quedan L y NL

- **¿PATH está en L?** No se sabe. Es la pregunta $\mathsf{L} = \mathsf{NL}$.
- **Savitch**, para $f(n) \geq \log n$: $\mathsf{NL} \subseteq \mathsf{SPACE}((\log n)^2)$. Para PATH, la máquina es `can_yield`.
- **Teorema (Immerman y Szelepcsényi):** $\mathsf{NL} = \mathsf{coNL}$. Solo el enunciado.
  Una máquina de NL puede convencerse de que **no** hay camino de s a t.
- En espacio, el complemento no cuesta. En NP no se sabe: NP frente a coNP.

El mapa de las clases



Por qué vale cada inclusión — clase de espacio — clase de tiempo

1	**L ⊆ NL:**	una máquina determinista es una no determinista que nunca elige.
2	**NL ⊆ P:**	el grafo de configuraciones tiene tamaño polinomial, y se recorre con BFS.
3	**P ⊆ NP:**	la misma razón que en 1.
4	**NP ⊆ PSPACE:**	se prueban todos los certificados, uno por uno, en las mismas celdas.
5	**PSPACE ⊆ EXPTIME:**	si se detiene, no repite configuraciones, y hay una cantidad exponencial.

De cada inclusión, por separado, no se sabe si es estricta.

Tres veces el mismo objeto

	El grafo de configuraciones	**Qué se hace con él**
Savitch	Tamaño $2^{O(f(n))}$	Recursión del punto medio: espacio $f(n)^2$
TQBF	Tamaño $2^{O(n^k)}$	La misma recursión, escrita como fórmula
PATH y NL	Tamaño polinomial	Es la instancia de PATH

La semana en una línea

El espacio se reutiliza. Una máquina con poco espacio es un grafo de configuraciones, y aceptar es **alcanzar** un nodo desde otro.

Laboratorio: Práctica 4, y un ejercicio opcional

La sesión de laboratorio es para la **Práctica 4**. Como ejercicio opcional, `lab/geography`:

1. Leer un grafo dirigido y un nodo de partida.
2. Decidir si el primer jugador tiene una estrategia ganadora, con una búsqueda recursiva que guarda solo el camino actual.
3. Si gana, imprimir una primera jugada ganadora. Hay un verificador: se acepta cualquiera.
4. Medir el tiempo con grafos de 10, 15 y 20 nodos, y anotar cómo crece.

Ojo con el tiempo

Dejen de probar jugadas al encontrar una ganadora: sin eso, los casos grandes dan TLE. Y usen una sola copia de las marcas de visitado, por referencia: copiarlas en cada llamada también se paga.

Para la próxima clase

- Leer los apuntes de la semana 14. La demostración completa de que TQBF es PSPACE-completo, con calma.
- Preguntas de repaso de la guía: el tope $2^{O(f(n))}$; qué guarda cada nivel de Savitch; las dos fórmulas a mano; "gana en tres jugadas" como fórmula; por qué BFS no muestra que PATH está en L.
- Próxima semana: los teoremas de jerarquía, el azar y lo que sigue abierto.

Pregunta

En $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$ alguna inclusión es estricta. ¿Cómo se podría demostrar que una máquina con más tiempo decide algo que ninguna con menos tiempo puede?