Complejidad de la lógica de primer orden y algunas de sus extensiones

IIC3263

Complejidad de la LPO

Queremos estudiar la complejidad de evaluar una consulta en LPO.

Vamos a estudiar dos nociones: Complejidad de los datos y Complejidad combinada

Pero antes vamos a repasar algunas nociones de complejidad.

- Necesitamos más clases que PTIME y NP
- Necesitamos modelos de computación paralela: Complejidad de circuitos

Para recordar: Máquinas de Turing

Definición

Máquina de Turing (determinista): $(Q, \Sigma, q_0, \delta, F)$

- Q *es un conjunto finito de estados*
- Σ *es un alfabeto finito tal que* $\vdash, \mathtt{B} \notin \Sigma$
- $q_0 \in Q$ *es el estado inicial*
- $F \subseteq Q$ *es un conjunto de estados finales*
- δ *es una función parcial:*

$$\delta \quad : \quad Q \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \to Q \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \times \{\leftarrow, \square, \rightarrow\}$$

δ *es llamada función de transición*

Máquinas de Turing: Funcionamiento

La cinta de la máquina de Turing es infinita hacia la derecha.

- El símbolo $\vdash$ es usado para demarcar la posición 0 de la cinta

Supuesto

- Si $\delta(q, \vdash)$ está definido: $\delta(q, \vdash) = (q', \vdash, X)$, con $X \in \{\square, \rightarrow\}$
- Si $a \in (\Sigma \cup \{\mathtt{B}\})$ y $\delta(q, a)$ está definido: $\delta(q, a) = (q', b, X)$, con $b \in (\Sigma \cup \{\mathtt{B}\})$

Máquinas de Turing: Funcionamiento

Σ es el alfabeto de entrada y $(\Sigma \cup \{\vdash, \mathtt{B}\})$ es el alfabeto de la cinta.

- Una palabra $w \in \Sigma^*$ de entrada de largo n es colocada en las posiciones $1, \ldots, n$ de la cinta
- Las posiciones siguientes $(n+1, n+2, \ldots)$ contienen el símbolo $\mathtt{B}$

Máquinas de Turing: Funcionamiento

Al comenzar a funcionar, la máquina se encuentra en el estado q_0 y su cabeza lectora está en la posición 1 de la cinta.

En cada instante la máquina se encuentra en un estado q y su cabeza lectora está en una posición p

- Si el símbolo en la posición p es a y $\delta(q, a) = (q', b, X)$, entonces:
  - La máquina escribe el símbolo b en la posición p de la cinta
  - Cambia de estado desde q a q'
  - Mueve la cabeza lectora a la posición $p - 1$ si X es $\leftarrow$, y a la posición $p + 1$ si X es $\rightarrow$. Si X es $\square$, entonces la cabeza lectora permanece en la posición p

La máquina acepta w si se detiene en un estado final

El lenguaje aceptado por una máquina de Turing

Definición

Dada una máquina de Turing $M = (Q, \Sigma, q_0, \delta, F)$*:*

$$L(M) \quad = \quad \{w \in \Sigma^* \mid M \text{ acepta } w\}$$

$L(M)$ es el lenguaje aceptado por M

Para recordar: Máquinas de Turing no determinista

Definición

Máquina de Turing no determinista: $(Q, \Sigma, q_0, \delta, F)$

- Q *es un conjunto finito de estados*
- Σ *es un alfabeto finito tal que* $\vdash, \mathtt{B} \notin \Sigma$
- $q_0 \in Q$ *es el estado inicial*
- $F \subseteq Q$ *es un conjunto de estados finales*
- δ *es una relación de transición:*

$$\delta \subseteq Q \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \times Q \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \times \{\leftarrow, \square, \rightarrow\}$$

Máquinas de Turing no determinista: Funcionamiento

Hacemos los mismos supuestos que para el caso determinista

- En particular sobre el uso del símbolo $\vdash$

La inicialización es igual que para el caso determinista

- Al comenzar a funcionar, la máquina se encuentra en el estado q_0 y su cabeza lectora está en la posición 1 de la cinta

Máquinas de Turing no determinista: Funcionamiento

En cada instante la máquina se encuentra en un estado q y su cabeza lectora está en una posición p que contiene un símbolo a.

- Sea $T = \{(q', b, X) \mid (q, a, q', b, X) \in \delta\}$. Si $T \neq \emptyset$, entonces la máquina elije $(q', b, X) \in T$ y:
  - escribe el símbolo b en la posición p de la cinta
  - cambia de estado desde q a q'
  - mueve la cabeza lectora a la posición $p - 1$ si X es $\leftarrow$, y a la posición $p + 1$ si X es $\rightarrow$. Si X es $\square$, entonces la cabeza lectora permanece en la posición p

Máquinas de Turing no deterministas: Lenguaje aceptado

Definición

Dada una máquina de Turing M no determinista con alfabeto Σ:

$$L(M) = \{w \in \Sigma^* \mid \text{existe alguna ejecución de } M \text{ con entrada } w \text{ que termina en un estado final}\}$$

Teorema

Para cada MT no determinista M, existe una MT determinista M' tal que $L(M) = L(M')$.

Complejidad de un algoritmo

Dado: MT determinista M con alfabeto Σ que para en todas las entradas

Definición

- *Paso de M: Ejecutar una instrucción de la función de transición*
- *$tiempo_M(w)$: Número de pasos ejecutados por M con entrada w*
- $t_M(n) = \text{máx}\{\ tiempo_M(w) \mid w \in \Sigma^*\ y\ |w| = n\}$, *para cada* $n \geq 0$

t_M: Tiempo de ejecución de M en el peor caso

Complejidad de un problema

Definición

Un lenguaje L puede ser aceptado en tiempo t si es que existe una MT determinista M tal que:

- *M para en todas las entradas*
- *$L = L(M)$*
- *t_M es $O(t)$, vale decir, existe $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tal que $t_M(n) \leq c \cdot t(n)$ para todo $n \geq n_0$*

El tiempo para computar una función f se define de la misma forma

- ¿Cómo se define una MT que calcula una función?

Clases de complejidad

Dado: Alfabeto Σ

Definición

DTIME(t): conjunto de todos los lenguajes $L \subseteq \Sigma^*$ que pueden ser aceptados en tiempo t

Dos clases fundamentales:

$$\text{PTIME} = \bigcup_{k \in \mathbb{N}} \text{DTIME}(n^k)$$

$$\text{EXPTIME} = \bigcup_{k \in \mathbb{N}} \text{DTIME}(2^{n^k})$$

PTIME: Conjunto de todos los problemas que pueden ser solucionados eficientemente

Clases de complejidad no deterministas

Dado: MT no determinista M con alfabeto Σ

Definición

- *Paso de M: Ejecutar una instrucción de la relación de transición*
- *$tiempo_M(w)$: Número de pasos de M con entrada w en la ejecución más corta que acepta a w*
- $t_M(n) = \text{máx}(\{n\} \cup \{\ tiempo_M(w) \mid w \in \Sigma^*, |w| = n \text{ y } M \text{ acepta } w\})$, *para cada* $n \geq 0$

¿Por que incluimos $\{n\}$ en la definición?

Clases de complejidad no deterministas

Definición

Un lenguaje L es aceptado en tiempo t por una MT M no determinista si:

- $L = L(M)$
- t_M *es* $O(t)$

Clases de Complejidad no deterministas

Dado: Alfabeto Σ

Definición

NTIME(t): Conjunto de todos los lenguajes que pueden ser aceptados en tiempo t por alguna MT no determinista

Dos clases fundamentales:

$$\text{NP} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(n^k)$$

$$\text{NEXPTIME} = \bigcup_{k \in \mathbb{N}} \text{NTIME}(2^{n^k})$$

Relación entre clases de complejidad

¿Cuál es la relación entre las clases de complejidad que acabamos de definir?

Fácil de demostrar	:	$\text{PTIME} \subseteq \text{NP} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME}$
No tan fácil de demostrar	:	$\text{PTIME} \subsetneq \text{EXPTIME}$
Aún sin resolver	:	$\text{¿PTIME} \subsetneq \text{NP?}$

También necesitamos clases definidas en término del espacio utilizado.

Espacio utilizado en una Máquina de Turing

Para introducir la noción de espacio utilizado en una MT tenemos que distinguir entre:

- el espacio ocupado por la entrada, y
- el espacio necesario para procesar la entrada

Para hacer esta distinción utilizamos MT con dos cintas:

- Cinta para la entrada: Sólo de lectura
- Cinta de trabajo: Como en una MT usual

El espacio utilizado se mide en términos de las celdas visitadas en la cinta de trabajo

Máquinas de Turing con cinta de trabajo

Definición

Máquina de Turing con cinta de trabajo (determinista):
$(Q, \Sigma, q_0, \delta, F)$

- Q *es un conjunto finito de estados*
- Σ *es un alfabeto finito tal que* $\vdash, \mathtt{B} \notin \Sigma$
- $q_0 \in Q$ *es el estado inicial*
- $F \subseteq Q$ *es un conjunto de estados finales*
- δ *es una función parcial:*

$$\delta \ : \ Q \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \ \rightarrow \\ Q \times \{\leftarrow, \square, \rightarrow\} \times (\Sigma \cup \{\vdash, \mathtt{B}\}) \times \{\leftarrow, \square, \rightarrow\}$$

Espacio utilizado en una MT

Dado: MT determinista M con alfabeto Σ que para en todas las entradas

Definición

- $espacio_M(w)$: *número de celdas visitadas en la cinta de trabajo de* M *al procesar* w
- $s_M(n) = \max\{ espacio_M(w) \mid w \in \Sigma^* \text{ y } |w| = n\}$, *para cada* $n \geq 0$

s_M: Espacio utilizado en la ejecución de M en el peor caso

Espacio utilizado para resolver un problema

Definición

Un lenguaje L puede ser aceptado en espacio s si es que existe una MT determinista M tal que:

- *M para en todas las entradas*
- *$L = L(M)$*
- *s_M es $O(s)$*

El espacio requerido para computar una función f se define de la misma forma

Clases de complejidad: Espacio

Dado: Alfabeto Σ

Definición

DSPACE(s): Conjunto de todos los lenguajes $L \subseteq \Sigma^*$ que pueden ser aceptados en espacio s

Tres clases fundamentales:

$$\begin{aligned} \text{LOGSPACE} &= \text{DSPACE}(\log n) \\ \text{PSPACE} &= \bigcup_{k \in \mathbb{N}} \text{DSPACE}(n^k) \\ \text{EXPSPACE} &= \bigcup_{k \in \mathbb{N}} \text{DSPACE}(2^{n^k}) \end{aligned}$$

Relación entre clases de complejidad

¿Cuál es la relación entre las clases de complejidad que acabamos de definir?

La relación es más compleja:

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE}$$

También se sabe que: $\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE}$

Y todavía nos faltan las clases no deterministas ...

Clases de complejidad no deterministas: Espacio

Dado: MT no determinista M con cinta de trabajo y alfabeto Σ

Definición

- $espacio_M(w)$: *Número mínimo de celdas visitadas en la cinta de trabajo de M, entre todas las ejecuciones de M que aceptan w*
- $s_M(n) = \text{máx}(\{1\} \cup \{ espacio_M(w) \mid w \in \Sigma^*, |w| = n \text{ y } M \text{ acepta } w\})$, *para cada* $n \geq 0$

¿Por que incluimos $\{1\}$ en la definición?

Clases de complejidad no deterministas: Espacio

Definición

Un lenguaje L es aceptado en espacio s por una MT M no determinista si:

- $L = L(M)$
- s_M *es* $O(s)$

Clases de Complejidad no deterministas: Espacio

Dado: Alfabeto Σ

Definición

NSPACE(s): Conjunto de todos los lenguajes que pueden ser aceptados en espacio s por alguna MT no determinista

Tres clases fundamentales:

$$\begin{aligned} \text{NLOGSPACE} &= \text{NSPACE}(\log n) \\ \text{NPSPACE} &= \bigcup_{k \in \mathbb{N}} \text{NSPACE}(n^k) \\ \text{NEXPSPACE} &= \bigcup_{k \in \mathbb{N}} \text{NSPACE}(2^{n^k}) \end{aligned}$$

Relación entre clases de complejidad

¿Cuál es la relación entre las clases de complejidad que definimos?

Se sabe que: $\mathrm{PSPACE} = \mathrm{NPSPACE}$ y $\mathrm{EXPSPACE} = \mathrm{NEXPSPACE}$

Relación final:

$$\mathrm{LOGSPACE} \subseteq \mathrm{NLOGSPACE} \subseteq \mathrm{PTIME} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{NEXPTIME} \subseteq \mathrm{EXPSPACE}$$

La noción de completitud

Dado: Clase de complejidad $\mathcal{C}$ que contiene a LOGSPACE

Definición

- *Decimos que L es hard para $\mathcal{C}$ si para todo $L' \in \mathcal{C}$ existe una reducción de L' a L que puede ser calculada en espacio logarítmico*
- *Decimos que L es completo para $\mathcal{C}$ (o $\mathcal{C}$-completo) si $L \in \mathcal{C}$ y L es hard para $\mathcal{C}$*

Problemas completos

Problema completo para NP: SAT

- ¿Puede nombrar otros?

Problema completo para NLOGSPACE: Graph Reachability

Problema completo para PSPACE: QBF

Complejidad de la LPO

Dado: Vocabulario $\mathcal{L}$

Definición

La complejidad de LPO se define como la complejidad del siguiente lenguaje:

$$L \quad = \quad \{(\mathfrak{A}, \varphi) \mid \mathfrak{A} \in \textsc{Struct}[\mathcal{L}],\ \varphi \textit{ es una } \mathcal{L}\textit{-oración y } \mathfrak{A} \models \varphi\}$$

Notación

Llamamos a esto complejidad combinada

Complejidad combinada de la LPO

¿Cuál es la complejidad combinada de la lógica de primer orden?

Teorema

L es PSPACE-completo

Ejercicio

Demuestre que el teorema es válido para cualquier vocabulario $\mathcal{L}$

Complejidad combinada de la LPO

El resultado anterior nos dice que es muy difícil evaluar una consulta en LPO

- ¿Cómo puede entonces funcionar un sistema de bases de datos?
- En general: Consultas son pequeñas comparadas con la base de datos

Complejidad de los datos de la LPO

Vamos a tomar en cuenta la diferencia entre el tamaño de las consultas y las bases de datos.

Dado: Vocabulario $\mathcal{L}$

Definición

La complejidad de evaluar una fórmula fija φ en LPO se define como la complejidad del siguiente lenguaje:

$$L_\varphi \quad = \quad \{\mathfrak{A} \mid \mathfrak{A} \in \textsc{Struct}[\mathcal{L}] \text{ y } \mathfrak{A} \models \varphi\}$$

Notación

Llamamos a esto complejidad de los datos

Complejidad de los datos de la LPO

Teorema

Para cada $\mathcal{L}$-oración φ se tiene que L_φ está en LOGSPACE

Ejercicio

Demuestre el teorema

Conclusión

Cada consulta puede ser evaluada eficientemente

Preguntas a responder ...

- ¿Cuan cerca está LPO de LOGSPACE? ¿Cuál es la complejidad exacta de LPO?
- ¿Existe algún lenguaje que sea completo para PTIME en términos de complejidad de los datos? ¿Puede ser este lenguaje LPO?
- ¿Hay consultas en algún lenguaje que no pueden ser evaluadas eficientemente?

Vamos a responder estas consultas. Empecemos por la tercera ...

La lógica de segundo orden: Sintaxis

Dado: Vocabulario $\mathcal{L}$

Definición

La lógica de segundo orden (LSO) sobre $\mathcal{L}$ es definida como la extensión de LPO que incluye las siguientes reglas:

- *Si $t_1, \ldots, t_k$ son $\mathcal{L}$-términos y X es una variable de segundo orden de aridad k (vale decir, una relación con $k \geq 1$ argumentos), entonces $X(t_1, \ldots, t_k)$ es una fórmula en LSO*
- *Si φ es una fórmula en LSO y X es una variable de segundo orden de aridad k, entonces $\exists X\varphi$ y $\forall X\varphi$ son fórmulas en LSO*

La lógica de segundo orden: Semántica

Notación

Dada una estructura $\mathfrak{A}$ con dominio A, una asignación σ es una función que asigna:

- *un valor en A a cada variable x de primer orden:* $\sigma(x) \in A$
- *un subconjunto de A^k a cada variable X de segundo orden con k argumentos:* $\sigma(X) \subseteq A^k$

La lógica de segundo orden: Semántica

Definición

La definición de la semántica de LSO incluye tres casos adicionales. Para una variable de segundo orden X con aridad k:

- $(\mathfrak{A}, \sigma) \models X(t_1, \ldots, t_k)$ *si y sólo si* $(\sigma(t_1), \ldots, \sigma(t_k)) \in \sigma(X)$
- $(\mathfrak{A}, \sigma) \models \exists X \varphi$ *si y sólo si existe* $S \subseteq A^k$ *tal que* $(\mathfrak{A}, \sigma[X/S]) \models \varphi$
- $(\mathfrak{A}, \sigma) \models \forall X \varphi$ *si y sólo si para todo* $S \subseteq A^k$*, se tiene que* $(\mathfrak{A}, \sigma[X/S]) \models \varphi$

La lógica de segundo orden: Ejemplo

¿Qué podemos decir en LSO?

Sea $\mathcal{L} = \{E(\cdot,\cdot)\}$ y $\mathcal{P}$ el conjunto de todas $\mathcal{L}$-estructuras $\mathfrak{A}$ tal que $E^{\mathfrak{A}}$ es una relación de sucesor.

- Vamos a demostrar que esta propiedad es expresable en LSO

Sea:

$$\mathit{first}(x) \quad = \quad \forall y \neg E(y,x)$$

La lógica de segundo orden: Ejemplo

Y sea:

$$\mathit{conectado}(x,y) \;=\; \forall P \Big[\Big(P(x) \wedge \forall u \forall v \, (P(u) \wedge E(u,v) \rightarrow P(v)) \Big) \rightarrow P(y) \Big]$$

Entonces la siguiente $\mathcal{L}$-oración Φ en LSO define $\mathcal{P}$:

$$\Phi \;= \forall x \forall y \forall z \Big(E(x,y) \wedge E(x,z) \rightarrow y = z \Big) \wedge$$
$$\forall x \forall y \forall z \Big(E(y,x) \wedge E(z,x) \rightarrow y = z \Big) \wedge$$
$$\exists x \Big(\mathit{first}(x) \wedge \forall y \, (x \neq y \rightarrow \mathit{conectado}(x,y)) \Big)$$

Lógica de Segundo Orden Existencial

Dado: Vocabulario $\mathcal{L}$

Definición

El fragmento existencial de la lógica de segundo orden, denotado como $\exists LSO$, es definido como el conjunto de todas las fórmulas en LSO sobre $\mathcal{L}$ de la forma:

$$\exists X_1 \exists X_2 \cdots \exists X_\ell \ \varphi$$

donde φ es una fórmula en LPO sobre el vocabulario $\mathcal{L}' = \mathcal{L} \cup \{X_1, \ldots, X_\ell\}$

Fragmento existencial de LSO: Ejemplo

¿Qué podemos decir en $\exists$LSO?

Sea $\mathcal{L} = \{E(\cdot,\cdot)\}$ y $\mathcal{P}$ el conjunto de todas $\mathcal{L}$-estructuras $\mathfrak{A}$ tal que $E^{\mathfrak{A}}$ es una relación de sucesor.

- ¿Podemos expresar esta propiedad en $\exists$LSO?

La siguiente $\mathcal{L}$-oración Ψ en $\exists$LSO define $\mathcal{P}$:

$$\Psi \;=\; \exists L \left(\begin{array}{l} \forall x\, \neg L(x,x) \;\wedge \\ \forall x \forall y \forall z\, (L(x,y) \wedge L(y,z) \rightarrow L(x,z)) \;\wedge \\ \forall x \forall y\, (L(x,y) \vee x = y \vee L(y,x)) \;\wedge \\ \forall x \forall y\, (E(x,y) \leftrightarrow (L(x,y) \wedge \neg \exists z\, (L(x,z) \wedge L(z,y)))) \end{array} \right)$$

Complejidad de la $\exists$LSO

Teorema

1. *Para cada vocabulario $\mathcal{L}$ y $\mathcal{L}$-oración φ en $\exists$LSO, se tiene que L_φ está en NP*
2. *Existe un vocabulario $\mathcal{L}$ y una $\mathcal{L}$-oración φ en $\exists$LSO tal que L_φ es NP-completo*

Ejercicio

Demuestre la primera parte del teorema

Vamos a demostrar la segunda parte del teorema ...

NP-hardness de $\exists$LSO: Demostración

Sea $\mathcal{L} = \{E(\cdot, \cdot)\}$

- E es utilizado para almacenar grafos

Sea Φ la siguiente $\mathcal{L}$-oración en $\exists$LSO:

$$\exists A \exists B \exists C \left(\begin{array}{l} \forall x\,(A(x) \vee B(x) \vee C(x)) \wedge \\ \forall x\,(\neg A(x) \vee \neg B(x)) \wedge \\ \forall x\,(\neg A(x) \vee \neg C(x)) \wedge \\ \forall x\,(\neg B(x) \vee \neg C(x)) \wedge \\ \forall x \forall y\,((E(x,y) \wedge A(x)) \rightarrow \neg A(y)) \wedge \\ \forall x \forall y\,((E(x,y) \wedge B(x)) \rightarrow \neg B(y)) \wedge \\ \forall x \forall y\,((E(x,y) \wedge C(x)) \rightarrow \neg C(y)) \end{array} \right)$$

NP-hardness de ∃LSO: Demostración

Lema

L_Φ es NP-hard

Demostración: Reducimos desde el problema de 3-coloración de grafos.

- Dado un grafo $G = (N, A)$, $\mathfrak{A}_G$ se define como una $\mathcal{L}$-estructura tal que:
  - El dominio de $\mathfrak{A}_G$ es N
  - $E^{\mathfrak{A}_G} = A$

Se tiene que G es 3-coloreable si y sólo si $\mathfrak{A}_G \in L_\Phi$ (es decir, $\mathfrak{A}_G \models \Phi$) □

Una segunda mirada a LPO

Acabamos de demostrar que la LSO no es una buena alternativa para un lenguaje de consulta.

- La complejidad de los datos del fragmento existencial es NP

Sabemos que la complejidad de los datos de la LPO está en LOGSPACE, por lo que LPO es una mejor alternativa

- ¿Qué tan buena es esta alternativa? ¿LPO está cerca de PTIME? ¿Al menos cerca de LOGSPACE?

Una segunda mirada a LPO

Vamos a demostrar que la LPO no es una buena alternativa en términos de expresividad.

- Vamos a introducir una noción de complejidad basada en circuitos Booleanos
- Vamos a definir clases de complejidad usando esta noción
- Vamos a mostrar que la LPO vive en una clase de complejidad que está propiamente contenida en LOGSPACE

Circuitos Booleanos

Definición

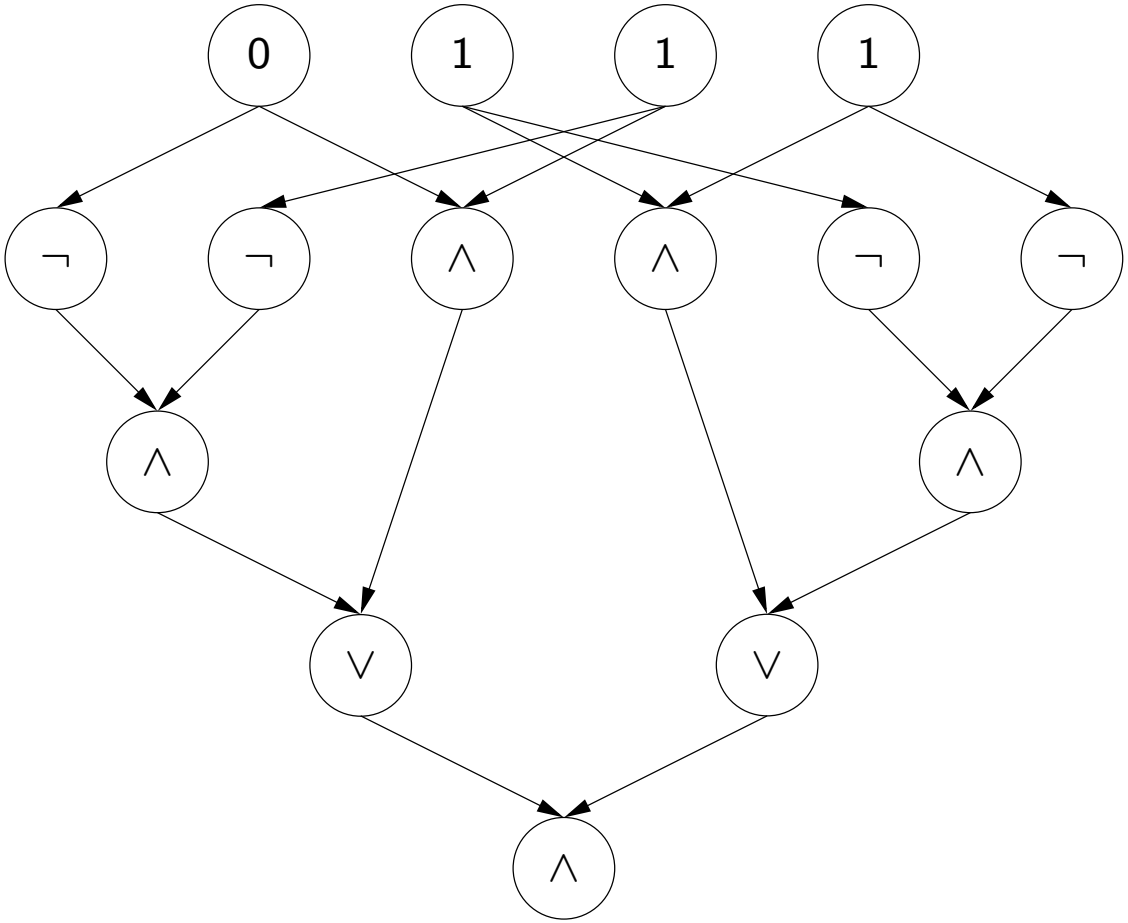
Un circuito Booleano es un grafo dirigido acíclico tal que:

- *Cada nodo sin arcos de entrada tiene etiqueta 0 ó 1*
- *Cada nodo con arcos de entrada tiene etiqueta $\neg$, $\wedge$ ó $\vee$*
- *Existe un único nodo sin arcos de salida*

Notación

- *Nodo de entrada: Nodo sin arcos de entrada*
- *Nodo interior: Nodo con arcos de entrada*
- *Nodo de salida: Nodo sin arcos de salida*

Circuito Booleano: Ejemplo

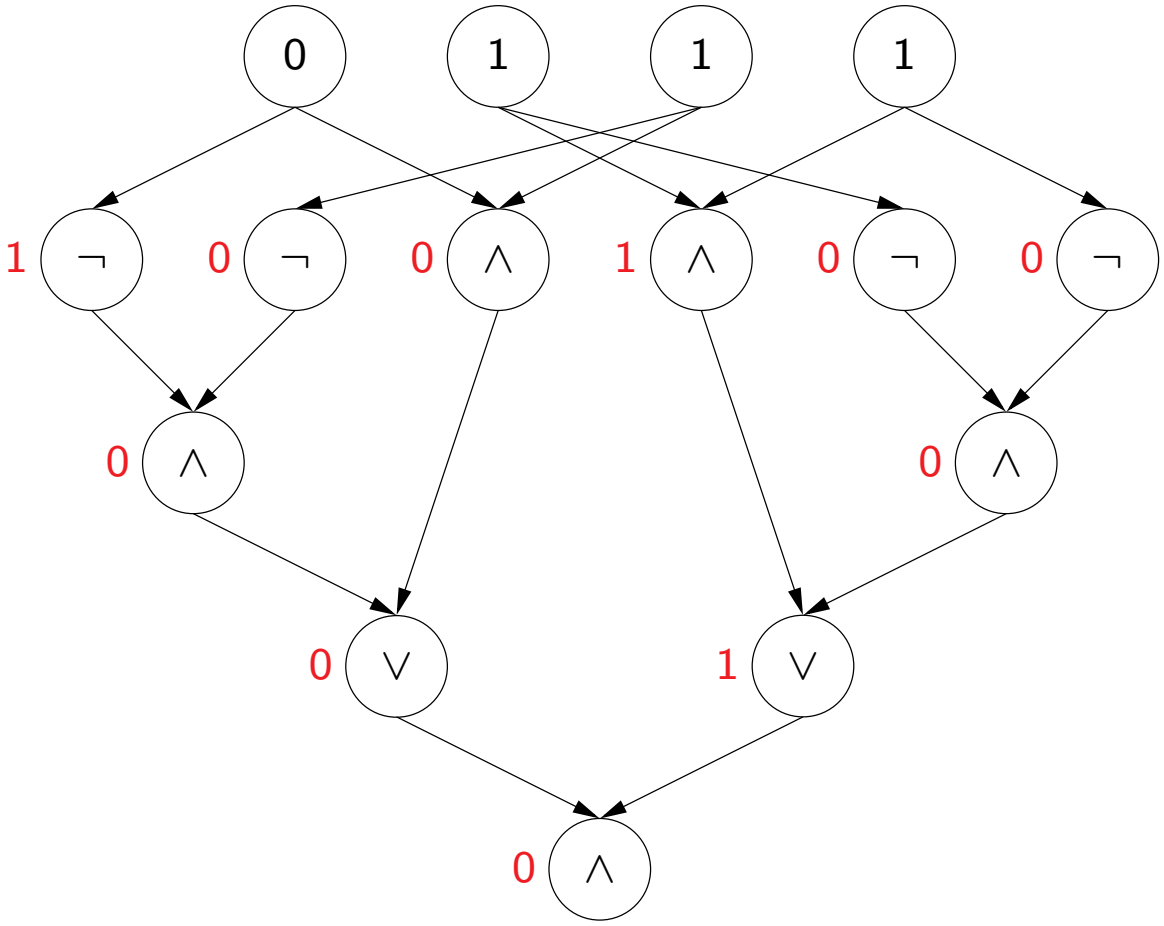


Circuitos Booleanos como funciones

Cada circuito Booleano puede ser interpretado como una función:

- Toma como entrada los valores en los nodos de entrada
- Asocia a cada nodo interior u el resultado de evaluar la etiqueta de u sobre los valores asociados a los nodos con arcos dirigidos a u
- Tiene como resultado el valor asociado al nodo de salida

Circuitos Booleanos como funciones: Ejemplo



Circuitos Booleanos como funciones

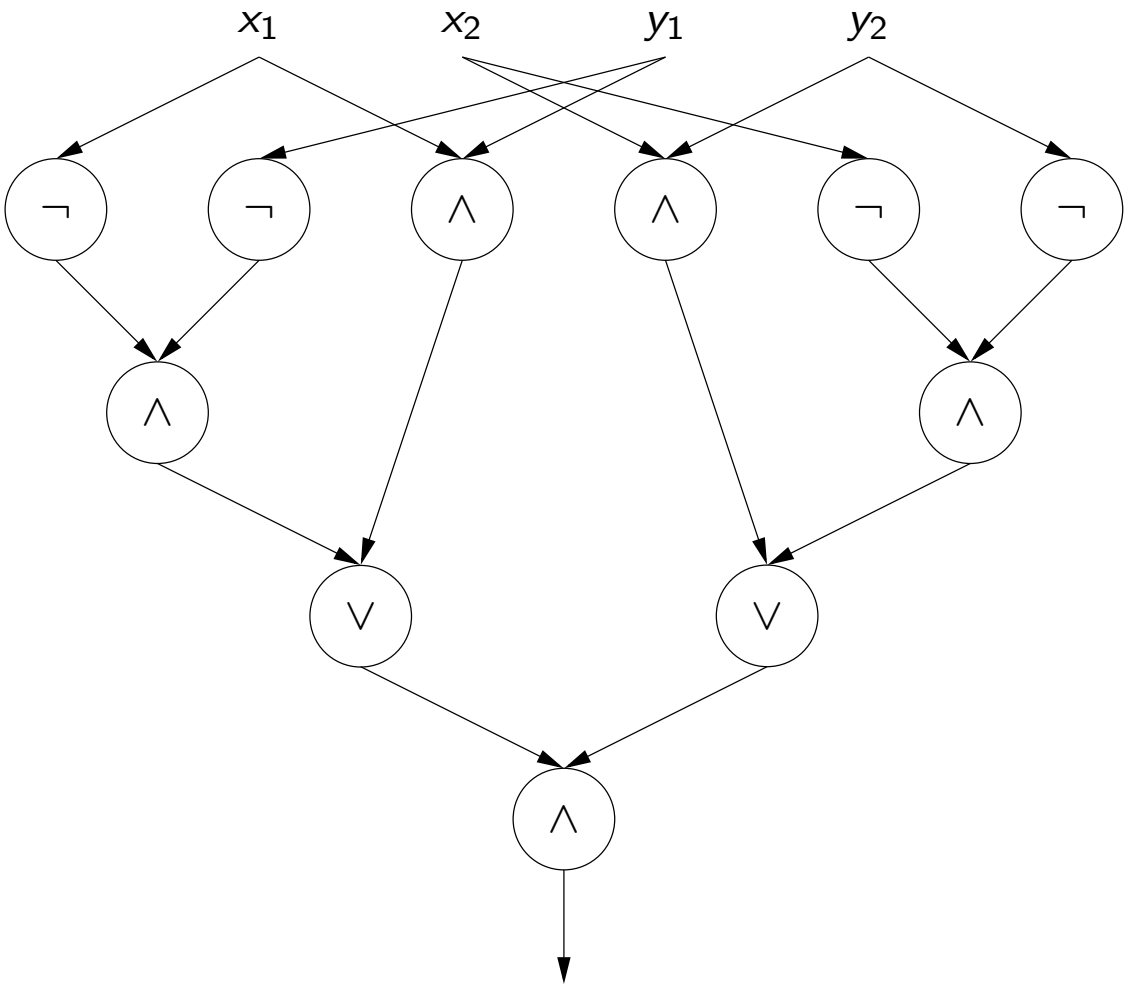
Circuito en el ejemplo anterior representa la función:

$$f(x_1, x_2, y_1, y_2) = \begin{cases} 1 & x_1 = y_1 \text{ y } x_2 = y_2 \\ 0 & \text{en otro caso} \end{cases}$$

Para indicar la función que representa un circuito:

- Reemplazamos las etiquetas 0 y 1 por variables
- Usamos un arco sin nodo de destino para indicar la salida

Circuito Booleano como función: Ejemplo



Circuitos Booleanos como funciones: Terminología

Definición

- *Tamaño de un circuito:* *Número de nodos con etiquetas* $\neg$, $\wedge$ *y* $\vee$
- *Profundidad de un circuito:* *Largo del camino más largo entre un nodo de entrada y el nodo de salida*

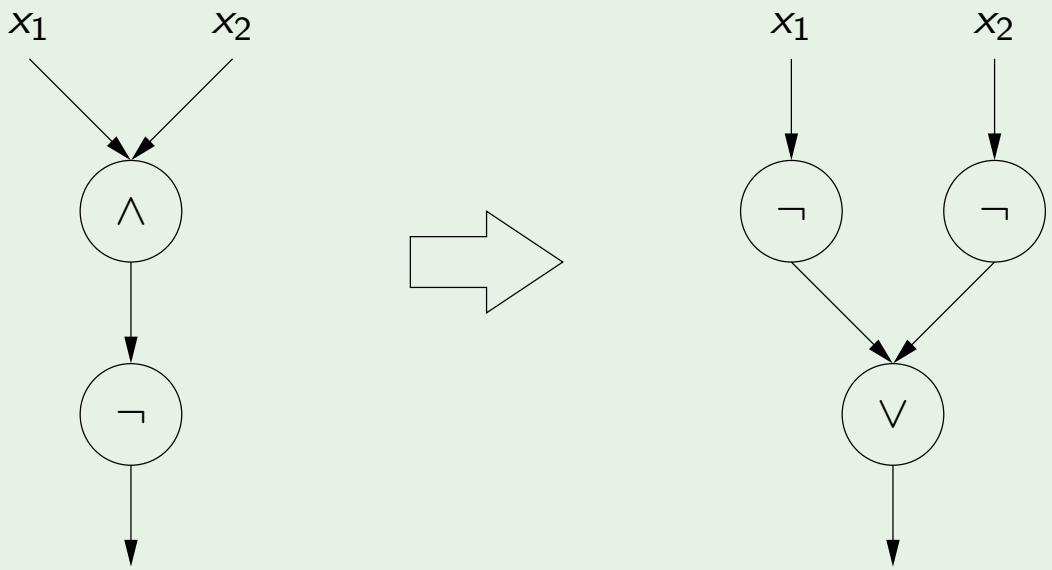
Ejemplo

El tamaño es 11 y la profundidad es 4 en el circuito de la transparencia anterior.

Circuitos Booleanos como funciones: Forma normal

Usando $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$ y $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$ podemos demostrar que cada circuito es equivalente a otro que sólo tiene negaciones en el primer nivel.

Ejemplo



Circuitos Booleanos como funciones: Forma normal

Aun más: El circuito generado tiene la misma profundidad que el circuito original y a lo más el doble de nodos.

De ahora en adelante: Suponemos que los circuitos sólo tienen negaciones en el primer nivel.

- Definimos el tamaño de un circuito sólo considerando el número de nodos con etiquetas $\wedge$ y $\vee$

Notación

- *tamaño*(C): *Tamaño del circuito* C
- *profundidad*(C): *Profundidad del circuito* C

Lenguajes aceptados por circuitos

Dado: Circuito $C(\bar{x})$ con n entradas

Definición

C define el lenguaje:

$$\{w \in \{0,1\}^n \mid C(w) = 1\}$$

Ejemplo

Para el caso anterior: $\{a_1a_2a_3a_4 \in \{0,1\}^4 \mid a_1 = a_3 \text{ y } a_2 = a_4\}$

Lenguajes aceptados por circuitos

El lenguaje definido por un circuito siempre es finito.

- Tenemos que usar familias de circuitos para poder generar lenguajes infinitos

Definición

Una familia de circuitos $\{C_n\}_{n \geq 0}$ *acepta* $L \subseteq \{0,1\}^*$ *si:*

- C_n *tiene* n *entradas* (C_0 *es 0 ó 1*)
- *Para cada palabra* w *de largo* n: $w \in L$ *si y sólo si* $C_n(w) = 1$

Lenguajes aceptados por circuitos

La noción de aceptación que acabamos de definir es muy general.

Proposición

Cada $L \subseteq \{0,1\}^*$ es aceptado por una familia de circuitos $\{C_n\}$.

Ejercicio

Demuestre la proposición.

Lenguajes aceptados por circuitos

Si queremos utilizar una familia de circuitos como un algoritmo tenemos que introducir una noción de uniformidad.

- Debe existir un algoritmo que genere los circuitos

Definición

Una MT determinista M genera una familia de circuitos $\{C_n\}$ si esta máquina con entrada 0^k genera C_k.

- *M genera C_k codificado como una palabra en $\{0,1\}^*$*

Clases de complejidad y circuitos

Es posible usar circuitos para definir clases de complejidad

Definición (Clase de complejidad AC^i)

Para $i \geq 0$: Un lenguaje $L \subseteq \{0,1\}^*$ está en AC^i si existe una familia de circuitos $\{C_n\}$, una MT determinista M y una constante k tal que:

1. *L es aceptado por $\{C_n\}$*
2. *M genera $\{C_n\}$ y funciona en espacio logarítmico*
3. *profundidad$(C_n) \leq k \cdot (\log n)^i$*

AC^i: Algunos comentarios

No imponemos restricciones al número de arcos que salen de un nodo.

No imponemos restricciones al número de arcos que llegan a un nodo.

- Si usamos compuertas con dos entradas obtenemos las clases NC^i

Los problemas que pueden ser resueltos en alguna de estas clases son llamados paralelizables (o también muy paralelizables).

AC^i: Poder de computación

AC^0: Circuitos tienen profundidad constante

Ejercicio

Muestre que la multiplicación de matrices esta en AC^0 ($+$ se interpreta como $\vee$ y $\cdot$ como $\wedge$)

AC^i: Poder de computación

AC^1: Circuitos tienen profundidad logarítmica. Esto nos permite resolver muchos problemas

Ejercicio

1. Muestre que $L = \{w \in \{0,1\}^* \mid w$ tiene un número par de símbolos $0\}$ está en AC^1
2. Muestre que el problema de verificar si dos nodos están conectados en un grafo está en AC^1

Suma en AC^0

Vamos a demostrar que la suma puede ser calculada en AC^0, vale decir, con circuitos de profundidad constante.

Queremos construir un circuito $C(x_n, \ldots, x_1, y_n, \ldots, y_1)$:

- Recibe como entrada dos números binarios $\bar{x} = x_n \cdots x_1$ y $\bar{y} = y_n \cdots y_1$
- Tiene $n + 1$ salidas que corresponden a la suma de estos números: $z_{n+1} z_n \cdots z_1$

Suma en AC^0

Notación

$$\begin{aligned} AND_i &= x_i \wedge y_i \\ OR_i &= x_i \vee y_i \\ XOR_i &= (x_i \wedge \neg y_i) \vee (\neg x_i \wedge y_i) \end{aligned}$$

Sea $r_0 = 0$ y r_i ($i \in [1, n]$) el i-ésimo resto en la suma de $\bar{x}$ y $\bar{y}$.

Entonces para $i \geq 1$:

$$r_i = \mathrm{AND}_i \vee (\mathrm{OR}_i \wedge r_{i-1})$$

Suma en AC^0

Desarrollando una vez obtenemos:

$$r_i \quad = \quad \mathrm{AND}_i \vee (\mathrm{OR}_i \wedge \mathrm{AND}_{i-1}) \vee (\mathrm{OR}_i \wedge \mathrm{OR}_{i-1} \wedge r_{i-2})$$

Si seguimos desarrollando obtenemos:

$$\begin{aligned}
r_1 &= \mathrm{AND}_1 \\
r_2 &= \mathrm{AND}_2 \vee (\mathrm{OR}_2 \wedge \mathrm{AND}_1) \\
&\cdots \\
r_n &= \mathrm{AND}_n \vee (\mathrm{OR}_n \wedge \mathrm{AND}_{n-1}) \vee (\mathrm{OR}_n \wedge \mathrm{OR}_{n-1} \wedge \mathrm{AND}_{n-2}) \vee \cdots \\
&\qquad \cdots \vee (\mathrm{OR}_n \wedge \mathrm{OR}_{n-1} \wedge \cdots \wedge \mathrm{OR}_2 \wedge \mathrm{AND}_1).
\end{aligned}$$

Suma en AC^0

Podemos usar los restos para definir el resultado de la suma:

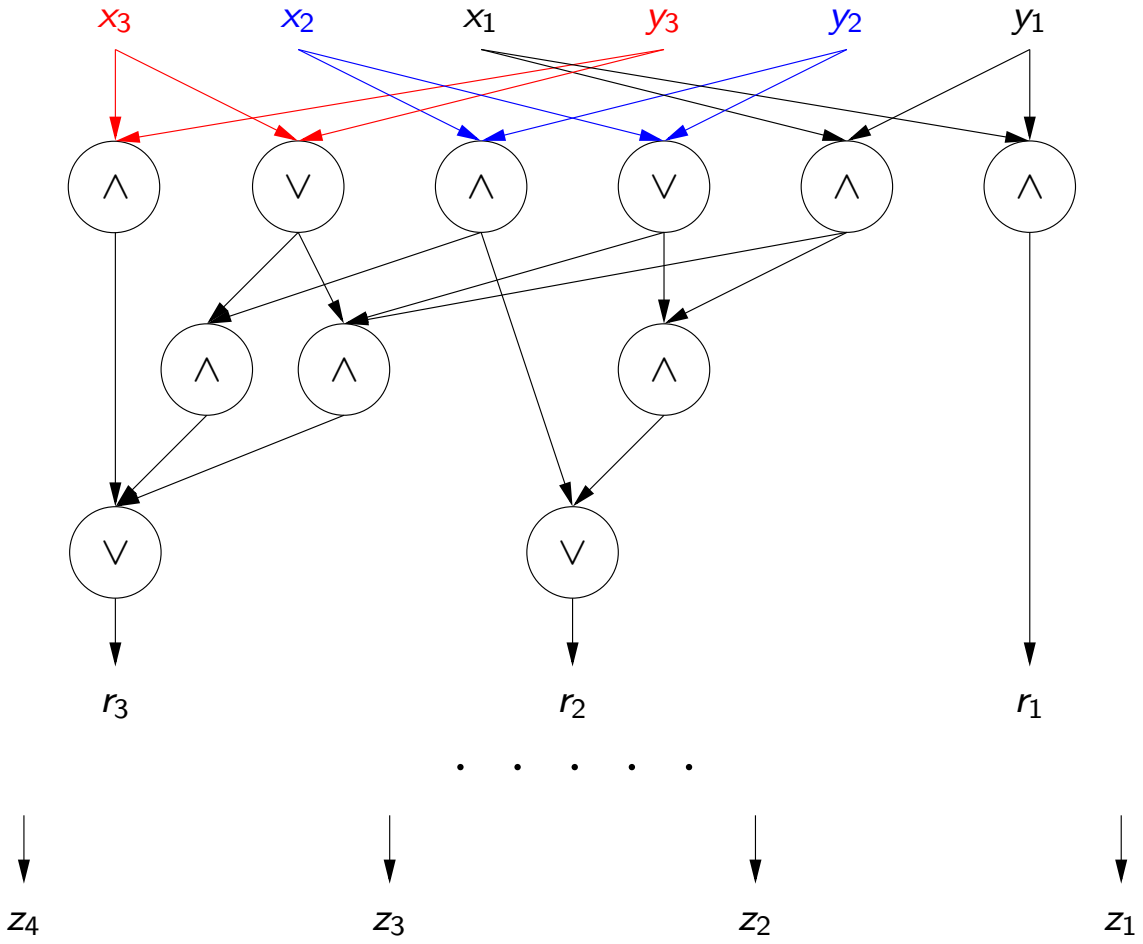
$$z_i \quad = \quad (\neg \mathrm{OR}_i \wedge r_{i-1}) \vee (\mathrm{XOR}_i \wedge \neg r_{i-1}) \vee (\mathrm{AND}_i \wedge r_{i-1})$$

En particular: $z_1 = \mathrm{XOR}_1$ y $z_{n+1} = r_n$

Podemos utilizar las definiciones de r_i y z_i para construir el circuito.

- Veamos un ejemplo: $C(x_3, x_2, x_1, y_3, y_2, y_1)$
- ¿Cuál es la profundidad del circuito?

Suma en AC^0: Ejemplo



Circuitos de tamaño polinomial

Si $L \in \mathrm{AC}^i$: Existe una familia de circuitos de tamaño polinomial que acepta L.

- Aun más: Tenemos que $L \in \mathrm{PTIME}$ por la condición de uniformidad

¿Qué clase de problemas pueden ser aceptados por familias de circuitos de tamaño polinomial?

AC^i: Resultados fundamentales

Sea $AC = \bigcup_{i \geq 0} AC^i$

Teorema

$AC^0 \subseteq LOGSPACE \subseteq NLOGSPACE \subseteq AC^1 \subseteq AC \subseteq PTIME$

Teorema (Furst-Saxe-Sipser)

$AC^0 \subsetneq LOGSPACE$

Demostración: $L = \{w \in \{0,1\}^* \mid w$ tiene un número par de símbolos $0\}$ no está en AC^0

Complejidad de la LPO: Circuitos Booleanos

¿Por qué nos interesan tanto los circuitos Booleanos?

Vamos a demostrar que la complejidad de los datos de la LPO es muy baja

- Está contenida en AC^0

Para empezar: Necesitamos representar estructuras como palabras en $\{0,1\}^*$

Complejidad de la LPO: Circuitos Booleanos

Dado: Vocabulario $\mathcal{L}$ sin constantes

- Constantes son reemplazadas por predicados unarios

Definimos la codificación de una $\mathcal{L}$-estructura $\mathfrak{A}$ ($\mathrm{enc}(\mathfrak{A})$) de la siguiente forma.

1. Tomamos un orden arbitrario en A: $a_1 < a_2 < \cdots < a_n$

   Este orden es usado para definir un orden lexicográfico para las k-tuplas ($k \geq 1$):

$$(a_1, \ldots, a_1, a_1) < (a_1, \ldots, a_1, a_2) < \cdots$$
$$\cdots < (a_1, \ldots, a_1, a_n) < (a_1, \ldots, a_2, a_1) < \cdots$$
$$\cdots < (a_n, \ldots, a_n, a_{n-1}) < (a_n, \ldots, a_n, a_n)$$

Complejidad de la LPO: Circuitos Booleanos

2. Para R en $\mathcal{L}$ con aridad k: $\mathrm{enc}(R)$ es una palabra w de largo n^k, tal que el i-ésimo elemento de w es 1 si y sólo si la i-ésima tupla en el orden lexicográfico de las k-tuplas está en $R^{\mathfrak{A}}$

3. Si $\mathcal{L} = \{R_1, \ldots, R_\ell\}$, entonces $\mathrm{enc}(\mathfrak{A}) = \mathrm{enc}(R_1) \cdots \mathrm{enc}(R_\ell)$

Ejemplo

Si $\mathfrak{A} = \langle A = \{1,2\}, P^{\mathfrak{A}} = \{1\}, R^{\mathfrak{A}} = \{(1,1),(2,1),(2,2)\}\rangle$, entonces:

$$\mathrm{enc}(\mathfrak{A}) \quad = \quad 101011$$

Complejidad de la LPO y AC^0

Dado: Vocabulario $\mathcal{L}$ sin constantes

Teorema

Para cada $\mathcal{L}$-oración φ, se tiene que L_φ está en AC^0

Demostración: Suponemos que en φ el conectivo $\neg$ sólo aparece en fórmulas de la forma $\neg(x = y)$ (es decir, $x \neq y$) y $\neg R(\bar{x})$

- ¿Por qué podemos suponer esto?

Para evaluar una fórmula φ en una estructura $\mathfrak{A}$ reemplazamos:

$$\exists x\, \psi(x, \bar{y}) \quad \text{por} \quad \bigvee_{a \in A} \psi(a, \bar{y})$$

$$\forall x\, \psi(x, \bar{y}) \quad \text{por} \quad \bigwedge_{a \in A} \psi(a, \bar{y})$$

Complejidad de la LPO y AC^0

Cuando hemos terminado de reemplazar todos los cuantificadores, podemos construir fácilmente el circuito buscado.

- ¿Cómo se maneja la igualdad?

La profundidad del circuito sólo depende de φ, por lo que es fija.

Complejidad de la LPO y AC^0

Es simple demostrar que la familia de circuitos puede ser generada por una MT determinista que trabaja en espacio logarítmico.

- Está máquina entrega el circuito 0 con entrada 0^k si las palabras de largo k no pueden representar una estructura

Concluimos que L_φ está en AC^0. □

Complejidad de la LPO y AC^0: Ejemplo

Sea $\varphi = \exists x\,(P(x) \wedge \forall y\,(x \neq y \rightarrow R(x,y)))$

Vamos a construir el circuito para φ y las estructuras con 3 elementos en el dominio

Primero reemplazamos $\exists x$ por la siguiente disyunción:

$$\Big(P(1) \wedge \forall y\,(1 \neq y \rightarrow R(1,y))\Big) \vee$$
$$\Big(P(2) \wedge \forall y\,(2 \neq y \rightarrow R(2,y))\Big) \vee$$
$$\Big(P(3) \wedge \forall y\,(3 \neq y \rightarrow R(3,y))\Big)$$

Complejidad de la LPO y AC^0: Ejemplo

Después reemplazamos $\forall y$ por las siguientes conjunciones:

$$\Big(P(1) \wedge ((1 \neq 1 \rightarrow R(1,1)) \wedge (1 \neq 2 \rightarrow R(1,2)) \wedge (1 \neq 3 \rightarrow R(1,3)))\Big) \vee$$
$$\Big(P(2) \wedge ((2 \neq 1 \rightarrow R(2,1)) \wedge (2 \neq 2 \rightarrow R(2,2)) \wedge (2 \neq 3 \rightarrow R(2,3)))\Big) \vee$$
$$\Big(P(3) \wedge ((3 \neq 1 \rightarrow R(3,1)) \wedge (3 \neq 2 \rightarrow R(3,2)) \wedge (3 \neq 3 \rightarrow R(3,3)))\Big)$$

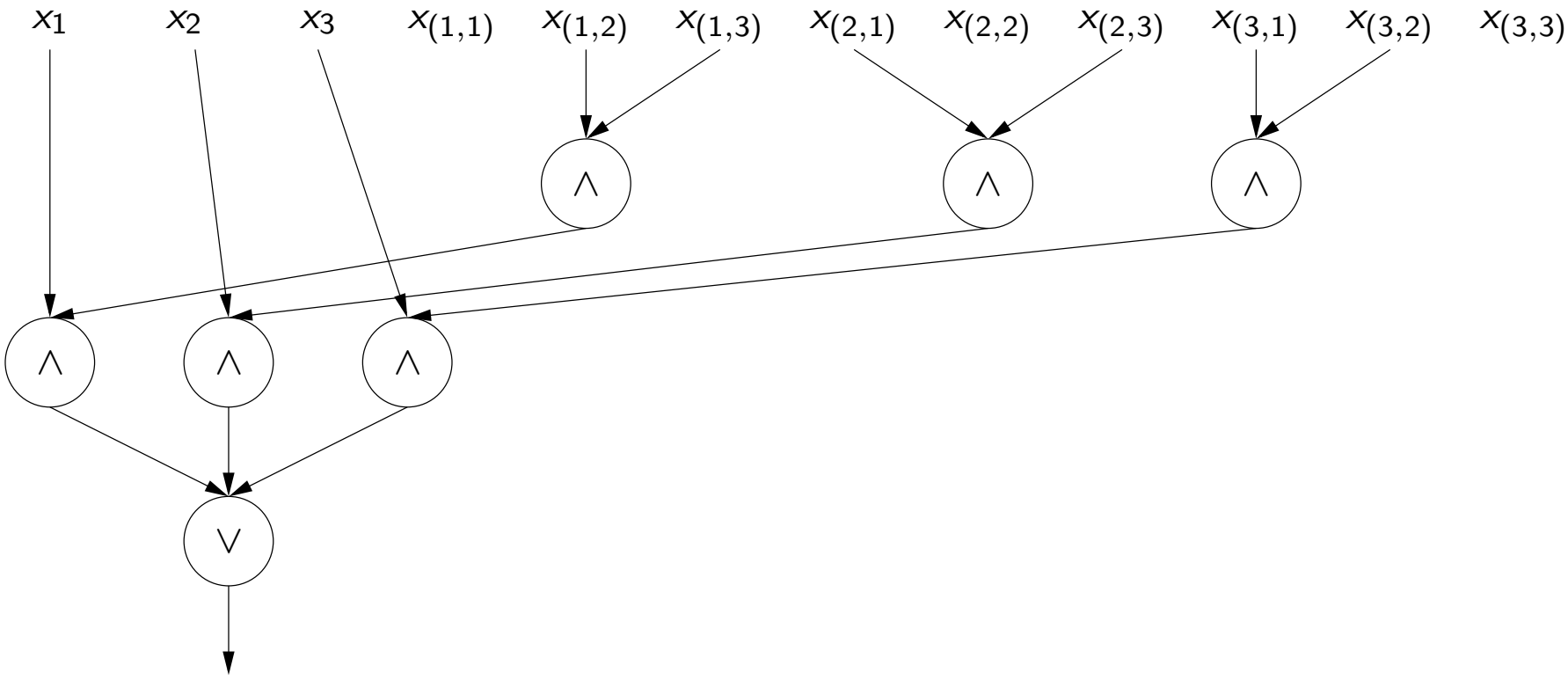
Complejidad de la LPO y AC^0: Ejemplo

Finalmente eliminamos la igualdad:

$$\Big(P(1) \wedge (R(1,2) \wedge R(1,3))\Big) \vee$$
$$\Big(P(2) \wedge (R(2,1) \wedge R(2,3))\Big) \vee$$
$$\Big(P(3) \wedge (R(3,1) \wedge R(3,2))\Big)$$

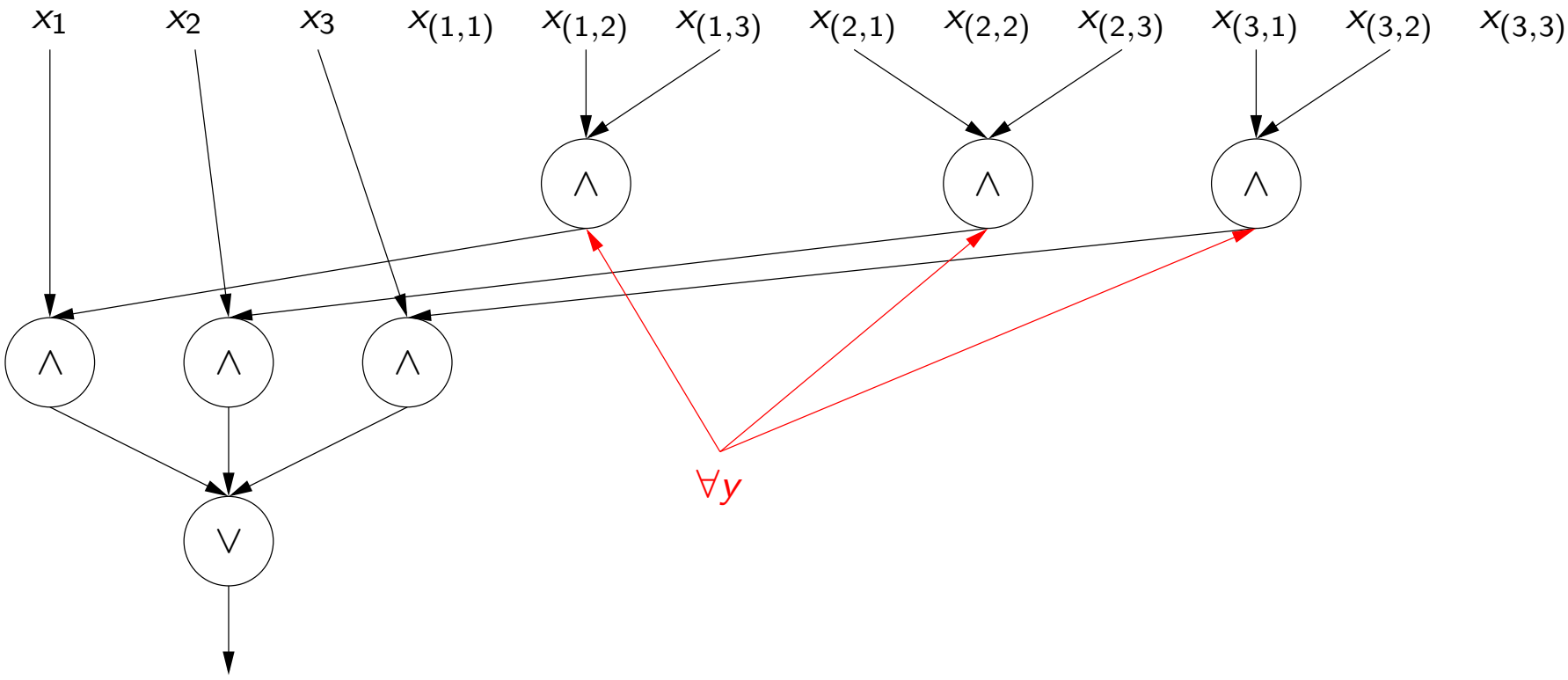
Complejidad de la LPO y AC^0: Ejemplo

Circuito para $\varphi = \exists x\,(P(x) \wedge \forall y\,(x \neq y \rightarrow R(x,y)))$ y las estructuras con 3 elementos en el dominio:



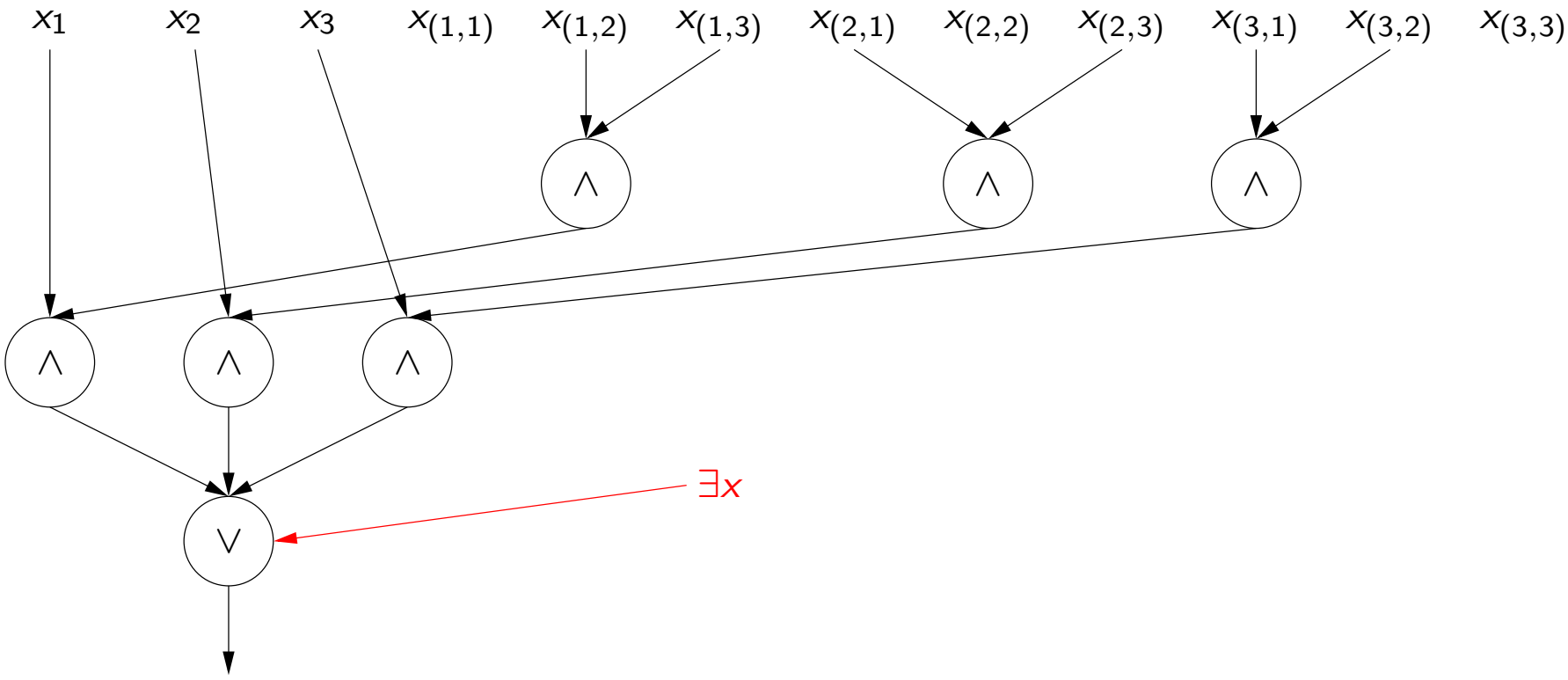
Complejidad de la LPO y AC^0: Ejemplo

Circuito para $\varphi = \exists x\,(P(x) \wedge \forall y\,(x \neq y \rightarrow R(x,y)))$ y las estructuras con 3 elementos en el dominio:



Complejidad de la LPO y AC^0: Ejemplo

Circuito para $\varphi = \exists x\,(P(x) \wedge \forall y\,(x \neq y \rightarrow R(x,y)))$ y las estructuras con 3 elementos en el dominio:



Complejidad de la LPO y AC^0: Consecuencias

Del Teorema de Furst-Saxe-Sipser obtenemos:

Corolario

Sea $\mathcal{L} = \{U(\cdot)\}$. El conjunto $\mathcal{P}$ de todas las $\mathcal{L}$-estructuras que tienen una cantidad par de elementos en U no es expresable en lógica de primer orden.

Pero esto ya lo habíamos demostrado.

- ¿Qué otras consecuencias tienen los resultados sobre la complejidad de la LPO?

Complejidad de la LPO y LOGSPACE

Complejidad basada en circuitos puede ser usada para responder una de las preguntas que teníamos pendientes.

- La complejidad de los datos de LPO está en una clase que está contenida en forma propia en LOGSPACE

Hay muchas propiedades que pueden ser computadas eficientemente y que no pueden ser expresadas en LPO

- Queremos encontrar lenguajes que estén más cerca de PTIME (¿qué *capturen* PTIME?)