Descriptive Complexity: First Order Reductions

Stathis Zachos, Petros Potikas, Ioannis Kokkinis and Aggeliki Chalki

ALMA
INTER-INSTITUTIONAL GRADUATE PROGRAM "ALGORITHMS, LOGIC AND DISCRETE MATHEMATICS"

Overview

Overview

Goal of the Section

FO is the set of boolean queries expressible in first order logic.

Goal of the Section

FO is the set of boolean queries expressible in first order logic.

L is the set of boolean queries computable by a deterministic Turing machine using at most logarithmic space.

Goal of the Section

FO is the set of boolean queries expressible in first order logic.

L is the set of boolean queries computable by a deterministic Turing machine using at most logarithmic space.

The goal of this section is to show that $FO \subseteq L$.

Logspace Turing Machines

A logspace-Turing Machine

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,
- has a write-only output tape,

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,
- has a write-only output tape,
- has a read-write work tape that contains $\mathcal{O}(\log n)$ bits.

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,
- has a write-only output tape,
- has a read-write work tape that contains $\mathcal{O}(\log n)$ bits.

Thus, it typically can:

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,
- has a write-only output tape,
- has a read-write work tape that contains $\mathcal{O}(\log n)$ bits.

Thus, it typically can:

- store a *logn*-bit number that points to a position in the input,

Logspace Turing Machines

A logspace-Turing Machine

- has a read-only input tape,
- has a write-only output tape,
- has a read-write work tape that contains $\mathcal{O}(\log n)$ bits.

Thus, it typically can:

- store a *logn*-bit number that points to a position in the input,
- work on strings (numbers etc) of size $\mathcal{O}(\log n)$ on the work tape.

Addition in Logarithmic Space

$$\begin{array}{rr} & 01011 \\ + & 00110 \\ \hline & 10001 \end{array}$$

The simple school algorithm works!

Addition in Logarithmic Space

$$\begin{array}{r r} & 01011 \\ + & 00110 \\ \hline & 10001 \end{array}$$

The simple school algorithm works!

The logspace Turing Machine examines the input positions and produces the output bits one by one from right to left.

Addition in Logarithmic Space

$$\begin{array}{rr} & 01011 \\ + & 00110 \\ \hline & 10001 \end{array}$$

The simple school algorithm works!

The logspace Turing Machine examines the input positions and produces the output bits one by one from right to left.

Only a the single bit carry has to be stored.

Multiplication in Logarithmic Space (1/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
r	1	0	0	1	0	

Again the school algorithm works, however we need some observations.

Multiplication in Logarithmic Space (1/3)

	5	4	3	2	1	0
b			1	1	0	
a	×		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
r	1	0	0	1	0	

Again the school algorithm works, however we need some observations.

If we forget the carries, the sum of column i is

$$\sum_{j+k=i+1} a_j b_k .$$

Multiplication in Logarithmic Space (1/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
r	1	0	0	1	0	

Again the school algorithm works, however we need some observations.

If we forget the carries, the sum of column i is

$$\sum_{j+k=i+1} a_j b_k .$$

E.g. for column 3 we have:

$$\sum_{j+k=4} a_j b_k = 1 + 1 + 0 = 10.$$

Multiplication in Logarithmic Space (2/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

Multiplication in Logarithmic Space (2/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

From the column sums we can compute the result bit and the carry at each position:

Multiplication in Logarithmic Space (2/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

From the column sums we can compute the result bit and the carry at each position:

$$c_0 = 0$$

$$c_i = \left\lfloor \frac{c_{i-1} + \sum_{j+k=i+1} a_j b_k}{2} \right\rfloor$$

Multiplication in Logarithmic Space (2/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

From the column sums we can compute the result bit and the carry at each position:

$$c_0 = 0$$

$$c_i = \left\lfloor \frac{c_{i-1} + \sum_{j+k=i+1} a_j b_k}{2} \right\rfloor$$

$$r_i = \left(c_{i-1} + \sum_{j+k=i+1} a_j b_k \right) \mod 2.$$

Multiplication in Logarithmic Space (2/3)

	5	4	3	2	1	0
b			1	1	0	
a	×		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

From the column sums we can compute the result bit and the carry at each position:

$$c_0 = 0$$

$$c_i = \left\lfloor \frac{c_{i-1} + \sum_{j+k=i+1} a_j b_k}{2} \right\rfloor$$

$$r_i = \left(c_{i-1} + \sum_{j+k=i+1} a_j b_k \right) \mod 2.$$

The carry is not necessary a single-bit!

Multiplication in Logarithmic Space (3/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

All we need to compute the previous sums are indices for the input bits and storing the previous element of the recurrence.

Multiplication in Logarithmic Space (3/3)

	5	4	3	2	1	0
b			1	1	0	
a	×		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

All we need to compute the previous sums are indices for the input bits and storing the previous element of the recurrence.

It is easy to see that $\sum_{j+k=i+1} a_j b_k \leq n$.

Multiplication in Logarithmic Space (3/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

All we need to compute the previous sums are indices for the input bits and storing the previous element of the recurrence.

It is easy to see that $\sum_{j+k=i+1} a_j b_k \leq n$.

Inductively we can show that $c_i \leq 2n$.

Multiplication in Logarithmic Space (3/3)

	5	4	3	2	1	0
b			1	1	0	
a	$\times$		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

All we need to compute the previous sums are indices for the input bits and storing the previous element of the recurrence.

It is easy to see that $\sum_{j+k=i+1} a_j b_k \leq n$.

Inductively we can show that $c_i \leq 2n$.

Indeed if $c_{i-1} \leq 2 \cdot n$ then $c_i = \frac{c_{i-1} + \sum_{j+k=i+1} a_j b_k}{2} \leq \frac{3n}{2} \leq 2n$.

Multiplication in Logarithmic Space (3/3)

	5	4	3	2	1	0
b			1	1	0	
a	×		0	1	1	
			1	1	0	
		1	1	0		
	0	0	0			
c		1	1	0	0	0
r	1	0	0	1	0	

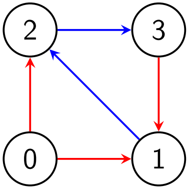
All we need to compute the previous sums are indices for the input bits and storing the previous element of the recurrence.

It is easy to see that $\sum_{j+k=i+1} a_j b_k \leq n$.

Inductively we can show that $c_i \leq 2n$.

Indeed if $c_{i-1} \leq 2 \cdot n$ then $c_i = \frac{c_{i-1} + \sum_{j+k=i+1} a_j b_k}{2} \leq \frac{3n}{2} \leq 2n$. Hence all the numbers we need can be stored in $\mathcal{O}(\log n)$ bits.

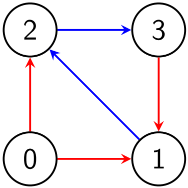
Binary Encoding of a Structure



$$G = \langle V, E, R, 0, 3 \rangle$$
$$n = 4$$
$$E = \{(1,2),(2,3)\}$$
$$R = \{(0,1),(0,2),(3,1)\}$$

The binary encoding of G is:

Binary Encoding of a Structure

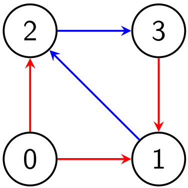


$$G = \langle V, E, R, 0, 3\rangle$$
$$n = 4$$
$$E = \{(1,2),(2,3)\}$$
$$R = \{(0,1),(0,2),(3,1)\}$$

The binary encoding of G is:

$$\mathrm{bin}(G) = \overbrace{0000001000010100}^{E}\overbrace{0110000000000100}^{R}\ \overbrace{00}^{0}\ \overbrace{11}^{3}\ .$$

Binary Encoding of a Structure



$$G = \langle V, E, R, 0, 3\rangle$$
$$n = 4$$
$$E = \{(1,2),(2,3)\}$$
$$R = \{(0,1),(0,2),(3,1)\}$$

The binary encoding of G is:

$$\mathrm{bin}(G) = \overbrace{0000001000010100}^{E}\overbrace{0110000000000100}^{R}\ \overbrace{00}^{0}\ \overbrace{11}^{3}\ .$$

Observe that $E(1,2)$ corresponds to bit

$$1 \cdot n + 2 + 1 = 1 \cdot 4 + 2 + 1 = 6.$$

Also $|\mathrm{bin}(G)| = n^2 + n^2 + \lceil \log n \rceil + \lceil \log n \rceil$.

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Proof

Let $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$.

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Proof

Let $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$. A boolean *FO*-query is determined by a sentence $\phi \in \mathcal{L}(\sigma)$.

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Proof

Let $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$. A boolean FO-query is determined by a sentence $\phi \in \mathcal{L}(\sigma)$. Let $\mathcal{A} \in \mathrm{STRUCT}(\sigma)$.

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Proof

Let $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$. A boolean *FO*-query is determined by a sentence $\phi \in \mathcal{L}(\sigma)$. Let $\mathcal{A} \in \mathrm{STRUCT}(\sigma)$. We will construct a logspace deterministic Turing machine M, such that:

Theorem

The set of boolean queries describable in first order logic can be computed in deterministic logspace, i.e. $FO \subseteq L$.

Proof

Let $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$. A boolean *FO*-query is determined by a sentence $\phi \in \mathcal{L}(\sigma)$. Let $\mathcal{A} \in \text{STRUCT}(\sigma)$. We will construct a logspace deterministic Turing machine M, such that:

$$\mathcal{A} \models \phi \iff M(\text{bin}(\mathcal{A})) \downarrow .$$

Proof

First we to compute the size of the universe.

Proof

First we to compute the size of the universe. M knows that its input is of the form $\mathrm{bin}(\mathcal{A})$ for some $\mathcal{A}$.

Proof

First we to compute the size of the universe. M knows that its input is of the form bin($\mathcal{A}$) for some $\mathcal{A}$. Hence M's input length is

$$f(n) = n^{a_1} + \ldots + n^{a_r} + s \cdot \lceil \log n \rceil$$

for some n.

Proof

First we to compute the size of the universe. M knows that its input is of the form $\text{bin}(\mathcal{A})$ for some $\mathcal{A}$. Hence M's input length is

$$f(n) = n^{a_1} + \ldots + n^{a_r} + s \cdot \lceil \log n \rceil$$

for some n. This n can be calculated as follows:

Proof

First we to compute the size of the universe. M knows that its input is of the form $\text{bin}(\mathcal{A})$ for some $\mathcal{A}$. Hence M's input length is

$$f(n) = n^{a_1} + \ldots + n^{a_r} + s \cdot \lceil \log n \rceil$$

for some n. This n can be calculated as follows:

M computes iteratively $f(1)$, $f(2)$, etc. until M computes an $f(j)$ that is equal to the size of M's input. This j is the required n.

Proof

First we to compute the size of the universe. M knows that its input is of the form $\text{bin}(\mathcal{A})$ for some $\mathcal{A}$. Hence M's input length is

$$f(n) = n^{a_1} + \ldots + n^{a_r} + s \cdot \lceil \log n \rceil$$

for some n. This n can be calculated as follows:

M computes iteratively $f(1)$, $f(2)$, etc. until M computes an $f(j)$ that is equal to the size of M's input. This j is the required n.

$\lceil \log j \rceil$ is simply the length of j's binary representation so it can easily be computed from j.

Proof

First we to compute the size of the universe. M knows that its input is of the form $\text{bin}(\mathcal{A})$ for some $\mathcal{A}$. Hence M's input length is

$$f(n) = n^{a_1} + \ldots + n^{a_r} + s \cdot \lceil \log n \rceil$$

for some n. This n can be calculated as follows:

M computes iteratively $f(1)$, $f(2)$, etc. until M computes an $f(j)$ that is equal to the size of M's input. This j is the required n.

$\lceil \log j \rceil$ is simply the length of j's binary representation so it can easily be computed from j.

Also it is is easy to see that $\log f(n) = \mathcal{O}(\log n)$.

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\ldots(Q_k x_k)\alpha(x_1, \ldots, x_k)$$

where $\alpha(x1, \ldots, x_k)$ is quantifier free.

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\ldots(Q_k x_k)\alpha(x_1,\ldots,x_k)$$

where $\alpha(x1,\ldots,x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\ldots(Q_k x_k)\alpha(x_1,\ldots,x_k)$$

where $\alpha(x1,\ldots,x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\ldots(Q_k x_k)\alpha(x_1, \ldots, x_k)$$

where $\alpha(x1, \ldots, x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

- $R_i(c_{j_1}, \ldots, c_{j_{a_i}})$

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\dots(Q_k x_k)\alpha(x_1,\dots,x_k)$$

where $\alpha(x1,\dots,x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

- $R_i(c_{j_1},\dots,c_{j_{a_i}})$
- $i \leq j$

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\dots(Q_k x_k)\alpha(x_1, \dots, x_k)$$

where $\alpha(x1, \dots, x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

- $R_i(c_{j_1}, \dots, c_{j_{a_i}})$
- $i \leq j$
- $\mathrm{BIT}(i, j)$

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\dots(Q_k x_k)\alpha(x_1,\dots,x_k)$$

where $\alpha(x1,\dots,x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

- $R_i(c_{j_1},\dots,c_{j_{a_i}})$
- $i \leq j$
- $\mathrm{BIT}(i,j)$
- $c_i \approx c_j$

Proof

We assume that ϕ is in prenex normal form:

$$\phi = (\exists x_1)(\forall x_2)\ldots(Q_k x_k)\alpha(x_1,\ldots,x_k)$$

where $\alpha(x1,\ldots,x_k)$ is quantifier free. We will construct the Turing machine by induction on k.

Induction Base: ϕ is quantifier-free sentence, i.e. it is a boolean combination of the following:

- $R_i(c_{j_1},\ldots,c_{j_{a_i}})$
- $i \leq j$
- $\mathrm{BIT}(i,j)$
- $c_i \approx c_j$

In that case M can test whether $\mathcal{A} \models \phi$ by only using a pointer in the input (which is of the form $\mathrm{bin}(\mathcal{A})$).

Proof

Induction Base (Cont'd): Assume for example that M wants to test the validity of $R_3(c_2, \max, c_1)$.

Proof

Induction Base (Cont'd): Assume for example that M wants to test the validity of $R_3(c_2, \max, c_1)$. Then M has to move its input-head to bit number:

Proof

Induction Base (Cont'd): Assume for example that M wants to test the validity of $R_3(c_2, \max, c_1)$. Then M has to move its input-head to bit number:

$$\underbrace{n^{a_1} + n^{a_2}}_{\text{bits for } R_1 \text{ and } R_2} + \underbrace{n^2 \cdot c_2 + n \cdot (n-1) + c_1 + 1}_{\text{position of } R_3(c_2,\max,c_1)}.$$

The above bit is '1' iff $\mathcal{A} \models R_3(c_2, \max, c_1)$.

Proof

Induction Base (Cont'd): Assume for example that M wants to test the validity of $R_3(c_2, \max, c_1)$. Then M has to move its input-head to bit number:

$$\underbrace{n^{a_1} + n^{a_2}}_{\text{bits for } R_1 \text{ and } R_2} + \underbrace{n^2 \cdot c_2 + n \cdot (n-1) + c_1 + 1}_{\text{position of } R_3(c_2,\max,c_1)}.$$

The above bit is '1' iff $\mathcal{A} \models R_3(c_2, \max, c_1)$. This way, M can test whether $\mathcal{A} \models \phi$.

Proof

Induction Base (Cont'd): Assume for example that M wants to test the validity of $R_3(c_2, \max, c_1)$. Then M has to move its input-head to bit number:

$$\underbrace{n^{a_1} + n^{a_2}}_{\text{bits for } R_1 \text{ and } R_2} + \underbrace{n^2 \cdot c_2 + n \cdot (n-1) + c_1 + 1}_{\text{position of } R_3(c_2,\max,c_1)}.$$

The above bit is '1' iff $\mathcal{A} \models R_3(c_2, \max, c_1)$. This way, M can test whether $\mathcal{A} \models \phi$.

Induction Hypothesis: Assume that all FO-queries with $k-1$ quantifiers are logspace-computable:

$$\phi = (\forall x_2) \ldots (Q_k x_k) \alpha(x2, \ldots, x_k)$$

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable.

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \ldots (Q_k x_k) \alpha(x_1, x_2, \ldots, x_k).$$

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \dots (Q_k x_k) \alpha(x_1, x_2, \dots, x_k).$$

In order to compute $\exists x_1 \psi(x_1)$ the Turing Machine M has to simply create all possible constants c in its work tape

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \dots (Q_k x_k) \alpha(x_1, x_2, \dots, x_k).$$

In order to compute $\exists x_1 \psi(x_1)$ the Turing Machine M has to simply create all possible constants c in its work tape and check for each one of them, whether $\psi(c)$ holds.

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \dots (Q_k x_k)\alpha(x_1, x_2, \dots, x_k).$$

In order to compute $\exists x_1 \psi(x_1)$ the Turing Machine M has to simply create all possible constants c in its work tape and check for each one of them, whether $\psi(c)$ holds. M can do this, since each possible constant can be represented by $\log n$ bits and $\psi(c)$ is an FO-sentence with $k - 1$ quantifiers,

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \dots (Q_k x_k) \alpha(x_1, x_2, \dots, x_k).$$

In order to compute $\exists x_1 \psi(x_1)$ the Turing Machine M has to simply create all possible constants c in its work tape and check for each one of them, whether $\psi(c)$ holds. M can do this, since each possible constant can be represented by $\log n$ bits and $\psi(c)$ is an FO-sentence with $k-1$ quantifiers, thus it is logspace computable by i.h.

Proof.

Induction Step: Our goal is to show that all FO-queries with k quantifiers are logspace computable. Assume that

$$\psi(x_1) = (\forall x_2) \dots (Q_k x_k)\alpha(x_1, x_2, \dots, x_k).$$

In order to compute $\exists x_1 \psi(x_1)$ the Turing Machine M has to simply create all possible constants c in its work tape and check for each one of them, whether $\psi(c)$ holds. M can do this, since each possible constant can be represented by $\log n$ bits and $\psi(c)$ is an FO-sentence with $k-1$ quantifiers, thus it is logspace computable by i.h. A universal quantifier is handled in a similar way. □

Overview

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important).

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important). A configuration of M looks like:

$$(q, i, w_1, w_2),$$

where

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important). A configuration of M looks like:

$$(q, i, w_1, w_2),$$

where

- q is M's current state

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important). A configuration of M looks like:

$$(q, i, w_1, w_2),$$

where

- q is M's current state
- i is the position of the cursor in the read-only input

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important). A configuration of M looks like:

$$(q, i, w_1, w_2),$$

where

- q is M's current state
- i is the position of the cursor in the read-only input
- w_1 are the contents of the work tape until and including the work tape cursor

Let M be a logspace bounded Turing Machine that can only accept or reject its input (thus the output tape is not important). A configuration of M looks like:

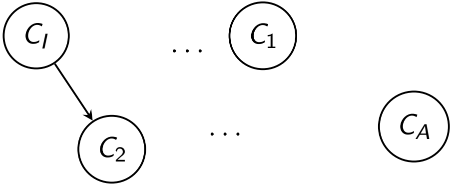
$$(q, i, w_1, w_2),$$

where

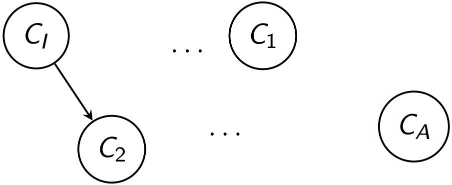
- q is M's current state
- i is the position of the cursor in the read-only input
- w_1 are the contents of the work tape until and including the work tape cursor
- w_2 are the rest of the work tapes' contents

All the configurations of M can bee seen as nodes in a graph.

All the configurations of M can bee seen as nodes in a graph.

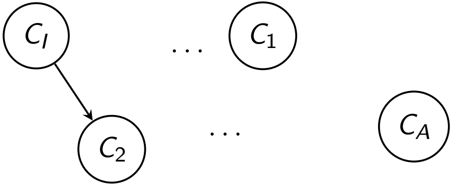


All the configurations of M can bee seen as nodes in a graph.



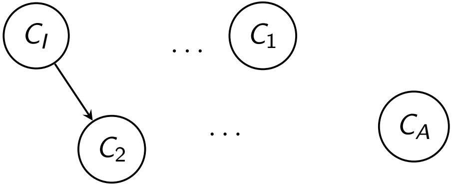
- An edge (C, C') corresponds to a transition. I.e. on configuration C, based on what M sees on the input tape and the work tape, M moves the cursors and possibly writes something to the output tape that leads to configuration C'.

All the configurations of M can bee seen as nodes in a graph.



- An edge (C, C') corresponds to a transition. I.e. on configuration C, based on what M sees on the input tape and the work tape, M moves the cursors and possibly writes something to the output tape that leads to configuration C'.
- Since configurations look like (q, i, w_1, w_2), there are at most $|Q| \cdot n \cdot |\Sigma|^{2 \log n} = \mathcal{O}(n^c)$ nodes in the graph.

All the configurations of M can bee seen as nodes in a graph.



- An edge (C, C') corresponds to a transition. I.e. on configuration C, based on what M sees on the input tape and the work tape, M moves the cursors and possibly writes something to the output tape that leads to configuration C'.
- Since configurations look like (q, i, w_1, w_2), there are at most $|Q| \cdot n \cdot |\Sigma|^{2 \log n} = \mathcal{O}(n^c)$ nodes in the graph.
- M accepts its input iff the accepting configuration (C_A) is reachable from the initial configuration (C_I).

The previous discussion indicates that every space complexity class should have some form of reachability problem as a natural complete problem.

The previous discussion indicates that every space complexity class should have some form of reachability problem as a natural complete problem.

We define the following problem:

$$\text{REACH} = \{(G, s, t) \mid G \text{ is directed and there is path from } s \text{ to } t\}.$$

The previous discussion indicates that every space complexity class should have some form of reachability problem as a natural complete problem.

We define the following problem:

$$\text{REACH} = \{(G, s, t) \mid G \text{ is directed and there is path from } s \text{ to } t\}.$$

We will show that REACH is *NL*-complete via *FO*-reductions.

REACH $\in NL$

The following simple non-determinstic logspace algorithm solves REACH.

REACH $\in NL$

The following simple non-determinstic logspace algorithm solves REACH.

```
b:= s;
while (not (b = t)) do {
      a := b;
      nondeterministically choose new b;
      if (not E(a,b)) then reject;
}
accept;
```

REACH $\in NL$

The following simple non-determinstic logspace algorithm solves REACH.

```
b:= s;
while (not (b = t)) do {
      a := b;
      nondeterministically choose new b;
      if (not E(a,b)) then reject;
}
accept;
```

The above algorithm needs only store a and b, which have size $\log n$.

Theorem

REACH *is complete for NL via FO-reductions.*

Theorem

REACH *is complete for NL via FO-reductions.*

Proof

- $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$

Theorem

REACH *is complete for NL via FO-reductions.*

Proof

- $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$
- $\tau_g = \langle E^2, s, t \rangle$ (i.e. the vocabulary of directed graphs with two specified nodes)

Theorem

REACH *is complete for NL via FO-reductions.*

Proof

- $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$
- $\tau_g = \langle E^2, s, t \rangle$ (i.e. the vocabulary of directed graphs with two specified nodes)
- Let N be the logspace nondeterministic Turing Machine that accepts a subset of STRUCT$[\sigma]$

Theorem

REACH *is complete for NL via FO-reductions.*

Proof

- $\sigma = \langle R_1^{a_1}, \ldots, R_r^{a_r}, c_1, \ldots, c_s \rangle$
- $\tau_g = \langle E^2, s, t \rangle$ (i.e. the vocabulary of directed graphs with two specified nodes)
- Let N be the logspace nondeterministic Turing Machine that accepts a subset of STRUCT$[\sigma]$
- We will construct an FO-reduction I : STRUCT$[\sigma] \to$ STRUCT$[\tau_g]$ such that for all $\mathcal{A} \in$ STRUCT$[\sigma]$

$$N(\mathrm{bin}(\mathcal{A})) \downarrow \Longleftrightarrow I(\mathcal{A}) \in \mathrm{REACH}.$$

Proof Sketch

- Assume that N uses at most $c \cdot \log n$ bits of work tape

Proof Sketch

- Assume that N uses at most $c \cdot \log n$ bits of work tape
- Remember that the number of N's states and the number of the relations and constants in σ are constants that do not depend on the input size.

Proof Sketch

- Assume that N uses at most $c \cdot \log n$ bits of work tape
- Remember that the number of N's states and the number of the relations and constants in σ are constants that do not depend on the input size.
- Let $a = max\{a_i \mid 1 \leq i \leq r\}$ and $k = 1 + a + c$

Proof Sketch

- Assume that N uses at most $c \cdot \log n$ bits of work tape
- Remember that the number of N's states and the number of the relations and constants in σ are constants that do not depend on the input size.
- Let $a = max\{a_i \mid 1 \leq i \leq r\}$ and $k = 1 + a + c$
- The reduction I will be a k-ary FO-query. I.e. the universe of $I(\mathcal{A})$ will consist of k-tuples from $\mathcal{A}$'s elements.

Proof Sketch

- A configuration of N can be encoded in a k-tuple of variables, i.e. $C = (p, r_1, \ldots, r_a, w_1, \ldots, w_c)$, where p and all the r_i's and w_i's are elements of the universe, i.e. $\log n$-bit numbers.

Proof Sketch

- A configuration of N can be encoded in a k-tuple of variables, i.e. $C = (p, r_1, \ldots, r_a, w_1, \ldots, w_c)$, where p and all the r_i's and w_i's are elements of the universe, i.e. $\log n$-bit numbers.
- N is looking at an 1 in the binary representation of relation R_i iff $\mathcal{A} \models R_i(r_1, \ldots, r_a)$

Proof Sketch

- A configuration of N can be encoded in a k-tuple of variables, i.e. $C = (p, r_1, \ldots, r_a, w_1, \ldots, w_c)$, where p and all the r_i's and w_i's are elements of the universe, i.e. $\log n$-bit numbers.
- N is looking at an 1 in the binary representation of relation R_i iff $\mathcal{A} \models R_i(r_1, \ldots, r_a)$
- The w_i's contain the contents of N's worktape

Proof Sketch

- A configuration of N can be encoded in a k-tuple of variables, i.e. $C = (p, r_1, \ldots, r_a, w_1, \ldots, w_c)$, where p and all the r_i's and w_i's are elements of the universe, i.e. $\log n$-bit numbers.
- N is looking at an 1 in the binary representation of relation R_i iff $\mathcal{A} \models R_i(r_1, \ldots, r_a)$
- The w_i's contain the contents of N's worktape
- p encodes the current state of N, which R_i or which c_i the input head is looking at and a pointer for the work tape

Proof Sketch

- A configuration of N can be encoded in a k-tuple of variables, i.e. $C = (p, r_1, \ldots, r_a, w_1, \ldots, w_c)$, where p and all the r_i's and w_i's are elements of the universe, i.e. $\log n$-bit numbers.
- N is looking at an 1 in the binary representation of relation R_i iff $\mathcal{A} \models R_i(r_1, \ldots, r_a)$
- The w_i's contain the contents of N's worktape
- p encodes the current state of N, which R_i or which c_i the input head is looking at and a pointer for the work tape
- since the number of states, relations and constants is independent of the input size and a pointer for the work tape needs $\mathcal{O}(\log \log n)$ bits, p has enough space to store all the necessary information (for large enough n).

Proof Sketch

- Now we build the desired k-ary FO-reduction

$$I = \lambda_{C,C'} \langle true, \phi_N, \alpha, \omega \rangle.$$

- True in the above relation means that the set of nodes in the graph $I(\mathcal{A})$ is equal to the set of all possible configurations
- Formulas ϕ_N, α and ω represent the edge relation, the source node s and the target node t in the created graph
- $\mathcal{A} \models \alpha(C)$ iff C is the unique initial configuration of N
- $\mathcal{A} \models \omega(C)$ iff C is the unique accepting configuration

Proof Sketch.

- $\mathcal{A} \models \phi_N(C, C')$ iff there is a valid move of N from C to C'. A move from C to C' has the following meaning:

Proof Sketch.

- $\mathcal{A} \models \phi_N(C, C')$ iff there is a valid move of N from C to C'. A move from C to C' has the following meaning:

  "if on configuration C we examine the input bit b then the input head moves to direction d_i, the work head moves to direction d_w and we write bit b' on the work tape."

Proof Sketch.

- $\mathcal{A} \models \phi_N(C, C')$ iff there is a valid move of N from C to C'. A move from C to C' has the following meaning:

  "if on configuration C we examine the input bit b then the input head moves to direction d_i, the work head moves to direction d_w and we write bit b' on the work tape."

  The above information can be extracted from the k variables $p, r_1, \ldots, r_a, w_1, \ldots, w_c$ that describe C and C'. $\square$

Overview

1. $FO \subseteq L$
2. NL-Completeness
3. L-Completeness
4. P-Completeness
5. On FO-Reductions

A slight modification of REACH gives us a natural complete problem for L.

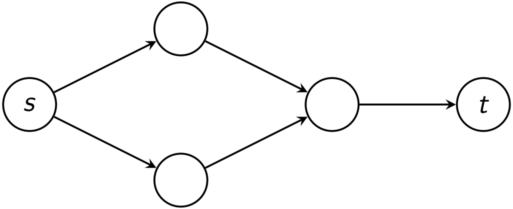
A slight modification of REACH gives us a natural complete problem for L.

The deterministic version of REACH is the following problem:

$$\text{REACH}_d = \{(G, s, t) \mid G \text{ is directed and there is a deterministic path from } s \text{ to } t\}.$$

The path P is deterministic if for every $(x, y) \in P$, (x, y) is the unique edge leaving x.

Difference Between REACH and REACH_d



It holds that $(G, s, t) \in \text{REACH}$ but $(G, s, t) \notin \text{REACH}_d$

$\text{REACH}_d \in L$

The following logspace algortihm answers REACH_d:

$$
\begin{array}{l}
b := s;\ \ i := 0;\ \ n = |G|; \\
\textbf{while}\ \Big[(b \neq t) \wedge (i < n) \wedge (\exists! a)E(b,a)\Big]\ \textbf{do}\ \{ \\
\qquad b := \text{ the unique } a \text{ for which } E(b,a) \\
\qquad i := i + 1 \\
\} \\
\textbf{if}\ b = t\ \textbf{then}\ \text{accept}\ \textbf{else}\ \text{reject.}
\end{array}
$$

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

Theorem

REACH_d *is complete for L via FO-reductions.*

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

Theorem

REACH_d *is complete for L via FO-reductions.*

Proof.

We repeat the same construction as we did for REACH and NL.

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

Theorem

REACH_d *is complete for* L *via* FO*-reductions.*

Proof.

We repeat the same construction as we did for REACH and NL. The only difference is that the Turing Machine is now a deterministic one,

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

Theorem

REACH_d *is complete for* L *via* FO*-reductions.*

Proof.

We repeat the same construction as we did for REACH and NL. The only difference is that the Turing Machine is now a deterministic one, thus every configuration has a unique next configuration,

REACH_d is a Natural Complete Problem for L

The definition of REACH_d was made such that the following theorem holds.

Theorem

REACH_d *is complete for L via FO-reductions.*

Proof.

We repeat the same construction as we did for REACH and NL. The only difference is that the Turing Machine is now a deterministic one, thus every configuration has a unique next configuration, which implies that every node in the constructed graph has a unique edge that leaves it. □

Overview

1. $FO \subseteq L$
2. NL-Completeness
3. L-Completeness
4. P-Completeness
5. On FO-Reductions

Using similar ideas we construct a natural complete problem for $P = ASPACE[\log n]$.

Using similar ideas we construct a natural complete problem for $P = ASPACE[\log n]$.

Definition

An alternating graph is a structure $G = \langle V, E, A, s, t \rangle$, where the edges are directed and the vertices are labelled universal (A) or existential ($V \setminus A$).

Using similar ideas we construct a natural complete problem for $P = ASPACE[\log n]$.

Definition

An alternating graph is a structure $G = \langle V, E, A, s, t \rangle$, where the edges are directed and the vertices are labelled universal (A) or existential ($V \setminus A$).

Let G be an alternating graph. P^G is the smallest binary relation that satisfies the following:

- $P^G(x, x)$

Using similar ideas we construct a natural complete problem for $P = ASPACE[\log n]$.

Definition

An alternating graph is a structure $G = \langle V, E, A, s, t \rangle$, where the edges are directed and the vertices are labelled universal (A) or existential ($V \setminus A$).

Let G be an alternating graph. P^G is the smallest binary relation that satisfies the following:

- $P^G(x, x)$
- If x is existential and for some edge (x, z) we have $P^G(z, y)$, then $P^G(x, y)$

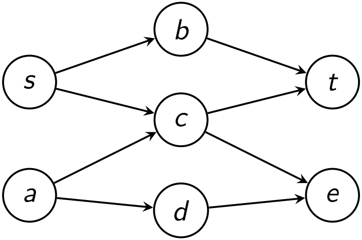
Using similar ideas we construct a natural complete problem for $P = ASPACE[\log n]$.

Definition

An alternating graph is a structure $G = \langle V, E, A, s, t \rangle$, where the edges are directed and the vertices are labelled universal (A) or existential ($V \setminus A$).

Let G be an alternating graph. P^G is the smallest binary relation that satisfies the following:

- $P^G(x, x)$
- If x is existential and for some edge (x, z) we have $P^G(z, y)$, then $P^G(x, y)$
- If x is universal, x has at least one outgoing edge and for all edges (x, z) we have $P^G(z, y)$, then $P^G(x, y)$



Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

P^G	s	a	b	c	d	e	t
s	1	0	0	0	0	0	1
a	0	1	0	0	0	1	0
b	0	0	1	0	0	0	1
c	0	0	0	1	0	1	1
d	0	0	0	0	1	1	0
e	0	0	0	0	0	1	0
t	0	0	0	0	0	0	1

Alternating Reachability

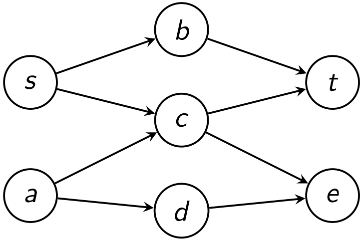
We define $\mathrm{REACH}_a = \left\{ (G, s, t) \;\middle|\; P^G(s, t) \right\}$.
The previous graph G with nodes s and t is a yes instance for REACH_a.

$\text{REACH}_a \in P$

make QUEUE empty; mark(t); **insert** t into QUEUE;
while QUEUE not empty **do** {
    remove first element, x, from QUEUE;
    for each unmarked vertex y such that E(y,x) **do** {
        delete edge (y,x);
        if y is existential or y has no outgoing edges **then**
            { mark(y); **insert** y into QUEUE}
        }
    };
if s is marked **then** accept **else** reject;

Remember: t is the target node, s is the source node and we wish to test whether there is an alternating path from s to t.

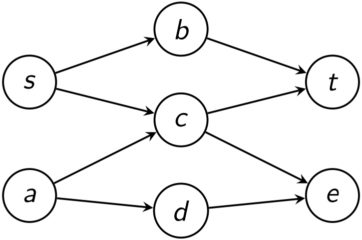
A Run of the Algorithm on G



Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

A Run of the Algorithm on G

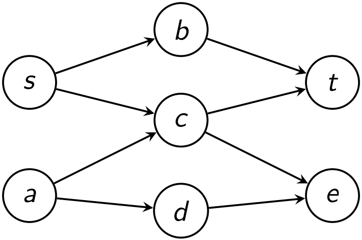


Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

- t is marked, added in QUEUE and then removed

A Run of the Algorithm on G

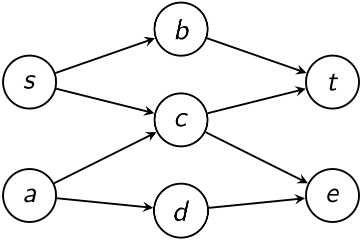


Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

- t is marked, added in QUEUE and then removed
- (b, t) and (c, t) are deleted and b, c are marked and added in QUEUE

A Run of the Algorithm on G

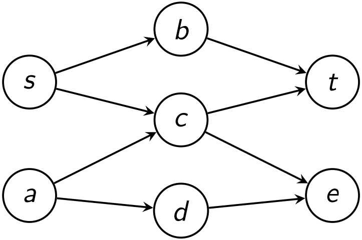


Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

- t is marked, added in QUEUE and then removed
- (b, t) and (c, t) are deleted and b, c are marked and added in QUEUE
- b is removed from QUEUE and (s, b) is deleted

A Run of the Algorithm on G

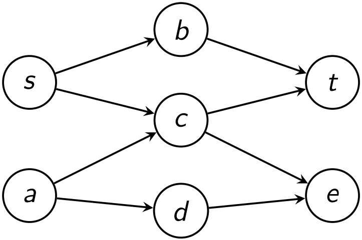


Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

- t is marked, added in QUEUE and then removed
- (b, t) and (c, t) are deleted and b, c are marked and added in QUEUE
- b is removed from QUEUE and (s, b) is deleted
- c is removed from QUEUE and (a, c) and (s, c) are deleted

A Run of the Algorithm on G



Universal Nodes: s and a

Existential Nodes: b, c, d, t, e

- t is marked, added in QUEUE and then removed
- (b, t) and (c, t) are deleted and b, c are marked and added in QUEUE
- b is removed from QUEUE and (s, b) is deleted
- c is removed from QUEUE and (a, c) and (s, c) are deleted
- s is marked and added in QUEUE (success!)

A Natural Complete Problem for $ASPACE[\log n]$

As before, REACH_a was defined such that the following theorem holds.

A Natural Complete Problem for $ASPACE[\log n]$

As before, REACH_a was defined such that the following theorem holds.

Theorem

REACH_a *is complete for P via FO-reductions.*

A Natural Complete Problem for $ASPACE[\log n]$

As before, REACH_a was defined such that the following theorem holds.

Theorem

REACH_a *is complete for P via FO-reductions.*

Proof Sketch.

The same construction as before works. The L-Turing machine is now an $ASPACE\,[\log n]$-Turing Machine. We have to make sure that the universal states are mapped to universal nodes. □

Overview

Closure under FO-reductions

Let C be a complexity class and $\mathcal{L}$ be a language that we use to express problems (e.g. FO,$SO\exists$).

Closure under *FO*-reductions

Let C be a complexity class and $\mathcal{L}$ be a language that we use to express problems (e.g. FO,$SO\exists$).

C is closed under FO-reductions if whenever $B \in C$ and $A \leq_{fo} B$ then $A \in C$.

Closure under FO-reductions

Let C be a complexity class and $\mathcal{L}$ be a language that we use to express problems (e.g. FO,$SO\exists$).

C is closed under FO-reductions if whenever $B \in C$ and $A \leq_{fo} B$ then $A \in C$.

$\mathcal{L}$ is closed under FO-reductions if whenever B is expressible in $\mathcal{L}$ and $A \leq_{fo} B$ then A is expressible in $\mathcal{L}$ too.

The Power of FO-reductions

As we have seen the complexity of queries expressible in FO-logic is relative low.

The Power of FO-reductions

As we have seen the complexity of queries expressible in FO-logic is relative low.

Thus the intuitive meaning of $A \leq_{fo} B$ is that A is not more difficult than B.

The Power of *FO*-reductions

As we have seen the complexity of queries expressible in *FO*-logic is relative low.

Thus the intuitive meaning of $A \leq_{fo} B$ is that A is not more difficult than B.

Despite the limited power of *FO*-logic almost all the complexity classes and languages that we will consider are closed under *FO*-reductions.

The Power of *FO*-reductions

As we have seen the complexity of queries expressible in *FO*-logic is relative low.

Thus the intuitive meaning of $A \leq_{fo} B$ is that A is not more difficult than B.

Despite the limited power of *FO*-logic almost all the complexity classes and languages that we will consider are closed under *FO*-reductions.

At least for complexity classes one should suspect that, since almost all of them are closed under logspace reductions and $FO \subseteq L$.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via *FO*-reductions.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via FO-reductions.
3. Show that $\mathcal{L}$ is closed under FO-reductions.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via FO-reductions.
3. Show that $\mathcal{L}$ is closed under FO-reductions.
4. Express T in $\mathcal{L}$.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via *FO*-reductions.
3. Show that $\mathcal{L}$ is closed under *FO*-reductions.
4. Express T in $\mathcal{L}$.

From 2-4 we can show that $C \subseteq \mathcal{L}$.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via *FO*-reductions.
3. Show that $\mathcal{L}$ is closed under *FO*-reductions.
4. Express T in $\mathcal{L}$.

From 2-4 we can show that $C \subseteq \mathcal{L}$. Indeed, let $A \in C$.

Methodology for Finding the Descriptive Analogue of a Complexity Class

Let $\mathcal{L}$ be a language and let C be a complexity class. In order to show that $\mathcal{L} = C$, i.e. that a query belongs in C if and only if it is expressible in $\mathcal{L}$, we do the following:

1. Create a C-algorithm that can test for every $\mathcal{L}$-sentence ϕ and every $\mathcal{L}$-structure $\mathcal{A}$, whether $\mathcal{A} \models \phi$. This shows that $\mathcal{L} \subseteq C$.
2. Find a boolean query T that is complete for C via FO-reductions.
3. Show that $\mathcal{L}$ is closed under FO-reductions.
4. Express T in $\mathcal{L}$.

From 2-4 we can show that $C \subseteq \mathcal{L}$. Indeed, let $A \in C$. Then $A \leq_{fo} T$, hence A is expressible in $\mathcal{L}$.

Summary

Today we've seen

Summary

Today we've seen :

- $FO \subseteq L$

Summary

Today we've seen :

- $FO \subseteq L$
- Reachability problems are typically the natural complete problems for space complexity classes.

Summary

Today we've seen :

- $FO \subseteq L$
- Reachability problems are typically the natural complete problems for space complexity classes.
- The natural complete problems for NL, L and P are REACH, REACH_d and REACH_a respevtively.

Summary

Today we've seen :

- $FO \subseteq L$
- Reachability problems are typically the natural complete problems for space complexity classes.
- The natural complete problems for NL, L and P are REACH, REACH_d and REACH_a respevtively.
- A strategy for finding the descriptive analogues of complexity classes.