Reductions

Peter Hajnal

Bolyai Institute, SZTE, Szeged

2023 fall

The Goal

The Goal

In the previous sections, we introduced several complexity classes ($\mathcal{L}$, $\mathcal{P}$, $\mathcal{D}$, $\mathcal{EXP}$).

The Goal

In the previous sections, we introduced several complexity classes ($\mathcal{L}$, $\mathcal{P}$, $\mathcal{D}$, $\mathcal{EXP}$). We saw examples of central mathematical problems: HAMILTON, $\overrightarrow{st}$-REACHABILITY, WORD PROBLEM, FACTORIZATION, etc.

The Goal

In the previous sections, we introduced several complexity classes ($\mathcal{L}$, $\mathcal{P}$, $\mathcal{D}$, $\mathcal{EXP}$). We saw examples of central mathematical problems: HAMILTON, $\overrightarrow{st}$-REACHABILITY, WORD PROBLEM, FACTORIZATION, etc.

A central question is to place individual problems in the introduced hierarchy.

The Goal

In the previous sections, we introduced several complexity classes ($\mathcal{L}$, $\mathcal{P}$, $\mathcal{D}$, $\mathcal{EXP}$). We saw examples of central mathematical problems: HAMILTON, $\overrightarrow{st}$-REACHABILITY, WORD PROBLEM, FACTORIZATION, etc.

A central question is to place individual problems in the introduced hierarchy.

The goal is to determine the location/complexity of an important problem as accurately as possible.

Breaking Down the Goal

Breaking Down the Goal

Determining the *location* has two tasks.

Breaking Down the Goal

Determining the *location* has two tasks.

(1) Providing an upper estimate of the complexity of a language/problem L (i.e., proving that $L \in \mathcal{C}_1$) requires giving an algorithm and analyzing it to show membership in class $\mathcal{C}_1$. Such results, predating the advent of computers, constitute the starting point of algorithm theory.

Breaking Down the Goal

Determining the *location* has two tasks.

(1) Providing an upper estimate of the complexity of a language/problem L (i.e., proving that $L \in \mathcal{C}_1$) requires giving an algorithm and analyzing it to show membership in class $\mathcal{C}_1$. Such results, predating the advent of computers, constitute the starting point of algorithm theory.

(2) Determining a lower estimate of the complexity of L (i.e., proving that $L \notin \mathcal{C}_2$) is more complex. It demands identifying a theoretical difficulty that prevents solving a problem efficiently, no matter how clever we are or how brilliant our ideas.

Breaking Down the Goal

Determining the *location* has two tasks.

(1) Providing an upper estimate of the complexity of a language/problem L (i.e., proving that $L \in \mathcal{C}_1$) requires giving an algorithm and analyzing it to show membership in class $\mathcal{C}_1$. Such results, predating the advent of computers, constitute the starting point of algorithm theory.

(2) Determining a lower estimate of the complexity of L (i.e., proving that $L \notin \mathcal{C}_2$) is more complex. It demands identifying a theoretical difficulty that prevents solving a problem efficiently, no matter how clever we are or how brilliant our ideas.

(1) is easy: Designing efficient algorithms.

Proving Difficulty

Proving Difficulty

Task (2) is much more challenging; we can say that almost no results have been obtained in this direction. Two important *attack directions* are mentioned:

Proving Difficulty

Task (2) is much more challenging; we can say that almost no results have been obtained in this direction. Two important *attack directions* are mentioned:

Proving Difficulty

Task (2) is much more challenging; we can say that almost no results have been obtained in this direction. Two important *attack directions* are mentioned:

(a) Replace the general model of Turing machines with a simpler computational model (which is not a universal computational concept), and try to prove lower estimates there. For example, for the SORTING problem (sorting n numbers in ascending order), we might work with just two comparisons of input numbers and branch based on the result. It is natural since most algorithms work this way. It can be demonstrated that at least $n \log n$ comparisons are needed to compute the output.

Proving Difficulty

Task (2) is much more challenging; we can say that almost no results have been obtained in this direction. Two important *attack directions* are mentioned:

(a) Replace the general model of Turing machines with a simpler computational model (which is not a universal computational concept), and try to prove lower estimates there. For example, for the SORTING problem (sorting n numbers in ascending order), we might work with just two comparisons of input numbers and branch based on the result. It is natural since most algorithms work this way. It can be demonstrated that at least $n \log n$ comparisons are needed to compute the output.

(b) Do not investigate (absolute) difficulty; instead, focus on relative difficulty. The goal is to prove only that a problem is at least as difficult as another.

2(b) Illustrated

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

A reduction is an assignment/calculation that, instead of answering this question, computes a new question: "I will describe a new input $\widetilde{\omega}$, and I will ask if it belongs to a new language $\widetilde{L}$."

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

A reduction is an assignment/calculation that, instead of answering this question, computes a new question: "I will describe a new input $\widetilde{\omega}$, and I will ask if it belongs to a new language $\widetilde{L}$."

"If someone can answer this question, then I can answer the original question." Moreover, the answer to the new question will be the same as the answer to my question.

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

A reduction is an assignment/calculation that, instead of answering this question, computes a new question: "I will describe a new input $\widetilde{\omega}$, and I will ask if it belongs to a new language $\widetilde{L}$."

"If someone can answer this question, then I can answer the original question." Moreover, the answer to the new question will be the same as the answer to my question.

Standing on the shoulders of $\widetilde{L}$, L is not hard.

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

A reduction is an assignment/calculation that, instead of answering this question, computes a new question: "I will describe a new input $\widetilde{\omega}$, and I will ask if it belongs to a new language $\widetilde{L}$."

"If someone can answer this question, then I can answer the original question." Moreover, the answer to the new question will be the same as the answer to my question.

Standing on the shoulders of $\widetilde{L}$, L is not hard. Of course, the calculation of the new question must be negligible.

2(b) Illustrated

For a language/problem L, an input ω essentially poses a question: Does ω belong to L?

A reduction is an assignment/calculation that, instead of answering this question, computes a new question: "I will describe a new input $\widetilde{\omega}$, and I will ask if it belongs to a new language $\widetilde{L}$."

"If someone can answer this question, then I can answer the original question." Moreover, the answer to the new question will be the same as the answer to my question.

Standing on the shoulders of $\widetilde{L}$, L is not hard. Of course, the calculation of the new question must be negligible.

$\widetilde{L}$ must be complex enough to allow us to formulate any problem described by L as a problem of $\widetilde{L}$.

2(b) Formally: Reductions

2(b) Formally: Reductions

Karp Reductions of Languages

Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \preccurlyeq_{\mathcal{C}} \widehat{L}$, if there exists a computable Turing machine R such that

2(b) Formally: Reductions

Karp Reductions of Languages

Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \preccurlyeq_{\mathcal{C}} \widehat{L}$, if there exists a computable Turing machine R such that

(i) R is a $\mathcal{C}$-complexity machine/procedure,

2(b) Formally: Reductions

Karp Reductions of Languages

Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \preccurlyeq_{\mathcal{C}} \widehat{L}$, if there exists a computable Turing machine R such that

(i) R is a $\mathcal{C}$-complexity machine/procedure,

(ii) $\omega \in L$ if and only if $\widetilde{\omega} \in \widehat{L}$, where $\widetilde{\omega}$ is the sequence calculated by R from ω.

Remarks

Remarks

The introduced relation is read as follows: L is reducible to $\widehat{L}$ in $\mathcal{C}$.

Remarks

The introduced relation is read as follows: L is reducible to $\widehat{L}$ in $\mathcal{C}$. This means: The decision task of language $\widehat{L}$ is *at least as hard* as that of L *modulo* $\mathcal{C}$.

Remarks

The introduced relation is read as follows: L is reducible to $\widehat{L}$ in $\mathcal{C}$. This means: The decision task of language $\widehat{L}$ is *at least as hard* as that of L *modulo* $\mathcal{C}$.

Other reduction concepts exist. The above definition lies in the work of Karp and is generally referred to as Karp reductions.

Remarks

The introduced relation is read as follows: L is reducible to $\widehat{L}$ in $\mathcal{C}$. This means: The decision task of language $\widehat{L}$ is *at least as hard* as that of L *modulo* $\mathcal{C}$.

Other reduction concepts exist. The above definition lies in the work of Karp and is generally referred to as Karp reductions. If emphasis is needed, we use the notation $\preccurlyeq_{\mathcal{C}}^{\text{Karp}}$.

Remarks

The introduced relation is read as follows: L is reducible to $\widehat{L}$ in $\mathcal{C}$. This means: The decision task of language $\widehat{L}$ is *at least as hard* as that of L *modulo* $\mathcal{C}$.

Other reduction concepts exist. The above definition lies in the work of Karp and is generally referred to as Karp reductions. If emphasis is needed, we use the notation $\preccurlyeq_{\mathcal{C}}^{\text{Karp}}$. In this course, we mostly encounter such reductions, and most of the time, we omit the upper index.

The first examples

The first examples

$$\text{CLIQUE} = \{\lceil G, k \rceil : \ G \text{ contains a clique of size } k\}$$

The first examples

$$\text{CLIQUE} = \{\lceil G, k \rceil : \ G \text{ contains a clique of size } k\}$$

$$\text{INDEPENDENT-VERTEX-SET} = \{\lceil G, k \rceil : G \text{ contains an independent vertex set of size } k\}$$

The first examples

$$\text{CLIQUE} = \{\lceil G, k \rceil : \ G \text{ contains a clique of size } k\}$$

$$\text{INDEPENDENT-VERTEX-SET} = \{\lceil G, k \rceil : G \text{ contains an independent vertex set of size } k\}$$

$$\text{VERTEX-COVER} = \{\lceil G, k \rceil : G \text{ has a vertex cover of size } k\}$$

The first examples

$$\text{CLIQUE} = \{\lceil G, k \rceil : \ G \text{ contains a clique of size } k\}$$

$$\text{INDEPENDENT-VERTEX-SET} = \{\lceil G, k \rceil : G \text{ contains an independent vertex set of size } k\}$$

$$\text{VERTEX-COVER} = \{\lceil G, k \rceil : G \text{ has a vertex cover of size } k\}$$

Note

The tasks above are reducible to each other in both directions.

Example of a Reduction

Example of a Reduction

Example

Example CLIQUE $\preccurlyeq_{\mathcal{P}}$ INDEPENDENT-VERTEX-SET.

Example of a Reduction

Example

Example CLIQUE $\preccurlyeq_{\mathcal{P}}$ INDEPENDENT-VERTEX-SET.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the CLIQUE problem.

Example of a Reduction

Example

Example CLIQUE $\preccurlyeq_{\mathcal{P}}$ INDEPENDENT-VERTEX-SET.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the CLIQUE problem. We compute the complement of G from its code. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, k \rceil$.

Example of a Reduction

Example

Example CLIQUE $\preccurlyeq_{\mathcal{P}}$ INDEPENDENT-VERTEX-SET.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the CLIQUE problem. We compute the complement of G from its code. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

Example of a Reduction

Example

Example CLIQUE $\preccurlyeq_{\mathcal{P}}$ INDEPENDENT-VERTEX-SET.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the CLIQUE problem. We compute the complement of G from its code. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

The complexity of computing $\widetilde{\omega}$ is polynomial (actually solvable in logarithmic space).

Example of a Reduction

Example of a Reduction

Example

Example INDEPENDENT-VERTEX-SET $\preccurlyeq_{\mathcal{P}}$ VERTEX-COVER.

Example of a Reduction

Example

Example INDEPENDENT-VERTEX-SET $\preccurlyeq_{\mathcal{P}}$ VERTEX-COVER.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the INDEPENDENT-VERTEX-SET problem.

Example of a Reduction

Example

Example INDEPENDENT-VERTEX-SET $\preccurlyeq_{\mathcal{P}}$ VERTEX-COVER.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the INDEPENDENT-VERTEX-SET problem. We compute $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil G, |V(G)| - k \rceil$.

Example of a Reduction

Example

Example INDEPENDENT-VERTEX-SET $\preccurlyeq_{\mathcal{P}}$ VERTEX-COVER.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the INDEPENDENT-VERTEX-SET problem. We compute $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil G, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

Example of a Reduction

Example

Example INDEPENDENT-VERTEX-SET $\preccurlyeq_{\mathcal{P}}$ VERTEX-COVER.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the INDEPENDENT-VERTEX-SET problem. We compute $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil G, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

The complexity of computing $\widetilde{\omega}$ is polynomial (actually solvable in logarithmic space).

Example of a Reduction

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem.

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem. We compute the complement and $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, |V(G)| - k \rceil$.

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem. We compute the complement and $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem. We compute the complement and $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

The complexity of computing $\widetilde{\omega}$ is polynomial (actually solvable in logarithmic space).

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem. We compute the complement and $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

The complexity of computing $\widetilde{\omega}$ is polynomial (actually solvable in logarithmic space).

Note that for CLIQUE, VERTEX-COVER, and INDEPENDENT-VERTEX-SET, no efficient algorithm is known.

Example of a Reduction

Example

Example VERTEX-COVER $\preccurlyeq_{\mathcal{P}}$ CLIQUE.

The reduction is extremely simple: Given an input $\omega = \lceil G, k \rceil$ for the VERTEX-COVER problem. We compute the complement and $|V(G)| - k$ from the codes of G and k. The new sequence $\widetilde{\omega}$ is $\lceil \overline{G}, |V(G)| - k \rceil$. From our earlier graph theory studies, the new *question* is equivalent to the original one.

The complexity of computing $\widetilde{\omega}$ is polynomial (actually solvable in logarithmic space).

Note that for CLIQUE, VERTEX-COVER, and INDEPENDENT-VERTEX-SET, no efficient algorithm is known. If there were one for any of them, it would have significant implications for the others.

Break

Turing Reductions

Turing Reductions

Definition

Turing Reductions of Languages Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is Turing reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \leq_{\mathcal{C}} \widehat{L}$, if there exists a $\mathcal{C}$-complexity oracle Turing machine O that decides L using $\widehat{L}$ as an oracle, i.e., for any input $\omega \in \Sigma^*$:

Turing Reductions

Definition

Turing Reductions of Languages Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is Turing reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \leq_{\mathcal{C}} \widehat{L}$, if there exists a $\mathcal{C}$-complexity oracle Turing machine O that decides L using $\widehat{L}$ as an oracle, i.e., for any input $\omega \in \Sigma^*$:

(i) If $\omega \in L$, then $O^{\widehat{L}}(\omega) = 1$.

Turing Reductions

Definition

Turing Reductions of Languages Let $L, \widehat{L} \subset \Sigma^*$ be two languages and $\mathcal{C}$ a complexity class. L is Turing reducible to $\widehat{L}$ in $\mathcal{C}$, denoted: $L \leq_{\mathcal{C}} \widehat{L}$, if there exists a $\mathcal{C}$-complexity oracle Turing machine O that decides L using $\widehat{L}$ as an oracle, i.e., for any input $\omega \in \Sigma^*$:

(i) If $\omega \in L$, then $O^{\widehat{L}}(\omega) = 1$.

(ii) If $\omega \notin L$, then $O^{\widehat{L}}(\omega) = 0$.

Turing Reduction

Turing Reduction

Turing Reduction

Let $L, \widehat{L} \subseteq \Sigma^*$ be two languages and $\mathcal{C}$ be a complexity class.

Turing Reduction

Turing Reduction

Let $L, \widehat{L} \subseteq \Sigma^*$ be two languages and $\mathcal{C}$ be a complexity class.

$L \preccurlyeq_{\mathcal{C}}^{\text{Turing}} \widehat{L}$ if and only if there exists a Turing machine R such that

Turing Reduction

Turing Reduction

Let $L, \widehat{L} \subseteq \Sigma^*$ be two languages and $\mathcal{C}$ be a complexity class.

$L \preccurlyeq_{\mathcal{C}}^{\text{Turing}} \widehat{L}$ if and only if there exists a Turing machine R such that

(i) R decides L and R is an L_2-oracle machine.

Turing Reduction

Turing Reduction

Let $L, \widehat{L} \subseteq \Sigma^*$ be two languages and $\mathcal{C}$ be a complexity class.

$L \preccurlyeq_{\mathcal{C}}^{\text{Turing}} \widehat{L}$ if and only if there exists a Turing machine R such that

(i) R decides L and R is an L_2-oracle machine.

(ii) The complexity of R belongs to $\mathcal{C}$.

Turing Reduction

Turing Reduction

Let $L, \widehat{L} \subseteq \Sigma^*$ be two languages and $\mathcal{C}$ be a complexity class.

$L \preccurlyeq_{\mathcal{C}}^{\text{Turing}} \widehat{L}$ if and only if there exists a Turing machine R such that

(i) R decides L and R is an L_2-oracle machine.

(ii) The complexity of R belongs to $\mathcal{C}$.

The term in (i) involves an unknown concept that needs clarification.

Oracle Turing Machine

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

R is an O-oracle Turing machine if

- it has an extra question/oracle tape.

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

R is an O-oracle Turing machine if

- it has an extra question/oracle tape.

  Only writing is allowed on this tape (there is no head over the tape; the hand can only write moving to the right).

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

R is an O-oracle Turing machine if

- it has an extra question/oracle tape.

  Only writing is allowed on this tape (there is no head over the tape; the hand can only write moving to the right). The written characters are elements of $\Sigma \cup \{?\}$, i.e., elements of the alphabet of language O and a special '?' symbol.

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

R is an O-oracle Turing machine if

- it has an extra question/oracle tape.

  Only writing is allowed on this tape (there is no head over the tape; the hand can only write moving to the right). The written characters are elements of $\Sigma \cup \{?\}$, i.e., elements of the alphabet of language O and a special '?' symbol. Writing '?' on the question tape indicates the condition of a question.

Oracle Turing Machine

Definition: Oracle Turing Machine

Let $O \subset \Sigma^*$ be a language.

R is an O-oracle Turing machine if

- it has an extra question/oracle tape.

  Only writing is allowed on this tape (there is no head over the tape; the hand can only write moving to the right). The written characters are elements of $\Sigma \cup \{?\}$, i.e., elements of the alphabet of language O and a special '?' symbol. Writing '?' on the question tape indicates the condition of a question. It queries about the character sequence in Σ^* between the previous '?' (or the tape-start symbol) and it with respect to the oracle O.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

The transition function acts on the state of the current configuration; it only affects the second component.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

The transition function acts on the state of the current configuration; it only affects the second component. The first component changes only if the algorithm poses a question to the oracle.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

The transition function acts on the state of the current configuration; it only affects the second component. The first component changes only if the algorithm poses a question to the oracle. The change depends naturally on the relationship of the question character sequence to O.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

The transition function acts on the state of the current configuration; it only affects the second component. The first component changes only if the algorithm poses a question to the oracle. The change depends naturally on the relationship of the question character sequence to O.

From these, the run (a sequence of configurations generated from the initial configuration), and the defined language, are naturally derived.

Oracle Turing Machine (Continued)

Definition: Oracle Turing Machine (Continued)

- The set of states is of the form

$$\{\text{ORACLE-YES}, \text{ORACLE-NO}\} \times S_0$$

where S_0 is the set of states in the original Turing machine.

The transition function acts on the state of the current configuration; it only affects the second component. The first component changes only if the algorithm poses a question to the oracle. The change depends naturally on the relationship of the question character sequence to O.

From these, the run (a sequence of configurations generated from the initial configuration), and the defined language, are naturally derived. The cost of the question is 1 time unit and 0 space.

Comparison of the Two Reductions

Comparison of the Two Reductions

The Karp reduction is a very specific case of Turing reduction: After the usual computation, a *single* question can be posed about the belonging of $\widehat{L}$.

Comparison of the Two Reductions

The Karp reduction is a very specific case of Turing reduction: After the usual computation, a *single* question can be posed about the belonging of $\widehat{L}$.

The answer to the question also represents the computed bit.

Comparison of the Two Reductions

The Karp reduction is a very specific case of Turing reduction: After the usual computation, a *single* question can be posed about the belonging of $\widehat{L}$.

The answer to the question also represents the computed bit.

Turing reduction is, of course, a much stronger concept. We can think of $\widehat{L}$ as an unwritten subroutine.

Comparison of the Two Reductions

The Karp reduction is a very specific case of Turing reduction: After the usual computation, a *single* question can be posed about the belonging of $\widehat{L}$.

The answer to the question also represents the computed bit.

Turing reduction is, of course, a much stronger concept. We can think of $\widehat{L}$ as an unwritten subroutine.

The essence of the reduction is that if someone can efficiently write the $\widehat{L}$ subroutine, then L can be efficiently decided (assuming the contribution of R is considered efficient, i.e., it belongs to $\mathcal{C}$).

Comparison of the Two Reductions

The Karp reduction is a very specific case of Turing reduction: After the usual computation, a *single* question can be posed about the belonging of $\widehat{L}$.

The answer to the question also represents the computed bit.

Turing reduction is, of course, a much stronger concept. We can think of $\widehat{L}$ as an unwritten subroutine.

The essence of the reduction is that if someone can efficiently write the $\widehat{L}$ subroutine, then L can be efficiently decided (assuming the contribution of R is considered efficient, i.e., it belongs to $\mathcal{C}$).

We do not require the implementation of the subroutine, we only count the *invocation* of the subroutine and the reception of the result as a single step.

Transitivity

Transitivity

Finally, we mention an important property of some reductions.

Transitivity

Finally, we mention an important property of some reductions.

Transitivity of reductions

Transitivity

Finally, we mention an important property of some reductions.

Transitivity of reductions

(i) $\preccurlyeq_{\mathcal{P}}$ is transitive.

Transitivity

Finally, we mention an important property of some reductions.

Transitivity of reductions

(i) $\preceq_{\mathcal{P}}$ is transitive.

(ii) $\preceq_{\mathcal{L}}$ is transitive.

Transitivity

Finally, we mention an important property of some reductions.

Transitivity of reductions

(i) $\preccurlyeq_{\mathcal{P}}$ is transitive.

(ii) $\preccurlyeq_{\mathcal{L}}$ is transitive.

(iii) Let $s(n) \geq \log n$ be a nice space function. If $L_1 \preccurlyeq_{\mathcal{SPACE}(\mathcal{O}(s(n))} L_2$ and $L_2 \preccurlyeq_{\mathcal{L}} L_3$, then $L_1 \preccurlyeq_{\mathcal{SPACE}(\mathcal{O}(s(n))} L_3$

Transitivity of Polynomial Time Reductions

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial. Let p_1 and p_2 be the two polynomials giving the time bounds of R_1 and R_2. We can assume that p_2 is monotonically increasing.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial. Let p_1 and p_2 be the two polynomials giving the time bounds of R_1 and R_2. We can assume that p_2 is monotonically increasing.

Run R_1 on input ω, which calculates the character sequence $\widetilde{\omega}$.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial. Let p_1 and p_2 be the two polynomials giving the time bounds of R_1 and R_2. We can assume that p_2 is monotonically increasing.

Run R_1 on input ω, which calculates the character sequence $\widetilde{\omega}$. Then run R_2 on $\widetilde{\omega}$, leading to the computation of $\widetilde{\widetilde{\omega}}$.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial. Let p_1 and p_2 be the two polynomials giving the time bounds of R_1 and R_2. We can assume that p_2 is monotonically increasing.

Run R_1 on input ω, which calculates the character sequence $\widetilde{\omega}$. Then run R_2 on $\widetilde{\omega}$, leading to the computation of $\widetilde{\widetilde{\omega}}$.

The resulting Turing machine is denoted as R.

Transitivity of Polynomial Time Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{P}} L_2$ and $L_2 \preccurlyeq_{\mathcal{P}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Specifically, R_1 and R_2 are both polynomial. Let p_1 and p_2 be the two polynomials giving the time bounds of R_1 and R_2. We can assume that p_2 is monotonically increasing.

Run R_1 on input ω, which calculates the character sequence $\widetilde{\omega}$. Then run R_2 on $\widetilde{\omega}$, leading to the computation of $\widetilde{\widetilde{\omega}}$.

The resulting Turing machine is denoted as R. We will show that R verifies the $L_1 \preccurlyeq_{\mathcal{P}} L_3$ reduction.

Transitivity of Polynomial Time Reductions (Continued)

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$.

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

We still need to show that R is polynomial.

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

We still need to show that R is polynomial. The time complexity of R on input ω is $p_1(|\omega|) + p_2(|\widetilde{\omega}|)$.

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

We still need to show that R is polynomial. The time complexity of R on input ω is $p_1(|\omega|) + p_2(|\widetilde{\omega}|)$. $\widetilde{\omega}$ is computed by a machine with time bound p_1, so $|\widetilde{\omega}| \leq p_1(\omega)$.

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

We still need to show that R is polynomial. The time complexity of R on input ω is $p_1(|\omega|) + p_2(|\widetilde{\omega}|)$. $\widetilde{\omega}$ is computed by a machine with time bound p_1, so $|\widetilde{\omega}| \leq p_1(\omega)$.

Thus, for input ω, the upper bound on the runtime becomes

$$p_1(|\omega|) + p_2(|\widetilde{\omega}|) \leq p_1(|\omega|) + p_2(p_1(|\omega|)).$$

Transitivity of Polynomial Time Reductions (Continued)

$\omega \in L_1$ if and only if $\widetilde{\omega} \in L_2$. Which holds if and only if $\widetilde{\widetilde{\omega}} \in L_3$.

We still need to show that R is polynomial. The time complexity of R on input ω is $p_1(|\omega|) + p_2(|\widetilde{\omega}|)$. $\widetilde{\omega}$ is computed by a machine with time bound p_1, so $|\widetilde{\omega}| \leq p_1(\omega)$.

Thus, for input ω, the upper bound on the runtime becomes

$$p_1(|\omega|) + p_2(|\widetilde{\omega}|) \leq p_1(|\omega|) + p_2(p_1(|\omega|)).$$

This is a polynomial upper bound.

Transitivity of Logarithmic Space Reductions

Transitivity of Logarithmic Space Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{L}} L_2$ and $L_2 \preccurlyeq_{\mathcal{L}} L_3$.

Transitivity of Logarithmic Space Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{L}} L_2$ and $L_2 \preccurlyeq_{\mathcal{L}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions.

Transitivity of Logarithmic Space Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{L}} L_2$ and $L_2 \preccurlyeq_{\mathcal{L}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions. Specifically, R_1 and R_2 are both logarithmic.

Transitivity of Logarithmic Space Reductions

Assume that $L_1 \preccurlyeq_{\mathcal{L}} L_2$ and $L_2 \preccurlyeq_{\mathcal{L}} L_3$. Let R_1 and R_2 be the two algorithms verifying the two reductions. Specifically, R_1 and R_2 are both logarithmic. We construct an algorithm R from the two

reductions as follows: Run R_1 on input ω, which calculates the character sequence $\widetilde{\omega}$. Then run R_2 on $\widetilde{\omega}$, leading to the computation of $\widetilde{\widetilde{\omega}}$.

Transitivity of Logarithmic Space Reductions (Continued)

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good:

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation.

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation. This is expected to exceed logarithmic space.

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation. This is expected to exceed logarithmic space. Nevertheless, keep in mind this R algorithm's execution.

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation. This is expected to exceed logarithmic space. Nevertheless, keep in mind this R algorithm's execution. In the actual $\widetilde{R}$ reduction, we recognize fragments of R's execution.

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation. This is expected to exceed logarithmic space. Nevertheless, keep in mind this R algorithm's execution. In the actual $\widetilde{R}$ reduction, we recognize fragments of R's execution.

The worktapes of $\widetilde{R}$ correspond to R_1's worktapes plus R_2's worktapes.

Transitivity of Logarithmic Space Reductions (Continued)

The algorithm obtained this way is NOT good: The intermediate $\widetilde{\omega}$ requires a worktape during computation. This is expected to exceed logarithmic space. Nevertheless, keep in mind this R algorithm's execution. In the actual $\widetilde{R}$ reduction, we recognize fragments of R's execution.

The worktapes of $\widetilde{R}$ correspond to R_1's worktapes plus R_2's worktapes. We have two extra tapes instead of the previous tape, one for the output of R_1 and shared with R_2's input tape, and the other tape's content is the position index of R_1's output tape while the other tape's content is the position index of R_2's input tape.

Transitivity of Logarithmic Space Reductions (Continued)

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head. If R_1 writes, then we compare the new position with the desired position for reading.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head. If R_1 writes, then we compare the new position with the desired position for reading.

If the two positions match, we read the unwritten character from the state and stop the R_1 simulation,

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head. If R_1 writes, then we compare the new position with the desired position for reading.

If the two positions match, we read the unwritten character from the state and stop the R_1 simulation, continue the R_2 simulation.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head. If R_1 writes, then we compare the new position with the desired position for reading.

If the two positions match, we read the unwritten character from the state and stop the R_1 simulation, continue the R_2 simulation. If the two positions differ, we continue the R_1 simulation.

Transitivity of Logarithmic Space Reductions (Continued)

$\widetilde{R}$ performs the simulation of R_2 without the content of the $\widetilde{\omega}$ input tape.

We need to work for every read operation.

Now we know which character of the calculated $\widetilde{\omega}$ we are interested in. We start the simulation of R_1. During the simulation, we do not write down the calculated characters, only store the position of the output head. If R_1 writes, then we compare the new position with the desired position for reading.

If the two positions match, we read the unwritten character from the state and stop the R_1 simulation, continue the R_2 simulation. If the two positions differ, we continue the R_1 simulation.

(iii) The ideas of the previous proof still work here.

Lemma

Lemma

Lemma

Lemma

Lemma

(i) $L \preccurlyeq_{\mathcal{P}} \widehat{L}$ and $\widehat{L} \in \mathcal{P}$, then $L \in \mathcal{P}$.

Lemma

Lemma

(i) $L \preccurlyeq_{\mathcal{P}} \widehat{L}$ and $\widehat{L} \in \mathcal{P}$, then $L \in \mathcal{P}$.

(ii) $L \preccurlyeq_{\mathcal{L}} \widehat{L}$ and $\widehat{L} \in \mathcal{L}$, then $L \in \mathcal{L}$.

Proof of the Lemma

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Let ω be the given input.

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Let ω be the given input.

Perform the computation

$$\omega \to A(\omega) \in \Sigma^{p(n)} \to \widehat{A}(A(\omega)) \in \{\text{ACCEPT}, \text{REJECT}\}$$

for the following values of n.

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Let ω be the given input.

Perform the computation

$$\omega \to A(\omega) \in \Sigma^{p(n)} \to \widehat{A}(A(\omega)) \in \{\text{ACCEPT}, \text{REJECT}\}$$

for the following values of n.

The first step is limited by a polynomial p in the input size n. The longest input that we can compute falls into $\Sigma^{p(n)}$.

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Let ω be the given input.

Perform the computation

$$\omega \to A(\omega) \in \Sigma^{p(n)} \to \widehat{A}(A(\omega)) \in \{\text{ACCEPT}, \text{REJECT}\}$$

for the following values of n.

The first step is limited by a polynomial p in the input size n. The longest input that we can compute falls into $\Sigma^{p(n)}$. The second step is limited by a polynomial q in the input size. The total time is $(p + q \circ p)(n)$, which is a polynomial function of n.

Proof of the Lemma

(i) Consider a Turing machine A that performs the reduction from L to $\widehat{L}$, and $\widehat{A}$, which decides the membership problem for $\widehat{L}$ in $\mathcal{P}$.

Let ω be the given input.

Perform the computation

$$\omega \to A(\omega) \in \Sigma^{p(n)} \to \widehat{A}(A(\omega)) \in \{\text{ACCEPT}, \text{REJECT}\}$$

for the following values of n.

The first step is limited by a polynomial p in the input size n. The longest input that we can compute falls into $\Sigma^{p(n)}$. The second step is limited by a polynomial q in the input size. The total time is $(p + q \circ p)(n)$, which is a polynomial function of n.

The combined action of the two algorithms, based on the concept of Karp reduction, decides exactly the language L, which means L is also decidable in polynomial time.

Proof of the Lemma

Proof of the Lemma

(ii) Based on the ideas of part (i) and the previous lemma, it is obvious.

Break

Example of a Reduction

Example of a Reduction

Example

Let L be an arbitrary language in $\mathcal{NL}$. Then

$$L \preccurlyeq_{\mathcal{L}} \text{DIRECTED-REACHABILITY}.$$

Example of a Reduction

Example

Let L be an arbitrary language in $\mathcal{NL}$. Then

$$L \preccurlyeq_{\mathcal{L}} \text{DIRECTED-REACHABILITY}.$$

The proof of this example is essentially what has been discussed earlier.

Example of a Reduction

Example

Let L be an arbitrary language in $\mathcal{NL}$. Then

$$L \preccurlyeq_{\mathcal{L}} \text{DIRECTED-REACHABILITY}.$$

The proof of this example is essentially what has been discussed earlier. We summarize the essential ideas of the previous reasoning in a theorem.

The Theorem

The Theorem

Theorem

Let $L \in_T \mathcal{NL}$. It can be assumed that the Turing machine proving containment has two different halting configurations on inputs of given length (so accepting runs end in the same configuration). To $\omega \in \Sigma^*$, we assign the graph of the (directed) reduced configurations $\overrightarrow{G} = \overrightarrow{G}_{T,\omega}$ of the Turing machine. This assignment includes v_0 as the vertex corresponding to the initial configuration, and v_+ as the vertex corresponding to the accepting configuration. This assignment has the following properties:

(i) $\omega \in L$ if and only if $\lceil \overrightarrow{G}_{\omega,T}, v_0, v_+ \rceil \in \overrightarrow{st}$-REACHABILITY.

(ii) The assignment is computable and its space complexity is $\mathcal{O}(\log(n))$.

Consequences

Consequences

This example is much more general than the previous one. To see this, let's look at some consequences.

Consequences

This example is much more general than the previous one. To see this, let's look at some consequences.

Corollary

If $\overrightarrow{st}$-REACHABILITY $\in \mathcal{P}$, then $\mathcal{NL} \subset \mathcal{P}$.

Consequences

This example is much more general than the previous one. To see this, let's look at some consequences.

Corollary

If $\overrightarrow{st}$-REACHABILITY $\in \mathcal{P}$, then $\mathcal{NL} \subset \mathcal{P}$.

The condition is true, this simply follows from descriptions and analysis of graph traversal algorithms, well-known in algorithm theory lectures.

Consequences

This example is much more general than the previous one. To see this, let's look at some consequences.

Corollary

If $\overrightarrow{st}$-REACHABILITY $\in \mathcal{P}$, then $\mathcal{NL} \subset \mathcal{P}$.

The condition is true, this simply follows from descriptions and analysis of graph traversal algorithms, well-known in algorithm theory lectures.

The above corollary is essentially re-proving $\mathcal{NL} \subset \mathcal{P}$, our earlier result.

Consequence

Consequence

Corollary

If DIRECTED-REACHABILITY $\in \mathcal{L}$, then $\mathcal{NL} = \mathcal{L}$.

Consequence

Corollary

If DIRECTED-REACHABILITY $\in \mathcal{L}$, then $\mathcal{NL} = \mathcal{L}$.

The conclusion is actually *only* $\mathcal{NL} \subset \mathcal{L}$ (the other direction of containment is obvious).

Consequence

Corollary

If DIRECTED-REACHABILITY $\in \mathcal{L}$, then $\mathcal{NL} = \mathcal{L}$.

The conclusion is actually *only* $\mathcal{NL} \subset \mathcal{L}$ (the other direction of containment is obvious).

Here, the truth of the condition is not known, and many believe it is not true.

Definition of Completeness

Definition of Completeness

The above reasoning is very important. The essence of the argument is that, based on the example, DIRECTED-REACHABILITY encapsulates the complexity of the entire $\mathcal{NL}$ language class. This leads to the creation of a more general concept:

Definition of Completeness

The above reasoning is very important. The essence of the argument is that, based on the example, DIRECTED-REACHABILITY encapsulates the complexity of the entire $\mathcal{NL}$ language class. This leads to the creation of a more general concept:

Definition

A language $\widehat{L}$ is *complete* in class $\mathcal{C}$ under complexity $\mathcal{R}$ reduction if:

(i) $\widehat{L} \in \mathcal{C}$,

(ii) for every $L \in \mathcal{C}$, $L \preccurlyeq_{\mathcal{R}} \widehat{L}$.

Special Cases

Special Cases

We highlight four special cases.

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

(i) $\widehat{L} \in \mathcal{NP}$,

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

(i) $\widehat{L} \in \mathcal{NP}$,

(ii) for every $L \in \mathcal{NP}$, $L \preccurlyeq_{\mathcal{P}} \widehat{L}$.

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

(i) $\widehat{L} \in \mathcal{NP}$,

(ii) for every $L \in \mathcal{NP}$, $L \preccurlyeq_{\mathcal{P}} \widehat{L}$.

The above convention is an alternative to working with $\preccurlyeq_{\mathcal{L}}$ reduction.

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

(i) $\widehat{L} \in \mathcal{NP}$,

(ii) for every $L \in \mathcal{NP}$, $L \preccurlyeq_{\mathcal{P}} \widehat{L}$.

The above convention is an alternative to working with $\preccurlyeq_{\mathcal{L}}$ reduction. This stricter interpretation will still be true for most later $\mathcal{NP}$-completeness-proving reductions.

Special Cases

We highlight four special cases.

Definition

The language $\widehat{L}$ is $\mathcal{NP}$-complete if it is complete in $\mathcal{NP}$ under $\mathcal{P}$ reduction. That is, $\widehat{L}$ is $\mathcal{NP}$-*complete* if

(i) $\widehat{L} \in \mathcal{NP}$,

(ii) for every $L \in \mathcal{NP}$, $L \preccurlyeq_{\mathcal{P}} \widehat{L}$.

The above convention is an alternative to working with $\preccurlyeq_{\mathcal{L}}$ reduction. This stricter interpretation will still be true for most later $\mathcal{NP}$-completeness-proving reductions. However, we only require the polynomial time complexity of the reductions, as we demand easier verifiability.

Special Cases

Special Cases

Definition

The language L is $\mathcal{NL}$-complete if it is complete in $\mathcal{NL}$ under $\mathcal{L}$ reduction.

Special Cases

Definition

The language L is $\mathcal{NL}$-complete if it is complete in $\mathcal{NL}$ under $\mathcal{L}$ reduction.

Definition

The language L is $\mathcal{P}$-complete if it is complete in $\mathcal{P}$ under $\mathcal{L}$ reduction.

Special Cases

Definition

The language L is $\mathcal{NL}$-complete if it is complete in $\mathcal{NL}$ under $\mathcal{L}$ reduction.

Definition

The language L is $\mathcal{P}$-complete if it is complete in $\mathcal{P}$ under $\mathcal{L}$ reduction.

Definition

The language L is $\mathcal{PSPACE}$-complete if it is complete in $\mathcal{P}SPACE$ under $\mathcal{P}$ reduction.

Hardness

Hardness

Definition

$\widehat{L}$ is *hard* for class $\mathcal{C}$ under $\mathcal{R}$ complexity reduction, if for every $L \in \mathcal{C}$, $L \preccurlyeq_{\mathcal{R}} \widehat{L}$.

Hardness

Definition

$\widehat{L}$ is *hard* for class $\mathcal{C}$ under $\mathcal{R}$ complexity reduction, if for every $L \in \mathcal{C}$, $L \preccurlyeq_{\mathcal{R}} \widehat{L}$.

In other words, hardness is the concept of completeness without condition (i). That is, we do not require membership in the class, only the reducibility of elements of the class.

Hardness

Definition

$\widehat{L}$ is *hard* for class $\mathcal{C}$ under $\mathcal{R}$ complexity reduction, if for every $L \in \mathcal{C}$, $L \preccurlyeq_{\mathcal{R}} \widehat{L}$.

In other words, hardness is the concept of completeness without condition (i). That is, we do not require membership in the class, only the reducibility of elements of the class.

A Concrete $\mathcal{NL}$-Complete Problem

A Concrete $\mathcal{NL}$-Complete Problem

Theorem

DIRECTED-REACHABILITY is $\mathcal{NL}$-complete.

A Concrete $\mathcal{NL}$-Complete Problem

Theorem

DIRECTED-REACHABILITY is $\mathcal{NL}$-complete.

When examining reductions, we saw the $\mathcal{NL}$-hardness. We previously saw the membership in $\mathcal{NL}$.

A Concrete $\mathcal{NL}$-Complete Problem

Theorem

DIRECTED-REACHABILITY is $\mathcal{NL}$-complete.

When examining reductions, we saw the $\mathcal{NL}$-hardness. We previously saw the membership in $\mathcal{NL}$.

Based on the theorem, our knowledge about DIRECTED-REACHABILITY extends to the entire $\mathcal{NL}$ class.

A Concrete $\mathcal{NL}$-Complete Problem

Theorem

DIRECTED-REACHABILITY is $\mathcal{NL}$-complete.

When examining reductions, we saw the $\mathcal{NL}$-hardness. We previously saw the membership in $\mathcal{NL}$.

Based on the theorem, our knowledge about DIRECTED-REACHABILITY extends to the entire $\mathcal{NL}$ class.

We already saw an example of this: $\overrightarrow{st}$-REACHABILITY$\in \mathcal{P}$. This led to $\mathcal{NL} \subset \mathcal{P}$.

Implication of Savitch's Algorithm

Implication of Savitch's Algorithm

Savitch's algorithm saves space. The DIRECTED-REACHABILITY is solved in $\log^2$ space. This immediately leads to the following theorem:

Implication of Savitch's Algorithm

Savitch's algorithm saves space. The DIRECTED-REACHABILITY is solved in $\log^2$ space. This immediately leads to the following theorem:

Theorem

$$\mathcal{NL} \subset \mathcal{SPACE}(\log^2 n).$$

Theorem

Theorem

For a nice function $s(n) > \log n$, any problem in $\mathcal{NSPACE}(s(n))$ can be reduced to a $\overrightarrow{st}$-REACHABILITY problem in $\mathcal{O}(s(n))$ space, whose vertex set has size $2^{\mathcal{O}(s(n))}$.

Theorem

For a nice function $s(n) > \log n$, any problem in $\mathcal{NSPACE}(s(n))$ can be reduced to a $\overrightarrow{st}$-REACHABILITY problem in $\mathcal{O}(s(n))$ space, whose vertex set has size $2^{\mathcal{O}(s(n))}$.

This reachability problem can be solved in $\mathcal{O}(s^2(n))$ space (Savitch's algorithm). This resulted in the following theorem:

Theorem

For a nice function $s(n) > \log n$, any problem in $\mathcal{NSPACE}(s(n))$ can be reduced to a $\overrightarrow{st}$-REACHABILITY problem in $\mathcal{O}(s(n))$ space, whose vertex set has size $2^{\mathcal{O}(s(n))}$.

This reachability problem can be solved in $\mathcal{O}(s^2(n))$ space (Savitch's algorithm). This resulted in the following theorem:

Theorem

Let $s(n) > \log n$ be a nice space function. Then

$$\mathcal{NSPACE}(s(n)) \subset \mathcal{SPACE}(\mathcal{O}(s^2(n))).$$

Implications

Implications

Specifically, we get the inclusion $\mathcal{NPSPACE} \subset \mathcal{PSPACE}$. That is,

Implications

Specifically, we get the inclusion $\mathcal{NPSPACE} \subset \mathcal{PSPACE}$. That is,

Theorem

$\mathcal{PSPACE} = \mathcal{NPSPACE}$.

Implications

Specifically, we get the inclusion $\mathcal{NPSPACE} \subset \mathcal{PSPACE}$. That is,

Theorem

$\mathcal{PSPACE} = \mathcal{NPSPACE}$.

We know that deterministic classes are closed under complementation. Specifically, we obtain the following:

Implications

Specifically, we get the inclusion $\mathcal{NPSPACE} \subset \mathcal{PSPACE}$. That is,

Theorem

$\mathcal{PSPACE} = \mathcal{NPSPACE}$.

We know that deterministic classes are closed under complementation. Specifically, we obtain the following:

Theorem

$\mathcal{NPSPACE}$ is closed under complementation.

Abstraction

Abstraction

Corollary

Let S be a class of nice space functions that is closed under squaring. Then $\mathcal{NPSPACE}(\cup_{s \in S} s(n)) = \mathcal{PSPACE}(\cup_{s \in S} s(n))$.

Abstraction

Corollary

Let S be a class of nice space functions that is closed under squaring. Then $\mathcal{NPSPACE}(\cup_{s\in S} s(n)) = \mathcal{PSPACE}(\cup_{s\in S} s(n))$.

Specifically, $\mathcal{NPSPACE}(\cup_{s\in S} s(n))$ is closed under complementation, i.e.,

$$\mathcal{NPSPACE}(\cup_{s\in S} s(n)) = co\mathcal{NPSPACE}(\cup_{s\in S} s(n)).$$

Abstraction

Corollary

Let S be a class of nice space functions that is closed under squaring. Then $\mathcal{NPSPACE}(\cup_{s\in S} s(n)) = \mathcal{PSPACE}(\cup_{s\in S} s(n))$.

Specifically, $\mathcal{NPSPACE}(\cup_{s\in S} s(n))$ is closed under complementation, i.e.,

$$\mathcal{NPSPACE}(\cup_{s\in S} s(n)) = co\mathcal{NPSPACE}(\cup_{s\in S} s(n)).$$

Thus, another special case: $\mathcal{EXPSPACE} = \mathcal{NEXPSPACE}$.

This is the End!

Thank you for your attention!