Complexity and Cryptography

Thomas Zeugmann

Hokkaido University
Laboratory for Algorithmics

https://www-alg.ist.hokudai.ac.jp/~thomas/COCRB/

Lecture 9: Important Complexity Classes

Fundamental Inclusions I

First we show that a logarithmic space bound can always be combined with a polynomial time bound.

Fundamental Inclusions I

First we show that a logarithmic space bound can always be combined with a polynomial time bound.

Theorem 1

(1) $\mathcal{L} \subseteq DTISP(n^{O(1)}, \log n)$,

(2) $\mathcal{NL} \subseteq NDTISP(n^{O(1)}, \log n)$,

Fundamental Inclusions I

First we show that a logarithmic space bound can always be combined with a polynomial time bound.

Theorem 1

(1) $\mathcal{L} \subseteq DTISP(n^{O(1)}, \log n)$,

(2) $\mathcal{NL} \subseteq NDTISP(n^{O(1)}, \log n)$,

Proof. Let M be a k-tape TM such that $S_M(n) = O(\log n)$. That is, there exists a constant $c > 0$ such that $S_M(n) \leqslant c \cdot \log n$. Recalling that a macro state consists of the head position on the input tape, the actual state of M, and for every work tape the content of all cells visited as well as the actual head position, we can bound the total number of *macro states* as follows:

Fundamental Inclusions II

$$n \cdot |Z| \left(|B|^{c \cdot \log n} c \cdot \log n \right)^{k-1} = O(n^{O(1)}) . \qquad (1)$$

Here n is the number of possibilities for the head position on the input tape. Moreover, the machine M can be in at most $|Z|$ many different states. On each work tape, the head can have visited at most $c \cdot \log n$ many positions, and thus it can write only strings of the same length on each of its work tapes. Since we have $|B|$ many symbols, and $k - 1$ many work tapes, the formula displayed above follows.

It remains to show the estimate $O(n^{O(1)})$.

Fundamental Inclusions III

Recall that

$$\begin{aligned}\log_{|B|} n &= \frac{\ln n}{\ln |B|} \text{ and } \log n = \frac{\ln n}{\ln 2} \text{, and hence} \\ c \cdot \log n &= c \cdot \frac{\ln n}{\ln 2} = c \cdot \frac{\ln |B|}{\ln 2} \cdot \log_{|B|} n = \hat{c} \cdot \log_{|B|} n\end{aligned}$$

Consequently, $|B|^{c \cdot \log n} = |B|^{\hat{c} \cdot \log_{|B|} n} = n^{\hat{c}}$. Furthermore, $\log n \leqslant n$ for $n \geqslant 1$, and therefore

$$\left(|B|^{c \cdot \log n} c \cdot \log n\right)^{k-1} \leqslant \left(c \cdot n^{\hat{c}+1}\right)^{k-1} = c^{k-1} n^{\tilde{c}} .$$

for $\tilde{c} = (\hat{c}+1)(k-1)$. Additionally, for $n \geqslant 2$ there exists an m such that $n^m \geqslant |Z|$. Thus, the Estimate (1) is proved and the theorem follows for the deterministic case.

Fundamental Inclusions IV

For the nondeterministic case, we have additionally to take into consideration that every polynomial is T-constructible. Hence, the NTM M can be combined with a clock for the particular polynomial time arising without changing the language accepted. We leave it as an exercise to show that the amount of space needed to implement the clock can be logarithmically bounded. Thus, (2) follows. ∎

Fundamental Inclusions IV

For the nondeterministic case, we have additionally to take into consideration that every polynomial is T-constructible. Hence, the NTM M can be combined with a clock for the particular polynomial time arising without changing the language accepted. We leave it as an exercise to show that the amount of space needed to implement the clock can be logarithmically bounded. Thus, (2) follows. ∎

The polynomial time bound just proved is essential to show the following inclusion:

Theorem 2

$\mathcal{NL} \subseteq \mathcal{P}$.

Fundamental Inclusions V

Proof. Let M be an NTM such that $S_M(n) \leqslant c \cdot \log n$ for a suitably chosen constant $c > 0$. We have to construct a deterministic TM $\tilde{M}$ that accepts the same language as M and that uses at most polynomial time, i.e., $T_{\tilde{M}}(n) \leqslant n^{O(1)}$.

The machine $\tilde{M}$ works as follows:

Fundamental Inclusions VI

(1) $\tilde{M}$ uses the same input w as M does. First, it writes all possible macro states of M on its first work tape. By Theorem 1 we already know that there are only polynomially many macro states.

Fundamental Inclusions VI

(1) $\tilde{M}$ uses the same input w as M does. First, it writes all possible macro states of M on its first work tape. By Theorem 1 we already know that there are only polynomially many macro states.

(2) Next, $\tilde{M}$ marks the one macro state of all the macro states written on its first work tape in which M starts its computation.

Fundamental Inclusions VI

(1) $\tilde{M}$ uses the same input w as M does. First, it writes all possible macro states of M on its first work tape. By Theorem 1 we already know that there are only polynomially many macro states.

(2) Next, $\tilde{M}$ marks the one macro state of all the macro states written on its first work tape in which M starts its computation.

(3) Then, $\tilde{M}$ marks all macro states that can be reached in one step by M from one of those already marked. If this increases the number of marked macro states, $\tilde{M}$ repeats Stage (3).
Otherwise, $\tilde{M}$ checks whether or not there is marked macro state in which M accepts the current input. If this is the case, $\tilde{M}$ accepts the current input. Otherwise, the input is rejected.

Fundamental Inclusions VII

By construction, we directly obtain $L(M) = L(\tilde{M})$. Finally, there are only polynomially many macro states to be read one time in each execution of Stage (3). Hence, $\tilde{M}$ executes at most $n^{O(1)}$ many steps. $\blacksquare$

Fundamental Inclusions VII

By construction, we directly obtain $L(M) = L(\tilde{M})$. Finally, there are only polynomially many macro states to be read one time in each execution of Stage (3). Hence, $\tilde{M}$ executes at most $n^{O(1)}$ many steps. $\blacksquare$

Note that the deterministic TM provided in the proof above also uses $n^{O(1)}$ many tape cells on its work tape.

Fundamental Inclusions VII

By construction, we directly obtain $L(M) = L(\tilde{M})$. Finally, there are only polynomially many macro states to be read one time in each execution of Stage (3). Hence, $\tilde{M}$ executes at most $n^{O(1)}$ many steps. $\blacksquare$

Note that the deterministic TM provided in the proof above also uses $n^{O(1)}$ many tape cells on its work tape.

Next, we aim to show that $\mathcal{NPSPACE} \subseteq \mathcal{PSPACE}$. This will be done by proving the following more general theorem:

Fundamental Inclusions VIII

Theorem 3 (Savitch (1970))

Let $f(n)$ *be an S-constructible function satisfying* $f(n) \geqslant \log n$. *Then, we have* $NSPACE(f(n)) \subseteq SPACE\left((f(n))^2\right)$.

We prove Savitch's theorem in the book in a more general context.

Fundamental Inclusions VIII

Theorem 3 (Savitch (1970))

Let $f(n)$ be an S-constructible function satisfying $f(n) \geqslant \log n$. Then, we have $NSPACE(f(n)) \subseteq SPACE\left((f(n))^2\right)$.

We prove Savitch's theorem in the book in a more general context.

So far, we have obtained the following insight:

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{NPSPACE} = \mathcal{PSPACE} .$$

Moreover, by Theorem 8 in Lecture 8 we also know that at least one of the inclusions must be *proper*. It is conjectured that *all* inclusions are proper. However, despite many efforts, so far none of the inclusions could be proved to be proper nor could any equality be shown.

Motivation

Question

Do there exist problems which can be considered to be the most difficult ones within their corresponding complexity class?

Motivation

Question

Do there exist problems which can be considered to be the most difficult ones within their corresponding complexity class?

We shall prove the *affirmative answer*, and refer to these problems as *complete problems*.

Motivation

Question

Do there exist problems which can be considered to be the most difficult ones within their corresponding complexity class?

We shall prove the *affirmative answer*, and refer to these problems as *complete problems*.

The importance of complete problems is easily explained. The efficient solution of a *complete* problem for $\mathcal{NL}$ or $\mathcal{NP}$ could be used to efficiently solve *all* of the problems in $\mathcal{NL}$ or $\mathcal{NP}$, respectively.

Motivation

Question

Do there exist problems which can be considered to be the most difficult ones within their corresponding complexity class?

We shall prove the *affirmative answer*, and refer to these problems as *complete problems*.

The importance of complete problems is easily explained. The efficient solution of a *complete* problem for $\mathcal{NL}$ or $\mathcal{NP}$ could be used to efficiently solve *all* of the problems in $\mathcal{NL}$ or $\mathcal{NP}$, respectively.

Next, we have to modify the deterministic TM model in a way such that strings can be computed as output. We use Σ to denote any fixed finite alphabet and Σ^* for denoting the free monoid over Σ.

Reductions I

Definition 1

A function $f \colon \Sigma^* \to \Sigma^*$ is said to be *log-space computable* if there exists a deterministic TM M_f satisfying the following properties:

(1) M_f possesses an input tape with a two-way read-only head, an output tape with a one-way write-only head and finitely many work tapes each of which has a two-way read-write head.

(2) On input w the machine M_f computes $f(w)$ and writes it on its output tape. While performing this computation, M_f uses on each of its work tapes at most $O(\log |w|)$ many tape cells.

Reductions II

Log-space computable functions have an interesting property which is stated as an exercise.

Exercise 1. *Show that for each log-space computable function the condition* $|f(w)| \leqslant |w|^{O(1)}$ *is satisfied.*

Reductions II

Log-space computable functions have an interesting property which is stated as an exercise.

Exercise 1. *Show that for each log-space computable function the condition $|f(w)| \leqslant |w|^{O(1)}$ is satisfied.*

Next, we define reductions.

Definition 2

Let $A, B \subseteq \Sigma^*$ be any two decidable languages. The language A is said to be *log-space reducible* to the language B (abbr. $A \leqslant_{\log} B$) if there exists a log-space computable function f such that for all $w \in \Sigma^*$ the condition $w \in A$ if and only if $f(w) \in B$ is satisfied.

Reductions II

Log-space computable functions have an interesting property which is stated as an exercise.

Exercise 1. *Show that for each log-space computable function the condition* $|f(w)| \leqslant |w|^{O(1)}$ *is satisfied.*

Next, we define reductions.

Definition 2

Let $A, B \subseteq \Sigma^*$ be any two decidable languages. The language A is said to be *log-space reducible* to the language B (abbr. $A \leqslant_{\log} B$) if there exists a log-space computable function f such that for all $w \in \Sigma^*$ the condition $w \in A$ if and only if $f(w) \in B$ is satisfied.

Exercise 2. *Let* $L_1, L_2, L_3 \subseteq \Sigma^*$ *be any decidable languages. Then we have: If* $L_1 \leqslant_{\log} L_2$ *and* $L_2 \leqslant_{\log} L_3$ *then* $L_1 \leqslant_{\log} L_3$*, i.e., log-space reducibility is transitive.*

Reductions III

Now, we are ready to define the notions of hardness and completeness.

Definition 3

Let $\mathcal{S}$ be a family of decidable languages over Σ^* and let L_0 be a language such that $L_0 \subseteq \Sigma^*$. The language L_0 is said to be *log-space hard* for $\mathcal{S}$ if $L \leqslant_{\log} L_0$ for every language $L \in \mathcal{S}$. If additionally $L_0 \in \mathcal{S}$ is satisfied then the language L_0 is said to be *log-space complete* for $\mathcal{S}$.

Reductions III

Now, we are ready to define the notions of hardness and completeness.

Definition 3

Let $\mathcal{S}$ be a family of decidable languages over Σ^* and let L_0 be a language such that $L_0 \subseteq \Sigma^*$. The language L_0 is said to be *log-space hard* for $\mathcal{S}$ if $L \leqslant_{\log} L_0$ for every language $L \in \mathcal{S}$. If additionally $L_0 \in \mathcal{S}$ is satisfied then the language L_0 is said to be *log-space complete* for $\mathcal{S}$.

Next, we ask in which sense a language $A \subseteq \Sigma^*$ is easier than a language $B \subseteq \Sigma^*$ provided $A \leqslant_{\log} B$. This is done via the following lemma:

Reductions IV

Lemma 1

Let M *be a TM such that* $S_M(n) \neq o(\log n)$. *If a language* $L \subseteq \Sigma^*$ *is log-space reducible to* $L(M)$ *then there exists a TM* $\widetilde{M}$ *such that* $L = L(\widetilde{M})$ *and* $S_{\widetilde{M}}(n) = O(S_M(n))$.

Reductions IV

Lemma 1

Let M *be a TM such that* $S_M(n) \neq o(\log n)$*. If a language* $L \subseteq \Sigma^*$ *is log-space reducible to* $L(M)$ *then there exists a TM* $\widetilde{M}$ *such that* $L = L(\widetilde{M})$ *and* $S_{\widetilde{M}}(n) = O(S_M(n))$*.*

Proof. The proof idea is to combine the acceptor TM M with a TM M_f that realizes the log-space translation of L into $L(M)$. But there is a problem. The space bound of M does not allow, in general, to write the result $f(w)$ of the translation of w via M_f on M's work tape(s). Hence, we have to modify M_f appropriately. We define a deterministic TM M_f' as follows:

Reductions V

On input w and input $\mathrm{bin}(k)$ on an auxiliary work tape, M'_f works as M_f does but writes only the kth symbol of $f(w)$ on its output tape. Since $|f(w)| \leqslant |w|^{O(1)}$ the space bound $O(\log n)$ for M'_f is ensured. The TM M'_f can count all attempts of M_f to write a symbol on its output tape until the kth one is reached which is then executed.

Finally, M is modified in way such that each change of the head position on the input tape of M is accompanied by setting the binary counter to the actual input head position and by computing the symbol to be read by executing M'_f on input w and $\mathrm{bin}(k)$ as described above. ∎

Reductions V

On input w and input $\mathrm{bin}(k)$ on an auxiliary work tape, M'_f works as M_f does but writes only the kth symbol of $f(w)$ on its output tape. Since $|f(w)| \leqslant |w|^{O(1)}$ the space bound $O(\log n)$ for M'_f is ensured. The TM M'_f can count all attempts of M_f to write a symbol on its output tape until the kth one is reached which is then executed.

Finally, M is modified in way such that each change of the head position on the input tape of M is accompanied by setting the binary counter to the actual input head position and by computing the symbol to be read by executing M'_f on input w and $\mathrm{bin}(k)$ as described above. ∎

Please note that the condition $S_M(n) \neq o(\log n)$ was essential for proving the latter lemma, since otherwise we could not have used the binary counter.

Reductions VI

Lemma 1 allows for the following corollary:

Corollary 1

Let $L, L' \subseteq \Sigma^*$ *be any languages.*

(1) *If* $L \in \mathcal{L}$ *and* $L' \leqslant_{\log} L$ *then* $L' \in \mathcal{L}$.

(2) *If* $L \in \mathcal{L}$ *and* $\emptyset \neq L' \neq \Sigma^*$ *then* $L \leqslant_{\log} L'$.

Proof. We leave it as an exercise to prove this corollary. ∎

Consequently, $\mathcal{L}$ constitutes the lowest level of log-space reducibility.

Remarks

It should also be noted that there are a couple of reducibility notions around which have been intensively studied in the literature. We mention here only *polynomial-time* reducibility which is defined analogously as log-space reducibility.

Remarks

It should also be noted that there are a couple of reducibility notions around which have been intensively studied in the literature. We mention here only *polynomial-time* reducibility which is defined analogously as log-space reducibility.
The only difference to Definition 2 is that the function f is now only required to be computable by a deterministic TM obeying a polynomial time bound for its computation time instead of the log-space bound required in Definition 1. If a language L_1 is polynomial-time reducible to a language L_2 then we write $L_1 \leqslant_{\text{poly}} L_2$.

Remarks

It should also be noted that there are a couple of reducibility notions around which have been intensively studied in the literature. We mention here only *polynomial-time* reducibility which is defined analogously as log-space reducibility.
The only difference to Definition 2 is that the function f is now only required to be computable by a deterministic TM obeying a polynomial time bound for its computation time instead of the log-space bound required in Definition 1. If a language L_1 is polynomial-time reducible to a language L_2 then we write $L_1 \leqslant_{\text{poly}} L_2$.

For getting a better understanding of polynomial-time reducibility, we recommend to solve the following exercise:

Reductions VII

Exercise 3. *Let* L_1, L_2 *be any two languages. If* $L_1 \leqslant_{\log} L_2$ *then* $L_1 \leqslant_{\text{poly}} L_2$.

The notion of reducibility also allows one to define an equivalence relation.

Reductions VII

Exercise 3. *Let* L_1, L_2 *be any two languages. If* $L_1 \leqslant_{\log} L_2$ *then* $L_1 \leqslant_{\text{poly}} L_2$.

The notion of reducibility also allows one to define an equivalence relation.

Definition 4

Let $L_1, L_2 \subseteq \Sigma^*$ be any two decidable languages. The languages L_1 and L_2 are said to be *equivalent* with respect to log-space reducibility (polynomial-time reducibility) if $L_1 \leqslant_{\log} L_2$ and $L_2 \leqslant_{\log} L_1$ ($L_1 \leqslant_{\text{poly}} L_2$ and $L_2 \leqslant_{\text{poly}} L_1$).

If L_1 and L_2 are equivalent with respect to log-space reducibility and polynomial-time reducibility then we write $L_1 \equiv_{\log} L_2$ and $L_1 \equiv_{\text{poly}} L_2$, respectively.

GAP I

Our next goal is to establish the existence of complete problems for the complexity class $\mathcal{NL}$ defined in the last lecture.
For that purpose, we define the *graph accessibility problem* (abbr. GAP).

GAP

Input: A directed graph $G = (V, E)$ with vertex set $V = \{v_1, \dots, v_m\}$ and a distinguished start node v_s and a distinguished end node v_e.

Problem: Does there exist a path between v_s and v_e?

If the graph G is given by its adjacency-list, then the input length n of GAP can be bounded by $O(m^2 \log m)$. Moreover, we can safely assume $n \geqslant m$.

GAP II

Next, we show GAP to be $\mathcal{NL}$-complete. This is done in two steps; i.e., by first showing GAP to be $\mathcal{NL}$-hard and then GAP to be in $\mathcal{NL}$.

Lemma 2

GAP *is $\mathcal{NL}$-hard.*

GAP II

Next, we show GAP to be $\mathcal{NL}$-complete. This is done in two steps; i.e., by first showing GAP to be $\mathcal{NL}$-hard and then GAP to be in $\mathcal{NL}$.

Lemma 2

GAP *is* $\mathcal{NL}$*-hard.*

Proof. Let M be a TM such that $S_M(n) \in O(\log n)$ and let w be an input to M. We define a graph $G_w = (V, E)$ as follows:

GAP III

The nodes of G_w are *all* the macro states of M that can occur under the space bound $S_M(|w|)$. Let v and v' be any two macro states of M (i.e., any two nodes of G_w). We define $(v, v') \in E$ iff M can reach macro state v' from macro state v in one step. Without loss of generality, we can assume that the macro state at the beginning of M's computation on input w is uniquely determined. Also without loss of generality, we can assume that, if M accepts w, then the accepting macro state of M is uniquely determined, too.

GAP III

The nodes of G_w are *all* the macro states of M that can occur under the space bound $S_M(|w|)$. Let v and v' be any two macro states of M (i.e., any two nodes of G_w). We define $(v, v') \in E$ iff M can reach macro state v' from macro state v in one step. Without loss of generality, we can assume that the macro state at the beginning of M's computation on input w is uniquely determined. Also without loss of generality, we can assume that, if M accepts w, then the accepting macro state of M is uniquely determined, too. Now, it is easy to see that the graph G_w is log-space computable from input w, since the number of nodes is uniformly polynomially bounded in $|w|$. If the nodes of G_w are appropriately numbered, then our construction implies

$$w \in L(M) \iff G_w \in \mathrm{GAP}\,.$$

So, every language from $\mathcal{NL}$ is log-space reducible to GAP. ∎

GAP IV

Next, we show GAP to be acceptable by an NTM.

Lemma 3

$\mathrm{GAP} \in \mathcal{NL}$.

GAP IV

Next, we show GAP to be acceptable by an NTM.

Lemma 3

$GAP \in \mathcal{NL}$.

Proof. Let any graph $G = (V, E)$ with vertex set $V = \{v_1, \ldots, v_m\}$ and a distinguished start node v_s and a distinguished end node v_e be given as input. Let n be the length of the input. As shown above, n can be bounded by $O(m^2 \log m)$. Thus, the space bound $\log n$ is sufficient to store any node number in binary.

GAP IV

Next, we show GAP to be acceptable by an NTM.

Lemma 3

$\text{GAP} \in \mathcal{NL}$.

Proof. Let any graph $G = (V, E)$ with vertex set $V = \{v_1, \dots, v_m\}$ and a distinguished start node v_s and a distinguished end node v_e be given as input. Let n be the length of the input. As shown above, n can be bounded by $O(m^2 \log m)$. Thus, the space bound $\log n$ is sufficient to store any node number in binary.

The NTM M works as follows: First, it stores the number s of the start node v_s. Then, non-deterministically any successor of v_s, say v_i, is chosen (that is, $(v_s, v_i) \in E$), s is erased, and the number i is stored as actual node number.

GAP V

Next, the process is iterated. That is, assuming j to be the actual node number, any successor of v_j, say v_k, is chosen and its number k is stored as actual node number and j is erased. The storing and erasing is done in a way such that the total amount of space used by M is $O(\log n)$.

GAP V

Next, the process is iterated. That is, assuming j to be the actual node number, any successor of v_j, say v_k, is chosen and its number k is stored as actual node number and j is erased. The storing and erasing is done in a way such that the total amount of space used by M is $O(\log n)$.

The graph G is accepted, if e is reached as actual node number. Otherwise, G is not accepted.

GAP V

Next, the process is iterated. That is, assuming j to be the actual node number, any successor of v_j, say v_k, is chosen and its number k is stored as actual node number and j is erased. The storing and erasing is done in a way such that the total amount of space used by M is $O(\log n)$.

The graph G is accepted, if e is reached as actual node number. Otherwise, G is not accepted.

Clearly, if there is a path from v_s to v_e in G, then there is an accepting computation of M on input G. Otherwise, no computation can accept G. ∎

GAP VI

By Definition 3, the Lemmata 2 and 3 directly imply the following corollary:

Corollary 2

GAP *is* $\mathcal{NL}$*-complete.*

GAP VI

By Definition 3, the Lemmata 2 and 3 directly imply the following corollary:

Corollary 2

GAP *is* $\mathcal{NL}$*-complete.*

Moreover, we immediately obtain the following corollary:

Corollary 3

Let $f(n) \neq o(\log n)$ *be a space bounding function. Then we have* GAP $\in$ *SPACE*$(f(n))$ *if and only if* $\mathcal{NL} \subseteq$ *SPACE*$(f(n))$.

GAP VI

By Definition 3, the Lemmata 2 and 3 directly imply the following corollary:

Corollary 2

GAP is $\mathcal{NL}$-complete.

Moreover, we immediately obtain the following corollary:

Corollary 3

Let $f(n) \neq o(\log n)$ be a space bounding function. Then we have $\text{GAP} \in \mathit{SPACE}(f(n))$ if and only if $\mathcal{NL} \subseteq \mathit{SPACE}(f(n))$.

Further $\mathcal{NL}$-complete problems are studied in the book.

$\mathcal{NP}$-complete Problems I

Now, we turn our attention to the class $\mathcal{NP}$ which contains many important problems. We start with a list of examples for decision problems that turn out to be all in $\mathcal{NP}$. We define these problems here as languages and assume any reasonable encoding of the input.

$\mathcal{NP}$-complete Problems I

Now, we turn our attention to the class $\mathcal{NP}$ which contains many important problems. We start with a list of examples for decision problems that turn out to be all in $\mathcal{NP}$. We define these problems here as languages and assume any reasonable encoding of the input.

Let $G = (V, E)$ be an undirected graph. A complete subgraph of size k of G is said to be a k-*Clique*. Here a graph is said to be complete if every vertex is connected to any other vertex. We set

$$\text{CLIQUE} = \{(G, k) \mid G \text{ possesses a } k\text{-Clique}\} .$$

$\mathcal{NP}$-complete Problems I

Now, we turn our attention to the class $\mathcal{NP}$ which contains many important problems. We start with a list of examples for decision problems that turn out to be all in $\mathcal{NP}$. We define these problems here as languages and assume any reasonable encoding of the input.

Let $G = (V, E)$ be an undirected graph. A complete subgraph of size k of G is said to be a k-*Clique*. Here a graph is said to be complete if every vertex is connected to any other vertex. We set

$$\text{CLIQUE} = \{(G, k) \mid G \text{ possesses a } k\text{-Clique}\} .$$

Let $G = (V, E)$ be an undirected graph. A set $U \subseteq V$ is said to be *independent* if $(u, v) \notin E$ for all $u, v \in U$, $u \neq v$. We set

$$\text{INDSET} = \{(G, k) \mid G \text{ possesses an independent set of size } k\} .$$

$\mathcal{NP}$-complete Problems II

Now, let $G = (V, E)$ be a directed graph. A *Hamiltonian path* is a path visiting all vertices of G exactly ones. We set

$$\text{dHAMILTON} = \{G \mid G \text{ possesses a Hamiltonian path}\} .$$

$\mathcal{NP}$-complete Problems II

Now, let $G = (V, E)$ be a directed graph. A *Hamiltonian path* is a path visiting all vertices of G exactly ones. We set

$$\text{dHAMILTON} = \{G \mid G \text{ possesses a Hamiltonian path}\} .$$

A *vertex cover* of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$. The size of a vertex cover V' is the cardinality of V'. We set

$$\text{VCOVER} = \{(G, k) \mid G \text{ possesses a vertex cover of size } k\} .$$

$\mathcal{NP}$-complete Problems III

Subset Sum Problem

Input: a number M and a vector $(a_0, \ldots, a_{n-1}) \in \mathbb{N}^n$.

Problem: Decide whether there exists a vector $(b_0, \ldots, b_{n-1}) \in \{0,1\}^n$ such that $M = \sum_{j=0}^{n-1} a_j b_j$.

As with any arithmetic problem, it is important to recall that the input integers are coded in binary. Then we define SUBSUM to be the language of all subset sum problems $((a_0, \ldots, a_{n-1}), M)$ for which there is a vector $(b_0, \ldots, b_{n-1}) \in \{0,1\}^n$ such that $M = \sum_{j=0}^{n-1} a_j b_j$.

$\mathcal{NP}$-complete Problems IV

Finally, we define the famous satisfiability problem.

Definition 5

Let $F = f(x_1, \ldots, x_n)$ be a Boolean formula consisting of the variables $x_1, \ldots, x_n$ and the Boolean operators $\vee, \wedge, \neg$. F is said to be *satisfiable* if there exists an assignment $(a_1, \ldots, a_n) \in \{0, 1\}^n$ to the variables $x_1, \ldots, x_n$ such that $F(a_1, \ldots, a_n) = 1$.

$\mathcal{NP}$-complete Problems IV

Finally, we define the famous satisfiability problem.

Definition 5

Let $F = f(x_1, \ldots, x_n)$ be a Boolean formula consisting of the variables $x_1, \ldots, x_n$ and the Boolean operators $\vee, \wedge, \neg$. F is said to be *satisfiable* if there exists an assignment $(a_1, \ldots, a_n) \in \{0, 1\}^n$ to the variables $x_1, \ldots, x_n$ such that $F(a_1, \ldots, a_n) = 1$.

The satisfiability problem is then the language

$$\text{SAT} = \{F \mid F \text{ is a satisfiable formula}\}.$$

$\mathcal{NP}$-complete Problems V

Let us ask what all the languages defined above do have in common. The general pattern is that it is presumably very hard to *find* a witness that any of its instances belongs to them. For example, in order to find a satisfying assignment one may have to try all possible assignments, i.e., all 2^n many Boolean vectors $a_1, \ldots, a_n \in \{0, 1\}^n$. The same clearly applies for SUBSUM. As for dHAMILTON, one may be forced to try all $n!$ permutations of the vertices of $G = (V, E)$, where $|V| = n$ in order to find a Hamiltonian path.

$\mathcal{NP}$-complete Problems V

Let us ask what all the languages defined above do have in common. The general pattern is that it is presumably very hard to *find* a witness that any of its instances belongs to them. For example, in order to find a satisfying assignment one may have to try all possible assignments, i.e., all 2^n many Boolean vectors $a_1, \ldots, a_n \in \{0, 1\}^n$. The same clearly applies for SUBSUM. As for dHAMILTON, one may be forced to try all $n!$ permutations of the vertices of $G = (V, E)$, where $|V| = n$ in order to find a Hamiltonian path.

On the other hand, it is for all the languages given above easy to *check* whether or not a witness is given. For instance, for any given assignment one can quickly check whether or not it is satisfying a given Boolean formula by a deterministic TM. Informally, this property may serve as a rule of thumb for deciding whether or not any given language belongs to $\mathcal{NP}$.

$\mathcal{NP}$-complete Problems VI

Next, we ask whether or not the class $\mathcal{NP}$ contains an $\mathcal{NP}$-complete language. The affirmative answer has been given by Cook (1971) and Levin (1973), who could show the following important theorem:

Theorem 4 (Cook (1971), Levin (1973))

SAT *is $\mathcal{NP}$-complete.*

We are not going to prove this theorem here, since there are many proofs in the literature. Furthermore, there is no need to prove any problem to be $\mathcal{NP}$-complete by using Cook's (1971) original proof technique.

$\mathcal{NP}$-complete Problems VI

Next, we ask whether or not the class $\mathcal{NP}$ contains an $\mathcal{NP}$-complete language. The affirmative answer has been given by Cook (1971) and Levin (1973), who could show the following important theorem:

Theorem 4 (Cook (1971), Levin (1973))

SAT *is $\mathcal{NP}$-complete.*

We are not going to prove this theorem here, since there are many proofs in the literature. Furthermore, there is no need to prove any problem to be $\mathcal{NP}$-complete by using Cook's (1971) original proof technique.

Instead, to show the $\mathcal{NP}$-completeness of any other language L it suffices to reduce SAT or any other language known to be $\mathcal{NP}$-complete to L.

$\mathcal{NP}$-complete Problems VII

Next, we exemplify this proof technique here.

A formula F is said to be in ℓ-CNF form if F is in conjunctive normal form and each clause contains precisely ℓ literals.

Let ℓ-SAT be the language of all satisfiable formulae in ℓ-CNF form. Then, we can show the following:

Theorem 5

ℓ-SAT is $\mathcal{NP}$-complete for all $\ell \geqslant 3$.

$\mathcal{NP}$-complete Problems VII

Next, we exemplify this proof technique here.
A formula F is said to be in ℓ-CNF form if F is in conjunctive normal form and each clause contains precisely ℓ literals.
Let ℓ-SAT be the language of all satisfiable formulae in ℓ-CNF form. Then, we can show the following:

Theorem 5

ℓ-SAT is $\mathcal{NP}$-complete for all $\ell \geqslant 3$.

Proof. Since SAT is in $\mathcal{NP}$ we have ℓ-SAT $\in \mathcal{NP}$, too. Thus, it suffices to log-space reduce SAT to ℓ-SAT.
First, we show that any formula F can be transformed into a sat-equivalent formula F' in CNF. Here by sat-equivalent we mean that F is satisfiable iff F' is satisfiable.

$\mathcal{NP}$-complete Problems VIII

Note that we cannot just transform F into its CNF, since the length of the CNF may be exponential in the length of F, thus violating our requirement to log-space reduce SAT to ℓ-SAT. For obtaining the desired transformation of F into a sat-equivalent formula F' in CNF, in general we need new auxiliary variables. Here by new we mean that these variables do not occur in F. We proceed as follows: In our first step we transform F into a logical equivalent formula F' by using de Morgan's rules as well as $\neg\neg x \equiv x$. Note the we use both $\neg x$ and $\overline{x}$ to denote negation.

Step 1. Using de Morgan's rules, we transform F into F' such that all negations in F' appear at the variables.

After a bit of reflection it is easy to see that Step 1 can be realized in log-space.

$\mathcal{NP}$-complete Problems IX

Let F' be the formula obtained so far. Next, we transform F' into a sat-equivalent CNF by using the following observation: If $F' = F_1 \vee F_2$ and $F_1,\ F_2$ are already in CNF, then we can replace F' by

$$(F_1 \vee y) \wedge (F_2 \vee \overline{y}) ,$$

where y is a new variable. Clearly, the new formula is sat-equivalent to F'. We refer to this rule as to Rule 1. Furthermore, we need Rule 2 displayed below to transform $F_1 \vee y$ and $F_2 \vee \overline{y}$ into a CNF. This is done as follows: Let $F_i = G_1 \wedge G_2 \wedge \ldots \wedge G_k$. Then $F_i \vee y^\alpha$ is equivalent to

$$(G_1 \vee y^\alpha) \wedge (G_2 \vee y^\alpha) \wedge \ldots \wedge (G_k \vee y^\alpha) ,$$

and we have again a conjunction of clauses.

Step 2. Apply Rules 1 and 2 recursively until a CNF is obtained.

$\mathcal{NP}$-complete Problems X

Next, we have to show that any formula in CNF can be transformed into a sat-equivalent formula in ℓ-CNF form. For the sake of presentation we handle here the case $\ell = 3$, only. Consider any clause $C = (z_1 \vee \cdots \vee z_k)$. In dependence on k we replace C by the following formula by using new variables y_i:

$$
\begin{aligned}
k = 1:&\quad (z_1 \vee y_1 \vee y_2) \wedge (z_1 \vee \overline{y}_1 \vee y_2) \wedge (z_1 \vee y_1 \vee \overline{y}_2) \wedge (z_1 \vee \overline{y}_1 \vee \overline{y}_2)\\
k = 2:&\quad (z_1 \vee z_2 \vee y_1) \wedge (z_1 \vee z_2 \vee \overline{y}_1)\\
k = 3:&\quad (z_1 \vee z_2 \vee z_3) \quad \text{i.e., we do not change } C\\
k > 3:&\quad (z_1 \vee z_2 \vee y_1) \wedge (\overline{y}_1 \vee z_3 \vee y_2) \wedge (\overline{y}_2 \vee z_4 \vee y_3) \wedge\\
&\quad \ldots \wedge (\overline{y}_{k-4} \vee z_{k-2} \vee y_{k-3}) \wedge (\overline{y}_{k-3} \vee z_{k-1} \vee z_k) =: \tilde{C}\,.
\end{aligned}
$$

Clearly, these formulae can be computed in log-space.

$\mathcal{NP}$-complete Problems XI

The sat-equivalence of the formulae obtained can be seen as follows: In case $k = 1$, the four clauses can be simultaneously satisfied if and only if z_1 is assigned the value 1, since independently of the assignments for y_1, y_2, in one of the four clauses the resulting evaluation is 0.

Analogously, one directly sees that in case $k = 2$ the two clauses can be simultaneously satisfied if and only if z_1 or z_2 is assigned the value 1.

For $k = 3$ nothing has to be shown.

$\mathcal{NP}$-complete Problems XII

Finally, for $k > 3$ it remains to show that C is satisfiable if and only if $\tilde{C}$ is satisfiable.

$\mathcal{NP}$-complete Problems XII

Finally, for $k > 3$ it remains to show that C is satisfiable if and only if $\tilde{C}$ is satisfiable.

Assume $(z_1 \vee \cdots \vee z_k)$ is satisfied by $z_i = 1$. If $i = 1$ or $i = 2$, then we set $y_j = 0$ for all $j = 1, \ldots, k-3$. So, the first clause in $\tilde{C}$ is satisfied by z_1 or z_2 and all remaining clauses in $\tilde{C}$ are satisfied by $\overline{y}_j$, $j = 1, \ldots, k-3$.

$\mathcal{NP}$-complete Problems XII

Finally, for $k > 3$ it remains to show that C is satisfiable if and only if $\tilde{C}$ is satisfiable.

Assume $(z_1 \vee \cdots \vee z_k)$ is satisfied by $z_i = 1$. If $i = 1$ or $i = 2$, then we set $y_j = 0$ for all $j = 1, \ldots, k-3$. So, the first clause in $\tilde{C}$ is satisfied by z_1 or z_2 and all remaining clauses in $\tilde{C}$ are satisfied by $\overline{y}_j$, $j = 1, \ldots, k-3$.

If $i \geqslant 3$, then we set $y_1 = y_2 = \cdots = y_{i-2} = 1$, $y_{i-1} = y_i = \cdots y_{k-3} = 0$. Now, by construction, in $\tilde{C}$ the first $i-2$ clauses are satisfied by by the y_i, the $(i-1)$st clause (containing z_i) is clearly satisfied by z_1, and the remaining $k-(i-2)-3$ clauses in $\tilde{C}$ are satisfied by $\overline{y}_i$.

$\mathcal{NP}$-complete Problems XIII

Next, assume $\tilde{C}$ to be satisfied. We distinguish the following 3 cases: If all $y_j = 0$, then $z_{k-1} \vee z_k$ must evaluate to 1, thus also C is satisfied. Analogously, if all $y_j = 1$, then $z_1 \vee z_2$ must evaluate to 1, and hence C is satisfied, too.

$\mathcal{NP}$-complete Problems XIII

Next, assume $\tilde{C}$ to be satisfied. We distinguish the following 3 cases: If all $y_j = 0$, then $z_{k-1} \vee z_k$ must evaluate to 1, thus also C is satisfied. Analogously, if all $y_j = 1$, then $z_1 \vee z_2$ must evaluate to 1, and hence C is satisfied, too.

It remains to consider the case that up to some i, $1 \leqslant i < k - 3$ we have $y_1 = \cdots = y_i = 1$ and $y_{i+1} = 0$. Now, the $(i+1)$st clause of $\tilde{C}$ can evaluate to 1 if and only if $z_{i+2} = 1$, that is, C is again satisfied. $\blacksquare$

$\mathcal{NP}$-complete Problems XIV

The importance of 3-SAT is its simple combinatorial structure which allows to apply it to prove the $\mathcal{NP}$-completeness of many other problems as shown below. Note that the condition $\ell \geqslant 3$ is essential.

Exercise 4. *Prove or disprove* 2-SAT $\in \mathcal{P}$.

$\mathcal{NP}$-complete Problems XIV

The importance of 3-SAT is its simple combinatorial structure which allows to apply it to prove the $\mathcal{NP}$-completeness of many other problems as shown below. Note that the condition $\ell \geqslant 3$ is essential.

Exercise 4. *Prove or disprove* 2-SAT $\in \mathcal{P}$*.*

Next, by reducing 3-SAT to CLIQUE one can easily prove the following theorem:

Theorem 6

CLIQUE *is* $\mathcal{NP}$*-complete.*

The proof is given in the book.

$\mathcal{NP}$-complete Problems XV

Having shown CLIQUE to be $\mathcal{NP}$-complete directly allows to prove VCOVER to be $\mathcal{NP}$-complete, too.

Theorem 7

VCOVER *is* $\mathcal{NP}$*-complete.*

$\mathcal{NP}$-complete Problems XV

Having shown CLIQUE to be $\mathcal{NP}$-complete directly allows to prove VCOVER to be $\mathcal{NP}$-complete, too.

Theorem 7

VCOVER *is* $\mathcal{NP}$*-complete.*

Proof. It is easy to see that VCOVER $\in \mathcal{NP}$. Next, we reduce CLIQUE to VCOVER. The reduction is almost trivial. Let $G = (V, E)$ and k be given. We map G to its complement graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) \mid u, v \in V,\ u \neq v,\ (u, v) \notin E\}$. Furthermore, k is mapped to $|V| - k$. We omit the details. ∎

$\mathcal{NP}$-complete Problems XV

Having shown CLIQUE to be $\mathcal{NP}$-complete directly allows to prove VCOVER to be $\mathcal{NP}$-complete, too.

Theorem 7

VCOVER *is* $\mathcal{NP}$*-complete.*

Proof. It is easy to see that VCOVER $\in \mathcal{NP}$. Next, we reduce CLIQUE to VCOVER. The reduction is almost trivial. Let $G = (V, E)$ and k be given. We map G to its complement graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) \mid u, v \in V,\ u \neq v,\ (u, v) \notin E\}$. Furthermore, k is mapped to $|V| - k$. We omit the details. ∎

Exercise 5. *Show* SUBSUM *to be* $\mathcal{NP}$*-complete.*

Example

Example 6

$F = \neg(\neg(x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_4 \vee (x_3 \wedge \overline{x}_5)))$.

Then, in Step 1, by using $\neg(\beta_1 \wedge \beta_2) \equiv \neg\beta_1 \vee \neg\beta_2$ or $\neg(\beta_1 \vee \beta_2) \equiv \neg\beta_1 \wedge \neg\beta_2$ we successively obtain:

$\neg(\neg(x_1 \vee x_2 \vee \overline{x}_3) \wedge (x_4 \vee (x_3 \wedge \overline{x}_5)))$

$$\begin{aligned}
&\equiv \neg\neg(x_1 \vee x_2 \vee \overline{x}_3) \vee \neg(x_4 \vee (x_3 \wedge \overline{x}_5)) \\
&\equiv (x_1 \vee x_2 \vee \overline{x}_3) \vee \neg(x_4 \vee (x_3 \wedge \overline{x}_5)) \\
&\equiv (x_1 \vee x_2 \vee \overline{x}_3) \vee (\overline{x}_4 \wedge \neg(x_3 \wedge \overline{x}_5)) \\
&\equiv (x_1 \vee x_2 \vee \overline{x}_3) \vee (\overline{x}_4 \wedge (\overline{x}_3 \vee x_5)) .
\end{aligned}$$

Example continued

Continuing our example, we thus obtain (where $\sim$ denotes sat-equivalence)

$$\begin{aligned}
& (x_1 \vee x_2 \vee \overline{x}_3) \vee (\overline{x}_4 \wedge (\overline{x}_3 \vee x_5)) \\
\sim\ & (x_1 \vee x_2 \vee \overline{x}_3 \vee y_1) \wedge (\overline{x}_4 \vee \overline{y}_1) \wedge (\overline{x}_3 \vee x_5 \vee \overline{y}_1)\,. \qquad (2)
\end{aligned}$$

Example continued

We finish our example here for the case of 3-CNF.

So, we have to apply the rules for $k > 3$ and $k = 2$. Applying the rule for $k > 3$ requires the introduction of a new variable y_2 and applying the rule for $k = 2$ requires the introduction of a new variable y_3. Thus, we finally obtain.

$$
\begin{aligned}
& (x_1 \vee x_2 \vee \overline{x}_3 \vee y_1) \wedge (\overline{x}_4 \vee \overline{y}_1) \wedge (\overline{x}_3 \vee x_5 \vee \overline{y}_1) \\
\sim\ & (x_1 \vee x_2 \vee y_2) \wedge (\overline{y}_2 \vee \overline{x}_3 \vee y_1) \wedge (\overline{x}_4 \vee \overline{y}_1 \vee y_3) \wedge (\overline{x}_4 \vee \overline{y}_1 \vee \overline{y}_3) \\
& \wedge (\overline{x}_3 \vee x_5 \vee \overline{y}_1) \; .
\end{aligned}
$$

Final Remarks I

As already mentioned, so far we do not know whether or not $\mathcal{P} = \mathcal{NP}$. Resolving this problem remains a huge challenge.

So, let us shortly discuss consequences of the two possible answers. If $\mathcal{P} \neq \mathcal{NP}$, then not much will change, since this conjecture is favored by many scientists. The main change, of course, is then the switch from conjecture to theorem, and all the theorems having a "…if $\mathcal{P} \neq \mathcal{NP}$" in their statement would be unconditionally true.

Final Remarks II

What are the consequences if we could prove that $\mathcal{P} = \mathcal{NP}$? Clearly this result would be also of fundamental epistemological importance. But the practical consequences may vary. If, for some important $\mathcal{NP}$-complete problem like 3-SAT someone finds a very efficient algorithm, say having running time $O(n^2)$, then the practical consequences would be heaven and hell at the same time. Heaven for those who need to find quickly solutions for $\mathcal{NP}$-complete problems, e.g., for many AI applications, for VLSI designers, for engineers.

Final Remarks II

What are the consequences if we could prove that $\mathcal{P} = \mathcal{NP}$? Clearly this result would be also of fundamental epistemological importance. But the practical consequences may vary. If, for some important $\mathcal{NP}$-complete problem like 3-SAT someone finds a very efficient algorithm, say having running time $O(n^2)$, then the practical consequences would be heaven and hell at the same time. Heaven for those who need to find quickly solutions for $\mathcal{NP}$-complete problems, e.g., for many AI applications, for VLSI designers, for engineers.

On the other hand, all tools currently in use for privacy protection, e.g., SSL, RSA, or PGP will become useless over night. Also, much of what mathematician are doing could then be done by a machine performing efficient theorem proving.

Final Remarks III

But it is also possible that the best polynomial time algorithm for any $\mathcal{NP}$-complete problem has a running time of order $O(n^c)$, where c is a six digit number, or even a 1000000 digit number. Of course, in this case the practical consequences would be much less dramatic, since the $\mathcal{NP}$-complete problems remain hard to *solve* for larger inputs. If the latter would be true, this would also explain why we have not found any such algorithm yet.

Last but not least, if $\mathcal{P} = \mathcal{NP}$ then randomization would not provide any principal gain.

Thank you!

Walter J. Savitch

Steven A. Cook

Leonid Levin