CS301: Computability and Complexity Theory (CC)

Lecture 11-12: NP-completeness

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

January 12, 2024

Table of contents

Section 1

Previously on CS301

Verifiers

Definition

A **verifier** for a language A is an algorithm V , where

$$A = \{w \mid V\ \mathit{accepts}\ < w, c > \ \mathit{for\ some\ string}\ c\}$$

We measure the time of a verifier only in terms of the length of w, so a *polynomial time verifier* runs in polynomial time in the length of w. A language A is *polynomially verifiable* if it has a polynomial time verifier

A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of A. This information is called a *certificate*, or *proof*, of membership in A
Observe that for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound

Class NP

Definition

NP is the class of languages that have polynomial time verifiers

The class NP is important because it contains many problems of practical interest. From the preceding discussion, both *HAMPATH* and *COMPOSITES* are members of NP

The term NP comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. Problems in NP are sometimes called NP-problems

Class NP

Theorem

A language is in NP iff it is decided by some nondeterministic polynomial time TM

Proof idea: We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate

P vs NP

As we saw, NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine; or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time

P is the class of languages where membership can be tested in polynomial time. We summarize this information as follows, where we loosely refer to polynomial time solvable as solvable "quickly"

P = the class of languages for which membership can be *decided* quickly

NP = the class of languages for which membership can be *verified* quickly

P vs NP

- The question of whether $\mathrm{P} = \mathrm{NP}$ is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics
- If these classes were equal, any polynomially verifiable problem would be polynomially decidable
- Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success
- Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search
- Doing so is presently beyond scientific reach

P vs NP

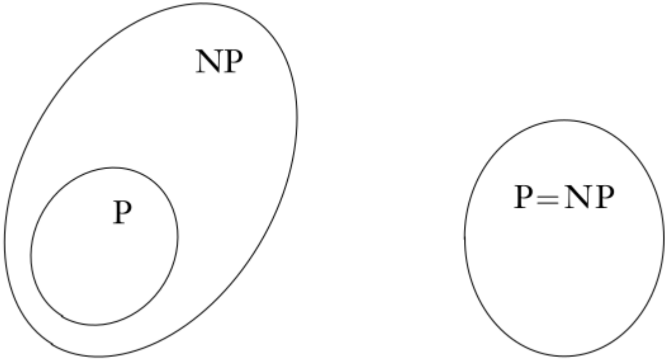

Figure: One of these two possibilities is correct

Section 2

Context setup

Context setup

Corresponding to 7.4

Context setup

- We continue the study of **NP**
- and that is all!

Section 3

NP-Completeness

NP-Completeness

- One important advance on the *P versus NP question* came in the early 1970s with the work of Stephen Cook and Leonid Levin
- They discovered certain problems in NP whose individual complexity is related to that of the entire class
- If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable
- These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons

NP-Completeness

On the theoretical side:

- A researcher trying to show that P is unequal to NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete one does
- Furthermore, a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal

On the practical side:

- The phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem
- Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, we believe that P is unequal to NP
- So proving that a problem is NP-complete is strong evidence of its nonpolynomiality

SAT

- The first NP-complete problem that we present is called the *satisfiability problem*
- Variables that can take on the values TRUE and FALSE are called Boolean variables
- We represent TRUE by 1 and FALSE by 0
- Boolean operations AND, OR, and NOT are represented by the symbols $\wedge, \vee, \neg$
- A Boolean formula is an expression involving Boolean variables and operations. For example: $\phi = (\neg x \wedge y) \vee (x \wedge \neg z)$
- A Boolean formula is *satisfiable* if some assignment of 0s and 1s to the variables makes the formula evaluate to 1
- The above formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1

SAT

The satisfiability problem is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{< \phi > \mid \phi \textit{ is a satisfiable boolean formula}\}$$

Theorem

$SAT \in P$ *iff* $P = NP$

Next, we study a method that is central to the proof of this theorem

Reducibility

- Previously we defined the concept of reducing one problem to another. When problem A reduces to problem B, a solution to B can be used to solve A
- Now we define a version of reducibility that takes the efficiency of computation into account
- When problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently

Reducibility

Definition

A function $f : \Sigma^* \rightarrow \Sigma^*$ is a **polynomial time computable function** if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input w

Definition

Language A is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language B, written $A \leq_P B$, if a polynomial time computable function $f : \Sigma^* \rightarrow \Sigma^*$ exists, where for every w

$$w \in A \iff f(w) \in B$$

The function f is called the **polynomial time reduction** of A to B

Reducibility

- Polynomial time reducibility is the efficient analog to mapping reducibility as defined previously
- Other forms of efficient reducibility are available, but polynomial time reducibility is a simple form that is adequate for our purposes
- As with an ordinary mapping reduction, a polynomial time reduction of A to B provides a way to convert membership testing in A to membership testing in B—but now the conversion is done efficiently
- To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$
- If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem

Reducibility

Theorem

If $A \leq_P B$ and $B \in P$, then $A \in P$

Proof.

Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows

$N =$ In input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus, M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial $\square$

3SAT

Now, we introduce $3SAT$, a special case of the satisfiability problem whereby all formulas are in a special form

- A *literal* is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$
- A *clause* is several literals connected with $\vee$s, as in $(x_1 \vee x_2 \vee x_3 \vee x_4)$
- A Boolean formula is in *conjunctive normal form*, called a *cnf-formula*, if it comprises several clauses connected with $\wedge$s
- It is a *3cnf-formula* if all the clauses have three literals

$$(x_1 \vee x_2 \vee x_3) \wedge (x_3 \vee x_5 \vee x_6) \wedge (x_3 \vee x_6 \vee x_4) \wedge (x_4 \vee x_5 \vee x_6)$$

3SAT

Let us consider the following language

$$3SAT = \{< \phi > \mid \phi \textit{ is a } 3cnf - formula\}$$

The next theorem presents a polynomial time reduction from the $3SAT$ problem to the $CLIQUE$ problem

Theorem

3SAT is polynomial time reducible to CLIQUE

3SAT to CLIQUE

Proof idea: The polynomial time reduction f that we demonstrate from $3SAT$ to $CLIQUE$ converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses

Proof.

Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

The reduction f generates the string $< G, k >$, where G is an undirected graph defined as follows

3SAT to CLIQUE

Proof.

The nodes in G are organized into k groups of three nodes each called the triples, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple, and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$.

For $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$ the next figure illustrates the construction $\square$

3SAT to CLIQUE

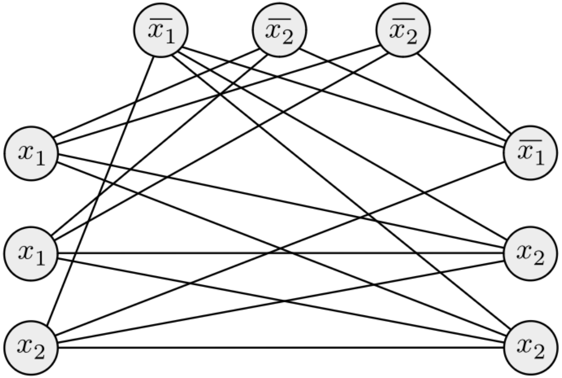

Figure: The graph that the reduction produces from ϕ

3SAT to CLIQUE

Proof.

Now we demonstrate why this construction works. We show that ϕ is satisfiable iff G has a *k-clique*.

Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a *k-clique*. The number of nodes selected is k because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore, G contains a *k-clique* $\square$

3SAT to CLIQUE

Proof.

Suppose that G has a *k-clique*. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore, each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal that is assigned $TRUE$. Therefore, ϕ is satisfiable □

NP-Completeness

- Previous theorems tell us that if *CLIQUE* is solvable in polynomial time, so is 3*SAT*
- At first glance, this connection between these two problems appears quite remarkable because, superficially, they are rather different.
- But polynomial time reducibility allows us to link their complexities.
- Now we turn to a definition that will allow us similarly to link the complexities of an entire class of problems

NP-Completeness

Definition

A language B is **NP-complete** if it satisfies two conditions:

- B is in NP
- every A in NP is polynomial time reducible to B

Theorem

If B is NP-complete and $B \in P$, then $P = NP$

Proof.

This theorem follows directly from the definition of polynomial time reducibility $\square$

NP-Completeness

Theorem

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete

Proof.

We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is *NP-complete*, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in NP is polynomial time reducible to C $\square$

SAT is NP-complete

The Cook-Levin theorem

Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it. However, establishing the first NP-complete problem is more difficult. Now we do so by proving that SAT is NP-complete.

Theorem

SAT is NP-complete

SAT is NP-complete

Proof idea: Showing that SAT is in NP is easy, and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to SAT.

To do so, we construct a polynomial time reduction for each language A in NP to SAT. The reduction for A takes a string w and produces a boolean formula ϕ that simulates the NP machine for A on input w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore, w is in A if and only if ϕ is satisfiable.

Actually constructing the reduction to work in this way is a conceptually simple task, though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn't surprising. The details are in the implementation of this idea.

SAT is NP-complete

Proof.

First, we show that *SAT* is in *NP*. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ

Next, we take any language A in *NP* and show that A is polynomial time reducible to *SAT*. Let N be a nondeterministic TM that decides A in n^k time for some constant k. For convenience, we actually assume that N runs in time $n^k - 3$; but only those students interested in details should worry about this minor point. The following notion helps to describe the reduction.

A tableau for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w, as shown in the next silde

SAT is NP-complete

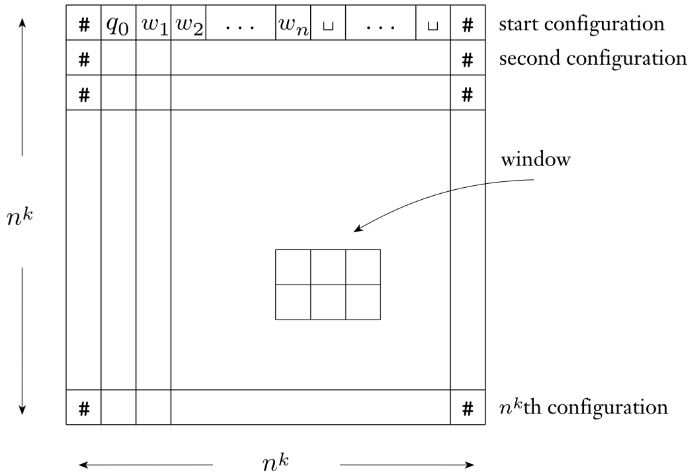

Figure: A tableau is an $n^k \times n^k$ table of configurations

SAT is NP-complete

Proof.

For convenience later, we assume that each configuration starts and ends with a # symbol. Therefore, the first and last columns of a tableau are all #s. The first row of the tableau is the starting configuration of N on w, and each row follows the previous one according to N's transition function. A tableau is accepting if any row of the tableau is an accepting configuration.

Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

SAT is NP-complete

Proof.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ. We begin by describing the variables of ϕ. Say that Q and Γ are the state set and tape alphabet of N, respectively. Let $C = Q \cup \Gamma \cup \{\#\}$. For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$

Each of the $(n^k)^2$ entries of a tableau is called a *cell*. The cell in row i and column j is called $\mathit{cell}[i,j]$ and contains a symbol from C. We represent the contents of the cells with the variables of ϕ. If $x_{i,j,s}$ takes on the value 1, it means that $\mathit{cell}[i,j]$ contains an s.

SAT is NP-complete

Proof.

Now we design ϕ so that a satisfying assignment to the variables does correspond to an accepting tableau for N on w. The formula ϕ is the AND of four parts and we describe each:

$$\phi_{cell} \wedge \phi_{start} \wedge \phi_{move} \wedge \phi_{accept}$$

As we mentioned previously, turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of boolean operations:

$$\phi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s \in C, s \neq t} \overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right]$$

SAT is NP-complete

Proof.

For example, the expression in the preceding formula:

$$\bigvee_{s \in C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s_1} \vee x_{i,j,s_1} \vee \ldots \vee x_{i,j,s_l}$$

where $C = \{s_1, s_2, \ldots, s_l\}$. Hence ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k.

SAT is NP-complete

Proof.

The first part of each fragment says that at least one variable is turned on in the corresponding cell. The second part of each fragment says that no more than one variable is turned on (literally, it says that in each pair of variables, at least one is turned off) in the corresponding cell. These fragments are connected by $\wedge$ operations.

The first part of ϕ_{cell} inside the brackets stipulates that at least one variable that is associated with each cell is on, whereas the second part stipulates that no more than one variable is on for each cell. Any assignment to the variables that satisfies ϕ (and therefore ϕ_{cell}) must have exactly one variable on for every cell. Thus, any satisfying assignment specifies one symbol in each cell of the table.

Parts ϕ_{start}, ϕ_{move}, and ϕ_{accept} ensure that these symbols actually correspond to an accepting tableau as follows.

SAT is NP-complete

Proof.

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{start} = & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\end{aligned}$$

SAT is NP-complete

Proof.

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau by stipulating that one of the corresponding variables is on:

$$q_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}$$

SAT is NP-complete

Proof.

Finally, formula ϕ_{move} guarantees that each row of the tableau corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal. We say that a 2×3 window is legal if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.

We could give a precise definition of legal window here, in terms of the transition function. But doing so is quite tedious and would distract us from the main thrust of the proof argument. Anyone desiring more precision should refer to the related analysis in the proof of theorem of the undecidability of the Post Correspondence Problem.

SAT is NP-complete

Proof.

For example, say that a, b, and c are members of the tape alphabet, and q_1 and q_2 are states of N. Assume that when in state q_1 with the head reading an a, N writes a b, stays in state q_1, and moves right; and that when in state q_1 with the head reading a b, N nondeterministically either

1. writes a c, enters q_2, and moves to the left, or
2. writes an a, enters q_2, and moves to the right

Expressed formally, $\delta(q_1, a) = \{(q_1, b, R)\}$ and $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$.

SAT is NP-complete

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Figure: Examples of legal windows

SAT is NP-complete

Proof.

Windows (a) and (b) are legal because the transition function allows N to move in the indicated way. Window (c) is legal because, with q_1 appearing on the right side of the top row, we don't know what symbol the head is over. That symbol could be an a, and q_1 might change it to a b and move to the right. That possibility would give rise to this window, so it doesn't violate N's rules.

Window (d) is obviously legal because the top and bottom are identical, which would occur if the head weren't adjacent to the location of the window. Note that $\#$ may appear on the left or right of both the top and bottom rows in a legal window. Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row, and it would then have moved to the left in state q_2 to appear on the right-hand end of the bottom row. Window (f) is legal because state q1 might have been immediately to the left of the top row, and it might have changed the b to a c and moved to the left. $\square$

SAT is NP-complete

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_2	a	a

(c)

b	q_1	b
q_2	b	q_2

Figure: Examples of illegal windows

Proof.

In window (a), the central symbol in the top row can't change because a state wasn't adjacent to it. Window (b) isn't legal because the transition function specifies that the b gets changed to a c but not to an a. Window (c) isn't legal because two states appear in the bottom row

SAT is NP-complete

Proof.

Next we want to prove that if the top row of the tableau is the start configuration and every window in the tableau is legal, each row of the tableau is a configuration that legally follows the preceding one.

We prove this claim by considering any two adjacent configurations in the tableau, called the upper configuration and the lower configuration. In the upper configuration, every cell that contains a tape symbol and isn't adjacent to a state symbol is the center top cell in a window whose top row contains no states. Therefore, that symbol must appear unchanged in the center bottom of the window. Hence it appears in the same position in the bottom configuration. The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function. Therefore, if the upper configuration is a legal configuration, so is the lower configuration, and the lower one follows the upper one according to N's rules.

SAT is NP-complete

Proof.

Now we return to the construction of ϕ_{move}. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula ϕ_{move} says that the settings of those six cells must be one of these ways, or

$$\phi_{move} = \bigwedge_{1 \leq i \leq n^k, 1 \leq j \leq n^k} (the\ (i,j) - window\ is\ legal)$$

SAT is NP-complete

Proof.

The $(i,j) - \textit{window}$ has $\textit{cell}[i,j]$ as the upper central position. We replace the text *the (i, j)-window is legal* in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{a_1,\ldots,a_6 \textit{ is a legal window}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

SAT is NP-complete

Proof.

Next, we analyze the complexity of the reduction to show that it operates in polynomial time. To do so, we examine the size of ϕ. First, we estimate the number of variables it has. Recall that the tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells. Each cell has l variables associated with it, where l is the number of symbols in C. Because l depends only on the TM N and not on the length of the input n, the total number of variables is $O(n^{2k})$.

We estimate the size of each of the parts of ϕ. Formula ϕ_{cell} contains a fixed-size fragment of the formula for each cell of the tableau, so its size is $O(n^{2k})$. Formula ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$. Formulas ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell of the tableau, so their size is $O(n^{2k})$. Thus, ϕ's total size is $O(n^{2k})$.

SAT is NP-complete

Proof.

That bound is sufficient for our purposes because it shows that the size of ϕ is polynomial in n. If it were more than polynomial, the reduction wouldn't have any chance of generating it in polynomial time. Actually, our estimates are low by a factor of $O(logn)$ because each variable has indices that can range up to n^k and so may require $O(logn)$ symbols to write into the formula, but this additional factor doesn't change the polynomiality of the result.

To see that we can generate the formula in polynomial time, observe its highly repetitive nature. Each component of the formula is composed many nearly identical fragments which differ only at the indices in a simple way. Therefore, we may easily construct a reduction that produces ϕ in polynomial time from the input w. □

SAT is NP-complete

- We have concluded the proof of the Cook–Levin theorem, showing that SAT is NP-complete. Showing the NP-completeness of other languages generally doesn't require such a lengthy proof
- Instead, NP-completeness can be proved with a polynomial time reduction from a language that is already known to be NP-complete.
- We can use SAT for this purpose; but using 3SAT, the special case of SAT, is usually easier

3SAT is NP-complete

Corollary

3SAT is NP-complete

Proof.

Convert SAT formula to CNF. Next, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original

For example, we replace clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$

3SAT is NP-complete

Proof.

In general, if a clause has l literlas:

$$a_1 \vee a_2 \vee \ldots \vee a_l$$

we can replace it with $l - 2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l)$$

$\square$

Section 4

Examination

Examination

- When: 22 Jan, 10 AM
- Where: see FMI website
- What: You will receive one of the following two subjects:
    1. Undecidability
    2. NP-Completeness
- How: pen&paper no auxiliary materials

Subject 1

Undecidability: Lecture 5 and 6 or Chapter 4.2 (pages 201 - 209) from Sipser (3rd edition)

1. What is undecidability?
2. The diagonalization method
3. (Un)Countable sets
4. $\mathcal{Q}$ is countable
5. $\mathcal{R}$ is uncountable
6. Some languages are not Turing-recognizable
7. A_{TM} is undecidable

Subject 2

NP-Completeness: Lecture 11 and 12 or Chapter 7.4 (pages 299 - 310) from Sipser (3rd edition)

1. Satisfiable boolean formula
2. Polinomial time reducibility
3. (3)CNF-formula
4. NP-complete
5. Cook-Levin theorem
6. 3SAT is NP-complete