# ECS 120 Lesson 25 – The Cook–Levin Theorem, Polynomial Reduction Proofs

Oliver Kreylos

Wednesday, May 30th, 2001

Last time, we looked at $\mathcal{NP}$, the class of problems solvable by nondeterministic Turing Machines in polynomial time, and at the "most difficult" problems in $\mathcal{NP}$ – the $\mathcal{NP}$-complete problems. Since the $\mathcal{NP}$-complete problems are those we generally classify as intractable, determining whether a given problem $P$ is $\mathcal{NP}$-complete or not is an important question. This can be done in two ways: Either by reducing *all* problems in $\mathcal{NP}$ to $P$, or by reducing *a single* $\mathcal{NP}$-complete problem to $P$. The second way is usually much easier, but it depends on a problem that we already know is $\mathcal{NP}$-complete. This first problem has to be proven the hard way.

## 1 The Cook–Levin Theorem

The first problem was proven to be $\mathcal{NP}$-complete independently by Steve Cook and Leonid Levin in 1971/1973. The so-called *satisfiability problem* asks whether a given boolean formula can be satisfied, i. e., whether there exists a setting of its variables to the values true/false such that the result of the formula is true. As a language, the problem is stated as

$$\text{SAT} = \left\{ \langle \phi \rangle \;\middle|\; \phi \text{ is a satisfiable boolean formula} \right\}$$

A boolean formula is just a formula involving variables that can be either true or false, which are combined by the logical operations and ($\wedge$), or ($\vee$) or not ($\neg$). For our discussion, we will assume that all variables are named $x_1, x_2, x_3, \ldots$ Examples for boolean formulas are $(x_1 \wedge x_2) \vee x_3$, $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_3)$, or $\neg(x_1 \wedge \neg x_2) \wedge \neg x_4$.

## 1.1 Boolean Formulas

To be precise, and to apply our knowledge about context-free grammars, let us define the exact notion of a boolean formula by a grammar over the alphabet $\Sigma = \{\wedge, \vee, \neg, (, ), x\}$. The grammar is defined by the rules

$$\begin{aligned} D &\rightarrow D \vee C \mid C \\ C &\rightarrow C \wedge A \mid A \\ A &\rightarrow \neg A | (D) \mid X \\ X &\rightarrow xX \mid x \end{aligned}$$

where $D$ is the start symbol. Examples for formulas according to these rules are $x \vee \neg x$ or $(x \vee xx) \wedge \neg xxx$. All variables generated from this grammar are of the form $xx \ldots x$; to make the format more readable we replace a variable $x^k$ consisting of $k$ $x$s by the symbol $x_k$. Then the above example formulas become $x_1 \vee \neg x_1$ and $(x_1 \vee x_2) \wedge \neg x_3$, respectively. In other words, our grammar encodes the index $k$ of a variable in unary by replicating its symbol $k$ times. Not very efficient, but simple.

## 1.2 $\mathcal{NP}$-completeness of SAT

To show that SAT is $\mathcal{NP}$-complete, we first have to show that it is itself in $\mathcal{NP}$. This is the easy part: A nondeterministic Turing Machine can "guess" the correct settings for all the variables in $\phi$. To verify the guess, the machine then evaluates $\phi$ on that setting, which can be done in polynomial time. The hard part is to show that *every other* problem in $\mathcal{NP}$ can be reduced to SAT in polynomial time. In doing so, we cannot assume anything specific about the problems we have to reduce from; so the proof must proceed in a very abstract manner.

In fact, the only thing that all problems in $\mathcal{NP}$ have in common is that, by definition, they are decided by some nondeterministic Turing Machine $N$ in polynomial time, say $O(n^k)$ for some $k$. And this is exactly the basis for the proof: Given machine $N$ and an input word $w$ of length $|w| = n$, we construct a boolean formula that is satisfiable exactly if $N$ accepts word $w$. If this construction exists, then the question whether $w$ is in the language defined by machine $N$ can be decided by constructing the formula, and then running the decider for SAT to give the correct answer. Furthermore, if the boolean formula can be constructed in polynomial time, then we can reduce

any problem $P \in \mathcal{NP}$ to SAT – the only assumption we made about $P$ was that is is in $\mathcal{NP}$.

The reduction itself is very similar to the one we did to show the Post Correspondence Problem undecidable: We will construct a boolean formula that "simulates" the computation of a nondeterministic Turing Machine. First, we have to make some observations: If $N$ is a Turing Machine as defined above, its time complexity must be $O(n^k)$ for some $k$. This means that in any branch of computation, $N$ performs at most $O(n^k)$ steps on any input word of length $n$. It also means that the longest possible portion of its tape that is ever used in any branch is at most $n^k$ characters long – a Turing Machine can only write one character in each step. Altogether, this means that any computation of $N$ on a word of length $n$ can be encoded as a rectangular *tableau* of $n^k \times n^k$ characters. Each row of the tableau will be a configuration of $N$, encoded in the usual way, and the sequence of configurations down the rows will be a valid computation chain of $N$ on word $w$. An example configuration for machine $N$ and word $w = w_1 w_2 \dots w_n$ is shown in Table 1.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| # | $q_0$ | $w_1$ | $w_2$ | $w_3$ | $\cdots$ | $w_n$ | $\sqcup$ | $\cdots$ | $\sqcup$ | # | $\mathrm{ID}_0$ |
| # | | | | | | | | | | # | $\mathrm{ID}_1$ |
| # | | | | | | | | | | # | $\mathrm{ID}_2$ |
| | | | | | $\vdots$ | | | | | | |
| | | | | | $\vdots$ | | | | | | |
| | | | | | $\vdots$ | | | | | | |
| | | | | | $\vdots$ | | | | | | |
| # | $u_1$ | $u_2$ | $\cdots$ | $u_i$ | $q_{\text{accept}}$ | $v_1$ | $v_2$ | $\cdots$ | $v_j$ | # | $\mathrm{ID}_{n^k}$ |

Table 1: A tableau for computation of a Turing Machine $N$ on an input word $w = w_1 w_2 \dots w_n$. If $N$ runs in time $O(n^k)$, then the tableau is of size $n^k \times n^k$.

We now proceed to construct a boolean formula that is satisfiable exactly if there exists a tableau for $N$ and input word $w$ such that the final row is an accepting configuration. This means that the constructed formula will be satisfiable if and only if $N$ accepts $w$, as required for the reduction. The idea in constructing this formula is to build it in such a way that it checks the computation chain represented by the tableau for validity; this is similar

to the idea of computation chain reductions we used to show the emptiness problem for CH–1 languages undecidable.

A tableau is a rectangular array of cells, where each cell contains exactly one symbols from the set $C = Q \cup \Gamma \cup \{\#\}$ consisting of all state symbols and tape characters of $N$, and the special separator #. To convert this array into a boolean formula, we must express the assignment of symbols to cells using boolean variables: For each tableau cell $c_{i,j}$ we create a set of variables $x_{i,j,s}$ for each symbol $s \in C$. The idea is to set $x_{i,j,s}$ to true exactly if cell $c_{i,j}$ contains symbol $s$.

We can now desribe the validity of the computation chain represented by the tableau as a boolean formula involving the variables $x_{i,j,s}$. We split the formula into four parts, each adressing one aspect of validity:

1. In the tableau, each cell contains exactly one symbol from $C$. If the variables $x_{i,j,s}$ are to represent a valid tableau, exactly one of the variables $x_{i,j,s}$ for each pair $(i, j)$ must be true. We enforce this by creating a boolean formula $\phi_{\text{cell}}$.

2. The first configuration of any computation chain must be an initial configuration. We enforce this using a formula $\phi_{\text{initial}}$.

3. The last configuration of an accepting computation must be an accepting configuration. We enforce this by a formula $\phi_{\text{accept}}$. To make the formula simpler, we assume that all tableaux for $N$ and $w$ contain exactly $n^k$ configurations; if $N$ performs less steps, we just replicate the last configuration of $N$ for the rest of the rows.

4. Every configuration in the tableau must be related to the one above it by $N$'s turnstile relation. We enforce this by a formula $\phi_{\text{move}}$.

Before we look at the four subformulas in more detail, it is important to note that the actual tableaux for computation of $N$ on $w$ are never constructed. The formula $\phi$ we build is merely a "verifier" for a given tableau. Instead of constructing tableaux and verifying them by evaluating $\phi$, we feed $\phi$ itself into the decider for the satisfiability problem; if $\phi$ is satisfiable, then there must be some satisfying setting of its variables, i. e., there must be some valid tableau corresponding to an accepting configuration chain in $N$. In this way, the tableaux are only an intermediate representation, but never actually constructed.

### 1.2.1 The Formula $\phi_{\text{cell}}$

For a setting of boolean variables $x_{i,j,s}$ to be a representation of a tableau, of all variables $x_{i,j,s}$ corresponding to a cell $c_{i,j}$ exactly one must be true. Remember that, by our reduction construction, variable $x_{i,j,s}$ is set to true if and only if cell $c_{i,j}$ contains symbol $s$. Since each cell contains exactly one symbol, exactly one variable in each group must be true. Formula $\phi_{\text{cell}}$ is given by

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left( \left( \bigvee_{s \in C} x_{i,j,s} \right) \wedge \left( \bigwedge_{\substack{s,t \in C \\ s \neq t}} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right)$$

The first part of the formula, the big disjunction, expresses that in each cell at least one variable must be true (each cell must contain at least one symbol); the second part, the big conjunction, expresses that no two variables can be true; in other words, each cell must contain at most one symbol.

### 1.2.2 The Formula $\phi_{\text{initial}}$

The initial configuration of $N$ on word $w$ is always $q_0\ w_1 w_2 \ldots w_n$, followed by any number of blanks. Thus, the formula $\phi_{\text{initial}}$ merely expresses which symbols are written into the first row of the tableau:

$$\phi_{\text{initial}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$

This formula expresses that the first cell must contain the symbol `#`, the second must contain the symbol $q_0$, etc.

### 1.2.3 The formula $\phi_{\text{accept}}$

An accepting configuration for $N$ consists of any left tape string, the accept state $q_{\text{accept}}$, and any right tape string. Therefore, by our definition, formula $\phi_{\text{accept}}$ must express that the symbol $q_{\text{accept}}$ must occur somewhere in the last row of the tableau:

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

### 1.2.4 The Formula $\phi_{\text{move}}$

The final formula, which expresses that every row in a valid tableau is related to the one above it by $N$'s turnstile relation, is the most complicated in the bunch. The idea to construct it is very similar to the creation of the dominos related to a machine's turnstile relation we saw in the undecidability proof for the Post Correspondence Problem. The basic observation is that a single computation step of a Turing Machine can change a configuration only locally, in the region around the current tape head position. By exploiting this fact, we can express the possibly infinite number of computation chains using only a finite set of rules. The rules making up $\phi_{\text{move}}$ are based on *windows*, $3 \times 2$ arrays of cells located anywhere in a tableau. A window shows how a three-symbol substring of one table row can be converted into a three-symbol substring in the next table row. Instead of giving all the details of how to construct the set of windows, we will only look at some examples of *legal windows* that can be parts of valid tableaux.

Let us assume that the transition function for $N$ contains the following mapping pairs: $(q_1, \mathtt{b}) \mapsto \{(q_2, \mathtt{c}, \mathrm{R})\}$ and $(q_2, \mathtt{b}) \mapsto \{(q_3, \mathtt{c}, \mathrm{L})\}$. Then a configuration containing the substring $\mathtt{a}\ q_1\ \mathtt{b}$ will change into a configuration containing the substring $\mathtt{ac}\ q_2$ at the same position. Similarly, a configuration containing the substring $\mathtt{a}\ q_2\ \mathtt{b}$ will change into a configuration containing the substring $q_3\ \mathtt{ac}$. The windows for these transitions are depicted in parts (a) and (b) in Table 2. Other windows involving the tape head are constructed from $N$'s transition function in the same way (an example is shown in part (c) of Table 2), and all windows that do not involve the tape head are constructed by creating every possible string of three symbols from $C$ and replicating it for both window rows – if the tape head is nowhere near, the tape contents must remain the same from one configuration to the next. An example for this kind of window is shown in part (d) of Table 2.

(a)

| | | |
|---|---|---|
| a | $q_1$ | b |
| a | c | $q_2$ |

(b)

| | | |
|---|---|---|
| a | $q_2$ | b |
| $q_3$ | a | c |

(c)

| | | |
|---|---|---|
| d | a | $q_1$ |
| d | a | c |

(d)

| | | |
|---|---|---|
| a | b | c |
| a | b | c |

Table 2: Four legal windows for a Turing Machine $N$. In part (c), the symbol to the right of $q_1$ in the upper row must be a b, this would be enforced by the window to the right of the shown one.

The formula $\phi_{\text{move}}$ contains a subformula for each possible window in the tableau, i. e., for all windows covering the cells $c_{i-1,j-1}$, $c_{i-1,j}$, $c_{i-1,j+1}$,

$c_{i,j-1}$, $c_{i,j}$ and $c_{i,j+1}$, for every $1 < i \leq n^k$, and every $1 \leq j \leq n^k$. Each of these subformulas is the disjunction of all the formulas describing the legal windows of $N$. For example, the formula expressing that the window centered at cell $c_{4,5}$ is the window shown in part (a) of Table 2 is:

$$\phi_{i,j,(\mathrm{a})} = x_{3,4,\mathtt{a}} \wedge x_{3,5,q_1} \wedge x_{3,6,\mathtt{b}} \wedge x_{4,4,\mathtt{a}} \wedge x_{4,5,\mathtt{c}} \wedge x_{4,6,q_2}$$

Overall, formula $\phi_{\text{move}}$ looks like:

$$\phi_{\text{move}} = \bigwedge_{1<i\leq n^k, 1\leq j\leq n^k} \Big( \bigvee_{\text{all cases } (c)} \phi_{i,j,(c)} \Big)$$

Where the $(c)$ refer to all cases of legal windows as described above. Finally, the formula

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{initial}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$$

expresses that a tableau, represented as a setting of boolean variables $x_{i,j,s}$, corresponds to an accepting branch of computation in Turing Machine $N$. Thus, the construction given here is a reduction from any $\mathcal{NP}$-complete problem, represented by a nondeterministic Turing Machine $N$, to the satisfiability problem for boolean formulas.

The only part left to prove is that the formula $\phi$ can be constructed in time polynomial in the length of the input word $w$. This is easy to see: The total size of any tableau is $n^k \times n^k$ cells, which are $n^{2k}$ cells in total. Each partial formula (except $\phi_{\text{initial}}$) consists of $O(n^{2k})$ subformulas, where each subformula can be constructed in time independent of the length of the input word – the subformulas only depend on $N$, not on $w$. We have seen that $\phi_{\text{initial}}$ is very simple and can also be constructed in polynomial time; therefore, the reduction is indeed polynomial.

This concludes our outline of the proof that SAT is $\mathcal{NP}$-complete. From now on, we can show problems to be $\mathcal{NP}$-complete by reducing them from a known $\mathcal{NP}$-complete problem, e. g., from SAT. As is turns out, it is often easier to reduce from a simplified version of SAT, which is also $\mathcal{NP}$-complete.

# 2 The Problem 3SAT

3SAT is a special case of SAT, in that it asks about the satisfiability of boolean formulas in a special format, *conjunctive normal form (CNF)*. In CNF (not to be confused with Chomsky Normal Form), a formula is a single

conjunction of *clauses*, i. e., sets of disjunctions, where each clause only consists of a set of *literals*, i. e., variables or negated variables. An example of a formula in CNF is $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_3)$. More generally, the format of formulas $\phi$ in CNF is

$$\phi = \bigwedge_{1 \le i \le n} \Big( \bigvee_{1 \le j \le m_i} L_{i,j} \Big), \quad \text{where } L_{i,j} \text{ is either } x_{i,j} \text{ or } \neg x_{i,j}$$

3SAT restricts the formulas even more by requiring that each clause contains exactly three literals. As it turns out, each general boolean formula can be converted into 3–CNF, i. e., CNF with exactly three literals per clause; so 3SAT is not actually a restriction of SAT after all. The proof for the $\mathcal{NP}$-completeness of 3SAT exploits this fact: Instead of reducing SAT to 3SAT, we adapt the proof of the Cook–Levin theorem to directly generate formula $\phi$ in 3–CNF. The $\mathcal{NP}$-completeness of 3SAT, proved in this way, has been used to show the $\mathcal{NP}$-completeness of a wide variety of seemingly unrelated problems.

# 3 The Problem VERTEX-COVER

One problem that can be shown $\mathcal{NP}$-complete by reduction from 3–SAT is the question whether a given graph has a *vertex cover* of at most $k$ nodes. A vertex cover of a graph is a subset $C \subset V$ of nodes, such that every edge in the graph is touched by one of the nodes in $C$. As an example, consider the graph shown in Figure 1; a vertex cover consisting of four nodes is indicated by the shaded nodes.

Just constructing any vertex cover is easy; one can be found by just selecting all vertices in the graph. More tricky is the task to find the *minimal vertex cover*, i. e., the one that consists of the minimum number of vertices. For example, the graph in Figure 1 contains another vertex cover consisting of only three vertices. For larger graphs, these are extremely difficult to find; in fact, the problem VERTEX-COVER is $\mathcal{NP}$-complete.

The proof for this claim works by reduction from 3–SAT. How can these seemingly unrelated problems, one about boolean formulas, the other one about graphs, be reduced to each other? Typically, in proofs like this one has to come up with a conversion construction, typically called "gadgets." Finding these gadgets is an act of creativity, but once they are found, the proofs usually are straightforward. For this specific reduction, we need two

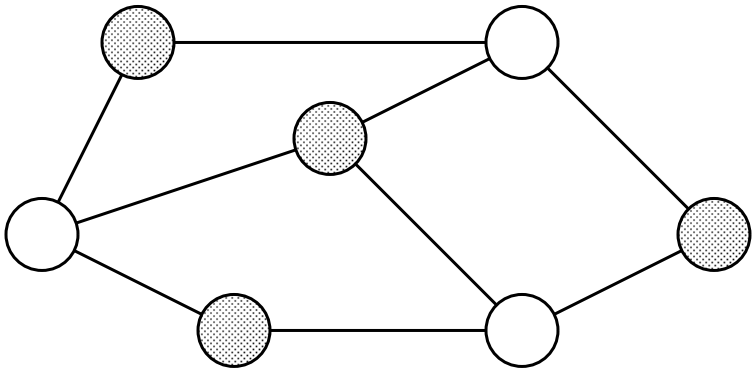

Figure 1: A graph and a vertex cover consisting of four nodes, indicated by the shaded nodes.

kinds of gadgets: One representing the variables of a formula in 3–CNF, and the other one representing the clauses. The variable gadget consists of two nodes, one representing a variable $x_i$ and the other one its complement $\neg x_i$, and a single edge connecting the two, see Figure 2, part (a). The clause gadget consists of three nodes, each one representing one literal from the clause, and three edges connecting the nodes, see Figure 2, part (b).

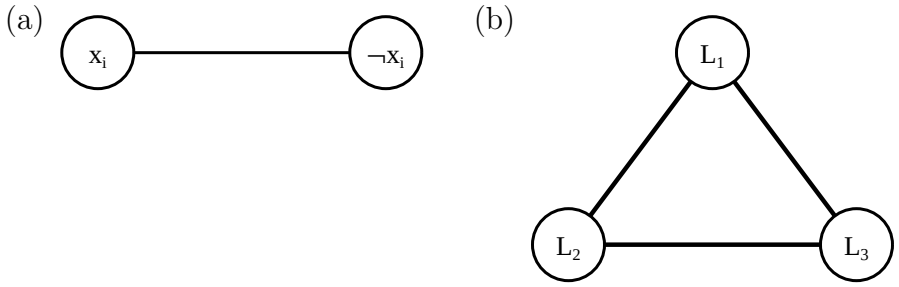


Figure 2: Gadgets used in the reduction from 3–SAT to VERTEX-COVER. (a) the variable gadget, (b) the clause gadget.

After constructing a variable gadget for each variable of $\phi$, and a clause gadget for each clause of $\phi$, the gadgets are connected to each other by adding an edge from each node in each clause gadget to the node labeled with the same literal. For example, the formula $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ is converted to the graph shown in Figure 3.

Let $n$ denote the number of variables in $\phi$, and let $m$ be the number of clauses. Then we claim that $\phi$ is satisfiable if and only if the constructed

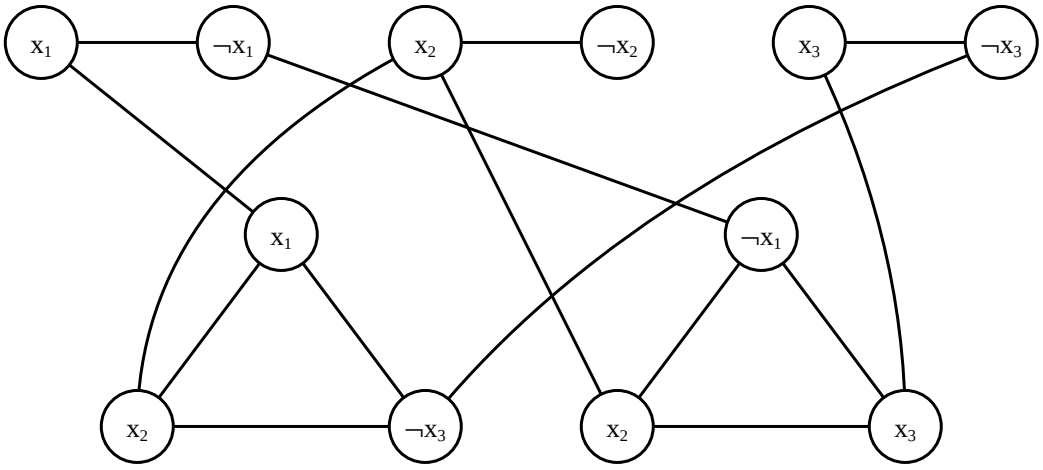


Figure 3: Graph constructed from the formula $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

graph has a vertex cover of at most $n + 2m$ nodes. To prove this claim, let us start with a satisfiable formula $\phi$. We then construct a vertex cover by including one node of each variable gadget and two nodes of each clause gadget: If $x_i$ is set to true in the satisfying assignment for $\phi$, the node labeled with $x_i$ is included; otherwise, the node labeled $\neg x_i$ is taken. In every clause of $\phi$, at least one literal must be true; we pick one of those and include the other two nodes of each clause gadget into the vertex cover. All edges in the variable and clause gadgets must be touched by selected nodes by construction; what about the edges connecting the two kinds of gadgets? For each clause gadgets, two of the edges going to variable gadgets are covered by the two nodes selected from the gadget; the other one must be touched by a node selected from a variable gadget, because the non-included node from each clause gadget corresponds to a literal set to true, which means the node corresponding to that literal in the variable gadgets is included. Altogether, we have a vertex cover consisting of $n + 2m$ nodes.

To prove the other way, we assume the constructed graph has a vertex cover of at most $n + 2m$ nodes. To cover all edges inside the gadgets, one node from each variable gadget and two nodes from each clause gadget must be included in the vertex cover. This accounts for all $n+2m$ nodes. To cover all edges between the two kinds of gadgets, the edge coming from the non-included node in each clause gadget must go to a selected variable gadget

node. If we set the variables of $\phi$ according to which node from each variable gadget is selected, i. e., $x_i$ is true if and only if the node corresponding to $x_i$ is included, then that assignment must satisfy $\phi$: In each clause gadget, one literal node is connected to an included literal node, which means that the literal is true. Thus, each clause has at least one true literal in it, which satisfies each clause and also the whole formula.

This way, we can convert an entity of 3–SAT into an entity of VERTEX-COVER, such that the formula $\phi$ is satisfiable if and only if the constructed graph has a vertex cover of at most $k = n + 2m$ nodes. Using the described construction, the graph can be built in polynomial time; therefore, the reduction shows that VERTEX-COVER is $\mathcal{NP}$-complete.