Chapter 16

Polynomial-Time Reducibility and $\mathcal{NP}$-Completeness

In the previous chapter, we have introduced the $\mathcal{P} = \mathcal{NP}$ problem as one of the most important problems in theoretical Computer Science. A very important breakthrough in solving this problem came in the early 1970s with the work of Stephen Cook and Leonid Levin. They discovered a certain class of problems in $\mathcal{NP}$ whose complexity is related to the complexity of the entire class $\mathcal{NP}$. This is the class of $\mathcal{NP}$-*Complete* problems, and they have an amazing feature that if there is a polynomial time algorithm (Turing machine) for *any* problem in the class of $\mathcal{NP}$-Complete problems, then there is a polynomial time algorithm for *every* problem in $\mathcal{NP}$. $\mathcal{NP}$-Complete problems are, in a sense, the hardest problems in the class $\mathcal{NP}$, as efficient solution to any of them would lead to efficient solutions to all $\mathcal{NP}$ problems.

At first sight, it may seem that any $\mathcal{NP}$-Complete problem must be extraordinary and completely artificial. However, this is not the case - there are many realistic problems that are $\mathcal{NP}$-Complete. We will see, in the course of our discussion, some problems that look reasonably simple, but which are proven to be in the $\mathcal{NP}$-Complete class.

Discovery of $\mathcal{NP}$-Complete problems has a great impact on the $\mathcal{P} = \mathcal{NP}$ problem. To prove that $\mathcal{P} = \mathcal{NP}$, we would "only" need to find a polynomial-time solution for *one* $\mathcal{NP}$-Complete problem, and we are free to choose any of them for this purpose. To prove that $\mathcal{P} \neq \mathcal{NP}$, we would still "only" need to prove that one problem from the $\mathcal{NP}$ class does not have a polynomial-time solution. Of course, the “only” part in both previous sentences is immensely hard, since no one so far has managed to do it, and the problem has been studied in great depth by many scientists for the last 50 years.

There are also more practical implications of the existence of the $\mathcal{NP}$-Complete class. If we encounter a problem that is $\mathcal{NP}$-Complete, we don’t need to waste time searching for polynomial-time solution to it, because it probably does not exist, since there is strong belief that the classes $\mathcal{P}$ and $\mathcal{NP}$ are not the same (but this has not been proven yet). Therefore, proving that a problem is $\mathcal{NP}$-Complete is a strong evidence of its nonpolynomiality. Therefore, when we encounter such a problem, we can turn our attention to developing efficient heuristics that work fast in most of the cases, or developing fast approximate algorithms for solving them, instead of searching for an efficient and precise solution.

16.1 Polynomial-Time Reducibility

We have already seen a concept of reducing one problem to another when we studied decidability. If we have problems A and B and we are able to transform every instance of the problem A to an instance of the problem B, then:

- If problem B is *decidable*, the problem A is *decidable* too.
- If problem A is *undecidable*, then problem B is *undecidable* too.

We have used this reasoning, for example, to conclude that L_u is undecidable, based on the fact that L_d was undecidable. We also used it to conclude that L_{halt} is undecidable, based on the fact that L_u was undecidable. A thing to note in the above reasoning is that if B is *undecidable*, we don’t know for sure that A is undecidable too. Here we extend this notion to *efficient* reducibility - if problem A is *efficiently* reducible to B and if there is an efficient (polynomial-time) solution to B, then there is also an efficient solution to A.

But before introducing the notion of efficient reducibility, we need to define what do we mean by an "efficient transformation". For that, we need the notion of an efficient (i.e. polynomial-time computable) function.

Definition 16.1: Polynomial-Time Computable Function

A function $f : \Sigma^* \to \Sigma^*$ is a *polynomial-time computable function* if some polynomial-time Turing machine M exists that, for any input string $w \in \Sigma^*$, halts with just $f(w)$ on its tape.

So far, we have mostly used Turing Machines as language recognisers, so we haven't seen too many polynomial-time computable functions. But most of the functions on strings that we deal with are polynomial-time computable. For example, a function that duplicates a string (that is, a function $f(w) = ww$ defined on strings w over some alphabet) is a polynomial-time computable, as we can easily find a Turing Machine that starts with w written on its tape, ends with ww on the tape and runs in polynomial-time with respect to the length of string w.

Definition 16.2: Polynomial-Time Reducibility

Language A is *polynomial-time mapping reducible* or simply *polynomial-time reducible* to language B, written $A \overset{f}{\propto} B$, if there exist a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$, such that

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

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

We will most often write simply $A \propto B$ instead of $A \overset{f}{\propto} B$, because we will just care that a polynomial-time reduction exists, and not what this polynomial-time reduction is. $A \propto B$ means that every string from A can efficiently be converted to a string from B. We will use the polynomial-time reducibility to efficiently convert (reduce) an (encoding of an) instance of one problem to an (encoding of an) instance of another problem. Note also that $\propto$ is not necessarily symmetric - $A \propto B$ does not always imply $B \propto A$.

"Regular" reducibility, as we have considered in decidability, allowed us to reduce testing of membership of a string in language A to testing membership of a string in language B. Polynomial-time reducibility means that this reduction can be done *efficiently*. To test whether $w \in A$, we efficiently convert w to $f(w)$, using some polynomial-time reduction f, and then test whether $f(w) \in B$. The following important theorem holds.

Theorem 16.1

If $A \overset{f}{\propto} B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.

Proof. Assume that $A \overset{f}{\propto} B$ and $B \in \mathcal{P}$. Since $B \in \mathcal{P}$, there exists a Turing machine M such that M decides (solves) B in polynomial-time $P(n)$, where P is some polynomial of n. Since $A \overset{f}{\propto} B$, then there is a Turing Machine F that computes f such that $w \in A \iff f(w) \in B$. Furthermore, $f(w)$ is computed in polynomial-time $Q(n)$, where n is the size of w. Furthermore, if the size of w is n, then the size of $f(w)$ is at most $Q(n)$. We haven't dealt much with space complexity of Turing Machines in this module, but it is obvious that a Turing Machine that has time complexity $Q(n)$ can visit at most $Q(n)$ different cells, at most one in every step, therefore that Turing Machine has space complexity $Q(n)$, where space complexity is the maximum number of cells used in computation of the Turing Machine for a string of a given size.

We can now construct a TM M' that will decide whether some $w \in A$ by first reducing w to $f(w)$ and then running M on $f(w)$ to decide whether $f(w) \in B$. Testing whether $f(w) \in B$ can be done in at most $P(Q(n))$ time, where n is the size of w. This because $f(w)$ is of size at most $Q(n)$, therefore it can be decides in time $P(Q(n))$. Therefore, for the input string w of size n, M' decides whether $w \in A$ in $Q(n) + P(Q(n))$ time, which is a polynomial time, since $P(n)$ and $Q(n)$ are polynomials, and composition of polynomials is a polynomial. □

Language/problem A being polynomial-time reducible to problem B can be interpreted as B being at least as hard as A. Another important theorem states the transitivity of the polynomial-time reducibility relation.

Theorem 16.2: Transitivity of Polynomial-Time Reducibility

If $A \propto B$ and $B \propto C$ then $A \propto C$.

Proof. If $A \propto B$, then there exists a polynomial-time computable function f such that $w \in A \iff f(w) \in B$. If $B \propto C$, then there exists a polynomial-time computable function g such that $v \in B \iff g(v) \in C$. Then $h = g \circ f : \Sigma^* \to \Sigma^*$ defined by $h(w) = (g \circ f)(w) = g(f(w))$ is a polynomial-time computable function and

$$w \in A \iff f(w) \in B \iff g(f(w)) = h(w) \in C.$$

Therefore, we have found a polynomial-time computable function h such that $w \in A \iff h(w) \in C$, which means that $A \propto C$. □

16.2 SAT, 3SAT and CLIQUE Problem

We will now see an example of how one problem can be reduced in polynomial time to a seemingly completely different problem. In the process, we will introduce three very important problems: SAT, 3SAT and CLIQUE. SAT is the first problem for which it was proven that it is $\mathcal{NP}$-Complete.

16.2.1 SAT Problem

A variable that can take only values TRUE or FALSE is called a *boolean variable*. We usually represent TRUE by 1 and FALSE by 0. The binary *Boolean operations* AND, OR and NOT, denoted by $\wedge$, $\vee$ and $\neg$ are given by

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \neg 0 = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \neg 1 = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

Note that $\neg x$ is usually denoted by $\overline{x}$. A *boolean formula* is an expression involving Boolean variables and operations. Some examples of Boolean formulas are

- $(\overline{x} \wedge y) \vee (x \wedge z)$
- $(x \wedge \overline{x}) \wedge y$
- $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_2 \vee x_3)$

A boolean formula is *satisfiable* if there exists an assignment of 0s and 1s to the variables that make the formula evaluate to 1. For example,

- $(\overline{x} \wedge y) \vee (x \wedge z)$ is satisfiable for $x = 0, y = 1, z = 0$.
- $(x \wedge \overline{x}) \wedge y$ is not satisfiable.
- $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_2 \vee x_3)$ is satisfiable for $x_2 = 1$ and any values for x_1 and x_3.

With all this in mind, we can now define the SAT problem as follows.

Problem 16.1: SAT Problem

The *satisfiability problem* (or *SAT* problem) is to test whether a Boolean formula is satisfiable. Formally, language SAT is defined as

$$\text{SAT} = \{\langle \phi \rangle | \phi \text{ is a satisfiable Boolean formula}\}.$$

16.2.2 3SAT

3SAT problem is similar to SAT problem, except that the Boolean formula has to be in a specific format.

A *literal* is a Boolean variable or a negated boolean variable (as in x or $\overline{x}$). A *clause* comprises of literals connected with $\vee$'s, e.g. $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A *Boolean formula* is in *conjuctive normal form*, also called a *cnf-formula*, if it comprises clauses connected by $\wedge$'s, e.g.

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_5 \vee x_6) \wedge (\overline{x_1} \vee x_5 \vee \overline{x_6}).$$

A cnf-formula is a *3cnf-formula* if all the clauses have exactly 3 literals, e.g.

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_4 \vee x_5) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_5}).$$

Problem 16.2: 3SAT Problem

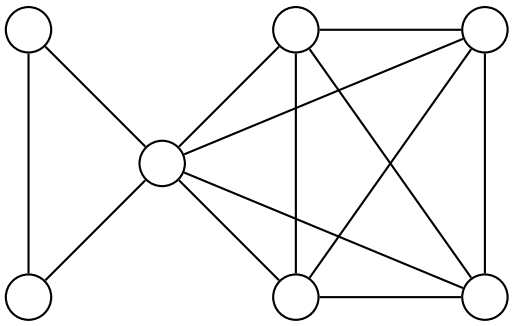
The 3SAT problem) is to test whether a 3cnf-formula is satisfiable. Formally, language 3SAT is defined as

$$\text{3SAT} = \{\langle\phi\rangle | \phi \text{ is a satisfiable 3cnf-formula}\}.$$

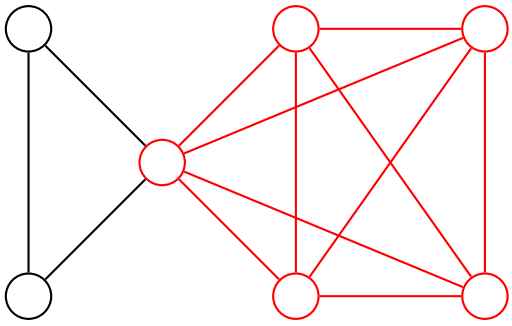
16.2.3 CLIQUE

Let G be an undirected graph (a graph where if there exists an edge from v_1 to v_2, then there also exists an edge from v_2 to v_1). $G' = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if $V' \subset V$ and E' comprises all the edges from E such that their both ends are in V'. A *clique* in a graph $G = (V, E)$ is a subgraph G' such that every two nodes from it are connected by an edge. A k-*clique* is a clique that contains k nodes.

Take, as an example, a graph from the picture below.



The nodes and edges of one 5-clique in this graph are highlighted in red in the picture below.



The CLIQUE problem now can be posed as follows.

Problem 16.3: CLIQUE Problem

The *CLIQUE Problem* is to determine whether a graph contains a clique of a specified size. Or, in the terms of languages,

$$\text{CLIQUE} = \{\langle G, k\rangle | G \text{ is an undirected graph with } k\text{-clique}\}.$$

16.2.4 Reducing 3SAT to CLIQUE

At first, 3SAT and CLIQUE look like two completely different, unrelated problems. After all, the former deals with satisfiability of logical formulas and the latter with searching for a particular subgraph of a graph. It is not obvious at all how these two problems could be related, and how solution of one of them could be used to find a solution of the other. However, the following theorem holds.

Theorem 16.3: 3SAT Is Polynomial-Time Reducible to CLIQUE

3SAT is polynomial-time reducible to CLIQUE.

Proof. We need to show that we can reduce every instance ϕ of 3SAT into an instance (G, k) of CLIQUE, such that ϕ is satisfiable if and only if G has a k-clique. Let us take an arbitrary instance ϕ of 3SAT

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

We create a graph G with the nodes organised into k triplets $t_1, t_2, \ldots, t_k$, where each triplet corresponds to one clause, and each node to one literal in the associated clause. Each two nodes are connected by an edge, except in the two cases:

1. Nodes corresponding to literals within the same clause are not connected.
2. A node corresponding to x is not connected to a node corresponding to $\overline{x}$.

Time it takes to create this graph is clearly linear with respect to the number of literals in the 3cnf-formula ϕ.

Now we need to show that the graph G constructed in this way has a clique (of size k) if and only if the 3cnf-formula ϕ is satisfiable. Let the 3cnf-formula ϕ be satisfiable. Then there is at least one literal in each clause whose value is 1 in the satisfiable assignment. In each triple t_i from G, we select one node that corresponds to a literal whose value is 1. If there is more than one literal in the satisfying assignment whose value is 1, we select an arbitrary one of them. The k nodes (together with the edges between them) selected in this way constitute the k-clique. For any two of these nodes it is true that

1. They do not come from the same triple (because we have selected only one node from each triple).
2. They do not correspond to x and $\overline{x}$, for some variable x (because not both x and $\overline{x}$ can be true in any satisfying assignment).

Therefore, if ϕ is satisfiable, there exist a k-clique in G.

Conversely, let us assume that there exists a k-clique in G. No two nodes from this clique can be in the same triple, therefore they cannot correspond to the literals from the same clause. Since there is k nodes in the clique, and no two nodes can correspond to the literals from the same clause, there has to be a node corresponding to a literal from each of the clauses. If we assign 1 to the literals that correspond to the nodes from the clique, we will get an assignment that satisfies ϕ:

1. At least one literal from each clause is assigned 1, therefore every clause evaluates to 1.
2. Assignment is not contradictory because the same variable will never be assigned both 0 and 1 (because the nodes corresponding to x and $\overline{x}$ are never connected by an edge).

Therefore, if there is a k-clique in G, then the 3cnf-formula ϕ is satisfiable.

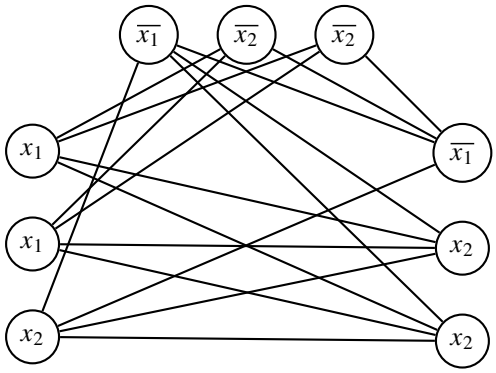
Therefore, we have shown that any instance of the 3SAT problem can be reduced in polynomial-time to an instance of the CLIQUE problem, such that the 3SAT instance has a solution if and only if the CLIQUE instance has a solution. Speaking in terms of the languages, every string from the language of all encodings of the instances of 3SAT which are satisfiable can, in polynomial time, be reduced to a string from the language of encodings of the instances of CLIQUE, for which there exists a clique of a given size. Therefore, 3SAT is polynomially reducible to CLIQUE. □

Let us see how this reduction works on an example. Take the 3cnf-formula

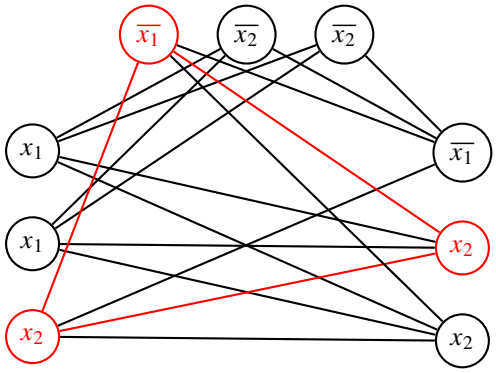
$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2).$$

This formula is reduced to the graph

Graph G that corresponds to this formula is



To find the solution to the instance ϕ of the 3SAT problem, we need to find the 3-clique in this graph. We can see that if we select the nodes corresponding to $\overline{x_1}$ (in the top triplet), x_2 (in the left triplet) and x_2 (in the right triplet), we get one 3-clique. The nodes and the corresponding edges between them are coloured in red in the picture below. Thus, taking $x_1 = 0, x_2 = 1$ gives us a satisfying assignment of the instance ϕ. Therefore, a solution of the instance of the CLIQUE problem led us to the solution of the corresponding instance of the 3SAT problem.



A side effect of Theorem 16.3 is that if CLIQUE is solvable in polynomial-time, then so is 3SAT. This is quite an interesting result, since we said that the two problems do not look in any way related. However, this shows that their complexities are related.

16.3 $\mathcal{NP}$-Completeness

Polynomial-time reducibility gives rise to a very important class of the problem, so called $\mathcal{NP}$-Complete problems.

Definition 16.3: $\mathcal{NP}$−Completeness

A language B is $\mathcal{NP}$-*Complete* if

1. B is in $\mathcal{NP}$
2. Every language A from $\mathcal{NP}$ is polynomial-time reducible to B.

Importance of the class of $\mathcal{NP}$-Complete problem is shown in the following theorem.

Theorem 16.4

If B is $\mathcal{NP}$-Complete and $B \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

$\mathcal{NP}$-Completeness sounds like a remarkable property that only a completely "artificial" languages would have. In fact, the fact that there exists a single language (artificial or otherwise) that is $\mathcal{NP}$-Complete is quite an unexpected result. It seems that, to prove that some language is $\mathcal{NP}$-Complete, we would need to prove

that every $\mathcal{NP}$ language is polynomial-time reducible to it. Luckily, the situation is not that dire, thanks to the following result.

Lemma 16.5

If B is $\mathcal{NP}$-Complete and $B \propto C$ for some $C \in \mathcal{NP}$, then C is also $\mathcal{NP}$-Complete.

Proof. We need to prove that $C \in \mathcal{NP}$ and that every language $A \in \mathcal{NP}$ is polynomial-time reducible to C. By assumption, $C \in \mathcal{NP}$. For any $A \in \mathcal{NP}$, we know that $A \propto B$ (because B is $\mathcal{NP}$-Complete). We also know that $B \propto C$. Due to transitivity of polynomial-time reducibility, we also know that $A \propto C$. Therefore, every $\mathcal{NP}$ language is polynomial-time reducible to C. □

Therefore, to prove that some language C is $\mathcal{NP}$-Complete, we do not need to prove that every $\mathcal{NP}$ language is polynomial-time reducible to it. We only need to find some other $\mathcal{NP}$-Complete language B and show that B is polynomial-time reducible to C. Note that the direction matters here - we need to prove $B \propto C$, not $C \propto B$. So, we "only" need our first $\mathcal{NP}$-Complete language/problem, and then we can use it to prove that other languages/problems are $\mathcal{NP}$-Complete. But for this first $\mathcal{NP}$-Complete problem, we need a full proof that any $\mathcal{NP}$ problem is polynomial-time reducible to it. That is, the proof that every $\mathcal{NP}$ problem is reducibe to that first $\mathcal{NP}$-Complete problem. This is how Cook and Levin showed in 1970s that SAT is $\mathcal{NP}$-Complete. This was the first discovered $\mathcal{NP}$-Complete problem, and it led to the read explosion in the field. Many other problems have since then been proven to belong to the class of $\mathcal{NP}$-Complete problems, all thanks to this first result.