Chapter 22

Introduction to NP-Completeness

Some problems have polynomial algorithms, and such problems are considered "easy," in the sense that they have efficient solutions. At the other extreme are problems (like the halting problem) that are unsolvable by *any* algorithm. Between these two extremes is a mysterious class of problems called NP-complete problems. Such problems are on the cusp between polynomial and non-polynomial.

Being able to understand this class and detect when a problem is in it is useful. Imagine you are setting out to write an algorithm that solves a particular problem. Your aim is to write an algorithm that is is efficient, that is, polynomial (of the lowest degree possible). But if you discover that the problem is NP-complete, then any effort to write a *polynomial* algorithm is all but certain to be wasted.

It is fitting to end our discussion of algorithm complexity here, on the horizon of what is comfortably computable. Sections 22.1 and 22.2 give background on problems, while sections 22.3 and 22.4 develop the notion of an NP-complete problem. Section 22.5 indicates how to show that a particular problem in NP-complete.

22.1 Decision Problems: SAT, 3SAT and Others

In the beginning, we need to specify what the word "problem" means in our present discussion. In doing so, we will be careful, but not too careful. Complete formality (which is necessary for deeper investigations) would mire us in myriad definitions, details and theorems. Our goal is a quick bird's-eye view, so a degree of informality is appropriate.

Informally, by **problem** we mean the type of question for which an algorithm might be designed to answer. So a problem might be something like list sorting, list searching, finding a shortest route between two locations, graph coloring, and so on. True, there are some problems (like the halting problem) that cannot be solved by algorithms. While we acknowledge their existence, our focus will be on problems solvable by algorithms.[1]

[1] Other problems, like, say, money, romantic or relationship problems, are outside of our scope. While it may be important to learn to solve these, you've been reading the wrong book! (But read the last two sentences of page 496.)

To simplify the discussion we will for the remainder of this chapter restrict our focus to problems whose solutions are simply answers of "Yes" or "No". A problem of this type is called a **decision problem**. For example, the following are decision problems: "*Is a particular integer n prime?*" "*Is a given graph G bipartite?*" "*Does a particular program halt?*" and "*Does a list X contain a particular entry x?*

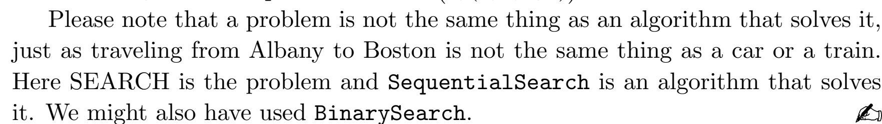
This remainder of this section is a sequence of examples that introduce various conventions, notations and lexicon used in discussing problems. Most especially, the examples call attention to the notion of an *instance* of a problem. An **instance** of a problem is a specific structure to which the problem applies.

By convention, specific problems are given all-cap names. For instance, we will call the problem of searching a list SEARCH. That is our first example.

Example 22.1. List searching (SEARCH). SEARCH is the problem of deciding whether a specific item is on a given list. An instance of SEARCH is a pair (x, X) where x is some entity and X is a list that could potentially contain x as an entry. A solution for the instance is either "Yes" or "No" depending on whether or not x appears in X. (Obviously Yes/No could be replaced with T/F, 1/0, etc.)

So $\big(2,(5,1,7,2,3,9,2)\big)$ is an instance of SEARCH, and its solution is "Yes." Another instance is $\big(8,(1,2,3,4,5,6,7,9)\big)$, and *its* solution is "No."

We know an algorithm for SEARCH, namely `SequentialSearch`. Algorithms that solve decision problems are best thought of as procedures that take problem instances as input and return "Yes" or "No". So `SequentialSearch`$\big(2,(2,3,9,2)\big)$ returns "Yes," while `SequentialSearch`$\big(8,(5,6,7,9)\big)$ returns "No."



Please note that a problem is not the same thing as an algorithm that solves it, just as traveling from Albany to Boston is not the same thing as a car or a train. Here SEARCH is the problem and `SequentialSearch` is an algorithm that solves it. We might also have used `BinarySearch`. ✍

An instance of a decision problem that results in an answer of Yes is called a **yes-instance** of the problem. An instance that results in an answer of No is called a **no-instance**, For example, $\big(2,(5,1,7,2,3,9,2)\big)$ is a yes-instance of SEARCH, while $\big(2,(5,6,7,9)\big)$ is a no-instance.

In the event of a yes-instance, we often want more information about just *why* or *how* the answer is Yes. A **witness** to a yes-instance of a problem is additional information that fulfills the promise that the answer is Yes. For example, given the yes-instance $\big(2,(5,1,7,2,3,9,2)\big)$ of SEARCH, a witness would be the information "Entry 4" (or just "4") because with this information we could go straight to the fourth list entry and see that it is indeed 2. Another witness is "Entry 7."

Example 22.2. Graph k-colorability (k-COLOR). For a given positive integer k, the problem k-COLOR asks whether a given graph has a proper k-coloring. Take for example the graph $G = \big(\{a,b,c,d,e\},\ \{ab,bc,ca,bd,de,ec\}\big)$ in Figure 22.1. This is a yes-instance of 3-COLOR because it *can* be properly 3-colored. A witness appears on the right, namely a proper 3-coloring by white, black and gray. This witness could be encoded as $\big\{\{a\},\{b,e\},\{c,d\}\big\}$, which lists the color classes of the

3-coloring. Another witness is $\{\{a, e\}, \{b\}, \{c, d\}\}$.

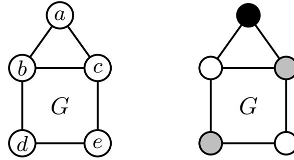


Fig. 22.1 Left: The graph G is a yes-instance of 3-COLOR. Right: A witness to this yes-instance.

The graph K_4 is a no-instance of 3-COLOR, and consequently there is no witness. But K_4 is a yes-instance of 4-COLOR. Any coloring where all four vertices are colored differently is a witness. ✍

Example 22.3. Boolean satisfiability (B-SAT). This problem asks whether an expression involving finitely many boolean variables and symbols $\vee, \wedge, \neg, \Rightarrow, \Leftrightarrow$ (and possibly parentheses) is satisfiable, that is, whether there exists an assignment of T/F values to the variables that makes the expression true. For example, the following expression is an instance of B-SAT using variables X, Y and Z.

$$(X \vee \neg Y) \Rightarrow (X \wedge Z).$$

This is a yes-instance of B-SAT because the assignment $(X, Y, Z) = (T, T, T)$ is a witness. The assignments $(X, Y, Z) = (F, T, F)$ and $(X, Y, Z) = (F, T, T)$ are two other witnesses. By contrast, the expression

$$(X \vee Y) \wedge (\neg X \vee Y) \wedge (X \vee \neg Y) \wedge (\neg X \vee \neg Y)$$

is a no-instance of B-SAT, because there is no witness. The expression is never true, no matter the values of the variables.

It would be easy to write a procedure that decides if any instance of B-SAT is satisfiable (that is, if it is a yes-instance). Just step through all the possible T/F assignments for the variables, and check to see if any of them make the expression true. But if there are n variables, then there are 2^n different T/F assignments. So such an algorithm could be at worst $\Theta(2^n)$. It is a sobering fact that no polynomial algorithms for B-SAT are known. ✍

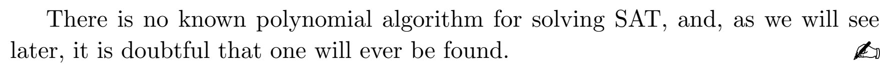
The next two problems are very important in the theory of NP-completeness. They involve some new definitions. A **literal** is a boolean variable or the negation of a boolean variable. For instance, the literals in the expression $(X \vee \neg Y) \Rightarrow (\neg X \wedge Z)$ are X, $\neg Y$, $\neg X$ and Z. For the remainder of this chapter, a boolean expression consisting of one or more literals joined together by $\vee$ will be called a **clause**. For example, $(X \vee \neg Y \vee Z \vee \neg X)$ is a clause, as are $(\neg X \vee \neg Y)$ and $(X \vee Y \vee Z \vee X)$. To be uniform and consistent, we enclose our clauses in parentheses, even when it is not technically necessary. Clauses can even be single literals, like (X) and $(\neg X)$.

The joining of boolean expressions with $\vee$ is sometimes called a **disjunction**, so we could also describe a clause as a disjunction of one or more literals.

A boolean expression is said to be in **conjunctive normal form** if it consists of one or more clauses joined by $\wedge$. Thus the expression $(X \vee \neg Y) \wedge (W \vee X \vee \neg Z) \wedge (Y)$ is in conjunctive normal form, as are $(W \vee Y) \wedge (X \vee \neg Y) \wedge (X \vee Z) \wedge (X \vee W)$ and $(W \vee X \vee \neg Z)$ and $(W) \wedge (X) \wedge (\neg Z)$. The joining of boolean expressions with $\wedge$ is sometimes called a **conjunction**, so an expression in conjunctive normal form is a conjunction of one or more clauses.

Example 22.4. Satisfiability (SAT). The problem SAT asks whether an expression in conjunctive normal form is satisfiable. For example, $(X \vee \neg Y) \wedge (W \vee X \vee \neg Z) \wedge (Y)$ is a yes-instance of SAT because the assignment $(X, Y, Z, W) = (T, T, T, T)$ makes it true (and is thus a witness). The assignment $(X, Y, Z, W) = (T, T, F, F)$ is another witness. By contrast, $(X \vee Y) \wedge (\neg X \vee Y) \wedge (X \vee \neg Y) \wedge (\neg X \vee \neg Y)$ is a no-instance of SAT because it is not satisfiable.

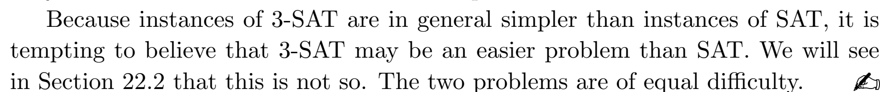
There is no known polynomial algorithm for solving SAT, and, as we will see later, it is doubtful that one will ever be found.



Example 22.5. 3-Satisfiability (3-SAT). This is a simplified version of SAT in which every clause has exactly three literals. Thus $(X \vee \neg Y \vee Y) \wedge (W \vee X \vee \neg Z)$ and $(U \vee W \vee X) \wedge (\neg W \vee X \vee Z) \wedge (\neg X \vee X \vee Z) \wedge (U \vee Y \vee W)$ are instances of 3-SAT, but $(X \vee \neg Y \vee Z) \wedge (W \vee X \vee \neg Z \wedge Y)$ is not.

The decision problem 3-SAT asks whether an arbitrary boolean expression in conjunctive normal form with three literals per clause is satisfiable.

Because instances of 3-SAT are in general simpler than instances of SAT, it is tempting to believe that 3-SAT may be an easier problem than SAT. We will see in Section 22.2 that this is not so. The two problems are of equal difficulty.



22.2 Polynomial Reductions of Problems

It can happen that the ability to efficiently solve one problem implies the ability to efficiently solve another. We now explore this important idea.

To start the discussion, take two decision problems A and B. Suppose there exists a procedure `A2B` that converts any instance I of A to an instance `A2B`(I) of B, in such a way that I is a yes-instance (or no-instance) of A if and only if `A2B`(I) is a yes-instance (or no-instance) of B.

Say we have procedure `SolveB` that solves B. That is, `SolveB`(J) returns "yes" if and only if J is a yes-instance of B. (Otherwise it returns "No.")

Under these circumstances, I will be a yes-instance of A if and only if `SolveB`$($`A2B`$(I))$ returns "Yes." This means `SolveB` can be used to solve A: Just use `A2B` to convert any instance of A to an instance of B, and then apply `SolveB` to the result. We have thus transformed A to B in such a way that solving A reduces to the problem of solving B.

Now suppose further that `A2B` is *polynomial.* To be precise, say `A2B` is $O(n^k)$.

(This means that `A2B` executes $O(n^k)$ steps when processing an input I of size n.) It follows that if an instance I of A has size n, then the corresponding instance $J = \texttt{A2B}(I)$ of B has size no greater than $O(n^k)$, because `A2B` can't expend more than $O(n^k)$ steps in generating the output J.

Finally, suppose that the procedure `SolveB` for B is also polynomial. Say it is $O(m^\ell)$ for some positive integer ℓ, so `SolveB` executes $O(m^\ell)$ steps when processing an input of size m. Now, if an instance I of A has size n, then $\texttt{A2B}(I)$ has size $O(n^k)$, so running $\texttt{SolveB}\big(\texttt{A2B}(I)\big)$ uses $O\left(\left(n^k\right)^m\right) = O\big(n^{k\ell}\big)$ steps. So solving A by applying `SolveB` to `A2B` is $O(n^{k\ell})$, that is to say, polynomial.

All of this is summarized in the box below. But first, a minor point about notation. In $\texttt{SolveB}\big(\texttt{A2B}(I)\big)$, the procedure `SolveB` accepts the output of $\texttt{A2B}(I)$. This is like the composition of two functions, so we agree to write $\texttt{SolveB}\big(\texttt{A2B}(I)\big)$ as $\texttt{SolveB} \circ \texttt{A2B}(I)$, and we use $\texttt{SolveB} \circ \texttt{A2B}$ as a name for the procedure defined by first running `A2B` and then applying `SolveB` to its output.

Fact 22.1. Consider two decision problems, A and B. Say there is a polynomial procedure `A2B` that converts all yes-instances of A to yes-instance of B (and all no-instances of A to no-instance of B).

Under these circumstances, given a polynomial procedure `SolveB` for B, the composition $\texttt{SolveB} \circ \texttt{A2B}$ is a polynomial procedure that solves A.

This gives the context for the main idea of this section.

Definition 22.1. A decision problem A can be **polynomially reduced** to a decision problem B if there is a polynomial procedure that converts yes-instances of A to yes-instances of B, and no-instances of A to no-instances of B.

So if A can be polynomially reduced to B, and there is an efficient way to solve B, then there is an efficient way to solve A. Thus the statement "A *can be polynomially reduced to* B" can be taken to mean "A *is no harder than* B" in the sense that if you can find a good algorithm for B, then you can find a good algorithm for A. We might even express this informally as A $\leq$ B. (Unfortunately, the word "reduced" may suggest that B may be *easier* than A, when the opposite is true. It would be better to say "A *can be polynomially embedded in* B." But unfortunately the lexicon is very entrenched.)

The next example shows SAT $\leq$ 3-SAT, which may seem surprising because SAT seems more complicated than 3-SAT.

Example 22.6. A polynomial reduction of SAT to 3-SAT. Recall the problem SAT (Example 22.4) asks if a boolean expression in conjunctive normal form is satisfiable. The problem 3-SAT (Example 22.5) asks the same question, but in the circumstance where each clause has three literals.

There is an easy way to reduce an instance I of SAT to an instance of 3-SAT.

First, any clause (L_1) of I having just one literal can be replaced by the logically equivalent clause $(L_1 \vee L_1 \vee L_1)$. (Here L_1 stands for either a boolean variable or its negation.) Likewise a clause $(L_1 \vee L_2)$ in I with two literals can be replaced with $(L_1 \vee L_2 \vee L_2)$.

Now suppose an I has a clause with *four* literals, like $(L_1 \vee L_2 \vee L_3 \vee L_4)$. Let V_1 be a new variable that does not appear in I. Check that $(L_1 \vee L_2 \vee L_3 \vee L_4)$ is satisfiable if and only if $(L_1 \vee L_2 \vee V_1) \wedge (\neg V_1 \vee L_3 \vee L_4)$ is satisfiable. So the clause $(L_1 \vee L_2 \vee L_3 \vee L_4)$ in I can be replaced by the conjunction $(L_1 \vee L_2 \vee V_1) \wedge (\neg V_1 \vee L_3 \vee L_4)$.

For example, the following instance I of SAT converts to an instance I' of 3-SAT such that I is satisfiable (is a yes-instance of SAT) if and only if I' is satisfiable (is a yes-instance of 3-SAT).

$$I = (L_1 \vee L_2 \vee L_3 \vee L_4) \wedge (L_5 \vee L_6) \wedge (L_7 \vee L_8 \vee L_9 \vee L_{10})$$

$$I' = (L_1 \vee L_2 \vee V_1) \wedge (\neg V_1 \vee L_3 \vee L_4) \wedge (L_5 \vee L_6 \vee L_6) \wedge (L_7 \vee L_8 \vee V_2) \wedge (\neg V_2 \vee L_9 \vee L_{10})$$

But how would we handle instances I of SAT that have clauses with more than four literals? Consider a clause with $n \geq 4$ literals, like

$$(L_1 \vee L_2 \vee L_3 \vee L_4 \vee \cdots \vee L_n). \tag{22.1}$$

Let $V_1, V_2, \ldots V_{n-3}$ be new variables, and check that the following conjunction of 3-literal clauses is satisfiable if and only the clause (22.1) is satisfiable:

$$\begin{aligned}
&(L_1 \vee L_2 \vee V_1) \wedge \\
&(\neg V_1 \vee L_3 \vee V_2) \wedge \\
&(\neg V_2 \vee L_4 \vee V_3) \wedge \\
&(\neg V_3 \vee L_5 \vee V_4) \wedge \\
&\quad \vdots \\
&(\neg V_{n-4} \vee L_{n-2} \vee V_{n-3}) \wedge \\
&(\neg V_{n-3} \vee L_{n-1} \vee L_n).
\end{aligned} \tag{22.2}$$

So we have seen that any instance I of SAT can be reduced to an instance I' of 3-SAT by replacing its clauses with conjunctions of 3-literal clauses.

It is easy to imagine harnessing this scheme to a polynomial procedure `SAT23SAT` that inputs instances I of SAT and returns corresponding instances I' of 3-SAT. The procedure simply scans the clauses of I. It replaces each (L_i) with $(L_i \vee L_i \vee L_i)$, and each $(L_i \vee L_j)$ with $(L_i \vee L_j \vee L_j)$. And each time a clause of form (22.1) is found, the procedure generates $n-3$ new variables V_i and replaces the clause (22.1) with the conjunction (22.2). The resulting instance I' of 3-SAT is satisfiable if and only if I is satisfiable.

In this way, SAT can be polynomially reduced to 3-SAT. ✍

22.3 The Classes P and NP

The time has come to describe two broad classes of problems called P and NP. The class P consists of all decision problems that are solvable by polynomial procedures. Let's record the definitions carefully, beginning with P.

Definition 22.2. The class P consists of all decision problems that can be decided with polynomial procedures.

In other words, a problem A belongs to P if a polynomial procedure `SolveA` can be written such that `SolveA`(I) returns "Yes" for every yes-instance I of A, and "No" for every no-instance I of A. Thus SEARCH $\in$ P because this problem is solved by the polynomial procedure `SequentialSearch`. (See Example 22.1.)

So A $\in$ P means that problem A can be decided with a polynomial procedure. This is so even if no such procedure is known. The statement A $\in$ P means that it is *possible* to find a polynomial procedure for deciding A. To prove A $\in$ P it suffices to write a procedure `SolveA` that decides A, *prove* that `SolveA` really does decide A, and *then* prove that `SolveA` is polynomial.

The class P is very natural. It consists of all the "good" decision problems that can be decided with efficient algorithms. When you set out to write an algorithm, you are aiming for a polynomial algorithm. You are hoping your problem is in P.

Before defining the class NP, a little context may be helpful. In the early days of computers (the 1950s and 1960s), computer scientists noticed that there are a great many problems that that have no known polynomial algorithm, but yet the truth of a witness—if given—can be checked in polynomial time.

One example of such a problem is 3-SAT. Given an instance of 3-SAT, like $I = (X_1 \vee X_2 \vee \neg X_3) \wedge (X_4 \vee \neg X_5 \vee X_6) \wedge \ldots$, we could decide if it is satisfiable by trying every possible T/F combination of the variables $X_1, X_2, X_3 \ldots$ and stopping as soon as (or if) a combination results in I being true. At that point we've found a witness $W = (W_1, W_2, \ldots)$, if one exists. (Here $W_i \in \{T, F\}$ is the value of X_i.) But if I has n variables, then there are 2^n possible T/F combinations for them, so an algorithm that takes this approach has exponential complexity. But if someone presented us with a $W = (W_1, W_2 \ldots)$ and claimed it were a witness to I, then this claim could be *checked* in polynomial time. (Start with the left-most clause of I, plug in the T/F values from W, check that this makes at least one literal of the clause true, then move on to the next clause, etc.)

So 3-SAT is a problem for which a witness can be checked in polynomial time. There are many problems of this type, and the class NP is designed to contain them.

Definition 22.3. The class NP consists of decision problems of the following type. A problem is in NP provided there is a procedure `CheckAnswer`(I, W) with the property that for any instance I of the problem and string W, `CheckAnswer`(I, W) returns "Yes" in polynomial time if W is a witness to I.

One example of a problem in NP is 3SAT. The reason is the (as noted on the previous page), a potential witness to an instance of 3-SAT can be checked in polynomial (in fact linear) time. One could certainly write a procedure `CheckAnswer3SAT`(I, W) that returns "Yes" in polynomial time if W is a witness to the instance I of 3SAT. Likewise, B-SAT and SAT belong to NP, as you can check.

Read Definition 22.4 carefully. It stipulates only that `CheckAnswer`(I, W) returns "Yes" in polynomial time if W is a witness to I. All bets are off if W is *not* a witness to I (which could happen if I is a no-instance, or if W is a "garbage" string). In that case it is *possible* that `CheckAnswer`(I, W) could return "No" in polynomial time, but it would not violate the definition if it returned "No" in non-polynomial time, or even if it halted and returned no output at all – or even if it looped forever. Because such lax demands are placed on `CheckAnswer`, it would seem that NP is a is truly a vast class of decision problems.

It is a fundamental fact that P $\subseteq$ NP. To prove this, assume A $\in$ P. This means that there is a polynomial procedure `SolveA`(I) that decides A. We need to show A $\in$ NP, which (by Definition 22.4) amounts to describing a procedure `CheckAnswer`(I, W) with the property that for any instance I of A, and string W, `CheckAnswer`(I, W) returns "Yes" in polynomial time if W is a witness to I. To do this, just arrange `CheckAnswer`(I, W) to ignore W and run `SolveA`(I). If W is a witness to I, then I is a yes-instance, so `SolveA`(I) returns "Yes" in polynomial time. Consequently `CheckAnswer`(I, W) returns "Yes" in polynomial time.

Three final remarks before moving on. First, we defined P and NP to be a classes (or sets) of problems. Admittedly, this is vague because we haven't said just what kind of mathematical entity a "problem" is. That is, if asked what exactly an element of P is, we might be hard-pressed to say. Nothing can be really proved in the presence of such murkiness. Careful treatments of P and NP codify problems as mathematical structures called **languages**, which are certain sets of strings of symbols. In such a setting, P is a set of languages. Though this has the advantage of precision, it also involves a lot of preliminary theory about languages. This book takes an informal approach and relies on your intuitive sense of what a problem is. This allows us to cover a lot of ground, but nothing will be completely *proved*.

Second, our focus on *decision* problems is not as limiting as it seems, because decision problems are benchmarks that measure the difficulty of problems in general. For example, let Problem A be "Is the integer n prime?" and let B be "Find the prime factorization of n." If it turns out that the decision problem A is problematic (which it is), then the non-decision-problem B will problematic too, because solving B would decide A.

Third, we showed above that P $\subseteq$ NP. But presently it is not at all clear that NP is any bigger than P, that is, it is not known whether or not P = NP. Most computer scientists think it's more likely that P $\subset$ NP, but no one knows for sure. This is probably the most significant unsolved problem in computer science today. The Clay Institute offers a prize of one million US dollars to anyone resolves it.

22.4 Cook's Theorem and NP-Completeness

We are ready to understand a major idea in computer science and complexity theory. Take two problems $A, B \in \mathrm{NP}$. In Section 22.2 we saw that if A can be polynomially reduced to B, then B is harder than (or as hard) as A, in the sense that any procedure that decides B is also powerful enough to decide A.

It is a startling fact that, in this sense, some problems in NP are the absolute hardest problems in NP. Such problems called *NP-complete* problems.

Definition 22.4. A problem is **NP-complete** if it is in NP, and any other problem in NP can be polynomially reduced to it.

The fact that NP-complete problems exist is astonishing. Think of it this way: There is no biggest integer in $\mathbb{Z}$. But NP is different. There *are* hardest problems in NP. An NP-complete problem is as hard as it gets.

How do we know that NP-complete problems even exist? A major theorem by Stephen Cook says that the boolean satisfiability problem SAT is NP-complete.

Theorem 22.2. (Cook) *The satisfiability problem* SAT *is NP-complete.*

The "proof" given below is not a real proof, but just an informal explanation of why Cook's theorem is plausible. Careful proofs of Cook's theorem use formal models of problems and computation called **languages** and **Turing machines**, which—in the interest of brevity and clarity—we have not developed. Still, the "proof" given below does capture the gist of a real proof.

Proof. We want to show that SAT is NP-complete. By Definition 22.4, we need to show that for any arbitrary problem $A \in \mathrm{NP}$, any instance I of A can be polynomially reduced to an instance I' of SAT. The basic idea is that if $A \in \mathrm{NP}$, then there is a polynomial procedure that either decides whether I is a yes-instance, or checks the validity of a witness to I. Below we will see how *the running of the procedure on* I is mirrored by a certain instance I' of SAT. We will show that I is a yes-instance of A, if and only if the procedure returns "Yes," if and only if I' is satisfiable.

For simplicity, we will work out the details with *one* particular problem in NP. (A real proof would show the approach works for *any* particular problem in NP.) The problem we will pick is that of deciding if a non-negative integer n is even. Call this problem PARITY. An instance of of PARITY is an integer $n \geq 0$. It is a yes-instance if n is even, and a no-instance if n is odd. We need to show how to convert the instance n of PARITY to an instance I' of SAT, such that n is even if and only if I' is satisfiable.

It is easy to write a polynomial procedure that decides if n is even. Here is one called `EvenTest`. It keeps subtracting 2 from $x := n$ until getting 0 or 1. If 0, then n is even (so "True" is returned). If 1, then n is odd (so "False" is returned).

Procedure: EvenTest(n) **Input:** An integer $n \geq 0$.
Returns: True if n is even; otherwise False.

```
begin
  x := n
  while x ≥ 2 do
  |  x := x − 2
  end
  if x = 0 then  Ans := True
  if x = 1 then  Ans := False
  return Ans
end
```

`EvenTest` has just two variables, x and Ans. Initially $x := n$. Call this Step 0. Then the **while** loop executes steps $1, 2, 3, \ldots$, in which x takes on values $n-2$, $n-4, n-6, \ldots$, respectively. In all, the loop executes $\lfloor n/2 \rfloor$ steps, so `EvenTest` does $1+\lfloor n/2 \rfloor+2$ steps. Hence its complexity is $O(n)$.

Take an instance n of PARITY. We will now reduce n to an instance I' of SAT. (Recall: this means n is even if and only if the boolean expression I' is satisfiable.) This will be done by building I' so that it mimics the running of `EvenTest`(n). First, translate the integer variable x in `EvenTest` to the following boolean variables:

$$\begin{array}{cccccc} X_{0,0} & X_{1,0} & X_{2,0} & X_{3,0} & \ldots & X_{n,0} \\ X_{0,1} & X_{1,1} & X_{2,1} & X_{3,1} & \ldots & X_{n,1} \\ X_{0,2} & X_{1,2} & X_{2,2} & X_{3,2} & \ldots & X_{n,2} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ X_{0,n} & X_{1,n} & X_{2,n} & X_{3,n} & \ldots & X_{n,n} \end{array}$$

The intended meaning is that $X_{j,k}$ is true if $x = j$ in step k of `EvenTest`(n). Otherwise $X_{j,k}$ is false. Notice that if $\lfloor n/2 \rfloor \leq k \leq n$, then `EvenTest` never even gets to step k, so some of these variables (like those in the last row) are unnecessary. They can be regarded as false.

The variable Ans in `EvenTest` is already boolean; it will also be a variable in I'.

Notice that `EvenTest`'s initial assignment of $x := n$ can be mirrored by the assertion that $X_{n,0}$ is true. Now imagine that `EvenTest` has just completed step k, at which point x has some value $m \geq 2$. Then in the next $(k+1)$th step, x will take on the value $m - 2$. This fact can be captured by the assertion that the following boolean expression is true:

$$X_{m,k} \Rightarrow X_{m-2,k+1}. \tag{22.3}$$

As $2 \leq m \leq n$ and $0 \leq k \leq n-1$, there are $(n-1)n = n^2 - n$ of these expressions.

If x attains the value 0 in some step k of `EvenTest`, then `EvenTest` assigns $Ans :=$ True, which amounts to asserting that this boolean expression is true:

$$X_{0,k} \Rightarrow Ans. \tag{22.4}$$

But if $x = 1$ at step k, then `EvenTest` assigns $Ans :=$ False, and this is true:

$$X_{1,k} \Rightarrow \neg Ans. \tag{22.5}$$

There are $n+1$ expressions of form (22.4), one for each possible value of k. Likewise there are $n+1$ expressios of form (22.5).

Boolean expressions (22.3), (22.4) and (22.5) reflect the running of `EvenTest`(n), with one exception. Obviously, at any step k of `EvenTest`(n), the variable x has only one value. But we have so far left open the possibility that two variables $X_{i,k}$ and $X_{j,k}$ could both be true, which would mean $x = i$ **and** $x = j$ at step k. To preclude this absurdity, we could demand that the following expressions are all true:

$$\neg(X_{i,k} \wedge X_{j,k}) \tag{22.6}$$

where $0 \leq i < j \leq n$ and $0 \leq k \leq n$. By the multiplication principle, there are $\binom{n}{2}(n+1)$ expressions of form (22.6).

Tallying (22.3), (22.4), (22.5) and(22.6), we can say that `EvenTest`(n) returns a value of "True" if and only if *all* expressions in the following table are true.

clause	meaning	quantity
$(X_{n,0})$	Variable x initially equals n.	1
$(X_{m,k} \Rightarrow X_{m-2,k+1})$	If x=m in step k, then x=$m-2$ in step k+1.	$n(n-1)$
$(X_{0,k} \Rightarrow Ans)$	If $x = 0$ at some step k, then Ans = True.	$n+1$
$(X_{1,k} \Rightarrow \neg Ans)$	If $x = 1$ at some step k, then Ans = False.	$n+1$
$\neg(X_{i,k} \wedge X_{j,k})$	At step k, variable x never has two values.	$(n+1)\binom{n}{2}$
(Ans)	`EvenTest` returns "True" (i.e., n is even)	1

Because $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$, and DeMorgan's law says $\neg(P \wedge Q) = \neg P \vee \neg Q$, this table can be rewritten as a collection of disjunctions:

clause	meaning	quantity
$(X_{n,0})$	Variable x initially equals n.	1
$(\neg X_{m,k} \vee X_{m-2,k+1})$	If x=m in step k, then x=$m-2$ in step k+1.	$n(n-2)$
$(\neg X_{0,k} \vee Ans)$	If $x = 0$ at some step k, then Ans = True.	n
$(\neg X_{1,k} \vee \neg Ans)$	If $x = 1$ at some step k, then Ans = False.	n
$(\neg X_{i,k} \vee \neg X_{j,k})$	At step k, variable x never has two values.	$n \cdot \binom{n}{2}$
(Ans)	`EvenTest` returns "True" (i.e., n is even)	1

Now let $I' = (X_{n,0}) \wedge (\neg X_{2,0} \vee X_{0,1}) \wedge (\neg X_{3,0} \vee X_{1,1}) \wedge \ldots \wedge (Ans)$ be the conjunction of all clauses in the above table. This is an instance of SAT, and we have shown that n is a yes-instance of PARITY, if and only if `EvenTest`(n) = True, if and only if I' is satisfiable.

Further, notice that the number of clauses in I' is a polynomial of degree 3, so I' can be built from n in a polynomial number of steps. Thus we have the desired polynomial reduction of PARITY to SAT.

A careful proof of Cook's theorem uses this approach to show that *any* problem in NP has a polynomial reduction to SAT. As we mentioned, the details involve development of some preliminary machinery, but the main idea is identical to what we have done above. □

The implications of Cook's theorem are staggering. For one, it says that no problem in NP is harder than SAT, in the sense that a polynomial procedure for SAT would apply to any other problem in NP, because any other problem is polynomially reducible to SAT. So if there were a polynomial algorithm for SAT, then every problem in NP would also be in P, and we'd conclude P = NP. Is there a polynomial algorithm for SAT? Does P = NP? No one knows.

What *is known* is that there are a great many NP-complete problems. We will encounter some in the next section. Keep in mind that if a polynomial procedure could be found for *any* single NP-complete problem, then *every* problem in NP would be solvable by a polynomial procedure, so P = NP. For this reason it is widely believed (but not known for certain) that NP-complete problems cannot be decided with polynomial algorithms.

22.5 Proving Problems are NP-Complete

Let's highlight the main points of this chapter so far.

- A decision problem is in P if some polynomial procedure can decide it.
- A decision problem is in NP if some polynomial procedure can check the validity of any proposed witness. (That is, for any witness W of a yes-instance, the procedure confirms that W is indeed a yes-instance in polynomial time.)
- A decision problem is NP-complete if it is in NP, any other problem in NP can be polynomially reduced to it.
- The problem SAT is NP-complete (Cook's Theorem).

As it turns out, SAT is not the only NP-complete problem. There are many. This section explains how to show that a particular decision problem in NP-complete. The general strategy for doing this is simple. Suppose we want to prove that some problem A is NP-complete. By the definition of NP-complete, this means that we need to show two things: First, A $\in$ NP. Second, we must show that any problem in NP can be polynomially reduced to A.

The second step is not has hard as you may expect. Suppose there is another problem B that is known to be NP-complete. All we need to do is show that B is polynomially reducible to A. Then, since any problem in NP can be polynomially reduced to B, and B can be polynomially reduced to A, it follows that any problem in NP can be polynomially reduced to A.

In summary, here is the procedure for showing a problem A is NP-complete.

How to show that a problem A is NP-complete

1. Show A $\in$ NP.
2. Show a known NP-complete problem can be polynomially reduced to A.

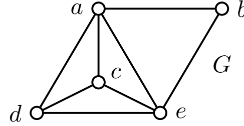
Our first example will carry out this process to show that 3-SAT is NP-complete. This will expand our list of known NP-complete problems to two: SAT and 3-SAT.

Example 22.7. The problem 3-SAT is NP-complete.

The first step is to show that 3-SAT is in NP. Recall that this means that we must show that the validity of a proposed witness to an instance of 3-SAT can be confirmed in polynomial time. So let I be an instance of 3-SAT. Say I consists of n clauses, containing variables $X_1, X_2, \ldots$. Let $W = (W_1, W_2, \ldots)$ be a potential witness to I. (Each W_i is a T/F value, standing for the truth assignment for the variable X_i.) A procedure can be written that scans the clauses of I, checking that, under the truth assignments W, each clause contains at least one true literal. As each clause has three literals, this involves checking a total of $3n$ literals. Thus such a procedure has complexity $O(n)$, which is polynomial.

The second step is to display a polynomial reduction of the NP-complete problem SAT to 3-SAT. This was done in Example 22.6. Thus 3-SAT is NP-complete. ✍

So now we know two NP-complete problems, SAT and 3-SAT. Next we come to a third NP-compete problem, that of finding cliques in graphs. A **k-clique** in a graph is a subset X of its vertices such that any two vertices in X are adjacent. (Thus X induces a complete K_k subgraph.) For example, the graph G below has eight 2-cliques, namely $\{a, b\}, \{a, e\}, \{a, c\}, \{a, d\}, \{c, d\}, \{c, e\}, \{d, e\}$ and $\{b, e\}$. It has four 3-cliques $\{a, b, e\}, \{a, c, e\}, \{a, d, c\}$ and $\{d, c, e\}$. It also has one 4-clique, $\{a, c, d, e\}$. Note that G has no 5-clique.



Given an integer k and a graph G, we can ask whether or not G has any k-cliques. Let's call this decision problem CLIQUE. An instance of CLIQUE can be thought of as a pair (k, G) where k is an integer and G is a graph. The pair (k, G) is yes-instance if G has a k-clique, otherwise it is a no-instance. Although this may seem an innocent enough problem, it is, in fact, NP-complete.

Example 22.8. The problem CLIQUE is NP-complete.

The first step is to show that CLIQUE is in NP. To do this we need to show that a proposed witness to an instance (k, G) can be verified in polynomial time. A proposed witness is a set $W = \{v_1, v_2, \ldots, v_k\} \subseteq V(G)$ of k vertices that could potentially be a clique in G. To check that this is really a clique, one would have to investigate all 2-element subsets $\{v_i, v_j\}$ of W and confirm each is a pair of adjacent vertices in G. Since there are $\binom{n}{2} = n(n-1)/2$ pairs to check, this can be done in $O(n^2)$ time. Thus CLIQUE is in NP.

In the second step we demonstrate a polynomial reduction of 3-SAT (which is NP-complete by Example 22.7) to CLIQUE. Doing this involves transforming any instance I of 3-SAT to an instance $I' = (k, G)$ of k-clique, such that I is a

yes-instance of 3-SAT (i.e., is satisfiable) if and only if (k, G) is a yes-instance of CLIQUE (i.e., the graph G has a k-clique).

We will illustrate this with an example, from which the general procedure will be clear. Consider the following instance of 3-SAT.

$I=(X_1 \vee X_2 \vee \neg X_4) \wedge (\neg X_1 \vee X_3 \vee X_4) \wedge (X_4 \vee \neg X_3 \vee X_2) \wedge (X_2 \vee X_4 \vee X_3) \wedge (\neg X_2 \vee X_3 \vee X_4)$

This instance of 3-SAT has 5 clauses. Our desired instance of CLIQUE will have the form (k, G), where $k = 5$ is the number of clauses in I. Make the graph G as follows. For each clause in I, G has three vertices, labeled by the literals in the clause. This is illustrated in Figure 22.8 (left), where the three vertices corresponding to different clauses in I fall within different gray regions. An edge connects any pair of vertices from different clauses, provided that the pair does not consist of a variable and its negation. (For example, in Figure 22.8 (left), there is an edge from the X_1 in C_1 to the X_3 in C_2, but there is no edge joining X_1 to $\neg X_1$.)

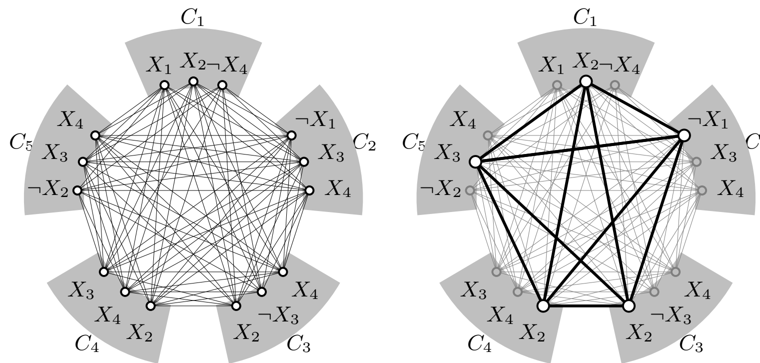


Fig. 22.2 Left: The graph G in the instance $(5, G)$. Right: A 5-clique in G.

Because no two vertices in a single gray region are adjacent, any 5-clique in G has one vertex from each of the five gray regions. For example the 5-clique in Figure 22.8 (right) has vertices labeled by $\neg X_1$, X_2 and X_3, one per gray region (i.e., one per clause in I). Consequently, if in I we set $X_1 = F$, $X_2 = T$ and $X_3 = T$ (and give the other variables arbitrary assignments), then I will be satisfied.

In fact, *any* 5-clique K in G consists of five vertices labeled with literals from I, one literal per clause. No two of these are labeled by a variable X_i and its negation $\neg X_i$, because by definition no edge in G joins two such vertices. Therefore no conflict can arise by setting each literal in K equal to T, and doing so results in a a truth assignment that satisfies I.

Conversely, given a truth assignment that makes I true, there is a 5-clique K in G got by selecting a true literal in each clause of I and using it as a vertex in K.

Consequently, I is satisfiable if and only if G has a 5-clique. In other words, I is a yes-instance of 3-SAT if and only if $(5, G)$ is a yes-instance of CLIQUE.

This same procedure works not just for the specific I above, but *any* instance I of 3-SAT. From I, generate the instance (k, G) by scanning through the clauses of I. For each clause C_i, generate three new vertices for G and label them with the literals in C_i. Then scan through the clauses of I again. For each vertex v labeled by a literal in C_i, connect it by an edge to all other vertices corresponding to the clauses C_j with $i \neq j$, *except* those whose label is the negation of the label of v. Let k be the number of clauses in I. Then, as above, I is satisfiable if and only if (k, G) is a yes-instance of CLIQUE, that is, if and only if G has a k-clique.

If I has n clauses, then building G involves generating $3n$ vertices, then adding in no more than $3n(n-1)/2$ edges. Thus building G can be done in $O(n^2)$ steps, so constructing the instance (k, G) is $O(n^2)$. Thus we have built a *polynomial* reduction of 3-SAT to CLIQUE. This completes the proof that CLIQUE is NP-complete. ✍

Our next example concerns graph coloring.

Example 22.9. Deciding if an arbitrary graph can be 3-colored is NP-complete.

Recall that a graph can be 3-colored if its vertices can be colored with three colors, in such a way that any two adjacent vertices have different colors. Given an arbitrary graph, one can ask whether or not it can be 3-colored. Let's call this decision problem 3COLOR. An instance of 3COLOR is a graph G. It is a yes-instance if it can be 3-colored; otherwise it is a no-instance. The problem 3COLOR is that of deciding whether an arbitrary instance G is a yes- or no-instance.

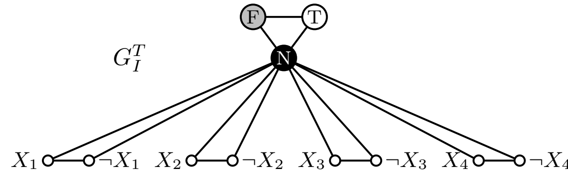
The first step in showing 3COLOR is NP-complete is to confirm that it is in NP. A witness to an instance G is an assignment three colors $\{1, 2, 3\}$ to the vertices of G, that is, a witness to G is a function $f : V(G) \to \{1, 2, 3\}$. Say G has n vertices. We can check if f is a valid coloring by examining all $\binom{n}{2} = n(n-1)/2$ pairs u, v of G's vertices, and confirming that $f(u) \neq f(v)$ when $uv \in E(G)$. This process involves $O(n^2)$ steps, so the problem of checking the witness is polynomial in the size of G. Hence 3COLOR $\in$ NP.

For the second step we describe a polynomial reduction of 3SAT to 3COLOR: For any instance I of 3SAT, we will construct a graph G_I with the property that I is satisfiable if and only if G_I can be 3-colored.

We illustrate this with a specific instance I of 3SAT. (Our construction of G_I will indicate a general recipe for converting any arbitrary I to a corresponding G_I.) So for the remainder of this example, let's consider the following instance I of 3SAT:

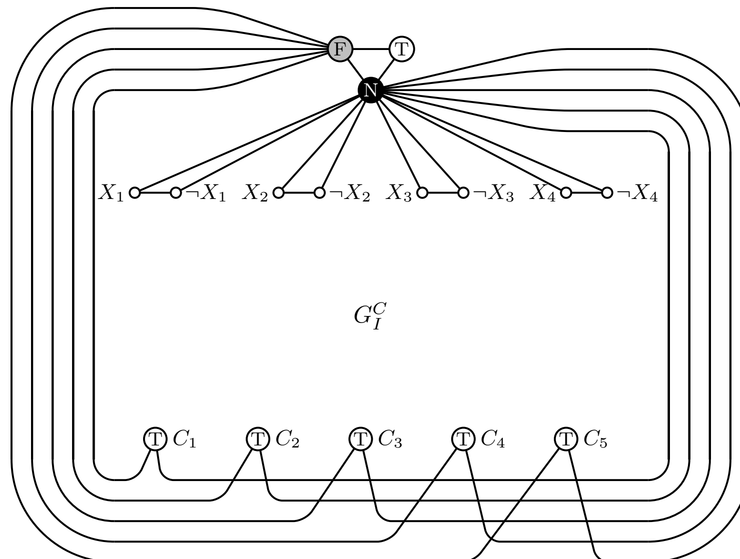
$$I = \underbrace{(X_1 \vee X_2 \vee \neg X_4)}_{C_1} \wedge \underbrace{(\neg X_1 \vee X_3 \vee X_4)}_{C_2} \wedge \underbrace{(X_4 \vee \neg X_3 \vee X_2)}_{C_3} \wedge \underbrace{(X_2 \vee X_4 \vee X_3)}_{C_4} \wedge \underbrace{(\neg X_2 \vee X_3 \vee X_4)}_{C_5}.$$

This instance has four variables $X_1, \ldots, X_4$ and five clauses $C_1, \ldots, C_5$. We now build a graph G_I whose 3-colorability mirrors the satisfiability of I. We build G_I in stages. The first piece G_I^T of G_I is shown below. It has a triangle at the top, plus eight vertices labeled by the four variables in I, along with their negations.

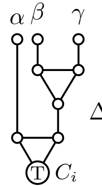


This graph G_I^T can certainly be 3-colored with "colors" $\{$T,F,N$\}$, which you should think of as standing for "True," "False" and "Neutral." The three vertices of the top triangle necessarily get different colors. Let's agree that its bottom vertex is colored N, as indicated. With this agreement, any vertex labeled X_i or $\neg X_i$ must be colored T or F. Further, because each vertex pair $\{X_i, \neg X_i\}$ is adjacent, X_i, and $\neg X_i$ have opposite T/F colors. So if the top triangle is colored as indicated, G_I^T has $2^4 = 16$ different 3-colorings, and each one corresponds to a particular True/False assignment to the variables in I. Thus 3-colorings of G_I^T model and mirror the ways that the variables in I can be instantiated. (In general, this kind of construction is sometimes called a **truth-setting gadget**.)

Some 3-colorings of G_I^T may correspond to True/False assignments that make I true, and others may make it false. The next step in constructing G_I is to add to G_I^T a mechanism that models the fact that each clause C_i in I is required to be true. This is accomplished by adding to G_I^T five new vertices labeled $C_1, \ldots, C_5$, each joined to the vertices at the top that are colored N and F, as shown below. Therefore each vertex C_i must be colored T. The intended meaning is that each clause C_i is true.

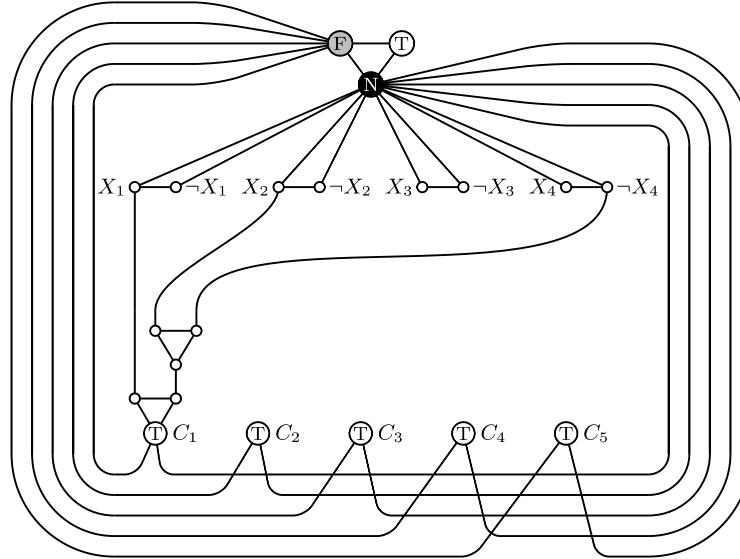


Of course this new graph G_I^C is still 3-colorable, as there are still $24 = 16$ ways to color the vertices labeled X_i and $\neg X_i$ with the colors T/F. We are next going to add mechanisms that make the graph 3-colorable if and only if the instance I is satisfiable. This will be done by inserting subgraphs Δ of the following form.



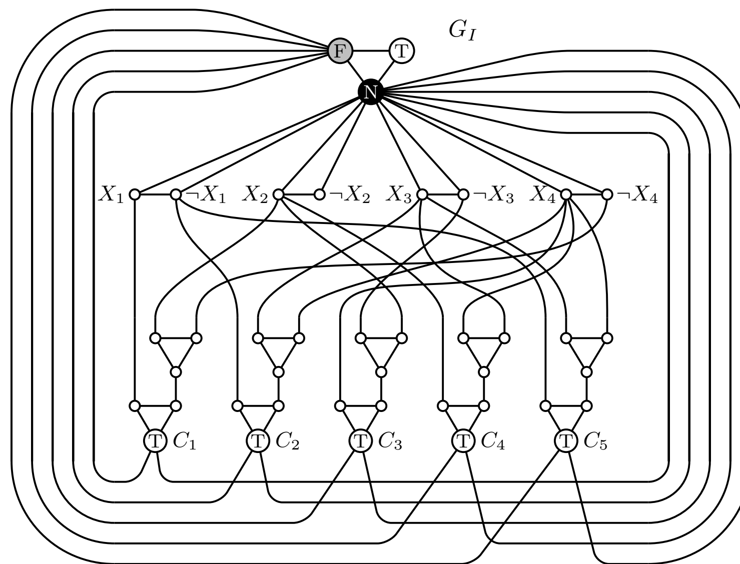
Please check that if the bottom vertex C_i of Δ is colored T, and each vertex α, β, γ is colored T or F, then in any 3-coloring of Δ (with colors T,F,N), *at least one of* α, β, γ *must also be colored* T. Conversely, if α, β, γ are assigned T/F colors and at least one of them is T, then this extends to a coloring of Δ with C_i colored T. Thus Δ is a type of "or" gate. If vertex C_i is T, then α or β or γ is T.

Now, the first clause C_1 of our instance I is $C_1 = (X_1 \vee X_2 \vee \neg X_4)$. Add to the previous graph G_I^C a copy of Δ for which the bottom vertex coincides with C_1, and the top three vertices (α, β, γ) attach to X_1, X_2 and $\neg X_4$, as indicated below.



Note that in any 3-coloring of this graph, at least one vertex X_1, X_2 and $\neg X_4$ must be colored T. This corresponds to an assignment of True/False to the literals in $C_1 = (X_1 \vee X_2 \vee \neg X_4)$ (and hence the variables X_1, X_2, X_4) that makes C_1 true.

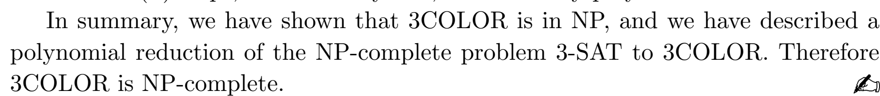
Finally, in a like fashion, insert copies of Δ for the remaining clauses $C_2, \ldots, C_5$, as follows. This is our final graph G_I that corresponds to the instance I.



By construction, the instance I of 3-SAT is satisfiable if and only if the graph G_I is 3-colorable. Although this was carried for a specific I, the recipe would apply to *any* instance I of 3-SAT. We thus have described a reduction of 3-SAT to 3-color.

But remember, our goal is a *polynomial* reduction of 3-SAT to 3-color. So it remains to verify that the process described above can be carried out in $f(n)$ steps, where f is a polynomial and n is the size of the 3-SAT instance I. Let's say n is the number of clauses in I. (This is somewhat arbitrary. We might also take n to be the total number of literals, which is three times the number of clauses.)

The first step of constructing G_I is to create its top triangle, which can be done in six steps (create the three vertices, add the three edges). There can be no more than $3n$ variables X_i, so the remainder of the truth-setting component can be built in no more than $2{\cdot}3n + 3{\cdot}3n$ steps (add two vertices for each variable, and then three edges for each variable). Then at the bottom we add n vertices labeled by the n clauses in I (n steps) and two connect edges from each one to N and F at the top ($2n$ steps). Finally, each "or" gate has five vertices and 10 edges, so inserting them one-per-clause takes $15n$ steps. Thus the total number of steps required to build G_I from I is no more than $6 + 2{\cdot}3n + 3{\cdot}3n + n + 2n + 15n = 6 + 36n$. So G_I is built in $O(n)$ steps, which is very fast, and certainly polynomial.



In summary, we have shown that 3COLOR is in NP, and we have described a polynomial reduction of the NP-complete problem 3-SAT to 3COLOR. Therefore 3COLOR is NP-complete. ✍