NP, NP completeness, and the Cook-Levin Theorem

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# NP, NP completeness, and the Cook-Levin Theorem

### 

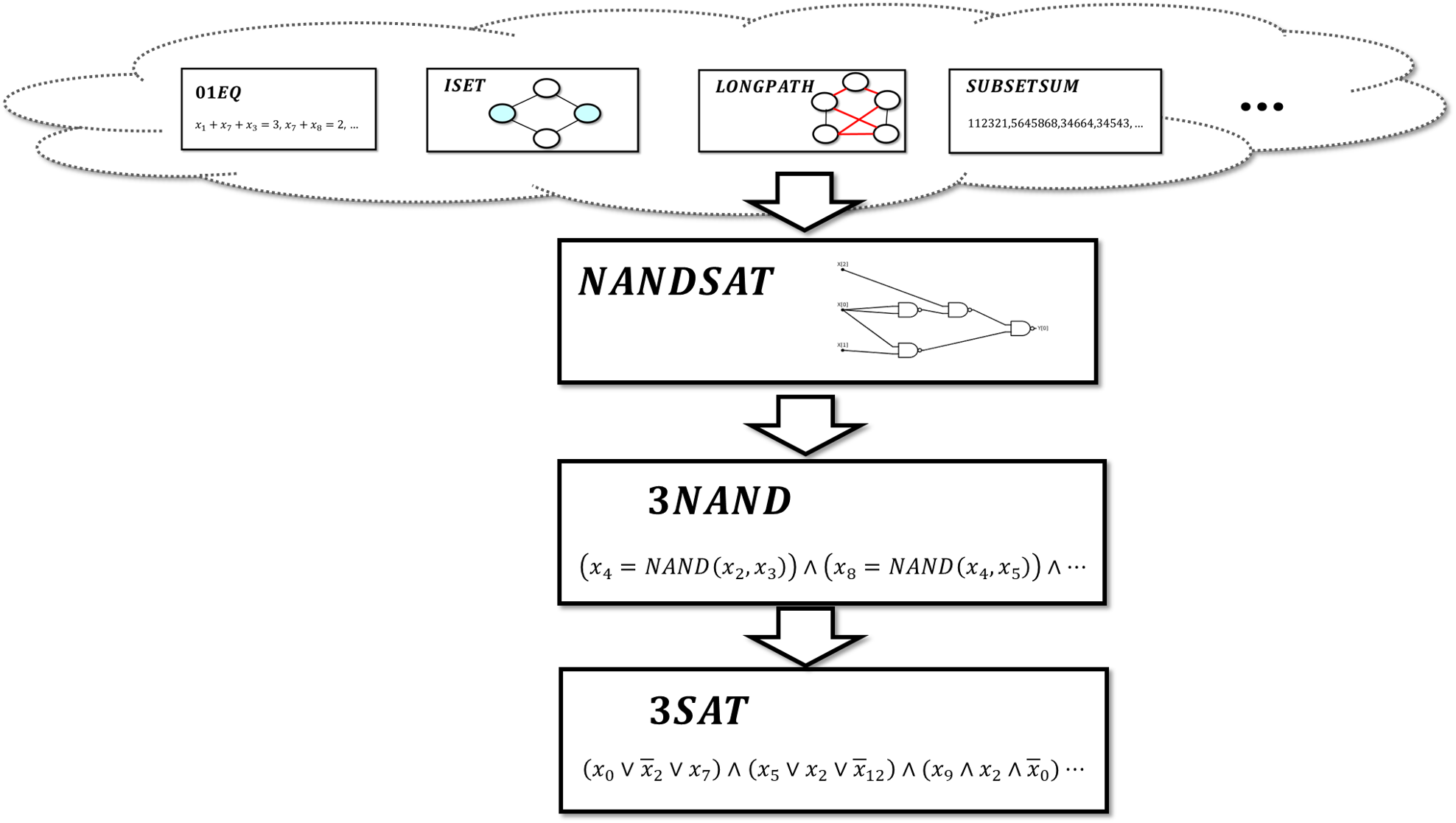
* Introduce the class capturing a great many important computational problems
* -completeness: evidence that a problem might be intractable.
* The vs problem.

*“In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually”*, Richard Karp, 1972

*“Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Twoness… 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?”* Eugene Lawler

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are *computationally equivalent* we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop.

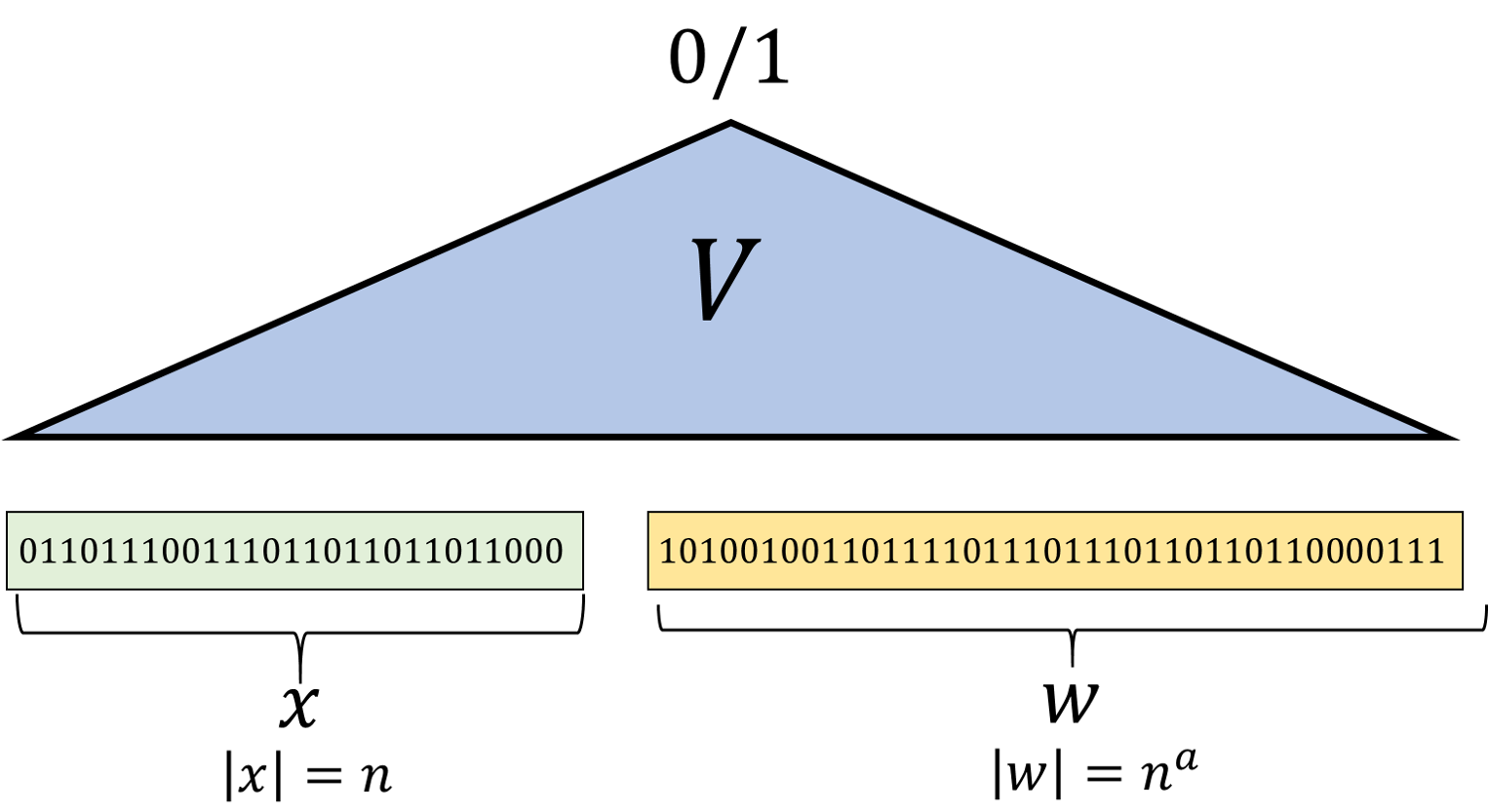
In fact, this result extends far beyond these particular problems. All of the problems we discussed in reductionchap, and a great many other problems, share the same commonality: they are all *search* problems, where the goal is to decide, given an instance , whether there exists a *solution* that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that *every* such search problem can be reduced to 3SAT.



Overview of the results of this chapter. We define to contain all decision problems for which a solution can be efficiently *verified*. The main result of this chapter is the *Cook Levin Theorem* (cook-levin-thm) which states that has a polynomial-time algorithm if and only if *every* problem in has a polynomial-time algorithm. Another way to state this theorem is that is  *complete*. We will prove the Cook-Levin theorem by defining the two intermediate problems and , proving that is complete, and then proving that .

## The class

To make the above precise, we will make the following mathematical definition. we define the class to contain all Boolean functions that correspond to a *search problem* of the form above. That is, a Boolean function is in if has the form that on input a string , if and only if there exists a “solution” string such that the pair satisfies some polynomial-time checkable condition. Formally, is defined as follows:



The class corresponds to problems where solutions can be *efficiently verified*. That is, this is the class of functions such that if there is a “solution” of length polynomial in that can be verified by a polynomial-time algorithm .

We say that is in if there exists some integer and such that and for every ,

In other words, for to be in , there needs to exist some polynomial-time computable verification function , such that if then there must exist (of length polynomial in ) such that , and if then for *every* such , . Since the existence of this string certifies that , is often referred to as a *certificate*, *witness*, or *proof* that .

See also NPdeffigfig for an illustration of NP-def. The name stands for “nondeterministic polynomial time” and is used for historical reasons; see the bibiographical notes. The string in NP-eq is sometimes known as a *solution*, *certificate*, or *witness* for the instance .

Show that the condition that in NP-def can be replaced by the condition that for some polynomial . That is, prove that for every , if and only if there is a polynomial-time Turing machine and a polynomial such that for every if and only if there exists with such that .

The “only if” direction (namely that if then there is an algorithm and a polynomial as above) follows immediately from NP-def by letting . For the “if” direction, the idea is that if a string is of size at most for degree polynomial , then there is some such that for all , . Hence we can encode by a string of exactly length by padding it with and an appropriate number of zeroes. Hence if there is an algorithm and polynomial as above, then we can define an algorithm that does the following on input with and :

* If then ignores and enumerates over all of length at most and outputs if there exists such that . (Since , this only takes a constant number of steps.)
* If then “strips out” the padding by dropping all the rightmost zeroes from until it reaches out the first (which it drops as well) and obtains a string . If tnen outputs .

Since runs in polynomial time, runs in polynomial time as well, and by definition for every , there exists such that if and only if there exists with such that .

The definition of means that for every and string , if and only if there is a *short and efficiently verifiable proof* of this fact. That is, we can think of the function in NP-def as a *verifier* algorithm, similar to what we’ve seen in godelproofdef. The verifier checks whether a given string is a valid proof for the statement “”. Essentially all proof systems considered in mathematics involve line-by-line checks that can be carried out in polynomial time. Thus the heart of is asking for statements that have *short* (i.e., polynomial in the size of the statements) proof. Indeed, as we will see in #chappvsnp, Kurt Gödel phrased the question of whether as asking whether “the mental work of a mathematician [in proving theorems] could be completely replaced by a machine”.

NP-def is *asymmetric* in the sense that there is a difference between an output of and an output of . You should make sure you understand why this definition does *not* guarantee that if then the function (i.e., the map ) is in as well.

In fact, it is believed that there do exist functions such that but . For example, as shown below, , but the function that on input a 3CNF formula outputs if and only if is *not* satisfiable is not known (nor believed) to be in . This is in contrast to the class which *does* satisfy that if then is in as well.

### Examples of functions in

We now present some examples of functions that are in the class . We start with the canonical example of the function.

is in since for every -variable formula , if and only if there exists a satisfying assignment such that , and we can check this condition in polynomial time.

The above reasoning explains why is in , but since this is our first example, we will now belabor the point and expand out in full formality the precise representation of the witness and the algorithm that demonstrate that is in . Since demonstrating that functions are in is fairly straightforward, in future cases we will not use as much detail, and the reader can also feel free to skip the rest of this example.

Using NPalternativeex, it is OK if witness is of size at most polynomial in the input length , rather than of precisely size for some integer . Specifically, we can represent a 3CNF formula with variables and clauses as a string of length , since every one of the clauses involves three variables and their negation, and the identity of each variable can be represented using . We assume that every variable participates in some clause (as otherwise it can be ignored) and hence that , which in particular means that the input length is at least as large as and .

We can represent an assignment to the variables using a -length string . The following algorithm checks whether a given satisfies the formula :

INPUT: 3CNF formula $\varphi$ on $k$ variables and with $m$ clauses, string $w \in \{0,1\}^k$   
OUTPUT: $1$ iff $w$ satisfies $\varphi$   
  
For{$j \in [m]$}  
 Let $\ell\_1 \vee \ell\_2 \vee \ell\_j$ be the $j$-th clause of $\varphi$   
 If{$w$ violates all three literals}  
 return $0$  
 Endif  
Endfor  
Return $1$

threesatverifieralg takes time to enumerate over all clauses, and will return if and only if satisfies all the clauses.

Here are some more examples for problems in . For each one of these problems we merely sketch how the witness is represented and why it is efficiently checkable, but working out the details can be a good way to get more comfortable with NP-def:

* is in since for every -variable instance of quadratic equations , if and only if there exists an assignment that satisfies . We can check the condition that satisfies in polynomial time by enumerating over all the equations in , and for each such equation , plug in the values of and verify that is satisfied.
* is in since for every graph and integer , if and only if there exists a set of vertices that contains no pair of neighbors in . We can check the condition that is an independent set of size in polynomial time by first checking that and then enumerating over all edges in , and for each such edge verify that either or .
* is in since for every graph and integer , if and only if there exists a simple path in that is of length at least . We can check the condition that is a simple path of length in polynomial time by checking that it has the form where each is a vertex in , no is repeated, and for every , the edge is present in the graph.
* is in since for every graph and integer , if and only if there exists a cut in that cuts at least edges. We can check that condition that is a cut of value at least in polynomial time by checking that is a subset of ’s vertices and enumerating over all the edges of , counting those edges such that and or vice versa.

### Basic facts about

The definition of is one of the most important definitions of this book, and is worth while taking the time to digest and internalize. The following solved exercises establish some basic properties of this class. As usual, I highly recommend that you try to work out the solutions yourself.

Prove that .

Suppose that . Define the following function : iff and . ( outputs on all other inputs.) Since we can clearly compute in polynomial time as well.

Let be some string. If then . On the other hand, if then for every , . Therefore, setting , we see that satisfies NP-eq, and establishes that .

People sometimes think that stands for “non polynomial time”. As PinNP shows, this is far from the truth, and in fact every polynomial-time computable function is in as well.

If is in it certainly does *not* mean that is hard to compute (though it does not, as far as we know, necessarily mean that it’s easy to compute either). Rather, it means that is *easy to verify*, in the technical sense of NP-def.

Prove that .

Suppose that and let be the polynomial-time computable function that satisfies NP-eq and the corresponding constant. Then given every , we can check whether in time by enumerating over all the strings and checking whether , in which case we return . If for every such then we return . By construction, the algorithm above will run in time at most exponential in its input length and by the definition of it will return for every .

PinNP and NPinEXP together imply that

The time hierarchy theorem (time-hierarchy-thm) implies that and hence at least one of the two inclusions or is *strict*. It is believed that both of them are in fact strict inclusions. That is, it is believed that there are functions in that cannot be computed in polynomial time (this is the conjecture) and that there are functions in for which we cannot even efficiently *certify* that for a given input . One function that is believed to lie in is the function defined as for every 3CNF formula . The conjecture that is known as the “” conjecture. It implies the conjecture (see npconppnpex).

We have previously informally equated the notion of with being “no harder than ” and in particular have seen in reductionsandP that if and , then as well. The following exercise shows that if then it is also “no harder to verify” than . That is, regardless of whether or not it is in , if has the property that solutions to it can be efficiently verified, then so does .

Let . Show that if and then .

Suppose that is in and in particular there exists and such that for every , . Suppose also that and so in particular there is a -time computable function such that for all . Define to be a Turing Machine that on input a pair computes and returns if and only if and . Then runs in polynomial time, and for every , iff there exists of size which is at most polynomial in such that , hence demonstrating that .

## From to 3SAT: The Cook-Levin Theorem

We have seen everal example of problems for which we do not know if their best algorithm is polynomial or exponential, but we can show that they are in . That is, we don’t know if they are easy to *solve*, but we do know that it is easy to *verify* a given solution. There are many, many, *many*, more examples of interesting functions we would like to compute that are easily shown to be in . What is quite amazing is that if we can solve 3SAT then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

### 

For every , .

We will soon show the proof of cook-levin-thm, but note that it immediately implies that , , and all reduce to . Combining it with the reductions we’ve seen in reductionchap, it implies that all these problems are *equivalent!* For example, to reduce to , we can first reduce to using cook-levin-thm and use the reduction we’ve seen in longpaththm from to . That is, since , cook-levin-thm implies that , and longpaththm implies that , which by the transitivity of reductions (transitiveex) means that . Similarly, since , we can use cook-levin-thm and quadeq-thm to show that , concluding that and are computationally equivalent.

There is of course nothing special about and here: by combining cook-levin-thm with the reductions we saw, we see that just like , *every* reduces to , and the same is true for and . All these problems are in some sense “the hardest in ” since an efficient algorithm for any one of them would imply an efficient algorithm for *all* the problems in . This motivates the following definition:

Let . We say that is  *hard* if for every , .

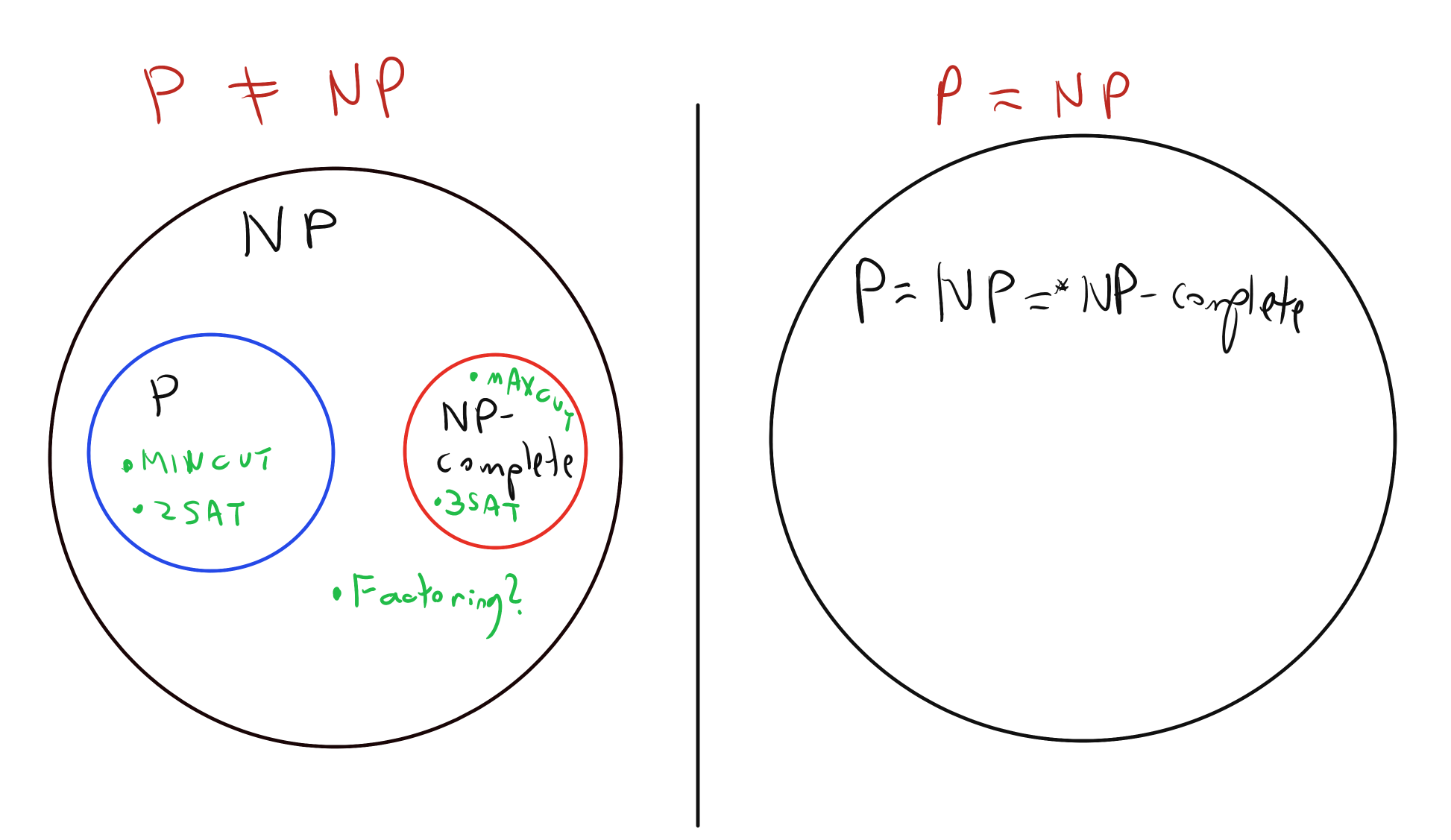
We say that is  *complete* if is hard and .

The Cook-Levin Theorem (cook-levin-thm) can be rephrased as saying that is hard, and since it is also in , this means that is complete. Together with the reductions of reductionchap, cook-levin-thm shows that despite their superficial differences, 3SAT, quadratic equations, longest path, independent set, and maximum cut, are all -complete. Many thousands of additional problems have been shown to be -complete, arising from all the sciences, mathematics, economics, engineering and many other fields. (For a few examples, see [this Wikipedia page](https://goo.gl/NomnoU) and [this website](https://goo.gl/nfJHWv).)

If a *single* -complete has a polynomial-time algorithm, then there is such an algorithm for every decision problem that corresponds to the existence of an *efficiently-verifiable* solution.

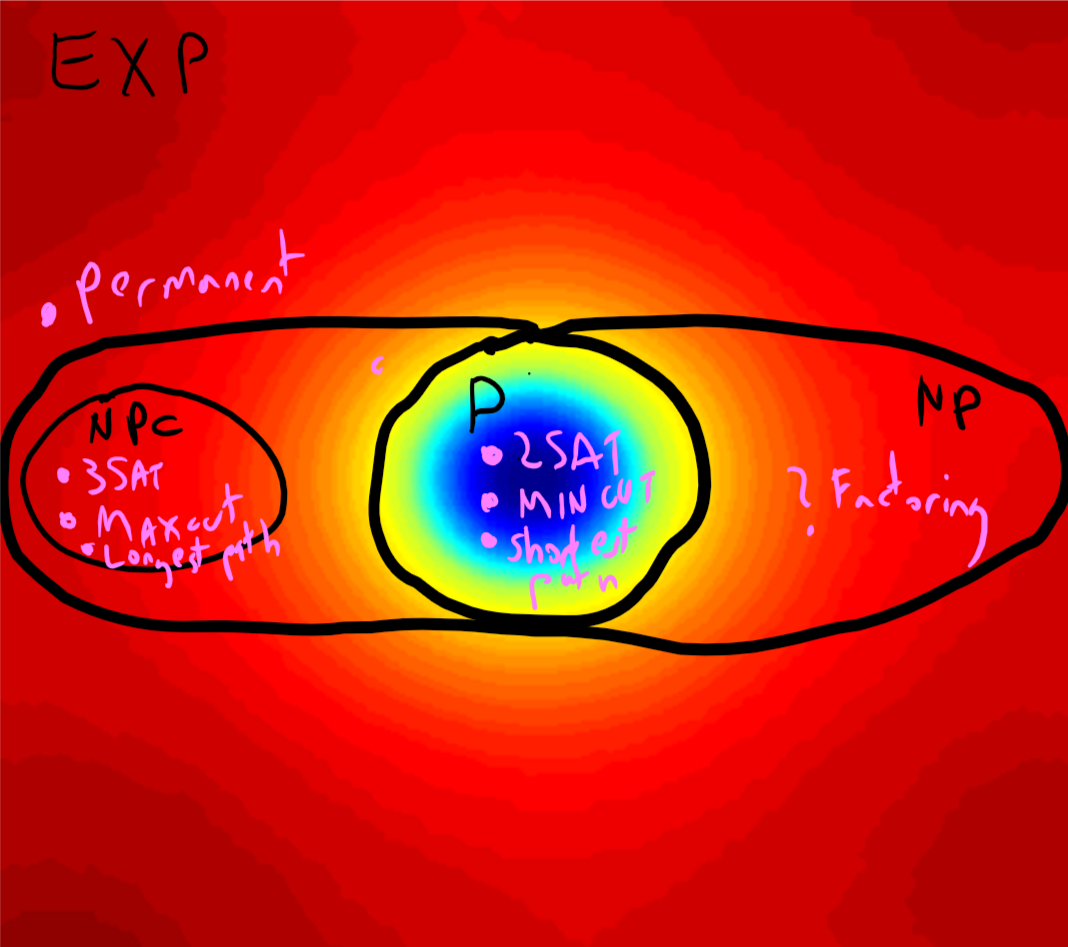
### What does this mean?

As we’ve seen in PinNP, . *The* most famous conjecture in Computer Science is that this containment is *strict*. That is, it is widely conjectured that . One way to refute the conjecture that is to give a polynomial-time algorithm for even a single one of the -complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for any of them has been found, supports that conjecture that indeed . In fact, for many of these problems (including all the ones we mentioned above), we don’t even know of a -time algorithm! However, to the frustration of computer scientists, we have not yet been able to prove that or even rule out the existence of an -time algorithm for 3SAT. Resolving whether or not is known as the  [vs problem](https://en.wikipedia.org/wiki/P_versus_NP_problem). A million-dollar prize has been [offered](http://www.claymath.org/millennium-problems/p-vs-np-problem) for the solution of this problem, a [popular book](https://www.amazon.com/dp/B00BKZYGUY) has been written, and every year a new paper comes out claiming a proof of or , only to wither under scrutiny.



The world if (left) and (right). In the former case the set of -complete problems is disjoint from and Ladner’s theorem shows that there exist problems that are neither in nor are -complete. (There are remarkably few natural candidates for such problems, with some prominent examples being decision variants of problems such as integer factoring, lattice shortest vector, and finding Nash equilibria.) In the latter case that the notion of -completeness loses its meaning, as essentially all functions in (save for the trivial constant zero and constant one functions) are -complete.

One of the mysteries of computation is that people have observed a certain empirical “zero-one law” or “dichotomy” in the computational complexity of natural problems, in the sense that many natural problems are either in (often in or ), or they are are hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as . However, it is believed that there exist problems in that are neither in nor are -complete, and in fact a result known as “Ladner’s Theorem” shows that if then this is indeed the case (see also ladner-ex and PNPscenariosfig).



A rough illustration of the (conjectured) status of problems in exponential time. Darker colors correspond to higher running time, and the circle in the middle is the problems in . is a (conjectured to be proper) superclass of and the -complete problems (or for short) are the “hardest” problems in , in the sense that a solution for one of them implies a solution for all other problems in . It is conjectured that all the -complete problems require at least time to solve for a constant , and many require time. The *permanent* is not believed to be contained in though it is -hard, which means that a polynomial-time algorithm for it implies that .

### The Cook-Levin Theorem: Proof outline

We will now prove the Cook-Levin Theorem, which is the underpinning to a great web of reductions from 3SAT to thousands of problems across great many fields. Some problems that have been shown to be -complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lattice, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of 3SAT, and through the Cook-Levin Theorem, to all problems in .

To prove cook-levin-thm we need to show that for every . We will do so in three stages. We define two intermediate problems: and . We will shortly show the definitions of these two problems, but cook-levin-thm will follow from combining the following three results:

1. is hard (nand-thm).
2. (threenand-thm).
3. (threenand-sat-thm).

By the transitivity of reductions, it will follow that for every ,

hence establishing cook-levin-thm.

We will prove these three results nand-thm, threenand-thm and threenand-sat-thm one by one, providing the requisite definitions as we go along.

## The Problem, and why it is hard.

The function is defined as follows:

* The **input** to is a string representing a NAND-CIRC program (or equivalently, a circuit with NAND gates).
* The **output** of on input is if and only if there exists a string (where is the number of inputs to ) such that .

Prove that .

We have seen that the circuit (or straightline program) evaluation problem can be computed in polynomial time. Specifically, given a NAND-CIRC program of lines and inputs, and , we can evaluate on the input in time which is polynomial in and hence verify whether or not .

We now prove that is hard.

### 

is hard.

### 

The proof closely follows the proof that (non-uniform-thm , see also unrollloopsec). Specifically, if then there is a polynomial time Turing machine and positive integer such that for every , iff there is some such that . The proof that gave us way (via “unrolling the loop”) to come up in polynomial time with a Boolean circuit on inputs that computes the function . We can then translate into an equivalent NAND circuit (or NAND-CIRC program) . We see that there is a string such that if and only if there is such satisfying which (by definition) happens if and only if . Hence the translation of into the circuit is a reduction showing .

### 

The proof is a little bit technical but ultimately follows quite directly from the definition of , as well as the ability to “unroll the loop” of NAND-TM programs as discussed in unrollloopsec. If you find it confusing, try to pause here and think how you would implement in your favorite programming language the function unroll which on input a NAND-TM program and numbers outputs an -input NAND-CIRC program of lines such that for every input , if halts on within at most steps and outputs , then .

Let . To prove nand-thm we need to give a polynomial-time computable function that will map every to a NAND-CIRC program such that .

Let be such a string and let be its length. By NP-def there exists and positive such that if and only if there exists satisfying .

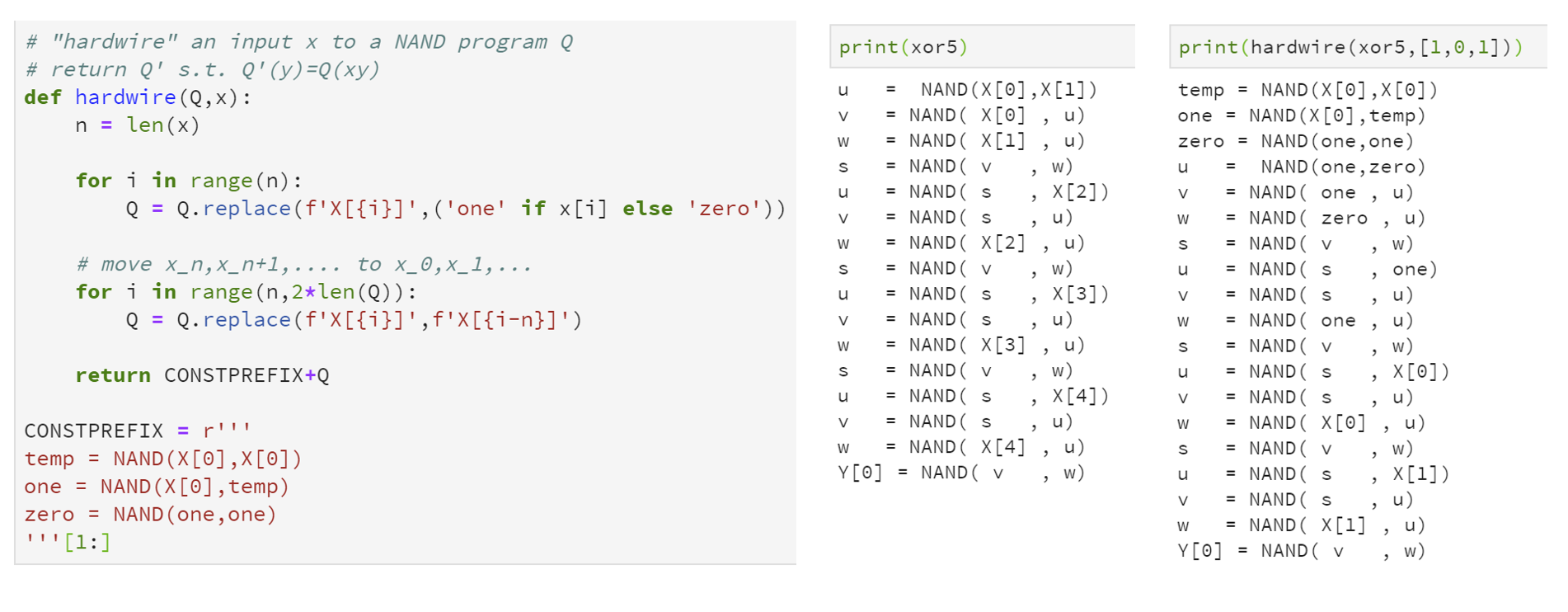
Let . Since there is some NAND-TM program that computes on inputs of the form with and in at most time for some constant . Using our “unrolling the loop NAND-TM to NAND compiler” of nand-compiler, we can obtain a NAND-CIRC program that has inputs and at most lines such that for every and .

We can then use a simple “hardwiring” technique, reminiscent of hardwiringrem to map into a circuit/NAND-CIRC program on inputs such that for every .

**CLAIM:** There is a polynomial-time algorithm that on input a NAND-CIRC program on inputs and , outputs a NAND-CIRC program such that for every , .

**PROOF OF CLAIM:** We can do so by adding a few lines to ensure that the variables zero and one are and respectively, and then simply replacing any reference in to an input with the corresponding value based on . See hardwiringfig for an implementation of this reduction in Python.

Our final reduction maps an input , into the NAND-CIRC program obtained above. By the above discussion, this reduction runs in polynomial time. Since we know that if and only if there exists such that , this means that if and only if , which is what we wanted to prove.



Given an -line NAND-CIRC program that has inputs and some , we can transform into a line NAND-CIRC program that computes the map for by simply adding code to compute the zero and one constants, replacing all references to X[] with either zero or one depending on the value of , and then replacing the remaining references to X[] with X[]. Above is Python code that implements this transformation, as well as an example of its execution on a simple program.

## The problem

The problem is defined as follows:

* The **input** is a logical formula on a set of variables which is an AND of constraints of the form .
* The **output** is if and only if there is an input that satisfies all of the constraints.

For example, the following is a formula with variables and constraints:

In this case since the assignment satisfies it. Given a formula on variables and an assignment , we can check in polynomial time whether , and hence . We now prove that is hard:

### 

.

### 

To prove threenand-thm we need to give a polynomial-time map from every NAND-CIRC program to a 3NAND formula such that there exists such that if and only if there exists satisfying . For every line of , we define a corresponding variable of . If the line has the form foo = NAND(bar,blah) then we will add the clause where and are the last lines in which bar and blah were written to. We will also set variables corresponding to the input variables, as well as add a clause to ensure that the final output is . The resulting reduction can be implemented in about a dozen lines of Python, see nandsattothreenandfig.



Python code to reduce an instance of to an instance of . In the example above we transform the NAND-CIRC program xor5 which has input variables and lines, into a formula that has variables and clauses. Since xor5 outputs on the input , there exists an assignment to such that and evaluates to *true* on .

To prove threenand-thm we need to give a reduction from to . Let be a NAND-CIRC program with inputs, one output, and lines. We can assume without loss of generality that contains the variables one and zero as usual.

We map to a formula as follows:

* has variables .
* The first variables will corresponds to the inputs of . The next variables will correspond to the lines of .
* For every , if the -th line of the program is foo = NAND(bar,blah) then we add to the constraint where and correspond to the last lines in which the variables bar and blah (respectively) were written to. If one or both of bar and blah was not written to before then we use instead of the corresponding value or in the constraint, where is the line in which zero is assigned a value. If one or both of bar and blah is an input variable X[i] then we use in the constraint.
* Let be the last line in which the output y\_0 is assigned a value. Then we add the constraint where is as above the last line in which zero is assigned a value. Note that this is effectively the constraint .

To complete the proof we need to show that there exists s.t. if and only if there exists that satisfies all constraints in . We now show both sides of this equivalence.

**Part I: Completeness.** Suppose that there is s.t. . Let be defined as follows: for , and for equals the value that is assigned in the -th line of when executed on . Then by construction satisfies all of the constraints of (including the constraint that since .)

**Part II: Soundness.** Suppose that there exists satisfying . Soundness will follow by showing that (and hence in particular there exists , namely , such that ). To do this we will prove the following claim : for every , equals the value assigned in the -th step of the execution of the program on . Note that because satisfies the constraints of , is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable y\_0 in the execution of on is equal to . To prove suppose, towards a contradiction, that it is false, and let be the smallest number such that is *not* equal to the value assigned in the -th step of the execution of on . But since satisfies the constraints of , we get that where (by the assumption above that is *smallest* with this property) these values *do* correspond to the values last assigned to the variables on the righthand side of the assignment operator in the -th line of the program. But this means that the value assigned in the -th step is indeed simply the NAND of and , contradicting our assumption on the choice of .



A instance that is obtained by taking a NAND-TM program for computing the function, unrolling it to obtain a instance, and then composing it with the reduction of threenand-thm.

## From to

The final step in the proof of cook-levin-thm is the following:

### 

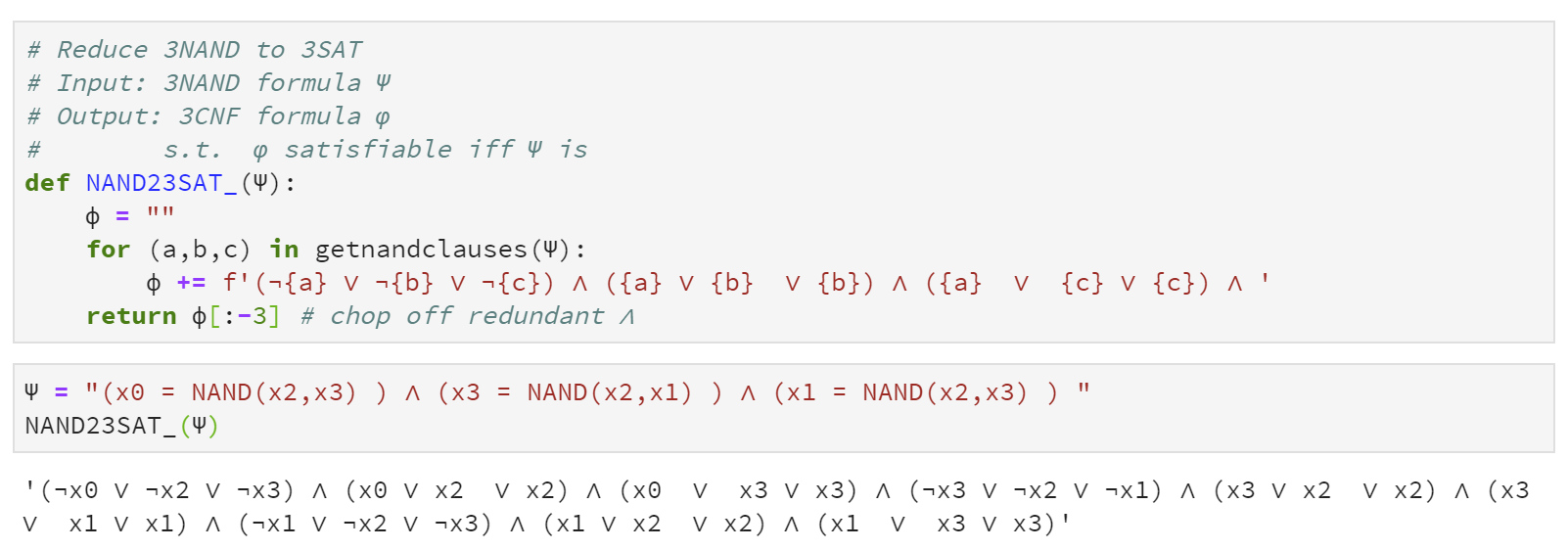
.

### 

To prove threenand-sat-thm we need to map a 3NAND formula into a 3SAT formula such that is satisfiable if and only if is. The idea is that we can transform every NAND constraint of the form into the AND of ORs involving the variables and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below.

### 

It is a good exercise for you to try to find a 3CNF formula on three variables such that is true if and only if . Once you do so, try to see why this implies a reduction from to , and hence completes the proof of threenand-sat-thm

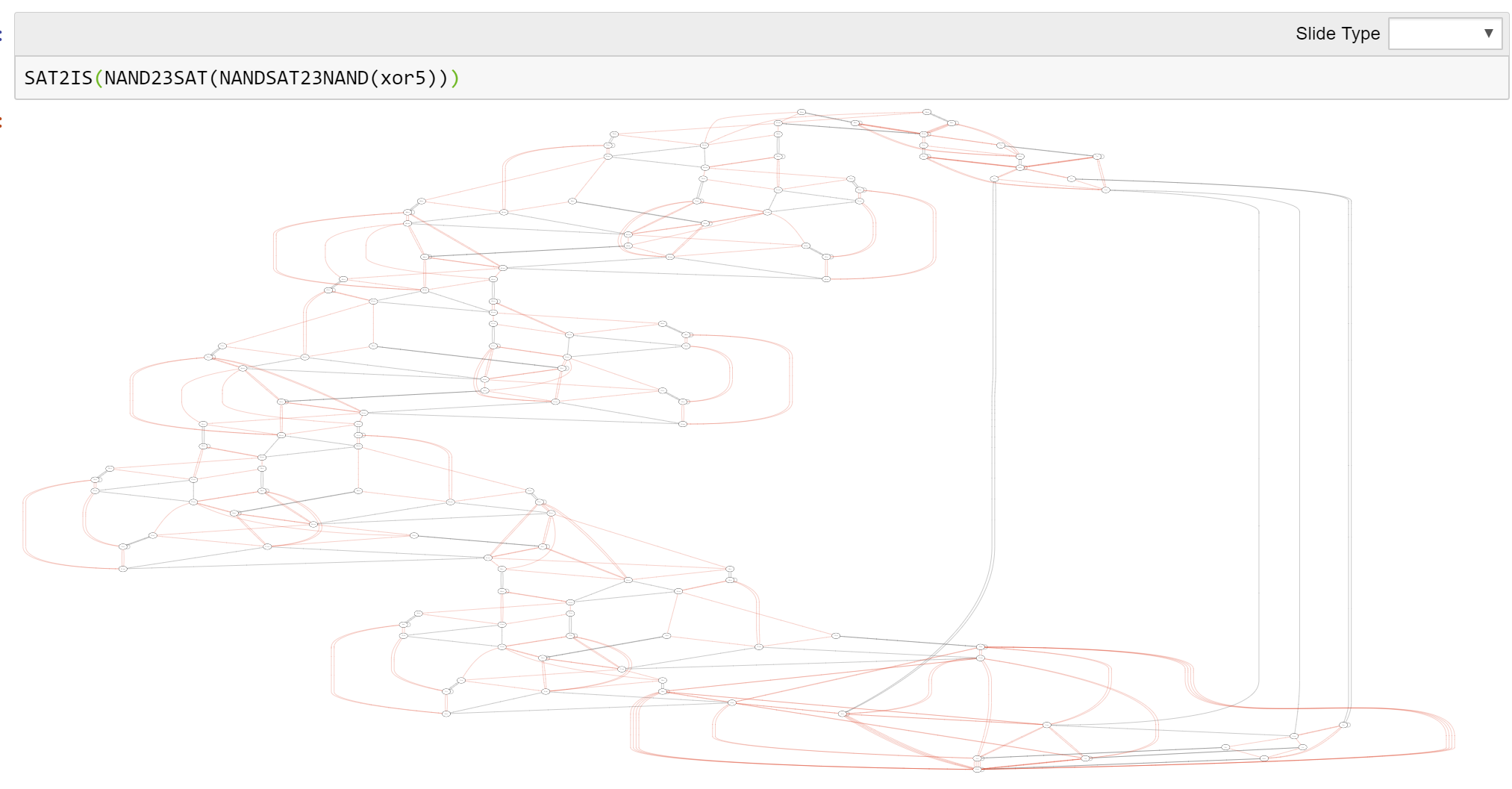


Code and example output for the reduction given in threenand-sat-thm of to .

The constraint

is satisfied if whenever . By going through all cases, we can verify that eq:NANDconstraint is equivalent to the constraint

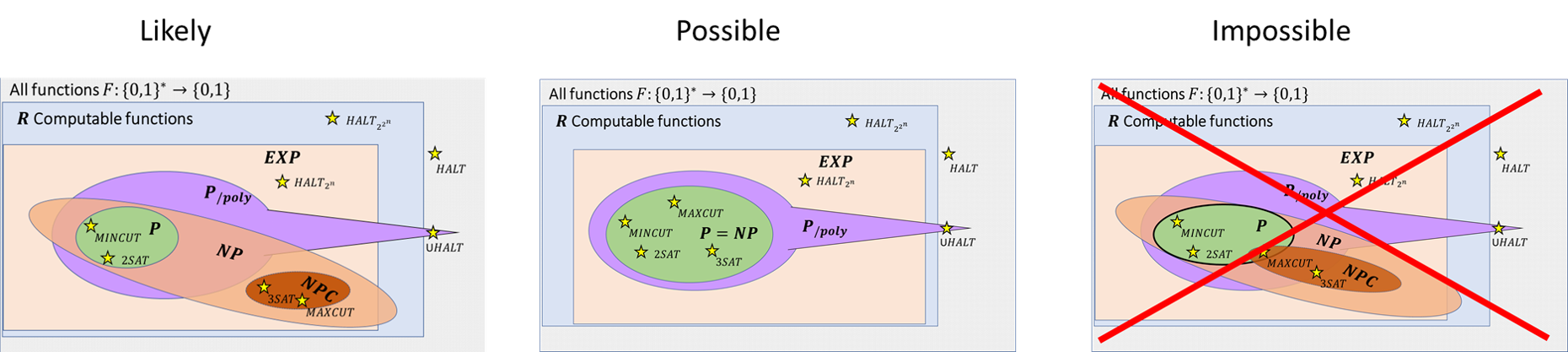
Indeed if then the first constraint of eq:CNFNAND is only true if . On the other hand, if either of or equals then unless either the second or third constraints will fail. This means that, given any 3NAND formula over variables , we can obtain a 3SAT formula over the same variables by replacing every constraint of with three constraints as in eq:CNFNAND.[[1]](#footnote-74) Because of the equivalence of eq:NANDconstraint and eq:CNFNAND, the formula satisfies that for every assignment to the variables. In particular is satisfiable if and only if is, thus completing the proof.



An instance of the *independent set* problem obtained by applying the reductions starting with the xor5 NAND-CIRC program.

## Wrapping up

We have shown that for every function in , , and so is -hard. Since in reductionchap we saw that , , and , all these problems are -hard as well. Finally, since all the aforementioned problems are in , they are all in fact -complete and have equivalent complexity. There are thousands of other natural problems that are -complete as well. Finding a polynomial-time algorithm for any one of them will imply a polynomial-time algorithm for all of them.



We believe that and all complete problems lie outside of , but we cannot rule out the possiblity that . However, we can rule out the possiblity that *some* -complete problems are in and other do not, since we know that if even one -complete problem is in then . The relation between and is not known though it can be shown that if one -complete problem is in then .

### 

* Many of the problems for which we don’t know polynomial-time algorithms are -complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for *all* of them.
* It is conjectured that which means that we believe that polynomial-time algorithms for these problems are not merely *unknown* but are *nonexistent*.
* While an -hardness result means for example that a full-fledged “textbook” solution to a problem such as MAX-CUT that is as clean and general as the algorithm for MIN-CUT probably does not exist, it does not mean that we need to give up whenever we see a MAX-CUT instance. Later in this course we will discuss several strategies to deal with -hardness, including *average-case complexity* and *approximation algorithms*.

## Exercises

### 

Prove that if there is no time algorithm for then there is some such that and is not complete.[[2]](#footnote-83)

Let be the function that on input a 3CNF formula return . Prove that if then . See footnote for hint.[[3]](#footnote-84)

Define to be the following function: the input is a CNF formula where each clause is the OR of one to three variables (*without negations*), and a number . For example, the following formula can be used for a valid input to : . The output if and only if there exists a satisfying assignment to in which exactly of the variables get the value . For example for the formula above since the assignment satisfies all the clauses. However since there is no single variable appearing in all clauses.

Prove that is -complete.

In the *employee recruiting problem* we are given a list of potential employees, each of which has some subset of potential skills, and a number . We need to assemble a team of employees such that for every skill there would be one member of the team with this skill.

For example, if Alice has the skills “C programming”, “NAND programming” and “Solving Differential Equations”, Bob has the skills “C programming” and “Solving Differential Equations”, and Charlie has the skills “NAND programming” and “Coffee Brewing”, then if we want a team of two people that covers all the four skills, we would hire Alice and Charlie.

Define the function s.t. on input the skills of all potential employees (in the form of a sequence of lists , each containing distinct numbers between and ), and a number , if and only if there is a subset of potential employees such that for every skill in , there is an employee in that has the skill .

Prove that is complete.

Prove that the “balanced variant” of the maximum cut problem is -complete, where this is defined as where for every graph and , if and only if there exists a cut in cutting at least edges such that .

Let be the following function: On input a list of regular expressions (represented as strings in some standard way), output if and only if there is a single string that matches all of them. Prove that is -hard.

## Bibliographical notes

Aaronson’s 120 page survey [@aaronson2016p] is a beautiful and extensive exposition to the vs problem, its importance and status. See also as well as Chapter 3 in Wigderson’s excellent book [@wigderson2017mathematics]. Johnson [@johnson2012brief] gives a survey of the historical development of the theory of completeness. The following [web page](https://goo.gl/bFHsd9) keeps a catalog of failed attempts at settling vs . At the time of this writing, it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that and about 50 claim to prove that .

Eugene Lawler’s quote on the “mystical power of twoness” was taken from the wonderful book “The Nature of Computation” by Moore and Mertens. See also [this memorial essay on Lawler](https://pure.tue.nl/ws/files/1506049/511307.pdf) by Lenstra.

1. The resulting formula will have some of the OR’s involving only two variables. If we wanted to insist on each formula involving three distinct variables we can always add a “dummy variable” and include it in all the OR’s involving only two variables, and add a constraint requiring this dummy variable to be zero. [↑](#footnote-ref-74)
2. **Hint:** Use the function that on input a formula and a string of the form , outputs if and only if is satisfiable and . [↑](#footnote-ref-83)
3. *Hint:* Prove and then use the fact that *is* closed under complement. [↑](#footnote-ref-84)