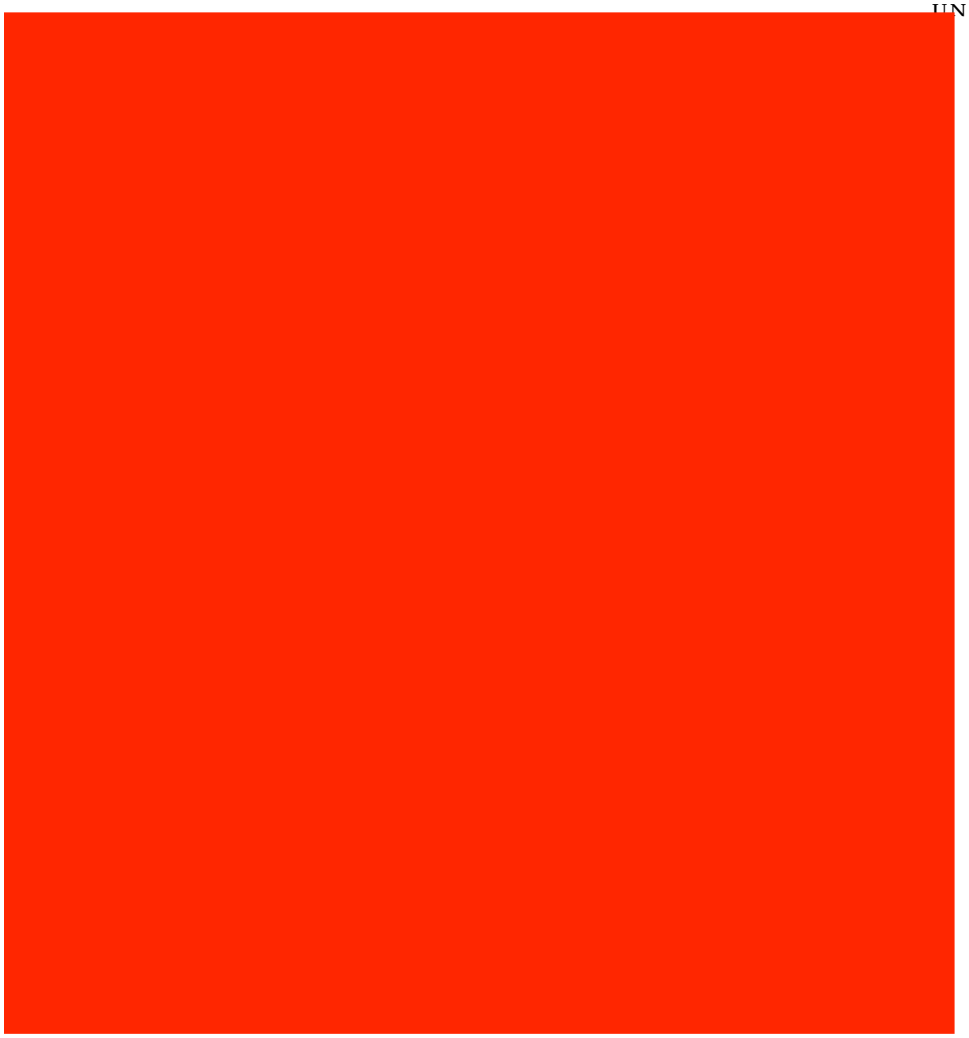
TheoryPrep Readings

May 2020

Selected from Mathematics for Computer Science (Lehman, Leighton, Meyer 2017), Building Blocks for Computer Science (Fleck 2013), and Introduction to Theoretical Computer Science (Barak, 2020)



8.3 THE HALTING PROBLEM

This covers the same material as the video so if you want to skip UNTIL THE GREEN ARROW that's fine. If you felt like another pass at the material would be good, seeing alternative presentations is a great way to build understanding!

Theorem 8.6 shows that there is *some* function that cannot be computed. But is this function the equivalent of the "tree that falls in the forest with no one hearing it"? That is, perhaps it is a function that no one actually *wants* to compute. It turns out that there are natural uncomputable functions:

Theorem 8.7 — Uncomputability of Halting function. Let $HALT : \{0,1\}^* \to \{0,1\}$ be the function such that for every string $M \in \{0,1\}^*$, $HALT(M,x) = 1$ if Turing machine M halts on the input x and $HALT(M,x) = 0$ otherwise. Then $HALT$ is not computable.

8.4 REDUCTIONS

The Halting problem turns out to be a linchpin of uncomputability, in the sense that Theorem 8.7 has been used to show the uncomputability of a great many interesting functions. We will see several examples of such results in this chapter and the exercises, but there are many more such results (see Fig. 8.6).

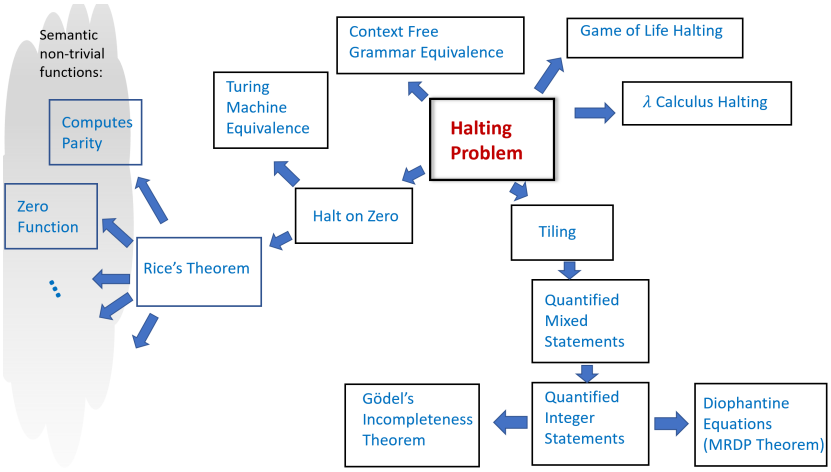


Figure 8.6: Some uncomputability results. An arrow from problem X to problem Y means that we use the uncomputability of X to prove the uncomputability of Y by reducing computing X to computing Y. All of these results except for the MRDP Theorem appear in either the text or exercises. The Halting Problem *HALT* serves as our starting point for all these uncomputability results as well as many others.

The idea behind such uncomputability results is conceptually simple but can at first be quite confusing. If we know that *HALT* is uncomputable, and we want to show that some other function *BLAH* is uncomputable, then we can do so via a *contrapositive* argument (i.e., proof by contradiction). That is, we show that **if** there exists a Turing machine that computes *BLAH* **then** there exists a Turing machine that computes *HALT*. (Indeed, this is exactly how we showed that *HALT* itself is uncomputable, by reducing this fact to the uncomputability of the function F^* from Theorem 8.6.)

For example, to prove that *BLAH* is uncomputable, we could show that there is a computable function $R : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every pair M and x, $HALT(M,x) = BLAH(R(M,x))$. The existence of such a function R implies that **if** *BLAH* was computable **then** *HALT* would be computable as well, hence leading to a contradiction! The confusing part about reductions is that we are assuming something we *believe* is false (that *BLAH* has an algorithm) to derive something that we *know* is false (that *HALT* has an algorithm). Michael Sipser describes such results as having the form *"If pigs could whistle then horses could fly"*.

A reduction-based proof has two components. For starters, since we need R to be computable, we should describe the algorithm to compute it. The algorithm to compute R is known as a *reduction* since the transformation R modifies an input to *HALT* to an input to *BLAH*, and hence *reduces* the task of computing *HALT* to the task of computing *BLAH*. The second component of a reduction-based proof is the *analysis* of the algorithm R: namely a proof that R does indeed satisfy the desired properties.

Reduction-based proofs are just like other proofs by contradiction, but the fact that they involve hypothetical algorithms that don't really exist tends to make reductions quite confusing. The one silver lining is that at the end of the day the notion of reductions is mathematically quite simple, and so it's not that bad even if you have to go back to first principles every time you need to remember what is the direction that a reduction should go in.

Remark 8.9 — Reductions are algorithms. A reduction is an *algorithm*, which means that, as discussed in Remark 0.3, a reduction has three components:

- **Specification (what):** In the case of a reduction from *HALT* to *BLAH*, the specification is that function $R : \{0,1\}^* \to \{0,1\}^*$ should satisfy that $HALT(M,x) = BLAH(R(M,x))$ for every Turing machine M and input x. In general, to reduce a function F to G, the reduction should satisfy $F(w) = G(R(w))$ for every input w to F.
- **Implementation (how):** The algorithm's description: the precise instructions how to transform an input w to the output $R(w)$.
- **Analysis (why):** A *proof* that the algorithm meets the specification. In particular, in a reduction from F to G this is a proof that for every input w, the output y of the algorithm satisfies that $F(w) = G(y)$.

8.4.1 Example: Halting on the zero problem

Here is a concrete example for a proof by reduction. We define the function $HALTONZERO : \{0,1\}^* \to \{0,1\}$ as follows. Given any string M, $HALTONZERO(M) = 1$ if and only if M describes a Turing machine that halts when it is given the string 0 as input. A priori *HALTONZERO* seems like a potentially easier function to compute than the full-fledged *HALT* function, and so we could perhaps hope that it is not uncomputable. Alas, the following theorem shows that this is not the case:

Theorem 8.10 — Halting without input. *HALTONZERO* is uncomputable.

P

The proof of Theorem 8.10 is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. In particular, try to think of what a reduction from *HALT* to *HALTONZERO* would look like. Doing so is an excellent way to get some initial comfort with the notion of proofs by reduction, which a technique we will be using time and again in this book.

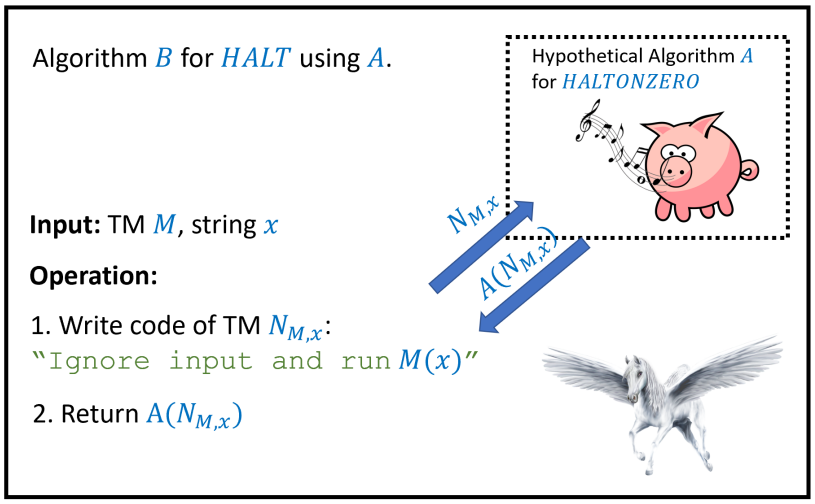


Figure 8.7: To prove Theorem 8.10, we show that *HALTONZERO* is uncomputable by giving a *reduction* from the task of computing *HALT* to the task of computing *HALTONZERO*. This shows that if there was a hypothetical algorithm A computing *HALTONZERO*, then there would be an algorithm B computing *HALT*, contradicting Theorem 8.7. Since neither A nor B actually exists, this is an example of an implication of the form "if pigs could whistle then horses could fly".

Proof of Theorem 8.10. The proof is by reduction from *HALT*, see Fig. 8.7. We will assume, towards the sake of contradiction, that *HALTONZERO* is computable by some algorithm A, and use this hypothetical algorithm A to construct an algorithm B to compute *HALT*, hence obtaining a contradiction to Theorem 8.7. (As discussed in Big Idea 9, following our "have your cake and eat it too" paradigm, we just use the generic name "algorithm" rather than worrying whether we model them as Turing machines, NAND-TM programs, NAND-RAM, etc.; this makes no difference since all these models are equivalent to one another.)

This remark is equivalent to the one in the video where we note that we'll use python since it's equivalent to NAND-TM

Since this is our first proof by reduction from the Halting problem, we will spell it out in more details than usual. Such a proof by reduction consists of two steps:

1. *Description of the reduction:* We will describe the operation of our algorithm B, and how it makes "function calls" to the hypothetical algorithm A.

2. *Analysis of the reduction:* We will then prove that under the hypothesis that Algorithm A computes *HALTONZERO*, Algorithm B will compute *HALT*.

Algorithm 8.11 — $HALT$ to $HALTONZERO$ reduction.

Input: Turing machine M and string x.
Output: Turing machine M' such that M halts on x iff M' halts on zero

```
1: procedure N_{M,x}(w)        # Description of the T.M. N_{M,x}
2:     return EVAL(M,x)                          # Ignore the
Input: w, evaluate M on x.
3: end procedure
4: return N_{M,x}              # We do not execute N_{M,x}: only
5: return its description
```

Our Algorithm B works as follows: on input M, x, it runs Algorithm 8.11 to obtain a Turing Machine M', and then returns $A(M')$. The machine M' ignores its input z and simply runs M on x.

In pseudocode, the program $N_{M,x}$ will look something like the following:

```
def N(z):
    M = r'.......'
    # a string constant containing desc. of M
    x = r'.......'
    # a string constant containing x
    return eval(M,x)
    # note that we ignore the input z
```

That is, if we think of $N_{M,x}$ as a program, then it is a program that contains M and x as "hardwired constants", and given any input z, it simply ignores the input and always returns the result of evaluating M on x. The algorithm B does *not* actually execute the machine $N_{M,x}$. B merely writes down the description of $N_{M,x}$ as a string (just as we did above) and feeds this string as input to A.

The above completes the *description* of the reduction. The *analysis* is obtained by proving the following claim:

Claim: For every strings M, x, z, the machine $N_{M,x}$ constructed by Algorithm B in Step 1 satisfies that $N_{M,x}$ halts on z if and only if the program described by M halts on the input x.

Proof of Claim: Since $N_{M,x}$ ignores its input and evaluates M on x using the universal Turing machine, it will halt on z if and only if M halts on x.

In particular if we instantiate this claim with the input $z = 0$ to $N_{M,x}$, we see that $HALTONZERO(N_{M,x}) = HALT(M,x)$. Thus if the hypothetical algorithm A satisfies $A(M) = HALTONZERO(M)$ for every M then the algorithm B we construct satisfies $B(M,x) = HALT(M,x)$ for every M, x, contradicting the uncomputability of $HALT$.

■

R

Remark 8.12 — The hardwiring technique. In the proof of Theorem 8.10 we used the technique of "hardwiring" an input x to a program/machine P. That is, modifying a program P that it uses "hardwired constants" for some of all of its input. This technique is quite common in reductions and elsewhere, and we will often use it again in this course.

8.5 RICE'S THEOREM AND THE IMPOSSIBILITY OF GENERAL SOFTWARE VERIFICATION

Start here

The uncomputability of the Halting problem turns out to be a special case of a much more general phenomenon. Namely, that *we cannot certify semantic properties of general purpose programs*. "Semantic properties" mean properties of the *function* that the program computes, as opposed to properties that depend on the particular syntax used by the program.

An example for a *semantic property* of a program P is the property that whenever P is given an input string with an even number of 1's, it outputs 0. Another example is the property that P will always halt whenever the input ends with a 1. In contrast, the property that a C program contains a comment before every function declaration is not a semantic property, since it depends on the actual source code as opposed to the input/output relation.

Checking semantic properties of programs is of great interest, as it corresponds to checking whether a program conforms to a specification. Alas it turns out that such properties are in general *uncomputable*. We have already seen some examples of uncomputable semantic functions, namely *HALT* and *HALTONZERO*, but these are just the "tip of the iceberg". We start by observing one more such example:

Theorem 8.13 — Computing all zero function. Let $ZEROFUNC : \{0,1\}^* \to \{0,1\}$ be the function such that for every $M \in \{0,1\}^*$, $ZEROFUNC(M) = 1$ if and only if M represents a Turing machine such that M outputs 0 on every input $x \in \{0,1\}^*$. Then $ZEROFUNC$ is uncomputable.

P

Despite the similarity in their names, *ZEROFUNC* and *HALTONZERO* are two different functions. For example, if M is a Turing machine that on input $x \in \{0,1\}^*$, halts and outputs the OR of all of x's coordinates, then $HALTONZERO(M) = 1$ (since M does halt on the input 0) but $ZEROFUNC(M) = 0$ (since M does not compute the constant zero function).

Proof of Theorem 8.13. The proof is by reduction to *HALTONZERO*. Suppose, towards the sake of contradiction, that there was an algorithm A such that $A(M) = ZEROFUNC(M)$ for every $M \in \{0,1\}^*$. Then we will construct an algorithm B that solves *HALTONZERO*, contradicting Theorem 8.10.

Given a Turing machine N (which is the input to *HALTONZERO*), our Algorithm B does the following:

1. Construct a Turing Machine M which on input $x \in \{0,1\}^*$, first runs $N(0)$ and then outputs 0.
2. Return $A(M)$.

Now if N halts on the input 0 then the Turing machine M computes the constant zero function, and hence under our assumption that A computes *ZEROFUNC*, $A(M) = 1$. If N does not halt on the input 0, then the Turing machine M will not halt on any input, and so in particular will *not* compute the constant zero function. Hence under our assumption that A computes *ZEROFUNC*, $A(M) = 0$. We see that in both cases, $ZEROFUNC(M) = HALTONZERO(N)$ and hence the value that Algorithm B returns in step 2 is equal to $HALTONZERO(N)$ which is what we needed to prove. ■

Another result along similar lines is the following:

Theorem 8.14 — Uncomputability of verifying parity. The following function is uncomputable

$$COMPUTES\text{-}PARITY(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases} \tag{8.4}$$

We leave the proof of Theorem 8.14 as an exercise (Exercise 8.6). I strongly encourage you to stop here and try to solve this exercise.

8.5.1 Rice's Theorem

Theorem 8.14 can be generalized far beyond the parity function. In fact, this generalization rules out verifying any type of semantic specification on programs. We define a *semantic specification* on programs to be some property that does not depend on the code of the program but just on the function that the program computes.

For example, consider the following two C programs

```
int First(int k) {
    return 2*k;
}

int Second(int n) {
    int i = 0;
    int j = 0
    while (j<n) {
        i = i + 2;
        j=  j + 1;
    }
    return i;
}
```

`First` and `Second` are two distinct C programs, but they compute the same function. A *semantic* property, would be either *true* for both programs or *false* for both programs, since it depends on the *function* the programs compute and not on their code. An example for a semantic property that both `First` and `Second` satisfy is the following: *"The program P computes a function f mapping integers to integers satisfying that $f(n) \geq n$ for every input n"*.

A property is *not semantic* if it depends on the *source code* rather than the input/output behavior. For example, properties such as "the

program contains the variable k" or "the program uses the `while` operation" are not semantic. Such properties can be true for one of the programs and false for others. Formally, we define semantic properties as follows:

Definition 8.15 — Semantic properties. A pair of Turing machines M and M' are *functionally equivalent* if for every $x \in \{0,1\}^*$, $M(x) = M'(x)$. (In particular, $M(x) = \perp$ iff $M'(x) = \perp$ for all x.)

A function $F : \{0,1\}^* \to \{0,1\}$ is *semantic* if for every pair of strings M, M' that represent functionally equivalent Turing machines, $F(M) = F(M')$. (Recall that we assume that every string represents *some* Turing machine, see Remark 8.3)

Recall this symbol means does not halt

There are two trivial examples of semantic functions: the constant one function and the constant zero function. For example, if Z is the constant zero function (i.e., $Z(M) = 0$ for every M) then clearly $F(M) = F(M')$ for every pair of Turing machines M and M' that are functionally equivalent M and M'. Here is a non-trivial example

Solved Exercise 8.1 — $ZEROFUNC$ is semantic. Prove that the function *ZEROFUNC* is semantic.

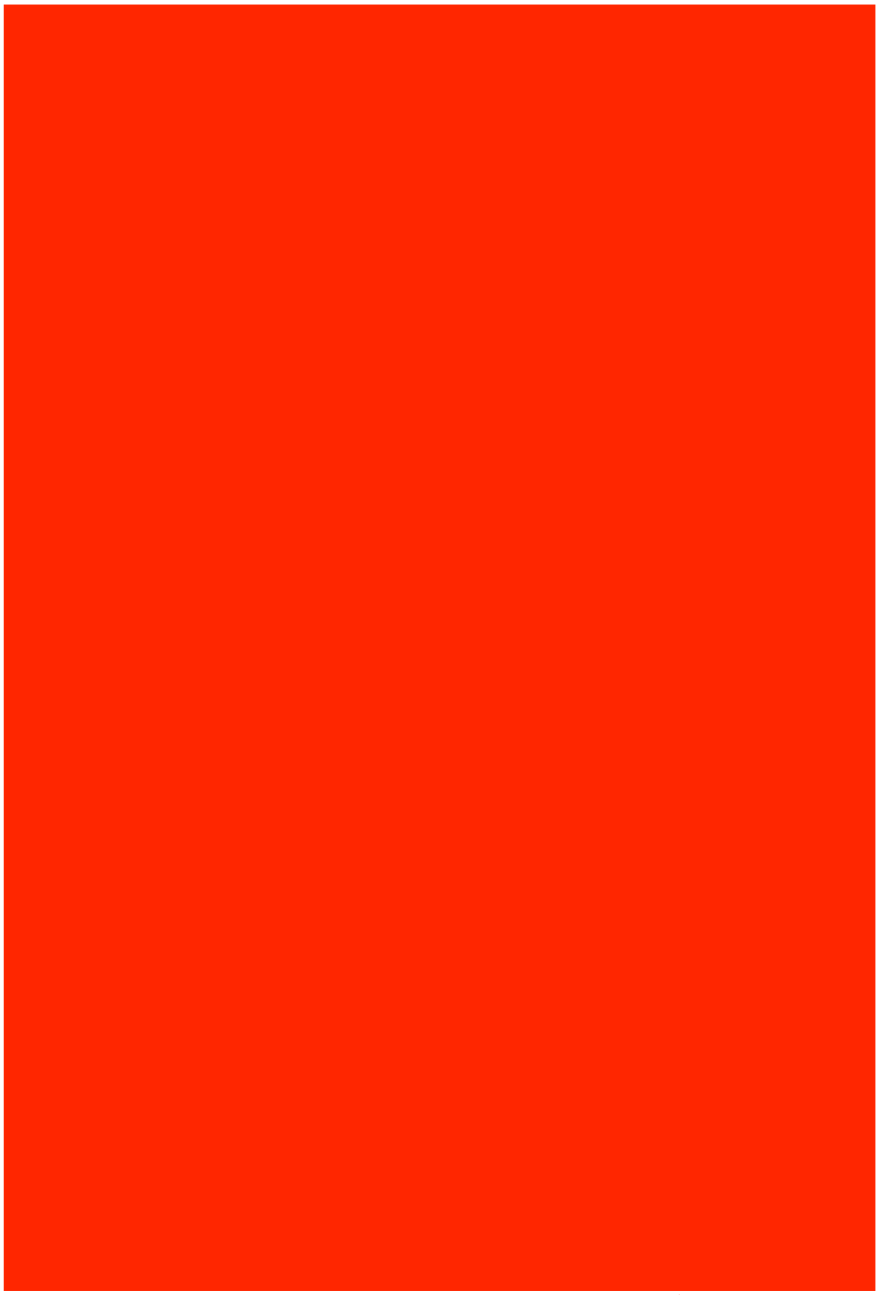
Solution:

Recall that $ZEROFUNC(M) = 1$ if and only if $M(x) = 0$ for every $x \in \{0,1\}^*$. If M and M' are functionally equivalent, then for every x, $M(x) = M'(x)$. Hence $ZEROFUNC(M) = 1$ if and only if $ZEROFUNC(M') = 1$.

Often the properties of programs that we are most interested in computing are the *semantic* ones, since we want to understand the programs' functionality. Unfortunately, Rice's Theorem tells us that these properties are all uncomputable:

Theorem 8.16 — Rice's Theorem. Let $F : \{0,1\}^* \to \{0,1\}$. If F is semantic and non-trivial then it is uncomputable.

Please don't remove this. We'll prove this on the problem set



Interesting discussion that provides motivation/ implications for this week's results. Feel free to skim

8.5.3 Is software verification doomed? (discussion)

Programs are increasingly being used for mission critical purposes, whether it's running our banking system, flying planes, or monitoring nuclear reactors. If we can't even give a certification algorithm that a program correctly computes the parity function, how can we ever be assured that a program does what it is supposed to do? The key insight is that while it is impossible to certify that a *general* program conforms with a specification, it is possible to write a program in the first place in a way that will make it easier to certify. As a trivial example, if you write a program without loops, then you can certify that it halts. Also, while it might not be possible to certify that an

arbitrary program computes the parity function, it is quite possible to write a particular program P for which we can mathematically *prove* that P computes the parity. In fact, writing programs or algorithms and providing proofs for their correctness is what we do all the time in algorithms research.

The field of *software verification* is concerned with verifying that given programs satisfy certain conditions. These conditions can be that the program computes a certain function, that it never writes into a dangerous memory location, that is respects certain invariants, and others. While the general tasks of verifying this may be uncomputable, researchers have managed to do so for many interesting cases, especially if the program is written in the first place in a formalism or programming language that makes verification easier. That said, verification, especially of large and complex programs, remains a highly challenging task in practice as well, and the number of programs that have been formally proven correct is still quite small. Moreover, even phrasing the right theorem to prove (i.e., the specification) if often a highly non-trivial endeavor.

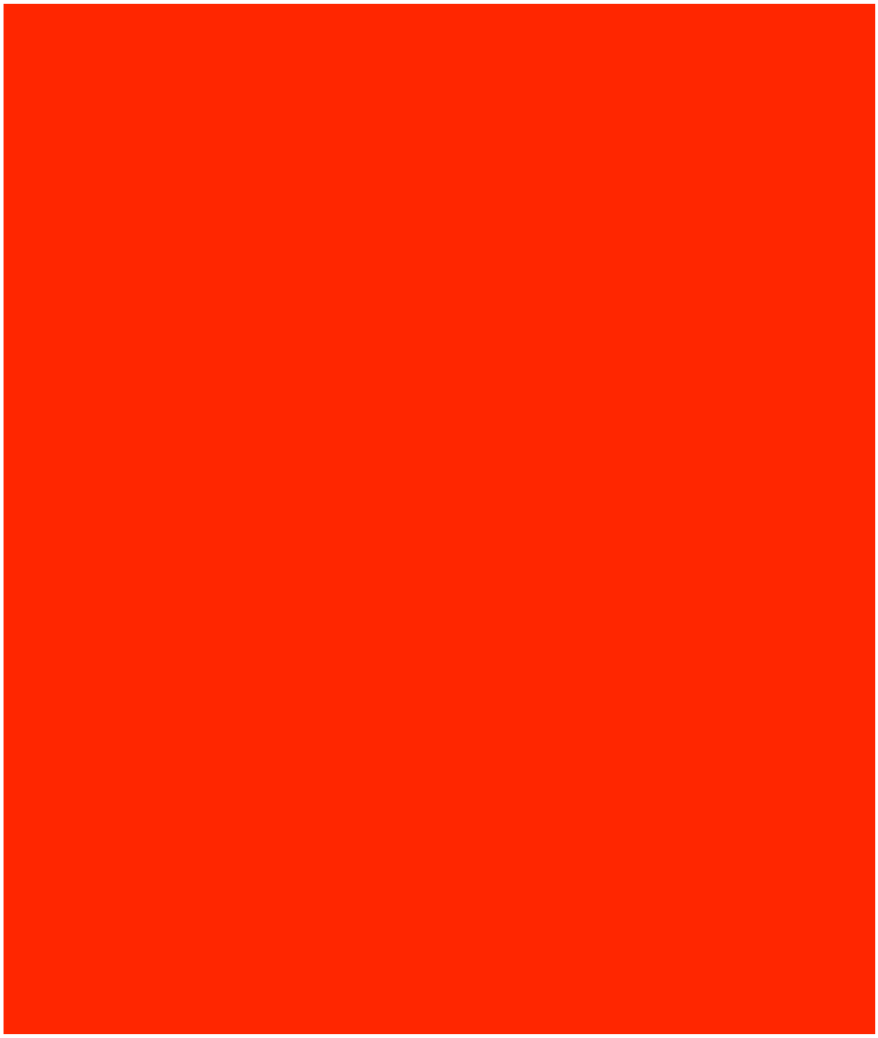


Figure 8.8: The set $\mathbf{R}$ of computable Boolean functions (Definition 6.4) is a proper subset of the set of all functions mapping $\{0,1\}^*$ to $\{0,1\}$. In this chapter we saw a few examples of elements in the latter set that are not in the former.

Learning Objectives:

- Introduce the notion of *polynomial-time reductions* as a way to relate the complexity of problems to one another.
- See several examples of such reductions.
- 3SAT as a basic starting point for reductions.

13
Polynomial-time reductions

This first part (up to section 13.1) is motivation and intuition. It's okay to skip over the details as long as you get the big picture

Consider some of the problems we have encountered in Chapter 11:

1. The *3SAT* problem: deciding whether a given 3CNF formula has a satisfying assignment.
2. Finding the *longest path* in a graph.
3. Finding the *maximum cut* in a graph.
4. Solving *quadratic equations* over n variables $x_0, \ldots, x_{n-1} \in \mathbb{R}$.

All of these problems have the following properties:

- These are important problems, and people have spent significant effort on trying to find better algorithms for them.
- Each one of these is a *search* problem, whereby we search for a solution that is "good" in some easy to define sense (e.g., a long path, a satisfying assignment, etc.).
- Each of these problems has a trivial exponential time algorithm that involve enumerating all possible solutions.
- At the moment, for all these problems the best known algorithm is not much faster than the trivial one in the worst case.

In this chapter and in Chapter 14 we will see that, despite their apparent differences, we can relate the computational complexity of these and many other problems. In fact, it turns out that the problems above are *computationally equivalent*, in the sense that solving one of them immediately implies solving the others. This phenomenon, known as **NP** *completeness*, is one of the surprising discoveries of theoretical computer science, and we will see that it has far-reaching ramifications.

In this chapter we will see that for each one of the problems of finding a longest path in a graph, solving quadratic equations, and finding

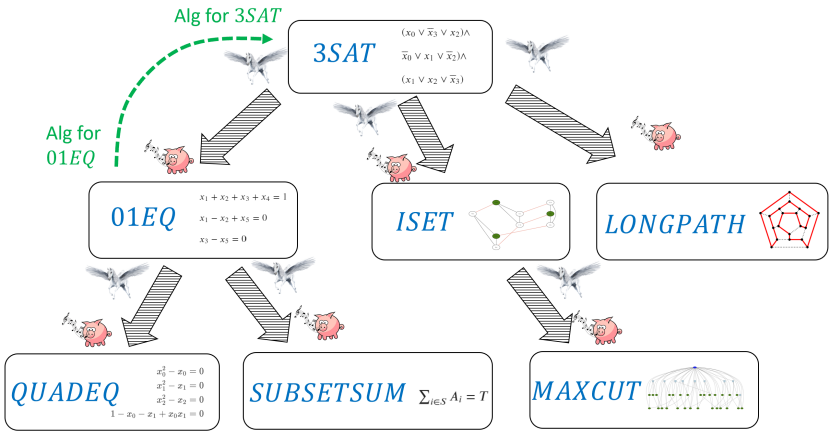


Figure 13.1: In this chapter we show that if the 3*SAT* problem cannot be solved in polynomial time, then neither can the *QUADEQ*, *LONGESTPATH*, *ISET* and *MAXCUT* problems. We do this by using the *reduction paradigm* showing for example "if pigs could whistle" (i.e., if we had an efficient algorithm for *QUADEQ*) then "horses could fly" (i.e., we would have an efficient algorithm for 3*SAT*.)

the maximum cut, if there exists a polynomial-time algorithm for this problem then there exists a polynomial-time algorithm for the 3SAT problem as well. In other words, we will *reduce* the task of solving 3SAT to each one of the above tasks. Another way to interpret these results is that if there *does not exist* a polynomial-time algorithm for 3SAT then there does not exist a polynomial-time algorithm for these other problems as well. In Chapter 14 we will see evidence (though no proof!) that all of the above problems do not have polynomial-time algorithms and hence are *inherently intractable*.

13.1 FORMAL DEFINITIONS OF PROBLEMS

For reasons of technical convenience rather than anything substantial, we concern ourselves with *decision problems* (i.e., Yes/No questions) or in other words *Boolean* (i.e., one-bit output) functions. We model the problems above as functions mapping $\{0,1\}^*$ to $\{0,1\}$ in the following way:

3SAT. The *3SAT problem* can be phrased as the function $3SAT : \{0,1\}^* \rightarrow \{0,1\}$ that takes as input a 3CNF formula φ (i.e., a formula of the form $C_0 \wedge \cdots \wedge C_{m-1}$ where each C_i is the OR of three variables or their negation) and maps φ to 1 if there exists some assignment to the variables of φ that causes it to evalute to *true*, and to 0 otherwise. For example

$$3SAT("(x_0 \vee \overline{x}_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x}_0 \vee \overline{x}_2 \vee x_3)") = 1 \quad (13.1)$$

since the assignment $x = 1101$ satisfies the input formula. In the above we assume some representation of formulas as strings, and

define the function to output 0 if its input is not a valid representation; we use the same convention for all the other functions below.

Quadratic equations. The *quadratic equations problem* corresponds to the function $QUADEQ : \{0,1\}^* \to \{0,1\}$ that maps a set of quadratic equations E to 1 if there is an assignment x that satisfies all equations, and to 0 otherwise.

Longest path. The *longest path problem* corresponds to the function $LONGPATH : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a simple path in G of length at least k, and maps (G,k) to 0 otherwise. The longest path problem is a generalization of the well-known Hamiltonian Path Problem of determining whether a path of length n exists in a given n vertex graph.

Maximum cut. The *maximum cut problem* corresponds to the function $MAXCUT : \{0,1\}^* \to \{0,1\}$ that maps a graph G and a number k to 1 if there is a cut in G that cuts at least k edges, and maps (G,k) to 0 otherwise.

All of the problems above are in **EXP** but it is not known whether or not they are in **P**. However, we will see in this chapter that if either *QUADEQ* , *LONGPATH* or *MAXCUT* are in **P**, then so is *3SAT*.

13.2 POLYNOMIAL-TIME REDUCTIONS

Suppose that $F, G : \{0,1\}^* \to \{0,1\}$ are two functions. A *polynomial-time reduction* (or sometimes just *"reduction"* for short) from F to G is a way to show that F is "no harder" than G, in the sense that a polynomial-time algorithm for G implies a polynomial-time algorithm for F.

Definition 13.1 — Polynomial-time reductions. Let $F, G : \{0,1\}^* \to \{0,1\}^*$. We say that F *reduces to* G, denoted by $F \leq_p G$ if there is a polynomial-time computable $R : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$,

$$F(x) = G(R(x)) \ . \tag{13.2}$$

We say that F and G have *equivalent complexity* if $F \leq_p G$ and $G \leq_p F$.

The following exercise justifies our intuition that $F \leq_p G$ signifies that "F is no harder than G.

Solved Exercise 13.1 — Reductions and P. Prove that if $F \leq_p G$ and $G \in \mathbf{P}$ then $F \in \mathbf{P}$.

■

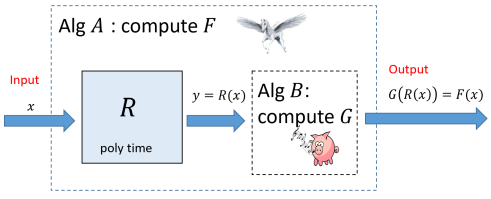


Figure 13.2: If $F \leq_p G$ then we can transform a polynomial-time algorithm B that computes G into a polynomial-time algorithm A that computes F. To compute $F(x)$ we can run the reduction R guaranteed by the fact that $F \leq_p G$ to obtain $y = F(x)$ and then run our algorithm B for G to compute $G(y)$.

P As usual, solving this exercise on your own is an excellent way to make sure you understand Definition 13.1.

It's okay if the input parts of this argument don't make sense. Focus on taking away the proof idea and don't stress too much about understanding all the details

Solution:

Suppose there was an algorithm B that compute F in time $p(n)$ where p is its input size. Then, (13.2) directly gives an algorithm A to compute F (see Fig. 13.2). Indeed, on input $x \in \{0,1\}^*$, Algorithm A will run the polynomial-time reduction R to obtain $y = R(x)$ and then return $B(y)$. By (13.2), $G(R(x)) = F(x)$ and hence Algorithm A will indeed compute F.

We now show that A runs in polynomial time. By assumption, R can be computed in time $q(n)$ for some polynomial q. In particular, this means that $|y| \leq q(|x|)$ (as just writing down y takes $|y|$ steps). This, computing $B(y)$ will take at most $p(|y|) \leq p(q(|x|))$ steps. Thus the total running time of A on inputs of length n is at most the time to compute y, which is bounded by $q(n)$, and the time to compute $B(y)$, which is bounded by $p(q(n))$, and since the composition of two polynomials is a polynomial, A runs in polynomial time.

■

Big Idea 20 A *reduction* $F \leq_p G$ shows that F is "no harder than G" or equivalently that G is "no easier than F".

A reduction from F to G can be used for two purposes:

- If we already know an algorithm for G and $F \leq_p G$ then we can use the reduction to obtain an algorithm for F. This is a widely used tool in algorithm design. For example in Section 11.1.4 we saw how the *Min-Cut Max-Flow* theorem allows to reduce the task of computing a minimum cut in a graph to the task of computing a maximum flow in it.
- If we have proven (or have evidence) that there exists *no polynomial-time algorithm* for F and $F \leq_p G$ then the existence of this reduction allows us to concludes that there exists no polynomial-time algorithm for G. This is the "if pigs could whistle then horses could fly" interpretation we've seen in Section 8.4. We show that if there was an hypothetical efficient algorithm for G (a "whistling pig") then since $F \leq_p G$ then there would be an efficient algorithm for F (a "flying horse"). In this book we often use reductions for this second purpose, although the lines between the two is sometimes blurry (see the bibliographical notes in Section 13.8).

The most crucial difference between the notion in Definition 13.1 and the reductions we saw in the context of *uncomputability* (e.g., in Section 8.4) is that for relating time complexity of problems, we need the reduction to be computable in *polynomial time*, as opposed to merely computable. Definition 13.1 also restricts reductions to have a very specific format. That is, to show that $F \leq_p G$, rather than allowing a general algorithm for F that uses a "magic box" that computes G, we only allow an algorithm that computes $F(x)$ by outputting $G(R(x))$. This restricted form is convenient for us, but people have defined and used more general reductions as well (see Section 13.8).

In this chapter we use reductions to relate the computational complexity of the problems mentioned above: 3SAT, Quadratic Equations, Maximum Cut, and Longest Path, as well as a few others. We will reduce 3SAT to the latter problems, demonstrating that solving any one of them efficiently will result in an efficient algorithm for 3SAT. In Chapter 14 we show the other direction: reducing each one of these problems to 3SAT in one fell swoop.

Transitivity of reductions. Since we think of $F \leq_p G$ as saying that (as far as polynomial-time computation is concerned) F is "easier or equal in difficulty to" G, we would expect that if $F \leq_p G$ and $G \leq_p H$, then it would hold that $F \leq_p H$. Indeed this is the case:

Solved Exercise 13.2 — Transitivity of polynomial-time reductions. For every $F, G, H : \{0,1\}^* \to \{0,1\}$, if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$.

Solution:

If $F \leq_p G$ and $G \leq_p H$ then there exist polynomial-time computable functions R_1 and R_2 mapping $\{0,1\}^*$ to $\{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $F(x) = G(R_1(x))$ and for every $y \in \{0,1\}^*$, $G(y) = H(R_2(y))$. Combining these two equalities, we see that for every $x \in \{0,1\}^*$, $F(x) = H(R_2(R_1(x)))$ and so to show that $F \leq_p H$, it is sufficient to show that the map $x \mapsto R_2(R_1(x))$ is computable in polynomial time. But if there are some constants c, d such that $R_1(x)$ is computable in time $|x|^c$ and $R_2(y)$ is computable in time $|y|^d$ then $R_2(R_1(x))$ is computable in time $(|x|^c)^d = |x|^{cd}$ which is polynomial.

13.3 REDUCING 3SAT TO ZERO ONE EQUATIONS

We will now show our first example of a reduction. The *Zero-One Linear Equations problem* corresponds to the function $01EQ : \{0,1\}^* \to \{0,1\}$ whose input is a collection E of linear equations in variables $x_0, \ldots, x_{n-1}$, and the output is 1 iff there is an assignment $x \in \{0,1\}^n$

of $0/1$ values to the variables that satisfies all the equations. For example, if the input E is a string encoding the set of equations

$$\begin{aligned} x_0 + x_1 + x_2 &= 2 \\ x_0 + x_2 &= 1 \\ x_1 + x_2 &= 2 \end{aligned} \tag{13.3}$$

then $01EQ(E) = 1$ since the assignment $x = 011$ satisfies all three equations. We specifically restrict attention to linear equations in variables $x_0, \ldots, x_{n-1}$ in which every equation has the form $\sum_{i \in S} x_i = b$ where $S \subseteq [n]$ and $b \in \mathbb{N}$.[1]

[1] If you are familiar with matrix notation you may note that such equations can be written as $Ax = \mathbf{b}$ where A is an $m \times n$ matrix with entries in $0/1$ and $\mathbf{b} \in \mathbb{N}^m$.

If we asked the question of whether there is a solution $x \in \mathbb{R}^n$ of *real numbers* to E, then this can be solved using the famous *Gaussian elimination* algorithm in polynomial time. However, there is no known efficient algorithm to solve $01EQ$. Indeed, such an algorithm would imply an algorithm for $3SAT$ as shown by the following theorem:

Theorem 13.2 — Hardness of $01EQ$. $3SAT \leq_p 01EQ$

Proof Idea:

A constraint $x_2 \vee \overline{x}_5 \vee x_7$ can be written as $x_2 + (1 - x_5) + x_7 \geq 1$. This is a linear *inequality* but since the sum on the left-hand side is at most three, we can also turn it into an *equality* by adding two new variables y, z and writing it as $x_2 + (1 - x_5) + x_7 + y + z = 3$. (We will use fresh such variables y, z for every constraint.) Finally, for every variable x_i we can add a variable x'_i corresponding to its negation by adding the equation $x_i + x'_i = 1$, hence mapping the original constraint $x_2 \vee \overline{x}_5 \vee x_7$ to $x_2 + x'_5 + x_7 + y + z = 3$. The main **takeaway technique** from this reduction is the idea of adding *auxiliary variables* to replace an equation such as $x_1 + x_2 + x_3 \geq 1$ that is not quite in the form we want with the equivalent (for $0/1$ valued variables) equation $x_1 + x_2 + x_3 + u + v = 3$ which is in the form we want.

★

```
def SAT2ZOE(ϕ):
    # Reduce 3SAT to 0/1 equations
    n = numvars(ϕ)
    E = ""
    for i in range(n): # add vars for negations
        E+= f"x{subscript(i)} + x{subscript(n+i)} = 1\n"
    m = 2*n
    for literals in getclauses(ϕ):
        # map each clause to equation
        def var(lit): # map literal to variable
            return f"x{subscript(n+int(lit[2:]))}" if lit[0]== "¬" else f"x{subscript(lit[1:])}"
        E+= " + ".join([var(lit) for lit in literals])
        E+= f" + x{subscript(m)} + x{subscript(m+1)} = 3\n"
        m += 2
    return Equation(E)
```

```
SAT2ZOE("(x0 ∨ ¬x3 ∨ x2 ) ∧ (x0 ∨ x1 ∨ ¬x2 ) ∧ (x1 ∨ x2 ∨ ¬x3 )")
```

$$\begin{aligned} x_0 + x_4 &= 1 \\ x_1 + x_5 &= 1 \\ x_2 + x_6 &= 1 \\ x_3 + x_7 &= 1 \\ x_0 + x_7 + x_2 + x_8 + x_9 &= 3 \\ x_0 + x_1 + x_6 + x_{10} + x_{11} &= 3 \\ x_1 + x_2 + x_7 + x_{12} + x_{13} &= 3 \end{aligned}$$

Figure 13.3: Left: Python code implementing the reduction of $3SAT$ to $01EQ$. Right: Example output of the reduction. Code is in our repository.

Proof of Theorem 13.2. To prove the theorem we need to:

1. Describe an algorithm R for mapping an input φ for $3SAT$ into an input E for $01EQ$.

2. Prove that the algorithm runs in polynomial time.

3. Prove that $01EQ(R(\varphi)) = 3SAT(\varphi)$ for every 3CNF formula φ.

We now proceed to do just that. Since this is our first reduction, we will spell out this proof in detail. However it straightforwardly follows the proof idea.

Algorithm 13.3 — $3SAT$ to $01EQ$ reduction.

Input: 3CNF formula φ with n variables $x_0, \ldots, x_{n-1}$ and m clauses.

Output: Set E of linear equations over $0/1$ such that $3SAT(\varphi) = 1$ -iff $01EQ(E) = 1$.

Let E's variables be $x_0, \ldots, x_{n-1}, x'_0, \ldots, x'_{n-1}, y_0, \ldots, y_{m-1}, z_0, \ldots, z_{m-1}$.
for $i \in [n]$ **do**
    add to E the equation $x_i + x'_i = 1$
end for
for j∈[m] **do**
    Let j-th clause be $w_1 \vee w_2 \vee w_3$ where w_1, w_2, w_3 are literals.
    for $a \in [3]$ **do**
        if w_a is variable x_i **then**
            set $t_a \leftarrow x_i$
        end if
        if w_a is negation $\neg x_i$ **then**
            set $t_a \leftarrow x'_i$
        end if
    end for
    Add to E the equation $t_1 + t_2 + t_3 + y_j + z_j = 3$.
end for
return E

The reduction is described in Algorithm 13.3, see also Fig. 13.3. If the input formula has n variable and m steps, Algorithm 13.3 creates a set E of $n+m$ equations over $2n+2m$ variables. Algorithm 13.3 makes an initial loop of n steps (each taking constant time) and then another loop of m steps (each taking constant time) to create the equations, and hence it runs in polynomial time.

Let R be the function computed by Algorithm 13.3. The heart of the proof is to show that for every 3CNF φ, $01EQ(R(\varphi)) = 3SAT(\varphi)$. We split the proof into two parts. The first part, traditionally known as the **completeness** property, is to show that if $3SAT(\varphi) = 1$ then $O1EQ(R(\varphi)) = 1$. The second part, traditionally known as the **soundness** property, is to show that if $01EQ(R(\varphi)) = 1$ then $3SAT(\varphi) = 1$.

(The names "completeness" and "soundness" derive viewing a solution to $R(\varphi)$ as a "proof" that φ is satisfiable, in which case these conditions corresponds to completeness and soundness as defined in Section 10.1.1. However, if you find the names confusing you can simply think of completeness as the "1-instance maps to 1-instance" property and soundness as the "0-instance maps to 0-instance" property.)

We complete the proof by showing both parts:

- **Completeness:** Suppose that $3SAT(\varphi) = 1$, which means that there is an assignment $x \in \{0,1\}^n$ that satisfies φ. We know that for every clause C_j in φ of the form $w_1 \vee w_2 \vee w_3$ (with w_1, w_2, w_3 being literals), $w_1 + w_2 + w_3 \geq 1$, which means that we can assign values to y_j, z_j in $\{0,1\}$ such that $w_1 + w_2 + w_3 + y_j + z_j = 3$. This means that if we let $x'_i = 1 - x_i$ for every $i \in [n]$, then the assignment $x_0, \ldots, x_{n-1}$, $x'_0, \ldots, x'_{n-1}$, $y_0, \ldots, y_{m-1}$, $z_0, \ldots, z_{m-1}$ satisfies the equations $E = R(\varphi)$ and hence $01EQ(R(\varphi)) = 1$.

- **Soundness:** Suppose that the set of equations $E = R(\varphi)$ has a satisfying assignment $x_0, \ldots, x_{n-1}$, $x'_0, \ldots, x'_{n-1}$, $y_0, \ldots, y_{m-1}$, $z_0, \ldots, z_{m-1}$. Then it must be the case that x'_i is the negation of x_i for all $i \in [n]$ and since $y_j + z_j \leq 2$ for every $j \in [m]$, it must be the case that for every clause C_j in φ of the form $w_1 \vee w_2 \vee w_3$ (with w_1, w_2, w_3 being literals), $w_1 + w_2 + w_3 \geq 1$, which means that the assignment $x_0, \ldots, x_{n-1}$ satisfies φ and hence $3SAT(\varphi) = 1$.

■

Anatomy of a reduction. A reduction is simply an algorithm, and like any algorithm, when we come up with a reduction, it is not enough to describe *what* the reduction does, but we also have to provide an *analysis* of *why* it actually works. Specifically, to describe a reduction R demonstrating that $F \leq_p G$ we need to provide the following:

- **Algorithm description:** This is the description of *how* the algorithm maps an input into the output. For example, Algorithm 13.3 above is the description of how we map an instance of $3SAT$ into an instance of $01EQ$ in the reduction demonstrating $3SAT \leq_p 01EQ$.

- **Algorithm analysis:** It is not enough to describe *how* the algorithm works but we need to also explain *why* it works. In particular we need to provide an *analysis* explaining why the reduction is both *efficient* (i.e., runs in polynomial time) and *correct* (satisfies that $G(R(x)) = F(x)$ for every x)). Specifically, the components of analysis of a reduction R include:

- **Efficiency:** We need to show that R runs in polynomial time. In most reductions we encounter this part is straightforward, as the reductions we typically use involve a constant number of nested loops, each involving a constant number of operations.

- **Completeness:** In a reduction R demonstrating $F \leq_p G$, the *completeness* condition is the condition that for every $x \in \{0,1\}^*$, if $F(x) = 1$ then $G(R(x)) = 1$. Typically we construct the reduction to ensure that this holds, by giving a way to map a "certificate/-solution" certifying that $F(x) = 1$ into a solution certifying that $G(R(x)) = 1$. For example, in the proof of Theorem 13.2 the satisfying assignment for the $3SAT$ formula φ can be mapped to a solution to the set of equations $R(\varphi)$.

- **Soundness:** This is the condition that if $F(x) = 0$ then $G(R(x)) = 0$ or (taking the contrapositive) if $G(R(x)) = 1$ then $F(x) = 1$. This is sometimes straightforward but can also be harder to show than the completeness condition, and in more advanced reductions (such as the reduction $SAT \leq_p ISET$ of Theorem 13.5) demonstrating soundness is the main part of the analysis.

Whenever you need to provide a reduction, you should make sure that your description has all these components. While it is sometimes tempting to weave together the description of the reduction and its analysis, it is usually clearer if you separate the two, and also break down the analysis to its three components of efficiency, completeness, and soundness.

13.3.1 Quadratic equations

Now that we reduced $3SAT$ to $01EQ$, we can use this to reduce $3SAT$ to the *quadratic equations* problem. This is the function $QUADEQ$ in which the input is a list of n-variate polynomials $p_0, \ldots, p_{m-1} : \mathbb{R}^n \to \mathbb{R}$ that are all of degree at most two (i.e., they are *quadratic*) and with integer coefficients. (The latter condition is for convenience and can be achieved by scaling.) We define $QUADEQ(p_0, \ldots, p_{m-1})$ to equal 1 if and only if there is a solution $x \in \mathbb{R}^n$ to the equations $p_0(x) = 0$, $p_1(x) = 0$, ..., $p_{m-1}(x) = 0$.

For example, the following is a set of quadratic equations over the variables x_0, x_1, x_2:

$$\begin{aligned} x_0^2 - x_0 &= 0 \\ x_1^2 - x_1 &= 0 \\ x_2^2 - x_2 &= 0 \\ 1 - x_0 - x_1 + x_0 x_1 &= 0 \end{aligned} \tag{13.4}$$

You can verify that $x \in \mathbb{R}^3$ satisfies this set of equations if and only if $x \in \{0,1\}^3$ and $x_0 \vee x_1 = 1$.

Theorem 13.4 — Hardness of quadratic equations.

$$3SAT \leq_p QUADEQ \tag{13.5}$$

Proof Idea:

Using the transitivity of reductions (Solved Exercise 13.2), it is enough to show that $01EQ \leq_p QUADEQ$, but this follows since we can phrase the equation $x_i \in \{0,1\}$ as the quadratic constraint $x_i^2 - x_i = 0$. The **takeaway technique** of this reduction is that we can use *nonlinearity* to force continuous variables (e.g., variables taking values in $\mathbb{R}$) to be discrete (e.g., take values in $\{0,1\}$).

⋆

Proof of Theorem 13.4. By Theorem 13.2 and Solved Exercise 13.2, it is sufficient to prove that $01EQ \leq_p QUADEQ$. Let E be an instance of $01EQ$ with variables $x_0, \ldots, x_{m-1}$. We define $R(E)$ to be the set of quadratic equations E' that is obtained by taking the linear equations in E and adding to them the n quadratic equations $x_i^2 - x_i = 0$ for all $i \in [n]$. Clearly the map $E \mapsto E'$ can be computed in polynomial time. We claim that $01EQ(E) = 1$ if and only if $QUADEQ(E') = 1$. Indeed, the only difference between the two instances is that:

- In the $01EQ$ instance E, the equations are over variables $x_0, \ldots, x_{n-1}$ in $\{0,1\}$.
- In the $QUADEQ$ instance E', the equations are overvariables $x_0, \ldots, x_{n-1} \in \mathbb{R}$ but we have the extra constraints $x_i^2 - x_i = 0$ for all $i \in [n]$.

Since for every $a \in \mathbb{R}$, $a^2 - a = 0$ if and only if $a \in \{0,1\}$, the two sets of equations are equivalent and $01EQ(E) = QUADEQ(E')$ which is what we wanted to prove.

■

13.4 THE INDEPENDENT SET PROBLEM

This is a repeat of the second video. If you've already watched the videos, skip to the next green arrow.

For a graph $G = (V, E)$, an independent set (also known as a *stable set*) is a subset $S \subseteq V$ such that there are no edges with both endpoints in S (in other words, $E(S, S) = \emptyset$). Every "singleton" (set consisting of a single vertex) is trivially an independent set, but finding larger independent sets can be challenging. The *maximum independent set* problem (henceforth simply "independent set") is the task of finding the largest independent set in the graph. The independent set

problem is naturally related to *scheduling problems*: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts. The independent set problem has been studied in a variety of settings, including for example in the case of algorithms for finding structure in protein-protein interaction graphs.

As mentioned in Section 13.1, we think of the independent set problem as the function $ISET : \{0,1\}^* \to \{0,1\}$ that on input a graph G and a number k outputs 1 if and only if the graph G contains an independent set of size at least k. We now reduce 3SAT to Independent set.

Theorem 13.5 — Hardness of Independent Set. $3SAT \leq_p ISET$.

Proof Idea:

The idea is that finding a satisfying assignment to a 3SAT formula corresponds to satisfying many local constraints without creating any conflicts. One can think of "$x_{17} = 0$" and "$x_{17} = 1$" as two conflicting events, and of the constraints $x_{17} \vee \overline{x}_5 \vee x_9$ as creating a conflict between the events "$x_{17} = 0$", "$x_5 = 1$" and "$x_9 = 0$", saying that these three cannot simultaneosly co-occur. Using these ideas, we can we can think of solving a 3SAT problem as trying to schedule non conflicting events, though the devil is, as usual, in the details. The **takeaway technique** here is to map each clause of the original formula into a *gadget* which is a small subgraph (or more generally "subinstance") satisfying some convenient properties. We will see these "gadgets" used time and again in the construction of polynomial-time reductions.

⋆

Proof of Theorem 13.5. Given a 3SAT formula φ on n variables and with m clauses, we will create a graph G with $3m$ vertices as follows. (See Fig. 13.4 for an example and Fig. 13.5 for Python code.)

- A clause C in φ has the form $C = y \vee y' \vee y''$ where y, y', y'' are *literals* (variables or their negation). For each such clause C, we will add three vertices to G, and label them (C, y), (C, y'), and (C, y'') respectively. We will also add the three edges between all pairs of these vertices, so they form a *triangle*. Since there are m clauses in φ, the graph G will have $3m$ vertices.
- In addition to the above edges, we also add an edge between every pair vertices of the form (C, y) and (C', y') where y and y' are *conflicting* literals. That is, we add an edge between (C, y) and (C, y') if there is an i such that $y = x_i$ and $y' = \overline{x}_i$ or vice versa.

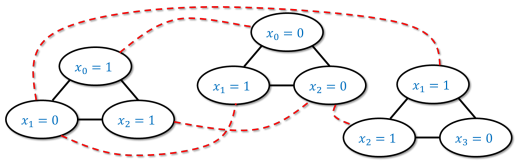


Figure 13.4: An example of the reduction of $3SAT$ to $ISET$ for the case the original input formula is $\varphi = (x_0 \vee \overline{x}_1 \vee x_2) \wedge (\overline{x}_0 \vee x_1 \vee \overline{x}_2) \wedge (x_1 \vee x_2 \vee \overline{x}_3)$. We map each clause of φ to a triangle of three vertices, each tagged above with "$x_i = 0$" or "$x_i = 1$" depending on the value of x_i that would satisfy the particular literal. We put an edge between every two literals that are *conflicting* (i.e., tagged with "$x_i = 0$" and "$x_i = 1$" respectively).

The above construction of G based on φ can clearly be carried out in polynomial time. Hence to prove the theorem we need to show that φ is satisfiable if and only if G contains an independent set of m vertices. We now show both directions of this equivalence:

Part 1: Completeness. The "completeness" direction is to show that if φ has a satisfying assignment x^*, then G has an independent set S^* of m vertices. Let us now show this.

Indeed, suppose that φ has a satisfying assignment $x^* \in \{0,1\}^n$. Then for every clause $C = y \vee y' \vee y''$ of φ, one of the literals y, y', y'' must evaluate to *true* under the assignment x^* (as otherwise it would not satisfy φ). We let S be a set of m vertices that is obtained by choosing for every clause C one vertex of the form (C, y) such that y evaluates to true under x^*. (If there is more than one such vertex for the same C, we arbitrarily choose one of them.)

We claim that S is an independent set. Indeed, suppose otherwise that there was a pair of vertices (C, y) and (C', y') in S that have an edge between them. Since we picked one vertex out of each triangle corresponding to a clause, it must be that $C \neq C'$. Hence the only way that there is an edge between (C, y) and (C, y') is if y and y' are conflicting literals (i.e. $y = x_i$ and $y' = \overline{x}_i$ for some i). But that would that they can't both evaluate to *true* under the assignment x^*, which contradicts the way we constructed the set S. This completes the proof of the completeness condition.

Part 2: Soundness. The "soundness" direction is to show that if G has an independent set S^* of m vertices, then φ has a satisfying assignment $x^* \in \{0,1\}^n$. Let us now show this.

Indeed, suppose that G has an independent set $S*$ with m vertices. We will define an assignment $x^* \in \{0,1\}^n$ for the variables of φ as follows. For every $i \in [n]$, we set x^*_i according to the following rules:

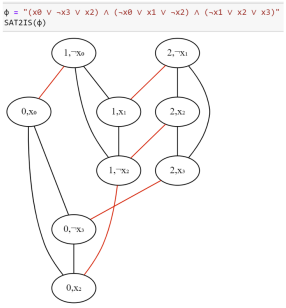
- If S^* contains a vertex of the form (C, x_i) then we set $x^*_i = 1$.
- If S^* contains a vertex of the form $(C, \overline{x_i})$ then we set $x^*_i = 0$.
- If S^* does not contain a vertex of either of these forms, then it does not matter which value we give to x^*_i, but for concreteness we'll set $x^*_i = 0$.

The first observation is that x^* is indeed well defined, in the sense that the rules above do not conflict with one another, and ask to set x^*_i to be both 0 and 1. This follows from the fact that S^* is an *independent set* and hence if it contains a vertex of the form (C, x_i) then it cannot contain a vertex of the form $(C', \overline{x_i})$.

We now claim that x^* is a satisfying assignment for φ. Indeed, since S^* is an independent set, it cannot have more than one vertex inside each one of the m triangles $(C, y), (C, y'), (C, y'')$ corresponding to a

clause of φ. Hence since $|S^*| = m$, it must have exactly one vertex in each such triangle. For every clause C of φ, if (C, y) is the vertex in S^* in the triangle corresponding to C, then by the way we defined x^*, the literal y must evaluate to *true*, which means that x^* satisfies this clause. Therefore x^* satisfies all clauses of φ, which is the definition of a satisfying assignment.

This completes the proof of Theorem 13.5 ■




```python
# Reduction SAT to Independent Set
def SAT2IS(ϕ):
    n = numvars(ϕ)
    G = nGraph() #(engine='neato')

    def nname(c,v): return f"({c},{v})"

    c = 0
    # nodes[literal] = vertices corresponding to this literal
    nodes = {f"x{i}":[] for i in range(n)}
    nodes.update(  {f"-x{i}":[] for i in range(n)})

    c=0
    for C in getclauses(ϕ):
        for u in C:
            G.node(nname(c,u),f"{c},{subscript(u)}",fontsize="12")
            negu = u[1:] if u[0]=="-" else "-"+u
            # add conflicting edges
            for v in nodes[negu]: G.edge(nname(c,u),v,color="red")
            nodes[u].append(nname(c,u))
        # add triangle edges
        G.edges([[nname(c,C[0]), nname(c,C[1])],[nname(c,C[1]), nname(c,C[2])],[nname(c,C[0]), nname(c,C[2])]])
        c += 1

    return G
```

Figure 13.5: The reduction of 3SAT to Independent Set. On the righthand side is *Python* code that implements this reduction. On the lefthand side is a sample output of the reduction. We use black for the "triangle edges" and red for the "conflict edges". Note that the satisfying assignment $x^* = 0110$ corresponds to the independent set $(0, \neg x_3), (1, \neg x_0), (2, x_2)$.

Solved Exercise 13.3 — Clique is equivalent to independent set. The maximum clique problem corresponds to the function $CLIQUE : \{0,1\}^* \to \{0,1\}$ such that for a graph G and a number k, $CLIQUE(G, k) = 1$ iff there is a S subset of k vertices such that for *every* distinct $u, v \in S$, the edge u, v is in G. Such a set is known as a *clique*.

Prove that $CLIQUE \leq_p ISET$ and $ISET \leq_p CLIQUE$. ■

This is recommended but not mandatory

Solution:

If $G = (V, E)$ is a graph, we denote by $\overline{G}$ its *complement* which is the graph on the same vertices V and such that for every distinct $u, v \in V$, the edge $\{u, v\}$ is present in $\overline{G}$ if and only if this edge is *not* present in G.

This means that for every set S, S is an independent set in G if and only if S is a *clique* in $\overline{S}$. Therefore for every k, $ISET(G, k) = CLIQUE(\overline{G}, k)$. Since the map $G \mapsto \overline{G}$ can be computed efficiently, this yields a reduction $ISET \leq_p CLIQUE$. Moreover, since $\overline{\overline{G}} = G$ this yields a reduction in the other direction as well. ■

13.5 REDUCING INDEPENDENT SET TO MAXIMUM CUT

The rest of the chapter is more polynomial time reductions. It's all optional from here on out. As a disclaimer some of the proofs that follow are quite tough.

We now show that the independent set problem reduces to the *maximum cut* (or "max cut") problem, modeled as the function *MAXCUT*

that on input a pair (G, k) outputs 1 iff G contains a cut of at least k edges. Since both are graph problems, a reduction from independent set to max cut maps one graph into the other, but as we will see the output graph does not have to have the same vertices or edges as the input graph.

Theorem 13.6 — Hardness of Max Cut. $ISET \leq_p MAXCUT$

Proof Idea:

We will map a graph G into a graph H such that a large independent set in G becomes a partition cutting many edges in H. We can think of a cut in H as coloring each vertex either "blue" or "red". We will add a special "source" vertex s^*, connect it to all other vertices, and assume without loss of generality that it is colored blue. Hence the more vertices we color red, the more edges from s^* we cut. Now, for every edge u, v in the original graph G we will add a special "gadget" which will be a small subgraph that involves u,v, the source s^*, and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in G then the corresponding cut in H will be "penalized" in the sense that it would not cut as many edges. Once we set for ourselves this objective, it is not hard to find a gadget that achieves it— see the proof below. Once again the **takeaway technique** is to use (this time a slightly more clever) gadget.

⋆

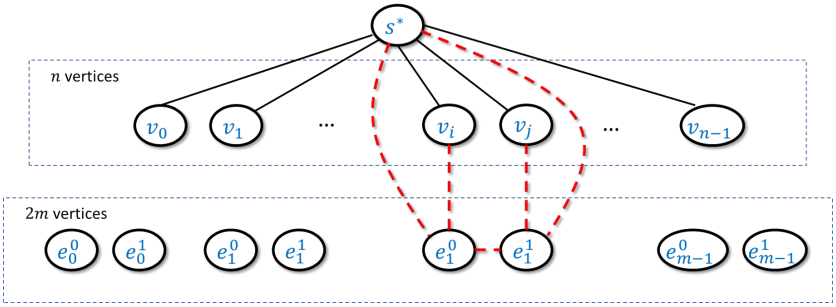


Figure 13.6: In the reduction of *ISET* to *MAXCUT* we map an n-vertex m-edge graph G into the $n + 2m + 1$ vertex and $n + 5m$ edge graph H as follows. The graph H contains a special "source" vertex s^*,n vertices $v_0, \ldots, v_{n-1}$, and $2m$ vertices $e_0^0, e_0^1, \ldots, e_{m-1}^0, e_{m-1}^1$ with each pair corresponding to an edge of G. We put an edge between s^* and v_i for every $i \in [n]$, and if the t-th edge of G was (v_i, v_j) then we add the five edges $(s^*, e_t^0), (s^*, e_t^1), (v_i, e_t^0), (v_j, e_t^1), (e_t^0, e_t^1)$. The intent is that if cut at most one of v_i, v_j from s^* then we'll be able to cut 4 out of these five edges, while if we cut both v_i and v_j from s^* then we'll be able to cut at most three of them.

Proof of Theorem 13.6. We will transform a graph G of n vertices and m edges into a graph H of $n + 1 + 2m$ vertices and $n + 5m$ edges in the following way (see also Fig. 13.6). The graph H contains all vertices of G (though not the edges between them!) and in addition H also has:

* A special vertex s^* that is connected to all the vertices of G

* For every edge $e = \{u, v\} \in E(G)$, two vertices e_0, e_1 such that e_0 is connected to u and e_1 is connected to v, and moreover we add the edges $\{e_0, e_1\}, \{e_0, s^*\}, \{e_1, s^*\}$ to H.

Theorem 13.6 will follow by showing that G contains an independent set of size at least k if and only if H has a cut cutting at least $k + 4m$ edges. We now prove both directions of this equivalence:

Part 1: Completeness. If I is an independent k-sized set in G, then we can define S to be a cut in H of the following form: we let S contain all the vertices of I and for every edge $e = \{u, v\} \in E(G)$, if $u \in I$ and $v \notin I$ then we add e_1 to S; if $u \notin I$ and $v \in I$ then we add e_0 to S; and if $u \notin I$ and $v \notin I$ then we add both e_0 and e_1 to S. (We don't need to worry about the case that both u and v are in I since it is an independent set.) We can verify that in all cases the number of edges from S to its complement in the gadget corresponding to e will be four (see Fig. 13.7). Since s^* is not in S, we also have k edges from s^* to I, for a total of $k + 4m$ edges.

Part 2: Soundness. Suppose that S is a cut in H that cuts at least $C = k + 4m$ edges. We can assume that s^* is not in S (otherwise we can "flip" S to its complement $\overline{S}$, since this does not change the size of the cut). Now let I be the set of vertices in S that correspond to the original vertices of G. If I was an independent set of size k then we would be done. This might not always be the case but we will see that if I is not an independent set then it's also larger than k. Specifically, we define $m_{in} = |E(I, I)|$ be the set of edges in G that are contained in I and let $m_{out} = m - m_{in}$ (i.e., if I is an independent set then $m_{in} = 0$ and $m_{out} = m$). By the properties of our gadget we know that for every edge $\{u, v\}$ of G, we can cut at most three edges when both u and v are in S, and at most four edges otherwise. Hence the number C of edges cut by S satisfies $C \leq |I| + 3m_{in} + 4m_{out} = |I| + 3m_{in} + 4(m - m_{in}) = |I| + 4m - m_{in}$. Since $C = k + 4m$ we get that $|I| - m_{in} \geq k$. Now we can transform I into an independent set I' by going over every one of the m_{in} edges that are inside I and removing one of the endpoints of the edge from it. The resulting set I' is an independent set in the graph G of size $|I| - m_{in} \geq k$ and so this concludes the proof of the soundness condition.

■

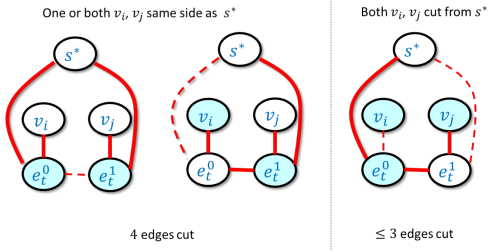
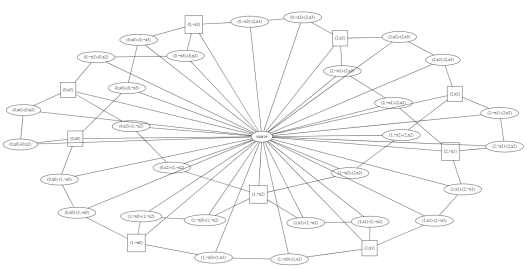


Figure 13.7: In the reduction of independent set to max cut, for every $t \in [m]$, we have a "gadget" corresponding to the t-th edge $e = \{v_i, v_j\}$ in the original graph. If we think of the side of the cut containing the special source vertex s^* as "white" and the other side as "blue", then the leftmost and center figures show that if v_i and v_j are not both blue then we can cut four edges from the gadget. In contrast, by enumerating all possibilities one can verify that if both u and v are blue, then no matter how we color the intermediate vertices e_t^0, e_t^1, we will cut at most three edges from the gadget. The figure above contains only the gadget edges and ignores the edges connecting s^* to the vertices $v_0, \ldots, v_{n-1}$.

13.6 REDUCING 3SAT TO LONGEST PATH

Note: This section is still a little messy; feel free to skip it or just read it without going into the proof details. The proof appears in Section 7.5 in Sipser's book.

One of the most basic algorithms in Computer Science is Dijkstra's algorithm to find the *shortest path* between two vertices. We now show



```python
# Reduction IS to MAXCUT
def IS2MAXCUT(G):
    G =nxgraph(G)
    H =Graph(engine='sfdp')
    s ="source"
    H.node(s) # create source node

    for v in G.nodes():
        H.node(v,shape="square")
        H.edge(s,v)

    j =0
    for (u,v) in G.edges():
        g1 = "e"+str(j)+"a"
        g2 = "e"+str(j)+"b"
        H.node(g1,label=u+"-"+v)
        H.node(g2,label=u+"-"+v)
        # add gadget
        H.edges([(s,g1),(s,g2),(g1,g2),(u,g1),(v,g2)])
        j +=1


    return H
```

Figure 13.8: The reduction of independent set to max cut. On the righthand side is Python code implementing the reduction. On the lefthand side is an example output of the reduction where we apply it to the independent set instance that is obtained by running the reduction of Theorem 13.5 on the 3CNF formula $(x_0 \vee \overline{x}_3 \vee x_2) \wedge (\overline{x}_0 \vee x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$.

that in contrast, an efficient algorithm for the *longest path* problem would imply a polynomial-time algorithm for 3SAT.

Theorem 13.7 — Hardness of longest path.

$$3SAT \leq_p LONGPATH \tag{13.6}$$

Proof Idea:

To prove Theorem 13.7 need to show how to transform a 3CNF formula φ into a graph G and two vertices s, t such that G has a path of length at least k if and only if φ is satisfiable. The idea of the reduction is sketched in Fig. 13.9 and Fig. 13.10. We will construct a graph that contains a potentially long "snaking path" that corresponds to all variables in the formula. We will add a "gadget" corresponding to each clause of φ in a way that we would only be able to use the gadgets if we have a satisfying assignment.

⋆

Proof of Theorem 13.7. We build a graph G that "snakes" from s to t as follows. After s we add a sequence of n long loops. Each loop has an "upper path" and a "lower path". A simple path cannot take both the upper path and the lower path, and so it will need to take exactly one of them to reach s from t.

Our intention is that a path in the graph will correspond to an assignment $x \in \{0,1\}^n$ in the sense that taking the upper path in the i^{th} loop corresponds to assigning $x_i = 1$ and taking the lower path corresponds to assigning $x_i = 0$. When we are done snaking through all the n loops corresponding to the variables to reach t we need to pass through m "obstacles": for each clause j we will have a small gadget consisting of a pair of vertices s_j, t_j that have three paths between them. For example, if the j^{th} clause had the form $x_{17} \vee \overline{x}_{55} \vee x_{72}$ then one path would go through a vertex in the lower loop corresponding to x_{17}, one path would go through a vertex in the upper loop corresponding to x_{55} and the third would go through the lower loop cor-

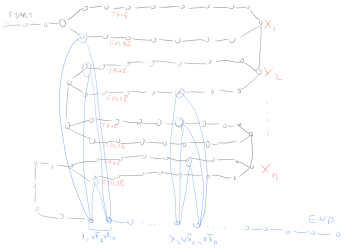


Figure 13.9: We can transform a 3SAT formula φ into a graph G such that the longest path in the graph G would correspond to a satisfying assignment in φ. In this graph, the black colored part corresponds to the variables of φ and the blue colored part corresponds to the vertices. A sufficiently long path would have to first "snake" through the black part, for each variable choosing either the "upper path" (corresponding to assigning it the value `True`) or the "lower path" (corresponding to assigning it the value `False`). Then to achieve maximum length the path would traverse through the blue part, where to go between two vertices corresponding to a clause such as $x_{17} \vee \overline{x}_{32} \vee x_{57}$, the corresponding vertices would have to have been not traversed before.

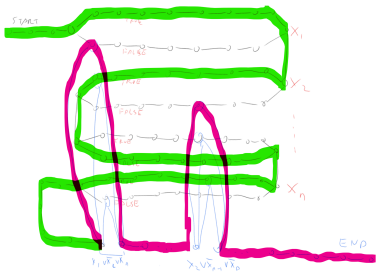


Figure 13.10: The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.

responding to x_{72}. We see that if we went in the first stage according to a satisfying assignment then we will be able to find a free vertex to travel from s_j to t_j. We link t_1 to s_2, t_2 to s_3, etc and link t_m to t. Thus a satisfying assignment would correspond to a path from s to t that goes through one path in each loop corresponding to the variables, and one path in each loop corresponding to the clauses. We can make the loop corresponding to the variables long enough so that we must take the entire path in each loop in order to have a fighting chance of getting a path as long as the one corresponds to a satisfying assignment. But if we do that, then the only way if we are able to reach t is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause j where we cannot reach t_j from s_j without using a vertex we already used before.

■

13.6.1 Summary of relations

We have shown that there are a number of functions F for which we can prove a statement of the form "If $F \in \mathbf{P}$ then $3SAT \in \mathbf{P}$". Hence coming up with a polynomial-time algorithm for even one of these problems will entail a polynomial-time algorithm for $3SAT$ (see for example Fig. 13.11). In Chapter 14 we will show the inverse direction ("If $3SAT \in \mathbf{P}$ then $F \in \mathbf{P}$") for these functions, hence allowing us to conclude that they have *equivalent complexity* to $3SAT$.

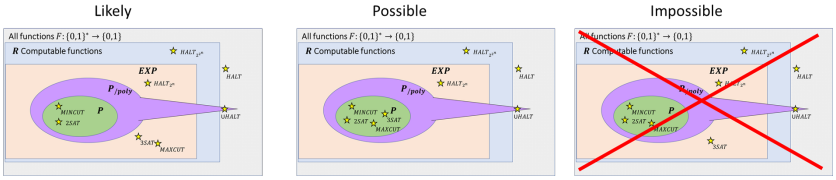


Figure 13.11: So far we have shown that $\mathbf{P} \subseteq \mathbf{EXP}$ and that several problems we care about such as $3SAT$ and $MAXCUT$ are in **EXP** but it is not known whether or not they are in **EXP**. However, since $3SAT \leq_p MAXCUT$ we can rule out the possiblity that $MAXCUT \in \mathbf{P}$ but $3SAT \notin \mathbf{P}$. The relation of $\mathbf{P_{/poly}}$ to the class **EXP** is not known. We know that **EXP** does not contain $\mathbf{P_{/poly}}$ since the latter even contains uncomputable functions, but we do not know whether ot not $\mathbf{EXP} \subseteq \mathbf{P_{/poly}}$ (though it is believed that this is not the case and in particular that both $3SAT$ and $MAXCUT$ are not in $\mathbf{P_{/poly}}$).

Lecture Recap

- The computational complexity of many seemingly unrelated computational problems can be related to one another through the use of *reductions*.
- If $F \leq_p G$ then a polynomial-time algorithm for G can be transformed into a polynomial-time algorithm for F.
- Equivalently, if $F \leq_p G$ and F does *not* have a polynomial-time algorithm then neither does G.
- We've developed many techniques to show that $3SAT \leq_p F$ for interesting functions F. Sometimes we can do so by using *transitivity* of reductions: if $3SAT \leq_p G$ and $G \leq_p F$ then $3SAT \leq_p F$.