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

- The 3SAT problem: deciding whether a given 3CNF formula has a satisfying assignment.
- Finding the longest path in a graph.
- Finding the maximum cut in a graph.
- 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 involves 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 problem 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
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 problem do not have polynomial-time algorithms and hence are inherently intractable.

Decision problems. For reasons of technical conditions rather than anything substantial, we will 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:

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

\[
3SAT ((x_0 \land x_1 \land x_2) \lor (x_1 \lor x_2 \lor x_3) \lor (x_0 \land x_2 \land x_3)^*) = 1 \quad (13.1)
\]

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.
The quadratic equations problem corresponds to the function
QUADEQ : \{0, 1\}^* → \{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.

The longest path problem corresponds to the function LONGPATH : \{0, 1\}^* → \{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.

The maximum cut problem corresponds to the function MAXCUT : \{0, 1\}^* → \{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.

13.1 REDUCTIONS

Suppose that \(F, G : \{0, 1\}^* \rightarrow \{0, 1\}\) are two functions. A reduction 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 \(F\). The formal definition is as follows:

**Definition 13.1 — Reductions.** Let \(F, G : \{0, 1\}^* \rightarrow \{0, 1\}^*\). We say that \(F\) reduces to \(G\), denoted by \(F \leq_p G\) if there is a polynomial-time computable \(R : \{0, 1\}^* \rightarrow \{0, 1\}^*\) 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 \(G \in P\) then \(F \in P\).
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.

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. The “quicksort” algorithm reduces the task of sorting to the task of partitioning an array to the elements smaller and bigger than some “pivot” element. Dijkstra’s shortest path algorithm reduces the task of finding the shortest path between two elements to the task of implementing a priority queue data structure.

- 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 will often use reductions for the second purpose, although the lines between the two is sometimes blurry (see the bibliographical notes).

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:
Lemma 13.2 For every $F, G, H : \{0, 1\}^* \rightarrow \{0, 1\}$, if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$.

We leave the proof of Lemma 13.2 as Exercise 13.1. Pausing now and doing this exercise is an excellent way to verify that you understood the definition of reductions.

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.

13.2 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 $01EQ : \{0, 1\}^* \rightarrow \{0, 1\}$ 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

\[
\begin{align*}
x_0 + x_1 + x_2 &= 2 \\
x_0 + x_2 &= 1 \\
x_1 + x_2 &= 2
\end{align*}
\]

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 $b \in \mathbb{N}$.\(^1\)

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

\(^1\) If you are familiar with matrix notation you may note that such equations can be written as $Ax = b$ where $A$ is an $m \times n$ matrix with entries in 0/1 and $b \in \mathbb{N}^m$.\)
elimination algorithm in polynomial time. However, there is no known efficient algorithm to solve $01\text{EQ}$. Indeed, such an algorithm would imply an algorithm for $3\text{SAT}$ as shown by the following theorem:

**Theorem 13.3 — Hardness of $01\text{EQ}$.** $3\text{SAT} \leq_p 01\text{EQ}$

**Proof Idea:**
A constraint $x_2 \lor x_5 \lor x_7$ 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 $x_2 \lor x_5 \lor x_7$ to $x_2 + x_5' + x_7 + y + z = 3$.

**Proof of Theorem 13.3.** To prove the theorem we need to:

1. Describe an algorithm $R$ for mapping an input $\varphi$ for $3\text{SAT}$ into an input $E$ for $01\text{EQ}$.

2. Prove that the algorithm runs in polynomial time and that $01\text{EQ}(R(\varphi)) = 3\text{SAT}(\varphi)$ for every $3\text{CNF}$ formula $\varphi$.

We proceed to do just that.

### 13.3 Quadratic Equations

Recall that in the *quadratic equation* problem, 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.) The task is to find out whether there is a solution $x \in \mathbb{R}^n$ to the equations $p_0(x) = 0, p_1(x) = 0, \ldots, p_{m-1}(x) = 0$. 

{#3sat2zoeqreductionfig}
For example, the following is a set of quadratic equations over the variables $x_0, x_1, x_2$:

\[
\begin{align*}
  x_0^2 - x_0 &= 0 \\
  x_1^2 - x_1 &= 0 \\
  x_2^2 - x_2 &= 0 \\
  1 - x_0 - x_1 + x_0x_1 &= 0
\end{align*}
\]  

(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 $x_0 \lor x_1 = 1$.

We will show how to reduce 3SAT to the problem of Quadratic Equations.

**Theorem 13.4 — Hardness of quadratic equations.**

\[
3SAT \leq_p QUADEQ
\]  

(13.5)

where 3SAT is the function that maps a 3SAT formula $\varphi$ to 1 if it is satisfiable and to 0 otherwise, and QUADEQ is the function that maps a set $E$ of quadratic equations over $\{0, 1\}^n$ to 1 it has a solution and to 0 otherwise.

**Proof Idea:**

At the end of the day, a 3SAT formula can be thought of as a list of equations on some variables $x_0, \ldots, x_{n-1}$. Namely, the equations are that each of the $x_i$'s should be equal to either 0 or 1, and that the variables should satisfy some set of constraints which corresponds to the OR of three variables or their negation. To show that $3SAT \leq_p QUADEQ$ we need to give a polynomial-time reduction that maps a 3SAT formula $\varphi$ into a set of quadratic equations $E$ such that $E$ has a solution if and only if $\varphi$ is satisfiable. The idea is that we can transform a 3SAT formula $\varphi$ first to a set of cubic equations by mapping every constraint of the form $(x_{12} \lor x_{15} \lor x_{24})$ into an equation of the form $(1-x_{12})x_{15}(1-x_{24}) = 0$. We can then turn this into a quadratic equation by mapping any cubic equation of the form $x_i x_j x_k = 0$ into the two quadratic equations $y_{i,j} = x_i x_j$ and $y_{i,j,k} = 0$.

* 

**Proof of Theorem 13.4.** To prove Theorem 13.4 we need to give a polynomial-time transformation of every 3SAT formula $\varphi$ into a set of quadratic equations $E$, and prove that $3SAT(\varphi) = QUADEQ(E)$.

We now describe the transformation of a formula $\varphi$ to equations $E$ and show the completeness and soundness conditions. Recall that a 3SAT formula $\varphi$ is a formula such as $(x_{17} \lor \overline{x}_{101} \lor x_{57}) \land (x_{18} \lor \overline{x}_{19} \lor \overline{x}_{101}) \land \ldots$. That is, $\varphi$ is composed of the AND of $m$ 3SAT clauses
where a 3SAT clause is the OR of three variables or their negation. A quadratic equations instance \( E \) is composed of a list of equations, each of involving a sum of variables or their products, such as \( x_{19}x_{52} - x_{12} + 2x_{33} = 2 \), etc.. We will include the constraints \( x_i^2 - x_i = 0 \) for every \( i \in [n] \) in our equations, which means that we can restrict attention to assignments where \( x_i \in \{0, 1\} \) for every \( i \).

There is a natural way to map a 3SAT instance into a set of cubic equations \( E' \), and that is to map a clause such as \( (x_{17} \lor x_{101} \lor x_{57}) \) (which is equivalent to the negation of \( \overline{x}_{17} \land x_{101} \land \overline{x}_{57} \)) to the equation \((1 - x_{17})x_{101}(1 - x_{57}) = 0\). Therefore, we can map a formula \( \varphi \) with \( n \) variables \( m \) clauses into a set \( E' \) of \( m + n \) cubic equations on \( n \) variables (that is, one equation per each clause, plus one equation of the form \( x_i^2 - x_i = 0 \) for each variable to ensure that its value is in \( \{0, 1\} \)) such that every assignment \( a \in \{0, 1\}^n \) to the \( n \) variables satisfies the original formula if and only if it satisfies the equations of \( E' \).

To make the equations quadratic we introduce for every two distinct \( i, j \in [n] \) a variable \( y_{i,j} \) and include the constraint \( y_{i,j} - x_i x_j = 0 \) in the equations. This is a quadratic equation that ensures that \( y_{i,j} = x_i x_j \) for every such \( i, j \in [n] \). Now we can turn any cubic equation in the \( x \)'s into a quadratic equation in the \( x \) and \( y \) variables. For example, we can “open up the parentheses” of an equation such as \( (1 - x_{17})x_{101}(1 - x_{57}) = 0 \) to \( x_{101} - x_{17}x_{101} - x_{101}x_{57} + x_{17}x_{101}x_{57} = 0 \). We can now replace the cubic term \( x_{17}x_{101}x_{57} \) with the quadratic term \( y_{17,101}x_{57} \). This can be done for every cubic equation in the same way, replacing any cubic term \( x_i x_j x_k \) with the term \( y_{i,j} x_k \). The end result will be a set of \( m + n + \binom{n}{2} \) equations (one equation per clause, one equation per variable to ensure \( x_i^2 - x_i = 0 \), and one equation per pair \( i, j \) to ensure \( y_{i,j} = x_i x_j = 0 \)) on the \( n + \binom{n}{2} \) variables \( x_0, \ldots, x_{n-1} \) and \( y_{i,j} \) for all pairs of distinct variables \( i, j \).

To complete the proof we need to show that if we transform \( \varphi \) to \( E \) in this way then the 3SAT formula \( \varphi \) is satisfiable if and only if the equations \( E \) have a solution. This is essentially immediate from the construction, but as this is our first reduction, we spell this out fully:

- **Completeness**: We claim that if \( \varphi \) is satisfiable then the equations \( E \) have a solution. To prove this we need to show how to transform a satisfying assignment \( a \in \{0, 1\}^n \) to the variables of \( \varphi \) (that is, \( a_i \) is the value assigned to \( x_i \)) to a solution to the variables of \( E \). Specifically, if \( a \in \{0, 1\}^n \) is such an assignment then by design \( a \) satisfies all the cubic equations \( E' \) that we constructed above. But then, if we assign to the \( n + \binom{n}{2} \) variables the values \( a_0, \ldots, a_{n-1} \) and \( \{a_i, a_j\} \) for all \( \{i, j\} \subseteq [n] \) then by construction this will satisfy all the quadratic equations of \( E \) as well.
• **Soundness:** We claim that if the equations $E$ have a solution then $\varphi$ is satisfiable. Indeed, suppose that $z \in \mathbb{R}^{n+\binom{n}{2}}$ is a solution to the equations $E$. A priori $z$ could be any vector of $n + \binom{n}{2}$ numbers, but the fact that $E$ contains the equations $x_i^2 - x_i = 0$ and $y_{i,j} - x_i x_j = 0$ means that if $z$ satisfies these equations then the values it assigns to $x_i$ must be in $\{0, 1\}$ for every $i$, and the value it assigns to $y_{i,j}$ must be $x_i x_j$ for every $\{i, j\} \subseteq [n]$. Therefore by the way we constructed our equations, the value assigned $x$ must be a solution of the original cubic equations $E'$ and hence also of the original formula $\varphi$, which in particular implies $\varphi$ is satisfiable.

This reduction can be easily implemented in about a dozen lines of Python or any other programming language, see Fig. 13.3.

### 13.4 THE INDEPENDENT SET PROBLEM

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. But it also arises in very different settings, including trying to find structure in protein-protein interaction graphs.

To phrase independent set as a decision problem, we think of it as a function $ISET : \{0, 1\}^* \rightarrow \{0, 1\}$ 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 will 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 $x_{17} \lor x_5 \lor x_9$ as creating a conflict between the events “$x_{17} = 0$”, “$x_5 = 1$” and “$x_9 = 0$”, saying that these three cannot simultaneously 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.
**Proof of Theorem 13.5.** Given a 3SAT formula \( \varphi \) 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)

- A clause \( C \) in \( \varphi \) has the form \( C = y \lor y' \lor y'' \) 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 \( \varphi \), 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.

The above construction of \( G \) based on \( \varphi \) can clearly be carried out in polynomial time. Hence to prove the theorem we need to show that \( \varphi \) 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 \( \varphi \) has a satisfying assignment \( x^* \), then \( G \) has an independent set \( S^* \) of \( m \) vertices. Let us now show this.

Indeed, suppose that \( \varphi \) has a satisfying assignment \( x^* \in \{0, 1\}^n \). Then for every clause \( C = y \lor y' \lor y'' \) of \( \varphi \), one of the literals \( y, y', y'' \) must evaluate to true under the assignment \( x^* \) (as otherwise it would not satisfy \( \varphi \)). 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 \( \varphi \) 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 $\varphi$ 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 $\varphi$. 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 $\varphi$. Hence since $|S^*| = m$, it must have exactly one vertex in each such triangle. For every clause $C$ of $\varphi$, 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 $\varphi$, which is the definition of a satisfying assignment.

This completes the proof of Theorem 13.5.

### Solved Exercise 13.2 — Clique is equivalent to independent set.

The maximum clique problem corresponds to the function $\text{CLIQUE} : \{0, 1\}^* \to \{0, 1\}$ such that for a graph $G$ and a number $k$, $\text{CLIQUE}(G, k) = 1$ 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 $\text{CLIQUE} \leq_p \text{ISET}$ and $\text{ISET} \leq_p \text{CLIQUE}$.

### 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 $k$, $\text{ISET}(G, k) = \text{CLIQUE}(\overline{G}, k)$. Since the map $G \mapsto \overline{G}$ can be computed efficiently,
this yields a reduction \( \text{ISET} \leq_p \text{CLIQUE} \). Moreover, since \( \overline{G} = G \)
this yields a reduction in the other direction as well.

### 13.5 REDUCING INDEPENDENT SET TO MAXIMUM CUT

**Theorem 13.6 — Hardness of Max Cut.** \( \text{ISET} \leq_p \text{MAXCUT} \)

**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 \( u, v, s^* \), 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.

* 

**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: the graph \( H \) will contain all vertices of \( G \) (though not the edges between them!) and in addition to that will contain:

* 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. 13.5). 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 would be done. This might not always be the case but we will see that if \( I \) is not an independent set then its 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.

**13.6 REDUCING 3SAT TO LONGEST PATH**

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

\[
3SAT \leq_p \text{LONGPATH} \quad (13.6)
\]

**Proof Idea:**

To prove Theorem 13.7 need to show how to transform a 3CNF formula \( \varphi \) into a graph \( G \) and two vertices \( s, t \) such that \( G \) has a path of length at least \( k \) if and only if \( \varphi \) is satisfiable. The idea of the reduction is sketched in Fig. 13.7 and Fig. 13.8. 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 \( \varphi \) 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 \( x_{17} \lor x_{55} \lor x_{72} \) 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 corresponding 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.

\[\square\]

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 \( 3\text{SAT} \leq_p F \) for interesting functions \( F \). Sometimes

\[\square\]
we can do so by using transitivity of reductions: if \( 3\text{SAT} \leq_p G \) and \( G \leq_p F \) then \( 3\text{SAT} \leq_p F \).

### 13.7 Exercises

#### Exercise 13.1 — Transitivity of reductions. Prove that if \( F \leq_p G \) and \( G \leq_p H \) then \( F \leq_p H \).

### 13.8 Bibliographical Notes

Several notions of reductions are defined in the literature. The notion defined in Definition 13.1 is often known as a mapping reduction, many to one reduction or a Karp reduction.

The maximal (as opposed to maximum) independent set is the task of finding a “local maximum” of an independent set: an independent set \( S \) such that one cannot add a vertex to it without losing the independence property (such a set is known as a vertex cover). Unlike finding a maximum independent set, finding a maximal independent set can be done efficiently by a greedy algorithm, but this local maximum can be much smaller than the global maximum.

Reduction of independent set to max cut taken from these notes. Image of Hamiltonian Path through Dodecahedron by Christoph Sommer.

We have mentioned that the line between reductions used for algorithm design and showing hardness is sometimes blurry. An excellent example for this is the area of SAT Solvers (see [Gom+08]). In this field people use algorithms for SAT (that take exponential time in the worst case but often are much faster on many instances in practice) together with reductions of the form \( F \leq_p SAT \) to derive algorithms for other functions \( F \) of interest.

\(^4\) TODO: Maybe mention either in exercise or in body of the lecture some NP hard results motivated by science. For example, shortest superstring that is motivated by genome sequencing, protein folding, maybe others.