Algorithms (XVI)

Yu Yu

Shanghai Jiaotong University
Chapter 8. **NP**-complete problems
Search problems
Efficient algorithms

We have developed algorithms for

- finding shortest paths in graphs,
- minimum spanning trees in graphs,
- matchings in bipartite graphs,
- maximum increasing subsequences,
- maximum flows in networks,
- ......

All these algorithms are efficient, because in each case their time requirement grows as a polynomial function (such as $n$, $n^2$, or $n^3$) of the size of the input.
Exponential search space

In all these problems we are searching for a solution (path, tree, matching, etc.) from among an exponential population of possibilities.

All these problems could in principle be solved in exponential time by checking through all candidate solutions, one by one.

But an algorithm whose running time is $2^n$, or worse, is all but useless in practice.

*The quest for efficient algorithms is about finding clever ways to bypass this process of exhaustive search, using clues from the input in order to dramatically narrow down the search space.*

Now will see some other “search problems” in which again we are seeking a solution with particular properties among an exponential chaos of alternatives. The fastest algorithms we know for them are all exponential.
The instances of **Satisfiability or sat**: 

$$(x \lor y \lor \bar{z})(x \lor \bar{y})(y \lor \bar{z})(z \lor \bar{x})(\bar{x} \lor \bar{y} \lor \bar{z})$$

That is, a **Boolean formula in conjunctive normal form (CNF)**. It is a collection of **clauses** (the parentheses), each consisting of the **disjunction** (logical or, denoted $\lor$) of several **literals**, where a literal is either a **Boolean variable** (such as $x$) or the **negation** of one (such as $\bar{x}$).

A **satisfying truth assignment** is an assignment of **false** or **true** to each variable so that every clause contains a literal whose value is **true**.

The **sat** problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.
Satisfiability (cont’d)

SAT is a typical search problem.

We are given an instance $I$ (that is, some input data specifying the problem at hand, in this case a Boolean formula in conjunctive normal form), and we are asked to find a solution $S$ (an object that meets a particular specification, in this case an assignment that satisfies each clause).

*If no such solution exists, we must say so.*
Search problems

A search problem is specified by an algorithm $C$ that takes two inputs, an instance $I$ and a proposed solution $S$, and runs in time polynomial in $|I|$. We say $S$ is a solution to $I$ if and only if $C(I, S) = \text{true}$. 
The traveling salesman problem

In the traveling salesman problem \((\text{TSP})\) we are given \(n\) vertices 1, \ldots, \(n\) and all \(n(n - 1)/2\) distances between them, as well as a \textit{budget} \(b\). We are asked to find a \textit{tour}, a cycle that passes through every vertex exactly once, of total cost \(b\) or less – or to report that no such tour exists.

That is, we seek a \textit{permutation} \(\tau(1), \ldots, \tau(n)\) of the vertices such that when they are toured in this order, the total distance covered is at most \(b\):

\[
d_{\tau(1), \tau(2)} + d_{\tau(2), \tau(3)} + \cdots + d_{\tau(n), \tau(1)} \leq b.
\]

We have defined the \textit{TSP} as a search problem: given an instance, find a tour within the budget (or report that none exists).

But why are we expressing the traveling salesman problem in this way, when in reality it is an \textit{optimization} problem, in which the shortest possible tour is sought?
Turning an optimization problem into a search problem does not change its difficulty at all, because the two versions *reduce to one another*.

Any algorithm that solves the optimization TSP also readily solves the search problem: find the optimum tour and if it is within budget, return it; if not, there is no solution.

Conversely, an algorithm for the search problem can also be used to solve the optimization problem:

- First suppose that we somehow knew the cost of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget.
- We can find the optimum cost by *binary search*. 
Then why search instead of optimization?

Isn’t any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being optimal?

*The solution to a search problem should be easy to recognize, or as we put it earlier, polynomial-time checkable.*

Given a potential solution to the TSP, it is easy to check the properties “is a tour” (just check that each vertex is visited exactly once) and “has total length $\leq b$.”

*But how could one check the property “is optimal”*?
Euler Path

**Euler Path:**
Given a graph, find a path that *contains each edge exactly once*.

The answer is yes if and only if

(a) the graph is connected and

(b) every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.

Using above, it is easy to see that there is a polynomial time algorithm for Euler Path.
Rudrata Cycle

**Rudrata Cycle:**
Given a graph, find a cycle that visits each vertex exactly once.

In the literature this problem is known as the *Hamilton cycle problem.*
A **cut** is a set of edges whose removal leaves a graph disconnected.

**MINIMUM CUT**: given a graph and a budget $b$, find a cut with at most $b$ edges.

This problem can be solved in polynomial time by $n - 1$ **max-flow computations**: give each edge a capacity of 1, and find the maximum flow between some fixed node and every single other node. The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut.
Balanced Cut

In many graphs, the smallest cut leaves just a singleton vertex on one side – it consists of all edges adjacent to this vertex.

Far more interesting are small cuts that partition the vertices of the graph into nearly equal-sized sets.

**Balanced Cut:**
Given a graph with $n$ vertices and a budget $b$, partition the vertices into two sets $S$ and $T$ such that $|S|, |T| \geq n/3$ and such that there are at most $b$ edges between $S$ and $T$. 
INTEGRAL LINEAR PROGRAMMING (ILP):
We are given a set of linear inequalities $Ax \leq b$, where $A$ is an $m \times n$ matrix and $b$ is an $m$-vector;
an objective function specified by an $n$-vector $c$;
and finally, a goal $g$ (the counterpart of a budget in maximization problems).

We want to find a nonnegative integer $n$-vector $x$ such that $Ax \leq b$ and $c \cdot x \geq g$.

But there is a redundancy here: the last constraint $c \cdot x \geq g$ is itself a linear inequality and can be absorbed into $Ax \leq b$.

So, we define ILP to be following search problem: given $A$ and $b$, find a nonnegative integer vector $x$ satisfying the inequalities $Ax \leq b$. 

INTEGER LINEAR PROGRAMMING (ILP):
We are given a set of linear inequalities $Ax \leq b$, where $A$ is an $m \times n$ matrix and $b$ is an $m$-vector;
an objective function specified by an $n$-vector $c$;
and finally, a goal $g$ (the counterpart of a budget in maximization problems).

We want to find a nonnegative integer $n$-vector $x$ such that $Ax \leq b$ and $c \cdot x \geq g$.

But there is a redundancy here: the last constraint $c \cdot x \geq g$ is itself a linear inequality and can be absorbed into $Ax \leq b$.

So, we define ILP to be following search problem: given $A$ and $b$, find a nonnegative integer vector $x$ satisfying the inequalities $Ax \leq b$. 

Integer linear programming
Three-dimensional matching

**3D matching:**
There are $n$ boys and $n$ girls, but also $n$ pets, and the compatibilities among them are specified by a set of *triples*, each containing a boy, a girl, and a pet.

Intuitively, a triple $(b, g, p)$ means that boy $b$, girl $g$, and pet $p$ get along well together.

We want to find $n$ disjoint triples and thereby create $n$ harmonious households.
**INDEPENDENT SET:**
Given a graph and an integer $g$, find $g$ vertices, no two of which have an edge between them.

**VERTEX COVER:** Given a graph and an integer $b$, find $b$ vertices cover (touch) every edge.

**CLIQUE:**
Given a graph and an integer $g$, find $g$ vertices such that all possible edges between them are present.
LONGEST PATH: Given a graph $G$ with nonnegative edge weights and two distinguished vertices $s$ and $t$, along with a goal $g$.

We are asked to find a path from $s$ to $t$ with total weight at least $g$. To avoid trivial solutions we require that the path be simple, containing no repeated vertices.
KNAPSACK: We are given integer weights \( w_1, \ldots, w_n \) and integer values \( v_1, \ldots, v_n \) for \( n \) items.

We are also given a weight capacity \( W \) and a goal \( g \).

We seek a set of items whose total weight is at most \( W \) and whose total value is at least \( g \).

The problem is solvable in time \( O(nW) \) by dynamic programming.
Subset sum

SUBSET SUM:
Find a subset of a given set of integers that adds up to exactly $W$. 
**NP-complete problems**
Hard problems, easy problems
### Hard problems, easy problems

<table>
<thead>
<tr>
<th>Hard problems (NP-complete)</th>
<th>Easy problems (in P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3SAT</td>
<td>2SAT, HORNSAT</td>
</tr>
<tr>
<td>TRAVELING SALESMAN PROBLEM</td>
<td>MINIMUM SPANNING TREE</td>
</tr>
<tr>
<td>LONGEST PATH</td>
<td>SHORTEST PATH</td>
</tr>
<tr>
<td>3D MATCHING</td>
<td>BIPARTITE MATCHING</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>UNARY KNAPSACK</td>
</tr>
<tr>
<td>INDEPENDENT SET</td>
<td>INDEPENDENT SET ON TREES</td>
</tr>
<tr>
<td>INTEGER LINEAR PROGRAMMING</td>
<td>LINEAR PROGRAMMING</td>
</tr>
<tr>
<td>RUDRATA PATH</td>
<td>EULER PATH</td>
</tr>
<tr>
<td>BALANCED CUT</td>
<td>MINIMUM CUT</td>
</tr>
</tbody>
</table>
Recall a search problem is defined by:

▶ There is an efficient checking algorithm $C$ that takes as input the given instance $I$, as well as the proposed solution $S$, and outputs $true$ if and only if $S$ really is a solution to instance $I$.

▶ Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance.

We denote the class of all search problems by $NP$. 

NP
Recall a search problem is defined by:

- There is an efficient checking algorithm $C$ that takes as input the given instance $I$, as well as the proposed solution $S$, and outputs $true$ if and only if $S$ really is a solution to instance $I$.
- Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance.

We denote the class of all search problems by $NP$. 
Recall a search problem is defined by:

- There is an efficient checking algorithm $C$ that takes as input the given instance $I$, as well as the proposed solution $S$, and outputs true if and only if $S$ really is a solution to instance $I$. 

- Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance.

We denote the class of all search problems by $NP$. 
Recall a search problem is defined by:

- There is an efficient checking algorithm $C$ that takes as input the given instance $I$, as well as the proposed solution $S$, and outputs $\text{true}$ if and only if $S$ really is a solution to instance $I$.

- Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance.
Recall a search problem is defined by:

- There is an efficient checking algorithm $C$ that takes as input the given instance $I$, as well as the proposed solution $S$, and outputs true if and only if $S$ really is a solution to instance $I$.
- Moreover the running time of $C(I, S)$ is bounded by a polynomial in $|I|$, the length of the instance.

We denote the class of all search problems by $\textbf{NP}$.
We've seen many examples of search problems that are solvable in polynomial time. In such cases, there is an algorithm that takes as input an instance $I$ and has a running time polynomial in $|I|$. If $I$ has a solution, the algorithm returns such a solution; and if $I$ has no solution, the algorithm correctly reports so. The class of all search problems that can be solved in polynomial time is denoted $P$. 
We’ve seen many examples of \textbf{NP} search problems that are \emph{solvable in polynomial time}. 
We’ve seen many examples of \textbf{NP} search problems that are \emph{solvable in polynomial time}.

In such cases, there is an algorithm that takes as input an instance $I$ and has a running time polynomial in $|I|$. 
We’ve seen many examples of \textbf{NP} search problems that are \textit{solvable in polynomial time}.

In such cases, there is an algorithm that takes as input an instance $I$ and has a running time polynomial in $|I|$. If $I$ has a solution, the algorithm returns such a solution;
We’ve seen many examples of \textbf{NP} search problems that are \textit{solvable in polynomial time}.

In such cases, there is an algorithm that takes as input an instance $I$ and has a running time polynomial in $|I|$. If $I$ has a solution, the algorithm returns such a solution; and if $I$ has no solution, the algorithm correctly reports so.
We’ve seen many examples of \textbf{NP} search problems that are *solvable in polynomial time*. 

In such cases, there is an algorithm that takes as input an instance $I$ and has a running time polynomial in $|I|$. If $I$ has a solution, the algorithm returns such a solution; and if $I$ has no solution, the algorithm correctly reports so.

The class of all search problems that can be solved in polynomial time is denoted \textbf{P}.
Why \textbf{P} and \textbf{NP}? 
Why \textbf{P} and \textbf{NP}?

\textbf{P}: \textit{polynomial time}
Why $P$ and $NP$?

$P$: polynomial time

$NP$: nondeterministic polynomial time
P ≠ NP?

Input: A mathematical statement \( \phi \) and \( n \).

Problem: Find a proof of \( \phi \) of length \( \leq n \) if there is one.

The task of finding a proof for a given mathematical assertion is a search problem and is therefore in \( \text{NP} \), because a formal proof of a mathematical statement is written out in excruciating detail, it can be checked mechanically, line by line, by an efficient algorithm. So if \( P = \text{NP} \), there would be an efficient method to prove any theorem, thus eliminating the need for mathematicians!
Input: A mathematical statement $\varphi$ and $n$.
Problem: Find a proof of $\varphi$ of length $\leq$ if there is one.
P ≠ NP?

Input: A mathematical statement \( \varphi \) and \( n \).

Problem: Find a proof of \( \varphi \) of length \( \leq \) if there is one.

The task of finding a proof for a given mathematical assertion is a search problem and is therefore in \( \text{NP} \),
P ≠ NP?

**Input:** A mathematical statement \( \varphi \) and \( n \).

**Problem:** Find a proof of \( \varphi \) of length \( \leq \) if there is one.

The task of finding a proof for a given mathematical assertion is a search problem and is therefore in \textbf{NP}, because a formal proof of a mathematical statement is written out in excruciating detail, it can be checked mechanically, line by line, by an efficient algorithm.
Input: A mathematical statement $\varphi$ and $n$.

Problem: Find a proof of $\varphi$ of length $\leq$ if there is one.

The task of finding a proof for a given mathematical assertion is a search problem and is therefore in \textbf{NP}, because a formal proof of a mathematical statement is written out in excruciating detail, it can be checked mechanically, line by line, by an efficient algorithm.

So if $P = \text{NP}$, there would be an efficient method to prove any theorem, thus eliminating the need for mathematicians!
Reductions, again

Even if we believe $P \neq NP$, what about the specific problems on the left side of the table? On the basis of what evidence do we believe that these particular problems have no efficient algorithm? Such evidence is provided by reductions, which translate one search problem into another. We will show that the problems on the left side of the table are all, in some sense, exactly the same problem, the hardest search problems in $NP$. If even one of them has a polynomial time algorithm, then every problem in $NP$ has a polynomial time algorithm.
Even if we believe $P \neq NP$, what about the specific problems on the left side of the table?
Reductions, again

Even if we believe $\mathbf{P} \neq \mathbf{NP}$, what about the specific problems on the left side of the table?

On the basis of *what evidence* do we believe that these particular problems have no efficient algorithm?
Even if we believe $P \neq NP$, what about the specific problems on the left side of the table?

On the basis of *what evidence* do we believe that these particular problems have no efficient algorithm?

Such evidence is provided by *reductions*,
Reductions, again

Even if we believe $\mathbf{P} \neq \mathbf{NP}$, what about the specific problems on the left side of the table?

On the basis of *what evidence* do we believe that these particular problems have no efficient algorithm?

Such evidence is provided by *reductions*, which translate one search problem into another.
Reductions, again

Even if we believe $\mathbf{P} \neq \mathbf{NP}$, what about the specific problems on the left side of the table?

On the basis of *what evidence* do we believe that these particular problems have no efficient algorithm?

Such evidence is provided by *reductions*, which translate one search problem into another.

We will show that the problems on the left side of the table are all, in some sense, *exactly the same problem*,
Reductions, again

Even if we believe $\mathbf{P} \neq \mathbf{NP}$, what about the specific problems on the left side of the table?

On the basis of what evidence do we believe that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the problems on the left side of the table are all, in some sense, exactly the same problem, the hardest search problems in $\mathbf{NP}$.
Even if we believe $P \neq NP$, what about the specific problems on the left side of the table?

On the basis of *what evidence* do we believe that these particular problems have no efficient algorithm?

Such evidence is provided by *reductions*, which translate one search problem into another.

We will show that the problems on the left side of the table are all, in some sense, *exactly the same problem*, the *hardest* search problems in $NP$.

If even one of them has a polynomial time algorithm, then every problem in $NP$ has a polynomial time algorithm.
Reduction between search problems

A reduction from search problem $A$ to search problem $B$ is a polynomial time algorithm $f$ that transforms any instance $I$ of $A$ into an instance $f(I)$ of $B$, together with another polynomial time algorithm $h$ that maps any solution $S$ of $f(I)$ back into a solution $h(S)$ of $I$. If $f(I)$ has no solution, then neither does $I$. These two translation procedures $f$ and $h$ imply that any algorithm for $B$ can be converted into an algorithm for $A$ by bracketing it between $f$ and $h$. 
Reduction between search problems

A reduction from search problem $A$ to search problem $B$ is a polynomial time algorithm $f$ that transforms any instance $I$ of $A$ into an instance $f(I)$ of $B$, together with another polynomial time algorithm $h$ that maps any solution $S$ of $f(I)$ back into a solution $h(S)$ of $I$. If $f(I)$ has no solution, then neither does $I$. These two translation procedures $f$ and $h$ imply that any algorithm for $B$ can be converted into an algorithm for $A$ by bracketing it between $f$ and $h$. 
Reduction between search problems

A reduction from search problem $A$ to search problem $B$ is a polynomial time algorithm $f$ that transforms any instance $I$ of $A$ into an instance $f(I)$ of $B$, together with another polynomial time algorithm $h$ that maps any solution $S$ of $f(I)$ back into a solution $h(S)$ of $I$. If $f(I)$ has no solution, then neither does $I$. These two translation procedures $f$ and $h$ imply that any algorithm for $B$ can be converted into an algorithm for $A$ by bracketing it between $f$ and $h$. 
A reduction from search problem $A$ to search problem $B$ is a polynomial time algorithm $f$ that transforms any instance $I$ of $A$ into an instance $f(I)$ of $B$, together with another polynomial time algorithm $h$ that maps any solution $S$ of $f(I)$ back into a solution $h(S)$ of $I$.

If $f(I)$ has no solution, then neither does $I$. 

Reduction between search problems

A reduction from search problem $A$ to search problem $B$ is a polynomial time algorithm $f$ that transforms any instance $I$ of $A$ into an instance $f(I)$ of $B$, together with another polynomial time algorithm $h$ that maps any solution $S$ of $f(I)$ back into a solution $h(S)$ of $I$.

If $f(I)$ has no solution, then neither does $I$.

These two translation procedures $f$ and $h$ imply that any algorithm for $B$ can be converted into an algorithm for $A$ by \textit{bracketing it between $f$ and $h$}.
NP-completeness

Definition
An A search problem is \textit{NP}-complete if all other search problems reduce to it.

Remark
It is not clear a priori that \textit{NP}-complete problems exist at all.
Definition
A search problem is **NP-complete** if all other search problems reduce to it.
**NP-completeness**

**Definition**
A search problem is **NP-complete** if all other search problems reduce to it.

**Remark**
*It is not clear a priori that NP-complete problems exist at all.*
The two ways to use reductions

Assume there is a reduction from a problem A to a problem B. A \rightarrow B.

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

Reductions also have the convenient property that they compose. If A \rightarrow B and B \rightarrow C, then A \rightarrow C.
The two ways to use reductions

Assume there is a reduction from a problem $A$ to a problem $B$.  

1. If we can solve $B$ efficiently, then we can also solve $A$ efficiently.
2. If we know $A$ is hard, then $B$ must be hard too.

Reductions also have the convenient property that they compose. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. 
The two ways to use reductions

Assume there is a reduction from a problem $A$ to a problem $B$.

$$A \rightarrow B.$$
The two ways to use reductions

Assume there is a reduction from a problem $A$ to a problem $B$.

$A \rightarrow B$.

- If we can solve $B$ efficiently, then we can also solve $A$ efficiently.
Assume there is a reduction from a problem $A$ to a problem $B$.

\[ A \rightarrow B. \]

- If we can solve $B$ efficiently, then we can also solve $A$ efficiently.
- If we know $A$ is hard, then $B$ must be hard too.
The two ways to use reductions

Assume there is a reduction from a problem $A$ to a problem $B$.

$$A \rightarrow B.$$ 

- If we can solve $B$ efficiently, then we can also solve $A$ efficiently.
- If we know $A$ is hard, then $B$ must be hard too.

Reductions also have the convenient property that they *compose*. 
The two ways to use reductions

Assume there is a reduction from a problem $A$ to a problem $B$.

$$ A \rightarrow B. $$

- If we can solve $B$ efficiently, then we can also solve $A$ efficiently.
- If we know $A$ is hard, then $B$ must be hard too.

Reductions also have the convenient property that they *compose*.

If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. 

Factoring

The difficulty of factoring is of a different nature than that of the other hard search problems we have just seen. For example, nobody believes that factoring is $\text{NP}$-complete. One evidence is that a number can always be factored into primes. Another difference: factoring succumbs to the power of quantum computation, while $\text{sat}$, $\text{tsp}$ and the other $\text{NP}$-complete problems do not seem to.
The difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen.
Factoring

The difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen.

For example, nobody believes that FACTORING is NP-complete.
Factoring

The difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen.

For example, nobody believes that FACTORING is NP-complete. One evidence is that a number can always be factored into primes.
The difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen.

For example, nobody believes that FACTORING is NP-complete. One evidence is that a number can always be factored into primes.

Another difference: FACTORING succumbs to the power of quantum computation, while SAT, TSP, and the other NP-complete problems do not seem to.
The reductions
Rudrata \((s, t)\)-Path $\rightarrow$ Rudrata Cycle

Recall the Rudrata Cycle problem: given a graph, is there a cycle that passes through each vertex exactly once?

We can also formulate the closely related Rudrata \((s, t)\)-Path problem, in which two vertices \(s\) and \(t\) are specified, and we want a path starting at \(s\) and ending at \(t\) that goes through each vertex exactly once.

Is it possible that Rudrata Cycle is easier than Rudrata \((s, t)\)-PATH?

We will show by a reduction that the answer is no. The reduction maps an instance \((G = (V, E), s, t)\) of Rudrata \((s, t)\)-Path into an instance \(G' = (V', E')\) of Rudrata Cycle as follows: \(G'\) is simply \(G\) with an additional vertex \(x\) and two new edges \(\{s, x\}\) and \(\{x, t\}\).
Recall the Rudrata Cycle problem: given a graph, is there a cycle that passes through each vertex exactly once?
Recall the **Rudrata Cycle** problem: given a graph, is there a cycle that passes through each vertex exactly once?

We can also formulate the closely related **Rudrata (s, t)-Path** problem, in which two vertices s and t are specified, and we want a path starting at s and ending at t that goes through each vertex exactly once.
Recall the Rudrata Cycle problem: given a graph, is there a cycle that passes through each vertex exactly once?

We can also formulate the closely related Rudrata (s, t)-Path problem, in which two vertices s and t are specified, and we want a path starting at s and ending at t that goes through each vertex exactly once.

Is it possible that Rudrata Cycle is easier than Rudrata (s, t)-PATH?
Recall the Rudrata Cycle problem: given a graph, is there a cycle that passes through each vertex exactly once?

We can also formulate the closely related Rudrata \((s, t)\)-Path problem, in which two vertices \(s\) and \(t\) are specified, and we want a path starting at \(s\) and ending at \(t\) that goes through each vertex exactly once.

Is it possible that Rudrata Cycle is easier than Rudrata \((s, t)\)-PATH? We will show by a reduction that the answer is no.
Recall the Rudrata Cycle problem: given a graph, is there a cycle that passes through each vertex exactly once?

We can also formulate the closely related Rudrata \((s, t)\)-Path problem, in which two vertices \(s\) and \(t\) are specified, and we want a path starting at \(s\) and ending at \(t\) that goes through each vertex exactly once.

Is it possible that Rudrata Cycle is easier than Rudrata \((s, t)\)-PATH? We will show by a reduction that the answer is no.

The reduction maps an instance \((G = (V, E), s, t)\) of Rudrata \((s, t)\)-Path into an instance \(G' = (V', E')\) of Rudrata Cycle as follows: \(G'\) is simply \(G\) with an additional vertex \(x\) and two new edges \(\{s, x\}\) and \(\{x, t\}\).
3Sat $\rightarrow$ Independent Set

In 3Sat the input is a set of clauses, each with three or fewer literals, for example $(\overline{x} \lor y \lor \overline{z})(x \lor \overline{y} \lor z)(x \lor y \lor \overline{z})$, and the aim is to find a satisfying truth assignment.

In Independent Set the input is a graph and a number $g$, and the problem is to find a set of $g$ pairwise non-adjacent vertices.

Given an instance $I$ of 3Sat, we create an instance $(G, g)$ of Independent Set as follows.

▶ Graph $G$ has a triangle for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, and has additional edges between any two vertices that represent opposite literals.

▶ The goal $g$ is set to the number of clauses.
In $3\text{Sat}$ the input is a set of clauses, each with three or fewer literals, for example

$$(\bar{x} \lor y \lor \bar{z})(x \lor \bar{y} \lor z)(x \lor y \lor z)(\bar{x} \lor y),$$

and the aim is to find a satisfying truth assignment.
In 3Sat the input is a set of clauses, each with three or fewer literals, for example

\[(\bar{x} \lor y \lor \bar{z})(x \lor \bar{y} \lor z)(x \lor y \lor z)(\bar{x} \lor y),\]

and the aim is to find a satisfying truth assignment.

In Independent Set the input is a graph and a number \(g\), and the problem is to find a set of \(g\) pairwise non-adjacent vertices.
In 3Sat the input is a set of clauses, each with three or fewer literals, for example

\[(\bar{x} \lor y \lor \bar{z})(x \lor \bar{y} \lor z)(x \lor y \lor z)(\bar{x} \lor y),\]

and the aim is to find a satisfying truth assignment.

In Independent Set the input is a graph and a number \(g\), and the problem is to find a set of \(g\) pairwise non-adjacent vertices.

Given an instance \(I\) of 3Sat, we create an instance \((G, g)\) of Independent Set as follows.
In $3$Sat the input is a set of clauses, each with three or fewer literals, for example
\[(\bar{x} \lor y \lor \bar{z})(x \lor \bar{y} \lor z)(x \lor y \lor z)(\bar{x} \lor y),\]
and the aim is to find a satisfying truth assignment.

In Independent Set the input is a graph and a number $g$, and the problem is to find a set of $g$ pairwise non-adjacent vertices.

Given an instance $I$ of $3$Sat, we create an instance $(G, g)$ of Independent Set as follows.

- Graph $G$ has a triangle for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, and has additional edges between any two vertices that represent opposite literals.
In **3Sat** the input is a set of clauses, each with **three or fewer literals**, for example

\[(\bar{x} \lor y \lor \bar{z})(x \lor \bar{y} \lor z)(x \lor y \lor z)(\bar{x} \lor y)\],

and the aim is to find a satisfying truth assignment.

In **Independent Set** the input is a graph and a number \(g\), and the problem is to find a set of \(g\) pairwise non-adjacent vertices.

Given an instance \(I\) of **3Sat**, we create an instance \((G, g)\) of **Independent Set** as follows.

- Graph \(G\) has a **triangle** for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, and has additional edges between any two vertices that represent opposite literals.
- The goal \(g\) is set to the number of clauses.
\textbf{Sat} \rightarrow \textbf{3Sat}

This is an interesting and common kind of reduction, from a problem to a special case of itself. Given an instance $I$ of Sat, use exactly the same instance for 3Sat, except that any clause with more than three literals, $(a_1 \lor a_2 \lor \cdots \lor a_k)$ (where the $a_i$'s are literals and $k > 3$), is replaced by a set of clauses, $(a_1 \lor a_2 \lor y_1)(\overline{y_1} \lor a_3 \lor y_2)(\overline{y_2} \lor a_4 \lor y_3) \cdots (\overline{y_{k-3}} \lor a_{k-1} \lor a_k)$, where the $y_i$'s are new variables. Call the resulting 3Sat instance $I'$. The conversion from $I$ to $I'$ is clearly polynomial time. $I'$ is equivalent to $I$ in terms of satisfiability, because for any assignment to the $a_i$'s, \[\{(a_1 \lor a_2 \lor \cdots \lor a_k)\text{ is satisfied} \iff \{\text{there is a setting for the } y_i \text{'s for which } (a_1 \lor a_2 \lor y_1)(\overline{y_1} \lor a_3 \lor y_2)(\overline{y_2} \lor a_4 \lor y_3) \cdots (\overline{y_{k-3}} \lor a_{k-1} \lor a_k)\text{ are all satisfied}\}\]
This is an interesting and common kind of reduction, from a problem to a special case of itself.
This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an instance $I$ of $\text{SAT}$, use exactly the same instance for $\text{3SAT}$, except that any clause with more than three literals, $(a_1 \lor a_2 \lor \cdots \lor a_k)$ (where the $a_i$’s are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \lor a_2 \lor y_1)(\bar{y}_1 \lor a_3 \lor y_2)(\bar{y}_2 \lor a_4 \lor y_3) \cdots (\bar{y}_{k-3} \lor a_{k-1} \lor a_k),$$

where the $y_i$’s are new variables.
This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an instance $I$ of $\text{SAT}$, use exactly the same instance for $\text{3SAT}$, except that any clause with more than three literals, $(a_1 \lor a_2 \lor \cdots \lor a_k)$ (where the $a_i$'s are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \lor a_2 \lor y_1)(\bar{y}_1 \lor a_3 \lor y_2)(\bar{y}_2 \lor a_4 \lor y_3) \cdots (\bar{y}_{k-3} \lor a_{k-1} \lor a_k),$$

where the $y_i$'s are new variables.

Call the resulting $\text{3SAT}$ instance $I'$. The conversion from $I$ to $I'$ is clearly polynomial time.
This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an instance $I$ of $\text{SAT}$, use exactly the same instance for $\text{3SAT}$, except that any clause with more than three literals, $(a_1 \lor a_2 \lor \cdots \lor a_k)$ (where the $a_i$'s are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \lor a_2 \lor y_1)(\bar{y}_1 \lor a_3 \lor y_2)(\bar{y}_2 \lor a_4 \lor y_3) \cdots (\bar{y}_{k-3} \lor a_{k-1} \lor a_k),$$

where the $y_i$'s are new variables.

Call the resulting $\text{3SAT}$ instance $I'$. The conversion from $I$ to $I'$ is clearly polynomial time.

$I'$ is equivalent to $I$ in terms of satisfiability, because for any assignment to the $a_i$'s,

$$\left\{ (a_1 \lor a_2 \lor \cdots \lor a_k) \text{ is satisfied} \right\} \iff \left\{ \text{there is a setting for the } y_i \text{'s for which } (a_1 \lor a_2 \lor y_1)(\bar{y}_1 \lor a_3 \lor y_2)(\bar{y}_2 \lor a_4 \lor y_3) \cdots (\bar{y}_{k-3} \lor a_{k-1} \lor a_k) \text{ are all satisfied} \right\}$$
In fact, \(\text{3Sat}\) remains hard even under the further restriction that no variable appears in more than three clauses. Suppose that in the \(\text{3Sat}\) instance, variable \(x\) appears in \(k > 3\) clauses. Then replace its first appearance by \(x_1\), its second appearance by \(x_2\), and so on, replacing each of its \(k\) appearances by a different new variable. Finally, add the clauses:

\[(\bar{x}_1 \lor x_2)(\bar{x}_2 \lor x_3) \cdots (\bar{x}_k \lor x_1)\]

In the new formula no variable appears more than three times (and in fact, no literal appears more than twice).
In fact, 3SAT remains hard even under the further restriction that no variable appears in more than three clauses.
In fact, \( \text{3Sat} \) remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the \( \text{3Sat} \) instance, variable \( x \) appears in \( k > 3 \) clauses.
In fact, 3Sat remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3Sat instance, variable $x$ appears in $k > 3$ clauses. Then replace its first appearance by $x_1$, its second appearance by $x_2$, and so on, replacing each of its $k$ appearances by a different new variable.
In fact, 3Sat remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3Sat instance, variable $x$ appears in $k > 3$ clauses. Then replace its first appearance by $x_1$, its second appearance by $x_2$, and so on, replacing each of its $k$ appearances by a different new variable. Finally, add the clauses

$$\overline{x}_1 \lor x_2 \overline{x}_2 \lor x_3 \overline{x}_3 \lor x_4 \cdots \overline{x}_k \lor x_1.$$
In fact, $3\text{Sat}$ remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the $3\text{Sat}$ instance, variable $x$ appears in $k > 3$ clauses. Then replace its first appearance by $x_1$, its second appearance by $x_2$, and so on, replacing each of its $k$ appearances by a different new variable. Finally, add the clauses

$$(\overline{x_1} \lor x_2)(\overline{x_2} \lor x_3) \cdots (\overline{x_k} \lor x_1).$$

In the new formula no variable appears more than three times (and in fact, no literal appears more than twice).
**Independent Set → Vertex Cover**

To reduce *Independent Set* to *Vertex Cover* we just need to notice that a set of nodes $S$ is a vertex cover of graph $G$ (that is, $S$ touches every edge in $E$) if and only if the remaining nodes, $V - S$, are an independent set of $G$.

Therefore, to solve an instance $(G, g)$ of *Independent Set*, simply look for a vertex cover of $G$ with $|V| - g$ nodes. If such a vertex cover exists, then take all nodes not in it. If no such vertex cover exists, then $G$ cannot possibly have an independent set of size $g$. 
To reduce Independent Set to Vertex Cover we just need to notice that a set of nodes $S$ is a vertex cover of graph $G = (V, E)$ (that is, $S$ touches every edge in $E$) if and only if the remaining nodes, $V - S$, are an independent set of $G$. 
To reduce **Independent Set** to **Vertex Cover** we just need to notice that a set of nodes $S$ is a vertex cover of graph $G = (V, E)$ (that is, $S$ touches every edge in $E$) if and only if the remaining nodes, $V - S$, are an independent set of $G$.

Therefore, to solve an instance $(G, g)$ of **Independent Set**, simply look for a vertex cover of $G$ with $|V| - g$ nodes.
To reduce \textbf{Independent Set} to \textbf{Vertex Cover} we just need to notice that a set of nodes \( S \) is a vertex cover of graph \( G = (V, E) \) (that is, \( S \) touches every edge in \( E \)) if and only if the remaining nodes, \( V - S \), are an independent set of \( G \).

Therefore, to solve an instance \((G, g)\) of \textbf{Independent Set}, simply look for a vertex cover of \( G \) with \(|V| - g\) nodes.

If such a vertex cover exists, then take all nodes not in it. If no such vertex cover exists, then \( G \) cannot possibly have an independent set of size \( g \).
Define the complement of a graph $G = (V, E)$ to be $\overline{G} = (V, \overline{E})$, where $\overline{E}$ contains precisely those unordered pairs of vertices that are not in $E$.

Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $G$. To paraphrase, these nodes have no edges between them in $G$ if and only if they have all possible edges between them in $\overline{G}$.

Therefore, we can reduce Independent Set to Clique by mapping an instance $(G, g)$ of Independent Set to the corresponding instance $(\overline{G}, g)$ of Clique; the solution to both is identical.
Define the complement of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in $E$. Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $G$. To paraphrase, these nodes have no edges between them in $G$ if and only if they have all possible edges between them in $\bar{G}$. Therefore, we can reduce Independent Set to Clique by mapping an instance $(G, g)$ of Independent Set to the corresponding instance $(\bar{G}, g)$ of Clique; the solution to both is identical.
Define the **complement** of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in $E$.

Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $G$. 
**Independent Set → Clique**

Define the **complement** of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in $E$.

Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $\bar{G}$.

To paraphrase, these nodes have no edges between them in $G$ if and only if they have all possible edges between them in $\bar{G}$. 
Define the **complement** of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in $E$.

Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $G$.

To paraphrase, these nodes have no edges between them in $G$ if and only if they have all possible edges between them in $\bar{G}$.

Therefore, we can reduce **Independent Set** to **Clique** by mapping an instance $(G, g)$ of **Independent Set** to the corresponding instance $(\bar{G}, g)$ of **Clique**;
Define the complement of a graph $G = (V, E)$ to be $\bar{G} = (V, \bar{E})$, where $\bar{E}$ contains precisely those unordered pairs of vertices that are not in $E$.

Then a set of nodes $S$ is an independent set of $G$ if and only if $S$ is a clique of $G$. To paraphrase, these nodes have no edges between them in $G$ if and only if they have all possible edges between them in $\bar{G}$.

Therefore, we can reduce Independent Set to Clique by mapping an instance $(G, g)$ of Independent Set to the corresponding instance $(\bar{G}, g)$ of Clique; the solution to both is identical.
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet. Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0,...,p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.) Then any matching must contain either the two triples $(b_0, g_1, p_0)$, $(b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1)$, $(b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states: it behaves like a Boolean variable!

To then transform an instance of 3Sat to one of 3D Matching, we start by creating a copy of the preceding gadget for each variable $x$. Call the resulting nodes $p_x^1$, $b_x^0$, $g_x^1$, and so on. The intended interpretation is that boy $b_x^0$ is matched with girl $g_x^1$ if $x = \text{true}$, and with girl $g_x^0$ if $x = \text{false}$.
3Sat → 3D Matching

Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

To then transform an instance of 3Sat to one of 3D Matching, we start by creating a copy of the preceding gadget for each variable $x$. Call the resulting nodes $p_{x1}$, $b_{x0}$, $g_{x1}$, and so on. The intended interpretation is that boy $b_{x0}$ is matched with girl $g_{x1}$ if $x = \text{true}$, and with girl $g_{x0}$ if $x = \text{false}$. 
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples.

Therefore, this "gadget" has two possible states: it behaves like a Boolean variable!
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples $(b_0, g_1, p_0), (b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1), (b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match.
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples $(b_0, g_1, p_0)$, $(b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1)$, $(b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states:
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples $(b_0, g_1, p_0)$, $(b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1)$, $(b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states: it behaves like a Boolean variable!
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples $(b_0, g_1, p_0)$, $(b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1)$, $(b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states: it behaves like a Boolean variable!

To then transform an instance of 3Sat to one of 3D Matching, we start by creating a copy of the preceding gadget for each variable $x$. 
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys $b_0$ and $b_1$ and the two girls $g_0$ and $g_1$ are not involved in any other triples. (The four pets $p_0, \ldots, p_3$ will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples $(b_0, g_1, p_0)$, $(b_1, g_0, p_2)$ or the two triples $(b_0, g_0, p_1)$, $(b_1, g_1, p_3)$, because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states: it behaves like a Boolean variable!

To then transform an instance of 3Sat to one of 3D Matching, we start by creating a copy of the preceding gadget for each variable $x$.

Call the resulting nodes $p_{x1}$, $b_{x0}$, $g_{x1}$, and so on.
Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.

Suppose that the two boys \( b_0 \) and \( b_1 \) and the two girls \( g_0 \) and \( g_1 \) are not involved in any other triples. (The four pets \( p_0, \ldots, p_3 \) will of course belong to other triples as well; for otherwise the instance would trivially have no solution.)

Then any matching must contain either the two triples \((b_0, g_1, p_0), (b_1, g_0, p_2)\) or the two triples \((b_0, g_0, p_1), (b_1, g_1, p_3)\), because these are the only ways in which these two boys and girls can find any match. Therefore, this “gadget” has two possible states: it behaves like a Boolean variable!

To then transform an instance of 3Sat to one of 3D Matching, we start by creating a copy of the preceding gadget for each variable \( x \).

Call the resulting nodes \( p_{x1}, b_{x0}, g_{x1}, \) and so on. The intended interpretation is that boy \( b_{x0} \) is matched with girl \( g_{x1} \) if \( x = \text{true} \), and with girl \( g_{x0} \) if \( x = \text{false} \).
Next we must create triples that somehow mimic clauses. For \( c = (x \lor \overline{y} \lor z) \), we introduce a new boy \( b_c \) and a new girl \( g_c \). They will be involved in three triples, one for each literal in the clause.

And the pets in these triples must reflect the three ways whereby the clause can be satisfied:

1. \( x = \text{true} \)
2. \( y = \text{false} \)
3. \( z = \text{true} \).

For (1), we have the triple \((b_c, g_c, p_{x1})\), where \( p_{x1} \) is the pet \( p_1 \) in the gadget for \( x \).

If \( x = \text{true} \), then \( b_{x0} \) is matched with \( g_{x1} \) and \( b_{x1} \) with \( g_{x0} \), and so pets \( p_{x0} \) and \( p_{x2} \) are taken. In which case \( b_c \) and \( g_c \) can be matched with \( p_{x1} \).

But if \( x = \text{false} \), then \( p_{x1} \) and \( p_{x3} \) are taken, and so \( g_c \) and \( b_c \) cannot be accommodated this way.
Next we must create triples that somehow mimic clauses.
Next we must create triples that somehow mimic clauses.

For
\[ c = (x \lor \neg y \lor z), \]

we introduce a new boy \( b_c \) and a new girl \( g_c \).
Next we must create triples that somehow mimic clauses.

For
\[ c = (x \lor \overline{y} \lor z), \]
we introduce a new boy \( b_c \) and a new girl \( g_c \).

They will be involved in three triples, one for each literal in the clause.
Next we must create triples that somehow mimic clauses. For

\[ c = (x \lor \bar{y} \lor z), \]

we introduce a new boy \( b_c \) and a new girl \( g_c \).

They will be involved in three triples, one for each literal in the clause. And the pets in these triples must reflect the three ways whereby the clause can be satisfied:
3Sat $\rightarrow$ 3D Matching (cont’d)

Next we must create triples that somehow mimic clauses.

For

$$c = (x \lor \overline{y} \lor z),$$

we introduce a new boy $b_c$ and a new girl $g_c$.

They will be involved in three triples, one for each literal in the clause. And the pets in these triples must reflect the three ways whereby the clause can be satisfied:

1. $x = \text{true}$,
2. $y = \text{false}$,
3. $z = \text{true}$.

For (1), we have the triple $(b_c, g_c, p_{x1})$, where $p_{x1}$ is the pet $p_1$ in the gadget for $x$. 
Next we must create triples that somehow mimic clauses.

For

\[ c = (x \lor \overline{y} \lor z), \]

we introduce a new boy \( b_c \) and a new girl \( g_c \).

They will be involved in three triples, one for each literal in the clause. And the pets in these triples must reflect the three ways whereby the clause can be satisfied:

1. \( x = \text{true} \),
2. \( y = \text{false} \),
3. \( z = \text{true} \).

For (1), we have the triple \((b_c, g_c, p_{x1})\), where \( p_{x1} \) is the pet \( p_1 \) in the gadget for \( x \).

- If \( x = \text{true} \), then \( b_{x0} \) is matched with \( g_{x1} \) and \( b_{x1} \) with \( g_{x0} \), and so pets \( p_{x0} \) and \( p_{x2} \) are taken. In which case \( b_c \) and \( g_c \) can be matched with \( p_{x1} \).
Next we must create triples that somehow mimic clauses.

For
\[ c = (x \lor \bar{y} \lor z), \]
we introduce a new boy \( b_c \) and a new girl \( g_c \).

They will be involved in three triples, one for each literal in the clause. And the pets in these triples must reflect the three ways whereby the clause can be satisfied:

1. \( x = \text{true} \),
2. \( y = \text{false} \),
3. \( z = \text{true} \).

For (1), we have the triple \((b_c, g_c, p_{x1})\), where \( p_{x1} \) is the pet \( p_1 \) in the gadget for \( x \).

- If \( x = \text{true} \), then \( b_{x0} \) is matched with \( g_{x1} \) and \( b_{x1} \) with \( g_{x0} \), and so pets \( p_{x0} \) and \( p_{x2} \) are taken. In which case \( b_c \) and \( g_c \) can be matched with \( p_{x1} \).
- But if \( x = \text{false} \), then \( p_{x1} \) and \( p_{x3} \) are taken, and so \( g_c \) and \( b_c \) cannot be accommodated this way.
We do the same thing for the other two literals of the clause, which yield triples involving \( b \) and \( g \) with either \( p_y^0 \) or \( p_y^2 \) (for the negated variable \( y \)) and with either \( p_z^1 \) or \( p_z^3 \) (for variable \( z \)).

We have to make sure that for every occurrence of a literal in a clause there is a different pet to match with \( b \) and \( g \).

But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete: From any matching we can recover a satisfying truth assignment by simply looking at each variable gadget and seeing with which girl \( b_x^0 \) was matched.

And from any satisfying truth assignment we can match the gadget corresponding to each variable \( x \) so that triples \((b_x^0, g_x^1, p_x^0)\) and \((b_x^1, g_x^0, p_x^2)\) are chosen if \( x = \text{true} \) and triples \((b_x^0, g_x^0, p_x^1)\) and \((b_x^1, g_x^1, p_x^3)\) are chosen if \( x = \text{false} \); and for each clause match \( b_c \) and \( g_c \) with the pet that corresponds to one of its satisfying literals.
3Sat $\rightarrow$ 3D Matching (cont’d)

We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).
We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).

We have to make sure that for every occurrence of a literal in a clause $c$ there is a different pet to match with $b_c$ and $g_c$. 
We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).

We have to make sure that for every occurrence of a literal in a clause $c$ there is a different pet to match with $b_c$ and $g_c$. But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.
We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).

We have to make sure that for every occurrence of a literal in a clause $c$ there is a different pet to match with $b_c$ and $g_c$.

But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete:
3Sat $\rightarrow$ 3D Matching (cont’d)

We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).

We have to make sure that for every occurrence of a literal in a clause $c$ there is a different pet to match with $b_c$ and $g_c$. But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete:

From any matching we can recover a satisfying truth assignment by simply looking at each variable gadget and seeing with which girl $b_x0$ was matched.
We do the same thing for the other two literals of the clause, which yield triples involving $b_c$ and $g_c$ with either $p_{y0}$ or $p_{y2}$ (for the negated variable $y$) and with either $p_{z1}$ or $p_{z3}$ (for variable $z$).

We have to make sure that for every occurrence of a literal in a clause $c$ there is a different pet to match with $b_c$ and $g_c$.

But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete:

From any matching we can recover a satisfying truth assignment by simply looking at each variable gadget and seeing with which girl $b_{x0}$ was matched.

And from any satisfying truth assignment we can match the gadget corresponding to each variable $x$ so that triples $(b_{x0}, g_{x1}, p_{x0})$ and $(b_{x1}, g_{x0}, p_{x2})$ are chosen if $x = \text{true}$.
We do the same thing for the other two literals of the clause, which yield triples involving \( b_c \) and \( g_c \) with either \( p_{y0} \) or \( p_{y2} \) (for the negated variable \( y \)) and with either \( p_{z1} \) or \( p_{z3} \) (for variable \( z \)).

We have to make sure that for every occurrence of a literal in a clause \( c \) there is a different pet to match with \( b_c \) and \( g_c \). But this is easy: by an earlier reduction we can assume that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for unnegated.

The reduction now seems complete:

From any matching we can recover a satisfying truth assignment by simply looking at each variable gadget and seeing with which girl \( b_{x0} \) was matched.

And from any satisfying truth assignment we can match the gadget corresponding to each variable \( x \) so that triples \((b_{x0}, g_{x1}, p_{x0})\) and \((b_{x1}, g_{x0}, p_{x2})\) are chosen if \( x = \text{true} \) and triples \((b_{x0}, g_{x0}, p_{x1})\) and \((b_{x1}, g_{x1}, p_{x3})\) are chosen if \( x = \text{false} \); and for each clause \( c \) match \( b_c \) and \( g_c \) with the pet that corresponds to one of its satisfying literals.
But one last problem remains:

In the matching defined so far, some pets may be left unmatched. In fact, if there are $n$ variables and $m$ clauses, then exactly $2^n - m$ pets will be left unmatched. But this is easy to fix: Add $2^n - m$ new boy-girl couples that are "generic animal-lovers," and match them by triples with all the pets!
But one last problem remains:
But one last problem remains: In the matching defined so far, some pets may be left unmatched.
But one last problem remains: In the matching defined so far, some pets may be left unmatched.

In fact, if there are $n$ variables and $m$ clauses, then exactly $2n - m$ pets will be left unmatched.
But one last problem remains: In the matching defined so far, some pets may be left unmatched.

In fact, if there are $n$ variables and $m$ clauses, then exactly $2n - m$ pets will be left unmatched.

But this is easy to fix: Add $2n - m$ new boy-girl couples that are “generic animal-lovers,” and match them by triples with all the pets!
3D Matching $\rightarrow$ ZOE

In ZOE (Zero-One Equations) we are given an $m \times n$ matrix $A$ with $0-1$ entries, and we must find a $0-1$ vector $x = (x_1, \ldots, x_n)$ such that the $m$ equations $Ax = 1$, are satisfied.

Here is how we express an instance of 3D Matching ($m$ boys, $m$ girls, $m$ pets, and $n$ boy-girl-pet triples) in the language of ZOE.

We have 0-1 variables $x_1, \ldots, x_n$, one per triple, where $x_i = 1$ means that the $i$th triple is chosen for the matching, and $x_i = 0$ means that it is not chosen.

Now all we have to do is write equations stating that the solution described by the $x_i$'s is a legitimate matching.

For each boy (or girl, or pet), suppose that the triples containing him (or her, or it) are those numbered $j_1, j_2, \ldots, j_k$; the appropriate equation is then $x_{j_1} + x_{j_2} + \cdots + x_{j_k} = 1$. 
In ZOE (Zero-One Equations) we are given an $m \times n$ matrix $A$ with 0–1 entries, and we must find a 0–1 vector $x = (x_1, \ldots, x_n)$ such that the $m$ equations

$$Ax = 1,$$

are satisfied.
3D Matching $\rightarrow$ ZOE

In ZOE (Zero-One Equations) we are given an $m \times n$ matrix $A$ with 0–1 entries, and we must find a 0–1 vector $x = (x_1, \ldots, x_n)$ such that the $m$ equations

$$Ax = 1,$$

are satisfied.

Here is how we express an instance of 3D Matching ($m$ boys, $m$ girls, $m$ pets, and $n$ boy-girl-pet triples) in the language of ZOE.
In ZOE (Zero-One Equations) we are given an $m \times n$ matrix $A$ with $0 - 1$ entries, and we must find a $0 - 1$ vector $x = (x_1, \ldots, x_n)$ such that the $m$ equations

$$Ax = 1,$$

are satisfied.

Here is how we express an instance of 3D Matching ($m$ boys, $m$ girls, $m$ pets, and $n$ boy-girl-pet triples) in the language of ZOE.

We have $0 - 1$ variables $x_1, \ldots, x_n$, one per triple, where $x_i = 1$ means that the $i$th triple is chosen for the matching, and $x_i = 0$ means that it is not chosen.
3D Matching $\rightarrow$ ZOE

In ZOE (Zero-One Equations) we are given an $m \times n$ matrix $A$ with 0–1 entries, and we must find a 0–1 vector $x = (x_1, \ldots, x_n)$ such that the $m$ equations

$$Ax = 1,$$

are satisfied.

Here is how we express an instance of 3D Matching ($m$ boys, $m$ girls, $m$ pets, and $n$ boy-girl-pet triples) in the language of ZOE.

We have 0–1 variables $x_1, \ldots, x_n$, one per triple, where $x_i = 1$ means that the $i$th triple is chosen for the matching, and $x_i = 0$ means that it is not chosen.

Now all we have to do is write equations stating that the solution described by the $x_i$’s is a legitimate matching.

For each boy (or girl, or pet), suppose that the triples containing him (or her, or it) are those numbered $j_1, j_2, \ldots, j_k$; the appropriate equation is then

$$x_{j_1} + x_{j_2} + \cdots + x_{j_k} = 1.$$
ZOE $\rightarrow$ Subset Sum

This is a reduction between two special cases of ILP: one with many equations but only 0−1 coefficients, and the other with a single equation but arbitrary integer coefficients. The reduction is based on: 0−1 vectors can encode numbers! We are looking for a set of columns of $A$ that, added together, make up the all-1's vector. But if we think of the columns as binary integers (read from top to bottom), we are looking for a subset of the integers corresponding to the columns of $A$ that add up to the binary integer 11...1. And this is an instance of Subset Sum. The reduction seems complete!
This is a reduction between two special cases of ILP:
This is a reduction between two special cases of ILP: one with many equations but only $0 - 1$ coefficients,
This is a reduction between two special cases of ILP: one with many equations but only 0 – 1 coefficients, and the other with a single equation but arbitrary integer coefficients.
This is a reduction between two special cases of ILP: one with many equations but only $0 - 1$ coefficients, and the other with a single equation but arbitrary integer coefficients.

The reduction is based on: $0 - 1$ vectors can encode numbers!
This is a reduction between two special cases of ILP: one with many equations but only 0–1 coefficients, and the other with a single equation but arbitrary integer coefficients.

The reduction is based on: 0–1 vectors can encode numbers!

We are looking for a set of columns of \( A \) that, added together, make up the all-1’s vector.
This is a reduction between two special cases of ILP: one with many equations but only \(0 - 1\) coefficients, and the other with a single equation but arbitrary integer coefficients.

The reduction is based on: \(0 - 1\) vectors can encode numbers!

We are looking for a set of columns of \(A\) that, added together, make up the all-1’s vector.
But if we think of the columns as binary integers (read from top to bottom), we are looking for a subset of the integers corresponding to the columns of \(A\) that add up to the binary integer \(11\ldots 1\).
This is a reduction between two special cases of ILP: one with many equations but only 0–1 coefficients, and the other with a single equation but arbitrary integer coefficients.

The reduction is based on: 0–1 vectors can encode numbers!

We are looking for a set of columns of $A$ that, added together, make up the all-1’s vector.
But if we think of the columns as binary integers (read from top to bottom), we are looking for a subset of the integers corresponding to the columns of $A$ that add up to the binary integer $11\ldots1$.

And this is an instance of Subset Sum. The reduction seems complete!
Except for one detail: carry. Because of carry, say 5-bit binary integers can add up to 11111 = 31, for example, 5 + 6 + 20 = 31 or, in binary, 00101 + 00110 + 10100 = 11111, even when the sum of the corresponding vectors is not (1, 1, 1, 1, 1).

But this is easy to fix: Think of the column vectors not as integers in base 2, but as integers in base $n + 1$, one more than the number of columns. This way, since at most $n$ integers are added, and all their digits are 0 and 1, there can be no carry, and our reduction works.
Except for one detail: carry.
Except for one detail: *carry*.

Because of carry, say 5-bit binary integers can add up to $11111 = 31$, for example, $5 + 6 + 20 = 31$ or, in binary,

$$00101 + 00110 + 10100 = 11111,$$

even when the sum of the corresponding vectors is not $(1, 1, 1, 1, 1)$. 
Except for one detail: carry.

Because of carry, say 5-bit binary integers can add up to \( 11111 = 31 \), for example, \( 5 + 6 + 20 = 31 \) or, in binary,

\[
00101 + 00110 + 10100 = 11111,
\]

even when the sum of the corresponding vectors is not \((1, 1, 1, 1, 1)\).

But this is easy to fix: Think of the column vectors not as integers in base 2, but as integers in base \( n + 1 \), one more than the number of columns.
Except for one detail: carry.

Because of carry, say 5-bit binary integers can add up to $11111 = 31$, for example, $5 + 6 + 20 = 31$ or, in binary,

$$00101 + 00110 + 10100 = 11111,$$

even when the sum of the corresponding vectors is not $(1, 1, 1, 1, 1)$.

But this is easy to fix: Think of the column vectors not as integers in base 2, but as integers in base $n + 1$, one more than the number of columns. This way, since at most $n$ integers are added, and all their digits are 0 and 1, there can be no carry, and our reduction works.
Special cases

3Sat is a special case of Sat, or, Sat is a generalization of 3Sat. By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from 3Sat to Sat, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged. In other words, functions \( f \) and \( h \) from the reduction diagram are both the identity.

It is a very useful and common way of establishing that a problem is \( \text{NP} \)-complete: Simply notice that it is a generalization of a known \( \text{NP} \)-complete problem. For example, the Set Cover problem is \( \text{NP} \)-complete because it is a generalization of Vertex Cover.
Special cases

$3\text{Sat}$ is a special case of $\text{Sat}$, or, $\text{Sat}$ is a generalization of $3\text{Sat}$. 
3Sat is a special case of Sat, or, Sat is a generalization of 3Sat.

By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).
**Special cases**

$3\text{Sat}$ is a special case of $\text{Sat}$, or, $\text{Sat}$ is a generalization of $3\text{Sat}$.

By special case we mean that the instances of $3\text{Sat}$ are a subset of the instances of $\text{Sat}$ (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from $3\text{Sat}$ to $\text{Sat}$, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged.
3Sat is a special case of Sat, or, Sat is a generalization of 3Sat.

By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from 3Sat to Sat, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged.
In other words, functions $f$ and $h$ from the reduction diagram are both the identity.
3Sat is a special case of Sat, or, Sat is a generalization of 3Sat.

By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from 3Sat to Sat, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged. In other words, functions $f$ and $h$ from the reduction diagram are both the identity.

It is a very useful and common way of establishing that a problem is NP-complete:
3Sat is a special case of Sat, or, Sat is a generalization of 3Sat.

By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from 3Sat to Sat, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged. In other words, functions $f$ and $h$ from the reduction diagram are both the identity.

It is a very useful and common way of establishing that a problem is NP-complete: Simply notice that it is a generalization of a known NP-complete problem.
3Sat is a special case of Sat, or, Sat is a generalization of 3Sat.

By special case we mean that the instances of 3Sat are a subset of the instances of Sat (in particular, the ones with no long clauses), and the definition of solution is the same in both problems (an assignment satisfying all clauses).

Consequently, there is a reduction from 3Sat to Sat, in which the input undergoes no transformation, and the solution to the target instance is also kept unchanged. In other words, functions \( f \) and \( h \) from the reduction diagram are both the identity.

It is a very useful and common way of establishing that a problem is \textbf{NP}-complete: Simply notice that it is a generalization of a known \textbf{NP}-complete problem. For example, the \textsc{Set Cover} problem is \textbf{NP}-complete because it is a generalization of \textsc{Vertex Cover}.
ZOE $\rightarrow$ ILP

In ILP we are looking for an integer vector $x$ that satisfies $Ax \leq b$, for given matrix $A$ and vector $b$.

To write an instance of ZOE in this precise form, we need to rewrite each equation of the ZOE instance as two inequalities, and to add for each variable $x_i$ the inequalities $x_i \leq 1$ and $-x_i \leq 0$. 
In ILP we are looking for an integer vector $x$ that satisfies $Ax \leq b$, for given matrix $A$ and vector $b$. 
In ILP we are looking for an integer vector $x$ that satisfies $Ax \leq b$, for given matrix $A$ and vector $b$.

To write an instance of ZOE in this precise form, we need to rewrite each equation of the ZOE instance as two inequalities, and to add for each variable $x_i$ the inequalities $x_i \leq 1$ and $-x_i \leq 0$. 

ZOE $\rightarrow$ ILP
ZOE $\rightarrow$ Rudrata Cycle

In Rudrata Cycle we seek a cycle in a graph that visits every vertex exactly once. We shall prove it \textit{NP}-complete in two stages:

1. First we will reduce ZOE to a generalization of Rudrata Cycle, called Rudrata Cycle with Paired Edges.
2. Then we shall see how to get rid of the extra features of that problem and reduce it to the plain Rudrata Cycle.
In Rudrata Cycle we seek a cycle in a graph that visits every vertex exactly once.
**In Rudrata Cycle** we seek a cycle in a graph that visits every vertex exactly once.

We shall prove it **NP**-complete in two stages:
In **Rudrata Cycle** we seek a cycle in a graph that visits every vertex exactly once.

We shall prove it **NP-complete** in two stages:

1. First we will reduce **ZOE** to a generalization of **Rudrata Cycle**, called **Rudrata Cycle with Paired Edges**.
In Rudrata Cycle we seek a cycle in a graph that visits every vertex exactly once.

We shall prove it NP-complete in two stages:

1. First we will reduce ZOE to a generalization of Rudrata Cycle, called Rudrata Cycle with Paired Edges.
2. Then we shall see how to get rid of the extra features of that problem and reduce it to the plain Rudrata Cycle.
We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that
1. visits all vertices once,
2. for every pair of edges $(e, e')$ in $C$, traverses either edge $e$ or edge $e'$ exactly one of them.

Notice that we allow two or more parallel edges between two nodes, since now the different copies of an edge can be paired with other copies of edges in ways that do make a difference.
We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges.
Rudrata Cycle with Paired Edges

We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that

1. visits all vertices once,
2. for every pair of edges $(e, e')$ in $C$, traverses either edge $e$ or edge $e'$ exactly one of them.

Notice that we allow two or more parallel edges between two nodes, since now the different copies of an edge can be paired with other copies of edges in ways that do make a difference.
We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that

1. visits all vertices once,
We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that

1. visits all vertices once,

2. for every pair of edges $(e, e')$ in $C$, traverses either edge $e$ or edge $e'$ exactly one of them.
We are given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. We seek a cycle that

1. visits all vertices once,
2. for every pair of edges $(e, e')$ in $C$, traverses either edge $e$ or edge $e'$ exactly one of them.

Notice that we allow two or more parallel edges between two nodes, since now the different copies of an edge can be paired with other copies of edges in ways that do make a difference.
Given an instance of ZOE, \( Ax = 1 \) (where \( A \) is an \( m \times n \) matrix with 0−1 entries, and thus describes \( m \) equations in \( n \) variables), the graph we construct has the very simple structure:

▶ A cycle that connects \( m + n \) collections of parallel edges.
▶ For each variable \( x_i \) we have two parallel edges (corresponding to \( x_i = 1 \) and \( x_i = 0 \)).
▶ For each equation \( x_{j1} + \cdots + x_{jk} = 1 \) involving \( k \) variables we have \( k \) parallel edges, one for every variable appearing in the equation.

Any Rudrata cycle must traverse the \( m + n \) collections of parallel edges one by one, choosing one edge from each collection. This way, the cycle "chooses" for each variable a value 0 or 1 and, for each equation, a variable appearing in it.
Given an instance of ZOE, $A x = 1$ (where $A$ is an $m \times n$ matrix with 0–1 entries, and thus describes $m$ equations in $n$ variables),
Given an instance of ZOE, $Ax = 1$ (where $A$ is an $m \times n$ matrix with 0–1 entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:
Given an instance of ZOE, $A \mathbf{x} = \mathbf{1}$ (where $A$ is an $m \times n$ matrix with 0–1 entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:

- A cycle that connects $m + n$ collections of parallel edges.
ZOE $\rightarrow$ Rudrata Cycle with Paired Edges

Given an instance of ZOE, $\mathbf{Ax} = \mathbf{1}$ (where $\mathbf{A}$ is an $m \times n$ matrix with 0–1 entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:

- A cycle that connects $m + n$ collections of parallel edges.
- For each variable $x_i$ we have two parallel edges (corresponding to $x_i = 1$ and $x_i = 0$).
Given an instance of ZOE, $Ax = 1$ (where $A$ is an $m \times n$ matrix with 0−1 entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:

- A cycle that connects $m + n$ collections of parallel edges.
- For each variable $x_i$ we have two parallel edges (corresponding to $x_i = 1$ and $x_i = 0$).
- for each equation $x_{j_1} + \cdots + x_{j_k} = 1$ involving $k$ variables we have $k$ parallel edges, one for every variable appearing in the equation.
Given an instance of ZOE, $Ax = 1$ (where $A$ is an $m \times n$ matrix with 0–1 entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:

- A cycle that connects $m + n$ collections of parallel edges.
- For each variable $x_i$ we have two parallel edges (corresponding to $x_i = 1$ and $x_i = 0$).
- For each equation $x_{j_1} + \cdots + x_{j_k} = 1$ involving $k$ variables we have $k$ parallel edges, one for every variable appearing in the equation.

Any Rudrata cycle must traverse the $m + n$ collections of parallel edges one by one, choosing one edge from each collection.
Given an instance of ZOE, $\mathbf{Ax} = 1$ (where $\mathbf{A}$ is an $m \times n$ matrix with $0-1$ entries, and thus describes $m$ equations in $n$ variables), the graph we construct has the very simple structure:

- A cycle that connects $m + n$ collections of parallel edges.
- For each variable $x_i$ we have two parallel edges (corresponding to $x_i = 1$ and $x_i = 0$).
- for each equation $x_{j_1} + \cdots + x_{j_k} = 1$ involving $k$ variables we have $k$ parallel edges, one for every variable appearing in the equation.

Any Rudrata cycle must traverse the $m + n$ collections of parallel edges one by one, choosing one edge from each collection. This way, the cycle “chooses” for each variable a value 0 or 1 and, for each equation, a variable appearing in it.
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere, and there is one place left: the set $C$ of pairs of edges such that exactly one edge in each pair is traversed. For every equation (recall there are $m$ in total), and for every variable $x_i$ appearing in it, we add to $C$ the pair $(e, e')$ where $e$ is the edge corresponding to the appearance of $x_i$ in that particular equation, and $e'$ is the edge corresponding to the variable assignment $x_i = 0$. This completes the construction.
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere,
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere, and there is one place left:
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere, and there is one place left: the set $C$ of pairs of edges such that exactly one edge in each pair is traversed.
The structure of the matrix $\mathbf{A}$ (and not just its dimensions) must be reflected somewhere, and there is one place left: the set $C$ of pairs of edges such that exactly one edge in each pair is traversed.

For every equation (recall there are $m$ in total), and for every variable $x_i$ appearing in it,
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere, and there is one place left: the set $C$ of pairs of edges such that exactly one edge in each pair is traversed.

For every equation (recall there are $m$ in total), and for every variable $x_i$ appearing in it, we add to $C$ the pair $(e, e')$ where $e$ is the edge corresponding to the appearance of $x_i$ in that particular equation, and $e'$ is the edge corresponding to the variable assignment $x_i = 0$. This completes the construction.
The structure of the matrix $A$ (and not just its dimensions) must be reflected somewhere, and there is one place left: the set $C$ of pairs of edges such that exactly one edge in each pair is traversed.

For every equation (recall there are $m$ in total), and for every variable $x_i$ appearing in it, we add to $C$ the pair $(e, e')$ where $e$ is the edge corresponding to the appearance of $x_i$ in that particular equation, and $e'$ is the edge corresponding to the variable assignment $x_i = 0$.

This completes the construction.
Take any solution of this instance of Rudrata Cycle with Paired Edges. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of ZOE. If a variable $x_i$ has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with $x_i$ on the equation side must be traversed, since they are paired in $C$ with the $x_i = 0$ edge. So, in each equation exactly one of the variables appearing in it has value 1, i.e., all equations are satisfied. The other direction is straightforward as well: from a solution to the instance of ZOE one easily obtains an appropriate Rudrata cycle.
Take any solution of this instance of Rudrata Cycle with Paired Edges.
Take any solution of this instance of Rudrata Cycle with Paired Edges. It picks a value for each variable and a variable for every equation.
Take any solution of this instance of **Rudrata Cycle with Paired Edges**. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of ZOE.
Take any solution of this instance of Rudrata Cycle with Paired Edges. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of ZOE.

If a variable $x_i$ has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with $x_i$ on the equation side must be traversed,
Take any solution of this instance of **Rudrata Cycle with Paired Edges**. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of ZOE.

If a variable $x_i$ has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with $x_i$ on the equation side must be traversed, since they are paired in $C$ with the $x_i = 0$ edge.
Take any solution of this instance of **Rudrata Cycle with Paired Edges**. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of **ZOE**.

If a variable $x_i$ has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with $x_i$ on the equation side must be traversed, since they are paired in $C$ with the $x_i = 0$ edge.

So, in each equation exactly one of the variables appearing in it has value 1, i.e., all equations are satisfied.
Take any solution of this instance of **Rudrata Cycle with Paired Edges**. It picks a value for each variable and a variable for every equation. We claim that the values thus chosen are a solution to the original instance of **ZOE**.

If a variable $x_i$ has value 1, then the edge $x_i = 0$ is not traversed, and thus all edges associated with $x_i$ on the equation side must be traversed, since they are paired in $C$ with the $x_i = 0$ edge.

So, in each equation exactly one of the variables appearing in it has value 1, i.e., all equations are satisfied.

The other direction is straightforward as well: from a solution to the instance of **ZOE** one easily obtains an appropriate **Rudrata** cycle.
Getting rid of the edge pairs

Consider the graph shown in Figure 8.12, and suppose that it is a part of a larger graph $G$ in such a way that only the four endpoints $a$, $b$, $c$, $d$ touch the rest of the graph.

We claim that this graph has the following important property: in any Rudrata cycle of $G$ the subgraph shown must be traversed in one of the two ways shown in bold in Figure 8.12(b) and (c).

This gadget behaves just like two edges $\{a, b\}$ and $\{c, d\}$ that are paired up in the Rudrate Cycle with Paired Edges.
Getting rid of the edge pairs

Consider the graph shown in Figure 8.12, and suppose that it is a part of a larger graph $G$ in such a way that only the four endpoints $a, b, c, d$ touch the rest of the graph.
Getting rid of the edge pairs

Consider the graph shown in Figure 8.12, and suppose that it is a part of a larger graph \( G \) in such a way that only the four endpoints \( a, b, c, d \) touch the rest of the graph.

We claim that this graph has the following important property:

*in any Rudrata cycle of \( G \) the subgraph shown must be traversed in one of the two ways shown in bold in Figure 8.12(b) and (c).*
Consider the graph shown in Figure 8.12, and suppose that it is a part of a larger graph $G$ in such a way that only the four endpoints $a, b, c, d$ touch the rest of the graph.

We claim that this graph has the following important property:

*in any Rudrata cycle of $G$ the subgraph shown must be traversed in one of the two ways shown in bold in Figure 8.12(b) and (c).*

This gadget behaves just like two edges $\{a, b\}$ and $\{c, d\}$ that are paired up in the Rudrate Cycle with Paired Edges.
We go through the pairs in $C$ one by one. To get rid of each pair $\{a, b\}, \{c, d\}$ we replace the two edges with the gadget in Figure 8.12(a). For any other pair in $C$ that involves $\{a, b\}$, we replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where $f$ is from the gadget: the traversal of $\{a, f\}$ is now an indication that edge $\{a, b\}$ in the old graph would be traversed. Similarly, $\{c, h\}$ replaces $\{c, d\}$. After $|C|$ such replacements (performed in polynomial time, since each replacement adds only 12 vertices to the graph) we are done. The Rudrata cycles in the resulting graph will be in one-to-one correspondence with the Rudrata cycles in the original graph that conform to the constraints in $C$. 
We go through the pairs in $C$ one by one.
Rudrata Cycle with Paired Edges $\rightarrow$ Rudrata Cycle

We go through the pairs in $C$ one by one. To get rid of each pair $(\{a, b\}, \{c, d\})$ we replace the two edges with the gadget in Figure 8.12(a).
We go through the pairs in $C$ one by one. To get rid of each pair $(\{a, b\}, \{c, d\})$ we replace the two edges with the gadget in Figure 8.12(a).

For any other pair in $C$ that involves $\{a, b\}$, we replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where $f$ is from the gadget:
We go through the pairs in $C$ one by one. To get rid of each pair $(\{a, b\}, \{c, d\})$ we replace the two edges with the gadget in Figure 8.12(a).

For any other pair in $C$ that involves $\{a, b\}$, we replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where $f$ is from the gadget: the traversal of $\{a, f\}$ is from now on an indication that edge $\{a, b\}$ in the old graph would be traversed.
We go through the pairs in $C$ one by one. To get rid of each pair $(\{a, b\}, \{c, d\})$ we replace the two edges with the gadget in Figure 8.12(a).

For any other pair in $C$ that involves $\{a, b\}$, we replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where $f$ is from the gadget: the traversal of $\{a, f\}$ is from now on an indication that edge $\{a, b\}$ in the old graph would be traversed.

Similarly, $\{c, h\}$ replaces $\{c, d\}$.

---

**Rudrata Cycle with Paired Edges $\rightarrow$ Rudrata Cycle**
We go through the pairs in \( C \) one by one. To get rid of each pair \( (\{a, b\}, \{c, d\}) \) we replace the two edges with the gadget in Figure 8.12(a).

For any other pair in \( C \) that involves \( \{a, b\} \), we replace the edge \( \{a, b\} \) with the new edge \( \{a, f\} \), where \( f \) is from the gadget: the traversal of \( \{a, f\} \) is from now on an indication that edge \( \{a, b\} \) in the old graph would be traversed.

Similarly, \( \{c, h\} \) replaces \( \{c, d\} \).

After \(|C|\) such replacements (performed in polynomial time, since each replacement adds only 12 vertices to the graph) we are done.
We go through the pairs in $C$ one by one. To get rid of each pair $\{a, b\}, \{c, d\}$ we replace the two edges with the gadget in Figure 8.12(a).

For any other pair in $C$ that involves $\{a, b\}$, we replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where $f$ is from the gadget: the traversal of $\{a, f\}$ is from now on an indication that edge $\{a, b\}$ in the old graph would be traversed.

Similarly, $\{c, h\}$ replaces $\{c, d\}$.

After $|C|$ such replacements (performed in polynomial time, since each replacement adds only 12 vertices to the graph) we are done.

The Rudrata cycles in the resulting graph will be in one-to-one correspondence with the Rudrata cycles in the original graph that conform to the constraints in $C$. 
Rudrata Cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is $1$ if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance.

Conversely, if $G$ has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$ (it must use at least one edge of length $1 + \alpha$, and the total length of all $n-1$ others is at least $n-1$).

Thus Rudrata Cycle reduces to TSP.
Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance. Conversely, if $G$ has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$ (it must use at least one edge of length $1 + \alpha$, and the total length of all $n - 1$ others is at least $n - 1$).

Thus Rudrata Cycle reduces to TSP.
Rudrata Cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
Rudrata Cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u,v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.
**Rudrata Cycle $\rightarrow$ TSP**

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance.
Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance.

Conversely, if $G$ has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$. 

Thus Rudrata Cycle reduces to TSP.
**Rudrata Cycle $\rightarrow$ TSP**

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance.

Conversely, if $G$ has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$ (it must use at least one edge of length $1 + \alpha$, and the total length of all $n - 1$ others is at least $n - 1$).
Rudrata Cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the following instance of the TSP:

- The set of cities is the same as $V$.
- The distance between cities $u$ and $v$ is 1 if $\{u, v\}$ is an edge of $G$ and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP instance is equal to the number of nodes, $|V|$.

If $G$ has a Rudrata cycle, then the same cycle is also a tour within the budget of the TSP instance.

Conversely, if $G$ has no Rudrata cycle, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$ (it must use at least one edge of length $1 + \alpha$, and the total length of all $n - 1$ others is at least $n - 1$).

Thus Rudrata Cycle reduces to TSP.
If $\alpha = 1$, then all distances are either 1 or 2, and so this instance of the TSP satisfies the triangle inequality: if $i, j, k$ are cities, then $d_{ij} + d_{jk} \geq d_{ik}$.

This is a special case of the TSP which is of practical importance and which, as we shall see in Chapter 9, is in a certain sense easier, because it can be efficiently approximated.
If $\alpha = 1$, then all distances are either 1 or 2,
If $\alpha = 1$, then all distances are either 1 or 2, and so this instance of the TSP satisfies the triangle inequality:
If \( \alpha = 1 \), then all distances are either 1 or 2, and so this instance of the TSP satisfies the triangle inequality: if \( i, j, k \) are cities, then
\[
d_{ij} + d_{jk} \geq d_{ik}.
\]
If $\alpha = 1$, then all distances are either 1 or 2, and so this instance of the TSP satisfies the triangle inequality: if $i, j, k$ are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}.$$ 

This is a special case of the TSP which is of practical importance and which, as we shall see in Chapter 9, is in a certain sense easier, because it can be efficiently approximated.
If $\alpha$ is large, then the resulting instance of the TSP may not satisfy the triangle inequality, but has another important property:

- Either it has a solution of cost $n$ or less,
- Or all its solutions have cost at least $n + \alpha$ (which now can be arbitrarily larger than $n$).

As we shall see in Chapter 9, this important gap property implies that, unless $P = NP$, no approximation algorithm is possible.
If \( \alpha \) is large, then the resulting instance of the TSP may not satisfy the triangle inequality, but has another important property:
If $\alpha$ is large, then the resulting instance of the TSP may not satisfy the triangle inequality, but has another important property:

- either it has a solution of cost $n$ or less,
- or all its solutions have cost at least $n + \alpha$ (which now can be arbitrarily larger than $n$).
If $\alpha$ is large, then the resulting instance of the TSP may not satisfy the triangle inequality, but has another important property:

- either it has a solution of cost $n$ or less,
- or all its solutions have cost at least $n + \alpha$ (which now can be arbitrarily larger than $n$).

As we shall see in Chapter 9, this important gap property implies that, unless $P = NP$, no approximation algorithm is possible.
Any problem in \textbf{NP} \rightarrow \textbf{Sat}

First, we shall show that all problems in \textbf{NP} can be reduced to a generalization of \textbf{Sat}, i.e., \textbf{Circuit Sat}.

In \textbf{Circuit Sat} we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:

1. \textbf{AND} gates and \textbf{OR} gates have indegree 2.
2. \textbf{NOT} gates have indegree 1.
3. Known input gates have no incoming edges and are labeled \textit{false} or \textit{true}.
4. Unknown input gates have no incoming edges and are labeled "?".

One of the sinks of the dag is designated as the output gate.
ANY PROBLEM IN $\textbf{NP} \rightarrow \textbf{Sat}$

First, we shall show that all problems in $\textbf{NP}$ can be reduced to a generalization of $\textbf{Sat}$, i.e., $\textbf{Circuit Sat}$. 
First, we shall show that all problems in \textbf{NP} can be reduced to a generalization of \textbf{Sat}, i.e., \textbf{Circuit Sat}.

In \textbf{Circuit Sat} we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:
First, we shall show that all problems in \textbf{NP} can be reduced to a generalization of \textbf{Sat}, i.e., \textbf{Circuit Sat}.

In \textbf{Circuit Sat} we are given a (Boolean) circuit, a dag whose vertices are \textbf{gates} of five different types:

1. AND gates and OR gates have indegree 2.

   1. AND gates and OR gates have indegree 2.
First, we shall show that all problems in \textbf{NP} can be reduced to a generalization of \textbf{Sat}, i.e., \textbf{Circuit Sat}.

In \textbf{Circuit Sat} we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:

1. AND gates and OR gates have indegree 2.
2. NOT gates have indegree 1.
First, we shall show that all problems in $\text{NP}$ can be reduced to a generalization of $\text{Sat}$, i.e., $\text{Circuit Sat}$.

In $\text{Circuit Sat}$ we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:

1. AND gates and OR gates have indegree 2.
2. NOT gates have indegree 1.
3. Known input gates have no incoming edges and are labeled false or true.
First, we shall show that all problems in \( \textbf{NP} \) can be reduced to a generalization of \( \textbf{Sat} \), i.e., \( \textbf{Circuit Sat} \).

In \( \textbf{Circuit Sat} \) we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:

1. AND gates and OR gates have indegree 2.
2. NOT gates have indegree 1.
3. Known input gates have no incoming edges and are labeled false or true.
4. Unknown input gates have no incoming edges and are labeled “?”.
First, we shall show that all problems in \( \text{NP} \) can be reduced to a generalization of \( \text{Sat} \), i.e., \( \text{Circuit Sat} \).

In \( \text{Circuit Sat} \) we are given a (Boolean) circuit, a dag whose vertices are gates of five different types:

1. AND gates and OR gates have indegree 2.
2. NOT gates have indegree 1.
3. Known input gates have no incoming edges and are labeled false or true.
4. Unknown input gates have no incoming edges and are labeled “?”.

One of the sinks of the dag is designated as the output gate.
Circuit Sat

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic, until we obtain the value at the output gate. This is the value of the circuit for the particular assignment to the inputs.

Circuit SAT: Given a circuit, find a truth assignment for the unknown inputs such that the output gate evaluates to true, or report that no such assignment exists.
Circuit Sat

Given an assignment of values to the unknown inputs,
Circuit Sat

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic,
Circuit Sat

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic, until we obtain the value at the output gate.
Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic, until we obtain the value at the output gate.

This is the value of the circuit for the particular assignment to the inputs.
Circuit Sat

Given an assignment of values to the unknown inputs, we can evaluate the gates of the circuit in topological order, using the rules of Boolean logic, until we obtain the value at the output gate.

This is the value of the circuit for the particular assignment to the inputs.

Circuit SAT: Given a circuit, find a truth assignment for the unknown inputs such that the output gate evaluates to true, or report that no such assignment exists.
Circuit Sat (cont’d)

Circuit Sat is a generalization of Sat. Sat asks for a satisfying truth assignment for a circuit that has a simple structure:

▶ A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.

▶ Each clause is the OR of its literals.

▶ Each literal is either an unknown input gate or the NOT of one.

▶ There are no known input gates.
Circuit Sat (cont’d)

Circuit Sat is a generalization of Sat.
Circuit Sat (cont’d)

Circuit Sat is a generalization of Sat.

Sat asks for a satisfying truth assignment for a circuit that has a simple structure:

- A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.
- Each clause is the OR of its literals.
- Each literal is either an unknown input gate or the NOT of one.
- There are no known input gates.
Circuit Sat is a generalization of Sat.

Sat asks for a satisfying truth assignment for a circuit that has a simple structure:

- A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.
Circuit Sat is a generalization of Sat.

Sat asks for a satisfying truth assignment for a circuit that has a simple structure:

- A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.
- Each clause is the OR of its literals.
Circuit Sat is a generalization of Sat.

Sat asks for a satisfying truth assignment for a circuit that has a simple structure:

- A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.
- Each clause is the OR of its literals.
- Each literal is either an unknown input gate or the NOT of one.
**Circuit Sat** (cont’d)

**Circuit Sat** is a generalization of **Sat**.

**Sat** asks for a satisfying truth assignment for a circuit that has a simple structure:

- A bunch of AND gates at the top join the clauses, and the result of this big AND is the output.
- Each clause is the OR of its literals.
- Each literal is either an unknown input gate or the NOT of one.
- There are no known input gates.
Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g = \text{true}$: $(g)$. 
- $g = \text{false}$: $(\overline{g})$.
- $g = h_1 \lor h_2$: $(g \lor \overline{h_1}) (g \lor \overline{h_2}) (\overline{g} \lor h_1 \lor h_2)$.
- $g = h_1 \land h_2$: $(\overline{g} \lor h_1) (\overline{g} \lor h_2) (g \lor \overline{h_1} \lor \overline{h_2})$.
- $g = \text{NOT} h$: $(g \lor h) (\overline{g} \lor \overline{h})$. 
For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:
Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g = \text{true}$: $(g)$.
- $g = \text{false}$: $(\overline{g})$.
- $g = h_1 \lor h_2$: $(g \lor \overline{h_1})(g \lor \overline{h_2})(\overline{g} \lor h_1 \lor h_2)$.
- $g = h_1 \land h_2$: $(\overline{g} \lor h_1)(\overline{g} \lor h_2)(g \lor \overline{h_1} \lor \overline{h_2})$.
- $g = \text{NOT} h$: $(g \lor h)(\overline{g} \lor \overline{h})$. 

Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g = \text{true}$: $(g)$.
- $g = \text{false}$: $(\bar{g})$.
Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g = \text{true}$: $(g)$.
- $g = \text{false}$: $(\overline{g})$.
- $g = h_1 \text{ OR } h_2$: $(g \lor \overline{h_1})(g \lor \overline{h_2})(\overline{g} \lor h_1 \lor h_2)$. 

Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g =$ true: $(g)$.
- $g =$ false: $(\bar{g})$.
- $g =$ $h_1$ OR $h_2$: $(g \lor \bar{h}_1)(g \lor \bar{h}_2)(\bar{g} \lor h_1 \lor h_2)$.
- $g =$ $h_1$ AND $h_2$: $(\bar{g} \lor h_1)(\bar{g} \lor h_2)(g \lor \bar{h}_1 \lor \bar{h}_2)$. 
Circuit Sat $\rightarrow$ Sat

For each gate $g$ we create a variable $g$, and we model the effect of the gate using a few clauses:

- $g = \text{true}$: $(g)$.
- $g = \text{false}$: $(\bar{g})$.
- $g = h_1 \text{ OR } h_2$: $(g \lor h_1)(g \lor h_2)(\bar{g} \lor h_1 \lor h_2)$.
- $g = h_1 \text{ AND } h_2$: $(\bar{g} \lor h_1)(\bar{g} \lor h_2)(g \lor \bar{h}_1 \lor \bar{h}_2)$.
- $g = \text{NOT } h$: $(g \lor h)(\bar{g} \lor \bar{h})$. 
ANY PROBLEM IN NP $\rightarrow$ CIRCUIT SAT (informal)
Let $A$ be an $\text{NP}$-problem.
Let $A$ be an $\mathbf{NP}$-problem. We must discover a reduction from $A$ to Circuit Sat.
Let $A$ be an $\textbf{NP}$-problem. We must discover a reduction from $A$ to $\textsc{Circuit Sat}$.

This sounds very difficult, because we know almost nothing about $A$!
Let $A$ be an $\text{NP}$-problem. We must discover a reduction from $A$ to $\text{Circuit Sat}$.

This sounds very difficult, because we know almost nothing about $A$!

Recall: there is an algorithm $C$ that checks, given an instance $I$ and a proposed solution $S$, whether or not $S$ is a solution of $I$. 

Let $A$ be an $\textbf{NP}$-problem. We must discover a reduction from $A$ to $\text{Circuit Sat}$.

_This sounds very difficult, because we know almost nothing about $A$!_

Recall: there is an algorithm $C$ that checks, given an instance $I$ and a proposed solution $S$, whether or not $S$ is a solution of $I$.

Moreover, $C$ runs time polynomial in the length of $I$ (we can assume that $S$ is itself encoded as a binary string, and we know that the length of this string is polynomial in the length of $I$).
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm. For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs. If the polynomial algorithm in question solves a problem that requires a yes or no answer, as is the situation with $C$: Does $S$ encode a solution to the instance encoded by $I$? then this answer is given at the output gate.
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm.
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, \textit{whose input gates encode the input to the algorithm}.

For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs,
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm.

For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs.
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm.

For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs.

If the polynomial algorithm in question solves a problem that requires a yes or no answer,
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm.

For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs.

If the polynomial algorithm in question solves a problem that requires a yes or no answer, as is the situation with $C$:

\[
\text{Does } S \text{ encode a solution to the instance encoded by } I? 
\]
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm. For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs.

If the polynomial algorithm in question solves a problem that requires a yes or no answer, as is the situation with $C$:

$$\text{Does } S \text{ encode a solution to the instance encoded by } I?$$

then this answer is given at the output gate.
Recall now our argument in Section 7.7 that any polynomial algorithm can be rendered as a circuit, whose input gates encode the input to the algorithm.

For any input length (number of input bits) the circuit will be scaled to the appropriate number of inputs, but the total number of gates of the circuit will be polynomial in the number of inputs.

If the polynomial algorithm in question solves a problem that requires a yes or no answer, as is the situation with $C$:

$$\text{Does } S \text{ encode a solution to the instance encoded by } I?$$

then this answer is given at the output gate.
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the known inputs are the bits of $I$,
- the unknown inputs are the bits of $S$,
- the output is true if and only if the unknown inputs spell a solution $S$ of $I$.

In other words, the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance $I$ of $A$.

The reduction is complete.
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the known inputs are the bits of $I$,
- the unknown inputs are the bits of $S$,
- the output is true if and only if the unknown inputs spell a solution $S$ of $I$.

In other words, the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance $I$ of $A$.

The reduction is complete.
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the known inputs are the bits of $I$,
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the known inputs are the bits of $I$,
- the unknown inputs are the bits of $S$, 

The reduction is complete.
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the **known** inputs are the bits of $I$,
- the **unknown** inputs are the bits of $S$,
- the output is true if and only if the unknown inputs spell a solution $S$ of $I$.
We conclude that, given any instance \( I \) of problem \( A \), we can construct in polynomial time a circuit:

- the known inputs are the bits of \( I \),
- the unknown inputs are the bits of \( S \),
- the output is true if and only if the unknown inputs spell a solution \( S \) of \( I \).

In other words, the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance \( I \) of \( A \).
We conclude that, given any instance $I$ of problem $A$, we can construct in polynomial time a circuit:

- the known inputs are the bits of $I$,
- the unknown inputs are the bits of $S$,
- the output is true if and only if the unknown inputs spell a solution $S$ of $I$.

In other words, the satisfying truth assignments to the unknown inputs of the circuit are in one-to-one correspondence with the solutions of instance $I$ of $A$. The reduction is complete.