CSE331 Introduction to Algorithms
Lecture 17
Introduction to Computational Complexity

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

June 14, 2017

Introduction

- This is an introductory lecture to computational complexity.
- The goal is to classify computational problems as "easy" or "difficult".
- I will introduce two complexity classes, **P** and **NP**, and the notion of **NP**-hardness.
- The presentation will not be very formal.
- A more rigorous treatment is given in:
  - CSE332 Theory of Computation
  - CSE530 Algorithm & Complexity
- **Reference**: Chapter 34 of the textbook (p. 1048)

  Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.
- I will not be following the textbook closely in this lecture.

Languages

Definition

A *binary string* is a finite sequence of 0s and 1s. We denote by $\{0,1\}^*$ the set of all binary strings. The *length* $|x|$ of a binary string $x = x_1 x_2 \dots x_n$ is the number n of bits in x.

- For instance, 0, 1, 01, 10, 11, 00111010 are binary strings.
- $|0| = 1$ and $|0110| = 4$.
- The empty string λ is also a string, with length $|\lambda| = 0$.

Languages

Definition

A *language* is a set of strings. In other words, L is a language whenever $L \subseteq \{0,1\}^*$.

Example

A *palyndrome* is a string $x_1 x_2 \dots x_n$ such that $x_1 x_2 \dots x_n = x_n x_{n-1} \dots x_1$. The palyndromes form a language.

Definition

We say that an algorithm *decides* a language L if, for every input string $x \in L$, it returns 1, and for every input string $x \notin L$, it returns 0.
We say that it decides L in time $T(n)$ if, for every input x of size $|x| = n$, it runs in at most $T(n)$ time.

Languages

- Example: The algorithm below decides the set of palyndromes.

Pseudocode

1: **procedure** PALYNDROME($x = x_1 x_2 \dots x_n$)
2: **for** $i \leftarrow 1, \lfloor n/2 \rfloor$ **do**
3: **if** $x_i \neq x_{n-i+1}$ **then**
4: **return** 0
5: **return** 1

- This algorithm decides the set of palyndromes in time $O(n)$.

The Class **P**

- We introduce our first *complexity class* **P**, where P stands for *polynomial-time*.

Definition

A language L is in **P** if there exists an algorithm that decides L in time $O(n^c)$, for some constant c.

- Example: The set of palyndromes is in **P**, as it can be decided in $O(n^1)$ time.
- In this definition, the class **P** only applies to deciding languages.
- In the following, we show how it is related to more general computing problems.

Decision Problems

- A *decision problem* is a problem whose answer is a *Boolean* TRUE or FALSE, or equivalently 1 or 0.
- Example of a decision problem:

Problem (DECIDELCS)

Given two input binary sequences A and B and an integer k, the problem of deciding whether the length of their longest common subsequence (LCS) is at least k is called DECIDELCS.

- A *positive instance* of a decision problem is an input for which the answer is 1.
- So a positive instance of DECIDELCS is a triple A, B, k such that the length of LCS(A, B) is at least k.

Decision Problems

- The input to DecideLCS can be represented as a string.
- Example: $A = 001101$, $B = 0101$, $k = 3$.

$$\text{Encoding: } \underbrace{000001010001}_{A}\,11\,\underbrace{00010001}_{B}\,11\,\underbrace{0101}_{k}$$

- We encoded each bit of A, B, and k with 00 or 01, and we use 11 as a separator between the representations of A, B and k.
- So DecideLCS can also be viewed as a language. A string is in this language if the input that it encodes is a positive instance of DecideLCS.
- The algorithm presented in Lecture 12 allows to solve DecideLCS in $O(n^2)$ time. Therefore DecideLCS $\in \mathbf{P}$

Decision Problems

- More generally, for all computing problems we encountered in CSE331, the input can be encoded in a binary string.
- Reason: This is what is done internally by the computer.
- So every decision problem can be seen as the problem of deciding the language containing the encodings of its positive instances.
- Therefore, whenever we deal with a decision problem, we can ask whether it is in **P** or not.
- If it is in **P**, then intuitively, the problem is "easy", and we say that it is *tractable*.
- This can be misleading because a $\Theta(n^{20})$ algorithm is too slow even for small inputs.
- But in most cases, we either get small polynomial running times such as $O(n^3)$, or the best known algorithm is exponential.

Optimization Problems

- The problem of computing an LCS is not a decision problem, because the output is a sequence, not just 0 or 1.
- Computing an LCS is an optimization problem:

Definition

Let f be a function defined over a domain $\mathcal{D}$. The problem of finding $x^* \in \mathcal{D}$ such that $f(x^*)$ is minimum is called a *minimization problem*. The problem of finding $x^* \in \mathcal{D}$ such that $f(x^*)$ is maximum is called a *maximization problem*. An *optimization problem* is a minimization or a maximization problem. The solution x^* is called an *optimal solution*.

Optimization Problems

- Every optimization problem can be associated with a decision problem where the goal is to decide whether the optimal value $f(x^*)$ is more or less than some input value.
- For instance, the problem DecideLCS is a decision problem associated with the problem of computing the length of an LCS.
- We cannot say that LCSLength $\in$ **P** because the output is not a Boolean, but an integer.
- So we will say that it is *polynomial-time solvable* (or *tractable*.)
- Other optimization problems we encountered in CSE331 are also polynomial-time solvable: Closest Pair, Optimal binary search tree

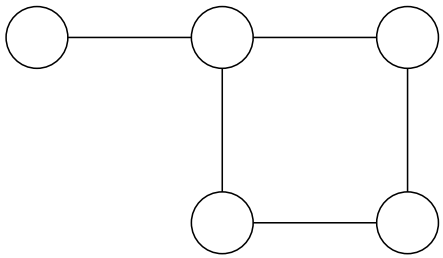
Other Problems

- Some problems are neither decision problems nor optimization problems. For instance, sorting, or matrix multiplication.
- In these problems, we want to compute $f(x)$, where x and $f(x)$ are strings representing the input and the output, and the size of the input is $|x| = n$.
- Such a problem is also said to be *polynomial-time solvable* or *tractable* if an algorithm can solve it in polynomial time $O(n^c)$.

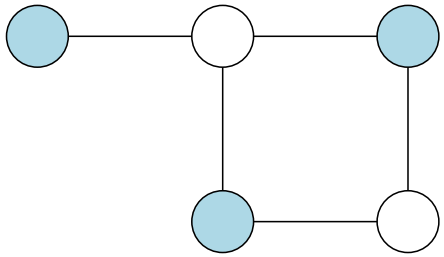
Vertex Cover

Definition (Vertex cover)

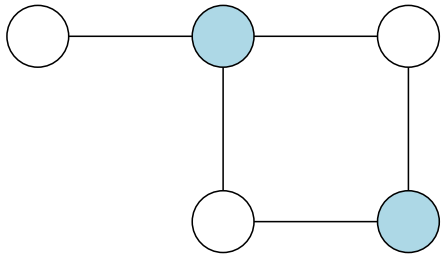
Given a graph $G(V, E)$ with vertex set V and edge set E, a *vertex cover* is a subset $V' \subseteq V$ of vertices such that each edge $e \in E$ is incident to at least one vertex in V'.



input graph



a vertex cover



minimum vertex cover

Problem (MinVertexCover)

The minimum vertex cover problem is to find a vertex cover of smallest cardinality.

3-SAT

- We are given r *Boolean* variables $z_1, \ldots, z_r$.
- So the value of any z_i is either 0 (false) or 1 (true).
- The *negation* of z_i is $\neg z_i = 1 - z_i$.
- A *clause* is a disjunction of terms in $z_1, \ldots, z_r, \neg z_1, \ldots, \neg z_r$.
  - Example: $z_1 \vee \neg z_3 \vee \neg z_4$, which means z_1 or not z_3 or not z_4.
- In this lecture, we will only consider clauses involving three variables.
  - Example: $z_2 \vee \neg z_3 \vee z_4$
  - But not $z_1 \vee \neg z_2 \vee z_3 \vee z_4$, and not $z_2 \vee z_4$
- A *truth assignment* is an assignment of value 0 or 1 to each z_i.

3-SAT

Problem (3-SAT)

Given a collection $C_1, \ldots, C_k$ of clauses with 3 variables each, decide whether there is a truth assignment that satisfies all the clauses.

Example:

- $C_1 = z_1 \vee z_2 \vee \neg z_3$
- $C_2 = z_1 \vee \neg z_3 \vee z_4$
- $C_3 = z_1 \vee \neg z_2 \vee \neg z_3$
- $C_4 = \neg z_2 \vee \neg z_3 \vee \neg z_4$
- $C_5 = z_2 \vee \neg z_3 \vee \neg z_4$

- With the truth assignment $z_1 = 0$, $z_2 = 1$, $z_3 = 1$, $z_4 = 1$, clauses C_1, C_2, and C_5 are satisfied.
- With the truth assignment $z_1 = z_2 = 1$, $z_3 = z_4 = 0$, all clauses are satisfied.
- So this is a positive instance of 3-SAT.

Reductions

- We can compare the complexity of two problems using the following relation.

Definition (Reduction)

A language $L \subset \{0,1\}^*$ is *polynomial-time reducible* to a language $L' \in \{0,1\}^*$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $\forall x \in \{0,1\}^*$, $x \in L \Leftrightarrow f(x) \in L'$.

- In this case, we say that L *reduces to* L', and we write $L \leqslant_p L'$.
- We can solve the problem L (i.e. decide the language L) as follows.
- First transform the instance x of L into an instance $f(x)$ of L' in polynomial time.
- Then solve the instance $f(x)$ of L'.

Reductions

- Intuitively, $L \leqslant_p L'$ means that L is not much harder than L'.
- So if L' is tractable, then L is tractable as well.
- Or, said differently, L is not harder than L' if we are willing to ignore polynomial factors in the running time.

Reductions

Proposition

If $L \leqslant_p L'$ and $L' \in \mathbf{P}$, then $L \in \mathbf{P}$.

Proof.

Suppose that $L \leqslant_p L'$ and $L' \in \mathbf{P}$. As $L' \in \mathbf{P}$, there exists a constant c_1 and a decision algorithm A running in $O(|x'|^{c_1})$ time such that $A(x') = 1$ iff $x' \in L'$. As $L \leqslant_p L'$, there exists a constant c_2 and a function f computable in $O(|x|^{c_2})$ time such that $x \in L$ iff $f(x) \in L'$.
Therefore, we have $x \in L$ iff $A(f(x)) = 1$.
As $f(x)$ can be computed in time $O(|x|^{c_2})$, the string $f(x)$ has length $O(|x|^{c_2})$. So $A(f(x))$ can be computed in time $O(|x|^{c_2} + (|x|^{c_2})^{c_1})$, which is polynomial in the input size $|x| = n$. It means that we can decide whether $x \in L$ in polynomial time by computing $A(f(x))$. □

Reductions

Proposition

If $L \leqslant_p L'$ and $L' \leqslant_p L''$, then $L \leqslant_p L''$.

Proof.

There exist polynomial-time computable functions f_1 and f_2 such that:

- $x \in L$ iff $f_1(x) \in L'$, and
- $y \in L'$ iff $f_2(y) \in L''$.

Therefore, $x \in L$ iff $f_2(f_1(x)) \in L''$. As f_1 and f_2 are polynomial-time computable, $f_2(f_1(x))$ can be computed in polynomial time. □

Reduction from 3-SAT to VertexCover

- We now show that 3-SAT reduces to VertexCover.
- More precisely, let DecideVertexCover be the decision problem associated with VertexCover. So given a graph G and an integer s, it consists in deciding whether G has a vertex cover of size s.

Theorem

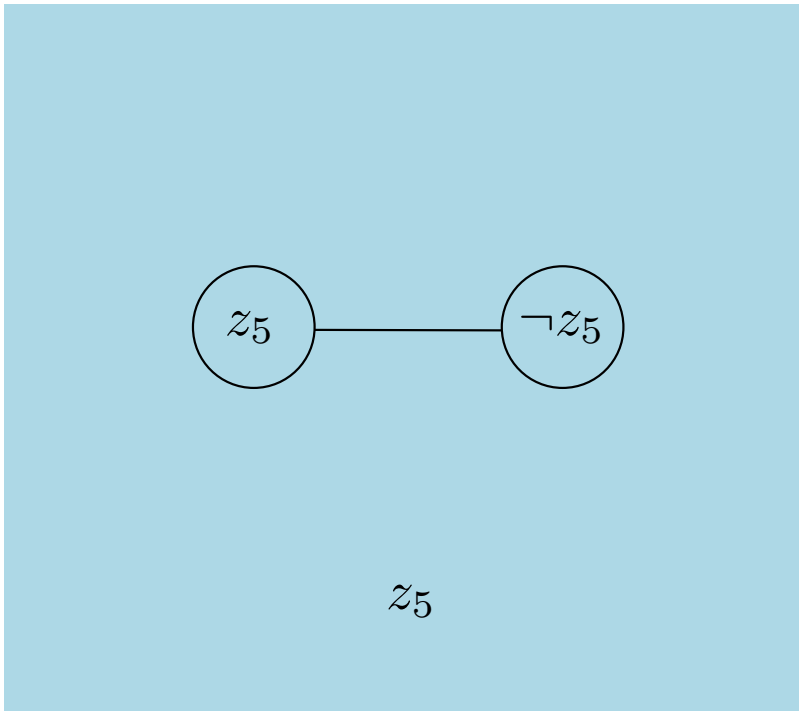
$$\text{3-SAT} \leqslant_p \text{DecideVertexCover}.$$

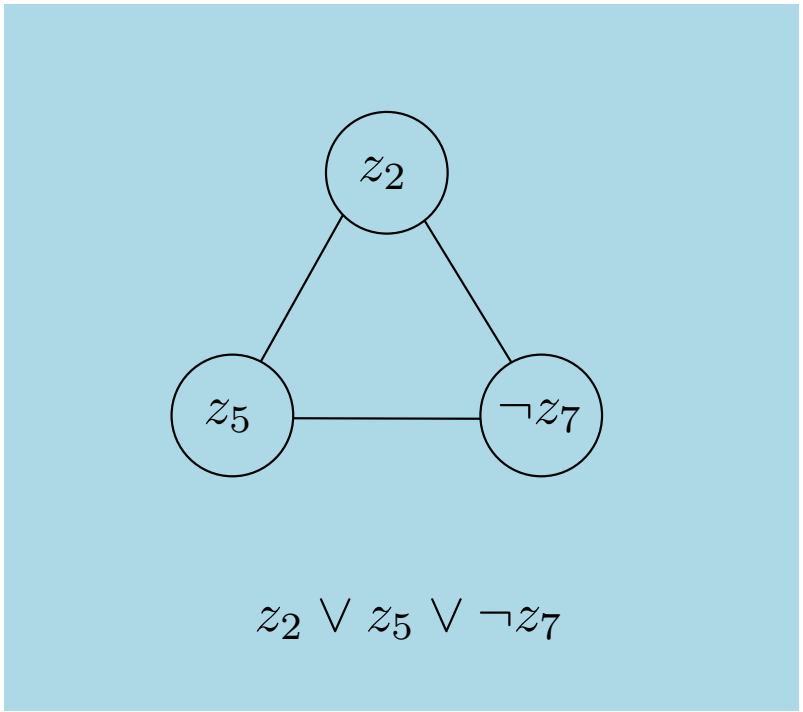
- We now prove this theorem.
- So given an instance of 3-SAT, we want to construct an instance of DecideVertexCover that is positive iff the original instance of 3-SAT is positive.

Reduction from 3-SAT to VertexCover

For each variable z_i we construct a *variable gadget* which is a graph formed by two nodes labeled z_i and $\neg z_i$, connected by an edge.

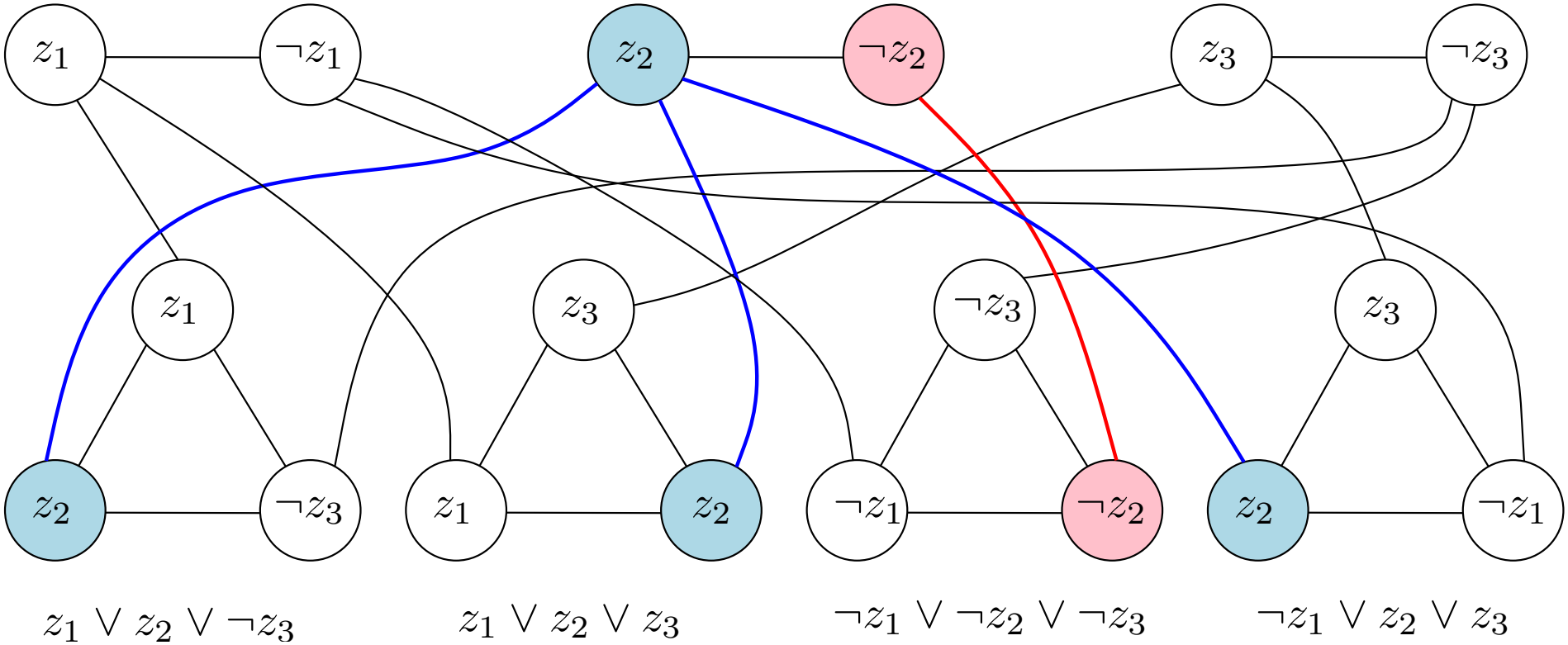
For each clause C_j we construct a *clause gadget* which is a triangle whose nodes are labeled by the literals of C_j.





Reduction from 3-SAT to VertexCover

- We connect each node of each variable gadget to all the nodes in clause gadgets that have the same label.



Reduction from 3-SAT to VertexCover

- Given an instance of 3-SAT with k clauses and r variables, we have constructed an instance G of vertex cover with $3k + 2r$ vertices.

Lemma

If the 3-SAT instance is satisfiable, then there is a vertex cover of size $2k + r$.

Proof:

- Suppose that the 3-SAT instance is satisfiable, and consider one assignment that satisfies it.
- Then for each i, if $z_i = 1$, we cover vertex z_i in the corresponding variable gadget, and otherwise we cover $\neg z_i$.
- So the edge in each variable gadget is covered.

Reduction from 3-SAT to VertexCover

- Let $a \vee b \vee c$ be a clause in our 3-SAT instance. Then a, b or c must be true in our variable assignment. Wlog, assume it is a. Then we cover b and c.

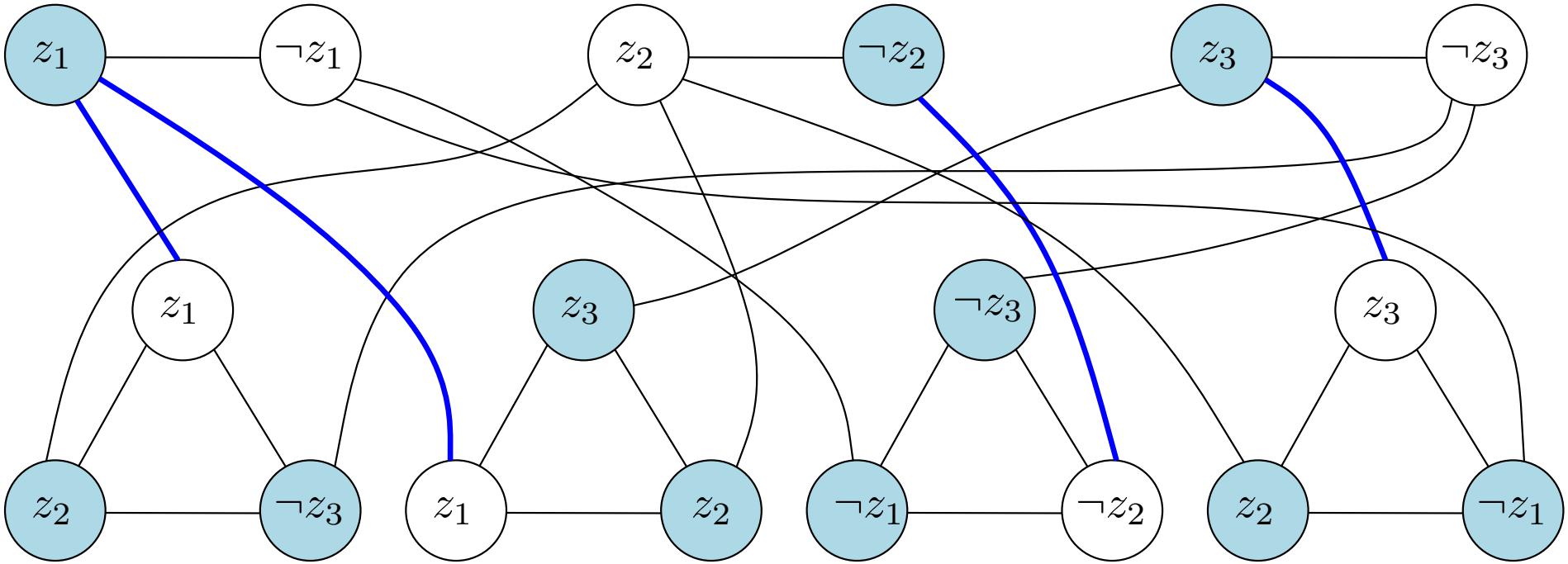


Figure: Vertex cover corresponding to the truth assignment $(z_1, z_2, z_3) = (1, 0, 1)$

Reduction from 3-SAT to VertexCover

- Edges (a, b), (a, c) and (b, c) are covered as b and c are covered.
- The edge connecting a to the corresponding variable clause is also covered, because the node representing a in the variable gadget is covered.
- The edges connecting b and c to variable gadgets are covered because b and c are covered.
- Therefore, all edges are covered.
- As we picked 1 vertex per variable gadget and 2 per clause gadget, our vertex cover has size $2k + r$. □

Reduction from 3-SAT to VertexCover

Lemma

Every vertex cover of G contains at least one node of each variable gadget and 2 nodes of each clause gadget.

Proof.

The edge in each variable gadget can only be covered by one (or both) of its nodes. So the vertex cover need to include at least one node from each variable gadget.
For each clause gadget, if two of the nodes are not covered, then the edge between them is not covered. So we need to cover at least two of its vertices. □

Reduction from 3-SAT to VertexCover

Lemma

If the there is a vertex cover of size $2k + r$, then the 3-SAT instance is satisfiable.

Proof.

By the lemma on Slide 27, exactly one literal in each variable gadget must be in the cover. We use the corresponding variable assignment.
By the same lemma, exactly two literals of each clause gadget must be in the cover. Then the third literal must be connected to a vertex of a variable gadget that is in the cover. So the clause is satisfied. □

Reduction from 3-SAT to VertexCover

- To summarize, our instance of 3-SAT is satisfiable iff the graph G that we constructed has a vertex cover of size $2k + r$.
- So given an instance of 3-SAT, we have constructed in polynomial time an instance of vertex cover that is equivalent.
- It completes the proof that $3\text{-SAT} \leqslant_p \text{DecideVertexCover}$.
- If we have an efficient algorithm for vertex cover, it means that given an instance of 3-SAT, we can solve it efficiently by constructing the graph G, and then running the vertex cover algorithm on it.
- Problem: No efficient algorithm for 3-SAT is known, and it seems currently out of reach. (See later in this lecture.)
- So our reduction shows that Vertex Cover is hard.
- It will often be the case: We use reduction to prove *hardness* result.

The Complexity Class **NP**

Definition

A language $L \subseteq \{0,1\}^*$ is in the class **NP** if there is a polynomial-time algorithm A that takes two strings as input, and a polynomial p, such that

$$x \in L \Leftrightarrow \exists y \in \{0,1\}^* : |y| \leqslant p(|x|) \text{ and } A(x,y) = 1.$$

- We say that y is a *certificate* for x and that A *verifies* L in polynomial time.

Example

3-SAT$\in$ **NP**

(Proof on next slide.)

The Complexity Class **NP**

- The following algorithm A verifies 3-SAT in polynomial time.

An algorithm A that verifies 3-SAT

1. Let I be the 3-SAT instance encoded by x, and let r be its number of variables.
2. If $|y| \neq r$, return 0.
3. If the truth assignment $(z_1, z_2, \ldots, z_r) \leftarrow (y_1, y_2, \ldots, y_r)$ satisfies all the clauses in I, then return 1. Otherwise, return 0.

- This algorithm simply takes as input an instance of 3-SAT and a truth assignment, and it checks whether this truth assignment is a solution.

The Complexity Class **NP**

- Intuitively, a problem is in **NP** if a solution can be *verified* in polynomial time.
- Difference with **P**: For a problem to be in **P**, you need to be able to *find* a solution in polynomial time, not just check it.
- For many problems in **NP**, we do not know how to find a solution in polynomial time.
- The 'N' in **NP** stands for *nondeterministic*. I will not explain why in CSE331.

The Complexity Class **NP**: More Examples

- DecideVertexCover is in **NP**.
- Constructing a timetable (for instance for final exams) is in **NP**—more precisely, the decision problem associated with it.
- Any problem in **P** is also in **NP**. (See next slide.)

The Complexity Class **NP**

Theorem

$$\mathbf{P} \subseteq \mathbf{NP}$$

Proof.

Let L be a language in **P**. We will prove that $L \in \mathbf{NP}$.
As $L \in$**P**, there is a polynomial-time algorithm B such that

$$B(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$$

Let A be the algorithm taking two strings x, y as input, and that just runs B on x, ignoring y. So A runs polynomial time, and if $x \in L$, then $A(x, \lambda) = 1$, so $\exists y : A(x, y) = 1$. On the other hand, if $x \notin L$, then $A(x, y) = 0$ for every string y. □

The Complexity Class **NP**

- The converse is not known: We do not know whether $\mathbf{NP} \subseteq \mathbf{P}$.
- It is the most important open problem in theoretical computer science:

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}.$$

- Intuitively, the question is whether *finding* a solution is harder than *verifying* a solution.
- In other words: Can creativity be automated?
- The Clay Mathematics Institute offers a $1,000,000 prize for a correct solution:

P vs NP Problem

- Most experts conjecture that $\mathbf{P} \neq \mathbf{NP}$, but currently a proof seems to be out of reach.

NP-Completeness

Definition (NP-hardness and NP-completeness)

We say that a language $L' \subseteq \{0,1\}^*$ is **NP**-*hard* if $L \leqslant_p L'$ for any $L \in \mathbf{NP}$. We say that L' is **NP**-*complete* if L' is **NP**-hard and $L' \in \mathbf{NP}$.

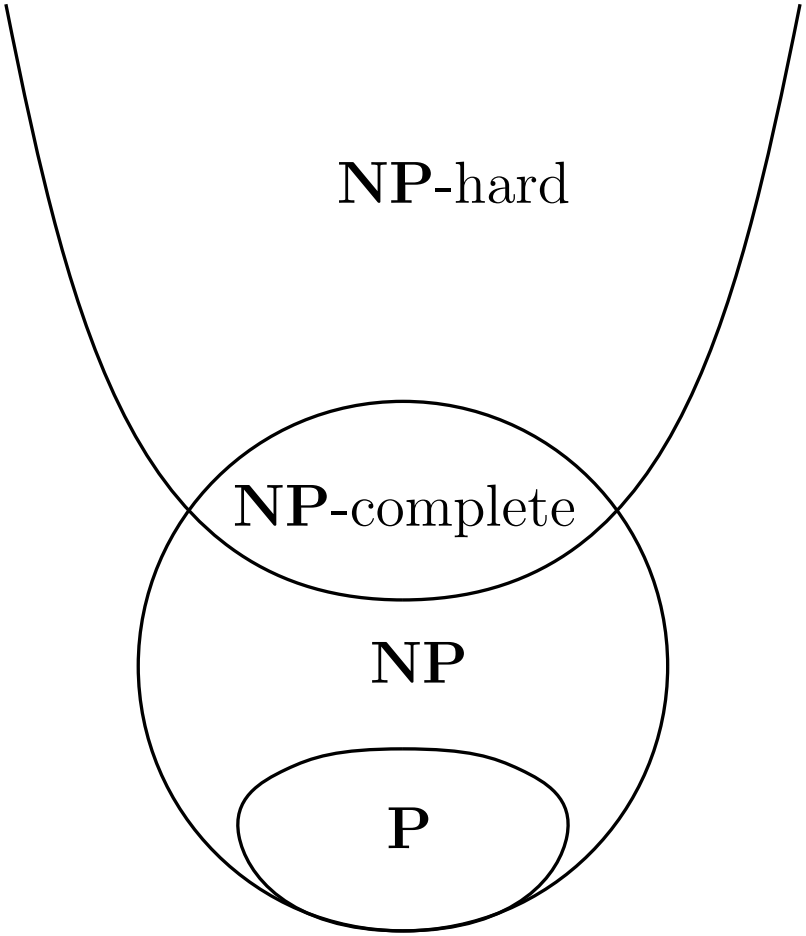
- Intuition: **NP**-complete languages are the hardest languages in **NP**.

Theorem

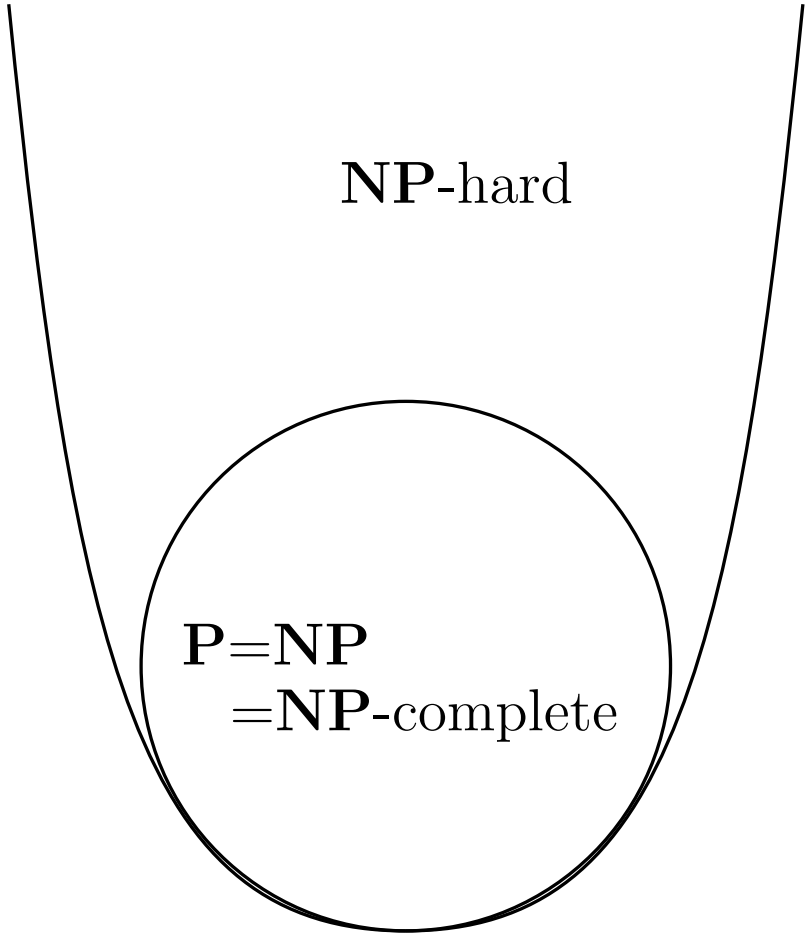
1. *If language L is* **NP**-*hard and $L \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.*
2. *If language L is* **NP**-*complete, then $L \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$.*

- Proof: Follows from the Proposition on Slide 19.

NP-Completeness



If $\mathbf{P} \neq \mathbf{NP}$



If $\mathbf{P} = \mathbf{NP}$

NP-Completeness

Theorem (Cook)

3-SAT *is* **NP**-*complete.*

- Not proved in CSE331. (Requires Turing machines.)
- As we have proved that 3-SAT reduces to vertex cover, and it is easy to see that DecideVertexCover is in **NP**, it follows that:

Corollary

DecideVertexCover *is* **NP**-*complete.*

- It is the usual way of proving that a problem is **NP**-complete: Prove that a known **NP**-complete problem reduces to this problem.
- 3-SAT is often used for this purpose.

NP-Completeness

- We cannot say that MinVertexCover is **NP**-complete, because it is not a decision problem, and hence it is not in **NP**.
- But MinVertexCover is **NP**-hard, because if we could construct an optimal vertex cover in polynomial time, then we could solve 3-SAT in polynomial time.
- Similarly, many optimization problems are **NP**-hard. For instance, finding an optimal timetable of final exams, finding a truth assignment that satisfies the largest number of clauses ...
- Many combinatorial optimization problems are **NP**-hard.
- It means that we do not know how to solve them in polynomial time, as otherwise it would prove that $\mathbf{P} = \mathbf{NP}$.

NP-Completeness

- In practice, most problems can easily be classified as either being in **P**, or **NP**-hard.
- So when we do not find a simple polynomial-time algorithm, it is likely that the problem is **NP**-hard.
  - Do not forget to try dynamic programming, because it is one of the few techniques that allows to solve seemingly difficult optimization problems in polynomial time.
- In order to confirm it, there is often a simple reduction from a known **NP**-hard problem.
- A list of **NP**-hard problems can be found in this book:

  Computers and Intractability by Garey and Johnson.
- When a problem is **NP**-hard, we need to resort to approximation algorithms, or heuristics.

Concluding Remarks

- Some problems are known to be intractable:

Problem (Halting)

Given the code of a computer program, and its input, the halting problem is to decide whether the program will finish running, or keep running forever.

- In fact, this problem is *undecidable*: No algorithm can solve it.
- So it is not in **P**, or even in **NP**.
- (Not covered in CSE331.)