CSE515 Advanced Algorithms
Lecture 14
Introduction to Computational Complexity II

Antoine Vigneron

antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

April 15, 2021

1 3-SAT

2 The complexity class **NP**

- **NP**-completeness

3 Conclusion

Introduction

- Reminder: Assignment 2 is due on Friday.
- This is the second part of our 3-lectures introduction to computational 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.

3-SAT

- We are given r *Boolean* variables $z_1, \dots, z_r$.
- 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 $z_1, \dots, z_r, \neg z_1, \dots, \neg z_r$.

Example

The clause $z_1 \vee \neg z_3 \vee \neg z_4$ means z_1 or not z_3 or not z_4.

- In this lecture, we will only consider clauses involving three variables.
  - For instance, $z_2 \vee \neg z_3 \vee z_4$, 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 $C_1, \dots, C_k$ of clauses with 3 variables each, decide whether there is a truth assignment that satisfies all the clauses.

Example

$$
\begin{aligned}
C_1 &= z_1 \lor z_2 \lor \neg z_3 \\
C_2 &= z_1 \lor \neg z_3 \lor z_4 \\
C_3 &= z_1 \lor \neg z_2 \lor \neg z_3 \\
C_4 &= \neg z_2 \lor \neg z_3 \lor \neg z_4 \\
C_5 &= z_2 \lor \neg z_3 \lor \neg z_4
\end{aligned}
$$

- 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.

Reduction from 3-SAT to VERTEX-COVER

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

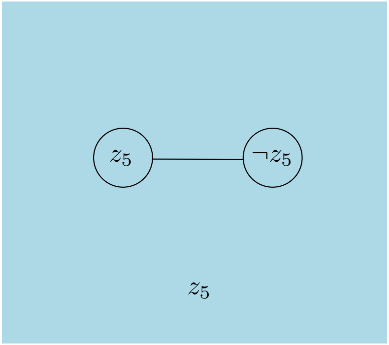
Theorem

$$\text{3-SAT} \leqslant_p \text{DECIDE-VERTEX-COVER}.$$

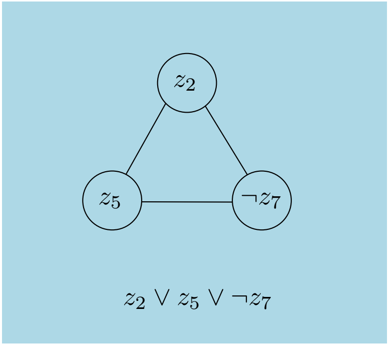
- We now prove this theorem. So given an instance of 3-SAT, we want to construct an instance of DECIDE-VERTEX-COVER that is positive iff the original instance of 3-SAT is positive.

Reduction from 3-SAT to Vertex-cover

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.

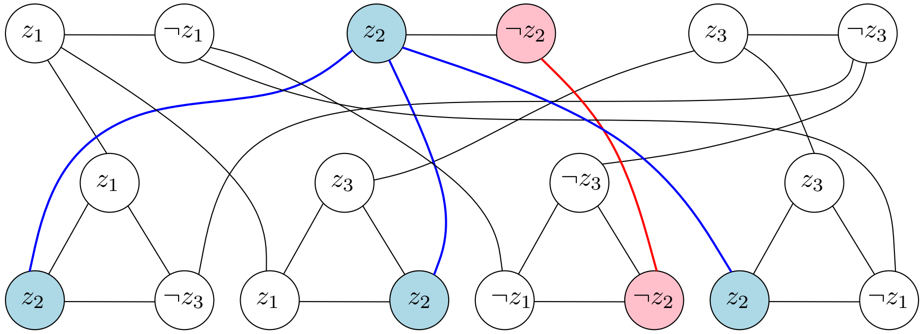


For each clause C_j we construct a *clause gadget* which is a triangle whose nodes are labeled by the literals of C_j.



Reduction from 3-SAT to VERTEX-COVER

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



$z_1 \vee z_2 \vee \neg z_3$ $z_1 \vee z_2 \vee z_3$ $\neg z_1 \vee \neg z_2 \vee \neg z_3$ $\neg z_1 \vee z_2 \vee z_3$

Reduction from 3-SAT to VERTEX-COVER

- 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 Vertex-cover

- 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.

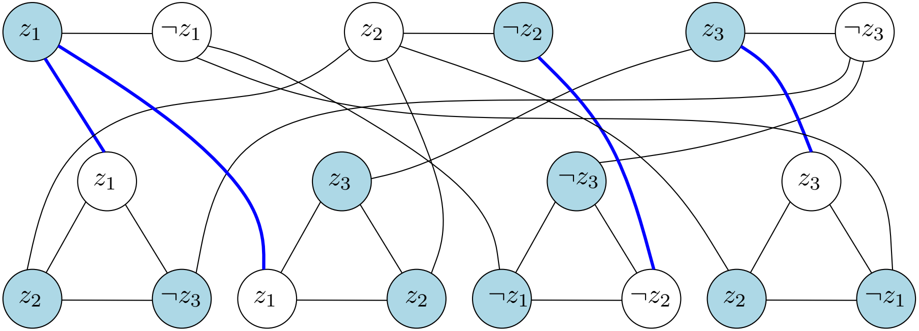


Figure: Vertex cover corresponding to the truth assignment $(z_1, z_2, z_3) = (1, 0, 1)$

Reduction from 3-SAT to VERTEX-COVER

- 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 Vertex-cover

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 VERTEX-COVER

Lemma

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

Proof.

By the lemma on Slide 12, 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. $\square$

Reduction from 3-SAT to VERTEX-COVER

- 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 $\mathrm{3\text{-}SAT} \leqslant_p$ DECIDE-VERTEX-COVER.
- 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 reductions 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 $(z_1, z_2, \dots, z_r) \leftarrow (y_1, y_2, \dots, y_r)$ 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 CSE515.

The Complexity Class **NP**: More Examples

- DECIDE-VERTEX-COVER 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 $L \in \mathbf{P}$, 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. $\square$

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 all $L \in \mathbf{NP}$.
We say that L' is **NP**-*complete* if L' is **NP**-hard and $L' \in \mathbf{NP}$.

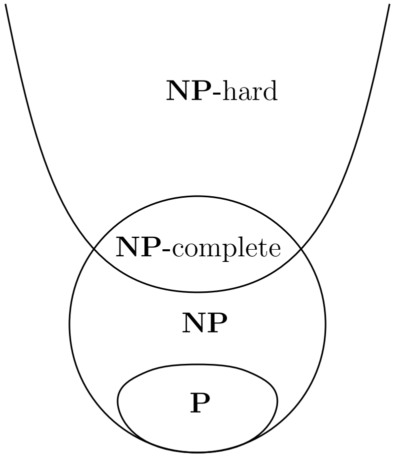
- Intuition: **NP**-complete languages are the hardest languages in **NP**.

Proposition

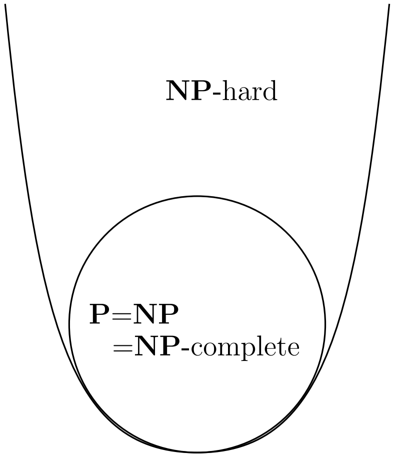
1. *If language L' is* **NP***-hard and $L' \leqslant_p L''$, then L'' is* **NP***-hard.*
2. *If language L' is* **NP***-hard and $L' \in$* **P***, then* **P** $=$ **NP***.*
3. *If language L' is* **NP***-complete, then $L' \in$* **P** *iff* **P** $=$ **NP***.*

- Proof: see exercise set.

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 CSE515. (Requires Turing machines.)
- As we have proved that 3-SAT reduces to VERTEX-COVER, and it is easy to see that DECIDE-VERTEX-COVER is in **NP**, it follows that:

Corollary

DECIDE-VERTEX-COVER *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 MIN-VERTEX-COVER is **NP**-complete, because it is not a decision problem, and hence it is not in **NP**.
- But MIN-VERTEX-COVER 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 and linear programming, because these are among the few techniques that allow 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.
  (not covered in CSE515)
- So it is not in **P**, or even in **NP**.