CSE331 Introduction to Algorithms
Lecture 25
Introduction to Computational Complexity I

Antoine Vigneron

antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021

Introduction

- This lecture, and the next two, form a short introduction 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
- **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 $x = x_1x_2\dots x_n$ 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 $x_1 x_2 \dots x_n = x_n x_{n-1} \dots x_1$. For instance 110011 is a palyndrome. 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 time at most $T(n)$.

Languages

Example

The algorithm below decides the set L of all palyndromes.

Pseudocode

procedure $\textsc{Palyndrome}(x = x_1 x_2 \ldots x_n)$
  for $i \leftarrow 1, \lfloor n/2 \rfloor$ **do**
    if $x_i \neq x_{n-i+1}$ **then**
      return 0
  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 $\text{LCS}(A, B)$ is at least k.

Decision Problems

- The input to DecideLCS can be represented as a string.

Example

$A = 001101$, $B = 0101$, $k = 3$.

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

- 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 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 19 allows to solve DecideLCS in $O(n^2)$ time. Therefore DecideLCS $\in$ P.

Decision Problems

- More generally, for all computing problems we encountered in CSE331, the input can be encoded as a binary string.
- Why? 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.

Example

$\textsc{DecideLCS}$ is a decision problem associated with LCS (the problem of computing an LCS).

- We cannot say that LCS $\in$ **P** because the output is not a Boolean, but a string.
- Similarly, $\textsc{LCSLength}$ (the problem of computing the *length* of an LCS) is not in **P** because the output is an integer.
- We will say that these problems are *polynomial-time solvable* (or *tractable*.)
- Other optimization problems we encountered in CSE331 are also polynomial-time solvable: Closest Pair, Optimal binary search tree.

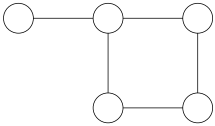
Other Problems

- 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 $O(n^c)$.

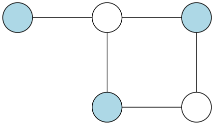
Vertex Cover

Definition (Vertex cover)

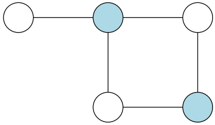
Given a graph $G(V, E)$ with vertex set V and edge set E, a *vertex cover* is a subset $V' \subseteq V$ of vertices such that each edge $e \in E$ is incident to at least one vertex in V'.



input graph



a vertex cover



minimum vertex cover

Vertex Cover

Problem (MIN-VERTEX-COVER)

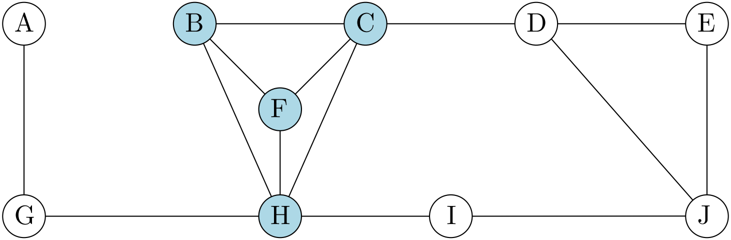
The minimum vertex cover problem is to find a vertex cover of smallest cardinality.

- This is a minimization problem.
- It is associated with the decision problem below:

Problem (VERTEX-COVER)

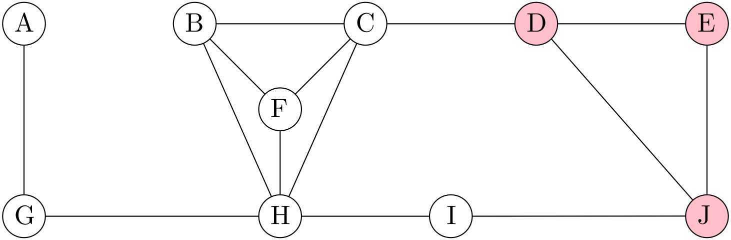
Given a graph G and an integer k, the vertex cover problem is to decide whether G has a vertex cover of size k.

The Clique Problem



- Every pair in B, C, F and E is connected by an edge.
- We say that $\{B, C, F, H\}$ is a *clique* of size 4 in this graph.

The Clique Problem



- $\{D, E, J\}$ is a clique of size 3.
- There are other cliques of size 3. Which ones?
- There is no clique of size 5. Why?

The Clique Problem

Definition

A *clique* in a graph $G(V, E)$ is a subset of vertices $C \subseteq V$ such that every pair of vertices in C is connected by an edge of E. The *size* of C is its cardinality $|C|$.

Problem (MAX-CLIQUE)

The maximum clique problem is to find a clique of maximum size in an input graph.

- Decision problem:

Problem (CLIQUE)

Given an input graph $G(V, E)$ and an integer k, the clique problem is to decide whether G has a clique of size k.

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 $f : \{0,1\}^* \rightarrow \{0,1\}^*$ 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 cannot be 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 $L' \in \mathrm{P}$, then $L \in \mathrm{P}$.

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 $A(f(x))$. $\square$

Reductions

Proposition

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

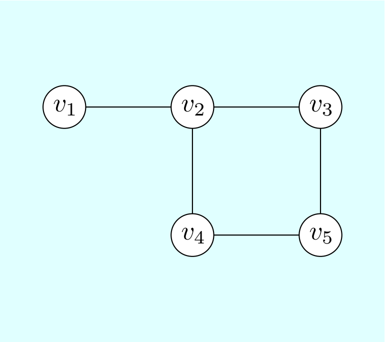
Proof.

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

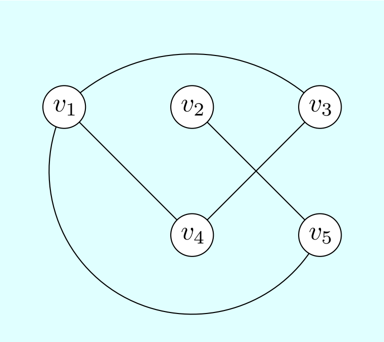
- $x \in L$ iff $f_1(x) \in L'$, and
- $y \in L'$ iff $f_2(y) \in L''$.

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. □

Example

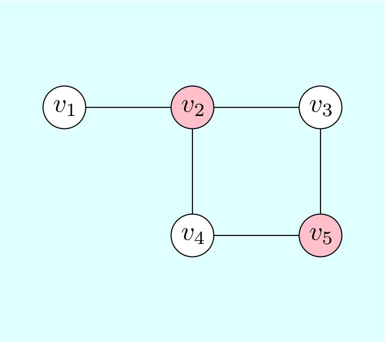


a graph G

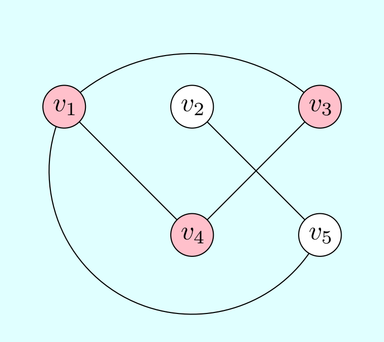


its complement $\bar{G}$

Example



$\{v_2, v_5\}$ is a vertex cover of size 2



$\{v_1, v_3, v_4\}$ is a clique of size 3

Example

Definition

The *complement* of the graph $G(V, E)$ is the graph $\bar{G}(V, \bar{E})$. In other words, an edge is in G iff it is not in $\bar{G}$.

Lemma

C is a vertex cover of G iff its complement $\bar{C} = V \setminus C$ is a clique in $\bar{G}$.

Example

Theorem

VERTEX-COVER $\leqslant_p$ CLIQUE. *In other words,* VERTEX-COVER *reduces to* CLIQUE.

Proof.

We transform an instance G of VERTEX-COVER into its complement $\bar{G} = f(G)$. Then G has a vertex cover of size k iff $\bar{G}$ has a clique of size $n - k$. □

- The reduction also works in the other direction, with the same proof:

Theorem

CLIQUE $\leqslant_p$ VERTEX-COVER.

Example

- It shows that CLIQUE and VERTEX-COVER have roughly the same complexity (i.e. within a polynomial factor).
- As we will see in the next lecture, these problems are hard, in the sense that no polynomial-time algorithm is currently known.