csce750 — Analysis of Algorithms
Fall 2019 — Lecture Notes: NP-Complete Problems

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

CLRS 34
GJ 1–3

The theory of **NP-completeness** can be used cast doubt on the existence of *any* polynomial-time algorithm for a given problem.

- So far, we have concentrated mostly on designing and analyzing efficient algorithms for various problems. Algorithms are evidence of how **easy** those problems are.
- By showing that a problem is NP-complete, we are giving evidence of how **hard** a problem is.

Practically, we can think of an NP-completeness proof as a 'license' to stop looking for an efficient algorithm, and settle for approximation or to consider only special cases.

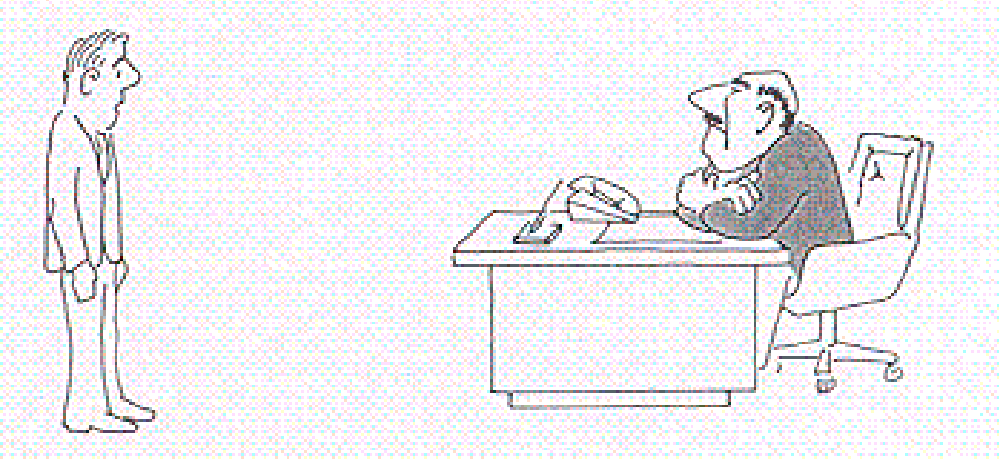
Note: Both CLRS and GJ define things at higher level of precision than we'll examine in the lecture: models of computation, abstract vs. concrete problems, encodings, *etc.*

2 Four classes of problems

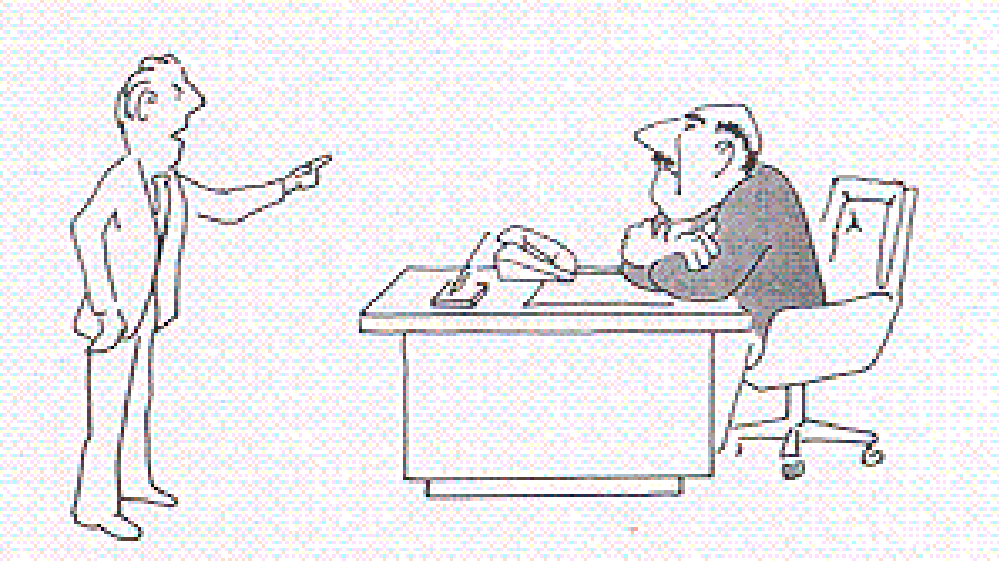
We'll examine four different classes of problems.

- P — problems that can be decided in polynomial time
- NP — problems that can be verified in polynomial time
- NP-hard — problems that can be reduced in polynomial time
- NP-complete — problems in both NP and NP-hard.

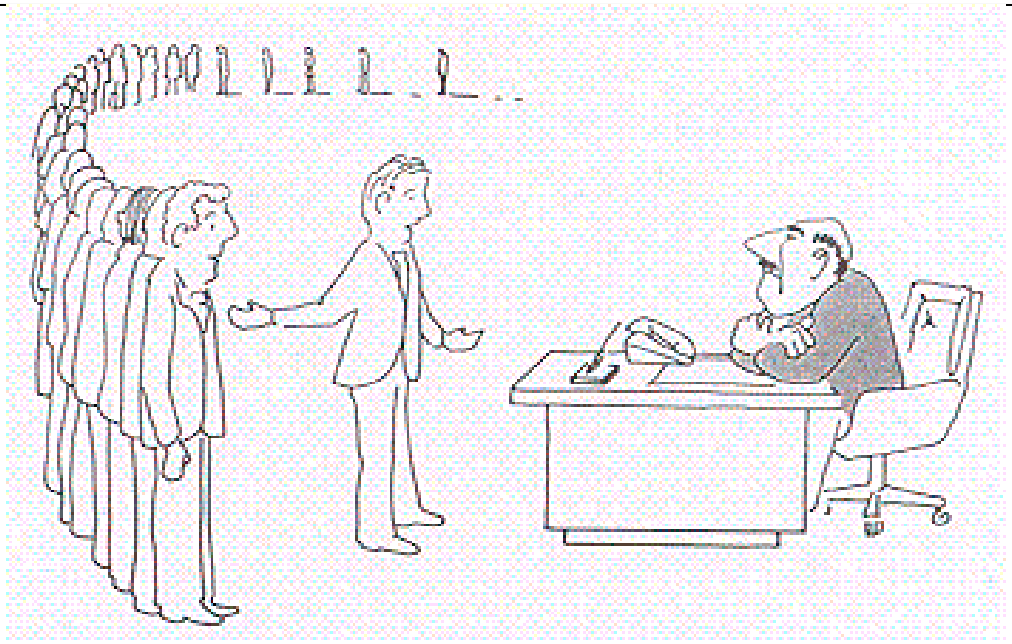
3 Illustration (from Garey and Johnson)



"I can't find an efficient algorithm. I guess I'm just too dumb."



"I can't find an efficient algorithm, because no such algorithm is possible."



"I can't find an efficient algorithm, but neither can any of these famous people."

4 Decision problems

A **decision problem** is a problem in which the correct output for each instance is either 'Yes' or 'No'.

HAM-CYCLE:
Instance: An undirected graph $G = (V, E)$.
Question: Does G contain a cycle that visits every vertex exactly once?

PATH:
Instance: A weighted directed graph G, a pair of vertices $u, v \in E(G)$, and a number k.
Question: Does G contain a path from u to v with total weight at most k?

5 Aside: Optimization problems

We can convert any **optimization problem** ('Find the largest…', 'Find the smallest…', etc) to a decision problem by including a bound on the objective function as part of the input.

6 Problems as languages

We can think of a decision problem as a **formal language**.

- The **input** is a (binary) string s.
- The **output** is either 'Yes' or 'No'.

The **language** of a problem is the set of binary input strings, under a suitable encoding, for which the correct output is 'Yes'.

7 Example language

HAM-CYCLE:
Instance: An undirected graph $G = (V, E)$.
Question: Does G contain a cycle that visits every vertex exactly once?

The language $L_{\text{HAM-CYCLE}}$ is the set of strings describing undirected graphs that have Hamiltonian cycles.

8 Deciding a language

Definition: An algorithm **decides a language** if it correctly determines whether its input string is a member of that language.

Specifically, the algorithm should:

- Terminate for any input instance.
- Return 'yes' if the input instance is in the language.
- Return 'no' if the input instance is not in the language.

9 P

Intuition: P is the set of all problems that can be decided in polynomial time.

Definition: P is the set of all languages L for which there exists an algorithm A and a constant c, such that

- A decides L, and
- the worst-case run time of A is $O(n^c)$.

10 How to prove: P

To prove that your problem L is in P:

- Describe an algorithm whose input is an instance of L.
- Show that your algorithm decides L.
  - If the input is a Yes instance, must return Yes.
  - If the input is a No instance, must return No.
- Show that the worst-case run time of your algorithm is bounded by some polynomial.

11 Verification algorithms

A **verification algorithm** accepts two inputs:

- An ordinary string x (the 'real' input)
- A **certificate** y (a 'proof' that the correct answer for x is Yes).

A verification algorithm produces one of two outputs:

- 'Yes', or
- 'No'.

The **language verified** by a verification algorithm is

$$L = \{x \mid \text{there exists } y \text{ for which } A(x, y) \text{ outputs Yes.}\}$$

12 Verification example: HAM-CYCLE

We can form a verification algorithm for HAM-CYCLE that accepts two inputs:

- An undirected graph $G = (V, E)$.
- An ordered list of vertices $(v_1, \ldots, v_m)$.

The algorithm would:

- Output 'Yes' if:
  - the sequence $(v_1, \ldots, v_m)$ contains all of the vertices of V, with no duplicates, and
  - E contains an edge (v_i, v_{i+1}) for each $i = 1, \ldots, m-1$.
  - E contains an edge (v_m, v_1).
- Output 'No' otherwise.

13 Verification example: PATH

We can form a verification algorithm for PATH that accepts two inputs:

- A weighted directed graph G, a pair of vertices $u, v \in E(G)$, and a number k.
- An ordered list of vertices $(v_1, \ldots, v_m)$.

The algorithm would:

- Output 'Yes' if:
  - $v_1 = u$,
  - $v_m = v$,
  - E contains an edge (v_i, v_{i+1}) for each $i = 1, \ldots, m$, and
  - the total weight of all these edges is at most k.
- Output 'No' otherwise.

14 NP

Intuition: NP is the set of all problems for which 'Yes' answers can be verified in polynomial time.

Definition: NP is the set of all languages L for which there exists a verification algorithm A and a constant c, such that

- A verifies L, and
- the worst-case run time of A is $O(n^c)$.

~~"non-polynomial time"~~ "nondeterministic polynomial time"

15 How to prove: NP

To prove that your problem L is in NP:

- Decide what to use as the certificate. That is, for each yes instance, explain what a certificate should be.
- Describe an algorithm whose input is an instance of L and a certificate.
- Show that your algorithm verifies L.
  - If it's a Yes instance and the certificate is correct, it returns Yes.
  - Otherwise, it returns No.
    - If it's a Yes instance but the certificate is not correct.
    - If it's a No instance.
- Show that the worst-case run time of your algorithm is bounded by some polynomial.

16 P $\subseteq$ NP

Every problem in P is also in NP.

Why?

17 NP $\subseteq$ P?

Are there problems in NP that are not in P?

That is: Are all problems that are polynomially verifiable also polynomially solvable?

Note: If NP $\subset$ P, then P $=$ NP.

18 Reductions

We can show relationships between problems using **reductions**:

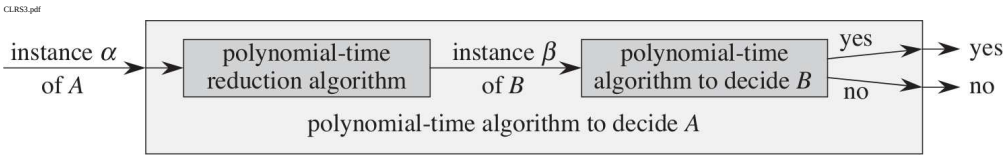
Definition: A language L_1 is **polynomial-time reducible** to another language L_2 if there exists a polynomial time algorithm f such that

$$x \in L_1 \text{ if and only if } f(x) \in L_2$$

If L_1 is polynomial-time reducible to L_2, we write $L_1 \leq_P L_2$

19 Polynomial-time reductions preserve polynomial-time solvability

Lemma (34.2): If $L_2 \in \mathrm{P}$ and $L_1 \leq_P L_2$, then $L_1 \in \mathrm{P}$.



Intuition: $L_1 \leq_P L_2$ means that L_1 is "no harder" to solve than L_2.

20 NP-hard

Definition: A language L is **NP-hard** if, for every $L' \in NP$, $L' \leq_P L$.

21 Watch out!

It is very easy to get the direction of the reduction wrong.

- L' — known NP-complete problem
- L — problem you want to show is NP-hard

The reduction must be an algorithm whose

- input is instance of problem L', and
- output is an equivalent instance of problem L.

22 How to prove: NP-hard

To prove that your problem L is in NP-hard:

- Choose some other problem L' that is already known to be NP-complete.
- Describe an algorithm that converts an instance of L' into an instance of L.
- Double check the direction: Does your reduction algorithm accept an instance of L' as its input?
- Show that your algorithm is a reduction.
  - If it's a Yes instance of L', it should produce a Yes instance of L.
  - If it's a No instance of L', it should produce a No instance of L.
- Show that the worst-case run time of your algorithm is polynomial.

23 NP-complete

Definition: A language that is both NP and NP-hard is called **NP-complete**.

Intuition: NP-complete is the set of the "hardest" problems in NP.

24 Why is NP-complete important?

Theorem 34.4: If any language in NP-complete is polynomial-time solvable, then $P = NP$.

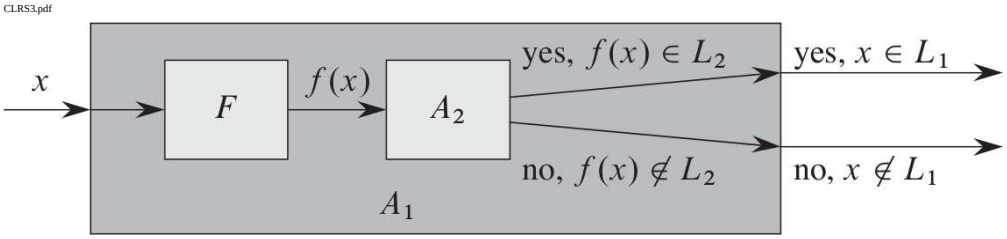
25 The Cook-Levin Theorem

SAT:
Instance: A Boolean formula ϕ consisting of variables, parentheses, and and/or/not operators.
Question: Is there an assignment of True/False values to the variables that makes the formula evaluate to True?

Theorem (Cook-Levin): Boolean satisfiability is NP-complete.

(The surprisingly accessible proof is covered in CSCE551.)

26 NP-completeness proofs



Intuition: This is essentially a proof by contradiction. We're showing that, if we have a polynomial-time algorithm to decide L_2, then we can use it form a polynomial-time algorithm to decide L_1.

27 How to prove: NP-complete

To prove that your problem L is in NP-complete:

- Prove that your problem is in NP.
- Prove that your problem is NP-hard.

28 Example: 3-SAT

Definition: A Boolean formula in **conjunctive normal form** is a series of clauses.

- Each clause is an OR of literals (that is, variables or negations of variables).
- The complete formula is the AND of all of the clauses.

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_5)$$

3-SAT:
Instance: A CNF formula with 3 literals in each clause.
Question: Is there an assignment of True/False values to the variables that makes the formula evaluate to True?

Theorem: 3-SAT is NP-complete.

Proof: Reduction from SAT. (CLRS 1082)

29 Example: Vertex cover

VERTEX COVER:
Instance: A graph G and an integer K.
Question: Is there a set of K vertices in G that touches each edge at least once?

30 VERTEX COVER is in NP

Theorem: VERTEX COVER is in NP.

Proof: Use the set of vertices that covers the graph as the certificate. Verify that there are at most K vertices (constant time) and that each edge touches at least one of them (linear time).

31 VERTEX COVER is NP-hard

Theorem: VERTEX COVER is NP-hard.

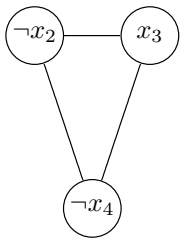
Proof: Reduction from 3-SAT. Given an arbitrary instance ϕ of 3-SAT with n variables and m clauses, form an instance (G, K) of VERTEX COVER as follows.

- **Truth-setting components**: One vertex for each literal in ϕ, with each x connected to $\neg x$.

  Idea: A vertex cover must contain at least one vertex from each pair.

32 Vertex cover (reduction continued)

- **Clause-satisfaction components**: Three vertices for each clause, connected to each other.



  Idea: A vertex cover must select at least two vertices from each of these triangles.

33 Vertex cover (reduction continued)

- **Connecting edges:** A edge connecting each node in each clause-satisfaction component to the corresponding truth-setting node.

- **Number of vertices allowed in cover:** Choose $K = n + 2m$.

34 Vertex cover (correctness of reduction)

Theorem: If ϕ is satisfiable, then G has a vertex cover of size $n + 2m$.

Proof: Given a satisfying assignment t for ϕ, consider this set of vertices:

- In truth-setting components, choose x_i if t sets x_i to true, or $\neg x_i$ otherwise.
- In clause-satisfaction components, find the first literal set to true, and choose the *other two* vertices.

Clearly this set has size $n + 2m$. Note that it covers all edges (1) within truth-setting components, (2) within clause-satisfaction components, and (3) between truth-setting and clause-satisfaction components. Therefore, it is a vertex cover for G.

35 Vertex cover (correctness of reduction)

Theorem: If G has a vertex cover of size $n + 2m$, then ϕ is satisfiable.

Proof: Suppose there exists a vertex cover A of G with size $n + 2m$. By construction, this cover must include

- one vertex in each truth-setting component, and
- two vertices in each clause-satisfaction component.

Let t be the truth assignment that makes x_i true iff its truth-setting vertex is in the vertex cover A.

- For each clause C in ϕ, the corresponding clause-satisfaction component has exactly one vertex v that is not in the vertex cover.
- An edge connects v to a truth-setting vertex u. Since A is a vertex cover, and $v \notin A$, we know that $u \in A$.
- Therefore, the literal associated with u is satisfied in t, which implies that C is satisfied.

Because t satisfies each clause of ϕ, it satisfies ϕ.

36 More examples

Additional reduction examples:

- CLRS 34.5
- GJ
- Fenner's notes
- Erickson's notes
- All over the internet.