CSC373: Algorithm Design, Analysis and Complexity
Fall 2017

Allan Borodin

October 18, 2017

Week 6 : Annoucements

- We have begun grading the test and the plan is to return those in a week. The assignment grading will be completed after that.
- Discussion of the term test questiions will be part of tomorrow tutorials. Not sure if we will post solutions sketches. Why not?
- **Next assignment:** The first two questions for Assignment 2 have been posted. Due date: Thursday, November 9 at 2PM. We plan to post 3 more questions on complexity theory to complete the assignment.
- We are making this assignment and the next assignment shorter than the last assignment. We try to post questions as soon as the relevant material is presented. It is best to start working on questions as soon as possible and not wait until the due date.
- I believe that a course should take say 10 or at most 12 hours per week. Of course, if you wait for the week that an assignment is due, the time spent on the course that week could be considerably more.

Todays agenda

Note: Todays agenda will stretch into next week

- Review of basic aspects of complexity theory.
- Encoding of inputs/outputs
- Functions, search problems, and optimization problems
- Definitions of decision problems (equivalently) languages in the classes P and NP
- Polynomial time reductions and the concept of NP completeness.
- Polynomial time reductions vs polynomial time transformations.
- A tree of polynomial time transformations starting from 3SAT.
- Examples of polynomial time transformations.

Computational complexity theory

Computational complexity aims to measure the amount of resources *required* for various computational problems.

In order to makes this into a theory, we need to have precise models of computation and measures of complexity within those models.

We will later introduce the Turing machine model, a precise mathematical model, and claim that all "classical" computational models can be "efficiently" simulated by Turing machines. For our purposes, "efficiently" will mean within a polynomial factor.

For what is called "fine-grained complexity", where we care say about the difference between n^2 and n^3, we use other formalizations and appropriate definitions of "efficiently". This may be one of the topics we will consider in the last week of the course.

What are the main measures of complexity that we study?

- sequential time. For the P vs NP issue, this is the relevant measure of complexity.
- memory referred to as space in complexity theory.
- parallel time
- In randomized algorithms, we also consider the amount of randomness used.
- And, in addition, we study tradeoffs bewteen these resources. For example, time vs space.

We now continue our discussion from last week with a brief review.

Encoding of inputs and outputs

- We are always assuming that inputs and outputs are encoded as strings over some finite alphabet S with at least 2 symbols.
  - S^n denotes the set of all finite strings of length n over the alphabet S.
  - The empty string is the only string of length 0.
  - $S^* = \bigcup_{n \geq 0} S^n$ denotes the set of all finite strings over the alphabet S
- We can use as many symbols as we want but 2 suffices for our purpose. (**Note:** finitely many symbols on a keyboard.)
- We can also encode in unary, but that causes an exponential blowup in representation.
- We can encode a set of inputs or outputs $w_1, \ldots, w_n$ by having a special symbol (say #) to separate the inputs but again this can all be encoded back into 2 symbols.
- If we need to distinguish components of an input, we can denote such an encoding of many inputs (or outputs) as $\langle w_1, \ldots, w_n \rangle$.
  - We will usually not need to be so careful about the distinction between an object G and its encoding $\langle G \rangle$.

What does it mean to be efficiently computable?

- Let n denote "size" (i.e. the encoded length) of the inputs and outputs of the problem.
- Using diagonalization, it is not difficult to show that

For any computable time bound $T(n)$, there are computable functions (in particular, decision problems) not computable within time T(n) for inputs/outputs of size n.

- Following Cobham and Edmonds (circa 1965), we will equate the intuitive concept of "efficiently computable" with computable in polynomial time (i.e. time bounded by a polynomial function of the encoded length of the inputs and outputs).
  - This has sometimes been called the Extended Church-Turing Thesis.
  - As we reamrked last class, this hypothesis is not literally believed in contrast to the Church-Turing Thesis.
  - But it is an abstraction that has led to great progress in computing.
- **Informal claim restated:** Any function (polynomial time) computable is (polynomial time) computable by a Turing machine.

What are the objects of study?

- Since it suffices to encode all inputs as finite strings over the alphabet $S = \{0, 1\} = \{\text{false}, \text{true}\}$, we often refer to the complexity issues of such computations as Boolean complexity theory.
  - This is in contrast to, say, arithmetic complexity theory, or the theory of real valued computation.
- Our general objects of study (for Boolean complexity theory) are the computation of:
  1. Functions $f : S^* \to S^*$
  2. Decision problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - ★ Given x, determine (i.e. output YES or NO) if there exists a y satisfying $R(x, y)$.
  3. Search problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - ★ Given x, output a y satisfying $R(x, y)$ or say that no such y exists.
  4. Optimization problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation, and $c : S^* \times S^* \to \Re$ be an objective function.
     - ★ Given x, output a y satsifying $R(x, y)$ so as to minimize (or maximize) $c(x, y)$ or say that no such y exists.

Polynomial time computable functions

- A function $f : S^* \to S^*$ is computable in polynomial time $T(\cdot)$ if:
  1. $T(n)$ is a polynomial.
  2. There is an algorithm (to be precise a Turing machine or an idealized RAM program with an appropriate instruction set) such that: For all inputs $w \in S^*$, the algorithm halts using at most $T(n)$ "time steps" where $n = |w| + |f(w)|$.
- We can define polynomial time search problems, optimization problems and decision problems similarly.
- We will generally not be dealing with functions where $|f(w)| > |w|$. So it will suffice to let n be the encoded length of the input.
- In particular, for decision problems, n will always refer to the length of the encoded input.
- We let P denote the class of decision problems that are solvable (decideable) in polynomial time. We sometimes abuse notation and also use P to denote any problem computable in polynomial time.

Computational Complexity Theory

- Classifies problems according to their computational difficulty.
- The class P (Polynomial Time) [Cobham, Edmonds, 1965]
- P consists of all problems that have an efficient (e.g. $n, n^2 \ldots$) algorithm. (n is the input length)

Examples in P

- Addition, Multiplication, Square Roots
- Shortest Path (Google Maps)
- Network flows (Internet Routing)
- Pattern matching (Spell Checking, Text Processing)
- Fast Fourier Transform (Audio and Image Processing, Oil Exploration)
- Recognizing Prime Numbers [Agrawal-Kayal-Saxena 2002]
- ...

Polynomial time continued

- One of the nice properties of the Turing machine model is that the concept of a computational step and hence time is well defined.
- **Warning:** If say a RAM has a multiplication operation, then by repeatedly squaring we can compute 2^{2^n} in n operations.
  - We argue that we cannot assume such an operation only costs 1 time unit since the operands will have 2^n bits and hence we might be encoding an exponential time computation within such operations.
- In particular, it can be shown that if we do not account properly for the cost of RAM operations, then we can factor integers (and thus be able to break some encryption schemes such as RSA) in polynomial time but such a computational algorithm would not be considered realistic.
- One way to deal with such RAM models is to say that any operation on operands of length m requires time m.

Classical vs quantum computation

- Quantum computation takes advantage of principles of quantum mechanics (such as "entangled states") which allow a quantum state involving n 'qubits' to be a "superposition" of up to 2^n possible bit strings.
- In 1994 Peter Shor proved that a quantum computer can factor numbers into primes in polynomial time.
- So if large quantum computers can be built, they could crack the RSA encryption scheme. (There are other ways that RSA could be broken (side channel attacks.)
- **The Catch:** Despite substantial effort, physicists have so far been unable to build an actual quantum computer large enough to process more than a dozen or so bits of information.
- **Caveat:** Quantum cryptography provides a promising approach to secure communication in which security (rather than complexity) depends on principles of quantum mechanics.

Note

Quantum computation does NOT change the Church-Turing thesis, that is, what is computable. But it does seem to change what is computable in polynomial time.

What have we been doing so far in this course

- So far, we have almost entirely been presenting polynomial time algorithms.
- As one exception, we did consider the pseudo polynomial time DP for the knapsack problem. Additiionally, we presented the $O(n^2 2^n)$ time DP for the travelling salesman problem improving upon the brute-force $n!$ time algorithm.
- Many times we didn't care if the operands (say in interval scheduling) were real numbers or integers.
- We just assumed that we could do basic arithmetic operations and comparisons in one step.
- This was not a problem because we didn't use algorithms that would build up large integers. (Note that using only addition, repeated doubling can only produce a number as large as 2^n in n steps).

NP-hardness

- We have often been refering to *NP*-hardness when we considered computing optimal solutions to a number of optimization problems.
- We alluded to the widely held belief that such problems cannot be computed efficiently (for all inputs).
- This will be expressed as the conjecture (sometimes called Cook's Hypothesis) that $P \neq NP$. When we say that an optimization problem is *NP*-hard, it implies that we cannot optimally solve such a problem if we assume $P \neq NP$.
- We will later consider approximate optimization algorithms.
- What is the evidence we have for believing in this conjecture?
  - Briefly stated, the main evidence is the extensive number of problems which can be shown to be "equivalent" in the sense that if any one of them can be computed efficiently (i.e. in P) then they all are.
  - These are problems that have been studied for many years (decades and in certain cases centuries) without anyone being able to find polynomial time algorithms.

What is NP?

- The class NP (Nondeterministic Polynomial Time)
- NP consists of all search problems for which a solution can be efficiently (i.e. in polynomial time) verified.
- More specifically, a set (or language) L is in the class NP if there is a polynomial time computable $R(x, y)$ and a polynomial time computable function f such that for all $x \in L$, there is a certificate y such that $|y| \leq f(|x|)$ and $R(x, y)$.

Examples in NP (besides everything in P)

- Given an integer x (in say binary of decimal representation), is it a composite number? (This is in fact a polynomial time computable decision problem.)
- Given a graph G, can it be vertex colored in 3 colors?
- Given a set $S = \{a_i\}$ of integers can it partitioned into two subsets S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_i \in S_2} a_i$?

P versus NP

- P: Problems for which solutions/certificates can be efficiently found
- NP: Problems for which solutions/certificateds can be efficiently verified

Conjecture

$$P \neq NP$$

- Most computer scientists believe this conjecture.
- But is seems to be incredibly hard to prove.

Why is proving $P \neq NP$ difficult?

- One reason is that some search problems in NP turn out to be relatively easy. An example is the maximum bipartite matching problem introduced in Week 4). More generally, matching in any graph is polynomial time solvable but this is not an easy result.

The matching problem for undirected graphs

Given a large group of people, we want to pair them up to work on projects. We know which pairs of people are compatible, and (if possible) we want to put them all in compatible pairs.

- If there are 50 or more people, a brute force approach of trying all possible pairings would take billions of years.
- However in 1965 Jack Edmonds found an ingenious efficient algorithm. So this problem is in P.
- There is often a "fine line" between what is and what is not known to be efficiently solvable (e.g. polynomial time 2SAT vs NP-hard 3SAT).

NP-Complete Problems

- These are the hardest *NP* problems.
- A problem A is *p*-reducible to a problem B if an "oracle" (i.e. a subroutine) for B can be used to efficiently solve A.
- If A is *p*-reducible to B, then any efficient procedure for solving B can be turned into an efficient procedure for A.
- If A is *p*-reducible to B and B is in P, then A is in P.

Definition

A problem B is *NP*-complete if B is in *NP* and every problem A in *NP* is *p*-reducible to B.

Theorem

If A is NP-complete and A is in P, then $P = NP$.

To show $P = NP$ you just need to find a fast (polynomial-time) algorithm for any one *NP*-complete problem!!!

Conjunctive normal form propositional formulas

The SAT problem is defined in terms of inputs given as conjunctive normal form (CNF) formulas.
Here are the standard definitions regarding CNF propositional formulas:

- A literal is a propositional variable x or its negation $\bar{x}$.
- A clause $C = \ell^1 \vee \ell^2 \ldots \vee \ell^r$ is a disjunction of literals.
- A CNF formula $F = C_1 \wedge C_2 \ldots \wedge C_m$ is a conjunction of clauses.
- A CNF formula is a kCNF formula if every clause has at most k literals. For our purposes we will abuse terminology and say that every clause has exactly k literals. It is always assumed that no variable appears twice in any clause.

$$\left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

Figure: An example of a 3CNF formula

Sat and 3SAT

A formula is *satisfiable* if there is an assignment of truth values (i.e. TRUE, FALSE) to the propositional variables such that the formula evaluate to TRUE. For CNF formulas, this means that there is an assignment of truth values such that every clause is TRUE.

SAT (resp. 3SAT) is the decision problem that determines ifn a CNF (resp. 3CNF) formula is satisfiable.

Following the initial results of Cook and Karp, our development of *NP* complete problems rests on showing that SAT is *NP* complete. We have to show how to reduce *any NP* decision problem to SAT.

We delay the proof of that Theorem until after some examples of reductions. Some reductions will be relatively easy and some not.

Many research papers were written in the early 1970's establishiing NP-completeness for specific problems. The monograph by Garey and Johnson was a valued reference keeping track of many examples. There are web sites devoted to particular domains listing NP complete problems.

- A great many (thousands) of problems have been shown to be *NP*-complete.
- Most scheduling related problems (delivery trucks, exams etc) are *NP*-complete.
- The following simple exam scheduling problem is *NP*-complete:

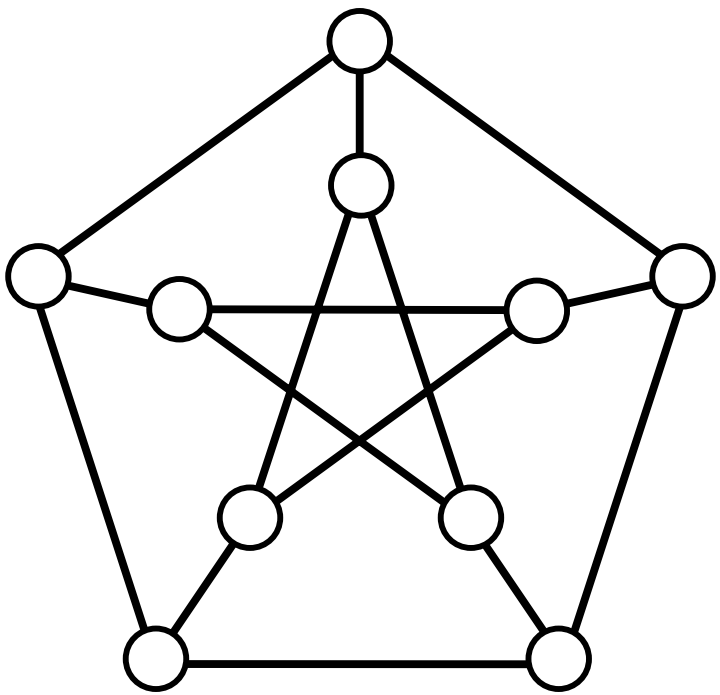
Example

- We need to schedule N examinations, and only three time slots are available.
- We are given a list of exam conflicts: A conflict is a pair of exams that cannot be offered at the same time, because some student needs to take both of them.
- **Problem:** Determine if there is a way of assigning each exam to one of the time slots T_1, T_2, T_3, so that no two conflicting exams are assigned to the same time slot.
- This problem is known as the graph 3-colourability problem.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.

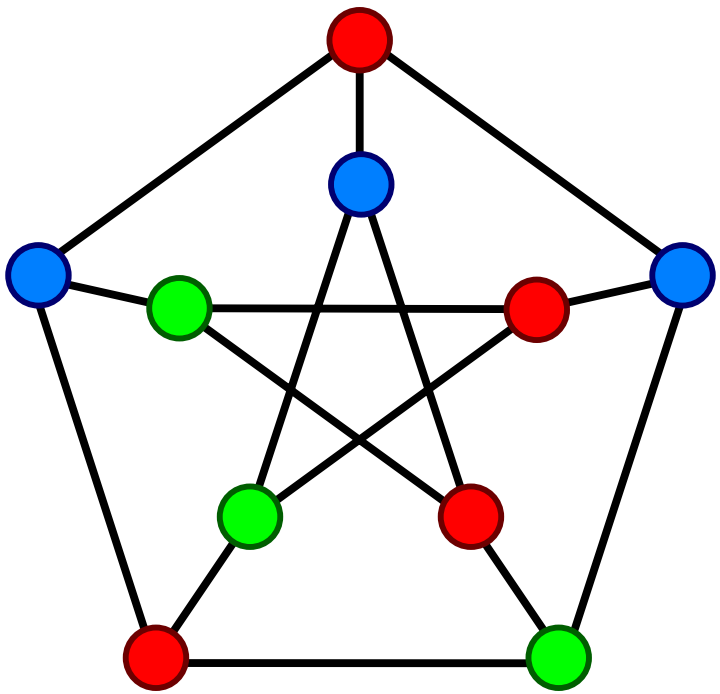


Imagine trying to decide this when there are say hundreds or thousands of nodes.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



Imagine trying to decide this when there are say hundreds or thousands of nodes.

Some more remarks on graph coloring

- The natural graph coloring optimization problem is to color a graph with the fewest number of colors.
- We can phrase it as a search or decision problem by saying that the input is a pair (G, k) and then
  1. The search problem is to find a k-coloring of the graph G if one exists.
  2. The decision problem is to determine whether or not G has a k coloring.
  3. Clearly, solving the optimization problem solves the search problem which in turn solves the decision problem.
  4. Conversely, if we can efficiently solve an NP complete decision problem then we can efficiently solve the search and optimization problems. This can be shown in general but we can also show it for specific problems (e.g., for graph coloring as you will show).
- Formally it is the graph coloring decision problem which is NP-complete. More precisely, the graph coloring decision problem for any fixed $k \geq 3$ is NP-complete. However, 2-Colorability is in P.
- Search or optimization problems that reduce to an NP-complete decision problem are then called NP-hard.

Reducing Graph 3-Colourability to 3SAT

We begin our examples of reductions between NP decision problems with a reduction that is implied by the fact that graph 3-colouring is in NP and hence must reduce to 3SAT which is NP-complete. This reduction would be considered a relatively easy reduction (certainly in hindsight) but it illustrates how reductions can be between problems coming from what are traditionally thought of as different research areas.

- We are given a graph G with nodes, say $V = \{v_1, v_2, \ldots, v_n\}$
- We are given a list of edges, say $(v_3, v_5), (v_2, v_6), (v_3, v_6), \ldots$
- We need to find a 3CNF formula F which is satisfiable if and only if G can be colored with 3 colors (say red, blue, green). **Note:** Any permutation or renaming of the colors does the change what follows.
- We use three different types of Boolean VARIABLES
  $r_1, r_2, \ldots, r_n$ (r_i means node i is colored red)
  $b_1, b_2, \ldots, b_n$ (b_i means node i is colored blue)
  $g_1, g_2, \ldots, g_n$ (g_i means node i is colored green)

Here we are abusing terminology as "means" is really "intended meaning"

- Here are the CLAUSES for the formula F:
  - We need one clause for each node:
    $(r_1 \vee b_1 \vee g_1)$ (node 1 gets at least one color)
    $(r_2 \vee b_2 \vee g_2)$ (node 2 gets at least one color)
    ...
    $(r_n \vee b_n \vee g_n)$ (node n gets at least one color)
  - We could put in clauses saying that no node gets colored with more than one color but coloring a node with more than one color can only make it more difficult to color so we really don't need these clauses.
  - We need 3 clauses for each edge: For the edge (v_3, v_5) we need
    $(\overline{r_3} \vee \overline{r_5})$ (v_3 and v_5 not both red)
    $(\overline{b_3} \vee \overline{b_5})$ (v_3 and v_5 not both blue)
    $(\overline{g_3} \vee \overline{g_5})$ (v_3 and v_5 not both green)
- The size of the formula F is comparable to the size of the graph G.
- **Check:** G is 3-colorable if and only if F is satisfiable.

On the nature of this polynomial time reduction

- If we consider the previous reduction of 3-coloring to 3-SAT, it can be seen as a very simple type of reduction.
- Namely, given an input w to the 3-coloring problem, it is transformed (in polynomial time) to say $h(w)$ such that

$$w \in \{G \mid G \text{ can be 3-colored}\} \text{ iff}$$
$$h(w) \in \{F \mid F \text{ is a satisfiable 3CNF formula}\}.$$

- If we express the problems as decision probems, the reduction of bipartite matching to maximum flows is also a transformation.

Polynomial time transformations

- We say that a language L_1 is polynomial time transformable to L_2 if there exists a polynomial time function h such that

$$w \in L_1 \text{ iff } h(w) \in L_2.$$

- The function h is called a polynomial time transformation.

Polynomial time reductions and transformations

- In practice, when we are reducing one NP problem to another NP problem, it will be a polynomial time transformation.
- We will use the same notation $\leq_p$ to denote a polynomial time reduction and polynomial time transformation.
- As we have observed before if $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$.
- The contrapositive says that if $L_1 \leq_p L_2$ and $L_1 \notin P$, then $L_2 \notin P$.

$\leq_p$ is transitive

- An important fact that we will use to prove NP completeness of problems is that polynomial time reductions are transitive.
- That is $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ implies $L_1 \leq_p L_3$.

- The proof for transformations is easy to see. For say that $L_1 \leq_p L_2$ via g and $L_2 \leq_p L_3$ via h, then $L_1 \leq_p L_3$ via $h \circ g$;
that is, $w \in L_1$ iff $h(g(w) \in L_3$.

Polynomial reductions/transformations continued

- One fact that holds for polynomial time transformation but is believed not to hold for polynomial time reductions is the following:

NP closed under polynomial time transformation

If $L_1 \leq_p L_2$ and $L_2 \in NP$ then $L_1 \in NP$.

- The closure of NP under polynomial time transformations is also easy to see. Namely,

Suppose

- $L_2 = \{w \mid \exists y, |y| \leq q(|w|)$ and $R(w, y)\}$ for some polynomial time relation R and polynomial q, and
- $L_1 \leq_p L_2$ via h.

Then

$L_1 = \{x \mid \exists y', |y'| \leq q(|h(x)|$ and $R'(x, y')\}$ where $R'(x, y') = R(h(x), y')$

Polynomial reductions/transformations continued

- On the other hand we do not believe that NP is closed under general polynomial time reductions.
- Specifically, for general polynomial time transformations we have $\bar{L} \leq_p L$. Here $\bar{L} = \{w | w \notin L\}$ is the language complement of L.
- We do not believe that NP is closed under language complementation.
- For example, how would you provide a short verification that a propositional formula F is *not* satisfiable? Or how would you show that a graph G *cannot* be 3-coloured?
- While we will use polynomial time transformations between decision problems/languages we need to use the more general polynomial time reductions to say reduce a search or optimization problem to a decision problem.

So how do we show that a problem is *NP* complete?

- Showing that a language (i.e. decision problem) L is *NP* complete involves establishing two facts:

1. L is in *NP*
2. Showing that L is *NP*-hard; that is showing

$$L' \leq_p L \text{ for every } L' \in NP$$

- Usually establishing (??) is relatively easy and is done directly in terms of the definition of $L \in NP$.
  - That is, one shows how to verify membership in L by exhibiting an appropriate certificate. (It could also be established by a polynomial time transformation to a known $L \in NP$.)
- Establishing (??), i.e. *NP*-hardness of L, is usually done by reducing some known *NP* complete problem L' to L.

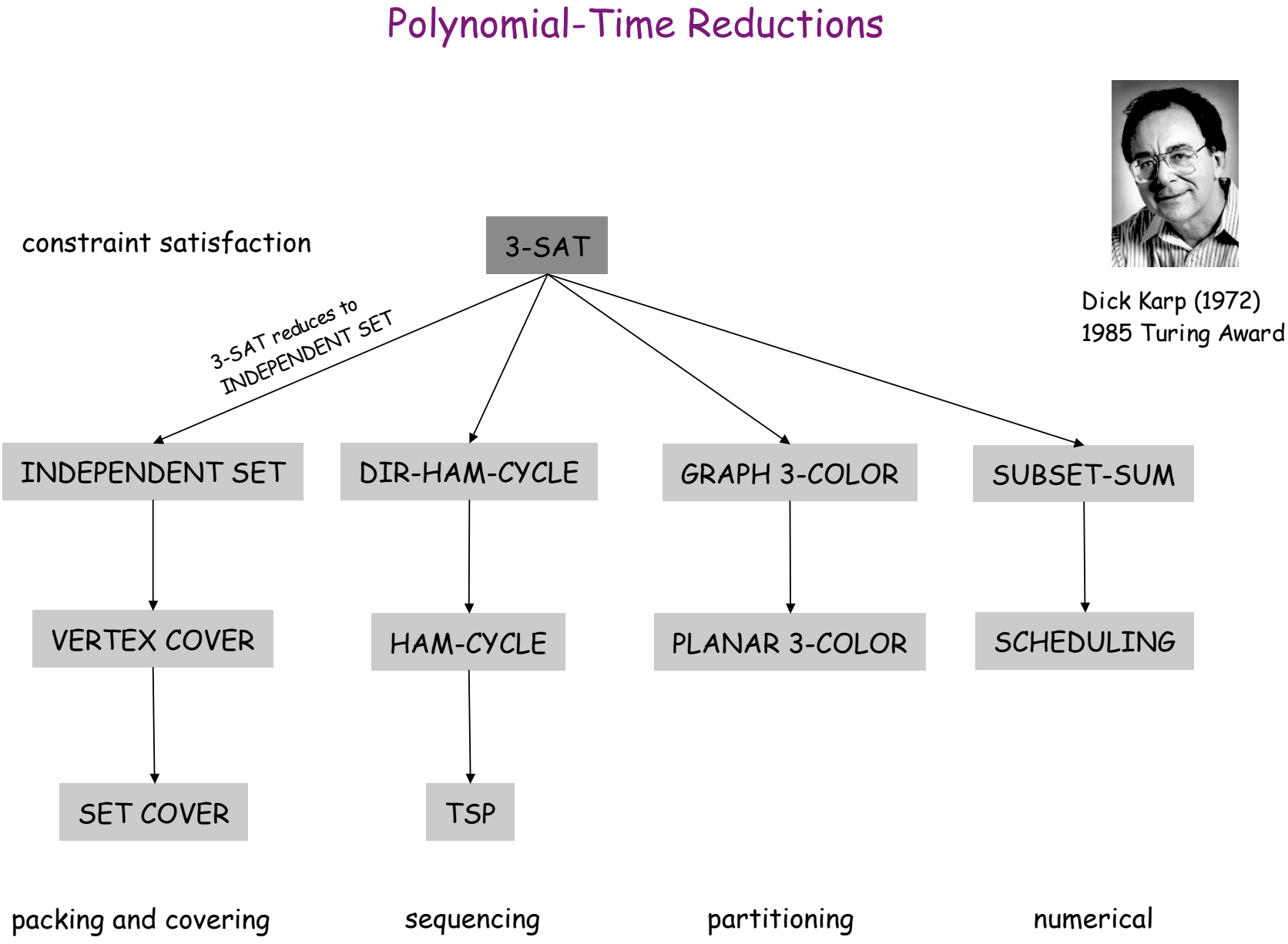
But how do we show that there are any *NP* complete problems?

How do we get started?

- Once we have established that there exists at least one *NP* complete problem then we can use polynomial time reductions and transitivity to establish that many other *NP* problems are *NP* hard.
- Following Cook's original result, we will show that *SAT* (and even 3*SAT*) is NP complete "from first principles".

- It is easy to see that *SAT* is in *NP*.
- We will (later) show that *SAT* is *NP* hard by showing how to encode an arbitrary "non-ddeterfministic" polynomial time (Turing) computation by a *CNF* formula. We can simply think of such a computation as one that "guesses" a certificate (i.e. makes non-deterministic Turing machine operations) and then verifies the certificate.

A tree of reductions/transformations



Dick Karp (1972)
1985 Turing Award

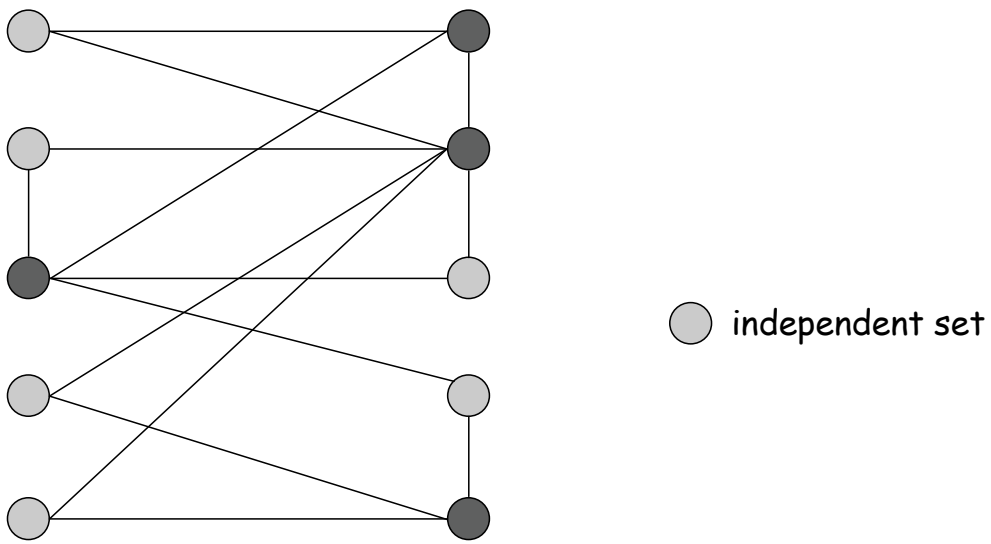
A little history of NP-completenes

- In his original 1971 seminal paper, Cook was interested in theorem proving. Stephen Cook won the Turing award in 1982
- Cook used the general notion of polynomial time reducibility which is called polynomial time Turing reducibility and sometimes called Cook reducibility.
- Cook established the NP completeness of 3SAT as well as a problem that includes CLIQUE $= \{(G, k) | G$ has a k clique $\}$.
- Independently, in the (former) Soviet Union, Leonid Levin proved an analogous result for SAT (and other problems) as a search problem.
- Following Cook's paper, Karp exhibited over 20 prominent problems that were also NP-complete.
- Karp showed that polynomial time transformations (sometimes called polynomial many to one reductions or Karp reductions) were sufficient to establish the NP completness of these problems.

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

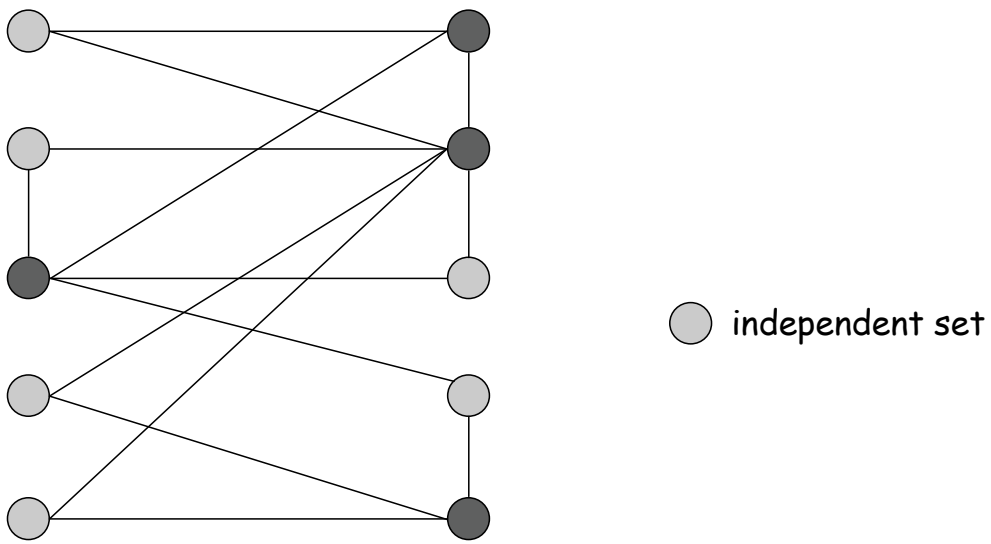


- Question: Is there an independent set of size 6?

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

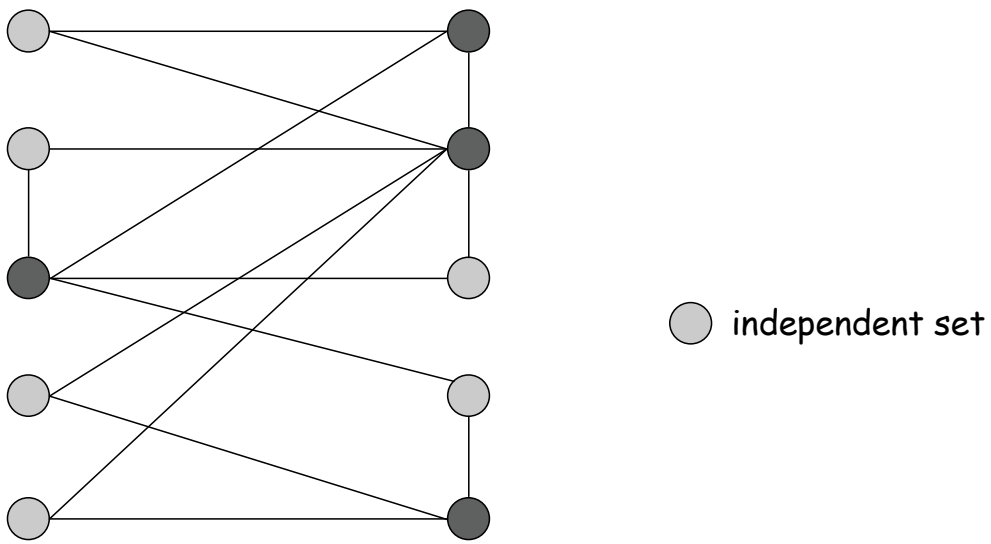


- Question: Is there an independent set of size 6? Yes.

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

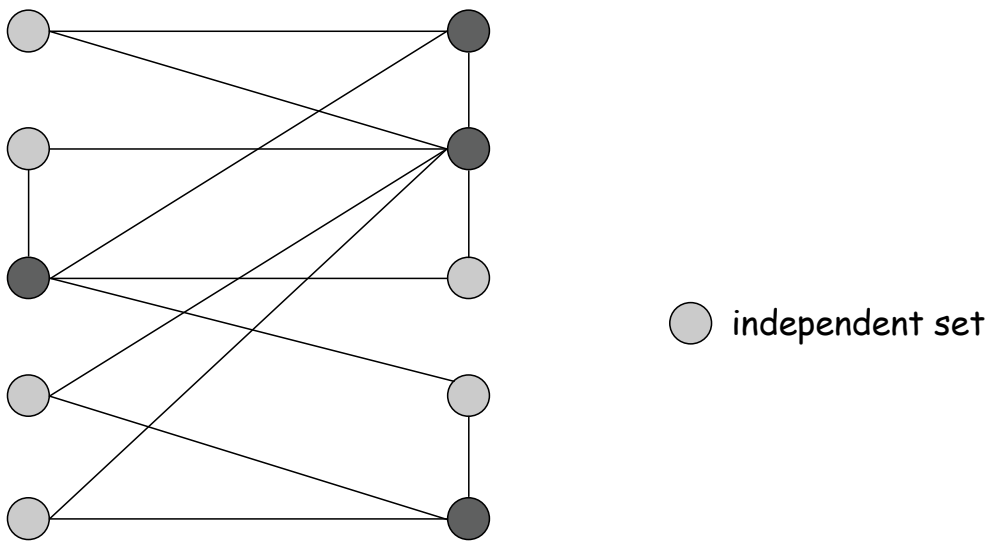


- Question: Is there an independent set of size 6? Yes.
- Question: Is there an independent set of size 7?

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?



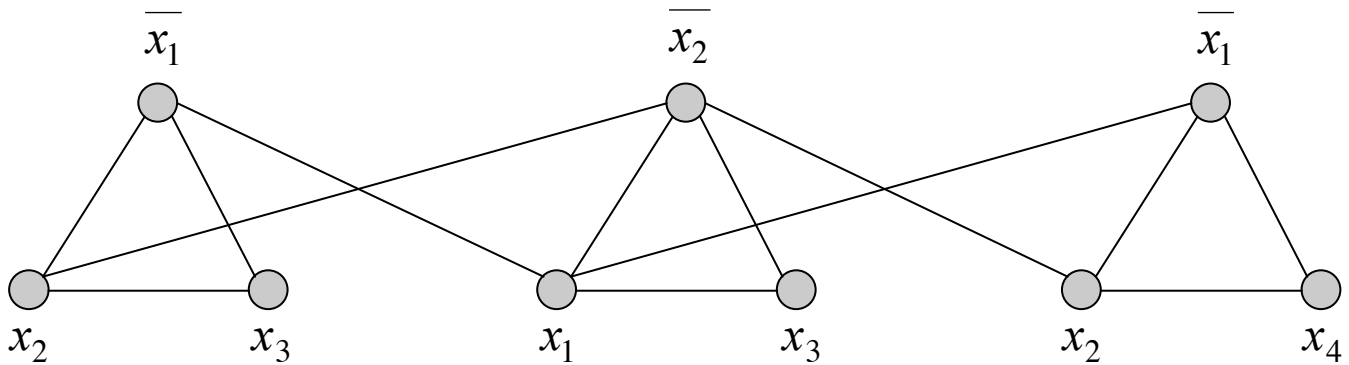
- Question: Is there an independent set of size 6? Yes.
- Question: Is there an independent set of size 7? No.

3SAT reduces to Independent Set

Claim

3SAT $\leq_p$ Independent Set

- Given an instance F of 3SAT with k clauses, we construct an instance (G, k) of Independent Set that has an independent set of size k iff F is satisfiable.
- G contains 3 vertices for each clause; i.e. one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

Subset Sum

Subset Sum

- Given a set of integers $S = \{w_1, w_2, \ldots, w_n\}$ and an integer W.
- Is there a subset $S' \subseteq S$ that adds up to exactly W?

Example

- Given $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ and $W = 3754$.
- Question: Do we have a solution?
- Answer: Yes. $1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754$.

3SAT reduces to Subset Sum

Claim

3SAT $\leq_p$ Subset Sum

- Given an instance F of 3SAT, we construct an instance of Subset Sum that has solution iff F is satisfiable.
- In the following array (next slide), rows represent integers represented in decimal. For each propositional variable we have a column specifying that each variable has just one truth assignment (i.e., *true* $= 1$) and for each clause we have a column saying that the clause is satisfiable. The "dummy rows" makes it possible to sum each column to 4 if and only if there is at least one literal set to true. Note that the decimal representation nsures that addition in each column will not carry over to the next column.

3SAT reduces to Subset Sum continued

The figure illustrates how a specific 3CNF formula is transformed into a set of integers and a target (bottow row).

$$C_1 = \bar{x} \vee y \vee z$$
$$C_2 = x \vee \bar{y} \vee z$$
$$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$$

	x	y	z	C_1	C_2	C_3
x	1	0	0	0	1	0
¬ x	1	0	0	1	0	1
y	0	1	0	1	0	0
¬ y	0	1	0	0	1	1
z	0	0	1	1	1	0
¬ z	0	0	1	0	0	1
dummies to get clause columns to sum to 4	0	0	0	1	0	0
	0	0	0	2	0	0
	0	0	0	0	1	0
	0	0	0	0	2	0
	0	0	0	0	0	1
	0	0	0	0	0	2
W	1	1	1	4	4	4

Reviewing how to show some L is NP complete.

- We must show $L \in NP$. To do so, we provide a polynomial time verification predicate $R(x, y)$ and polynomial length certificate y for every $x \in L$; that is, $L = \{x | \exists y, R(x, y) \text{ and } |y| \leq q(|x|)\}$.
- We must show that L is NP hard (say with respect to polynomial time tranformations); that is, for some known NP complete L', there is a polynomial time transducer function h such that $x \in L'$ iff $h(x) \in L$. This then establishes that $L' \leq_p L$.
- Warning The reduction/transformation $L' \leq_p L$ must be in the correct direction and h must be defined for every input x; that is, one must also show that if $x \notin L'$ then $h(x) \notin L$ as well as showing that if $x \in L'$ then $h(x) \in L$.