Lecture 19 Intractability I: P, NP, and NP-completeness	L19.1 6.046 05/02/2017

Today

- Problem types: decision, search, optimization
- Polynomial time (P) vs. Non-deterministic polynomial time (NP)
- Reductions
- NP-completeness
- Cook's theorem

Reading for this and the next lecture:
CLRS Chpt 34

Problems: Seemingly related problems can be of vastly different difficulties

Example 1: Shortest vs. Longest Path

Shortest path: Given a weighted graph $G=(V,E,w)$ and vertices s & t, find the shortest (simple) path from s to t. → Polynomial-time algs. exist

Longest path: Given the same input, find the longest (simple) path from s to t.
→ No polynomial-time alg. known

Example 2: MST vs. TSP

Minimum Spanning Tree (MST): Given a weighted graph $G=(V,E,w)$, find a spanning tree of minimum weight. → Polynominal-time algs. exist: $O(|E|\log|V|)$

Traveling Salesman Problem (TSP): Given the same input, find a spanning simple cycle of minimum weight. → No polynomial-time alg. known: Current best $O(|V|\cdot 2^{|V|})$ ops

Example 3: Linear vs. Integer Programming

Linear Programming: Given a matrix A and vectors $\vec{b}$ & $\vec{c}$, find the real vector $\vec{x}$ that maximizes $\vec{c}\cdot\vec{x}$ subject to the constraints $A\vec{x}\leq\vec{b}$.
→ Polynomial-time algs. exist.

Integer Programming: Given a matrix A and vectors $\vec{b}$ & $\vec{c}$, find the integer vector $\vec{x}$ that maximizes $\vec{c}\cdot\vec{x}$ subject to the constraints $A\vec{x}\leq\vec{b}$.
→ No polynomial-time alg. known.

Example 4: 2d- vs. 3d-Matching

2d-Matching: Given a bipartite graph connecting men to compatible women, find a perfect matching of men to women. → Poly-time algs. exist.

3d-Matching: Given a tri-partite graph connecting men to compatible women, men to compatible restaurants, and women to compatible restaurants, find a collection of man-woman-restaurant triplets so that each man, woman, and restaurant is in a triplet and none are in more than one triplet.

→ No polynomial-time algorithm known.

Conclusions:

(1) Seemingly small changes to problem statements can lead to drastic changes in our algorithmic understanding (and our current ability to solve).

(2) Longest path, TSP, & Integer programming are problems that have been studied intensely by thousands of researchers over a substantial amount of time, yet they have resisted discovery of polynomial-time algorithms.

Questions:

(1) Are there polynomial-time algorithms for these problems waiting to be found?
(→ huge open problem)

(2) Given a problem, can we tell that it is hard, to avoid "wasting time" trying to find a polynomial time algorithm? (→ yes)

Decision, Search, & Optimization Problems

MST (optimization version): Given a weighted graph $G=(V,E,w)$, find a spanning tree of minimum weight. Result is a tree or "graph not connected".

MST (search version): Given a weighted graph $G=(V,E,w)$ and a budget K, find a spanning tree whose weight $\leq K$ or report that none exists.
Result is a tree or "insufficient budget" or "graph not connected".

MST (decision version): Given a weighted graph $G=(V,E,w)$ and a budget K, decide whether or not there exists a spanning tree with weight $\leq K$.
Result is "yes" or "no".

The existence of a polynomial-time solution to the optimization version of the problem implies a polynomial-time solution to the search version (why?).

Likewise, the existence of a polynomial-time solution to the search version implies one for the decision version.

[Note that sometimes the inference goes in the other direction. For example, a poly-time solution of a search problem can lead to a poly-time solution to the corresponding optimization version by carrying out binary search to find the optimal objective value.]

Conversely, an inability to solve the decision version in poly-time $\Rightarrow$ an inability to solve the search version in poly-time $\Rightarrow$ an inability to solve the optimization version in poly-time. (Why? $p \Rightarrow q \iff \neg q \Rightarrow \neg p$)

↳ For showing intractability, we tend to focus on decision version.

Thus, we will be concerned with the decision versions of our problems:

Shortest Path: Is there a path of length at most K from s to t?

Longest Path: Is there a simple path of length at least K from s to t?

MST: Is there a spanning tree of weight $\leq K$?

TSP: Is there a spanning simple cycle of weight $\leq K$?

Linear Programming: Is there a real vector $\vec{x}$ s.t. $\vec{c} \cdot \vec{x} \geq K$ and $A\vec{x} \leq \vec{b}$?

Integer Programming: Is there an integer vector $\vec{x}$ s.t. $\vec{c} \cdot \vec{x} \geq K$ and $A\vec{x} \leq \vec{b}$?

2d-matching: Is there a perfect 2d-matching?

3d-matching: Is there a perfect 3d-matching?

Polynomial vs. Non-deterministic Polynomial Time

A decision problem Π is solvable in polynomial time if there exists a polynomial-time algorithm A such that:

$\forall x$: (x is a "yes" input for Π) $\iff$ ($A(x)$ ouputs "yes")

We write $\Pi \in P$ — problems solvable in polynomial time.

(shortest path, MST, LP) $\in P$

What can we say for longest path, TSP, integer programming, and 3d-matching?

For every "yes" instance there exists a polynomially short certificate that can be verified in polynomial time to establish that the instance is truly a "yes" instance.

3d-matching: If G is a "yes" instance of a 3d-matching problem, then a certificate for it is a collection of $\frac{|V|}{3}$ woman-man-restaurant triplets. And it can be checked in polynomial time whether or not the collection is truly a perfect 3d-matching.

Longest path: If we have a "yes" instance of a longest path problem (G, K, s, t), then a certificate for it is a path from s to t whose length is at least K. Again, it can be checked in polynomial time whether or not the path is valid and its length is at least K.

Non-Deterministic Polynomial Time (NP) captures problems with polynomially short & polynomial-time verifiable certificates of their "yes" instances.

Formally, we say that $\Pi \in NP$ if there exists a polynomial-time verification algorithm V_Π and a constant c such that

$$\forall x : (\Pi(x) = \text{"yes"}) \iff (\exists \text{ certificate } y \text{ s.t. } |y| \leq |x|^c \text{ & } V_\Pi(x,y) = \text{"yes"})$$

Note: (i) NP does not stand for "non-polynomial" but "non-deterministic polynomial" (because these problems can be solved by a non-deterministic polynomial algorithm that guesses the solution and then checks it).

(ii) $P \subseteq NP$

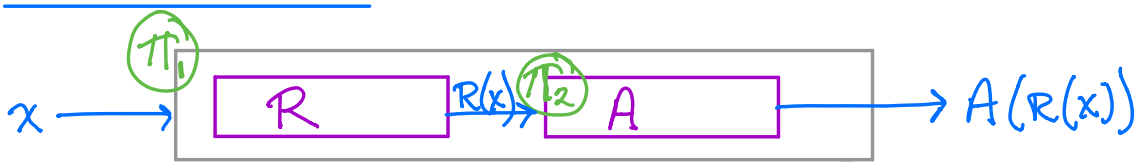
Proof: If $\Pi \in P$, $\exists$ poly-time algorithm A s.t. $\forall x$: $(\Pi(x) = \text{"yes"}) \iff (A(x)$ outputs "yes"). Now define $V_\Pi(x,y) = A(x)$. It follows that $\forall x$: $(\Pi(x) = \text{"yes"}) \iff (V_\Pi(x,y) = \text{"yes"})$.

(iii) Believed State of the World

The "P vs NP" problem is considered one of the most important problems in mathematics.

The Clay Institute offers a $1M prize for its resolution.

Reductions



A polynomial-time reduction from problem Π_1 to problem Π_2 is useful in 2 situations:

(1) we have a poly-time algorithm for Π_2 and want to obtain a poly-time algorithm for Π_1.

(2) we believe Π_1 is a hard problem and want to deduce that Π_2 is also hard.

Formally, a polynomial-time reduction from Π_1 to Π_2 is a polynomial-time algorithm R such that

- if x is an input to Π_1, then $R(x)$ is an input to Π_2
- $\Pi_1(x) = \text{"yes"} \iff \Pi_2(R(x)) = \text{"yes"}$

↳ And if a poly-time reduction from Π_1 to Π_2 exists, we write $\Pi_1 \leq_p \Pi_2$ (Π_2 is at least as hard as Π_1)

- If $\Pi_1 \leq_p \Pi_2$ and $A(\cdot)$ is a poly-time alg. for Π_2, then $A(R(\cdot))$ is a poly-time alg. for Π_1.
- If there is no poly-time alg. for Π_1 and $\Pi_1 \leq_p \Pi_2$, then there is no poly-time algorithm for Π_2.

Exercise: $\Pi_1 \leq_p \Pi_2$ and $\Pi_2 \leq_p \Pi_3 \Rightarrow \Pi_1 \leq_p \Pi_3$

NP-Completeness

Defn: A problem Π is **NP-hard** if,
$\forall \Pi' \in NP, \quad \Pi' \leq_p \Pi$

Defn: A problem Π is **NP-complete** if,
① $\Pi \in NP$, and
② Π is NP-hard

A problem is NP-hard if it is at least as hard as any problem in NP. It is also NP-complete if it is also in NP (as well as being NP-hard).

We know of numerous NP-complete problems:

- Cook [1971], see below
- Karp [1972] provides a list of 21 NP-complete problems
- Since then the list has grown to contain thousands of NP-complete problems.

It is believed that NP-completeness is one of the biggest exports from Computer Science to the other Sciences and Engineering.

Cook's Theorem

Imagine a circuit made up of 3 types of boolean logic gates: AND, OR, and NOT

fan-in (= # inputs): 2 2 1

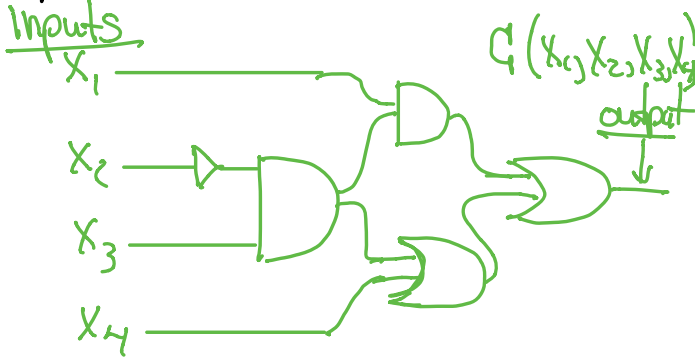
fan-out (= # outputs): unbounded unbounded

(Note: CLRS indicates case of unbounded fan-in for AND & OR)

unbounded ↔ one output with unbounded branching.

Can be represented by graph with nodes labelled AND, OR, & NOT

Inputs and output are binary variables $\{0,1\}$



Assume no feedback, so graph is DAG, and one output.

Circuit-SAT: Given a circuit $C(x_1, \ldots, x_n)$, is there an input for which the output of C is 1?

Theorem [Cook, 1971]: Circuit-SAT is NP-complete.

Proof (some parts sketched):

First, we show that Circuit-SAT $\in$ NP. Given a circuit C' and a proposed satisfying assignment $x_1, \ldots, x_n$, we can check whether $C'(x_1, \ldots, x_n) = 1$ in polynomial time.

It remains to show that Circuit-SAT is NP-hard.

- If we pick any problem $\Pi \in$ NP, we want to show that $\Pi \leq_p$ Circuit-SAT.
- Because $\Pi \in$ NP, $\exists$ poly-time V_Π s.t. for every "yes" input x there exists a certificate y s.t. $V_\Pi(x, y) =$ "yes"
- Need to design reduction from Π to Circuit-SAT.
  - (i) Let x be an input to Π.
  - (ii) Reduction builds circuit C_x that is satisfiable iff $\Pi(x) =$ "yes". $C_x(y)$ is basically an implementation of $V_\Pi(x, y)$.
  - (iii) The technically detailed part of the proof is to show that for any given x, a poly-time alg. $V_\Pi(x, y)$ of y (and of runtime $p_1(|y|)$) can be implemented by a circuit of size $p_2(|y|)$.

Sketch of (iii): On any input y the computation of $V_\Pi(x,y)$ proceeds in $p(|y|)$ steps.

- In every step, the state of the program is a $(|y| + p_3(|y|) + \log p_4(|y|))$-long vector containing input, contents of memory, and program counter.
- The starting state is only the input.
- In going from state t to state $t+1$, the transition is accomplished through a computation that reads parts of the state vector and produces the next state vector. The computation itself can be represented by an AND, OR, NOT circuit
- Output state: Memory location corresponding to $V_\Pi(x,y)$.
- So the computation of $V_\Pi(x,y)$, for any fixed x and variable y, can be implemented as a layered circuit with $p(|y|)$ layers s.t. $C_x(y) = V_\Pi(x,y)$, for all y.
- Thus, C_x is satisfiable iff $\exists y$ s.t. $V_\Pi(x,y) = 1$