NP completeness and Intractability of Problems

Most of the course so far has been focused on good algs.

But it is instructive to also study when we cannot give good algs.

Why?

- No point looking for good algs when they don’t exist.
- Or when you have compelling evidence that they may not exist.

Most of the time

Today we will see what is usually the “compelling evidence” given.

We usually say

“cannot solve problem Q fast unless \( P=NP \)”

So:

- What is \( NP \) and \( P \)
- How would you prove such a result for your favorite problem \( Q \).

Along the way:

- Formal definitions
- Poly time
- Decision problems
- \( P \) and \( NP \)
- Reductions
- \( NP \) completeness
So far we talked about efficient algorithms, and finally today we will talk about limitations.

1. **Polynomial time**

   For problem $Q$, an algorithm runs in polynomial time if there exists a constant $c > 0$ such that for all inputs $I$ to the problem,
   
   if $|I| = \text{number of bit to write down } I$

   then
   
   \[
   \text{runtime of algo on input } I \leq c|I|^c.
   \]

   Hence Dijkstra's algorithm runs in poly-time.

   Why?

   **Input:** write $n$ in binary
   
   $m$
   
   list of $m$ $(u, v)$ pairs
   
   length of each edge
   
   name of source

   $O(ln n)$ bit
   
   $O(ln m)$ bit
   
   $O(m \ln n)$ bit
   
   $O(m \log D)$ bit
   
   $O(ln n)$ bit

   $\Rightarrow$ total input $|I| \leq O(m \ln n + m \log D)$ bits

   **Runtime:** $O(m \ln n)$ operations.

   if the edge lengths $\leq D$

   $\Rightarrow$ maximum distance $\leq nD$

   $\Rightarrow$ all numbers are $\leq O(\ln n + \ln D)$ bits

   $\Rightarrow$ overall runtime $\leq poly(|I|)$
In contrast: here’s an algo that is not in polytime.

\[ \text{Add}(p,q): \]
1. if \( q = 0 \) return \( p \)
else
\[ p \leftarrow p + 1, \quad q \leftarrow q - 1 \]
goto 1

Input length = \( O(\log p + \log q) = \Omega(1) \)

runtime = \( O(q) \) which is exponential in \( \Omega(1) \)

Here’s two less contrived examples:–

1. **Ford Fulkerson (basic version)**

   Input: graph, capacities in \((0, 1, \ldots, C)\), \( s, t \)
   all takes \( O(m(\log n + \log C)) \) bit

   But runtime \( \geq \Theta(mF^+) \) on some instances
   and \( F^+ \) can be as large as \( C \) \( \Rightarrow \) \( \Theta(mC) \)

Example:

![Diagram of network flow problem](image)
However: Edmonds-Karp "Fattest path" has runtime $O(m^2 \log F)$
—" "Shortest path" $O(m^2 n)$

Both are poly time. (to see that $O(m^2 \log F)$ is OK,
note that $F \leq mc$

Hedges max capacity.

\( \underline{2} \) Knapsack Dynamic Program:

- **Input:** \( n \) items with sizes and values, knapsack size \( B \)

\[ O(n(\log S_{\max} + \log V_{\max}) + \log B) \text{ bit} \]

Runtime of DP: at least \( O(nB) \), have to fill in table

but that is exponential in \( |I| \)

- if \( B = \) much larger than \( n \).

Say \( B = 2^m \) then runtime is \( \leq n \cdot 2^m \)

\( S_{\max} = 2^m \)

\( V_{\max} = 2^m \)

DP shows that if numbers are small then knapsack is easy.

Now: we'll show that if numbers are large then
knapsack is "hard".

Well, we will refer to a theorem...
By the way, it's not that we want to just aim for poly time
- an algo that runs in time $n^{100}$ is not useful for any me
  (if $n \geq 2$ say)

better to have an algo that runs in time $2^n$

But: (a) it's a start, once you get $n^{100}$, chances are you can
do better.

For many problems people have beaten the exponent down
quite a bit — at least for problems people care about

(b) It's a low bar. And somehow we say that this
problem has no poly time algo, we're saying
it's not even meeley this low bar.

Ideal situation: problems lie in 2 buckets

Very fast algs
(near linear time)

| Cannot be done
in poly time

but sadly there's stuff there. We hope they
are not intractable problems...
A problem is a decision problem if only allowed answers are YES/NO.

Example: **3-COLORING**
- Input is a graph $G$
- Output = YES if $\exists$ an assignment of 3 colors to vertices of $G$ s.t. adjacent vertices have different colors.
- No otherwise

Correct answer:
- YES

Example: **Bipartite Perfect Matching**
- Input: bipartite graph $G = (L, R, E)$ with $|L| = |R|$
- Output: does $\exists$ a matching of size $|L| = |R|$?
OR: Connectivity  Input: Graph G
       output: is G connected?

OR: KNPACK: Knapsack instance \( B, v_1, \ldots, v_n \)
       and target value \( K \)

output: does \( \exists \) a solution that fits into the knapsack
       and has value \( \geq K \)?

\[ \text{Note: Any optimization problem (Find a blah of max value)} \]
\[ \text{gives rise to a natural decision problem that takes an} \]
\[ \text{additional parameter } K \text{ and asks} \]
\[ \text{"Does there exist a } \text{ blah of value } \geq K?" \]

\[ \text{Fact: The decision problem is no more difficult than the} \]
\[ \text{optimization problem.} \]

\[ \text{Why? If you can solve the optimization problem, you can} \]
\[ \text{solve the decision problem.} \]

\[ \text{(In other words, you can reduce the dec. problem to its} \]
\[ \text{opt. version)} \]
OK, we’ve seen ① poly-time and ② decision problems.

Here’s what we’d ideally like to prove (maybe):

There are problems which we cannot solve in polynomial time.

**Good News**: this can be proved 😊

**Bad News**: not what we’re looking for, anyways 😞

**Theorem**: In fact, there are uncomputable problems.
- The halting problem is not computable at all.
- There are other problems which can be solved in exponential time but provably not solvable in poly-time (Exp-complete problems, for example, cannot be solved in polynomial time, by the time-hierarchy theorem).

But these are not necessarily the problems we wanted to solve, so the answer is not v. satisfying.
The class \( \text{NP} \) (non-deterministic poly-time) versus the class \( \text{P} \) (poly-time).

\( \text{P} \) is easy. It contains all decision problems which have poly-time algorithms.

So \( \text{P} \) contains \text{CONNECTIVITY, BIPARTITE PERFECT MATCHING,} and many other problems we care about.

\( \text{NP} \): it is class of decision problems whose "solutions" can be verified in polynomial time.

Let's be precise now: Fix a problem \( Q \).

Want: I a program called \( V \) (for verifier) that does the following:

- Suppose \( I \) is a \text{YES}-instance of \( Q \).
  That is, you should say \text{YES} when given \( I \) as input.
  Then: I witness \( W \) of poly(\( |I| \)) size such that \( V(I,W) \) says \text{OK} in poly(\( |I| \)) time.

- Suppose \( I \) is a \text{NO}-instance of \( Q \).
  Then: \( \forall W, \; V(I,W) \) says \text{NO} (again in poly time).
That's the definition of a problem being in \( \text{NP} \). Note:-

1. Verifier does not have to find a witness. It just has to verify that a witness \( W \) is "good" for input \( I \).

2. Witness must be short (poly-size)
   Verifier must be fast (poly-time).

---

**Examples:**

1. Suppose \( Q \) is a problem in \( \text{P} \) (has poly-time algo)
   
   then \( Q \in \text{NP} \).

   Why? \( Q \) has a poly-time algo \( A \) that says YES on YES instance.
   NO on NO instance.

   So set \( V(I, W) := A(I) \)
   
   that is ignore "witness".

2. \( \text{3COLORING} \in \text{NP} \).

   **PF:** Witness is a \( 3\)-coloring of \( G \).
   
   Verifier takes this coloring and checks
   
   (a) all vertices colored
   
   (b) no edge has same color on both ends.

   Runs in linear time!
(3) **COMPOSITES** is a decision problem that takes a number $N$ and outputs YES if $N$ is not prime
NO if $N$ is prime.

$\text{COMPOSITES} \in \text{NP}$

**Proof:** witness is a factorization $N = a \cdot b = N$
Verify that $a, b$ integers, not 1.
$a \cdot b = N$.

(4) **PRIMES** is opposite of **COMPOSITE**.

**Thm:** PRIMES $\in$ NP \cite{Pratt}

**Hmm:** How would you quickly prove to me that a number is a prime?
A bit tricky, but can be done.

(5) **Linear Programming (Decision version) $\in$ NP**.

Given an LP and a value $K$, does $\exists$ a solution of value $\geq K$.

**Hey:** witness can be solution $x$ to the LP.

**Q:** Why is witness of size $\text{poly}(\text{size of LP})$?

**Ans:** Can show using fact that solution is a corner.

\[ \text{optimal} \]
Indeed: if solution is a corner, then is intersection of \( n \) constraints where \( n = \# \) of variables.

So \( a_i^T x = b_i \) \( \Rightarrow \) \( n \) vars, \( n \) constraints

\( a_i^T w x = b_i \) \( \rightarrow \) \( \text{some } n \) constraints

call this system \( \hat{A} x = \hat{b} \) \( \Rightarrow \) \( x = \hat{A}^{-1} \hat{b} \)

Thm: Entries here are not too large using Cramer's Rule

Point is: sometimes not completely easy to see that problem \( \in \text{NP} \).

---

**NON-3COLORABLE**: Given \( G \), is \( G \) NOT 3-COLORABLE?

Hmm: how do you give a witness for not being 3COLORABLE

Can list all 3colorings, but too many of them.

In fact **NON-3COLORABLE** believed to \( \not\in \text{NP} \).
Anyway: Many problems we care about are in NP. We can check if a solution is indeed a solution.

The big question of the area is:

\[ P = \text{NP?} \quad \text{or} \quad P \subsetneq \text{NP}. \]

“Is finding a solution more difficult than verifying one?”

Or is it only polynomially harder?
Good. We've tried for a long time to show that $P = NP$ or $P \neq NP$ and failed.

But the general consensus is: $P \neq NP$.

We think the world looks like

![Diagram showing the relationship between P, NP, NP-complete, and other problems]

And we use this as a way to give "compelling evidence" that some statement is true.

Eg: we say statement $X$ is true unless $P = NP$.

"Something is very strange"
or "something unlikely happens"

And the "hardness" proofs we give are all of this form.
We would have loved to say:

My problem $Q$ (which happens to be in $NP$) cannot be solved in poly time

But we currently cannot rule out that all problems in $NP$ can be solved in linear time 😞

So what we say is the following:

Problem $Q$ cannot be solved in poly time unless $P=NP$.

Or rephrasing that a bit:

If problem $Q$ can be solved in poly time

$\Rightarrow$ all problems in $NP$ can be solved in polytime!!

(which is very unlikely, we think)

In the rest of this lecture (and maybe a bit of the next) talk about how to show such a statement.
IV. Reductions (Poly time Reductions specifically).

Two problems, $Q$ and $Q'$, say that $Q \leq_p Q'$ in poly time if:

Given algorithm to solve $Q'$, can solve $Q$ in poly time using polynomial calls to $A'$.

(Assume $A'$ takes 1 unit of time on each call)

Examples: Shortest path reduces to mincost maxflow.

Max finding reduces to sorttrip

Min finding reduces to max finding

Min finding reduces to max flow

Max flow reduces to LP.

Etc.

Denote $Q \leq_p Q'$ "no harder" than poly time
A special kind of Reductions (Karp reductions)

\( \text{Q Karp-reduces to Q' if} \)

There is a polytime computable map \( f : \text{instances of Q} \rightarrow \text{instances of Q} \)

Such that \( I \in \text{YES} \text{ instance of Q} \)

\( \Rightarrow f(I) \text{ is YES instance of Q'} \)

Use this because it keeps the theory cleaner. (can do without, see text)

It's like calling the algorithm \( A' \) for \( Q' \) just once and returning the same answer.

 Alg for Q (I):
   return \( A'(f(I)) \)
OK: NP completeness.

"The hardest problems in NP"

Problem $Q$ is NP complete if

1. $Q$ is in NP

2. $\forall Q' \in \text{NP}, \ Q' \leq_p Q$.

But since $Q_1 \leq_p Q_2$ and $Q_2 \leq_p Q_3 \Rightarrow Q_1 \leq_p Q_3$

We can say:

Problem $Q$ is NP-complete if

1. $Q \in \text{NP}$

2. $\exists$ another NP-complete problem $Q'$ s.t. $Q' \leq Q$

This is what we always use.
OK: So you are worried that your problem is NP-complete? 

1. Is it a decision problem? If not, cannot be in NP. 
   (Type mismatch) 
   Maybe consider its decision version?

2. Show it is in NP. 
   Can find witnesses for YES instances that are easily verified.

3. Find some other NP-complete problem $Q'$ and prove $Q' \leq_p Q$ 

   "If you can solve $Q$ efficiently, you can solve $Q'$ efficiently."

But $Q'$ is NP-complete so this is unlikely.

So $Q$ having efficient also is unlikely.