CSE332: Data Abstractions

Lecture 27: A Few Words on NP

Tyler Robison

Summer 2010

Easily one of the most important questions in Computer Science:

Does P=NP?

- Of course, we need to go into what these terms mean
- P and NP are *classes* of problems
  - P: Class of problems that can be solved in polynomial time
  - NP: Class of problems where an answer can be *verified* in polynomial time
    - We'll get into what that means
- The question is, are these sets equivalent?
  - A question that computer scientists & mathematicians have been grappling with for a long time
  - Most believe that P!=NP, but no one's proven it
  - One such proof recently in the news (P!=NP; probably not valid)

Wow, that’s fantastic… who cares?

- P=NP would mean that many ‘difficult’ problems that could previously only be solved in exponential time could now be solved in polynomial time
  - Some algorithms (such as cryptography) are based around the ‘difficulty’ of brute-forcing it, but the ease of which an answer can be verified
  - You can break many online encryptions now… with enough computing power
    - Say, an enormous # of computers
    - Or one computer running for several centuries
    - And you don’t break the scheme itself; you break it for a single session
  - If P=NP, much of existing cryptography would (in theory) be insecure

Why, cont.

- Proving equivalence (or non equivalence) of two problem classes interesting mathematically
- Proving (or disproving) **P** = **NP** is among the most vexing and important open questions in computer science and probably mathematics
  - A $1M prize, the Turing Award, and eternal fame await
  - Sort of the “Fermat’s Last Theorem” of the CS world (except, this is unsolved)

Topic doesn't really belong in CSE332

- This lecture mentions some highlights of **NP**, the **P** vs. **NP** question, and **NP**-completeness
- It should not be part of CSE332:
  - We don't spend enough time to do it justice
  - To really cover it, a much larger block of time is needed, and after relevant theory background
  - It's not on the final
- But you are all (?) "in transition"
  - Due to recent shifting around of CS curriculum
  - Encourage you to take Algorithms or Theory to learn more
    - Remember the Dijsktra's quote : "computer science is no more about computers than astronomy is about telescopes" – they are quite relevant here
  - Anyway, next academic year, this lecture drops out of CSE332
- And, it's an interesting (& important) problem

P

- **P**: The class of *problems* that can be solved by algorithms running in polynomial time; $O(n^{\mathbf{k}})$ for some constant **k**
  - Note: For purposes of this discussion, consider logn, nlogn, etc. as roughly the same as polynomial: nlogn < n^2, so it's 'about that fast'
    - Contrast with exponential time: very, *very* slow
  - Every problem we have studied is in **P**
    - Examples: Sorting, minimum spanning tree, …
  - Yet many problems don't have efficient algorithms!
  - While we may have been quite concerned with getting sorting down from $O(n^2)$ to O(nlogn), in the grand scheme of things, both are pretty good
    - Really, polynomial time is sufficiently 'quick'
      - Yes, even something insane like $O(n^{24601})$
    - Exponential time is not; very quickly becomes infeasible to solve (precisely, anyway)

NP

- **NP**: The class of *problems* for which polynomial time algorithms exist to check that an answer is correct
  - Given this potential answer, can you *verify* that it's correct in polynomial time?
  - To solve from scratch, we only know algorithms that can do it in exponential time
    - If P=NP, then that would mean we'd have polynomial time algorithms for solving NP problems
  - Ex: We saw Dijkstra's algorithm for finding shortest path in polynomial time
    - For an unweighted graph, finding the *longest path* (that doesn't repeat vertices) is in NP
      - There is a bit more to it than that; need to modify the problem slightly

More NP

- We know **P** $\subseteq$ **NP**
  - That is, if we already know how to solve a problem in polynomial time, we can verify a solution for it in polynomial time too
- NP stands for "non-deterministic polynomial time" for technical reasons
- *Many* details being left out, but this is the gist

- There are many important problems for which:
  - We know they are in **NP** (we can verify solutions in polynomial time)
  - We do not know if they are in **P** (but we *highly* doubt it)
  - The best algorithms we have to solve them are exponential
    - $O(\mathbf{k}^{\mathbf{n}})$ for some constant **k**

NP Example One: Satisfiability

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

- Input: a logic formula of size **m** containing **n** variables
  - Various logical ands, ors, nots, implications, etc.
  - Can assign true or false to each variable, evaluate according to rules, etc.
- Output: An assignment of Boolean values to the variables in the formula such that the formula is true
  - That is, find an assignment for x1, x2, … xn such that the equation is true; if such an assignment exists
- A good problem to solve, in that you can use logic to represent many other problems
  - An older branch of AI looked into encoding an agent's knowledge this way, then reasoning about the world by evaluating expressions

NP Example One: Satisfiability

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

- We can solve it via 'brute-force':
  - Try every possible variable assignment
    - {x1=true,x2=true,…xn=true}, {x1=false…}…
  - How many possibilities do we need to try?
    - n variables, 2 possible values for each, so 2^n possible assignments
  - Not so bad for n=5… looking less bright for n=1,000
- So exponential time to solve by checking all possibilities
- We can verify it quickly though
  - If I give you an assignment {x1=false, x2=….}, you can do it in polynomial time: just evaluate the expression
- If P=NP, a O($\mathbf{m}^k\mathbf{n}^k$) algorithm to solve this exists

NP Example One: Satisfiability

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

- Quite a few NP problems are like this in that they:
  - Are relatively simple to explain
  - Can be solved easily but slowly by brute-force: simply try all possibilities

NP Example Two: Subset sum

14	17	5	2	3	2	6	7	6	17

31?

Input: An *array* of *n* numbers and a target-sum *sum*

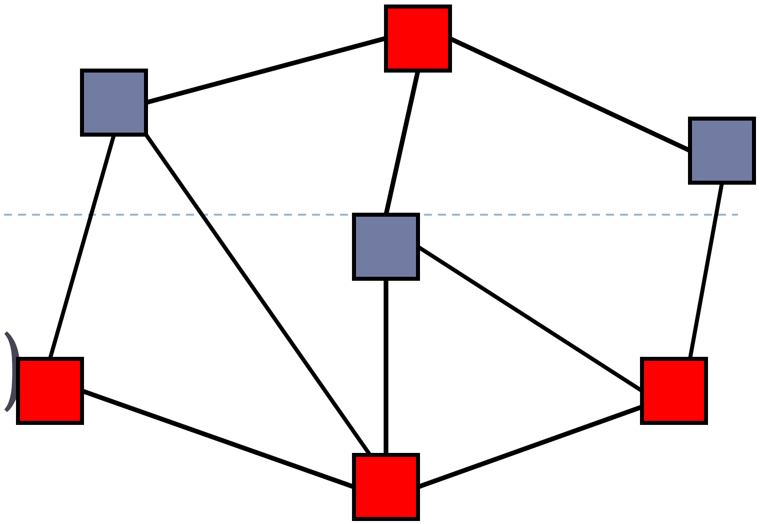
Output: A subset of the numbers that add up to *sum* if one exists

$O(2^n)$ algorithm: Try every subset of array

$O(n^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Given a subset that allegedly adds up to sum, add them up in $O(n)$

NP Example Three: Vertex Cover (modified)



Input: A graph (**V**,**E**) and a number **m**

Output: A subset **S** of **V** such that for every edge (**`u`**,**`v`**) in **E**, at least one of **`u`** or **`v`** is in **S** and **|S|=m** (if such an **S** exists)

That is, every vertex in the graph is 'covered' by being in **S**, or being adjacent to something in **S**, and the size of **S** is m

$O(2^{\mathbf{m}})$ algorithm: Try every subset of vertices of size **m**

$O(m^{\mathbf{k}})$ algorithm: Unknown, probably does not exist

Verifying a solution: See if **S** has size **m** and covers edges

NP Example Four: Traveling Salesman

Input: A complete directed graph (**V**,**E**) and a number **m.**

Say, a graph of cities with edges as travel times

Output: A path that visits each vertex exactly once and has total cost < **m** if one exists

$O(2^{|V|})$ algorithm: Try every valid path including all vertices; pick one of cost **m**

$O(|V|^k)$ algorithm: Unknown, probably does not exist

Verifying a solution: Traverse the graph in that order, keep track of the cost as you go; at the end, compare against **m**

More?

- Thousands of different problems that:
  - Have real applications
  - Nobody has polynomial algorithms for

- Widely believed: None of these problems have polynomial algorithms
  - That is, P!=NP
  - For *optimal* solutions, but some can be *approximated* more efficiently

NP-Completeness

What we have been able to prove is that many problems in **NP** are actually **NP**-complete (one sec for why that's important)

To be NP-complete, needs to have 2 properties:

1. Be in NP (that is, a solution to it can be verified in polynomial time)
2. Be NP-hard: On an intuitive level, being NP-hard means that it is *at least as hard as any other problem in NP*
   - What it boils down to: If we have a polynomial time solution to an NP-hard problem, we can alter it to solve any problem in NP in polynomial time

All four of our examples are **NP**-complete

P=NP ?

- If we gave an algorithm that solved an NP-complete problem in polynomial time, we could then use it to solve ***all*** NP problems in polynomial time
  - Because of our definition of NP-complete
- To show P=NP, you just need to find a polynomial time solution to a single NP-complete problem
  - Or, to show P!=NP, you need to show that no polynomial time algorithm exists for a particular NP problem

Hard problems

There are problems in each of these categories:

- We know how to solve efficiently
- We do not know how to solve efficiently:
  - For example, NP-complete problems
- We know we *cannot* solve efficiently (exponential time): see a Theory course
- We know we cannot solve at all: see CSE311/CSE322
  - The Halting Problem

A key art in computer science:
When handed a problem, figure out which category it is in!