CSE332: Data Abstractions

Lecture 27: A Few Words on NP

Dan Grossman

Spring 2010

This does not 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:
  - 30 minutes can't due this rich and important topic justice
  - It's a major component (approx. 2 weeks) of CSE312
  - It's not on the final
- But in Spring 2010, you are all "in the transition"
  - None of you will take CSE312 because you took CSE321
  - So want to mention what you're missing
  - Encourage you to take CSE421 or CSE431 to learn more
- So, next academic year, this lecture drops out of CSE332

NP

- **P**: The class of *problems* for which polynomial time ($O(n^{\mathbf{k}})$ for some constant **k**) algorithms exist (to solve the problem)
  - Every problem we have studied is in **P**
    - Examples: Sorting, minimum spanning tree, …
  - Many problems don't have efficient algorithms!
    - Misleading to have your instructor pick the problem! ☺
- **NP**: The class of *problems* for which polynomial time algorithms exist to check that an answer is "yes"
- There are many important problems for which:
  - We know they are in **NP**
  - We do not know if they are in **P** (but we *highly* doubt it)
  - The best algorithms we have are exponential
    - $O(\mathbf{k}^{\mathbf{n}})$ for some constant **k**

Outline

- A few example problems
  - Checking a solution vs. finding a solution
- **P** == **NP** ?
- **NP**-completeness
- Why it's called **NP**
- **NP** is not as hard as it gets

Subset sum

14	**17**	5	**2**	**3**	**2**	6	**7**	6	17

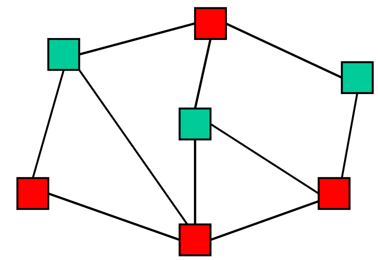
31?

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

$O(2^{\mathbf{n}})$ algorithm: Try every subset of array
$O(n^{\mathbf{k}})$ algorithm: Unknown, probably does not exist

Verifying a solution: Given a subset that allegedly adds up to sum, add them up in $O(n)$
Verifying no solution exists: hard in general as far as we know

Vertex Cover: Optimal

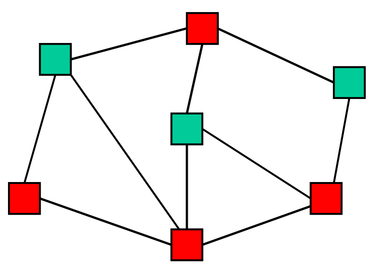


Input: A graph (**V**,**E**)
Output: A minimum size subset **S** of **V** such that for every edge (**u**,**v**) in **E**, at least one of **u** or **v** is in **S**

$O(2^{|\mathbf{V}|})$ algorithm: Try every subset of vertices; pick smallest one
$O(|\mathbf{V}|^{\mathbf{k}})$ algorithm: Unknown, probably does not exist

Verifying a solution:
- Hmm, hard to verify an answer is *optimal* (smalles **|S|**)
- Can recast vertex cover as a *decision problem*

Vertex Cover: Decision Problem



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)

$O(2^{\mathbf{m}})$ algorithm: Try every subset of vertices of size **m**
$O(m^{\mathbf{k}})$ algorithm: Unknown, probably does not exist

Verifying a solution: Easy, see if **S** has size **m** and covers edges

Good enough: Binary search on **m** can solve the original problem

Traveling Salesman

[Like vertex cover, usually interested in the optimal solution, but we can ask a yes/no question and rely on binary search for optimal]

Input: A complete directed graph (**V**,**E**) and a number **m**
Output: A path that visits each vertex exactly once and has total cost < **m** if one exists

$O(2^{|\mathbf{V}|})$ algorithm: Try every subset of vertices; pick smallest one
$O(|\mathbf{V}|^{\mathbf{k}})$ algorithm: Unknown, probably does not exist

Verifying a solution: Easy

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

Output: An assignment of Boolean values to the variables in the formula such that the formula is true

O(**m***2**ⁿ**) algorithm: Try every variable assignment

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

Verifying a solution: Evaluate the formula under the assignment

Outline

- A few example problems
  - Checking a solution vs. finding a solution
- **P** == **NP** ?
- **NP**-completeness
- Why it's called **NP**
- **NP** is not as hard as it gets

More?

- Thousands of different problems that:
  - Have real applications
  - Nobody has polynomial algorithms for
- Widely believed: None of these problems have polynomial algorithms
  - For *optimal* solutions, but some can be *approximated*
- But: Nobody has ever proven that a single problem is:
  - In **NP**: A solution can be verified in polynomial time
  - And not in **P**: Cannot be solved in polynomial time

P==NP ?

- Proving (or disproving) **P** != **NP** is the most vexing and important open question in computer science and probably mathematics
  - A $1M prize, the Turing Award, and eternal fame await
- Clearly **P** $\subseteq$ **NP**
  - If there is a polynomial algorithm, then we can just "verify" a solution exists by running the algorithm
- If **P==NP**, then all sorts of strange things / problems arise
  - Most cryptography would stop working, for example
  - But nobody has been able to prove **P != NP**

NP-Completeness

What we have been able to prove is that many problems in **NP** are actually **NP**-complete:

Definition: A problem is **NP**-complete if the discovery of a polynomial algorithm for it means *every* problem in **NP** has a polynomial-time algorithm, i.e., **P==NP**

All four of our examples are **NP**-complete
- There are thousands more

How do you prove a problem is **NP**-complete?
- Take CSE421

Why it's called NP

- Your instructor finds the "polynomial time to verify a solution" definition of **NP** intuitive

- An equivalent definition (not obvious it's equivalent) is "there exists a polynomial time algorithm if the algorithm is allowed to make correct guesses at every step"
  - This "guessing" is technically non-determinism in the sense you will learn (or have learned) about in CSE322
  - **NP** stands for non-deterministic polynomial time

Hard problems

There are problems in each of these categories:

- We know how to solve efficiently: most of this course
- We do not know how to solve efficiently:
  - For example, NP-complete problems
- We know we cannot solve efficiently: see CSE431
- We know we cannot solve at all: see CSE311/CSE322
  - Canonical example: The halting problem

A key art in computer science:
When handed a problem, figure out which category it is in!
Example: Don't waste time on an algorithm for an intractable problem!