P & NP

CSC 373 - Fall 2015

Reminder

- Assignment 1 due on Friday October 30th, 9:59pm
- "Prof in a prop" costume selection on Piazza
- Test 2 conflict accommodation request by November 6th
  - Email to ins373f@cs.utoronto.ca

So far …

- The course so far: techniques for designing efficient algorithms, e.g. dynamic programming and greedy algorithms
- What happens if you cannot find an efficient algorithm? Is that your fault or the problem's?
- Showing that a problem has an efficient algorithm is relatively, easy. "All' that is needed is to demonstrate an algorithm.

So far … (Cont.)

- Proving that no efficient algorithm exists for a particular problem is difficult. How can we prove the nonexistence of something?
- We will now learn about NP Complete Problems, which provide us with a way to approach this question.

NP-Complete Problems

- This is a very large class of thousands of practical problems for which:
  - it is not known if the problems have "efficient" solutions
  - it is known that if any one of the NP-Complete Problems has an efficient solution then all of the NP-Complete Problems have efficient solutions
  - there is a large body of tools that often permit us to prove when a new problem is NP-complete.

NP-Complete Problems (Cont.)

- Proving that a problem is NP-Complete does not prove that the problem is hard. It does indicate that the problem is very likely to be hard.
- Time permitting, we will also discuss what to do if you find that the problem that you really need to solve is NP-complete (other than giving up).

Content of the Lecture

- Input size of problems
- Optimization problems vs. decision problems
- Polynomial time algorithms, the class P
- The class NP
- Problems in the two classes
- The class Co-NP

Encoding the Inputs of Problems

- In order to formally discuss how hard a problem is, i.e., how much time it requires to solve as a function of its input, we need to be much more formal than before about the input size of a problem.
- Now how can we encode the inputs of the problems?

Encoding Graphs

- A graph G may be represented by its adjacency matrix A = $[a_{ij}]$.
- G can then be encoded as the binary string:

  $a_{11}\ldots a_{n1}a_{21}\ldots a_{2n}\ldots a_{n1}\ldots a_{nn}$

  of length n^2.
- When the binary string is given, the computer can count the number of bits and then determine n, the vertices and edges.
- In general, the inputs of any problem can be encoded as binary strings.

Input size of problems

- **Standard Definition**: The input size of a problem is the minimum number of bits ({0,1}) needed to encode the input of the problem.
  - In most cases we do not need to determine the size exactly.

Input size example: Composite

- **Problem**: Given a positive integer n, are there integers j, k>1 such that n = jk?
- **Question**: What is the input size of the problem?
- **Answer**: Any integer n>0 can be represented in the binary number system as:

$$n = \sum_{i=0}^{k} a_i 2^i \quad \text{where } k = \lceil \log_2(n+1) \rceil - 1$$

- Therefore, a natural measure of input size is $\lceil \log_2(n+1) \rceil$

Input size example: Sorting

- **Problem**: Sort n integers a_1, a_2, …, a_n.
- **Question**: What is the input size of this problem?
- **Solution**: Using the fixed length encoding writes a_i as binary string of length:

$$m = \lceil \log_2 \max(|a_i| + 1) \rceil$$

- This encoding gives input size nm.

Warning

- Running time of algorithms, unless otherwise specified, should be expressed in terms of input size.
- For determining whether n is composite, we need to compare it against the first n-1 numbers. This takes $\Theta(n)$.
- But note that the size of the input is $\log_2 n$. So the number of comparisons performed is actually

$$\Theta(n) = \Theta\left(2^{size(n)}\right)$$

Input size of problems

- Two functions *f(n)* and *g(n)* are of the same type if

$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large n, where a_1, a_2, b_1, b_2, c_1, c_2 are some positive constants.

- For example all polynomials are of the same type, but polynomials and exponentials are of different types.

Input size example: Integer multiplication

- **Problem**: Compute a x b
- **Question**: What is the input size?
- **Solution**: The (minimum) input size is

$$s = \lceil \log_2(a+1) \rceil + \lceil \log_2(b+1) \rceil.$$

- A natural choice is to use

$$t = \log_2 \max(a, b)$$

as the input size since

$$\frac{s}{2} \leq t \leq s.$$

Decision Problems

- **Definition**: A decision problem is a question that has two possible answers, yes or no.
- **Definition**: An optimization problem requires an answer that is an optimal configuration
- **Remark**: An optimization problem usually has a corresponding decision problem
  - Example: MST vs. Decision Spanning Tree (DST)

Decision Problem: MST

- **Optimization problem: Minimum Spanning Tree** Given a weighted graph G, find a minimum spanning tree (MST) of G.
- **Decision problem: Decision Spanning Tree (DST)** Given a weighted graph G and an integer K, does G have a spanning tree of weight at most K?
  - The inputs are of the form (G,K). So we will write $(G,k) \in DST$ or $(G,k) \notin DST$ to denote respectively yes and no answers.

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding simpler decision problem.
- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually be trivial.
- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

Example

- If we know how to solve MST we can solve DST which asks if there is an Spanning Tree with weight at most K. How?
- First solve the MST problem and then check if the MST has cost <= K. If it does, answer Yes. If it doesn't, answer No.

Polynomial-Time Algorithms

- **Definition:** An algorithm is *polynomial-time* if its running time is $O(n^k)$, where 'k' is a constant independent of 'n', and 'n' is the input size of the problem that the algorithm solves.
- Examples:
  - DFS
  - Kruskals's MST

The Class NP

- Usually NP defined in terms of "verifier":
  - Algorithm V that takes two inputs (x,c) such that:
  - For all yes-instances 'x', there is a string 'c' such that V(x,c) outputs True in polynomial time.
  - For all no-instances 'x', for all strings 'c', V(x,c) outputs False.
  - 'c' is called "certificate"
- Notice the symmetry: possible to verify yes-instance in polynomial time, but nothing known about a no-instance

The Class NP (Cont.)

- All decision problems D that can be solved by a "generate-and-verify" algorithm with the following structure:

On input x:

```
generate all “certificates” c, and for each one:

    #verification phase

    if verify (x,c): return true

return false
```

Where the verification phase runs in worst-case polynomial time, as a function of size(x).

- The time require to run the entire algorithm may be exponential.

NP Example

- COMPOSITE (given positive integer x, does x have any factors?) belongs to NP because it is solved by generate-and-verify algorithm where generate phase loops over all integers c in {2,3,...,x-1} and the algorithm is:
- for all integers c = 2,3,...,x-1:

```
        if c divides x:

            return True

    return False  # no divisor found
```

  - verify(x,c): return (c mod x == 0)

NP Example: Vertex Cover

- Input: Undirected graph G, positive integer k
- Question: Does G contain a vertex cover of size k?
- On input <G,k,c>:

    verify that c is a subset of exactly k vertices →O(kn)

    check that c forms a vertex cover in G→ O(mk)

    Output True if both conditions holds; False otherwise

- "Generate and Verify Algorithms"

On input <G,k>:

    For all subsets of vertices c:

        If c is a subset of exactly k vertices and c forms a vertex cover

        Return true

    Return false

Polynomial vs. Non-polynomial Time

- Nonpolynomial-time algorithms are *impractical* . For example, to run an algorithm of time complexity 2^n for n = 100 on a computer which does 1 terraoperation (10^{12} operations) per second
- It takes $2^{100}/10^{12} \sim 10^{18.1}$ seconds ~ 4.10^{10} years

Review: P And NP Summary

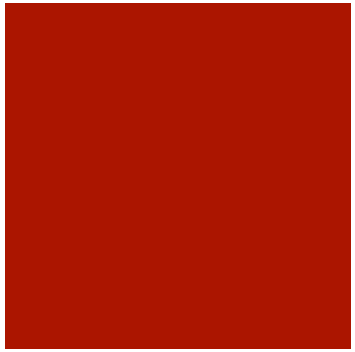
- **P** = set of problems that can be solved in polynomial time
- **NP** = set of problems for which a solution can be verified in polynomial time
  - Examples: Fractional Knapsack,…, Hamiltonian Cycle
- Clearly **P** $\subseteq$ **NP**
- Open question: Does **P** = **NP**?
  - Most suspect not

Amusing analogy
(thanks to lecture notes at University of Utah)

- Students believe that every problem assigned to them is NP-complete in difficulty level, as they have to find the solutions.
- Teaching Assistants, on the other hand, find that their job is only as hard as NP, as they only have to verify the student's answers.
- When some students confound the TAs, even verification becomes hard

Why Care?

NP Contains Lots of Problems We Don't Know to be in P



- Classroom Scheduling
- Packing objects into bins
- Scheduling jobs on machines
- Finding cheap tours visiting a subset of cities
- Allocating variables to registers
- Finding good packet routings in networks

OK, OK, I care...

But where do I begin
if I want to reason about
the P=NP problem?

How can we prove that $NP \subseteq P$?

I would have to show that every set in NP has a polynomial time algorithm…

How do I do that?
It may take a long time!
Also, what if I forgot one of the sets in NP?

We can describe
just one problem L in NP,
such that
if this problem L is in P,
then NP $\subseteq$ P.

It is a problem that can
capture all other problems
in NP.

The “Hardest” Set in NP

Class co-NP

- Note that if $L \in \mathcal{NP}$, there is no guarantee that $\bar{L} \in \mathcal{NP}$ since having certificates for yes-instance does not mean that we have certificates for no-instance.
- The class of decision problems 'L' such that $\bar{L} \in \mathcal{NP}$, is called co-NP.
- So if D in co-NP, there is a verifier V(x,c) running in polynomial time:
  - V(x,c) = False for some 'c' whenever x is a no-instance
  - V(x,c) = True for all c whenever x is a yes-instance

Example

- Composite is in NP so, Prime problem would be in co-NP
- DENSE Problem:
  - Input: Undirected graph G = (V,E), positive integer k.
  - Output: Does every subset of k vertices contain at least one edge between vertices in the subset?

Example (Cont.)

- Dense in co-NP:

On input G,k,c:

    If c is a subset of k vertices and G contains no edge between any two vertices in c

    Return false

Return true

- Verifier runs in polynomial and returns False for some c iff (G,k) not in DENSE

Example (Cont.)

- DENSE is the "complement " of Independent-Set (IS):
  - Input: Undirected graph G, positive integer k.
  - Output: Does G contain some independent set of size k (a subset of vertices with no edge between any two vertices in the subset)