IE418: Integer Programming

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

16th February 2005

The Goal(s) of Computational Complexity

1. Provide a method of quantifying problem difficulty
2. Compare the relative difficulty of two different problems
3. Rigorously define the meaning of an efficient algorithm
4. State that one algorithm for a problem is "better" than another

Computational Complexity

- The ingredients that we need to build a theory of computational complexity for problem classification are the following
  1. A class $\mathcal{C}$ of problems to which the theory applies
  2. A (nonempty) subclass $\mathcal{C}_{\mathcal{E}} \subseteq \mathcal{C}$ of "easy" problems
  3. A (nonempty) subclass $\mathcal{C}_{\mathcal{H}} \subseteq \mathcal{C}$ of "hard" problems
  4. A relation $\lhd$ "not more difficult than" between pairs of problems
- Our goal is just to put some definitions around this machinery
  - Thm: $Q \in \mathcal{C}_{\mathcal{E}}, P \lhd Q \Rightarrow P \in \mathcal{C}_{\mathcal{E}}$
  - Thm: $P \in \mathcal{C}_{\mathcal{H}}, P \lhd Q \Rightarrow Q \in \mathcal{C}_{\mathcal{H}}$

Ingredient #1 — Problem Class $\mathcal{C}$

- The theory we develop applies only to decision problems
- Problems that have a "yes-no" answer.
  - **Opt:** $\max\{c^T x \mid x \in S\}$
  - **Decision:** $\exists x \in S$ such that $c^T x \geq k$?
- Example: The Bin Packing Problem **BPP**
  - We are given a set S of items, each with a specified integral size, and a specified constant C, the size of a bin.
  - **Opt**: Determine the smallest number of subsets into which one can partition S such that the total size of the items in each subset is at most C
  - **Decision**: For a given constant K determine whether S can be partitioned into K subsets such that that the total size of the items in each subset is at most C

Turning **Opt** to **Decision**

- Suppose you know that $l \leq z^* \leq u$, $l, z^*, u \in \mathcal{Z}$
  - z^* is optimal value to **Opt**
- How can you solve **Opt** by solving a series of **Decision** problems?
  - `for (k=l; k<=u; k++) dec(k)`
  - Requires (at most) $u - l + 1$ calls to `dec(k)`
- Better to use *binary search*
  - `1. if(u-l<=1) z = l; exit();`
  - `2. k=(l+u)/2; if (dec(k)==false) l=k; else u=k; goto 1;`
- Requires at most $\log(u - l) + 1$ calls to `dec(k)`
- The log is important—For reason to be explained later.

Measuring the Difficulty of a Problem

- We are interested in knowing the difficulty of a problem, not an instance.
  - Recall: a problem (or model) is an infinite family of instances whose objective function and constraints have a specific structure.

Possible methods of evaluation

1. Empirical
   - Doesn't given us any real guarantee about the difficulty of a given instance
2. Average case running time
   - Difficult to analyze and depends on specifying a probability distribution on the instances.
3. Worst case running time
   - Addresses these problems and is usually easier to analyze.

Comparison of Three Approaches

Empirical	1. Depends on programming language, compiler, etc. 2. Time consuming and expensive 3. Often inconclusive
Average-Case	1. Depends on probability distribution 2. Intricate mathematical analysis 3. No information on distribution of outcomes
Worst-Case	1. Influenced by pathological instances 2. A "pessimistic" view of the world

- The complexity theory we develop is based on a *worst-case* approach.
- Complexity theory is built on a basic set assumptions called the model of computation. Our model of computation is something called a Turing machine
- To deal with this topic in full rigor would require a full semester course—that I couldn't probably teach.

Ingredients #2 and #3

- To define "easy" and "hard", we need to make a few definitions so we can define the running time of an algorithm.
- The running time of an algorithm depends on size of the input. (Duh.)
- A time complexity function specifies, as a function of the problem size, the largest[1] amount of time needed by an algorithm to solve any problem instance.
- How do we measure problem size?
  - The length of the amount of information necessary to represent the problem in a *reasonable* encoding scheme.
  - Example: TSP, N, c_{ij}
  - Example: Knapsack: N, a_j, c_j, b

[1] Here is our "worst case"

What is Reasonable?

- Don't be stupid (pad the input data with unnecessary information)
- Represent numbers in binary notation.
  - That's how computers do it anyway
- An integer $2^n \leq x < 2^{n+1}$ can be represented by a vector $(\delta_0, \delta_1, \ldots, \delta_n)$, where $x = \sum_{i=0}^{n} \delta_i 2^i$
- It requires a *logarithmic* number of bits to represent $x \in \mathbb{Z}$
- Again, we always assume that numbers are *rational*, so they can be encoded with two integers.

- TSP on n cities with costs $c_{ij} \in \mathbb{Z}$, $\max_{i,j} c_{ij} = \theta$, then requires $\leq \log(n) + n^2 \log(\theta)$ bits to represent an instance.

Running Time—Elementary Operations

for $i = 1 \cdots p$ do
  for $j = 1 \cdots q$ do
    $c_{ij} = a_{ij} + b_{ij}$

- How many elementary operations?
- How long does an elementary operation take?
- This may depend on the encoding!
  - All "reasonable" encodings would take at most on the order of $\log \theta$ time, where $\theta = \max_{i,j}\{a_{ij}, b_{ij}\}$
- In what follows, assume all elementary operations (addition, multiplication, comparison, etc.) can be accomplished in constant time.
- Most books (yours too) assume that $\log \theta$ is negligible

Computational Complexity: Big O Notation

- Provides a special way to compare relative sizes of functions
- Big O notation makes use of approximations that highlight large scale differences
    - For purposes of this course, that is all that we can about
- Let f and g be real-valued functions that are defined on the same set of real numbers. Then f **is** $O(g(x))$ if and only if there exist positive *constant* real numbers c and x_0 such that

$$|f(x)| \leq c \cdot |g(x)|, \text{for } x \geq x_0$$

- Is $f(x) = 100x^2 + 3x = O(x^2)$?
- Is $f(x) = 6x^3 = O(x^2)$?
- Is $\log(x) = O(x)$?

Big O Notation—Examples

- We can prove that any polynomial function is "big O" of the polynomial that is its highest term
- Therefore, find orders for the following functions:
    - $f(x) = 7x^5 + 5x^3 - x + 4$
    - $g(x) = \frac{(x-1)(x+1)}{4}$
    - $h(x) = \sum_{i=1}^{x} i$
- $x \neq O(\log x)$
- $2^n \neq O(n^p)$ for any constant p
- Polynomials are "bigger than" logarithms, Exponentials are "bigger than" polynomials

The Limit Trick

If $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$, then $g(x) \neq O(f(x))$

Ready for (Somewhat Formal) Definitions

- Given a problem P, and algorithm A that solves P, and an instance X of problem P.
  - $L(X) \equiv$ The length (in a reasonable encoding) of the instance
  - $f_A(X) \equiv$ the number of elementary calculations required to run algorithm A on instance X.
  - $f_A^*(l) \equiv \sup_X \{f_A(X) : L(X) = l\}$ is the *running time* of algorithm A
- If $f_A^*(l) = O(l^p)$ for some positive constant integer p, A is **polynomial**

Types of Polynomiality

- A is strongly polynomial if $f_A^*(l)$ is bounded by a polynomial function that does not involve the data size (magnitude of numbers).
- A is weakly polynomial if it is polynomial and not strongly polynomial. The l in $O(l^p)$ contains terms involving $log\ \theta$
- An algorithm is said to be an exponential-time algorithm if $f_A^*(l) \neq O(l^p)$, for any p
- Note: If I can solve **dec** if polynomial time, then I can also solve **opt** in polynomial time
  - Why?
  - Also see N&W I.5, Thm: 4.1

I don't think *Polynomiality* is a word

One Last Type of Polynomiality

- A *pseudopolynomial algorithm* A is one that is polynomial in the length of the data when encoded in *unary*.
  - *Unary* means that we are using a one-symbol alphabet. (not binary)
- Practically, it means that A is polynomial in the parameters and the magnitude of the instance data θ—*not* $\log \theta$.
- Example: The Integer Knapsack Problem
  - There is an $O(Nb)$ algorithm for this problem, where N is the number of items and b is the size of the knapsack.
  - This is not a polynomial-time algorithm
  - If b is bounded by a polynomial function of n, then it is

Knapsack In More Detail

- **Knapsack**: N, a_j, c_j, b
- For an instance of **Knapsack** X, what is the length of the input L(X)?
- What are the numbers c_j, a_j, b? Assume they are *rational*.
  - So they can be expressed as the ratio of two integers.
  - Assume $a_j \leq b$
  - $\theta = \max_{j \in N} c_j$
  - $L(X) = \log N + (2N + 2) \log b + 2N \log \theta$
- Is $Nb = O(L(X))$?
  - $\exists p \in \mathbb{Z}$ such that $Nb \leq ((2N + 1) \log b)^p$?
  - **No!**
  - Note if Nb replaced by $N \log b$, then **Yes!**

The problem class $\mathcal{NP}$

- $\mathcal{NP} \neq$ "Non-polynomial"
- $\mathcal{NP} \equiv$ the class of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.
- What the Heck!!?!?!?!?!?!?!?!?
- $\mathcal{NP} \approx$ the class of decision problems with the property that for every instance for which the answer is "yes", there is a short certificate
- The certificate is your "proof" that what you are telling me is the truth

$\mathcal{NP}$: Examples

Example: 0-1IP

$\exists x \in \mathbb{B}^n$ such that $Ax \leq b, c^T x \geq K$?

1. You say the answer is "Yes". I say "prove it."
2. You give me the vector x: This is a "short certificate"
3. The 0-1 vector x can be checked such that $Ax \leq b, c^T x \geq K$?

Example: Optimization

Is the optimal solution z^* to P such that $z^* \geq k$?

- This is *not* necessarily in NP
- Just because the **dec** $\in \mathcal{NP}$ does not imply **opt** $\in \mathcal{NP}$

The Class co-$\mathcal{NP}$

- The class of problems for which the "complement" problem to P is $\in \mathcal{NP}$
- co-$\mathcal{NP}$ $\approx$ the class of decision problems with the property that for every instance for which the answer is "no", there is a short certficate

Example: 0-1IP

$\not\exists x \in \mathbb{B}^n$ such that $Ax \leq b, c^T x \geq K$?

1. You say "no." I say "prove it."
2. You give me what? Is this a short (polynomial length) certificate?

co-$\mathcal{NP}$, More examples

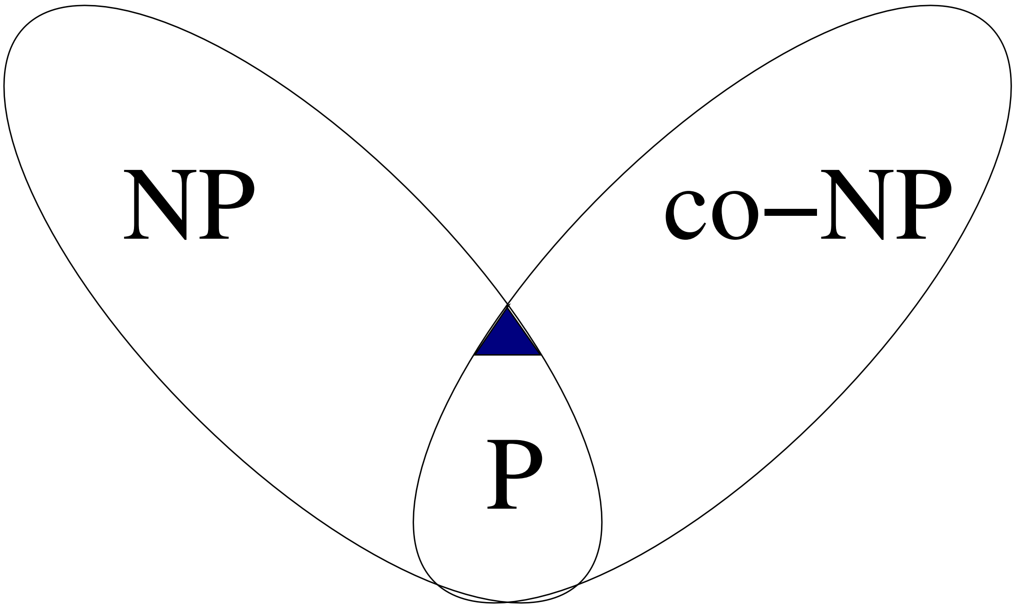
LP

$\not\exists x \in \mathbb{R}^n_+$ such that $Ax \leq b, c^T x \geq K$?

1. You say "no." I say "prove it."
2. You give me What?.
3. Hint: (x, π) is optimal if and only if $Ax \leq b, x \geq 0, \pi^T A \geq c, \pi \geq 0, c^T x = b^T \pi$
4. $\pi \in \mathbb{R}^m \mid \pi^T A \geq c, \pi \geq 0, \pi^T b < K \Rightarrow \not{e}xistsx \in \mathbb{R}^n \mid Ax \leq b, x \geq 0, c^T x \geq K$
5. Is π a short certificate?

The Class $\mathcal{P}$

- $\mathcal{P}$ is the class of problems for which there exists a polynomial algorithm.
- $\mathcal{P} \in \mathcal{NP} \cap$ co-$\mathcal{NP}$



Problems Solvable in Polynomial Time

- Shortest path problem with nonnegative weights: $O(m^2)$
  - Note that the number of operations is independent of the magnitude of the edge weights **Strongly Polynomial**
- Solving systems of equations: $O(n^3)$
  - The magnitude of the numbers that occur is bounded by the largest determinant of any square submatrix of (A, b).
  - Since det A involves $n! < n^n$ terms, this l¡argest number is bounded by $(n\theta)^n$, where θ is the largest entry of (A, b).
  - This means that the size of their representation is bounded by a polynomial function of n and $\log \theta$.
    - $\log((n\theta)^n) = n \log(n\theta)$: Polynomial in the size of the input
- Assignment Problem: $O(nm + n^2 logn)$, $O(\sqrt{n}m \log(nC))$

Recap

- We have our class(es) of problems $\mathcal{P}, \mathcal{NP}$, co-$\mathcal{NP}$
- We know class of "easy" problems. (Problems in $\mathcal{P}$)
- We need our last ingredient
- The relation "not (significantly) more difficult than" ($\lhd$)
  - For this we need the concept of problem reductions.
- Next time: Polynomial reductions
- The class $\mathcal{NPC}$
- A couple sample reductions
- The end of computational complexity
- Read N&W: I.5.1—I.5.6