Computational Complexity

IE 496 Lecture 6

Dr. Ted Ralphs

Reading for This Lecture

- N&W Sections I.5.1 and I.5.2
- Wolsey Chapter 6
- Kozen Lectures 21-25

Introduction to Computational Complexity

- What is the goal of computational complexity theory?
  - To provide a method of quantifying problem difficulty in an absolute sense.
  - To provide a method comparing the relative difficulty of two different problems.
- We would like to be able to rigorously define the meaning of *efficient algorithm*.
- Complexity theory is built on the basic set of assumptions set forth by the *model of computation*.

Problems and Instances

- What is the difference between a *problem* and a *problem instance*?
- To define these terms rigorously takes a great deal of mathematical machinery.
- We will do so only within the context of mathematical programming.
  - Loosely, a *problem* or *model* is an infinite family of *instances* whose objective function and constraints have a specific structure.
  - An instance is obtained by specifying values for the various problem parameters.
- Recall the distinction between *model* and *data* in AMPL.

Running Time and Complexity

- Running time is a measure of the efficiency of an algorithm.
- Computational complexity is a measure of the difficulty of a problem.
- The computational complexity of a problem is the running time of the best possible algorithm.
- In most cases, we cannot prove that the best known algorithm is the also the best possible algorithm.
- We can therefore only provide an upper bound on the computational complexity in most cases.
- That is why complexity is usually expressed using “big O” notation.
- A case in which we know the exact complexity is comparison-based sorting, but this is unusual.

Comparing Algorithms

- So far, we have defined complexity as a tool for comparing the difficulty of two different problems.
- This machinery can also be used to compare two algorithms for the same problem.
- In this way, we can judge whether one algorithm is "better" than another one.
- Note that worst case analysis is far from perfect for this job.
- The simplex algorithm has an exponential worst case running time, but does extremely well in practice.

Polynomial Time Algorithms

- Algorithms whose running time is bounded by a polynomial function are called *polynomial time algorithms*.
- For the purposes of this class, we will call an algorithm *efficient* if it is polynomial time.
- Problems for which a polynomial time algorithm exists are called *polynomially solvable*.
- The class of all problems which are known to be polynomially solvable occupies a special place in optimization theory.
- For most interesting problems, it is not known whether or not there is a polynomial algorithm.
- This is one of the great unsolved problems in mathematics.
- If you can solve it, the American Mathematical Society will give you one million dollars and you will become instantly famous.
- We'll come back to this.

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.
- Solving a system of equations: $O(n^3)$.
  - Note that 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 largest 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 $\log\theta$.
- Minimum weight spanning tree problem: $O(\min\{m \log n, m + n \log n\})$
- Assignment Problem: $O(\min\{n(m + n \log n), n^3\})$

The Case of Linear Programming

- General linear programming is polynomially solvable.
- Note, however, that the simplex algorithm is *not* polynomial time!
- In practice, the expected running time *is* polynomial.
- A polynomial-time algorithm (the ellipsoid method) for LP was not found until 1979!
- Although this algorithm has not had a big practical impact, it's theoretical impact has been large.
- This is one of the biggest cases against using worst-case analysis.

Pseudopolynomial Time Algorithms

- A *pseudopolynomial algorithm* 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.
- Hence, to store an integer k, we would need k symbols.
- 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 in general.
  - If b is bounded by a polynomial function of n, then it is.
  - However, it is pseudopolynomial.

Certificates

- Suppose you had the optimal solution LP and wanted to prove to someone else it was optimal.
- You could simply produce the primal and dual solutions.
- Can optimality be verified in polynomial time?
  - In $O(mn)$ operations, one could verify optimality.
  - However, what is the magnitude of the numbers?
  - They are the ratio of two integers, each of which can be represented in a size that is polynomially bounded.
- Information that can be used to check the output of an algorithm for correctness in polynomial time is called a *certificate*.
- If a binary string has a size polynomial in the length of the input, then it is said to be *short*.
- Obviously, a certificate must be short.

Importance of Certificates

- Every polynomially solvable problem has a certificate.
- It is not known whether every problem with a certificate is polynomially solvable.
- Until 1979, linear programming was one problem with a certificate that was not known to be polynomially solvable.

Problem Reductions

- Suppose we are given two problems X_1 and X_2.
- We want to show that if we solve one, we can also solve the other.
- We say X_1 is polynomially reducible to X_2 if
  1. there is an algorithm for X_1 that uses the algorithm for X_2 as a subroutine, and
  2. the algorithm runs in polynomial time under the assumption that the subroutine runs in constant time.
- This implies immediately that if X_2 is polynomially solvable and X_1 is polynomially reducible to X_2, then X_1 is polynomially solvable.

Decision Problems

- A *decision problem* or *feasibility problem* is a problem for which the answer is either yes or no.
- For technical reasons, most of complexity theory is defined in terms of decision problems.
- Any optimization problem is polynomially reducible to a decision problem (why?).
- Example: The Bin Packing Problem
  - We are given a set S of items, each with a specified integral size, and a specified constant C, the size of a *bin*.
  - Optimization problem: 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 problem: 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.

Feasibility and Polynomial Transformation

- A decision problem will be defined rigorously as a pair (D, F) of sets of binary strings such that $F \subseteq D$.
- The members of D are the instances and the members of F are the feasible instances.
- Given $d \in D$, we must decide whether $d \in F$.
- Suppose we have two problems $X_1 = (D_1, F_1)$ and $X_2 = (D_2, F_2)$.
- In addition, we have a function $g : D_1 \rightarrow D_2$ such that
  - For $d \in D_1$, $g(d) \in F_2 \Leftrightarrow d \in F_1$.
  - $g(d)$ is computable in time polynomial in the length of d.
- In this case, we say that X_1 *is polynomially transformable to* X_2.

Certificates for Decision Problems

- A *certificate of feasibility* for a decision problem is information that can be used to verify a “yes” answer in polynomial time.
- If such a certificate exists, it must be short.
- (Imperfect) Example: The meeting room problem
  - Decision: Is there anyone in this room that I don’t know?
  - There is a certificate for this problem. What is it?
- Example: General integer programming
  - What is the decision version of this problem
  - Is there a certificate of feasibility for this problem?

Nondeterministic Algorithms

- This concept is a little hard to fathom at first, so be Zen...
- A nondeterministic algorithm works on a feasibility problem (D, F) as follows.
- The input to the algorithm is an instance $d \in D$.
- The algorithm has two stages
  - Guessing Stage: Randomly guess a string Q.
  - Checking Stage: Check whether Q can be used to verify the feasibility of d. If so, output $d \in F$. If not, there is no output.
- There are two properties required.
  - We require that if $d \in F$, then there must exist a certificate Q_d that verifies the feasibility of d.
  - The running time of the algorithm is the time it takes to check a certificate that verifies feasibility.

Nondeterministic Polynomial Time Algorithms

- These algorithms are called nondeterministic because the guessing stage is random.
- A nondeterministic polynomial time algorithm is one for which the running time is a polynomial in the size of the input.
- The class of problems for which there exists a nondeterministic polynomial time algorithm is denoted $\mathcal{NP}$.
- The essential property of problems in $\mathcal{NP}$ is that for every feasible instance, there exists a certificate that can be checked in polynomial time.
- Examples of problems in $\mathcal{NP}$.
  - General integer programming feasibility.
  - The decision version of bin packing.
- General integer programming *infeasibility* is not in $\mathcal{NP}$.

$\mathcal{P}$, $\mathcal{NP}$, and $co\mathcal{NP}$

- The class of problem for which *deterministic* polynomial-time algorithms exist is denoted $\mathcal{P}$.
- Obviously, $\mathcal{P}$ is a subset of $\mathcal{NP}$.
- It is not known whether $\mathcal{P} = \mathcal{NP}$ (the million dollar question).
- $co\mathcal{NP}$ is the class of problems for which the complement is in $\mathcal{NP}$.
- In other words, it is the class of decision problem for which there is a certificate verifying a no answer.
- $\mathcal{P}$ is also a subset of $co\mathcal{NP}$.
- If the decision version of an optimization problem is in $\mathcal{NP} \cap co\mathcal{NP}$, then there exists a certificate of optimality.
- It is unlikely that there exist many problems in $\mathcal{NP} \cap co\mathcal{NP}$ that are not also in $\mathcal{P}$.

The Class $\mathcal{NPC}$

- It is interesting to ask what are the hardest problems in $\mathcal{NP}$?
- We say that a problem X is in the class $\mathcal{NPC}$ if every problem in $\mathcal{NP}$ is polynomially reducible to X.
- Surprisingly, such problems exist!
- Even more surprisingly, this class contains almost every interesting integer programming problem that is not known to be in $\mathcal{P}$!

Proposition 1. *If $X \in \mathcal{NPC}$, then $X \in \mathcal{P} \Leftrightarrow \mathcal{P} = \mathcal{NP}$.*

Proposition 2. *If $X_1 \in \mathcal{NPC}$ and X_1 is polynomially reducible to X_2, then $X_2 \in \mathcal{NPC}$.*

The Satisfiability Problem

- This is the first problem proven to be $\mathcal{NP}$-complete.
- The problem is described by
  1. a finite set $N = \{1, \ldots, n\}$ (the *literals*), and
  2. m pairs of subsets of N, $C_i = (C_i^+, C_i^-)$ (the *clauses*).
- An instance is feasible if the set

$$\left\{ x \in \mathbb{B}^n \mid \sum_{j \in C_i^+} x_j + \sum_{j \in C_i^-} (1 - x_j) \geq 1 \text{ for } i = 1, \ldots, m \right\}$$

  is nonempty.
- This problem is obviously in $\mathcal{NP}$ (why?).
- In 1971, Cook defined the class $\mathcal{NP}$ and showed that satisfiability was $\mathcal{NP}$-complete, even if each clause only contains three literals.
- The proof is beyond the scope of this course.

Proving $\mathcal{NP}$-completeness

- After satisfiability was proven to be $\mathcal{NP}$-complete, it was easy to prove many other problems $\mathcal{NP}$-complete.
- This is done by polynomial reduction.
- Example: The k-Clique Problem
  - Does a given graph have a clique of size k?
  - Although it seems simple, this problem is $\mathcal{NP}$-complete.
  - This problem is easily shown to be in $\mathcal{NP}$.
  - To prove it is in $\mathcal{NP}$-complete, we reduce 3-satisfiability to it.

The Line Between $\mathcal{P}$ and $\mathcal{NP}$-complete

- Generally speaking, most interesting problems are either known to be in $\mathcal{P}$ or are $\mathcal{NP}$-complete.
  - The problems known to be in $\mathcal{P}$ are generally "easy" to solve.
  - The problems in $\mathcal{NPC}$ are generally "hard" to solve.
- This is very intriguing!
- The line between these two classes is also very thin!
  - Consider a 0-1 matrix A, an cost vector $c \in \mathbb{Z}^n$, $z \in \mathbb{Z}$ defining the decision problem

$$\{x \in \mathbb{B}^n \mid Ax \leq 1, cx \geq z\}$$

  - If we limit the number of nonzero entries in each column to 2, then this problem is known to be in $\mathcal{P}$ (what is it?).
  - If we allow the number of nonzero entries in each column to be three, then this problem is $\mathcal{NP}$-complete!

$\mathcal{NP}$-hard Problems

- The class $\mathcal{NP}$-hard extends $\mathcal{NP}$-complete to include problems that are not in $\mathcal{NP}$.
- If $X_1 \in \mathcal{NPC}$ and X_1 reduces to X_2, then X_2 is said to be $\mathcal{NP}$-hard.
- Thus, all $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard.
- The primary reason for this definition is so we can classify optimization problems that are not in $\mathcal{NP}$.
- It is common for people to refer to optimization problems as being $\mathcal{NP}$-complete, but this is technically incorrect.

Karp's 21 $\mathcal{NP}$-complete Problems

SAT is the original NP-complete problem to which all others can be reduced. From there, came Karp's original list of 21 $\mathcal{NP}$-complete problems.

- 0-1 IP
- Clique $\Rightarrow$ Set packing, Vertex covering $\Rightarrow$ Set covering, Feedback node/arc set, Directed Hamiltonian cycle $\Rightarrow$ Undirected Hamiltonian cycle
- 3-SAT $\Rightarrow$ Graph coloring $\Rightarrow$ Clique cover, Set Partitioning $\Rightarrow$ Hitting set, Steiner tree, 3-D matching, Knapsack $\Rightarrow$ Job sequencing, Partition $\Rightarrow$ Max cut

Theory versus Practice

- In practice, it is true that most problem known to be in $\mathcal{P}$ are "easy" to solve.
- This is because most known polynomial time algorithms are of relatively low order.
- It seems very unlikely that $\mathcal{P} = \mathcal{NP}$.
- If so, the reduction is likely to be prohibitively expensive.
- For similar reasons, although all $\mathcal{NP}$-complete problems are "equivalent" in theory, they are not in practice.
- TSP vs. QAP

Final Notes

- Note that the material in this lecture assumes a sequential model of computation.
- We can ignore most details of the model of computation if we are only interested in separating the complexity class of a problem.
- For parallel algorithms, the situation is much more difficult.
- In theory, we could apply the same framework.
- However, the details of the model of computation become much more important.