Integer Programming
ISE 418

Lecture 25

Dr. Ted Ralphs

Reading for This Lecture

- N&W Sections I.5.1 and I.5.2
- Wolsey Chapter 6
- "Algorithms in C++," R. Sedgewick

Introduction to Computational Complexity

- What is the goal of computational complexity theory?
  - To provide a method of quantifying problem difficulty and comparing the relative difficult of different problems.
  - To provide a method of comparing the relative efficiency of different algorithms for the same problem.
- The theory provides us with a rigorous notion of an *efficiently solvable problem* and an *efficient algorithm*.
- It is built on a basic set assumptions called the *model of computation*.
- We will not concern ourselves too much with the details of a particular model here.
- To deal with this topic in full rigor would require a full semester course.

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 do so here in the context of mathematical programming.
  - Loosely, a *problem* or *model* is a (typically) 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.
- This is similar to the distinction between *model* and *data* in a modeling language.

Measuring the Difficulty of an Instance

- In some sense, the difficulty of a problem instance is easy to measure.
- We solve the problem instance and see how long it takes (*the running time*).
- Note that this inherently depends on the algorithm and the computing platform.
- We want a measure independent of both these variables, however.
  - We cannot simply use the "real" time of execution on a real computer.
  - For now, assume a fixed (typically deterministic) algorithm is used.
  - Execution time is then the total number of elementary operations executed on a theoretical model of a computer called a *Turing machine*.

Turing Machines

- The complexity framework we use today is based on the concept of a *Turing machine* proposed by A.M. Turing.
- A Turing machine is what we would think of today as *algorithm* or, more specifically, a *computer program*.
- Corresponding to any given algorithm, there is a Turing machine that takes an input and produces an output through a sequence of steps.
- The specific sequence might be different for different inputs, i.e., we have a notion of *conditional branching*.
- Turing later conceived of something like what we think of as a *computer*, which was able to load a "program" into its memory.
- This became known as a *universal Turing machine*.
- These concepts heavily influenced the inventions that lead to modern computer architectures.

Elementary Operations

- *Elementary operations* are very loosely defined to be additions, subtractions, multiplications, comparisons, etc.
- In most cases, we will assume that each of these can be performed in constant time.
- Again, this is a good assumption as long as the size of the numbers remains "small" as the calculation progresses.
- Generally we will want to ensure that the numbers can be encoded in a size polynomial in the size of the input.
- This justifies our assumption about constant time operations.
- We will see later, we may have to be very careful about checking this assumption.

Measuring the Difficulty of a Problem

- On the previous slide, we discussed how to measure the efficiency with which an instance can be solved with a given algorithm.
- The difficulty of an entire family of instances problem is a more difficult concept to define.
  - We must consider all possible algorithms for solving the problem.
  - We must also consider the performance of the algorithm on all possible instances.
- We first derive a method of summarizing the efficiency of a single fixed algorithm across a family of instances.
- Ideally, we would like to do this in a way that allows comparison to other algorithms.
- To evaluate that difficulty of a problem, we then use this method of summarization to choose the "best" algorithm.
- The difficulty of the family of instances is then taken to be the efficiency with which instances can be solved using this "best" algorithm.

Running Time Functions

- Deriving a method of summarizing running time across an entire family of instances involves several logical steps.
- First, we observe that it makes little sense to compare running times across all possible instances simultaneously.
  - The running time depends inherently on certain properties of the instances that are independent of a particular algorithm.
  - We therefore first divide the instances into sets of instances that are "comparable."
  - The most obvious universal property on which running time depends and which is measurable for all instances across all problems is *size* (defined next).
- Second, we must choose a *summary statistic* to represent the performance of a given algorithm across a given set of instances
- Finally, we produce a *running time function* that describes the dependence of the summary statistic on the chosen property of the instance.

The Size of a Problem

- The time needed to solve a problem instance with a given algorithm depends on certain properties of the instance.
- The most obvious universal property on which running time depends and which is measurable for all instances across all problems is "*size*."
- What exactly do we mean by "size?"
- In many cases, the *size* of an instance can be taken to be the number of input parameters.
- For a general MILP, this would be roughly determined by the number of variables and constraints.
- The running time of certain algorithms, however, depends explicitly on the *magnitude* of the input data.

Measuring the Size of a Problem

- We will define the *size* of an instance to be the amount of information required to represent the instance.
- This is still not a clear definition because it depends on both
  - our formulation of the problem (in the case of mathematical optimization) and
  - our method of representing the data in memory (the *alphabet*).
- Because computers store numbers in binary format, we use the size of a *binary* encoding (a two symbol alphabet) as our standard measure.
- In other words, the size of a number l is the number of bits required to represent it in binary, i.e., $\log_2 l$.
- As long as the magnitude of the input data is bounded, this is equivalent to considering the number of input parameters.
- In practice, the magnitude of the input data is *usually*, but not always, bounded.

More on the Size of a Problem

- Note that many combinatorial problems are defined *implicitly*, i.e., independent of a particular formulation.
- An example of this is the Euclidean Traveling Salesman Problem.
- The input data for an instance of the TSP is simply the coordinates of each customer location.
- Hence, the size of an instance is determined by the number of locations (assuming the magnitude of the coordinates is bounded).
- If we formulate the TSP as an integer program, the size of the input to the solver will be larger.
- This is because it includes a distance matrix that can be computed from the coordinates.

Summary Statistics

- There are a few obvious ways in which performance can be summarized across a set of instances.
  - Best case running time
  - Average case running time
  - Worst case running time
- Best case doesn't given us any useful guarantee about the difficulty of a randomly selected instance in the family.
- Average case is challenging to analyze and depends on specifying a probability distribution on the instances.
- Worst case addresses these problems and is usually easier to analyze.
- As such, our running time function will consider the worst case (theoretical running time of an instance as a function of its size.

Asymptotic Analysis

- So far, we have determined that our measure of running time will be a function of instance size (a positive integer).
- Determining the exact function is still problematic at best.
- We will only really be interested in approximately how quickly the function grows "in the limit".
- To determine this, we will use *asymptotic analysis*.
- Some notation
  - Running time function names are usually f, g, or T.
  - At risk of confusion, we denote the input size to be n (a value in $\mathbb{N}_+$).
  - We also use m as a variable taking on values in $\mathbb{N}_+$.
  - We use a, b, and c to denote constants taking values in $\mathbb{N}_+$.
  - Although it is common practice, I will try not to refer to a function by the notation "$f(n)$" because $f(n)$ is a value, not a function.
    * Correct: "f is a polynomial function."
    * Incorrect: "$f(n)$ is a polynomial function."

Comparing Functions

- We compare running time functions based on *asymptotic growth rates* of the running times.
- Consider algorithm A with running time given by f and algorithm B with running time given by g.
- We are interested in knowing

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}$$

- What are the four possibilities?

Θ Notation

- We now define the set

$$\Theta(g) = \{f : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \; \forall n \geq n_0\}$$

- If $f \in \Theta(g)$, then we say that f and g *grow at the same rate* or that they are *of the same order*.
- Note that

$$f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$$

- We also know that if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant c, then $f \in \Theta(g)$.
- If the limit doesn't exist, we don't know.

Big-O Notation

- Similarly, we can define the set

$$O(g) = \{f : \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

- If $f \in O(g)$, then we say that "f is big-O of g" or that g *grows at least as fast as* f.
- Note that if $f \in O(g)$, then either $f \in \Theta(g)$ or $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Some other notation
  - $f \in \Omega(g) \Leftrightarrow g \in O(f)$.
  - $f \in o(g) \Leftrightarrow f \in O(g) \setminus \Theta(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
  - $f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

Comparing Functions

- The notation we have just defines gives us a way of ordering functions.
- We can can interpret
  - $f \in O(g)$ as "$f \leq g$,"
  - $f \in \Omega(g)$ as "$f \geq g$,"
  - $f \in o(g)$ as "$f < g$,"
  - $f \in \omega(g)$ as "$f > g$," and
  - $f \in \Theta(g)$ as "$f = g$."
- This gives us a method for comparing algorithms based on their running time functions.
- Note that most of the relational properties of real numbers (transitivity, reflexivity, symmetry) work here also.
- However, not every pair of functions is comparable.

Order Relations

- For polynomials, the order relation from the previous slide can be used to divide the set of functions into equivalence classes.
- We will only be concerned with what equivalence class the function belongs to.
- Note that class membership is invariant under multiplication by scalars and addition of "low-order" terms.
- For polynomials, the class is determined by the largest exponent on any of the variables.
- For example, all functions of the form $f(n) = an^2 + bn + c$ are $O(n^2)$.

Running Time and Problem Complexity

- The running time function 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 function associated with 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

- Re call that complexity analysis is a tool both for
  - comparing the difficulty of two different problems and
  - for comparing two algorithms for the same problem.
- Using this analysis, 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.

Aside: Space Complexity

- So far, we have discussed only the amount of computing time required to solve a problem.
- The amount of memory required to execute a given algorithm may also be an issue.
- This is known as *space complexity*.
- We can analyze space complexity in an analogous manner.

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 non-negative 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.