Computational Optimization
ISE 407

Lecture 8

Dr. Ted Ralphs

Reading for this Lecture

- "Computers and Intractability: A Guide to the Theory of NP-Completeness," Garey and Johnson.
- Grötschel, Lovász, and Schrijver, Chapter 1.
- "Computational Complexity: A Modern Approach", S. Arora and B.Barak.

The Complexity of a Problem

- We have seen how to measure the difficulty of an instance by calculating its running time.
- We have also seen how to measure the complexity of an algorithm by computing its running time function.
- How do we measure the difficulty of a problem?
- We must consider all possible algorithms for solving the problem.
- The formal *time complexity* of a problem specified by language L is the running time function of the "best" algorithm.
- Although the set of all possible running time functions is not well-ordered, we'll nevertheless use this concept to divide problems into classes.

Complexity Classes

- Based on the concepts just discussed, the running time function can be used to separate problems into equivalence classes.
- If we do this using the notion of "polynomial equivalence," we obtain the classes described in the classical theory of NP-completeness.
- This is the scheme used in the vast majority of the literature on mathematical optimization.
- Determining the class to which a problem belongs can be done by determining the running time function or by what we call *reduction*.

Decision Problems and Optimization Problems

- A *decision problem* or *feasibility problem* is a problem for which the answer is either yes or no.
- For primarily historical reasons, complexity theory is defined in terms of decision problems.
- Any optimization problem can be solved by a sequence of decision problems (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.

Formally

- The formal model of computation underlying the original NP-completeness theory is the *deterministic* Turing machine (DTM).
  - A DTM specifies an *algorithm* for computing a Boolean function.
  - The DTM executes a program, reading the input from a *tape*.
  - We equate a given DTM with the program it executes.
  - The output is `YES` or `NO`.
  - A `YES` answer is returned if the machine reaches an *accepting state*.
- As before, a problem is specified in the form of a *language* (subset of the possible inputs over a given *alphabet* (Γ) that yield output `YES`).
- A DTM that produces the correct output for inputs w.r.t. a given language is said to *recognize the language*.
- Informally, we can then say that the DTM represents an "algorithm that solves the given problem correctly."
- Although the original theory used the DTM as the model of computation, we could equivalently use the RAM introduced in the previous lecture.

Polynomially Solvable Problems

- The first basic class of problem we'll define are those for which there exists an algorithm with a worst-case running time function that is a polynomial.
- The class of all such problems is denoted simply by P.
- Polynomial time algorithms are the fundamental building blocks from which we will build up other classes of problems in a recursive fashion.
- For reasons that will become clear, we tend to divide the world of problem into those that are known to be in P and "everything else."

Reduction in Terms of Oracles (Cook Reduction)

- Suppose we are given two problems P_1 and P_2, specified by languages L_1 and L_2.
- We want to show that if we can solve P_2 (recognize L_2), we can also solve P_1 (recognize L_1).
- We say P_1 is *polynomially reducible* to P_2 if
  1. there is an algorithm for P_1 that uses the algorithm for P_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 P_2 is polynomially solvable and P_1 is polynomially reducible to P_2, then P_1 is polynomially solvable.
- A subroutine that we assume runs in constant time for the purpose of doing a reduction is called an *oracle*.
- The reduction described here was first introduced by Cook (1971) in his seminal work "On the complexity of theorem-proving procedures."

Reduction in Terms of Languages (Karp Reduction)

- A problem specified by language L_1 can be *reduced* to a problem specified by language L_2 if there is a poly-time transformation that maps
  - each string in L_1 to a string in L_2, and
  - each string not in L_1 to a string not in L_2.
- A problem specified by a language L is said to be *complete* for a class if all problems in the class can be reduced to it.
- This definition is equivalent to the one in terms of oracles.
- Note, however, that the certificate produced is for the language L_2.
- To produce a certificate for the language L_1, we need a final transformation step of the certificate for L_2 into one for L_1.
- This notion of reduction was introduced by Karp (1972) in his seminal work "Reducibility among combinatorial problems."

Polyonomial Equivalence

- If P_1 can be reduced to P_2 (Cook or Karp) and vice versa, then the problems are said to be *polynomially equivalent*.
- This defines an equivalence relation that can be used to derive a set of equivalence classes.
- Using these kinds of reduction to divide problems has pros and cons.
  - On one hand, "equivalence" can be determined without knowing the precise running time functions.
  - On the other hand, the classes obtained in this way are not very "fine-grained."
  - These classes lump together many problems that are not really "equivalent" in actual practice.
- As mentioned earlier, the resulting division is mainly between problems that can be solved in polynomial time and those that can't.
- There other possible notions of equivalence, but this is the one that has been endorsed by the research community.

Cook Versus Karp

- Cook and Karp can both be thought in terms of oracles.
  - Cook allows multiple oracle calls.
  - Karp allows only one.
- The two notions of reduction lead to (potentially) different sets of equivalence classes.
- It is not known whether these notions actually lead to exactly the same classes or not.
- There are other notions of reductions that put other limits on the number of oracle calls (logarithmic, etc.).

Certificates

- A *certificate* is a "proof" we can check that certifies the output of a given computation is correct.
- The idea is that checking the validity of such a certificate is more efficient than solving the original problem.
- Formally, suppose we have a problem specified by language L and an algorithm for computing a boolean function M with the property that

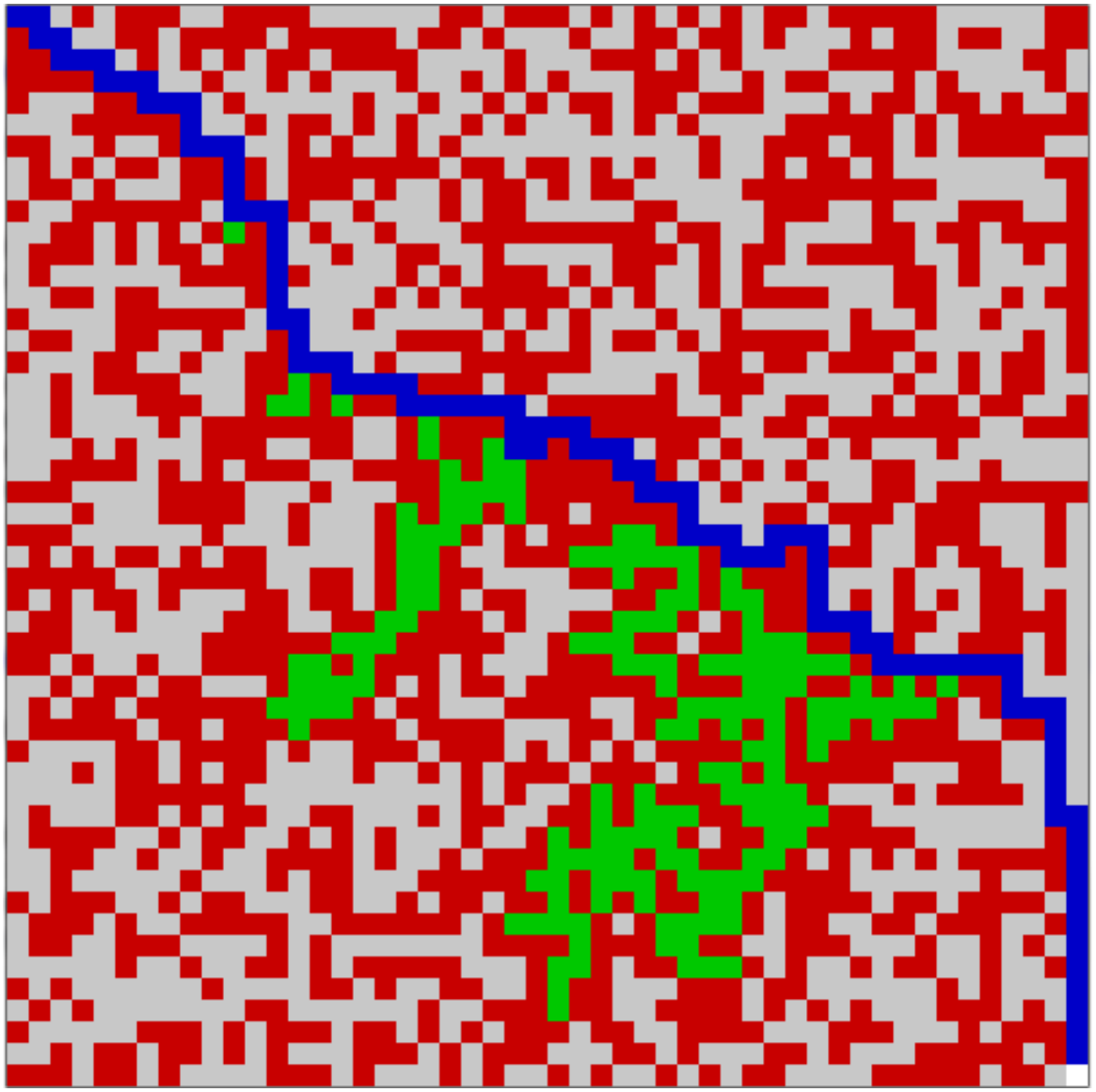
$$x \in L \Leftrightarrow \exists u \in \{0,1\}^* \text{ such that } M(x,u) = 1$$

- The vector u associated with $x \in L$ is the *certificate* for x.
- If we have such an algorithm and can specify a general structure for the associated certificates, then we say the problem itself has a certificate.
- Example: Certificate of optimality for linear programming
  - Given primal and dual solutions, we can verify optimality in $O(mn)$ operations.
  - We have to verify that the magnitude of the numbers is not too big.
  - They are the ratio of two integers, each of which has an encoding length that is polynomially bounded.

Certificates and Algorithms

- A certificate that can be checked in polynomial time is sometimes informally called *short*.
- One way of producing a certificate is just to record the sequence of steps that resulted in the answer.
- In general, however, many "dead ends" may be discarded in producing the final certificate.
- Example: Consider the problem if finding a path through a maze.
  - If we use the naive approach of enumerating all paths, we may explore many dead ends.
  - Once a path is found, the path itself serves as a certificate that such a path exists.
  - We can discard all of the superfluous paths leading to dean ends.
- Thus, every polynomially solvable problem has a short certificate.
- It is not known whether every problem with a short certificate is polynomially solvable.
- Until 1979, linear programming was one problem with a short certificate that was not known to be polynomially solvable.

Certificate For Path Finding



Certificates for Decision Problems

- For many decision problems, the certificate for the YES is easier to verify.
- This is because it usually involves a question of existence, where the NO answer requires proving non-existence.
- (Imperfect) Example: The meeting room problem
  - Decision: Is there anyone in this room that I don't know?
  - There is a short certificate for the YES answer. What is it?
- When the problem is a decision version of an optimization problem, the "solution" serves as a certificate.
  - Consider the Bin Packing Problem.
  - A feasible partition of S into subsets serves as a certificate when there exists a feasible packing.
  - There is no (known) short certificate to show that there *does not* exist a packing.

Non-deterministic Turing Machines

- A *non-deterministic Turing machine* (NDTM) can be thought of as a Turing machine with an infinite number of parallel processors.
- As previously, we could also consider a non-deterministic RAM computer.
- An NDTM follows all possible execution paths simultaneously.
- Informally, if there is a conditional branch in the algorithm, we are able to follow both (all) branches in parallel.
- The algorithm returns `YES` if an accepting state is reached on *any* path.
- The running time of an NDTM is the *minimum* running time of any execution path that ends in an accepting state.
- Alternatively, the running time is the minimum time required to verify that some path (given as input) leads to an accepting state.

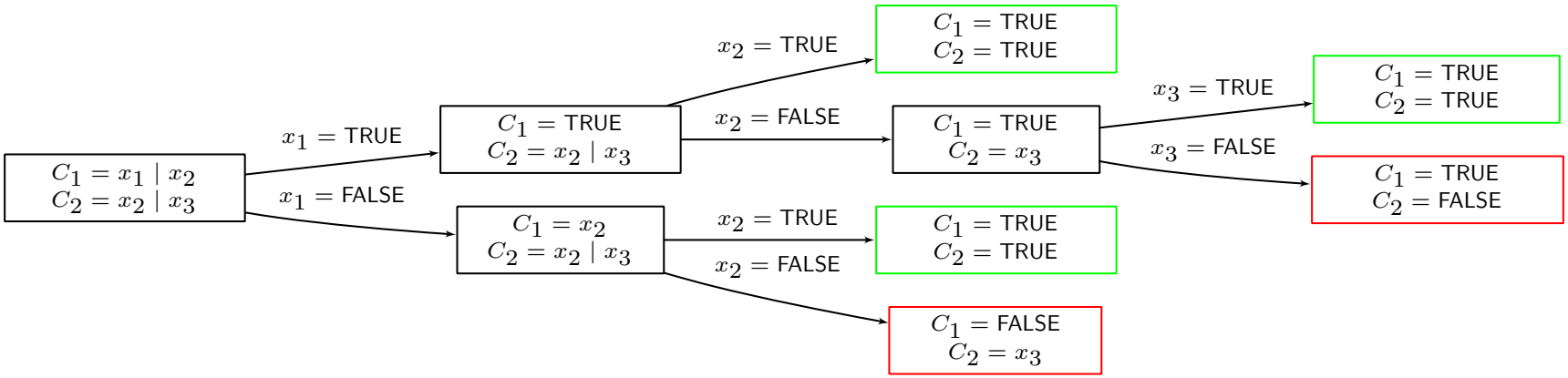
Example: The Satisfiability Problem

- The Satisfiability (SAT) 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

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

  is nonempty.
- There are 2^n possible assignments of values to literals.
- A naive algorithm would explore the "tree" of all 2^n possibilities.
  - This algorithm has an exponential running time of $O(2^n)$ on a DTM.
  - On an NDTM, however, it's running time is $O(mn)$ (the time to construct and check the feasibility of one solution).

Enumeration Tree for SAT Example



Another Way to Think About It

- A nondeterministic algorithm is an algorithm that corresponds to an NDTM.
- The input to the algorithm is a string $s \in \Gamma^*$ (set of all strings formed from characters in Γ).
- Conceptually we can think of the algorithm as having two stages
  - Guessing Stage: Randomly guess a string q (the certificate).
  - Checking Stage: Check whether q can be used to verify that $d \in L$. If so, output `YES`. If not, there is no output.
- There are two properties required.
  - We require that if $d \in L$, then there must exist a certificate that verifies this.
  - The running time of the algorithm is the maximum time it takes to check a certificate that verifies $d \in L$.

NDTMs and Certificates

- Non-deterministic algorithms are so called because the guessing stage is random.
- We can use the description of the path that eventually leads to an accepting state as the certificate.
  - For combinatorial problems, this is usually equivalent to the "solution."
  - Recall the SAT Problem from earlier.
- If the running time of the NDTM is polynomial, then the certificate for the YES answer is short.
- If no accepting path is found, there is no short certificate in general.
  - The *YES* answer is an "existential" statement ($?\exists x$ s.t. ...).
  - The *NO* answer is a "universal" statement ($\forall x \ldots$)

More Complexity Classes

- As described earlier, languages are grouped into *classes* based on the *best worst-case running time function* of any TM that recognizes the language.
- The running time function is as described previously.
  - The class P is the set of all languages for which there exists a DTM that recognizes the language in time polynomial in the length of the input.
  - The class NP is the set of all languages for which there exists an NDTM that recognizes the language in time polynomial in the length of the input.
  - The class coNP is the set of languages whose complements are in NP.
- Additional classes are formed hierarchically by the use of *oracles*, as described earlier.
- Obviously, P is a subset of NP (and coNP).
- It is not known whether P = NP (the million dollar question).

Defining NP and coNP Using Certificates

- Another way of describing the class NP is as the class for which there exists a short certificate for the YES answer.
- This is equivalent to the definition in terms of running time on an NDTM.
- coNP is then the class of problems for which there exists a short certificate for the NO answer.
- Examples
  - Upper bound on the optimization version of bin packing is in NP.
  - Lower bound on the optimization version of bin packing is in coNP.
- If the decision version of an optimization problem is in $NP \cap coNP$, then there exists a certificate of optimality.
- It is unlikely that there exist many problems in $NP \cap coNP$ that are not also in P.

The Class NP-complete

- What are the "hardest" problems in NP?
- As before, we say that a problem specified by a language L is *complete* for NP if every language in NP is polynomially reducible to L.
- This class of problem is denoted NPC (NP-complete).
- Surprisingly, such problems exist!
- Even more surprisingly, this class contains almost every interesting integer programming problem that is not known to be in P!

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

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

The SAT Problem Revisited

- Recall the SAT Problem from earlier.
- This problem is obviously in NP (why?).
- In 1971, Cook defined the class NP and showed that satisfiability was NP-complete, even if each clause only contains three literals.
- This means that all problems in NP somehow share the essential flavor of the SAT problem.
- The solution of all can be reduced to some sort of exponential enumeration scheme with backtracking and many "dead ends."
- When the answer is YES, here must always be a short path that avoids the dead ends and certifies the answer YES.
- So far, we do not know of any algorithms for solving such an NP-complete problem that guarantees *deterministic* polynomial running time.
- The proof is beyond the scope of this course.

Proving NP-completeness

- After satisfiability was proven to be NP-complete, it was easy to prove many other problems 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 NP-complete.
  - This problem is easily shown to be in NP.
  - To prove it is in NP-complete, we reduce 3-satisfiability to it.

The Line Between P and NP-complete

- Generally speaking, most interesting problems are either known to be in P or are NP-complete.
  - The problems known to be in P are generally "easy" to solve.
  - The problems in 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 P (what is it?).
  - If we allow the number of nonzero entries in each column to be three, then this problem is NP-complete!

NP-hard Problems

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

Theory versus Practice

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