College of Computer Science
Northeastern University
COM 3391: Advanced Algorithms

Lecture 1
29 September 1999
Scribe: Rajmohan Rajaraman

Polynomial-Time Reductions, NP-Completeness, and Approximations

1 Introduction

This course will cover some basic topics in the design and analysis of approximation algorithms. The study of approximation algorithms has developed from the seeming intractability of a number of widely-applicable *NP-hard* optimization problems. These optimization problems are unlikely to admit efficient (polynomial-time computable) optimal solutions. Consequently, a number of techniques have been designed to provide approximate (near-optimal) solutions that can be obtained efficiently. Of course, we would like to sacrifice as little optimality as possible, while gaining as much as possible in efficiency. Trading off optimality in favor of efficiency is the paradigm of approximation algorithms.

In this lecture, we provide a brief overview of notions of complexity, approximations, and polynomial-time transformations among problems. Much of the material covered in this lecture is taken from [CLR90] and an excellent set of lecture notes prepared by Rajeev Motwani of Stanford University [Mot92]. We begin with a formal definition of an optimization problem.

Definition 1 (Optimization Problem) *An* **optimization problem** Π *is characterized by a collection of instances, a specification for valid solutions, and a function that evaluates solutions.*

- **Instances:** *D, a set of input instances.*
- **Solutions:** *For each instance I, a set $S(I)$ of solutions.*
- **Value:** *Function $f : S(I) \rightarrow \mathbf{R}$.* ■

Example 1 The minimum spanning tree problem is an optimization problem. The set of instances is the set of all weighted undirected graphs. For a given instance graph G, the set $S(G)$ of solutions for G is the set of all trees that span every vertex of G. Finally, the value of a solution tree T is simply the sum of the weights of the edges of T. ■

A polynomial-time algorithm $\mathcal{A}$ for a given optimization problem Π takes as input an instance I of Π and returns a solution for I in time $O(n^c)$, where n is the size of I and c is a nonegative constant. Examples of optimization problems for which *optimal* polynomial-time algorithms exist include the longest subsequence problem, activity selection problem, minimum spanning tree problem, shortest paths problem, maximum flow and several other problems that we usually encounter in a first algorithms course.

The running time of a polynomial-time algorithm is required to be polynomial in the *size* of the input instance. What is the size of an instance? The size of an instance is defined to be the length of the instance, that is, the number of bits in the representation the instance. This number, however, depends on the particular representation or encoding of the instance. We would have liked our definition of "polynomial-time algorithm" to be independent of the encoding. Unfortunately, the size of an instance depends highly on the encoding used. As a result, a polynomial-time algorithm with respect to one encoding may not be polynomial-time with respect to another. For example, consider the problem of testing whether a given number n is prime. A simple algorithm is to test whether n is divisible by any of the numbers 2 through $\lfloor \sqrt{n} \rfloor$. If it is, then we output no; otherwise, we output yes. Assuming unit time for a division operation, the running time of this algorithm is $O(\sqrt{n})$. Whether this time is polynomial in the size of the input depends crucially on how the input is represented. If the input is represented in unary, then the length of the instance specified by integer n is n. Therefore, in this case, the algorithm is polynomial-time. If the instance is represented in binary, however, the length of the instance is $\log n$, and the running time is $O(\sqrt{n}) = O(2^{(\log n)/2})$, which is exponential in the length of the instance.

For our purposes, we will use a standard encoding. In this encoding, we will represent any integer in binary, and any set as a sequence of encodings of its elements, separated by commas and enclosed in braces. This encoding is considered "reasonable" and is commonly used in complexity theory. The formal definition of a reasonable encoding is somewhat technical and we omit the discussion here. The reader is referred to [GJ79] for details.

In this course, we are going to study optimization problems for which it is unlikely that polynomial-time algorithms producing optimal solutions exist. Before delving into this collection of problems, however, we review the theory of *NP-completeness* that provides a basis for analyzing the complexity of various problems. This theory is best understood, and indeed defined, in terms of *decision problems.*

Definition 2 (Decision problems) *A* **decision problem** Π *is characterized by a set of input instances and a question:*

- **Instances:** *D, a set of input instances.*
- **Question:** *A yes/no question asking whether input instance I has a certain property.* ■

Every optimization problem can be framed as a decision problem. For example, one decision version of the minimum spanning tree problem is as follows.

Example 2 In the decision MST problem, the set of input instances is the set

$$\{\langle G, k\rangle : G \text{ is a weighted undirected graph and } k \text{ is an integer}\}.$$

The yes/no question is: Is there a spanning tree of G with weight at most k? ■

2 Complexity classes P, NP, and NPC

We now define the complexity classes P and NP. The class P is the class of decision problems that can be solved in polynomial time. The class NP is the class of decision problems that can be "verified" in polynomial time. More precisely, a decision problem Π is in NP if for each instance of Π there exists a polynomial-size certificate for each yes-instance of Π that can be verfied in polynomial time. For a precise definition of the class NP, the reader is referred to [CLR90] (page 927). It is easy to see that every problem that can be solved in polynomial time can also be verified in polynomial time. We thus have:

$$\mathrm{P} \subseteq NP.$$

All of the problems that we will study in the course have polynomial-time verification algorithms. In fact, in most cases, these verification algorithms will be straightforward.

A decision problem that can be verfied in polynomial time, yet for which no polynomial-time algorithm is known, is the HAMILTONIAN CYCLE problem.

Definition 3 *The* HAMILTONIAN CYCLE *problem is to determine whether a given undirected graph has a cycle that covers all the vertices exactly once.*

- **Instance:** *Undirected unweighted graph $G = (V, E)$.*
- **Question:** *Is there a simple cycle in G that visits every vertex exactly once?* ■

Given a graph G, a cycle in G that visits every vertex of a graph G exactly once is called a *hamiltonian cycle*. A certificate for a yes-instance of the above problem is the hamiltonian cycle itself. It is easy to see that given a cycle and a graph, it can be verified in polynomial time whether the cycle is hamiltonian in the graph.

While HAMILTONIAN CYCLE is easily seen to be in NP, the question of whether it is in P is still open. In fact, HAMILTONIAN CYCLE is one of a large collection of problems in NP, called *NP-complete problems*, that are strongly believed to be not in P. That is, it is unlikely that HAMILTONIAN CYCLE can be solved in polynomial time. The class of NP-complete problems, henceforth referred to as NPC, is a class of decision problems in NP that satisfy the following property: if any problem Π in NPC is solvable in polynomial time, then every problem in NP is in P. Thus, the problems in NPC are the "hardest" problems in NP. In order to provide a more formal characterization of NPC, we introduce the important concept of *polynomial-time reductions*.

Definition 4 (Polynomial-time reductions) *We say that a decision problem Π_1 is polynomial-time reducible to another decision problem Π_2 if there exists a polynomial-time computable function $f : D(\Pi_1) \to D(\Pi_2)$ (from the instances of Π_1 to the instances of Π_2) such that the following condition holds: for every instance I of Π_1, I is a yes-instance of Π_1 if and only if $f(I)$ is a yes-instance of Π_2. We write this as $\Pi_1 \leq_P \Pi_2$.* ■

There are two crucial characteristics of a polynomial-time reduction worth noting in Definition 4. First, the function f is polynomial-time computable. Second, f maps yes-instances of Π_1 to yes-instances of Π_2 and no-instances of Π_1 to no-instances of Π_2. The following theorem is immediate from Definition 4.

Theorem 1 *If $\Pi_1 \in P$ and $\Pi_2 \leq_P \Pi_1$, then $\Pi_2 \in P$.*

Proof: Let $\mathcal{A}$ be an algorithm for Π_1. Let f be the polynomial-time computable function that provides the reduction from Π_2 to Π_1. Then, a polynomial-time algorithm for Π_2 is the following: Given any instance I of Π_2, return the output of $\mathcal{A}$ on instance $f(I)$. It is easy to see from the definition of $\leq_P$ that this is a valid algorithm for Π_2. Furthermore, the algorithm runs in polynomial time. ■

Using the definition of $\leq_P$, we now formally define the class NPC as follows.

Definition 5 (NP-complete problems) *A decision problem* Π *is in NPC (or alternatively,* Π *is NP-complete) if the following two conditions hold:*

- Π *is in NP.*
- *For every problem* $\Pi_1 \in NP$, $\Pi_1 \leq_P \Pi$. ■

A problem that satisfies the second condition of Definition 5 is referred to as an *NP-hard* problem. Thus, a problem Π is NP-complete if and only if Π is in NP and Π is NP-hard. The definitions of polynomial-time reductions and NP-completeness now directly yield the following theorem that establishes the desired property of NP-complete problems mentioned at the outset.

Theorem 2 *If any NP-complete problem is solvable in polynomial time, then* $P = NP$*. If any problem in NP is not solvable in polynomial-time, then no problem in NPC is solvable in polynomial-time.*

Proof: Let Π be a problem in NPC that is solvable in polynomial time. Consider any problem $\Pi_1 \in$ NP. By the second condition of Definition 5, it follows that there exists a polynomial-time computable function f that reduces Π_1 to Π. Therefore, Π_1 can be solved in polynomial time. The second claim of the theorem also follows using a similar argument. ■

While Theorem 2 formally establishes that NP-complete problems are the "hardest" problems in NP, it remains to be seen whether this class is even non-empty! In order to prove that a problem Π is NP-complete, we are faced with the daunting task of establishing that *every problem* in NP is polynomial-time reducible to Π. While this is indeed a challenging task, the transitivity of polynomial-time reductions suggests that once we know an NP-complete problem, proving the NP-completeness of other problems is much easier. The proof of the following lemma is left as an easy exercise.

Lemma 1 *If* Π_1 *is NP-complete,* $\Pi_2 \leq_P \Pi_1$*, and* $\Pi_2 \in NP$*, then* Π_2 *is also NP-complete.* ■

The first problem that was proven to be NP-complete was the satisfiability problem (SAT).

Definition 6 (SAT) *An instance of* SAT *is given by a boolean formula that consists of boolean variables* $x_1, x_2, \ldots, x_n$ *connected by means of boolean connectives* $\wedge$, $\vee$, *and* $\neg$ *and parenthesis for scoping. The question is to determine whether there exists an assignment for the* n *variables that satisfies the formula.* ■

The following special case of SAT, which is sometimes more useful for deriving polynomial-time reductions, is also NP-complete.

Definition 7 **(3SAT)** *3SAT is a special case of SAT in which the input formula is represented as conjunction of clauses, where each clause is a disjunction of at most three literals.* ■

3 Example reductions

We now illustrate the use of polynomial-time reductions in establishing the NP-completeness of two decision problems. We first consider the CLIQUE problem.

Definition 8 **(CLIQUE)** *A* **clique** *of an undirected graph $G = (V, E)$ is a subset V' of V such that every pair of vertices in V' is connected by an edge in E. An instance of CLIQUE is a pair (G, k), where G is an undirected graph and k is a positive integer. The question is whether there exists a clique in G of size at least k.* ■

The proof of the following theorem is also given in [CLR90] (page 947). The reduction is from 3SAT.

Theorem 3 CLIQUE *is NP-complete.*

Proof: We first prove that CLIQUE is in NP. The certificate for a given yes-instance of clique $\langle G, k\rangle$ is simply the set of k vertices that purportedly form a clique in G. The correctness of such a certificate can be verified in polynomial time by checking whether there is an edge in G between any two vertices in the certificate.

Our second obligation is a polynomial-time reduction from an NP-complete problem to CLIQUE. We show that 3SAT $\leq_P$ CLIQUE. An instance F of 3SAT is specified as a conjunction of clauses, each clause being a disjunction of at most three literals. Let the clauses be labeled C_1 through C_m. Let the formula F consist of n boolean variables labeled x_1 through x_n.

We construct the following instance $\langle G, k\rangle$ of CLIQUE. For each occurence of a literal in the formula, we have a vertex in G. Thus, the number of vertices in G is at most $3m$. We have no edges between vertices that represent literals belonging to the same clause. For any two vertices that represent literals ℓ_1 and ℓ_2 in different clauses, we have an edge if and only if ℓ_1 and ℓ_2 can be simultaneously made true; that is, if and only if ℓ_1 is not equal to $\neg\ell_2$. Finally, we let k equal m. This completely specifies the CLIQUE instance. The running time of this construction is polynomial in m, and hence, polynomial in the size of F. This construction is illustrated on page 948 of [CLR90].

We now prove that F is satisfiable if and only if G has a clique of size k. Let F be satisfiable. This implies that there exists an assignment σ that makes true at least one literal ℓ_i in each clause C_i. Consider the m vertices corresponding to these m literal occurences. They all belong to different clauses and are all made simultaneously true by σ. Therefore, there is an edge between any two of these vertices, yielding a clique of size k in G.

Now suppose that G has a clique of size k. We show that F is satisfiable. Since there are no edges between vertices that represent literals in the same clause, the clique consists of vertices representing exactly one literal from each clause. Since all of these vertices are adjacent to each other, it implies that there exists an assignment σ that makes all the corresponding literals simultaneously true. This assignment σ thus satisfies each clause, and hence the formula F. ■

Similarly, one can use a reduction from CLIQUE to prove that the VERTEX COVER problem, defined below, is NP-complete.

Definition 9 (VERTEX COVER) *A* **vertex cover** *of an undirected graph $G = (V, E)$ is a subset V' of V such that every edge in E is incident on at least one vertex in V'. An instance of* VERTEX COVER *is a pair (G, k), where G is an undirected graph and k is a positive integer. The question is whether there exists a vertex cover of G of size at least k.* ■

The reduction in Theorem refthm:vc uses the fact that a clique V' of a graph G yields a vertex cover $V - V'$ in the *complement graph* $\bar{G}$ and vice versa.

Theorem 4 VERTEX COVER *is NP-complete.*

Proof: We first show that VERTEX COVER is in NP. To see this, we note that a certificate for a yes-instance of VERTEX COVER is simply the set of vertices that form the vertex cover. The correctness of such a certificate can be verified by checking whether every edge in G is adjacent to at least one vertex in this certificate. Clearly, this verification process is polynomial-time.

We now prove that CLIQUE $\leq_P$ VERTEX COVER. Let $\langle G = (V, E), k\rangle$ be an instance of CLIQUE. Let $\bar{G} = (V, \bar{E})$ denote the complement of G; that is, $\bar{G}$ has the same set of vertices as G, and edge e is in $\bar{E}$ iff e is not in E. Consider the instance $\langle \bar{G}, |V| - k\rangle$ for VERTEX COVER. Clearly, the construction takes time polynomial in the size of the instance $\langle G, k\rangle$.

We now show that G has a clique of size k iff $\bar{G}$ has a vertex cover of size $|V| - k$. This follows from the following claim: S is a clique in G iff $V - S$ is a vertex cover in $\bar{G}$. S is a clique in G iff $\bar{G}$ has no edge between any two vertices in S. This is true iff every edge in $\bar{G}$ has at least one vertex in $V - S$. Which is equivalent to the statement that $V - S$ is a vertex cover in $\bar{G}$. This completes the reduction. ■

4 Notions of approximation

For any NP-complete problems, for example, the problems SAT, 3SAT, CLIQUE, and VERTEX COVER considered in Section 2 and 3, it is unlikely that we can devise a *polynomial-time* algorithm that solves *every instance* of the problem *optimally* in polynomial time. We can relax our requirements in the following three ways:

- **Super-polynomial time algorithms:** Instead of requiring poly-time algorithms, we may demand a slightly super-polynomial time algorithm. Indeed, for certain problems such as KNAPSACK, which we study in Lecture 2, we can obtain efficient slightly super-polynomial time algorithms that solve the probem optimally. It turns out, however, that the fastest known algorithms for almost all of the interesting NP-complete problems is exponential time.

- **Focus on a subset of the instances:** We can also relax our objective by focusing on a subset of the instances, rather than demanding that we solve every instance optimally. The trouble with this approach is to define the particular subset of instances we are interested in. While it may be possible to define the notion of a "random instance" in some cases and analyze the expected running-time of algorithms by probabilistic means, this is difficult to justify for most problems.

- **Approximation algorithms:** A third relaxation, which is the one we will concentrate on during this course, is to allow for approximate solutions. That is, rather than solving optimally for every instance, we will demand *polynomial-time* algorithms that solve *near-optimally* for *every instance.*

What does "near-optimal" mean? One notion of approximation is that of an *absolute performance guarantee*, in which the value of the solution returned by the approximation algorithm differs from the optimal value by an absolute constant. Let Π be an optimization problem and let I be an instance of Π. Given an algorithm $\mathcal{A}$ for Π, let $\mathcal{A}(I)$ denote both the solution as well as the value of the solution returned by $\mathcal{A}$ on instance I. Also, let $OPT(I)$ denote the optimal value for instance I.

Definition 10 *An approximation algorithm $\mathcal{A}$ for problem Π has an* **absolute performance guarantee** *of k if for the following condition holds for all instances I of Π:*

$$|\mathcal{A}(I) - \mathrm{OPT}(I)| \leq k.$$

■

In the following lecture, we provide evidence that the notion of absolute performance guarantee is not appropriate for many NP-complete problems. We then define a more suitable notion of *relative performance guarantee*, which is the notion that we will employ for the remainder of the course.

References

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, New York, 1979.

[Mot92] R. Motwani. Lecture notes on approximation algorithms: Volume I. Technical Report STAN-CS-92-1435, Department of Computer Science, Stanford University, 1992. Postscript document is available at http://Theory.Stanford.EDU/~rajeev/postscripts/approximations.ps.gz (134 pages).