6.893 Approximability of Optimization Problems	September 8, 1999
Lecture 1	
Lecturer: Madhu Sudan	*Scribe: Rocco Servedio*

1 Course Overview

This is a course about the approximability of optimization problems. The optimization problems which we will consider are all "hard" in that no efficient algorithms are known for finding optimal solutions (and there are good reasons for believing that no efficient algorithms exist). However, as we will see, these optimization problems lie across a broad spectrum in terms of how easy (or difficult) it is to find *approximately* optimal solutions.

The course will start out with some positive results: we will review some well-known approximation algorithms for various optimization problems. This will hopefully highlight some of the differences between finding optimal and approximate solutions to optimization problems.

We will then spend quite a bit of time on negative (nonapproximability) results, which essentially say that for various optimization problems it is as difficult to find an approximately optimal solution as it is to find a truly optimal solution. These results were discovered in the past decade and rely heavily on the theory of probabilistic proof systems; a large component of the course will consist of developing the theory of these proof systems and exploring their connections with nonapproximability. In particular, we will spend quite a bit of time studying constructions of probabilistically checkable proof systems (PCP); this will include error-correcting codes, program checking, and low-degree testing for polynomials and linearity testing. We will also discuss complexity classes of optimization problems, reductions between optimization problems, and the use of gadgets in these reductions.

Finally, toward the end of the course we will study some recent approximation algorithms which were discovered in the context of the PCP-related nonapproximability theory outlined above. We will cover Arora's recent polynomial-time approximation algorithm for the Euclidean traveling salesman problem in the plane and related algorithms.

In the rest of today's lecture we went over some of the history of combinatorial optimization as a way of motivating basic definitions which we will use throughout the course.

2 Combinatorial Optimization: A Brief History

Combinatorics questions deal with the existence or enumeration of objects from within some finite space, whereas optimization questions are about finding the "best" object from within some (not necessarily finite) space of candidates. Combinatorial optimization can be viewed as the study of finding the "best" object from within some finite space of objects. In one form or another, mathematicians have been studying combinatorial optimization problems at least since the turn of the century – an early book on the subject is Lawler's *Combinatorial Optimization: Networks and Matroids* from 1976. Two important topics from the early history of combinatorial optimization are network flow and linear programming, which we discuss in the next subsection.

2.1 Some Early Results

2.1.1 Network Flow

A *flow graph* is a directed graph $G = (V, E)$ which has two distinguished vertices s and t (source and sink, respectively) and where each directed edge $e \in E$ has an associated nonnegative integer capacity $c(e)$. A *flow* in G is a function f from E to the nonnegative real numbers which satisfies $f(e) \leq c(e)$ for every edge e, and which also satisfies conservation of flow (the total flow into each vertex, except for s and t, equals the total flow out of each vertex); the *value* of a flow f is the total flow out of s minus the total flow into s. The *maximum flow problem* is the following: given a flow graph G, find a flow which achieves the maximum value.

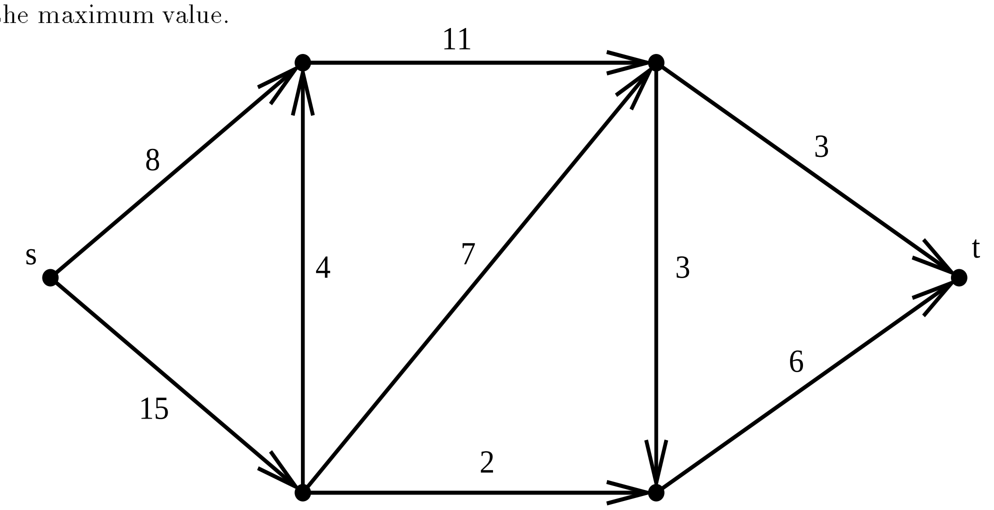

Figure 1: A flow graph with capacitated edges.

At first this problem might not seem like a combinatorial optimization problem at all – since a flow is a real-valued function, it might seem as if we are searching an infinite space rather than a finite one. Ford and Fulkerson studied this problem in the 1950s and showed that in fact it is a discrete combinatorial problem (one need only consider integer-valued flows, of which there are a finite number for any given flow graph). They gave an algorithm which always succeeds in finding the maximum flow, and proved the *max-flow-min-cut* theorem, which asserts that the value of the maximum flow equals the size of the minimum cut in the graph.

2.1.2 Linear Programming

Linear programming is another classical combinatorial optimization problem. An instance of the linear programming problem is an n-dimensional vector c, an n-by-m-dimensional matrix A, and an m-dimensional vector b (all vectors and matrices are over the real numbers). The linear programming problem is the following: find an n-dimensional vector x which maximizes $\sum_{i=1}^{n} c_i x_i$ subject to the constraints $\sum_{i=1}^{n} A_{ij} x_i \leq b_j$ for $j = 1, \ldots, m$.

Here too, it seems at first that we are searching a continuous space rather than a discrete one. Dantzig and Von Neumann realized that the maximum value must occur at a vertex of the simplex defined by the constraint hyperplanes, and that hence the search space is in fact a finite one. Dantzig proposed the *simplex* algorithm and proved that it will always succeed in finding an optimal solution.

2.2 The Importance of Polynomial Time

Neither Ford and Fulkerson's original algorithm for network flow nor Dantzig's simplex algorithm will always run in time polynomial in the description length of the input instance. However, these two algorithms were not challenged as inefficient for many years, because polynomial running time had not yet been equated with algorithmic efficiency. In 1965 Jack Edmonds wrote a paper titled "Paths, trees, and flowers" which underscored the importance of polynomial runtime. In this paper he gave a relatively intricate algorithm for finding a perfect matching in a nonbipartite graph. Edmonds pointed out that while this problem could be trivially solved by brute force search, such an approach would take exponential time, whereas his algorithm required only polynomial time. In a companion paper Edmonds drew a distinction between problems for which polynomial time algorithms were known (such as perfect matchings) and problems for which no such algorithms were known (such as Hamiltonian paths) and conjectured that no polynomial-time algorithm exists for the latter problem.

2.3 Early History of NP

In 1971 Stephen Cook published his important paper "The complexity of theorem-proving procedures," in which he formally defined the class NP and introduced the notion of NP-completeness. He showed that the satisfiability problem is NP-complete, thus proving that it is as hard as the hardest language in NP. Shortly afterwards, Richard Karp enumerated many combinatorial problems (including CLIQUE, GRAPH COLORING, VERTEX COVER, TSP, BIN PACKING, and PARTITION) and proved that all these problems are NP-complete. At approximately the same time, Leonid Levin (then in the U.S.S.R.) published a paper which established very similar results. (In addition to these results, in his two-page paper Levin also gave an algorithm for the satisfiability problem which is provably optimal; the hitch is that it is totally unclear how to analyze its running time.) The publication of Levin's results was slightly delayed because in addition to the problems which he did show were NP-complete he also wanted to prove that linear programming, primality, and graph isomorphism are NP-complete as well. We now have stronge evidence that these problems are not NP-complete.

3 NP and NP-Completeness

A *language* is a collection of binary strings $L \subseteq \{0,1\}^*$. (No real loss of generality is incurred by restricting ourselves to a binary alphabet since any finite alphabet can be efficiently encoded using a binary alphabet.) The class NP is defined as follows:

Definition 1 *A language L is in NP if there is a polynomial-time machine M such that for any input $x \in \{0,1\}^*$,*

- *If $x \in L$ then $M(x,y)$ accepts for some $y \in \{0,1\}^*$ with $|y|$ bounded by a polynomial in $|x|$;*
- *If $x \notin L$ then $M(x,y)$ rejects for all $y \in \{0,1\}^*$.*

The "standard" definition of NP is that it is the class of languages recognizable in nondeterministic polynomial time. This definition is easily seen to be equivalent to the one we have given; we note that in our definition nondeterminism is captured by the condition that there must exist *some* short string y which causes M to accept input x. It is useful to think of y as a "short proof" that x belongs to the language L and to view the computation of M on x, y as a verification of this proof. Note that languages in NP are somewhat one-sided; if $x \in L$ then there is a short proof that $x \in L$, but if $x \notin L$ then there may not exist any short proof that $x \notin L$.

3.1 Examples of Languages in NP

Our first example of a language in NP is the language 3SAT. An instance of 3SAT is a collection of m clauses $C_1, \ldots, C_m$, where each clause C_j is of the form $x_{i_1} \vee x_{i_2} \vee \overline{x}_{i_3}$ for some $i_1, i_2, i_3 \in \{1, \ldots, n\}$. (Of course, this is all represented using some fixed reasonable representation scheme over a binary alphabet). A collection of clauses $\phi = C_1, \ldots, C_m$ belongs to the language 3SAT if there is some assignment of TRUE and FALSE to the variables $x_1, \ldots, x_n$ which simultaneously satisfies each of the m clauses $C_1, \ldots, C_m$. To see that the language 3SAT is in NP, we note that the desired machine M takes as its two inputs ϕ (a representation of the 3SAT formula) and y (a representation of the truth assignment to $x_1, \ldots, x_n$) and simply verifies that under the given truth assignment, every clause is satisfied. This can clearly be done in polynomial time.

Another interesting language in NP is the language SHORTPROOF. Fix some formal system for expressing mathematical statements: the language SHORTPROOF consists of those statements S in the formal system which have the property that there is a proof of S in the formal system whose length is bounded by a polynomial in the length of S. (We have described the language SHORTPROOF rather informally; it is tedious but straightforward to give a completely rigorous description.) It is clear that SHORTPROOF is in NP – the string y is simply the proof of statement S and the polynomial-time machine works by verifying that y is a valid proof of S according to the rules of the formal system. We note that a polynomial-time algorithm for recognizing membership in SHORTPROOF would essentially put all mathematicians out of business – such an algorithm could be used to find a short proof for any mathematical statement which has a short proof, and this is more or less exactly what mathematicians do (or try to do).

3.2 Reductions and NP-Completeness

Although 3SAT seems very "combinatorial" and SHORTPROOF seems very "logical," these languages are equivalent in an important sense since both are NP-complete. To define NP-completeness we need the notion of a *reduction* between languages.

Definition 2 *Let L_1, L_2 be languages. We say that L_1* reduces to L_2 in polynomial time *(abbreviated $L_1 \le L_2$) if there is a polynomial-time algorithm f such that $\forall x \in \{0,1\}^*$ we have $x \in L_1 \iff f(x) \in L_2$.*

If $L_1 \le L_2$ then a polynomial-time algorithm for determining membership in L_2 implies the existence of a polynomial-time algorithm for membership in L_1 as well: given a string x for which membership in L_1 is to be determined, simply run the algorithm for membership in L_2 on the string $f(x)$.

Definition 3 *A language L is* NP-hard *if for every L' in NP we have $L' \le L$. A language L is* NP-complete *if $L \in NP$ and L is NP-hard.*

Thus an NP-complete language is at least as hard as any language in NP. A major achievement of theoretical computer science in the 1970s was the discovery that many important computational problems are NP-complete, including 3SAT, SHORTPROOF and many NP optimization problems.

3.3 NP Optimization Problems

An NP optimization problem is described by two functions `FEASIBLE`$(\cdot,\cdot)$ and `OBJECTIVE`$(\cdot,\cdot)$ and a bit whose value is either `MAX` or `MIN`. The functions `FEASIBLE` and `OBJECTIVE` are each polynomial-time computable; the function `FEASIBLE` is boolean-valued and the function `OBJECTIVE` is real- or integer-valued. For a given NP optimization problem, `FEASIBLE`(X,Y) is true iff Y represents a feasible solution to the problem instance X. As an example, for the traveling salesman problem `FEASIBLE`(X,Y) is true iff X represents a edge-weighted graph G and Y represents a tour which starts and ends at the same node and touches each node at least once. The value of `OBJECTIVE`(X,Y) measures the quality of the solution Y; for the traveling salesman problem, `OBJECTIVE`(X,Y) is the sum of the weights of the edges traversed in the tour Y. The bit `MAX/MIN` determines whether the goal is to maximize or minimize the the objective function; for the traveling salesman problem the bit is `MIN`.

There are actually three different versions of any given NP optimization problem. In the *search* version, given a problem instance X the goal is to find a feasible solution Y which optimizes `OBJECTIVE`(X,Y), i.e. either maximizes or minimizes it depending on whether the bit is `MAX` or `MIN`. In the *computation* version, given an instance X the goal is to find the maximum/minimum value of `OBJECTIVE`(X,Y) over all feasible solutions Y. In the *decision* version, given an instance X and a number B, if the bit is `MAX` then the goal is to determine whether there exists a feasible solution Y such that `OBJECTIVE`$(X,Y) \ge B$ (if the bit is `MIN` then $\ge$ is replaced by $\le$). The decision version of an NP optimization problem is the one that corresponds to a formal language – the language is simply the set of all strings representing X, B such that the condition of the previous sentence is satisfied.

4 Approximation of NP Optimization Problems

4.1 Early History

The most basic dichotomy for NP optimization problems is whether the language associated with the decision version of the problem is polynomial-time recognizable or is NP-complete. Often a seemingly minor change in the problem definition will move a problem over the boundary from P to NP; see Table 1 for several examples of this phenomenon. (Exact definitions of all the problems listed in Table 1 can be found in Garey and Johnson's book *Computers and Intractability: A Guide to the Theory of NP-Completeness.*)

In P	NP-complete
EDGE COVER	VERTEX COVER
MINIMUM CUT	MAXIMUM CUT
MINIMUM SPANNING TREE	MINIMUM STEINER TREE
SHORTEST PATH	LONGEST PATH
CHINESE POSTMAN	TRAVELING SALESMAN
2D ASSIGNMENT	3D ASSIGNMENT

Table 1: The computational complexity of some optimization problems.

At this relatively crude level of analysis – is or isn't the problem NP-complete? – all the problems on the right side of Table 1 are equivalent. In the 1970s various researchers including Garey, Johnson, Sahni and Gonzales started to consider the question of how easy it might be to find *approximate* solutions, rather than optimal ones, to some of these problems. They discovered that different problems which are "equivalent" in the sense of NP-completeness frequently seemed to have widely different approximability behaviors. By the end of the 1970s, the state of the art in approximation for the following four problems was as described below:

- For the KNAPSACK problem: an algorithm such that for any $\delta > 0$ the algorithm runs in time poly$(n, 1/\delta)$ and finds a solution which is guaranteed to be within a $(1+\delta)$-factor of the optimal solution.
- For the MINIMUM MAKESPAN problem: an algorithm such that for any $\delta > 0$ the algorithm runs in time poly$(n^{1/\delta})$ and finds a solution which is guaranteed to be within a $(1+\delta)$-factor of the optimal solution.
- For the VERTEX COVER problem: an algorithm that runs in polynomial time and finds a vertex cover which is at most twice as large as the minimum vertex cover.
- For the GRAPH COLORING problem: nothing known.

Even though the languages associated with these problems are all reducible to each other, the standard reductions do not preserve approximability.

4.2 Complexity of Approximation: A Brief Timetable

In the 1970s through most of the 1980s, relatively little work was done on the computational complexity of approximation for NP optimization problems; instead, more research was directed toward finding approximation algorithms for various optimization problems. In the 1970s Johnson gave simple 2-approximation algorithms for MAX3SAT and MAXCUT, and Lovasz and Johnson independently gave an $O(\log n)$-approximation algorithm for SET COVER. In the 1980s Hochbaum and Shmoys gave approximation algorithms for various scheduling and packing problems, and Leighton and Rao gave approximation algorithms for some flow problems. However, even on the algorithmic side approximation was not really a major research topic, as witnessed by the fact that the 1985 book *Combinatorial Optimization: Annotated Bibliographies* did not include a chapter on approximation algorithms.

The situation changed in 1988 when Papadimitriou and Yannakakis published a paper "Optimization, Approximation, and Complexity Classes" which established the first successful complexity-theoretic point of view on approximability of optimization problems. They defined complexity classes of optimization problems, gave a notion of approximation-preserving reductions between optimization problems, gave a completeness notion and proved that several natural and important optimization problems, including MAX3SAT, are complete under this notion. In the next few years many other optimization problems were shown to satisfy this completeness notion as well.

At this point, "hard-to-approximate" was a different hardness assumption than "hard-to-solve-exactly" (i.e. the P $\neq$ NP assumption). In 1990, an important paper of Feige, Goldwasser, Lovasz, Safra and Szegedy established a connection between approximation and probabilistic proof systems. On a high level, their work showed that being able to find approximately optimal solutions corresponds to being able to find proofs which have "only a few errors." This notion may seem strange at first, but such "only slightly incorrect" proofs are important in the theory of probabilistically checkable proofs. The work of [FGLSS] and subsequent work led to the collapsing of the two different hardness assumptions mentioned above in many cases – for many NP-complete optimization problems it is now known that even *approximating* the solution within some bound is also NP-complete. Apart from its inherent interest, this progress in the complexity of approximation has been useful in focusing algorithmic research – designers of approximation algorithms can avoid expending effort on trying to design approximation algorithms where appxoximation is known to be NP-complete.

5 Optimization Problems and Approximation Algorithms

Recall that an optimization problem Π has three components: a polynomial-time boolean function $\texttt{Feasible}(\cdot,\cdot)$, a polynomial-time real-valued function $\texttt{Objective}(\cdot,\cdot)$, and a bit $\texttt{MAX/MIN}$. Throughout this section, unless stated otherwise we assume that the value of the bit is $\texttt{MAX}$.

Definition 4 *We say that Y is an* optimal solution *to the problem instance X if Y maximizes $\texttt{Objective}(X, Y')$ over all Y' such that $\texttt{Feasible}(X, Y')$ is true.*

Definition 5 *For $R \geq 1$, we say that a solution Y'* achieves performance ratio R *if*

$$\texttt{Objective}(X, Y') \geq \frac{\texttt{Objective}(X, Y)}{R}$$

where Y is an optimal solution.

Equivalently, we will say that Y' is an *R-approximate solution.*

Definition 6 *We say that an algorithm A is a R-*approximation algorithm *for problem Π if, given a problem instance X, algorithm A outputs a solution $A(X)$ which has a performance ratio of at least R.*

Note that we do *not* require that the approximate solution is always *exactly* $1/R$ times the optimal solution; if this were the case, then we could determine the value of the optimal solution simply by multiplying by R. Also note that the value R can be a function of the problem size n such as $\log n$.

Definition 7 *We say that an optimization problem Π is R-*approximable *if there is an R-approximation algorithm for Π which runs in polynomial time.*

From the preceding definitions, it is clear that a 1-approximation algorithm is an algorithm which always succeeds in finding an optimal solution, i.e. $R \geq 1$ always. For minimization problems, the inequalitity in the preceding paragraph is replaced by

$$\texttt{Objective}(X, Y') \leq R \cdot \texttt{Objective}(X, Y)$$

and hence we will always have $R \geq 1$ for minimization problems as well.

In the next few lectures we will see a range of approximation algorithms which have many different values of R.