6.854 Advanced Algorithms	
Lecture 12: October 20, 2003	Lecturers: Erik Demaine, David Karger
Scribe: Jason Eisenberg	

Approximation Algorithms

12.1 Introduction

So far in class, we have studied problems which are efficiently solvable (solvable in polynomial time), and we have asked how quickly we can solve them. For the next few lectures, however, we will consider problems which are not known to be efficiently solvable.

12.1.1 NP-Completeness

In studying such problems, we encounter the notion of NP-completeness. NP-complete problems comprise a family of thousands of distinct combinatorial and optimization problems for which

- no efficient algorithms are known; we can, however, use brute force to solve these problems in exponential time
- an efficient reduction exists from every other NP-complete problem; thus, if we have a black box which is able to solve one of these problems efficiently, we can solve all of the problems efficiently

Below are listed some examples of NP-complete problems. Each problem except SAT is formulated as an optimization problem, although strictly speaking it is the decision version of each problem that is NP-complete. As an aside, there does exist an optimization problem related to SAT, called MAXSAT. In this problem, we are given a boolean formula, and we must find an assignment to the variables which maximizes the number of satisfied clauses.

Satisfiability (SAT): Given a boolean formula, is there an assignment to the variables which satisfies the formula (makes the formula evaluate to true)?

Bin Packing: Given a set of items of specified sizes and unit-size bins, determine the minimum number of bins required to hold the items.

Max Independent Set: Given a graph, find a maximum-size subset of vertices such that no two vertices in the subset are adjacent.

Knapsack: Given a knapsack of fixed size and a set of items, each of a specified value and size, determine the maximum total value of any set of items which fits in the knapsack.

Parallel Machine Scheduling: Given a set of identical machines and a set of tasks, each of specified duration, find an assignment of tasks to machines which minimizes the time required for all machines to complete their assigned tasks.

Traveling Salesman Problem (TSP): Find a minimum-distance route through a set of cities which allows a salesman to begin and end in the same city and visit every other city exactly once.

That all of these problems cannot be solved efficiently depends on the assumption that P $\neq$ NP. Although this conjecture is not proven, it is widely believed to be true, and so we will simply assume that P $\neq$ NP.

12.1.2 Coping with NP-Completeness

Since we do not know how to solve NP-complete problems efficiently, what can we do?

heuristics: One possibility is to abandon searching for polynomial-time algorithms and to instead concentrate on developing heuristics which are almost polynomial-time in practice for instances which are not too large. But in general, it would be surprising to obtain algorithms which are even subexponential, as achieving this goal would have ramifications as to whether P and NP are equal.

average-case analysis: Rather than looking at the worst-case performance of algorithms for these problems, we can analyze the algorithms' behavior only on certain classes of inputs. This is done by determining their expected performance over some specified distribution of the input instances. But there is often much disagreement over which distribution to use.

approximation algorithms: We can attempt to find polynomial-time approximation algorithms which can be proven to be *approximately* correct.

We will explore the topic of approximation algorithms over the next few lectures.

12.2 Optimization Problems

Before discussing approximation algorithms, we must establish some terminology for optimization problems.

Definition 1 *An optimization problem has a set of* **problem instances**.

Definition 2 *Each instance I has a* **solution set S(I)**.

Definition 3 *The maximization/minimization problem is to find a solution $s \in S(I)$ of maximum/minimum objective* **value f(s)**. *We will assume the input and output of f are integers composed of a polynomial number of bits.*

Definition 4 *The value $f(s)$ of an optimum solution s for instance I is denoted* **OPT(I)**.

Each of the optimization problems given in Section 12.1.1 fits into this optimization framework. For example, consider the Max Independent Set problem. Each problem instance is a graph; the solution set for a graph consists of all subsets of vertices such that no two vertices in each subset are adjacent; and the value of a solution is the number of vertices in the subset.

Although we would like to refer to these optimization problems as being NP-complete, this term is usually reserved for decision problems and languages. Instead, we use the concept of NP-hardness.

Definition 5 *An optimization problem is* **NP-hard** *if some other NP-hard problem can be reduced to it in polynomial time.*

Usually, the NP-hard problem used in the reduction is the corresponding decision problem of whether $OPT(I)$ is at least (or at most) some value k.

12.3 Absolute Approximation Algorithms

Definition 6 *An* **approximation algorithm** *is a polynomial-time algorithm which when given an instance I, returns a solution s in the solution space $S(I)$.*

For example, in the bin-packing problem, one possible approximation algorithm is to simply place each item in its own bin. But doing so most likely produces a poor quality solution. To address the issue of quality, let us consider absolute approximation algorithms.

Definition 7 *Given an instance I, an* **α-absolute approximation algorithm** *finds a solution of value at most $OPT(I) + \alpha$.*

Note that this definition only makes sense for minimization problems; an α-absolute approximation algorithm for a maximization problem would return a solution of value at least $OPT(I) - \alpha$. Further, observe that when designing an absolute approximation algorithm, we would like α to be as small as possible.

12.3.1 Algorithms for Graph Coloring

Consider the problem of **planar graph coloring**, in which we are given a planar graph (one which can be drawn in a plane without its edges crossing) and we must find a coloring of the vertices such that no two neighboring vertices have the same color. As the following theorem demonstrates, this problem possesses an absolute approximation algorithm.

Theorem 1 *A 2-absolute approximation algorithm exists for planar graph coloring.*

Proof: By the Five Color Theorem, every planar graph is 5-colorable. Further, note that empty graphs (graphs without edges) are 1-colorable, bipartite graphs are 2-colorable, while all other graphs require at least 3 colors. These observations lead to the following algorithm:

1. If the graph is empty or bipartite, color it optimally.

2. Otherwise, color it with 5 colors.

Since this algorithm only uses 5 colors when the optimum number of colors is at least 3, it is a 2-absolute approximation algorithm. ■

We can also consider the problem of **edge-coloring**, in which we color the edges rather than the vertices. Unlike in planar graph coloring, there is no constant upper bound on the number of colors required, since the optimum number $OPT(I)$ is lower-bounded by the maximum vertex degree Δ. Nevertheless, we have the following theorem due to Vizing:

Theorem 2 *The edges of any graph can be colored using at most $\Delta + 1$ colors.*

Since his proof of the theorem is constructive, it provides us with an algorithm for finding an edge-coloring using at most 1 more color than the optimum.

Corollary 1 *A 1-absolute approximation algorithm exists for edge-coloring.*

12.3.2 Proving Negative Examples by Scaling

Although these coloring problems possess absolute approximation algorithms, most NP-hard problems do not. In fact, for most of these problems, we can prove that an absolute approximation algorithm cannot exist unless P equals NP. Such proofs use a technique called **scaling**. In scaling, we first increase (scale) certain parameters of the problem instance. We then show that if an absolute approximation algorithm existed, the solution it would provide for the modified instance could be rescaled to yield an optimum solution for the original instance. But this would imply the existence of an efficient algorithm for an NP-hard problem, and thus P would equal NP.

The following two examples illustrate the use of scaling.

Claim 1 *An absolute approximation algorithm does not exist for the Knapsack problem.*

Proof: Consider a Knapsack problem instance I in which each of the items i has profit p_i, and suppose we have an α-absolute approximation algorithm A for the problem. If we double the profit of each item to $2p_i$ to form a new instance (call it $2I$), the resulting optimum solution $OPT(2I)$ is twice the original optimum solution $OPT(I)$, since the set of items which originally yielded a profit of $OPT(I)$ now yields a profit of $2OPT(I)$. If we then run A on instance $2I$, we obtain a solution of at least $OPT(2I) - \alpha = 2OPT(I) - \alpha$. Finally, dividing this result by 2 yields a solution to the original instance of at least $OPT(I) - \alpha/2$. Thus, we have improved the value of α by a factor of 2.

In general, scaling the original instance I by a factor of r and then dividing the resulting solution of A by r allows us to reduce α to α/r. Hence, if we choose r to be $\lceil 2\alpha \rceil$, we can reduce α to $\alpha/\lceil 2\alpha \rceil \leq 1/2$, implying A can be used to obtain a solution s for I of at least $OPT(I) - 1/2$. If we assume I has integer sizes and profits, the maximum achievable profit $OPT(I)$ is also an integer, and so s must equal $OPT(I)$. Consequently, we have an efficient algorithm for solving integer instances of Knapsack, which contradicts our assumption that P $\neq$ NP. ■

Claim 2 *An absolute approximation algorithm does not exist for Max Independent Set.*

Proof: Suppose we have an α-absolute approximation algorithm A. If we modify an instance I by making a copy of the graph (call this new instance $2I$), the size of the optimum independent set in $2I$ is twice that in I. Thus, $OPT(2I)$ equals $2OPT(I)$, which implies that running A on instance $2I$ yields an independent set of size of at least $OPT(2I) - \alpha = 2OPT(I) - \alpha$. To transform this solution into one for the original I, we count the number of vertices in the independent set of $2I$. One of the graphs must have at least half of these vertices, and thus that graph has an independent set of size at least $(2OPT(I) - \alpha)/2 = OPT(I) - \alpha/2$, implying we have reduced α by a factor of 2. Generalizing this result, if we make $\lceil 2\alpha \rceil$ copies of the graph, we can use A to find an independent set of size at least $OPT(I) - 1/2$ in I. But this must equal $OPT(I)$, since the number of vertices is integral. Thus, we have an efficient algorithm for solving Max Independent Set, which is a contradiction. ■

12.4 Relative Approximation Algorithms

Since absolute approximation algorithms are known to exist for so few optimization problems, a better class of approximation algorithms to consider are relative approximation algorithms. Because they are so commonplace, we will refer to them simply as approximation algorithms.

Definition 8 *An* $\boldsymbol{\alpha}$**-approximation algorithm** *finds a solution of value at most* $\alpha \cdot OPT(I)$.

Note that although α can vary with the size of the input, we will only consider those cases in which it is a constant. To illustrate the design and analysis of an α-approximation algorithm, let us consider the Parallel Machine Scheduling problem, a generic form of load balancing.

Parallel Machine Scheduling: Given m machines m_i and n jobs with processing times p_j, assign the jobs to the machines to minimize the load

$$\max_i \sum_{j \in i} p_j,$$

the time required for all machines to complete their assigned jobs. In scheduling notation, this problem is described as P || Cmax.

A natural way to solve this problem is to use a greedy algorithm called **list scheduling**.

Definition 9 *A* **list scheduling** *algorithm assigns jobs to machines by assigning each job to the least loaded machine.*

Note that the order in which the jobs are processed is not specified. To analyze the performance of list scheduling, we must somehow compare its solution for each instance I (call this solution $A(I)$) to the optimum $OPT(I)$. But we do not know how to obtain an analytical expression for $OPT(I)$. Nonetheless, if we can find a meaningful lower bound $LB(I)$ for $OPT(I)$ and can prove that $A(I) \leq \alpha \cdot LB(I)$ for some α, we then have

$$\begin{aligned} A(I) &\leq \alpha \cdot LB(I) \\ &\leq \alpha \cdot OPT(I). \end{aligned}$$

Using this idea of lower-bounding $OPT(I)$, we can now determine the performance of list scheduling.

Claim 3 *List scheduling is a 2-approximation algorithm for Parallel Machine Scheduling.*

Proof: Consider the following two lower bounds for the optimum load $OPT(I)$:

- the maximum processing time $p = \max_j p_j$
- the average load $L = \sum_j p_j/m$

The maximum processing time p is clearly a lower bound, as the machine to which the corresponding job is assigned requires at least time p to complete its tasks. To see that the average load is a lower bound, note that if all of the machines could complete their assigned tasks in less than time L, the maximum load would be less than the average, which is a contradiction. Now suppose machine m_i has the maximum runtime $c_{\max}$, and let job j be the last job that was assigned to m_i. At the time job j was assigned, m_i must have had the minimum load (call it L_i), since list scheduling assigns each job to the least loaded machine. Thus,

$$\begin{aligned} L_i &\leq \text{average load when } j \text{ assigned} \\ &\leq \text{final average load } L, \end{aligned}$$

since the average load can only increase. Assigning job j to m_i added at most p to L_i, which implies that

$$\begin{aligned} c_{\max} &\leq L_i + p \\ &\leq L + p \\ &\leq 2OPT(I) \quad (L \text{ and } p \text{ are lower bounds for } OPT(I)). \end{aligned}$$

The solution returned by list scheduling is $c_{\max}$, and thus list scheduling is a 2-approximation algorithm for Parallel Machine Scheduling. ■

It is possible to show that by modifying list scheduling to assign the jobs in decreasing order of processing time, we obtain a 4/3-approximation algorithm. Further, note that list scheduling is an online algorithm. Newer online algorithms are able to achieve an α of about 1.8.

12.5 Polynomial Approximation Schemes

The obvious question to now ask is how good an α we can obtain.

Definition 10 *A* **polynomial approximation scheme (PAS)** *is a set of algorithms $\{A_\epsilon\}$ for which each A_ϵ is a polynomial-time $(1+\epsilon)$-approximation algorithm.*

Thus, given any $\epsilon > 0$, a PAS provides an algorithm that achieves a $(1+\epsilon)$-approximation. How do we devise a PAS? The most common method used is k-enumeration.

Definition 11 *An approximation algorithm using* **k-enumeration** *finds an optimal solution for the k most important elements in the problem and then uses an approximate polynomial-time method to solve the remainder of the problem.*

For example, an approximation algorithm which uses k-enumeration to solve Parallel Machine Scheduling is as follows:

1. Enumerate all possible assignments of the k largest jobs.
2. For each of these partial assignments, list schedule the remaining jobs.
3. Return as the solution the assignment with the minimum load.

Note that in enumerating all possible assignments of the k largest jobs, the algorithm will always find the optimal assignment for these jobs. The following claim demonstrates that this algorithm provides us with a PAS.

Claim 4 *For any fixed m, k-enumeration yields a polynomial approximation scheme for Parallel Machine Scheduling.*

Proof: As in the proof of Claim 3, let us consider the machine m_i with maximum runtime $c_{\max}$ and the last job j that m_i was assigned. If this job is not among the k largest, it was assigned during list scheduling, at which point there were at least k larger jobs which had already been scheduled. Thus, when job j was assigned, the average load (call it L_{assigned}) must have been at least kp_j/m, which implies that

$$\begin{aligned} p_j &\leq \frac{mL_{\text{assigned}}}{k} \\ &\leq \frac{mL}{k}. \end{aligned}$$

Since $c_{\max}$ is the sum of p_j and the load on m_i before job j was assigned (which was shown in the proof of Claim 3 to be at most L), we have

$$\begin{aligned} c_{\max} &\leq L + p_j \\ &\leq (1 + \frac{m}{k})L \\ &\leq (1 + \frac{m}{k})OPT(I). \end{aligned}$$

Given an $\epsilon > 0$, if we let k equal m/ϵ,

$$c_{\max} \leq (1+\epsilon)OPT(I).$$

This bound on $c_{\max}$ also holds if job j is among the k largest. In this case, job j is scheduled optimally, and $c_{\max}$ thus equals $OPT(I)$. Finally, to determine the running time of the algorithm, note that because each of the k largest jobs can be assigned to any of the m machines, there are $m^k = m^{m/\epsilon}$ possible assignments of these jobs. Since the list scheduling performed for each of these assignments takes $O(n)$ time, the total running time is $O(nm^{m/\epsilon})$, which is polynomial because m is fixed. Thus, given an $\epsilon > 0$, the algorithm is a $(1+\epsilon)$-approximation, and so we have a polynomial approximation scheme. ■

Obviously, we would prefer an approximation algorithm for which m does not have to be fixed. As a first step towards achieving this goal, let us reconsider Parallel Machine Scheduling when there are only k possible sizes (or types) of jobs, where k is bounded by a constant. In this case, it is possible to find an optimum solution in polynomial time using dynamic programming. Note that each set of jobs can be described by its "profile," the number of each type of job, and the number of possible profiles is at most n^k, which is polynomial in the input size. The dynamic program computes the function $M(a_1, a_2, \ldots, a_k)$, the minimum number of machines needed to complete the a_i type-i jobs in some fixed time T. After enumerating the set X of all profiles which can be completed by a single machine in time T and using X to initialize the appropriate entries in the table, the remaining entries are computed using

$$M(a_1, a_2, \ldots, a_k) = 1 + \min_{(x_1, x_2, \ldots, x_k) \in X} M(a_1 - x_1, a_2 - x_2, \ldots, a_k - x_k).$$

Thus, the minimum number of machines required to complete a profile y is found by exhaustively removing all possible single-machine profiles from y and looking up the minimum number of machines required to complete the rest of y. Finally, to determine the optimum time required to complete all of the jobs, the dynamic program can be used as a "subroutine" in a binary search over the values of T.