CS 583: Approximation Algorithms: Introduction*

Chandra Chekuri

January 15, 2018

1 Introduction

Course Objectives

1. To appreciate that not all intractable problems are the same. **NP** optimization problems, identical in terms of exact solvability, can appear very different from the approximation point of view. This sheds light on why, in practice, some optimization problems (such as KNAPSACK) are easy, while others (like CLIQUE) are extremely difficult.
2. To learn techniques for design and analysis of approximation algorithms, via some fundamental problems.
3. To build a toolkit of broadly applicable algorithms/heuristics that can be used to solve a variety of problems.
4. To understand reductions between optimization problems, and to develop the ability to relate new problems to known ones.

The complexity class **P** contains the set of problems that can be solved in polynomial time. From a theoretical viewpoint, this describes the class of tractable problems, that is, problems that can be solved efficiently. The class **NP** is the set of problems that can be solved in *non-deterministic* polynomial time, or equivalently, problems for which a solution can be *verified* in polynomial time. **NP** contains many interesting problems that often arise in practice, but there is good reason to believe $\mathbf{P} \neq \mathbf{NP}$. That is, it is unlikely that there exist algorithms to solve **NP** optimization problems efficiently, and so we often resort to heuristic methods to solve these problems.

Heuristic approaches include backtrack search and its variants, mathematical programming methods, local seach, genetic algorithms, tabu search, simulated annealing etc. Some methods are guaranteed to find an optimal solution, though they may take exponential time; others are guaranteed to run in polynomial time, though they may not return an optimal solution. Approximation algorithms fall in the latter category; however, though they do not find an optimal solution, we can give guarantees on the *quality* of the solution found.

*Based on previous scribed lecture notes of Nitish Korula and Sungjin Im.

Approximation Ratio

To give a guarantee on solution quality, one must first define what we mean by the quality of a solution. We discuss this more carefully in the next lecture; for now, note that each *instance* of an optimization problem has a set of feasible solutions. The optimization problems we consider have an **objective function** which assigns a (real/rational) number/value to each feasible solution of each instance I. The goal is to find a feasible solution with minimum objective function value or maximum objective function value. The former problems are minimization problems and the latter are maximization problems.

For each instance I of a problem, let $\mathrm{OPT}(I)$ denote the value of an optimal solution to instance I. We say that an algorithm $\mathcal{A}$ is an α-approximation algorithm for a problem if, for *every* instance I, the value of the feasible solution returned by $\mathcal{A}$ is within a (multiplicative) factor of α of $\mathrm{OPT}(I)$. Equivalently, we say that $\mathcal{A}$ is an approximation algorithm with *approximation ratio* α. For a minimization problem we would have $\alpha \geq 1$ and for a maximization problem we would have $\alpha \leq 1$. However, it is not uncommon to find in the literature a different convention for maximization problems where one says that $\mathcal{A}$ is an α-approximation algorithm if the value of the feasible solution returned by $\mathcal{A}$ is at least $\frac{1}{\alpha} \cdot \mathrm{OPT}(I)$; the reason for using convention is so that approximation ratios for both minimization and maximization problems will be ≥ 1. In this course we will for the most part use the convention that $\alpha \geq 1$ for minimization problems and $\alpha \leq 1$ for maximization problems.

Remarks:

1. The approximation ratio of an algorithm for a minimization problem is the *maximum* (or supremum), over all instances of the problem, of the ratio between the values of solution returned by the algorithm and the optimal solution. Thus, it is a bound on the *worst-case* performance of the algorithm.

2. The approximation ratio α can depend on the size of the instance I, so one should technically write $\alpha(|I|)$.

3. A natural question is whether the approximation ratio should be defined in an *additive* sense. For example, an algorithm has an α-approximation for a minimization problem if it outputs a feasible solution of value at most $\mathrm{OPT}(I) + \alpha$ for all I. This is a valid definition and is the more relevant one in some settings. However, for many **NP** problems it is easy to show that one cannot obtain any interesting additive approximation (unless of course $P = NP$) due to scaling issues. We will illustrate this via an example later.

Pros and cons of the approximation approach:

Some advantages to the approximation approach include:

1. It explains why problems can vary considerably in difficulty.

2. The analysis of problems and problem instances distinguishes easy cases from difficult ones.

3. The worst-case ratio is *robust* in many ways. It allows *reductions* between problems.

4. Algorithmic ideas/tools are valuable in developing heuristics, including many that are practical and effective.

As a bonus, many of the ideas are beautiful and sophisticated, and involve connections to other areas of mathematics and computer science.

Disadvantages include:

1. The focus on *worst-case measures* risks ignoring algorithms or heuristics that are practical or perform well *on average*.
2. Unlike, for example, integer programming, there is often no incremental/continuous tradeoff between the running time and quality of solution.
3. Approximation algorithms are often limited to cleanly stated problems.
4. The framework does not (at least directly) apply to decision problems or those that are inapproximable.

Approximation as a broad lens

The use of approximation algorithms is not restricted solely to **NP**-Hard optimization problems. In general, ideas from approximation can be used to solve many problems where finding an exact solution would require too much of any resource.

A resource we are often concerned with is *time*. Solving **NP**-Hard problems exactly would (to the best of our knowledge) require exponential time, and so we may want to use approximation algorithms. However, for large data sets, even polynomial running time is sometimes unacceptable. As an example, the best exact algorithm known for the MATCHING problem in general graphs requires $O(n^3)$ time; on large graphs, this may be not be practical. In contrast, a simple greedy algorithm takes near-linear time and outputs a matching of cardinality at least $1/2$ that of the maximum matching; moreover there have been randomized sub-linear time algorithms as well.

Another often limited resource is *space*. In the area of data streams/streaming algorithms, we are often only allowed to read the input in a single pass, and given a small amount of additional storage space. Consider a network switch that wishes to compute statistics about the packets that pass through it. It is easy to exactly compute the average packet length, but one cannot compute the median length exactly. Surprisingly, though, many statistics can be approximately computed.

Other resources include programmer time (as for the MATCHING problem, the exact algorithm may be significantly more complex than one that returns an approximate solution), or communication requirements (for instance, if the computation is occurring across multiple locations).

2 Formal Aspects

2.1 NP Optimization Problems

In this section, we cover some formal definitions related to approximation algorithms. We start from the definition of optimization problems. A problem is simply an infinite collection of *instances*. Let Π be an optimization problem. Π can be either a minimization or maximixation problem. Instances I of Π are a subset of Σ^* where Σ is a finite encoding alphabet. For each instance I there is a set of feasible solutions $\mathcal{S}(I)$. We restrict our attention to real/rational-valued optimization problems; in these problems each feasible solution $S \in \mathcal{S}(I)$ has a value $val(S, I)$. For a minimization problem Π the goal is, given I, find $\mathrm{OPT}(I) = \min_{S \in \mathcal{S}(I)} val(S, I)$.

Now let us formally define NP optimization (NPO) which is the class of optimization problems corresponding to NP.

Definition 1 Π *is in NPO if*

- *Given $x \in \Sigma^*$, there is a polynomial-time algorithm that decide if x is a valid instance of Π. That is, we can efficiently check if the input string is well-formed. This is a basic requirement that is often not spelled out.*
- *For each I, and $S \in \mathcal{S}(I)$, $|S| \leq poly(|I|)$. That is, the solution are of size polynomial in the input size.*
- *There exists a poly-time decision procedure that for each I and $S \in \Sigma^*$, decides if $S \in \mathcal{S}(I)$. This is the key property of NP; we should be able to* verify *solutions efficiently.*
- *$val(I, S)$ is a polynomial-time computable function.*

We observe that for a minimization NPO problem Π, there is a associated natural decision problem $L(\Pi) = \{(I, B) : \mathrm{OPT}(I) \leq B\}$ which is the following: given instance I of Π and a number B, is the optimal value on I at most B? For maximization problem Π we reverse the inequality in the definition.

Lemma 2 $L(\Pi)$ *is in NP if Π is in NPO.*

2.2 Relative Approximation

When Π is a minimization problem, recall that we say an approximation algorithm $\mathcal{A}$ is said to have approximation ratio α iff

- $\mathcal{A}$ is a polynomial time algorithm
- for all instance I of Π, $\mathcal{A}$ produces a feasible solution $\mathcal{A}(I)$ s.t. $val(\mathcal{A}(I), I) \leq \alpha\, val\,(\mathrm{OPT}(I), I)$. (Note that $\alpha \geq 1$.)

Approximation algorithms for maximization problems are defined similarly. An approximation algorithm $\mathcal{A}$ is said to have approximation ratio α iff

- $\mathcal{A}$ is a polynomial time algorithm
- for all instance I of Π, $\mathcal{A}$ produces a feasible solution $\mathcal{A}(I)$ s.t. $val(\mathcal{A}(I), I) \geq \alpha\, val\,(\mathrm{OPT}(I), I)$. (Note that $\alpha \leq 1$.)

For maximization problems, it is also common to see use $1/\alpha$ (which must be ≥ 1) as approximation ratio.

2.3 Additive Approximation

Note that all the definitions above are about relative approximations; one could also define *additive* approximations. $\mathcal{A}$ is said to be an α-additive approximation algorithm, if for all I, $val(\mathcal{A}(I)) \leq \mathrm{OPT}(I) + \alpha$. Most NPO problems, however, do not allow any additive approximation ratio because $\mathrm{OPT}(I)$ has a scaling property.

To illustrate the scaling property, let us consider Metric-TSP. Given an instance I, let I_β denote the instance obtained by increasing all edge costs by a factor of β. It is easy to observe that for each $S \in \mathcal{S}(I) = \mathcal{S}(I_\beta)$, $val(S, I_\beta) = \beta val(S, I_\beta)$ and $\mathrm{OPT}(I_\beta) = \beta \mathrm{OPT}(I)$. Intuitively, scaling edge by a factor of β scales the value by the same factor β. Thus by choosing β sufficiently large, we can essentially make the additive approximation(or error) negligible.

Lemma 3 *Metric-TSP does not admit an α additive approximation algorithm for any polynomial-time computable α unless $P = NP$.*

Proof: For simplicity, suppose every edge has integer cost. For the sake of contradiction, suppose there exists an additive α approximation $\mathcal{A}$ for Metric-TSP. Given I, we run the algorithm on I_β and let S be the solution, where $\beta = 2\alpha$. We claim that S is the optimal solution for I. We have $val(S, I) = val(S, I_\beta)/\beta \leq \mathrm{OPT}(I_\beta)/\beta + \alpha/\beta = \mathrm{OPT}(I) + 1/2$, as $\mathcal{A}$ is α-additive approximation. Thus we conclude that $\mathrm{OPT}(I) = val(S, I)$, since $\mathrm{OPT}(I) \leq val(S, I)$, and $\mathrm{OPT}(I)$, $val(S, I)$ are integers. This is impossible unless $P = NP$. □

Now let us consider two problems which allow additive approximations. In the Planar Graph Coloring, we are given a planar graph $G = (V, E)$. We are asked to color all vertices of the given graph G such that for any $vw \in E$, v and w have different colors. The goal is to minimize the number of different colors. It is known that to decide if a planar graph admits 3-coloring is NP-complete [4], while one can always color any planar graph G with using 4 colors (this is the famous 4-color theorem) [1, 5]. Further, one can efficiently check whether a graph is 2-colorable (that is, if it is bipartite). Thus, the following algorithm is a 1-additive approximation for Planar Graph Coloring: If the graph is bipartite, color it with 2 colors; otherwise, color with 4 colors.

As a second example, consider the Edge Coloring Problem, in which we are asked to color edges of a given graph G with the minimum number of different colors so that no two adjacent edges have different colors. By Vizing's theorem [6], we know that one can color edges with either $\Delta(G)$ or $\Delta(G) + 1$ different colors, where $\Delta(G)$ is the maximum degree of G. Since $\Delta(G)$ is a trivial lower bound on the minimum number, we can say that the Edge Coloring Problem allows a 1-additive approximation. Note that the problem of deciding whether a given graph can be edge colored with $\Delta(G)$ colors is NP-complete [2].

2.4 Hardness of Approximation

Now we move to hardness of approximation.

Definition 4 (Approximability Threshold) *Given a minimization optimization problem Π, it is said that Π has an approximation threshold $\alpha^*(\Pi)$, if for any $\epsilon > 0$, Π admits a $\alpha^*(\Pi) + \epsilon$ approximation but if it admits a $\alpha^*(\Pi) - \epsilon$ approximation then $P = NP$.*

If $\alpha^*(\Pi) = 1$, it implies that Π is solvable in polynomial time. Many NPO problems Π are known to have $\alpha^*(\Pi) > 1$ assuming that $P \neq NP$. We can say that approximation algorithms try to decrease the upper bound on $\alpha^*(\Pi)$, while hardness of approximation attempts to increase lower bounds on $\alpha^*(\Pi)$.

To prove hardness results on NPO problems in terms of approximation, there are largely two approaches; a direct way by reduction from NP-complete problems and an indirect way via gap reductions. Here let us take a quick look at an example using a reduction from an NP-complete problem.

In the (metric) k-center problem, we are given an undirected graph $G = (V, E)$ and an integer k. We are asked to choose a subset of k vertices from V called centers. The goal is to minimize the maximum distance to a center, i.e. $\min_{S \subseteq V, |S|=k} \max_{v \in V} \text{dist}_G(v, S)$, where $\text{dist}_G(v, S) = \min_{u \in S} \text{dist}_G(u, v)$.

The k-center problem has approximation threshold 2, since there are a few 2-approximation algorithms for k-center and there is no $2-\epsilon$ approximation algorithm for any $\epsilon > 0$ unless $P = NP$. We can prove the inapproximability using a reduction from the decision version of Dominating Set: Given an undirected graph $G = (V, E)$ and an integer k, does G have a dominating set of size at most k? A set $S \subseteq V$ is said to be a dominating set in G if for all $v \in V$, $v \in S$ or v is adjacent to some u in S. Dominating Set is known to be NP-complete.

Theorem 5 ([3]) *Unless $P = NP$, there is no $2-\epsilon$ approximation for k-center for any fixed $\epsilon > 0$.*

Proof: Let I be an instance of Dominating Set Problem consisting of graph $G = (V, E)$ and integer k. We create an instance I' of k-center while keeping graph G and k the same. If I has a dominating set of size k then $\text{OPT}(I') = 1$, since every vertex can be reachable from the Dominating Set by at most one hop. Otherwise, we claim that $\text{OPT}(I') \geq 2$. This is because if $\text{OPT}(I') < 2$, then every vertex must be within distance 1, which implies the k-center that witnesses $\text{OPT}(I')$ is a dominating set of I. Therefore, the $(2 - \epsilon)$ approximation for k-center can be used to solve the Dominating Set Problem. This is impossible, unless $P = NP$. □

3 Designing Approximation Algorithms

How does one design and more importantly analyze the performance of approximation algorithms? This is a non-trivial task and the main goal of the course is to expose you to basic and advanced techniques as well as central problems. The purpose of this section is to give some high-level insights. We start with how we design polynomial-time algorithms. Note that approximation makes sense mainly in the setting where one can find a feasible solution relatively easily but finding an *optimum* solution is hard. In some cases finding a feasible solution itself may involve some non-trivial algorithm, in which case it is useful to properly understand the structural properties that guarantee feasibility, and then build upon it.

Some of the standard techniques we learn in basic and advanced undergraduate algorithms courses are recursion based methods such as divide and conquer, dynamic programming, greedy, local search, combinatorial optimization via duality, and reductions to existing problems. How do we adapt these to the approximation setting? Note that intractability implies that there are no efficient characterizations of the optimum solution value.

Greedy and related techniques are often fairly natural for many problems and simple heuristic algorithms often suggest themselves for many problems. (Note that the algorithms may depend on being able to solve some existing problem efficiently. Thus, knowing a good collection of general poly-time solvable problems is often important.) The main difficulty is in analyzing their performance. The key challenge here is to identify appropriate *lower bounds* on the optimal value (assuming that the problem is a minimization problem) or *upper bounds* on the optimal value (assuming that the problem is a maximization problem). These bounds allow one to compare the output of the algorithm and prove an approximation bound. In designing poly-time algorithms we often prove that greedy algorithms do not work. We typically do this via examples. This skill is also useful in proving that some candidate algorithm does *not* give a good approximation. Often the bad examples lead one to a new algorithm.

How does one come up with lower or upper bounds on the optimum value? This depends on the problem at hand and knowing some background and related problems. However, one would like to find some automatic ways of obtaining bounds. This is often provided via linear programming relaxations and more advanced convex programming methods including semi-definite programming, lift-and-project hierarchies etc. The basic idea is quite simple. Since integer linear programming is NP-Complete one can formulate most discrete optimization problems easily and "naturally" as an integer program. Note that there may be many different ways of expressing a given problem as an integer program. Of course we cannot solve the integer program but we can solve the linear-programming relaxation which is obtained by removing the integrality constraints on the variables. Thus, for each instance I of a given problem we can obtain an LP relaxation $LP(I)$ which we typically can be solve in polynomial-time. This automatically gives a bound on the optimum value since it is a relaxation. How good is this bound? It depends on the problem, of course, and also the specific LP relaxation. How do we obtain a feasible solution that is close to the bound given by the LP relaxation. The main technique here is to *round* the fractional solution x to an integer feasible solution x' such that x''s value is close to that of x. There are several non-trivial rounding techniques that have been developed over the years that we will explore in the course. We should note that in several cases one can analyze combinatorial algorithms via LP relaxations even though the LP relaxation does not play any direct role in the algorithm itself. Finally, there is the question of which LP relaxation to use. Often it is required to "strengthen" an LP relaxation via addition of constraints to provide better bounds. There are some automatic ways to strengthen any LP and often one also needs problem specific ideas.

Local search is another powerful technique and the analysis here is not obvious. One needs to relate the value of a local optimum to the value of a global optimum via various *exchange* properties which define the local search heuristic. For a formal analysis it is necessary to have a good understanding of the problem structure.

Finally, dynamic programming plays a key role in the following way. Its main use is in solving to optimality a *restricted* version of the given problem or a subroutine that is useful as a building block. How does one obtain a restricted version? This is often done by some clever proprocessing of a given instance.

Reductions play a very important role in both designing approximation algorithms and in proving inapproximability results. Often reductions serve as a starting point in developing a simple and crude heuristic that allows one to understand the structure of a problem which then can lead to further improvements.

Discrete optimization problems are brittle — changing the problem a little can lead to substantial changes in the complexity and approximability. Nevertheless it is useful to understand problems and their structure in broad categories so that existing results can be leveraged quickly and robustly. Thus, some of the emphasis in the course will be on classifying problems and how various parameters influence the approximability.

References

[1] Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. part i: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[2] Ian Holyer. The np-completeness of edge-coloring. *SIAM Journal on computing*, 10(4):718–720, 1981.

[3] Wen-Lian Hsu and George L Nemhauser. Easy and hard bottleneck location problems. *Discrete Applied Mathematics*, 1(3):209–215, 1979.

[4] Larry Stockmeyer. Planar 3-colorability is polynomial complete. *SIGACT News*, 5(3):19–25, July 1973.

[5] Robin Thomas. An update on the four-color theorem. *Notices of the AMS*, 45(7):848–859, 1998.

[6] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.