CSE 632: Analysis of Algorithms II **Fall 2017**

Lecture 1 (08/30/2017): Introduction

Lecturer: Shi Li *Scribe: Shi Li*

1.1 Introduction

Please see the course webpage `www.cse.buffalo.edu/~shil/courses/CSE632` regarding the administrative information of the course.

In the CSE431/531 course, we have learned the complexity class **P** and **NP** and the notion of NP-completeness and NP-hardness. Unfortunately, many natural problems are NP-hard. If $\mathbf{P} \neq \mathbf{NP}$, then for these problems, we can not find an algorithm that

1. in polynomial time (as we argued in the definition of **P**, we shall say an algorithm is efficient if it runs in polynomial time),
2. finds optimum solution,
3. for any instance.

Thus, one of the three requirements have to be relaxed when we handle a NP-hard problem. This leads to three different categories of approaches to handle NP-hard problems.

1. **Relaxing requirement 3: for any instance.** An analysis showing an algorithm works for any instance is called "worst-case analysis". To relax a worst-case analysis, we can study special cases of the problems for which efficient algorithms exist. However this often requires the cases studied to be very special and thus too restricted. If a special case is also NP-hard, and we are applying the worst-case analysis within this special case, then this does not lead to a new approach to handle NP-hard problems.

   Another approach of this type is to consider "random inputs": we assume the input instance comes from some distribution and we only require the algorithm works for most of the instances in this distribution. Recently many problems have been studied using approach. However, it is often not known what the right distribution for input instances is. For many problems, assuming that the distribution is the uniform distribution over all input instances is not realistic and makes the problems easy to solve.

2. **Remove requirement 1: algorithm is efficient.** In this category, we need to find algorithms that find optimum solutions for all instances, but not necessarily in polynomial time. This approach has been taken by those in the field of operations research who solve integer programming formulations of optimization problems (this has lead to many generic techniques for solving IP such as cutting-plane generation and branch-and-bound), or those in the area of artificial intelligence who use A^*-search to explore the full set of solutions in a clever way.

   Fast exponential time algorithms have been studied in the field of theoretical computer science. In this approach, we are trying to find faster and faster algorithms for solving NP-hard problems, even though the algorithms run in exponential time. For example, one may try to improve the running time from $2^{O(n)}$ to $2^{O(\sqrt{n})}$, or $2^{O(n^{1/3})}$. Even within the $2^{O(n)}$ range, a 1.5^n-time algorithm is much faster than a 2^n-time algorithm. For example, the trivial algorithm for 3SAT runs in time $2^n\text{poly}(n)$, and the current best randomized algorithm for 3SAT runs in expected time $1.321^n\text{poly}(n)$.

3. **Remove requirement 2: finds optimum solutions.** This is by far the most common approach. Our goal is to find efficient algorithms that find solutions that are good enough for all instances. There has been an enormous study of various types of heuristics and meta-heuristics such as simulated annealing, genetic algorithms, and tabu search. These techniques often yield good results in practice.

   The focus of this course is the "approximation algorithm" framework for optimization problems, whose goal is to find a solution that minimize or maximize an objective function. We try to find a solution that closely approximates the optimal solution in terms of its objective value. How well an approximation algorithm performs is measured by its approximation ratio.

Definition 1.1 *An α-approximation algorithm for an optimization problem is a polynomial- time algorithm that for all instances of the problem produces a solution whose value is within a factor of α of the value of an optimal solution.*

For minimization problems, the approximation ratio α is always at least 1: an algorithm is an α-approximation algorithm if it finds a solution whose cost is at most α times the optimal cost. For maximization problems, we shall use the convention that the approximation ratio α is at most 1: an algorithm is α-approximation algorithm if it finds a solution whose value is at least α times the optimal value. Sometimes, the approximation ratio of an algorithm for a maximization problem is defined as $\frac{\text{value(optimum solution)}}{\text{value(solution we found)}}$, which is at least 1; however, we shall not use this definition in the course.

Why do we study approximation algorithms?

- Because we need to solve NP-hard optimization problems. Approximation algorithms will provide useful insights for designing heuristics to solve practical problems.
- It provides a method to measure the levels of difficulty of various NP-hard problems.
- The worst case analysis is robust: it gives guaranteed quality for all scenarios.
- The notion of approximation appears everywhere in many other areas (linear and sub-linear time algorithms, streaming and online algorithms, property testing and probabilistic learning methods), and the techniques developed in studying approximation algorithms can be applied to these areas.
- It is fun since many of the ideas in designing approximation algorithms are mathematically elegant.

However, there are also some disadvantages when one tries to apply approximation algorithms directly to practical problems:

- The framework only applies to optimization problems with specified objectives. It does not applied to, for example, decision problems, and problems without objective functions (some problems from machine learning are of this type).
- Though the worst case analysis is robust, sometimes it is too pessimistic. The worst approximation ratio may come from some pathological cases that rarely happen in practice. Focusing

on such cases may ignore algorithms that perform well for practical instances.

- There is often no smooth tradeoff between running time of algorithms and the guaranteed approximation ratios. For most problems, we can guarantee some α-approximation ratio with a polynomial time algorithm, but to improve the α-approximation ratio, we need to use exponential time algorithms.
- It is often limited to clean problems. To apply the framework to practical problems, one need to remove many side constraints and identify the hard cores of these problems.