Introduction to Approximation Algorithms

CSCI 1440/2440

2025-10-08

We informally define the complexity classes P and NP, and state the conjecture that P $\neq$ NP. We then introduce approximation algorithms, and a recipe for analyzing them.

1 *P versus NP*

After a hard day's work solving CSCI 1440/2440 homework problems, you can't wait to see the latest cute dog[1] memes on Facebook. You proceed by opening your favorite browser, and visiting `facebook.com`. You enter your login and password, and seconds later, you are ready for a healthy dose of memes. Life is good!

On the contrary, life would not be so great if, after entering your credentials (login and password), you had to wait several minutes (or maybe hours, or days!) to access your account. Likewise, you wouldn't be very happy if a program that enumerates all possible credentials could find yours, in a relatively short period of time. If that were the case, then your account (and mine) would be very easy to hack, and Facebook wouldn't know if you prefer cat or dog memes—a most unfortunate situation!

Note that, checking whether your credentials are valid (Facebook's job) is an easy task. On the other hand, blindly trying all possible combinations could be an impossible task (i.e., it could take forever!). With an alphabet of just 52 letters, there are something like 10^{82} possible credentials,[2] more than the number of atoms in the universe![3]

The guess-and-check structure of this problem is at the heart of the P versus NP problem, the most celebrated open question in computer science. This is such a fundamental and important problem that you could win a million dollars if you were to solve it.[4]

In a nutshell, a problem is in the class NP (nondeterministic polynomial time), if verifying a proposed solution to the problem is easy, but generating such a solution is not. In contrast, a problem is in P (polynomial time) if both verifying and generating a solution is easy. Here "easy" means, can be accomplished in polynomial time.[5]

Intuitively, it might seem that validating vs. generating solutions (e.g., credentials) are fundamentally different computational tasks. If you feel this way, then you are in the same boat as the majority of computer scientists today, who conjecture that P $\neq$ NP. But are these fundamentally different computational problems; or has sufficient ingenuity merely eluded us thus far? This is the P vs. NP question.

[1] Or maybe cat; we are not judging!

[2] Assuming logins and passwords must be between 5 and 25 characters, ignoring duplicate constraints (i.e., that no two logins can be the same).

[3] There are an estimated 10^{80} atoms in the universe. `http://www.wolframalpha.com/input/?i=number+of+atoms+in+universe`

[4] `http://www.claymath.org/millennium-problems/p-vs-np-problem`

[5] We recognize that our explanation is circular. We are not defining these classes formally; we are only providing informal intuition about them.

2 *Algorithmic Design Goals*

If it turns out that P $\neq$ NP, then for so-called NP-hard optimization problems, it will not be possible to design algorithms that satisfy all three of the following desiderata:

1. solve all problem instances
2. run in polynomial time
3. compute optimal solutions

So, we expect to have to relax at least one of these three criteria. Different relaxations lead to different algorithmic techniques.

If we have reason to believe our algorithm need only handle particular inputs, then a sensible approach is to relax requirement 1, and develop algorithms that handle these special cases only.

If instead, we are only willing to relax requirement 2, i.e., not require polynomial-time solutions, then one might try to exploit the structure of the problem to design an optimal algorithm (e.g., design admissible heuristics for a branch and bound search). However, this approach does not *guarantee* that a solution will be returned in a reasonable amount of time, even if the algorithm tends to works very well in practice (i.e., on common problem instances).

Relaxing requirement 3 means designing algorithms that are not guaranteed to return optimal solutions. There are two prevalent ways to relax this requirement. The first way, which is typical of the AI community, is to design an algorithm—usually called a **heuristic**—that runs fast (on all inputs) but does not necessarily yield optimal solutions, and then to empirically evaluate the quality of its solutions: i.e., how suboptimal they are. Like the first two approaches, when the empirical tests reflect reality, this approach translates into good results in practice, as heuristics can be tailored to the problem instances on which they are tested.

Another approach, which is typical of the (CS) theory community, is to design an **approximation algorithm**, i.e., an algorithm whose solution quality is *guaranteed* to relate—somehow—to the optimal solution regardless of the input: i.e., *in the worst case*. We will explore this latter approach in this lecture.

3 *Recipe for Analyzing Approximation Algorithms*

Like the design of any algorithm, the design of an approximation algorithm is an art.[6] One needs to think hard about the problem at hand, construct a reasonable algorithm, and then exploit any apparent structure to carry out an analysis. That said, there is a scientific

[6] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011

component to the analysis, in that there is a generic recipe that is often followed to make progress. The recipe is as follows:

Call the value of the optimal solution OPT, and the value of the solution obtained by the approximation algorithm APX.

1. Upper bound the value of OPT by UB, perhaps by analyzing some simpler algorithm that is guaranteed to perform better than (or as well as) OPT: i.e., $\text{UB} \geq \text{OPT}$.
2. Lower bound the value of APX by LB, perhaps by analyzing some simpler algorithm that is guaranteed to perform worse than (or as well as) APX: i.e., $\text{APX} \geq \text{LB}$.
3. Since $\text{OPT} \geq \text{APX}$, it follows that $\text{UB} \geq \text{OPT} \geq \text{APX} \geq \text{LB}$. So:

$$\frac{\text{APX}}{\text{OPT}} \geq \frac{\text{LB}}{\text{OPT}} \geq \frac{\text{LB}}{\text{UB}}$$

   The holy grail in approximation algorithm research is to design an algorithm for which $\frac{\text{LB}}{\text{UB}}$ is a constant, meaning it is independent of the problem's inputs. In such cases, we say that we have found a **constant-factor** approximation algorithm. E.g., the constant is α, then we conclude that $\text{APX} \geq \alpha\,\text{OPT}$.

4 A Simple Example

The alternative to a constant-factor approximation is a factor that grows as some part of the problem description grows: e.g., the number of bidders, the number of goods, etc. In this section, we analyze a simple approximation mechanism—a lottery—which is arguably too simple, as it yields a linear, not a constant, factor approximation.

Recall that the second-price auction is welfare-maximizing. In particular, the welfare it achieves is $v_{(1)}$, the value of the highest bidder. Thus, $\text{OPT} = v_{(1)}$.

As an alternative, consider a lottery, which allocates one good at random to one of n bidders.[7] We denote the welfare of a lottery by APX. On average,

[7] participants, I suppose

$$\text{APX} = \frac{1}{n}\sum_{i=1}^{n} v_i$$

Lower bound. As $\sum_{i=1}^{n} v_i \geq \max_{i\in[n]} v_i = v_{(1)}$, the lottery always achieves welfare at least $1/n(v_{(1)})$.

The lottery is thus a non-constant approximation of the optimal, as the approximation ratio is linear in the number of bidders:

$$\frac{\text{APX}}{\text{OPT}} = \frac{\left(\frac{1}{n}\right)\sum_{i=1}^{n} v_i}{v_{(1)}} \geq \frac{\left(\frac{1}{n}\right) v_{(1)}}{v_{(1)}} = \frac{1}{n}$$

Tight lower bound This linear approximation ratio is the best possible for the lottery, because not only is the upper bound on OPT *tight* (i.e., $\text{OPT} = v_{(1)}$), but moreover, our lower bound on APX is also tight, as the following example shows.

Consider values s.t. $v_1 = 1$ and $v_i = 0$, for all $i \in \{2, \ldots, n\}$. In this case, the lottery yields expected welfare of

$$\text{APX} = \left(\frac{1}{n}\right)(1) + \left(\frac{n-1}{n}\right)(0) = \frac{1}{n}.$$

References

[1] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.