CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 25: Conclusions

Harvard SEAS - Fall 2024 *2024-12-03*

1 Announcements

- Anurag OH after class; Salil no OH this Thurs.
- Fill out Q evaluation.
- Problem set 9 due tomorrow 12/4; see Salil's Ed post for a definitional correction.
- Problem sets 8/9 revision videos and late revision videos (for half credit) due 12/11.
- Final exam 12/19, 9am. Cheat sheet allowed; stay tuned for details.
- Stay tuned for Ed post about review sessions, extra OH/sections, practice exams.
- Votes on T-shirt designs due 12/4.

Recommended Reading:

- Roughgarden IV, Epilogue
- MacCormick, Chapter 18

2 An Algorithmicist's Workflow

When confronted with a real-world algorithmic problem (like Web Search, Interval Scheduling, Population demographics, Shortest routes on maps, ChessboardWithBombs, Register Allocation, Wireless spectrum allocation, Kidney Exchange, Lyber, Arithmetic Overflow, etc.), you can tackle it using the skills from CS1200 (and future classes) by looping through the following steps:

1. Mathematically model
    - What kinds of **input data**? Key-value pairs (CS1650), graphs and combinatorial structures (AM107, Math 152), logical formulas (Math 141A, Phil 144), programs (CS 1520, CS 1530), ...
    - What kind of **output/queries/objective function**?
    - Is it a **one-shot** computational problem or **data structure** problem? How many queries will be made? Does it need dynamic updating or can it be static? For instance, Static Predecessor, Dynamic dictionaries. (CS 1240)
    - How much **details**/structure to incorporate? We have seen many cases where incorporating a restriction makes the problem easier. For instance, SAT vs 2-SAT, Bipartite Matching vs Matching, directed vs undirected graphs, unlimited universe vs bounded universe for sorting, edge weights vs unweighted graphs...)

- Social context and **EthiCS**? (CS37, CS79, CS1050, CS1260, CSx3xx, CS1780, CS2080, CS2260, CS 2261, CS2380)
- What is the right **computational model** and relevant **resources** to consider? (CS61, CS1210, CS14xx, CS15xx, CS16xx)

2. Look for related problems (in class, in the literature, on the web) and try to obtain an algorithm by **reduction to** other problems

   - IntervalScheduling, AreaOfConvexPolygon reducing to Sorting
   - Duplicate Search reducing to Dynamic dictionaries
   - Bipartite Matching/ ChessBoardWithBombs/ 2-Coloring reducing to Shortest Paths
   - Independent Set/Coloring/3D Matching reducing to SAT, or SubsetSum reducing to SAT modulo Theory of Difference Arithmetic (caveat: this will generally not lead to a polynomial time algorithm, but SAT solvers can lead to fast heuristic algorithms)

3. If coming up with a reduction to a related problem did not succeed, you would have to come up with a new algorithm for the problem. Try to apply **algorithmic techniques**:

   - BFS
   - Greedy
   - Exhaustive Search
   - Divide and Combine (MergeSort, Randomized Selection)
   - Data Structures (Sorted Arrays, BSTs, Hash Tables)
   - Randomized algorithms
   - Many more in CS1240, CS1280, CS1820, AM121, CS1650, CS22xx: dynamic programming, DFS, priority queues, approximation algorithms, local search, linear/integer/convex programming

     ⋮

4. Try to show hardness/unsolvability by **reduction from** other problems

   - NP-hardness for combinatorial search problems
   - Unsolvability for program verification and logic problems
   - More in CS1210, CS1270, CS2210, Math141b, Phil144

5. And/or settle for weaker guarantees

   - Find more structure to incorporate in your modelling, or
   - Use heuristics that only run efficiently or are only correct on some inputs (such as SAT solvers)
   - Settle for approximation algorithms
   - Monte Carlo algorithms instead of Las Vegas algorithms
   - More in CS1240, CS1280, CS1520, CS1820, AM121

6. Importantly, keep the landscape of complexity classes in view:

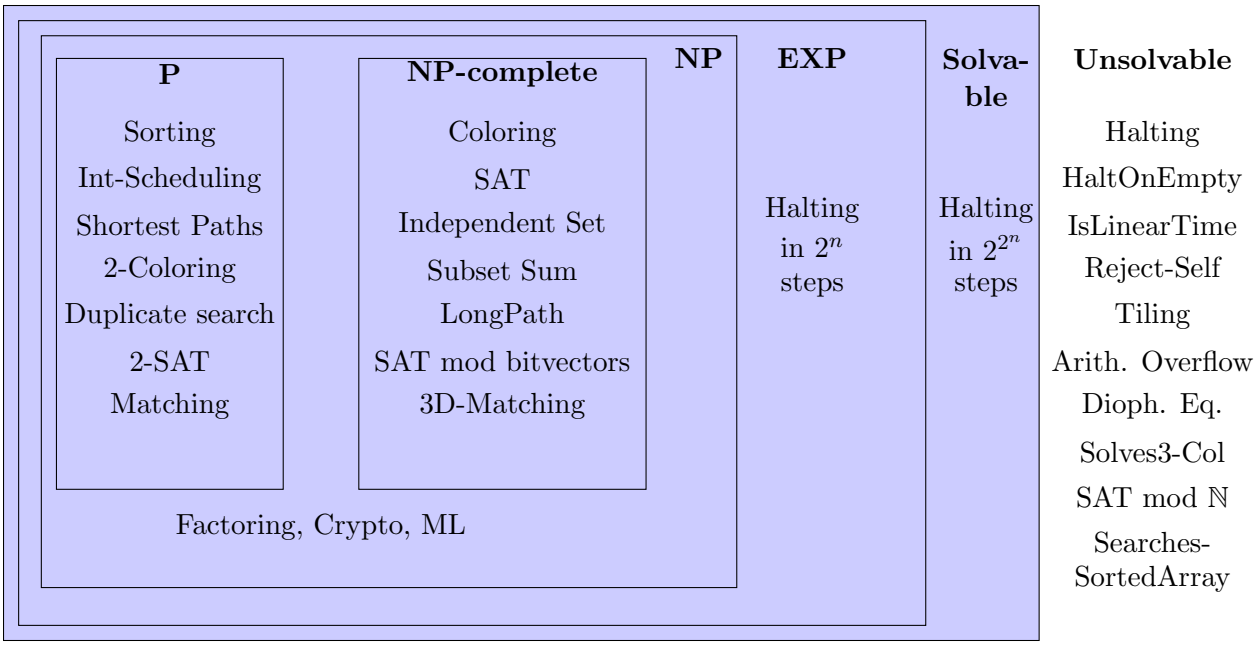

Figure 1: The complexity classes and computational problems in these classes. This diagram shows a separation between P and NP-complete problems, but there is also the possibility that P = NP = NP-complete, in which case the entire NP circle collapses into one class.

3 Other Takeaways

Universality. Various forms of 'universality' came up throughout the course. We saw the *Church–Turing Thesis* whereby our mathematically precise models of computation (e.g. RAM, Word-RAM, and Turing Machines) fully capture our intuitive notion of what is computable, and the *Extended Church–Turing Thesis* whereby some of these models (e.g. Word-RAM and TMs) also capture the physical time needed to solve problems, up to a polynomial slackness (in a sequential and deterministic computation). These theses were backed by *simulation* theorems, whereby we could compile programs in one model into programs in another model. Thus, each of (strongly) Turing-equivalent is universal in the sense that it can simulate any other reasonable model of computation. A stronger form of simulation comes from the *universal* programs (e.g. Universal (Word-RAM), Universal TM), whereby we can have one fixed program U that simulates any other program P given as an input. These forms of universality are part of what has made the theory of computation stand the test of time, even as physical computing technology has changed so much. Even modern computers can be simulated by Turing's 1936 model of computation. NP-completeness captures another form of universality, but for problems (like SAT) rather than models or programs. More on theoretical, applied, and/or philosophical aspects of these topics is covered in courses like CS1210, CS61, CS1520, CS1530, Math 141b, and Phil 144.

A Rigorous Mathematical Theory. Another major takeaway from CS1200 is that computation and the problems we wish to solve with it *can* be modelled precisely using mathematics, and in

fact this modelling predated and inspired the development of computing technology. This allows us to rigorously analyze the correctness and efficiency of the algorithms and data structures we design (and more broadly solutions to problems that arise throughout computer science). Furthermore, having a precise model of what is and is not an algorithm allows us to understand the limits of algorithms, proving that some problems are inherently unsolvable or intractable: no matter how hard we try, we will not discover an algorithm that bypasses these barriers.

Open Problems. Despite how rich and well-developed the theory of algorithms is, there is still much that we do not know. The P vs. NP problem stands out as the most famous of the open problems, addressing whether we can always do much better than brute-force exhaustive search, but many more fine-grained questions are also important and open, such as whether Sorting can be done in time $O(n)$ time and Matching can be done in time $O(n+m)$. When we bring in other resources like randomization or quantum computing, does this change what can be solved in polynomial time? (The conjectured answers are no and yes, respectively.) Although proving separations between complexity classes (like $\mathsf{P} \neq \mathsf{NP}$) seems to be a negative endeavor (showing that some problems are hard for efficient algorithms), one particularly important positive application of hardness is cryptography. Proving that the kinds of problems needed for cryptography are actually hard in the strong, average-case sense that cryptography needs (much more than $\mathsf{P} \neq \mathsf{NP}$) is another major open problem. For more on approaches to questions like these, you can take a course on complexity theory (CS 1210, CS 2210), fine-grained complexity (MIT 6.1420), quantum computing (CS 2310, QSE 210a), and/or cryptography (CS 1270).

4 CS1200 Learning Outcomes

It is a good time to reflect on the extent to which you have acquired the skills we set out to develop (as enumerated in the Syllabus):

- To mathematically abstract computational problems and models of computation
- To design and implement algorithms using a toolkit of algorithmic techniques
- To recognize and formalize inherent limitations of computation
- To rigorously analyze algorithms and their limitations via mathematical proof
- To appreciate the technology-independent mathematical theory of computation as an intellectual endeavor as well as its relationship with the practice of computing.
- To engage effectively in a collaborative theoretical computer science learning community, supporting your peers' learning as well as your own
- To clearly communicate mathematical proofs about computation to peers, conveying both high-level intuition and formal details.

5 Algorithms in a World of AI

Massive advances in AI and Machine Learning are transforming the practice of computing and many areas of science. How will it affect theoretical computer science and the theory of algorithms

as we have seen it in CS1200? With how quickly things are changing, it is impossible to make confident predictions, but here we share some thoughts.

Colloquially, people use the term 'algorithms' very broadly to incorporate the methods used in AI/ML systems as well as the theory of algorithms as we have studied it in CS1200. However, AI/ML systems differ substantially from what we have studied in CS1200 in several respects:

- They are designed to learn *models* (e.g. deep neural networks) to solve a problem (e.g. recognizing cat images, or responding to questions) from being given large amounts of *training data*. Thus, there are really two separate algorithmic components: the *training algorithm* which takes the training data and produces the model, and the *model* itself, which can be run on new inputs to solve the problem that it's been trained to solve.

- In the practice of AI, there is typically not a precisely formulated computational problem that is being solved. Instead, the goal is to produce a model that 'behaves like' whatever (real-world) process generated the training data.

There is some hope that ML will eventually replace some of the manual design of algorithms as we have studied in CS1200. For example, by training an ML model on many examples of graphs with maximum matchings, perhaps it can learn an algorithm for correctly solving the Maximum Matching problem. Nevertheless, it seems that there is likely to remain a major role for the theory of algorithms as we have covered in CS1200, at least for the foreseeable future, and possibly in perpetuity. Several potential reasons for this are:

- The massive data storage, computation, and networking systems on which ML models are trained and executed are based on the kind of fundamental algorithms and data structures that we studied at the beginning of the course (Storing & Searching Data), and optimizing these systems relies on formulating and analyzing computational models using skills like we developed in the second unit (Computational Models).

- At least in the near term, AI systems are still very far from having the kind of creativity and logical reasoning needed to extrapolate algorithms for complicated combinatorial problems like Maximum Matching merely given examples of solved instances.

- Furthermore, to train the algorithm we would need to produce optimally solved instances (which we currently do not know how to do without writing a Maximum Matching algorithm ourselves). And if we only provide solved instances, and no indication of how those instances were solved, the ML algorithm does not seem to have enough information to learn a *fast* algorithm for the problem, or even distinguish a fast algorithm from a slow one.

- Even if an ML system learns to solve a problem well on instances that are 'similar to' its training data, it is likely to err on new 'outlier' instances that are very different from its training data. That is, it won't provide worst-case correctness like we have demanded in CS1200, and at present, even generalization to larger input sizes typically fails. For high-stakes applications (e.g. in a medical application or a safety-critical system), such risks may not be acceptable.

- To address the above issues, a dream in the future is that we could train a large language model to perform *program synthesis*: take a *specification* of a computational problem (either

in mathematical logic as we saw in Lecture 24 or in natural language prose) and produce the *code* of an algorithm solving the problem in the worst case (ideally, one that is as efficient as possible, perhaps with runtime objectives specified also in the specification). We can't hope for such a language model to *always* succeed (for reasons of unsolvability) but it's conceivable that one day it will be as good as or even better than humans at this sort of algorithm design. Note that even if that happens, making use of such a language model relies on the skills from CS1200, in precisely formulating what computational problem we want solved and knowing what is realistic to expect in terms of efficiency.

- Even if the dream above is achieved, how can we be sure that the code generated by an ML model is correct? One way is for the ML model to also generate a *proof* that the algorithm is correct, which can then be verified using formal proof systems (such as Coq and Lean). The latter systems are in turn based on algorithms for mathematical logic, like we have studied in the context of SAT and SMT Solvers.

- Combining ideas and techniques from machine learning and traditional algorithms can yield solutions to problems that are better than what either paradigm could achieve on its own. For example, *data-driven algorithm design* and *learning-augmented algorithms* use predictions from an ML model to tune an algorithm's parameters and improve its performance.. And *reductions* between computational problems (as we have studied) can be used to apply an ML model designed for one task to solve another.

- The *limits* of algorithms we studied (i.e. NP-completeness and Unsolvability) apply to ML systems, and tell us that there are problems that ML systems will never be able to solve, no matter how much they improve. Furthermore, we rely on our understanding of the limits of algorithms to design cryptographic algorithms that protect the security of our communications and computations. In the context of ML, there are additional security and privacy concerns that are best addressed using worst-case analysis as developed in the theory of algorithms (since an attacker will try to make the 'worst case' happen).

- Many of the skills and ways of thinking developed in cs1200—mathematical abstraction of computational problems and computational models; rigorous analysis of correctness and efficiency; understanding, communicating, and collaborating on technical concepts; and reductions between problems and models—are useful throughout computer science (not just the study of algorithms), including in ML.

- The theory of algorithms and their limitations has intrinsic mathematical beauty and philosophical value, independent of its implementation in physical computing technology.

You can learn more about the algorithmic aspects of AI and ML in many of the CSx28x and CSx8xx classes.

6 Where to Learn More

- Theory of Computation seminar: https://toc.seas.harvard.edu/events-seminars

- Many other CS courses, especially x2x0. Look at grad (2xx0) courses as well. (CS1200 may serve as a sufficient substitute for CS1210/CS1240 in some of them.)

- Read more of our textbooks (Roughgarden, MacCormick, CLRS, and the references therein)
- Come talk to us in office hours!