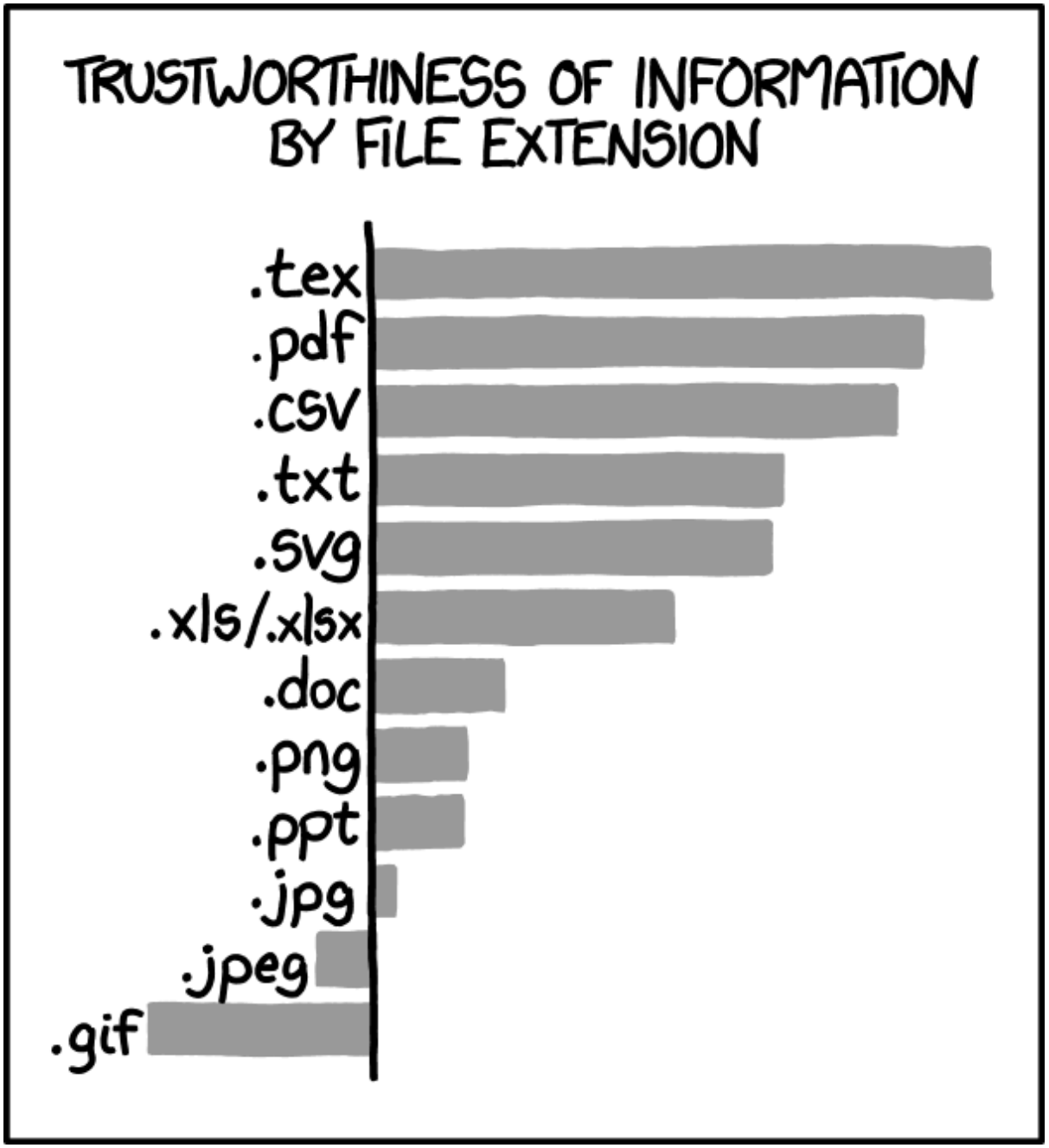
CSCI6454: Advanced Algorithms

ADITHYA BHASKARA
PROFESSOR: HUCK BENNETT

READINGS: VARIOUS

UNIVERSITY OF COLORADO BOULDER

EDITION 1



Contents

Preface

To the interested reader,

This document is a compilation of lecture notes scribed during the Spring 2025 semester for CSCI6454: Advanced Algorithms at the University of Colorado Boulder. The content in these course notes is from lecture and several resources cited herein. The author only takes responsibility for the scribing of the content and lectures into the notes. This course was taught by Huck Bennett, Ph. D.

While much effort has been put in to remove typos and mathematical errors, it is very likely that some errors, both small and large, are present. If an error needs to be resolved, please contact Adithya Bhaskara at adithya@colorado.edu.

Best Regards,
Adithya Bhaskara

REVISED: January 16, 2025

1 Introduction to Fine-Grained Complexity

1.1 Lecture 1: January 14, 2025

1.1.1 Introduction to Fine-Grained Complexity

We hope to precisely understand the complexity of several algorithmic problems beyond the coarse information given by standard complexity classes, like P and NP. A central theme is proving lower bounds for the complexity of algorithms; this is a difficult task in general. We use reductions as an essential tool.

The high-level idea behind fine-grained complexity is to explain hardness, or the lack of algorithmic progress, on fundamental computational problems by giving reductions from well-studied problems. Recall the P v. NP problem. Loosely, problems in P are "easy," to solve, and problems in NP are "easy" to verify positive answers to. The hardest problems in NP, the NP-complete problems do not admit polynomial-time algorithms unless $\mathsf{P} = \mathsf{NP}$. To show that a problem is NP-complete, we show that it is in NP, and exhibit a polynomial-time reduction from a known NP-complete problem.

The theory behind NP-completeness is robust and quite nice. But, we'd like finer results. Knowing whether a problem is in P or not doesn't really narrow down how efficiently we can solve it in practice. Even quadratic-time algorithms may be inefficient in production.

These ideas motivate fine-grained complexity. Here, we start by taking a well-studied problem, say L_1, and making a precise conjecture about the running time of optimal algorithms for L_1. Then, we give an efficient reduction from L_1 to a problem L_2, for which we are showing hardness. We start by providing some hypotheses corresponding to certain problems, and later, we'll show fairly precise running time bounds for problems in P.

Example 1. *Under certain assumptions, there is no $O(n^{2-\epsilon})$-time algorithm for the* Edit Distance *problem. This matches the $O(n^2)$ dynamic programming algorithm.*

Computational Problem 1 (k-SAT)**.**

- *Given a CNF formula φ with k literals in each clause,*
- *Decide: Does there exist a satisfying assignment?*

Hypothesis 1.1.1: Exponential Time Hypothesis (ETH)

The 3-SAT problem takes $2^{\Omega(n)}$ time.

Remark. *Note that* P $\neq$ NP *asserts that* 3-SAT *has no polynomial-time algorithm, whereas ETH asserts that* 3-SAT *has no subexponential-time algorithm. ETH implies* P $\neq$ NP*.*

Hypothesis 1.1.2: Strong Exponential Time Hypothesis (SETH)

For every $\epsilon > 0$, there exists $k \in \mathbb{Z}^+$ such that there is no $O\left(2^{(1-\epsilon)n}\right) = O((2-\epsilon)^n)$-time algorithm for k-SAT.

Remark. *SETH, at a high level, claims that 2^n-time is essentially optimal for k-SAT for large k.*

Computational Problem 2 (k-SUM)**.**

- *Given arrays $A_1, \ldots, A_k$ each with n integers in $[-n^c, n^c]$,*
- *Decide: Does there exist $a_1 \in A_1, \ldots, a_k \in A_k$ such that $a_1 + \cdots + a_k = 0$?*

Some variants include assuming that $A_1 = \cdots = A_k$, or wanting to find $a_1 \in A_1, \ldots, a_k \in A_k$ where $a_1 + \cdots + a_{k-1} = a_k$.

Naively, we can solve k-SUM by trying all possible choices a_i and checking the sum. This can be done in $O(n^k)$ time. We can do better. For now, let $k = 3$. We start by computing all possible sums $a_1 + a_2$. Then, define $S = \{a_1 + a_2 : a_1 \in A_1, a_2 \in A_2\}$. Sort A_3. For each $s \in S$, search in A_3 for $-s$. If $-s$ is found, there exists a solution. Otherwise, there is no solution. This provides an $O(n^2 \log n)$ algorithm; computing the sums takes $O(n^2)$ time, sorting takes $O(n \log n)$ time, and the search procedure takes $O(n^2 \log n)$ time.

Hypothesis 1.1.3: No $O(n^{2-\epsilon})$-Time Algorithm for 3-Sum

For every $\epsilon > 0$, there is no $O\left(n^{2-\epsilon}\right)$-time algorithm for 3-SUM.

1.2 Lecture 2: January 16, 2025

1.2.1 Fine-Grained Complexity II

Now, we introduce a new problem.

Computational Problem 3 (All Pairs Shortest Path (APSP))**.**

- *Given a weighted graph $G = (V, E, w)$ on n vertices,*
- *Find: The lengths of the shortest paths $d(v_i, v_j)$ between every $v_i, v_j \in V$.*

One way to solve APSP is to run Dijkstra's algorithm on each vertex. Recall that Dijkstra's algorithm finds a single-source shortest path tree. We could also similarly use the Floyd-Warshall dynamic programming algorithm.

Hypothesis 1.2.1: No $O(n^{3-\epsilon})$-Time Algorithm for APSP

For every $\epsilon > 0$, there exists no $O(n^{3-\epsilon})$-time algorithm for APSP.

We now introduce the Floyd-Warshall dynamic programming algorithm in detail. Let $D_k[i, j]$ store the length of the shortest path from v_i to v_j, with the restriction that the path only passes through $v_1, \ldots, v_k$.

Example 2. *To illustrate our notion, consider the below figure.*

[AB: Insert Huck's example, see lecture notes.]

The idea is to build D_k from D_{k-1} comes from noting that either

1. the shortest path from v_i to v_j only using $v_1, \ldots, v_k$ uses v_k, or
2. it does not.

[AB: Insert figure here; see lecture notes.] From this basic idea, we get a simple algorithm.

Algorithm 1.2.1 Floyd-Warshall

Require: $G = (V, E, w)$.

procedure Floyd_Warshall(E)

$$D_0 \leftarrow \begin{cases} 0 & i = j \\ w_{i,j} & i \neq j, (v_i, v_j) \in E \\ \infty & (v_i, v_j) \notin E \end{cases}$$

for $k \in \{1, \ldots, n\}$ **do**
  for $i \in \{1, \ldots, n\}$ **do**
    for $j \in \{1, \ldots, n\}$ **do**
      $D_k[i, j] = \min\{D_{k-1}[i, k] + D_{k-1}[k, j], D_{k-1}[i, j]\}$
return D_n ▷ $D_n[i, j] = d(v_i, v_j)$

Consider the following problem.

Computational Problem 4 (Orthogonal Vectors (OV))**.**

- *Given two sets* $V = \{v_1, \dots, v_n\} \subseteq [0,1]^d$ *and* $W = \{w_1, \dots, w_n\} \subseteq [0,1]^d$,
- *Decide: Does there exist* $v_i \in W$ *and* $w_j \in W$ *such that* $\langle v_i, w_j \rangle = \sum_{k=1}^{d} v_i[k] w_j[k] = 0$*?*

The naive algorithm is to compare every pair, compute the inner products, and compare to 0. We have n^2 inner products to compute, and each takes d time, so the naive algorithm takes $O(dn^2)$ time. Think of $d = \log^k(n)$ for some k. In this case, $O(dn^2) = O(n^2 \log^k(n)) = \tilde{O}(n^2)$.

Hypothesis 1.2.2: No $O(n^{2-\epsilon})$-Time Algorithm for OV

For every $\epsilon > 0$, there exists no $O(n^{2-\epsilon})$-time algorithm for OV with $d = \text{polylog}(n)$.

So far, we have 5 hypotheses; what are the relations herein? We'll attack this question after a brief interlude on reductions.

We write $L_1 \leq L_2$ if problem L_1 reduces to problem L_2; that is, L_1 is no harder than L_2; and if L_1 is hard, then L_2 is hard. Equivalently, if L_2 is easy, then L_1 is easy. Consider the following definition, making precise the notion of a "fine-grained reduction."

Definition 1.2.1: (Virginia Vassilevska Williams) Fine-Grained Reductions

Suppose L_1 and L_2 are computational problems. Let $\ell_1 = \ell_1(n)$ and $\ell_2 = \ell_2(n)$ be associated runtime bounds, say conjectured optimal bounds. Then, L_1 (ℓ_1, ℓ_2)-reduces to L_2 if for every $\epsilon > 0$, there exists $\delta > 0$ and an algorithm R running in time $O(\ell_1(n)^{1-\delta})$ and making q calls to an oracle for L_2 on inputs of size $n_1, \dots, n_q$ where

$$\sum_{i=1}^{q} \ell_2(n_i)^{1-\epsilon} \leq O(\ell_1(n)^{1-\delta}).$$

Remark. *To understand Definition 1.2.1, consider the below figure. [AB: Add Figure; see Lecture Notes]*

Theorem 1.2.1: (Ryan Williams, 2004): SETH $\implies$ OV

Hypothesis 1.1.2 implies Hypothesis 1.2.2.

Proof. Using Definition 1.2.1, we can restate this theorem as follows: for every $k \in \mathbb{Z}^+$, k-SAT $(2^n, n^2)$-reduces to OV with $d = \text{polylog}(n)$. We exhibit such a reduction, which will be of exponential-time.

Let $\varphi(x_1, \dots, x_n) = \bigwedge_{j=1}^{m} C_j$ be our k-SAT instance, with each C_j a k-clause. Partition $X = \{x_i\}_{i=1}^{n}$ into $X_1 = \{x_1, \dots, x_{\frac{n}{2}}\}$ and $X_1 = \{x_{\frac{n}{2}+1}, \dots, x_n\}$. Consider the $N = 2^{\frac{n}{2}}$ assignments to each of X_1 and X_2 independently. Then, let $V_1, V_2 \subseteq \{0,1\}^m$. For an assignment, a_1 to literals in X_1, $v_{a_1} \in V_1$ is defined by

$$v_{a_1}[j] = \begin{cases} 0 & a_1 \text{ satisfies } C_j \\ 1 & \text{otherwise} \end{cases}.$$

Define $v_{a_2} \in V_2$ similarly corresponding to a_2 to literals in X_2. We claim that $\langle v_{a_i}, v_{a_2} \rangle = 0$ if and only if (a_1, a_2) is a satisfying assignment. Note that $m = O(n^k) = O((2\log_2 N)^k) = O(\log^k N)$, where $N = 2^{\frac{n}{2}}$. A $O\left(N^{2-\epsilon}\right)$-time algorithm for OV would then imply an $O\left(N^{2-\epsilon}\right) = O\left(\left(2^{\frac{n}{2}}\right)^{2-\epsilon}\right) = O\left(2^{\left(1-\frac{\epsilon}{2}\right)n}\right)$-time algorithm for k-SAT. □

Example 3. *Consider $n = 6$, $m = 3$, and $k = 3$ with*

$$\varphi(x_1, \dots, x_6) = ([AB : Finish!])$$

Appendices

Bibliography