COMS E6998-15: Fine-grained complexity (Spring '26) February 3, 2026

Lecture 3: OV-Hardness of Sequence Problems

Instructor: *Josh Alman* Scribes: *Jack Parkhouse*

1 Introduction

We begin with a review of Schöning's randomized algorithm for k-SAT, and then conclude our discussion of SAT algorithms by analyzing said algorithm's runtime and proving certain bounds on its success probability. We then introduce the next topic - subsequence problems - and provide proof sketches for certain results on the hardness of solving these problems.

2 Schöning's SAT algorithm

2.1 Algorithm description

We would like an algorithm that solves k-SAT with runtime $2^{n(1-\frac{1}{\mathcal{O}(k)})}$ - i.e., an algorithm that is demonstrably superior to the 2^k-runtime brute-force algorithm, but which asymptotically approaches the brute-force runtime as k grows large. Schöning's algorithm is one such algorithm. The procedure is as follows:

1. Begin by picking a random initial assignment $\alpha : \{x_1, \ldots, x_n\} \to \{\mathrm{T}, \mathrm{F}\}$ to our formula over n variables (where each clause has at most k literals).
2. Repeat the following steps n times:
   (a) Find a clause c not satisfied by α. (If there is no such c, then halt and return SAT.)
   (b) Pick a random variable x_i in c and flip its assignment $\alpha(x_i)$.
3. If, after repeating steps (a) and (b) n times, we still have not returned SAT, return UNSAT.

We make a few observations about this algorithm:

- **This algorithm runs in** $\text{poly}(n)$**.** This is straightforward to see - we run at most n iterations.
- **If the input is unsatisfiable, then we always return UNSAT.** This is also straightforward - if there is no satisfying assignment to the formula, we will never return SAT; we only return SAT if we encounter a satisfying assignment.
- **If we repeat this algorithm many times, then the probability we incorrectly return UNSAT is very small.** To make precise what we mean by "very small," we will describe some success probability p - that is, the probability that we return SAT given that the formula is indeed satisfiable. We will show that this p is not too small, and thus repeating it some $\frac{\mathcal{O}(1)}{p}$ times will give us a very high success probability.

2.2 Success probability of Schöning's algorithm

Suppose there exists some satisfying assignment to our formula $\beta : \{x_1, \ldots, x_n\} \to \{T, F\}$. First, we will provide a lower bound on p in terms of the following two events: (1) the original random assignment that we have chosen, α, differs from β on γ variables; and (2) each of the first γ steps in the algorithm flips the assignment of a variable x_i on which α and β disagree. The concurrence of these two events describes one possible avenue by which we arrive at β at the termination of the algorithm - over the course of at most $\gamma \leq n$ steps, we correct at all γ discrepancies between α and β.

Assume for the moment that α and β differ on γ variables. Each iteration has a $\frac{1}{k}$ chance of flipping an x_i variable on which α and β disagree, since for any clause α doesn't satisfy, at least one of the k variables of that clause must be satisfied by β. Therefore, the probability that each of the first γ steps in the algorithm flips the assignment of a variable x_i on which α and β disagree is at least

$$(\frac{1}{k})^\gamma$$

Furthermore, given γ, there are $\frac{\binom{n}{\gamma}}{2^n}$ choices of α that have distance γ from β. To find a lower bound on p, we take the union bound over all possible values of γ.

$$\begin{aligned}
p &\geq \sum_{\gamma=0}^{n} \left(\frac{1}{k}\right)^\gamma \frac{\binom{n}{\gamma}}{2^n} \\
&= \frac{1}{2^n} \sum_{\gamma=0}^{n} \left(\frac{1}{k}\right)^\gamma \binom{n}{\gamma} \\
&= \frac{1}{2^n} \sum_{\gamma=0}^{n} \binom{n}{\gamma} \left(\frac{1}{k}\right)^\gamma (1)^{n-\gamma} \\
&= \frac{1}{2^n} \left(1 + \frac{1}{k}\right)^n && \text{(By the Binomial Theorem)} \\
&= \frac{1}{2^n} \cdot 2^{\frac{n}{O(k)}} && (\ e = \lim_{n\to\infty} \left(1 + \tfrac{1}{n}\right)^n) \\
&= \frac{1}{2^{n(1 - \frac{1}{O(k)})}}
\end{aligned}$$

Some concluding remarks: So why does this matter? This algorithm is noteworthy because we have demonstrated that there are indeed some nontrivial savings in runtime from attempting some correction on an initial assignment, even if that correction is random in nature, over a purely random assignment without correction. Indeed, this discussion motivates **Super SETH**, which states that there is no algorithm for SAT whose runtime savings can improve over $\frac{1}{\mathcal{O}(k)}$ in the exponent, ruling out even a seemingly modest improvement of $\frac{1}{\mathcal{O}(\log k)}$.

3 Subsequence problems

3.1 The Longest Common Subsequence problem

We now turn our attention to the world of subsequence problems. Subsequence problems include problems, such as Longest Common Subsequence (LCS), find numerous applications in situations as diverse as file version control (e.g., `diff`), computational linguistics, and bioinformatics.

Nearly all subsequence problems leverage dynamic programming in some form; it seems natural that the recursive structure of dynamic programming would provide a useful framework when it comes to subdivision of the original problem. Briefly, we define the dynamic programming approach as follows: We wish to find

$$M[i,j] = \mathsf{LCS}(\text{first } i \text{ characters of } S, \text{first } j \text{ characters of } T)$$

where

$$M[i,j] = \max\left\{M[i-1,j], M[i,j-1], M[i-1,j-1] + (S[i] \texttt{ == } T[j])\right\}$$

where the third term checks for equality between the ith character in S and the jth character in T. We claim without proof that the recursive dynamic programming procedure above computes LCS in $\mathcal{O}(n^2)$ time; no improvements in this runtime beyond a logarithmic factor are known yet.

As a concrete example of computing LCS, consider two sequences of strings:

$$S = ABAABACDE$$
$$T = BABBABEDC$$

The longest common subsequence is given by the longest in-order (i.e., left-to-right) sequence of characters, not necessarily adjacent, that can be matched between S and T. For our above example, the longest common subsequence is highlighted below:

$$S = A\color{red}{B}\color{purple}{A}\color{cyan}{A}\color{green}{B}\color{black}{A}\color{blue}{C}\color{black}{DE}$$
$$T = \color{red}{B}\color{purple}{A}\color{black}{BB}\color{cyan}{A}\color{green}{B}\color{black}{ED}\color{blue}{C}$$

where the matching colors corresponds to the characters from S and T that are matched; thus, the longest common subsequence between S and T has length 5 and is BAABC.

3.2 OVC and LCS

We will first discuss a foundational proof of the subsequence problem universe, an OV-hardness result, proven independently and simultaneously by Abboud, Backurs, and Vassilevska Williams [ABW15] and Bringmann and Künnemann [BK15].

Theorem 1. *Assuming OVC, there is no $\mathcal{O}(n^{2-\varepsilon})$-time algorithm for* LCS *for any $\varepsilon > 0$.*

Proof. The following is a proof sketch. We have as input two sets $A, B \subseteq \{0,1\}^d; |A| = |B| = n$. We would like to know if there exists $a \in A, b \in B$ such that $\langle a, b\rangle = 0$; in other words, we would like to solve the OV decision problem on A and B. Our procedure will require two overarching steps: 1) convert our vectors into n strings of length poly(d), i.e., devise a mapping from $\{0,1\}^d \to \Sigma^{\text{poly}(d)}$, where Σ is our alphabet of letters; and 2) convert our n strings from A into a single string, and similarly for B, and then computing LCS between A and B, where the result would correspond to the existence of a pair of orthogonal vectors between A and B.

We begin by tackling part (1) of our two-part procedure. Our proof will make use of several intermediate results which we describe below.

Lemma 2. *There are four strings s_0, s_1, t_0, t_1 such that*

$$\mathsf{LCS}(s_0, t_0) = \mathsf{LCS}(s_0, t_1) = \mathsf{LCS}(s_1, t_0) > \mathsf{LCS}(s_1, t_1)$$

Proof. Choose $s_0 = AB; s_1 = AA; t_0 = BA; t_1 = BB$. □

Lemma 3. *There are mappings $VG_A, VG_B : \{0,1\}^d \to \Sigma^{poly(d)}$ such that for all $a, b \in \{0,1\}^d$, if $\langle a, b\rangle = 0$, then $\mathsf{LCS}(VG_A(a), VG_B(b)) = \beta$; and if $\langle a, b\rangle \neq 0$, then $\mathsf{LCS}(VG_A(a), VG_B(b)) = \beta - 1$.*

Proof. We construct the "vector gadgets" (VGs) which satisfy the desired properties above. Consider the following construction:

$$VG_A(a) = s_{a[1]} X s_{a[2]} X \dots s_{a[d]}$$
$$VG_B(b) = t_{b[1]} X t_{b[2]} X \dots t_{b[d]}$$

where $s_{a[i]}$ and $t_{b[j]}$ are defined as in Lemma 2, and which are chosen based on the value of $a[i]$ or $b[j]$ (which are either 0 or 1), and where $X = C^{3d}$. We introduce a "blowup" substring X in order to enforce that each "coordinate gadget" $s_{a[i]}$ or $t_{b[j]}$ would strictly match to the corresponding substring in the opposing string in the longest common subsequence, but which increases the total length of each string by only a polynomial-in-d factor.

Now, we claim that if we find a match in the manner described above, then the longest common subsequence has length $(d-1)3d + d - \langle a, b\rangle$. The full proof of this claim involves some tedious case analysis, which we omit here. The crux of the argument relies on the X substrings - because we have chosen the length of each X substring to be $3d$, they are sufficiently long such that, in building the longest common subsequence, there are no additional gains from matching merely a portion of the coordinate gadgets $s_{a[i]}$ or $t_{b[j]}$ to each other over matching an X substring; in other words, there is no way to match up bits and pieces of the coordinate gadgets to find the longest common substring given the X portions.

As an aside, we note that we can slightly modify our coordinate gadgets so that we achieve equality between the LCS of all combinations of strings s_0, s_1, t_0, t_1 from Lemma 2. Consider the following modification:

$$VG_A(a) = YsXsX \dots sX$$
$$VG_B(b) = tXtX \dots tXY$$

where $Y = D^{\beta-1}$. Introducing these Y subsections allows us to enforce another condition: if the longest common substring is less than length $\beta - 1$, then we need not consider such a result - the sequence that gets returned is the sequence of $\beta - 1$ Ds.

□

Now, we are ready to begin part (2) of our two-part procedure. Recall that we wish to convert our n strings, which we constructed in part (1), into one string, for each set A and B; formally, we wish to convert these strings into single strings $x, y \in \Sigma^{n \cdot \text{poly}(d)}$. We wish to perform this conversion such that if there exists an orthogonal pair between A and B, then $\mathsf{LCS}(x, y) \geq \tau$; otherwise, $\mathsf{LCS}(x, y) < \tau$.

Consider the following construction:

$$\begin{aligned} x &= VG_A(a_1)ZVG_A(a_2)\ldots ZVG_A(a_n)ZVG_A(a_1)Z\ldots ZVG_A(a_n) \\ y &= Z^n VG_B(b_1)ZVG_B(b_2)\ldots ZVG_B(b_n)Z^n \end{aligned}$$

where $Z = E^{d^{100}}$ (*n.b.* d^3 is likely sufficient; for eudaemonic purposes, we choose an exorbitantly large number). We make note of a few things: 1) x is composed of two concatenated copies of itself, with a Z in between; i.e., it is of the form xZx; this is to ensure that any LCS matching can be made regardless of the order in which each vector gadget appears in x and y, which is important to mirror the orderless property of the orthogonal vectors problem; and 2) in order to keep the proportion of Z-substring relatively balanced between x and y, we pad the beginning and end of y with additional Z-substrings.

This argument requires extensive case analysis as well, which is omitted here; see [ABW15] for the full proof.

□

Note: There exists a proof of Theorem 1 using only a binary alphabet by Rubinstein and Song [RS20].

References

[ABW15] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Quadratic-time hardness of lcs and other sequence similarity measures. *arXiv preprint arXiv:1501.07053*, 2015.

[BK15] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97. IEEE, 2015.

[RS20] Aviad Rubinstein and Zhao Song. Reducing approximate longest common subsequence to approximate edit distance. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1591–1600. SIAM, 2020.