6.1420/6.S974 Fixed-Parameter and Fine-Grained Complexity **MIT**
Lecture 5: OV Hypothesis implies hardness for string problems **September 15, 2024**
Virginia Vassilevska Williams and Ryan Williams (notes by V)

Today: We defined the Orthogonal Vectors problem (OV) in previous lectures, showing that SETH implies that OV on n vectors of dimension $\omega(\log n)$ requires $n^{2-o1)}$ time. We also saw a simple reduction from OV to the Diameter problem in sparse graphs. Today we will develop reductions from OV to two sequence similarity problems: Longest Common Substring with Don't Cares (LC*) and Longest Common Subsequence (LCS). Such reductions have been designed for many other string problems: Frechet Distance, Edit Distance, Dynamic Time Warping and so on. The reductions have various similarities but are all different due to the different gadgets that are employed.

1 Some structure of OV and string similarity problems

Given an instance $S = \{0,1\}^d$ of OV with $|S| = n, S = \{s_1, \ldots, s_n\}$ the problem asks to compute:

$$\bigvee_{i,j\in[n]} \bigwedge_{c\in[d]} (\neg s_i[c] \vee \neg s_j[c]) \,.$$

Meanwhile, in string similarity problems, one is given two N length strings a and b and one considers all possible *alignments* of a and b and needs to compute the best such alignment according to some quality measure which is typically computed symbol by symbol in the alignment.

Let's differentiate between *substrings* and *subsequences* of a string $a = a_1a_2 \ldots a_n$ where each $a_i \in \Sigma$ is a letter.

A *substring* of a is $a_ia_{i+1} \ldots a_j$ for some $j \geq i$, i.e. it is a consecutive list of characters in a.

A *subsequence* of a is some $a_{i_1}a_{i_2} \ldots a_{i_k}$ for some choice of $i_1 < i_2 < \ldots < i_k$. That is, a substring is a subsequence where $i_j = i_1 + (j - 1)$. A more general subsequence of a is a string of not necessarily consecutive characters of a, as long as they appear in the same order as in a.

What is an *alignment* of two strings a and b? For problems that involve substrings, an alignment is a just a way to put the symbols of a on top of the symbols of b, shifted by some amount. E.g. if b is shifted $i - 1$ places to the right, the alignment may look like this:

$$\begin{array}{cccccccccc} a_1 & a_2 & \ldots & a_i & a_{i+1} & \ldots & & a_N & \\ & & \ldots & b_1 & & b_2 & \ldots & b_{N-i+1} & \ldots \end{array}$$

For most sequence similarity problems, a more general alignment allows gaps between symbols in addition to shifts, aligning symbols of one string with symbols or gaps in the other string (in the same order). E.g. if _ signifies a gap, the following is an alignment of $xmatch$ with $meanny$. This more general alignment can be viewed as a walk on sequences, or sometimes as a subsequence problem.

$$\begin{array}{cccccccc} x & m & \_ & a & t & c & h & \\ & m & e & a & n & \_ & n & y \end{array}$$

A typical sequence similarity problem defines a cost/gain of matching symbols/gaps in an alignment, and the goal is to find an alignment that minimizes/maximizes the total cost.

Suppose that we want to reduce OV to a maximization problem (minimization is analogous): A natural attempt would be as follows:

- Create two gadgets f and g that map $\{0, 1\}$ to symbols from an alphabet Σ so that matching $f(1)$ and $g(1)$ would give small/zero quality, whereas matching $f(x)$ and $g(y)$ for any $(x, y) \neq (1, 1)$ would give large quality.

  This step implements the inner $\vee$ in the definition of OV: $\neg s_i[c] \vee \neg s_j[c]$.

- Create gadgets F and G that map binary strings of length d to strings over Σ so that the max alignment of $F(s)$ and $G(t)$ has large quality if s and t are orthogonal and small quality otherwise. These gadgets typically look like this: for $F(s)$, string $f(s[1]), f(s[2]), \ldots, f(s[d])$ one after the other (i.e. using the symbol gadgets on each bit of s), sometimes adding some extra strings around each $f(s[i])$. Then, the quality of aligning $F(s)$ completely with $G(t)$ is proportional to the number of coordinates i for which $(s[i], t[i]) \neq (1, 1)$. Thus when s and t are orthogonal, the quality is maximized.

  This step implements the $\bigwedge$ in the definition of OV: $\bigwedge_{c\in[d]}(\ldots)$.

- Finally, figure out a way to glue the gadgets $F(s_1), F(s_2), \ldots, F(s_n)$ next to each other with various symbols inbetween, and similarly $G(t_1), G(t_2), \ldots, G(t_n)$. This creates the final strings a and b. The goal of this step is that the only good alignments are those that align some $F(s_i)$ exactly on top of some $G(t_j)$ and where the quality of the alignment is completely determined by the quality of the alignment of $F(s_i)$ and $G(t_j)$ which then means that the best alignment will allow us to determine if there exist s_i, t_j that are orthogonal.

  This step implements the outer $\bigvee$ in the definition of OV: $\bigvee_{i,j\in[n]}(\ldots)$.

Finally one wants the reduction to produce strings of length $N = n(d)^{o(1)}$ so that an $O(N^{2-\varepsilon})$ time algorithm for the string problem implies an $O(n^{2-\delta})$ time algorithm for OV (for small d).

2 Longest Common Substring with Don't Cares

In the Longest Common Substring with Don't Cares (LS*) problem, one is given two n-length strings a, b where a is over a finite alphabet Σ and b is over $\Sigma \cup \{*\}$. The question is: what is the longest string c that appears both in a and b as a substring (consecutive letters)?

In b, $*$ can represent any letter of Σ. So the question is, what is the longest substring of a that matches a substring of b? (Think about how this can be thought of as an alignment problem.)

For instance, the LS* of $abceaaad$ and $*rbc**ak$ is $bceaaa$ of length 6.

There is known algorithm for LS* that runs in $O(n^{2-\varepsilon})$ time for any $\varepsilon > 0$, although a quadratic time algorithm is very easy to obtain, e.g. via dynamic programming (try it!).

The related Longest Common Substring problem is similar, but b is also over Σ (there are no $*$). This problem can be solved in $O(n)$ time! Another simpler variant allows for b to have $*$s but instead of looking for a substring of a and b, it asks whether b itself matches a substring of a. This problem also has a fast algorithm: $O(n \log n)$ time.

We will show that OV reduces to LS* so that any truly subquadratic algorithm for LS* implies a truly subquadratic time algorithm for OV. In fact, our reduction can be modified to also work for $\Sigma = \{0, 1\}$.

We will follow the gadget approach outlined above. Define bit gadgets: for every $b \in \{0, 1\}$:

$$f(b) = b \text{ and } g(b) = 0 \text{ if } b = 1 \text{ and } * \text{ if } b = 0.$$

By design we get that $f(b)$ matches $g(b')$ as long as $(b, b') \neq (1, 1)$.

Now let's define vector gadgets that take any vector $s \in \{0,1\}^d$ to a length d sequence:

$$F[s] = f(s[1])f(s[2])\dots f(s[d]) = s \in \{0,1\}^d,$$

$$G[s] = g(s[1])g(s[2])\dots g[s[d]) \in \{1,*\}^d.$$

By design, $G[s_j]$ exactly matches $F[s_i]$ if and only if for every $c \in [d]$, $(s_i[c], s_j[c]) \neq (1,1)$ which is if and only if $s_i \cdot s_j = 0$. In other words, if $s_i \cdot s_j = 0$, the LS* of $G[s_j]$ and $F[s_i]$ is $= d$, and if $s_i \cdot s_j \neq 0$ then the LS* of $G[s_j]$ and $F[s_i]$ is $< d$.

Now we want to form the final strings a, b. Let X and Y be new letters in our alphabet. Y will not appear anywhere in a.

Let

$$a = F[s_1]XF[s_2]X\dots XF[s_n]$$

and

$$b = G[s_1]YG[s_2]Y\dots YG[s_n].$$

Because Y does not match any symbol in a, the LS* of a in b is the largest out of the LS*s of a and $G[s_j]$ over all j.

Consider

$$a = F[s_1]XF[s_2]X\dots XF[s_n]$$

and

$$G[s_j] \in \{1,*\}^d.$$

Some $*$ of $G[s_j]$ could potentially be matched with some X of a. E.g. you could align $\dots 010X01\dots$ from a with $010*01$ from $G[s_j]$ and this does not correspond to an alignment of $F[s_i]$ with $G[s_j]$!

Fortunately, this issue is easy to fix. We redefine the gadgets F and G as follows. Let Z and W be brand new symbols. For any any vector $s \in \{0,1\}^d$ form a length $3d$ sequence:

$$F[s] = Zf(s[1])WZf(s[2])W\dots Zf(s[d])W \in \{0,1,Z,W\}^{3d},$$

$$G[s] = Zg(s[1])WZg(s[2])W\dots Zg[s[d])W \in \{1,*,Z,W\}^{3d}.$$

We get that the LS* length of $F[s_i]$ and $G[s_j]$ is $3d$ if $s_i \cdot s_j = 0$ and $< 3d$ otherwise.

a and b remain the same:

$$a = F[s_1]XF[s_2]X\dots XF[s_n]$$

and

$$b = G[s_1]YG[s_2]Y\dots YG[s_n].$$

Their length is now $n-1+3dn$. As before, since Y does not appear in a, the LS* of a and b is the same as the longest over all $j \in [n]$ of the LS*s of a and $G[s_j]$.

As X does not appear in $G[s_j]$ for any j, either the LS* is the longest substring of $F[s_i]$ and $G[s_j]$ for some i and j, or some $*$ of some $G[s_j]$ is aligned with some X of a.

Suppose that the latter thing happens. If the LS* length is more than 1, then either the symbol to the left of $*$ is aligned with the symbol to the left of X or the symbol to the right of $*$ is aligned with the symbol to the right of X.

The symbol to the left of $*$ by design is Z and the one to the left of X is W and these are distinct. Similarly, the symbol to the right of $*$ by design is W and the one to the right of X is Z and these are distinct. Hence the maximum LS* length one can get by aligning $*$ from some $G[s_j]$ to some X in a is 1. This is not useful since one can always align some $Xf(s_i[c])Z$ in a with $X*Z$ from $G[s_\ell]$ where s_ℓ is not the all 1s vector to get a substring of length 3; wlog we can assume that S contains such an s_ℓ.

Hence, the LS* of a and $G[s_j]$ is the LS* of $F[s_i]$ and $G[s_j]$ for some i and the reduction is finished: The length of the LS* of a and b is $3d$ if there is a pair of orthogonal vectors and it's $< 3d$ otherwise.

Since a and b both have length $O(dn)$, any truly subquadratic algorithm computing their longest common substring would determine whether the LS* length is $3d$ or $< 3d$ and would solve OV on the given vectors $s_1, \ldots, s_n$.

The alphabet that we used was $\Sigma = \{0, 1, X, Y, Z, W\}$. Two get a reduction for $\Sigma = \{0, 1\}$, replace X by 000, Y by 111, Z and W by 01.

Exercise 2.1 *Show that substring of a and b that matches any of the 0s of some X to 0 or $*$s in b can have length at most 5. Similarly, show that any substring of a and b matching any of the 1s of some Y to 1s of a can have length at most 5.*

Once you solve the above exercise, we get that any substring of a and b of length > 5 cannot use any symbols of X or Y and thus must be a substring of $F[s_i]$ and $G[s_j]$ for some $i, j \in [n]$. We easily get that the longest common substring of $F[s_i]$ and $G[s_j]$ (which both have length $5d$) is of length $5d$ if $s_i \cdot s_j = 0$ and it's $< 5d$ otherwise. This completes the reduction for the binary alphabet case as well.

3 Longest Common Subsequence

In the Longest Common Subsequence (LCS) problem one is given two n length strings a and b over some finite alphabet Σ and one wants to find the longest string s that appears as a *subsequence* of both a and b. A subsequence doesn't need to have consecutive letters: the letters of s only need to appear in the same order in a and b but they need not be consecutive.

For instance the LCS of $abracadabra$ and $baaxxaac$ is $baaaa$.

The fastest known algorithm for LCS runs in $O(n^2/\log^2 n)$ time. On the slides we will give a reduction from OV to LCS with some intuition on the proof, without all the details. The reduction is similar in spirit to the one for LS* but it becomes more complicated because subsequences are more complicated than substrings.