1 The Polynomial Method in Algorithms

Last time, we saw how the OV can be reduced to a number of sequencing problems, by carefully considering the “structure” of the OV problem, as an OR of AND of ORs, and making “sequencing gadgets” that mimic the logic of OV. Today, we will look at algorithms for computing OV faster which exploit this structure. In particular, we will look at a method for computing OV by expressing its propositional logic form (OR of AND of ORs) as polynomials. This is a very generic method that works for many problems besides just OV. (To give two examples, the method can also be used to get faster algorithms for solving systems of polynomial equations [LPT+ 17], and a completely different algorithm for $k$-SAT, even for counting the number of SAT assignments to a $k$-CNF, in $2^{n - n/O(k)}$ time [CW21]. We’ll see another example in the next lecture.) Maybe polynomials will help you get a faster algorithm for your favorite problem!

As before, the exercises are optional, but recommended! They are there to get you thinking about the problems. Answers to the exercises for these lecture notes can be found at the end of this PDF.

Recall the OV problem.

**ORTHOGONAL VECTORS (OV)**

**Input:** $v_1, \ldots, v_n \in \{0, 1\}^d$, i.e., $n$ vectors in $d$ dimensions

**Decide:** Are there $i, j$ such that $\langle v_i, v_j \rangle = 0$?

I’ll sometimes denote the problem as $OV_{n,d}$ when I want to emphasize that the number of vectors is $n$ and their dimensionality is $d$. OV is a very basic and versatile problem. Here is an interesting application: partial match queries.

In the Partial Match problem, we are given a “database” of $n$ binary strings, and a list of $n$ “queries” which are strings in $\{0, 1, ?\}^*$. (Here, “?” represents a wildcard.) We say that a query $q = q_1 \cdots q_d$ matches a string $x = x_1 \cdots x_d$ if for all $i \in [d]$, if $q_i \in \{0, 1\}$ then $q_i = x_i$. We want to determine, for all $n$ queries, which queries match some string in the database. This is a very important problem in search that has confounded researchers for decades.

The following theorem shows that faster algorithms for OV are equivalent to faster algorithms for Partial Match:

**Theorem 1.1** $OV$ and Partial Match are subquadratic-time equivalent. Formally, for “reasonable” time functions $T$, there is an $\varepsilon > 0$ such that $OV$ is in $n^{2-\varepsilon} \cdot T(O(d))$ time if and only if there is a $\delta > 0$ such that Partial Match is in $n^{2-\delta} \cdot T(O(d))$ time.

Recall that OV is a decision problem, while Partial Match, as stated above, is a function problem: you have to output $n$ bits, one bit for each query. Nevertheless, these two problems are fine-grained equivalent!

You will prove a partial result towards this equivalence, on your problem set.

2 Simple Algorithms for OV

Let’s start by listing some known algorithms for finding an orthogonal pair.
1. Trivial brute-force algorithm for OV: $O(n^2d)$ time.

2. There is a “folklore” algorithm which is faster, when the dimension $d \ll \log(n)$: it runs in $O(dn \cdot 2^d)$ time.

**Exercise 2.1** Can you find such an algorithm? (The solution in the appendix is 2.5 lines of text.)

3. There is also an algorithm running in $\text{poly}(d) \cdot (n + 2^d)$ time.

   One of your problem set questions is to find such an algorithm. Here is one route to solving it. We can reduce OV to the following “subset query” problem (also on the problem set):

   Given subsets $S_1, \ldots, S_n \subseteq \{1, \ldots, d\}$, are there $i, j \in [n]$ such that $S_i$ is a proper subset of $S_j$?\(^1\)

   This subset query problem can then be solved by dynamic programming in the desired running time. (That’s all the hints we’ll give here! If you need more, ask on piazza or in office hours.)

4. For large dimensionality $d$, there is an algorithm running faster than $n^2d$:

   **Theorem 2.1** (Gum-Lipton’01) \([GL01]\) OV is in $O(n^2 \cdot d^{\omega-2})$ time, where $\omega < 2.373$ is the matrix multiplication exponent.

   The proof (which we now show) indicates an important connection between OV and Matrix Multiplication. Consider an $n \times d$ matrix $A$ with rows equal to the vectors $v_1, \ldots, v_n$. Observe that for all $i, j$, $(A \cdot A^T)[i, j]$ equals $\langle v_i, v_j \rangle$ (where $A^T$ is the transpose of $A$).

   We can therefore view the OV problem in the following way: we are given an $n \times d$ 0-1 matrix $A$, and wish to determine if $A \cdot A^T$ contains any 0-entry. We want to do this in time significantly less than the total number of entries in $A \cdot A^T$.

### 3 Towards a Not-So-Simple Algorithm

From the reduction from CNF-SAT to OV given in lecture 1, and the sparsification lemma for $k$-SAT as described in lecture 2) we have:

**Theorem 3.1** $\text{SETH} \implies$ for every $\varepsilon > 0$, OV is not in $n^{2-\varepsilon} \cdot 2^{o(d)}$ time.

Therefore, assuming SETH, there is no OV algorithm that runs in time that is both “subquadratic in $n$ and subexponential in $d$”.

In this lecture, we’ll describe the algorithm of [Abboud, Williams, Yu 2015] \([AWY15]\) which shows:

**Theorem 3.2** (Main Theorem) For every $1 \leq c(n) \ll 2^{\log n}$, $OV_{n,c \cdot \log(n)}$ can be solved in (randomized) time $n^{2-1/O(\log c)}$.

**Corollary 3.1** For every $c \geq 1$, there is an $\varepsilon > 0$ such that $OV_{n,c \cdot \log(n)}$ is in $n^{2-\varepsilon}$ time.

\(^1\)Recall $[n] := \{1, \ldots, n\}$. We will use this notation often in these notes!
If we could swap the “for all \( c \)” and “there is an \( \varepsilon \)” quantifiers, we would refute SETH and the Orthogonal Vectors Hypothesis! This follows from the fact that \( k \)-SAT can be sparsified: for example, we can reduce \( k \)-SAT instances with any number of clauses to \( 2^{n/1000} \) \( k \)-SAT instances with at most \( c_k n \) clauses, for a constant \( c_k \) that only depends on \( k \).

This algorithm is not-so-simple, but it will illustrate a quite powerful method. We will translate the OV problem into a problem about polynomial evaluation, then solve the polynomial evaluation more quickly than the obvious algorithm would do.

### 3.1 The Starting Point: A Self-Reduction

We begin with a simple “self-reduction” for OV. (Informally, a “self-reduction” for a problem \( \Pi \) is a reduction from \( \Pi \) on instances of length \( n \) to a number of smaller instances of \( \Pi \).)

**Theorem 3.3** For any parameter \( s \in [n] \), we can reduce any instance of OV\(_{n,d}\) to \( n^2/s^2 \) instances of OV\(_{2s,d}\).

**Proof.** Given an instance of OV with \( n \) vectors in \( d \) dimensions, the reduction works as follows:

1. Divide the \( n \) vectors into \( O(n/s) \) groups, where each group has at most \( s \) vectors each.
2. For all \( (n/s)^2 \) pairs of groups \( G, G' \), call OV\(_{2s,d}\) on the union of the two groups \( G \cup G' \).
3. If any call returns “yes”, then output “yes” (there’s an OV pair) else output “no”.

\( \Box \)

The obvious algorithm for OV\(_{2s,d}\) takes \( O(s^2d) \) time, so if we applied that algorithm to step (2) of the self-reduction, we’d only get a running time of \( O((n/s)^2 \cdot s^2d) \leq O(n^2d) \) time. No improvement at all. The key idea behind the **Main Theorem** is the following:

Represent the function OV\(_{2s,d}\) in some "interesting" way, so that we can evaluate OV\(_{2s,d}\) on many pairs of groups fast.

(In particular, we will represent OV\(_{2s,d}\) as a multivariate polynomial, and use fast matrix multiplication to evaluate the function quickly on many points!)

This will speed-up the execution of step (2) of the self-reduction.

Ideally, we want to compute step (2) in \( \tilde{O}(n^2/s^2) \) time for the **largest possible** choice of \( s \).\(^2\) Then, applying the self-reduction, we’ll get an \( \tilde{O}(n^2/s^2) \) time algorithm for OV\(_{n,d}\). (It turns out we can set \( s = n^{1/O(\log c)} \), which is exactly how we get \( n^{2-1/O(\log c)} \) time in the **Main Theorem**.)

### 3.2 Representing OV With Propositional Logic

OK, how do we represent OV\(_{2s,d}\)? (To keep the notation simple, we’ll just look at OV\(_{s,d}\) in the following.)

First of all, we can think of OV\(_{s,d}\) as a Boolean logic expression:

Given \( nd \) bits \( V = v_1[1], \ldots, v_1[d], \ldots, v_n[1], \ldots, v_n[d] \) (encoding \( n \) vectors of \( d \) bits each), we can write

\[
OV_{s,d}(V) = \bigvee_{i,j \in [s]} \bigwedge_{k \in [d]} (\neg v_i[k] \lor \neg v_j[k]).
\]

(Virginia observed exactly this fact in the previous lecture. The logical expression directly encodes the condition:

\(^2\)Our \( \tilde{O} \) notation will mean that \( \tilde{O}(f(n, s)) \leq f(n, s) \cdot c \log^c n \), for some constant \( c \geq 1 \).
There exists $i, j$ such that for all $k$, $v_i[k] \cdot v_j[k] = 0$.

Our goal is to evaluate this “OR of AND of ORs” on many pairs of inputs, quickly. To do this, we will use a randomized representation of $OV_{n,d}$ to get a nice polynomial representing it. The key idea is the marvelous XOR Trick that was originally used to prove lower bounds in circuit complexity [Raz87].

### 3.3 Converting OV to a Polynomial With the XOR Trick(s)

Given a clause $C = (y_1 \lor \cdots \lor y_L)$ and a parameter $k$ (think of $k \ll L$), the XOR trick randomly reduces $C$ to a “circuit” $G$, which is an OR of only $k$ XORs of random subsets of $y_1, \ldots, y_L$. That is, we take $k$ independent and uniform random subsets $R_1, \ldots, R_k$ of $\{y_1, \ldots, y_L\}$, compute the XOR of the bits in each $R_i$, then take the OR of the $k$ outcomes. In other words, our reduction looks like:

$$
(y_1 \lor \cdots \lor y_L) \rightarrow \bigvee_{i=1}^{k} \left[ \sum_{j=1}^{L} \alpha_{i,j} y_j \mod 2 \right],
$$

where all $\alpha_{i,j}$ are chosen from $\{0, 1\}$ independently and uniformly at random. We’re replacing a “big” OR on $L$ variables, with a “small” OR of XORs of up to $L$ variables.

**Lemma 3.1 (XOR Trick Lemma, Part I)** For every $k$ and $L$, there is a distribution of formulas $\mathcal{D}$, each of the form “$\lor_{i}[\sum_{j=1}^{L} y_j \mod 2]$”, such that for all $y \in \{0, 1\}^L$,

1. If $(y_1 \lor \cdots \lor y_L) = 0$, then for every $G$ drawn from $\mathcal{D}$, $G(y_1, \ldots, y_L) = 0$.
2. If $(y_1 \lor \cdots \lor y_L) = 1$, then $\Pr_{G \sim \mathcal{D}}[G(y_1, \ldots, y_L) = 0] = 1/2^k$.

In case (1), we always compute the right answer with $G$. In case (2), there is probability only $1/2^k$ of outputting the wrong answer.

**Proof.**

1. If all $y_1, \ldots, y_L$ are 0, then every XOR of a subset of $y_1, \ldots, y_L$ is also 0, and every OR of those XORs is also 0.

2. If at least one of $y_1, \ldots, y_L$ is 1, let $S \subseteq [L]$ be the (non-empty) subset of $i$’s such that $y_i = 1$.

We claim that, for every nonempty $S \subseteq [L]$, and randomly chosen $R \subseteq [L]$, $\Pr_R[|R \cap S| \text{ is odd}] = 1/2$.

**Exercise 3.1** Try to prove the claim!

The claim implies that, if $(y_1 \lor \cdots \lor y_L) = 1$, then each random XOR $[\sum_{j=1}^{L} \alpha_{i,j} y_j = 1 \mod 2]$ has probability $1/2$ of evaluating to 1. Therefore the OR of $k$ of them has probability at least $1 - 1/2^k$ of evaluating to 1. This completes the proof.

The XOR Trick Lemma shows how to reduce an “OR of $L$” into a “OR of $k$ XORs”. There is a completely analogous version of the trick using AND instead of OR. Namely:

**Lemma 3.2 (XOR Trick Lemma, Part II)** For every $k$ and $L$, there is a distribution of formulas $\mathcal{D}$, each of the form “$\land_{i}[\sum_{j=1}^{L} y_j \mod 2]$”, such that for all $y \in \{0, 1\}^L$,

1. If $(y_1 \land \cdots \land y_L) = 1$, then for every $G$ drawn from $\mathcal{D}$, $G(y_1, \ldots, y_L) = 1$.
2. If $(y_1 \land \cdots \land y_L) = 0$, then $\Pr_{G \sim \mathcal{D}}[G(y_1, \ldots, y_L) = 1] = 1/2^k$. 


Exercise 3.2 Prove Part II.

Now, we will use the XOR Trick to randomly reduce the formula for $OV_{s,d}$ into a polynomial over $F_2$ (the field of two elements, mod 2). The formal theorem we’ll prove is:

**Theorem 3.4 (OV Conversion Theorem)** For every $s, d$, there is a distribution $D$ of polynomials over $F_2$, where each polynomial has $s \cdot d$ variables and at most $M(s, d) := \text{poly}(s) \cdot \left(2^{2d}O(\log s)\right)$ monomials, such that for all possible inputs $v_1, \ldots, v_s \in \{0, 1\}^d$ to the $OV_{s,d}$ problem,

$$\Pr_{p \sim D}[OV_{s,d}(v_1, \ldots, v_s) = p(v_1, \ldots, v_s) \pmod{2}] \geq \frac{3}{4}.$$  

Moreover, we can construct a random $p$ from the distribution $D$ in $\text{poly}(M(s, d))$ time.

Such a distribution $D$ is often called a probabilistic polynomial, similar to a probabilistic algorithm. Its behavior is just like that of a good probabilistic algorithm: on every input, the correct answer is output with at least $3/4$ probability.

**Proof.** Recall we were working with

$$OV_{s,d}(V) = \bigvee_{i,j \in [s], i \neq j} \bigwedge_{k \in [d]} (-v_i[k] \lor -v_j[k]).$$

We apply the XOR Trick Part I and Part II to the $\bigvee$ and the $\bigwedge$, as follows.

(Step 1) To the big OR over $\binom{s}{2}$ terms, we apply the XOR Trick with parameter $k = 3$.

(Step 2) To each of the $\binom{s}{2}$ ANDs in the formula, we apply the XOR Trick (Part II) with $k = 3 + 2 \log(s)$.

First, let’s check that

$$\Pr_{p \sim D}[OV_{s,d}(v_1, \ldots, v_s) = p(v_1, \ldots, v_s)] \geq \frac{3}{4}.$$  

On any given input, the replacement to the top OR (the XOR Trick, Part I) contributes error at most $1/8$ to the result, and the replacement of each of the $\binom{s}{2}$ ANDs (the XOR Trick, Part II) contributes error at most $s^2 \cdot 1/(8s^2) \leq 1/8$. Therefore (by the union bound) the total error is at most $1/4$.

Now let’s show that the object we constructed from applying these Tricks corresponds to a polynomial with at most $\text{poly}(s) \cdot \left(2^{2d}O(\log s)\right)$ monomials.

Remember that modulo 2, AND is the same as multiplication, XOR is the same as addition, and

$$\text{NOT}(x) = 1 + x \pmod{2}.$$  

Therefore, by DeMorgan’s laws, the OR function on $k$ inputs $x_1, \ldots, x_k$ can be written as the degree-$k$ polynomial $1 + (1 + x_1) \cdots (1 + x_k) \pmod{2}$.

Let’s consider the effect of Step 1 on the OV formula. It replaces the big OR at the top with an OR of three XORs of $O(s^2)$ inputs. If we rewrite this as a polynomial using the above observations, we obtain a degree-3 polynomial on $O(s^2)$ variables, where each input to this polynomial is a formula from Step 2. This polynomial has $\text{poly}(s)$ monomials. Then, for each of the $O(s^2)$ ANDs in Step 2, we are applying the XOR Trick to replace each AND of $d$ inputs with an AND of $(3 + 2 \log(s))$ XORs, each of at most $d$ inputs.
Exercise 3.3  From here, show that the formula we got from applying the XOR Tricks in Steps 1 and 2 corresponds to a polynomial with at most $M(s, d) := \text{poly}(s) \cdot \left(\frac{2^d}{\log s}\right)$ monomials. Also check that we can construct this polynomial in $\text{poly}(M(s, d))$ time.

What does this theorem mean? It shows that we can convert the $OV_{s,d}$ function, which is an “OR of AND of ORs” into a polynomial with at most $M(s, d)$ monomials over $F_2$, for a certain $M(s, d)$. We can view such an $F_2$-polynomial as simply an XOR of at most $M(s, d)$ ANDs. That is, we are converting the $OV_{s,d}$ function which is a “depth three” circuit into a simpler type of logical expression, a “depth two” circuit. But there are two caveats.

1. Our conversion only works in a randomized way, so there is a decent chance that our randomized expression gives a wrong answer on a given input.
2. The original OV problem was a formula of size $O(s^2d)$. Our new expression has size $\text{poly}(s) \cdot \left(\frac{2^d}{\log s}\right)$, which could be much larger. (Remember we are interested in the case where $d = c \log(n)$, for a constant $c$.) For example, suppose $s = n^{0.1}$. Then the expression would have size $\text{poly}(n) \cdot \left(\frac{2^{c \log(n)}}{\log n}\right)$, which could be much larger than $n^2$. That would mean that even the conversion in the OR Conversion Theorem would already take super-quadratic running time... no way we’d get a subquadratic algorithm for OV in that case!

4 Solving OV Faster

Let’s turn to how we can use the polynomial representation to solve OV faster. First, let’s consider a simpler case, where we had a deterministic reduction from $OV_{s,d}$ to one polynomial $P_{s,d}$. Given an instance of OV with $n$ vectors in $d$ dimensions, we could then modify our original OV self-reduction as follows:

1) Divide the $n$ vectors into $O(n/s)$ groups, where each group has at most $s$ vectors each.

(2') Evaluate $P_{s,d}$ on all $(n/s)^2$ pairs of groups, where for large $s$, we want this “batch evaluation” to run in $\tilde{O}(n/s^2)$ time.

(3') If any evaluation returns true, then output ”yes” (there’s an OV pair) else ”no OV pair”.

How on earth might this run faster than $n^2$ time? The main question is how to implement (2'). Even if there is just one polynomial $P_{s,d}$ with $M = \left(\frac{2^d}{O(\log s)}\right)$ monomials, the obvious way of implementing (2') requires at least $\left(\frac{2^d}{O(\log s)}\right) \cdot n^2/s^2$ time, which (as far as we can tell) is never less than $n^2$ time. The key is the following lemma.

Lemma 4.1 (Batch Evaluation Lemma) Given any $F_2$-polynomial $P$ with $2m$ variables and at most $N^{0.1}$ monomials, and given $A, B \subseteq \{0, 1\}^m$, $|A| = |B| = N$, we can evaluate $P$ on all $N^2$ points in the Cartesian product $A \times B$ in $\tilde{O}(N^{1.1} \cdot m + N^2)$ time.

That is, given any polynomial that is “sparse enough”, we can evaluate it on many pairs of points in nearly optimal running time! Note that the obvious algorithm would take $\Omega(N^{2.1})$ time, and we’re taking only $\tilde{O}(N^2)$ time (assuming $m$ is small). We’ll cover how to prove the Batch Evaluation Lemma in the next lecture.

For us, we’ll choose the following parameters:

6
$N = n/s$ (there are $n/s$ groups) and $m = s \cdot d$ (each group has at most $s$ vectors of $d$ bits each).

If we can set $s$ so that the number of monomials in our polynomial $P_{s,d}$ is at most $N^{0.1} = (n/s)^{0.1}$, then we can apply the Batch Evaluation Lemma. It will turn out that setting $s = n^{1/O(\log c)}$ suffices (the calculations are done below).

### 4.1 The Randomized Case

However, we do not have a deterministic reduction from $OV_{s,d}$ to a single polynomial. We have a randomized reduction that produces one of potentially many possible polynomials. To handle that case, we have to be slightly more sophisticated, and run multiple trials of the evaluation process. Our modified self-reduction for OV (now turning into a real OV algorithm) becomes:

1. Divide the $n$ vectors into $O(n/s)$ groups, where each group has at most $s$ vectors each.
2. For $t = 60 \log(n)$ independent trials, draw random polynomials $P_1, \ldots, P_t$ from the distribution $\mathcal{D}$ that represent $OV_{2s,d}$, from the OV Conversion Theorem.
3. For $i = 1, \ldots, t$, evaluate $P_i$ on all $(n/s)^2$ pairs of groups (recall this is $(n/s)^2$ different inputs, each of length $2sd$).
4. For all $(n/s)^2$ pairs of groups with vectors $v_1, \ldots, v_{2s}$, compute $\text{MAJORITY}(P_1(v_1, \ldots, v_{2s}), \ldots, P_t(v_1, \ldots, v_{2s}))$.
5. If any $\text{MAJORITY}$ outputs 1, then output "yes" (there’s an OV pair) else "no".

#### Exercise 4.1

Why do steps (2a), (2b), (2c) work, and give us a randomized algorithm that outputs the correct answer with high probability?

Hint: Using a Chernoff bound on the $P_i$, you can show that for every pair of groups, the probability that $\text{MAJORITY}(P_1, \ldots, P_t)$ outputs the incorrect answer for that pair of groups is less than $1/n^3$. Therefore by a union bound over all pairs of groups, the probability that we get an incorrect answer from some pair of groups is less than $1/n$.

Assuming the above exercise, our algorithm for OV gives the correct answer with high probability. We now calculate how to set $s$ so that the algorithm runs as fast as possible.

Remember that the number of monomials in each $P_i$ is at most $\text{poly}(s)(c\log n)\big(D\log s\big)/\big(D\log(n)\big)$ for a constant $D > 1$.

We will apply the Batch Evaluation Lemma for $s := n^{\delta/(\log c)}$, for some constant $\delta > 0$. When we plug that setting of $s$ into our expression for the number of monomials, we get:

$$\text{poly}(n^{\delta/(\log c)}) \cdot \left(\frac{c\log(n)}{D\delta \log(n)/(\log c)}\right) \leq n^{D\delta/(\log c)} \cdot \left(\frac{e \cdot \log(n)}{D\delta \log(n)/(\log c)}\right)$$

(since $\left(\frac{N}{K}\right) \leq \left(eN/K\right)^K$)

$$\leq n^{D\delta/(\log c)} \cdot \left(\frac{e \cdot (c \log c)}{D\delta}\right)^{D\delta \log(n)/(\log c)}$$

$$\leq n^{D\delta/(\log c)} \cdot \left(\frac{e \cdot (c \log c)}{D\delta}\right)^{D\delta \log(n)/(\log c)}$$

$$\leq n^{D\delta/(\log c)} \cdot \left(\frac{e \cdot (D\delta)}{2D\delta \log(n)}\right)^{2D\delta \log(n)/(\log c)}$$

(since $\log(c \log c)/(\log c) \leq \log(c^2)/(\log c) \leq 2$)

$$\leq n^{D\delta/(\log c)} \cdot n^{2D\delta \log(\log c) /(D\delta)} \leq n^{D\delta/(\log c) + 2D\delta \log(\log c) /(D\delta)}.$$
Now, as $\delta \to 0$, we have $D\delta/(\log c) \to 0$ and $2D \cdot \delta \log(c/(D\delta)) \to 0$. Therefore we can always set $\delta$ to be a small enough constant so that the exponent of $n$ is less than 0.1.

Applying the Batch Evaluation Lemma, we can execute Step (2b) in time

$$\tilde{O}((n/s)^{1.1} \cdot s \cdot d + n^2/s^2) \leq \tilde{O}(n^{2-\delta/(\log c)}).$$

Therefore the entire algorithm takes at most this much time, and we are done!

References


A Answers to (Almost All) Exercises

A.1 An Algorithm for OV running in $O(dn2^d)$ time

For all possible $2^d$ vectors $u \in \{0,1\}^d$, check if there is a vector $v_i$ in our set equal to $u$. If so, then check if there is a vector $v_j$ in our set such that $\langle v_j, u \rangle = 0$. Both checks can be done in $O(dn)$ time by passing through the list of $n$ vectors, so the total running time is $O(dn2^d)$.

A.2 A Crucial Claim in the XOR Trick

Let $L > 0$ be an integer. We claim that for every nonempty subset $S \subseteq [L]$, and for a uniformly randomly chosen $R \subseteq [L]$, the probability that $|R \cap S|$ is odd is $1/2$.

Let $i^*$ be any element in $S$. Consider all $2^{L-1}$ subsets $R$ of $[L]$ that do not contain $i^*$, and all $2^{L-1}$ sets $R$ that do contain $i^*$. Pair these sets up in the natural way: pair up two sets if they agree on all elements except for $i^*$. Now observe that for each pair, exactly one of these sets has odd intersection with $S$, while the other has even intersection. Therefore exactly $2^{L-1}$ sets have odd intersection with $S$. 

8
A.3 Part II of the XOR Trick

We pick $k$ random XORs as before, with the following modification: In each of the random XORs, include $+1$ if the number of variables in the XOR we chose is even, otherwise include $+0$ (add nothing). Then, if all $L$ inputs are 1, then every random XORs is odd, so the AND of all of the XORs is always true. Similarly as before, if at least one of the $L$ bits is 0, then there’s probability 1/2 that the random XOR is even.

A.4 Why Taking Majority Works (Applying a Chernoff Bound)

First, we prove the probability bound on $\text{Majority}(P_1, \ldots, P_t)$.

Remember that on any given input $x$, $\Pr[P_i(x) = \text{OV}_{2s,d}(x)] \geq 3/4$.

Let $X_i$ be the random variable which is 1 if the condition $[P_i(x) = \text{OV}_{2s,d}(x)]$ is true, and 0 otherwise.

Since the $P_i$ are independent, the $X_1, \ldots, X_t$ are independent random variables, each having the property that $\Pr[X_i = 1] \geq 3/4$.

Let $X = \sum_i X_i$, and note that its expectation $\mathbb{E}[X] \geq 3t/4$.

One convenient form of the Chernoff bound (a.k.a. Hoeffding bound) tells us that for all $d \in (0, 1)$,

$$\Pr[X < \mathbb{E}[X](1 - d)] \leq e^{-d^2 t/2}.$$ 

Therefore we have

$$\Pr[\text{Majority}(P_1(x), \ldots, P_t(x)) \neq \text{OV}_{2s,d}(x)] = \Pr[\sum_i X_i < t/2]$$
$$= \Pr[X < t/2]$$
$$= \Pr[X < 3t/4 \cdot (1 - d)] \text{ for } d = 1/3: 3/4 \cdot 2/3 = 1/2$$
$$\leq e^{-(1/3)^2t/2} = e^{-t/18}.$$ 

Recall that we set $t = 60 \log(n)$, so $e^{-t/18} < 1/n^3$. So the probability that for some group, we conclude the wrong answer for $\text{OV}_{2s,d}$ is

$$\Pr[\text{For some group input } x, \text{Majority}(P_1(x), \ldots, P_t(x)) \neq \text{OV}_{2s,d}(x)] \leq (n/s)^2 \cdot 1/n^3 \quad \text{by the union bound}$$
$$< 1/n.$$ 

Therefore the algorithm outputs the correct answer for every group (and therefore the self-reduction is correct) with probability at least $1 - 1/n$. 

9