Fine-grained complexity of Orthogonal Vectors Problem and Variants

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Kamuju Sushma
(112001014)

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**Fine-grained complexity of Orthogonal Vectors Problem and Variants**" is a bonafide work of **Kamuju Sushma (Roll No. 112001014)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Krishnamoorthy Dinesh
Assistant Professor
Department of Computer Science & Engineering
Indian Institute of Technology Palakkad

Acknowledgements

I would like to acknowledge and give my deepest gratitude to my supervisor, Dr.Dinesh Krishnamoorthy, for their invaluable guidance, unwavering support, and constant encouragement throughout the entire process of completing this thesis.

I also thank my senior, Chandana S (112303001) for taking part in the project discussions and for many of the technical ideas discussed in this thesis. I am also grateful to IIT Palakkad for providing the necessary resources and a conducive environment for research and learning.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given a decision problem, we often analyze the problem based on the amount of resources that are needed to solve it i.e., time and space required as a function of the input size. Designing an algorithm for the problems gives an upper bound on the amount of resources needed. An efficient way to solve a problem is if it has an algorithm that runs in polynomial time of input length or solvable in space that is logarithmic in input length which is one way to define space efficiency, which we are very much interested in. The theory of NP-completeness (for time) and NL-completeness (for space) answers what set of problems can be efficiently solved using the above mentioned notions of efficiency.
Looking at the problems that belong to class P which are solvable in polynomial time, there are several problems that seems to have defied any improvement. In this project, we aim to look at such problems and also build a theory similar to NP-completeness for them. This has already been done in literature (For a recent survey, see [1]).

1.1 Motivation: Fine-grained complexity

The classification of problems via the theory of NP-hardness enables us to distinguish the very hard problems (like the Satisfiability Problem, which are covered by NP-completeness) and on the other hand, some problems can be solvable in polynomial time. If the input data is very large then the time taken to run the algorithm will also be very high for polynomial-time algorithms. If we look within the problems solvable in polynomial time, there are many problems for which we do not know the best algorithms for them. Every problem we encounter, we would ideally try to achieve a linear time-solving algorithm to it, which is the best algorithm a problem can have. But it is not always possible to do so, instead we try to get a better algorithm than the current best-known algorithm to it.

We have two ways to get a better algorithm. One way is to design a better algorithm for it. Another way is to do fine-grained reductions between the problems using the following manner.

Consider you know a better algorithm for problem B and you want to get a better algorithm for A, if you can somehow reduce the problem A to B then you can use the algorithm of B. This results in a better algorithm for A.
In this thesis, we focused on both ways to design algorithm, in the first way of designing algorithm we focused on defining the 3SUM problem running in $O(n^2)$ (where n is the size of input) and proved its correctness (see Chapter 2). Secondly, we focused on designing and proving reductions between orthogonal vectors problem and its variants (see Chapter 3).

This shows that using reductions if we succeeded in finding better algorithms for one of the orthogonal vector problems, we succeed in finding better algorithms for all of them.

1.2 Problems of interest

This thesis describes various algorithms but primarily focuses on the orthogonal vectors(OV) problem and its variants. Here, the input is 2 collections($\in \{0,1\}^d$) each of size n.
For the given input of 2 collections, the algorithm should find if there is a pair of vectors from A and B respectively, where their dot product is 0. The best-known algorithm for this has the time complexity $O(n^2 \log n)$ (OV).

There are many variants to the Orthogonal Vectors Problem, for example, $\mathsf{OV}Search$, which takes input as the 2 vectors collections and outputs the pair of vectors whose inner product is 0. Similarly, there is a variant called OV' which takes a single collection and outputs a pair of vectors from it if there is any orthogonal pair. Another variant is the $MaxIP+$, which takes in 2 collections as input and outputs the maximum inner product a pair of vectors from both collections can have. The problem $CountOV$ takes in both the collections as an input and outputs the number of orthogonal pairs of vectors present in them.

Then, we look at the Hitting Set problem. It is a decision problem, which takes both collections as input and outputs yes if there is a vector in 1st collection whose inner product with all vectors in the second collection is not zero. Finally, we look at the $3SUM$ problem where, from the given list of numbers, should find if there is a triplet (distinct numbers forming the triplet) which sums to 0. The best-known algorithm is $O(n^2)$. We will discuss these problems in future sections.

Now, let's look at the DFA evaluation problem, which is solvable in linear time, denoted as $O(n)$, where n is the size of the input string. We have the transition function based on the input string bit and the present state, so it is possible to get the next possible state in $O(1)$ time. This is because for a DFA, going to multiple states from a single state is not possible. So the overall time complexity of the problem is $O(n)$. However, in the case of NFA, it is not easy as compared to DFA because, for an NFA from a state, it is possible to go from a single state to all the states. So the overtime complexity if we try in brute force will be $\sigma(Q+Q^2+\ldots+Q^n)$. Using dynamic programming, we have a better algorithm for this, it is not possible to have a better solution than this because of "reduction".

1.3 Contributions of this thesis

Designed and proved the correctness of the reductions mentioned as solid lines in the diagram. They are described in this thesis.

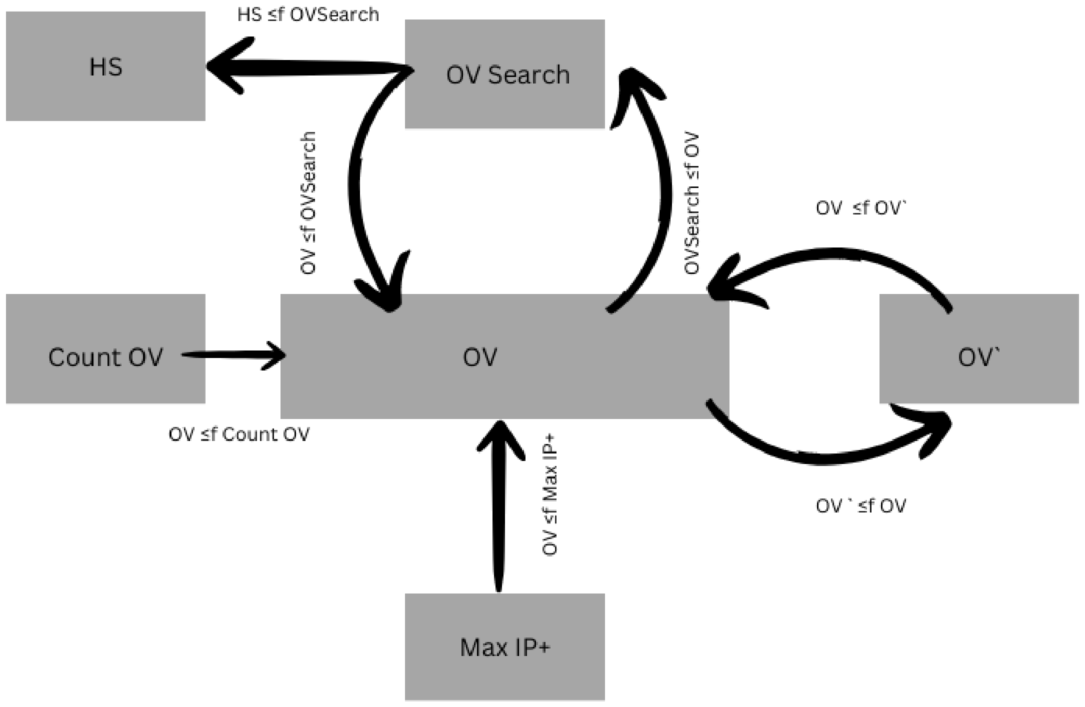

Fig. 1.1 Reductions

S. No.	Reduction	Description
1.	$OV \leq_f MaxIP+$ (theorem:5)	Reduction from OV to MaxIP+
2.	$OV' \leq_f OV_d$ (theorem:2)	Reduction from OV' to OV_d $OV' \leq_f OV_d$
3.	$OV \leq_f OV'$ (theorem: 1)	Reduction from OV to OV' $OV \leq_f OV'$
4.	$OVSearch \leq_f OV$ (theorem: 3)	Reduction from OV Search to OV
5.	$HS \leq_f OVSearch$ (theorem:4)	Reduction from HS to OV Search
6.	$OV \leq_f CountOV$ (theorem:6)	Reduction from OV to Count OV

Table 1.1 Reductions between variants of Orthogonal Vectors Problem

Chapter 2

Fine Grained Complexity: An Introduction

2.1 Reductions and Fine Grained Reduction

Fine-grained Reduction:

A fine-grained reduction is a transformation from one computational problem to another. Consider a scenario where we have two problems A, B and we attempt to improve the algorithm for A. What the fine-grained reduction states is that improving the algorithm for problem B could lead to an improvement in the algorithm for problem A.

The main difference between fine-grained reduction and polynomial reduction is the introduction of time bounds. In fine-grained reduction, the transformed algorithm of A should have lower time complexity compared to the original, while polynomial reduction doesn't impose any such constraint.

Essentially, the time complexity of solving problem A via fine-grained reduction is the sum of $\sigma(x)$ and the time complexity of solving problem B, where $\sigma(x)$ represents the time taken to transform an instance of A into an instance of B. This method focuses on reducing the overall time it takes, which directly solves problem A through the reduction.

Therefore, improving the transformation process from A to B becomes important in fine-grained reduction, with a focus on minimizing both $\sigma(x)$. Hence if there is a better algorithm for B then there will be a better algorithm for A.

2.2 Two problems: NFA Acceptance and Orthogonal Vectors

In this section, we introduce two more problems the NFA acceptance problem and the Orthogonal vectors problem.

Our aim is to understand a fine-grained reduction from NFA acceptance to Orthogonal Vectors problem as a part of the literature survey. This result is proved in [2].

We start by giving the definitions of the two problems.

Definition 1. ***(NFA acceptance)*** *Given a NFA $N = (Q, \Sigma, \Delta, s, F)$ with set of states Q as a lookup table of order $|Q| \times |\Sigma|$ of size m and a string $s \in \Sigma^*$ of length n, check if the NFA N accepts s on some non-deterministic choice.*

We now define the orthogonal vectors problem.

Definition 2. ***(Orthogonal vectors)*** *Given two collections, $A \subseteq \{0,1\}^d$ and $B \subseteq \{0,1\}^d$, where the length of each vector in both sets is d. Let v_1 be any vector from set A, and v_2 be any vector from set B. The task is to determine whether there exists a pair of vectors v_1 and v_2 such that their dot product $\langle v_1, v_2 \rangle$ equals 0.*

Output `true` if such a pair exists; otherwise, output `false`.

We start by giving standard algorithms for these problems and argue their correctness. Following this, we briefly discuss the connection between NFA acceptance and the Orthogonal Vectors problem and use this to motivate the Orthogonal vectors hypothesis.

NFA acceptance: An algorithm and its correctness The result in this section is from [2]. Below we are explaining this result as a part of literature survey. The algo-

Algorithm 2.2 Algorithm for NFA acceptance

```
Initialize a boolean array $array[n+1][Q]$
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow 0$ to $Q-1$ do
        if $i = 0$ then
            if $j = 0$ then
                $array[i][j] \leftarrow$ true
            else
                $array[i][j] \leftarrow$ false
            end if
        else
            if $array[i-1][j]$ then
                Initialize an empty vector $v$
                $v \leftarrow$ transitionFunction(state[$i$], string[$j$])
                for each $k$ in $v$ do
                    $array[i][k] \leftarrow$ true
                end for
            end if
        end if
    end for
end for
if $array[n][m-1]$ then
    print("The string is accepted")
else
    print("The string is not accepted")
end if
```

rithm 2.1 is a standard dynamic programming based algorithm. We are skipping the proof of correctness.

Time Complexity The two for loops iterate over every pair (i, j) where $i < j$. The part in the innermost for loop takes $O(m)$ to iterate over the n states. Hence, the overall runtime is $O((n+1)\cdot \mid Q \mid \cdot \mid Q \mid). = O(m \cdot n)$.

A natural question at this point is: can NFA acceptance be solved in $o(m \cdot n)$ time. We see that the answer to this question is related to the Orthogonal vectors problem defined before. We now describe algorithms for orthogonal vector problems, discuss their runtime and argue correctness.

Orthogonal vectors Firstly observe that n vectors from A and n vectors form B, one could iterate over all n^2 pairs and check if their dot product is 0. This is a trivial algorithm for this problem which takes a time of $O(n^2 d)$. Below, we give an improved algorithm that runs in $O(n^2 \log n)$ irrespective of the value of d.

We first describe an algorithm for this problem which is better than the brute-force algorithm in terms of its dependence on n in Algorithm 2.3. The idea is as follows: we maintain an array of order $n \times 2^d$ which is populated with 0 or 1 as follows: for each $v_1 \in A$ and for each $w \in \{0,1\}^d$, the array$[v_1][w] = \langle v_1, w \rangle$, that is we compute the inner product between the two vectors. We repeat this for all the vectors in A. Finally, we go over the entire array and check if there is a $v_2 \in B$ such that array$[v_1][_2]$ is 0. If we find such a v_2, the algorithm returns true. If no such v_2 is found for all $v_1 \in A$ the algorithm returns false. We now discuss the correctness of Algorithm 2.3 and its runtime.

Proof of Correctness of Algorithm 2.3

Proof. To argue correctness, we need to show that there exists an $V1 \in A$ and $V2 \in B$ with their inner product being 0 if and only if Algorithm 2.3 returns true.

Algorithm 2.4 An Algorithm for Orthogonal Vectors with runtime $O(nd2^d)$

```
function dotProduct( x,  y)
    int res $\leftarrow 0$
    for $i \leftarrow 0$ to $d - 1$ do
        res $\leftarrow$ res $+ x[i] \times y[i]$
    end for
    return res
end function

size $= 2^d$
int array$[n]$[size]
int$B'[size]$ contains all possible vectors from $\{0, 1\}^d$ of size d
for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow 0$ to size $- 1$ do
        array$[i][j] \leftarrow$ dotProduct$(A[i], B'[j])$
    end for
end for

for $i \leftarrow 0$ to $n - 1$ do
    for $j \leftarrow 0$ to size $- 1$ do
        if array$[i][j] == 0$ and $B'[j] \in B$ then
            Return true
        end if
    end for
end for
Return false
```

We argue the forward direction:

The first of the two loops enumerates over all the vectors in A and all possible 2^d vectors over $\{0, 1\}^d$ and populates the array with the inner product value. Since we know that inner product of v_1, v_2 is zero, we must have array$[i][j] = 0$ where i is the index of v_1 in the set A and j is the integer corresponding to the binary representation of v_2. Hence, the algorithm will return true.

We now argue the reverse direction:

Observe that second of the two for loops does a brute force search over all possible pairs of vectors chosen from A and all d length Boolean vectors. Since, the algorithm returned true, it must be that an array entry with 0 was encountered and the associated d length binary string is in B. Hence A, B has an orthogonal pair.

□

Time Complexity of Algorithm 2.3 The time complexity of the algorithm is primarily determined by the nested loops, iterating over all possible pairs of vectors from sets A and B. Specifically, the iteration involves $2^d \times n$ pairs, where 2^d represents the size of set B and n is the number of vectors in set A. Within each iteration, the dot product computation requires d multiplications, resulting in a time complexity of $2^d \times n \times d$. We again iterate over the array$array$ and if the entry is 0 we check if that corresponding $B[j]$ is present in set B which takes $2^d \times n \times n$ time. Therefore, the overall time complexity is $O(2^d \times n \times d)$.

2.3 Orthogonal Vectors Hypothesis

Final $O(n^2 \log n)$ algorithm for Orthogonal Vectors Problem The final algorithm is: run the brute-force algorithm along with the algorithm given above and take the output of the algorithm that terminates the first. Hence, the runtime of this algorithm is $O(\min\{n^2 d, n \cdot d \cdot 2^d)\})$. We now consider two cases: one when n is small (compared to 2^d) and other when n is large (compared to 2^d). If n is small (i.e., $n <<< 2^d$) the brute force algorithm will terminate first and if it is huge (i.e., $n >>> 2^d$) the algorithm 2.3 will terminate. The minimum is achieved when $d = \log n$ which gives a runtime of $O(n^2 \log n)$ irrespective of the value of d.

This is one of the best known algorithms for this problem. Several efforts to improve the above algorithm has not succeeded so far. This led researchers to consider the following hypothesis which is called as the orthogonal vectors hypothesis.

Orthogonoal vector hypothesis *For any $0 < \epsilon < 1$, there is no deterministic algorithm that can solve the orthogonal vectors problem in $O(n^{2-\epsilon}\mathsf{poly}(d))$ time.*

The orthogonal vectors problem was first studied in [3]. The best known algorithm for this problem runs in $O(n^{2-1/c}\mathsf{poly}(\log n))$ when $d = c \log n$ for any constant c [4]. No further improvement is known till date. This hypothesis is one of the central hypothesis in the area of fine grained complexity. In the next section, we attempt to give a better algorithm for this problem, when the input is structured.

2.4 Another approach for Orthogonal Vectors Problem:

Let the number of set bits in the vector $x \in A$ be k_1. Similarly, let the number of set bits in the vector $y \in B$ be k_2.(In binary notation, a set bit is represented by 1)

We partition collections A and B as follows:

- Partition A into A_1 and A_2, where A_1 contains elements with $k_1 \leq \frac{n}{2}$ and A_2 contains elements with $k_1 > \frac{n}{2}$.
- Partition B into B_1 and B_2, where B_1 contains elements with $k_2 \leq \frac{n}{2}$ and B_2 contains elements with $k_2 > \frac{n}{2}$.

Observation: We can choose v_1 and v_2 from:

1. A_1, B_1

2. A_2, B_1

3. A_1, B_2

we don't need to check the pairs from $A2$ and $B2$ because there dot product is never 0. The time complexity of this approach can be calculated as follows: $|x|$ = cardinal number of set X

$$
\begin{aligned}
&(|A_1| \cdot |B_1| + |A_2| \cdot |B_1| + |A_1| \cdot |B_2|) \cdot d \\
&= |A_1|(B_1 + B_2) \cdot d + |A_2| \cdot |B_1| \cdot d \\
&= |A_1| \cdot n \cdot d + |A_2| \cdot |B_1| \cdot d \\
&= |A_1| \cdot n \cdot d + |n - A_1| \cdot |B_1| \cdot d \\
&= |A_1| \cdot n \cdot d + (n) \cdot |B_1| \cdot d - |A_1| \cdot |B_1| \cdot d \\
&= n \cdot d \cdot (|A_1| + |B_1|) - |A_1| \cdot |B_1| \cdot d \\
&\leq n \cdot d \cdot (|A_1| + |B_1|)
\end{aligned}
$$

We can utilize this approach when we have knowledge of the count domain on $|A|$ and $|B|$. If $(|A_1| + |B_1|) < n^{1-e}$ then we have the better algorithm.

2.5 More reasons to study Orthogonal Vectors

We give more reasons to study the orthogonal vectors problem. The well known k-SAT problem asks the following: given a Boolean CNF formula ϕ on n variables where each clause has k literals, check if ϕ is satisfiable.

A result due to [3] showed that k-SAT is reducible to Orthogonal Vectors Problem. That is, if we can have better algorithm for OV, since k-SAT is reducible to OV, this would give a better algorithm for k-SAT.

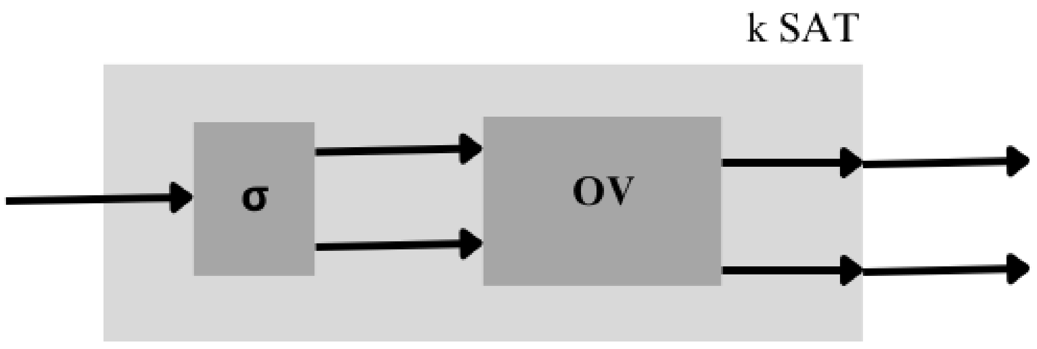

Fig. 2.1 $kSAT \leq OV$

2.6 A fine grained reduction illustrated

Consider a person has come to you with a problem A and asked you to help them find a better algorithm to it. Recalling the discussion from introduction, it is not always possible to find a new algorithm to the problem, but at least if we can identify the problems which are similar to it or somehow related, then it is possible to provide a better algorithm for problem A using fine grained reduction. Let us consider we wanted a better algorithm to A and we know a algorithm to problem B which has less time complexity. If it is possible to transform an input instance of A to input instance of B (via the map σ) such that yes instances of A gets mapped to yes instances of B and vice versa, then using the algorithm of B we can actually get better algorithm for A. We actually transform the problem A to B and solve it. Crucially, we also require the reduction σ and the algorithm for B to run faster than the algorithm for A. That is, $\mathrm{TC}(\sigma) + \mathrm{TC}(B) < \mathrm{TC}(A)$ is necessary to do a fine-grained reduction. Below we give an example of such a reduction.

In general it is not that we should call or use the problem B just once but we can

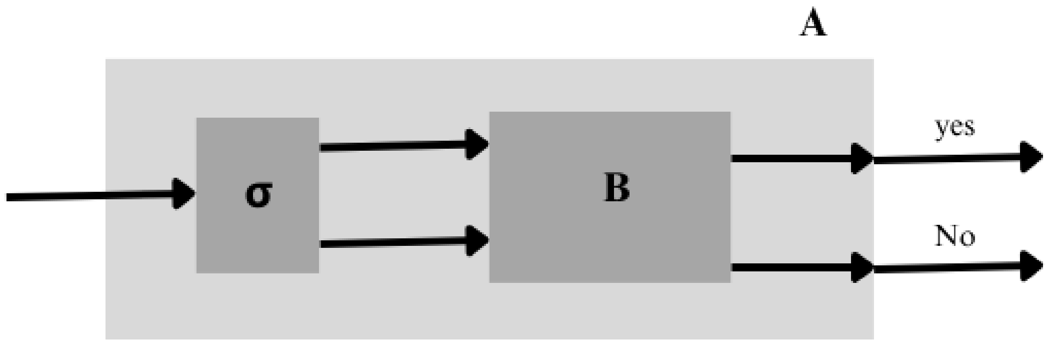

Fig. 2.2 $A \leq B$

actually call it multiple times and that type of reduction is known as **Turing reduction**.

We first define the MinIP problem below.

Definition 3 (Minimum Inner Product MinIP)**.** *Let n, d be non-negative integers.* ***Input:*** *Collections $A, B,$ of n vectors each from $\mathbb{R}^d$.*

Output: *Compute* $\min_{(v_1,v_2)\in A\times B}\langle v_1, v_2\rangle$.

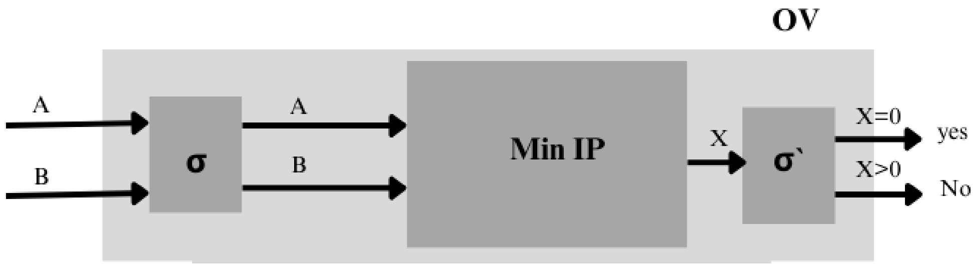

Fig. 2.3 OV $\leq_f$ $MinIP$

Claim: OV 2 $\leq_f$ $MinIP$ We now describe how the reduction works.

1. Transform the input of problem OV to Min IP input; here, the input remains the same.
2. Give the transformed input to the algorithm of Min IP.
3. Since Min IP is not a decision problem, we perform another transformation on the output of Min IP and use result of it as the output ofOV.

2.7 The 3SUM problem

3-SUM Problem In this section, we describe the 3SUM problem and give three algorithms for this problem (each improving the previous algorithm). This is also widely studied in the context of fine grained complexity. We include the details here as a part of literature survey. .

Definition 4. ***(3SUM Problem)*** *Given a list of n ,where $n \geq 3$, integers $a_1, a_2, \dots, a_n$, check if there exists three distinct integers a_i, a_j, a_k in this list such that $a_i + a_j + a_k = 0$.*

Below we give a brute-force algorithm for this problem which goes over all the triplets of numbers from the list and checks if they sum to 0. This means that the algorithm enumerates all $O(\binom{n}{3}) = O(n^3)$. The correctness follows since we explicitly check all the triplets.

Algorithm 2.5 Find Triplets with Sum 0 **3SUM (Brute-force)**

```
for $i \leftarrow 0$ to $n-3$ do
    for $j \leftarrow i+1$ to $n-2$ do
        for $k \leftarrow j+1$ to $n-1$ do
            $sum \leftarrow \text{array}[i] + \text{array}[j] + \text{array}[k]$
            if $sum = 0$ then
                return                                  ▷ Found triplet whose sum is 0
            end if
        end for
    end for
end for
```

A $O(n^2 \log n)$-time algorithm for 3SUM The algorithm is as follows, first we sort the input array then enumerate over all possible pairs and check if the negation of sum of that pair is present in the array (using binary search).

We use the function `binarySearch` which takes in 2 arguments, the array and the key to be searched. This function returns true if the key is present in the array and false otherwise.

Algorithm 2.6 Find Triplets with Sum 0

```
iterate over the input , count the frequency of $zero$ .
if the frequency is $\geq 3$ then return 1,
if the frequency is 2 then retain only 1 $zero$.
sort the input array using merge sort
for $i \leftarrow 0$ to $n-2$ do
    for $j \leftarrow i+1$ to $n-1$ do
        $t \leftarrow \text{arr}[i] + \text{arr}[j]$
        if binarySearch(arr, $-t$) then

            return 1
        end if
    end for
end for
return 0
```

Proof of correctness: We show that the algorithm returns 1 if and only if the input array has three integers that sum to 0. (write justification for using frequency as parameter) *Forward direction*: Suppose that the algorithm returns 1, then clearly the binary search succeeded in finding a number in the array whose value is $-(arr[i] + arr[j])$(Here binary search will succeed in finding because we sorted the array in line 4 in the Algorithm 2.6) Hence there must be a triplet in the input array that sums to 0.

Reverse direction: Suppose that the input array has three distinct numbers a, b, c such that $a+b+c = 0$. Without loss of generality assume $a \leq b \leq c$. Let p and q be the indices of the elements a and b in the sorted input array. When the outer for loop i becomes p and the inner for loop j becomes q, the value of t becomes $arr[p] + arr[q] = a + b$. In the next step, we search for the element $-(a + b) = c$ which must appear in the array. Since the array is sorted, the binary search will succeed and return 1.

The line 4 of the algorithm 2.6 sorts the input array which costs $O(n \log n)$. The two for loops iterate over every pair (i, j) where $i < j$. The part in the inner most for loop, takes $O(\log n)$ to do the binary search on a sorted array of size n. The overall time for running loops is $O(\binom{n}{2} \log n) = O(n^2 \log n)$. Hence the overall runtime is $O(n \log n + n^2 \log n) = O(n^2 \log n)$.

$O(n^2)$ time algorithm for 3SUM In this section, we describe a better algorithm which runs in $O(n^2)$ time which is better than the algorithms 2.5, 2.6(In fact this is the best known algorithm for this problem)

The result in this section is from [5]. Below we are explaining this result as a part of literature survey

Algorithm 2.7 Find Triplets with Sum 0

```
1: sort the input array using merge sort ;
2: for $i \leftarrow 0$ to $n-3$ do
3:     $a \leftarrow$ array[$i$]
4:     $start \leftarrow i+1$
5:     $end \leftarrow n-1$
6:     while $start < end$ do
7:         $b \leftarrow$ array[$start$]
8:         $c \leftarrow$ array[$end$]
9:         $sum \leftarrow a+b+c$
10:        if $sum = 0$ then
11:            return 1;
12:        else
13:            if $sum > 0$ then
14:                $end \leftarrow end - 1$
15:            else
16:                $start \leftarrow start + 1$
17:            end if
18:        end if
19:    end while
20: end for
21: return 0;
```

2.7.1 Proof of correctness

We show that the algorithm returns 1 if and only if the input array has three integers that sum to 0.

Forward direction: Suppose that the algorithm returns 1, this means the algorithm found a triplet whose sum is 0.we now show that the elements of the found triplet has distinct indices. Let the indices of the triplet be p, q, r. For $x \in \{i, start, end\}$ in the algorithm the value at the index x in the *array* is being assigned to distinct $y \in \{a, b, c\}$. Hence there is a one-to-one mapping between $\{p, q, r\}$ and $\{i, start, end\}$.

The variable i begins from 0 and goes till $n-3$. The variable *start* begins from $i+1$

which clearly conveys $i < start$. At line 6 of Algorithm 2.7 we compare *start* and *end* which always ensures $start < end$. Hence, we conclude that $i < start < end$. Since $\{p, q, r\}$ and $\{i, start, end\}$ are same the triplet a, b, c whose sum is 0 are having distinct indices in the input *array*.

Reverse Direction: If there a triplet whose sum is 0 in the array then algorithm will return 1. Let the triplet be (a, b, c). Without loss of generality let $a \leq b \leq c$ and the indices be ℓ, m, p respectively. We prove this by arguing the proof of the invariant while the $i = \ell$ in the for loop:

1. The range of *start* is from $\ell + 1$ to m
2. The range of *end* is from p to $n - 1$
3. If $start < m$ and $end > p$ then at the end of while loop ,either start will increase by 1 or end will decrease by 1.

Proof of (1) ,(2) by induction on number of iterations of while loop. The induction hypothesis is as follows ,

At the beginning of t^{th} iteration. $start_t$ lies in the range $l + 1$ to m and end_t lies in the range p to $n - 1$.

Here $start_t$ and end_t denote the value of start ,end (respectively) at the beginning of t^{th} iteration. At $t = 0$, $start_0 = l + 1$ and $end_0 = n - 1$ hence the induction hypothesis holds. Assume that the induction hypothesis holds for some value of t, We now show that the induction Hypothesis holds for $t + 1$. $start_{t+1}$ is atleast $start_t$ from the algorithm as we never decrement start . By induction hypothesis $start_t$ is atleast $\ell + 1$. Hence $start_{t+1}$ is atleast $\ell + 1$.

We now show $start_{t+1} \leq m$. We prove this by contradiction. Suppose $start_{t+1} > m$. Then, along with the induction hypothesis $start_t \leq m$ means that the *start* value is being increased in the $t + 1$ iteration. So this implies $a + array[start_t] + array[end_t] < 0$. We observe that $start_t$ should be m . If not $start_{t+1} \leq start_t + 1 \leq m$ that contradicts to our previous assumption $start_{t+1} > m$. Hence $a + array[start_t] + array[end_t] = a + array[m] + array[end_t]$.By induction hypothesis $p \leq end_t$. Hence $a + array[m] + array[end_t] \geq a + array[m] + array[p] = 0$ which is a contradiction.So $start_{t+1} \leq m$.

end_{t+1} is atmost $start_t$ from the algorithm as we never increment end . By induction hypothesis end_t is end $n-1$. Hence $start_{t+1}$ is atmost $n-1$.

We now show $end_{t+1} \geq p$. We prove this by contradiction. Suppose $end_{t+1} < p$. Then, along with the induction hypothesis $end_t \geq p$ means that the end value is being decreased in the $t+1$ iteration. So this implies $a + array[start_t] + array[end_t] > 0$. We observe that end_t should be p . if not $end_{t+1} \geq end_t + 1 \geq p$ that contradicts to our previous assumption $end_{t+1} < p$. Hence $a + array[start_t] + array[end_t] = a + array[start_t] + array[p]$.By induction hypothesis $p \leq end_t$. Hence $a + array[start_t] + array[p] \leq a + array[m] + array[p] = 0$ which is a contradiction.So $end_{t+1} \geq p$. Conclusion $l+1 \leq$ start $\leq m$ and $p \leq$ end $\leq$ start $\leq n-1$.

proof of (3) , assuming there is only triplet whose $sum = 0$. only when $start = m$ and $end = p$ the $suma + arr[start] + arr[end] = 0$. If not then the $sum > 0$ or $sum < 0$. from the algorithm if the $sum > 0$ then end will be decremented and if the $sum < 0$ then $start$ will be incremented .

Proof of correctness of algorithm : According to algorithm it does not increment the $start$ when it is m and it does not decrement the end when it is p. We fix the $start$, we know the range of end to be (p,n-1) so $end > p$, as the array is sorted $arr[end] > arr[p]$, $a + arr[m] + arr[end] > a + arr[m] + arr[p]$. We know $a + arr[m] + arr[p] = 0$. So $a + arr[m] + arr[p] > 0$ from the algorithm if the sum is greater than 0 then end will be decremented Hence the algorithm does not increment $start$.

we fix the end,we know the range of $start$ to be (l+1,m) so $start < m$, as the array is sorted $arr[start] > arr[m]$, $a + arr[start] + arr[p] < a + arr[m] + arr[p]$. we know $a + arr[m] + arr[p] = 0$.so $a + arr[start] + arr[p] < 0$. The sum is less than 0 so $start$ will be incremented Hence the algorithm does not decrement end.

Fix the *end*, and increment *start*, when the *start* is is the range of $l+1$ to $m-1$ the value is less than 0

2.7.2 Time Complexity

The first line of the algorithm sorts the input array which costs $O(n \log n)$. The outer loop iterate over every single element of list of size n . The inner loop takes $O(n)$. Hence, the overall time is $O(n^2)$. Hence the overall runtime is $O(n \log n + n^2) = O(n^2)$.

Chapter 3

Fine-grained Reductions between variants of Orthogonal Vectors

In this chapter, we consider some known variants of orthogonal vectors problem and fine grained reductions among them. The orthogonal vectors problem was introduced in Chapter 2 where its importance is discussed in Section 2.5.

3.1 Variants of Interest

We start with the usual variant of the orthogonal vectors problem (defined in 2), which we denote by OV, where the task is to check, given A, B check if there exists an $v_1 \in A$ and $v_2 \in B$ such that $\langle v_1, v_2 \rangle = 0$ which we defined in Chapter 2 (Section 2.2). This motivates three other variants of the problem where we switch the quantifiers on $v_1 \in A$ and $v_2 \in B$ and the $\langle v_1, v_2 \rangle$ final check. The variants are listed in Table 3.1.

In total, there should have been eight variants, but the other four variants will essentially be complement of the above four problems and hence are not listed. By a *collection*, we mean a multi-set.

We define these problems starting with OV below which are the decision variants.

Definition 5 (Orthogonal Vectors OV)**.** *Let n, d be non-negative integers.*

Problem	Quantification on $v_1 \in A$	Quantification on $v_2 \in B$	$\langle v_1, v_2 \rangle$ check
OV	$\exists$	$\exists$	$\langle v_1, v_2 \rangle = 0$
NOV	$\exists$	$\exists$	$\langle v_1, v_2 \rangle \neq 0$
HS	$\exists$	$\forall$	$\langle v_1, v_2 \rangle \neq 0$
AOV	$\exists$	$\forall$	$\langle v_1, v_2 \rangle = 0$

Table 3.1 Variants of Orthogonal Vectors Problem

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *yes, if there exists distinct vectors $v_1 \in A$, $v_2 \in B$ such that $\langle v_1, v_2 \rangle = 0$, and no otherwise.*

Definition 6 (Non-Orthogonal Vectors NOV)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *yes, if there exists distinct vectors $v_1 \in A$, $v_2 \in B$ such that $\langle v_1, v_2 \rangle \neq 0$, and no otherwise.*

Definition 7 (Hitting Set HS)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *yes, if there exists $v_1 \in A$ for all $v_2 \in B$ such that $\langle v_1, v_2 \rangle \neq 0$, and no otherwise.*

Definition 8 (Always Orthogonal AOV)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *yes, if there exists $v_1 \in A$ for all $v_2 \in B$ such that $\langle v_1, v_2 \rangle = 0$, and no otherwise.*

We complete the decision variants with the following single set variant of OV.

Definition 9 (OV$'$)**.** *Let n, d be non-negative integers.*

Input: *A collection A of n vectors from $\{0,1\}^d$.*

Output: *yes, if there exits distinct vectors $v_1 \in A$, $v_2 \in A$ such that $\langle v_1, v_2 \rangle = 0$, and no otherwise.*

These problem variants have been studied (or is defined implicitly) in the following works. More precisely MaxIP^+ has been studied in the context of inapproximability within sub-quadratic time [6]. As mentioned before Orthogonal vectors has been studied in [3, 4, 7]. We could not find any references for AOV, HS and NOV though they are very natural variants to be defined.

We now define related variants which includes functional (MaxIP, MaxIP^+, MinIP) search variant ($\mathsf{OV}\mathit{Search}$) and counting variants (countOV).

Definition 10 (Maximum Inner Product MaxIP)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\mathbb{R}^d$.*

Output: *Compute $\max_{(v_1,v_2)\in A\times B}\langle v_1, v_2\rangle$.*

Definition 11 (Minimum Inner Product MinIP)**.** *Let n, d be non-negative integers.* ***Input:*** *Collections A, B, of n vectors each from $\mathbb{R}^d$.*

Output: *Compute $\min_{(v_1,v_2)\in A\times B}\langle v_1, v_2\rangle$.*

Definition 12 (Maximum Inner Product MaxIP^+)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\mathbb{R}^d_{\geq 0}$.*

Output: *Compute $\max_{(v_1,v_2)\in A\times B}\langle v_1, v_2\rangle$.*

We now define the search variant of OV.

Definition 13 (Search Orthogonal Vectors $\mathsf{OV}\mathit{Search}$)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *Output $(v_1, v_2) \in A \times B$ such that $\langle v_1, v_2\rangle = 0$, if it exists and null set otherwise.*

Definition 14 (Count Orthogonal Vectors countOV)**.** *Let n, d be non-negative integers.*

Input: *Collections A, B, of n vectors each from $\{0,1\}^d$.*

Output: *Output $|\{(v_1, v_2) \in A \times B \mid \langle v_1, v_2\rangle = 0\}|$.*

3.2 Fine grained reductions between the variants

3.2.1 OV and OV′ are equivalent under fine grained reductions

In this section, as a warm up, we show that $\mathsf{OV} \leq_f \mathsf{OV}'$ and $\mathsf{OV}' \leq_f \mathsf{OV}$.

Theorem 1. $\mathsf{OV} \leq_f \mathsf{OV}'$

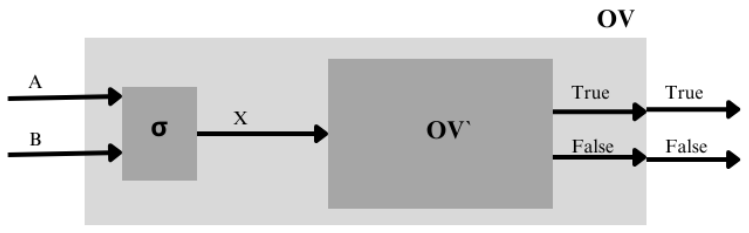

Fig. 3.1 $\mathsf{OV} \leq \mathsf{OV}'$

Proof. Let A, B be collections of $\{0,1\}^d$ of size n each. We show the following:

1. There exists an algorithm σ that takes in the sets A, B and outputs X in time $O(nd)$.
2. For any A, B the following holds:

$$(A, B) \text{ is an yes instance of } \mathsf{OV} \iff X \text{ (where } X = \sigma(A,B)) \text{ is an yes instance of } \mathsf{OV}'$$

Proof of (1) : Given A, B, construct $A' \subseteq \{0,1\}^{d+2}$ where $A' = \{(a_1, \ldots, a_d, 1, 0) \mid (a_1, \ldots, a_d) \in A\}$ and similarly, $B' = \{(b_1, \ldots, b_d, 0, 1) \mid (b_1, \ldots, b_d) \in B\}$. The output of σ is $A' \cup B'$. Clearly, σ can be computed in $O(nd)$ time.

Proof of (2): We start with the forward direction: suppose (A, B) is a yes instance of OV. Hence, there exists $v_1 \in A$ and $v_2 \in B$ such that $\langle v_1, v_2 \rangle = 0$. By the map σ defined above, there exists v_1' and v_2' in X where v_1' is v_1 concatenated with $(1, 0)$ and v_2' is v_2 concatenated with $(0, 1)$. By construction, $\langle v_1', v_2' \rangle = \langle v_1, v_2 \rangle$. Since $\langle v_1, v_2 \rangle = 0$, we can conclude that X is a yes instance of OV'.

We now argue the reverse direction: suppose X is a yes instance of OV'. Then there exists distinct vectors $v_1', v_2' \in X$ such that $\langle v_1', v_2' \rangle = 0$. By construction, there exists v_1, v_2 such that v_1' is v_1 concatenated with (a, b) and v_2' is v_2 concatenated with (c, d) where (a, b) as well as (c, d) belongs to $\{(0, 1), (1, 0)\}$. Since $\langle v_1', v_2' \rangle$ is 0, it can never be the case that $(a, b) = (c, d)$, for else, we get the inner product to be at least 1. Hence, by construction it must be that v_1 and v_2 has not come from the same set. Hence either $(v_1, v_2) \in A \times B$ or $(v_2, v_1) \in A \times B$. As already argued previously, the inner product of v_1, v_2 must be same as v_1', v_2'. Hence, (A, B) is a yes instance of OV.

□

Theorem 2. $\mathsf{OV}' \leq_f \mathsf{OV}$

Proof. Let A, B, X be collections vectors in $\{0, 1\}^d$ where A, B and X are of size n. We now give the reduction and prove its correctness:

1. There exists an algorithm σ that takes in the sets A, B and outputs X in time $O(nd)$.
2. For any x the following holds:

$$X \text{ is an yes instance of } \mathsf{OV}' \iff (A, B) \text{ where } (A, B) = \sigma(X) \text{ is an yes instance of } \mathsf{OV}$$

Proof of (1): Consider the algorithm σ that takes in the set X and outputs A,B where $A = B = X$. Observe that this takes time of $O(nd)$.

Proof of (2): We start with forward direction: Consider an yes instance X of OV'. This implies there is pair of vectors $v_1, v_2 \in X$ whose inner product is 0. Then there will be

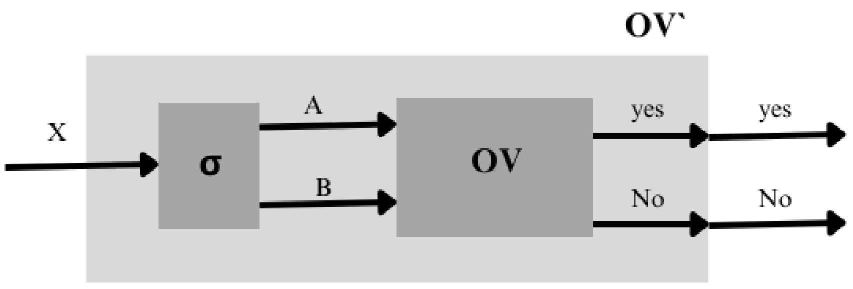

Fig. 3.2 $\mathsf{OV}' \leq_f \mathsf{OV}$

pair of vectors in $A \times B$ such that their inner product is 0. Hence A, B is an yes instance of OV. We now argive the reverse direction: Suppose X is no instance of OV', then there are no pair of vectors in X that have inner product as 0, in which case it is not possible to get any pair of vectors in $A \times B$ whose inner product is 0. Hence A, B is no instance of OV. $\square$

3.2.2 OV and its Search Variant

In this section, we show the reduction between OV Search and OV ($\mathsf{OV}Search \leq_f \mathsf{OV}$)

Theorem 3. $\mathsf{OV}search \leq_f \mathsf{OV}$

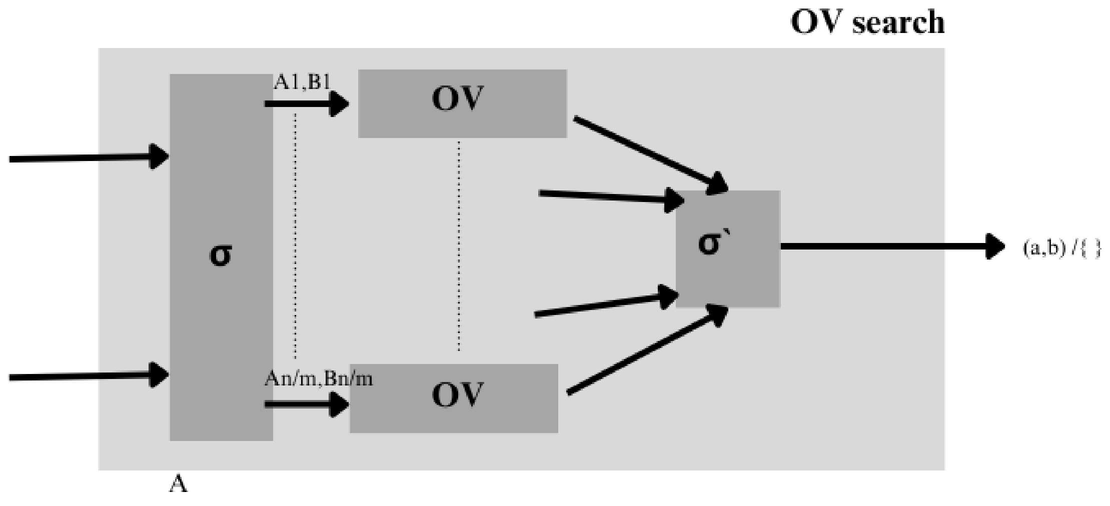

Fig. 3.3 OV Search$\leq$ OV

Proof. Let A, B be collections of vectors from $\{0,1\}^d$ both of size n. We first describe the reduction which consists of σ and σ' following which we argue the correctness.

1. The algorithm σ takes in the set A, B and divides A into smaller collections each of size m (where $m < n$, which will be fixed later) at random. Repeat the same with B and outputs $\{A_1, A_2, \ldots, A_{\frac{n}{m}}\}, \{B_1, B_2, \ldots, B_{\frac{n}{m}}\}$ in time $2 \cdot nd = O(nd)$

2. The algorithm σ' does brute force searches over all pairs A_i and B_j to find it any one of them have an orthogonal pair using the an algorithm for OV. This involves calling the OV algorithm $\left(\frac{n}{m}\right)^2$. If one of the pairs A_i, B_j returns a yes, we brute-force over all elements in A_i, B_j and find the orthogonal pair. If no pair was found, we return that there is no pair. This takes $O(m^2 d)$ time.

3. For any A_i, B_i where $A_i \in \{A_1, A_2, \ldots, A_{\frac{n}{m}}\}, \quad B_j \in \{B_1, B_2, \ldots, B_{\frac{n}{m}}\}$ the following holds:

$$(A, B) \in \mathsf{OV} \iff \exists i, j (A_i, B_j) \text{ is a yes instance of } \mathsf{OV}.$$

Proof of 3: We first prove the forward direction:

Consider yes instance of OV. This implies there is pair of vectors, say, v_1, v_2 where $v_1 \in A$ and $v_2 \in B$ such that their dot product is 0. After dividing A, B into smaller collections let $v_1 \in A_i, V_2 \in B_j$. Hence $(A_i, B_j) \in \mathsf{OV}$.

We now prove the reverse direction: Consider a no instance (A, B) of OV. This implies these exists no pair of vectors in (A, B) such that their dot product is 0. Hence none of the pairs in $\{A_1, A_2, \ldots, A_{\frac{n}{m}}\} \times \{B_1, B_2, \ldots, B_{\frac{n}{m}}\}$ belongs to OV.

$\square$

The time taken to do OV for all the $(\frac{n}{m})^2$ sets is $\left(\frac{n}{m}\right)^2 \times \mathrm{OV}(m, d)$.

Time Complexity We start by assuming that there is a deterministic algorithm that solved OV in time $\mathsf{OV}(n, d) := n^{2-\epsilon}\mathsf{poly}(d)$ time for some $\epsilon > 0$. We then show that $\mathsf{OV}Search$ can be solved in $n^{2-2\epsilon}\mathsf{poly}(d)$ time.

- From the description of σ and σ', the overall time taken by the $\mathsf{OV}Search$ algorithm is $\left(\frac{n}{m}\right)^2 \cdot \mathrm{OV}(m, d) + m^2 \cdot d + O(nd)$. Ignoring the last term (as the rest of the terms are larger) and substituting for $\mathsf{OV}(m, d)$, this can be simplified as,

$$\begin{aligned}\left(\frac{n}{m}\right)^2 \cdot m^{2-\epsilon} \cdot \mathrm{poly}(d) + m^2 \cdot d &\leq \left(\frac{n}{m}\right)^2 \cdot m^{2-\epsilon} \cdot \mathrm{poly}(d) + m^2 \cdot \mathrm{poly}(d) \\ &\leq (\left(\frac{n}{m}\right)^2 \cdot m^{2-\epsilon} + m^2)\mathrm{poly}(d)\end{aligned}$$

- Choosing $m = n^{(\frac{2}{2+\epsilon})}$, the time complexity is $n^{(2-2\epsilon)} \cdot \mathrm{poly}(d)$. This can be argued by observing[1] that $n^{\frac{2}{2+\epsilon}} \leq n^{1-\epsilon}$.

  We remark that a similar divide and conquer based idea has also been used in related fine-grained settings (See [8] for details).

[1]The details are as follow: $n^{\left(\frac{1}{1+\frac{\epsilon}{2}}\right)} = n^{(1+\frac{\epsilon}{2})^{-1}} = n^{1-\left(\frac{\epsilon}{2}\right)+\left(\frac{\epsilon}{2}\right)^2+\cdots} \leq n^{1-\epsilon}$, where the second equality follows from Taylor expansion.

3.2.3 Hitting set reduces to OV

In this section, we show that the hitting set problem reduces to OVSearch. Since we have already shown in Theorem **??** that OVSearch reduces to OV, this implies that Hitting set reduces to OV.

Theorem 4. HS $\leq_f$ OV$Search$

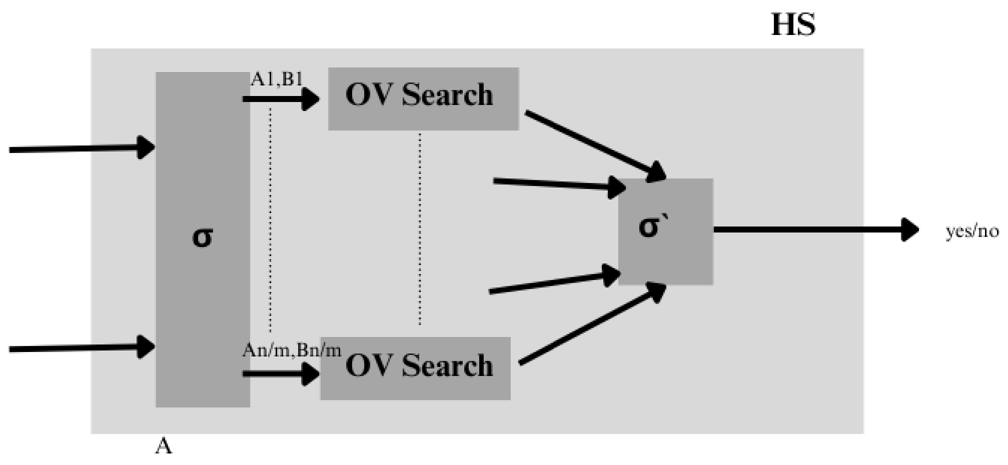

Fig. 3.4 HS $\leq$ OV$Search$

Proof. Let A, B be collections of vectors from $\{0,1\}^d$ both of size n. We first describe the reduction which consists of σ and σ' following which we argue the correctness.

1. Consider the algorithm σ that takes in the set A, B and divides A into smaller sets each of size m (where $m < n$, which will be fixed later) arbitrarily. Repeat the same B and outputs $\{A_1, A_2, \ldots, A_{\frac{n}{m}}\}, \{B_1, B_2, \ldots, B_{\frac{n}{m}}\}$. This is doable in time $2 \cdot nd = \mathrm{O}(nd)$

2. The algorithm σ' does the following: for each $i \in \{1, 2, \ldots, n/m\}$ and for each $j \in \{1, 2, \ldots, n/m\}$, σ' runs OV$Search$ on the pair (A_i, B_j) and removes all the vectors

in A_i (to get A'_i) for which the inner product with some element in B_j is zero. This can be found from the result of $\mathsf{OV}\mathit{Search}$. We repeat this for all the B_j's. Once all the B_j's are checked, if the resulting A_i is non-empty, return yes. Else proceed to the next A_i. At the end, if all A_i's turned out to be empty, return no.

3. For any A_i, B_i where $A_i \in \{A_1, A_2, \dots, A_{\frac{n}{m}}\}, \quad B_j \in \{B_1, B_2, \dots, B_{\frac{n}{m}}\}$ the following holds:

$$A, B \text{ is yes instance of } HS \iff \exists A_i \text{ such that } \forall j \text{ the OV search on } (A_i, B_j) \text{ returns a null set.}$$

Proof of (3): We first prove the forward direction:
Consider a yes instance of HS, this implies let $v_1 \in A$ such that for all $v_2 \in B$, $\langle v_1, v_2 \rangle \neq 0$. Hence, $v_1 \in A_i$ for some i. Hence, for any $j \in \{1, 2, \dots, n/m\}$, if we perform an $\mathsf{OV}\mathit{Search}$ on (A_i, B_j), it is bound to return a null set by the hitting set property.

We now prove the reverse direction.
Let (A, B) be a no instance of HS. Then, for any $v_1 \in A$, there exists $v_2 \in B$ such that $\langle v_1, v_2 \rangle = 0$. Pick any A_i and any $v_1 \in A_i$. By the above statement, there exists a B_j such that for some $v_2 \in B_j$ the inner product of v_1 and v_2 is 0. Hence $\mathsf{OV}\mathit{Search}$ on the pair (A_i, B_j) will return (v_1, v_2) and hence is not a null set.

Time Complexity We now argue the time complexity of this reduction. Suppose that $\mathsf{OV}\mathit{Search}$ can be solved in time $t(n, d) = O(n^{2-\epsilon}\mathsf{poly}(d))$ for some $\epsilon > 0$

- We first estimate the time needed to process each A_i. The reduction performs an $\mathsf{OV}\mathit{Search}$ for A_i, B_j for every j in the worst case. However, since each round will eliminate one element from A_i, this cannot go for more than m steps. Hence, in the worst case, we make calls to $\mathsf{OV}\mathit{Search}$ for m times, resulting in $m \cdot t(m, d)$ time.

- Since, we iterate over all of elements in A_1 to maintain the set while processing different B_js, this will take an extra m time to maintain the is a vectors in A_i which has a non-zero dot product with all vectors of B.

- Hence, the overall time taken for all $A_1, A_2, \ldots, A_{n/m}$ is $(m \cdot t(m,d) + m) \cdot (n/m)$.

The above expression for runtime can be simplified as follows.

$$\begin{aligned} n \cdot m^{2-\epsilon} \cdot \text{poly}(d) + n &\leq n \cdot m^{2-\epsilon} \cdot \text{poly}(d) + n \cdot \text{poly}(d) \\ &= (n \cdot m^{2-\epsilon} + n) \cdot \text{poly}(d) \end{aligned}$$

Choosing $m = n^{\frac{1}{1+\left(\frac{1}{1-\epsilon}\right)}}$, we get the above runtime to be $O(n^{2-\epsilon}\mathsf{poly}(d))$.

3.2.4 OV reduces to Max Inner Product

Theorem 5. *$OV \leq$ MaxIP+*

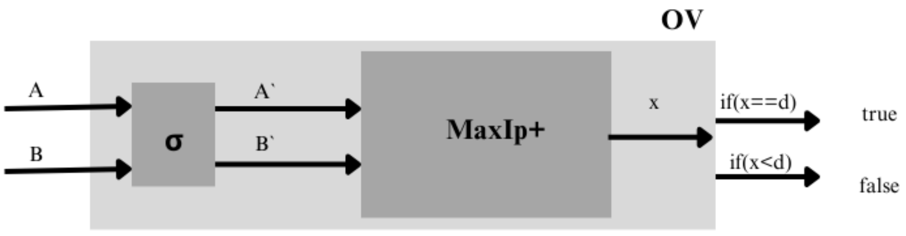

Fig. 3.5 MaxIp+ $\leq$ OV

Proof. Let A, B be collections of $\{0,1\}^d$ of size n each. We show the following:

1. There exits an algorithm σ that takes collections A, B and outputs A', B' in time $\mathcal{O}(8 \cdot \text{nd}) = \mathcal{O}(\text{nd})$.

2. For any A, B the following holds:

(A, B) is a yes instance of $OV \iff$ MAXIP+ instance of (A', B') (where $(A', B') = \sigma(A, B)$) returns d.

Proof of (1): Given A, B, Similarly, for all $b \in B$, $b' \in B'$ such that b' is given by $[1-b, 1-b, b]$ so the length of each vector in B' becomes 3d where the length of each vector in B is d. Which clearly takes $\mathcal{O}(nd)$ time clearly to construct.

Proof of (2): Consider A, B as yes instances of OV. This implies there is a pair of vectors, let $a \in A$ and $b \in B$, whose dot product is 0 $\langle a, b\rangle = 0$. Let $a' \in A'$ and $b' \in B'$ where $a' = a||\overline{a}||\overline{a}$ and $b' = \overline{b}||\overline{b}||b$.

In the following section, $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ and $\mathbf{y} = (y_1, y_2, \ldots, y_d)$

- The operation $(x \parallel y)$ indicates $(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n)$.
- The operation $\langle x, y\rangle$ indicates $\sum_{i=1}^{d} x_i y_i$.
- $\overline{x}$ indicates $(1 - x_1, 1 - x_2, \ldots, 1 - x_n)$

Here, we are modifying the a' and b' in the following manner :

1. $a' = a \,||\, \overline{a} \,||\, \overline{a}$
2. $b' = \overline{b} \,||\, \overline{b} \,||\, b$

Lemma 1. $\langle a', b'\rangle = d - \langle a, b\rangle$

Proof. The proof is as follows:

$$\boxed{\begin{aligned}
\langle a', b' \rangle &= \sum_{i=1}^{3d} (a'_i \cdot b'_i) \\
&= \sum_{i=1}^{d} a_i(1-b_i) + \sum_{i=d+1}^{2d} (1-a_{i-d})(1-b_{i-d}) + \sum_{i=2d+1}^{3d} (1-a_{i-2d})(b_{i-2d}) \\
&= \sum_{i=1}^{d} a_i(1-b_i) + \sum_{i=1}^{d} (1-a_i)(1-b_i) + \sum_{i=1}^{d} (1-a_i)b_i \\
&= \sum_{i=1}^{d} a_i(1-b_i) + (1-a_i)(1-b_i) + (1-a_i)b_i \\
&= \sum_{i=1}^{d} a_i - a_ib_i + 1 - a_i - b_i + a_ib_i + b_i - a_ib_i \\
&= d - \langle a, b \rangle
\end{aligned}}$$

from lemma1 $\langle a', b' \rangle = d - \langle a, b \rangle$ as we know that $\langle a, b \rangle = 0, so \quad \langle a', b' \rangle = d.$ Consider no instance of OV this means there is no pair of vectors from A, B whose dot product is 0,this implies $\forall a \in A, \forall b \in B, \langle a, b \rangle > 0$. from lemma1$\langle a', b' \rangle = d - \langle a, b \rangle$ as $\langle a, b \rangle > 0$ there is no possibility that $\langle a', b' \rangle$ can ever become d.

□

Theorem 6. *$OV \leq$ Count OV*

Proof. Let A, B be collections of $\{0,1\}^d$ of size n each. We show the following:

1. The algorithm σ takes in A, B and outputs A, B.

2. For any A, B the following holds:

$$A, B \text{ are } \textit{yes} \text{ instance of } OV \iff \textit{countOV} \text{ returns } x, \text{ where } x > 0.$$

3. The algorithm σ' takes the value x, outputs *yes* if x is greater than zero, else *no.*

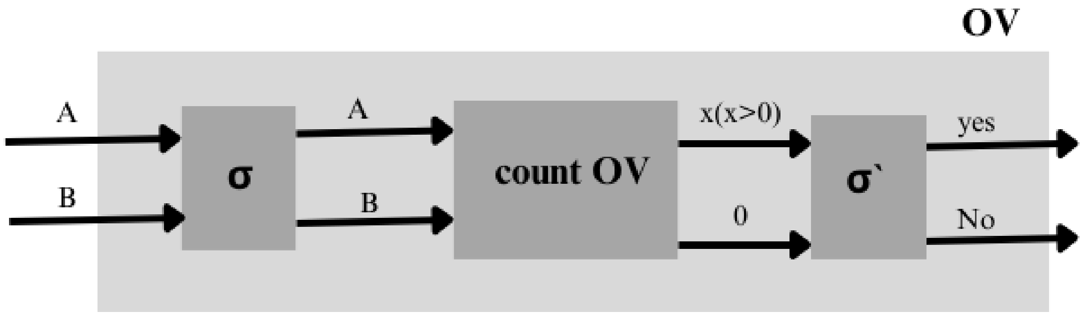

Fig. 3.6 MaxIp+ $\leq$ OV

Proof of (2) & (3): We will first argue the forward direction:

Consider A, B to be yes instances of OV; this implies there is an orthogonal pair of vectors. Let it be $v_1 \in A$, $v_2 \in B$. *CountOV* returns the number of orthogonal pairs of vectors in A, B. Since v_1, v_2 already exist, x should be at least 1.

We will now argue the reverse direction:

A, B is *no* instance of OV this means there is no orthogonal vectors in A, B so *CountOV* will return 0.

$\square$

In this chapter, we study fine-grained reductions between many problems that seem to be not solvable in linear time.

Chapter 4

Conclusion and Open Problems

In this report, we looked at the concept of fine-grained complexity and how it helps determine the time complexity of some problems. We saw various reductions between OV variants and similar and analysed its correctness and time complexity.
we did try to do reduction between $MaxIP+$ and OV i.e., $MaxIP+ \leq OV$ but we couldn't get it.
The problems NOV and AOV are defined but couldn't get any reduction between them.

References

[1] V. V. Williams, *On Some Fine-Grained Questions in Algorithms AND Complexity.* World Scientific, 2018, pp. 3447–3487. [Online]. Available: https://worldscientific.com/doi/abs/10.1142/9789813272880_0188

[2] A. Potechin and J. Shallit, "Lengths of words accepted by nondeterministic finite automata," *Information Processing Letters*, vol. 162, p. 105993, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020019020300806

[3] R. Williams, "A new algorithm for optimal 2-constraint satisfaction and its implications," *Theor. Comput. Sci.*, vol. 348, no. 2-3, pp. 357–365, 2005. [Online]. Available: https://doi.org/10.1016/j.tcs.2005.09.023

[4] A. Abboud, R. R. Williams, and H. Yu, "More applications of the polynomial method to algorithm design," in *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, 2015, pp. 218–230. [Online]. Available: https://doi.org/10.1137/1.9781611973730.17

[5] A. Gajentaan and M. H. Overmars, "On a class of $O(n^2)$ problems in computational geometry," *Computational geometry*, vol. 5, no. 3, pp. 165–185, 1995.

[6] L. Chen and R. Williams, "An equivalence class for orthogonal vectors," in *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, 2019, pp. 21–40. [Online]. Available: https://doi.org/10.1137/1.9781611975482.2

[7] A. Abboud, R. Williams, and H. Yu, “More applications of the polynomial method to algorithm design,” in *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms.* SIAM, 2014, pp. 218–230.

[8] V. V. Williams and R. R. Williams, “Subcubic equivalences between path, matrix, and triangle problems,” *J. ACM*, vol. 65, no. 5, pp. 27:1–27:38, 2018. [Online]. Available: https://doi.org/10.1145/3186893