UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Intractability And The Tractability Of The Orthogonal Vectors Problem

by

Karina M. Meneses-Cime

in the
Department of Mathematics

June 2018

Contents

1 Introduction **1**
- 1.1 Orthogonal Vectors Problem (OVP) 2
- 1.2 Edit Distance 3
- 1.3 Outline 4

2 From k-SAT to Orthogonal Vectors **6**

3 An In-Depth Look At Orthogonal Vectors **11**

4 From OVP To Edit Distance **17**

Bibliography **33**

Chapter 1

Introduction

The famous problem of P vs. NP attempts to understand the distinction between the problems whose solutions can be computed in polynomial time and those whose solutions can be verified in polynomial time. The theory behind P vs. NP looks at the problems in the class P, those that can be solved in polynomial time, as the easiest type of problems. Meanwhile, the problems in the class NP-Complete are considered to be hard problems. An NP-Complete problem, Π, is an NP problem (that is, one whose solutions can be verified in polynomial time) such that every other NP problem can be reduced to Π in polynomial time. The very first known NP-complete problem is the famous satisfiability problem which asks if, given a formula F in conjunctive disjunctive normal (CNF) form, F has a satisfying truth assignment. To show that an NP problem Π' is NP-Complete we only need to give a polynomial reduction from satisfiability to Π'.

The satisfiability problem has proven to be one of the central problems the theory of computation. If there does not exist a deterministic algorithm which runs in polynomial that solves the satisfiability problem then P $\neq$ NP. However, showing that such algorithm does or does not exist is one of Millennium problems. An exhaustive search algorithm for satisfibaility takes time approximately $O(2^n)$, where n is the number of variables. Despite efforts on improving the algorithm for satisfiability, the fastest known algorithms to-date at best shave off logarithmic factors. The fastest algorithm runs in time $O(2^{n(1-\frac{1}{O(\log mn)})})$ where m is the number of clauses and n the number of variables in the instance at hand ([1], [2]). The apparent intractability of this problem led to the introduction of the Strong Exponential Time Hypothesis (SETH). In their 1999 paper,"Which Problems Have Strongly Exponential Complexity?"[3] Paturi and Impagliazzo introduced the idea that under the assumption that the time needed to answer an instance of the satisfiability problem with k many literals in each clause of

the given CNF-formula and n variables approaches $O(2^n)$ as k grows larger, one can prove the intractability of many different kinds of problems.

Under SETH, we are not confined to working with only the classes P and NP. Many of the work in the field deals with much finer-grained reductions. Consider the following problem. Given a set S of n integers, does S contain three elements that sum to zero? This problem is known as 3SUM and has a simple running algorithm of time $O(n^2)$[4]. Despite efforts, the best algorithm up to date shaves off only logarithmic factors in the exponent and runs in time $O(\frac{n^2(\log\log n)^{O(1)}}{\log^2 n})$ [5].

This leads us to the suspicion that perhaps there is no better algorithm and to our next point. Under SETH we are able to show the intractability of problems which run in polynomial time. That is, given some problem with an algorithm that is known to run in time $O(n^k)$, we are able to show that there does not exist an algorithm of run-time $\Omega(n^{k-\epsilon})$ for $\epsilon > 0$ that solves the original problem unless SETH fails. In this paper we discuss two such reductions. We will show that under SETH there is no better algorithm than the naive algorithm for the Orthogonal Vectors problem as well as the Edit Distance problem. We then discuss the known algorithms for the Orthogonal Vectors Problems, and present a not well known algorithm for the OVP problem.

1.1 Orthogonal Vectors Problem (OVP)

An instance of the Orthogonal Vector Problem consists of sets A and B of d-dimensional binary vectors with $|A| = |B| := N$; the problem is to determine whether there exist vectors $a \in A$ and $b \in B$ such that $a \cdot b = 0$. The obvious algorithm for this problem is to check every possible pair of vectors and has a run-time of $\tilde{O}(N^2)$. Here we use the notation $\tilde{O}$ to mean big-O notation ignoring logarithmic factors.

We have already discussed the importance of the satisfiability problem. k-Satisfiability (k-SAT) is a particular version of a satisfiability instance where each clause in the given formula F contains exactly k literals. The obvious algorithm for this problem is to check every single truth assignment and runs in time $O(2^n)$ where n is the number of variables specified in the instance. k-SAT and SETH are our vehicles to showing that indeed OVP has no algorithm with run time $\tilde{O}(N^{2-\epsilon})$ for $\epsilon > 0$ unless SETH is false.

The reduction of k-SAT to OVP is particularly important as it is the first step in showing the intractability of further problems. For example, in a 2018 paper, Backurs et al. [6] have shown tight approximation bounds for Graph Diameter and Eccentricities under SETH. The first step in showing such bounds begins with a reduction from k-SAT to

OVP. Likewise, in this thesis, OVP is the first step of showing the hardness of the Edit Distance [7].

Besides serving as an intermediate step in important reductions, OVP can serve as a tool in rejecting SETH. That is, if we can find an algorithm that solves OVP in sub-quadratic time then SETH is false. Much work has gone into giving sub-quadratic algorithms for OVP. Through an application of the Sparsification Lemma [8], it is enough to consider the case when the dimensionality of the vectors, d, is $c \log N$ for constant $c > 0$. In 2015, Abboud, Williams and Yu ([7], [9]) showed, using the probabilistic method, that there is an algorithm for OVP that runs in time $n^{2-\frac{1}{O(\log c)}}$. The probabilistic method focuses on utilizing tools such as Markov's Inequality, the Chernoff bound and the Lovasz local lemma to show that an event takes place. In the use of the probabilistic method, one usually has a parameter p which determines the probability that such event happens. Thus, the probabilistic method involves a randomness. In order to use such techniques to argue for the run-time of deterministic algorithms, one must de-randomize these approaches. In [7], the algorithm given runs in the stated time with a high probability of obtaining a correct answer while in [9] this algorithm is derandomized. In this thesis we do not use the probabilistic method. Instead we give simple and intuitive algorithms. In comparison, in this thesis we discuss an algorithm that has been discussed only in an online forum [10] that runs in time $\tilde{O}(N^c)$ but now with $c < 2$ which does not use the probabilistic method. We do so in hope of understanding exactly what it is that makes OVP intractable.

1.2 Edit Distance

In the last section of the thesis, we discuss an example of the most useful property of OVP: its ability to transform into different problems. As mentioned before, OVP is known as an intermediate bridge between different types of problems, including Graph Diameter, Eccentricity, Batch Partial Match, and Vector Domination ([6], [7]). In this thesis, we are particularly concerned with the reduction of OVP into Edit Distance. An Edit Distance problem takes as input two strings x and y. It considers the actions of deleting, adding and replacing an entry in the string and assigns each a cost of 1. The edit distance $d(x, y)$ of strings x and y is the minimum cost required to transform x into y. The Edit Distance problem is to compute $d(x, y)$.

The last part of this thesis, starts by discussing the dynamic programming algorithm usually utilized to solve the edit distance problem. We'll see that this runs in time $O(N^2)$ where N is the size of the strings. We will then give the reduction from OVP to Edit Distance [11]. In conjunction with the reduction from k-SAT to OVP, this gives

a reduction from k-SAT to Edit Distance. This means that there is no sub-quadratic algorithm for Edit Distance unless SETH is false.

1.3 Outline

In chapter 2 we define k satisfiability. The obvious algorithm to solve an instance of k-SAT with n variables runs in time $O(2^n)$. Despite efforts this algorithm, up-to logarithmic factors in the exponent is also the fastest known. The hardness of k-satisfiability leads us to hypothesize that there does not exist a substantially faster algorithm for k-SAT. This is a famous hypothesis known as the Strong Exponential Time Hypothesis. In this chapter, we state SETH as well as explore its implications. In particular, we present the problem of orthogonal vectors. We see that the obvious algorithm for OVP runs in time $O(N^2)$ where N is the size of the vector sets in the fixed instance of OVP. Much like k-SAT, despite efforts, the fastest known algorithm, up to logarithmic factors, also runs in time $O(N^2)$. Because of this, we are motivated to show the intractability of OVP. In fact we show that if one believes in the intractability of k-SAT then one must also believe in the intractability of OVP. That is, we give a reduction from k-SAT to OVP and conclude the chapter with the result that for any $\epsilon > 0$ there does not exist an algorithm of run time $O(N^{2-\epsilon})$ unless SETH is false. We visit a widely used theorem that goes by the name of "the sparsification lemma" [8] that allows us to assume that any given instance of k-SAT can be reduced to a number of k-SAT instances such that the number of clauses m in each instance is $O(n)$. This in turn will mean that we may concentrate on OVP instances where the vector dimensional is $O(\log N)$.

If we can solve OVP in time $O(N^{2-\epsilon})$ for some $\epsilon > 0$ then we can solve k-SAT in time $O(2^{n-\delta})$ for some $\delta > 0$ and SETH is false. This would not prove that P = NP but it would give us better k-SAT algorithms. This motivates us to understand the hardness of OVP. In chapter 3 we continue exploring the OVP problem. In chapter 3, we revisit the usual $O(N^2)$ algorithm and explore techniques that make certain OVP instances easier to work with. By the sparsification lemma [8], we only consider instances where the dimensionality d of the vectors is $d = c \log N$ for some fixed constant $c > 0$. Following [9], we prove that when $c < 1$ there exists an algorithm that runs in time $O(N^{\max\{1,2c\}}) \leq O(N^2)$. We take this a step further and present the following result: Fix $c < 2$. For an instance of OVP where $d = c \log N$ there is an algorithm with running time $O(N^c)$ [10]. In contrast the best algorithm for OVP to date utilizes the probabilistic method and is given in [9]. This algorithm assumes $c \leq 2^{o(\frac{\log N}{\log \log N})}$ and runs in time $O(N^{2-\frac{1}{O(\log c)}})$.

The result for OVP is not unique. In fact, under SETH we are able to show hardness for many different problems. However, a particular property of OVP is its versatility. In chapter 4 we explore one example by considering the hardness of the Edit Distance problem. The Edit Distance problem is to convert one string s into another string t over some fixed alphabet in the fastest way possible. We convert this into a decision problem by computing the edit distance $d(x, y)$ of two given strings and deciding whether or not $d(x, y) < C$ for some fixed bound C. We take an in-depth look at the the usual dynamic algorithm used to compute the edit distance of two strings. Such algorithm runs in time $O(n^2)$ where n is the maximum of the length of x and y. Much like k-SAT and OVP, the fastest known algorithm runs in time $O(\frac{n^2}{\log^2 n})$[12]. That is, as fast as the standard textbook algorithm up to logarithmic factors. Just as with k-SAT and OVP, it is possible to show the intractability of Edit Distance under the hypothesis of SETH. We thus give an outline of the reduction from OVP to Edit distance as in the paper by A. Backurs and P. Indyk [11]. This reduction runs in time $O(n)$ and gives the following result: For every $\epsilon > 0$, there is no $O(n^{2-\epsilon})$ algorithm for the Edit distance decision problem unless SETH is false.

Chapter 2

From k-SAT to Orthogonal Vectors

An instance of the satisfiability problem takes as input a formula F in CNF form over a set of variables $x_1, \ldots, x_n$ and asks for a satisfying truth assignment. An instance of k-satisfiability (k-SAT) takes an instance of the satisfiability problem where each of the clauses $C_1, \ldots, C_m$ contains at most k literals. An instance of k-SAT is said to be "**satisfiable**" if it has a satisfying truth assignment. The trivial algorithm simply checks every truth assignment and runs in time equal to the number of possible truth assignments, $O(2^n)$. The apparent intractability of SAT leads to the following two hypothesis.

Definition 2.1 (ETH and SETH [3])**.** Let

$$s_k = \inf\{\delta : \text{there exists an algorithm for } k\text{-SAT with running time } O(2^{\delta n})\}.$$

It is clear that $s_k \leq 1$. The Exponential Time Hypothesis (ETH) states that $s_k > 0$ for $k \geq 3$. The Strong ETH (SETH) states that in addition $\lim\limits_{k\to\infty} s_k = 1$.

The versatility of k-SAT is exploited through SETH. When one encounters a problem which, after many failed attempts, proves itself to be particularly grim in respect to improving its run time, SETH is a tool that may help showcase the intractability of such problem.

An equivalent way to state SETH considers any k. In this form, SETH states the following: For all $\epsilon > 0$ there is a k such that k-SAT requires time $\Omega(2^{(1-\epsilon)n})$. The way to use SETH to show the intractability of some decision problem is to give a valid, fast reduction from k-SAT to the problem. By valid reduction we mean that given an

instance of k-SAT the reduction gives an instance of the problem at hand such that the instance of k-SAT is satisfiable if and only if the problem at hand has answer "Yes." By fast reduction, we mean that the transformation of the k-SAT instance into the problem at hand can be computed faster than the time it takes to compute a solution of the problem at hand. The fast requirement is imperatively necessary since it is possible to give a reduction between two problems in such a way where the reduction takes more time than to compute the answer. What remains is that in the end, we take more time to write down the problem than to solve it so that no matter what we do, the problem will always have run time at least the time we took to write it down.

In what follows, we use SETH to show the $\tilde{O}(N^2)$ hardness of OVP.

An instance of the Orthogonal Vector Problem consists of sets A, B of d-dimensional vectors over the alphabet $\{0, 1\}$ and where $|A| = |B| =: N$. The objective is to output answer to the question, "Do there exist vectors $a \in A$ and $b \in B$ such that $a \cdot b = 0$?" If the answer is "Yes" then we say that the instance of OVP, (A, B) "**has a solution**."

We will need to provide a valid, fast reduction from any instance of k-SAT to a particular instance of OVP. Valid here means that the instance of k-SAT has a satisfying truth assignment if and only if there is a pair of orthogonal vectors in our instance of OVP. Fast here means that the instance of OVP can be computed in time strictly less than $\tilde{O}(N^2)$. Note again, that if we give a reduction in time $\tilde{O}(N^{2+\eta})$ regardless of the time needed to compute the answer of the OVP instance at hand, the total computation time would be at least $\tilde{O}(N^{2+\eta})$. In this case, we know that the obvious algorithm for OVP runs in time $\tilde{O}(N^2)$ so that regardless of improving the algorithm for OVP, the total computation time is $\tilde{O}(N^{2+\eta})$. For our purposes we need to show that if there exists a fast algorithm for OVP then a reduction and a run of such algorithm is still fast.

We find that when dealing with k-SAT, the time of reduction necessary may be problematic. The reduction from a exponential time problem to a polynomial time problem creates a barrier when simply trying to write down the equivalent OVP instance. In our reduction below, the number m of clauses in our k-SAT instance will become the dimension, d, of our vectors. If the dimension of our vectors gets too large then writing them down could take possibly more time than actually computing the solution. Luckily, this is not the first reduction with this problem. To overcome this barrier, we make use of an important Theorem.

Theorem 2.2 (Sparsification Lemma [8]). *For every $k \in N$ and $\epsilon > 0$ there exists an algorithm sparse(F, ϵ) and a function $c(k, \epsilon)$ such that the following properties hold:*

1. *sparse*(F, ϵ) *takes as input a k-CNF formula F over n variables and gives a list of t many k-CNF formulas* $\{F_1, \ldots, F_t\}$ *such that F is satisfiable if and only if at least one of the* F_i*s is satisfiable.*

2. *Each* F_i *uses only the variables* $x_1, \ldots, x_n$*.*

3. *Each* F_i *has at most* $c(k, \epsilon)n$ *clauses.*

4. *sparse*(F, ϵ) *runs in time* $O(2^{\epsilon n})$ *and* $t < 2^{\epsilon n}$*.*

We will now give the reduction.

Theorem 2.3. *Fix* $\epsilon > 0$*. There is an algorithm SATtoOVP which does the following:*

1. *Given an instance* ϕ *of k-SAT as input, SATtoOVP*(ϕ, ϵ) *outputs* $t < 2^{\epsilon n}$ *many instances,* $(A_1, B_1) \cdots (A_t, B_t)$*, of OVP.*

2. ϕ *is good if and only* (A_i, B_i) *is good for some* $i \in [t]$*.*

3. *SATtoOVP*(ϕ, ϵ) *runs in time* $O(2^{(\frac{1}{2}+\epsilon)n} n)$*.*

Proof. Given an instance of k-SAT consisting of variables $x_1, \ldots, x_n$, clauses $C_1, \ldots, C_{m'}$ and CNF-formula F, apply the sparsification lemma to obtain formulas $F_1, \ldots, F_t$. By 1 in Theorem 2.2, we have that F is satisfiable if and only some F_i for $i \in [t]$ is satisfiable. Thus, solving the satisfiability questions for all the F_i is equivalent to solving the satisfiability question for our original F.

Fix $i \in [t]$. We reduce each F_i to an OVP question. By 2 in Theorem 2.2, the number of variables in each F_i is no more than n. Let α be a truth assignment on the first $\frac{n}{2}$ variables and define the vector v_α coordinate by coordinate by setting

$$(v_\alpha)_i = \begin{cases} 0 & \text{if } \alpha \text{ makes some literal in } C_i \text{ true} \\ 1 & \text{otherwise} \end{cases}$$

Let A be the set of all vectors v_α derived from every possible truth assignment α on the first $\frac{n}{2}$ variables. We assume for convenience that n is even. We have that $|A| = N = 2^{\frac{n}{2}}$.

Similarly, let β be a truth assignment on the last $\frac{n}{2}$ vectors and define the variables v_β by

$$(v_\beta)_i = \begin{cases} 0 & \text{if } \alpha \text{ makes some literal in } C_i \text{ true} \\ 1 & \text{otherwise} \end{cases}$$

Set B to be the set of all possible such vectors. We also have that $|B| = N = 2^{\frac{n}{2}}$.

Now we claim that the truth assignment γ defined by

$$\gamma(x_i) = \begin{cases} \alpha(x_i) & \text{if } 1 \le i \le \frac{n}{2} \\ \beta(x_i) & \text{if } \frac{n}{2} < i \le n \end{cases}$$

is a satisfying truth assignment for F_i if and only if $v_\alpha \cdot v_\beta = 0$.

Indeed, γ satisfies F_i if and only if γ satisfies each C_i for all $i \in [m]$. γ satisfies each C_i for all $i \in [m]$ if and only if either $(v_\beta)_i = 0$ or $(v_\alpha)_i = 0$ so that it is required for $(v_\beta)_i \cdot (v_\alpha)_i = 0$.

We have proven that every F_i can be reduced to an equivalent OVP problem. In what follows we show the sufficient runtime to transform each F_i into its particular OVP counterpart. For each formula F_i apply the following algorithm $SINGLEOVP$.

Define Function $SINGLEOVP$
//Function $SINGLEOVP$ takes in an instance ϕ of k-SAT
//and outputs an equivalent instance of OVP
Intialize empty sets of vectors A and B
//In the reduction given above, the size of the vector sets
//A and B are given by the possible number of truth assignments
//on half of the variables for formula F_i.
//That is, $|A| = |B| = N \le 2^{\frac{n}{2}}$.
for each truth assignment α to $x_1, \ldots, x_{\frac{n}{2}}$ **do**
  for each clause $C_i \in F$ **do**
    Set $(v_\alpha)_i = \begin{cases} 0 & \text{if } \alpha \text{ makes } C_i \text{ true} \\ 1 & \text{otherwise} \end{cases}$
    Put v_α in A
  end for
end for
for each truth assignment α to $x_{\frac{n}{2}+1}, \ldots, x_n$ **do**
  for each clause $C_i \in F$ **do**
    Set $(v_\beta)_i = \begin{cases} 0 & \text{if } \alpha \text{ makes } C_i \text{ true} \\ 1 & \text{otherwise} \end{cases}$
    Put v_β in B
  end for
end for
Output A and B
End Function $SINGLEOVP$

The run time for the conversion of each F_i can be bounded as follows. There are $2^{\frac{n}{2}}$ possible truth assignments on each half of the variables and by 3 of Theorem 2.2 $m = O(n)$ many clauses in our given instance of SAT. Furthermore, the time it takes to check if α satisfies a certain clause is $k = O(1)$. Thus, this algorithm takes time $O(2^{\frac{n}{2}}mk) = O(2^{\frac{n}{2}}n)$.

Now we define $SATtoOVP(\phi, \epsilon)$.

Begin function $SATtoOVP$
//Function $SATtoOVP$ takes in an instance of $k-$SAT with $x_1, \ldots, x_n$,
//$C_1, \ldots, C_{m'}$ clauses and CNF-formula F.
//$\epsilon > 0$ is fixed.
Compute sparse(F, ϵ)
// sparse gives us t formulas with the properties given in 2.2.
for all formulas F_i given by sparse(F, ϵ) **do**
  Output the OVP instance given by $SINGLEOVP(F_i)$
end for

Properties 1 and 2 of Theorem 2.2 follow from our discussions above. For property 3 we examine the overall runtime of $SATtoOVP$. By property 4 of Theorem 2.2, sparse(F, ϵ) runs in time $O(2^{\epsilon n})$. Computing $SINGLEOVP(F_i)$ for all $i < 2^{\epsilon n}$ runs in time $O(2^{\frac{1}{2}n}n2^{\epsilon n}) = O(2^{(\frac{1}{2}+\epsilon)n}n)$ so in total we get a running time of $O(2^{\epsilon n} + 2^{(\frac{1}{2}+\epsilon)n}n) = O(2^{(\frac{1}{2}+\epsilon)n}n)$.

□

From here, it is easy to argue the following.

Theorem 2.4. *There does not exist an algorithm for OVP with time $O(N^{2-\delta})$ for constant $\delta > 0$ unless SETH fails.*

Proof. For the sake of contradiction, assume that there exists an $O(N^{2-\delta})$ time algorithm for OVP for fixed $\delta > 0$. Fix $k \geq 3$. We will prove $s_k < 1 - \frac{\delta}{4}$. Set $\epsilon = \frac{\delta}{4}$.

Apply the algorithm of OVP with run-time $O(N^{2-\delta})$ for fixed $\delta > 0$ to each of the $O(2^{\epsilon n})$ instances of OVP. Each of these OVP instances can be solved with runtime $O((2^{\frac{n}{2}})^{2-\delta}) = O(2^{n-\frac{\delta n}{2}}) = O(2^{n(1-\delta)})$. As there are at most $2^{\epsilon n}$ of these instances, the total run time to solve all of the OVP instances is $O(2^{\epsilon n}2^{n-\frac{\delta}{2}n}) = O(2^{(1-\frac{\delta}{2}+\epsilon)n})$.

Thus, solving ϕ runs in time $O(2^{n(\epsilon+\frac{1}{2})}n) + O(2^{(1-\frac{\delta}{2}+\epsilon)n}) = O(2^{n(\frac{\delta}{4}+\frac{1}{2})}n + 2^{(1-\frac{\delta}{4})n}) = O(2^{(1-\frac{\delta}{4})n})$. In other words, for any $k \geq 3$, $s_k < 1 - \frac{\delta}{4}$ which contradicts SETH.

□

Chapter 3

An In-Depth Look At Orthogonal Vectors

An instance of the Orthogonal Vector Problem consists of sets A, B of d-dimensional vectors over the alphabet $\{0,1\}$ and where $|A| = |B| =: N$. The objective is to output an answer to the question, "Do there exist vectors $a \in A$ and $b \in B$ such that $a \cdot b = 0$?" We have seen in 2, that if we can give an algorithm for OVP that runs in time $O(N^{2-\epsilon})$ for some $\epsilon > 0$ then we will have refuted SETH. In fact, in Theorem 2.3, we reduced an instance of k-SAT to OVP. In the process, we applied the sparsification lemma, Theorem 2.2, and obtained $\phi_1, \ldots, \phi_t$ instances of k-SAT with the property that each had at most cn clauses for some fixed constant $c > 0$. In our conversion of each such instance ϕ_i to an OVP instance ϕ_i', the dimensionality of the vectors in A and B is given by the total number of clauses in ϕ_i. That is, ϕ_i' enjoys the property that $d = cn$. Recall that ϕ_i' also has the property that $|A| = |B| = N = 2^{\frac{n}{2}}$ so that $n = 2\log N$. Thus, ϕ_i' has vectors with dimensionality $d = cn = c2\log N = O(\log N)$. For this reason, in what follows we focus on looking at different approaches to solve instances of OVP with dimensionality $O(\log N)$ in time faster than $O(N^2)$.

The obvious algorithm for OVP is to check every possible combination of pairings of vectors. As there are N^2 many possible pairs this algorithm runs in time $\tilde{O}(N^2)$. The fastest algorithm for OVP is given in [7] and uses the probabilistic method to obtain a runtime of $O(n^2 \log N)$. In a 2016 paper, R. Williams and T. Chan [9] showed that under the assumption that $c \leq 2^{o(\frac{\log N}{\log\log N})}$, the number of distinct $u, v \in \{0,1\}^{c\log N}$ such that $u \cdot v = 0$ can be counted in $N^{2-\frac{1}{O(\log c)}}$ time deterministically. What follows is inspired by such result. Take note of the restriction that $c \leq 2^{o(\frac{\log N}{\log\log N})}$. In our following algorithms we take advantage of the implications of restricting the dimensionality of our vectors to $O(\log N)$. Indeed, note that since $d = c\log N$ then $2^d = N^c$. Thus, there

are only $2^d = N^c$ possible d-dimensional vectors where $c < 1$ while there are $N > N^c$ vectors in A (or B respectively). This means that we must have copies of vectors in sets A and B.

In what follows we assume that we have eliminated duplicates from our given OVP instances. There exist various implementations of algorithms in C++ that eliminate repetitions in vectors and run in time $\tilde{O}(n)$. One such implementation is to sort the vector and then remove repetitions after scanning through the vector only once. Given a vector V, sorting V takes time $O(\log(|V|))$ while scanning through it to check for repetitions takes time $O(|V|)$ giving us a total running time of $O(|V|\log(|V|) + |V|) = \tilde{O}(|V|)$.

Since the running time of our algorithms will be greater than $\tilde{O}(|A|) = \tilde{O}(|B|)$, removing repetitions does affect the running time of the overall algorithms.

In [9], R. Williams and T. Chan state that OVP, with the restriction of $d = c\log N$, can be solved in time $\min\{O(cN^2\log N), O(N^{c+1})\}$. This means that when $c > 1$, we seek to solve OVP in time $O(cN^2\log N)$ and when $c < 1$ we may solve it in time $O(N^{c+1})$. We claim that in fact, when $c < 1$, OVP can be solved in time $\tilde{O}(N^{\max\{1,2c\}})$. In what follows, we give an explicit algorithm for this run time.

Theorem 3.1. *(An improvement of [9] from [10]) For an instance of OVP where* $d = c\log N$*, and* $c < 1$ *there is an algorithm with running time* $\tilde{O}(N^{\max\{1,2c\}})$.

Proof. For each vector for each $b \in B$, consider the set $\phi_b := \{v \in \Omega : v \cdot b = 0\}$. Each set takes time $\tilde{O}(2^d)$ to compute. Now apply the following algorithm.

Remove duplicates from A and store them in data structure Π_A
Remove duplicates from B and store them in data structure Π_B
for all $b \in \Pi_B$ **do**
  Compute ϕ_b
  for all $v \in \phi_b$ **do**
    if $v \in \Pi_A$ **then**
      Output "Yes" and halt
    end if
  end for
end for
Output "No"

Building Π_A and Π_B takes time $\tilde{O}(N)$. There are at most 2^d vectors in Π_B and at most 2^d vectors in ϕ_b for every b. Searching for $v \in \Pi_A$ can be done in time $\tilde{O}(N)$

inside an efficient data structure. Thus, this gives an algorithm with running time $\tilde{O}(2^d 2^d + N) = \tilde{O}(2^{2d} + N) = \tilde{O}(N^{2c} + N) = \tilde{O}(\max\{N^{2c}, N\})$. □

We now take this a step further. Previously, we took advantage of the fact that the sets A and B contain repetitions. We now take advantage of the structure given by the all possible orthogonal vectors to a set. Doing this leads us to the following result.

Theorem 3.2. *For an instance of OVP where $d = c \log N$, $c < 2$, there is an algorithm with running time $O(N^c)$.*

Notice that this is an improvement over the algorithms given in 3.1 and [9] as we expand our constant c selection to $c < 2$. Furthermore, the fastest algorithm thus far for OVP assumes that $c \leq 2^{o(\frac{\log N}{\log \log N})}$ and runs in time $N^{2 - \frac{1}{O(\log c)}}$ deterministically. Such algorithm is obtained using the probabilistic method and requires more sophisticated techniques than the ones presented in this paper. Instead, we give a simple deterministic algorithm for OVP that does not rely on probability but that competes with the result of algorithm in [9].

Throughout this chapter, we consider the following sets of 4-dimensional vectors, $A = \{0111, 0101, 1101\}$ and $B = \{0001, 0010, 1001\}$ as examples for our definitions.

Notation: In what follows, the ith coordinate in a vector w is denoted w_i.

Definition 3.3 (Orthogonalization of a vector)**.** For a vector $b = b_1...b_d$, the orthogonalization of b, $o(b)$ is defined as the vector with entries over the set $\{0, \times\}$ with entries defined as follow: $o(b)_i = \begin{cases} 0 & \text{if } b_i = 1 \\ \times & \text{otherwise} \end{cases}$

For example, for $0001 \in B, o(0001) = \times\times\times 0$. This vector represents all the vectors that can be obtained by replacing $\times$ with either a 0 or a 1. These vectors are also exactly all of the vectors orthogonal to 0001. Consider the orthogonalization of the set B, $o(B) = \{o(b) : b \in B\}$. If $a \in A$ is a vector such that $a_i = 0$ if $o(b)_i = 0$ for all i, then a is orthogonal to b. To search for such vector $a \in A$ easier, we build the following set.

Definition 3.4 (Orthogonal Extension of a set)**.** Let A be a set of vectors from $\{0, 1\}^d$. The orthogonal extension of a set A is

$$A_{OE} = \{a' \in \{0, \times\}^d : \text{there exists } a \in A, a'_i = 0 \Rightarrow a_i = 0\}$$

The orthogonal extension of a set A contains every possible way to represent a vector in A. That is, if $w \in A_{OE}$ then there is a way to replace all $\times$s, some with 0's and some

with 1's, to obtain a vector in A. In our example, the orthogonal extension of A is the following:

$$A_{OE} = \{0\times\times\times, 0\times0\times, \times\times0\times, \times\times\times\times\}$$

Take $w = \times\times0\times$. Replace $w_0 = 0, w_1 = 1, w_3 = 1$ and we obtain the string $0101 \in A$.

Theorem 3.5. *$v \in \{0,1\}^d$ is orthogonal to some $a \in A$ if and only if $o(v) \in A_{OE}$.*

Proof. ($\Rightarrow$)If $o(v)_i = 0$ then $v_i = 1$ which forces $a_i = 0$ since $v_i a_i = 0$ for all $i \in [d]$. Thus $o(v) \in A_{OE}$.

($\Leftarrow$)If $o(v) \in A_{OE}$ then there exists some $a \in A$ such that if $o(v)_i = 0$ then $a_i = 0$ for all $i \in [d]$. If $v_i = 0$ then $a_i v_i = 0$. On the other hand, if $v_i = 1$ then $o(v)_i = 0$ which implies $a_i = 0$ and so $v_i a_i = 0$. □

We thus have that $o(b) \in A_{OE}$ for some $b \in B$ if and only if there is a pair of orthogonal vectors in our OVP instance.

We now discuss the time complexity necessary to build A_{OE}. First, we define some useful tools.

Definition 3.6 (Abstraction)**.** Let $w \in \{0,1\}^d$. The abstraction of w, $\alpha(w)$ is defined coordinate by coordinate by $\alpha(w)_i = \begin{cases} 0 & \text{if } w_i = 0 \\ \times & \text{if } w_i = 1 \end{cases}$

In general, the abstraction of a set of vectors $A \subset \{0,1\}^d$, is $\alpha(A) = \{\alpha(a) : a \in A\}$.

Thus, for our example where $A = \{0111, 0101, 1101\}$,

$$\alpha(A) = \{0\times\times\times, 0\times0\times, \times\times0\times\}$$

Definition 3.7 (0-replacement)**.** Let $w \in \{0,\times\}^d$. The 0-replacement set of w, $\otimes(w)$, is defined by $\otimes(w) = \{w' \in \{0,\times\} : w' \text{ is obtained by replacing exactly one 0 by a } \times \text{ in } w\}$. For a set $X \subset \{0,\times\}^d$, the 0-replacement set is defined as $\otimes(X) = \bigcup_{x \in X} \otimes(x)$.

The recursive 0-replacement process defined on a set A puts $X_1 = \alpha(A)$ and $X_{i+1} = \otimes(X_i)$.

For $\alpha(A) := X_1$ we have

$$\otimes(X_1) = \{\times\times\times\times, \times\times 0\times, 0\times\times\times\} := X_2 \text{ and}$$

$$\otimes(X_2) = \{\times\times\times\times\} := X_3$$

It is easy to see that for the 0-replacement process of any set A, there exists a $j \in \mathbb{Z}^+$ such that $\{\times\times\times\times\} = X_j$. Thus, the sets $X_1, ..., X_j$ can be computed in a finite amount of time. Furthermore, the following results will allow us to specify an algorithm for to compute A_{OE}.

Theorem 3.8. *Let A be a set of d-dimensional vectors and let $X_1, ..., X_d$ be the sets given by the 0-replacement process then $\bigcup_{z\in[d]} X_z = A_{OE}$.*

Proof. If $v \in A_{OE}$ then there is some $a \in A$ such that if $v_i = 0$ then $a_i = 0$. If $\alpha(a) \neq v$, because $\alpha(a) \in X_1$ maintains this property, then $\alpha(a)$ differs in finitely many places from v by means of having a 0 instead of an $\times$. Thus, taking the 0-replacement of $\alpha(a)$, that is, replacing each 0 with an $\times$ one at a time, yields v in a set X_z for $z \in [d]$ so that $v \in \bigcup_{z\in[d]} X_z$.

Take $v \in \bigcup_{z\in[d]} X_z$. If $v \in \alpha(A)$ then $v \in X_1 \subset \bigcup_{z\in[d]} X_z$, otherwise v is obtained for some $a \in A$ by replacing 1s by $\times$s and some 0s to $\times$. Thus, if $v_i = 0$ then $a_i = 0$. □

To compute A_{OE}, we do the following:

Initialize X_0 as an empty array of maximum size 2^d
Initialize $X_1 = \alpha(A)$
while $X_1 \neq \emptyset$ **do**
  Pick $w \in X_1$
  Remove w from X_1
  Add w to X_0
  for $v \in \otimes(w)$ **do**
    if $v \notin X_0 \cup X_1$ **then**
      Add v to X_1
    end if
  end for
end while

Theorem 3.9. *For a set $A \subset \{0,1\}^d$, computing A_{OE} takes time $\tilde{O}(d2^d + N)$.*

Proof. We refer to the algorithm given above and discuss its run-time. First, initializing X_0 takes time $\tilde{O}(1)$.

Now, in order to compute $\alpha(A)$, we simply run the following algorithm.

Initialize $\alpha(A)$ to be an empty set of maximum size N
for $a \in A$ **do**
  Define a d-dimensional vector v_i by
  $$v_i = \begin{cases} x & \text{if } a_i = 0 \\ 0 & \text{if } a_i = 1 \end{cases}$$
  Add v to $\alpha(A)$
end for

We see that initializing $\alpha(A)$ to be an empty array runs in time $O(1)$. We then have that running through every $a \in A$ and scanning every coordinate in a takes time $O(dN)$. Thus, computing $\alpha(A)$ takes time $O(N + dN) = O(N + c\log(N)N) = \tilde{O}(N)$.

Note that the line if $v \notin X_0 \cup X_1$ guarantees that every $v \in \{0, \times\}$ appears in X_0 at most once. Since there are at 2^d d-dimensional vectors over $\{0, \times\}$ this guarantees that the loop executes at most 2^d times. Now for a $d-$dimensional vector w, $| \otimes (w)| \leq d$ so that the inner for loop runs in time $\tilde{O}(d)$. Thus, this the construction of A_{OE} runs in time $\tilde{O}(d2^d)$. The result now follows.

□

From this, we are now ready to give a faster algorithm for OVP in a special instance.

Theorem 3.10. *For an instance of OVP where* $d = c\log N$, $c < 2$, *there is an algorithm with running time* $O(N^c)$.

Proof. Build $o(B)$
Build A_{OE}
for $b \in o(B)$ **do**
  if $b \in A_{OE}$ **then**
    Output "Yes"
  end if
end for
Output "No"

Building $o(B)$ runs in time $\tilde{O}(N)$. By the previous theorem, building A_{OE} runs in time $\tilde{O}(d2^d + N)$ and checking for membership in A_{OE} runs time $O(\log N)$ with an efficient data structure. Thus, the algorithm runs in time $\tilde{O}(N + d2^d) = \tilde{O}(N^c)$.

We return to the example sets given at the beginning of the chapter. From above, $o(B) = \{\times\times\times 0, \times\times 0\times, 0\times\times 0\}$ and $A_{OE} = \{0\times\times\times, 0\times 0\times, \times\times 0\times, \times\times\times\times\}$. Since $\times\times 0\times \in A_{OE}$ then the answer is yes for this instance of OVP. □

Chapter 4

From OVP To Edit Distance

In the previous chapters we explored the intractability of OVP. However, OVP is most famous for its versatility as a bridge to show intractability of other problems. k-SAT is an exponentially hard problem while OVP is a quadratic hard problem. That is, while there remains to find a sub-exponential algorithm for k-SAT, there remains to find a sub-quadratic algorithm for OVP. When discussing P vs NP we are interested in only exponentially hard problems. We are looking to find polynomial run-time algorithms for problems with easy exponential time solutions. However, notice that OVP's easy solution runs in time $O(dN^2)$ and that furthermore, we were able to show the hardness of OVP despite it being a problem with a polynomial run-time. This is precisely what makes OVP versatile.

In what follows we convey this property of OVP by introducing a new problem, Edit Distance. The Edit Distance problem takes two strings x and y over some alphabet Σ and asks for the minimum number of operations required to transform some string x into another string y. To talk about measuring such distance we define the function $d_e(x, y) : \Sigma \times \Sigma \to \mathbb{R}_{\geq 0}$ to be the minimum number of symbol insertions, deletions and substitutions required to transform x into y.

Theorem 4.1. *d_e forms a metric.*

Proof. We have that $x \in \Sigma$ requires no actions to transform into itself and thus $d_e(x, x) = 0$. Furthermore, if $d_e(x, y) = 0$ then x is identical to y and thus $x = y$.

Every substitution of a letter, substitute τ with τ', can be reversed by substituting τ' with τ. Every deletion of a letter τ can be undone by the insertion of the letter τ and conversely every insertion of a letter τ can be undone by the deletion of a letter τ. Thus, it follows that if n actions are minimally required to change x into y then at most n

actions are required to change y into x. Further, it must be the case that also n actions are required to change y into x, for if not let Φ be actions needed to change y into x, then the reverse of Φ yields a shorter conversion of x into y violating the minimality of n. We thus have that $d_e(x, y) = d_e(y, x)$.

It remains to show that $d_e(x, z) \leq d_e(x, y) + d_e(y, z)$ for all strings y. Although $d_e(x, z)$ is fixed, there may be many minimal ways to transform x into z. Let $\Phi_1, \ldots, \Phi_k$ be all the possible ways to transform x into z. For each Φ_j, applying Φ_j to x yields a list of strings $x_1^j, \ldots, x_n^j$ such that $x_1^j = x$ and $x_n^j = y$. A string z is optimal if $x_i^j = z$ for some $i \in [n]$ and $j \in [k]$. If z is optimal, it follows that $d_e(x, y) + d_e(y, z) = d_e(x, z)$. If z is not optimal then the actions of converting x to y and then y to z is a way to convert x into z that is not minimal. Thus, $d_e(x, z) \leq d_e(x, y) + d_e(y, z)$. □

The naive approach to solving an instance of Edit Distance utilizes a dynamic programming algorithm [13]. Let $n = \max(|x|, |y|)$. The dynamic programming algorithm approach uses a $|x| \times |y|$ table to keep track of possible efficient routes to convert the string x into string y and runs in time $O(n^2)$. Suppose string x consists of symbols $x_1 \ldots x_n$ and string y of symbols $y_1 \ldots y_n$, the mentioned table constructs $d_e(x_i, y_j)$ for all $i \in [n]$ and $j \in [m]$ and then looks for the shortest combination of such i's and j's such the sum is $d_e(x, y)$.

Suppose we are given strings x =BCAAD and y =CAAB. Our task is to find the minimum number of deletions, substitutions and insertions necessary to convert x into y. We first note that the shortest way to do this may involve deleting every character in the initial source string x. Thus, instead of considering just x we consider the string ␣ x to make the point that when the whole of x is deleted, ␣ remains. We also do this with string y. The algorithm to compute $d_e(x, y)$ begins by initializing an empty $n + 1$ by $m + 1$ matrix and associating the source string ␣ x with the horizontal axis and the target string ␣ y with the vertical axis as shown below.

	␣	B	C	A	A	D
␣						
C						
A						
A						
D						

Entry (i, j) in the table will correspond to the number of least computations necessary to convert the substring $x_1 \ldots x_i$ into the substring $y_1 \ldots y_j$. We start filling out the easiest entries in such table, $(i, 1)$ for $i = 1, 2, 3, 4, 5$ and $(1, j)$ for $j = 1, 2, 3, 4, 5, 6$. It is clear $(i, 1) = i - 1$ for all $i = 1, 2, 3, 4, 5$ as the shortest way of converting an empty

string into a non empty string consists of inserting the non-empty string characters into the empty string. Likewise, $(1, j) = j - 1$ for $j = 1, 2, 3, 4, 5, 6$ since the shortest way of converting a non-empty string into an empty string is to simply delete all characters. We thus have the following table.

	␣	B	C	A	A	D
␣	0	1	2	3	4	5
C	1					
A	2					
A	3					
D	4					

Now we fill out the rest of the table inductively by columns. As mentioned above, moving down on the table indicates an insertion of a character and moving right indicates a deletion. By process of elimination, moving diagonally must indicate substitution. Indeed, consider entry (2,2) on the above table. The shortest way to convert the string x_1 =B into y_1 =C is to replace B with a C. This induces a cost of +1 in our table so entry (2,2) = (1,1) + 1 = 0+1 = 1.

What does it mean if we would have picked to add one to the entry right above (2,2) (That is (1,2))? This would first force us to find the cost to transform the string B into ␣ as per indicated by that entry. Then by moving down we are inserting the necessary letter C. What this looks like is B → ␣ → C, two moves. Similarly, if we were to take the entry to the left of (2,2), our first move would indicate a deletion. That is, B → ␣ → C, again two moves.

From this we learn that moving down represents insertion, moving left represents deletion and moving diagonally represents substitution. At each entry we look for the smallest cost possible and thus for the minimum of these three entries. If we choose to operate with the diagonal at entry (i, j) and if $x_i = y_j$ then we simply leave that characters alone. That is, there is no extra cost for that move. We then have that we set entry $(i, j) = \min((i-1, j), (i, j-1), (i-1, j-1) + \text{cost}(i, j))$ where $\text{cost}(i, j) = 0$ if $x_i = y_j$ and 1 otherwise. From this, we recursively fill our table.

	␣	B	C	A	A	D
␣	0	1	2	3	4	5
C	1	1	1	2	3	4
A	2	2	2	1	2	3
A	3	3	3	2	1	2
D	4	4	4	3	2	1

$d_e(x, y) = (n, m)$. Indeed, the fastest way to turn x to y is to delete B from x which has a 1 operation. A more interesting example is below.

	␣	Z	A	P	A	T	O
␣	0	1	2	3	4	5	6
P	1	1	2	2	3	4	5
A	2	2	1	2	2	3	4
P	3	3	2	1	2	3	4
O	4	4	3	2	2	3	3

In short, we have introduced the following algorithm to compute $d_e(x, y)$.

First define the function cost:

Define function cost(i,j)
if $x_i = y_j$ **then**
  Return 0
else
  Return 1
end if
End function

Now define the algorithm to compute $d_e(x, y)$:

Initialize an empty $|x| \times |y|$ 2D matrix
for i in range($|x|$) **do**
  Set $(i, 1) = i - 1$
end for
for j in range($|j|$) **do**
  Set $(1, j) = j - 1$
end for
for $i = 2$ in range($|x|$) **do**
  for $j = 2$ in range ($|y|$) **do**
    Set $(i, j) = \min((i, j - 1) + 1, (i - 1, j) + 1, (i, j)$+cost$(i, j))$
  end for
end for

This runs in time $O(n^2)$. Despite efforts, the fastest algorithm to compute Edit Distance runs in time $O(\frac{n^2}{\log^2 n})$ [12]. Ignoring logarithmic factors this is as fast as the usual textbook algorithm. This gives us a motivation to show the intractability of OVP. To do this, we use SETH and give a reduction from k-SAT to edit distance. As it was mentioned earlier, such reduction will not be explicitly between k-SAT and edit distance. Instead, we utilize the result given by Theorem 2.3 and use OVP as an intermediate

bridge between k-SAT and edit distance. What we will give is an explicit reduction from OVP to Edit distance. Indeed, such reduction exists and is given in [11]. In this reduction we implicitly convert the problem of finding $d_e(x, y)$ to the decision problem of "Is $d_e(x, y) = C$?," for some explicit, fixed C.

Theorem 4.2. *[11] There exists a reduction from OVP to Edit distance which runs in time $O(N)$.*

Proof. We give such reduction. Without loss of generality we assume that the first entry in every vector in B is 1.

We work from the inside out. We first give vector gadgets for each coordinate in each vector. Then we will concatenate them to make up vector gadgets for each vector in sets A and B. Lastly, we concatenate these vectors to get an instance of two strings, each representing either set. In the end we construct two strings s_A and s_B representative of each corresponding set A and B where $d_e(s_A, s_B) = C$ if there are no two orthogonal vectors in A and B and $d_e(s_A, s_B) \leq C - 2$ otherwise.

First, we will talk about the vector gadgets of each coordinate. Throughout the reduction, we will define parameters of length l_c, l_v, and l_s. The reason behind choosing these parameters will become apparent as we progress into discussing the validity of the reduction.

Coordinate Gadgets:

Define

$$\mathrm{CG}_1(i) = \begin{cases} 2^{l_c}01112^{l_c} & \text{if } i = 0 \\ 2^{l_c}00012^{l_c} & \text{if } i = 1 \end{cases}$$

$$\mathrm{CG}_2(i) = \begin{cases} 2^{l_c}00112^{l_c} & \text{if } i = 0 \\ 2^{l_c}11112^{l_c} & \text{if } i = 1 \end{cases}$$

for $i \in \{0, 1\}$ and $l_c = 1000 \cdot d$.

These are our coordinate vector gadgets. At every step of complexity, we must address how our construction impacts the edit distance of the objects at hand. Here, note that the edit distance between the two coordinate vector gadgets is 1 when the product of the inputs of each coordinate gadget is zero and 3 when the product of the inputs is nonzero. That is, for $i, j \in \{0, 1\}$,

$$d_e(CG_1(i), CG_2(j)) = \begin{cases} 1 & \text{if } i \cdot j = 0 \\ 3 & \text{if } i \cdot j = 1 \end{cases}$$

Now we define the vector gadgets. In what follows, we denote the concatenation of two strings x and y as $x \cup y$. First, let $l_v = (1000d)^2$, and define

$$\textsc{Pad}_L = \textsc{Pad}_R = \textsc{Pad} = 3^{l_v}$$

$$\textsc{Edge}_L = \textsc{Edge}_R = 4^{l_v}$$

$$\textsc{Conc}_a = \bigcup_{i \in [d]} \text{CG}_1(a_i)$$

$$\textsc{Conc}_b = \bigcup_{i \in [d]} \text{CG}_2(b_i)$$

Vector Gadgets:

Now define the vector gadgets as follows.

$$VG_1(a) = \textsc{Edge}_L \textsc{Conc}_a \textsc{Pad} \textsc{Conc}_{a'} \textsc{Edge}_R \text{ where } a' = 10^{d-1}$$

$$V_2(b) = \textsc{Pad}_L \textsc{Conc}_b \textsc{Pad}_R$$

We thus have the following strings for VG_1 and VG_2 respectively.

$EDGE_L = 4^{l_v}$ | $CONC_a = \bigcup_{i\in[d]} CG_1(a_i)$ | $PAD = 3^{l_v}$ | $CONC_{a'} = \bigcup_{i\in[d]} CG_1(a'_i)$ | $EDGE_R = 4^{l_v}$

444 ... 444 | $2^{l_v}0$ xx 12^{l_v} ... $2^{l_v}0$ xx 12^{l_v} | 333 ... 333 | $2^{l_v}0$ xx 12^{l_v} ... $2^{l_v}0$ xx 12^{l_v} | 444 ... 444

333 ... 333 | $2^{l_v}0$ xx 12^{l_v} ... $2^{l_v}0$ xx 12^{l_v} | 333 ... 333

$PAD_L = 3^{l_v}$ | $CONC_b = \bigcup_{i\in[d]} CG_2(b_i)$ | $PAD_R = 3^{l_v}$

We must now talk about $d_e(VG_1(a), VG_2(b))$. This part of the proof is what makes the construction of this reduction work.

Claim: Let $|\textsc{Conc}_a| = |\textsc{Conc}_b| = |\textsc{Conc}_{a'}| = \mu$.

$$d_e(VG_1(a), VG_2(b)) = \begin{cases} V_\text{o} := 2l_v + \mu + d & \text{if } a \cdot b = 0 \\ V_\text{no} := 2l_v + \mu + d + 2 & \text{if } a \cdot b \geq 1 \end{cases}$$

Before proving the claim, we give intuition of a fundamental notion. We wish to make the notion of "aligning" strings precise. Intuitively, an alignment between two strings occurs when the necessary actions to transform string x into a string y only include preserving or substituting symbols. However, one may go in a round-about way of doing this. For example, one may delete symbols only to insert them again with the same or a different symbol and only to find out that the operation could have been done in a smaller amount of steps by substituting or preserving the symbol. However, the transformation of string x into string y, takes a particular set of instructions that once applied to x yield y. These instructions can always be applied onto string x by traversing the string from left to right and applying the necessary actions on the scanned symbol at hand. If applied in such manner, the tape head will always scan the first and last symbols of strings x and y at the same times whenever x aligns with y. This notion is useful as we seek to generalize the intuition of alignment to substrings.

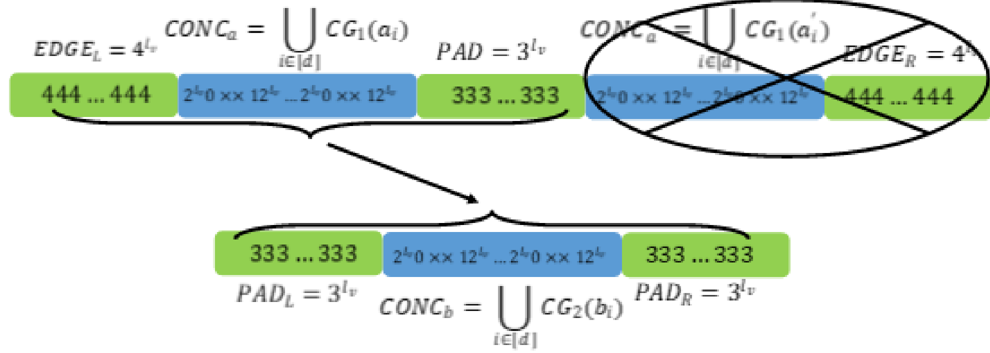
Definition 4.3. Consider a transformation of string s into string t, a contiguous sub-string of s, $s' = s_0 \dots s_k$ and a contiguous sub-string of t, $t' = t_0 \dots t_p$. We say that s' is aligned with t' if there is a time in the conversion of s into t such that s_0 is scanned at the same time as t_0 and s_k is scanned at the same time as t_k.

This means that either $s_i = t_i$ or s_i is eventually replaced with t_i for all $i \in [k]$. Here we use the word eventually as it may be the case that s_i takes a long route to be replaced by t_i. Take for example the action of deleting s_i and then inserting t_i. The presented notion of alignment still allows us to say that s_i is aligned with t_i although s_i was not explicitly substituted with t_i. This notion will be critical to the rest of the proof as we will need the gadgets to have a certain alignment and we show that the alignment we want is in fact the optimal alignment.

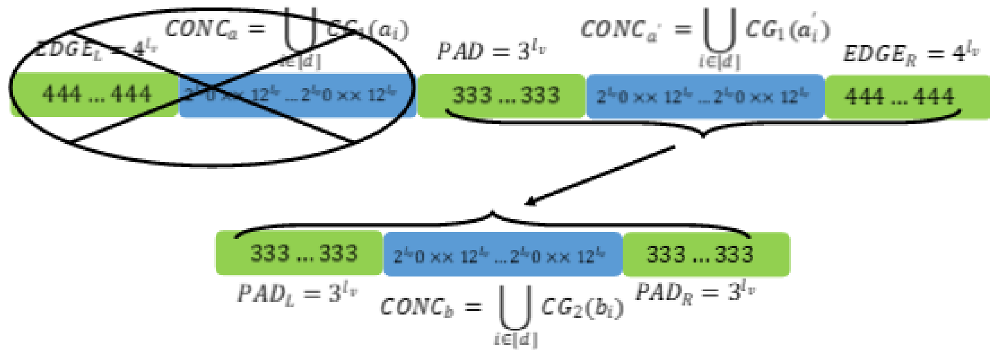
Upper bound of claim:

The upper bounds of our claims will always be the "easy" cases as we simply give the alignment we actually want to be optimal.

One of the alignments we show to be optimal occurs when $a \cdot b = 0$. Delete $\textsc{Conc}_{a'}$ and $\textsc{Edge}_R$. This costs $\mu + l_v$. Then convert $\textsc{Edge}_L$ into $\textsc{Pad}_L$ which adds a cost of l_v. Finally, convert $\textsc{Conc}_a$ into $\textsc{Conc}_b$. Because a and b are orthogonal, this only costs d. We then get a final distance of $2l_c + \mu + d$. The following image summarizes this alignment.



On the other hand, if $a \cdot b \neq 0$ then delete $\textsc{Edge}_l$ and $\textsc{Conc}_a$. This costs $\mu + l_v$. Now convert $\textsc{Edge}_R$ into $\textsc{Pad}_R$ which induces a cost of l_v. Convert $\textsc{Conc}_{a'}$ into $\textsc{Conc}_b$. Since a' is zero every where but its first coordinate and b's first coordinate is 1 we have that $a' \cdot b = 1$. Thus, converting $\textsc{Edge}_r$ into $\textsc{Pad}_r$ gives us a cost of $d + 2$. Thus, the distance is $2l_c + \mu + d + 2$. This case is summarized with the following image.



Putting the previous arguments together we have that $d_e(VG_1(a), VG_2(b))$ is bounded above by our claim.

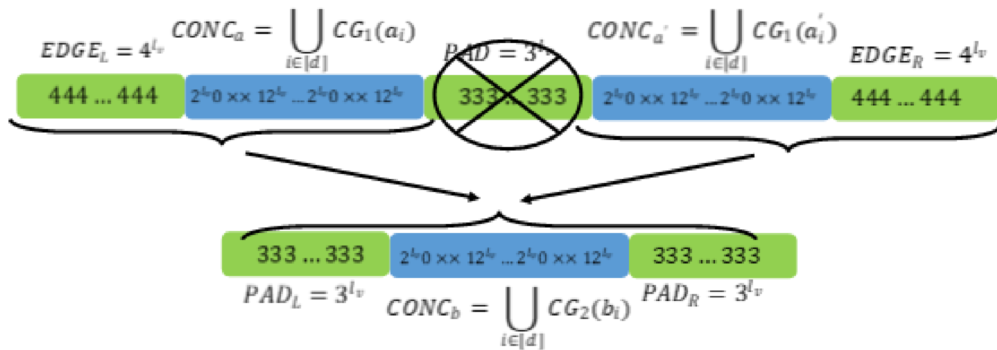
We now show that the alignment that we have chosen is in fact the optimal. Note that in our upper bound we had an alignment between $\textsc{Conc}_b$ and either $\textsc{Conc}_a$ or $\textsc{Conc}_{a'}$. In what follows we show that this is indeed the ideal route in which to convert $VG_1(a)$ to $VG_2(b)$.

Lower bound of claim:

We will prove the lower bound by exhaustion. In particular, we must show that the cases below yield a greater bound than our intuitive alignment above.

Concatenated Middle Strings:

In what follows, we refer to a contiguous sub-string of some string x as a "part" of x. First one may ask, what would happen if in the conversion of $VG_1(a)$ into $VG_2(b)$, aligning parts of $\textsc{Conc}_b$ with parts of the merged string $\textsc{Conc}_a\textsc{Conc}_{a'}$ gives a smaller cost than our desired alignment. This can be pictured below.



In this case, we would be required to delete $\textsc{Pad}$ and change substrings $\textsc{Pad}_L$ and $\textsc{Pad}_R$ into $\textsc{Edge}_L$ and $\textsc{Edge}_R$ respectively. This induces as cost of at least $3l_v$ so that $d_e(VG_1(a), VG_2(b)) \geq 3l_v$. However, simply by deleting substrings $\textsc{Conc}_a, \textsc{Conc}_{a'}, \textsc{Conc}_b, \textsc{Edge}_R$ and replacing every 4 in $\textsc{Pad}_L$ with a 3, in order to convert $\textsc{Pad}_L$ into $\textsc{Edge}_L$ we obtain the upper bound $2l_v + 3\mu$. By simple counting we obtain $\mu = d_e(4 + l_c)$ and through our choice of l_c and l_v one can confirm that $l_v > 3\mu$ so that $2l_v + 3\mu < 3l_v$ which shows that this is not optimal.

Thus, it follows that some parts of $\textsc{Conc}_b$ must be aligned to either some parts of $\textsc{Conc}_a$ or some parts of $\textsc{Conc}_{a'}$ but not both. However, regardless of which string parts the parts of $\textsc{Conc}_b$ are aligned to, we prove that $d_e(VG_1(a), VG_2(b)) \geq 2l_v + \mu$. Indeed suppose that some parts of $\textsc{Conc}_b$ aligns with some parts of $\textsc{Conc}_{a'}$ ($\textsc{Conc}_a$ respectively). This means that we must substitute or delete every symbol in $\textsc{Edge}_L\textsc{Conc}_a$ ($\textsc{Conc}_{a'}\textsc{Edge}_R$ respectively). This induces a cost of $l_v + \mu$. Furthermore, because all symbols in $\textsc{Edge}_R$ ($\textsc{Edge}_L$ respectively) are 4's while all symbols in $\textsc{Pad}_R$ ($\textsc{Pad}_L$ respectively) are 3's, we will a cost of l_v in order to edit $\textsc{Edge}_R$ ($\textsc{Edge}_L$) into $\textsc{Pad}_R$ ($\textsc{Pad}_L$). Thus, we have established a minimum cost of $2l_v + \mu$. The rest of the cases will be dedicated to computing lower bounds for the contribution of vector gadgets $\textsc{Conc}_a$, $\textsc{Conc}_{a'}$, and $\textsc{Conc}_b$.

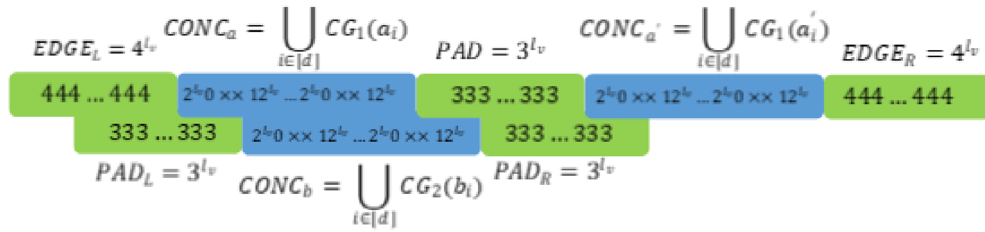
Alignment with $\textsc{Conc}_{a'}$:

We have proved that $\textsc{Pad}$ acts as a divider between $\textsc{Conc}_a$ and $\textsc{Conc}_{a'}$. This means that if $\textsc{Conc}_b$ has an alignment with $\textsc{Conc}_a$ ($\textsc{Conc}_{a'}$ respectively), then every symbol in $\textsc{Conc}_{a'}\textsc{Edge}_R$ ($\textsc{Edge}_L\textsc{Conc}_a$ resp.) must be either deleted or substituted. This operation will then contribute to a cost of $\mu+l_v$. We conclude that the optimal alignment cost is at least $\mu + l_v$. Furthermore, at least part of only one, either $\textsc{Conc}_a$ or $\textsc{Conc}'_a$ must be aligned with $\textsc{Conc}_b$.

Without loss of generality, we look at the case where $\textsc{Conc}_a$ has an alignment with $\textsc{Conc}_b$. $\textsc{Conc}_a$ is made up of concatenated coordinate vectors. In each one, we have blocks of 1's decided by whether the vector at each coordinate is orthogonal or not. Denote the ith run of 1s in $\textsc{Conc}_a$ as $(\textsc{Conc}_a)_i$ and likewise denote the jth run of 1s in $\textsc{Conc}_b$ as $(\textsc{Conc}_b)_j$.

Case 1:

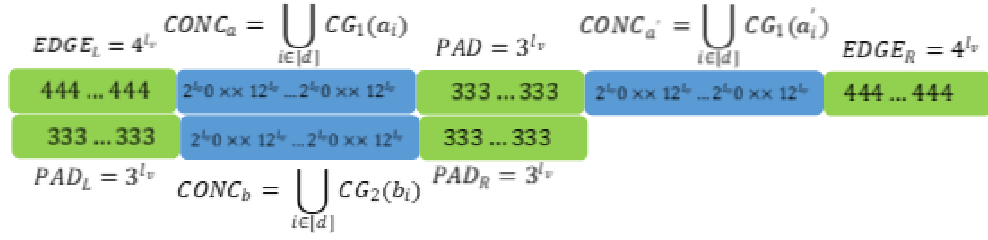
Suppose that we have an alignment between two "off" blocks of 1's. That is, there is an alignment between block i of $\textsc{Conc}_a$ and j of $\textsc{Conc}_b$ for $i \neq j$, $i, j \in [d]$ as shown below.



We then have that the number of symbols by which $\textsc{Conc}_a$ and C_b differ by after the i and j block of 1's is at least $2l_c$. We thus have an addition to the minimum cost of our edit distance of $2l_c \geq d+2$. We then have an overall minimum cost of $2l_v + \mu + d + 2$ as desired.

Case 2:

If no such alignment between two "off" blocks exists then for all $i \in [d]$ the ith block of 0s and 1s from $\textsc{Conc}_a$ has an alignment with the ith block of 0s and 1s in $\textsc{Conc}_b$.



If the ith run of 0s and 1s in $\textsc{Conc}_a$ is aligned with the ith run of 0s and 1s in $\textsc{Conc}_b$ then by design, the optimal way to convert the ith run of 0s and 1s from $\textsc{Conc}_a$ to the ith run of $\textsc{Conc}_b$. This gives a cost of $1 + 2a_i'b_i'$.

In the runs of 0s and 1s where we do not have alignment we must get rid of the runs of 1s. We have that since each run of 1s has at least 2 1s in C_a and each run of 1st has at least 1 1s in C_b, then the edit cost is at least $2 + 1 = 3$. Let $I = \{i : i\text{th run in } \textsc{Conc}_{a'} \text{ has an alignment with the } i\text{th run in } \textsc{Conc}_b\}$. We have that edit cost over all runs is the following.

$$\sum_{i:i\in I}(1+2a_i'b_i) + \sum_{i:i\in[d]-I} 3 \geq \sum_{i}^{d}(1+2a_i'b_i) = d + 2(a'\cdot b)$$

Since all of these changes are minimal independent changes taken to convert string $VG_1(a)$ into $VG_2(b)$, we have $d_e(VG_1(a), VG_2(b)) \geq 2l_v + \mu + d + 2(a' \cdot b)$.

Alignment with $\textsc{Conc}_a$

The very last case happens when $\textsc{Conc}_{a'}$ has alignment with $\textsc{Conc}_b$. This case is almost identical to the case when $\textsc{Conc}_b$ has no alignment with $\textsc{Edge}_l\textsc{Conc}_a$ as we did not make assumptions about a. By symmetry, it holds that minimal cost in this case is

$$\sum_{i:i\in I}(1+2a_ib_i) + \sum_{i:i\in[d]-I} 3 \geq \sum_{i}^{d}(1+2a_ib_i) = d + 2(a\cdot b) \quad (2)$$

Putting (1) and (2) together we obtain the bound $d_e(VG_1(a), VG_2(b)) \geq d + 2\min(a \cdot b, a' \cdot b)$. Since $a' \cdot b = 1$ holds for every $a, b \in \{0,1\}^d$ and $a \cdot b \geq 1$ if and only if a and b are not orthogonal the claim follows.

Pattern Gadgets:

Before building the complete Set Gadgets, we first take a step back into a different notion.

Definition 4.4. Define the pattern distance of x and y, $d_p(x, y)$, to be the minimum edit distance between x and any contiguous sub-string of y.

Pattern distance will be useful to us by letting us argue bounds of only the concatenated vector gadgets. Define the set pattern gadgets as follows:

$$SG_1 = \bigcup_{a \in A} 5^{l_s} VG_1(a) 5^{l_s}$$

$$SG_2 = (\bigcup_{i=1}^{|A|-1} 5^{l_s} VG_2(\mathbf{1}) 5^{l_s})(\bigcup_{b \in B} 5^{l_s} VG_2(b) 5^{l_s})(\bigcup_{i=1}^{|A|-1} 5^{l_s} VG_2(\mathbf{1}) 5^{l_s})$$

where $l_s = 1000d|VG_1(a)| = O(d^3)$ and $\mathbf{1} = 1^d$.

Claim:

$$d_p(SG_1, SG_2) = NV_{no} \text{ if } a \cdot b \geq 1 \text{ for all } a \in A, b \in B$$

$$d_p(SG_1, SG_2) \leq NV_{no} - V_{no} + V_o \text{ if } a \cdot b = 0 \text{ for some } a \in A, b \in B$$

Upper Bound:

Consider the transformation of SG_1 in to the contiguous sub-sequence $\bigcup_{b \in B} 5^{l_s} VG_2(b) 5^{l_s}$ of SG_2. We obtain the transformation by aligning the vector gadget of the ith element in A with the vector gadget of the ith element in B. By our previous results, this results on the cost V_o if the ith element of A is orthogonal to the ith element of B and V_{no} otherwise. Thus, if there is no pair of orthogonal vectors in A or B, we have that the cost is $|A|V_{no} = NV_{no}$. Otherwise, there exists some elements $a \in A$ and $b \in B$ such that $a \cdot b = 0$. We subtract the cost created by a non-orthogonal pair of vectors and add the cost of a orthogonal pair of vectors. This gives us $NV_{no} - V_{no} + V_o$.

Lower Bound:

Now we claim that if there does not exist a pair of orthogonal vectors in sets A, B then $d_p(SG_1, SG_2) \leq NV_{no}$. To show this, we will focus on every vector gadget that makes up SG_1 and SG_2. We will show that every vector gadget independently contributes a cost of at least V_{no}. We consider various cases.

Fix $a \in A$. **Case 1:**

$VG_1(a)$ has an alignment with $VG_2(b)$ for more than one $b \in B$. In this case, we have that we must remove the 5's in between $VG_2(b)$ gadgets. Thus, we have a minimum cost of $2l_s > V_{no}$.

Case 2:

$VG_1(a)$ has an alignment with a single $VG_2(b)$ for unique $b \in B$. Let I be the contiguous substring of SG_2 that achieves $d_p(SG_1, SG_2)$. Two scenarios exist.

Case 2.1:

The first is that the vector gadget $VG_2(b)$ is fully contained in the intermediate substring I.

Case 2.1.1:

In this case, if no symbol of $VG_1(a)$ or $VG_2(b)$ is aligned with a 5 then by our previous results, $d_e(VG_1(a), VG_2(b)) \geq V_{no}$.

Case 2.1.2:

On the other hand, suppose symbols with indexes J_a in $VG_1(a)$ and symbols with indexes J_b in $VG_2(b)$ align with 5s in the intermediate string I, then each of these contribute a cost of 1 to the final distance. Thus, we may produce a lower bound by first deleting all symbols with indexes J_a from $VG_1(a)$ to produce $VG_1'(a)$ and also all symbols with indexes J_b from $VG_1(b)$ to produce $VG_2'(b)$ then computing $d_e(VG_1'(a), VG_2'(b))$. From our results above, it follows that the pattern distance to which $VG_1(a)$ and $VG_2(b)$ contribute is at least $d_e(VG_1'(a), VG_2'(b)) + |J_a| + |J_b| = d_e(VG_1(a), VG_2(b)) = V_{no}$ as required.

Case 2.2:

The string is not fully contained in I. If $VG_2(b)$ is not fully contained in I then it must be the case that $VG2(b)$ is either the rightmost or the leftmost vector gadget in the sub string I. WLOG, suppose it is the rightmost vector gadget. Then it follows that the symbols from $VG_2(b)$ missing in I are replaced 5s which must be substituted or deleted. Thus, this still induces a distance of V_{no}.

Since each vector gadget $VG_1(a)$ provides its own independent cost we have that in total the cost induced for all the vector gadgets of type 1 is V_{no} and so the total cost is NV_{no}.

From this, it follows that $d_p(SG_1, SG_2) \geq NV_{no}$. To obtain equality, we simply choose $I = \bigcup_{b \in B} 5^{l_s} VG_2(b) 5^{l_s}$ and run our original bound between every $VG_1(a)$ and $VG_2(b)$ over all N vector gadgets.

Set Gadgets

We now transform our problem from Pattern to Edit distance. Define

$$SG'_1 = 6^{|SG_2|}SG_1 6^{|SG_2|}$$

$$SG'_2 = SG_2$$

Claim:

If there exist $a \in A$ and $b \in B$ such that $a \cdot b = 0$ then $d_e(SG'_1, SG'_2) \leq 2|SG_2| + NV_{no} - 2$. Otherwise, $d_e(SG'_1, SG'_2) = 2|SG_2| + NV_{no}$.

Case 1:

Suppose that there does exist a pair of orthogonal vectors. We then have that an upper bound for $d_e(SG'_1, SG'_2)$ is to delete all 6s from SG'_1 and then convert the remaining string VG_1 into $VG'_2 = VG_2$. Deleting all 6 symbols costs $2|SG_2|$ and converting the remaining string gives a cost of $\leq NV_{no} - V_{no} + V_o = NV_{no} - 2$ by the above discussion.

Case 2:

Suppose that there does not exist a pair of orthogonal vectors. To convert SG'_1 into SG'_2 we must at least delete or replace the 6s in SG'_1 which gives a cost of $2|SG'_2|$ external to the cost given by the substring SG_1 of SG'_1. Now, the substring SG_1 of SG'_1 gives a cost of at least $d_p(SG_1, SG'_2) = 2|SG_2| + NV_{no}$ by the above results. Thus, we get the lower bound $d_e(SG'_1, SG'_2) \geq 2|SG_2| + NV_{no}$. Equality follows as the same computation gives the upper bound.

Runtime: We now explore the run time of the conversion from OVP to Edit Distance.

For each vector coordinate, the transformation takes computing time $2l_c + 4$. For each $a \in A$, its corresponding vector gadget, $VG_1(a)$ takes computing time $3l_v + 2d_e(2l_c + 4)$, and for each $b \in B$, its corresponding vector gadget, $VG_2(b)$ takes computing time $2l_v + 2l_c + 4$.

When we convert into a pattern instance by defining SG_1 and SG_2, we concatenate 5's at the front and back of each $VG_1(a)$ and $VG_2(b)$. This costs us $2l_s + (3l_v + 2d_e(2l_c + 4))$ and $2l_s + (2l_1 + 2l_0 + 4)$ for each $VG_1(a)$ and $VG_2(b)$ respectively.

The pattern gadget SG_1 then has computation time $N(2l_s + (3l_v + 2d_e(2l_c + 4)))$. On the other hand, the gadget SG_2 has computation time $2(N-1)(2l_s + (2l_v + 2l_c + 4)) + N(l_s + (2l_v + 2l_c + 4))$.

We then transform pattern to Edit Distance by defining $SG_1' = 6^{|SG_2|}SG_1 6^{|SG_2|}$ and $SG_2' = SG_2$. This leaves the cost of computing SG_2' fixed at $2(N-1)(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (2l_v + 2l_c + 4))$ and adds onto the cost of computing SG_1' by $2|SG_2|$ so that the final cost of computing SG_1' is $4(N-1)(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (3l_v + 2d_e(2l_c + 4)))$.

Now, the cost of computing SG_1' is

$$4(N-1)(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (3l_v + 2d_e(2l_c + 4)))$$

where $l_c = 1000d, l_v = (1000d)^2, l_s = O(d^3)$ so that the cost is

$$\begin{gathered}4(N-1)(2O(d^3) + (2(1000d)^2 + 2(1000d) + 4)) + \\ N(2O(d^3) + (2(1000d)^2 + 2(1000d) + 4)) + \\ N(2O(d^3) + (3(1000d)^2 + 2d_e(2(1000d) + 4))) \\ = O(d^3N)\end{gathered}$$

Similarly, the cost of computing SG_2' is

$$\begin{gathered}2(N-1)(2l_s + (2l_v + 2l_c + 4)) + N(2l_s + (2l_v + 2l_c + 4)) \\ = 2(N-1)(2O(d^3) + (2(1000d)^2 + 2(1000d) + 4)) + N(2O(d^3) + (2(1000d)^2 + 2(1000d) + 4)) \\ = O(d^3N)\end{gathered}$$

then it follows that the edit distance instance can be computed in time $O(d^3N) = \tilde{O}(N)$. $\square$

We recall that we may transform the problem of computing the edit distance of two strings x, y to a decision problem by asking if $d_e(x, y) \leq C$. We have shown that there is a reduction from OVP to the decision problem of Edit distance. Thus, we have the following result.

Theorem 4.5. *Edit Distance cannot be computed time $\tilde{O}(N^{2-\epsilon})$ for $\epsilon > 0$ unless SETH fails.*

Proof. Suppose that edit distance can be computed in $\tilde{O}(N^{2-\epsilon})$ time for some $\epsilon > 0$. Given an OVP instance, apply the reduction above. The reduction only takes time $\tilde{O}(N)$ and yields two edit distance strings x and y. Now compute $d_e(x, y)$. We assumed that there is algorithm that can do this in time $\tilde{O}(N^{2-\epsilon})$ for some $\epsilon > 0$. Thus, in time $\tilde{O}(N^{2-\epsilon})$, we will know if $d_e(x, y) \leq 2|SG_2| + NV_{no} - 2$. By 4.2 this happens if and only if the answer to the original OVP problem is "No." Otherwise, we will obtain $d_e(x, y) = 2|SG_2| + NV_{no}$ and the answer to the original OVP problem will be "Yes." We have thus solved the OVP problem in time $O(N^{2-\epsilon}) + \tilde{O}(N) = \tilde{O}(N^{2-\epsilon})$ which by Theorem 2.4 violates SETH. □

Bibliography

[1] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. A duality between clause width and clause density for SAT. In *IEEE Conf. on Computational Complexity (CCC'06)*, pages 252–260, 2006.

[2] Evgeny Dantsin and Edward A. Hirsch. Worst-case upper bounds. In *Handbook of Satisfiability*, volume 185, chapter 12, pages 403–424. IOS Press, Amsterdam, 2009.

[3] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62:367–375, January 2001.

[4] Jeff Erickson. Lower bounds for linear satisfiability problems. In *Symposium on Discrete Algorithms (SODA'95)*, pages 388–395, 1999.

[5] Timothy M. Chan. More logarithmic-factor speedups for 3SUM, (median,+)-convolution, and some geometric 3SUM-hard problems. In *Symposium on Discrete Algorithms (SODA'18)*, pages 881–897, 2018.

[6] Arturs Backurs, Liam Roditty, Gilad Segal, Virginia Vassilevska Williams, and Nicole Wein. Towards tight approximation bounds for graph diameter and eccentricities. In *Symposium on the Theory of Computing (STOC'18)*, 2018.

[7] Amir Abboud, Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Symposium on Discrete Algorithms (SODA'15)*, pages 218–230, 2015.

[8] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63: 512–530, December 1999.

[9] Timothy Chan and Ryan Williams. Deterministic APSP, orthogonal vectors, and more: Quickly derandomizing Razborov-Smolensky. In *Symposium on Discrete Algorithms (SODA'16)*, pages 1246–1255, 2016.

[10] Pairwise comparison of bit vectors. Theoretical Computer Science Stack Exchange, January 2017. URL `https://cstheory.stackexchange.com/q/37361`.

[11] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *CoRR*, 2014.

[12] William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[13] Gonzalo Navarro. A guided tour to approximate string matching. *JACM Computing Suveys*, 33(1):31–88, 2001.