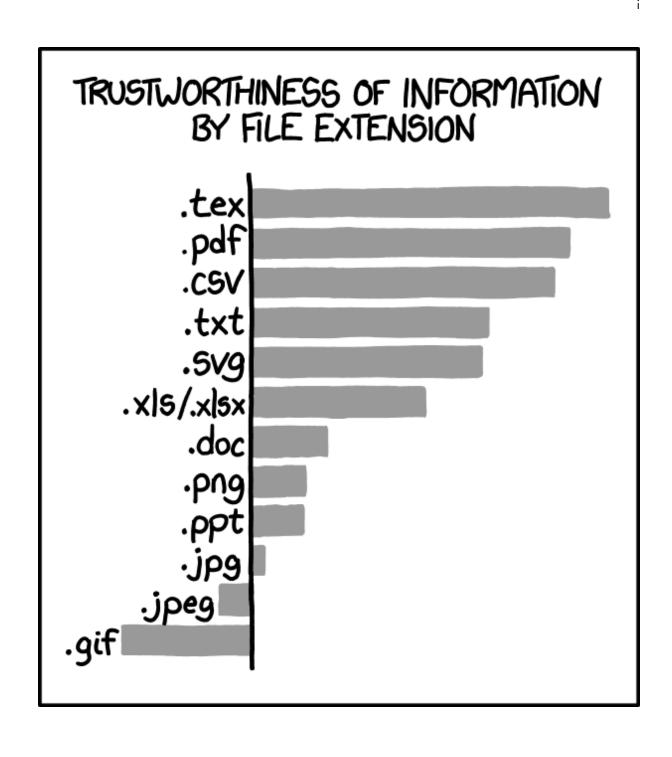
# CSCI6454: Advanced Algorithms

PROFESSOR: HUCK BENNETT, SCRIBE: ADITHYA BHASKARA

UNIVERSITY OF COLORADO BOULDER





# Contents

Preface			iii
1	Intro	oduction to Fine-Grained Complexity	1
	1.1	Lecture 1: January 14, 2025	
		1.1.1 Introduction to Fine-Grained Complexity	
	1.2	Lecture 2: January 16, 2025	
		1.2.1 Fine-Grained Complexity II	
	1.3	Lecture 3: January 21, 2025	
		1.3.1 Graph Diameter	8
	1.4	Lecture 4: January 23, 2025	9
		1.4.1 Smarter k-SAT, Part I	9
	1.5	Lecture 5: January 28, 2025	12
		1.5.1 Smarter k-SAT, Part II	12
	1.6	Lecture 6: January 30, 2025	
		1.6.1 Nontrivial Nondeterministic Algorithms: SETH and Other Assumptions	13
2	Mat	rix Multiplication	15
		Lecture 6: January 30, 2025	15
		2.1.1 Introduction to Matrix Multiplication	
	2.2	Lecture 7: February 4, 2025	16
		2.2.1 Strassen's Algorithm & Tensor Rank Techniques	
	2.3	Lecture 8: February 6, 2025	19
		2.3.1 Matrix Multiplication Verification	19
	2.4	Lecture 9: February 11, 2025	20
		2.4.1 Sparse Matrix Multiplication	20
3	ND	Hard Graph Problems	21
J		Lecture 10: February 13, 2025	
	3.1	3.1.1 Graph Coloring	
		3.1.1 Graph Coloring	21
ΑI	Algorithms as a Word Cloud		
Appendices			
Lis	ist of Theorems and Definitions		

# **Preface**

To the interested reader,

This document is a compilation of lecture notes scribed during the Spring 2025 semester for CSCI6454: Advanced Algorithms at the University of Colorado Boulder. The content of these course notes is from lecture and several resources cited herein. I only take credit for the scribing of the content and lectures into the notes. This course was taught by Huck Bennett, Ph. D. and the notes would not be possible without Huck's great lectures.

I have taken liberties, deviating from the lecture notes, when it comes to notation and ordering of content between chapters.

This course is designed as a graduate third course in algorithms. Hence, we assume as prerequisites the topics covered in undergraduate algorithms and introductory graduate courses. At CU Boulder, the prerequisites correspond to CSCI 3104 and CSCI 5454.

While much effort has been put in to remove typos and mathematical errors, it is very likely that some errors, both small and large, are present. I take full responsibility for all remaining errors. If an error needs to be resolved, please contact me at adithya@colorado.edu.

Best Regards, Adithya Bhaskara

REVISED: February 17, 2025

# Introduction to Fine-Grained Complexity

# 1.1 Lecture 1: January 14, 2025

# 1.1.1 Introduction to Fine-Grained Complexity

We hope to precisely understand the complexity of several algorithmic problems beyond the coarse information given by standard complexity classes like **P** and **NP**. A central theme is proving lower bounds for the complexity of algorithms; this is a difficult task in general. Here, reductions are an essential tool.

The high-level idea behind fine-grained complexity is to explain hardness, or the lack of algorithmic progress, on fundamental computational problems by giving reductions from well-studied problems. Recall the  $P ext{ v. } NP$  problem. Loosely, problems in P are "easy," to solve, and problems in NP are "easy" to verify positive answers to. The hardest problems in NP, the NP-complete problems, do not admit polynomial-time algorithms unless P = NP. To show that a problem is NP-complete, we show that it is in NP, and exhibit a polynomial-time reduction from a known NP-complete problem. We assume familiarity with the technical definitions and implications, and defer the reader to [HMU01, Sip12] for details.

The theory behind NP-completeness is robust and quite nice. But, we'd like finer results. Knowing whether a problem is in P or not doesn't really narrow down how efficiently we can solve it in practice. Even quadratic-time algorithms may be inefficient in production.

These ideas motivate fine-grained complexity. Here, we start by taking a well-studied problem, say  $L_1$ , and making a precise conjecture about the running time of optimal algorithms for  $L_1$ . Then, we give an efficient reduction from  $L_1$  to a problem  $L_2$ , for which we are showing hardness. We start by providing some hypotheses corresponding to certain problems, and later, we'll show fairly precise running time bounds for problems in  $\bf P$ .

**Example 1.** Under certain assumptions, there is no  $O(n^{2-\epsilon})$ -time algorithm for the EDIT DISTANCE problem. This matches the  $O(n^2)$  dynamic programming algorithm.

#### Computational Problem 1 (k-SAT).

- ullet Given a CNF formula  $\varphi$  with k literals in each clause,
- Decide: Does there exist a satisfying assignment?

# Hypothesis 1.1.1: [IP01] Exponential Time Hypothesis (ETH)

The 3-SAT problem takes  $2^{\Omega(n)}$  time.

**Remark.** Note that  $P \neq NP$  asserts that 3-SAT has no polynomial-time algorithm, whereas ETH asserts that 3-SAT has no subexponential-time algorithm. Hence, ETH implies  $P \neq NP$ .

# Hypothesis 1.1.2: [IP01, IPZ01] Strong Exponential Time Hypothesis (SETH)

For every  $\epsilon > 0$ , there exists  $k \in \mathbb{Z}^+$  such that there is no  $O\left(2^{(1-\epsilon)n}\right) = O((2-\epsilon)^n)$ -time algorithm for  $k\text{-}\mathrm{SAT}$ .

**Remark.** SETH, at a high level, claims that  $2^n$ -time is essentially optimal for k-SAT with large k.

Remark. In [IP01, IPZ01], ETH and SETH are instead stated as follows. Let

$$s_k = \inf \left\{ \delta > 0 : \exists \text{ an algorithm to solve } k\text{-SAT in } O^*\left(2^{\delta n}\right) \text{ time} \right\}.$$

ETH is the statement that for  $k \geq 3$ ,  $s_k > 0$ . Assuming ETH,  $s_k$  is increasing infinitely often, [IPZ01]. Let  $s_{\infty} = \lim_{k \to \infty} s_k$ . Then, SETH is the statement that  $s_{\infty} = 1$ .

# Computational Problem 2 (k-SUM).

- Given arrays  $A_1, ..., A_k$  each with n integers in  $[-n^c, n^c]$ ,
- Decide: Does there exist  $a_1 \in A_1, ..., a_k \in A_k$  such that  $a_1 + \cdots + a_k = 0$ ?

Some variants include assuming that  $A_1 = \cdots = A_k$ , or wanting to find  $a_1 \in A_1, \ldots, a_k \in A_k$  where  $a_1 + \cdots + a_{k-1} = a_k$ .

A naive algorithm for k-Sum is to try all possible choices  $a_i$  and verify the sum. This gives rise to an  $O(n^k)$ -time algorithm.

Question. Can we do better?

Answer. Yes, we can!

6:

For now, let k = 3. Consider the following, slightly better, algorithm.

# Algorithm 1.1.1 Better 3-SUM

return FALSE

```
      1: procedure 3-SUM-BETTER (A_1, A_2, A_3)

      2: S \leftarrow \{a_1 + a_2 : a_1 \in A, a_2 \in A_2\}
      \triangleright O(n^2)

      3: Sort A_3.
      \triangleright O(n \log n)

      4: for s \in S do
      \triangleright O(n^2 \log n)

      5: if -s \in A_3 then return TRUE
```

Algorithm 1.1.1 takes  $O(n^2 \log n)$  time, using binary search on the sorted array  $A_3$ .

Question. Can we do better?

Answer. Yes, we can!

Our first example of a helpful reduction is as follows. Given a 3-Sum instance with arrays of length n, we can construct n 2-Sum instances. This procedure gives us an  $O(n^2)$  algorithm for 3-Sum, which we describe in Algorithm 1.1.2.

# Algorithm 1.1.2 Best 3-Sum

```
1: procedure 3-SUM-BETTER(A_1, A_2, A_3)
2: Sort A_1 and A_2.
3: for a_3 \in A_3 do
4: A_2 + a_3 \leftarrow [a_{2,1} + a_3, ..., a_{2,n} + a_3]
5: if 2-SUM(A_1, A_2 + a_3) then return TRUE
6: return FALSE
```

Sorting takes  $O(n \log n)$  time. Then, we can n calls of 2-SuM that each take O(n) time since the arrays are sorted. Therefore, Algorithm 1.1.2 runs in  $O(n^2)$  time. We have the following hypothesis.

# Hypothesis 1.1.3: **No** $O(n^{2-\epsilon})$ -Time Algorithm for 3-Sum

For every  $\epsilon > 0$ , there is no  $O\left(n^{2-\epsilon}\right)$ -time algorithm for 3-SUM.

# 1.2 Lecture 2: January 16, 2025

# 1.2.1 Fine-Grained Complexity II

Now, we introduce a new problem.

Computational Problem 3 (ALL PAIRS SHORTEST PATH (APSP)).

- Given a weighted graph G = (V, E, w) on n vertices,
- Find: The lengths of the shortest paths  $d(v_i, v_i)$  between every  $v_i, v_i \in V$ .

One way to solve APSP is to run Dijkstra's algorithm on each vertex. Recall that Dijkstra's algorithm finds a single-source shortest path tree. We could also similarly use the Floyd-Warshall dynamic programming algorithm which we introduce below.

Let  $D_k[i,j]$  store the length of the shortest path from  $v_i$  to  $v_j$ , with the restriction that the path only passes through  $v_1, \ldots, v_k$ .

**Example 2.** To illustrate this notion, consider the below graph.

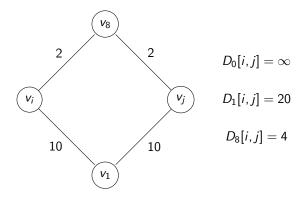


Figure 1.1: Example of  $D_k[i,j]$ 

Illustrated in Figure 1.2, the idea behind Floyd-Warshall is to build  $D_k$  from  $D_{k-1}$ , noting that either

- 1. the shortest path from  $v_i$  to  $v_i$  only using  $v_1, \ldots, v_k$  uses  $v_k$ , or
- 2. it does not.

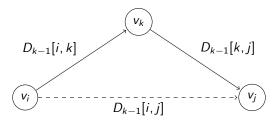


Figure 1.2: Update of  $D_k[i,j]$ 

From this basic idea, we get a simple algorithm.

# Algorithm 1.2.3 Floyd-Warshall

```
Require: G = (V, E, w).

1: procedure FLOYD_WARSHALL(E)

2: D_0 \leftarrow \begin{cases} 0 & i = j \\ w_{i,j} & i \neq j, (v_i, v_j) \in E \\ \infty & (v_i, v_j) \notin E \end{cases}

3: for k \in \{1, ..., n\} do

4: for i \in \{1, ..., n\} do

5: for j \in \{1, ..., n\} do

6: D_k[i,j] = \min\{D_{k-1}[i,k] + D_{k-1}[k,j], D_{k-1}[i,j]\}

7: return D_n \triangleright D_n[i,j] = d(v_i, v_j)
```

It turns out that as far as we know, Algorithm 1.2.3 with complexity  $O(n^3)$  is the best we can do.

# Hypothesis 1.2.1: No $O(n^{3-\epsilon})$ -Time Algorithm for APSP

For every  $\epsilon > 0$ , there exists no  $O(n^{3-\epsilon})$ -time algorithm for APSP.

We now turn to a new problem.

Computational Problem 4 (ORTHOGONAL VECTORS (OV)).

- Given two sets  $V = \{v_1, ..., v_n\} \subseteq \{0, 1\}^d$  and  $W = \{w_1, ..., w_n\} \subseteq \{0, 1\}^d$ ,
- Decide: Does there exist  $v_i \in W$  and  $w_j \in W$  such that  $\langle v_i, w_j \rangle = \sum_{k=1}^d v_i[k]w_j[k] = 0$ ?

The naive algorithm is to compare every pair, compute the inner products, and compare to 0. We have  $n^2$  inner products to compute, and each takes d time, so the naive algorithm takes  $O(dn^2)$  time. Think of  $d = \log^k(n)$  for some k. In this case,  $O(dn^2) = O(n^2 \log^k(n)) = \tilde{O}(n^2)$ .

# **Hypothesis 1.2.2:** No $O(n^{2-\epsilon})$ -Time Algorithm for OV

For every  $\epsilon > 0$ , there exists no  $O(n^{2-\epsilon})$ -time algorithm for OV with d = polylog(n).

So far, we have 5 hypotheses; what are the relations herein? We'll attack this question after a brief interlude on reductions.

We write  $L_1 \leq L_2$  if problem  $L_1$  reduces to problem  $L_2$ ; that is,  $L_1$  is no harder than  $L_2$ ; and if  $L_1$  is hard, then  $L_2$  is hard. Equivalently, if  $L_2$  is easy, then  $L_1$  is easy. Consider the following definition, making precise the notion of a "fine-grained reduction."

# **Definition 1.2.1: ◎ [VW19] Fine-Grained Reductions**

Suppose  $L_1$  and  $L_2$  are computational problems. Let  $\ell_1=\ell_1(n)$  and  $\ell_2=\ell_2(n)$  be associated conjectured optimal runtime bounds. Then,  $L_1$  ( $\ell_1,\ell_2$ )-reduces to  $L_2$  if for every  $\epsilon>0$ , there exists  $\delta>0$  and an algorithm R for  $L_1$  running in time  $O(\ell_1(n)^{1-\delta})$  and making q calls to an oracle for  $L_2$  on inputs of size  $n_1,\ldots,n_q$  where

$$\sum_{i=1}^q \ell_2(n_i)^{1-\epsilon} \leq O(\ell_1(n)^{1-\delta}).$$

We write  $L_1 \leq_{\ell_1,\ell_2} L_2$ . If both  $L_1 \leq_{\ell_1,\ell_2} L_2$  and  $L_2 \leq_{\ell_1,\ell_2} L_1$ , we say  $L_1 =_{\ell_1,\ell_2} L_2$ .

# Theorem 1.2.1: $\bigcirc$ [Wil04] SETH $\Longrightarrow$ OV

Hypothesis 1.1.2 implies Hypothesis 1.2.2.

*Proof.* Using Definition 1.2.1, we can restate this theorem as follows: for every  $k \in \mathbb{Z}^+$ , k-SAT  $(2^n, n^2)$ -reduces to OV with d = polylog(n). We exhibit such a reduction, which will be of exponential-time.

Let  $\varphi(x_1,\ldots,x_n)=\bigwedge_{j=1}^m C_j$  be our k-SAT instance, with each  $C_j$  a k-clause. Partition  $X=\{x_i\}_{i=1}^n$  into  $X_1=\{x_1,\ldots,x_{\frac{n}{2}}\}$  and  $X_2=\{x_{\frac{n}{2}+1},\ldots,x_n\}$ . Consider the  $N=2^{\frac{n}{2}}$  assignments to each of  $X_1$  and  $X_2$  independently. Then, let  $V_1,V_2\subseteq\{0,1\}^m$ . For an assignment  $a_1$  to literals in  $X_1,v_{a_1}\in V_1$  is defined by

$$v_{a_1}[j] = egin{cases} 0 & a_1 ext{ satisfies } C_j \ 1 & ext{otherwise} \end{cases}.$$

Define  $v_{a_2} \in V_2$  similarly corresponding to  $a_2$  with literals in  $X_2$ . We claim that  $\langle v_{a_i}, v_{a_2} \rangle = 0$  if and only if  $[a_1, a_2]$  is a satisfying assignment. Note that  $m = O(n^k) = O((2\log_2 N)^k) = O(\log^k N)$ , where  $N = 2^{\frac{n}{2}}$ . A  $O(N^{2-\epsilon})$ -time algorithm for OV would then imply an  $O(N^{2-\epsilon}) = O\left(\left(2^{\frac{n}{2}}\right)^{2-\epsilon}\right) = O\left(2^{\left(1-\frac{\epsilon}{2}\right)n}\right)$ -time algorithm for k-SAT.

**Example 3.** To illustrate the reduction in the proof of Theorem 1.2.1, consider k-SAT with n = 6, m = 3, and k = 3 and formula

$$\varphi(x_1,\ldots,x_6)=(x_1\vee\neg x_2\vee x_6)\wedge(x_2\vee x_3\vee x_4)\wedge(\neg x_4\vee\neg x_3\vee x_6).$$

For  $a_1 = [0, 1, 0]$ , we have  $v_{a_1} = [1, 0, 0] \in V_1$  and for  $a_2 = [0, 1, 1]$ , we have  $v_{a_2} = [0, 1, 0] \in V_2$ . Indeed  $\varphi(a_1, a_2) = 1$  and  $\langle v_{a_1}, v_{a_2} \rangle = 0$ .

# 1.3 Lecture 3: January 21, 2025

# 1.3.1 Graph Diameter

Consider the following problem.

Computational Problem 5 (Graph Diameter).

- Given a graph G = (V, E, w),
- Find:  $\max_{u,v \in V} d(u,v)$  where d(u,v) is the shortest path distance between  $u,v \in V$ .

An easy algorithm is to run Floyd-Warshall and output the largest distance found. This takes  $O(n^3)$  time.

# 

The OV problem  $(n^2, n^2)$ -reduces to GRAPHDIAMETER on undirected, unweighted graphs with O(n) vertices and  $\tilde{O}(n)$  edges.

*Proof.* Let 
$$V = [v_1', \dots, v_n'] \subseteq \{0, 1\}^d$$
 and  $W = [w_1', \dots, w_n'] \subseteq \{0, 1\}^d$ 

**Remark.** Theorem 1.3.1 immediately shows that assuming Hypothesis 1.2.2 or SETH, there exists no  $O\left(n^{2-\epsilon}\right)$ -time algorithm to  $\left(\frac{3}{2}-\delta\right)$ -approximate GraphDiameter for all  $\epsilon,\delta>0$ .

[AB: Finish this! I was very tired during lecture...]

# 1.4 Lecture 4: January 23, 2025

# 1.4.1 Smarter k-SAT, Part I

In this section, we detail faster algorithms for solving 3-SAT. But first, we recall some standard facts; 2-SAT is in **P** and k-SAT is **NP**-complete for  $k \ge 3$ . The brute force algorithm takes  $O(2^n)$  time. Now, we show how to convert  $\varphi$ , a k-SAT formula with n variables and m clauses, into a 3-SAT formula. Write

$$\varphi = \bigwedge_{i=1}^{m} \left( x_i^1 \vee \dots \vee x_i^k \right)$$

$$\iff \bigwedge_{i=1}^{m} \left[ \left( x_i^1 \vee x_i^2 \vee y_i^1 \right) \wedge \left( \overline{y_i^1} \vee x_i^3 \vee y_i^2 \right) \wedge \dots \wedge \left( \overline{y_i^{k-2}} \vee x_{k-2} \vee y_i^{k-3} \right) \wedge \left( \overline{y_i^{k-3}} \vee x_{k-1} \vee x_k \right) \right]$$

Our new 3-SAT formula has n+m(k-3) literals and m(k-3) clauses, which is apparent from the line above.

We also present the useful sparsification lemma. Morally, we can reduce solving a k-SAT instance with n variables to solving several k-SAT instances each with n variables O(n) clauses. Informally, "every k-SAT formula has O(n) clauses."

#### **Theorem 1.4.1:** The Sparsification Lemma

Let  $\epsilon>0$  and  $k\geq 3$ . Then, there is a  $2^{\epsilon n}$  poly(n)-time algorithm that takes a k-SAT formula  $\varphi$  on n variables as input and produces  $2^{\epsilon n}$  k-SAT formulas  $\varphi_1,\ldots,\varphi_{2^{\epsilon n}}$ , each of which has n variables and  $n\left(\frac{k}{\epsilon}\right)^{O(k)}=O(n)$  clauses. Furthermore,  $\varphi$  is satisfiable if and only if  $\bigvee_i \varphi_i$  is satisfied.

With the help of Theorem 1.4.1, we can show that the "strong" exponential time hypothesis indeed implies the exponential time hypothesis. The proof follows by contraposition, first applying the sparsification lemma to the k-SAT formula, and then converting each  $\varphi_i$  into a 3-SAT formula. Indeed, SETH implies ETH.

Now, we tackle 3-SAT itself. A very simple, but clever optimization gives us a  $O(1.913^n)$ -time algorithm. Let  $\varphi$ ; a denote a partial assignment to  $\varphi$  according to a.

# **Algorithm 1.4.4** 3-SAT Easy Recursive Algorithm; $O(1.913^n)$

- 1: **procedure** 3-SAT-EASY-RECURSIVE( $\varphi$ )
- 2: **if**  $\varphi$  only has 1 and 2-clauses **then return**  $2\text{-}\mathrm{SAT}(\varphi)$

⊳ in poly(n) time

- 3:  $C \leftarrow \ell_1 \lor \ell_2 \lor \ell_3$
- 4: **for** each of 7 partial satisfying partial assignments  $\mathbf{a}_C$  to C **do**
- 5: **return** 3-SAT-EASY-RECURSIVE( $\varphi$ ; **a**<sub>C</sub>)
- 6: **return** FALSE

Algorithm 1.4.4 takes time

$$T(n) = 7T(n-3) + \text{poly}(n)$$
  
=  $7^{\frac{n}{3}} \text{ poly}(n) = 2^{\frac{\log_2 7}{3}} \text{ poly}(n) = O(1.913),$ 

as desired.

Question. Can we do better?

Answer. Yes, we can!

To improve, we take the approach of [MS85]. Consider a clause  $C = x_1 \vee x_2 \vee x_2$ . Note that

$$C = x_1 \vee x_2 \vee x_2 \iff x_1 \vee (\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3).$$

We can think of the above formula as either setting the truth of one, two, or three literals. This gives rise to a modification of Algorithm 1.4.4, making 3 recursive calls. This gives us the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + poly(n) = O^*(\alpha^n)$$

where  $\alpha$  is the unique real root to  $\alpha^n - \alpha^{n-1} - \alpha^{n-2} - \alpha^{n-3} = 0$ , or equivalently,  $\alpha^3 - \alpha^2 - \alpha - 1$ . Here,  $\alpha \le 1.84$ . So,  $T(n) = (1.84^n)$ .

Question. Can we do better?

Answer. Yes, we can!

Local search algorithms for 3-SAT turn out to be very powerful. We proceed, following [Sch99, Spi07]. Here, we harness the power of randomness.

# Algorithm 1.4.5 Schöning SAT

```
2: Guess a uniformly random assignment to \mathbf{a} \in \{0,1\}^n to \varphi.

3: for j \in \{1, ..., \beta_n\} do

4: if \varphi is satisfied by a then return a

5: else
```

6: Choose an unsatisfied clause C in  $\varphi$ , and choose a variable  $x_i$  in C uniformly at random.

7: Flip the assignment  $a_i$  to  $x_i$  in **a**.

1: **procedure** SCHÖNING\_SAT\_FRAMEWORK $(\varphi, \beta_n)$ 

8: return a

**Example 4.** Say  $C = (x_1, x_2, x_3)$  with  $a_1 = 0$ ,  $a_2 = 0$ , and  $a_3 = 0$ . Since C is unsatisfied, we can flip one of the literals uniformly at random to satisfy C.

Importantly, how do we choose  $\beta_n$ ? Suppose that the initial assignment  $\mathbf{a}$  agrees with the satisfying assignment  $\mathbf{a}^*$  on a "decent" fraction of coordinates with "decent" probability. The steps in the loop can be thought of as doing a random walk on  $\{0,1\}^n$ . At each step,  $\mathbf{a}$  gets closer to  $\mathbf{a}^*$  with probability at least  $\frac{1}{3}$  since at least one variable in each unsatisfied clause differs between  $\mathbf{a}$  and  $\mathbf{a}^*$ .

For "Easy"-Schöning, take  $\beta_n = n$ . Say u is the number of coordinates in which  $\mathbf{a}$  and  $\mathbf{a}^*$  are different. Then, the probability of Algorithm 1.4.5 finding a satisfying assignment is given by

$$\begin{split} \Pr[\text{satisfying assignment found}] &\geq \sum_{i=0}^n \Pr[u=i] \Pr[\text{first } i \text{ choices match}] \\ &= \sum_{i=0}^n \frac{1}{2^n} \binom{n}{i} \Pr[\text{first } i \text{ choices match}] \\ &\geq \frac{1}{2^n} \sum_{i=0}^n \binom{n}{i} \frac{1}{3^i} \\ &= \frac{1}{2^n} \left(1 + \frac{1}{3}\right)^n \\ &= \left(\frac{2}{3}\right)^n. \end{split}$$

Intuitively, we sample a random assignment  $\mathbf{a}$  which differs from  $\mathbf{a}^*$  in i coordinates. We want to keep flipping assignments so that  $\mathbf{a} = \mathbf{a}^*$ . The above procedure succeeds with probability at least  $\left(\frac{2}{3}\right)^n$ . So, we can repeat Algorithm 1.4.5 roughly  $\left(\frac{3}{2}\right)^n$  times with  $\beta_n = n$  to find a solution with high probability.

# 1.5 Lecture 5: January 28, 2025

# 1.5.1 Smarter k-SAT, Part II

To get an even better runtime, we can refine the procedure described in the previous section. What if we apply Algorithm 1.4.5 with  $\beta_n = 3n$ ? Let's analyze the probability that

- 1. the initial assignment  $\mathbf{a}$  differs with  $\mathbf{a}^*$  on exactly i coordinates, and
- 2. the first 3i steps of the walk include at least 2i choices satisfying all clauses.

For (1), the analysis is very similar to "Easy"-Schöning. First, recall that for  $i \ge 2$ ,

$$\binom{3i}{i} \ge \frac{1}{\sqrt{5i}} \frac{3^{3i}}{2^{2i}}$$

by Stirling's approximation. Then,

 $\Pr[\text{at least } 2i \text{ of the first } 3i \text{ choices satisfy all clauses}] \geq \binom{3i}{i} \left(\frac{1}{3}\right)^{2i} \left(\frac{2}{3}\right)^i \geq \frac{1}{\sqrt{5i}} \frac{3^{3i}}{2^{2i}} \left(\frac{1}{3}\right)^{2i} \left(\frac{2}{3}\right)^i.$ 

Then,

 $\begin{aligned} \Pr[\text{satisfying assignment found}] &\geq \sum_{i=0}^n \Pr[u=i] \Pr[\text{at least } 2i \text{ of the first } 3i \text{ choices satisfy all clauses}] \\ &\geq \frac{1}{2^n} \sum_{i=0}^n \binom{n}{i} \binom{3i}{i} \left(\frac{1}{3}\right)^{2i} \left(\frac{2}{3}\right)^i \\ &\geq \frac{1}{2^n} \sum_{i=0}^n \binom{n}{i} \frac{1}{\sqrt{5i}} \frac{3^{3i}}{2^{2i}} \left(\frac{1}{3}\right)^{2i} \left(\frac{2}{3}\right)^i \\ &\geq \frac{1}{2^n \sqrt{5n}} \sum_{i=0}^n \binom{n}{i} \frac{1}{\sqrt{5i}} \frac{3^{3i}}{2^{2i}} \left(\frac{1}{3}\right)^{2i} \left(\frac{2}{3}\right)^i \\ &= \frac{1}{2^n \sqrt{5n}} \sum_{i=0}^n \binom{n}{i} \frac{1}{2^i} \\ &= \frac{1}{2^n \sqrt{5n}} \left(1 + \frac{1}{2}\right)^n \\ &= \frac{1}{\sqrt{5n}} \left(\frac{3}{4}\right)^n. \end{aligned}$ 

Proceeding similarly, we can run Algorithm 1.4.5 roughly  $O^*\left(\frac{4}{3}\right)^n$  times with  $\beta=3n$  to find a solution with high probability.

Question. Is this optimal?

Answer. No! But, we'll end here.

# 1.6 Lecture 6: January 30, 2025

# 1.6.1 Nontrivial Nondeterministic Algorithms: SETH and Other Assumptions

Consider the following definition.

# **Definition 1.6.1:** ® NTIME

A language  $L \subseteq \{0,1\}^n$  is in NTIME $(f_n)$  if there exists an algorithm M such that for  $x \in \{0,1\}^*$  with |x| = n, M runs in time f(n) and

- 1. if  $x \in L$ , then there exists  $w \in \{0, 1\}^*$  with M(x, w) accepting, and
- 2. if  $x \notin L$ , for all  $w \in \{0, 1\}^*$ , M(x, w) is rejecting.

Remark. The familiar complexity class NP is given by

$$\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathsf{NTIME}(n^k).$$

Herein, we reference [CGI+16]. Recall that k-SAT  $\in$  **NP**.

Question. Can we say anything about verifying that a k-SAT formula is unsatisfiable?

# **Hypothesis 1.6.1:** ® **NSETH**

For every  $\epsilon > 0$ , there exists k with  $\overline{k\text{-SAT}} \notin \mathsf{NTIME}\left(2^{(1-\epsilon)n}\right)$ .

**Remark.** *NSETH* essentially hypothesizes that k-SAT has no nontrivial verification algorithm.

The key takeways of [CGI+16] are that

- 1. assuming NSETH,  $\overline{k\text{-}\mathrm{SAT}}$  does not have nontrivial nondeterministic algorithms,
- 2. 3-Sum and APSP do have nontrivial nondeterministic algorithms, and
- 3. assuming (1), (2) and NSETH, there is no fine-grained reduction from k-SAT to 3-SUM or APSP, as opposed to the k-SAT to OV reduction.

Consider the following theorem.

# Theorem 1.6.1: $\odot$ [CGI<sup>+</sup>16] Nondeterministically Solving $\overline{3\text{-Sum}}$ in $\tilde{O}(n^{\frac{3}{2}})$ Time

There is a  $\tilde{O}\left(n^{\frac{3}{2}}\right)$ -time nondeterministic algorithm for  $\overline{\text{3-Sum}}$ .

*Proof.* Take the variant of 3-SUM where we are given a single list  $A = [a_1, ..., a_n]$  where  $a_i \in [-n^c, n^c]$ , and we are to decide if there exist  $a_i, a_j, a_k \in A$  with  $a_i + a_j + a_k = 0$ . So, for  $\overline{3\text{-SUM}}$ , we must verify that for all  $i, j, k \in \{1, ..., n\}$ ,  $a_i + a_j + a_k \neq 0$ . The witness is a triple (p, t, S) where

- 1. p is a prime which is at most the  $n^{1.5\text{TH}}$  prime number,
- 2.  $S = \{(i, j, k) : a_i + a_j + a_j \equiv 0 \pmod{p}\}$ , and
- 3.  $t \in \mathbb{Z} \cap [0, 3cn^{1.5} \log n]$  such that t = |S|.

We wish to show that such a witness exists. Let R be the set of all pairs ((i,j,k),p) where p is a prime which is at most the  $n^{1.5\text{TH}}$  prime number and  $a_i+a_j+a_k\equiv 0\pmod{p}$ . Then,  $|R|\leq n^3\log(3n^c)\leq 3cn^3\log n$ . There are  $n^3$  triples (i,j,k), and  $3n^c$  gives an upper bound on the magnitude of  $a_i+a_j+a_k$ , and the logarithm is an upper bound on the number of prime factors of this quantity. Now, by an averaging argument, there must exist  $p_0$  which is at most the  $n^{1.5\text{TH}}$  prime number such that the number of pairs  $((i,j,k),p_0)$  is at most  $\frac{|R|}{n^{1.5}}\leq 3cn^{1.5}\log n$ . We now show verification given input A. First, check that for all  $r\in\{1,\ldots,t\}$ ,

$$a_{ir} + a_{ir} + a_{kr} \equiv 0 \pmod{p}$$
,  $a_{ir} + a_{ir} + a_{kr} \neq 0$ .

Then compute the number of triples summing to 0 modulo p compare this with t. To compute all triple sums, map

$$A = [a_1, \dots, a_n] \mapsto q_A(x) = \sum_i x^{a_i \bmod p}$$

and use the fast Fourier transform to compute  $(q_A(x))^3$ . If  $b_j$  is the coefficient of  $x^j$  in  $(q_A(x))^3$ ,  $b_3$  is the number of triples in  $\{a_1 \bmod p, \ldots, a_n \bmod p\}$  summing to j. Now, since  $\deg(q_A(x))^3 \le 3(p-1) < 3p$ , we can just check

$$b_0 + b_p + b_{2p} = t$$
,

and accept if we have equality, and reject otherwise.

For complexity, the first check for  $r \in \{1, ..., t\}$  takes time  $O(t) = O\left(n^{\frac{3}{2}}\log n\right)$ . The fast Fourier transform computation takes time  $O(p\log p) = \tilde{O}\left(n^{\frac{3}{2}}\right)$  where  $p = O\left(n^{\frac{3}{2}}\log n\right)$ . In short, our procedure takes  $O\left(n^{\frac{3}{2}}\log n\right)$ , as desired.

# Matrix Multiplication

# 2.1 Lecture 6: January 30, 2025

# 2.1.1 Introduction to Matrix Multiplication

Consider the following problem.

Computational Problem 6 (MATRIX MULTIPLICATION).

- Given matrices  $A, B \in \mathbb{R}^{n \times n}$ ,
- Find: C = AB, where  $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ .

Naively, we immediately have an  $O(n^3)$  algorithm. We compute  $n^2$  inner products, each taking time O(n).

Question. Can we do better?

Answer. Yes, we can!

# **Definition 2.1.1:** Matrix Multiplication Exponent

The Matrix Multiplication exponent  $\omega$  is

$$\omega = \inf \left\{ \omega' > 0 : \forall \epsilon, \exists \text{ an algorithm to solve } \mathrm{MATRIX} \ \mathrm{MULTIPLICATION} \ \mathrm{in} \ \mathit{O} \left( \mathit{n}^{\omega' + \epsilon} \right) \ \mathrm{time} \right\}.$$

**Remark.** We know  $\omega \in [2,3]$ . It is a conjecture that  $\omega = 2$ . The best known upper bound is  $\omega < 2.371539$  due to [ADW<sup>+</sup>24].

We show how to improve  $\omega$  by exploring Strassen's algorithm, using the divide and conquer paradigm. The key idea is to write A and B in  $2 \times 2$  block form, and compute the product blockwise. That is,

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

# 2.2 Lecture 7: February 4, 2025

# 2.2.1 Strassen's Algorithm & Tensor Rank Techniques

Naively, a divide and conquer approach for matrix multiplication, following the outline of the previous section, would give the recurrence  $8T\left(\frac{n}{2}\right) + O(n^2)$  since we'd have 8 multiplications, each taking  $O(n^2)$  time. But,  $T(n) = O(n^{\log_2 8}) = O(n^3)$ . It turns out that if we're clever, we can reduce 8 multiplications to 7. In this section, we explore this approach, [Str69]. Those familiar with Karatsuba's algorithm for multiplication should notice parallels.

# Algorithm 2.2.1 Strassen

```
1: procedure STRASSEN(A, B)

2: M_1 \leftarrow \text{STRASSEN}(A_{11} + A_{22}, B_{11} + B_{22})

3: M_2 \leftarrow \text{STRASSEN}(A_{21} + A_{22}, B_{11})

4: M_3 \leftarrow \text{STRASSEN}(A_{11}, B_{12} - B_{22})

5: M_4 \leftarrow \text{STRASSEN}(A_{22}, B_{21} - B_{11})

6: M_5 \leftarrow \text{STRASSEN}(A_{11} + A_{12}, B_{22})

7: M_6 \leftarrow \text{STRASSEN}(A_{21} - A_{11}, B_{11} + B_{12})

8: M_7 \leftarrow \text{STRASSEN}(A_{12} - A_{22}, B_{21} + B_{22})

9: return \begin{bmatrix} M_1 + M_4 - M_3 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}
```

To analyze runtime for Strassen's algorithm, we have that  $7T\left(\frac{n}{2}\right) + O(n^2) = O\left(n^{\log_2 7}\right) = O\left(n^{2.808}\right)$ . Now, we seek to generalize Strassen's method. We follow [Alm21].

# **Definition 2.2.1:** • Order 3 Tensors

Let  $\mathbb F$  be a field. An order 3 tensor  $T\in\mathbb F^{a imes b imes c}$  over  $\mathbb F$  is a bilinear map

$$T: \mathbb{F}^a \times \mathbb{F}^b \to \mathbb{F}^c$$
.

Remark. Equivalently, we can think of order 3 tensors as

- 1. 3-dimensional array structures over  $\mathbb{F}$ ,
- 2. a trilinear map  $T: \mathbb{F}^a \times \mathbb{F}^b \times \mathbb{F}^c \to \mathbb{F}$ , or
- 3. a trilinear polynomial

$$\sum_{i=1}^{a} \sum_{j=1}^{b} \sum_{k=1}^{c} T[i, j, k] x_i y_j z_k.$$

We can think of  $n \times n$  square matrix multiplication as a bilinear map  $\mathbb{F}^{n^2} \times \mathbb{F}^{n^2} \to \mathbb{F}^{n^2}$ .

# **Definition 2.2.2:** Matrix Multiplication Tensor

Let  $\mathbb{F}$  be a field. The  $a \times b \times c$  matrix multiplication tensor  $\langle a, b, c \rangle$  is given by

$$\sum_{i=1}^{a} \sum_{k=1}^{b} \sum_{j=1}^{c} x_{i,k} y_{k,j} z_{i,j} = \sum_{i=1}^{a} \sum_{j=1}^{c} z_{i,j} \left( \sum_{k=1}^{b} x_{i,k} y_{k,j} \right),$$

corresponding to a bilinear map  $\mathbb{F}^{ac} \times \mathbb{F}^{cb} \to \mathbb{F}^{ab}$ .

**Remark.** If we substitute in the entries of x and y from the matrices, the coefficient of  $z_{i,j}$  corresponds to the i, j entry in the matrix product.

#### **Definition 2.2.3:** Tensor Rank

Let T be a tensor over finite sets of variables X, Y, and Z. A tensor over  $\mathbb{F}$  has rank 1 if there exist coefficients  $\alpha_x$ ,  $\beta_y$ ,  $\gamma_z \in \mathbb{F}$  with

$$T = \left(\sum_{x \in X} \alpha_x X\right) \left(\sum_{y \in Y} \beta_y y\right) \left(\sum_{z \in Z} \gamma_z z\right).$$

Then, the rank, rank (T), of a general tensor T is the minimum of rank 1 tensors that sum up to T.

We now detail an intimate connection between tensor rank and fast matrix multiplication.

# Theorem 2.2.1: Upper Bounding Matrix Multiplication Tensor Rank → Fast Algorithms

If rank  $(\langle q, q, q \rangle) \leq r$ , then there exists a  $O(n^{\log_q r})$ -time matrix multiplication algorithm.

Proof. By the definition of tensor rank, we have that

$$\langle q, q, q \rangle = \sum_{i=1}^{q} \sum_{j=1}^{q} \sum_{k=1}^{q} x_{i,k} y_{k,j} z_{i,j}$$

$$= \sum_{\ell=1}^{r} \left( \sum_{i=1}^{q} \sum_{k=1}^{q} \alpha_{i,k}^{(\ell)} x_{i,k} \right) \left( \sum_{k=1}^{q} \sum_{j=1}^{q} \beta_{k,j}^{(\ell)} y_{k,j} \right) \left( \sum_{i=1}^{q} \sum_{j=1}^{q} \gamma_{i,j}^{(\ell)} y_{i,j} \right)$$

for coefficients  $\alpha_{i,k}^{(\ell)}, \beta_{k,j}^{(\ell)}, \gamma_{i,j}^{(\ell)} \in \mathbb{F}$ . Assume  $q \mid n$  without loss of generality, and form  $q \times q$  blocks of  $X, Y \in \mathbb{F}^{n \times n}$ . Then, each block  $X_{i,k}, Y_{k,j} \in \mathbb{F}^{\frac{n}{q} \times \frac{n}{q}}$ . Then, substitute in the blocks into the right-hand side of the above equation, and compute the coefficient of each  $z_{i,j}$  to compute  $Z_{i,j}$ . Recurse to compute products of the  $\frac{n}{q} \times \frac{n}{q}$  matrices. Each term corresponding to a rank 1 tensor indexed by  $\ell$  takes  $3q^2$  additions and one multiplication of  $\frac{n}{q} \times \frac{n}{q}$  matrices. This gives us the recurrence

$$T(n) = rT\left(\frac{n}{q}\right) + O(n^2) = O\left(n^{\log_q r}\right),$$

as desired.

Applying Theorem 2.2.1 from above, we have the following results.

# 

Note rank  $(\langle 2, 2, 2 \rangle) \leq 7$ , so we have an  $O(n^{\log_2 7})$ -time matrix multiplication algorithm.

# Corollary 2.2.2: Pan78]'s Bound

Note rank  $(\langle 70, 70, 70 \rangle) \leq 143640$ , so we have an  $O(n^{\log_{70} 143640})$ -time matrix multiplication algorithm.

# **Definition 2.2.4:** Rectangular Matrix Multiplication Exponent

The Rectangular Matrix Multiplication exponent  $\omega$  is

$$\omega(\textbf{a},\textbf{b},\textbf{c}) = \inf \left\{ \omega' > 0 : \forall \epsilon > 0, \exists \text{ an algorithm for multiplying } A \in \mathbb{R}^{n^a \times n^b} \text{ and } B \in \mathbb{R}^{n^b \times n^c} \text{ in } O\left(n^{\omega' + \epsilon} \text{ time}\right) \right\}.$$

# **Definition 2.2.5:** Dual Matrix Multiplication Exponent

The dual MATRIX MULTIPLICATION exponent lpha is

$$\alpha = \sup \{ \alpha' > 0 : \omega(1, 1, \alpha') = 2 \}.$$

**Remark.** By [WXXZ23],  $\alpha \geq$  0.321334. Also,  $\alpha = 1$  if and only if  $\omega = 2$ .

# Oraft: February 17, 2025

# 2.3 Lecture 8: February 6, 2025

# 2.3.1 Matrix Multiplication Verification

Consider the following problem.

Computational Problem 7 (MATRIX MULTIPLICATION VERIFICATION).

- Given matrices  $A, B, C \in \mathbb{R}^{n \times n}$ ,
- *Decide: AB* = *C*?

Naively, we immediately have an  $O(n^{\omega})$  algorithm by an easy reduction to MATRIX MULTIPLICATION.

Question. Can we do better?

Answer. Yes, we can!

We present a natural randomized algorithm, due to [Fre79].

# Algorithm 2.3.2 Freivalds' MATRIX MULTIPLICATION VERIFICATION

- 1: **procedure** Freivalds\_MMV(*A*, *B*, *C*)
- 2: Sample  $x \sim \{0, 1\}^n$  uniformly at random.
- 3: **return** ABx = Cx

 $\triangleright$  Compute ABx as A(Bx)

For correctness, let D = AB - C and observe that Dx = 0 if and only if ABx = Cx. If D = 0, then Dx = 0 with probability 1. If  $D \neq 0$ , D has a nonzero row d. Then,

$$\Pr_{x}[Dx=0] \leq \Pr_{x}[\langle d, x \rangle = 0] \leq \frac{1}{2}.$$

Algorithm 2.3.2 runs in  $O(n^2)$  time. Note that Freivalds' procedure uses n random bits. Naturally, we ask how to partially derandomize while still retaining  $o(n^\omega)$  time.

# 2.4 Lecture 9: February 11, 2025

# 2.4.1 Sparse Matrix Multiplication

Another natural question is how we can do better in the case of sparse matrices. Here, we follow [YZ05].

Computational Problem 8 (Sparse Matrix Multiplication).

- Given matrices  $A, B \in \mathbb{R}^{n \times n}$ , each with at most m nonzero entries,
- Find C = AB.

We have two critical observations that we rely on. First, let  $A, B \in \mathbb{R}^{n \times n}$  where A has columns  $a_i$  and B has rows  $b_i$ . Then,  $AB = \sum_{i=1}^n a_i b_i$ ; this is an outer product characterization of matrix multiplication. Also, for any permutation  $\pi$ ,  $\sum_{i=1}^n a_i b_i = \sum_{i=1}^n a_{\pi(i)} b_{\pi(i)}$ . We may compute this product using  $O\left(+n^2\right)$  [AB: todo].

We present the algorithm by [YZ05]. [AB: TODO!!!]

# 3

# NP-Hard Graph Problems

# 3.1 Lecture 10: February 13, 2025

# 3.1.1 Graph Coloring

We introduce the graph coloring problem.

Computational Problem 9 (GRAPH k-COLORING).

- Given a unweighted and undirected graph G = (V, E) and  $k \in \mathbb{Z}^+$ ,
- Decide: Does there exist an admissible k-coloring of G?

For now, take k = 3. Note that k = 2 corresponds to deciding whether the given graph is bipartite, easily done in polynomial time. There are several algorithms for determining whether G is 3-colorable. We describe certain techniques at a high level below.

# Algorithm 3.1.1 3-Coloring via a Reduction to 2-Coloring

Require: G = (V, E).

- 1: **procedure** 3-Color-Reduction-2-Color(*G*)
- 2: **for** each subset  $R \subseteq V$  with  $|R| \leq \frac{n}{3}$  **do**
- 3: Attempt to 2-color  $(V \setminus R, E)$  with GREEN and BLUE
- 4: **return** the coloring if successful.

The running time of Algorithm 3.1.1 is

$$\left(\sum_{i=1}^{\binom{n}{3}} \binom{n}{i}\right) \operatorname{poly}(n).$$

Then, we analyze the highest term and see that [AB: ...]

$$\binom{n}{\frac{n}{3}} =$$

#### Algorithm 3.1.2 3-Coloring via a Reduction to 2-Coloring

Require: G = (V, E).

- 1: **procedure** 3-Color-Reduction-2-Sat(*G*)
- 2: for  $v \in V$  do
- 3: Repeatedly randomly choose an assignment  $\ell(v)$  of two colors to v
- 4: Find a 3-coloring respecting  $\ell(v_1), \dots, \ell(v_n)$  by a reduction to 2-SAT.

The idea behind Algorithm 3.1.2 is to fix a legal 3-coloring  $c^*$  of G, and for each vertex, we can randomly select two colors from {Red, Green, Blue}. We want one of them to correspond with  $c^*$ . Then,

$$Pr[all \ n \ guesses \ agree \ with \ c^*] = \left(\frac{2}{3}\right)^n$$

**Lemma.** Suppose that we have guesses  $\ell(v) \in \{\text{Red}, \text{Green}, \text{Blue}\}^2$  that agree with  $c^*$  for all n vertices. Then, we can find a 3-coloring of G in poly(n) time.

*Proof.* Construct a 2-SAT instance with one variable  $x_i$  for each vertex  $v_i \in G$ . The values that can be assigned to  $x_i$  are in one-to-one correspondence with the colors in  $\ell(v_i)$ . Then, for each edge  $(v_i, v_j)$ , add either one or two 2-clauses, depending on  $|\ell(v_i) \cap \ell(v_j)|$ .

By our probabilistic analysis, and using a polynomial time 2-SAT algorithm [AB: cite: e.g., papadimitriou], we see that Algorithm 3.1.2 runs in  $O^*\left(\left(\frac{3}{2}\right)^n\right)$  time.

We now provide an algorithm for k-coloring in general. But first, consider the following theorems.

# Theorem 3.1.1: Fast Multiplication of Multivariate Polynomials

There exists an algorithm that multiplies polynomials  $p(x_1, ..., x_n)$  and  $q(x_1, ..., x_n)$  with degree at most d in each variable using  $O^*(2^n d)$  arithmetic operations.

*Proof.* The proof is by reduction to univariate polynomial multiplication and applying the fast Fourier transform.

# **Definition 3.1.1:** Independent Set Polynomial of a Graph

Define the independent set polynomial of G = ([n], E) as

$$p_G(x_1, ..., x_n) = \sum_{S \subseteq [n]} (\mathbf{1}_{S ext{ is an independent set}}) \prod_{i \in S} x_i.$$

# Theorem 3.1.2: A Necessary and Sufficient Condition for k-Colorability

The graph G = ([n], E) is k-colorable if and only if  $p_G(x_1, ..., x_n)^k$  contains the monomial  $z(x_1 \cdots x_n)$  for some  $z \in \mathbb{Z}^+$ .

*Proof.* A k-coloring of G is a partition of the vertices of G into k independent sets  $S_1, \ldots, S_k$ . If  $S_1, \ldots, S_k$  corresponds to a legal k-coloring, then  $p_G(x_1, \ldots, x_k)$  contains each of the k monomials  $\prod_{i \in S_j} x_i$ . So,  $p_G(x_1, \ldots, x_n)^k$  contains  $x_1 \cdots x_n$ . On the other hand, if  $p_G(x_1, \ldots, x_n)^k$  contains  $x_1 \cdots x_n$ , then  $p_G$  contains k disjoint monomials corresponding to a partition of the input graph into k independent sets.

The running time is  $O^*(2^n)$  to compute  $p_G$ , and  $O^*(2^n)$  to compute  $p_G(x_1, ..., x_k)^n$ .

# **Appendices**

# Bibliography

- [ADW<sup>+</sup>24] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication, 2024.
- [Alm21] Josh Alman. Algorithms for matrix multiplication, 2021.
- [CGI+16] Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, ITCS '16, page 261–270, New York, NY, USA, 2016. Association for Computing Machinery.
- [Fre79] Rūsiņš Freivalds. Fast probabilistic algorithms. 1979.
- [HMU01] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley series in computer science. Addison-Wesley, 2001.
- [IP01] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. volume 62, pages 367–375, 2001.
- [IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? In *Journal of Computer and System Sciences*, volume 63, pages 512–530, 2001.
- [MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2n steps. In *Discrete Applied Mathematics*, volume 10, pages 287–295, 1985.
- [Pan78] Victor Yakovlevich Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In 19th Annual Symposium on Foundations of Computer Science, pages 166–176, 1978.
- [RVW13] Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory* of Computing, STOC '13, page 515–524, New York, NY, USA, 2013. Association for Computing Machinery.
- [Sch99] Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 410. IEEE Computer Society, 1999.
- [Sip12] Michael Sipser. Introduction to the Theory of Computation. Cengage Learning, 2012.
- [Spi07] Daniel Spielman. Lecture notes on schöning's algorithm for 3-sat, April 2007.
- [Str69] Volker Strassen. Gaussian elimination is not optimal. Numer. Math., 13(4):354–356, 1969.

BIBLIOGRAPHY 28

[VW19] Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity, 2019.

- [Wil04] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365, 2004. Automata, Languages and Programming: Algorithms and Complexity (ICALP-A).
- [WXXZ23] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega, 2023.
- [YZ05] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.