**AI2615** 算法设计与分析 2020-2021 春季学期

# Lecture 4 – Fast Fourier Transform and Graph Algorithms

2021 年 3 月 19 日

*Lecturer:* 张驰豪 *Scribe:* 陶表帅

In this lecture, we will learn one more divide-and-conquer-based algorithm, *Fast Fourier Transform* (FFT), that computes the product of two polynomials in $O(d \log d)$ time, where $d$ is the degree of the two polynomials. We will then start to learn graph algorithms. In this lecture, we will learn algorithms to find all the connected components in a graph.

Before we proceed to Fast Fourier Transform, we will first go through some complex number basics.

## 1 Complex Number Basics

A *complex number* is a number that can be expressed as $a+bi$, where $a, b$ are two real numbers and $i$ is the *imaginary unit* which satisfies $i^2 = -1$. The set of all complex numbers is denoted by $\mathbb{C}$. For the complex number $z = a + bi$, $a$ is called the *real part* and $b$ is called the *imaginary part*. A complex number $a + bi$ can be expressed as a point in a 2-dimensional plane with coordinate $(a, b)$, or a vector $(a, b)$. The $x$-axis is called the *real axis* and the $y$-axis is called the *imaginary axis*.

A complex number can also be expressed by the *polar form* characterized by parameters $r \in \mathbb{R}_{\geq 0}$ and $\theta$:

$$z = r(\cos\theta + i \sin\theta),$$

where $r$ is the length of the vector $(a, b)$ and $\theta$ is the angle between the vector and the real axis. By Euler's formula, we have

$$z = r(\cos\theta + i \sin\theta) = r \cdot e^{i\theta}.$$

Given a complex number $z = a + bi$, its *complex conjugate*, denoted by $\overline{z}$, is given by

$$\overline{z} = a - bi.$$

In polar form, we have

$$\overline{z} = e^{-i\theta},$$

since $\overline{z} = r(\cos\theta - i \sin\theta) = r(\cos(-\theta) + i \sin(-\theta)) = e^{-i\theta}$.

### 1.1 Complex Matrices

Given two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$ with complex entries, their *inner product* is defined by

$$\langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i=1}^{n} \overline{a_i} b_i.$$

A vector $\mathbf{a}$ is called a *unit vector* if $\langle \mathbf{a}, \mathbf{a} \rangle = 1$. Two vectors $\mathbf{a}, \mathbf{b}$ are *orthogonal* if $\langle \mathbf{a}, \mathbf{b} \rangle = 0$, and they are *orthonormal* if $\langle \mathbf{a}, \mathbf{b} \rangle = 0$ and both $\mathbf{a}$ and $\mathbf{b}$ are unit vectors. A set of vectors form an *orthogonal set* if every pair of distinct vectors are orthogonal. A set of vectors form an *orthonormal set* if it is an orthogonal set and each vector is a unit vector. In other words, a set of vectors form an orthonormal set if every pair of distinct vectors are orthonormal.

A square matrix $A$ is an *orthonormal matrix* if all its columns are orthonormal vectors.[1] [2] If a matrix $A$ is orthonormal, all its rows are also orthonormal vectors.

Given a complex matrix $A \in \mathbb{C}^{n \times m}$, its *conjugate transpose*, denoted by $A^*$, is a $m \times n$ matrix obtained by first taking the transpose of $A$ and then taking the complex conjugate of each entry:

$$\left(A^*\right)_{i,j} = \overline{A_{j,i}}.$$

When $A$ is a real matrix, its conjugate transpose is just its transpose: $A^* = A^\top$.

The theorem below follows straightforwardly from the definition of orthonormal matrices.

**Theorem 1.** *If a square matrix $A$ is orthonormal, then $A$ is invertible and we have $A^{-1} = A^*$.*

## 2 Fast Fourier Transform for Polynomial Multiplication

**Problem 2.** Given two polynomials $p(x)$ and $q(x)$ with degree $d-1$, compute its product $r(x) = p(x)q(x)$.

We consider the word RAM model where integer additions and multiplications can be done in constant time.

We will write

$$p(x) = \sum_{i=0}^{d-1} a_i x^i \qquad \text{and} \qquad q(x) = \sum_{i=0}^{d-1} b_i x^i$$

in this section. Each polynomial is encoded by its coefficients. Therefore, $p(x)$ and $q(x)$ are encoded by $(a_0, a_1, \ldots, a_{d-1})$ and $(b_0, b_1, \ldots, b_{d-1})$ respectively.

The product of $p(x)$ and $q(x)$, $r(x) = p(x)q(x)$, is given by

$$r(x) = \sum_{i=0}^{2d-2} c_i x^i \qquad \text{where} \qquad c_i = \sum_{k=0}^{i} a_k b_{i-k}.$$

If we compute $r(x)$ directly using the equality above, this will takes time $O(d^2)$.

[1] Conventionally, an orthonormal matrix is also called an *orthogonal matrix*. In particular, when we say a matrix is orthogonal, we also require its columns being orthonormal vectors, not just being orthogonal vectors. This inconsistent usage of the word "orthogonal" in a set of vectors and in a square matrix may be misleading. Therefore, in this course, we will avoid saying a matrix is orthogonal, and we will always say it is orthonormal.

[2] In some textbooks, orthonormal matrix only refers to a real matrix, and a complex matrix satisfying the orthonormality is called a *unitary matrix*. We will not make such a distinction, and use the word "orthonormal matrix" for both real and complex matrices.

The problem of multiplying two $d$-bit integers can be viewed as a special case of the problem of multiplying two polynomials with degree $d-1$, if we take $x=2$ and restrict the coefficients of the polynomials to be binary. Algorithm 4 in the last lecture can be adapted to work here, and we have an algorithm with time complexity $O(d^{\log_2 3})$. In this lecture, we will present another divide-and-conquer-based algorithm *Fast Fourier Transform* (FFT) with time complexity $O(d \log d)$.
The supreme efficiency of FFT makes it embedded into hardware chips nowadays.

## 2.1 Polynomial Interpolation

A polynomial $p(x)$ of degree $d-1$ can *interpolate* $d$ points $(\alpha_0, p(\alpha_0)), (\alpha_1, p(\alpha_1)), \ldots, (\alpha_{d-1}, p(\alpha_{d-1}))$ that are on the curve of $p(x)$. The *interpolation theorem* below states that $d$ points interpolated by the polynomial $p(x)$ uniquely determine $p(x)$.

**Theorem 3** (Interpolation Theorem). *Given $d$ points $(x_0, y_0), (x_1, y_1), \ldots, (x_{d-1}, y_{d-1})$ such that $x_i \neq x_j$ for any $i \neq j$, there exists a unique polynomial $p(x)$ with degree at most $d-1$ such that $p(x_i) = y_i$ for each $i = 0, 1, \ldots, d-1$.*

*Proof.* For each $i = 0, 1, \ldots, d-1$, we have $y_i = \sum_{t=0}^{d-1} a_t x_i^t$, which can be expressed by the equation below

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{d-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{d-1} & x_{d-1}^2 & \cdots & x_{d-1}^{d-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix}. \tag{1}$$

Let $A$ be the square matrix on the right-hand side. $A$ is a *Vandermonde matrix* with determinant

$$|A| = \prod_{0 \leq i < j \leq d-1} (x_j - x_i).$$

Since $x_i \neq x_j$ for all $i \neq j$, $|A| \neq 0$, so $A$ is invertible. We have

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{bmatrix} = A^{-1} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{d-1} \end{bmatrix},$$

which yields unique $a_0, a_1, \ldots, a_{d-1}$. □

The algorithm FFT proceeds in two steps:

- Interpolation Step: We choose $2d-1$ distinct numbers $\alpha_0, \alpha_1, \ldots, \alpha_{2d-2}$ and compute the values of

$$p(\alpha_0)q(\alpha_0), p(\alpha_1)q(\alpha_1), \ldots, p(\alpha_{2d-2})q(\alpha_{2d-2}).$$

This gives us an interpolation

$$(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{2d-2}, r(\alpha_{2d-2}))$$

for the polynomial $r(x)$ we want to compute.

- Recovery Step: Recover the polynomial $r(x)$ by its interpolation obtained in the previous step.

## 2.2 Interpolation Step

Let $D = 2d - 1$. For simplicity, we assume $D$ is an integer power of 2, and let $n = \log_2 D$. We describe an algorithm that chooses $D$ distinct numbers $\alpha_0, \alpha_1, \ldots, \alpha_{D-1}$ and computes $p(\alpha_0), p(\alpha_1), \ldots, p(\alpha_{D-1})$. The same algorithm can be used to compute $q(\alpha_0), q(\alpha_1), \ldots, q(\alpha_{D-1})$. Since we are considering word RAM model, it takes $O(1)$ time to compute each $p(\alpha_i)q(\alpha_i)$. Thus, it takes $O(D)$ time to obtain the interpolation $(\alpha_0, r(\alpha_0)), (\alpha_1, r(\alpha_1)), \ldots, (\alpha_{D-1}, r(\alpha_{D-1}))$ of $r(x)$ by combining $p(\alpha_0), p(\alpha_1), \ldots, p(\alpha_{D-1})$ and $q(\alpha_0), q(\alpha_1), \ldots, q(\alpha_{D-1})$.

We suppose that the degree of $p(x)$ is $D-1$ instead of $d-1$. This can be viewed by setting $a_d = a_{d+1} = \cdots = a_{D-1} = 0$. This will not change the overall time complexity asymptotically since $D = O(d)$.

The naïve computation for each $p(\alpha_i) = \sum_{t=0}^{D-1} a_t \alpha_i^t$ requires $O(D)$ time, and computing all those $p(\alpha_i)$'s requires $O(D^2)$ time. This is no better than directly computing $r(x)$.

The naïve divide and conquer algorithm for computing each $p(\alpha_i)$ by the "left-right decomposition" $p(\alpha_i) = p_1(\alpha_i)\alpha_i^{\frac{D}{2}} + p_2(\alpha_i)$ has time complexity characterized by $T(D) = 2T(D/2) + O(1)$, which is $T(D) = O(D)$ by master theorem. This is no better than just directly compute $p(\alpha_i)$. Intuitively, there is no sophistication in this divide and conquer algorithm: if we break down this algorithm, it merely computes the $D-1$ additions in $p(\alpha_i) = \sum_{t=0}^{D-1} a_t \alpha_i^t$ by a different order.

The FFT algorithm considers a more clever way to decompose $p(\alpha_i)$. Instead of the "left-right decomposition", we consider the following "alternative decomposition":

$$p(\alpha_i) = p_e(x^2) + x \cdot p_o(x^2),$$

where

$$p_e(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{D-2} x^{\frac{D-2}{2}} \quad \text{and} \quad p_o(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{D-1} x^{\frac{D-2}{2}}.$$

Here, the subscripts $e$ and $o$ stand for "even" and "odd" respectively.

By choosing $\alpha_0, \alpha_1$ such that $\alpha_0 = -\alpha_1$, we have $p_e(\alpha_0^2) = p_e(\alpha_1^2)$, and we only need to compute $p_e$ once for $\alpha_0$ and $\alpha_1$. The same observation holds for $p_o$. By making $\alpha_{2i+1} = -\alpha_{2i}$ for each $i = 0, 1, \ldots, \frac{D}{2} - 1$, we only need to compute each of $p_e$ and $p_o$ for $\frac{D}{2}$ times.

Let $T(D)$ be the time complexity to compute the $D$ values $p(\alpha_0), p(\alpha_1), \ldots, p(\alpha_{D-1})$. Computing $p_e$ for $\frac{D}{2}$ times has time complexity $T(\frac{D}{2})$, and the same holds for $p_o$. To combine, we need to compute $p_e(\alpha_{2i}^2) +$

$\alpha_{2i} \cdot p_o(\alpha_{2i}^2)$ for each $i$, which requires $O(D)$ time. Therefore, we have $T(D) = 2T(\frac{D}{2}) + O(D)$, which implies $T(D) = O(D \log D)$. It seems that we are done. However, there is an important missing piece in the above analysis. This missing piece requires the introduction of complex numbers.

In the first level of recursion, we have chosen the $D$ values $\alpha_0, \alpha_1, \ldots, \alpha_{D-1}$ such that $\alpha_{2i+1} = -\alpha_{2i}$ for each $i$. In the second level of recursion, we are given $\frac{D}{2}$ values

$$\alpha_0^2, \quad \alpha_2^2, \quad \ldots, \quad \alpha_{\frac{D}{2}-1}^2$$

and we need to evaluate polynomials $p_e$ and $p_o$ at these values. To continue the divide and conquer process, we need to further make sure

$$\alpha_0^2 = -\alpha_2^2, \quad \alpha_4^2 = -\alpha_6^2, \quad \alpha_8^2 = -\alpha_{10}^2, \quad \cdots$$

This is impossible if $\alpha_i$'s are real numbers, and this is where we need the help from complex numbers.

To choose $\alpha_0, \alpha_1, \ldots, \alpha_{D-1}$ that makes the divide and conquer algorithm work, it is helpful to go through a small example. An example for $D = 8$ with 3 levels of recursions is shown in Fig. 1.

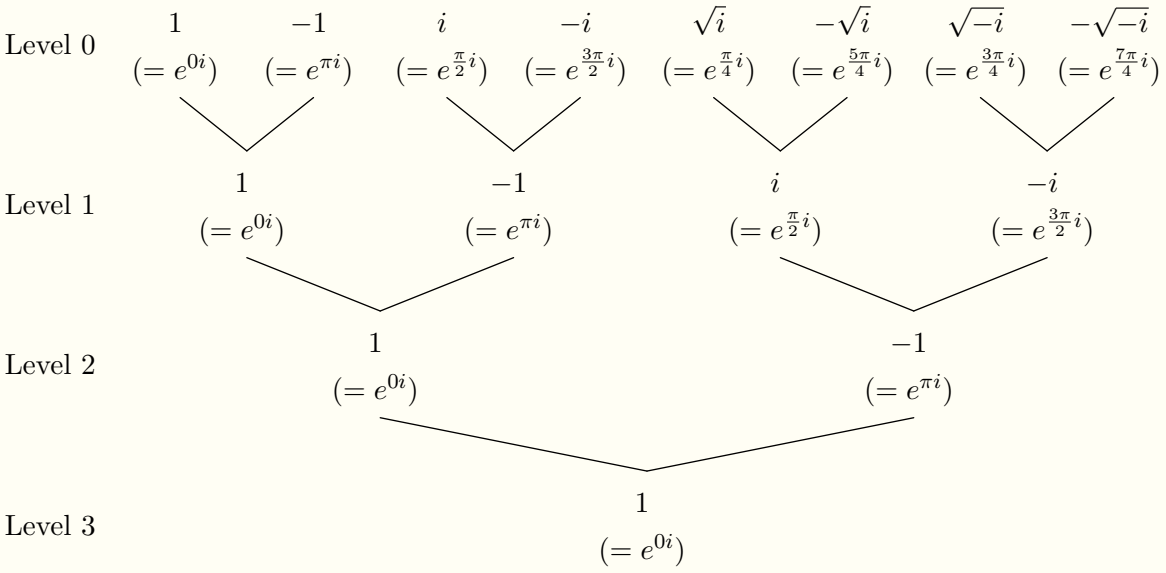


**Figure 1**: An example for $D = 8$: For the 8 values $\alpha_0, \alpha_1, \ldots, \alpha_7$ we choose at level 0, we need to make sure that, for every pair of values at any level, the square of the two values are equal. It is easy to find $\alpha_0, \alpha_1, \ldots, \alpha_7$ in a bottom-up way. In level 3, we set the value to 1. For the two values in level 2, we need to make sure the square of both values equals to 1. Therefore, the two values in level 2 are 1 and $-1$. For each value $\alpha$ at any intermediate level, there are always two numbers, namely $\sqrt{\alpha}$ and $-\sqrt{\alpha}$, whose squares are $\alpha$. When we have eventually computed all the 8 values in level 0, we will see that they are $e^{0i}, e^{\frac{\pi}{4}i}, e^{\frac{2\pi}{4}i}, \ldots, e^{\frac{7\pi}{4}i}$ that uniformly partition the unit circle into 8 arcs. These 8 values are paired such that every pair of numbers corresponds to a pair of vectors pointing at the opposite directions.

In general, FFT chooses $\alpha_t = \omega^t$ (for each $t = 0, 1, \ldots, D-1$) where $\omega = e^{\frac{2\pi}{D}i}$. The $D$ vectors corresponding to these $D$ values uniformly partition the unit circle into $D$ arcs. They are paired such that every pair of numbers corresponds to a pair of vectors pointing at the opposite directions. That is, each $\omega^t = e^{\frac{2\pi t}{D}i}$ with $t < \frac{D}{2}$ is paired with $\omega^{t+D/2} = e^{(\frac{2\pi t}{D}+\pi)i}$ (from the polar representations, it is easy to see that $\omega^{t+D/2}$ is obtained by rotating $\omega^t$ with $180^o$). The $\frac{D}{2}$ values computed at the next levels are

$$\omega^0, \omega^2, \omega^4, \ldots, \omega^{D-2},$$

and they correspond to the following pairs in the previous level

$$\left(e^{0i}, e^{\pi i}\right), \left(e^{\frac{\pi}{D}i}, e^{\frac{(D+1)\pi}{D}i}\right), \left(e^{\frac{2\pi}{D}i}, e^{\frac{(D+2)\pi}{D}i}\right), \ldots, \left(e^{\frac{(D-2)\pi}{2D}i}, e^{\frac{(2D-2)\pi}{2D}i}\right).$$

Figure 2 presents an example with $D = 8$ of how values of $\omega^0, \omega^1, \ldots, \omega^7$ are paired in all the three levels.

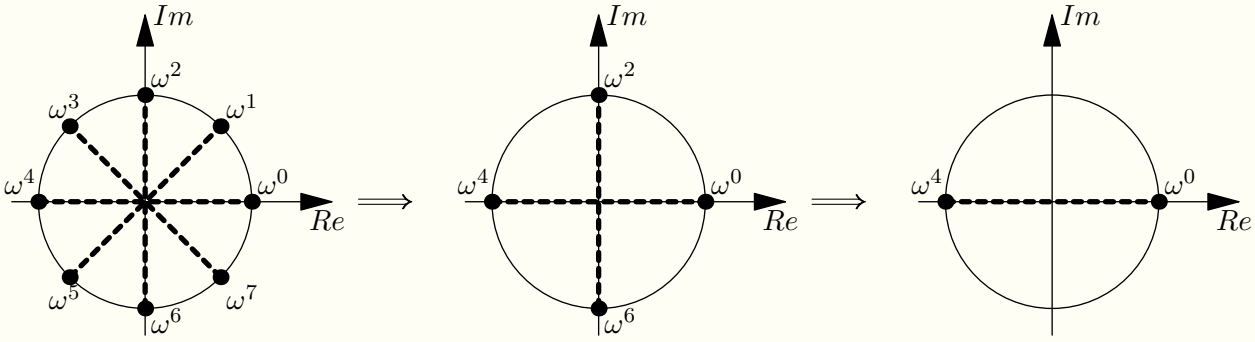


**Figure 2**: An example with $D = 8$ of how values are paired in all levels.

Putting together, Algorithm 1 describes the FFT algorithm for the interpolation step.

**Algorithm 1** The Fast Fourier Transform

FFT$(p, \omega)$: // $p$ is a polynomial of degree $D-1$, $\omega = e^{\frac{2\pi}{D}i}$, and FFT$(p, \omega)$ should output $D$ values: $p(\omega^0), p(\omega^1), \ldots, p(\omega^{D-1})$

1: if $\omega = 1$, **return** $p(1)$
2: $p_e(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{D-2} x^{\frac{D-2}{2}}$
3: $p_o(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{D-1} x^{\frac{D-2}{2}}$
4: $\left(p_e(\omega^0), p_e(\omega^2), \ldots, p_e(\omega^{D-2})\right) \leftarrow$FFT$(p_e, \omega^2)$
5: $\left(p_o(\omega^0), p_o(\omega^2), \ldots, p_o(\omega^{D-2})\right) \leftarrow$FFT$(p_o, \omega^2)$
6: **for** $t = 0, 1, \ldots, D-1$:
7: $\quad p(\omega^t) = p_e(\omega^{2t}) + \omega^t \cdot p_o(\omega^{2t})$
8: **endfor**
9: **return** $\left(p(\omega^0), p(\omega^1), \ldots, p(\omega^{D-1})\right)$

By applying Algorithm 1 again for $(q, \omega)$ and computing the product $p(\alpha_t)q(\alpha_t)$ for each $t = 0, 1, \ldots, D-1$, we obtain an interpolation $\left\{\left(\omega^t, r(\omega^t)\right) \mid t = 0, 1, \ldots, D-1; \omega = e^{\frac{2\pi}{D}i}\right\}$ for $r(x)$. As we have analyzed, the interpolation step requires $O(D \log D) = O(d \log d)$ time.

### 2.3 Recovery Step

We have obtained an interpolation $\left\{\left(\omega^t, r(\omega^t)\right) \mid t = 0, 1, \ldots, D-1; \omega = e^{\frac{2\pi}{D}i}\right\}$ for $r(x)$. It remains to recover (the coefficients of) the polynomial $r(x)$.

Let $A : \mathbb{C} \to \mathbb{C}^{D \times D}$ be a function that maps a complex number to a $D \times D$ complex matrix defined as follows:

$$A(x) \triangleq \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & x & x^2 & \cdots & x^{D-1} \\ 1 & x^2 & x^4 & \cdots & x^{2(D-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x^{D-1} & x^{2(D-1)} & \cdots & x^{(D-1)(D-1)} \end{bmatrix},$$

where $(A(x))_{i,j} = x^{(i-1)(j-1)}$. Similar to (1), the interpolation of $r(x) = \sum_{i=0}^{D-1} c_i x^i$ can be rewritten as

$$\begin{bmatrix} r(1) \\ r(\omega) \\ \vdots \\ r(\omega^{D-1}) \end{bmatrix} = A(\omega) \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{D-1} \end{bmatrix}. \tag{2}$$

**Proposition 4.** *The matrix $\frac{1}{\sqrt{D}} A(\omega)$ is orthonormal, where $\omega = e^{\frac{2\pi}{D}i}$.*

*Proof.* Let $\mathbf{c}_1, \ldots, \mathbf{c}_D$ be the $D$ column vectors corresponding to the $D$ columns of $\frac{1}{\sqrt{D}} A(\omega)$. For any $i, j \in \{1, \ldots, D\}$, we have

$$\langle \mathbf{c}_i, \mathbf{c}_j \rangle = \sum_{k=1}^{D} \frac{1}{D} \cdot \overline{\omega^{(k-1)(i-1)}} \cdot \omega^{(k-1)(j-1)} = \frac{1}{D} \sum_{k=1}^{D} \omega^{(k-1)(j-i)} = \begin{cases} 1 & \text{if } i = j \\ \frac{1}{D} \cdot \frac{1-\omega^{(j-i)D}}{1-\omega^{j-i}} = 0 & \text{if } i \neq j \end{cases},$$

where the last equality is due to $\omega^D = 1$. This shows that $\frac{1}{\sqrt{D}} A(\omega)$ is orthonormal. □

By Theorem 1, we have

$$A(\omega)^{-1} = \left(\sqrt{D} \cdot \frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{-1} = \frac{1}{\sqrt{D}} \left(\frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{-1} = \frac{1}{\sqrt{D}} \left(\frac{1}{\sqrt{D}} \cdot A(\omega)\right)^{*} = \frac{1}{D} A(\omega)^{*}.$$

Therefore, we have

$$\left(A(\omega)^{-1}\right)_{i,j} = \frac{1}{D} \cdot \overline{(A(\omega))_{j,i}} = \frac{1}{D} \cdot \omega^{-(i-1)(j-1)} = \frac{1}{D} \cdot \left(\omega^{-1}\right)^{(i-1)(j-1)},$$

which implies

$$A(\omega)^{-1} = \frac{1}{D} A\left(\omega^{-1}\right).$$

Putting it into (2), we have

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{D-1} \end{bmatrix} = \frac{1}{D} \cdot A\left(\omega^{-1}\right) \cdot \begin{bmatrix} r(1) \\ r(\omega) \\ \vdots \\ r(\omega^{D-1}) \end{bmatrix}. \tag{3}$$

Finally, to recover the coefficients $c_0, c_1, \ldots, c_{D-1}$ of the polynomial $r(x)$, we only need to apply Algorithm 1 by calling FFT$(s, \omega^{-1})$, where $s(x)$ is a polynomial of degree $D-1$ with coefficients $r(1), r(\omega), \ldots, r(\omega^{D-1})$. Specifically, it is easy to verify that

$$\left(\left(\omega^{-1}\right)^0, \left(\omega^{-1}\right)^1, \ldots, \left(\omega^{-1}\right)^{D-1}\right)$$

consists of the same $D$ values $(\omega^0, \omega^1, \ldots, \omega^{D-1})$ but with a clockwise order. In particular, $\omega^{-1}$ is also a $D$-th root of 1. This ensures that the pairings in all levels of the FFT algorithm work as before.

We put everything together and obtain Algorithm 2.

**Algorithm 2** Polynomial multiplication by FFT

MULTIPLY$(p, q)$: // $p, q$ are two polynomials with degree at most $d$

1: let $D$ be the smallest integer power of 2 such that $\frac{D}{2} - 1$ is an upper bound for the degrees of $p$ and $q$
2: let $\omega = e^{\frac{2\pi}{D} i}$
3: $(p_0, p_1, \ldots, p_{D-1}) \leftarrow$FFT$(p, \omega)$
4: $(q_0, q_1, \ldots, q_{D-1}) \leftarrow$FFT$(q, \omega)$
5: for each $t = 0, 1, \ldots, D-1$, compute $r_t \leftarrow p_t \cdot q_t$
6: let $s(x) = \sum_{t=0}^{D-1} r_t x^t$
7: $(c_0, c_1, \ldots, c_{D-1}) \leftarrow$FFT$(s, \omega^{-1})$
8: let $r(x) = \sum_{t=0}^{D-1} \frac{c_t}{D} x^t$
9: **return** $r$

It is easy to see that Algorithm 2 can be done in $O(D \log D) = O(d \log d)$, since it invokes function FFT for a constant number of times, and the remaining steps can be done in linear time $O(D)$.

**Theorem 5.** *Multiplying two polynomials of degree at most $d$ by Fast Fourier Transform can be done in $O(d \log d)$ time.*

## 3 Graph and Its Connected Components

We begin to study graph algorithms. We use $\{i, j\}$ and $(i, j)$ to represent an edge in an *undirected graph* and an edge in an *directed graph* respectively (as $\{\cdot\}$ is used for sets and $(\cdot)$ is used for ordered sets). We will

use $n = |V|$ and $m = |E|$ to denote the number of vertices and the number of edges in a graph $G = (V, E)$. Given a graph $G = (V, E)$ (directed or undirected), there are two common ways to encode its edges.

- Adjacency matrix: an $n \times n$ binary matrix such that $A_{i,j} = 1$ if $(i, j) \in E$ (when $G$ is directed) or $\{i, j\} \in E$ (when $G$ is undirected). Notice that the adjacency matrix for an undirected graph is always symmetric.
- Adjacency list: each vertex is associated with a linked list that stores all its neighbors.

Storing edges requires $\Theta(n^2)$ space if an adjacency matrix is used, and it requires $\Theta(m + n)$ space if an adjacency list is used. An adjacency list requires less space. However, some operations such as finding an edge and deleting an edge that can be done in $O(1)$ time in an adjacency matrix may have time complexity $\Theta(n)$ in an adjacency list.

## 3.1 Connectivity and Connected Components

**Definition 6.** Given an undirected graph $G = (V, E)$, a *connected component* is an induced subgraph $G' = (V', E')$ such that there is a path between every pair of vertices in $G'$ and there is no path from any vertex $u \in V'$ and any vertex $v \in V \setminus V'$. An undirected graph with only one connected component is *connected*.

**Definition 7.** Given a directed graph $G = (V, E)$, a *strongly connected component* is an induced subgraph $G' = (V', E')$ such that there is a path between every ordered pair of vertices in $G'$ and there do not exist two vertices $u \in V'$ and $v \in V \setminus V'$ such that $u$ is reachable from $v$ and $v$ is reachable from $u$. A directed graph with only one connected component is *strongly connected*.

For a strongly connected component in a directed graph, we require that there is a path between every *ordered* pair of vertices in it. In particular, $(u, v)$ and $(v, u)$ are two different ordered pairs, and there is a path from $u$ to $v$ and a path from $v$ to $u$ if $u$ and $v$ are in the same strongly connected component. If we contract each strongly connected component to a single vertex, we get a *meta-graph* that is a *directed acyclic graph* (DAG). See Figure 3 for an example about strongly connected components of a directed graph and the meta-graph.
In this section, we will study algorithms that count the number of connected components (strongly connected components) in an undirected graph (directed graph). Notice that the (strong) connectivity of a graph can be checked by checking if the number of its (strongly) connected components is exactly 1.

### 3.1.1 Connectivity for Undirected Graphs

**Problem 8.** Given an undirected graph $G = (V, E)$, count the number of connected components of it.

We will use a subroutine EXPLORE($u$) that searches all vertices that are in the same connected component containing vertex $u$. A global Boolean vector "visited" of length $n$ has been initialized such that visited[$v$]

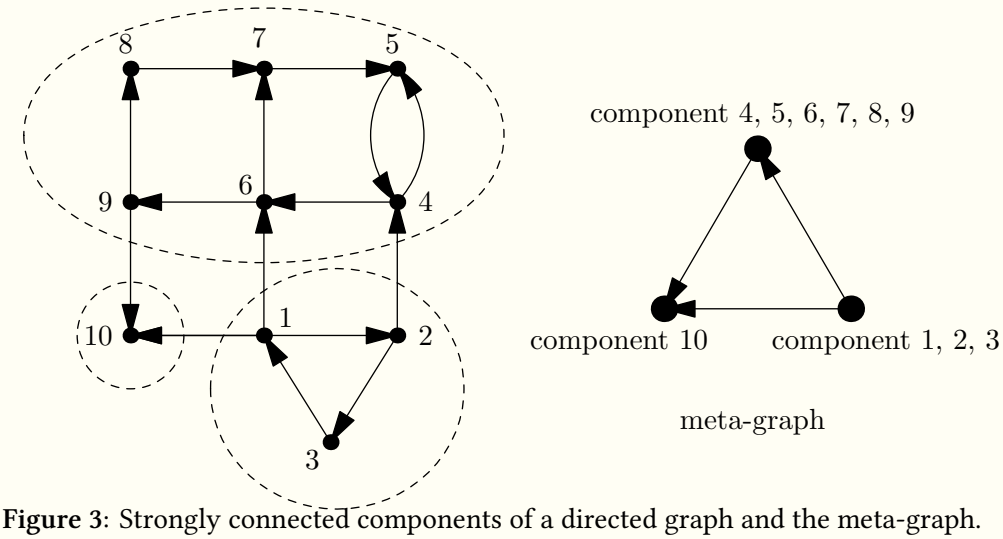


**Figure 3**: Strongly connected components of a directed graph and the meta-graph.

is set to **false** for each $v \in V$ indicating $v$ has not been visited yet. The search algorithm presented in Algorithm 3 is called a *depth-first-search* (DFS).

**Algorithm 3** Depth-First-Search

EXPLORE($u$): // a global Boolean vector "visited" of length $n$ has been initialized to **false**

1: visited[$u$] = **true**
2: **for** each $v \in N(u)$: // $N(u)$ denotes the set of neighbors of $u$
3: **if** visited[$v$] = **false**: EXPLORE($v$)
4: **endfor**

To check if an undirected graph $G = (V, E)$ is connected, we pick an arbitrary vertex $u$ and implement EXPLORE($u$). If visited[$v$] is **true** for each vertex $v \in V$, then $G$ is connected. Otherwise, $G$ is not.
The time complexity for this algorithm depends on the data structure we use to store the edges. If an adjacency matrix is used, the time complexity is $O(n^2)$; if an adjacency list is used, the time complexity is $O(m + n)$.
We leave it as an exercise to generalize this algorithm to count the number of connected components in an undirected graph.

### 3.1.2 Count Number of Strongly Connected Components in Directed Graphs

**Problem 9.** Given a directed graph $G = (V, E)$, count the number of strongly connected components of it.

Direct application of the algorithm in the previous section will not work here. Using the graph in Figure 3 as an example, if we start at vertex 1 and implement EXPLORE(1), then all the vertices in the graph will be

reachable from 1. This will lead to the wrong conclusion that the graph has only one strongly connected component.

**Definition 10.** Given a directed graph $G = (V, E)$, a vertex $u$ is a *source* if $(v, u) \notin E$ for any $v \in V$, and a vertex $u$ is a *sink* if $(u, v) \notin E$ for any $v \in V$.

**Theorem 11.** *A directed acyclic graph always has at least one source and at least one sink.*

**Exercise 12.** Prove Theorem 11.

We need to start from a vertex $u$ such that the strongly connected component containing it corresponds to a *sink* in the meta-graph. If we implement EXPLORE($u$) on such a vertex $u$, then all the vertices reachable from $u$ form a strongly connected component. Suppose for the sake of contradiction that there is a vertex $v$ that is reachable from $u$ and $v$ is in a different strongly connected component. In the meta-graph, there is a path from the vertex $C_u$ representing the strongly connected component containing $u$ to the vertex $C_v$ representing the strongly connected component containing $v$. It then follows that $C_u$ cannot be a sink, which contradicts to our assumption.

This observation gives us an iterative algorithm for Problem 9. We iteratively find a vertex $u$ such that the strongly connected component containing it corresponds to a *sink* in the meta-graph, find the strongly connected component that contains $u$, and then remove this component from the graph. The algorithm terminates when all the strongly connected components are found. Using the graph in Figure 3 as an example again, vertex 10 is in a strongly connected component that is a sink in the meta-graph, so we first implement EXPLORE(10) and find the strongly connected component that contains the single vertex 10; after removing vertex 10, the strongly connected component with vertices $4, 5, 6, 7, 8, 9$ is a sink in the meta-graph, so we shall find one of these 6 vertices, say, vertex 4, and implement EXPLORE(4), which will find this strongly connected component; finally, one of vertices $1, 2, 3$ will be found and the strongly connected component with these three vertices will be found.

Now, the only remaining problem is to find a vertex such that the strongly connected component containing it corresponds to a sink in the meta-graph. Algorithm 3 can be modified to do this. Algorithm 4 does a *post-order tree traversal* for the DFS tree. Other than the Boolean vector "visited", we also need a global integer vector "label" and an integer variable num that is initialized to 0.

We iteratively apply Algorithm 4 until all vertices are visited. This is described in Algorithm 5.

If edges are stored by an adjacency list, Algorithm 5 runs in time $O(m+n)$, since each edge $(u, v)$ is visited exactly once at EXPLORE2($u$). We have the following proposition after the execution of Algorithm 5.

**Proposition 13.** *For any two strongly connected components $C_1$ and $C_2$ such that there is a path from $C_1$ to $C_2$ in the meta-graph, we have*

$$\max_{i \in C_1} \text{label}[i] > \max_{j \in C_2} \text{label}[j].$$

**Algorithm 4** Depth-First-Search

EXPLORE2($u$): // the integer variable "num" is initialized to 0.

1: visited[$u$] = **true**
2: **for** each $v \in N(u)$:
3: **if** visited[$v$] = **false**: EXPLORE($v$)
4: **endfor**
5: label[$u$]←num++

**Algorithm 5** Label all vertices

1: set visited[$u$] = **false** for each $u \in V$
2: set num = 0
3: **while** there exist $u$ with visited[$u$] = **false**, **do**
4: EXPLORE2($u$)
5: **endwhile**

*Proof.* We consider two cases. If Algorithm 5 visits component $C_2$ before component $C_1$, then all vertices in $C_2$ will be labeled and vertices $C_1$ will only be labeled in a future while-loop iteration. Since the integer num is strictly increasing throughout algorithm 5, the first vertex visited in $C_1$ will have a higher label than any vertices in $C_2$. On the other hand, if $C_1$ gets visited first, the first vertex visited in $C_1$ will be labeled only when all vertices that are reachable from this vertex are labeled, which includes all vertices in $C_2$, in which case the proposition also follows. □

By Proposition 13, after executing Algorithm 5, the vertex with the highest label must be in a strongly connected component that corresponds to a *source* in the meta-graph. By reverse all edges in $G$, we can use Algorithm 5 to find a sink.

Finally, we can find all the strongly connected components in $G$ by iteratively find a vertex $u$ in a strongly connected component corresponding to a sink in the meta-graph, find the component containing $u$ by Algorithm 3, and remove this component from the graph. Notice that Algorithm 5 only needs to be executed once: when we find and remove a strongly connected component, we only need to find a vertex in the remainder graph that has the highest label without executing Algorithm 5 again. The overall time complexity for finding all the strongly connected components is therefore $O(m+n)$.

### 3.2 2SAT

In this section, we show that the problem 2SAT can be reduced to the problem of finding strongly connected components in a directed graph.

**Problem 14** (2SAT). Given a 2-cnf Boolean formula, decide if it has a satisfying assignment.

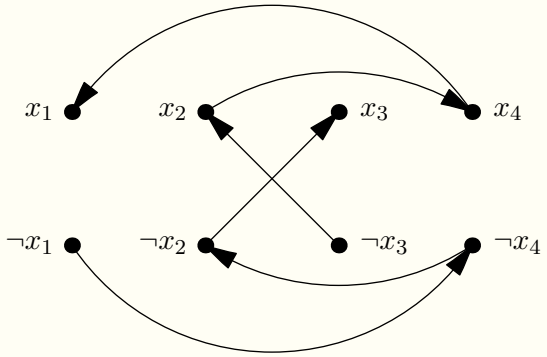


**Figure 4**: The graph corresponding to the formula $(x_2 \vee x_3) \wedge (x_1 \vee \neg x_4) \wedge (\neg x_2 \vee x_4)$.

A single clause $(x_i \vee x_j)$ in a 2-cnf formula is equivalent to $(\neg x_i \rightarrow x_j)$, or its contra-positive $(\neg x_j \rightarrow x_i)$. This simple observation provides us a natural way to construct a directed graph with $2n$ vertices for a 2-cnf formula with $n$ variables. For each variable $x_i$, we construct two vertices $x_i$ and $\neg x_i$ in the directed graph. For each term $(x_i \vee x_j)$, we construct two directed edges $(\neg x_i, x_j)$ and $(\neg x_j, x_i)$. An example of this construction is shown in Figure 4.

Naturally, an assignment to the 2-cnf formula corresponds to a choice of $n$ vertices in the graph such that exactly one vertex is chosen from each vertex pair $\{x_i, \neg x_i\}$. Moreover, if we choose a vertex, we have to choose all the other vertices that are reachable from it. This is easy to see if we view directed edges as logical implications: if we choose a vertex that corresponds to the assignment of a single variable, we need to assign the values for many other variables implied by this assignment. For example, if we choose vertex $\neg x_1$ in Figure 4, we have to choose $\neg x_4$ as well, since the clause $(x_1 \vee \neg x_4)$ in the formula is equivalent to $(\neg x_1 \rightarrow \neg x_4)$. For the similar reason, after $\neg x_4$ is chosen, we have to choose $\neg x_2$ and then $x_3$.

**Proposition 15**. *Given a 2-cnf formula $\phi$ and a directed graph $G$ representing it, $\phi$ has a satisfying assignment if and only if $x_i$ and $\neg x_i$ are not in the same strongly connected component for all $i = 1, \ldots, n$.*

*Proof.* To show the only-if direction, we need to show that $\phi$ being satisfiable implies each pair of $x_i$ and $\neg x_i$ are not in the same strongly connected component. We prove the contra-positive of this statement. Suppose there exists a pair of vertices $x_i$ and $\neg x_i$ that are in the same strongly connected component. There is a path from $x_i$ to $\neg x_i$ and there is a path from $\neg x_i$ to $x_i$. We know that exactly one of $x_i$ and $\neg x_i$ needs to be chosen. However, this is impossible, as choosing either one would require the choice of the other (remember that we need to choose all vertices that are reachable from a chosen vertex). Thus, $\phi$ should not have been satisfiable, as we cannot assign both **true** and **false** to $x_i$.

The proof for the if direction is left as an exercise. □

**Exercise 16**. Complete the proof for Proposition 15.

Proposition 15 straightforwardly provides us an algorithm for Problem 14. We construct the directed

graph, use algorithms in the previous section to identify all its strongly connected components, and check if there exists $i \in \{1, \ldots, n\}$ such that $x_i$ and $\neg x_i$ are in the same component. It is easy to check that the time complexity of this algorithm is $O(m+n)$, where $n$ is the number of variables and $m$ is the number of clauses.