Abstract

The class will focus on two themes: linear algebra and probability, and their many applications in algorithm design. We will cover a number of classical examples, including: fast matrix multiplication, FFT, error correcting codes, cryptography, efficient data structures, combinatorial optimization, routing, and more. We assume basic familiarity (undergraduate level) with linear algebra, probability, discrete mathematics and graph theory.
# Satisfiability

## 7.1 2-SAT

## 7.2 3-SAT

# Hash functions: the power of pairwise independence

## 8.1 Pairwise independent bits

## 8.2 Application: de-randomized MAXCUT

## 8.3 Optimal sample size for pairwise independent bits

## 8.4 Hash functions with large ranges

## 8.5 Application: collision free hashing

## 8.6 Efficient dictionaries: storing sets efficiently

## 8.7 Bloom filters

# Min cut

## 9.1 Karger’s algorithm

## 9.2 Improving the running time

# Routing

## 10.1 Deterministic routing is bad

## 10.2 Solution: randomized routing

# Expander graphs

## 11.1 Edge expansion

## 11.2 Spectral expansion

## 11.3 Cheeger inequality

## 11.4 Random walks mix fast

## 11.5 Random walks escape small sets

## 11.6 Randomness efficient error reduction in randomized algorithms
0 Preface: Mathematical background

0.1 Fields
A field is a set \( F \) endowed with two operations: addition and multiplication. It satisfies the following conditions:

- **Associativity:** \((x + y) + z = x + (y + z)\) and \((xy)z = x(yz)\).
- **Commutativity:** \(x + y = y + x\) and \(xy = yx\).
- **Distributivity:** \(x(y + z) = xy + xz\).
- **Unit elements \((0,1)\):** \(x + 0 = x\) and \(x1 = x\).
- **Inverse:** If \(x \neq 0\) then there exists \(1/x\) such that \(x(1/x) = 1\).

You probably know many infinite fields: the real numbers \( \mathbb{R} \), the rationals \( \mathbb{Q} \) and the complex numbers \( \mathbb{C} \). For us, the most important fields will be finite fields, which have a finite number of elements.

An example is the binary field \( F_2 = \{0, 1\} \), where addition corresponds to XOR and multiplication to AND. This is an instance of a more general example, of prime finite fields. Let \( p \) be a prime. The field \( F_p \) consists of the elements \( \{0, 1, \ldots, p - 1\} \), where addition and multiplication are defined modulo \( p \). One can verify that it is indeed a field. The following fact is important, but we will not prove it.

**Fact 0.1.** If a finite field \( F \) has \( q \) elements, then \( q \) must be a prime power. For any prime power there is exactly one finite field with \( q \) elements, which is called the finite field of order \( q \), and denoted \( F_q \).

0.2 Polynomials
Univariate polynomials over a field \( F \) are given by

\[
f(x) = \sum_{i=0}^{n} f_i x^i.
\]

Here, \( x \) is a variable which takes values in \( \mathbb{F} \), and \( f_i \in \mathbb{F} \) are constants, called the coefficients of \( f \). We can evaluate \( f \) at a point \( a \in \mathbb{F} \) by plugging \( x = a \), namely

\[
f(a) = \sum_{i=0}^{n} f_i a^i.
\]

The degree of \( f \) is the maximal \( i \) such that \( f_i \neq 0 \).
Multi-variate polynomials are defined in the same way: if $x_1, \ldots, x_d$ are variables that

$$f(x_1, \ldots, x_d) = \sum_{i_1, \ldots, i_d} f_{i_1, \ldots, i_d}(x_1)^{i_1} \cdots (x_d)^{i_d}$$

where $i_1, \ldots, i_d$ range over a finite subset of $\mathbb{N}^d$. The total degree of $f$ is the maximal $i_1 + \cdots + i_d$ for which $f_{i_1, \ldots, i_d} \neq 0$.

### 0.3 Matrices

An $n \times m$ matrix over a field $F$ is a 2-dimensional array $A_{i,j}$ for $1 \leq i \leq n$, $1 \leq j \leq m$. If $A$ is an $n \times m$ matrix, $B$ a $m \times \ell$ matrix, then their product is the $n \times \ell$ matrix given by

$$(AB)_{i,j} = \sum_{k=1}^{m} A_{i,k}B_{j,k}.$$

If $A$ is an $n \times n$ matrix, its determinant is

$$\det(A) = \sum_{\pi \in S_n} (-1)^{\text{sign}(\pi)} \prod_{i=1}^{n} A_{i,\pi(i)}.$$

Here, $\pi$ ranges over all permutations of $\{1, \ldots, n\}$.

**Fact 0.2.** If $A$ is an $n \times n$ matrix, then $\det(A) = \det(A^T)$, where $A^T$ is the transpose of $A$.

**Fact 0.3.** If $A, B$ are $n \times n$ matrices, then $\det(AB) = \det(A) \det(B)$.

### 0.4 Probability

In this class, we will only consider discrete distributions and discrete random variables, which take a finite number of possible values. Let $X$ be a random variable, such that $\Pr[X = x_i] = p_i$ where $x_i \in \mathbb{R}$, $p_i \geq 0$ and $\sum p_i = 1$. Its expectation (average) is

$$\mathbb{E}[X] = \sum_i p_i x_i$$

and its variance is

$$\text{Var}[X] = \mathbb{E} \left[ (X - \mathbb{E}[X])^2 \right] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

If $X, Y$ are any two random variables then $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$; if they are independent then also $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$ (in fact, it suffices if $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$ for that to hold). If $X, Y$ are joint random variables, let $X|Y = y$ be the marginal random variable of $X$ conditioned on $Y = y$. The conditional expectation $\mathbb{E}[X|Y]$ is a random variable which depends on $Y$. We have the useful formula

$$\mathbb{E}\left[ \mathbb{E}[X|Y] \right] = \mathbb{E}[X].$$

We will frequently use the following two common bounds.
Claim 0.4 (Markov inequality). Let \( X \geq 0 \) be a random variable. Then
\[
\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}.
\]

Proof. Let \( S = \{ i : x_i \geq a \} \). Then
\[
\mathbb{E}[X] = \sum p_i x_i \geq \sum p_i x_i \geq \sum p_i a = \Pr[X \in S]a = \Pr[X \geq a]a.
\]

Claim 0.5 (Chebychev inequality). Let \( X \in \mathbb{R} \) be a random variable where \( \mathbb{E}[X^2] < \infty \). Then
\[
\Pr \left[ |X - \mathbb{E}[X]| \geq a \right] \leq \frac{\text{Var}(X)}{a^2}.
\]

Proof. Let \( Y = |X - \mathbb{E}[X]|^2 \). Then \( \mathbb{E}[Y] = \text{Var}(X) \) and, by applying Markov inequality to \( Y \) (note that \( Y \geq 0 \)) we have
\[
\Pr \left[ |X - \mathbb{E}[X]| \geq a \right] = \Pr[Y^2 \geq a^2] \leq \frac{\mathbb{E}[Y^2]}{a^2} = \frac{\text{Var}(X)}{a^2}.
\]

We will also need tail bounds for the sum of many independent random variables. This is given by the Chrenoff bounds. We state two versions of these bounds: one for absolute (additive) error and one for relative (multiplicative) error.

Theorem 0.6 (Chernoff bounds). Let \( Z_1, \ldots, Z_n \in \{0, 1\} \) be independent random variables. Let \( Z = \sum Z_i \) and \( \mu = \mathbb{E}[Z] \). Then for any \( \lambda > 0 \) it holds that

(i) Absolute error:
\[
\Pr \left[ |Z - \mathbb{E}[Z]| \geq \lambda n \right] \leq 2 \exp(-2\lambda^2 n).
\]

(ii) Relative error:
\[
\Pr \left[ Z \geq (1 + \lambda)\mu \right] \leq \exp \left( -\frac{\lambda^2 \mu}{2 + \lambda} \right).
\]
1 Matrix multiplication

Matrix multiplication is a basic primitive in many computations. Let \( A, B \) be \( n \times n \) matrices. Their product \( C = AB \) is given by

\[
C_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.
\]

The basic computational problem is how many operations (additions and multiplications) are required to compute \( C \). Implementing the formula above in a straightforward manner requires \( O(n^3) \) operations. The best possible is \( 2n^2 \), which is the number of inputs. The matrix multiplication exponent, denoted \( \omega \), is the best constant such that we can multiply two \( n \times n \) matrices in \( O(n^\omega) \) operations (to be precise, \( \omega \) is the infimum of these exponents). As we just saw, \( 2 \leq \omega \leq 3 \). To be concrete, we will consider matrices over the reals, but this can be defined over any field.

Open Problem 1.1. What is the matrix multiplication exponent?

The first nontrivial algorithm was by Strassen [Str69] in 1969, who showed that \( \omega \leq \log_2 7 \approx 2.81 \). Subsequently, researchers were able to improve the exponent. The best results to date are by Le Gall [LG14] who get \( \omega \leq 2.373 \). Here, we will only describe Strassen’s result, as well as general facts about matrix multiplication. A nice survey on matrix multiplication, describing most of the advances so far, can be found in the homepage of Yuval Filmus: http://www.cs.toronto.edu/~yuvalf/.

1.1 Strassen’s algorithm

The starting point of Strassen's algorithm is the following algorithm for multiplying \( 2 \times 2 \) matrices.

1. \( p_1 = (a_{1,1} + a_{2,2})(b_{1,1} + b_{2,2}) \)
2. \( p_2 = (a_{2,1} + a_{2,2})b_{1,1} \)
3. \( p_3 = a_{1,1}(b_{1,2} - b_{2,2}) \)
4. \( p_4 = a_{2,2}(b_{2,1} - b_{1,1}) \)
5. \( p_5 = (a_{1,1} + a_{1,2})b_{2,2} \)
6. \( p_6 = (a_{2,1} - a_{1,1})(b_{1,1} + b_{1,2}) \)
7. \( p_7 = (a_{1,2} - a_{2,2})(b_{2,1} + b_{2,2}) \)
8. \( c_{1,1} = p_1 + p_4 - p_5 + p_7 \)
9. \( c_{1,2} = p_3 + p_5 \)
10. \( c_{2,1} = p_2 + p_4 \)
11. \( c_{2,2} = p_1 - p_2 + p_3 + p_6 \)

It can be checked that this program uses 7 multiplications and 18 additions/subtractions, so 25 operations overall. The naive implementation of multiplying two \( 2 \times 2 \) matrices requires 8 multiplications and 4 additions, so 12 operations overall. The main observation of Strassen is that really only the number of multiplications matter.

In order to show it, we will convert Strassen’s algorithm to a normal form. In such a program, we first compute some linear combinations of the entries of \( A \) and the entries of \( B \), individually. We then multiply them, and take linear combinations of the results to get the entries of \( C \). We define it formally below.

**Definition 1.2 (Normal form).** A normal-form program for computing matrix multiplication, which uses \( M \) multiplications, has the following form:

(i) For \( 1 \leq i \leq M \), compute linear combinations \( \alpha_i \) of the entries of \( A \).

(ii) For \( 1 \leq i \leq M \), compute linear combinations \( \beta_i \) of the entries of \( B \).

(iii) For \( 1 \leq i \leq M \), Compute the product \( p_i = \alpha_i \cdot \beta_i \).

(iv) For \( 1 \leq i, j \leq n \), compute \( c_{i,j} \) as a linear combination of \( p_1, \ldots, p_M \).

**Lemma 1.3.** Any program for computing matrix multiplication, which uses \( M \) multiplications (and any number of additions), can be converted to a normal form with at most \( 2M \) multiplications.

Note that in the normal form, the program computes \( 2M \) multiplications and \( O(Mn^2) \) additions.

**Proof.** First, we can convert any program for computing matrix multiplication to a straight-line program. In such a program, every step computes either the sum or product of two previously computed variables. In our case, let \( z_1, \ldots, z_N \) be the values computed by the program. The first \( 2n^2 \) are the inputs: \( z_1, \ldots, z_{n^2} \) are the entries of \( A \) (in some order), and \( z_{n^2+1}, \ldots, z_{2n^2} \) are the entries of \( B \). The last \( n^2 \) variables are the entries of \( C \) which we wish to compute. Every intermediate variable \( z_i \) is either:

- A linear combination of two previously computed variables: \( z_i = \alpha z_j + \beta z_k \), where \( j, k < i \) and \( \alpha, \beta \in \mathbb{R} \).

- A product of two previously computed variables: \( z_i = z_j \cdot z_k \), where \( j, k < i \).

In particular, note that each \( z_i \) is some polynomial of the inputs \( \{a_{i,j}, b_{i,j} : 1 \leq i, j \leq n\} \). Next, we show that as the result is bilinear in the inputs, we can get the computation to respect that structure. We decompose \( z_t(A, B) \) as follows:

\[
z_t(A, B) = a_t + b_t(A) + c_t(B) + d_t(A, B) + e_t(A, B),
\]

where
• $a_t$ is a constant (independent of the inputs)
• $b_t(A)$ is a linear combination of the entries of $A$
• $c_t(B)$ is a linear combination of the entries of $B$
• $d_t(A, B)$ is a bi-linear combination of the entries of $A, B$; namely, a linear combination of $\{a_{i,j}b_{k,\ell} : 1 \leq i, j, k, \ell \leq n\}$
• $e_t(A, B)$ is the rest. Namely, any monomial which is quadratic in $A$ or in $B$.

Note that inputs have only a linear part (either $b_t$ or $c_t$); and that outputs have only a bilinear part ($d_t$). The main observation is that we can compute all of the linear and bilinear parts directly, without computing $e_t$ at all, via a straight line program.

(i) Sums: If $z_t = z_i + z_j$ with $i, j < t$, then

• $a_t = a_i + a_j$.
• $b_t(A) = b_i(A) + b_j(A)$.
• $c_t(B) = c_i(B) + c_j(B)$.
• $d_t(B) = d_i(B) + d_j(B)$.

The same holds for general linear combinations, $z_t = \alpha z_i + \beta z_j$.

(ii) Multiplications: If $z_t = z_i \cdot z_j$ with $i, j < t$, then

• $a_t = a_i \cdot a_j$.
• $b_t(A) = a_i \cdot b_j(A) + a_j \cdot b_i(A)$.
• $c_t(B) = a_i \cdot c_j(B) + a_j \cdot c_i(B)$.
• $d_t(A, B) = a_i \cdot d_j(A, B) + a_j \cdot d_i(A, B) + b_i(A)c_j(B) + b_j(A)c_i(B)$.

Note that $c_t$ are constants independent of the inputs. So, the only actual multiplications we do is in computing $b_i(A)c_j(B)$ and $b_j(A)c_i(B)$. To get to a normal form, we can compute:

• Linear combinations of the entries of $A$: $a_t(A)$ for $1 \leq t \leq N$.
• Linear combinations of the entries of $B$: $b_t(B)$ for $1 \leq t \leq N$.
• Products: $b_i(A)c_j(B)$ and $b_j(A)c_i(B)$ whenever $z_t = z_i \cdot z_j$.
• $d_t(A, B)$ are linear combination of these products, and previously computed $d_i(A, B), d_j(A, B)$ for $i, j < t$.
• Output: linear combination of $d_t(A, B)$ for $1 \leq t \leq N$.

Note that we only need the linear combinations which enter the multiplication gates, which gives the lemma.
Next, we show how to use matrix multiplication programs in normal form to compute products of large matrices. This will show that only the number of multiplications matter.

**Theorem 1.4.** If two $m \times m$ matrices can be computed using $M = m^\alpha$ multiplications (and any number of additions) in a normal form, then for any $n \geq 1$, any two $n \times n$ matrices can be multiplied using only $O((mn)^\alpha \log(mn))$ operations.

So for example, Strassen’s algorithm is an algorithm in normal form which uses 7 multiplications to multiply two $2 \times 2$ matrices. So, any two $n \times n$ matrices can be multiplied using $O(n^{\log_2 7} \log n^{O(1)}) \leq O(n^{2.81+\varepsilon})$ operations for any $\varepsilon > 0$. So, $\omega \leq \log_2 7 \approx 2.81$.

**Proof.** Let $T(n)$ denote the number of operations required to compute the product of two $n \times n$ matrices. We assume that $n$ is a power of $m$, by possible increasing it to the smallest power of $m$ larger than it. This might increase $n$ to at most $nm^s$. Now, the main idea is to compute it recursively. We partition an $n \times n$ matrix as an $m \times m$ matrix, whose entries are $(n/m) \times (n/m)$ matrices. Let $C = AB$ and let $A_{i,j}, B_{i,j}, C_{i,j}$ be these sub-matrices of $A, B, C$, respectively, where $1 \leq i, j \leq m$. Then, observe that (as matrices) we have

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k}B_{k,j}.$$ 

We can apply any algorithm for $m \times m$ matrix multiplication in normal form to compute $\{C_{i,j}\}$, as the algorithm never assumes that the inputs commute. So, to compute $\{C_{i,j}\}$, we:

1. For $1 \leq i \leq M$, compute linear combinations $\alpha_i$ of the $A_{i,j}$.
   
2. For $1 \leq i \leq M$, compute linear combinations $\beta_i$ of the $B_{i,j}$.
   
3. For $1 \leq i \leq M$, Compute $p_i = \alpha_i \beta_i$.
   
4. For $1 \leq i, j \leq m$, compute $C_{i,j}$ as a linear combination of $p_1, \ldots, p_M$.

Note that $\alpha_i, \beta_i, p_i$ are all $(n/m) \times (n/m)$ matrices. How many operations do we do? steps (i), (ii), (iv) each require $Mm^2$ additions of $(n/m) \times (n/m)$ matrices, so in total require $O(Mn^2)$ additions. Step (iii) requires $M$ multiplications of matrices of size $(n/m) \times (n/m)$.

So, we get the recursion formula

$$T(n) = m^\alpha T(n/m) + O(m^\alpha n^2).$$

This solves to $O((mn)^\alpha)$ if $\alpha > 2$ and to $O((mn)^2 \log n)$ if $\alpha = 2$. Let’s see explicitly the first case, the second being similar.

Let $n = m^s$. This recursion solves to a tree of depth $s$, where each node has $m^\alpha$ children. The number of nodes at depth $i$ is $m^{\alpha i}$, and the amount of computation that each makes is $O(m^\alpha(n/m)^2)$. Hence, the total amount of computation at depth $i$ is $O(m^\alpha \cdot m^{(\alpha-2)i}n^2)$. As long as $\alpha > 2$, this grows exponentially fast in the depth, and hence controlled by the last level (at depth $s$) which takes $O(m^\alpha \cdot m^{(\alpha-2)s}m^{2s}) = O((mn)^\alpha)$. \qed
1.2 Verifying matrix multiplication

Assume that someone gives you a magical algorithm that is supposed to multiply two matrices quickly. How would you verify it? one way is to compute matrix multiplication yourself, and compare the results. This will take time $O(n^\omega)$. Can you do better? the answer is yes, if we allow for randomization. In the following, our goal is to verify that $AB = C$ where $A, B, C$ are $n \times n$ matrices over an arbitrary field.

Function MatrixMultVerify

<table>
<thead>
<tr>
<th>Input: $n \times n$ matrices $A, B, C$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Is it true that $AB = C$?</td>
</tr>
<tr>
<td>1. Choose $x \in {0, 1}^n$ randomly.</td>
</tr>
<tr>
<td>2. Return TRUE if $A(Bx) = Cx$, and FALSE otherwise.</td>
</tr>
</tbody>
</table>

Clearly, if $AB = C$ then the algorithm always returns true. Moreover, as all the algorithm does is iteratively multiply an $n \times n$ matrix with a vector, it runs in time $O(n^2)$. The main question is: can we find matrices $A, B, C$ where $AB \neq C$, but where the algorithm returns TRUE with high probability? The answer is no, and is provided by the following lemma, applied to $M = AB - C$.

Lemma 1.5. Let $M$ be a nonzero $n \times n$ matrix. Then $\Pr_{x \in \{0, 1\}^n}[Mx = 0] \leq 1/2$.

In particular, if we repeat this $t$ times, the error probability will reduce to $2^{-t}$.

Proof. The matrix $M$ has some nonzero row, lets say it is $a_1, \ldots, a_n$. Then,

$$\Pr_{x \in \{0, 1\}^n}[Mx = 0] \leq \Pr \left[ \sum a_i x_i = 0 \right].$$

Let $i$ be minimal such that $a_i \neq 0$. Then, $\sum a_i x_i = 0$ iff $x_i = \sum_{j> i} (-a_j / a_i) x_j$. Hence, for any fixing of $\{x_j : j > i\}$, there is at most one value for $x_i$ which would make this hold.

$$\Pr \left[ \sum a_i x_i = 0 \right] = \mathbb{E}_{x_j, \ldots, x_n \in \{0, 1\}} \Pr \left[ \sum_{x_i \in \{0, 1\}} a_i x_i = 0 \right]$$

$$= \mathbb{E}_{x_j, \ldots, x_n \in \{0, 1\}} \left[ \Pr_{x_i \in \{0, 1\}} \left[ x_i = \sum_{j > i} (-a_j / a_i) x_j \right] \right]$$

$$\leq 1/2.$$
1.3 Application: checking if a graph contains a triangle

Let $G = (V, E)$ be a graph. Our goal is to find whether $G$ contains a triangle, and more generally, enumerate the triangles in $G$. Trivially, this takes $n^3$ time. We will show how to improve it using fast matrix multiplication. Let $|V| = n$, $A$ be the $n \times n$ adjacency matrix of $G$, $A_{i,j} = 1_{(i,j) \in E}$. Observe that

$$(A^2)_{i,j} = \sum_k A_{i,k}A_{k,j} = \text{number of paths of length two between } i, j$$

So, to check $G$ contains a triangle, we can first compute $A^2$, and then use it to detect if there is a triangle.

**Function TriangleExists(A)**

| Input: An $n \times n$ adjacency matrix $A$ |
| Output: Is there a triangle in the graph? |

1. Compute $A^2$
2. Check if there is $1 \leq i, j \leq n$ with $A_{i,j} = 1$ and $(A^2)_{i,j} \geq 1$.

The running time of step 1 is $O(n^\omega)$, and of step 2 is $O(n^2)$. Thus the total time is $O(n^\omega)$.

1.4 Application: listing all triangles in a graph

We next describe a variant of the TriangleExists algorithms, which lists all triangles in the graph. Again, the goal is to improve upon the naive $O(n^3)$ algorithm which tests all possible triangles.

The enumeration algorithm will be recursive. At each step, we partition the vertices to two sets and recurse over the possible 8 configurations. To this end, we will need to check if a triangle $i,j,k$ exists in $G$ with $i \in I, j \in J, k \in K$ for some $I, J, K \subset V$. The same algorithm works.

**Function TriangleExists(A; I,J,K)**

| Input: An $n \times n$ adjacency matrix $A$, $I, J, K \subset \{1, \ldots, n\}$ |
| Output: Is there a triangle $(i, j, k)$ with $i \in I, j \in J, k \in K$. |

1. Let $A_1, A_2, A_3$ be $I \times J, J \times K, I \times K$ submatrices of $A$.
2. Compute $A_1A_2$.
3. Check if there is $i \in I, k \in K$ with $(A_1A_2)_{i,k} = 1$ and $(A_3)_{i,k} = 1$.

We next describe the triangle listing algorithm. For simplicity, we assume $n$ is a power of two.
Function TrianglesList(A; I,J,K)

Input: An \( n \times n \) adjacency matrix \( A \), \( I,J,K \subset \{1,\ldots,n\} \)

Output: Listing of all triangles \((i,j,k)\) with \( i \in I, j \in J, k \in K \).

1. If \( n = 1 \) check if triangle exists. If so, output it.
2. If CheckTriangle \((I,J,K)\) is False return.
3. Partition \( I = I_1 \cup I_2, J = J_1 \cup J_2, K = K_1 \cup K_2 \).
4. Run TrianglesList \((I_a,J_b,K_c)\) for all \( 1 \leq a,b,c \leq 2 \).

We will run TrianglesList \((A; V,V,V)\) to enumerate all triangles in the graph.

**Lemma 1.6.** If \( G \) has \( m \) triangles, then TrianglesList outputs all triangles, and runs in time \( O(n^\omega m^{1-\omega/3}) \).

In particular, if \( \omega = 2 \), the algorithm runs in time \( O(n^2 m^{1/3}) \).

**Proof.** It is clear that the algorithm lists all triangles, and every triangle is listed once. To analyze its running time, consider the tree defined by the execution of the algorithm. A node at depth \( d \) corresponds to three matrices of size \( n/2^d \times n/2^d \). It either has no children (if there is no triangle in the corresponding sets of vertices), or has 8 children. Let \( \ell_i \) denote the number of nodes at depth \( i \), then we know that

\[
\ell_i \leq \min(8^i, 8m).
\]

The first bound is obvious, the second follows because for any node at depth \( i \), its parent at depth \( i-1 \) must contain a triangle, and all the triangles at a given depth are disjoint. The computation time at level \( i \) is given by

\[
T_i = \ell_i \cdot O((n/2^i)^\omega)
\]

Let \( i^* \) be the level at which \( 8^{i^*} = 8m \). The computation time up to level \( i^* \) is given by

\[
\sum_{i \leq i^*} T_i \leq \sum_{i \leq i^*} 8^i \cdot O((n/2^i)^\omega) = \sum_{i \leq i^*} O(n^{\omega}2^{i(3-\omega)}) = O(n^{\omega}2^{i^*(3-\omega)}) = O(T_{i^*}).
\]

The computation time after level \( i^* \) is given by

\[
\sum_{i \geq i^*} T_i \leq \sum_{i \geq i^*} m \cdot O((n/2^i)^\omega) \leq m \cdot O((n/2^{i^*})^\omega) = O(T_{i^*}).
\]

So the total running time is controlled by that of level \( i^* \), and hence

\[
\sum T_i = O(T_{i^*}) = O(n^{\omega}m^{1-\omega/3}).
\]

**Open Problem 1.7.** How fast can we find one triangle in a graph? How about \( m \) triangles?
2 Fast Fourier Transform and fast polynomial multiplication

The Fast Fourier Transform is an amazing discovery with many applications. Here, we motivate it by the problem of computing quickly the product of two polynomials.

2.1 Univariate polynomials

Fix a field, say the reals. A univariate polynomial is

\[ f(x) = \sum_{i=0}^{n} f_i x^i, \]

where \( x \) is the variable and \( f_i \in \mathbb{R} \) are the coefficients. We may assume that \( f_n \neq 0 \), in which case we say that \( f \) has degree \( n \).

Given two polynomials \( f, g \) of degree \( n \), their sum is given by

\[ (f + g)(x) = f(x) + g(x) = \sum_{i=0}^{n} (f_i + g_i) x^i. \]

Note that given \( f, g \) as their list of coefficients, we can compute \( f + g \) in time \( O(n) \).

The product of two polynomials \( f, g \) of degree \( n \) each is given by

\[ (fg)(x) = f(x)g(x) = \left( \sum_{i=0}^{n} f_i x^i \right) \left( \sum_{j=0}^{n} g_j x^j \right) = \sum_{i=0}^{n} \sum_{j=0}^{n} f_i g_j x^{i+j} = \sum_{i=0}^{2n} \left( \sum_{j=0}^{\min(i,n)} f_j g_{i-j} \right) x^i. \]

So, in order to compute the coefficients of \( fg \), we need to compute \( \sum_{j=0}^{\min(i,n)} f_j g_{i-j} \) for all \( 0 \leq i \leq 2n \). This trivially takes time \( n^2 \). We will see how to do it in time \( O(n \log n) \), using Fast Fourier Transform (FFT).

2.2 The Fast Fourier Transform

Let \( \omega_n \in \mathbb{C} \) be a primitive \( n \)-th root of unity, \( \omega_n = e^{2\pi i/n} = \cos(2\pi/n) + i \sin(2\pi/n) \). The order \( n \) Fourier matrix is given by

\[ (F_n)_{ij} = (\omega_n)^{ij} = (\omega_n)^{ij \mod n} \]

What is so special about it? well, as we will see soon, we can multiply it by a vector in time \( O(n \log n) \), whereas for general matrices this takes time \( O(n^2) \). To keep the description simple, we assume from now on that \( n \) is a power of two.

**Theorem 2.1.** For any \( x \in \mathbb{C}^n \), we can compute \( F_n x \) using \( O(n \log n) \) additions and multiplications.
Proof. Decompose $F_n$ into four $n/2 \times n/2$ matrices as follows. First, reorder the rows to list first all $n/2$ even indices, then all $n/2$ odd indices. Let $F'_n$ be the new matrix, with re-ordered rows. Decompose

$$F'_n = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$ 

What are $A, B, C, D$? If $1 \leq a, b \leq n/2$ then

- $A_{a,b} = (F_n)_{2a,b} = (\omega_n)^{2ab} = (\omega_{n/2})^{ab} = (F_{n/2})_{a,b}$.
- $B_{a,b} = (F_n)_{2a,b+n/2} = (\omega_n)^{2ab+an} = (\omega_n)^{2ab} = (F_{n/2})_{a,b}$.
- $C_{a,b} = (F_n)_{2a+1,b} = (\omega_n)^{2ab+b} = (F_{n/2})_{a,b} \cdot (\omega_n)^b$.
- $D_{a,b} = (F_n)_{2a+1,b+n/2} = (\omega_n)^{2ab+b+an+n/2} = -(F_{n/2})_{a,b} \cdot (\omega_n)^b$.

So, let $P$ be the $n/2 \times n/2$ diagonal matrix with $P_{a,b} = (\omega_n)^b$. Then

$$A = B = F_{n/2}, \quad C = -D = F_{n/2}P.$$ 

In order to compute $F_n x$, decompose $x = (x', x'')$ with $x', x'' \in \mathbb{C}^{n/2}$. Then

$$F'_n x = (Ax + By, Cx + Dy) = (F_{n/2}(x + y), F_{n/2}P(x - y)).$$ 

Let $T(n)$ be the number of additions and multiplications required to multiply $F_n$ by a vector. Then to compute $F_n x$ we need to:

- Compute $x + y, x - y, P(x - y)$, which takes time $3n$ since $P$ is diagonal.
- Compute $F_{n/2}(x + y)$ and $F_{n/2}P(x - y)$. This takes time $2T(n/2)$.
- Reorder the entries of $F'_n x$ to compute $F_n x$, which takes another $n$ steps.

So we obtain the recursion formula

$$T(n) = 2T(n/2) + 4n.$$ 

This recursion solves to $T(n) \leq 4n \log n$:

$$T(n) \leq 2 \cdot 4(n/2) \log(n/2) + 4n = 4n \log n - 4n + 4n.$$ 

Open Problem 2.2. Can we compute $F_n x$ faster than $O(n \log n)$? maybe as fast as $O(n)$?
2.3 Inverse FFT

The inverse of the Fourier matrix is the complex conjugate of the Fourier matrix, up to scaling.

Lemma 2.3. \((F_n)^{-1} = \frac{1}{n} \overline{F_n}\).

Proof. We have \(\overline{F_{n,a,b}} = \overline{\omega_n^{ab}} = \omega_n^{-ab}\). So

\[
(F_n \overline{F_n})_{a,b} = \sum_{c=0}^{n-1} \omega_n^{ac} \omega_n^{-cb} = \sum_{c=0}^{n-1} (\omega_n)^{(a-b)c}.
\]

If \(a = b\) then the sum equals \(n\). We claim that when \(a \neq b\) the sum is zero. To see that, let \(S = \sum_{i=0}^{n-1} \omega_n^{ic}\), where \(c \neq 0 \mod n\). Then

\[
(\omega_n)^c \cdot S = \sum_{i=1}^{n} (\omega_n)^{ic} = \sum_{i=0}^{n-1} (\omega_n)^{ic} = S.
\]

Since \(\omega_n\) has order \(n\), we have \(\omega_n^c \neq 1\), and hence \(S = 0\). This concludes that

\[
F_n \overline{F_n} = nI_n.
\]

Corollary 2.4. For any \(x \in \mathbb{C}^n\), we can compute \((F_n)^{-1}x\) using \(O(n \log n)\) additions and multiplications.

Proof. We have

\[
F_n^{-1}x = \frac{1}{n} \overline{F_n \overline{x}} = \frac{1}{n} \overline{F_n \overline{x}}.
\]

We can conjugate \(x\) to obtain \(\overline{x}\) in time \(O(n)\); compute \(F_n \overline{x}\) in time \(O(n \log n)\); and conjugate the output and divide by \(n\) in time \(O(n)\).

2.4 Fast polynomial multiplication

Let \(f(x) = \sum_{i=0}^{n-1} f_i x^i\) be a polynomial. We identify it with the list of coefficients \((f_0, f_1, \ldots, f_{n-1}) \in \mathbb{C}^n\). Its order \(n\) Fourier transform is defined as its evaluations on the \(n\)-th roots of unity:

\[
\hat{f}_i = f(\omega_n^i).
\]

Lemma 2.5. Let \(f(x)\) be a polynomial of degree \(\leq n - 1\). Its Fourier transform can be computed in time \(O(n \log n)\).
Proof. We have \( \hat{f}_j = \sum_{i=0}^{n-1} f_i \cdot \omega_n^{ij} \). So
\[
\hat{f} = F_n \begin{pmatrix}
f_0 \\
f_1 \\
\vdots \\
f_{n-1}
\end{pmatrix}.
\]

Corollary 2.6. Let \( f \) be a polynomial of degree \( \leq n-1 \). Given the evaluations of \( f \) at the \( n \)-th roots of unity, we can recover the coefficients of \( f \) in time \( O(n \log n) \).

Proof. Compute \( f = (F_n)^{-1} \hat{f} \).

The Fourier transform of a product has a simple formula:
\[
(\hat{fg})_i = (fg)(\omega_n^i) = f(\omega_n^i) \cdot g(\omega_n^i) = \hat{f}_i \cdot \hat{g}_i.
\]

So, we can multiply two polynomials as follows: compute their Fourier transform; multiply it coordinate-wise; and then perform the inverse Fourier transform. Note that if \( f, g \) have degrees \( d, e \), respectively, then \( fg \) has degree \( d + e \). So, we need to choose \( n > d + e \) to compute their product correctly.

### Fast polynomial multiplication

<p>| Input: Polynomials ( f, g ) |</p>
<table>
<thead>
<tr>
<th>Output: The product ( fg )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0. Let ( n ) be the smallest power of two, ( n \geq \text{deg}(f) + \text{deg}(g) + 1 ).</td>
</tr>
<tr>
<td>1. Pad ( f, g ) to length ( n ) if necessary (by adding zeros)</td>
</tr>
<tr>
<td>2. Compute ( \hat{f} = F_n f )</td>
</tr>
<tr>
<td>3. Compute ( \hat{g} = F_n g )</td>
</tr>
<tr>
<td>4. Compute ( (\hat{fg})_i = \hat{f}_i \hat{g}_i ) for ( 1 \leq i \leq n )</td>
</tr>
<tr>
<td>5. Return ( fg = (F_n)^{-1} \hat{fg} ).</td>
</tr>
</tbody>
</table>

2.5 Multivariate polynomials

Let \( f, g \) be multivariate polynomials. For simplicity, let’s consider bivariate polynomials. Let
\[
f(x, y) = \sum_{i,j=0}^{n} f_{i,j} x^i y^j, \quad g(x, y) = \sum_{i,j=0}^{n} g_{i,j} x^i y^j.
\]
Their product is

\[(fg)(x, y) = \sum_{i,j,i',j'=0}^{n} f_{i,j} g_{i',j'} x^{i+i'} y^{j+j'} = \sum_{i,j=0}^{2n} \left( \sum_{i'=0}^{\min(n, i)} \sum_{j'=0}^{\min(n, j)} f_{i',j'} g_{i-i',j-j'} \right) x^{i} y^{j}.
\]

Our goal is to compute \(fg\) quickly. One approach is to define a two-dimensional FFT. Instead, we would reduce the problem of multiplying two bivariate polynomials of degree \(n\) in each variable, to the problem of multiplying two univariate polynomials of degree \(O(n^2)\), and then apply the algorithm using the standard FFT.

Let \(N\) be large enough to be determined later, and define the following univariate polynomials:

\[F(z) = \sum_{i,j=0}^{n} f_{i,j} z^{Ni+j}, \quad G(z) = \sum_{i,j=0}^{n} g_{i,j} z^{Ni+j}.
\]

We can clearly compute \(F, G\) from \(f, g\) in linear time, and as \(\deg(F), \deg(G) \leq (N+1)n\), we can compute \(F \cdot G\) in time \(O((Nn) \log(Nn))\). The only question is whether we can infer \(f \cdot g\) from \(F \cdot G\).

**Lemma 2.7.** Let \(N \geq 2n + 1\). If \(H(z) = F(z)G(z) = \sum H_i z^i\) then

\[(fg)(x, y) = \sum_{i,j=0}^{2n} H_{Ni+j} x^{i} y^{j}.
\]

**Proof.** We have

\[H(z) = F(z)G(z) = \left( \sum_{i=0}^{n} f_{i,j} z^{Ni+j} \right) \left( \sum_{i=0}^{n} g_{i',j'} z^{N(i'+j')} \right)
= \sum_{i,j,i',j'=0}^{n} f_{i,j} g_{i',j'} z^{N(i+i')+(j+j')}.
\]

We need to show that the only solutions for

\[N(i+i')+(j+j') = Ni^*+j^*
\]

where \(0 \leq i, i', j, j' \leq n\) and \(0 \leq i^*, j^* \leq 2n\), are those which satisfy \(i+i' = i^*, j+j' = j^*\). As \(0 \leq j+j', j^* \leq 2n\) and \(N > 2n\), if we compute the value modulo \(N\) we get that \(j+j' = j^*\), and hence also \(i+i' = i^*\). \(\square\)

**Corollary 2.8.** We can compute the product of two bivariate polynomials of degree \(n\) in time \(O(n^2 \log n)\).
3 Secret sharing schemes

Secret sharing is a method for distributing a secret amongst a group of participants, each of whom is allocated a share of the secret. The secret can only be reconstructed when an allowed groups of participants collaborate, and otherwise no information is learned about the secret. Here, we consider the following special case. There are \( n \) players, each receiving a share. The requirement is that every group of \( k \) players can together learn the secret, but any group of less than \( k \) players can learn nothing about the secret. A method to accomplish that is called an \((n,k)\)-secret sharing scheme.

**Example 3.1** \((3,2)\)-secret sharing scheme). Assume a secret \( s \in \{0, 1\} \). The shares \((S_1, S_2, S_3)\) are joint random variables, defined as follows. Sample \( x \in \mathbb{F}_5 \) uniformly, and set \( S_1 = s + x, S_2 = s + 2x, S_3 = s + 3x \). Then each of \( S_1, S_2, S_3 \) is uniform in \( \mathbb{F}_5 \), even conditioned on the secret, but any pair define two independent linear equations in two variables \( x, s \) which can be solved.

We will show how to construct \((n,k)\) secret sharing schemes for any \( n \geq k \geq 1 \). This will follow [Sha79].

### 3.1 Construction of a secret sharing scheme

In order to construct \((n,k)\)-secret sharing schemes, we will use polynomials. We will later see that this is an instance of a more general phenomena. Let \( \mathbb{F} \) be a finite field of size \( |\mathbb{F}| > n \). We choose a random polynomial \( f(x) \) of degree \( k - 1 \) as follows: \( f(x) = \sum_{i=0}^{k-1} f_i x^i \) where \( f_0 = s \) and \( f_1, \ldots, f_{k-1} \in \mathbb{F} \) are chosen uniformly. Let \( \alpha_1, \ldots, \alpha_n \in \mathbb{F} \) be distinct nonzero elements. The share for player \( i \) is \( S_i = f(\alpha_i) \). Note that the secret is \( s = f(0) \). For example, the \((3,2)\)-secret sharing scheme corresponds to \( \mathbb{F} = \mathbb{F}_5, \alpha_1 = 1, \alpha_2 = 2, \alpha_3 = 3 \).

**Theorem 3.2.** This is an \((n,k)\)-secret sharing scheme.

The proof will use the following definition and claim.

**Definition 3.3** (Vandermonde matrices). Let \( \alpha_1, \ldots, \alpha_k \in \mathbb{F} \) be distinct elements in a field. The Vandermonde matrix \( V = V(\alpha_1, \ldots, \alpha_k) \) is defined as follows:

\[
V_{i,j} = (\alpha_i)^{j-1}.
\]

**Lemma 3.4.** If \( \alpha_1, \ldots, \alpha_k \in \mathbb{F} \) are distinct elements then \( \det(V(\alpha_1, \ldots, \alpha_k)) \neq 0 \).

**Proof sketch.** We will show that \( \det(V(\alpha_1, \ldots, \alpha_k)) = \prod_{i<j}(\alpha_j - \alpha_i) \). In particular, it is nonzero whenever \( \alpha_1, \ldots, \alpha_k \) are distinct. To see that, let \( x_1, \ldots, x_k \in \mathbb{F} \) be variables, and define the polynomial

\[
f(x_1, \ldots, x_k) = \det(V(x_1, \ldots, x_k)).
\]

First, note that if we set \( x_i = x_j \) for some \( i \neq j \), then \( f_{|x_i=x_j} = 0 \). This is since the matrix \( V(x_1, \ldots, x_k) \) with \( x_i = x_j \) has two identical rows (the \( i \)-th and \( j \)-th rows), and hence its
determinant is zero. This then implies (and we omit the proof here) that \( f(x) \) is divisible by \( x_i - x_j \) for all \( i \neq j \). So we can factor

\[
f(x_1, \ldots, x_k) = \prod_{i > j} (x_i - x_j) g(x_1, \ldots, x_k)
\]

for some polynomial \( g(x_1, \ldots, x_k) \). Next, we claim that \( g \) is a constant. This will follow by comparing degrees. Recall that for an \( n \times n \) matrix \( V \) we have

\[
det(V) = \sum_{\pi \in S_n} (-1)^{\text{sign}(\pi)} \prod_{i=1}^{n} V_{\pi(i), i},
\]

where \( \pi \) ranges over all permutations of \( \{1, \ldots, n\} \). In our case, \( V_{i,j} = x_{\pi(i)}^j \) is a polynomial of degree \( j \), and hence

\[
\deg(det(V(x_1, \ldots, x_n))) = \sum_{j=0}^{n-1} j = \left( \begin{array}{c} n \\ 2 \end{array} \right).
\]

Observer that also

\[
\deg(\prod_{i > j} (x_i - x_j)) = \left( \begin{array}{c} n \\ 2 \end{array} \right).
\]

So it must be that \( g \) is a constant. One can further verify that in fact \( g = 1 \), although we would not need that.

If we substitute \( x_i = \alpha_i \) we obtain that, as \( \alpha_1, \ldots, \alpha_n \) are distinct that

\[
det(V(\alpha_1, \ldots, \alpha_n)) = \prod_{i > j} (\alpha_i - \alpha_j) \neq 0.
\]

\[\blacksquare\]

**Proof of Theorem 3.2.** We need to show two things: (i) any \( k \) players can recover the secret, and (ii) any \( k - 1 \) learn nothing about it.

(i) Consider any \( k \) players, say \( i_1, \ldots, i_k \). Each share \( S_{i_j} \) is a linear combination of the \( k \) unknown variables \( f_0, \ldots, f_{k-1} \). We will show that they are linearly independent, and hence the players have enough information to solve for the \( k \) unknowns, and in particular can recover \( f_0 = s \). Let \( V = V(\alpha_{i_1}, \ldots, \alpha_{i_k}) \). By definition we have \( S_{i_j} = (Vf)_j \), where we view \( f = (f_0, \ldots, f_{k-1}) \) as a vector in \( \mathbb{F}^k \). Since \( \det(V) \neq 0 \), the players can solve the system of equations and obtain \( f_0 = (V^{-1}(S_{i_1}, \ldots, S_{i_k}))_1 \).

(ii) Consider any \( k - 1 \) players, say \( i_1, \ldots, i_{k-1} \). We will show that for any fixing of \( f_0 = s \), the random variables \( S_{i_1}, \ldots, S_{i_{k-1}} \) are independent and uniform over \( \mathbb{F} \). To see that, let \( V = V(0, \alpha_{i_1}, \ldots, \alpha_{i_{k-1}}) \) and let \( f = (f_0, \ldots, f_{k-1}) \in \mathbb{F}^k \) be chosen uniformly. Then, \( (f_0, S_{i_1}, \ldots, S_{i_{k-1}}) = Vf \) is also uniform in \( \mathbb{F}^k \). In particular, \( f_0 \) is independent of \( (S_{i_1}, \ldots, S_{i_{k-1}}) \), and hence the distribution of \( (S_{i_1}, \ldots, S_{i_{k-1}}) \), which happens to be uniform in \( \mathbb{F}^{k-1} \), is independent of the choice of \( f_0 \). Thus, the \( k - 1 \) learn no information about the secret.
We can in fact generalize this construction. Let $M$ be an $(n + 1) \times k$ matrix over a field $\mathbb{F}$ with the following properties:

- The first row of $M$ is $(1, 0, \ldots, 0)$.
- Each $k$ rows of $M$ are linearly independent.

For example, if $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$ are nonzero and distinct, then $M_{i,j} = \alpha_j^{i-1}$ achieves this, where we set $\alpha_0 = 0$ and $0^0 = 1$. The $(n,k)$ secret sharing scheme that uses $M$ goes as follows: randomly choose $f_1, \ldots, f_{k-1} \in \mathbb{F}$, compute $S = M(s, f_1, \ldots, f_k)^T$, and give $S_i$ to the $i$-th player as its share. It is easy to verify that the proof of Theorem 3.2 extends to this more general case. We will later see that such matrices play an important role in other domains, such as coding theory (MDS codes) and pseudo-randomness ($k$-wise independent random variables).

### 3.2 Lower bound on the share size

In our construction, the shares were elements of $\mathbb{F}$, and hence their size grew with the number of players. One may ask whether this is necessary, or whether there are better constructions which achieve smaller shares. Here, we will only analyze the case of linear constructions (such as above), although similar bounds can be obtained by general secret sharing schemes.

**Lemma 3.5.** Let $M$ be a $n \times k$ matrix over a field $\mathbb{F}$, with $n \geq k + 2$, such that any $k$ rows in $M$ are linearly independent. Then $|\mathbb{F}| \geq \max(k, n-k)$. In particular, $|\mathbb{F}| \geq n/2$.

We note that the condition $n \geq k + 2$ is tight: the $(k + 1) \times k$ matrix whose first $k$ rows form the identity matrix, and its last row is all ones, has this property over any field, and in particular $\mathbb{F}_2$. We also note that there is a conjecture (called the MDS conjecture) which speculates that in fact $|\mathbb{F}| \geq n - 1$, and our construction above is tight.

**Proof.** We can apply any invertible linear transformation to the columns of $M$, without changing the property that any $k$ rows are linearly independent. So, we may assume that

$$M = \begin{pmatrix} I & R \end{pmatrix},$$

where $I$ is the $k \times k$ identity matrix, and $R$ is a $k \times (n-k)$ matrix.

Next, we argue that $R$ cannot contain any 0. Otherwise, if for example $R_{i,j} = 0$, then the following $k$ rows of $M$ are linearly dependent: the $i$-th row of $R$, and the $k-1$ rows of the identity matrix which exclude row $j$. So, $R_{i,j} \neq 0$ for all $i,j$.

Hence, we may scale the rows of $R$ so that $R_{i,1} = 1$ for all $1 \leq i \leq n-k$. Moreover, we can then scale the columns of $R$ so that $R_{1,i} = 1$ for all $1 \leq i \leq k$. There are now two cases to consider:
(i) If $|\mathbb{F}| < k$ then $R_{2,i} = R_{2,j}$ for some $1 \leq i < j \neq k$. But then the following $k$ rows are linearly dependent: the first and second rows of $R$, and the $k - 2$ rows of the identity matrix which exclude rows $i, j$. So, $|\mathbb{F}| \geq k$.

(ii) If $|\mathbb{F}| < n - k$ then $R_{i,2} = R_{j,2}$ for some $1 \leq i < j \leq n - k$. But then the following $k$ rows are linearly dependent: the $i$-th and $j$-th rows of $R$, and the $k - 2$ rows of the identity matrix which exclude rows $1, 2$. So, $|\mathbb{F}| \geq n - k$. 

$\square$
4 Error correcting codes

4.1 Basic definitions

An error correcting code allows to encode messages into (longer) codewords, such that even in the presence of errors, we can decode the original message. Here, we focus on “worst case errors”, where we make no assumptions on the distribution of errors, but instead limit the number of errors.

Definition 4.1 (Error correcting code). Let $\Sigma$ be a finite set, $n \geq k \geq 1$. An error correcting code over the alphabet $\Sigma$ of message length $k$ and codeword length $n$ (also called block length) consists of

- A set of codewords $C \subset \Sigma^n$ of size $|C| = |\Sigma|^k$.
- A one-to-one encoding map $E : \Sigma^k \to C$.
- A decoding map $D : \Sigma^n \to \Sigma^k$.

We require that $D(E(m)) = m$ for all messages $m \in \Sigma^k$.

To describe the error correcting capability of a code, define the distance of $x, y \in \Sigma^n$ as the number of coordinates where they differ, $\text{dist}(x, y) = |\{i \in [n] : x_i \neq y_i\}|$.

Definition 4.2 (Error correction capability of a code). A code $(C, E, D)$ can correct up to $e$ errors if for any message $m \in \Sigma^k$ and any $x \in \Sigma^n$ such that $\text{dist}(E(m), x) \leq e$, it holds that then $D(x) = m$.

Example 4.3 (Repetition code). Let $\Sigma = \{0, 1\}$, $k = 1$, $n = 3$. Define $C = \{000, 111\}$, $E : \{0, 1\} \to \{0, 1\}^3$ by $E(0) = 000, E(1) = 111$. Define $D : \{0, 1\}^3 \to \{0, 1\}$ by $D(x_1, x_2, x_3) = \text{Majority}(x_1, x_2, x_3)$. Then $(C, E, D)$ can correct up to 1 errors.

If we just care about combinatorial bounds (that is, ignore algorithmic aspects), then a code is defined by its codewords. We can define $E : \Sigma^k \to C$ by any one-to-one way, and $D : \Sigma^n \to \Sigma^k$ by mapping $x \in \Sigma^n$ to the closest codeword $E(m)$, breaking ties arbitrarily. From now on, we simply describe codes by describing the set of codewords $C$. Once we start discussing algorithms for encoding and decoding, we will revisit this assumption.

Definition 4.4 (Minimal distance of a code). The minimal distance of $C$ is the minimal distance of any two distinct codewords, $\text{dist}_{\text{min}}(C) = \min_{x \neq y \in C} \text{dist}(x, y)$.

Definition 4.5 ($(n, k, d)$-code). An $(n, k, d)$-code over an alphabet $\Sigma$ is a set of codewords $C \subset \Sigma^n$ of size $|C| = |\Sigma|^k$ and minimal distance $\geq d$. 

23
Lemma 4.6. Let $C$ be an $(n,k,2e+1)$-code. Then it can decode from $e$ errors.

Proof. Let $x \in C$, $y \in \Sigma^n$ be such that $\text{dist}(x,y) \leq e$. We claim that $x$ is the unique closest codeword to $y$. Assume not, that is there is another $x' \in C$ with $\text{dist}(x',y) \leq e$. Then by the triangle inequality, $\text{dist}(x,x') \leq \text{dist}(x,y) + \text{dist}(y,x') \leq 2e$, which contradicts the assumption that the minimal distance of $C$ is $2e+1$. \qed

Moreover, if $C$ has minimal distance $d$, then there exist $x, x' \in C$ and $y \in \Sigma^n$ such that $\text{dist}(x,y) + \text{dist}(x',y) = d$, so the bound is tight. So, we can restrict our study to the existence of $(n,k,d)$-codes.

4.2 Basic bounds

Lemma 4.7 (Singleton bound). Let $C$ be an $(n,k,d)$-code. Then $k \leq n - d + 1$.

Proof. We have $C \subset \Sigma^n$ of size $|C| = |\Sigma|^k$. Let $C' \subset \Sigma^{n-d+1}$ be the code obtained by deleting the first $d-1$ coordinates from all codewords if $C$. Note that all codewords remain distinct, as we assume the minimal distance is at least $d$. So, $|C'| = |\Sigma|^k$. This implies that $k \leq n - d + 1$. \qed

An MDS code (Maximal Distance Separable) is a code for which $k = n - d + 1$. We will later see an example of such a code (Reed-Solomon code).

Lemma 4.8 (Hamming bound). Let $C$ be an $(n,k,2e+1)$-code over $\Sigma$ with $|\Sigma| = q$. Then

$$q^k \sum_{i=0}^{e} \binom{n}{i}(q-1)^i \leq q^n.$$ 

Proof. For each codeword $x \in C$ define the ball of radius $e$ around it, 

$$B(x) = \{y \in \Sigma^n : \text{dist}(x,y) \leq e\}.$$ 

These ball cannot intersect by the minimal distance requirement. Each ball contains $\sum_{i=0}^{e} \binom{n}{i}(q-1)^i$ elements, and there are $q^k$ such balls. The lemma follows. \qed

The singleton bound and the hamming bound are incomparable, in the sense that in some regimes one is superior to the other, and in other regimes the opposite holds. The following example demonstrates that.

Example 4.9. Let $d = 3$, corresponding to correcting $1$ error. The singleton bound gives $k \leq n - 2$ over any alphabet. The hamming bound gives (for $e = 1$)

$$q^k(1 + (q-1)n) \leq q^n.$$ 

For binary codes, eg $q = 2$, it gives $2^k(n+1) \leq 2^n$, which gives $k \leq n - \log_2(n+1)$, which is a stronger bound than that given by the singleton bound. On the other hand, if we take $q$ to be very large, then the hamming bound gives $k \leq n - 1 + o_q(1)$, which is weaker than the singleton bound.
4.3 Existence of asymptotically good codes

An \((n,k,d)\)-code is said to be asymptotically good (or simply good) if \(k = \alpha n, d = \beta n\) for some constants \(\alpha, \beta > 0\). More precisely, we consider families of codes with growing \(n\) and fixed \(\alpha, \beta > 0\). The singleton bound implies that \(\alpha + \beta \leq 1\), and MDS codes achieve that. It is unknown if over binary alphabet it is achievable, and it is one of the major open problems in coding theory. Here, we will show that good codes exist for some constants \(\alpha, \beta\), without trying to optimize them.

**Lemma 4.10.** There exists a family \(C \subset \{0,1\}^n\) of size \(2^{n/10}\) such that \(\text{dist}_{\text{min}}(C) \geq n/10\).

**Proof.** The proof is probabilistic. For \(N = 2^{n/10}\) let \(x_1, \ldots, x_N \in \{0,1\}^n\) be uniformly chosen. We claim that with high probability, \(C = \{x_1, \ldots, x_N\}\) is as claimed. To see that, let's consider the probability that \(\text{dist}(x_i, x_j) \leq n/10\) for some fixed \(1 \leq i < j \leq N\). The number of choices for \(x_i\) is \(2^n\). Given \(x_i\), the number of choices for \(x_j\) of distance at most \(n/10\) from \(x_i\) is \(\sum_{i=0}^{n/10} \binom{n}{i}\). This should be divided by the total number of pairs, which is \(2^{2n}\). So,

\[
\Pr[\text{dist}(x_i, x_j) \leq n/10] \leq \frac{2^n \sum_{i=0}^{n/10} \binom{n}{i}}{2^{2n}} \leq \frac{n \binom{n}{n/10}}{2^n}.
\]

We need some estimates for the binomial coefficient. A useful one is

\[
\binom{n}{m}^m \leq \binom{n}{m} \leq \left(\frac{en}{m}\right)^m.
\]

So,

\[
\binom{n}{n/10} \leq \left(\frac{en}{n/10}\right)^{n/10} \leq ((10e)^{1/10})^n \leq (1.4)^n.
\]

So,

\[
\Pr[\text{dist}(x_i, x_j) \leq n/10] \leq \frac{n(1.4)^n}{2^n} = n(0.7)^n.
\]

Now, the probability that there exists some pair \(1 \leq i < j \leq N\) such that \(\text{dist}(x_i, x_j) \leq n/10\) can be upper bounded by the union bound,

\[
\Pr[\exists 1 \leq i < j \leq N, \text{dist}(x_i, x_j) \leq n/10] \leq \sum_{1 \leq i < j \leq N} \Pr[\text{dist}(x_i, x_j) \leq n/10] \\
\leq N^2 n(0.7)^n = 2^{2n/10} n(0.7)^n \leq n(0.81)^n.
\]

So, the probability that \(\text{dist}_{\text{min}}(C) \leq n/10\) is at most most \(n(0.81)^n\), which is exponentially small. Hence, with very high probability, the randomly chosen code will be a good code. \(\square\)

We can get the same bounds without using probability. Consider the following process for choosing \(x_1, x_2, \ldots, x_N \in \{0,1\}^n\). Pick \(x_1\) arbitrarily, and delete all points of distance \(\leq n/10\) from it; pick \(x_2\) from the remaining points, and delete all points of distance \(\leq n/10\) from it; and so on. This process guarantees that the minimum distance of the code is at least \(n/10\), and we can repeat this process \(20\) times to get codes with a minimum distance of \(n/2\).

25
from it; pick $x_3$ from the remaining points, and so on. Continue in such a way until all points are exhausted. The number of points chosen $N$ satisfies that

$$N \geq \frac{2^n}{\sum_{i=0}^{n/10} \binom{n}{i}}.$$ 

This is because we have initially a total of $2^n$ points, and at each point we delete at most $\sum_{i=0}^{n/10} \binom{n}{i}$ undelete points. The same calculations as before show that $N \geq 2^{n/10}$.

4.4 Linear codes

A special family of codes are linear codes. Let $\mathbb{F}$ be a finite field. In a linear code, $\Sigma = \mathbb{F}$ and $C \subset \mathbb{F}^n$ is a $k$-dimensional subspace. The encoding map is a linear map: $E(x) = Ax$ where $A$ is a $n \times k$ matrix over $\mathbb{F}$. Note that $\text{rank}(A) = k$, as otherwise the set of codewords will have dimension less than $k$. In practice, nearly all codes are linear, as the encoding map is easy to define. However, the decoding map needs inherently to be nonlinear, and is usually the hardest to compute.

Claim 4.11. Let $C$ be a linear code. Then $\text{dist}_{\text{min}}(C) = \min_{0 \neq x \in C} \text{dist}(0, x)$.

Proof. If $x_1, x_2 \in C$ have the minimal distance, then $\text{dist}(x_1, x_2) = \text{dist}(0, x_1 - x_2)$ and $x_1 - x_2 \in C$. \hfill $\square$

We can view the decoding problem from either erasures or errors as a linear algebra problem. Let $A$ be an $n \times k$ matrix. Codewords are $C = \{Ax : x \in \mathbb{F}^k\}$, or equivalently the subspace spanned by the columns of $A$.

Decoding from erasures. The problem of decoding from erasures is equivalent to the following problem: given $y \in (\mathbb{F} \cup \{?\})^n$, find $x \in \mathbb{F}^k$ such that $(Ax)_i = y_i$ for all $y_i \neq ?$. Equivalently, we want the sub-matrix formed by keeping only the rows $\{i \in [n] : y_i \neq ?\}$ to form a rank $k$ matrix. So, the requirement that a linear code can be uniquely decoded from $e$ erasures, is equivalent to the requirement that if any $e$ rows are deleted in the matrix, it still has rank $k$. Clearly, $e \leq n - k$. We will see a code achieving this bound, the Reed-Solomon code. It will be based on polynomials.

Decoding from errors. The problem of decoding from $e$ errors is equivalent to the following problem: given $y \in \mathbb{F}^n$, find $x \in \mathbb{F}^k$ such that $(Ax)_i \neq y_i$ for at most $e$ coordinates. Equivalently, we want to find a vector spanned by the columns of $A$, which agrees with $y$ in at least $n - e$ coordinates. If the code has minimal distance $d$, then we know that this is mathematically possible whenever $e < d/2$; however, finding this vector is in general computationally hard. We will see a code where this is possible, and which moreover has the best minimal distance, $d = n - k + 1$. Again, it will be Reed-Solomon code.
5  Reed-Solomon codes

Reed-Solomon codes are an important group of error-correcting codes that were introduced by Irving Reed and Gustave Solomon in the 1960s. They have many important applications, the most prominent of which include consumer technologies such as CDs, DVDs, Blu-ray Discs, QR Codes, satellite communication and so on.

5.1 Definition

Reed-Solomon codes are defined as the evaluation of low degree polynomials over a finite field. Let \( F \) be a finite field. Messages are in \( F^k \) are treated as the coefficients of a univariate polynomial of degree \( k-1 \), and codewords are its evaluations on \( n < |F| \) points. So, Reed-Solomon codes are defined by specifying \( F, k \) and \( n < |F| \) points \( \alpha_1, \ldots, \alpha_n \in F \), and its codewords are

\[
C = \{ (f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_n)) : f(x) = \sum_{i=0}^{k-1} f_i x^i, f_0, \ldots, f_{k-1} \in F \}.
\]

We define this family of codes in general as \( \text{RS}_F(n, k) \), and if needs, we can specify the evaluation points. An important special case is when \( n = |F| \), and we evaluate the polynomial on all field elements.

**Lemma 5.1.** The minimal distance of \( \text{RS}_F(n, k) \) is \( d = n - k + 1 \).

**Proof.** As \( C \) is a linear code, it suffices to show that for any nonzero polynomial \( f(x) \) of degree \( \leq k - 1 \), \( |\{x \in F : f(x) = 0\}| \leq k - 1 \). Hence, \( |\{i \in [n] : f(\alpha_i) = 0\}| \geq n - k + 1 \). Now, this follows from the fundamental theorem of algebra: a nonzero polynomial of degree \( r \) has at most \( r \) roots. We prove it below by induction on \( r \).

If \( r = 0 \) then \( f \) is a nonzero constant, and so it has no roots. So, assume \( r \geq 1 \). Let \( \alpha \in F \) be such that \( f(\alpha) = 0 \). Let's shift the input so that the root is at zero. That is, define \( g(x) = f(x + \alpha) \), so that \( g(0) = 0 \) and \( g(x) \) is also a polynomial of degree \( r \). Express it as

\[
g(x) = \sum_{i=0}^{r} f_i (x + \alpha)^i = \sum_{i=0}^{r} g_i x^i.
\]

Since \( g_0 = g(0) = 0 \), we get that \( g(x) = x h(x) \) where \( h(x) = \sum_{i=0}^{r-1} g_{i+1} x^i \) is a polynomial of degree \( r-1 \), and hence \( f(x) = g(x - \alpha) = (x - \alpha) h(x - \alpha) \). By induction, \( h(x - \alpha) \) has at most \( r - 1 \) roots, and hence \( f \) has at most \( r \) roots. \( \square \)

Recall that the Singleton bound shows that in any \( (n, k, d) \) code, \( d \leq n - k + 1 \). Codes which achieve this bound, i.e for which \( d = n - k + 1 \), are called MDS codes (Maximal Distance Separable). What we just showed is that Reed-Solomon codes are MDS codes. In fact, for prime fields, it is known that Reed-Solomon are the only MDS codes [Bal11], and it is conjecture to be true over non-prime fields as well (except for a few exceptions in characteristic two).
5.2 Decoding Reed-Solomon codes from erasures

We first analyze the ability of Reed-Solomon codes to recover from erasures. Assume that we are given a Reed-Solomon codeword, with some coordinates erased. Let $S$ denote the set of remaining coordinates. That is, for $S \subset \{1, \ldots, n\}$ we know that $f(\alpha_i) = y_i$ for all $i \in S$, where $y_i \in \mathbb{F}$. The question is: for which sets $S$ is this information sufficient to uniquely recover the polynomial $f$?

Equivalently, we need to solve the following system of linear equations, where the unknowns are the coefficients $f_0, \ldots, f_{k-1}$ of the polynomial $f$:

$$k-1 \sum_{j=0}^{k-1} f_j \alpha_i^j = y_i, \quad \forall i \in S.$$ 

In order to analyze this, let $V = V(\{\alpha_i : i \in S\})$ be a $|S| \times k$ Vandermonde matrix given by $V_{i,j} = \alpha_i^j$ for $i \in S$, $0 \leq j \leq k - 1$. Then, we want to solve the system of linear equations $Vf = y$,

where $f = (f_0, \ldots, f_{k-1}) \in \mathbb{F}^k$ and $y = (y_i : i \in S) \in \mathbb{F}^{|S|}$. Clearly, we need $|S| \geq k$ for a unique solution to exist. As we saw, whenever $|S| = k$ the matrix $V$ is invertible, hence there is a unique solution. So, as long as $|S| \geq k$, we can restrict to $k$ equations and uniquely solve for the coefficients of $f$.

**Corollary 5.2.** The code $\text{RS}_\mathbb{F}(n,k)$ can be uniquely decoded from $n-k$ erasures.

5.3 Decoding Reed-Solomon codes from errors

Next, we study the harder problem of decoding from errors. Again, let $f(x) = \sum_{i=0}^{k-1} f_i x^i$ be an unknown polynomial of degree $k-1$. We know its evaluation on $\alpha_1, \ldots, \alpha_n \in \mathbb{F}$, but with a few errors, say $e$. That is, we are given $y_1, \ldots, y_n \in \mathbb{F}$, such that $y_i \neq f(\alpha_i)$ for at most $e$ of the evaluations. If we knew the locations of the errors, we would be back at the decoding from erasures scenario; however, we do not know them, and enumerating them is too costly. Instead, we will design an algebraic algorithm, called the Berlekamp-Welch algorithm, which can detect the locations of the errors efficiently, as long as the number of errors is not too large (interestingly enough, the algorithm was never published as an academic paper, and instead is a patent).

Define a ”error locating” polynomial $E(x)$ as follows:

$$E(x) = \prod_{i : y_i \neq f(\alpha_i)} (x - \alpha_i).$$

The decoder doesn’t know $E(x)$. However, we will still use it in the analysis. It satisfies the following equation:

$$E(\alpha_i) (f(\alpha_i) - y_i) = 0 \quad \forall 1 \leq i \leq n.$$ 

Let $N(x) = E(x)f(x)$. Note that $\deg(E) = e$ and $\deg(N) = \deg(E) + \deg(f) = e + k - 1$.

We established the following claim.
Claim 5.3. There exists polynomials $E(x), N(x)$ of degrees $\deg(E) = e, \deg(N) = e + k - 1$ such that

$$N(\alpha_i) - y_i E(\alpha_i) = 0 \quad \forall 1 \leq i \leq n.$$ 

Proof. We have $N(\alpha_i) = E(\alpha_i)f(\alpha_i)$. This is equal to $E(\alpha_i)y_i$ as either $y_i = f(\alpha_i)$ or otherwise $E(\alpha_i) = 0$. 

The main idea is that we can find such polynomials by solving a system of linear equations.

Claim 5.4. We can efficiently find polynomials $\tilde{E}(x), \tilde{N}(x)$ of degrees $\deg(\tilde{E}) \leq e, \deg(\tilde{N}) \leq e + k - 1$, not both zero, such that

$$\tilde{N}(\alpha_i) - y_i \tilde{E}(\alpha_i) = 0 \quad \forall 1 \leq i \leq n.$$ 

Proof. Let

$$\tilde{E}(x) = \sum_{j=0}^{e} a_j x^j, \quad \tilde{N}(x) = \sum_{j=0}^{e+k-1} b_j x^j,$$

where $a_j, b_j$ are unknown coefficients. They need to satisfy the following system of $n$ linear equations:

$$\sum_{j=0}^{e} a_j \cdot \alpha_i^j - y_i \sum_{j=0}^{e+k-1} b_j \cdot \alpha_i^j = 0 \quad \forall 1 \leq i \leq n.$$ 

We know that this system has a nonzero solution (since we know that $E, N$ exist by our assumptions). So, we can find a nonzero solution by linear algebra. 

Note that it is not guaranteed that $\tilde{E} = E, \tilde{N} = N$, so we are not done yet. However, the next claim shows that we can still recover $f$ from any $\tilde{E}, \tilde{N}$ that we find.

Claim 5.5. If $e \leq (n - k)/2$ then $\tilde{N}(x) = \tilde{E}(x)f(x)$.

Proof. Consider the polynomial $R(x) = \tilde{N}(x) - \tilde{E}(x)f(x)$. Note that for any $i$ such that $f(\alpha_i) = y_i$, we have that

$$R(\alpha_i) = \tilde{N}(\alpha_i) - \tilde{E}(\alpha_i)f(\alpha_i) = \tilde{N}(\alpha_i) - \tilde{E}(\alpha_i)y_i = 0.$$ 

So, $R$ has at least $n - e$ roots. On the other hand, $\deg(R) \leq \max(\deg(\tilde{N}), \deg(\tilde{E}) + \deg(f)) \leq e + k - 1$. So, as long as $n - e > e + k - 1$, it has more zeros than its degree, and hence must be the zero polynomial. 

Corollary 5.6. The code $RS_F(n, k)$ can be uniquely decoded from $(n - k)/2$ errors.

Proof. Given $\tilde{N}, \tilde{E}$ we can solve for $f$ such that $\tilde{N}(x) = \tilde{E}(x)f(x)$, either by polynomial division, or by solving a system of linear equations.

Note that this is the best we can do, as the minimal distance is $n - k + 1$. 

29
6 Polynomial identity testing and finding perfect matchings in graphs

We describe polynomial identity testing in this chapter. It is a key ingredient in several algorithms. Here, we motivate it by the problem of checking whether a graph has a perfect matching. We will first present a combinatorial algorithm, based on Hall’s marriage theorem. Then, we will introduce a totally different algorithm based on polynomials and polynomial identity testing.

6.1 Perfect matchings in bi-partite graphs

Let $G = (U, V, E)$ be a bi-partite graph with $|U| = |V| = n$. Let $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$. A perfect matching is a matching of each node in $U$ to an adjacent distinct node in $V$. That is, it is given by a set of edges $(u_1, v_{\pi(1)}), (u_2, v_{\pi(2)}), \ldots, (u_n, v_{\pi(n)})$, where $\pi \in S_n$ is a permutation.

We would like to characterize when a give graph has a perfect matching. For $u \subset U$, let $\Gamma(u) \subseteq V$ denote the set of neighbors of $u$ in $V$. For $U' \subset U$, let $\Gamma(U') = \cup_{u \in U'} \Gamma(u)$ be the neighbors of the vertices of $U'$.

**Theorem 6.1** (Hall marriage theorem). $G$ has a perfect matching if and only if

$$|\Gamma(U')| \geq |U'| \quad \forall U' \subseteq U.$$  

**Proof.** The condition is clearly necessary: if $G$ has a perfect matching $\{(u_i, v_{\pi(i)}) : i \in [n]\}$ for some permutation $\pi$ then if $U' = \{u_{i_1}, \ldots, u_{i_k}\}$ then $v_{\pi(i_1)}, \ldots, v_{\pi(i_k)} \in \Gamma(U')$, and hence $|\Gamma(U')| \geq k = |U'|$.

The more challenging direction is to show that the condition given is sufficient. To show that, assume towards a contradiction that $G$ has no perfect matching. Assume without loss of generality (after possibly renaming the vertices) that $M = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ is the largest partial matching in $G$. Let $u \in U \setminus \{u_1, \ldots, u_m\}$. We will build a partial matching for $\{u, u_1, \ldots, u_m\}$, which would violate the maximality of $M$.

If $(u, v) \in E$ for some $v \notin \{v_1, \ldots, v_m\}$, then clearly $M$ is not maximal, as we can add the edge $(u, v)$ to it. More generally, we say that a path $P$ in $G$ is an augmenting path starting at $u$, if it is of the form

$$P = u, v_{i_1}, u_{i_1}, v_{i_2}, u_{i_2}, \ldots, v_{i_k}, u_{i_k}, v$$

with $v \notin \{v_1, \ldots, v_m\}$. Note that all the even edges in $P$ are in the matching $M$ (namely, $(v_{i_1}, u_{i_1}), \ldots, (v_{i_k}, u_{i_k}) \in M$), and all the odd edges are not in the matching $M$ (namely, $(u, v_{i_1}), (u_{i_1}, v_{i_2}), \ldots, (u_{i_k}, v) \notin M$). Such a path would also allow us to increase the matching size by one, by taking

$$M' = \{(u, v_{i_1}), (u_{i_1}, v_{i_2}), \ldots, (u_{i_k}, v)\} \cup \{(u_j, v_j) : j \notin \{i_1, \ldots, i_k\}\}.$$  

So, by our assumption that $M$ is a partial matching of maximal size, there are no augmenting paths in $G$ which start at $u$. 

30
We say that a path $P$ in $G$ is an alternating path starting at $u$, if each other edge in it belongs to the matching $M$. Such a path cannot end at some $v_i$ because then we can add $u_i$ to it. So, it ends at some $u_{i_k}$ and hence has the form

$$P = u, v_{i_1}, u_{i_1}, v_{i_2}, u_{i_2}, \ldots, v_{i_k}, u_{i_k}.$$ 

Let $X \subseteq \{u_1, \ldots, u_m\}$, $Y \subseteq \{v_1, \ldots, v_m\}$ be all the nodes in $u_1, \ldots, u_m$ and in $v_1, \ldots, v_m$, respectively, which can be reached by some alternating path starting at $u$. Note that $Y$ is nonempty as $u$ has at least one neighbor, and by assumption all the neighbors of $u$ are in $\{v_1, \ldots, v_m\}$. Assume without loss of generality that $Y = \{v_1, \ldots, v_k\}$ for some $k \leq m$. We claim that $X = \{u_1, \ldots, u_k\}$, i.e. all the nodes matched to $Y$ by the matching. This follows immediately from the construction. Furthermore, we claim that $Y = \Gamma(X)$, since if $(u_i, v_j) \in E$ with $u_i \in X$, then fix an augmenting path from $u$ to $u_i$. Either it already contains $v_j$, or otherwise we can extend it to $v_j$. In either case, $v_j \in Y$. But then for $S = X \cup \{u\}$ we have $|S| = k + 1$ and $|\Gamma(S)| = |T| = k$, a contradiction to the assumptions of the theorem.

So, we have a mathematical criteria to check if a bi-partite graph has a perfect matching. Moreover, it can be verified that the proof of Hall’s marriage theorem is in fact algorithmic: it can be used to find larger and larger partial matchings in a graph. This is much better than verifying the conditions of the theorem, which naively would take time $2^n$ to enumerate all subsets of $U$. We will now see a totally different way to check if a graph has a perfect matching, using polynomial identity testing.

### 6.2 Polynomial representation

We will consider multivariate polynomials in $x_1, \ldots, x_n$ of degree $d$. How can we represent them? one way is explicitly, by their list of coefficients:

$$f(x_1, \ldots, x_n) = \sum_{e_1, \ldots, e_n \geq 0 : \sum e_i \leq d} f_{e_1, \ldots, e_n} x_1^{e_1} \cdots x_n^{e_n}.$$ 

For large degrees, this can be quite large: the number of monomials in $n$ variables of degree $\leq d$ is $\binom{n + d}{n}$, which is exponentially large if both $n, d$ are large. This means that evaluating the polynomial on a single input would take exponential time.

Another way is via a computation. Consider for example $f(x) = (x + 1)^n$. It has $2^n + 1$ monomials, so evaluating it via summing over all monomials would take exponential time. However, we can compute it in time $O(n)$ by first evaluating $x + 1$, and then squaring it iteratively $n$ times. This shows that some polynomials of exponential degree can in fact be computed in polynomial time. In order to define this more formally, we introduce the notion of an algebraic circuit.

**Definition 6.2 (Algebraic circuit).** Let $\mathbb{F}$ be a field, and $x_1, \ldots, x_n$ be variables taking values in $\mathbb{F}$. An algebraic circuit computes a polynomial in $x_1, \ldots, x_n$. It is defined by a directed acyclic graph (DAG), with multiple leaves (nodes with no incoming edges) and a single root.
(node with no outgoing edges). Each leaf is labeled by either a constant \( c \in \mathbb{F} \) or a variable \( x_i \), which is the polynomial it computes. Internal nodes are labeled in one of two ways: they are either sum gates, which compute the sum of their inputs, or they are product gates, which compute the product of their inputs. The polynomial computed by the circuit is the polynomial computed by the root.

So for example, we can compute \((x + 1)^{2^n}\) using an algebraic circuit of size \(O(n)\):

- It has two leaves: \( v_1, v_2 \) which compute the polynomials \( f_{v_1}(x) = 1, f_{v_2}(x) = x \).
- The node \( v_3 \) is a sum gate with two children, \( v_1, v_2 \). It computes the polynomial \( f_{v_3}(x) = x + 1 \).
- For \( i = 1, \ldots, n \), let \( v_{i+3} \) be a multiplication gate with two children, both being \( v_{i+2} \). It computes the polynomial \( f_{v_{i+3}}(x) = f_{v_{i+2}}(x)^2 = (x + 1)^2 \).
- The root is \( v_{n+3} \), which computes the polynomial \((x + 1)^{2^n}\).

**Definition 6.3.** Let \( f(x_1, \ldots, x_n) \) be a polynomial. It is said to be efficiently computable if \( \deg(f) \leq \text{poly}(n) \) and there is an algebraic circuit of size \( \text{poly}(n) \) which computes \( f \). The class of polynomials which can be efficiently computed by algebraic circuits is called VP.

An interesting example for a polynomial in VP is the determinant of a matrix. Let \( M \) be a matrix with entries \( x_{ij} \). The determinant polynomial is

\[
\det(M)((x_{ij} : 1 \leq i, j \leq n)) = \sum_{\pi \in S_n} (-1)^{\text{sign}(\pi)} \prod_{i=1}^{n} x_{i,\pi(i)}.
\]

The sum is over all the permutations on \( n \) elements. In particular, the determinant is a polynomial in \( n^2 \) variables of degree \( n \). As a sum of monomials, it has \( n! \) monomials, which is very inefficient. However, we know that the determinant can be computed efficiently by Gaussian elimination. Although we would not show this here, it turns out that it can be transformed to an algebraic circuit of polynomial size which computes the determinant.

Another interesting polynomial is the permanent. It is defined very similarly to the determinant, except that we do not have the signs of the permutations:

\[
\text{per}(M)((x_{i,j} : 1 \leq i, j \leq n)) = \sum_{\pi \in S_n} \prod_{i=1}^{n} x_{i,\pi(i)}.
\]

A direct calculation of the permanent, by summing over all monomials, requires size \( n! \). There are more efficient ways (such as Ryser formula [Rys63]) which gives an arithmetic circuit of size \( O(2^n n^2) \), but these are still exponential. It is suspected that no sub-exponential algebraic circuits can compute the permanent, but we do not know how to prove this. The importance of this problem is that the permanent is complete, in the sense that many “counting problems” can be reduced to computing the permanent of some specific matrices.

**Open Problem 6.4.** What is the size of the smallest algebraic circuit which computes the permanent?
6.3 Polynomial identity testing

A basic question in mathematics is whether two objects are the same. Here, we will consider the following problem: given two polynomials \( f(x), g(x) \), possibly via an algebraic circuit, is it the case that \( f(x) = g(x) \)? Equivalently, since we can create a circuit computing \( f(x) - g(x) \), it is sufficient to check if a given polynomial is zero. If this polynomial is given via its list of coefficients, we can simply check that all of them are zero. But, this can be a very expensive procedure, as the number of coefficients can be exponentially large. For example, verifying the formula for the determinant of a Vandermonde matrix directly would take exponential time if done in this way, although we saw a direct proof of this formula.

We will see that using randomness, it can be verified if a polynomial is zero or not. This will be via the following lemma, called that Schwartz-Zippel lemma [Zip79, Sch80], which generalizes the fact that univariate polynomials of degree \( d \) have at most \( d \) roots, to multivariate polynomials.

**Lemma 6.5.** Let \( f(x_1, \ldots, x_n) \) be a nonzero polynomial of degree \( d \). Let \( S \subseteq \mathbb{F} \) of size \( |S| > d \). Then

\[
\Pr_{a_1, \ldots, a_n \in S} [f(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}.
\]

Note that the lemma is tight, even in the univariate case: if \( f(x) = \prod_{i=1}^{d} (x - i) \) and \( S = \{1, \ldots, s\} \) then \( \Pr_{a \in S} [f(a) = 0] = \frac{d}{s} \).

**Proof.** The proof is by induction on \( n \). If \( n = 1 \), then \( f(x_1) \) is a univariate polynomial of degree \( d \), hence it has at most \( d \) roots, and hence \( \Pr_{a_1 \in S} [f(a_1) = 0] \leq \frac{d}{|S|} \).

If \( n > 1 \), we express the polynomial as

\[
f(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_i \cdot f_i(x_1, \ldots, x_{n-1}),
\]

where \( f_0, \ldots, f_d \) are polynomials in the remaining \( n - 1 \) variables, and where \( \deg(f_i) \leq d - i \). Let \( e \leq d \) be maximal such that \( f_e \neq 0 \). Let \( a_1, \ldots, a_n \in S \) be chosen independently and uniformly. We will bound the probability that \( f(a_1, \ldots, a_n) = 0 \) by

\[
\Pr_{a_1, \ldots, a_n \in S} [f(a_1, \ldots, a_n) = 0] = \Pr [f(a_1, \ldots, a_n) = 0 | f_e(a_1, \ldots, a_{n-1}) = 0] \cdot \Pr [f_e(a_1, \ldots, a_{n-1}) = 0]
+ \Pr [f(a_1, \ldots, a_n) = 0 | f_e(a_1, \ldots, a_{n-1}) \neq 0] \cdot \Pr [f_e(a_1, \ldots, a_{n-1}) \neq 0]
\leq \Pr [f_e(a_1, \ldots, a_{n-1}) = 0] + \Pr [f(a_1, \ldots, a_n) = 0 | f_e(a_1, \ldots, a_{n-1}) \neq 0].
\]

We bound each term individually. We can bound the probability that \( f_e(a_1, \ldots, a_{n-1}) = 0 \) by induction:

\[
\Pr_{a_1, \ldots, a_{n-1} \in S} [f_e(a_1, \ldots, a_{n-1}) = 0] \leq \frac{\deg(f_e)}{|S|} \leq \frac{d - e}{|S|}.
\]
Next, fix $a_1, \ldots, a_{n-1} \in S$ such that $f_e(a_1, \ldots, a_{n-1}) \neq 0$. The polynomial $f(a_1, \ldots, a_n, x)$ is a univariate polynomial in $x$ of degree $e$, hence it has at most $e$ roots. Thus, for any such fixing of $a_1, \ldots, a_{n-1}$ we have

$$\Pr_{a_n \in S} [f(a_1, \ldots, a_{n-1}, a_n) = 0] \leq \frac{e}{|S|}.$$  

This implies that also

$$\Pr_{a_1, \ldots, a_n \in S} [f(a_1, \ldots, a_n) = 0 | f_e(a_1, \ldots, a_{n-1}) \neq 0] \leq \frac{e}{|S|}.$$  

We conclude that

$$\Pr_{a_1, \ldots, a_n \in S} [f(a_1, \ldots, a_n) = 0] \leq \frac{d - e}{|S|} + \frac{e}{|S|} = \frac{d}{|S|}.$$

\[\square\]

**Corollary 6.6.** Let $f, g$ be two different multivariate polynomials of degree $d$. Fix $\varepsilon > 0$ and let $s \geq d/\varepsilon$. Then

$$\Pr_{a_1, \ldots, a_n \in \{1, \ldots, s\}} [f(a_1, \ldots, a_n) = g(a_1, \ldots, a_n)] \leq \varepsilon.$$

Note that if we have an efficient algebraic circuit which computes $f, g$, then we can run this test efficiently using a randomized algorithm, which evaluates the two circuits on a randomly chosen joint input.

### 6.4 Perfect matchings via polynomial identity testing

We will see an efficient way to find if a bipartite graph has a perfect matching, using polynomial identity testing.

Define the following $n \times n$ matrix: $M_{i,j} = x_{i,j}$ if $(u_i, v_j) \in E$, and $M_{i,j} = 0$ otherwise. The determinant of $M$ is

$$\det(M) = \sum_{\pi \in S_n} (-1)^{\text{sign}(\pi)} \prod_{i=1}^{n} M_{i,\pi(i)}.$$

**Lemma 6.7.** $G$ has a perfect matching iff $\det(M)$ is not the zero polynomial.

**Proof.** Each term $\prod_{i=1}^{n} M_{i,\pi(i)}$ is the monomial $\prod_{i=1}^{n} x_{i,\pi(i)}$ if $\pi$ corresponds to a perfect matching; and it zero otherwise. Moreover, each monomial appears only once, and hence monomials cannot cancel each other. \[\square\]

We got an efficient randomized algorithm to test if a bi-partite graph has a perfect matching: run the polynomial identity testing algorithm on the polynomial $\det(M)$.  

34
Corollary 6.8. Let $\mathbb{F} = \mathbb{F}_p$ be a finite field for $p \geq \frac{n}{\varepsilon}$. Define a randomized $n \times n$ matrix $M$ over $\mathbb{F}$ as follows: if $(i, j) \in E$ then sample $M_{i,j} \in \mathbb{F}$ uniformly, and if $(i, j) \notin E$ then set $M_{i,j} = 0$. Then

- If $G$ has no perfect matchings then always $\det(M) = 0$.
- If $G$ has a perfect matching then $\Pr[\det(M) = 0] \leq \varepsilon$.

The main advantage of the polynomial identity testing algorithm over the “combinatorial” algorithm based on Hall marriage theorem, that we saw before, is that the polynomial identity testing algorithm can be parallelized. It turns out that the determinant can be computed in parallel by $\text{poly}(n)$ processors in time $O(\log^2 n)$, and in particular we can check if a graph has a perfect matching in that time if parallelism is allowed. On that other hand, all implementations of the algorithm based on Hall marriage theorem requires at least $\Omega(n)$ time, even when parallelism is allowed.
7 Satisfiability

Definition 7.1 (CNF formulas). A CNF formula over boolean variables is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals (variables or their negation). A $k$-CNF is a CNF formula where each clause contains exactly $k$ literals.

For example, the following is a 3-CNF with 6 variables:

$$
\varphi(x_1, \ldots, x_6) = (x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor \neg x_3 \lor x_5) \land (\neg x_1 \lor \neg x_2 \lor x_4) \land (x_1 \lor \neg x_2 \lor x_6).
$$

The $k$-SAT problem is the computational problem of deciding whether a given $k$-CNF has a satisfying assignment. Many constraint satisfaction problems can be cast as a $k$-SAT problem, for example: verifying that a chip works correctly, scheduling flights in an airline, routing packets in a network, etc. As we will shortly see, 2-SAT can be solved in polynomial time (in fact, in linear time); however, for $k \geq 3$, the $k$-SAT problem is NP-hard, and the only known algorithms solving it run in exponential time. However, even there we would see that we can improve upon full enumeration (which takes $2^n$ time). We will present an algorithm that solves 3-SAT in time $\approx 2^{0.41n}$. The same algorithm solves $k$-SAT for any $k \geq 3$ in time $2^{c_k n}$ where $c_k < 1$.

Both the polynomial algorithm for 2-SAT and the exponential algorithm for 3-SAT, $k \geq 3$, will be based on a similar idea: analyzing a random walk on the space of possible solutions.

### 7.1 2-SAT

Let $x = (x_1, \ldots, x_n)$ and let $\varphi(x)$ be a 2-SAT given by

$$
\varphi(x) = C_1(x) \land \ldots \land C_m(x),
$$

where each $C_i$ is the OR of two literals. We say that an assignment $x \in \{0,1\}^n$ satisfies a clause $C_i$ if $C_i(x) = 1$. We will analyze the following simple looking algorithm.

<table>
<thead>
<tr>
<th>Function Solve-2SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> 2–CNF $\varphi$</td>
</tr>
<tr>
<td><strong>Output:</strong> $x \in {0,1}^n$ such that $\varphi(x) = 1$</td>
</tr>
</tbody>
</table>

1. Set $x = 0$.
2. While there exists some clause $C_i$ such that $C_i(x) = 0$:
   2.1 Let $x_a, x_b$ be the two variables participating in $C_i$.
   2.2 Choose $\ell \in \{a, b\}$ uniformly: $\Pr[\ell = a] = \Pr[\ell = b] = 1/2$.
   2.3 Flip $x_\ell$.
3. Output $x$.

We will show the following theorem.
Theorem 7.2. If \( \varphi \) is a satisfiable 2-CNF, then with probability at least 1/4 over the internal randomness of the algorithm, it outputs a solution within 4n^2 steps.

We note that if we wish a higher success probability (say, 99%) then we can simply repeat the algorithm a few times, where in each phase, if the algorithm does not find a solution within the first 4n^2 steps, we restart the algorithm. The probability that the algorithm still doesn’t find a solution after \( t \) restarts is at most \((3/4)^t\). So, to get success probability of 99% we need to run the algorithm 9 times (since \((3/4)^9 \leq 1\%\)).

We next proceed to the proof. For the proof, fix some solution \( x^* \) for \( \varphi \) (if there is more than one, choose one arbitrarily). Let \( x^t \) denote the value of \( x \) in the \( t \)-th iteration of the loop. Note that it is a random variable, which depends on our choice of which clause to choose and which variables to flip in the previous steps. Define \( d_t = \text{dist}(x^t, x^*) \) to be the hamming distance between \( x^t \) and \( x^* \) (that is, the number of bits where they differ). Clearly, at any stage \( 0 \leq d_t \leq n \), and if \( d_t = 0 \) then we reached a solution, and we output \( x^t = x^* \) at iteration \( t \).

Consider \( x^t \), the assignment at iteration \( t \), and assume that \( d_t > 0 \). Let \( C = \ell_a \lor \ell_b \) be a violated clause, where \( \ell_a \in \{x_a, \neg x_a\} \) and \( \ell_b \in \{x_b, \neg x_b\} \). This means that either \((x^*)_a \neq (x^t)_a \) or \((x^*)_b \neq (x^t)_b \) (or both), since \( C(x^*) = 1 \) but \( C(x^t) = 0 \). If we choose \( \ell \in \{a, b\} \) such that \((x^t)_\ell \neq (x^*)_\ell \), then the distance between \( x^{t+1} \) and \( x^* \) decreases by one; otherwise, it increases by one. So we have:

\[
d_{t+1} = d_t + \Delta_t,
\]

where \( \Delta_t \in \{−1, 1\} \) is a random variable that satisfies \( \Pr[\Delta_t = −1|x^t] \geq 1/2 \).

Another way to put it, the sequence of distances \( d_0, d_1, d_2, \ldots \) is a random walk on \( \{0, 1, \ldots, n\} \). It starts at some arbitrary location \( d_0 \). In each step, we move to the left (getting closer to 0) with probability \( \geq 1/2 \), and otherwise move to the right (getting further away from \( n \)). We will show that after \( O(n^2) \) steps, with high probability, this has to terminate: either some satisfying assignment has been found, or otherwise we hit 0 and output \( x^* \). We do so by showing that a random walk tends to drift far from its origin.

For simplicity, we first analyze the slightly simpler case where the random walk is symmetric, that is \( \Pr[\Delta_t = −1|x^t] = \Pr[\Delta_t = 1|x^t] = 1/2 \). We will then show how to extend the analysis to our case, where the probability for \( −1 \) could be larger (intuitively, this should only help us get to 0 faster; however proving this formally is a bit technical).

Lemma 7.3. Let \( y_0, y_1, \ldots \) be a random walk, defined as follows: \( y_0 = 0 \) and \( y_{t+1} = y_t + \Delta_t \), where \( \Delta_t \in \{−1, 1\} \) and \( \Pr[\Delta_t = 1|y_t] = 1/2 \) for all \( t \geq 0 \). Then, for any \( t \geq 0 \),

\[
\mathbb{E}[y_t^2] = t.
\]

Proof. We prove this by induction on \( t \). It is clear for \( t = 0 \). We have

\[
\mathbb{E}[y_{t+1}^2] = \mathbb{E}[(y_t + \Delta_t)^2] = \mathbb{E}[y_t^2] + 2\mathbb{E}[y_t\Delta_t] + \mathbb{E}[\Delta_t^2].
\]
By induction, $E[y_t^2] = t$. Since $\Delta_t \in \{-1, 1\}$ we have $E[\Delta_t^2] = 1$. To conclude, we need to show that $E[y_t \Delta_t] = 0$. We show this via the rule of conditional expectations:

$$E[y_t \Delta_t] = E_{y_t}[E_{\Delta_t}[y_t \Delta_t | y_t]] = E_{y_t}[y_t \cdot E_{\Delta_t}[\Delta_t | y_t]] = E_{y_t}[y_t \cdot 0] = 0.$$ 

We now prove the more general lemma, where we allow a consistent drift.

**Lemma 7.4.** Let $y_0, y_1, \ldots$ be a random walk, defined as follows: $y_0 = 0$ and $y_{t+1} = y_t + \Delta_t$, where $\Delta_t \in \{-1, 1\}$ and $Pr[\Delta_t = 1 | y_t] \geq 1/2$ for all $t \geq 0$. Then, for any $t \geq 0$, 

$$E[y_t^2] \geq t/2.$$ 

Note that the same result holds by symmetry if we assume instead that $Pr[\Delta_t = -1 | y_t] \geq 1/2$ for all $t \geq 0$.

**Proof.** The proof is by a “coupling argument”. Define a new random walk $y'_0, y'_1, \ldots$, where $y'_0 = 0, y'_{t+1} = y'_t + \Delta'_t$. In general, we would have that $y'_t, \Delta'_t$ depend on $y_1, \ldots, y_t$. So, fix $y_1, \ldots, y_t$, and assume that $Pr[\Delta_t = 1 | y_t] = \alpha$ for some $\alpha \geq 1/2$. Define $\Delta'_t$ as:

$$\Delta'_t(y_t, \Delta_t) = \begin{cases} 
-1 & \text{if } \Delta_t = -1 \\
1 & \text{if } \Delta_t = 1, \text{ with probability } 1/2\alpha \\
-1 & \text{if } \Delta_t = 1, \text{ with probability } 1 - 1/2\alpha 
\end{cases}$$

It satisfies the following properties:

- $\Delta'_t \leq \Delta_t$.
- $Pr[\Delta'_t = 1 | y_t, \Delta_t] = 1/2$.

Note that the random walk $y'_0, y'_1, \ldots$, is a symmetric random walk, which further satisfies $y'_t \leq y_t$ for all $t \geq 0$. By the previous lemma,

$$E[(y'_t)^2] = t.$$ 

Note moreover that since the random walk $y'_0, y'_1, \ldots$ is symmetric, $Pr[y'_t = a] = Pr[y'_t = -a]$ for any $a \in \mathbb{Z}$. In particular, $Pr[y'_t \geq 0] \geq 1/2$. Whenever $y'_t \geq 0$ we have $y_t^2 \geq (y'_t)^2$. So we have

$$E[y_t^2] \geq E[y_t^2 \cdot 1_{y_t \geq 0}] \geq E[(y'_t)^2 \cdot 1_{y_t \geq 0}] = E[(y'_t)^2]/2 = t/2.$$ 

We return now to the proof of Theorem 7.2. The proof will use Lemma 7.4, but the analysis is more subtle.
Proof of Theorem 7.2. Recall that $x^0, x^1, \ldots$ is the sequence of “guesses” for a solution which the algorithm explores. Let $T \in \mathbb{N}$ denote the random variable of the step at which the algorithm outputs a solution. The challenge in the analysis is that not only the sequence is random, for also $T$ is a random variable. For simplicity of notation later on, set $x^{t} = x^{T}$ for all $t > T$.

Let $y_0, y_1, \ldots$ be defined as $y_t = d_t - d_0$, where recall that $d_t = \text{dist}(x^{t}, x^{*})$. In order to analyze this random walk, we define a new random walk $z_0, z_1, \ldots$ as follows: set $z_t = y_T$ for $t \leq T$, and for $t \geq T$ set $z_{t+1} = z_t + \rho_t$, where $\rho_t \in \{-1, +1\}$ is uniformly and independently chosen. We will argue that the sequence $z_t$ satisfies the conditions of Lemma 7.4. Namely, if we define $\Delta_t = z_{t+1} - z_t$ then $\text{Pr}[\Delta_t = -1 | z_t] \geq 1/2$.

To show this, let us condition on $x^0, \ldots, x^t$. If none of them are solutions to $\varphi$ then $\Delta_t = y_{t+1} - y_t$, and conditioned on $x^0, \ldots, x^t$ not being solutions, we already showed that the probability that $\Delta_t = -1$ is $\geq 1/2$. If, on the other hand, $x^0, \ldots, x^t$ contain a solution, then $\Delta_t = \rho_t$ and the probability that $\Delta_t = -1$ is exactly $1/2$. In either case we have

$$\text{Pr}[\Delta_t = -1 | x^0, \ldots, x^t] \geq 1/2.$$ 

This then implies that

$$\text{Pr}[\Delta_t = -1 | z_t] \geq 1/2.$$ 

Thus, we may apply Lemma 7.4 to the sequence $z_0, z_1, \ldots$ and obtain that for any $t \geq 1$ we have

$$\mathbb{E}[z_t^2] \geq t/2.$$ 

Next, for $t \geq 1$ let $T_t = \min(T, t)$. We may write $z_t$ as

$$z_t = y_{T_t} + \sum_{i=T_t}^{t-1} \eta_i,$$

where the sum is empty if $T_t = t$. In particular, conditioning on $T$ gives that

$$\mathbb{E}[z_t^2 | T] = \mathbb{E} \left[ \left( y_{T_t} + \sum_{i=T_t}^{t-1} \eta_i \right)^2 \bigg| T \right] = \mathbb{E}[y_{T_t}^2 | T] + t - T_t.$$

Averaging over $T$ gives then that

$$\mathbb{E}[z_t^2] = \mathbb{E}[y_{T_t}^2] + t - \mathbb{E}[T_t].$$

Next, observe that any $y_t = d_t - d_0$ is a difference of two numbers between 0 and $n$, and hence $|y_t| \leq n$. In particular, $\mathbb{E}[y_{T_t}^2] \leq n^2$. Thus we have

$$\mathbb{E}[T_t] = \mathbb{E}[y_{T_t}^2] + t - \mathbb{E}[z_t^2] \leq n^2 + t/2.$$ 

Wet $t_0 = 4n^2$. We have

$$\mathbb{E}[T_{t_0}] \leq (3/4)t_0.$$
By Markov inequality we have that
\[ \Pr[T \geq t_0] = \Pr[T_{t_0} = t_0] \leq \Pr[T_{t_0} \geq (4/3)\mathbb{E}[T_{t_0}]] \leq 3/4. \]
So, with probability at least 1/4, we have that \( T < t_0 \), which means the algorithm finds a solution within \( t_0 = 4n^2 \) steps.

## 7.2 3-SAT

Let \( \varphi \) be a 3-CNF. Finding if \( \varphi \) has a satisfying assignment is NP-hard, and the best known algorithms take exponential time. However, we can still improve upon the naive \( 2^n \) full enumeration, as the following algorithm shows. The algorithm we analyze is due to Schoning [Sch99].

Let \( m \geq 1 \) be a parameter to be determined later.

<table>
<thead>
<tr>
<th>Function Solve-3SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: 3-CNF ( \varphi )</td>
</tr>
<tr>
<td><strong>Output</strong>: ( x \in {0,1}^n ) such that ( \varphi(x) = 1 )</td>
</tr>
<tr>
<td>1. Choose ( x \in {0,1}^n ) randomly.</td>
</tr>
<tr>
<td>2. For ( i = 1, \ldots, m ):</td>
</tr>
<tr>
<td>2.1 If ( \varphi(x) = 1 ), output ( x ).</td>
</tr>
<tr>
<td>2.2 Otherwise, let ( C_i ) be some clause such that ( C_i(x) = 0 ), with variables ( x_a, x_b, x_c ).</td>
</tr>
</tbody>
</table>
| 2.2 Choose \( \ell \in \{a,b,c\} \) uniformly: \[
\Pr[\ell = a] = \Pr[\ell = b] = \Pr[\ell = c] = 1/3.
\] |
| 2.3 Flip \( x_\ell \). |
| 3. Output FAIL. |

Our goal is to analyze the following question: what is the success probability of the algorithm? as before, assume \( \varphi \) is satisfiable, and choose some satisfying assignment \( x^* \). Define \( x^t \) to be the value of \( x \) at the \( t \)-th iteration of the algorithm, and let \( d_i = \text{dist}(x^t, x^*) \).

**Claim 7.5.** The following holds

(i) \( \Pr[d_0 = k] = 2^{-n}{n \choose k} \) for all \( 0 \leq k \leq n \).

(ii) \( d_{t+1} = d_t + \Delta_t \) where \( \Delta_t \in \{-1,1\} \) satisfies \( \Pr[\Delta_t = -1|d_t] \geq 1/3 \).

**Proof.** For (i), note that \( d_0 \) is the distance of a random string from \( x^* \), so equivalently, it is the hamming weight of a uniform element of \( \{0,1\}^n \). The number of elements of hamming weight \( k \) is \( {n \choose k} \). For (ii), if \( x_a, x_b, x_c \) are the variables appearing in an unsatisfied clause at iteration \( t \), then at least one of them disagrees with the value of \( x^* \). If we happen to choose it, the distance will decrease by one, otherwise, it will increase by one. 

\[ 40 \]
For simplicity, let’s assume from now on that \( \Pr[\Delta_t = -1|d_t] = 1/3 \), where the more general case can be handled similar to the way we handled it for 2-SAT.

**Claim 7.6.** Assume that \( d_0 = k \). The probability that the algorithm finds a satisfying solution is at least
\[
\left( \frac{m}{(m + k)/2} \right) \left( \frac{1}{3} \right)^{(m-k)/2} \left( \frac{2}{3} \right)^{(m-k)/2}.
\]

*Proof.* Consider the sequence of steps \( \Delta_0, \ldots, \Delta_{m-1} \). If there are \( k \) more \(-1\) than \(+1\) in this sequence, then starting at \( d_0 = k \), we will reach \( d_m = 0 \). The number of such sign sequences is \( \binom{m}{(m+k)/2} \), the probability for seeing a \(-1\) is \( \frac{1}{3} \), and the probability for seeing a \(+1\) is \( \frac{2}{3} \).

**Claim 7.7.** For any \( 0 \leq k \leq n \), the probability that the algorithm finds a solution is at least
\[
2^{-n} \binom{n}{k} \binom{m}{(m + k)/2} \left( \frac{2}{3} \right)^{(m-k)/2}.
\]

*Proof.* We require that \( d_0 = k \) (which occurs with probability \( 2^{-n} \binom{n}{k} \)) and, conditioned on that occurring, apply Claim 7.6.

We now need to optimize parameters. Fix \( k = \alpha n, m = \beta n \) for some constants \( \alpha, \beta > 0 \). We will use the following approximation: for \( n \geq 1 \) and \( 0 < \alpha < 1 \),
\[
\binom{n}{\alpha n} \approx \left( \frac{1}{\alpha} \right)^{\alpha n} \left( \frac{1}{1-\alpha} \right)^{(1-\alpha)n} \approx 2^{H(\alpha)n},
\]
where \( H(\alpha) = \alpha \log_2 \left( \frac{1}{\alpha} \right) + (1-\alpha) \log_2 \left( \frac{1}{1-\alpha} \right) \) is the entropy function. Then
\[
\binom{n}{k} \approx 2^{H(\alpha)n},
\]
\[
\binom{m}{(m + k)/2} \approx 2^{H \left( \frac{1}{2} + \frac{\alpha}{2\beta} \right) \beta n}
\]
\[
\left( \frac{1}{3} \right)^{(m+k)/2} \left( \frac{2}{3} \right)^{(m-k)/2} = 3^{-\beta n} 2^{(\beta-\alpha)/2 \cdot n}.
\]

So, we can express the probability of success as \( \approx 2^{-\gamma n} \), where
\[
\gamma = 1 - H(\alpha) - H \left( \frac{1}{2} + \frac{\alpha}{2\beta} \right) \beta + \log_2 3 \cdot \beta - \frac{\beta-\alpha}{2}.
\]

Our goal is to choose \( 0 < \alpha < 1 \) and \( \beta \geq \alpha \) to minimize \( \gamma \). The minimum is obtained for \( \alpha = 1/3, \beta = 1 \), which gives \( \gamma \approx 0.41 \).

So, we have an algorithm that runs in time \( O(m) = O(n) \), and finds a satisfiable assignment with probability \( \approx 2^{-\gamma n} \). To find a satisfiable assignment with high probability, we simply repeat it \( N = 5 \cdot 2^m \) times. The probability it fails in all these executions is at most
\[
(1 - 2^{-\gamma n})^N \leq \exp(-2^{-\gamma n} N) \leq \exp(-5) \leq 1%.
\]

**Corollary 7.8.** We can solve 3-SAT in time \( O(2^{\gamma n}) \) for \( \gamma \approx 0.41 \).
Hash functions: the power of pairwise independence

Hash functions are used to map elements from a large domain to a small one. They are commonly used in data structures, cryptography, streaming algorithms, coding theory, and more - anywhere where we want to store efficiently a small subset of a large universe. Typically, for many of the applications, we would not have a single hash function, but instead a family of hash functions, where we would randomly choose one of the functions in this family as our hash function.

Let $\mathcal{H} = \{h : U \rightarrow R\}$ be a family of functions, mapping elements from a (typically large) universe $U$ to a (typically small) range $R$. For many applications, we would like two, seemingly contradicting, properties from the family of functions:

- Functions $h \in \mathcal{H}$ should “look random”
- Functions $h \in \mathcal{H}$ are succinctly described, and hence processed and stored efficiently.

The way to resolve this is to be more specific about what do we mean by “looking random”. The following definition is such a concrete realization, which although is quite weak, it is already very useful.

**Definition 8.1** (Pairwise independent hash functions). A family $\mathcal{H} = \{h : U \rightarrow R\}$ is said to be pairwise independent, if for any two distinct elements $x_1 \neq x_2 \in U$, and any two (possibly equal) values $y_1, y_2 \in R$,

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \text{ and } h(x_2) = y_2] = \frac{1}{|R|^2}.$$ 

We investigate the power of pairwise independent hash functions in this chapter, and describe a few applications. For many more applications we recommend the book [LLW06].

8.1 Pairwise independent bits

To simplify notations, let us consider the case of $R = \{0, 1\}$. We also assume that $|U| = 2^k$ for some $k \geq 1$, by possibly increasing the size of the universe by a factor of at most two. Thus, we can identify $U = \{0, 1\}^k$, and identify functions $h \in \mathcal{H}$ with boolean functions $h : \{0, 1\}^k \rightarrow \{0, 1\}$. Consider the following construction:

$$\mathcal{H}_2 = \{h_{a,b}(x) = \langle a, x \rangle + b \pmod{2} : a \in \{0, 1\}^k, b \in \{0, 1\}\}.$$ 

One can check that $|\mathcal{H}_2| = 2^{k+1} = 2|U|$, which is much smaller than the set of all functions from $\{0, 1\}^k$ to $\{0, 1\}$ (which has size $2^{2^k}$). We will show that $\mathcal{H}_2$ is pairwise independent. To do so, we need the following claim.

**Claim 8.2.** Fix a $x \in \{0, 1\}^k$, $x \neq 0^k$. Then

$$\Pr_{a \in \{0, 1\}^k}[(a, x) \pmod{2} = 0] = \frac{1}{2}.$$ 

42
Proof. Fix \( i \in [k] \) such that \( x_i = 1 \). Then
\[
\Pr_{a \in \{0,1\}^k} [(a,x) \pmod{2} = 0] = \Pr \left[ a_i = \sum_{j \neq i} a_j x_j \pmod{2} \right] = \frac{1}{2}.
\]

**Lemma 8.3.** \( \mathcal{H}_2 \) is pairwise independent.

**Proof.** Fix distinct \( x_1, x_2 \in \{0,1\}^k \) and (not necessarily distinct) \( y_1, y_2 \in \{0,1\} \). In all the calculations below of \( \langle a, x \rangle + b \), we evaluate the result modulo 2. We need to prove
\[
\Pr_{a \in \{0,1\}^k, b \in \{0,1\}} [(a,x_1) + b = y_1 \text{ and } (a,x_2) + b = y_2] = \frac{1}{4}.
\]
If we just randomized over \( a \) then by Claim 8.2, for any \( y \in \{0,1\} \) we have
\[
\Pr_{a} [(a,x_1) \oplus (a,x_2) = y] = \Pr_{a} [(a,x_1 \oplus x_2) = y] = \frac{1}{2}.
\]
Randomizing also over \( b \) gives us the desired result:
\[
\Pr_{a,b} [(a,x_1) + b = y_1 \text{ and } (a,x_2) + b = y_2] = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}.
\]

Next, we describe an alternative viewpoint, which will justify the name ”pairwise independent bits”.

**Definition 8.4** (Pairwise independent bits). A distribution \( D \) over \( \{0,1\}^n \) is said to be pairwise independent, if for any distinct \( i,j \in [n] \) and any \( y_1, y_2 \in \{0,1\} \) we have
\[
\Pr_{x \sim D} [x_i = y_1 \text{ and } x_j = y_2] = \frac{1}{4}.
\]
We note that we can directly use \( \mathcal{H}_2 \) to generate pairwise independent bits. Assume that \( n = 2^k \). Identify \( h \in \mathcal{H}_2, h : \{0,1\}^k \to \{0,1\} \), with the string \( u_h \) in \( \{0,1\}^n \) giving by concatenating all the evaluations of \( h \) is some pre-fixed order. Let \( D \) be the distribution over \( \{0,1\}^n \) obtained by sampling \( h \in \mathcal{H}_2 \) uniformly and outputting \( u_h \). Then the condition that \( D \) is pairwise independent is equivalent to that of \( \mathcal{H}_2 \) be pairwise independent. Note that the construction above gives a distribution \( D \) supported on \( |\mathcal{H}_2| = 2n \) elements in \( \{0,1\}^n \), much less than the full space. In particular, we can represent a string \( u \) in the support of \( D \) by specifying the hash function which generated it, which only requires \( \log |\mathcal{H}| = \log n + 1 \) bits.

**Example 8.5.** Let \( n = 4 \). A uniform string from the following set of \( 8 = 2^3 \) strings is pairwise independent:
\[
\{0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111\}.
\]
8.2 Application: de-randomized MAXCUT

Let \( G = (V, E) \) be a simple undirected graph. For \( S \subset V \) let \( E(S, S^c) = \{(u, v) \in E : u \in S, v \in S^c\} \) be the number of edges which cross the cut \( S \). The MAXCUT problem asks to find the maximal number of edges in a cut.

\[
\text{MAXCUT}(G) = \max_{S \subset V} |E(S, S^c)|.
\]

Computing the MAXCUT of a graph is known to be NP-hard. Still, there is a simple randomized algorithm which approximates it within factor 2. Below, let \( V = \{v_1, \ldots, v_n\} \).

**Lemma 8.6.** Let \( x_1, \ldots, x_n \in \{0, 1\} \) be uniformly and independently chosen. Set \( S = \{v_i : x_i = 1\} \).

Then

\[
\mathbb{E}_S[|E(S, S^c)|] \geq \frac{|E(G)|}{2} \geq \frac{\text{MAXCUT}(G)}{2}.
\]

**Proof.** For any choice of \( S \) we have

\[
|E(S, S^c)| = \sum_{(v_i, v_j) \in E} 1_{v_i \in S} 1_{v_j \in S^c}.
\]

Note that every undirected edge \( \{u, v\} \) in \( G \) is actually counted twice in the calculation above, once as \( (u, v) \) and once as \( (v, u) \). However, clearly at most one of these is in \( E(S, S^c) \).

By linearity of expectation, the expected size of the cut is

\[
\mathbb{E}_S[|E(S, S^c)|] = \sum_{(v_i, v_j) \in E} \mathbb{E}[1_{v_i \in S} 1_{v_j \in S^c}] = \sum_{(v_i, v_j) \in E} \mathbb{E}[1_{x_i = 1} 1_{x_j = 0}]
\]

\[
= \sum_{(v_i, v_j) \in E} \Pr[x_i = 1 \text{ and } x_j = 0] = 2|E(G)| \cdot \frac{1}{4} = \frac{|E(G)|}{2}.
\]

This implies that a random choice of \( S \) has a non-negligible probability of giving a 2-approximation.

**Corollary 8.7.** \( \Pr_S\left[|E(S, S^c)| \geq \frac{|E(G)|}{2}\right] \geq \frac{1}{2|E(G)|} \geq \frac{1}{n^2} \).

**Proof.** Let \( X = |E(S, S^c)| \) be a random variable counting the number of edges in a random cut. Let \( \mu = |E(G)|/2 \), where we know that \( \mathbb{E}[X] \geq \mu \). Note that whenever \( X < \mu \), we in fact have that \( X \leq \mu - 1/2 \), since \( X \) is an integer and \( \mu \) a half-integer. Also, note that always \( X \leq |E(G)| \leq 2\mu \). Let \( p = \Pr[X \geq \mu] \). Then

\[
\mathbb{E}[X] = \mathbb{E}[X|X \geq \mu] \Pr[X \geq \mu] + \mathbb{E}[X|X \leq \mu - 1/2] \Pr[X \leq \mu - 1/2]
\]

\[
\leq 2\mu \cdot p + (\mu - 1/2) \cdot (1 - p)
\]

\[
\leq \mu - 1/2 + 2\mu p.
\]

So we must have \( 2\mu p \geq 1/2 \), which means that \( p \geq 1/(4\mu) \geq 1/(2|E(G)|) \).
In particular, we can sample $O(n^2)$ sets $S$, compute for each one its cut size, and we are guaranteed that with high probability, the maximum will be at least $|E(G)|/2$.

Next, we derandomize this randomized algorithm using pairwise independent bits. As a side benefit, it will reduce the computation time from testing $O(n^2)$ sets to testing only $O(n)$ sets.

**Lemma 8.8.** Let $x_1, \ldots, x_n \in \{0, 1\}$ be pairwise independent bits (such as the ones given by $H_2$). Set

$$S = \{v_i : x_i = 1\}.$$ 

Then

$$\mathbb{E}_S [|E(S, S^c)| \geq \frac{|E(G)|}{2}].$$

**Proof.** The only place where we used the fact that the bits were uniform in the proof of Lemma 8.6, was in the calculation

$$\Pr[x_i = 1 \text{ and } x_j = 0] = \frac{1}{4}$$

for all distinct $i,j$. However, this is also true for pairwise independent bits (by definition). \[ \square \]

In particular, for one of the $O(n)$ sets $S$ that we generate in the algorithm, we must have that $|E(S, S^c)|$ exceeds the average, and hence $|E(S, S^c)| \geq |E(G)|/2$.

### 8.3 Optimal sample size for pairwise independent bits

The previous application showed the usefulness of having small sample spaces for pairwise independent bits. We saw that we can generate $O(n)$ binary strings of length $n$, such that choosing one of them uniformly gives us pairwise independent bits. We next show that this is optimal.

**Lemma 8.9.** Let $X \subset \{0, 1\}^n$ and let $D$ be a distribution supported on $X$. Assume that $D$ is pairwise independent. Then $|X| \geq n$.

**Proof.** Let $X = \{x_1, \ldots, x_m\}$ and let $p_i = \Pr[D = x^i]$. For any $i \in [n]$, we construct a vector $v_i \in \mathbb{R}^m$ as follows:

$$(v_i)_\ell = \sqrt{p_\ell} \cdot (-1)^{(x_\ell)_i}.$$  

We will show that the set of vectors $\{v_1, \ldots, v_n\}$ are linearly independent in $\mathbb{R}^m$, and hence we must have $|X| = m \geq n$.

As a first step, we show that $\langle v_i, v_j \rangle = 0$ for all $i \neq j$:

$$\langle v_i, v_j \rangle = \sum_{\ell=1}^m p_\ell \cdot (-1)^{(x_\ell)_i + (x_\ell)_j} = \mathbb{E}_{x \sim D} [(-1)^{x_i + x_j}]$$

$$= \Pr_{x \sim D}[x_i + x_j = 0 \ (\text{mod } 2)] - \Pr_{x \sim D}[x_i + x_j = 1 \ (\text{mod } 2)]$$

$$= \frac{1}{2} - \frac{1}{2} = 0.$$
Next, we show that this implies that $v_1, \ldots, v_n$ must be linearly independent. Assume towards contradiction that this is not the case. That is, there exist coefficients $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$, not all zero, such that
\[
\sum \alpha_i v_i = 0.
\]
However, for any $j \in [n]$, we have
\[
0 = \langle \sum \alpha_i v_i, v_j \rangle = \sum \alpha_i \langle v_i, v_j \rangle = \alpha_j \|v_j\|^2 = \alpha_j.
\]
So $\alpha_j = 0$ for all $j$, a contradiction. Hence, $v_1, \ldots, v_n$ are linearly independent and hence $|X| = m \geq n$.

### 8.4 Hash functions with large ranges

We now consider the problem of constructing a family of hash functions $\mathcal{H} = \{h : U \to R\}$ for large $R$. For simplicity, we will assume that $|R|$ is prime, although this requirement can be somewhat removed. So, let's identify $R = \mathbb{F}_p$ for a prime $p$. We may assume that $|U| = p^k$, by possibly increase the size of the universe by a factor $p$. This allows us to identify $U = \mathbb{F}_p^k$.

Define the following family of hash functions:
\[
\mathcal{H}_p = \{h_{a,b}(x) = \langle a, x \rangle + b : a \in \mathbb{F}_p^k, b \in \mathbb{F}_p\}.
\]

Note that $|\mathcal{H}| = p^{k+1} = |U| \cdot |R|$, and observe that for $p = 2$ this coincides with our previous definition of $\mathcal{H}_2$. We will show that $\mathcal{H}_p$ is pairwise independent. As before, we need an auxiliary claim first.

**Claim 8.10.** Let $x \in \mathbb{F}_p^k, x \neq 0$. Then for any $y \in \mathbb{F}_p$,
\[
\Pr_{a \in \mathbb{F}_p^k} \left[ \langle a, x \rangle = y \right] = \frac{1}{p}.
\]

**Proof.** Fix $i \in [k]$ such that $x_i \neq 0$. Then
\[
\Pr_{a \in \mathbb{F}_p^k} \left[ \langle a, x \rangle = y \right] = \Pr \left[ a_i x_i = y - \sum_{j \neq i} a_j x_j \right].
\]

Now, for every fixing of $\{a_j : j \neq i\}$, we have that $a_i x_i$ is uniformly distributed in $\mathbb{F}_p$, hence the probability that it equals any specific value is exactly $1/p$.

**Lemma 8.11.** $\mathcal{H}_p$ is pairwise independent.

**Proof.** Fix distinct $x_1, x_2 \in \mathbb{F}_p^k$ and (not necessarily distinct) $y_1, y_2 \in \mathbb{F}_p$. All the calculations below of $\langle a, x \rangle + b$ are in $\mathbb{F}_p$. We need to show
\[
\Pr_{a \in \mathbb{F}_p^k, b \in \mathbb{F}_p} \left[ \langle a, x_1 \rangle + b = y_1 \text{ and } \langle a, x_2 \rangle + b = y_2 \right] = \frac{1}{p^2}.
\]
If we just randomized over $a$ then by the claim, then for any $y \in \mathbb{F}_p$ by the claim,

$$\Pr_a[\langle a, x_1 \rangle - \langle a, x_2 \rangle = y] = \Pr_a[\langle a, x_1 - x_2 \rangle = y] = \frac{1}{p}.$$ 

Randomizing also over $b$ gives us the desired result.

$$\Pr_{a,b}[\langle a, x_1 \rangle + b = y_1 \text{ and } \langle a, x_2 \rangle + b = y_2] =$$

$$\Pr_{a,b}[\langle a, x_1 \rangle - \langle a, x_2 \rangle = y_1 - y_2 \text{ and } b = \langle a, x_1 \rangle + y_1] =$$

$$\frac{1}{p} \cdot \frac{1}{p} = \frac{1}{p^2}.$$ 

\[\square\]

### 8.5 Application: collision free hashing

Let $S \subset U$ be a set of objects. A hash function $h : U \to R$ is said to be collision free for $S$ if it is injective on $S$. That is, $h(x) \neq h(y)$ for all distinct $x, y \in S$. We will show that if $R$ is large enough, then any pairwise independent hash family contains many collision free hash functions for any small set $S$. This is extremely useful: it allows to give lossless compression of elements from a large universe to a small range.

**Lemma 8.12.** Let $\mathcal{H} : \{ h : U \to R \}$ be a pairwise independent hash family. Let $S \subset U$ be a set of size $|S|^2 \leq |R|$. Then

$$\Pr_{h \in \mathcal{H}}[h \text{ is collision free for } S] \geq \frac{1}{2}.$$ 

**Proof.** Let $h \in \mathcal{H}$ be uniformly chosen, and let $X$ be a random variable that counts the number of collisions in $S$. That is,

$$X = \sum_{\{x,y\} \subset S} 1_{h(x) = h(y)}.$$ 

The expected value of $X$ is

$$\mathbb{E}[X] = \sum_{\{x,y\} \subset S} \Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \binom{|S|}{2} \frac{1}{|R|} \leq \frac{|S|^2}{2|R|} \leq \frac{1}{2}.$$ 

By Markov’s inequality,

$$\Pr_{h \in \mathcal{H}}[h \text{ is not collision free for } S] = \Pr[X \geq 1] \leq \frac{1}{2}.$$ 

\[\square\]
8.6 Efficient dictionaries: storing sets efficiently

We now show how to use pairwise independent hash functions, in order to design efficient dictionaries. Fix a universe $U$. For simplicity, we will assume that for any $R$ we have a family of pairwise independent hash functions $\mathcal{H} = \{ h : U \rightarrow R \}$, and note that while our previous constructions required $R$ to be prime (or in fact, a prime power), this will at most double the size of the range, which at the end will only change our space requirements by a constant factor.

Given a set $S \subset U$ of size $|S| = n$, we would like to design a data structure which supports queries of the form “is $x \in S$?”. Our goal will be to do so, while minimizing both the space (memory) requirements and the time it takes to answer a query. If we simply store the set as a sorted list of $n$ elements, then the space (memory) requirements are $O(n \log |U|)$, and each query takes time $O(\log |U| + \log n)$, by doing a binary search on the list. We will see that these can be improved via hashing.

First, consider the following simple hashing scheme. Fix a range $R = \{1, \ldots, n^2\}$. Let $\mathcal{H} = \{ h : U \rightarrow [n^2]\}$ be a pairwise independent hash function. We showed that a randomly chosen $h \in \mathcal{H}$ will be collision free on $S$ with probability at least $1/2$. So, we can sample $h \in \mathcal{H}$ until we find such an $h$, which on average would require two attempts. Let $A$ be an array of length $n^2$. It will be mostly empty, except that we set $A[h(x)] = x$ for all $x \in S$. Now, to check whether $x \in S$, we compute $h(x)$ and check whether $A[h(x)] = x$ or not. Thus, the query time is only $O(\log |U|)$. However, we pay in our space requirements are big: to store $n$ elements, we maintain an array of size $n^2$, which requires at least $n^2$ bits (and possibly even $O(n^2 \log |U|)$, depending on how efficient we are in storing the empty cells).

We describe a two-step hashing scheme due to Fredman Komlós and Szemerédi [FKS84] which avoids this large waste of space. It will use only $O(n \log n + \log |U|)$ space, but would still allow for query time of $O(\log |U|)$. As a preliminary step, we apply the collision free hash scheme we just described. So, we may assume from now on that $U = O(n^2)$ and that $S \subset U$ has size $|S| = n$.

**Step 1.** We first find a hash function $h : U \rightarrow [n]$ which has only $n$ collisions. Let $\text{Coll}(h, S)$ denote the number of collisions of $h$ for $S$, namely

$$\text{Coll}(h, S) = \left| \{ \{x, y\} \subset S : h(x) = h(y) \} \right|.$$ 

If $\mathcal{H} = \{ h : U \rightarrow [n]\}$ is a family of pairwise independent hash functions, then

$$\mathbb{E}_{h \in \mathcal{H}}[\text{Coll}(h, S)] = \sum_{\{x, y\} \subset S} \Pr[h(x) = h(y)] = \binom{|S|}{2} \frac{1}{n} \leq \frac{|S|^2}{2n} \leq \frac{n}{2}.$$ 

By Markov’s inequality, we have

$$\Pr_{h \in \mathcal{H}}[\text{Coll}(h, S) \geq n] \leq 1/2.$$ 

So, after on average two iterations of randomly choosing $h \in \mathcal{H}$, we find such a function $h : U \rightarrow [n]$ such that $\text{Coll}(h, S) \leq n$. We fix it from now on. Note that it is represented using only $O(\log n)$ bits.
Step 2. Next, for any $i \in [n]$ let $S_i = \{x \in S : h(x) = i\}$. Observe that $\sum |S_i| = n$ and

$$\sum_{i=1}^{n} \left( \frac{|S_i|}{2} \right) = \text{Coll}(h, S) \leq n.$$ 

Let $n_i = |S_i|^2$. Note that $\sum n_i = 2\text{Coll}(h, S) + \sum |S_i| \leq 3n$. We will find hash functions $h_i : U \rightarrow [n_i]$ which are collision free on $S_i$. Choosing a uniform hash function from a pairwise independent set of hash functions $\mathcal{H}_i = \{h : U \rightarrow [n_i]\}$ succeeds on average after two samples. So, we only need $O(n)$ time in total (on expectation) to find these functions. As each $h_i$ requires $O(\log n)$ bits to be represented, we need in total $O(n \log n)$ to represent all of them.

Let $A$ be an array of size $3n$. Let $\text{offset}_i = \sum_{j<i} n_j$. The sub-array $A[\text{offset}_i : \text{offset}_i + n_i]$ will be used to store the elements of $S_i$. Initially $A$ is empty. We set

$$A[\text{offset}_i + h_i(x)] = x \quad \forall x \in S_i.$$ 

Note that there are no collisions in $A$, as we are guaranteed that $h_i$ are collision free on $S_i$. We will also keep the list of $\{\text{offset}_i : i \in [n]\}$ in a separate array.

Query. To check whether $x \in S$, we do the following:

- Compute $i = h(x)$.
- Read $\text{offset}_i$.
- Check if $A[\text{offset}_i + h_i(x)] = x$ or not.

This can be computed using $O(\log n)$ bit operations.

Space requirements. The hash functions $h$ requires $O(\log n)$ bits. The hash functions $\{h_i : i \in [n]\}$ require $O(n \log n)$ bits. The array $A$ requires $O(n \log n)$ bits.

Setup time. The setup algorithm is randomized, as it needs to find good hash functions. It has expected running time is $O(n \log n)$ bit operations.

- To find $h$ takes $O(n \log n)$ time, as this is how long it takes to verify that it is collision free.
- To find each $h_i$ takes $O(|S_i| \log n)$ time, and in total it is $O(n \log n)$ time.
- To set up the arrays of $\{\text{offset}_i : i \in [n]\}$ and $A$ takes $O(n \log n)$ time.

RAM model vs bit model. Up until now, we counted bit operations. However, computers can operate on words efficiently. A model for that is the RAM model, where we can perform basic operations on log $n$-bit words. In this model, it can be verified that the query time is $O(1)$ word operations, space requirements are $O(n)$ words and setup time is $O(n)$ word operations.
8.7 Bloom filters

Bloom filters allow for even more efficient data structures for set membership, if some errors are allowed. Let $U$ be a universe, $S \subset U$ a subset of size $|S| = n$. Let $h: U \rightarrow [m]$ be a uniform hash function, for $m$ to be determined later. The data structure maintains a bit array $A$ of length $m$, initially set to zero. Then, for every $x \in S$, we set $A[h(x)] = 1$. In order to check if $x \in S$, we compute $h(x)$, read the value $A[h(x)]$ we answer “yes” if $A[h(x)] = 1$ and “no” otherwise. This has the following guarantees:

- No false negative: if $x \in S$ we will always say “yes”.
- Few false positives: if $x \notin S$, we will say “yes” with probability $\frac{|\{i: A[i] = 1\}|}{m}$, assuming $h$ is a fully random function.

So, if for example we set $m = 2n$, then the probability for $x \notin S$ that we say ”no” it at least $1/2$. In fact, the probability is greater, since when hashing $n$ elements to $2n$ values there will be some collisions, and so the number of 1’s in the array will be less than $n$. It turns out that to get probability $1/2$ we only need $m \approx 1.44n$. This is since, the probability that $A[i] = 1$, over the choice of $h$, is

$$\Pr_h[A[i] = 0] = \Pr[h(x) \neq i, \forall x \in S] = \left(1 - \frac{1}{m}\right)^n \approx e^{-n/m}.$$

So, for $m = n/\ln(2) \approx 1.44n$, the expected number of 0’s in $A$ is $m/2$.

Note that a bloom filter uses $O(n)$ bits, which is much less than the $O(n \log |U|)$ bits we needed when we did not allow for any errors. It can be shown that we don’t need $h$ to be uniform (which can be costly to store); a $k$-wise independent hash family (an extension of pairwise independent) for $k = O(\log n)$ suffices, and such a function can be stored using only $O(\log^2 n)$ bits.
9 Min cut

Let $G = (V, E)$ be a graph. A cut in this graph is $|E(S, S^c)|$ for some $S \subset V$, that is, the number of edges which cross a partition of the vertices. Finding the maximum cut is NP-hard, and the best algorithms solving it run in exponential time. We saw an algorithm which finds a 2-approximation for the max-cut. However, finding the minimum cut turns out to be solvable in polynomial time. Today, we will see a randomized algorithm due to Karger [Kar93] which achieves this is a beautiful way. To formalize this, the algorithm will find

$$\text{min-cut}(G) = \min_{\emptyset \neq S \subseteq V} |E(S, S^c)|.$$

9.1 Karger’s algorithm

The algorithm is very simple: choose a random edge and contract it. Repeat $n - 1$ times, until only two vertices remain. Output this as a guess the for min-cut. We will show that this outputs the min-cut with probability at least $2/n^2$, hence repeating it $3n^2$ times (say) will yield a min-cut with probability at least $(1 - 2/n^2)^{3n^2} \geq 99\%$.

Formally, to define a contraction we need to allow graphs to have parallel edges.

**Definition 9.1** (edge contraction). Let $G = (V, E)$ be an undirected graph with $|V| = n$ vertices, potentially with parallel edges. For an edge $e = (u, v) \in E$, the contraction of $G$ along $e$ is an undirected graph on $n - 1$ vertices, where we merge $u, v$ to be a single node, and delete any self-loops that may be created in this process.

We can now present the algorithm.

<table>
<thead>
<tr>
<th>Karger</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Undirected graph $G = (V, E)$ with $</td>
</tr>
<tr>
<td><strong>Output</strong>: Cut in $G$</td>
</tr>
</tbody>
</table>

1. Let $G_n = G$.
2. For $i = n, \ldots, 3$ do:
   1. Choose a uniform edge $e_i \in G_i$.
   2. Set $G_{i-1}$ to be the contraction of $G_i$ along $e_i$.
3. Output the cut in $G$ corresponding to $G_2$.

Next, we proceed to analyze the algorithm. We start with a few observations. By contracting along an edge $e = (u, v)$, we make a commitment that $u, v$ belong to the same side of the cut, so we restrict the number of potential cuts. Hence, any cut of $G_i$ is a cut of $G$, and in particular

$$\text{min-cut}(G) \leq \text{min-cut}(G_i), \quad i = n, \ldots, 2.$$
In order to analyze the algorithm, let’s fix from now on a min-cut $S \subset V(G)$. We will analyze the probability that the algorithm never chooses an edge in $S$. Hence, after $n-2$ contractions, the output will be exactly the cut $S$. First, we bound the min-cut by cuts for which one side is a single vertex.

**Claim 9.2.** $\min\text{-cut}(G) \leq \min_{v \in G} \deg(v) \leq 2\frac{|E|}{|V|}$.

*Proof.* For any vertex $v \in V$, we can form a cut by taking $E(\{v\}, V \setminus \{v\})$. It has $\deg(v)$ many edges. Since $2|E| = \sum_{v \in V} \deg(v)$ we can bound

$$\min\text{-cut}(G) \leq \min_{v \in V} \deg(v) \leq 2\frac{|E|}{|V|}.$$ 

\[ \square \]

**Claim 9.3.** Let $G = (V, E)$ be a graph, $S \subset V$ be a minimal cut in $G$. Let $e \in E$ be a uniform chosen edge. Then

$$\Pr_{e \in E}[e \in E(S, S^c)] \leq 2\frac{|E(S, S^c)|}{|V|}.$$ 

*Proof.* We saw that $|E(S, S^c)| \leq 2|E|/|V|$. So, the probability that $e$ is in the cut is

$$\Pr_{e \in E}[e \in E(S, S^c)] \leq \frac{|E(S, S^c)|}{|E|} \leq 2\frac{|E|}{|V|} = 2\frac{|V|}{|V|}.$$ 

\[ \square \]

**Theorem 9.4.** For any min-cut $S$ in $G$, $\Pr[\text{algorithm outputs } S] \geq \frac{1}{\binom{n}{2}} \geq \frac{2}{n^2}$.

*Proof.* Fix a min-cut $S \subset V$. The algorithm outputs $S$ if it never chooses an edge in $S$. So

$$\Pr[\text{algorithm outputs } S] = \Pr[e_n, \ldots, e_3 \notin E(S, S^c)]$$

$$= \prod_{i=3}^{3} \Pr[e_i \notin E(S, S^c)|e_n, \ldots, e_{i+1} \notin E(S, S^c)].$$

To analyze this, assume that $e_n, \ldots, e_{i+1} \notin E(S, S^c)$, so that $S$ is still a cut in $G_i$. Thus, it is also a min-cut in $G_i$, and we proved that in this case, $Pr[e_i \in E(S, S^c)] \leq 1/2|V(G_i)|$ = $1/2i$. So

$$\Pr[\text{algorithm outputs } S] \geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\ldots\left(1 - \frac{2}{3}\right)$$

$$= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3}$$

$$= \frac{2}{n(n-1)}.$$ 

\[ \square \]

We get a nice corollary: the number of min-cuts in $G$ is at most $\binom{n}{2}$. Can you prove it directly?

52
9.2 Improving the running time

The time it takes to find a min-cut by Karger’s algorithm described above is $O(n^4)$: we need $O(n^2)$ iterations to guarantee success with high probability. Every iteration requires $n-2$ rounds of contraction, and every contraction takes $O(n)$ time. We will see how to improve this running time to $O(n^2 \log n)$ due to Karger and Stein [KS96]. The main observation guiding this is that most of the error comes when the graph becomes rather small; however, when the graphs are small, the running time is also smaller. So, we can allow to run multiple instances for smaller graphs, which would boost the success probability without increasing the running time too much.

<table>
<thead>
<tr>
<th>Fast-Karger</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Undirected graph $G = (V,E)$ with $</td>
</tr>
<tr>
<td><strong>Output</strong>: Cut in $G$</td>
</tr>
<tr>
<td>0. If $n = 2$ output the only possible cut.</td>
</tr>
<tr>
<td>1. Let $G_n = G$ and $m = n/\sqrt{2}$.</td>
</tr>
<tr>
<td>2. For $i = n, \ldots, m+1$ do:</td>
</tr>
<tr>
<td>2.1 Choose a uniform edge $e_i \in G_i$.</td>
</tr>
<tr>
<td>2.2 Set $G_{i-1}$ to be the contraction of $G_i$ along $e_i$.</td>
</tr>
<tr>
<td>3. Run recursively $S_i = \text{Fast-Karger}(G_i)$ for $i = 1, 2$.</td>
</tr>
<tr>
<td>4. Output $S \in {S_1, S_2}$ which minimizes $E(S, S^c)$.</td>
</tr>
</tbody>
</table>

**Claim 9.5.** Fix a cut $S$. Run the original Karger algorithm, but output $G_m$ for $m = n/\sqrt{2}$. Then the probability that $S$ is still a cut in $G_m$ is at least $1/2$.

**Proof.** Repeating the analysis we did, but stopping once we reach $G_m$, gives
\[
\Pr[S \text{ is a cut of } G_m] = \Pr[e_n, \ldots, e_{m+1} \notin E(S, S^c)] \geq \frac{n-2}{n} \cdot \ldots \cdot \frac{m-1}{m+1} = \frac{m(m-1)}{n(n-1)}.
\]
So, if we set $m = n/\sqrt{2}$, this probability is at least $1/2$. \hfill $\Box$

**Theorem 9.6.** Fast-Karger runs in time $O(n^2 \log n)$ and returns a minimal cut with probability at least $1/4$.

**Proof.** We first analyze the success probability. Let $P(n)$ be the probability that the algorithm succeeds on graphs of size $n$. We will prove by induction that $P(n) \geq 1/4$. We proved that
\[
\Pr[\text{min-cut}(G_m) = \text{min-cut}(G)] \geq \frac{1}{2}.
\]
So we have
\[
P(n) \geq \frac{1}{2} \cdot (1 - P(m))^2 \geq \frac{1}{2} \cdot \left(\frac{3}{4}\right)^2 = \frac{9}{32} \geq \frac{1}{4},
\]
53
We next analyze the running time. Let $T(n)$ be the running time of the algorithm on graphs of size $n$. Then

$$T(n) = 2 \cdot T(n / \sqrt{2}) + O(n^2).$$

This solves to $T(n) = O(n^2 \log n)$. 

$\square$
10 Routing

Let $G = (V, E)$ be an undirected graph, where nodes represent processors and edges represent communication channels. Each node wants to send a message to another node: $v \rightarrow \pi(v)$, where $\pi$ is some permutation on the vertices. However, messages can only traverse on edges, and each edge can only carry one message at a given time unit. A routing scheme is a method of deciding on paths for the messages obeying these restrictions, which tries to minimize the time it takes for all messages to reach their destination. If more than one packet needs to traverse an edge, then only one packet does so at any unit time, and the rest are queued for later time steps. The order of sending the remaining packets does not matter much. For example, you can assume a FIFO (First In First Out) queue on every edge.

Here, we will focus on the hypercube graph $H$, which is a common graph used in distributed computation.

**Definition 10.1 (Hypercube graph).** The hypercube graph $H_n = (V, E)$ has vertices corresponding to all $n$-bit strings $V = \{0, 1\}^n$ and edges which correspond to bit flips,

$$E = \{(x, x \oplus e_i) : x \in \{0, 1\}^n, i \in [n]\},$$

where $e_i$ is the $i$-th unit vector, and $\oplus$ is bitwise xor.

An oblivious routing scheme is a scheme where the path of sending $v \rightarrow \pi(v)$ depends just on the endpoints $v, \pi(v)$, and not on the targets of all other messages. Such schemes are easy to implement, as each node $v$ can compute them given only their local knowledge of their target $\pi(v)$. A very simple one for the hypercube graph is the “bit-fixing scheme”: in order to route $v = (v_1, \ldots, v_n)$ to $u = (u_1, \ldots, u_n)$, we flip the bits in order whenever necessary. So for example, the path from $v = 10110$ to $u = 00101$ is

$$10110 \rightarrow 00110 \rightarrow 00100 \rightarrow 00101.$$

We denote by $P_{\text{fix}}(v, u)$ the path from $v$ to $u$ according to the bit-fixing routing scheme.

10.1 Deterministic routing is bad

Although the maximal distance between pairs of vertices in $H$ is $n$, routing based on the bit-fixing scheme can incur a very large overhead, due to the fact that edges can only carry one message at a time.

**Lemma 10.2.** There are permutations $\pi : \{0, 1\}^n \rightarrow \{0, 1\}^n$ for which the bit-fixing scheme requires at least $2^n/n$ time steps to transfer all messages.

**Proof.** Assume $n$ is even, and write $x \in \{0, 1\}^n$ as $x = (x', x'')$ with $x', x'' \in \{0, 1\}^n$. Consider any permutation $\pi : \{0, 1\}^n \rightarrow \{0, 1\}^n$ which maps $(x', 0)$ to $(0, x')$ for all $x' \in \{0, 1\}^{n/2}$. These $2^{n/2}$ paths all pass through a single vertex $(0, 0)$. As it has only $n$ outgoing edges, we need at least $2^{n/2}/n$ time steps to send all these packages. □
In fact, a more general theorem is true, which shows that any deterministic oblivious routing scheme is equally bad. Here, a deterministic oblivious routing scheme is any scheme in which if \( \pi(v) = u \) then the path from \( v \) to \( u \) depends only on \( v, u \), and more is decided in some deterministic fixed way.

**Theorem 10.3.** For any deterministic routing scheme, there exists a permutation \( \pi : \{0, 1\}^n \to \{0, 1\}^n \) which requires at least \( \frac{2^n}{\sqrt{n}} \) time steps.

We will not prove this theorem. Instead, we will see how randomization can greatly enhance performance.

### 10.2 Solution: randomized routing

We will consider the following oblivious routing scheme, which we call the *two-step bit-fixing scheme*. It uses randomness on top of the deterministic bit-fixing routing scheme described earlier.

**Definition 10.4 (Two-step bit fixing scheme).** In order to route a packet from a source \( v \in \{0, 1\}^n \) to a target \( \pi(v) \in \{0, 1\}^n \) do the following:

(i) Sample uniformly an intermediate target \( t(v) \in \{0, 1\}^n \).

(ii) Follow the path \( P_{\text{fix}}(v, t(v)) \).

(iii) Follow the path \( P_{\text{fix}}(t(v), \pi(v)) \).

Observe that \( t : \{0, 1\}^n \to \{0, 1\}^n \) is not necessarily a permutation, as each \( t(v) \) is sampled independently, and hence there could be collisions. Still, we prove that with very high probability, all packets will be delivered in linear time.

**Theorem 10.5.** With probability \( \geq 1 - 2^{-(n-1)} \), all packets will be routed to their destinations in at most \( 14n \) time steps.

In preparation for proving it, we first prove a few results about the deterministic bit-fixing routing scheme.

**Claim 10.6.** Let \( v, v', u, u' \in \{0, 1\}^n \). Let \( P = P_{\text{fix}}(v, u) \) and \( P' = P_{\text{fix}}(v', u') \). If the paths separate at some point, they never re-connect. That is, let \( w_1, \ldots, w_m \) be the vertices of \( P \) and let \( w'_1, \ldots, w'_m' \) be the vertices of \( P' \). Assume that \( w_i = w'_j \) but \( w_{i+1} \neq w'_{j+1} \). Then

\[ w_\ell \neq w'_\ell' \quad \forall \ell \geq i+1, \ell' \geq j+1. \]

**Proof.** By assumption \( w_i = w'_j \) and \( w_{i+1} \neq w'_{j+1} \). Then \( w_{i+1} = w_i \oplus e_a \) and \( w'_{j+1} = w'_j \oplus e_b \) with \( a \neq b \). Assume without loss of generality that \( a < b \). Then \( (w_\ell)_a = (w_{i+1})_a = (w_i)_a \oplus 1 \) for all \( \ell \geq i+1 \), while \( (w'_{\ell'})_a = (w'_{j+1})_a = (w'_j)_a = (w_i)_a \) for all \( \ell' \geq j+1 \). Thus, for any \( \ell \geq i+1, \ell' \geq j+1 \) we have \( w_\ell \neq w'_{\ell'} \), which means that the paths never intersect again. \( \square \)
Lemma 10.7. Fix \( \pi : \{0,1\}^n \rightarrow \{0,1\}^n \) and \( v \in \{0,1\}^n \). Let \( e_1, \ldots, e_k \) be the edges of \( P_{\text{fix}}(v, \pi(v)) \). Define
\[
S_v = \{ v' \in V : v' \neq v, P_{\text{fix}}(v', \pi(v')) \text{ contains some edge from } e_1, \ldots, e_k \}.
\]
Then the packet sent from \( v \) to \( \pi(v) \) will reach its destination after at most \( k + |S_v| \) steps.

Proof. Let \( S = S_v \) for simplicity of notation. The proof is by a charging argument. Let \( p_v \) be the packet sent from \( v \) to \( \pi(v) \). We assume that packets carry “tokens” on them. Initially, there are no tokens. Assume that at some time step, the packet \( p_v \) it is supposed to traverse an edge \( e_i \) for some \( i \in [k] \), but instead another packet \( p_{v'} \) is sent over \( e_i \) at this time step (necessarily \( v' \in S \)). In such a case, we generate a new token and place it on \( p_{v'} \). Next, if for some \( v' \in S \), a packet \( p_{v'} \) with at least one token on it is supposed to traverse an edge \( e_j \) for \( j \in [k] \), but instead another packet \( p_{v''} \) is sent over it at the same time step (again, necessarily \( v'' \in S \)), then we move one token from \( p_{v''} \) to \( p_{v'} \).

We will show that any packet \( p_{v'} \) for \( v' \in S \) can have at most one token on it at any given moment. This shows that at most \( |S| \) tokens are generated overall. Thus, \( p_v \) is delayed for at most \( |S| \) steps and hence reaches its destination after at most \( k + |S| \) steps.

To see that, observe that tokens always move forward along \( e_1, \ldots, e_k \). That is, if we follow a specific token, it starts at some edge \( e_i \), follows a path \( e_i, \ldots, e_j \) for some \( j \geq i \), and then traverses an edge outside \( e_1, \ldots, e_k \). At this point, by Claim 10.6, it can never intersect the path \( e_1, \ldots, e_k \) again. So, we can never have two tokens which traverse the same edge at the same time, and hence two tokens can never be on the same packet. \( \Box \)

Lemma 10.8. Let \( t : \{0,1\}^n \rightarrow \{0,1\}^n \) be uniformly chosen. Let \( P(v) = P_{\text{fix}}(v, t(v)) \). Then with probability at least \( 1 - 2^{-n} \) over the choice of \( t \), for any path \( P(v) \), there are at most \( 6n \) other paths \( P(w) \) which intersect some edge of \( P(v) \).

Proof. For \( v, w \in \{0,1\}^n \), let \( X_{v,w} \in \{0,1\} \) be the indicator random variable for the event that \( P(v) \) and \( P(w) \) intersect in an edge. Our goal is to upper bound \( X_v = \sum_{w \neq v} X_{v,w} \) for all \( v \in \{0,1\}^n \). Before analyzing it, we first analyze a simpler random variable.

Fix an edge \( e = (u, u + e_a) \). For \( v \in \{0,1\}^n \), let \( Y_{e,v} \in \{0,1\} \) be an indicator variable for the event that the edge \( e \) belongs to the path \( P(w) \). The number of paths which pass through \( e \) is then \( \sum_{w \in \{0,1\}^n} Y_{e,w} \). Now, the path between \( w \) and \( t(w) \) paths through \( e \) iff \( w_i = u_i \) for all \( i > a \), and \( t(w)_i = u_i \) for all \( i < a \). Let \( A_e = \{ w \in \{0,1\}^n : w_i = u_i \ \forall i > a \} \). Only \( w \) with \( w \in A_e \) has a nonzero probability for the path from \( w \) to \( t(w) \) to go through \( e \). Note that \( |A_e| = 2^a \). Moreover, for any \( w \in A_e \), the probability that \( P(w) \) indeed passes through \( e \) is given by
\[
\Pr[Y_{e,w} = 1] = \Pr[t(w)_i = u_i \ \forall i < a] = 2^{-(a-1)}.
\]

Hence, the expected number of paths \( P(w) \) which go through any edge \( e \) is at most 2, since
\[
\mathbb{E} \left[ \sum_{w \in \{0,1\}^n} Y_{e,w} \right] = \sum_{w \in A_e} \Pr[Y_{e,w} = 1] = 2^a \cdot 2^{-(a-1)} = 2.
\]

57
Next, the paths $P(v), P(w)$ intersect if some $e \in P(v)$ belongs to $P(w)$. This implies that

$$X_{v,w} \leq \sum_{e \in P(v)} Y_{e,w}.$$ 

Hence we can bound

$$E[X_v] = E \left[ \sum_{w \neq v} X_{v,w} \right] \leq E \left[ \sum_{e \in P(v)} \sum_{w \neq v} Y_{e,w} \right].$$

In order to bound $E[X_v]$, note that once we fix $t(v)$ then $P(v)$ becomes a fixed list of $n$ edges. Hence

$$E[X_v|t(v)] = \sum_{e \in P(v)} E \left[ \sum_{w \neq v} Y_{e,w} \right] \leq 2n.$$ 

This then implies the same bound once we average over $t(v)$ as well,

$$E[X_v] \leq 2n.$$ 

This means that for every $v$, the path $P(v)$ intersects on average most $2n$ other paths $P(w)$. Next, we show that this if we slightly increase the bound, this happens with very high probability. This requires a tail bound. In our case, a multiplicative version of the Chernoff inequality.

**Theorem 10.9** (Chernoff bound, multiplicative version). Let $Z_1, \ldots, Z_N \in \{0,1\}$ be independent random variables. Let $Z = Z_1 + \ldots + Z_N$ with $E[Z] = \mu$. Then for any $\lambda > 0$,

$$\Pr[Z \geq (1 + \lambda)\mu] \leq \exp \left( -\frac{\lambda^2 \mu}{2 + \lambda} \right).$$

In our case, let $v \in \{0,1\}^n$, fix $t(v) \in \{0,1\}^n$ and let $Z_1, \ldots, Z_N$ be the random variables $\{X_{v,w} : w \neq v\}$. Note that they are indeed independent, their sum is $Z = X_v$, and that $\mu = E[X_v] \leq 2n$. Taking $\lambda = 2$ gives

$$\Pr[X_v \geq 6n] \leq \exp(-2n).$$

By the union bound, the probability that $X_v \geq 6n$ for some $v \in \{0,1\}^n$ is bounded by

$$\Pr[\exists v \in \{0,1\}^n, X_v \geq 6n] \leq 2^n \exp(-2n) \leq 2^{-n}.$$ 

\[\square\]

**Proof of Theorem 10.5.** Let $t : \{0,1\}^n \to \{0,1\}^n$ be uniformly chosen. By Lemma 10.8, we have that $|S_v| \leq 6n$ for all $v \in \{0,1\}^n$ with probability at least $1 - 2^{-n}$. By Lemma 10.7, this implies that the packet send from $v$ to $t(v)$ would reach its destination in at most $n + 6n = 7n$ time steps. Analogously, we can send the packets from $t(v)$ to $\pi(v)$ in at most $7n$ time steps (note that this is exactly the same argument, except that the starting point is now randomized, instead of the end point). Again, the success probability of this phase is $1 - 2^{-n}$. By the union bound, the choice of $t$ is good for both phases with probability at least $1 - 2 \cdot 2^{-n}$. \[\square\]
11 Expander graphs

Expander graphs are deterministic graphs which behave like random graphs in many ways. They have a large number of applications, including in derandomization, constructions of error-correcting codes, robust network design, and many more. Here, we will only give some definitions and describe a few of their properties. For a much more comprehensive survey see [HLW06].

11.1 Edge expansion

Let $G = (V, E)$ be an undirected graph. We will focus here on $d$-regular graphs, but many of the results can be extended to non-regular graphs as well. Let $E(S, T) = |E \cap (S \times T)|$ denote the number of edges in $G$ with one endpoint in $S$ and the other in $T$. For a subset $S \subset V$, its edge boundary is $E(S, S^c)$. We say that $G$ is an edge expander, if any set $S$ has many edges going out of it.

Definition 11.1. The Cheeger constant of $G$ is

$$h(G) = \min_{S \subset V, 1 \leq |S| \leq |V|/2} \frac{E(S, S^c)}{|S|}.$$ 

Simple bounds are $0 \leq h(G) \leq d$, with $h(G) = 0$ iff $G$ is disconnected. A simple example for a graph which large edge expansion is the complete graph. If $G = K_n$, the complete graph on $n$ vertices, then $d = n - 1$ and

$$h(G) = \min_{S \subset V, 1 \leq |S| \leq |V|/2} \frac{E(S, S^c)}{|S|} = \min_{S \subset V, 1 \leq |S| \leq |V|/2} |S^c| = n/2.$$ 

Our interest however will be in constructing large but very sparse graphs (ideally with $d = 3$) for which $h(G) \geq c$ for some absolute constant $c > 0$. Such graphs are “highly connected” graphs. For example, the following lemma shows that by deleting a few edges in such graphs, we can only disconnect a few vertices. This is very useful for example in network design, where we want the failure of edges to effect as few nodes as possible.

Lemma 11.2. To disconnect $k$ vertices from the rest of the graph, we must delete at least $k \cdot h(G)$ edges.

Proof. If after deleting some number of edges, a set $S \subset V$ of size $|S| = k$ gets disconnected from the graph, then we must have deleted at least $E(S, S^c) \geq k \cdot h(G)$ many edges. 

There are several constructions of expander graphs which are based on number theory. The constructions are simple and beautiful, but the proofs are hard. For example, a construction of Selberg has $V = \mathbb{Z}_p \cup \{\infty\}$ and $E = \{(x, x+1), (x, x-1), (x, 1/x) : x \in V\}$. It is a 3-regular graph, and it can be proven to have $h(G) \geq 3/32$. The degree 3 is the smallest we can hope for.
Claim 11.3. If $G$ is 2-regular on $n$ vertices then $h(G) \leq 4/n$.

Proof. If $G$ is 2-regular, it is a union of cycles. If it is disconnected then $h(G) = 0$. Otherwise, it is a cycle $v_1, v_2, \ldots, v_n, v_1$. If we take $S = \{v_1, \ldots, v_{n/2}\}$ then $E(S, S^c) = 2$, $|S| = n/2$ and hence $h(G) \leq 4/n$. ☐

11.2 Spectral expansion

We will now describe another notion of expansion, called spectral expansion. It must seem less natural, but we will later see that it is essentially equivalent to edge expansion. However, the benefit will be that it is easy to check if a graph has a good spectral expansion, while we don’t know of an efficient way to test for edge expansion (other than computing it directly, which takes exponential time).

For a $d$-regular graph $G = (V, E)$, $|V| = n$, let $A$ be the adjacency matrix of $G$. That is, $A$ is an $n \times n$ matrix with $A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$.

Note that $A$ is a symmetric matrix, hence its eigenvalues are all real. We first note a few simple properties of them.

Claim 11.4.

(i) All eigenvalues of $A$ are in the range $[-d, d]$.

(ii) The vector $\vec{1}$ is an eigenvector of $A$ with eigenvalue $d$.

(iii) If $G$ has $k$ connected components then $A$ has $k$ linearly independent eigenvectors with eigenvalue $d$.

(iv) If $G$ is bi-partite then $A$ has eigenvalue of $-d$.

Parts (iii), (iv) are in fact iff, but we will only show one direction.

Proof. (i) Let $v \in \mathbb{R}^n$ be an eigenvector of $A$ with eigenvalue $\lambda$. Let $i$ be such that $|v_i|$ is maximal. Then

$$\lambda v_i = (Av)_i = \sum_{j \sim i} v_j,$$

and hence $|\lambda| |v_i| \leq \sum_{j \sim i} |v_j| \leq d |v_i|$, which gives $|\lambda| \leq d$.

(ii) For any $i \in [n], (A\vec{1})_i = \sum_{j \sim i} 1 = d$.

(iii) Let $\vec{1}_S \in \mathbb{R}^n$ be the indicator vector for a set $S \subset V$. If $G$ has $k$ connected components, say $S_1, \ldots, S_k \subset V$, then $\vec{1}_{S_1}, \ldots, \vec{1}_{S_k}$ are all eigenvectors of $G$ with eigenvalue $d$. 60
(iv) If \( G \) is bi-partite, say \( V = V_1 \cup V_2 \) with \( E \subseteq V_1 \times V_2 \), then the vector \( v = \bar{1}_{V_1} - \bar{1}_{V_2} \) has eigenvalue \(-d\). Indeed, if \( i \in V_1 \) then
\[
(Av)_i = \sum_{j \sim i} v_j = -d
\]
since if \( j \sim i \) then \( j \in V_2 \) and hence \( v_j = -1 \). Similarly if \( i \in V_2 \).

Let \( \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \) be the eigenvalues of \( G \). We know that \( \lambda_1 = d \) and that \( \lambda_n \geq -d \).

**Definition 11.5.** A \( d \)-regular graph \( G \) is a \( \lambda \)-expander for \( 0 \leq \lambda \leq d \) if \( |\lambda_2|, \ldots, |\lambda_n| \leq \lambda \).

It is very simple to check spectral expansion: we can simply compute the eigenvalues of the matrix \( A \). Surprisingly, having a nontrivial spectral expansion (namely \( \lambda \leq d - \varepsilon \)) is equivalent to having a nontrivial edge expansion (namely \( h \geq \varepsilon' \)). We next see that \( \lambda \)-expanders have a stronger property than just edge expansion: the number of edges between any two large sets, is close to the expected number in a random \( d \)-regular graph.

**Lemma 11.6** (Expander mixing lemma). Let \( G \) be a \( d \)-regular \( \lambda \)-expander. Let \( S, T \subset V \). Then
\[
\left| E(S, T) - \frac{d|S||T|}{n} \right| \leq \lambda \sqrt{|S||T|}.
\]

**Proof.** We can write \( E(S, T) = \bar{1}_S^T A \bar{1}_T \). We decompose
\[
\bar{1}_S = \sum \alpha_i v_i
\]
and
\[
\bar{1}_T = \sum \beta_i v_i.
\]
Then
\[
E(S, T) = \sum \alpha_i \beta_i \lambda_i.
\]
The terms for \( i = 1 \) correspond to the “random graph” case: \( \alpha_1 = \langle \bar{1}_S, v_1 \rangle = |S|/\sqrt{n}, \beta_1 = |T|/\sqrt{n} \) and \( \lambda_1 = d \), so
\[
\alpha_1 \beta_1 \lambda_1 = \frac{d|S||T|}{n}.
\]
We can thus bound
\[
\left| E(S, T) - \frac{d|S||T|}{n} \right| = \left| \sum_{i=2}^n \alpha_i \beta_i \lambda_i \right| \leq \lambda \sum_{i=2}^n |\alpha_i| |\beta_i| \leq \lambda \sqrt{\sum_{i=2}^n \alpha_i^2} \cdot \sqrt{\sum_{i=2}^n \beta_i^2},
\]
where we used the Cauchy-Schwarz inequality \( \langle u, v \rangle \leq ||u||_2 ||v||_2 \) for vectors \( u, v \). To conclude, note that \( |S| = \langle \bar{1}_S, \bar{1}_S \rangle = \sum \alpha_i^2 \) and similarly \( |T| = \sum \beta_i^2 \). Hence
\[
\left| E(S, T) - \frac{d|S||T|}{n} \right| \leq \lambda \sqrt{|S||T|}.
\]

\( \square \)
For example, if $|S| = \alpha n$, $|T| = \beta n$ then

$$E(S, T) = (\alpha \beta d \pm \sqrt{\alpha \beta \lambda})n.$$ 

So, as long as $\lambda/d \ll \sqrt{\alpha \beta}$ then we have a good estimate on the number of edges between $S$ and $T$.

**Corollary 11.7.** Let $G = (V, E)$ be a $d$-regular $\lambda$-expander. Let $I \subset V$ be an independent set. Then $|I| \leq \frac{\lambda}{d} |V|$. 

**Proof.** If $I \subset V$ is an independent set then $E(I, I) = 0$. Plugging this to the expander mixing lemma gives

$$\frac{d|I|^2}{|V|} \leq \lambda |I|$$

which gives $|I| \leq \frac{\lambda}{d} |V|$. \qed

So, we can prove strong properties of a graph whenever $\lambda \ll d$. But, how small can $\lambda$ be as a function of the degree $d$? Alon and Boppana proved that $\lambda \geq 2\sqrt{d-1}(1+o(1))$. This bound is tight, and graphs which attain it are called Ramanujan graphs. We will prove a slightly weaker bound.

**Lemma 11.8.** Let $G = (V, E)$ be a $d$-regular $\lambda$-expander. Assume that $|V| = n \gg d$. Then $\lambda \geq \sqrt{d}(1-o(1))$. 

**Proof.** For any real matrix $M$ it holds that $Tr(MM^t) = \sum |\lambda_i(M)|^2$. As $A$ is symmetric, we have

$$Tr(A^2) = \sum \lambda_i^2.$$ 

On the other hand,

$$Tr(A^2) = \sum_{i=1}^n (A^2)_{i,i} = \sum_{i,j=1}^n A^2_{i,j} = 2|E| = nd.$$ 

So $\sum \lambda_i^2 = nd$. We have $\lambda_1 = d$ and hence

$$\sum_{i=2}^n \lambda_i^2 = nd - d^2 = d(n - d).$$

As we have $|\lambda_i| \leq \lambda$ for all $i \leq 2$, we conclude that

$$\lambda^2 \geq \frac{d(n - d)}{n - 1} = d \left( 1 - \frac{d - 1}{n - 1} \right) = d(1 - o(1)).$$ \qed
11.3 Cheeger inequality

We will prove the following theorem, relating spectral expansion and edge expansion.

**Theorem 11.9** (Cheeger inequality). For any $d$-regular graph $G$,
\[
\frac{1}{2} (d - \lambda_2) \leq h(G) \leq \sqrt{2d(d - \lambda_2)}.
\]

Let $v_1, \ldots, v_n$ be the eigenvectors of $A$ corresponding to eigenvalues $\lambda_1, \ldots, \lambda_n$. Since the matrix $A$ is symmetric, we can choose orthonormal eigenvectors, $\langle v_i, v_j \rangle = 1_{i=j}$. In particular, $v_1 = \frac{1}{\sqrt{n}} \mathbf{1}$. We will only prove the lower bound on $h(G)$, which is easier and is sufficient for our goals - to show a nontrivial edge expansion, it suffices to show that $\lambda_2 \ll d$.

We start with a general characterization of $\lambda_2$.

**Claim 11.10.** \( \lambda_2 = \sup_{w \in \mathbb{R}^n, \langle w, \mathbf{1} \rangle = 1} \frac{w^T Aw}{w^T w} \).

Proof. Let $w \in \mathbb{R}^n$ be such that $\langle w, \mathbf{1} \rangle = 0$. We can decompose $w = \sum_{i=1}^n \alpha_i v_i$. Then $w^T w = \sum_i \alpha_i^2$ and $w^T A w = \sum_{i=2}^n \lambda_i \alpha_i^2$. Since $\lambda_1 \leq \lambda_2$ for all $i \geq 2$, we obtain that $w^T A w \leq \lambda_2 w^T w$.

Clearly, if $w = v_2$ then $w^T A w = \lambda_2 w^T w$, hence the claim follows. \( \square \)

So, to prove the lower bound $h(G) \geq (d - \lambda_2)/2$, which is equivalent to $\lambda_2 \geq d - 2h(G)$, we just need to exhibit a vector $w$. We do so in the next lemma.

**Lemma 11.11.** Let $S \subset V$ be a set for which $h(G) = \frac{E(S, S^c)}{|S|}$. Define $w \in \mathbb{R}^n$ by
\[
w_i = \mathbf{1}_S - \frac{|S|}{n} \mathbf{1}.
\]

Then $\langle w, \mathbf{1} \rangle = 0$ and
\[
\frac{w^T A w}{w^T w} \geq d - 2h(G).
\]

Proof. It is clear that $\langle w, \mathbf{1} \rangle = 0$. For the latter claim, we first compute
\[
w^T w = (\mathbf{1}_S - \frac{|S|}{n} \mathbf{1})^T (\mathbf{1}_S - \frac{|S|}{n} \mathbf{1}) = |S| - \frac{|S|^2}{n}.
\]

Next, $A w = A \mathbf{1}_S - \frac{|S|}{n} \mathbf{1}$ and since $\langle w, \mathbf{1} \rangle = 0$ we have
\[
w^T A w = w^T A \mathbf{1}_S = \mathbf{1}_S^T A \mathbf{1}_S - \frac{|S|^2}{n} = E(S, S) - \frac{|S|^2}{n} = d|S| - E(S, S^c) - \frac{|S|^2}{n}.
\]

We now plug in $E(S, S^c) = |S|h(G)$ and obtain that
\[
w^T A w = w^T w \left(d - \frac{|S|}{|S| - |S|^2/n} h(G)\right) = w^T w \left(d - \frac{n}{n - |S|} h(G)\right) \geq w^T w (d - 2h(G))
\]
since $|S| \leq n/2$ and hence $n/(n - |S|) \leq 2$. \( \square \)
11.4 Random walks mix fast

We saw one notion under which expanders are “robust” - deleting a few edges can only disconnect a few vertices. Now we will see another - random walks mix fast. This will require a bound on all $\lambda_i$ for $i \geq 2$, which is how we defined $\lambda$-expanders.

A random walk in a $d$-regular graph $G$ is defined as one expects: given a current node $i \in V$, a neighbour $j$ of $i$ is selected uniformly, and we move to $j$. There is a simple characterization of the probability distribution on the nodes after one step, given by the normalized adjacency matrix. Define

$$\tilde{A} = \frac{1}{d} A.$$ 

We can describe distributions over $V$ as vectors $\pi \in (\mathbb{R}^+)^n$, where $\pi_i$ is the probability that we are at node $i$.

**Claim 11.12.** Let $\pi \in (\mathbb{R}^+)^n$ be a distribution over the nodes. After taking one step in the random walk on $G$, the new distribution over nodes is given by $\tilde{A}\pi$.

**Proof.** Let $\pi'$ be the distribution over the nodes after the random walk. The probability that we are at node $i$ after the random walk is the sum over all its neighbours $j \sim i$, of the probability that we were at node $j$ before the step, and that we chose to go from $j$ to $i$. This latter probability is $1/d$ always, as the graph is $d$-regular. So

$$\pi'_i = \sum_{j \sim i} \pi_j \cdot (1/d) = (1/d)(A\pi)_i = (\tilde{A}\pi)_i.$$

We next use this observation to show that if $\lambda_2 \ll d$ then random walks in $G$ converge fast to the uniform distribution. The distance between distributions is the statistical distance, given by

$$\text{dist}(\pi, \pi') = \frac{1}{2} \sum_i |\pi_i - \pi'_i| = \frac{1}{2} \|\pi - \pi'\|_1.$$

It can be shown that this is also equivalent to the largest probability in which an event can distinguish $\pi$ from $\pi'$,

$$\text{dist}(\pi, \pi') = \max_{F \subseteq [n]} \sum_{i \in F} \pi_i - \pi'_i.$$

Below, we denote by $U = (1/n)\mathbf{1}$ the uniform distribution over the nodes.

**Lemma 11.13.** Let $\pi_0$ be any starting distribution over the nodes of $V$. Let $\pi_1, \pi_2, \ldots$ be the distributions obtained by performing a random walk on the nodes. Then

$$\|\pi_t - U\|_1 \leq n(\lambda/d)^t.$$
Proof. Decompose \( \pi_0 = \sum \alpha_i v_i \) where \( \alpha_1 = \langle \pi_0, v_1 \rangle = 1/\sqrt{n} \langle \pi_0, \mathbf{1} \rangle = 1/\sqrt{n} \) and hence \( \alpha_1 v_1 = (1/n) \mathbf{1} = U \) is the uniform distribution. We have that \( \pi_t = \mathbf{A}_t \pi_0 \). The eigenvectors of \( \mathbf{A} \) are \( v_1, \ldots, v_n \) with eigenvalues \( 1 = \lambda_1/d, \lambda_2/d, \ldots, \lambda_n/d \). Hence

\[
\pi_t = \sum_{i=1}^{n} (\lambda_i/d)^t v_i.
\]

Thus, the difference between \( \pi_t \) and the uniform distribution is given by

\[
\pi_t - U = \sum_{i=2}^{n} (\lambda_i/d)^t v_i.
\]

In order to bound \( \|\pi_t - U\|_1 \), it will be easier to first bound \( \|\pi_t - U\|_2 \), and then use the Cauchy-Schwarz inequality: for any \( w \in \mathbb{R}^n \) we have

\[
\|w\|_1^2 = \left( \sum_{i=1}^{n} |w_i| \right)^2 \leq n \sum_{i=1}^{n} |w_i|^2 = n \|w\|_2^2.
\]

Now, \( |\lambda_i| \leq \lambda \) for all \( i \geq 2 \). So,

\[
\|\pi - U\|_2^2 = \sum_{i=2}^{n} (\lambda_i/d)^{2t} \leq n (\lambda/d)^{2t}.
\]

Hence,

\[
\|\pi - U\|_1 \leq n (\lambda/d)^t.
\]

\[
\Box
\]

**Corollary 11.14.** The diameter of a \( d \)-regular \( \lambda \)-expander is at most \( 2 \log n \frac{\log d/\lambda}{\log(d/\lambda)} \).

Proof. Fix any \( i, j \in V \). The probability of a random walk of length \( t \) which starts at \( i \) to reach \( j \) is \( 1/n \pm n(\lambda/d)^t \). If \( t = \frac{c \log n}{\log(d/\lambda)} \) then the error term is bounded by

\[
n(\lambda/d)^t \leq n^{-(c-1)}.
\]

So, for \( c > 2 \) the error term is < \( 1/n \), and hence there is a positive probability to reach \( j \) from \( i \) within \( t \) steps. In particular, their distance is bounded by \( t \).

\[
\Box
\]

### 11.5 Random walks escape small sets

We next show another property of random walks on expanders: they don’t stay trapped in small sets.
Lemma 11.15. Let \( S \subset V \). Let \( i_0 \in V \) be uniformly chosen, and let \( i_1, i_2, \ldots, i_t \in V \) be nodes obtained by a random walk starting at \( i_0 \). Then

\[
\Pr[i_0, i_1, \ldots, i_t \in S] \leq \left( \frac{|S|}{n} + \left( 1 - \frac{S}{n} \right) \frac{\lambda}{d} \right)^t.
\]

Proof. We analyze the event that \( i_0, i_1, \ldots, i_t \in S \) by analyzing the conditional probabilities:

\[
\Pr[i_0, i_1, \ldots, i_t \in S] = \Pr[i_0 \in S] \cdot \Pr[i_1 \in S|i_0 \in S] \cdot \cdots \cdot \Pr[i_t \in S|i_0, i_1, \ldots, i_{t-1} \in S].
\]

Clearly, \( \Pr[i_0 \in S] = \frac{|S|}{n} \). However, we will present it in another way. Let \( \pi_0 = (1/n) \bar{1} \) be the uniform distribution, and let \( \Pi_S \) be the projection to \( S \). That is, \( \Pi_S \) is a diagonal \( n \times n \) matrix with \( (\Pi_S)_{i,i} = 1_{i \in S} \). Then

\[
\Pr[i_0 \in S] = \bar{1}^T \Pi_S \pi_0.
\]

Let \( \pi'_0 \) be the conditional distribution of \( i_0 \), conditioned on \( i_0 \in S \). It is the uniform distribution over \( S \). Equivalently,

\[
\pi'_0 = \frac{\Pi_S \pi_0}{\Pr[i_0 \in S]}.
\]

The distribution of \( i_1 \), conditioned on \( i_0 \in S \), is given by \( \pi_1 = \Pi \pi'_0 \). The probability that \( i_1 \in S \) conditioned on \( i_0 \in S \) is given by

\[
\Pr[i_1 \in S|i_0 \in S] = \Pi^T \Pi_S \pi_1,
\]

Let \( \pi'_1 \) be the distribution of \( i_1 \), conditioned on \( i_0, i_1 \in S \). Then

\[
\pi'_1 = \frac{\Pi_S \pi_1}{\Pr[i_1 \in S|i_0 \in S]} = \frac{\Pi_S \bar{A} \pi'_0}{\Pr[i_1 \in S|i_0 \in S]} = \frac{\Pi_S \bar{A} \Pi_S \pi_0}{\Pr[i_1 \in S|i_0 \in S]} = \frac{\Pi_S \bar{A} \Pi_S \pi_0}{\Pr[i_0, i_1 \in S]}
\]

Similarly,

\[
\Pr[i_2 \in S|i_0, i_1 \in S] = \Pi^T \Pi_S \pi'_1,
\]

and

\[
\pi'_2 = \frac{\Pi_S \bar{A} \pi'_1}{\Pr[i_2 \in S|i_0, i_1 \in S]} = \frac{\Pi_S \bar{A} \Pi_S \bar{A} \Pi_S \pi_0}{\Pr[i_0, i_1, i_2 \in S]}
\]

More generally, and exploiting the fact that \( \Pi_S^2 = \Pi_S \), we have that the conditioned distribution of \( i_t \), conditioned on \( i_0, \ldots, i_t \in S \), is given by

\[
\pi'_t = \frac{(\Pi_S \bar{A} \Pi_S)^t \pi_0}{\Pr[i_0, \ldots, i_t \in S]}
\]

Since \( \pi'_t \) is a distribution, \( \Pi^T \pi'_t = 1 \). Hence we can compute

\[
\Pr[i_0, i_1, \ldots, i_t \in S] = \Pi^T (\Pi_S \bar{A} \Pi_S)^t \pi_0.
\]
Let $M = \Pi_S\tilde{A}\Pi_S$. As it is a symmetric matrix, its eigenvalues are all real. Let $\mu \in \mathbb{R}$ denote its largest eigenvalue in absolute value. We will shortly bound $|\mu|$. This will then imply that $\|Mv\|_2 \leq |\mu|\|v\|_2$ for any vector $v \in \mathbb{R}^n$, and hence $\|M^tv\|_2 \leq |\mu|^t\|v\|_2$. Thus

$$\Pr[i_0, i_1, \ldots, i_t \in S] \leq \|\tilde{1}\|_2\|M^t\pi_0\|_2 \leq \sqrt{n} \cdot |\mu|^t\|\pi_0\|_2 = |\mu|^t.$$ 

In order to bound $|\mu|$, let $w \in \mathbb{R}^n$ denote the eigenvector corresponding to $\mu$, where we normalize $\|w\|_2 = 1$. Since $\mu w = Mw = \Pi_S(\tilde{A}\Pi_S w)$ we must have that $w$ is supported on $S$, and hence $\Pi_S w = w$. Decompose $w = \alpha v_1 + w^\perp$ where $\alpha = \langle w, v_1 \rangle$ and $\langle w^\perp, v_1 \rangle = 0$, and let $\beta = \|w^\perp\|_2$. We have

$$1 = \|w\|_2^2 = \alpha^2\|v_1\|_2^2 + \|w^\perp\|_2^2 = \alpha^2 + \beta^2$$

and

$$|\mu| = |w^TMw| = |w^T\tilde{A}w| = |\alpha^2 + (w^\perp)^T\tilde{A}w^\perp| \leq \alpha^2 + (\lambda/d)\beta^2 = \alpha^2 + (\lambda/d)(1 - \alpha^2).$$

So, to bound $|\mu|$ we need to bound $|\alpha|$. As $w$ is supported on $S$ we have

$$|\alpha| = |\langle w, v_1 \rangle| = \frac{1}{\sqrt{n}}|\langle w, \tilde{1} \rangle| = \frac{1}{\sqrt{n}}|\langle w, \tilde{1}_S \rangle| \leq \frac{1}{\sqrt{n}}\|w\|_2\|\tilde{1}_S\|_2 = \sqrt{|S|/n}$$

and hence

$$\alpha^2 \leq \frac{|S|}{n}.$$ 

We thus obtain the bound

$$|\mu| \leq \frac{|S|}{n} + \left(1 - \frac{|S|}{n}\right)\frac{\lambda}{d}.\quad\square$$

### 11.6 Randomness efficient error reduction in randomized algorithms

We describe an application of Lemma 11.15 in error reduction for randomized algorithms. Let $A(x,r)$ be a randomized algorithm which computes a boolean function $f(x) \in \{0,1\}$. Lets assume for simplicity it has a one-sided error (the analysis can be extended to two-sided error, but we won’t do that here). That is, we assume that

- If $f(x) = 0$ then $A(x,r) = 0$ always.
- If $f(x) = 1$ then $\Pr_r[A(x,r) = 1] \geq 1/2$.

Lets say that we want to increase the success probability for inputs $x$ for which $f(x) = 1$ from $1/2$ to $1 - \varepsilon$, where for simplicity we take $\varepsilon = 2^{-t}$. A simple solution is to simply repeat the algorithm $t$ times, and output 0 only if in all the runs it outputted 0. If $A$ uses $m$ random bits (eg $r \in \{0,1\}^m$) then the new algorithm will use $mt$ random bits. However, this can be improved using expanders.
Lemma 11.16. Let $G = (V, E)$ be some $d$-regular $\lambda$-expander for $V = \{0, 1\}^m$, where $\lambda < d = O(1)$ are constants. We treat nodes of $G$ as assignments to the random bits of $A$. Consider the following algorithm: choose a random $r_0 \in V$, and let $r_1, \ldots, r_t$ be obtained by a random walk on $G$ starting at $r_0$. On input $x$, we run $A(x, r_0), \ldots, A(x, r_t)$, and output 0 only if in all the runs output 0. Then

1. The new algorithm is a one-sided algorithm with error $2^{-\Omega(t)}$.

2. The new algorithm uses only $m + O(t)$ random bits.

Proof. If $f(x) = 0$ then $A(x, r) = 0$ for all $r$, hence we will return 0 always. So assume that $f(x) = 1$. Let $B = \{r \in \{0, 1\}^m : A(x, r) = 0\}$ be the “bad” random strings, on which the algorithm makes a mistake. By assumption, $|B| \leq |V|/2$. We will return 0 only if $r_0, \ldots, r_t \in B$. However, we know that

$$\Pr[r_0, \ldots, r_t \in B] \leq \left(\frac{1}{2} + \frac{1}{2} \cdot \frac{\lambda}{d}\right)^t = p^t$$

where $p = p(\lambda, d) < 1$ is a constant. So the probability of error is $2^{-\Omega(t)}$. The number of random bits required is as follows: $m$ random bits to choose $r_0$, but only $\log d = O(1)$ random bits to choose each $r_i$ given $r_{i-1}$. So the total number of random bits is $m + O(t)$.  \qed
References


