Discrete Mathematics

My notes from CS 70

Matthew Signorotti

Summer 2019

Contents

1 Propositional Logic **2**
1.1 De Morgan's laws 2

2 Proofs **3**
2.1 Common Proof Strategies 3

3 Graphs **4**
3.1 Types of Graphs 4

4 Modular Arithmetic **5**
4.1 Finite Fields with Modular Arithmetic 5
4.2 The Fundamental Theorem of Arithmetic 5
4.3 Computing Greatest Common Denominators and Inverses: the Extended GCD Algorithm 6
4.4 The Chinese Remainder Theorem 7
4.5 Fermat's Little Theorem 7

5 Polynomials **7**
5.1 Key Polynomial Theorems 7
5.2 Lagrange Interpolation 8

6 Applications of Modular Arithmetic and Polynomials **8**
6.1 Encryption Schemes 8
6.2 Shamir's Secret Sharing 9
6.3 Error-Correcting Codes 9
6.3.1 Dropped Packets 9
6.3.2 Corrupted Packets 10
6.3.3 The Berlekamp-Welch Algorithm and Its Proof 10

7 Countability **12**
7.1 The Pigeonhole Principle 12
7.2 The Cantor-Schröder-Bernstein Theorem 12

8 Computability **12**

9 The Stable Marriage Problem **13**
9.1 The Gale-Shapley Algorithm 13

10 Probability **13**

1 Propositional Logic

A **proposition** is a rigorously posed statement which is presumably either true or false. **Propositional formulae** are functions of the "boolean" values, true and false. Propositional logic provides compact mathematical symbols for common English phrases:

- $\wedge$: and
- $\vee$: or
- $\neg$: not/the negation of
- $\forall$: for all
- $\exists$: there exists/for some

1.1 De Morgan's laws

The famous **De Morgan's laws** observe that if P and Q are boolean values, then $\neg(P \wedge Q) = (\neg P) \vee (\neg Q)$, and also $\neg(P \vee Q) = (\neg P) \wedge (\neg Q)$.

2 Proofs

Mathematical **proofs** build upon elementary logical deductions to reach a desired statement. They are the building blocks of **theorems**, the main results of mathematics. Proofs use assumptions known as **axioms** and often presume certain **definitions** for ideas.

2.1 Common Proof Strategies

- **Direct proof**: Start from a premise P, and follow a sequence of logic to directly derive a conclusion Q.
- **Proof by contraposition**: Instead of directly proving that P leads to Q, show that $\neg Q$ leads to $\neg P$. This approach assumes that P and Q are each either true or false.[1]
- **Proof by contradiction**: Start by assuming the negation ($\neg Q$) of the desired conclusion Q, and show that it implies a **contradiction**, a seeming inconsistency with a proposition already thought to be true.
- **Proof by induction**: First prove that a statement holds for $n = 0$; then show that if it holds for some $n \in \mathbb{N} = \{0, 1, \ldots\}$, it must also hold for $n + 1$.
  - For this latter, "inductive" step, you may also assume the proposition holds not just for n, but also for all $i \in \mathbb{N}, \leq n$. This **strong induction** technique is analogous to knocking down an infinite line of dominoes: if the nth domino has fallen, then certainly so have the 1st through $(n-1)$st.
- **Constructive proof**: Provide an algorithm which uses widely accepted steps to reach some conclusion.
- **Proof by cases**: Consider all the ways in which something could be, and show that for all of these possibilities, the same conclusion follows.

[1] This assumption that all propositions are either true or false is the "law of the excluded middle." Some mathematicians and logicians consider this law a flawed assumption. For example, consider the statement "This proposition is false." The law of the excluded middle implies that this proposition is either true or false, but intuition tells us that neither case can be.

3 Graphs

Graphs are abstract mathematical constructs defined by their vertices and edges. **Vertices** are generic objects which are identified in some way, perhaps, for example, through a numeric ID. **Edges** are arrows between vertices and may be directed or undirected. A vertex's **degree** is the number of edges coming out of it.

Graphs have many interesting properties. According to the **handshaking lemma**, the sum of degrees of all the vertices in an undirected graph is twice the number of edges. A **Eulerian tour** is a closed "walk" around a graph (ending at the start vertex) which uses every edge exactly once; such a tour exists if and only if every vertex has even degree.

3.1 Types of Graphs

Numerous special types of graphs exist. Some, like planar graphs, have very well-studied properties.

- **Trees** are graphs which are connected and acyclic. Trees are equivalently defined as connected graphs with one fewer edge than vertices.
- **Complete graphs** have an edge between every pair of vertices.
- **Bipartite graphs** can be partitioned into two sets of vertices such that no edge connects vertices in the same set.
  - A graph is bipartite if and only if it can be 2-colored. (A graph is **n-colored** if each vertex is assigned one of n colors and no two adjacent vertices share a color.)
- **Planar graphs** can be drawn on paper so that no edges intersect. Graphs are thought to be always either planar or non-planar.
  - **Euler's formula for planar graphs**: vertices+faces = edges+2 (proof by induction)
  - Planar graphs are sparse, as for vertices ≥ 3, edges $\leq 3 \cdot$vertices-6 (proof applying Euler's formula). For bipartite planar graphs, we have a tighter bound of edges $\leq 2 \cdot$ vertices $- 4$, again for vertices ≥ 3.

  - Planar graphs' vertices can always be 6-colored (easy proof), 5-colored (hard proof), and even 4-colored (an extremely difficult computer-assisted proof).
  - **Kuratowski's theorem**: A graph is planar if and only if it contains the complete graph on 5 vertices (K_5) or the bipartite-complete graph on 6 vertices ($K_{3,3}$).

- An **n-dimensional hypercube** is made by copying two $(n-1)$-dimensional hypercubes and connecting corresponding vertices in the two copies.

4 Modular Arithmetic

The core idea of **modular arithmetic** is to perform addition, subtraction, and multiplication of integers as normally, but take the result of each operation modulo (remainder) some positive integer p.

4.1 Finite Fields with Modular Arithmetic

Modular arithmetic provides a finite alternative to common infinite fields such as the real numbers, which are difficult to represent on computers. One field is the set of integers $\{0, 1, \ldots, p-1\}$, taken modulo some prime p. This set happens to satisfy the field properties and provides an arithmetic system following similar rules as the real numbers. Since this set is finite, it is called a **finite field**, or a **Galois field**. Although it would be tedious to prove every field property for this finite field, we will see soon that for a prime modulus, this set satisfies the field property that all nonzero elements have a multiplicative inverse.

4.2 The Fundamental Theorem of Arithmetic

Every positive integer can be uniquely factorized into a multiplication of prime numbers (proof by induction).

4.3 Computing Greatest Common Denominators and Inverses: the Extended GCD Algorithm

A multiplicative inverse for $x \pmod m$ exists if and only if $gcd(x, m) = 1$. (The *gcd* of two integers is their **greatest common divisor**, the largest positive integer which is a factor of, or evenly divides, both integers.) Hence, all nonzero elements of $\{0, 1, \ldots, m-1\}$ will have inverses if m is prime. But otherwise, some element of $\{0, 1, \ldots, m-1\}$ will share a common divisor with m, and will lack a multiplicative inverse mod m. Hence, modular arithmetic fails to be a field when m is not prime.

Following is the **extended GCD algorithm**, `ExtendedGCD`, which takes inputs $x, m \in \mathbb{N}$ and returns $(a, b, gcd(x, m))$ such that $ax + bm = gcd(x, m)$. If an inverse exists, $gcd(x, m) = 1$ and thus $a = x^{-1} \pmod m$.

```
def ExtendedGCD(x, m):
    if m == 0:
        return (1, 0, x)
    else:
        (a', b', c) = ExtendedGCD(m, x mod m)
        return (b', a' - b'*floor(x/m), c)
```

1. If $m = 0$, return $(1, 0, x)$ (this is the base case).
2. Otherwise, recursively call `ExtendedGCD`$(m, x \bmod m)$.
   (a) This call will output (a', b', c) such that $a'm + b'(x \bmod m) = c = gcd(m, x \bmod m)$. Since $x \bmod m = x - \lfloor x/m \rfloor m$ ($\lfloor \cdot \rfloor$ indicates the floor operation), then $c = a'm + b'x - b'\lfloor x/m \rfloor m = b'x + (a' - b'\lfloor x/m \rfloor)m$.
   (b) Note that $gcd(x, m) = gcd(m, x \bmod m)$ (the proof is left to the reader), so $gcd(x, m) = b'x + (a' - b'\lfloor x/m \rfloor)m$. That is the justification for the below return statement.
   (c) Since $x \bmod m < m$, each argument strictly decreases every other iteration, so `ExtendedGCD` always successfully reaches the base case.
3. Return $(b', a' - b'\lfloor x/m \rfloor, c)$.

4.4 The Chinese Remainder Theorem

If $n_1, n_2, \ldots, n_k$ are mutually **coprime**, or have no common denominators greater than 1, then the system of equations $x \equiv a_i \pmod{n_i} \forall i$ has a unique solution modulo $N = \prod_i n_i$. The proof for the existence of a solution is constructive; the proof of uniqueness uses the existence conclusion and a counting argument.

4.5 Fermat's Little Theorem

If p is a prime number (has no factors besides 1 and p), then for all $a \in \{0, \ldots, p-1\}$, $a^{p-1} \equiv 1 \pmod{p}$.

A more general form is **Euler's theorem**, which states that if $\phi(n)$ is the number of numbers in $1, \ldots, n$ which are coprime to n, then $a^{\phi(n)} \equiv 1 \pmod{n}$. If n happens to be a prime, certainly $\phi(n) = n - 1$. Another interesting result is that if $gcd(m, n) = 1$, $\phi(mn) = \phi(m)\phi(n)$.

5 Polynomials

A degree-n **polynomial** (in some field) is a function of the form $f(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_0$. Two polynomials are said to be equal if their outputs are equal over the entire field.

5.1 Key Polynomial Theorems

1. Like with real numbers, long division works on polynomials over a field. Given a degree-m dividend polynomial p and nonzero degree-n divisor d, long division will return the *unique* degree-$(m-n)$ quotient q and degree-$< n$ remainder r such that $p(x) = q(x)d(x) + r(x)$ (constructive proof).

    (a) Note that r may be nonzero. Not all p are perfectly divided by some d, so polynomials over a field do not form a field themselves.

2. If p has degree d and $p(a) = 0$, then p can be written as $p(x) = (x-a)q(x)$, with q of degree $d-1$ (proof applying theorem 1).

    (a) This is a special exception to our observation that not all divisors perfectly divide a polynomial.

3. A nonzero, degree-d polynomial has $\leq d$ roots (proof by contradiction and induction on d using theorem 2).

4. Distinct degree-$\leq d$ polynomials agree on at most d points (direct proof applying theorem 3).

5. $d+1$ distinct points uniquely define some degree-$\leq d$ polynomial (consequence of theorem 4 and the following constructive Lagrange interpolation algorithm).

5.2 Lagrange Interpolation

From any distinct $n+1$ points, one can reconstruct a polynomial's coefficient representation by the **Lagrange interpolation** process:

1. For $i = 1, \ldots, n, n+1$, construct the polynomials $\Delta_i(x) = \prod_{i \neq j}(x - x_j) / \prod_{i \neq j}(x_i - x_j)$. Δ_i will be 1 at x_i and 0 at x_j for $j \neq i$.

2. Construct the polynomial as $f(x) = \sum_i y_i \Delta_i(x)$.

   (a) This output is itself a degree-n polynomial fitting $n+1$ points, and hence is the unique desired polynomial, by the last of the theorems in the next section.

An alternative procedure to Lagrange interpolation is posing the problem as a system of linear equations.

6 Applications of Modular Arithmetic and Polynomials

6.1 Encryption Schemes

Encryption schemes allow different parties to communicate without third-party eavesdropping. The **one-time pad scheme** assumes that the authentic parties know a common bitstring "pad" which they can XOR to encrypt and decrypt a message. This scheme has a couple issues. It is logistically difficult to agree on one-time pads. Also, reusing one-time pads poses a security vulnerability: XORing two encrypted messages yields the XOR of the true messages, and this can sometimes reveal statistical trends in a message.

In the **RSA public-key scheme**, each person has two random large prime numbers, p and q, and e such that $gcd\,(e, (p-1)(q-1)) = 1$. The public "key" is $(N = pq, e)$, and the private key is $(p, q, d = e^{-1} \pmod{(p-1)(q-1)})$. Guessing large primes p and q is efficient due to quick primality-testing algorithms and because the number of primes $\leq n$ is at least $n/\ln(n)$, a significant proportion of huge integers. The sender encrypts a message m as $c = m^e \pmod{N}$, and the receiver decrypts the message as $m = c^d \pmod{N}$. Correctness is proven using Fermat's little theorem; security is assumed because factoring N is very inefficient. However, the size of the primes required to prevent easy factoring makes RSA slower than current state-of-the-art methods.

6.2 Shamir's Secret Sharing

Say we need n people to recover some desired secret s, a number in some field. Define a polynomial p such that $p(0) = s$, and choose $p(i)$ uniformly at random for $i = 1, \ldots, n-1$. Then distribute $p(1), \ldots, p(n)$ to the n people. Only together can they uniquely recover the degree-$(n-1)$ polynomial; any fewer than n points of p and the original secret could just as well be anything.

6.3 Error-Correcting Codes

Suppose you would like to send a length-n message $m_1, \ldots, m_n$. Directly sending these packets $\{m_i\}$ fails to deliver the message with certainty when the channel is noisy or may drop packets. With the alternative approach of **Reed-Solomon encoding**, you uniquely fit a degree-$\leq n-1$ polynomial p to $(1, m_1), \ldots, (n, m_n)$, and the recipient will attempt to reconstruct p and thus the message from received points. Typically, you might work modulo some sufficiently large number s, but the core idea extends to the real numbers.

6.3.1 Dropped Packets

Firstly, consider that k packets may be dropped. In this case, simply send points of the polynomial p evaluated at $1, 2, \ldots, n+k$, and the recipient will receive at least the n packets needed to reconstruct p. Note that n *distinct* points uniquely identify the desired degree-$\leq n-1$ polynomial p, so if we are working in a modular arithmetic finite field, the modulus s must satisfy $s \geq n + k$.

6.3.2 Corrupted Packets

If up to k packets may be corrupted, send $n + 2k$ packets. A little deeper thinking shows that after corruption, a message sent with $< n + 2k$ packets could correspond to multiple messages. Yet, I argue $n + 2k$ packets are sufficient to reconstruct the polynomial with certainty: You could look at any subset of $n+k$ received packets, and $\geq n$ will be consistent with the true polynomial. Of these $n + k$ points, either some point is corrupted, hence no degree-$\leq n - 1$ polynomial fits these $n + k$ points (polynomial theorem 4), or all $n + k$ are uncorrupted and just the true polynomial fits (polynomial theorem 5). There must be some uncorrupted subset of $n + k$ points, under the assumption that at most k points can be corrupted.

6.3.3 The Berlekamp-Welch Algorithm and Its Proof

However, trying to fit a polynomial to a bunch of random sets of $n + k$ points is very slow. An empirically faster method is the **Berlekamp-Welch algorithm**. The following description is still inferior to real-world implementations which use the fast Fourier transform, but the core ideas involved are the same.

1. Find a nonzero solution to $n + 2k$ linear equations of the form $q'(i) = r_i e'(i)$, with $i = 1, \ldots, n + 2k$. Here, r_i is the ith received packet, which may or may not be corrupted. q' is defined so that $q'(i) = c_{n+k-1} i^{n+k-1} + \cdots + c_0$. e' is a **monic polynomial** in that $e'(i) = d_k i^k + \cdots + d_0$, with $d_k = 1$ (the monic property).

    (a) These equations will always have a solution for $\leq k$ corruptions. If only we used our previous slow algorithm to figure out which indices $\{e_i\}$ are corrupted, we could construct e as an "error-locator" polynomial of the form $e(i) = (i - e_1)(i - e_2) \cdots (i - e_k) = i^k + d_{k-1} i^{k-1} + \cdots + d_0$. (We assume at most k errors. And if there were fewer than k errors, we could set the extra e_i such that $e_i > n + 2k$.) Then, defining q so that $q(i) = p(i)e(i)$ satisfies $q(i) = r_i e(i)$ for $i = 1, \ldots, n + 2k$: for the $\leq k$ corrupted points i, $e(i) = 0 = q(i)$, and for the uncorrupted i, $p(i) = r_i$ so certainly $q(i) = p(i)e(i) = r_i e(i)$. These q and e must solve the system of equations, provided there are at most k corruptions.

2. Once solution polynomials q' and e' are determined, use polynomial long division to attempt to divide q' by e'. If the result has no remainder, and there were indeed at most k errors, then the algorithm has certainly yielded $p = q'/e'$, by the following proof. Otherwise, the algorithm fails.

   (a) If there are no remainder and k errors at most, then the original p has been recovered. (A maximum of k errors is a handy assumption to allow for this proof.) Once again, let e be a true, valid degree-k error-locator polynomial, with a root at i if and only if i is an error index (for $i = 1, \ldots, n + 2k$). Let q be the corresponding polynomial defined so that $q(i) = p(i)e(i)$. p is exactly the polynomial the Berlekamp-Welch algorithm attempts to recover. As described under the previous step, it turns out that $q(i) = r_i e(i)$. Likewise, for the solution q' and e' we have obtained, $q'(i) = r_i e'(i)$ — these were the very equations being solved — and some multiplication of equalities gives that

$$q(i)e'(i) = r_i e(i)e'(i) = q'(i)e(i)$$

   for all $i = 1, \ldots, n + 2k$.

   Now, let's say we perform long division on q' and e' and obtain $p' = q'/e'$, plus no remainder. How does the true polynomial p compare to the obtained polynomial p'? By definition $q(i) = p(i)e(i)$; furthermore, as previously noted $q(i) = r_i e(i)$. By the same logic, $q'(i) = p'(i)e'(i) = r_i e'(i)$. Multiplication yields that $q(i)e'(i) = p(i)e(i)e'(i)$ and $q'(i)e(i) = p'(i)e(i)e'(i)$. For all i then, since $q(i)e'(i) = q'(i)e(i)$,

$$p(i)e(i)e'(i) = p'(i)e(i)e'(i).$$

   Both e and e' are of degree k since there are $\leq k$ corruptions, so $e(i)e'(i) \neq 0$ for at least $n + 2k - 2(k) = n$ of the indices $i = 1, \ldots, n + 2k$. At those indices, $e(i)e'(i)$ has a multiplicative inverse and $p(i) = p'(i)$. p and p', which are both of degree $< n$, agree on at least n points. Therefore, p and p' must be the same polynomial (polynomial theorem 4 or 5). We have concluded the correctness of the Berlekamp-Welch algorithm: if long division yielded the polynomial p', and there were at most k errors, then p' is the desired polynomial p.

7 Countability

A set is **countable** if and only if a **bijection** exists between the set and the natural numbers, $\mathbb{N}$. A bijection is a one-to-one correspondence between pairs in two sets, or more formally, it can be considered a function which is both **injective** (no two inputs map to the same output) and **surjective** (it reaches every element in the output set). We assume that for any set, a bijection either exists or does not exist. We are again leveraging the law of excluded middle, which is addressed in a prior footnote.

7.1 The Pigeonhole Principle

The **pigeonhole principle** is the counting argument that if n pigeons are to be placed into $m < n$ holes, at least one hole will have multiple pigeons. Of course, the pigeonhole principle can apply to much more general mathematical objects than pigeons.

7.2 The Cantor-Schröder-Bernstein Theorem

If there exist injections from $A \to B$ and $B \to A$, then there exists a bijection between A and B.

8 Computability

Computers cannot compute everything! **Computability** is whether some program will always correctly solve a problem in finite time. The **halting problem** is uncomputable by construction of the following program:

```
def Turing(P, x):
    if TestHalt(P, x):
        loop forever
    else:
        halt
```

We assume that the program Turing can be passed as its own two arguments `P` and `x`. Then running `Turing(Turing, Turing)` poses a paradoxical contradiction.

Recursively enumerable problems have programs which terminate correctly if the answer is true, and terminate correctly or run forever if the answer is false. **Co-recursively enumerable** problems are the opposite: their programs may terminate and correctly output true or false, or they might run forever if the answer is true.

9 The Stable Marriage Problem

Say we have two distinct sets of n individuals whom we would like to pair (for instance, students and companies), and each of whom has a ranked preference list of individuals from the opposite set. A **stable matching** is a matching which has no **rogue pair**, which is two individuals, one from each set, who are unpaired yet would prefer each other over their current partners. A stable matching always exists, by a constructive proof (given with the following algorithm). The related problem of choosing roommates, in which all individuals are in the same group ("residents" of an apartment), may not have a stable matching by simple counterexample.

9.1 The Gale-Shapley Algorithm

More rigorously, let us denote the two sets as A and B. The Gale-Shapley algorithm constructs a stable matching by, essentially, progressively ruling out many of the n^2 possible pairings.

1. Each iteration, each individual $a \in A$ proposes to their top choice $b \in B$ who has not already rejected a.
    (a) Each individual $b \in B$ rejects all but their top suitor.
2. The process repeats until there are no rejections.

The Gale-Shapley algorithm has worst-case n^2 runtime. It can also be shown that the algorithm is A-optimal: in the stable matching output, every $a \in A$ will be matched with its most preferred partner out of all the partners a is matched with in any stable matching.

10 Probability

My probability notes are consolidated in a different document.