Proof and Algorithms

Xiaoyi Cui

This is just a brief summary of the contents of the lecture. Please note: most of the calculations and demonstrations shown in class are neglected. I claim no originality of this notes.

Axiomatic mathematics — the components of a theory

Mathematics are commonly put into axiomatic systems, which is any set of axioms from which some or all axioms can be used in conjunction to logically derive theorems.

Undefined terms: objects or manipulations that are explicit or intuitive in its own.

Axioms: properties (which are assumed to be true) that undefined terms need to satisfy are called axioms/postulates (from the Greek word meaning "worthy").

Definition: the means for binding a concept and a set of associated properties that describe the concept.

Any statement that is not an axiom or definition needs to be proved.

Theorems: Important statements that have been proved are called theorems.

Propositions: Less important (true) statements.

Lemma: subtheorems that will help prove part of a more important theorem or proposition.

Corollary: a statement whose truth is an immediate consequence of some other theorem or proposition.

Example 1 (Euclid's Postulates) *Undefined terms: straight line, point, circle, angle, etc.*

Axioms: "Let the following be postulated":

- *"To draw a straight line from any point to any point."*
- *"To produce [extend] a finite straight line continuously in a straight line."*
- *"To describe a circle with any centre and distance (radius)."*
- *"That all right angles are equal to one another."*
- *"That, if a straight line falling on two straight lines make the interior angles on the same side less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which are the angles less than the two right angles."*

Example 2 (Boolean Algebra)

Further properties of axioms: they need to be independent and consistent. I.e., let I labels the axioms of a system $\{A_i\}_{i\in I}$, then we have that

$$(\forall i)((\forall j \in I\backslash\{i\})A_j \to A_i) \iff \mathbb{F},$$

and

$$(\forall i)(\forall j \in I\backslash\{i\})(\wedge_j A_j \to \neg A_i) \iff \mathbb{F}.$$

Proof techniques

Assume that we would like to show that $P \to Q$ is true.

Trivial proof

From truth table of $P \to Q$ and that P is false, deduce that the statement is true.

Direct proof

$$P \to Q \Leftarrow (P \to S_1) \wedge (S_1 \to S_2) \wedge \cdots \wedge (S_n \to Q)$$

Proof by contraposition

Consider $(\neg Q \to \neg P) \Leftrightarrow (P \to Q)$.

Proof by contradiction

Assume that $P \to Q$ is false (i.e., $P \wedge \neg Q$ is true), conclude that it is logically equivalent to a contradiction, or $\neg P$, or Q. Consider $\neg(P \to Q) \Leftrightarrow \mathbb{F}$, or $\neg(P \to Q) \Leftrightarrow \neg P$, or $\neg(P \to Q) \Leftrightarrow Q$.

Proof by case

If $P \Rightarrow P_1 \vee P_2 \vee P_3 \vee \cdots$, then $\wedge_i(P_i \to Q) \Rightarrow P \to Q$.

Example 3 *If $n \in \mathbb{Z}$, then $n^3 - n$ is even.*

Proof by construction

Use the implications with existential quantifiers. $(P(a) \to Q) \Rightarrow (\exists x)(P(x) \to Q)$.

Proof of biconditionals

Example 4 *The following statements are equivalent:*

- A_1

- $\cdots$
- A_n.

There are essentially two approaches using graphic representation.

Further examples

Infinite numbers of primes

Theorem 1 *There is an infinite number of distinct primes.*

Proof. Suppose instead that there is only a finite number of primes. Denote them as $\{P_1, \cdots, p_n\}$ where n is a positive integer.

Consider the positive integer $q = (p_1 p_2 \cdots P_n) + 1$. Since $q \notin \{p_1, P_2, \cdots P_n\}$, q must be composite.

But none of the primes P_k divide q and $\{P_1, P_2, \cdots p_n\}$ are all the primes, q cannot be composite. This leads to a contradiction. Therefore, there must be an infinite number of primes. Q.E.D.

$P =$"S is the set of all primes", $Q =$"$|S|$ is finite". Now assume that $P \wedge \neg Q$ is true.
$W =$"$(p_1 p_2 \cdots P_n) + 1$ is composite", and we have that $(P \wedge \neg Q) \Rightarrow W$.

Now we have that $(P \wedge \neg Q) \Rightarrow \neg W$.

$(P \wedge \neg Q) \Rightarrow (W \wedge \neg W)$, hence $\mathbb{T} \Rightarrow \neg(P \wedge \neg Q)$, i.e., $P \Rightarrow Q$.

Prime Divisibility Property

Theorem 2 *Let p be a prime. If p divides the product $a_1 a_2 \cdots a_n$, then p divides at least one of the factors a_i.*

We shall look into different cases. First, if $n = 2$, and then, if $n > 2$.

Exercise 1 *Use the above result to prove the following proposition: every integer $n > 2$ can be uniquely written as a product of primes in ascending order.*

Mathematical Induction

Theorem 3 *If $\{P(i)\}$ is a set of statements such that*

- *$P(1)$ is true,*
- *$P(i) \to P(i+1)$ is true for $i > 1$,*

then P_k is true for all positive integers k. This can be stated more succinctly as $[P(1) \wedge (\forall i)(P(i) \to P(i+1))] \Rightarrow [(\forall k)P(k)]$.

Remark 1 *In the above theorem, the second condition can be replaced by "$\wedge_{j \leq i} P(j) \to P(i+1)$ is true for $i > 1$". (Why?) This is called the theorem of complete induction.*

Exercise 2 *Prove that "every integer $n > 2$ can be written as a product of primes".*

Exercise 3 *Let $n > 1$. Suppose we have a $2^n \times 2^n$ chess board, with one square missing, and a box full of L-shaped tiles. Each tile can cover 3 squares on the chess board. No matter which square on the chess board is missing, we can entirely cover the remaining squares with the tiles.*

The well-ordering principal is an axiom which says "every nonempty set of natural numbers has a smallest element".

Theorem 4 *The following are equivalent[1]: 1) the well-ordering principle (WOP) 2) the theorem of mathematical induction (MI) and 3) the theorem of complete induction (CI).*

[1] If you are interested in the proof, or more aspects, try looking for "Piano axioms" in Wikipedia.

Computability, provability and Gödel's theorem

Note: this is a beautiful theory that I can not resist to talk about. But it is hard presenting the full story, as do assigning homeworks ;)

Definition 1 *An algorithm is a finite sequence of unambiguous steps for solving a problem or completing a task in a finite amount of time.*

Algorithms are like functions — they take input, and produce output. It is important to know about the domain. On the other hand, we are interested in “multi-variable" things.

If the size of a problem increases, will the algorithms become complicated? And how to decide the relation between the size and the computability?

Question 1: is the problem solvable?

Question 2: for how long can one solve a problem?

Theorem 5 (Cantor's Uncountability Theorem) *There are uncountably many infinite sequences of 0's and 1's.*

Definition 2 *A 01-sequence $f(i)$ is computable if there is a program which given input i computes $f(i)$. A 01-sequence program is a string of symbols which associates output “0” or “1” to each input in $\mathbb{N}$ within a finite number of steps (“halts”).*

Proposition 1 *The set of computable 01-sequences cannot be listed in a computable way. Similarly the set of 01-sequence program can not be listed in a computable way[2].*

[2] The second fact is easy — otherwise it would contradict the first one. For the first fact, use the diagonalization technique

The notion of 01-sequence and 01-sequence programs can be generalized — firstly, the range of the sequence can be taken on $\mathbb{N}$ as opposed to $\{0, 1\}$, and secondly, the sequences can become arrays.

Generalize the notion of computable sequence, we have the following notions.

Definition 3 • *Let X and Y be two sets, a partial function from X to Y is any pair $\langle D(f), f\rangle$, where $D(f) \subset X$.*

- *A partial function f from $\mathbb{N}^m$ to $\mathbb{N}^n$ is called computable if there exists a program that, wherever a vector x is entered in the input, gives as an output $f(x)$ if x is in the domain, and 0 otherwise.*
- *A partial function f from $\mathbb{N}^m$ to $\mathbb{N}^n$ is called semi-computable if there exists a program that, wherever a vector x is entered in the input, gives as an output $f(x)$ if x is in the domain, and 0 or works infinitely long without stopping otherwise.*
- *A partial function f from $\mathbb{N}^m$ to $\mathbb{N}^n$ is called noncomputable if it does not satisfies the above condition.*

Proposition 2 *There exists noncomputable partial functions*[3]*.*

[3] Use the fact that $Func(\mathbb{N}, \mathbb{N})$ is uncountable, while the semi-computable functions are countable.

Proposition 3 *For semicomputable functions, that it is computable is equivalent to that the character function of its domain is computable.*

Example of Fermat's problem (semicomputable function, but not computable): $f(n) = 1$ if there exists a positive integral solution to the equation $x^{n+2} + y^{n+2} = z^{n+2}$.

How to describe semi-computable and computable partial functions? Church's Thesis (usual form) (a) A function f is semicomputable if and only if it is partial recursive. (b) A function f is computable if and only if both f and $\chi_{D(f)}$ are partial recursive[4].

[4] Although hard to realize, there are plenty of partial recursive functions. And by a class of operations we can generate a great more. Church's result tells us what those are, and how to construct new partial recursive functions from the old ones

Theorem 6 *There is no program which each input p, determines if p is a program which halts on all of its inputs.*

Theorem 7 *There is no program $R(p, i)$ which for each program p and each input i, can determine "yes" or "no" if p halts on i.*

Lemma 1 *(Gödel number.) There exists a primitive recursive function $Gd(k, t)$ (Gödel's function) with the following property: for any $N \in \mathbb{N}$ and any finite sequence $a_1 \cdots a_N \in \mathbb{N}$ of length N, there exists $t \in \mathbb{N}$ such that $Gd(k, t) = a_k$ for all $1 \le k \le N$*[5]

[5] In other words, the function Gd allows us to consider integers as encoding arbitrarily long sequences of integers: $Gd(k, t)$ is the k-th member of the sequence encoded by t, and the existence assertion ensures that each sequence has an encoding. In this way, the sequence $a_1 \cdots a_N \in \mathbb{N}$ uniquely corresponds to a number t, which we call the Gödel number.

Lemma 2 *(Self-reference Lemma.) Given any formula $P(x)$ in the language that has one free variable, we can effectively construct a closed formula Q_P that says, "my number does not belong to the set defined by P". In other words, Q_P is true if and only if $P(|\neg Q_P|)$ is false*[6]*.*

[6] From here, P can be viewed as a proposition-valued function on $\mathbb{N}$, and Gödel number $|-|$ assigns a natural number to the corresponding proposition.

Exercise 4 *There is another form of self-reference lemma, which does not apply the negation, meaning that one can reconstruct a proposition from its Gödel number. Try to give this form of the lemma.*

Theorem 8 (Tarski) *TRUTH, the set of numbers which encode true sentences of number theory, is not definable in number theory.*

1. Suppose that truth is definable by a formula P.
2. Then there is a formula Q_P that says âĂIJI am not true.âĂİ
3. The formula Q_P cannot be false (because of its semantics).
4. The formula Q_P cannot be true (because of its semantics).
5. Therefore, truth is not definable.

Theorem 9 (Gödel's First Incompleteness Theorem) *Any adequate axiomatizable theory is incomplete. In particular the sentence "This sentence is not provable" is true but not provable in the theory.*

1. Provability is definable by a formula P.
2. There is a formula Q_P that says "I am not provable".
3. The formula Q_P cannot be false (because of its semantics, since otherwise it would be provable, and hence true).
4. Therefore, Q_P is true.
5. Therefore, Q_P is not provable (because of its semantics).

Theorem 10 (Gödel's Second Incompleteness Theorem) *In any consistent axiomatizable theory (axiomatizable means the axioms can be computably generated) which can encode sequences of numbers (and thus the syntactic notions of "formula", "sentence", "proof") the consistency of the system is not provable in the system.*

Complexity of algorithms

Definition 4 • *A function f is said to be in big-O of g, if there exists N and c constant, such that for all $n > N$, $|f(n)| \leq c \cdot |g(n)|$.*

- *A function f is said to be in big-Ω of g, if there exists N and c constant, such that for all $n > N$, $|f(n)| \geq c \cdot |g(n)|$.*
- *A function f is said to be in big-Θ of g if it is in $O(g) \cap \Omega(g)$*[7]*.*

[7] Common reference functions (the "g" function used in the definitions) are constant, linear, power law, logarithmic, exponential, and n-log. But exponential algorithms are considered as unrealistic, since it is practically impossible to do large computations using them.

Exercise 5 *Show that if $f_1 \in \Theta(g_1)$ and $f_2 \in \Theta(g_2)$, then $f_1 + f_2 \in \Theta(max\{g_1, g_2\})$, and $f_1 \cdot f_2 \in \Theta(g_1 \cdot g_2)$.*

Exercise 6 *Try to give sufficient conditions for each of those above definitions using limit.*

Example 5 (Searching Problem) *Find the first place of appearance of a character in a string of length N; return 0 if it is not found.*

Sequential Search, pseudocode, average behavior: linear.

Binary Search, pseudocode, average behavior: logarithmic.

To compare differential algorithms, one needs to compare the number of critical operations, which, in the above example, is the times of comparisons. In practice, one can also compare the time for some n, and then use a fitting with corresponding law.

Example 6 (Counting critial operations) *The following fragment of pseudocode represents part of an algorithm. The critical operations are the comparison on line 3 and the swap on line 4. It is easier to focus on line 3, since it will be executed every time. What is a good big-Θ estimate of the number of comparisons?*

```
1: for i = 1 to n
2:     for j = 1 to n
3:         if a[i] < a[j]
4:             swap a[i] and a[j]
```

Notice that the loop at line 1 (with index i) will be executed n times. The inner loop (line 2, with index j) will also be executed $n - i + 1$ times. Each time line 3 is executed we need to add 1 to the comparison count. The total number of comparisons will be $\frac{n\cdot(n+1)}{2}$.

Example 7 (Pattern Matching) *Find the first occurrence of a sub-string of length m (pattern) inside a string of length n.*

The obvious algorithm: best case: compare m times; the worst time, compare $m \cdot (n - m + 1)$-times, so in average $\frac{m\cdot(n-m)}{2}$ times, and hence the complexity is $\Theta(m \cdot (n - m))$.

What can we do better? Comparisons are done multiple times...

Knuth-Morris-Pratt: create a table of shift to denote the number of shifts one should do if there is a mismatch at the place k for all $1 \leq k \leq m$. The outcome is two-folded: firstly, one can shift more than one step after a mismatch; and secondly, once a shift is done, there is no need to examine the characters in the n-string to the left of the miss.

Exercise 7 *Compute the shift table for the string "mama", "aaaab" and "baaaa".*

Alternatively, the KMP method can be understood using the theory of borders. In that approach, it is crucial to define and compute a different table $b[m]$ from the pattern $p[m]$. Note that the shift table $s[m]$ can also be obtained using the similar method.

```
1: void kmpPreprocess()
```

```
{
  int i=0, j=-1;
  b[i]=j;
  while (i<m)
  {
    while (j>=0 && p[i]!=p[j]) j=b[j];
    i++; j++;
    b[i]=j;
  }
}
```

Exercise 8 *Analyze the complexity of the above code.*

Note: we shall not cover the following in class, so you can safely ignore those.

In practice, KMP doesn't do much better than the obvious algorithm for most types of text and pattern. This is because most misses occur on or near the first character in the pattern.

Boyer-Moore: create two tables: one for the text (last table) and one for the pattern (shift table)

- *searches for the pattern from right to left;*
- *knows about all possible characters in the pattern;*
- *if the corresponding char is not in the pattern set, move m places;*
- *if it is in the pattern set, then the shift is decided by the pattern itself.*

The shift table: similar to the KMP shift table, but the comparison starts from right to left. We want the smallest shift which Don't Repeat the Miss [8] *and which do Repeat the Hits* [9].

The last table: Denote the table by L and index the table by the characters in the text alphabet, in lexicographical order. The value under text character x is the position of the rightmost occurrence of x in the pattern, or 0 if x does not appear in the pattern.

```
integer BoyerMoore(string T, string p)
L lastTable(p)
D shiftTable(p)
i = 1 # start at the first character of the text
while i < n - m + 1 # pattern will extend past text if i > n-m+l
  j = m # start at the right of the pattern
  while (j > 1) and (T[i+j-1] == p[j]) # still in pattern and a hit
    j = j-1
  if j == 0 # has every character in the pattern been matched?
    return i # yes, exit and return position of match
  i = i + Max(D[j], j-L[T[i+j-1]]) # no, shift and try again
```

[8] I.e., we don't shift the same letter in the pattern to the spot where the miss just occurred.

[9] I.e., we need every character shifted to the right of the miss to match the previous character known to reside in that spot.

```
11: return -1 # pattern not found
12: end BoyerMoore
```