CS 4810 — Fall'13

Syllabus | Lectures | Homework | Resources

Lecture 1: Boolean Functions

Boolean values

There are two *Boolean values*, TRUE and FALSE. For convenience, we will denote these values as 1 and 0. Boolean values are often also called *bits*. We will use Boolean values to encode information.

Basic Boolean operations

Basic Boolean operations are functions that take one or two Boolean values as input and output a Boolean value. We will focus the following operations:

$$\text{NOT}(x) := \begin{cases} 1 & \text{if } x = 1, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{AND}(x, y) := \begin{cases} 1 & \text{if } x = 1 \text{ and } y = 1, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{OR}(x, y) := \begin{cases} 1 & \text{if } x = 1 \text{ or } y = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The following notation is common:

$$\neg x \text{ for } \text{NOT}(x),$$

$$x \wedge y \text{ for } \text{AND}(x, y),$$

$$x \vee y \text{ for } \text{OR}(x, y).$$

Truth table

We can represent the Boolean operations NOT, AND, OR by the following *truth table*.

x	y	$\text{NOT}(x)$	$\text{AND}(x, y)$	$\text{OR}(x, y)$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

De Morgan's Laws

There are redundancies among the operations NOT, AND, OR. In particular, we can write AND in terms of NOT, OR. Similarly, OR can be expressed in terms of NOT, AND.

$$\neg(x \wedge y) = \neg x \vee \neg y,$$

$$\neg(x \vee y) = \neg x \wedge \neg y.$$

We can check that these identities hold by looking at the above truth table.

Boolean functions

> An n-bit *Boolean function* f is a function that take n Boolean values as input and outputs a Boolean value, i.e.,
>
> $$f{:}\{0,1\}^n \to \{0,1\}.$$

Boolean function can express interesting algorithmic questions:

$$\text{PRIMES}_n(x) := \begin{cases} 1 & \text{if } x \text{ is (the binary encoding of) a prime number,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{RIEMANN}_n(x) := \begin{cases} 1 & \text{if } x \text{ is (the binary encoding of) a proof of the Riemann hypothesis,} \\ 0 & \text{otherwise.} \end{cases}$$

(The Riemann hypothesis is an important open question in mathematics. If you resolve the hypothesis, you get a Millennium Prize.)

We will try to understand whether these kind of functions can be expressed by a small number of basic Boolean operations. (It turns out that both functions above can be expressed by a number of basic Boolean operations that is polynomial in n.)

Lecture 2: Boolean Circuits

Boolean circuit

A *Boolean circuit* with n inputs and 1 output is a directed acyclic graph that consists of the following kind of nodes:

- *Input nodes*, labeled $1, \ldots, n$, without ingoing edges (*sources*).
- Any number of *gates*, labeled with NOT, AND, OR.
  - All AND, OR gates have two incoming edges.
  - All NOT gates have one incoming edge.
- One *output gate*, without outgoing edges (*sink*). (All other gates have at least one outgoing edge.)

Circuit computation

Suppose we assign Boolean values $x_1, \ldots, x_n$ to the input nodes of a circuit. Then, we can propagate these values through the gates of the circuit by applying the corresponding Boolean operations.

More formally, we would process the gates of the circuit in topological order. To process a gate u labeled with a Boolean operation $T \in \{\text{NOT}, \text{AND}, \text{OR}\}$, we apply the operation T to the values assigned to the ancestors of u and we assign the output of this operation to u. (Since we process nodes in topological order, the ancestors of u have already been processed at the time we process u.)

We say a Boolean circuit C with n inputs and 1 output *computes* an n-bit Boolean function $f:\{0,1\}^n \to \{0,1\}$ if for every input $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, the function value $f(x)$ is equal to the value propagated to the output gate of C when we assign $x_1, \ldots, x_n$ to the input nodes of C.

One can show (e.g., by induction) that a circuit computes only one function. In particular, the function does not depend on the order we process the gates. We write f_C to denote the function computed by a circuit C.

Example: parity

Consider the following n-bit Boolean function

$$\text{PARITY}_n(x) := \begin{cases} 1 & \text{if } x_1 + \ldots + x_n \text{ is an odd number,} \\ 0 & \text{otherwise.} \end{cases}$$

For $n = 2$, we can express PARITY in terms of NOT, AND, OR as follows

$$\text{PARITY}_2(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

The parity of 2 bits is called *exclusive-or*. A common notation for $\text{PARITY}_2(x_1, x_2)$ is $x_1 \oplus x_2$.

If n is a power of 2, we can construct a circuit for PARITY recursively based on the following identity

$$\text{PARITY}_n(x) = \text{PARITY}_{n/2}(x_1, \ldots, x_{n/2}) \oplus \text{PARITY}_{n/2}(x_{n/2+1}, \ldots, x_n).$$

Circuit complexity

For a given Boolean functions, there could be many ways of computing it by a circuit. But some circuits are better than others.

We measure the efficiency of a circuit C by its *size*

$$\text{size}(C) := \text{the number of gates of } C.$$

For a Boolean function f, we define its *size* as the minimum size of circuit that computes f.

Another measure for the efficiency of a circuit is its *depth*

$$\begin{aligned} &\text{depth}(C) \\ &:= \text{length of longest path from an input node to the output gate} \end{aligned}$$

Generic upper bound

> **Theorem:** For every n-bit Boolean function f there exists a circuit of size at most $10 \cdot 2^n - 10$ to compute it.

Proof: We will use induction on n. If we fix one of input bit of the function f, then we are left with an $n-1$-bit Boolean function.

For $n = 1$, the theorem is true because for every 1-bit function, we can find a circuit of size at most 10. Suppose $n > 1$. Let f_0 and f_1 be the two $n-1$ bit functions obtained by fixing the first bit of f, i.e.,

$$f_0(x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n),$$

$$f_1(x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n).$$

Both f_0 and f_1 are $n-1$ bit functions. The following identity allows us to express the original function f in terms of f_0 and f_1 as well as a few basic operations

$$f = (\neg x_1 \wedge f_0) \vee (x_1 \wedge f_1).$$

Therefore, we can construct a circuit for f from a circuit for f_0 and a circuit for f_1 as well as 4 additional gates (one NOT gate, one OR gate, and two AND gates). It follows that $\text{size}(f) \leq \text{size}(f_0) + \text{size}(f_1) + 4$. By induction hypothesis, both $\text{size}(f_0)$ and $\text{size}(f_1)$ are at most $10 \cdot 2^{n-1} - 10$. By combining these bound, we get

$$\text{size}(f) \leq 10 \cdot 2^n - 20 + 4 = 10 \cdot 2^n - 16 \leq 10 \cdot 2^n - 10.$$

Lecture 3: Hard functions

Generic lower bound

Theorem: For every $n \in \mathbb{N}$ with $n \geq 100$, there exists an n-bit Boolean function that requires circuits of size at least $2^n/100n$.

Proof: The idea is to compare the number of n-bit Boolean functions $f:\{0,1\}^n \to \{0,1\}$ to the number of circuits of size at most $s = 2^n/100n$. If the number of functions is larger than the number of such circuits, it means that there exists a functions that is not computed by such a circuit.

Let us first count the number of n-bit Boolean functions. In general, the number of functions from a finite set X to a finite set Y is equal to $|Y|^{|X|}$. (You should convince yourself of this fact.) Hence, the number of n-bit Boolean functions is 2^{2^n}.

Now let's count the number of size-s circuits. A circuit of size s consists of a directed acyclic graph (DAG) on $s' = s + n \leq 2s$ nodes (n input nodes and s gates). Each node has in-degree at most 2. (The in-degree of a node is the number of ingoing edges.) Furthermore, each input node (in-degree 0) is labeled with an input variable $i \in [n]$. Each gate is labeled with a Boolean operation, either AND, OR, or NOT.

We claim that we can describe such a circuit completely by a bit string of length at most $10s \log s$. First, we give every node a unique ID. Since there are only s' nodes in the circuit, we only need $\log s'$ bits for the ID. (We can use the numbers from 1 to s' as IDs.) Our description of the circuit will consist of s' parts, each of length at most $10 \log s'$ (one part per vertex). To describe a single vertex u, we store the ID of u ($\log s'$ bits), the IDs of the vertices that have incoming arcs to u ($2 \log s'$ bits) and the label of u, which is either a Boolean function or the index of an input/output variable (at most $\log n \leq \log s'$ bits). We don't need to store the outgoing arcs of a vertex, because those will be stored as incoming arcs by other vertices. In total, we store at most $4 \log s'$ bits per vertex. Hence, the description of the whole circuit is indeed shorter than $10s \log s$ bits.

It follows that the number of circuits of size s is at most $2^{10s \log s}$ (the number of bit strings of length $10s \log s$). For $s = 2^n/100n$, this number is much less than 2^{2^n}. (Check this calculation.) It follows that an overwhelming majority of n-bit Boolean functions does not have circuits of size $2^n/100n$.

\

Remark: While the theorem shows that some functions require large circuits, the proof of the theorem does not give an explicit construction of such a function. (The proof is non-constructive.) It's a famous open problem to show strong lower bounds on the circuit size of concrete functions. For example, the following funtion is conjectured to require circuits of exponential size

$$\text{CIRCUIT}-\text{SAT}_n(x) := \begin{cases} 0 & \text{if } x \text{ is the binary encoding of a circuit} \\ & \text{that computes the constant 0 function,} \\ 1 & \text{otherwise.} \end{cases}$$

Straight-line Programs

In this part, we will discuss an exact correspondence between circuits and a simple kind of programs.

Here is an example of a straight-line program:

Input: x_1, x_2

Instructions:

- $y_1 := \neg x_1$
- $y_2 := \neg x_2$
- $y_3 := x_1 \wedge y_2$
- $y_4 := y_1 \wedge x_2$
- $y_5 := y_3 \vee y_4$

Output: y_5

This program outputs the parity of the two input bits. We can construct a circuit that corresponds to this program in the following way: For every instruction, we have a gate in the circuit that applies the same Boolean operation and has incoming edges from nodes corresponding to the right-hand-side variables of the instruction. The size of the resulting circuit is exactly equal to the number of instructions of the straight-line program.

Definition: A *straight-line program* is a sequence of instructions of the following form: Each instruction applies a basic Boolean operation (NOT, AND, OR) to input variables or previously assigned variables and assigns the outcome of this operation to a new variable. The output of the program is the outcome of last instruction. The *length* of a straight-line program is the number of instructions.

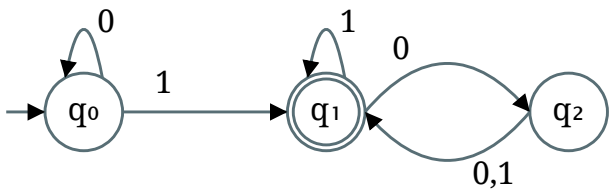
Exercise: A Boolean function has a straight-line program of length at most ℓ if and only if it has a Boolean circuit of size at most ℓ.

CS 4810 – Fall'13

Syllabus | Lectures | Homework | Resources

Lecture 4: Deterministic Finite Automata

The following figure shows the *state diagram* of a deterministic finite automaton (DFA),



The automaton has three *states* q_0, q_1, and q_2. The state q_0 is designated as the *start state* and q_1 is designated as an *accept state*. The directed edges between states specify the *transitions* of the automaton when reading an input string.

This automaton accepts the set of strings that contain 1 and have an even number of 0's after the last 1.

Languages

An *alphabet* Σ is a finite set of symbols, e.g., $\Sigma = \{0,1\}$ or $\Sigma = a, b, c$. (We will usually content ourselves with the binary alphabet.) A *string* w over Σ is a finite sequence of symboles, written as $w = x_1 \cdots x_n$ for $x_1, \ldots, x_n \in \Sigma$. The *empty string* (string of length zero) is denoted ε. The set of all strings is denoted Σ^*. A *language* L is any subset of strings, so that $L \subseteq \Sigma^*$.

We will use languages to encode computational problems. (A language L can be thought to encode the function $f{:}\Sigma^* \to \{0,1\}$ with $f(w) = 1$ if and only if $w \in L$).

Operations on languages: Let $A, B \subseteq \Sigma^*$ be two languages. The following operations on languages are useful:

- Basic set-theoretic operations, *union* $A \cup B$, *intersection* $A \cap B$, and *complementation* $\overline{A} := \Sigma^* \setminus A$. (These operations correspond to OR, AND, NOT.)
- *Concatentation* $AB := \{uv \mid u \in A, v \in B\}$.
- *Kleene star* $A^* := A^0 A^1 A^2 \ldots$, where A^i denotes the i-fold concatenation of A with itself. (The corner cases $i = 0, 1$ are defined so that the rule $A^i A^j = A^{i+j}$ holds. Concretely, $A^0 = \{\varepsilon\}$ and $A^1 = A$).

(Notice that the Kleene star notation is compatible with the notation for the set of all strings.)

Deterministic Finite Automaton

Definition: A *deterministic finite automaton* M consists of the following parts:

- a *transition function* δ:$Q \times \Sigma \to Q$ for an *alphabet* Σ and a set of *states* Q,
- a *start state* $q_0 \in Q$,
- a set of *accept states* $F \subseteq Q$.

Language of a DFA

Definition: A DFA M *accepts* a string $w = x_1 \cdots x_n$ if there exists a sequence of states $r_0, r_1, \ldots, r_n \in Q$ such that

- the first state is the start statem $r_0 = q_0$
- the ith state is obtained by reading the ith symbol, $r_i = \delta(r_{i-1}, x_i)$
- the final state is an accept state, $r_n \in F$.

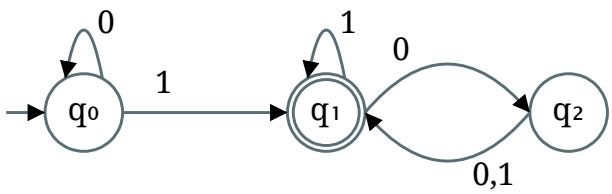
The language of M is defined as

$$L(M) = \{w \mid M \text{ accepts } w\}.$$

If L is the language of some deterministic finite automaton, we say that L is *regular*.

Example proof

Recall the automaton M from the beginning:



Lemma: The language of the automaton M is the set of all strings $w \in \{0,1\}^*$ such that w contains a 1 and ends with an even number of 0's after the last 1.

We will prove this lemma by inductions. To make the proof easier, we will show a stronger statement. (Proofs by induction often become easier if the statement is strengthened.)

Define the *repeated transition function* δ^*:$\Sigma^* \to Q$,

$$\delta^*(w) := \text{state of } M \text{ after reading } w$$

This function satisfies $\delta^*(wx) = \delta(\delta^*(w), x)$ for every string $w \in \Sigma^*$ and symbol $x \in \Sigma$. Furthermore, $L(M) = \{w \mid \delta^*(w) \in F\}$. Hence, the following claim implies the lemma.

Claim: The repeated transition function of M is

$$\delta^*(w) = \begin{cases} q_0 & \text{if } w \text{ does not contains } 1 \ (A), \\ q_1 & \text{if } w \text{ contains } 1 \text{ and ends with an even number of } 0\text{'s after the the last } 1 \ (B) \\ q_2 & \text{if } w \text{ contains } 1 \text{ and ends with an odd number of } 0\text{'s after the the last } 1 \ (C) \end{cases},$$

Proof of claim: By induction on the length of w. (At this point, the proof is completely mechanical.)

If $w = \varepsilon$, then $\delta^*(w) = q_0$ and the claim is true (because $\varepsilon \in A$).

Suppose length of $w = ux$ for $x \in \{0,1\}$ is greater than 0. The induction hypothesis is that the claim holds for all strings that are shorter than w. (In particular, the claim holds for the substring u of w.)

We consider three cases:

» $u \in A$: By induction hypothesis, $\delta^*(u) = q_0$. Therefore, .

$$\delta^*(w) = \delta(q_0, x) = \begin{cases} q_0 & \text{if } x = 0, \\ q_1 & \text{if } x = 1. \end{cases}$$

The claim is true in this case because $u0 \in A$ and $u1 \in B$.

» $u \in B$: By induction hypothesis, $\delta^*(u) = q_1$. Therefore,

$$\delta^*(w) = \delta(q_1, x) = \begin{cases} q_2 & \text{if } x = 0, \\ q_1 & \text{if } x = 1. \end{cases}$$

The claim is true in this case because $u0 \in C$ and $u1 \in B$.

» $u \in C$: By induction hypothesis, $\delta^*(u) = q_1$. Therefore,

$$\delta^*(w) = \delta(q_1, x) = \begin{cases} q_1 & \text{if } x = 0, \\ q_1 & \text{if } x = 1. \end{cases}$$

The claim is true in this case because both $u0 \in B$ and $u1 \in B$.

We conclude that the claim is true in all three cases, which proves the claim.

Lecture 5: Closure Properties

We will show that the set of regular languages is closed under basic set-theoretic operations, *intersection*, *union*, and *complement*.

These closure properties are useful to show that languages are regular or to show that they are not.

> **Theorem:** Let $A, B \subseteq \Sigma^*$ be two regular languages over Σ. Then, the following languages are also regular:
>
> - *intersection:* $A \cap B = \{w \in \Sigma^* \mid w \in A \text{ and } w \in B\}$,
> - *union:* $A \cup B = \{w \in \Sigma^* \mid w \in A \text{ or } w \in B\}$,
> - *complement:* $\bar{A} = \{w \in \Sigma^* \mid w \notin A\}$ and $\bar{B} = \{w \in \Sigma^* \mid w \notin B\}$.
>
> Furthermore, if M_A and M_B are DFAs for A and B, then the languages above have DFAs with a number of states that is polynomial in the number of states of M_A and M_B.

Proof: Suppose M_A and M_B are deterministic finite automata for A and B. Let δ_A:$Q_A \times \Sigma \to Q_A$ and δ_B:$Q_B \times \Sigma \to Q_B$ be the transition functions of M_A and M_B. Here, Q_A and Q_B are the states sets of M_A and M_B. Let $F_A \subseteq Q_A$ and $F_B \subseteq Q_B$ be the accept states of M_A and M_B.

Complement: We are to construct a DFA M for the complement $\bar{A}$. Let M be the same automaton as M_A except with accept states $F = Q_A \setminus F_A$. (Concretely, M has the same transition function, set of states, alphabet, and start state as M_A.) Then, M accepts a word $w \in \Sigma^*$ if and only if $\delta^*(w) \in F = Q_A \setminus F_A$. By construction, M_A is in the same state after reading w, i.e., $\delta^*(w) = \delta_A^*(w)$. Hence, M accepts w if and only if $\delta_A(w) \notin F_A$, which means that M_A does not accept w. It follows that $L(M) = \Sigma^* \setminus L(M_A) = \bar{A}$.

Intersection: We are to construct a DFA M for the intersection $A \cap B$. The idea is that M simulates both M_A and M_B at the same time. The state space of M is the product set $Q = Q_A \times Q_B$. The transition function is

$$\delta\big((q_A, q_B), x\big) = \big(\delta_A(q_A, x), \delta_B(q_B, x)\big)$$

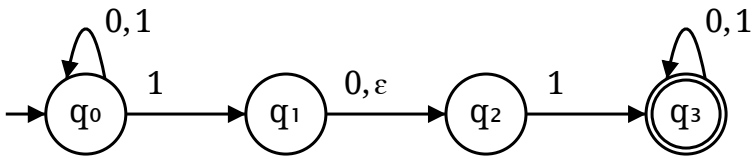
The start state of M has the start space of M_A in the first coordinate and the start space of M_B in the second coordinate. Finally the accept states of M are $F = F_A \times F_B$. With this construction, M is in state $\delta^*(w) = \big(\delta_A^*(w), \delta_A^*(w)\big)$ after reading w. Hence, M accepts w if and only $\delta_A^*(w) \in F_A$ and $\delta_B^*(w) \in F_B$, which means that $w \in A \cap B$.

Union: We could construct an automaton for $A \cup B$ similar to the product construction above. However, we can also use the properties we have proved so far to conclude that $A \cup B$ is regular whenever A and B are regular. By de Morgan's law, we can express the union in terms of complements and intersection, $A \cup B = \overline{(\bar{A} \cap \bar{B})}$. Since we have already shown that regular languages are closed under complement and intersection, it follows that $A \cup B$ is regular.

Lecture 6: Non-Deterministic Finite Automata

Example

The following figure shows the *state diagram* of a non-deterministic finite automaton (NFA),



The automaton accepts the set of all strings that contain 11 or 101 as a substring.

Non-deterministic Finite Automata

Definition: A *non-deterministic finite automaton* (NFA) M consists of the following

- a set of *states* Q and *transitions* $T \subseteq Q \times Q \times (\Sigma \cup \{\varepsilon\})$ between states labeled by alphabet symbols or the empty string ε,
- a start state $q_0 \in Q$,
- a set of accept states $F \subseteq Q$.

Language of Non-deterministic Finite Automata

Definition: A *computation path* of an NFA M for a string $w \in \Sigma^*$ is a sequence of transitions $(r_0, r_1, x_1), (r_1, r_2, x_2), \ldots, (r_{m-1}, r_m, x_m) \in T$ such that $r_0 = q_0$ is the start state and $w = x_1 \cdots x_m$. An NFA M *accepts* a string w if there exists a computation path for w that ends in an accept state, so that $r_m \in F$.

The *language* of an NFA is the set of strings that M accepts,

$$L(M)$$
$$= \{w \in \Sigma^* \mid \text{there exits an accepting computation path in } M \text{ for } w\}.$$

Closure Properties of Regular Languages

> **Theorem:** The set of regular languages is closed under union, concatenation, and Kleene star operations.

Lemma: Let M_1 and M_2 be NFAs, then there exists an NFA M with

$$L(M) = L(M_1) \cup L(M_2).$$

Proof sketch: Create a designated start state with ε-transitions to the start state of M_1 and the start state of M_2. An accepting computation path in M corresponds to either an accepting computation path in M_1 or an accepting computation path in M_2. $\square$.

Lemma: Let M_1 and M_2 be NFAs, then there exists an NFA M with

$$L(M) = L(M_1)L(M_2).$$

Proof sketch: Add ε-transitions from the accept states of M_1 to the start state of M_2. An accepting computation path in M corresponds to an accepting computation paths in M_1 followed by an accepting computation path in M_2. $\square$.

Lemma: Let M_0 be an NFA, then there exists an NFA M with

$$L(M) = L(M_0)^*.$$

Proof sketch: Create a designated start state with a ε-transition to the start state of M_0. Add ε-transitions from the accept states of M_0 to this designated start state. Make this designated start state the unique accept state of M. An accepting computation path in M corresponds to a finite sequence of accepting computation paths in M_0. $\square$.

Lecture 7: Non-deterministic vs Deterministic Finite Automata

The following theorem shows that deterministic and non-deterministic finite automata accept the same class of languages.

> **Theorem:** For every non-deterministic finite automaton N, there exists a deterministic finite automaton M that accepts the same language, $L(M) = L(N)$.

Proof: The idea is to have states in M that correspond to subsets of states in N. Let Q_N be the states of N. We want that for every word $w \in \Sigma^*$, the automaton M is in the following state after reading w,

$$\begin{aligned}&\delta_M^*(w)\\ &= \{q \in Q_N\\ &\mid N \text{ has a computation path for the word } w \text{ that ends in the state } q\}\end{aligned}.$$

If we have an automaton M with this property, then it has the same language as N if we choose the accept states of M as

$$F_M = \{Q \subseteq Q_N \mid Q \text{ contains an accept state of } N\}.$$

Let q_0 be the start state of N. As start state for M, we choose

$$Q_0 = \{q \in Q_N \mid q \text{ can be reached from } q_0 \text{ by } \varepsilon\text{-transitions}\}.$$

Finally, the transition function of M is defined as

$$\begin{aligned}\delta_M(Q, x) = \{q \in Q_N | &q \text{ can be reached from a state in } Q \text{ by } \varepsilon\text{-transitions}\\ &\text{and exactly one } x\text{-transition}\}.\end{aligned}$$

With this transition function and start state, the repeated transition function δ_M^* indeed has the property above (proof by induction). Therefore,

$$\begin{aligned}w \in L(M) \Leftrightarrow &\delta_M^*(w) \in F_M\\ \Leftrightarrow &N \text{ has a computation for } w \text{ that ends in an accept state of } N\\ \Leftrightarrow &w \in L(N).\end{aligned}$$

Lecture 8: Regular Expressions

Regular Expressions

> **Definition:** A *regular expression* over an alphabet Σ is a formula with alphabet symbols $x \in \Sigma$, the empty set $\emptyset$, and the empty string ε as constants, and union, concatenation, and Kleene star as operations.

Order of precedence: Kleene star has the highest precedence, followed by concatenation, and then union. For example, $0 \cup 10^*$ is short for $0 \cup (1(0^*))$.

Example: The following expression describes the set of strings that contain 11 or 101 as substring.

$$(0 \cup 1)^* 1 (0 \cup \varepsilon) 1 (0 \cup 1)^*$$

The *language* of a regular expression R is defined recursively. In the base case, R is a constant. Then,

- $L(x) = \{x\}$ for all $x \in \Sigma$,
- $L(\varepsilon) = \{\varepsilon\}$,
- $L(\emptyset) = \emptyset$.

Otherwise, a regular expression is obtained by applying an operation to one or two smaller expressions. In this case,

- $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$ (union),
- $L(R_1 R_2) = L(R_1) L(R_1)$ (concatenation),
- $L(R^*) = L(R)^*$ (Kleene star).

Regular Expressions vs. Finite Automata

The following theorem shows that the set of languages generated by regular expressions is the same as the set of languages accepted by finite automata.

> **Theorem:** For every language A, there exists a regular expression R with $L(R) = A$ if and only if there exists a finite automaton M with $L(M) = A$.

To show this theorem, we will first show how to construct an automaton from a given regular expression. Then, we will show how to construct a regular expression from a given finite automaton.

From regular expression to finite automaton: Let R be a regular expression. We can construct a finite automaton M with $L(M) = L(R)$ recursively. The idea is to use the fact that the set of languages of finite automata is closed under union, concatenation, and Kleene star operations.

In the base case, R is a constant, either $R = x$ for $x \in \Sigma$, $R = \varepsilon$ or $R = \emptyset$. In all cases, $L(R)$ is finite. Hence, there exists a finite automaton for $L(R)$. (See Problem 2 in Homework 2.)

Otherwise, R is an operation applied to one or two smaller expressions. Either, $R = R_1 \cup R_2$, $R = R_1 R_2$, or $R = R_1^*$. Since R_1 and R_2 are smaller regular expressions, we can construct automata M_1 and M_2 with $L(M_1) = L(R_1)$ and $L(M_2) = L(R_2)$. Then, by closure properties of finite automata (see Lecture 6), there exist finite automata for the languages $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$ and $L(M_1)^*$. Therefore, there exists a finite automaton for $L(R)$.

From finite automaton to regular expression: Let M be a finite automaton. We are to construct a regular expression with the same language as M. The idea is to transform M step-by-step into a regular expression. We will refer to the intermediate objects as *generalized finite automata* (GFA). In contrast to the usual kind of finite automata, generalized finite automata have transitions labeled by regular expressions. We say that a generalized finite automaton M accepts a word w if there exists an accepting computation path of M with transitions $R_1, \ldots, R_m$ such that $w \in L(R_1) \cdots L(R_m)$.

We say that a GFA has *dedicated start and accept states*, if it has exactly one start state and exactly one accept state and if no transitions go into the start state and no transitions go out of the accept state.

> **Lemma:** Suppose M is a GFA with dedicated start and accept states and with at least one additional state (not the start or accept state). Then, there exists a GFA M' with dedicated start and accept states that has fewer states than M, but accepts the same language, $L(M') = L(M)$.

Proof: Let q_0 be an additional state of the automaton M (not the start of accept state). We show how to remove this state and update the transitions between the remaining states such that the language of the automaton doesn't change. For simplicity, we will assume that M contains a transition labeled by R_0 from q_0 to itself. (We can take $R_0 = \emptyset$ if M doesn't contain such a transition.)

> **Construction of M':** Let $q_0, q_1, \ldots, q_m$ be the states of M. Let $R_{i\to j}$ be the label of the transition from q_i to q_j, or if there is no such transition, let $R_{i\to j} = \emptyset$. For all $i, j \geq 1$, we label the transition from q_i to q_j in M' by
>
> $$R'_{i\to j} = R_{i\to j} \cup R_{i\to 0} R^*_{0\to 0} R_{0\to j}.$$

We claim that the GFA M' has the same language as M. The reason is that every computation path in M that uses q_0 can be shortcut to an equivalent computation path in M'. Similarly, every computation path in M' can be extended to an equivalent computation path in M. $\square$

Lectures 9 & 10: Limitations of Finite Automata

Pumping Lemma

The following theorem, known as *pumping lemma* gives a way to show that a language is not regular.

> **Theorem:** Suppose L is the language of a finite automaton M. Then, there exists a number p (called *pumping length*) such that for every word $w \in L$ of length at least p, there exists a decomposition $w = xyz$ with the following properties:
>
> » $xy^iz \in L$ for every integer $i \geq 0$,
>
> » $|y| > 0$,
>
> » $|xy| \leq p$.

Proof: We may assume that M doesn't have ε-transitions. (We could even assume M is deterministic.) We choose p as the number of states of M. Let $w \in L$ be a word in the language L. We are to show a decomposition of w with the properties above.

Consider a computation path of M that accepts w. Since w has length at least p, one state of M appears at least twice on this computation path (by the *pigeonhole principle*). Let q be the first repeated state on the computation path. We divide the computation path into three parts: from the start state to the first occurance of q, from the first occurance of q to its second occurance, and from the second occurance of q to an accept state.

Let x, y, z be the strings corresponding to these three parts of the computation path. Then, $w = xyz$. We are to check that this decomposition satisfies the above properties. First, $|y| > 0$ because M doesn't have ε-transitions. Second, $|xy| \leq p$ because q is the first repeated state and therefore all other states on the first and second part of the path are distinct. Finally, $xy^iz \in L$ for every integers $i \geq 0$, because we can obtain accepting computation paths for these strings by repeating the second part of the paths i times. □

Examples

Lemma: The language $\{0^n 1^n \mid n \in \mathbb{N}\}$ is not regular.

Proof: We will show that for any potential pumping length p, there exists a word $w \in L$ of length $|w| \geq p$ that does not admit a decomposition as in the pumping lemma. Then, from the pumping lemma it follows that the language cannot be the language of a finite automata.

Let p be a natural number. Consider the word $w = 0^p 1^p$ in L. Let $w = xyz$ be any decomposition of this word that satisfies the second and third condition of the pumping lemma. Then, $x = 0^s$ and $y = 0^t$ for integers $s \geq 0$ and $t \geq 1$ (because $|xy| \leq p$ and $|y| \geq 1$). But then $xy^2z = 0^{p+t}1^p \notin L$, which means that the decomposition doesn't satisfy the first condition of the pumping lemma. □

\

The next example shows that sometimes one has to be careful to choose the right word w in the language to show that the conditions of the pumping lemma are not satisfied.

Lemma: The language $\{ww \mid w \in \{0,1\}^*\}$ is not regular.

Proof: Same proof strategy as before.

Let p be a natural number. Consider the word 0^p10^p1 in L. Let $w = xyz$ be any decomposition of this word that satisfies the second and third condition of the pumping lemma. As before, $x = 0^s$ and $y = 0^t$ for integers $s \geq 0$ and $t \geq 1$. Again $xy^2z = 0^{p+t}10^p1$ is not in L, which means that the decomposition doesn't satisfy the first condition of the pumping lemma. (Convince yourself that for example, for $w = (01)^p \in L$ the proof would fail.) □

\

The following example shows that sometimes it is useful to "pump down" (which means to choose $i = 0$ to show that the first condition in the pumping lemma is violated).

Lemma: The language $\{0^i 1^j \mid i \geq j\}$ is not regular.

Proof: Same proof strategy as before.

Let p be a natural number. Consider the word 0^p1^p. Let $w = xyz$ be any decomposition of this word that satisfies the second and third condition of the pumping lemma. As before, $x = 0^s$ and $y = 0^t$ for integers $s \geq 0$ and $t \geq 1$.

But then, the word $xy^0z = 0^{p-t}1^p$ is not in L, which means that the first condition in the pumping lemma is not satisfied. (Convince yourself that xy^iz is in the language for all integers $i \geq 1$.) $\square$

Lecture 11: Turing machines

What is computation?

The idea of computation is ancient (especially arithmetic computation, like addition and multiplication). Also computing devices have been around for a long time. (Think about tally sticks and abaci)

If we want to reason about the limitations of computation, we need a rigorous mathematical definition of computation. Turing machines provide such a definition. Before the formal definition, we will discuss an "informal definition" of computation. Then we will see that Turing machines indeed capture this informal definition.

"Informal definition" of computation

At an informal level, we can define computation as *the process of manipulating data according to simple rules*. If we represent data as a binary string $w \in \{0,1\}^n$, then we can visualize a (discrete) process that manipulates data by the following kind of diagram:

$$w \longrightarrow w^{(1)} \longrightarrow \cdots \longrightarrow w^{(i)} \longrightarrow w^{(i+1)} \longrightarrow \cdots \longrightarrow w^{(n)}$$

To make the notion of "simple rules" more concrete, the idea of *locality* is helpful; we want that a computation step $w^{(i)} \longrightarrow w^{(i+1)}$ only changes a small part of the data. To further concretize this idea, we will add a "marker," denoted †, to indicate which part of the data is allowed to change. With this marker, the computation looks as follows:

$$\dagger w \longrightarrow u^{(1)} \dagger v^{(1)} \longrightarrow \cdots \longrightarrow u^{(i)} \dagger v^{(i)} \longrightarrow u^{(i+1)} \dagger v^{(i+1)} \longrightarrow \cdots$$

The rules of the computation specify how the data around the marker changes in one step depending on the current data around the marker.

Turing machines are a further formalization along this line of thought.

Turing machine

We can visualize a Turing machine as the union of three components:

- a *finite-state control* (akin to a deterministic finite automaton)
- a *tape*, extending infinitely in one direction
- a read / write *head*, pointing to a tape position

The computation starts with the head and input placed at the beginning of the tape. In each computation step, the control reads the symbols under the head, writes a symbol under the head, and moves the head one position to the left or to the right. The computation halts if the control accepts or rejects.

Definition: A **Turing machine** M consists of the following:

- *input alphabet* Σ (e.g., $\Sigma = \{0, 1\}$).
- *tape alphabet* Γ,
  - contains all symbols of input alphabet, $\Sigma \subseteq \Gamma$,
  - contains a designated *blank symbol* $\square \in \Gamma \setminus \Sigma$.
- *transition function* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{R}, \mathtt{L}\}$,
  - set of states Q,
  - $\mathtt{R}$ and $\mathtt{L}$ indicate left and right movement of head,
- *special states*: *start* $q_0 \in Q$, *accept* $q_{\text{accept}} \in Q$, and *reject* $q_{\text{reject}} \in Q \setminus \{q_{\text{accept}}\}$.

Configurations of Turing machines

A configuration describes the state of the whole machine (as opposed to just the control) during some step of a computation. In particular, the configuration contains the content of the tape, the position of the head, and the state of the control. We represent a **configuration** by a string C,

$$C = uqv$$

where $u \in \Gamma^*$ is the tape content before the head position, $q \in Q$ is the state of the machines, and $v \in \Gamma^+$ is the tape content after the head position, including the symbol directly under the head.

Definition: For a Turing machine M, a configuration C **yields in one step** a configuration C', denoted $C \vdash_C C'$, if one of the following rules is matched:

» Suppose $\delta(q, a) = (q', a', \mathtt{R})$ for some $q, q' \in Q$ and $a, a' \in \Gamma$. Then, for all $u, v \in \Gamma^*$,

$$uqav \vdash_\delta ua'q'v \text{ and } uqa \vdash_\delta ua'q'\square$$

» Suppose $\delta(q, a) = (q', a', \mathtt{L})$ for some $q, q' \in Q$ and $a, a' \in \Gamma$. Then, for all $u, v \in \Gamma^*$ and $b \in \Gamma$,

$$ubqav \vdash_\delta uqba'v \text{ and } qav \vdash_\delta q'a'v$$

For every configuration C, there exists exactly one configuration C' such that $C \vdash_\delta C'$ (the computation is *deterministic*).

Computation of Turing machines

Definition: We say a configuration C **yields in a finite number of steps** a configuration C', denoted $C \vdash_\delta^* C'$, if there exists a sequence of configurations $C_0, \ldots, C_n$ such that $C_i \vdash_\delta C_{i+1}$ for all $i \in \{0, \ldots, n-1\}$. A Turing machine M **accepts** on input $w \in \Sigma^*$ if $q_0 w \vdash_\delta^* C$ for some configuration C that contains the accept state. Similarly, M **rejects** on w if $q_0 w \vdash_\delta^* C$ for some configuration C that contains the reject state. Finally we say that M **halts** on input w if it either accepts or rejects on w.

Definition: The language of a Turing machine M,

$$L(M) := \{w \in \Sigma^* \mid M \text{ accepts on input } w\}.$$

Decidability

We say a language $L \subseteq \Sigma^*$ is **recognizable** (by a Turing machine) if there exists a Turing machine M such that $L(M) = L$. A language L is **decidable** if there exists a Turing machine M that accepts every $w \in L$ and rejects every $w \notin L$.

Theorem: A language L is decidable if and only if both L and $\neg L$ are recognizable.

Proof: One direction is trivial. A machine that decides L also recognizes L. If we exchange the accept and reject state of this machine, it recognizes $\neg L$.

It remains to show that if both L and $\neg L$ are recognizable then L is decidable. Let M_Y and M_N be machines with $L(M_Y) = L$ and $L(M_N) = \neg L$. Then, we can build a Turing machine M that decides L as follows:

Operation of M on input w:

- Repeat the following, starting with $i = 0$:
  - Simulate M_Y for i steps on input w. Accept if M_Y accepts in i steps.
  - Simulate M_N for i steps on input w. Reject if M_Y accepts in i steps.
  - Increment i by one.

The machine M accepts w if and only if M_Y accepts w, and M rejects w if and only if M_N accepts w. □

Example: Powers of Two

Lemma: The following language is decidable,

$$\{0^{2^n} \mid n \in \mathbb{N}_0\}.$$

Proof: We are to construct a Turing machine M that decides the above language. A general way to implement a Turing machine is to sweep over the tape content repeatedly, and on each sweep, the machine behaves like a DFA (with the difference that the machine may also write to the tape). The additional power of Turing machines over finite automata comes from the fact that they may sweep over the data multiple times.

Operation of M on input w:[1]

- Reject if w is not of the form 0^+.
- Repeat the following:
  - Accept if exactly one (uncrossed) 0 is on the tape.
  - Reject if the number (uncrossed) 0's is odd and bigger than 1.
  - Cross out every other 0 on the tape (say by replacing them with the symbol $\mathtt{X} \in \Gamma$)..comment

Each of these steps can be implemented by one sweep over the data, using a constant number of states. Hence, the above algorithm as a whole can be implemented as a Turing machine.

It remains to argue that the above algorithm decides the language $\{0^{2^n} \mid n \in \mathbb{N}_0\}$. (We need to show that the machine always halts and if it halts its answer is correct.) We may assume that w has the form 0^+. (Otherwise, the machine rejects, which is correct.) During the operation of the machine, we will maintain the invariant that there is at least one uncrossed 0 on the tape. The third step of the loop is reached only if the number of uncrossed 0's is even and at least two (by the invariant). Hence, each full iteration exactly halves the number of uncrossed 0's. Thus, if $w = 0^m$, there are exactly $m/2^i$ uncrossed 0's after i full iterations of the loop. We conclude that M accepts only if $1 = m/2^i$ for some $i \in \mathbb{N}_0$, which means m is a power of 2, and M rejects only if $m/2^i$ is an odd number bigger than 1, which means that m is not a power of 2. Furthermore, the machine halts after at most i iterations, where i is the smallest integer such that $m/2^i \leq 1$. □

---

1. As very first step, we mark the beginning of the tape, using special symbols from the tape alphabet, $\hat{}0$ and $\hat{}1$. This marker will allow us to detect in future iterations if we reached the beginning of the tape. For all other purposes, these special symbols will be treated as regular symbols, e.g., $\hat{}0$ is treated as 0 and $\hat{}1$ is treated as 1.↩

Lecture 12: Church–Turing Thesis

Church–Turing Thesis

.center **Intuitive notion of computation**
equals
Turing-machine model of computation.

The thesis is not a mathematical statement and therefore it is not possible to prove it (in the usual mathematical sense). Instead we should view the thesis as a scientific hypothesis. Nevertheless, the thesis makes interesting mathematical predictions that we can prove or disprove rigorously. (A disproof of a prediciton of the Church–Turing thesis would falsify the hypothesis.)

Computing on Configurations

We will discuss examples of other computational models (variants of Turing machines) and verify that these models are no more powerful than ordinary Turing machines. (In this way, we verify predictions of the Church–Turing thesis.)

In order to simulate another computational model by Turing machines, we will follow a simple strategy:

» Identify a notion of configuration for the competing computational model.
» Encode these configurations as strings.
» Argue that Turing machines can manipulate these strings in a way that simulates the computation of the competing model.

Multi-tape Turing machines

Suppose we augment Turing machine with multiple tapes and multiple independent heads. As before, the finite-state control determines the behavior of the heads depending on what the heads read in the current step. Formally, we have a transition function $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathtt{L}, \mathtt{R}\}^k$ to control k independent heads.

Theorem: Single-tape Turing machines can simulate multi-tape Turing machines.

Proof Sketch: For simplicity we consider the case $k = 2$. We can encode the configuration of a 2-tape machine M as a string C of the form

$$C = u_1 q a_1 v_1 \# u_2 q a_2 v_2.$$

Here, $q \in Q$ is the state of the control, $u_i \in \Gamma^*$ is the content of tape i before the head, $v_i \in \Gamma^*$ is the content of tape i after the head, and $a_i \in \Gamma$ is the symbol that head i is reading in the current step. (The hash symbol $\# \notin \Gamma$ delimits the contents of the two tapes.)

The following single-tape Turing machine M' simulates the computation of the 2-tape machine M. For simplicity, we can choose the tape alphabet of M' such that it contains all elements of $\Gamma \cup Q \cup \{\#\}$ (but we could also choose to encode everything as binary strings).

Operation of M' on input w:

- Write the start configuration $C_0 = q_0 w \# q_0 \square$ of M on the tape of M'.
- Repeat the following steps:
  - Read the configuration C of M on the tape of M'.
  - Accept if C is an accept-configuration for M and reject if C is a reject-configuration for M.
  - Manipulate the tape content such that it becomes the subsequent configuration C' of M.

Each of the steps can be implemented by a constant number of passes over the tape. After i full iterations of the loop, the tape of M' contains the configuration of M after i computaton steps. Therefore, M' accepts or rejects on input w if and only if M accepts or rejects on input w. $\square$

Non-deterministic Turing machines

Suppose we allow non-deterministic transitions in the finite-state control of a Turing machine M. Formally, we have a transition function δ such that $\delta(q, a) \subseteq Q \times \Gamma \times \{\mathtt{L}, \mathtt{R}\}$ describes the possible transitions if the control is in state q and the head reads a. Then, a configuration C of M can yield multiple configurations C' in one step, denoted $C \vdash_\delta C'$.

We say that a non-deterministic Turing machine **accepts** on input w if there *exists* a sequence of configurations $C_0, \ldots, C_m$ such that C_0 is the start configuration, $C_i \vdash_\delta C_{i+1}$ for all $i \in \{0, \ldots m-1\}$, and C_m is an accept-configuration. The language $L(M)$ of a non-deterministic Turing machine M

is the set of strings $w \in \Sigma^*$ such that M accepts on input w. (Unlike for deterministic machines, we don't define what it means for a non-deterministic machine to reject.)

> **Theorem:** A language L is recognizable by non-deterministic Turing machines if and only if it is recognizable by deterministic Turing machines.

Proof: Let M be a non-deterministic Turing machine. We are to construct a deterministic Turing machine M' with the same language $L(M') = L(M)$. The idea is that M' enumerates all possible computation paths of M.

Operation of M' on input w:

- Enumerate all binary strings x (e.g., in lexicographic order) and perform the following operations:
  - Check that x encodes a sequence of configuration $C_0, \ldots, C_m$.
  - Check that $C_0 = q_o w$ is the start configuration.
  - Check that $C_i \vdash_\delta C_{i+1}$ for all $i \in 0, \ldots, m-1$.
  - Check that C_m is an accept configuration.
  - Accept if all checks are passed.

The machine M' can perfom each step of the loop by doing a finite number of passes over the tape content. If M' accepts, then there exists an accepting computation path for M and therefore M accepts. On the other hand, if M accepts, then there exists an accepting computation path and an encoding of the path as a binary string x. The machine M' will accept when the enumeration reaches the string x. □

Computing on Turing Machines

Going the beyond idea of encoding configurations as strings and computing on them, we can encode whole machines as strings and compute on them. For a Turing machine M, we write

$$\langle M \rangle$$

to denote an encoding of it as a binary string. We assume that the formatting of the encoding is so that it is easy for another Turing machine to manipulate it. For convenience, we also assume that every binary string encodes some Turing

machine. (For example, we can say if a binary string doesn't respect the usual formatting, then it decodes to a dummy machine M_0.)

Another property of such encodings that is sometimes useful is that for every machine M, there are infinitely many binary strings that decode to M or to a machine that behaves exactly in the same way as M. We can assume this property because by adding unreachable dummy states to M, we can get infinitely many machines that behave exactly like M.

> In the following, we will use the notation $\langle \cdot \rangle$ to encode arbitrary mathematical objects as binary strings.

Universal Turing machines

We will show that there exists a Turing machine U that can simulate arbitrary Turing machines. Here, it is interesting that the number of states of U and its tape alphabet are fixed, but it can still simulate machines with many more states and many more tape alphabet symbols. A Turing machine with this property is called universal. (In contrast, the machines in our previous constructions were allowed to depend on the machine they are simulating.)

The fact that modern computers are multi-purpose devices and can, in principle, execute arbitrary programs stems from this formal idea of universality.

> **Theorem:** There exist a Turing machine U that on input $\langle M, w \rangle$ simulates the operation of the Turing machine M on input $w \in \{0,1\}^*$.

Proof: It's convenient to first build a two-tape machine U_0 that achieves this task. Then, we can simulate U_0 by a single-tape machine U using our previous construction for multi-tape machines.

Operation of U_0 on input $\langle M, w \rangle$.

- Write M's start configuration $C_0 = q_0 w$ as binary string C_0 on tape 1.
- Write M's description as binary string $\langle M \rangle$ on tape 2.
- Repeat the following steps:
  - *Invariant:* tape 1 holds the binary encoding of a configuration C of M.
  - Locate on tape 1 the state of M (encoded as binary string).regular.grey.
  - Locate on tape 1 the tape symbol under M's head (encoded as binary string).comment.
  - Look up the corresponding part in the transition table of M on tape 2.
  - Accept or reject if the current state is the accept or reject of M.
  - Update the content of tape 1 according to this transition.

It's important to note the finite control of U_0 cannot "remember" the current state of M or the tape symbol that M is currently reading. Nevertheless, U_0 can find the positions of these items on the tapes. (We can implement this process by moving markers across the tape until we find a match.)

After i full iterations of the loop, tape 1 holds the configuration of M after i steps on input w. Therefore, the machine U_0 on $\langle M, w \rangle$ accepts or rejects if and only if M on w accepts or rejects. □

Acceptance problem for Turing machines

> **Theorem:** The acceptance problem A_{TM} for Turing machines is recognizable,
>
> $$A_{\mathrm{TM}} = \{\langle M, w \rangle \mid M \text{ accepts on input } w\}.$$

Proof: The universal Turing machine U from the previous theorem recognizes this language, because U accepts on input $\langle M, w \rangle$ if and only if M accepts on input w. □

Lecture 13: Undecidable Languages

Diagonalization

Theorem: Every set S has cardinality strictly smaller than its powerset 2^S.

Proof sketch: The theorem is especially interesting for infinite sets. For concreteness, we prove it only for finite sets, but our proof would also work for infinite sets. (Exercise.) Suppose $S = \{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$. We can encode every subset $A \subseteq S$ as an n-bit string $x_A \in \{0,1\}^n$.

Let $x_1, \ldots, x_n \in \{0,1\}^n$ be a list of n binary strings of length n. We are to show that there exists an n-bit binary string $x \in \{0,1\}^n \setminus \{x_1, \ldots, x_n\}$ not contained in the list. Let us arrange the strings as an $n \times n$ matrix,

	1	2	$\cdots$	n
x_1	$x_{1,1}$	$x_{1,2}$	$\cdots$	$x_{1,n}$
x_2	$x_{2,1}$	$x_{2,2}$	$\cdots$	$x_{2,n}$
$\vdots$	$\vdots$		$\ddots$	$\vdots$
x_n	$x_{n,1}$	$x_{n,2}$	$\cdots$	$x_{n,n}$

An explicit n-bit string not in this list is the string $\bar{x}_{1,1}\bar{x}_{2,2}\cdots\bar{x}_{n,n}$. The reason is that this string differs from x_i in (at least) the i-th coordinate. It follows that the number n-bit strings is larger than n. □

Existence of undecidable languages

We can use the previous theorem as a black-box to show that some languages are **undecidable**.

Theorem: Not every language is decidable.

Proof: The set of all Turing machines has the same cardinality as the set of binary strings $\{0,1\}^*$. On the other hand, the set of all languages is the power set of the set of binary strings. Therefore, the set of all languages has strictly

larger cardinality than the set of Turing machines (by the previous theorem). Hence, there exists a language that is not decided by any Turing machine. (A Turing machine decides at most one language.) □

Explicit undecidable language

The previous theorem shows that undecidable languages exist but it doesn't give any hint how such languages might look like. The following theorem shows an explicit language that is undecidable.

Theorem: The following language is undecidable,

$$L_{\text{diag}} = \{\langle M \rangle \mid M \text{ on input } \langle M \rangle \text{ rejects}\}.$$

Proof: Every machine M fails to decide the language L on input $w = \langle M \rangle$. If $w \in L_{\text{diag}}$, then M rejects w. On the other hand, if $w \notin L_{\text{diag}}$, then M does not reject w. In either case, M fails to decide L_{diag} on input w. □

Remark: In lecture 3, we have seen a non-constructive proof that some n-bit functions require large circuits, concretely size $\Omega(2^n/n)$. The proof used the same kind of counting argument that showed the existence of undecidable languages in this lecture. Here, we managed to use diagonalization to come up with an explicit undecidable language. *Exercise:* Does the same kind of diagonalization give us an explicit function that requires larger circuits? Why not?

Lecture 14: Reductions

Decidability and Reductions

> **Definition:** A function $f{:}\Sigma^* \to \Sigma^*$ is a **reduction** from language A to language B if $w \in A \Leftrightarrow f(w) \in B$ for every $w \in \Sigma^*$.

We say a function $f{:}\Sigma^* \to \Sigma^*$ is **computable** if there exists a Turing machine that for every input $w \in \Sigma^*$ halts with just $f(w)$ written on its tape.

We say A *reduces* to B, denoted $A \leq_m B$, if there exists a computable reduction from A to B.

> **Theorem:** If $A \leq_m B$ and A is undecidable, then so is B.

Proof: We prove the contrapostive. If B is decidable, then A is decidable. Suppose M_B decides B and M_f computes f. Then, the following machine M decides A:

> Operation of M on input w:
> - Simulate M_f on w to compute $f(w)$.
> - Simulate M_B on $f(w)$.
> - Accept if M_B accepts and reject if M_B rejects.

Since $f(w) \in B$ if and only if $w \in A$, the machine M decides A.

Example 1: Acceptance of Turing Machines

In lecture 13, we showed that the diagonal language L_{diag} for Turing machines is undecidable

$$L_{\text{diag}} = \{\langle M \rangle \mid M \text{ on input } \langle M \rangle \text{ rejects}\}.$$

In lecture 12, we introduced the acceptance problem A_{TM} for Turing machines and showed that this language is recognizable (by a universal Turing machine).

Lemma: The acceptance problem for Turing machines is undecidable,

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ on input } w \text{ accepts}\}.$$

Proof: We will show that $L_{\text{diag}} \leq_m A_{\text{TM}}$. Consider the computable function that maps M to $\langle M', w\rangle$, where $w = \langle M\rangle$ and M' is a machine that accepts if and only if M rejects. This function is a reduction from L_{diag} to A_{TM}. Therefore, $L_{\text{diag}} \leq_m A_{\text{TM}}$, which shows that A_{TM} is undecidable.

Example 2: Non-emptiness

Lemma: The non-emptiness property for Turing machines is undecidable,

$$\text{NONEMPTY} = \{\langle M\rangle \mid L(M) \neq \emptyset\}.$$

Proof: We will show $A_{\text{TM}} \leq_m \text{NONEMPTY}$. Consider the computable function f that maps $\langle M, w\rangle$ to $\langle M_w\rangle$, where M_w is the following machine:

Operation of M_w on input u:
- If $u = w$, simulate M on input w and accept if M accepts.
- If $u \neq w$, reject.

The language of M_w is non-empty if and only if M accepts w. Therefore, f is a reduction from A_{TM} to NONEMPTY, which shows that NONEMPTY is undecidable.

Example 3: Halting Problem

Lemma: The halting problem for Turing machines is undecidable,

$$\text{HALT} = \{\langle M, w\rangle \mid M \text{ halts on input } w\}.$$

Proof: We will show $A_{\text{TM}} \leq_m \text{HALT}$. Consider the computable function f that maps $\langle M, w\rangle$ to $\langle M', w\rangle$, where M' is the following machine:

Operation of M' on input u:

- Simulate M on u.
- Accept if M accepts (but do not reject if M rejects).

The machine M' halts on some input u if and only if M accepts u. Therefore, $\langle M, w \rangle \in A_{\text{TM}}$ if and only if $\langle M', w \rangle \in \text{HALT}$, which means that the function f is a reduction from A_{TM} to HALT.

Lecture 15: General undecidability criterion

Rice's Theorem

We will show that every non-trivial property of languages of Turing machines is undecidable *(Rice's theorem)*. To show this theorem, we will formalize what properties of languages are and what it means for them to be non-trivial for Turing machines.

Let $2^{\{0,1\}^*}$ be the set of all languages over the binary alphabet. A **property** of languages is a subset $P \subseteq 2^{\{0,1\}^*}$. We say L *satisfies* P if $L \in P$ and we say that L *violates* P if $L \notin P$. For example, we can choose P to be the set of languages that contain the string 0000.

We say that P is a **non-trivial** property of languages of Turing machines if there exists Turing machines M_Y and M_N such that $L(M_Y)$ satisfies the property and $L(M_N)$ violates the property. Here is an example of a property that is non-trivial for languages of Turing machines,

$$P_{\text{empty}} = \{\emptyset\}.$$

The property is non-trivial because there are Turing machines that recognize the empty language and there are Turing machines that recognize a non-empty language. Here is an examples of a property that is trivial for languages of Turing machines,

$$P_{\text{recognizable}} = \{L \mid L \text{ is recognizable}\}.$$

> **Theorem:** Let P be any non-trivial property of languages of Turing machines. Then, the following language is undecidable,
>
> $$L_P = \{\langle M \rangle \mid \ L(M) \text{ satisfies } P\}.$$

Proof: We distinguish two cases, $\emptyset \in P$ and $\emptyset \notin P$.

Case $\emptyset \notin P$: In this case, we will show that $A_{\text{TM}} \leq_m L_P$, which implies that L_P is undecidable. (See the notes for lecture 14 for the proof that the acceptance problem A_{TM} is undecidable.) Let M_Y be a Turing machine such

that $L(M_Y) \in P$ satisfies P. (We know it exists because P is a non-trivial property of languages of Turing machines.) Consider the computable function f that maps $\langle M, w \rangle$ to $\langle M'_w \rangle$, where M'_w is the following Turing machine:

> Operation of M'_w on input u:
> - Simulate M on input w.
> - If M accepts, simulate M_Y on input u and accept if M_Y accepts.

By construction, the machine M'_w accepts on input u if and only if M accepts w and M_Y accepts u. Therefore, $L(M'_w) = L(M_Y)$ if M accepts on w and $L(M'_w) = \emptyset$ if M does not accept w. Since $L(M_Y) \in P$ and $\emptyset \notin P$, we see that $L(M'_w)$ satisfies P if and only if M accepts w. Thus, f is a reduction from A_{TM} to L_P.

Case $\emptyset \in P$: We will reduce this case to the previous case. The complement property $\neg P$ is also a non-trivial property of languages of Turing machines and it satisfies $\emptyset \notin \neg P$. The proof for the previous case applies and we get that $L_{\neg P}$ is undecidable. Since $L_P = \neg L_{\neg P}$, we can conclude that L_P is undecidable (undecidability is closed under complementation). $\square$