Models of Computation

CS 365

Eric Blais

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 365 during Winter 2021 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

The original version of notes is available at https://cs.uwaterloo.ca/~eblais/cs365/. It was made with Book Theme: https://github.com/alex-shpak/hugo-book. This set of notes is more like a latex type-up of the original web version.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Languages

1.1 Alphabets, Strings, and Languages

All areas of mathematics are concerned with the study of the properties of specific types of mathematical objects: integers for number theory, graphs for graph theory, vector spaces for linear algebra, etc.

So what is the mathematical object that we study in theoretical computer science? It might be tempting to answer "computers", or "algorithms", or "computational problems", but none of these answers is really accurate, simply because these concepts are not precise enough to enable rigorous mathematical analysis.

Instead, the main objects of study in theoretical computer science are much simpler: alphabets, strings, and languages. We begin by introducing these three (deceptively simple-looking) concepts.

alphabet

An **alphabet** is a nonempty finite set.

The elements of an alphabet are called **symbol**s.

string

A **string** over alphabet Σ is a finite sequence of symbols from Σ.

The **length** of the string x, denoted $|x|$, is the number of symbols in x.

The **empty string** is the unique string of length 0; we denote it by ε.

An **encoding** is a mapping from the objects to the set of strings such that no two objects are mapped to the same string.

For any non-empty string x of length $|x| = n$, we write $x = (x_1, \ldots, x_n)$ so that x_I is the ith symbol in x for any $i = 1, \ldots, n$.

language

A **language** over alphabet Σ is a set of strings over Σ.

A language is **finite** if it contains a finite number of strings. The **cardinality of a finite language** L is the number of strings that it contains and is denoted $|L|$.

The **empty language** $\varnothing$ is the unique language that contains no strings.

The language that contains every string over Σ is denoted Σ^*.

The language that contains the strings of length exactly n over Σ is denoted Σ^n.

1.2 String Operations

concatenation

The **concatenation** of the strings $x, y \in \Sigma^*$ of lengths $|x| = k$ and $|y| = \ell$ is the string

$$xy = (x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_\ell).$$

square

The **square** of $x \in \Sigma^*$ is the string $x^2 = xx$.

More generally, for any $n \geq 1$ the nth power of x is string x^n obtained by concatenating n copies of x.

The 0th power of x is $x^0 = \varepsilon$.

primitive string

A string $x \in \Sigma^*$ is **primitive** if $x \neq y^n$ for any $y \in \Sigma^*$ and $n \geq 2$.

For any strings $x, y \in \Sigma^*$:

prefix

x is a **prefix** of y if there exists $z \in \Sigma^*$ such that $y = xz$.

suffix

x is a **suffix** of y if there exists $z \in \Sigma^*$ such that $y = zx$.

substring

x is a **substring** of y if there exists $w, z \in \Sigma^*$ such that $y = wxz$.

subsequence

The string x is a **subsequence** of the string y if there exist indices $j_1 < j_2 < \cdots < j_{|x|}$ such that $x_i = y_{j_i}$ for each $i = 1, 2, \ldots, |x|$.

reversal

The **reversal** of a string x of length $|x| = n$ is the string

$$x^R = (x_n, x_{n-1}, \ldots, x_2, x_1).$$

The string x is a **palindrome** if $x = x^R$.

1.3 Language Operations

Given two languages A and B over Σ,

- (Union) $A \cup B = \{x \in \Sigma^* : x \in A \text{ or } x \in B\}$;
- (Intersection) $(A \cap B) = \{x \in \Sigma^* : x \in A \text{ and } x \in B\}$;
- (Set difference) $A \setminus B = \{x \in \Sigma^* : x \in A \text{ and } x \notin B\}$;
- (Symmetric difference) $A \triangle B = (A \setminus B) \cup (B \setminus A)$; and
- (Complement) $\overline{A} = \Sigma^* \setminus A$.

concatenation

The **concatenation** of two languages A and B over Σ is the language

$$AB = \{xy : x \in A \text{ and } y \in B\}$$

obtained by concatenating every string in A with every string in B.

square

The **square** of a language L over Σ is the language $L^2 = LL$ obtained by concatenating L with itself.

More generally, for any language $n \geq 1$, the nth power of L is the language

$$L^n = \{x^{(1)}x^{(2)} \cdots x^{(n)} : x^{(1)}, x^{(2)}, \ldots, x^{(n)} \in L\}$$

obtained by concatenating n copies of L.

The 0th power of L is defined to be $L^0 = \{\varepsilon\}$.

start operation

For any language L, the **start operation** on L yields the language

$$L^* = \bigcup_{n \geq 0} L^n.$$

The star operation is also called the Kleene star operation, after Stephen Kleene who introduced it.

1.4 Countability and Uncountability

A really important notion in the study of the theory of computation is the uncountability of some infinite sets, along with the related argument technique known as the diagonalization method.

$|S| \leq |T|$

The cardinalities of two sets S and T, denoted $|S|$ and $|T|$, satisfy the inequality $|S| \leq |T|$ if and only if there is a one-to-one mapping from the elements of S to those of T.

same cardinality

Two sets S and T have the **same cardinality**, denoted $|S| = |T|$ if and only if $|S| \leq |T|$ and $|T| \leq |S|$.

The set if natural numbers is $\mathbb{N} = \{1, 2, 3, \ldots\}$. Note that in CS 360, 0 is included.

finite nonempty set

A nonempty set S is **finite** if $|S| = |\{1, 2, \ldots, n\}|$ for some natural number $n \in \mathbb{N}$. When this is the case, we write $|S| = n$ and say that S has cardinality $|S| = n$.

The empty set has cardinality $|\varnothing| = 0$ and is also finite.

The cardinality of set $\mathbb{N}$ of natural numbers is denoted $|\mathbb{N}| = \aleph_0$.

countable set

The set S is **countable** if and only if $|S| \leq |\mathbb{N}| = \aleph_0$.

Proposition 1.1

For any alphabet Σ, the set Σ^* of strings over Σ is countable.

A set is **uncountable** if it is not countable.

power set

For any set S, the **power set** $\mathcal{P}(S)$ is the set of all subsets of S.

Theorem 1.2: Cantor's Theorem

For any infinite countable set S, the power set $\mathcal{P}(S)$ is uncountable.

Corollary 1.3

For any alphabet Σ, the set of languages over Σ is uncountable.

2

Deterministic Finite Automata

deterministic finite automation

A **deterministic finite automation** (or DFA) is an abstract machine described by

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

The **size of the DFA** $M = (Q, \Sigma, \delta, s, F)$ is the number of states in Q.

DFA acceptance

The DFA $M = (Q, \Sigma, \delta, s, F)$ accepts the string $w \in \Sigma^n$ if and only if there is a sequence of states $r_0, r_1, \ldots, r_n \in Q$ where

- $r_0 = q_0$,
- $r_i = \delta(r_{i-1}, w_i)$ for each $i = 1, 2, \ldots, n$, and
- $r_n \in F$.

A string is rejected by a DFA if it is not accepted by that DFA.

language recognized by DFA

The **language recognized** by a DFA M is

$$L(M) = \{x \in \Sigma^* : M \text{ accepts } x\}$$

Note that in CS 360, this is written as $\mathrm{L}(M)$.

regular language

The language A is **regular** if there is a DFA M such that $A = L(M)$.

Proposition 2.1

Every finite language is regular.

Proposition 2.2

There exist languages that are not regular.

2.1 Nondeterministic Finite Automata

nondeterministic finite automaton

A **nondeterministic finite automaton** (or NFA) is an abstract machine described by

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is the transition relation,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

An **ϵ-transition** in the NFA M is a transition of the form $\delta(q, \epsilon)$ for some $q \in Q$. The NFA M can choose to follow this transition from q at any time, without processing any symbol from its input.

ϵ-reachable

A state $r \in Q$ is **ϵ-reachable** from a state $q \in Q$ in an NFA M if there are states $s_0, s_1, \ldots, s_\ell$ for some $\ell \geq 0$ that satisfy $s_0 = q$, $s_\ell = r$, and $s_i \in \delta(s_{i-1}, \epsilon)$ for each $i = 1, 2, \ldots, \ell$. The state $r \in Q$ is ϵ-reachable from a set $S \subseteq Q$ of states if it is reachable from some state $q \in S$.

language recognized by NFA

The **language recognized** by the NFA N is

$$L(N) = \{x \in \Sigma^* : N \text{ accepts } x\}.$$

Proposition 2.3

Every regular language can be recognized by an NFA.

2.2 Rabin-Scott Theorem

Theorem 2.4: Rabin-Scott Theorem

The set of languages that can be recognized by DFAs is exactly the same as the set of languages that can be recognized by NFAs.

Lemma 2.5

For every NFA N, there is a DFA MM that recognizes $L(N)$.

2.3 Regular Expressions

regular expression

A **regular expression** over the alphabet Σ is a nonempty string r over $\Sigma \cup \{|, ^*, (,), \epsilon, \varnothing\}$ that satisfies one of the following conditions:

- $r = \varnothing$,
- $r = \epsilon$,
- $r = \sigma$ for some $\sigma \in \Sigma$,
- $r = (s \mid t)$ for some regular expressions s, t,
- $r = (st)$ for some regular expressions s, t, or
- $r = (s^*)$ for some regular expression s.

length

The **length** of a regular expression r over Σ is the number of symbols from Σ in the string r.

2.4 Kleene's Theorem

Theorem 2.6: Kleene's Theorem

The language A is regular if and only if it can be described by a regular expression.

We can prove the theorem via two lemmas.

Lemma 2.7

For every regular expression r, the language $L(r)$ described by r can be recognized by an NFA.

Lemma 2.8

Every language L that can be recognized by a DFA can be described with a regular expression.

2.5 The Pumping Lemma

Lemma 2.9: pumping lemma for regular languages

For every regular language L, there is a number p such that for any string $s \in L$ of length at least p, we can write $s = xyz$ where

- $|y| > 0$,
- $|xy| \leq p$, and
- For each $i \geq 0$, $xy^iz \in L$.

A language L that satisfies the condition of the lemma is said to satisfy the *pumping property*.

A value p for which L satisfies the pumping property is known as a *pumping length* of L.

The Pumping Lemma is useful for showing that some languages are non-regular because it can be restated equivalently in the following contrapositive form:

Lemma 2.10: Contrapositive Form of the Pumping Lemma

If for every positive integer p, there exists a string $s \in L$ of length at least p such that for every decomposition $s = xyz$ with $|y| > 0$ and $|xy| \leq p$, there exists a value $i \geq 0$ for which $xy^iz \notin L$, then L is not a regular language.

2.6 Non-Regular Languages

$L = \{0^n1^n :\geq 0\}$ is not regular.

$L_<\{0^m1^n : n > m \geq 0\}$ is not regular.

$L_>\{0^m1^n : m > n \geq 0\}$ is not regular.

$L = \{0^{2^n} : n \geq 0\}$ is not regular.

One important warning about the Pumping Lemma is that the converse of that lemma is not true: there are languages that are not regular but still satisfy the pumping property. The following language gives one such example.

Proposition 2.11

The language $L_{\text{pal}^+} = \{ww^Rx \in \Sigma_2^* : |w| \geq 1\}$ satisfies the pumping property but it is not regular.

Another way to state the limitation of the pumping lemma is that it gives a necessary property that must hold for a language AA to be regular (since all regular languages satisfy the pumping property), but that this property does not characterize the class of regular languages since it is not a sufficient property to guarantee that a language is regular.

So does there exist a (natural) property of languages that characterizes regular languages? The answer is yes. This result is known as the Myhill-Nerode Theorem and is one of the topics covered in CS 462.

2.7 Properties of Regular Languages

We have already seen that the class of regular languages is closed under the union operation: given two regular languages A and B, their union $A \cup B$ is also a regular language. It is also closed under the complement, concatenation, and star operations.

language expansion

The **expansion** of the language A over Σ is the language

$$A^{\uparrow} = \{x \in \Sigma^* : \exists y \in A, |y| = |x|, \mathrm{d}(x, y) \leq 1\}.$$

The class of regular languages is closed under language expansion.

Topological Separation

It is interesting to ask about the regularity of languages obtained by taking subsets of other languages. Since every finite language is regular, all languages have a subset that is a regular language. Can we say more? The most natural possible strengthening of this observation is to ask whether every infinite language contains a subset that is an infinite regular language. The answer to this question is no.

There exists a language L over Σ_2 such that every infinite language $L' \subseteq L$ is non-regular.

What about when L is regular? It's easy to show that in this case it always contains a regular language as a strict subset.

For every infinite regular language L, there is an infinite language $L' \subsetneq L$ that is regular.

If you end up with the same proof of the last proposition that I obtained, you might find it rather unsatisfying. If so, the next natural question to ask is whether every infinite regular language L can be partitioned into two languages $L' \subseteq L$ and $L \setminus L'$ that are both infinite regular languages. This question is a type of topological separation problem, and the answer is yes.

For every infinite regular language L, there is a regular language $L' \subseteq L$ such that L' and $L \setminus L'$ are both infinite regular languages.

3

Context-Free Languages

3.1 Pushdown Automata

Looking back on the specific languages examined in that note, we can identify what appears to be the main limitation of finite automata causing this issue: to recognize languages like $\{0^n 1^n : n \geq 0\}$, a finite automaton would needs access to some memory.

The pushdown automaton model the model of computation when we give memory to finite automata in the form of a stack. This model of computation can indeed recognize a richer class of languages. And it also possesses very interesting structural properties of its own.

A pushdown automaton is a nondeterministic finite automaton that is augmented by having access to a stack. Pushdown automata can be described using transition diagrams.

When simulating a PDA on some input string, you want to keep track of three things:

- The symbol of the input string the automaton is currently examining,
- The current state of the automaton, and
- The string currently in the stack.

Since a PDA is nondeterministic, there will usually be multiple valid possible computational paths for any given input; as with NFAs, what we want to determine is whether any path ends in an accepting state.

We use the notation

$$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$$

to denote the alphabet obtained by adding the ϵ symbol to Σ.

pushdown automaton

A **(nondeterministic) pushdown automaton** (PDA) is an abstract machine defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ is the transition relation,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

A deterministic pushdown automaton can also be defined by adding a stack to DFAs, so technically we should call the object we will be studying in these notes "nondeterministic PDAs" throughout. But since we will only examine nondeterministic PDAs, we will just call them "PDAs" throughout.

PDA acceptance

The pushdown automaton M **accepts** the string $w \in \Sigma^*$ if and only if there exist a parameter $m \geq |w|$, symbols $y_1, \ldots, y_m \in \Sigma_\epsilon$, states $r_0, \ldots, r_m \in Q$, and strings $s_0, \ldots, s_m \in \Gamma^*$ such that

1. $w = y_1 \cdots y_m$,
2. $r_0 = q_0$ and $s_0 = \epsilon$,
3. $r_m \in F$, and
4. for each $i = 1, \ldots, m$, there exist $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$ such that
   - $(r_i, b) \in \delta(r_{i-1}, y_i, a)$,
   - $s_{i-1} = a\,t$, and
   - $s_i = b\,t$.

language of PDA

The **language** of a pushdown automata M is

$$L(M) = \{w \in \Sigma^* : M \text{ accepts } w\},$$

the set of strings accepted by M. We say that M recognizes $L(M)$.

Proposition 3.1

Every regular language can be recognized by a pushdown automaton.

We can also prove that PDAs are strictly more powerful than NFAs: we can show that there are non-regular languages that can be recognized by PDAs: there is a PDA that recognizes the language

$L = \{0^n 1^n : n \geq 0\}$.

3.2 Context-Free Grammars

A context-free grammar (or CFG) is a set of rules that can be applied to (individual) variables to generate strings of variables and symbols, as in this example:

$$\begin{aligned} S &\to AB \\ A &\to 0A \\ A &\to 0 \\ B &\to 1B \\ B &\to 1 \end{aligned}$$

We also use the short-hand notation to write multiple rules with the same initial variable on the same line, so that the rules above can also be expressed as

$$\begin{aligned} S &\to AB \\ A &\to 0A \mid 0 \\ B &\to 1B \mid 1 \end{aligned}$$

To generate a string from a grammar, we start with a special start variable (S in this case) and apply rules to one of the variables in the current string until all the symbols that remain are symbols. A string is generated from a CFG if it can be obtained in this way.

context-free grammar

A **context-free grammar** (or CFG) is a grammar

$$G = (V, \Sigma, R, S)$$

where

- V is a finite set of variables (or non-terminal symbols),
- Σ is a finite set of terminals ,
- R is a finite set of rules mapping V to $(V \cup \Sigma)^*$, and
- S is the start variable in V.

yield

The string uAv **yields** the string uwv in the context-free grammar $G = (V, \Sigma, R, S)$, denoted

$$uAv \Rightarrow uwv,$$

when $A \in V$ is a variable, $u, v, w \in (\Sigma \cup V)^*$ are some strings, and the rule $A \to w$ is in R.

derive

The string u **derives** the string v in G, denoted

$$u \overset{*}{\Rightarrow} v,$$

when $u = v$, $u \Rightarrow v$, or there is a sequence of $k \geq 1$ strings $u_1, \ldots, u_k \in (V \cup \Sigma)^*$ such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

language generated by CFG

The **language generated** by the context-free grammar $G = (V, \Sigma, R, S)$ is

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\},$$

the set of strings of terminal symbols that can be derived in G from the initial string containing only the start variable S.

The class of languages that can be described by context-free grammars is our main object of interest in this section. We call it the class of context-free languages.

context-free language

A language is **context-free** if it can be generated by a context-free grammar.

3.3 Equivalence of CFGs and PDAs

Theorem 3.2

A language can be generated by a context-free grammar if and only if it can be recognized by a pushdown automaton.

We prove the two directions of the "if and only if" statement separately.

Theorem 3.3

For every context-free grammar G, the language $L(G)$ can be recognized by a pushdown automaton.

Lemma 3.4

For every pushdown automaton M, there is a pushdown automaton $M' = (Q, \Sigma, \Gamma, \delta, q_0, F)$ which satisfies

1. F consists of a single accept state;
2. M' always empties the stack before accepting; and
3. Each transition $a; b \to c$ in δ has exactly one of b, c equal to ϵ, so that each transition either pushes a symbol onto the stack or pops a symbol from the stack.

and also satisfies $L(M) = L(M')$.

Using the lemma, we can establish the second direction of our main theorem.

Theorem 3.5

For every pushdown automaton M, the language $L(M)$ can be described by a context-free grammar.

3.4 Chomsky Normal Form

Chomsky normal form

The context-free grammar $G = (V, \Sigma, R, S)$ is in **Chomsky normal form** (or CNF) if every production rule is of the form

- $A \to BC$,
- $A \to a$, or
- $S \to \epsilon$

for some variable $A \in V$, some terminal symbol $a \in \Sigma$, and some variables $B, C \in V \setminus \{S\}$.

Theorem 3.6

For every context-free grammar G, there is a context-free grammar G' in Chomsky normal form such that $L(G) = L(G')$.

3.5 Pumping Lemma for CFLs

Proposition 3.7

If $G = (V, \Sigma, R, S)$ is a context-free grammar in Chomsky normal form with $|V| = m$ variables and G generates a string x of length $|x| \geq 2^m$, then $L(G)$ is an infinite language.

Lemma 3.8: Pumping lemma for context-free languages

For every context-free language $L \subseteq \Sigma^*$, there is a number p such that for every string $s \in L$ of length $|s| \geq p$, we can write $s = uvxyz$ where

- $|v| + |y| > 0$,
- $|vxy| \leq p$, and
- For each $i \geq 0$, $uv^i xy^i z \in L$.

We can use the pumping lemma to prove the following languages are not context-free.

$$L = \{0^n 1^n 2^n : n \geq 0\}$$
$$L = \{ww : w \in \Sigma_2^*\}$$
$$L = \{0^\ell 1^m 2^n : 0 \leq \ell \leq m \leq n\}$$
$$L = \{wxw^R : w, x \in \Sigma^*, |w| = |x|\}$$
$$L = \{x \in \Sigma_2^* : x \text{ contains two 1s in the middle third }\}$$
$$L = \{x\#w : x \text{ is a substring of } w\}$$
$$L = \{x \in \Sigma_2^* : x = x^R \text{ and } x \text{ has the same number of 0s and 1s }\}$$
$$L = \{0^k 1^n : n \text{ is a multiple of } k\}$$

3.6 Properties of Context-Free Languages

Proposition 3.9

Context-free languages are closed under the following operations: Union, Concatenation, Star operation, and Reversal.

Proposition 3.10

Context-free languages are not closed under the following operations: Intersection and Complementation.

Another way that we can explore the differences between regular languages and context-free languages is to consider the class of languages that can be recognized by deterministic pushdown automata. We call this class of languages deterministic context-free languages (DCFL).

The class of DCFLs is not the same as that of CFLs. In fact, they also do not share all of the same structural properties. For instance, like the class of regular languages, the class of DCFLs is closed under complementation.

Theorem 3.11

The classes of deterministic and (standard) context-free languages satisfy strict inclusion

$$\text{DCFL} \subsetneq \text{CFL}.$$

4

Computability

4.1 Turing Machines

A Turing machine is a finite automaton that has access to an infinite tape, divided into individual cells that each store a single symbol.

A Turing machine only sees the content of the single cell at the current tape head position, and it can only move the tape head position one cell to the right or to the left in each transition. Yet, as we will see in this part, even with such strong restrictions, Turing machines are just as powerful as any other reasonable model of computation.

There is a special symbol that denotes blank cells which have no content in them: we represent this symbol with a blank square □. [1]

deterministic Turing machine

A **deterministic Turing machine** (DTM) is an abstract machine defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the tape alphabet,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the initial state,
- $q_{acc} \in Q$ is the accept state, and
- $q_{rej} \in Q \setminus \{q_{acc}\}$ is the reject state.

[1] Note that in CS 360, the blank symbol is ␣ which is better.

configuration

A **configuration** of a Turing machine T is a string wqy with $w, y \in \Gamma^*$ and $q \in Q$ where

- q is the current state of the automaton,
- wy is the current string on the tape,
- the position of the tape head is on the first symbol in y.

The initial configuration on input x is q_0x.

The configuration wqy is:

- an *accepting configuration* if $q = q_{acc}$.
- a *rejecting configuration* if $q = q_{rej}$.
- a *halting configuration* if it is an accepting or a rejecting configuration.

yield

For any strings $w, y \in \Gamma^*$, symbols $a, b, c \in \Gamma$, and states $q, q' \in Q$, the configuration $waqby$ in M **yields** the configuration $wq'acy$, written

$$waqby \vdash wq'acy$$

when $\delta(q, b) = (q', c, L)$. Similarly,

$$waqby \vdash wacq'y$$

when $\delta(q, b) = (q', c, R)$.

derive

The configuration wqy **derives** the configuration $w'q'y'$ in M, written

$$wqy \vdash^* w'q'y'$$

if there is a finite sequence of configurations $w_1q_1y_1, \ldots, w_kq_ky_k$ such that

$$wqy \vdash w_1q_1y_1 \vdash \cdots \vdash w_kq_ky_k \vdash w'q'y'.$$

We can now formally what it means for a Turing machine to accept, reject, or halt on some input string.

accept, reject, halt

The Turing machine M with initial state q_0

- *accepts* x if (q_0, x) derives an accepting configuration;
- *rejects* x if (q_0, x) derives a rejecting configuration; and
- *halts* on x if it either accepts or rejects x.

decide

The Turing machine M **decides** $L \subseteq \Sigma^*$ if it accepts every $x \in L$ and rejects every $x \notin L$.

A language is **decidable** when there is a Turing machine that decides it.

The set of decidable languages is also known as the set of **recursive languages**.

recognize

The Turing machine M **recognizes** the language L if it accepts every $x \in L$ and either rejects or does not halt for every $x \notin L$.

A language is **recognizable** when there is a Turing machine that recognizes it.

The set of recognizable languages is also known as the set of **recursively enumerable languages**.

4.2 Church-Turing Thesis

Even though a Turing machine is a very restricted model of computation, we believe that it is just as powerful as any other reasonable model of computation can do. This remarkable statement is known as the Church-Turing thesis.

Church-Turing Thesis

Any decision problem that can be solved by an algorithm on any computer that we can construct in this universe corresponds to a language that can be decided by a Turing machine.

We can't prove that the Church-Turing thesis is true, but we can test it out by seeing if it applies to various models of computation that we define.

A **register Turing machine** has a finite number of registers which can each store one symbol from the tape alphabet. The transitions of these register Turing machines are determined by the current state, the content of the tape at the current head position, and the contents of all the registers. On each transition, the register Turing machine can overwrite the content of the current tape cell as well as all of the registers.

Proposition 4.1

Every language that can be decided by a register Turing machine can also be decided by a (standard) Turing machine.

Define a **subroutine Turing machine** to be a Turing machine that can call (standard) Turing machines to run as black-box subroutines on the input. A call to a subroutine can be represented as a special type of state in the subroutine Turing machine. All actions on the input tape (including movement of the current head position) persist after the subroutine halts, and two transitions are followed out of its special state; one for when the subroutine accepts and the other for when it rejects.

Proposition 4.2

Every language that can be decided by a subroutine Turing machine can also be decided by a (standard) Turing machine.

A **multitape Turing machine** is a Turing machine that has $k \geq 1$ tapes instead of one. It also has k tape heads, one per tape, which can move independently of each other, and the current symbol on each tape can be read and overwritten on each transition. In other words, the transition function of a multitape Turing machine is a function of the form

$$\delta : Q \times \Gamma^k \to Q \times (\Gamma \times \{L, R, N\})^k,$$

where L and R denote movement to the left and the right, as before, and N denotes no movement in either direction.

Initially, when we run a multitape Turing machine the input is written in tape 1 and the other tapes are empty.

Theorem 4.3

Every language that can be decided by a multitape Turing machine can also be decided by a (standard single-tape) Turing machine.

Other Models of Computation

Since its formulation, the Church-Turing thesis has been shown to hold not just for simple variants and extensions of Turing machines like the ones considered above, but for many other richer models of computation as well.

4.3 Universal Turing Machines

universal Turing machine

A **universal Turing machine** is a Turing machine U that on any input $\langle M, x \rangle$ which encodes a Turing machine M and a string x,

- Accepts if M accepts x; and
- Rejects if M rejects x.

Theorem 4.4

There exists a Universal Turing machine.

Turing-complete

A model of computation is **Turing-complete** if for every decidable language L, there is a machine/algorithm in this model that decides L.

Universal Turing machines give us a simple way to determine if a model of computation is Turing complete.

Proposition 4.5

If a universal Turing machine can be implemented in the model of computation $\mathcal{M}$, then $\mathcal{M}$ is Turing-complete.

This approach has been used to show that many programming languages (such as C, Python) and many other models of computation (such as Redstone computers in Minecraft or Conway's Game of Life) are Turing-complete.

4.4 Nondeterministic Turing Machines

nondeterministic Turing machine

A **nondeterministic Turing machine** (NTM) is an abstract machine defined by

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the tape alphabet,
- $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is the transition relation,
- $q_0 \in Q$ is the initial state,
- $q_{acc} \in Q$ is the accept state, and
- $q_{rej} \in Q \setminus \{q_{acc}\}$ is the reject state.

All the other notions of deterministic Turing machines carry over to the nondeterministic setting as well. We do need to be careful in defining what it means for a nondeterministic Turing machine to decide or to recognize a language:

decide

The nondeterministic Turing machine M **decides** $L \subseteq \Sigma^*$ if and only if

- for every $x \in L$, there exists a computational path that accepts x *and* all computational paths halt; and
- for every $x \notin L$, all computational paths reject x.

recognize

The nondeterministic Turing machine M **recognizes** $L \subseteq \Sigma^*$ if and only if

- for every $x \in L$, there exists a computational path that accepts x; and
- for every $x \notin L$, no computational path accepts x.

Theorem 4.6

The set of languages that can be decided by deterministic Turing machines is exactly the same as the set of languages that can be decided by non-deterministic Turing machines.

Theorem 4.7

The set of languages that can be recognized by deterministic Turing machines is exactly the same as the set of languages that can be recognized by non-deterministic Turing machines.

4.5 Decidable Languages

Theorem 4.8

Every context-free language is decidable.

Proposition 4.9

The language $L = \{0^n 1^n 2^n : n \geq 0\}$ is decidable.

Theorem 4.10

The class of decidable languages is closed under Union, Intersection, Complementation, Set Difference, Star operation, and Reversal.

4.6 Undecidable Languages

Theorem 4.11

There exist undecidable languages.

The language

$$A_{\text{TM}} = \{\langle M, x\rangle : M \text{ is a TM that accepts } x\}$$

is undecidable.

Once we have one explicit undecidable language, we can use it to show that other languages are also undecidable using reductions.

For example,

$$H_{\text{TM}} = \{\langle M, x\rangle : M \text{ halts on input } x\}$$

is undecidable.

4.7 Rice's Theorem

A property of recognizable languages is simply a subset of the set of all recognizable languages.

A property of recognizable languages is non-trivial if it is not the empty set or the set of all recognizable languages.

L_P

For every Turing machine language property P, let

$$L_P = \{\langle M \rangle : L(M) \in P\}$$

denote the language of all encodings of Turing machines that recognize a language in P.

Theorem 4.12: Rice's Theorem

For every non-trivial property P of recognizable language, the language L_P is undecidable.

4.8 Recognizability

Theorem 4.13

The language A_{TM} is recognizable.

Theorem 4.14

A language L is decidable if and only if both L and its complement $\overline{L}$ are both recognizable.

A corecognizable language is a language whose complement is recognizable. The last theorem states that the intersection of the set of recognizable languages and the set of corecognizable languages is exactly the set of decidable languages.

Corollary 4.15

The language $\overline{A_{\text{TM}}}$ is not recognizable.

5

Time Complexity

5.1 TIME Complexity Classes

The time cost of the Turing machine M on input x is the number of transitions it follows before it halts.

When M does not halt on x, we say that its time cost is either infinite or undefined, but in this part of the course we will only consider Turing machines that halt on all inputs.

By itself, the time cost measure above is not very useful: having to compare the time cost of two Turing machines on every possible input would not be practical or informative. Instead, we want to define a more global measure of the time cost of a Turing machine.

time cost

The **(worst-case) time cost** of the Turing machine M is the function $t : \mathbb{N} \to \mathbb{N}$ where $t(n)$ is the maximum time cost of M on any input x of length $|x| = n$.

The time cost is a measure of the efficiency of individual Turing machines. We can use this measure to obtain a notion of the complexity of (decidable) languages.

$\mathsf{TIME}(t)$

For every function $t : \mathbb{N} \to \mathbb{N}$, the time complexity class $\mathsf{TIME}(t)$ is the set of all languages that can be decided by a multitape Turing machine with worst-case time cost bounded above by $O(t)$.

The class $\mathsf{TIME}(n)$ is the set languages that can be computed by *linear-time* Turing machines and is particularly interesting.

Proposition 5.1

The class $\mathsf{TIME}(n)$ is a strict superset of the class of all regular languages.

Are there languages in *sublinear* time complexity classes? Most ambitiously, is the class $\mathsf{TIME}(1)$ of languages that can be computed with *constant-time* Turing machines non-empty? It is, and can be described precisely.

Proposition 5.2

For every $t \in o(n)$, $\mathsf{TIME}(t) = \mathsf{TIME}(1)$.

Theorem 5.3: Linear Speedup Theorem

For every constant $\epsilon > 0$, if there is a multitape Turing machine M that decides L with time cost $t(n)$, then there is also a multitape Turing machine M' that decides L and has time cost $\epsilon \cdot t(n) + n + 2$.

5.2 Time Hierarchy Theorem

time-constructible

The function $t : \mathbb{N} \to \mathbb{N}$ is **time-constructible** if there is a Turing machine that on input 0^n writes $t(n)$ ones on the tape in time $O(t(n))$.

Theorem 5.4: Time Hierarchy Theorem

For every time-constructible function $t : \mathbb{N} \to \mathbb{N}$, $\mathsf{TIME}(t(n)) \subsetneq \mathsf{TIME}(t(n) \log t(n))$.

Theorem 5.5

There exist non-time-constructible functions $t : \mathbb{N} \to \mathbb{N}$ with $t = \omega(n)$ for which $\mathsf{TIME}(t(n)) = \mathsf{TIME}(t(n) \log t(n))$.

5.3 The Class P

A **polynomial-time Turing machine** is a Turing machine with time cost $O(n^k)$ for some $k \geq 1$. The class P is the set of all languages that can be decided by some polynomial-time Turing machine. Equivalently, we can also define P in terms of TIME complexity classes.

$$\mathsf{P} := \bigcup_{k \geq 0} \mathsf{TIME}(n^k)$$

Most importantly, P is closed under subroutine calls: if a polynomial-time subroutine Turing machine M uses (black-box) calls to another polynomial-time Turing machine N, the total time complexity of M (including the time spent in the subroutines) is still polynomial.

This closure property of P will be especially useful when we start working with polynomial-time reductions, starting in the next lesson.

Finally, perhaps the most important reason that we spend so much time studying P is that it is a robust class of languages, in that for many different models of computation, the set of languages that can be decided in polynomial time happens to correspond exactly to P. In fact, it is believed that this robustness property holds for all reasonable models of computation.

Cobham-Edmonds Thesis

Any decision problem that can be solved in polynomial time by an algorithm on any physically-realizable computer corresponds to a language that is in P.

Proposition 5.6

every regular language is in P.

We define

$$\mathsf{E} := \bigcup_{k \geq 0} \mathsf{TIME}(2^{k \cdot n})$$

and

$$\mathsf{EXP} = \bigcup_{k \geq 0} \mathsf{TIME}(2^{n^k}).$$

Theorem 5.7

$\mathsf{P} \subsetneq \mathsf{E} \subsetneq \mathsf{EXP}$.

5.4 The Class NP

nondeterministic time complexity

For every function $t : \mathbb{N} \to \mathbb{N}$, the **nondeterministic time complexity** $\mathsf{NTIME}(t)$ is the set of all languages that can be decided by a nondeterministic multitape Turing machine with worst-case time cost bounded above by $O(t)$.

The class NP is the analogue of P for nondeterministic Turing machines: it is the set of all languages that can be decided by a polynomial-time nondeterministic Turing machine, and can be defined formally as follows:

$$\mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}(n^k)$$

There is another interpretation of NP which is more intuitive and also very useful.

A *verifier* is a Turing machine with an additional *certificate tape*. The input to a verifier is a pair (w, c), with the *input string* w written on the main tape and the *certificate string* c written on the certificate tape.

The execution of verifiers follows exactly the same rules as standard multitape Turing machines. However, the language decided or recognized by a verifier is defined differently.

The *language recognized* by the verifier V is

$$L(V) = \{w \in \Sigma^* : \exists c \text{ such that } V \text{ accepts } (w, c)\}.$$

The verifier V *decides* $L(V)$ if it always halts on all input pairs (w, c). A *polynomial-time verifier* is a verifier with time cost $O(n^k)$ for some constant $k \geq 0$.

Theorem 5.8

A language L is in NP if and only if it can be decided by a polynomial-time verifier.

5.5 NP-Completeness

The notion of NP-completeness aims to capture the notion of what it means to be one of the "hardest" languages in NP to compute. To make this notion precise, we need to introduce polynomial-time reductions.

polynomial-time reduction

Given two languages $A, B \subseteq \Sigma^*$, the language A is **polynomial-time reducible** to B, denoted

$$A \leq_P B$$

if there is a function $f : \Sigma^* \to \Sigma^*$ such that

1. For every $x \in \Sigma^*$, we have $x \in A \Leftrightarrow f(x) \in B$; and
2. There is a polynomial-time Turing machine M that on input $x \in \Sigma^*$, erases it and replaces it with $f(x)$ on the tape and then halts.

Polynomial-time reductions are most useful in establishing relations between different languages.

Lemma 5.9

For every two languages A, B that satisfy $A \leq_P B$,

- If $B \in P$, then $A \in P$ as well; and
- If $B \in NP$, then $A \in NP$ as well.

Note also the contrapositive of the lemma: if $A \leq_P B$ and $A \notin P$, then $B \notin P$ either. Both the direct and contrapositive forms of the lemma will prove to be useful.

NP-hard

The language L is NP-hard if every language $A \in NP$ satisfies $A \leq_P L$.

NP-complete

The language L is NP-complete if $L \in NP$; and L is NP-hard.

Proposition 5.10

If any NP-hard language L is in P, then $P = NP$.

There exists an NP-complete language.

5.6 Satisfiability

A Boolean variable is a variable that can take two possible values: True or False. A literal is a Boolean variable x or its negation $\overline{x}$. A clause is the OR (or disjunction) of one or more literals. A Boolean formula in conjunctive normal form (or CNF formula) is an AND (or conjunction) of one or more clauses. A Boolean formula is satisfiable if there exists some assignment of True and False values to its underlying Boolean variables that causes the formula to evaluate to True.

satisfiability language

The **satisfiability language** is

$$\mathsf{SAT} = \{\langle \phi \rangle : \phi \text{ is a satisfiable formula}\}.$$

It's easy to see $\mathsf{SAT} \in \mathsf{NP}$. The NP-completeness of SAT is obtained by considering tableaux. A tableau is a sequence of configurations of a Turing machine. A tableau is valid if the sequence corresponds exactly to the configurations of a Turing machine starting from its initial configuration all the way to the configuration where the machine halts. A valid tableau is accepting when the Turing machine is in an accepting state in its last configuration.

5.7 The Cook-Levin Theorem

Theorem 5.11: Cook-Levin Theorem

The language SAT is NP-complete.

Proof:
Check CS 360 notes. □

5.8 Beyond P and NP

We define complement class

$$\mathsf{coNP} = \{L \subseteq \Sigma^* : \overline{L} \in \mathsf{NP}\}.$$

Proposition 5.12

If $\mathsf{NP} \neq \mathsf{coNP}$ then $\mathsf{P} \neq \mathsf{NP}$.

If $\mathsf{NP} \neq \mathsf{coNP}$, then $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$.

Proposition 5.13

The language

$$\mathsf{FACTOR} = \{\langle M, N \rangle : M < N \in \mathbb{N} \text{ and } d \mid N \text{ for some } 1 < d < M\}$$

is in $\mathsf{NP} \cap \mathsf{coNP}$.

Theorem 5.14: Ladner's Theorem

If $\mathsf{P} \neq \mathsf{NP}$, there exist some languages in $\mathsf{NP} \setminus \mathsf{P}$ that are not NP-complete.

We could extend the notion of verifiers to obtain richer conditions for acceptance.

For example, let a $\forall\exists$-verifier be a Turing machine V with two extra tapes that include certificates c_1 & c_2 and that recognizes the language

$$L(V) = \{w \in \Sigma^* : \forall c_1 \exists c_2 \text{ such that } V \text{ accepts } (w, c_1, c_2)\}.$$

We can define the class of languages that can be efficiently decided using these verifiers as follows.

Π_2^{P} is the class of all languages that can be decided by polynomial-time $\forall\exists$-verifiers.

Theorem 5.15

If $\mathsf{P} = \mathsf{NP}$, then $\mathsf{P} = \Pi_2^{\mathsf{P}}$ as well.

The class Π_2^{P} is of course not the only possible extension of the classes NP and coNP. By extending the above generalization, we can obtain an infinite class of complexity classes: this set of complexity classes (and their union) is known as the polynomial hierarchy.

6

Space Complexity

6.1 SPACE Complexity Classes

space cost

The **(worst-case) space cost** of the Turing machine M is the function $s : \mathbb{N} \to \mathbb{N}$ where $s(n)$ is the maximum space cost of M on any input x of length $|x| = n$.

The notion of space cost applies to both single-tape and multitape Turing machines. (For multitape machines, we count the total number of visited cells in all the tapes to obtain the space cost.)

space complexity class

For every function $s : \mathbb{N} \to \mathbb{N}$, the **space complexity class** $\mathsf{SPACE}(s)$ is the set of all languages that can be decided by a multitape Turing machine with worst-case space cost bounded above by $O(s)$.

Observe that for every function $s : \mathbb{N} \to \mathbb{N}$ we have $\mathsf{TIME}(s) \subseteq \mathsf{SPACE}(s)$.

Theorem 6.1

For every function $s : \mathbb{N} \to \mathbb{N}$,

$$\mathsf{NTIME}(s) \subseteq \mathsf{SPACE}(s) \subseteq \mathsf{TIME}(2^{O(s)}).$$

The class of all languages that can be decided by Turing machines with polynomial space cost is called PSPACE.

$$\mathsf{PSPACE} := \bigcup_{k \geq 1} \mathsf{SPACE}(n^k).$$

The last theorem immediately lets us place PSPACE within the time complexity hierarchy.

Corollary 6.2

$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$.

6.2 Savitch's Theorem

We can consider the space complexity analogue of the P vs. NP problem by considering the space cost of nondeterministic Turing machines.

For every function $s : \mathbb{N} \to \mathbb{N}$, the time complexity class $\mathsf{NSPACE}(s)$ is the set of all languages that can be decided by a multitape nondeterministic Turing machine with worst-case space cost bounded above by $O(s)$.

We can use these complexity classes to define the nondeterministic analogue of PSPACE.

$$\mathsf{NPSPACE} := \bigcup_{k \geq 1} \mathsf{NSPACE}(n^k).$$

Savitch's Theorem shows that any nondeterministic Turing machine can be simulated by a deterministic Turing machine with at most a quadratic increase in the amount of space required.

Lemma 6.3

There is a Turing machine that decides Derive and has space cost only $O(s \log t)$ on input $\langle N, x, c_1, c_2, t \rangle$ when N has space cost $O(s)$ on input x.

Using the result in the lemma, we can complete the proof of Savitch's theorem.

Theorem 6.4: Savitch's Theorem

For every $s : \mathbb{N} \to \mathbb{N}$,

$$\mathsf{NSPACE}\big(s(n)\big) \subseteq \mathsf{SPACE}\big(s(n)^2\big).$$

Savitch's Theorem immediately provides a solution to the PSPACE vs. NPSPACE question.

Corollary 6.5

PSPACE = NPSPACE.

6.3 L and NL

input/work tape Turing machine

A **input/work tape Turing machine** is a multitape Turing machine whose input is on the first read-only tape and where the other tapes are standard read/write tapes.

space cost

The **space cost** of an input/work tape Turing machine M is the minimal function $s : \mathbb{N} \to \mathbb{N}$ such that on every input of length n, the machine M scans at most $s(n)$ distinct cells in the work tapes.

We can now redefine our space complexity classes with the refined notion of space cost.

For every function $s : \mathbb{N} \to \mathbb{N}$,

- $\mathsf{SPACE}(s)$ is the set of all languages that can be decided by a deterministic input/work tape Turing machine with worst-case space cost bounded above by $O(s)$; and
- $\mathsf{NSPACE}(s)$ is the set of all languages that can be decided by a nondeterministic input/work tape Turing machine with worst-case space cost bounded above by $O(s)$.

Of particular interest are the classes of languages that can be decided by Turing machine with logarithmic space cost:

$$\mathsf{L} = \mathsf{SPACE}(\log n) \text{ and } \mathsf{NL} = \mathsf{NSPACE}(\log n)$$

Proposition 6.6

$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP}$.

6.4 NL-Completeness

Does $\mathsf{L} = \mathsf{NL}$? Savitch's theorem shows that $\mathsf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$, but this does not resolve the question. We can make progress on it by studying the class of languages which are complete for NL.

log-space reduction

Given two languages $A, B \subseteq \Sigma^*$, the language A is **log-space reducible** to B, denoted

$$A \leq_{\mathsf{L}} B$$

if there is a function $f : \Sigma^* \to \Sigma^*$ such that

1. For every $x \in \Sigma^*$, we have $x \in A \Leftrightarrow f(x) \in B$; and
2. There is a logarithmic-space Turing machine M that on input $x \in \Sigma^*$, erases it and replaces it with $f(x)$ on the tape and then halts.

Lemma 6.7

For every two languages A, B that satisfy $A \leq_{\mathsf{L}} B$,

- If $B \in \mathsf{L}$, then $A \in \mathsf{L}$ as well; and
- If $B \in \mathsf{NL}$, then $A \in \mathsf{NL}$ as well.

NL-complete

The language L is NL-complete if

1. Every language $A \in \mathsf{NL}$ satisfies $A \leq_{\mathsf{L}} L$; and
2. $L \in \mathsf{NL}$.

The (s,t)-connectivity problem asks: given a directed graph $G = (V, E)$ and two vertices $s, t \in V$, does there exist a path from s to t in G? The corresponding language is

$$\mathsf{CONN} = \langle G, s, t \rangle : \text{ there is a path from } s \text{ to } t \text{ in } G.$$

Then CONN is NL-complete.

6.5 Immerman-Szelepcsenyi Theorem

Since we don't know whether L = NL or not, it's natural to turn to related questions, such as whether NL is equal to its "complement" class coNL or not.

We define $\text{coNL} = \{L \subseteq \Sigma^* : \overline{L} \in \text{L}\}$.

If we believe that L = NL, then this means that we must believe that NL = coNL as well.

Theorem 6.8: Immerman-Szelepscenyi Theorem

NL = coNL.

Proof:
See https://en.wikipedia.org/wiki/Immerman%E2%80%93Szelepcs%C3%A9nyi_theorem □

6.6 More on Space Complexity

There are many more interesting results and open problems related to space complexity. Here are two examples.

PSPACE-Completeness

Recall that the question of whether P and PSPACE are distinct is currently open. As was the case with the P vs. NP problem, there is one interesting way to try to tackle this question by considering languages that are complete for the larger class.

PSPACE-complete

The language L is PSPACE-complete if

- every $A \in$ PSPACE satisfies $A \leq_{\text{P}} L$; and
- $L \in$ PSPACE.

Just as NP-complete languages are known to be outside P if P ≠ NP, every PSPACE-complete language is guaranteed to be outside P if P and PSPACE are distinct.

Proposition 6.9

If L is PSPACE-complete and P $\subsetneq$ PSPACE, then $L \notin$ P.

We can identify an explicit PSPACE-complete problem by considering totally quantified Boolean formulae (t.q.b.f.), Boolean formulae along with ∀ (universal) and ∃ (existential) quantifiers for each variable. For example,

$$\phi(x, y, z) = \forall x \exists y \forall z : x \wedge (\overline{x} \vee y) \wedge (\overline{y} \vee z)$$

is a totally quantified Boolean formula.

Each totally quantified Boolean formula evaluates to either True or False, and the TQBF language includes the encoding of every true totally quantified Boolean formula.

$$\text{TQBF} = \{\langle \phi \rangle : \phi \text{ is a true t.q.b.f.}\}$$

Theorem 6.10

TQBF is PSPACE-complete.

The P vs. PSPACE would therefore be resolve if we can determine whether the TQBF language is in P or not.

Small Space Complexity Classes

To go in a completely different direction, we can also ask about very small space complexity classes. The smallest space complexity class is $\mathsf{SPACE}(1)$ which contains all languages that can be decided by Turing machines which require only a constant amount of space.

As it turns out, we have already studied this class previously: it corresponds exactly to the class of regular languages.

Theorem 6.11

The class $\mathsf{SPACE}(1)$ is the set of regular languages.

In fact, we can extend this result to show that there are some (slow-growing) functions $s = \omega(1)$ for which $\mathsf{SPACE}(s)$ is also equivalent to the class of regular languages.

Time and Space

So far, we have only considered the time and space complexity of a language separately. We can also combine them and consider classes of languages that can be decided by Turing machines with small time cost and small space cost.

The class SC is the class of languages that can be decided by a deterministic Turing machine with time cost $O(n^k)$ and space cost $O(\log^\ell n)$ for some parameters $k, \ell \in \mathbb{N}$.

The class name SC was chosen to stand for Steve's Class and is named after Stephen Cook for his early work on this class.

Note that the space requirement in the definition of SC corresponds to poly-logarithmic space complexity, a natural notion of space complexity.

$$\mathsf{PolyL} = \bigcup_{\ell \geq 1} \mathsf{SPACE}(\log^\ell n).$$

We can easily relate the class SC to P and to PolyL in the following way.

Proposition 6.12

$\mathsf{SC} \subseteq \mathsf{P} \cap \mathsf{PolyL}$.

Can the last proposition be strengthened to show equality between the two sides? Or, equivalently, are there problems that can be solve time-efficiently or space-efficiently, but not both at the same time? This problem is still open.

Is $\mathsf{SC} = \mathsf{P} \cap \mathsf{PolyL}$?

Index

R

S

T

U

Y