1 Regular Expressions

1.1 Alphabets, Words, Languages

language over alphabet Σ is a (finite or infinite) set of words over Σ. Σ^*: Set of all words over Σ

1.1.1 Inductive Definition of Strings

$\epsilon \in \Sigma^*$. If $w \in \Sigma^*$ and $\sigma \in \Sigma$, then $w\sigma \in \Sigma^*$

1.1.2 Prefixes, Suffixes, Substrings

If x,y,z are words and w=xyz, then (1) x is a **prefix** of w (2) y is a **substring** of w (3) z is a **suffix** of w. Note that x,y,z can all be empty

1.1.3 Simple and complex Alphabets

Alphabets can also be sets of words. A language can be seen as an alphabet, where each word is a symbol in the alphabet.

1.2.2 Syntax

Formal structure or *set of rules* defining a model

Definition. *(Regular Expressions).*

1. *a) The symbol $\emptyset$ is a regex*
   b) The symbol ϵ is a regex
   c) For each $\sigma \in \Sigma, \sigma$ is a regex
2. *If α and β are regex then*
   a) $(\alpha\beta)$ is a regex
   b) $(\alpha + \beta)$ is a regex
3. *If α is a regex, then (α^*) is a regex.*

1.2.3 Semantics

Meaning of a model, what it can express. The semantics of regular expressions assign a language $L(\alpha)$ to each RE α.

1.2.4 Operators on Languages

$L_1 \circ L_2 = \{uv \mid u \in L_1, v \in L_2\}$
$\{ab\}^* \circ \{ccc\}^* = \{\epsilon, ab, ccc, abccc, abab, ...\}$

Definition. *(RegEx: Semantics).*

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L(\sigma) = \{\sigma\}$, *for each* $\sigma \in \Sigma$
- *If α and β are regular expressions, then:*
  $L((\alpha\beta)) = L(\alpha) \cdot L(\beta)$
  $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$

Regular Language

a language that can be described by a regex. The complement of a regular language is also regular

1.3 Extended Notation

Precedence rules

(1) Brackets; (2) *; (3) concatenation; (4) +

$[a-z] \equiv a + ... + z$	
$\alpha? \equiv (\alpha + \epsilon)$	α *once or never*
$\alpha^+ \equiv \alpha\alpha^*$	α *at least once*
$\alpha^n \equiv \alpha...\alpha$	α *repeated n times*
$\alpha^{\{m,n\}}$	*min. m, max. n times*
Σ^k	*All strings of length* k

1.4 Equivalences of RegEx

Two regular expressions are equivalent if they describe the same language. We write $\alpha \equiv \beta$ if $L(\alpha) = L(\beta)$.

1.4.1 Equivalence Rules

Associativity	$\alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma$ $\alpha(\beta\gamma) \equiv (\alpha\beta)\gamma$
Commutativity	$\alpha + \beta \equiv \beta + \alpha$
Neutral elements	$\emptyset + \alpha \equiv \alpha \equiv \alpha + \emptyset$ $\epsilon\alpha \equiv \alpha \equiv \alpha\epsilon$
Distributivity	$\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$ $\epsilon\alpha \equiv \alpha \equiv \alpha\epsilon$
Idempotence	$(\alpha^*)^* \equiv \alpha^*$
Zero element	$\emptyset\alpha \equiv \emptyset \equiv \alpha\emptyset$ $\emptyset^* \equiv \epsilon$

1.4.2 Proofs of Equivalences

Proof method: Equality of sets The equality of two sets M_1, M_2 is best shown in two steps:

- Show that $M_1 \subseteq M_2$
- Show that $M_2 \subseteq M_1$
- $\forall w \in M_1$, show that $w \in M_2$
- $\forall w \in M_2$, show that $w \in M_1$

Show that $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$. In other words show that $L(\alpha(\beta + \gamma)) = L((\alpha\beta + \alpha\gamma)$

- Let α, β, γ be arbitrary.
- Here we only show: $L(\alpha(\beta + \gamma)) \subseteq L(\alpha\beta + \alpha\gamma)$
- Let w be an arbitrary string in $L(\alpha(\beta + \gamma))$:
  – Then there are $u \in L(\alpha)$ and $v \in L(\beta + \gamma)$ with $w = uv$
  – Then, $v \in L(\beta)$ or $v \in L(\gamma)$
  We distinguish between two cases:
  1. $v \in L(\beta)$: Since $u \in L(\alpha)$ and $v \in L(\beta), w = uv \in L(\alpha\beta) \implies w \in L(\alpha\beta + \alpha\gamma)$
  2. $v \in L(\gamma) : w = uv \in L(\alpha\gamma) \implies w \in L(\alpha\beta + \alpha\gamma)$
- The proof of $L(\alpha\beta + \alpha\gamma) \subseteq L(\alpha(\beta + \gamma))$ proceeds similarly.

1.4.3 Proofs of Non-Equivalences

Find a counter example

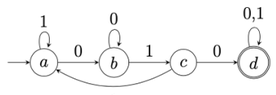
2 Finite Automata

2.1 Deterministic Finite Automata

Definition. *(DFA). **finite automaton** $\mathcal{A}$ is a quintuple consisting of*

- *finite set Q of **states***
- *an **input alphabet** Σ*
- ***transition function** $\delta : Q \times \Sigma \to Q$*
- *an **initial state** $s \in Q$*
- *a set $F \subseteq Q$ of **accepting states***

We write $\mathcal{A} = (Q; \Sigma, \delta, s, F)$



$\mathcal{A}_{010} = (Q_{010}, \{0,1\}, \delta_{010}, s_{010}, F_{010})$
$Q_{010} = \{a, b, c, d\}$
$s_{010} = a$
$F_{010} = \{d\}$
$\delta_{010}(a, 0) = b, \delta_{010}(a, 1) = a, ...$

2.1.2 Semantics

Informal Semantics

- Beginning in the initial state, the automaton reads the input character by character
- Using the transition function the automaton is able to switch states
- The automaton accepts the input if the finishing state is in F

Formal Semantics

Assigns to each finite Automaton $\mathcal{A}$ the language $L(\mathcal{A})$

Definition. *(DFA Semantics).The **extended transition function** $\delta^* : Q \times \Sigma^* \to Q$ of an automaton $\mathcal{A}$ defined inductively:*

- $\delta^*(q, \epsilon) = q$
- $\delta^*(q, u\sigma) = \delta(\delta^*(q, u), \sigma)$, *for* $u \in \Sigma^*, \sigma \in \Sigma$

*$\mathcal{A}$ **accepts** $w \iff \delta^*(s, w) \in F$*
*The language **decided** by $\mathcal{A}$: $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A}$ accepts $w\}$*

$\delta^*(q, w)$ is the state that $\mathcal{A}$ reaches after reading the string w when starting from state q is

2.2 Nondeterm. Finite Automata

We allow multiple transitions for the same symbol thus can not be perfectly reconstructed.

Definition. *(NFA) $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ consists of*

- *set Q of states*
- *input alphabet Σ*
- *initial state $s \in Q$*
- *set F of accepting states*
- *a **transisiton relation** $\delta \subseteq Q \times \Sigma \times Q$*

Similar to DFA except transition relation rather than transition function.

We say that the automaton **accepts** a word w if a computation exists in which w is completely read and an accepting state is reached.

2.2.2 NFA Semantics

Definition. *(Run) ρ through an NFA is a sequence of $q_0, \sigma_1, q_1, ...\sigma_n q_n$ where:*

- $\forall i \in \{0, ..., n\} : q_i \in Q$
- $\forall i \in \{1, ..., n\} : \sigma_i \in \Sigma$
- $\forall i \in \{1, ..., n\} : q_{i-1} \xrightarrow{\sigma_i} q_i$

A run is a sequence of the form *state, symbol, state, ..., symbol, state* with valid relations between the states.

$p \xrightarrow{w} q$: $\exists$ a run from state p to state q

2.3 NFA with ϵ-Transitions

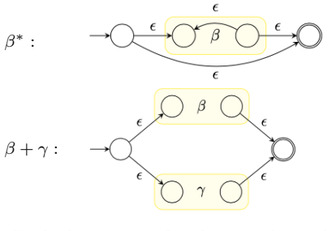
These transitions "increase the nondeterminism", since they add the possibility to change states without reading a symbol.

2.4 Converting REs into NFAs

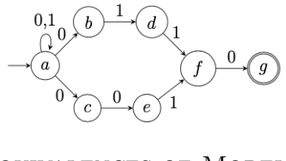
2.4.1 From REs to ϵ-NFAs

Theorem 2.1. *For every RE α $\exists$ an ϵ-NFAs $\mathcal{A}$ such that $L(\alpha) = L(\mathcal{A})$*

$\alpha = \emptyset$:
$\alpha = \epsilon$:
$\alpha = \sigma$:
$\alpha = \beta\gamma$:
$\alpha = \beta^*$:
$\alpha = \beta + \gamma$:



The individual parts can then be pieced together into one automaton:



3 Equivalences of Models

RE	Th. 2.1 ⟶	ϵ−NFA	$\mathcal{O}(\lvert\alpha\rvert)$
ϵ−NFA	⟶	NFA	ϵ−closures $\lvert Q\rvert$
ϵ−NFA	Th. 3.2 ⟶	DFA	Powerset $2^{\mathcal{O}(\lvert Q\rvert)}$
DFA	Th. 3.1 ⟶	NFA	Powerset $2^{\mathcal{O}(\lvert Q\rvert)}$
DFA	⟶	RE	State elim $4^{\mathcal{O}(\lvert Q\rvert)}$

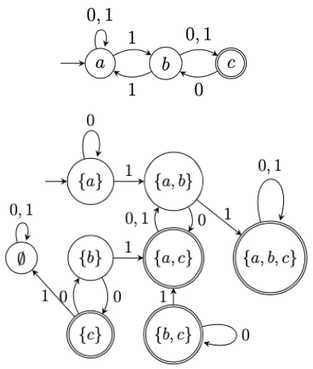
3.1 From NFAs to DFAs

Theorem 3.1. *For each NFA $\mathcal{A}$ exists a DFA $\mathcal{A}_D$ with $L(\mathcal{A}_D) = L(\mathcal{A})$*

3.1.1 Powerset Automaton

After reading a word w, the state of $\mathcal{A}_D$ should be the set of all states $\mathcal{A}$ for which there exists a run from initial state reading w

Convert the follwoing NFA to a DFA using the input 1011 as a guide.



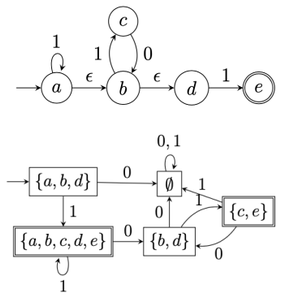
Don't forget to put the empty state $\emptyset$. Or write that all missing transitions lead to a sink state

3.2 From ϵ-NFAs to DFAs

Theorem 3.2. *For each ϵ-NFA $\mathcal{A}$ exists a DFA $\mathcal{A}_D$ with $L(\mathcal{A}_D) = L(\mathcal{A})$*

Very similar to the transformation of NFAs into DFAs with two differences:

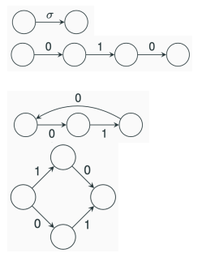
- Initial state: ϵ-closure(s)
- $\delta_D(S, \sigma) = \epsilon-\text{closure}(\{q \mid \exists p \in S : p \xrightarrow{\sigma, \mathcal{A}}\})$



ϵ-closure(p) Set of all reachable states from p without reading a symbol) ϵ-closure$(a) = \{a, b, d\}$

3.3 From REs to DFAs

σ of an RE
Concatenation
Repetition
Choice

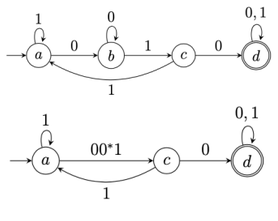


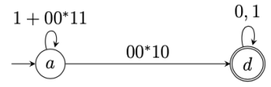
3.4 From DFAs to REs

Theorem 3.3. ***McNaughton, Yamada** For each DFA $\mathcal{A}$ there exists an RE α with $L(\alpha) = L(\mathcal{A})$*

State Elimination

- We use a hybrid automaton, whose transitions are labeled with regular expressions
- By successively removing states, a single regular expression is finally obtained





Giving us $(1 + 00^*11)^*00^*10(0 + 1)^*$

3.5 Structural Induction

- Proofs by structural induction prove that all elements of an inductively defined set have a certain property by
  – first proving the statement for the basic elements,
  – and then for the "composite" elements

4 Automata Minimization

4.1 Basic Idea

1. Unreachable states can be removed
2. States whose transitions lead to the same states and that are either all accepting or all nonaccepting can be merged
3. Sink states can be merged
4. In general: states that have the same acceptance behaviour for all subsequent input sequences (including ϵ) can be merged

4.2 Equivalence class Automaton

- The minimization of a DFA for a language L is based on a simple idea: Let x, y be two strings with $xz \in L \iff yz \in L \forall z$
  – So, the strings have the same behaviour with respect to L for all "extensions" z
  – Notation for this relation between x and $y : x \sim_L y$. $\sim_L$ is an **equivalence relation**
- Therefore, by reading either x or y, the states we reach are not distinguishable by reading any word starting from these states. $\implies$ We can merge these states into a single state
- The equivalence classes of $\sim_L$ define the states of minimal DFA for L. This DFA is called the **equivalence class automaton**

Equivalence Relation

A binary relation that is **reflexive, symmetric, transitive**

Equivalence Class

of an equivalence relation $\sim$ is a maximal set K of elements such that $x \sim y \forall x, y \in K$

4.3 Nerode Relation

Definition. *(Nerode Relation). Let $L \subseteq \Sigma^*$ be a language. The **Nerode relation** $\sim_L$ on Σ^* is defined as: $x \sim_L y \Leftrightarrow \forall z \in \Sigma^*$ it holds that: $xz \in L \Leftrightarrow yz \in L$.*

Note that it might be that $z = \epsilon$. Example: $[\epsilon] = \{w \in L \mid w$ does not contain the subword aaa and does not end with $a\}$

Proposition

If the index $(\sim_L) = k$ is finite $\implies$ $\exists$ a DFA for L with k states The **equivalence class automaton** $\mathcal{A}_L = (Q, \Sigma, \delta, s, F)$ for L is defined as follows:

- Q is the set of equivalence classes of $\sim_L$
- $s = [\epsilon]$
- $F = \{[x] | x \in L\}$
- $\forall x \in \Sigma^*, \sigma \in \Sigma : \delta([x], \sigma) = [x\sigma]$

4.4 Lower Bounds for DFAs

4.5 Myhill-Nerode Theorem

To prove a language is (not) regular: By showing it has (in)finite equivalence classes or construct a DFA, NFA or RE that recognizes it.

Theorem 4.1. *Nerode-Hill A language is regular iff $\sim_L$ has a finite number of equivalence classes*

4.6 Minimal Automaton: Uniqueness

Isomorphism

$\mathcal{A}_1$ and $\mathcal{A}_1$ are caleld isomorphic if a bijection $\pi : Q_1 \to Q_2$ exists where:

1. $\pi(s_1) = s_2$
2. $\forall q \in Q_1$ it holds that $q \in F_1 \iff \pi(q) \in F_2$
3. $\forall q \in Q_1$ and $\sigma \in \Sigma$ it holds that: $\pi(\delta_1(q, \sigma)) = \delta_2(\pi(q), \sigma)$

4.7 Algorithms: DFA Minimization

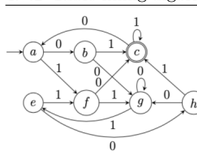
→ $\mathcal{A}_L$ is constructed by merging states from $\mathcal{A}$

- Two states p, q from $\mathcal{A}$ can be merged, if they are equivalent according to: $\forall z \in \Sigma^*$: $\delta^*(p, z) \in F \iff \delta^*(q, z) \in F$
- The basic idea behind our algorithm is to first calculate which states can not be merged
- Therefore, we consider an algorithm that calculates the set $N(\mathcal{A})$ of **non-equivalent states:** $N(\mathcal{A}) = \{(p, q) \mid p, q \in Q, \exists z : (\delta^*(p, z) \in F \not\Leftrightarrow \delta^*(q, z) \in F)\}$
- The calculation is done inductively:
  – Begin by computing pairs which are inequivalent due to $z = \epsilon$
  – Afterwards, compute pairs which are inequivalent due to transitions to other inequivalent pairs

4.7.1 Minimization Algorithm

- Input: $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
- Output: minimal DFA $\mathcal{A}$ with $L(\mathcal{A}' (= L(\mathcal{A})$
  1. Remove all states of $\mathcal{A}$ which are not reachable from s
  2. Compute the relation $N(\mathcal{A})$ using the table filling algorithm
  3. Merge iteratively all unmarked state pairs into one state
- Total runtime complexity: $\mathcal{O}(\lvert Q\rvert^2 \lvert\Sigma\rvert)$

4.7.2 Table filling algorithm



	a	b	c	e	f	g	h
a	■	X^1	X^0		X^1		X^1
b	■	■	X^0	X^1	X^1	X^1	
c	■	■	■	X^0	X^0	X^0	X^0
e	■	■	■	■	X^1		X^1
f	■	■	■	■	■	X^1	X^1
g	■	■	■	■	■	■	X^1
h	■	■	■	■	■	■	■

- Input: $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
- Output: Relation $N(\mathcal{A})$
  1. $N = \{(p, q), (q, p) \mid p \in F, q \notin F$
  2. $N' = \{(p, q) \notin N \mid \exists \sigma \in \Sigma : (\delta(p, \sigma), \delta(q, \sigma)) \in N)\}$
  3. $N = N \cup N'$
  4. If $N' \neq \emptyset$, continue with 2
  5. Output N

4.7.3 Form an RE to a DFA: In Full

1. Specify the language by a regex α
2. Convert α into an ϵ−NFA $\mathcal{A}_\infty$
3. Convert $\mathcal{A}_\infty$ into a DFA $\mathcal{A}_\in$
4. Convert $\mathcal{A}_\in$ into a minimal DFA $\mathcal{A}_\ni$

4.8 Further Insights

The Myhill Nerode Theorem can be used to show a language is (not) regular (Even though the pumping lemma is more suitable to show it is not). By demonstrating a language has (in)finitely many equivalence classses. To verify a set of equivalence classes one has to show:

1. Every string appears in a class
2. $\forall u, v$ in the same class it holds that $u \sim_L v$
2' All strings v of a class have the same set L/v
3. For the strings u, v from different classes it holds that $u \nsim_L v$
3' Different classes have different sets L/v

L/v is the set of all strings than can be appended to v such that the final word is in L.

4.9 Minimal NFAs

For every NFA A there is also a minimal NFA A with L(A) = L(A). However, in general, minimal NFAs are not unique up to isomorphism!

5 Limits, Closure Properties, Algorithms

5.1 Pumping Lemma

- Can show that a language is **not** regular
- Does not always work
- Can't be used to prove that a language is regular
- A 'cycle' in an accepting path can be repeated arbitrarily often
- If $\exists$ more words then equivalence classes ($|w| \geq |Q|$) $\implies$ $\exists$ a state that the DFA visits (at least) twice while reading w

Theorem 5.1. *(Pumping Lemma). L is regular* $\implies$ *$\exists\, n$ s.t. each $w \in L$ with $|w| \geq n$ can be decomposed into xyz s.t. that:(1) $y \neq \epsilon$ (2) $|xy| \leq n$ (3) $\forall k \geq 0 : xy^k z \in L(\mathcal{A})$*

Contraposition more useful

Corollary 5.1.1. *Let L be a language. Assume that for each $n > 0$ $\exists$ a string $w \in L$ with $|q| \geq n$ s.t. for each decomposition $w = xyz$ with (1) $y \neq \epsilon$ (2) $|xy| \leq n$ there $\exists$ a k0 s.t. $xy^k z \notin L$* $\implies$ *not regular.*

Example

$L_{a<b}$={w $\in \{a,b\}^* | \#_a(w) < \#_b(w)\}$

Let $n \in \mathbb{N}$ be arbitrary. Choose $w = a^n b^{n+1} \in L_{a<b} : \Rightarrow |w| = 2n+1 \geq n$

Let x, y, z be arbitrary strings with $w = xyz$ and: (1) $y \neq \epsilon$ (2) $|xy| \leq n$. Due to (2), y does not contain any b. Due to (1), y contains at l. one a.

Choose $k \geq 2$:
$\Rightarrow$ xy^2z contains at least $n+1$ a's but at most $n+1$ b's
$\Rightarrow$ $xy^2z \notin L_{a<b}$
$\Rightarrow$ $L_{a<b}$ is not regular.

Theorem 5.2. *(Jaffe, 78). L is regular* $\iff$ *$\exists\, n$ s.t. each string $w \in L$ with $|w| \geq n$ can be written as $w = xyz$ in at least one way s.t. condiitons hold: (1)$y \notin \epsilon$(2)$|xy| \leq n$(3)$\forall k \geq 0 : xy^k z \sim_L xyz$*

all xyk^z have to be in the same Nerode equivalence class as xyz

5.2 Regular Languages: Counting

We've encountered the intuitive criterion that a language is regular if it suffices to remember a constant amount of information while reading a word, independently of the input length.

5.3 Closure Properties and Synthesis of Automata

5.4 Synthesis of Finite Automata

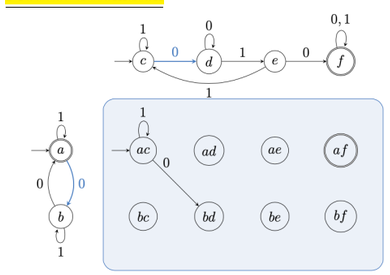
5.4.1 Boolean Operations

Theorem 5.3. *Let $\mathcal{A}_1, \mathcal{A}_2$ be DFAs.Then, DFAs can be constructed for the follwoing languages:*

	States	*Time*
$L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$	$\lvert Q_1\rvert\lvert Q_2\rvert$	$\mathcal{O}(\lvert Q_1\rvert\lvert Q_2\rvert\lvert\Sigma\rvert)$
$L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$	$\lvert Q_1\rvert\lvert Q_2\rvert$	$\mathcal{O}(\lvert Q_1\rvert\lvert Q_2\rvert\lvert\Sigma\rvert)$
$\Sigma^* - L(\mathcal{A}_1)$	$\lvert Q_1\rvert$	$\mathcal{O}(\lvert Q_1\rvert)$

Corollary 5.3.1. *The class of reg. languages is closed for intersection, union and complementation.*

Product automaton



1. Determine initial and accepting state. Only when both accept
2. Add transitions
3. Remove unreachable states!

5.4.2 Size

	DFA $\to$ DFA	DFA $\to$ NFA	NFA $\to$ NFA
$L_1 \cap L_2$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$
$L_1 \cup L_2$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert + \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert + \lvert Q_2\rvert)$
$L_1 - L_2$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert Q_2\rvert)$	$\lvert Q_1\rvert \times 2^{\mathcal{O}(\lvert Q_2\rvert)}$
$L_1 \circ L_2$	$\lvert Q_1\rvert \times 2^{\mathcal{O}(\lvert Q_2\rvert)}$	$\mathcal{O}(\lvert Q_1\rvert + \lvert Q_2\rvert)$	$\mathcal{O}(\lvert Q_1\rvert + \lvert Q_2\rvert)$
L_1^*	$2^{\mathcal{O}(\lvert Q_1\rvert)}$	$\mathcal{O}(\lvert Q_1\rvert)$	$\mathcal{O}(\lvert Q_1\rvert)$
$h(L_1)$	$2^{\mathcal{O}(\lvert Q_1\rvert \times \lvert h\rvert)}$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert h\rvert)$	$\mathcal{O}(\lvert Q_1\rvert \times \lvert h\rvert)$
$h^{-1}(L_1)$	$\mathcal{O}(\lvert Q_1\rvert)$	$\mathcal{O}(\lvert Q_1\rvert)$	$\mathcal{O}(\lvert Q_1\rvert)$

5.5 Closure Properties Reg Lang

5.5.1 Closure: Concatenation & Iteration

Theorem 5.6. *Let $\mathcal{A}_1, \mathcal{A}_2$ be DFAs (or NFAs) for L_1, L_2. Then DFAs (or NFAs) can be constructed for $L_1 \circ L_2$ and L_1^**

5.5.2 Closure: Homomorphism

Definition. *(Homomorphism) A function $h : \Sigma^* \to \Gamma^*$ is a **homomorphism** if for all strings $u, v \in \Sigma^* : h(uv) = h(u)h(v)$.*

From the definition it follows that $h(\epsilon) = \epsilon$

Theorem 5.7. *If L over Sigma and Γ is an alphabet, then h(L) for each homomorphism h: $\Sigma^* \to \Gamma^*$ is regular.*

5.6 Further Algos for Automata

5.6.1 Emptiness Test

1. Forget the edge labels
2. Insert a target node t and edges from all accepting nodes to t
3. Test if there is a path from s to t
4. If yes: output "$L(\mathcal{A}) \neq \emptyset$"

It might be simpler to backwards, from t to s

5.6.2 Word Problem

Check if specific word in a given language

DFA	simulate	$\mathcal{O}(\lvert w\rvert + \delta)$
NFA	simulate powerset aut.	$\mathcal{O}(\lvert w\rvert \times \delta)$
RE	$\to \epsilon$-NFA sim. powset aut.	$\mathcal{O}(\lvert w\rvert \times \alpha)$

5.6.3 Equivalence Test for DFAs

Two finite automata are equivalent if they decide the same language. You can show that two automata are not equivalent by giving a word that is accepted by one of the automata but not by the other. To show that two automata are equivalent, use (1) or (2)

(1) With minimal Automata

- Construct min. automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$
- Test if $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are isomorphic
  - Construct step-wise bijection π from (the states of) $\mathcal{A}'_1$ to $\mathcal{A}'_2$
  - Initialization: π maps the initial state onto the initial state
  - Then: Continue π according to the transitions of $\mathcal{A}'_1$ and $\mathcal{A}'_2$
  - "Both DFAs are now isomorphic according to $a \to c$ and $b \to d \Rightarrow$ DFAs are equiv." Also draw the graphs!

(2) With product Automata

5.6.4 Equivalence Test for NFAs and REs

The problem is (likely) even harder than NP.

5.6.5 Finiteness Test

We know that if we can pump a word, we can repeat a cycle over and over again, which would mean that the language is infinite. On the other hand, if there are no loops or cycles, the language must be finite.

Theorem 5.8. *The language of a DFA is infinite $\exists$ state q with:*

- *q is reachable from s,*
- *q lies on a cycle,*
- *An accepting state is reachable from q*

6 Context-Free Language (1)

Prove a language is context-free: construct PDA.

Context-free grammar (CFG)

$G = (V, \Sigma, S, P)$ consists of

- finite set V of **variables**
- an alphabet Σ
- a **start symbol** $S \in V$
- finite set P of **rules**: $P \subseteq V \times (V \cup \Sigma)^*$

where $\Sigma \cap V = \emptyset$ has to hold

Notation and terminology

- Instead of $(A, BC) \in P$ we write $A \to BC$
  - Elements of $V \cup \Sigma$: **symbols**
  - Elements of Σ: **terminals**
  - Elements of V: **nonterminals**
  - Rules in P: **Productions**

6.1 Semantics

Derivation

- A derivation is a sequence of derivation steps
- In a derivation step a variable X is replaced by the right-hand side γ of a rule $X \to \gamma$ e.g: $bSB \Rightarrow_G baSab$

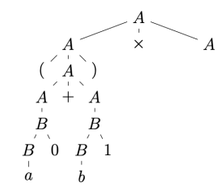
Derivation Step

- Let $G = (V, \Sigma, S, P)$ be a context-free grammar
- A string of terminals and non-terminals of G is a string from $(V \cup \Sigma)^*$
- If u and v are such strings of G, then v results from u in a **derivation step** if
  - $u = \alpha X \beta$
  - $v = \alpha\gamma\beta$

  for
  - a variable X
  - $\alpha, \beta, \gamma \in (\forall \cup \Sigma)^*$
  - a rule $X \to \gamma$ in P
- Notation: $u \Rightarrow_G v$

6.2 Derivation Trees

$A \to \quad B \mid A+A \mid A \times A \mid (A)$
$B \to \quad Ba \mid Bb \mid B0 \mid B1 \mid a \mid b$



$A \Rightarrow_\ell A \times A$
$\Rightarrow_\ell (A) \times A$
$\Rightarrow_\ell (A + A) \times A$
$\Rightarrow_\ell (B + A) \times A$
$\Rightarrow_\ell (B0 + A) \times A$
$\Rightarrow_\ell (a0 + A) \times A$
$\Rightarrow_\ell (a0 + B) \times A$
$\Rightarrow_\ell (a0 + B1) \times A$
$\Rightarrow_\ell (a0 + b1) \times A$

This is a derivation tree for the string $(a0 + b1) \times a + b$. *!When asked for a derivation write down the derivation, not the derivation tree! Put the l*

6.2.1 Leftmost Rightmost Derivations

- In each step, the left-/rightmost variable is replaced.
- For each derivation tree, there exist corresponding leftmost and rightmost derivations.
- Leftmost and rightmost derivations are not necessarily unique in a tree
- There are derivations that are neither leftmost nor rightmost

6.3 Ambiguity

For a given grammar and string there are multiple derivations. Unfortunately, there is no general procedure to test if a grammar is unambiguous.

Ambiguous grammar

(Definition) A context-free grammar G is called ambiguous if there exists a string w which admits two different derivation trees with respect to G. If there is only one derivation tree, G is called unambiguous.

6.4 Extended CFGs

allowing for more compact representations while preserving the semantics.

Extended Context-free grammar (CFG)

$G = (V, \Sigma, S, P)$ consists of

- set V of **variables**
- an alphabet Σ
- a **start symbol** $S \in V$
- finite set P which contains exactly one **rule**: $X \to \alpha_x$ for each variable $X \in V$ where α_x is a regular expression over $V\Sigma$

where $\Sigma \cap V = \emptyset$ has to hold

7 Normal Forms

7.1 Chomsky Normal Form (CNF)

7.1.1 Useful Symbol

A symbol X of a grammar is called useful if X appears in a derivation $S \Rightarrow^*_G w$ of a word $w \in \Sigma^*$ In the grammar: $S \to AB|C$; $A \to b$; $C \to ab$; $D \to c$. Only S and C are useful: they appear in the (only) derivation $S \Rightarrow C \Rightarrow ab$. A, B, D are all not useful. A is not useful because it can only be derived from S along with B, which has no rule.

Definition. *Chomsky Normal Form, CNF $G = (V, \Sigma, S, P)$ is in **Chomsky normal form** if*

- *G only contains useful symbols*
- *All rules of G are of the form*
  - *$X \to YZ$*
  - *$X \to \sigma$*

  with $X, Y, Z \in \Sigma$
- *If S does not appear on the right-hand side of any rule, the rule $S \to \epsilon$ is allowed*

Theorem 7.1. *(Chomsky 59)*

- *For every context-free language L, there exists a grammar G in CNF with $L(G) = L$.*
- *Any given grammar G_0 for L can be brought in CNF in time $\mathcal{O}(|G_0|^4)$.*

7.2 Conversion to CNF

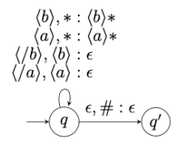
(G_1) Only useful symbols are allowed.
  (a) Remove non-generating rules
  (b) Remove non reachable rules
(G_2) Right-hand sides contain exactly one terminal symbol or only variables:
  (a) For every terminal symbol σ: Insert a new variable W_σ and replace σ by W_σ everywhere on the right-hand side, incl. in rules but only replace symbols σ not variables X!
  (b) Insert a new rule $W_\sigma \to \sigma$
(G_3) No $X \to \beta$ rules with $|\beta| > 2$ are allowed: We only have to look at the rules containing more than two variables. The first would be $S \to W_b DD$. To shorten this, we introduce a new proxy-variable S_1 and add the rules $S \to W_b S_1$ and $S_1 \to DD$.
(G_4) No ϵ-rules allowed
  (a) compute the set V_ϵ of all variables Y with $Y \to^* \epsilon$
  (b) $\forall$ right-hand side occurrence $X \to YZ$ or $X \to ZY$ of these variables, add new rules $X \to Z$
  (c) eliminate all rules of the form $X \to \epsilon$
  (d) remove useless variables (1) non-generating (2) non-reachable
(G_5) No unit rules ($X \to Y$) allowed: compute all pairs $U = \{(XY) \mid X \Rightarrow^* Y, X \neq Y\}$. For these pairs, check if a non-unitrule $Y \to \alpha$ occurs, and if so, a new rule $X \to \alpha$ is added. $Y \to \alpha$ is a non-unit rule if $= Z_1 Z_2$ or $\alpha = \sigma$ (two variables or a terminal symbol).
Eliminate unit rules
(G_6) If language L contains the string ϵ: if S does not appear on the right-hand side of any rule then Insert the rule $S \to \epsilon$. Else: Insert a new start symbol S and the rule $S \to \epsilon$

8 Pushdown Automata

8.1 Definition

A pushdwon automaton (PDA) $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, \tau_0, F)$ consists of

- finite set of states Q
- input alphabet Σ
- stack alphabet Γ
- finite transition relation $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$
- initial state $q_0 \in Q$
- initial stack symbol $\tau_0 \in \Gamma$
- set F of accepting states



The only visible difference to finite automata lies in the transitions.

8.1.1 Transition relation δ

(p, σ, τ, q, z)

- p current state
- σ input symbol with $\sigma \in \{\Sigma \cup \epsilon\}$
- τ symbol on top of stack
- q new state
- z replace τ with z

Given a state (Q), an input symbol ($\Sigma \cup \{\epsilon\}$) and a symbol on the top of our stack (Γ), this leads us to another state (Q), and we might make changes on our stack (Γ^*).

Instructions of the form:

$$\langle b \rangle, * : \langle b \rangle *$$

input, top of stack : new top of stack

* not a stack symbol, can be substituted with any other symbol in the stack alphabet.

8.1.2 Configurations

Describe the current "situation".

Definition. *(Configuration of a PDA) Let $A\ \mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, \tau_0, F)$ be a PDA. A **configuration** $(q, u, v$ of $\mathcal{A}$ consists of:*

- *a state $q \in Q$*
- *the remaining input $u \in^*$*
- *the stack content $v \in \Gamma^*$*

Start configuration with input w: (s,w,τ_0)

8.1.3 Configurations and Runs

Definition. *(Sucessor Configuration) Let $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, \tau_0, F)$ be a PDA. The succesor configuration $\vdash_\mathcal{A}$ is defined as follows:*

- *$\forall p, q \in Q, \sigma \in \Sigma, \tau \in \Gamma, u \in \Sigma^*, z, v \in \Gamma^*$:*
- *$(p, \sigma u, \tau v) \vdash_\mathcal{A} (q, u, zv)$ if $(p, \sigma, \tau, q, z) \in \delta$*
- *$(p, u, \tau v = \vdash_\mathcal{A} (q, u, zv)$ if $(p, \epsilon, \tau, q, z) \in \delta$*

*If $; \vdash_\mathcal{A} K'$ then K' is called a **successor configuration** of K*

Definition. *(Run of PDA). A run of a PDA $\mathcal{A}$ is sequence $K_1, \ldots, K_n$ of configurations with $K_I \vdash_\mathcal{A} K_{i+1} \forall i \in \{1, \ldots, n-1\}$*

8.1.4 Accepting

8.2 Empty Stack vs. Accepting States

Theorem 8.1. *For each PDA $\mathcal{A}$ whihc accepts with an empty stack, $\exists$ a PDA $\mathcal{B}$ which accepts with accepting states such thath $L(\mathcal{B}) = L(\mathcal{A})$ and vice versa.*

8.2.1 Empty Stack $\to$ Accepting States

8.2.2 Accepting States $\to$ Empty Stack

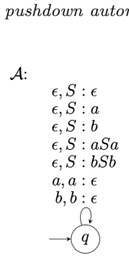
8.3 CFGs $\equiv$ PDAs

Theorem 8.2. *For each context-free grammar G, there exists a pushdown automata $\mathcal{A}$ with $L(\mathcal{A}) = L(G)$*

G:

$S \to \epsilon$
$S \to a$
$S \to b$
$S \to aSa$
$S \to bSb$

$\mathcal{A}$:



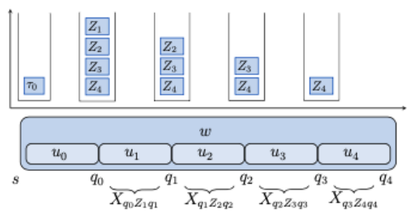
Theorem 8.3. *For each pushdown automaton $\mathcal{A}$ $\exists$ a context-free grammar G with $L(G) = L(A)$*

On a high level, we suppose that we have a PDA with initial stack symbol τ_0 and that we are in the initial state s, and we want to read the input word w:

First step $\mathcal{A}$ replaces the initial stack symbol with a string $Z_1 \ldots Z_k$ and reads a prefix u_0 of the input w:

The symbols $Z_1 \ldots Z_k$ are then successively deleted from the stack in the rest of the computation:

Thereby the substrings $u_1, \ldots, u_k$ of the input are read:



Idea for the grammar

For each combination $p, p' \in Q, \tau \in \Gamma$ the grammar G contains a variable $X_{p,\tau,p'}$ which generates all strings for which $\mathcal{A}$ has a partial computation from state p to state p' which overall deletes the symbol τ from the stack.

8.4 Deterministic PDAs

In deterministic pushdown automata (DPDAs), there may only be one transition (p, σ, τ, q, w) in δ for each combination of p, σ, τ.

We can have an ϵ-transition going out of q for stack symbol τ, only if for the same state q and stack symbol τ, there is no further transition.

$\exists\, (p, \epsilon, \tau, q, w) \implies$ there may not be a transition $(p, \sigma, \tau, q', w')$ for any σ.
(1) $(p, \epsilon, \tau, q_1, w_1)$ (2) $(p, \epsilon, \tau, q_2, w_2)$

9 Context-Free Language (2)

9.1 Pumping Lemma

To determine that a language is not context-free.

9.2 Application

Corollary 9.0.1. *Let L be a language. Assume that for each $n > 0 \exists$ a string $z \in L$ with $|z| \geq n$ such that for each decomposition $z = uvwxy$ with (1) $vx \neq \epsilon$ (2) $|vwx| \leq n$ $\exists$ a $k \geq 0$ where $uv^k wx^k y \notin L$* $\implies$ *L is not context-free.*

Let $n > 0$ be arbitrary. Choose $z = a^n b^n c^n \in L, |z| \geq n$ holds. Let $uvwxy$ be a decomposition of z with $u, v, w, x, y \in \{a, b\}^*$ satisfying (1) and (2). Due to $|vwx| \leq n$ and because there are n b's vx can never consist of more than two different symbols. Choose $k = 0 : uv^0 wx^0 y = uwy \notin L$ since... $\Rightarrow L$ is not context-free

- Decomposition z in $uvwxy$ no choice can be made, the argument must work for arbitrary decompositions which fulfill (1) and (2)
- k can be chosen freely and individually for each possible decomposition z
- In the context-free pumping lemma vwx can be anywhere in z.

9.3 CFG Closure Properties

Theorem 9.1. *The class of context-free languages is closed under: union, concatenation, the *–operator, the +–operator*

1. If L is context-free then $\{w^R \mid w \in L\}$ is as well
2. If $L \subseteq \Sigma^*$ is context-free and $h : \Sigma^* \to \Gamma^*$ a homomorphism, then $h(L)$ is context-free
3. If $L \subseteq \Sigma^*$ is context-free and $h : \Sigma^* \to \Gamma^*$ a homomorphism, then $h^{-1}(L)$ is context-free

Theorem 9.2. *The class of context-free languages is* ***not*** *closed under: intersection, complementation, set difference.*

Theorem 9.3. *If L_1 is context-free and L_2 is regular, then $L_1 \cap L_2$ is context-free.*

9.4 Closure Propert of det. CFGs

9.4.1 Closure Under Complementation

Theorem 9.4. *Deterministic context-free languages are closed under complementation.*

9.4.2 Union and Intersection

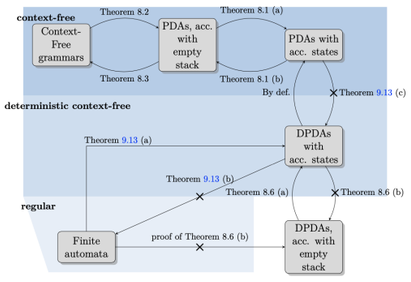
The class of deterministic context-free languages is **not** closed under union and intersection.

9.4.3 Not all contxt-free-langs have a DPDA

The class of deterministic context-free languages forms a proper subset of the class of context-free languages.

9.5 Relation Between Classes

Theorem 9.5. *1. Each regular language is deterministic context-free*
2. There are deterministic context-free languages which are not regular
3. There are context-free languages which are not deterministic context-free



10 Syntax Analysis

Syntax analysis is used to verify whether a given program is syntactically correct

10.0.1 Word Problem

Definition. *(Word Problem for CFGs) Input: A word $w \in \Sigma^*$ and a grammar G. Output: Is $w \in L(G)$*

Definition. *(Syntax Analysis Problem for CFGs). Input: A word $w \in \Sigma^*$ and a grammar G. Desired output: If $w \in L(G)$: Derivation tree*

10.1 Backtracking

10.1.1 Top-down Syntax Analysis

The backtracking algo is "brute-force" iteratively trying all derivations $S \Rightarrow_l^* w[1,m]X\alpha$ of the form $X \to \beta$ to generate the leftmost derivation. It's $\mathcal{O}(p^n)$ exp.

10.2 CYK Algorithm

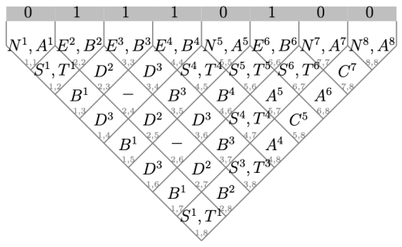
- Takes cubic time $\mathcal{O}(|w|^3|G|)$
- Uses dynamic programming

Solving the word problem

1. Write down the input word: 01110100.
2. Construct a triangular table. First row: one cell for each symbol of the input word, and each consecutive row has one symbol less: The numbers at the bottom of each cell represent the indices. For example, the (4,6) cell represents the substring from 4 to 6.
3. Fill the table. The first row check whether there is a rule that derives a given symbol.
4. Next row. In cell (1,2) check for a rule that concatenates its children, (1,1) and (2,2).
5. Cell (1,3), we cross over the results from the previous row. Concretely, we match (1,2) with (3,3) and (1,1) with (2,3).
6. Continue until final cell,
7. At this stage, we are particularly interested in S. Other variables would not be good enough.

10.2.1 Extended CYK ALgorithm

For solving the *syntax analysis problem* to obtain the derivation tree we extend the CYK algorithm slightly, to keep track of how we partitioned the word. B^2 means we split between the second and third symbol



10.3 Top-Down Syntax Analysis

Restricted CFG so syntax analysis can run in linear time and the next applicable rule only depends on the next input symbol.

10.3.2 LL(1) Grammars

Definition. *(LL(1) Grammar). Let G be a CFG without useless variables & left-recursion. $G \in LL(1)$ if $\forall$ variables X and rule $X \to \alpha, X \to \beta$, with $\alpha \neq \beta$ the following conditions hold:*

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. *If $\alpha \Rightarrow^* \epsilon$, then $FOLLOW(X) \cap FIRST(\beta) = \emptyset$*

For a sentential form α, let FIRST(α)= $\{\sigma \in \Sigma \mid \alpha \Rightarrow^* \sigma v, v \in \Sigma^*\} \cup \{\epsilon \mid a \Rightarrow^* \epsilon\}$
FIRST(α) contains all first terminal symbols of strings that can be derived from α, and possibly also ϵ, if it can be derived from α.

For a variable X let
FOLLOW(X)= $\{\sigma \in \Sigma \mid S \Rightarrow^* uX\sigma v, u, v \in \Sigma^*\}$
FOLLOW(X) contains all terminal symbols that may appear directly after X in a sentential form for X.

10.3.3 Look up table

Derived: $caAB$ Input: c a **b** a

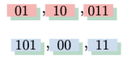
	a	**b**	*c*	(	)
S		AB	AB	$(S)S$	
A		ϵ	CA		
C			ca		
B		ba			

Simply fill in the rules corresponding to terminal symbols and variables

11 Turing Machines

Post Correspondence Problem

11.0.1 Pseudo PCP



- Given red & blue blocks with strings
- An arbitrary number of blocks is available for each block type
- Is there a word that can be constructed from red blocks as well as from blue blocks?

11.0.2 PCP

- Given a set of block types
- An arbitrary number of blocks is available for each block type.
- Can the blocks be arranged in a line such that the red (upper) word equals the blue (lower) word?

Definition. *(PCP).*
Input: *Sequence $(u_1, v_1), \ldots, (u_k, v_k)$ of pairs of non-empty strings.*
Question: *Is there an index sequence $i_1, \ldots, i_n$ with $n \geq 1$ such that $u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$?*

We call an index string $\vec{i} = i_1, \ldots, i_n$ a **solution** and the string $u_{i_1} u_{i_2} \ldots u_{i_n}$ a **solution string**.

Theorem 11.1. *PCP is not decidable.*

11.1 Turing Machines

11.1.1 Definition

Definition. *(Turing Machine (Syntax)) $M = (Q, \Gamma, \delta, s)$ consists of:*

- *a finite set of states Q*
- *working alphabet Γ with $\sqcup, \triangleright \in \Gamma$*
- *transition function δ: $Q \times \Gamma \to (Q \cup \{yes, no\}) \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$*
- *start state $s \in Q$*

where it holds that $Q, \Gamma, \{yes, no\}$ and $\{\leftarrow, \downarrow, \rightarrow\}$ are pairwise disjoint.

- The symbol $\triangleright$ signifying the left end of the tape must not be overwritten
- The head is not allowed to move left from $\triangleright$

11.1.2 Transitions

before: after, move

a: Read over 0 and 1; move left– if $\triangleright$ or # then move right; switch to *b*
b: If 0 then overwrite it by # and move right; switch to *c*...

11.1.3 Configurations

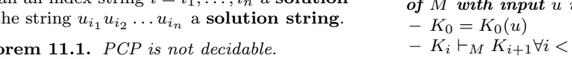
Definition. *(String-Tape Head Decription) (w,z) consists of*

- *a string $w \in \Gamma^*$*
- *a tape head position z $\mathbb{N}, 1 \leq z \leq |w|$*

Sometimes the notation (u, σ, v) is used instead of (w, z) with $w = u\sigma v$, and $|u\sigma| = z$. $(\triangleright 10, 0, 00\_)$

Definition. *(Configuration) of M is a tuple (q,(w,z)) consisting of*

- *a state $q \in Q \cup \{yes, no\}$*
- *a string-tape head description (w,z)*



Definition. *(Successor configuration $\vdash_M$) Let $K = (q, (w, z))$ be a configuration with $w[z] = \sigma$ and let $\delta(q, \sigma) = (q', \tau, d)$ with $\tau \in \Gamma, d \in \{\leftarrow, \downarrow, \rightarrow\}$.*

Then $K' = (q', (w', z'))$ is the successor configuration of K: $K \vdash_M K'$. We write $K \vdash_M^* K'$ if $\exists$ configurations $K_1, \ldots, K_l$ with $K \vdash_M K_1 \vdash_M \cdots \vdash_M K_l \vdash_M K'$

11.1.4 Semantics

When does a TM accept or reject? What does it's output for a particular input word mean?

Definition. *(TM (Semantics)) Let M be a TM.*

- $\Sigma \subseteq \Gamma - \{\triangleright, \sqcup\}$ *is called the* ***input/output alphabet***
- ***Start configuration*** *with input $u \in \Sigma^*$ is $K_0(u) = (s, (\triangleright u, 1))$*
- $(q, (w, z))$ *is called a* ***halting configuration*** *if $q \in \{yes, no\}$*
- *$K_0, K_1, \ldots, K_t$ is called a* ***finite computation of M with input*** *u if:*
  - *$K_0 = K_0(u)$*
  - *$K_i \vdash_M K_{i+1} \forall i < t$*
  - *K_t is a halting configuration*
- *...*

If a TM is used to answer yes-no questions it can be described as follows:

Definition. *(Language decided by M). A TM M decides a language L if for each $u \in Sigma^*$:*

- *$u \in L \implies$ M accepts u*
- *$u \notin L \implies$ M rejects u*

TM has to halt for every possible input word.

Definition. *(L(M)) L(M)= set of all words accepted by M*

11.1.5 Functions

Definition. *$f_M(u) = v \in \Sigma^*$ if*

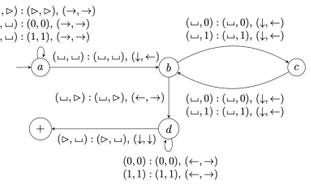
- *$K_0(u) \vdash_M^* (yes, (\triangleright v, 1))$ or*
- *$K_0(u) \vdash_M^* (yes, (\triangleright v \tau w, 1))$ for a $\tau \in \Gamma - \Sigma, w \in \Gamma^*$*

$f_M(u)$ is undefined ($f_M(u) = \perp$) if $\nexists$ such a v.

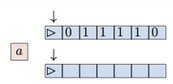
Function-calculating TMs must finish with the head pointing on the starting symbol $\triangleright$

Definition. *(Turing computable) A partial function $F : \Sigma^* \to \Sigma^*$ is Turing computable if $f = f_M$ for a Turing Machine M*

11.2 Multitape Turing Machines



11.2.1 Transition Functions



The starting state is a. The first tuple specifies that nothing is to the left of the first tape head, that the first tape head is on the starting symbol, and that the input is to the right of the tape head.

11.2.2 Robustness

Theorem 11.2. *For each multi-tape TM M $\exists$ a 1-tape TM M'. Furthermore M' halts for exactly the same inputs as M*

11.3 Church-Turing Thesis

The class of functions computable by Turing machines encompasses all intuitively computable functions. It is **not** provable.

11.4 Decidability & Computability

Definition. *(Decidable) A set $L \subseteq \Sigma^*$ is called decidable if $\exists$ a TM M which decides L.*

A TM that decides a language has to halt for all inputs over Σ^*, including those that are not in the language. As a consequence, if L is decidable, then the complement $\bar{L}$ of L is also decidable.

Decidability is concerned with yes-no questions, computability is about asking an openquestion demanding a result:

Definition. *(Computable, $\mathcal{R}$) A partial function $f : \Sigma^* \to \Sigma^*$ is called* ***computable*** *if it is Turing computable. $\mathcal{R}$ = the set of computable partial functions*

Note: Total functions are also partial functions

11.4.1 Algorithmic Problems vs. Languages & Functions

How can "real" algorithmic problems be expressed by such sets or functions?

Encodings All kinds of structures can be encoded by 0-1 strings.

Languages Algorithmic decision problems correspond to languages.

12 Undecidable Problems (1)

12.1 "Hello, World!" Problems

12.1.1 Autom. Verification of General Programs

Definition. *(Verification Problem) Input: Program P, property E Question: Does P feature property E?*

Theorem. *(Fermat's Theorem [Wiles 95]) $x^n + y^n = z^n$ has no integer solution for $x, y, z \in \mathbb{N}$ and $n \geq 3$*

Corollary. *This example program is not a "Hello, World!" program.*

There is no simple trick to avoid having to test infinitely many combinations.

Definition. *("Hello, World!" Problem) We denote a program for the "Hello, World!" problem as a "Hello, World!" tester. This tester will take a program P as input, and output "yes" if P is a "Hello, World!" program, and "no" otherwise.*
Input: *Program P*
Question: *Is P a "Hello, World!" program?*

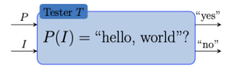
12.1.2 Tester

Theorem. *No "Hello, World!" testers exist*

Definition. *("Hello, World!" Probl. with Input)*
Input*: Program P, input I*
Question*: Does P return "hello, world" for the input I?*

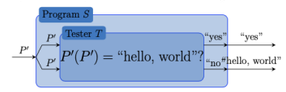
"Proof by Contradiction": "Hello World!" Tester theorem. Notation: Problem P, input I

Assume $\exists$ a tester T

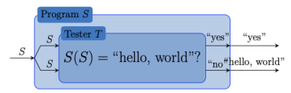


We construct a program S with input P' as follows:

1. Execute T with inputs P' (for P) and P' (for I). We input P' as both the program and input. $P'(P')$
2. Return:
   - "yes" if T returns the outptut "yes"
   - "hello, world", if T returns output "no"



We now consider S run with input S:



Both possible outputs lead to a contradiction of the assumption that T is a tester for the "Hello, World!" problem with input.

We will show using a reduction that $\nexists$ a tester for the "hello world" problem without input.

12.2 First undecidable Problem

Proof is similar to the proof that $\nexists$ program for solving the "hello, world!" problem.

Definition. *(TM-Diag)*
Input: Turing machine M
Question: Does M accept the input enc(M)

Can also be defined as language: TM-Diag = $\{\text{enc}(M) \mid M \text{ accepts the input enc}(M)\}$

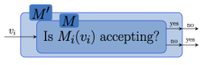
Theorem 12.1. TM-Diag *is not decidable*

Diagonalization

Enumerate all strings over Σ^* with $\Sigma = \{0, 1\}$, thus an enumeration of all Turing machines.

Consider the acceptance and termination behavior of M_i for the input v_j $\forall$ combos of i and j. Assume $\exists$ TM M that decides TM-Diag.

We modify M to M' by inverting the accepting behavior.



Now, there must exist $M_l = M'$ because we enumerated all TMs. Then it should hold:

$$\begin{aligned} & M_l \text{ accepts } v_l \\ \iff\ & M' \text{ accepts } v_l \\ \iff\ & M \text{ rejects } v_l \end{aligned}$$

This is a contradiction because we started by stating that M_l accepts v_l. Thus TM-Diag is not decidable.

12.3 Reductions

Problem A reduces to problem B iff a solver for B can be used to solve problem A.

If A reduces to B and B is "solvable" then A is "solvable"

If A reduces to B and A is "unsolvable" then B is "unsolvable"

Definition. *(Reduction). Let $L, L' \in \Sigma^*$. A total computable function $F = \Sigma^* \to \Sigma^*$ is called a* ***reduction*** *from L to L' if $\forall x \in \Sigma^* : x \in L \implies f(x) \in L'$*

Definition. *(Reducible). Let $L, L' \in \Sigma^*$ L is called* ***reducible*** *to L' if $\exists$ a reduction from L to L'. The notation to indicate this is $L \leq L'$*

Reductions as functions

A reduction from A to B is a function $f : \Sigma_1^* \to \Sigma_2^*$ such that:

1. f is **total** (defined for the entire domain)
2. f is **computable** (in finitely many steps)
3. $x \in L \iff f(x) \in L'$

12.3.1 Reductions & undecidable problems

Reductions can prove that a decision problem A' is undecidable. It is sufficient to show that a known undecidable problem A is reducible to A'.

12.3.2 CFG-Intersection & PCP

Theorem 12.2. PCP $\leq$ CFG-Intersection

12.4 Further undecidable Problems

Theorem 12.3. TM-Halt *is not decidable*

Proof Sketch Show that TM-Diag $\leq$ TM-Halt.

12.4.1 Undecidability of TM-Halt & TM-e-Halt

TM-Halt	*Does M halt for the input x?*
TM-e-Halt	*Does M halt for the input ϵ?*
TM-Diag	*Does M accept the input enc(M)?*

13 Undecidable Problems (2)

13.1 Complements of Decidable Languages

Lemma 13.1. *L is decidable $\implies$ $\bar{L}$ is decidable*
$\bar{L}$ undecidable $\implies$ L is undecidable

13.2 Rice's Theorem

Any non-trivial semantic property of algorithms is undecidable.

Theorem 13.2. *(Rice 53) Let S be a set of computable partial functions with $\emptyset \neq S \neq \mathcal{R}$. Then* TM-Func(S) *is undecidable.*

Proof

1. Define S the subset of computable functions that have the property we want to check e.g. $S = \{f \in \mathcal{R} | \forall x,$ it holds $f(x) = \perp \vee f(x) \neq 101\}$
2. Show that $S \neq \emptyset$ and $S \neq \mathcal{R}$ holds by giving an example.

13.3 From SPCP to PCP

13.4 Undecid. Grammar Problems

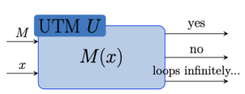
CFG-Intersection	*Is $L(G_1) \cap L(G_2) = \emptyset$?*
CFG-Unamb	*Is G unambiguous?*
CFGEqui	*Is $L(G_1) = L(G_2)$?*
CFGRegEqui	*Is $L(G_1) = L(\alpha)$?*
CFGAll	*Is $L(G) = \Sigma^*$?*
CFGCont	*Is $L(G_1) \subseteq L(G_2)$?*

G: Context-free grammar α: Reg. Expression

14 Variants, Restrictions & Extensions

14.1 Universal Turing Machines

A universal TM U takes as input a TM M, and can then simulate M.



U takes as input the machine M and x, which is the input to M. It computes $f_M(x)$

Binary Encoding for Turing Machines

A standardisation of the encoding of TM.

x	num(x)
$\leftarrow$	1
$\downarrow$	2
$\rightarrow$	3

x	num(x)
$\triangleright$	1
$\sqcup$	2
0	3
1	4

x	num(x)
s	1
yes	2
no	3

states(q)	enc(q)=0$^{\text{num}(q)}$1
symbols (σ)	enc(σ)=0$^{\text{num}(\sigma)}$1
directions (d)	enc(d)=0$^{\text{num}(d)}$1

transitions $\delta(q, \sigma) = (q, \sigma', d)$:
enc$(q, \sigma) = 1$ enc(q) enc(σ) enc(q') enc(σ') enc(d)

enc(M) = concatenation of the strings enc(q, σ)

1 serves as a "stopping" symbol indicating the end of the encoding.

Assigning to eacht 0-1 string w a TM M_w:

- The TM M with enc(M)=w, if $\exists$ such a TM M
- TM M_- which immediately switches to rejecting state by reading left border symbol $\triangleright$, otherwise

input $x = x_1 ... x_m$ is encoded by enc(x) = enc(x_1) ... enc(x_m)

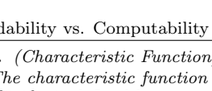
For $y = enc(x)$, we write $x = enc^{-1}(y)$

In a sense, $enc^{-1}(y)$ is a "decoding" function that transforms the encoding back to the original TM.

Existence of the Universal Turing Machine

Definition. *(Universal Turing Machine). TM U is a universal Turing machine if for 0-1-strings w and x the following conditions are satisfied:*

- *M_w accepts input $enc^{-1}(x) \implies U$ accepts input $w\#x$*
- *M_w rejects input $enc^{-1}(x) \implies U$ rejects input $w\#x$*
- *M_w doesn't halt for input $enc^{-1}(x) \implies U$ doesn't halt for input $w\#x$*



Theorem 14.1. *$\exists$ a universal Turing machine*

14.2 Semi-Decidability

Definition. *(Semi-decidability) A set $L \subseteq \Sigma^*$ is called semi-decidable if $\exists$ a TM M with L=$L(M)$*

- Halts and accepts for all
- Does not halt for or rejects all other inputs.

Every decidable language is also semi-decidable.

14.2.1 Examples of Semi-Decidable Languages

CFG-Intersection *Is $L(G_1) \cap L(G_2) \neq \emptyset$?* One can use the CYK algo to test whether $w \in L(G_1)$ & $w \in L(G_2)$

Further exmaples: TM-Diag, TM-Halt, PCP,...

14.2.2 Decidability & Semi-Decidability

Lemma 14.2. *Let L be a language. Then it holds: L decidable $\iff$ L and $\bar{L}$ semi-decidable*

14.2.3 Non Semi-Decidable Problems

Lemma 14.2 can be used to verify that a language is not semi-decidable.

Definition. (CFG-Disjunct)
Is $L(G_1) \cap L(G_2) = \emptyset$?

Lemma 14.3. CFG-Disjunct *is not semi-decidable.*

14.2.4 Semi-Decid. vs. Recursively Enumerable

Definition. *(Recursively Enumerable) A langueage L is recursively enumerable if $\exists$ a 2-tape TM M that progressively writes all element of L onto its second tape.*

Lemma 14.4. *L is recursively enumerable $\iff$ L is semi-decidable*

14.2.5 Decidability vs. Computability

Definition. *(Characteristic Function) Let L be a language. The characteristic function of L, $\chi_L : \Sigma^* \to \{0, 1\}$, is defined by:*

$$\chi_L(w) = \begin{cases} 1 & w \in L \\ 0 & w \notin L \end{cases}$$

The partial characteristic function of L, $\chi'_L : \Sigma^* \to \{0, 1\}$, is defined by:

$$\chi_L(w) = \begin{cases} 1 & w \in L \\ \perp & w \notin L \end{cases}$$

Lemma 14.5. *L decidable $\iff$ χ_L computable*
L semi-decidable $\iff$ χ'_L computable

15 Polynomial Time

Definition. MinSpanningTreeO
Input: Undirected Graph G=(V,E), weighting function l:$E \to \mathcal{N}$
Question: Spanning tree $T \subseteq E$ of G with minimal total weight

15.1 Prim's Algorithm

To calculate min spanning tree. Start at any node. Always choose the adjacent edge with the least weight and add it to the graph.

15.1.1 Comft. Bridge Building & Min Cycles

Definition. MinCycleO
Input: Undirected Graph G=(V,E), distance function l:$E \to \mathcal{N}$
Desired output: Cycle $K \subseteq E$ through all nodes with minimal total weight. If $\nexists$ such a cycle, the value should be ∞.

15.1.2 Minimal Cycles: Algo

Best algo has a complexty of $> n!$

15.2 Comp. Cost & Compl. Theory

15.2.1 Complexity

In this lecture measured in runtime

15.2.2 Asymptotic Worst-case Complexity

complexity: function of the input size.

15.2.3 Purpose of Complexity Theory

Problems are categorized according to their principal algorithmic difficulty into "efficiently solvable" and "not efficiently solvable"

Complexity classes

1. Mode of computation (deterministic, probabilistic, parallel etc.)
2. Type of considered resource (runtime, memory space...)
3. Growth behavior (log, exp..)

$P \neq NP$?

In polynoml time, can more problems be solved nondeterministically (NP) than determin. (P)

15.3 Efficiently solvable Decision Problems

The runtime is quantified as the number of steps of a computation. A single step composed of several operations (reading tape, change state..)

Definition. *(Runtime of TMs) If $K_0 \vdash_M K_1 \vdash_M K_2 \cdots \vdash_M K_m$ is a computation of a TM M for the input x and K_m a halting configuration, then we define:*
$t_M(x) = m$: runtime of M for the input x
If $\nexists$ such finite sequence: $t_M(x) = \perp$

Definition. *(Input size): len(input string)*

Definition. *(Time Bound). A function $T : \mathcal{N} \to \mathbb{R}$ is called a time bound for a TM M if $\exists$ an $n_0 \in \mathcal{N}$ such that $\forall x \in \Sigma^*$ with $|x| > n_0 : t_M(x) \leq T(|x|)$*

Definition. (TIME (T), P).*For $T : \mathcal{N} \to \mathbb{R}$ let* TIME *(T) be the set of all languages L for which $\exists$ a k-tape TM M with time bound T such that L=$L(M)$*
$P = \bigcup_{p \text{ polynomial}} TIME(p)$

15.3.2 Problems Not solvable in Polynomial Time

Definition. *(EXPTIME):* $\bigcup_{p \text{ polynomial}} TIME(2^p)$

15.3.3 Polynomial Time Bounds. n^k

Simplified expression for complexity

15.4 Optimization Problems vs. Decision Problems

Optimisation problems can be expressed as decision problems.

15.4.1 Travelling Salesperson Probl.

Shortest round trip through a set of cities visiting each city exactly once.

Definition. *(TSP - decision problem).*
Input: Distance function d, target value $k \in \mathcal{N}$
Question: $\exists$ TSP route f for d with dist(f) $\leq k$?

Definition. *(TSPO - optimisation problem).*
Input: Distance function d
Desired output: TSP route f for d with minimal total distance

Definition. *(TSPV - value problem).*
Input: Distance function d
Desired output: Distance dist(f) of a TSP route f for d with minimal total distance

Proof Sketch. Let d be a distance function for TSPV with n cities. It is then possible to use a binary search algorithm to find the value of the minimal distance. The TSP optimization problem can thus be reduced in polynomial time to the TSP decision problem.

Lemma 15.1. *If TSP is solvable in polynomial time, then TSPV is as well. If TSPV is solvable in polynomial time, then TSPO is as well.*

16 NP-Completeness

16.1 Examples of Difficult Computational Problems

Definition. (Knapsack).
Input: Weight limit W, desired Value V, m objects with values v nd weights w.
Question: Is there $I \subset \{1, \ldots, m\}$ such that $\Sigma_{i \in I} v_i \geq V$ and $\Sigma_{i \in I} w_i \geq W$?

Definition. (Col).
Input: Undirected graph G, number k
Question: Can the nodes of G be properly colored with k colors, i.e. such that the nodes connected by edges have different colors?

16.1.3 Graph Theory Refresher

A **walk** is a sequence of consecutive edges. Nodes and edges can be repeated.

A **path** is a walk that does not go through the same node more than once. Note that a path can start and finish in the same node.

A **closed walk** is a walk that starts and ends in the same node.

Definition. (EulerCycle)
Input: Undirected graph G
*Question: $\exists$ a closed walk which visits **each edge** exactly once?*

Theorem. *A connected graph G has an Euler cycle iff each node has even degree (number of adjacent nodes).*

Definition. (HamiltonianCycle)
Input: Undirected graph G
*Question: $\exists$ a closed walk which visits **each node** exactly once?*

16.1.6 Clique Problem

Definition. *(Clique). Two nodes u, v of an undirected graph $G = (V, E)$ are called adjacent if $(u, v) \in E$. A k-clique is a set C of k nodes which are pairwise adjacent.*

Definition. (Clique).
Input: Graph $G = (V,E)$, number k
Question: $\exists$ clique G with k nodes?

16.1.7 Prop. Logic Satisfiability (SAT, 3-SAT)

Definition. *(Propositional Logic Formula). Let $x_1, x_2, x_3, \ldots$ be variables*

- *Each x_i is a propositional logic formula*
- *If ϕ is such a formula, then $\neg\phi$ is as well.*
- *If ϕ_1, ϕ_2 are such formulas, then $\phi_1 \wedge \phi_2$ and $\phi_1 \vee \phi_2$ are too.*
- *A truth assignment $\alpha : \{x_1, x_2, ...\} \to \{0, 1\}$ assigns 0 or 1 to each variable.*
- *$\alpha \models \phi$ ("α models ϕ"): The formula ϕ evaluates to q under the assignment α*
- *A formula ϕ is satisfiable if $\exists \alpha$ with $\alpha \models \phi$*

16.2 NP

Stands for non-deterministically polynomial. It refers to problems for which checking the correctness of a proof candidate takes polynomial time.

Known algo $\in P$: EulerCycle; MinSpanTreeO

No known algo $\in P$: TSP; HamiltonianCycle; SAT; Col; Knapsack; 3-SAT, Clique

The NP have in common that they can be solved by "brute force".

Properties

1. They have a search space of proof candidates.
2. The proof candidates are polynomially large in the input.
3. Each proof candidate can be checked in polynomial time.

(note that we do not need to come up with the proof candidate)

For all these problems, there exists for each input a set of **proof candidates** e.g. for Col: all colorings with k colors.

16.2.1 Formalizing the definition of NP

Definition. *(Non-deterministically Accepting/Deciding). Let $\Sigma = \{0, 1\}$*

- *M **non-deterministically accepts** a string $w \in \Sigma^*$ if $\exists$ string $y \in \Sigma^*$ s.t. M (deterministically accepts the input (w,y)*
- *The string y is called the extra input (proof candidate)*
- *M **non-deterministically decides** a language $L \subseteq \Sigma^*$ if $\forall$ strings $w \in \Sigma^*$:*
  - *$w \in L \Rightarrow M$ accepts w non-deterministically $\exists$ y s.t. M accepts (w,y)*
  - *$w \notin L$: M does not accept w non-deterministically $\Rightarrow \nexists$ y s.t. M accepts (w,y)*

We denote the computation time of M for the input (w, y) by $t_M(w, y)$

Definition. (NTIME(T),NP) *Let $\Sigma = \{0, 1\}, T : \mathcal{N} \to \mathbb{R}$*

- (NTIME(T)= *class of all $L \subseteq \Sigma^*$ $\exists$ TM M which decides L non-deterministically s.t. $\forall$ $w,y, \in \Sigma^* : t_M(w, y) \leq T(|w|)$*
- $NP = \bigcup_{p \text{ polynomial}} NTIME(p)$

Proposition 16.1. $P \subseteq NP \subseteq EXPTIME$

16.3 NP-Completeness

16.3.1 Similarity of Difficult Decision Problems

They can all be linked together by polynomial reductions. This means that either they can all be solved in polynomial time or none of them can.

16.3.2 Polynomial Reductions

Definition. *(Polynomial Reduction $\leq_p$). A total function f from L to L' is a polynomial reduction if:*

1. *f is a reduction from L to L', i.e. $\forall w \in \Sigma^*$: $w \in L \iff f(w) \in L'$*
2. *f is computable in polynomial time*

L is polynomially reducible: $L \leq_p L'$

16.3.3 NP-hard Problems

Definition. *(NP-hard). A language L is called NP-hard if $\forall L' \in NP : L' \leq_p L$*

A problem is NP-hard if it is at least as hard as other problems in NP.

Proposition 16.2. TM-Halt *is NP-hard*

Any language $L \in NP$ can be reduced in polynomial time to TM-Halt

Definition. *(NP-complete). A langueage L is called NP-complete if L is NP-hard and $L \in NP$*

Proposition 16.3. *Let $L, L' \subseteq \Sigma^*$ with $L \leq_p L'$:*

1. *$L' \in P \implies L \in P$*
2. *$L' \in NP \implies L \in NP$*

Corollary 16.0.1. *Let $L, L' \subseteq \Sigma^*$ and let $L \leq_p L'$. If $L \notin P \implies L' \notin P$*

Theorem 16.5. *Let L be an NP-complete problem.*

- *If $L \in P \implies P = NP$*
- *If $L \notin P \implies P \neq NP$*

16.4 Cook-Levin Theorem

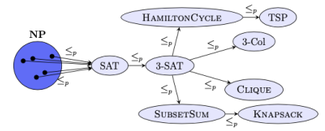
Prove that at least one problem is NP-complete.

Theorem 16.6. *(cook 71, Levin 73). SAT is NP-complete.*

17 NP-Complete Problems

Use polynomial reductions to show that other problems are also NP-complete.

17.1 Polynomial Reductions



Theorem 17.1. *The following problems are NP-complete:* 3-SAT; 3-Col; Clique; HamiltonianCycle; TSP; SubsetSum; Knapsack

Classify a language at first sight

Look at the exponents of the different alphabets. Is there any relation between the different exponents?

- **No**. The language is **regular** and **context-free**. EG: $L = a^m b^n \mid m, n \geq 0$
- **Yes, one** eg: $L = a^m b^n \mid m > n, m \neq n$ These languages are not regular, but are context-free and accepted by a PDA.
- **Yes, multiple** not regular and not context-free