IRZ definicije

Čipson

January 21, 2021

Contents

1 Preliminaries

A *model of computation* is a formal definition of the basic notions of algorithmic computation. It rigorously defines

- what is meant by the notion of the algorithm,
- what is the environment required to execute the algorithm,
- how the algorithm executes in this environment.

2 Finite Automata and Regular Expressions

A *deterministic finite automaton* (DFA) is a 5-tuple $(Q,\sum,\delta,q_0,F)$ where:

- Q is a finite set of states,
- $\sum$ is a finite input alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states, and
- δ is the transition function, i.e. $\delta : Q \times \Sigma \to Q$.

That is, $\delta(q, a)$ is a state (for each state q and input symbol a).
Note: δ is the program of DFA. Every DFA has its own, specific δ.

A string x is said to be *accepted by a DFA* $M = (Q, \sum, \delta, q_0, F)$ if $\delta(q_0, x) = p$ for some $p \in F$.

The *language accepted by a DFA* M is defined as the set $L(M) = x \in \sum^* | \delta(q_0, x) \in F$

A language L' is said to be a *regular set* (or just regular) if L' is accepted by some DFA (if L' is accepted by some DFA (i.e. if $\exists$ DFA M : L' = L(M))).

2.1 Nondeterminism

Question: Given an input word $a_1a_2 \dots a_n$, who decides whether or not there exists a sequence of transitions leading from initial to some final state?
Answer: NFA itself!

Question: How does NFA do that?
Answer: The NFA is not a realistic model of computation: it is assumed that NFA can always guess right. That is, it is assumed that NFA has the magic capability of choosing, from any given set of options, the right option, i.e. the option that leads to a success (if such an option exists; otherwise, NFA halts).
In particular, if there are several transitions from a state on the same input symbol, the NFA can immediately choose the one (if there is such) which eventually leads to some final state.

Question: If NFA is unrealistic, who needs it?
Answers:

- NFA (and other nondeterministic models that we will see later) can be used to find lower bounds on the time required to solve computational problems. The reasoning is as follows: If a problem P requires time T to be solved by a nondeterministic model M, then solving this problem on any deterministic version D of the model M must require at least time T (because D lacks the ability of prediction).
- Often it is much easier to design a NFA (or some other nondeterministic model) for a given problem P. We then try to construct an equivalent deterministic version (equivalent in the sense that it solves P too, regardless of the time needed).

2.1.1 Nondeterministic finite automaton

A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \sum, \delta, q_0, F)$ where:

- Q is a finite set of states,
- $\sum$ is a finite input alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states and
- δ is the transition function, i.e. $\delta : Q \times \sum \to 2^Q$
  That is, $\delta(q, a)$ is the set of all states p, such that there is a transition labeled a from q to p.

Note: δ is the program of NFA. Every NFA has its own specific δ.

A string x is said to be *accepted by a NFA* $M = (Q, \sum, \delta, q_0, F)$ if $\delta(q_0, x)$ contains some $p \in F$ (i.e.,$\delta(q_0, x) \cap F \neq \emptyset$).

The *language accepted by a NFA* $M = (Q, \sum, \delta, q_0, F)$ is the set
$L(M) = \{x \in \sum^* \mid \delta(q_0, x)$ contains a state in $F\}$

Every DFA is also NFA.

Let L be a set accepted by a NFA M. Then there exists a DFA M' that accepts L.

2.1.2 Nondeterministic finite automaton with ε moves

A *NFA with ε-moves* (NFA_ε) is a 5-tuple $(Q, \sum, \delta, q_0, F)$ where:

- Q is a finite set of states,
- $\sum$ is a finite input alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states and
- δ is the transition function, i.e. $\delta : Q \times (\sum \cup \{\varepsilon\}) \to 2^Q$
  That is, $\delta(q, a)$ is the set of all states p, such that there is a transition labeled a from q to p, where a is either a sybol in $\sum$ or ε.

A string x is said to be *accepted by a* NFA_ε $M = (Q, \sum, \delta, q_0, F)$ if the set
$L(M) = \{x \in \sum^* \mid \hat{\delta}(q_0, x)$ contains a state in $F\}$

Let L be a set accepted by a NFA_ε M. Then there exists a DFA M' that accepts L.

The set of all states reachable from the state q with ε-transitions only: ε-Closure(q).

Let $\sum$ be an alphabet. Let L_1 and L_2 be sets of words from $\sum^*$. The concatenation of L_1 and L_2, denoted L_1L_2, is the set
$L_1L_2 = \{xy \mid x \in L_1$ and y $\in L_2\}$ Words in L_1L_2 are formed by taking an x in L_1 and following it by a y in L_2, for all possible x, y.

Let $L \subseteq \sum^*$. Define $L^0 = \{\varepsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The *Kleene closure* (in short closure) of L, denoted L^*, is the set

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

and the *positive closure* of L, denoted L^+, is the set

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

L^* is the set of words that are constructed by concatenating *any number* of words from L.
L^+ is the same, but the case of *zero* words (whose concatenation is defined to be ε), is excluded.
Note: L^+ contains ε *iff* L contains ε.

2.1.3 Regular expressions

Let $\sum$ be alphabet. The *regular expressions* (r.e.) over $\sum$ (and the sets that they denote) are defined inductively as follows:

1. $\emptyset$ is a r.e.; it denotes the *empty set*, $\emptyset$;
2. ε is a r.r.; it denotes the set $\{\varepsilon\}$;
3. For each $a \in \sum$, a is a r.e.; it denotes the set {a};
4. If r and s are r.e.s denoting languages R and S, respectively, then
   - $(r + s)$ is a r.e.; it denotes the set $R \cup S$; (union of R and S)
   - (rs) is a r.e.; it denotes the set RS; (concatenation of R and S)
   - (r^*) is a r.e.; it denotes the set R^*. (Kleene closure of R)

Note. The basic r.e.s are defined explicitly (1,2,3). All the other r.e.s are defined inductively (4a,b,c). Definitions of this kind are called inductive. Properties of the defined objects are often proved by induction.

Conventions

1. We can omit many parentheses
   - if we assume that * has higher precedence than concatenation and concatenation has higher precedence than +.
   - if we abbreviate the expression rr^* by r^+.
2. When
   - necessary to distinguish between a regular expression r and the language denoted by r, we use $L(r)$ for the latter;
   - no confusion is possible we use r for both the regular expression and the language denoted by the regular expression.

Let r be an arbitrary r.e. Then there exists an NFA$_\varepsilon$ that accepts $L(r)$.
Proof idea: We use induction on the number of operators in r to show that, for any r.e. r, there exists an NFA$_\varepsilon$ $M = (Q, \sum, \delta, q_0, \{f_0\})$ with one final state and no transitions out of it, such that $L(M) = L(r)$.
Note: NFA$_\varepsilon$s with just one final state will enable us to easily combine them into larger NFA$_\varepsilon$s. No generality will be lost in this way. (Why? Show how an arbitrary general NFA can be transformed into such equivalent NFA$_\varepsilon$.)

Let M be an arbitrary DFA. There exists a r.e. that denotes $L(M)$.
Proof idea:

- We view $L(M)$ as a union of finitely many sets.
- Each of the sets corresponds to a final state of M and contains exactly the words that take M from its initial state to this final state.
- We then define these sets inductively (bottom up, by simpler sets). Simultaneously we construct to each such set the corresponding r.e.

3 Properties of regular sets

3.1 Pumping lemma

Let L be a regular set. Then there is a constant n (that only depends on L) such that the following holds:
if z is any word such that

$z \in L$ and $|z| \geq n$ (obstaja beseda z, ki je daljša od n)

then there exist words u, v, w such that

$z = uvw$, (sestavljena iz u,v,w)

$|uv| \leq n$,

$|v| \geq 1$, (sredinja beseda dolga vsaj 1)

$\forall i \geq 0 : uv^iw \in L$. (to besedo lahko poljubno ponavljamo)

In addition, n is at most the number of states of the smallest FA accepting L.

Informally:
Given any sufficiently long word z accepted by an FA, we can find a subword v (near the beginning of z) that may be repeated ("pumped") as many times as we like but the resulting word will still be accepted by the FA.

Applications - from p.101
Proving some languages are not regular
$(\exists z)(\forall u, v, w)(\exists i \geq 0)uv^i w \notin L \Rightarrow L$ not regular
(for 'good' n,z,u,v,w)
: If we prove, for a given L, that the left-hand side of '$\Rightarrow$' holds, then L is not regular.
There exist non-regular languages! For these we will need a model of computation that is more powerful than FA.

3.2 Closure under union, concatenation, and Kleene closure

The class of regular sets is closed under union, concatenation, and Kleene closure.
Remark: So the union $L_1 \cup L_2$ and concatenation $L_1 L_2$ of regular sets L_1, L_2 is a regular set, and the Kleene closure L^* of a regular set L is a regular set.

3.3 Closure under complementation and intersection

The class of regular sets is closed under complementation and intersection.
Remark: The theorem states that the complement $\sum^* - L$ of a regular set L is a regular set, and the intersection $L_1 \cap L_2$ of regular sets L_1, L_2 is a regular set.

3.4 Closure under substitution and homomorphism

Let Σ, Δ be alphabets. A *substitution* is a function f that maps each symbol of Σ to a language over Δ; i.e. $f(a) \subseteq \Delta^*$ for each a $\in \Sigma$. We extend f to words in Σ^* by defining $f(\varepsilon) = \varepsilon$ and $f(wa) = f(w)f(a)$; and then to languages by defining

$$f(L) = \bigcup_{x \in L} f(x)$$

The definition of substitution says nothing about the kind of the set L and the sets $f(a)$, $a \in \Sigma$. What if we additionally require that L and all $f(a)$, $a \in \Sigma$ are regular? Is then $f(L)$ regular too?

A homomorphism is a substitution h such that $h(a)$ contains a *single word* for each $a \in \Sigma$. We extend h to words and languages as in the case of substitution. The *inverse homomorphic image* of a word w is the set $h^{-1}(w) = \{x \mid h(x) = w\}$ and of a language L is the set $h^{-1}(L) = \{x \mid h(x) \in L\}$.

The class of regular sets is closed under substitution, homomorphism and inverse homomorphism.
Proof idea:

- (*substitution*) Let L and all $f(a)$, $a \in \Sigma$ be regular sets. Let L be denoted by r.e. r and $f(a)$ by r_a.
  Idea: replace each occurence of a by r by r_a. Then prove that the resulting r.e. r' denotes $f(L)$.
  (Use induction on the number of operators in r'.)

- (*homomorphism*) Closure under homomorphism follows directly from closure under substitution
  (because every homomorphism is by definition a (special) substitution)

- (*inverse homomorphism*) Let L be regular and h a homomorphism. We want to prove that $h^{-1}(L)$ is regular. Let M be DFA accepting L. We want to construct a DFA M' such that M' accepts $h^{-1}(L)$ *iff* M accepts L.
  Idea: construct M' so that when M' reads $a \in \Delta$, it simulates M on $h^{-1}(L)$.

3.5 Closure under quotient

The *quotient* of languages L_1 and L_2 is the set L_1/L_2 defined by
$L_1/L_2 = \{x \mid \exists y \in L_2 : xy \in L_1\}$.
Informally: L_1/L_2 contains prefixes of words in L_1 whose corresponding suffixes are words in L_2.
The class of regular sets is closed under quotient with arbitrary sets.

4 Decision algorithms for regular sets

4.1 Emptiness and finiteness of regular sets.

The set $L(M)$ accepted by a FA M with n states is:

1. *nonempty iff* M accepts a word of length l, where $l < n$.
2. *infinite iff* M accpets a word of length l, where $n \leq l < 2n$

Naive algorithms for both could systematically generate all words of appropriate lengths l and, for each generated word, check whether M accepts that word.
Both algorithms eventually halt and return a YES or NO.

4.2 Equivalence of finite automata.

Two finite automata M_1 and M_2 are said to be equivalent if they accept the same language, i.e. if $L(M_1) = L(M_2)$.
There exists an algorithm to decide whether two FAs are *equivalent*.
Proff:
Let M_1 and M_2 be FAs and $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Define a language L_3 as follows:

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

L_3 is regular (due to closure properties) and therefore accepted by some FA M_3. This M_3 is important because we can show that

$$M_3 \text{ accepts a word } \textit{iff } L_1 \neq L_2.$$

So we need to check whether M_3 accepts any word, i.e. whether L_3 is non-empty.

5 The Myhill-Nerode theorem and minimization of FA

Let $L \subseteq \Sigma^*$ be an arbitrary language. Define a relation R_L on Σ^* by

$$xR_Ly \text{ iff } \forall z \in \Sigma^* : xz \in L \Leftrightarrow yz \in L.$$

Remarks: Two words $x, y \in \Sigma^*$ are in relation R_L *iff* their arbitrary extensions xz,yz are either both in L or both outside L. Now, R_L is an *equivalence relation*. So, R_L partitions L into *equivalence classes*. The number of these is called the index of R_L and it can be finite or infinite.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Define a relation R_M on Σ^* by

$$xR_My \textit{ iff } \delta(q_0, x) = \delta(q_0, y).$$

Remarks: Two words $x, y \in \Sigma^*$ are in relation R_M *iff* they take M from q_0 to the same state q. R_M is *equivalence relation*. It partitions Σ^* into *equivalence classes, one for each state q reachable from* q_0. The number of the classes is the *index of* R_M. The index of R_M is finite (since Q is finite). $L(M)$ is the union of some equivalence classes (that correspond to final states $q \in F$).

We can prove that R_M is right invariant, i.e. that

$$xR_My \Rightarrow \forall z \in \Sigma^* : xzR_Myz$$

(Myhill-Nerode) The following statements are *equivalent*:

1. $L \subseteq \sigma^*$ is a regular set;
2. R_L is of finite index;
3. L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index.

Remarks: The theorem is useful when, for a given L, we have proved one of the items (1,2,3). Then by Myhill-Nerode theorem, the other two items hold too, and so reveal additional information about L.

A consequence of the Myhill-Nerode Theorem is that for every regular set there is an essentially unique *minimum state* DFA.

The minimum state DFA accepting a regular set L is *unique up to an isomorphism* (renaming of the states).

Proof idea:

- Let L be regular. By Myhill-Nerode theorem there are finitely many equivalence classes of R_L. Denote by $[x]$ the eq. class containing $x \in \Sigma^*$. So $\{[x] \mid x \in \Sigma^*\}$ is the set of all eq. classes of R_L.
- Construct a DFA $M = (Q, \Sigma, \delta, q_0, F)$ as follows:

  $Q := \{[x] \mid x \in \Sigma^*\}$; (each state will correspond to an eq. class of R_L)

  $\delta([x], a) := [xa]$, for $a \in \Sigma$

  $q_0 := [\varepsilon]$

$$F := \{[x] \mid x \in L\}$$

- *Note:* $\delta(q_0, w) = \delta(q_0, a_1a_2 \dots a_n) = [a_1a_2 \dots a_n] = [w]$.
  Thus, M accepts w *iff* $[w] \in F$. This means that M accepts L.
- It follows from the proof of Myhill-Nerode theorem that this M is the minimum state DFA for L.

6 Context-Free grammars and languages

A *context-free grammar* (CFG) is a 4-tuple $G = (V, T, P, S)$ where:

- V is a finite set of variables
- T is a finite set of terminals
- P is a finite set of productions, each of which is of the form $A \Rightarrow \alpha$, where $A \in V$ and α is a word in the language $(V \cup T)^*$;
- S is a special variable called the *start symbol*.

Let $A \to \beta$ be a production and $\alpha, \gamma \in (V \cup T)^*$ arbitrary string.

- We say that we apply the production $A \to \beta$ to $\alpha A \gamma$ and obtain $\alpha\beta\gamma$ if we substitute A by β in $\alpha A \gamma$. In this case we say that $\alpha A \gamma$ directly derives $\alpha\beta\gamma$ by the production $A \to \beta$.
- We say that two strings are in the relation ${}_G\Rightarrow$ if the 1^{st} directly derives the 2^{nd} one by one application of a production in G.
- Let $\alpha_1, \alpha_2, \dots, \alpha_m \in (V \cup T)^*, m \geq 1$, be strings.
  If $\alpha_1 \; {}_G\Rightarrow \alpha_2 \wedge \alpha_2 \; {}_G\Rightarrow \alpha_3 \wedge \dots \wedge \alpha_{m-1} \; {}_G\Rightarrow^* \alpha_m$
  then we say that α_1 derives α_m in G and denote this fact by $\alpha_1 \; {}_G\Rightarrow^* \alpha_m$.
  Note: The relation ${}_G\Rightarrow^*$ is reflexive and transitive closure on the relation ${}_G\Rightarrow$.

The *language generated* by a CFG $G = (V, T, P, S)$ is the set
$$L(G) = \{w \mid w \in T^* \wedge S \; {}_G\Rightarrow^* w\}.$$
So the language generated by G is the set of all terminal strings that can be derived from S.

A language L is called *context-free* (CFL) if $L = L(G)$ form some CFG G.

A string $\alpha \in (V \cup T)^*$ is called a *sentential form* if $S \; {}_G\Rightarrow^* \alpha$.

Two grammars G_1 and G_2 are said to be *equivalent* if $L(G_1) = L(G_2)$.

6.1 Derivation trees

Let $G = (V, T, P, S)$ be a CFG. A tree is called a *derivation* (or parse) tree for G if:

1. Every vertex **v** has a label that is a symbol in $V \cup T \cup \{\varepsilon\}$.
2. The label of the *root* is S.
3. If a vertex **v** is *interior* and has label A, then A must be in V.

4. If a vertex **v** has label A and vertices v_1,v_2,... ,v_k are the *sons* of v (from left to right) with labels $X_1, X_2, \ldots, X_k$, respectively, then $A \to X_1X_2 \ldots X_k$ must be a *production* in P.
5. If vertex **v** has label ε, then **v** is a *leaf* and is the only son of its father.

If we read the labels of all *leaves* visited by the *preorder traversal* of the tree we obtain a string that is called the *yield* of the *derivation tree*.

A *subtree* of a derivation tree is a particular vertex of the tree together with all of its descendants, edges among them, and their labels. If the root of a subtree is labeled A, then the subtree is called A-tree.
A subtree is just like a derivation tree, but the label of its root may not be the start symbol S of the grammar.

6.1.1 The relationship between derivation trees and derivations

Let $G = (V, T, P, S)$ be a CFG. Them $S \underset{G}{\Rightarrow}^* \alpha$ *iff* there is a derivation tree for G with yield α.
Proof idea: Induction on the number of interior vertices of the tree.

6.1.2 Leftmost and righmost derivations

A derivation is said to be *leftmost* if at *each step* of the derivation a production is applied to the *leftmost variable*. Similarly, a derivation is *rightmost* if at *each step* a production is applied to the *rightmost* variable.

6.1.3 Ambiguity

A CFG G is said to be *ambiguous* if some word has more than one derivation tree.
Equivalently: A CFG is ambiguous if some word has more than one leftmost (rightmost) derivation.

A CFL L is *inherently ambiguous* if every CFG for L is ambiguous.

6.2 Simplification of context-free grammars

There are several ways to *restrict the form of productions without reducing the power* of CFGs. If L is a nonempty CFL then L can be generated by a CFG G having the following properties:

- Each variable and terminal of G appears in the derivation of some word in L.
- There are no productions of the form $A \to \varepsilon$.
- If $\varepsilon \notin L$, there are no productions of the form $A \to \varepsilon$.
- If $\varepsilon \notin L$, we can require that

  every production is of the form $A \to BC$ or $A \to b$
  (Chomsky normal form) where A, B, C are variables and b is a terminal

  or, every production is of the form $A \to b\gamma$
  (Greibach normal form) where $b \in T$ and $\gamma \in V^*$ (a string of variables).

6.2.1 Elimination of useless symbols

Let $G = (V, T, P, S)$ be a grammar. A symbol X is useful if there exists a derivation $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ for some α, β, and $w \in T^*$. Otherwise X is useless.
Lemmas:

- Given a CFG $G = (V, T, P, S)$ with $L(G) \neq \emptyset$, we can effectively find an equivalent CFG $G' = (V', T, P', S)$ such that for each $A \in V'$ there is a $w \in T^*$ so that $A \Rightarrow^* w$
- Given a CFG $G' = (V', T, P', S)$, we can effectively find an equivalent CFG $G'' = (V'', T, P'', S)$ such that for each $X \in V'' \cup T$ there are $\alpha, \beta \in (V'' \cup T)^*$ so that $A \Rightarrow^* \alpha X \beta$.

Applying the lemmas *in this order*, we can convert a CFG G to the equivalent G" *without useless symbols*. (Applying the lemmas in the reverse order may fail to eliminate all useless symbols.)

Every nonempty CFL is generated by a CFG with no useless symbols.

6.2.2 Elimination of ε-productions

An ε-production is a production of the form $A \to \varepsilon$.
Clearly, if ε is in $L(G)$, we cannot eliminate all ε-productions from G. (Otherwise, ε would no longer be in the generated language.) But if ε is not in $L(G)$, we can eliminate all ε-productions from G.

If $L = L(G)$ for some CFG $G = (V, T, P, S)$, then $L - \{\varepsilon\}$ can be generated by a CFG G' that has no useless symbols and no ε-productions.
Proof idea:

- Determine for each $A \in V$ whether $A \Rightarrow^* \varepsilon$. If so, call A nullable.
- Then replace each production $B \to X_1 X_2 \dots X_n$ by all productions formed by striking out some subset of those X_i's that are nullable, but do not include $B \to \varepsilon$, even if all X_i's are nullable.

6.2.3 Elimination of unit productions

A unit production is a production of the form $A \to B$.
The right-hand side must be a single variable; all other productions, including $A \to a$ and ε-production, are non-unit.

Every CFL without ε can be generated by a grammar that has no useless symbols, no ε-productions, and no unit productions.

6.2.4 Chomsky normal form

Every CFL without ε can be generated by a grammar in which every production is of the form

$$A \to BC \text{ or } A \to a$$

where A, B, C are variables and a is a terminal.
TLDR "proof" p.142:

> The input CFG should be without useless variables, unit productions and ε-productions
>
> Replace all terminals in productions so they only appear this form: $A \to a$ (one variable $\to$ one terminal)
>
> Break up all productions that include more than 2 (≥ 3) variables into 2 parts. (the maximum amount of variables in a production is 2: $A \to BC$)

6.2.5 Greibach normal form

Every CFL without ε can be generated by a grammar in which every production is of the form.

$$A \to b\gamma$$

where A is a variable, b is a terminal, and γ is a (possibly empty) string of variables ($\gamma \in V^*$). *"proof" p.144*

6.2.6 Inherently ambiguous context-free languages

The CFL $L = \{a^n b^n c^m d^m \mid n, n \geq 1\} \bigcup \{a^n b^m c^m d^n \mid n, m \geq 1\}$ is inherently ambiguous.

7 Pushdown automata

A *pushdown automaton* (PDA) is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $q_0 \in Q$ is the initial state,
- $Z_0 \in \Gamma$ is the start symbol,
- $F \subseteq Q$ is the set of final states and
- δ is the transition function,
  i.e. a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

Note: δ can be viewed as a program of PDA. Every PDA has its own specific δ.

7.1 Moves of the PDA

The interpretation of the move

- $\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$ is that the PDA in state q, with input symbol a and Z the top symbol on the stack can, for any i, $1 \leq i \leq m$, enter state p_i, replace symbol Z by string γ_i, and advance the window one symbol. We call this the *regular move*.

- $\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$ is that the PDA in state q, independently of the input symbol being scanned and with Z the top symbol on the stack, can enter state p_i, and replace Z by γ_i, for any i, $i \leq i \leq m$. In this case, the window is not advanced. We call this the ε-move.

Conventions: the leftmost symbol of γ_i is placed highest on the stack and the rightmost symbol of γ_i lowest on the stack. We use $a, v, c, \ldots$ for input symbols, $u, v, w, \ldots$ for strings of input sybols, captal letters for stack symbols, and Greek letters for string of stack symbols.

7.2 Instantaneous descriptions of the PDA

An instantaneous description (ID) is a tripple (q, w, γ), where q is a state, w a string of input symbols to be read, and γ a string of stack symbols.

- If $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a PDA, we say that ID $(q, ax, Z\beta)$ can directly become $ID(p_i, ax, \gamma_i\beta)$, — written
$$(q, ax, Z\beta) \vdash (p_i, x, \gamma_i\beta),$$
if $\delta(q, a, Z)$ contains (p_i, γ_i). Here, a may be an input symbol or ε.
- We write ${}_M\vdash^*$ for the reflexive and transitive closure of ${}_M\vdash$ and say that an ID I can become ID J if $I {}_M\vdash^* J$. We write $I {}_M\vdash^k J$ if $I {}_M\vdash^* J$ in exactly k moves.

The subscript M can be dropped whenever the particular PDA M is understood.

7.3 Accepted languages of the PDA

For PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ we define two language:

- $L(M)$, the language accepted by final state, to be
$L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (p, \varepsilon, \gamma) \text{ for some } p \in F \text{ and } \gamma \in \Gamma^*\}$
- $N(M)$, the language accepted by empty stack, to be
$N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (p, \varepsilon, \varepsilon) \text{ for some } p \in Q\}$.

$L(M)$ contains a word w if after reading w, M can be (nondeterminism!) in some final state.
$N(M)$ contains a word w if after reading w, M can have (nondeterminism!) its stack empty. If acceptance is by empty stack, final states are irrelevant; in this case, we usually let $F = \emptyset$.

7.4 Deterministic pushdown automata

A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is said to be deterministic if δ fulfills two conditions for every $q \in Q$ and $Z \in \Gamma$:

1. $\delta(q, \varepsilon, Z) \neq \emptyset \Rightarrow \forall a \in \Sigma : \delta(q, a, Z) = \emptyset$
2. $\forall a \in \Sigma \cup \{\varepsilon\} : |\delta(q, a, Z)| \leq 1$

What does that mean? Condition 1 prevents the possibility of a choice between ε-moves and regular moves. Condition 2 prevents the possibility of a choice for ε-moves and the possibility of a choice for regular moves.
Note: Unlike FA, a PDA is assumed to be non-deterministic unless we state otherwise. In this case, we denote it by DPDA (for deterministic PDA).

7.5 Pushdown automata and context-free languages

7.5.1 Equivalence of acceptance by final state and empty stack

If $L = L(M_2)$ for some PDA M_2, then $L = N(M_1)$ for some PDA M_1.
Proof idea: Given an arbitrary $L = L(M_2)$, construct a PDA M_1 that simulates M_2 but erases the stack whenever M_2 enters a final state. So we have $L = N(M_1)$

If $L = L(M_1)$ for some PDA M_1, then $L = N(M_2)$ for some PDA M_2.

Proof idea: Given an arbitrary $L = L(M_1)$, construct a PDA M_2 that simulates M_1 but enters a final state whenever M_1 erases its stack. So we have $L = N(M_2)$

$\Rightarrow$ The class of languages accepted by PDAs by final state is the same as the class of languages accepted by PDAs by empty stack.

7.5.2 Equivalence of PDAs and CFLs

If L is a CFL, then there exists a PDA M sucha that $L = N(M)$. *Proof idea:* Let L be an arbitrary CFL. L can be generated by a CFG G in Greibach normal form. Constructa PDA M that simulates leftmost derivations of G. (It is easier to have M accept by empty stack.) So $L = N(M)$

If $L = N(M)$ for some PDA M, then L is a CFL *Proof idea:* Let M be an arbitrary PDA. Construct a CFG G in such a way that a leftmost derivation in G of a sentence x is a simulation of the PDA M when given the input x. So $L = L(G)$, a CFL.

The class of languages accepted by PDAs is exactly the class of CFLs.

7.5.3 Deterministic vs. nondeterministic PDAs

DPDAs are less powerful than PDAs (they dont accept the same class of languages).

$\{ww^R \mid w \in (0+1)^*\}$ is accepted by a PDA and by no DPDA.

8 Properties of Context-Free Languages

8.1 The Pumping Lemma for CFLs

Let L be a CFL. Then there is a constant n (depending on L only) such that the following holds:
if z is any word such that

$z \in L$ and $|z| \geq n$,

then there exist words u, v, w, x, y such that

$z = uvwxy$,

$|vx| \geq 1$,

$|vwx| \leq n$, and

$\forall i \geq 0 : uv^iwx^iy \in L$

Informally: Given any sufficiently long word z in a CFL L, we can find two short sub-words v and x close together that may be repeated, both the same arbitrary number of times, and the resulting word is still in L.

There exist languages that are not context-free! For such languages we'll need a model of computation more powerful than PDA.

8.2 Closure properties for CFL's

The class of CFLs is closed under

- union,
- concatenation,
- Kleene closure,
- substitution (and hence homomorphism),
- inverse homomorphism

The class of CFLs is not closed under

- intersection (unless $\downarrow$),
- complementation

If L is a CFL and R is a regular set, then $L \cap R$ is a CFL

8.3 Emptiness and finiteness of CFLs

There are decision algorithms to determine whether or not a CFL is:

1. empty;
2. finite.

Proof idea: Let $G = (V, T, P, S)$ be a CFG

- $L(G)$ is nonempty *iff* S generates some string of terminals
- $L(G')$ is finite *iff* the graph has no cycles.
  G' is the Chomsky Normal Form (with no useless symbols) of G

8.4 Membership

The Membership for CFGs is the question "Given a CFG $G = (V, R, P, S)$ and a word $x \in T^*$, is $x \in L(G)$?"
Question: Does there exist a decision algorithm such that, given an arbitrary CFG G and an arbitrary word $x \in T^*$, answers the question "Is x a member of $L(G)$?"
Answer: The answer is YES; there is the following naive algorithm:

1. Convert G to Greibach normal form (GNF) G'
2. If $x = \varepsilon$ then test whether $S \underset{G}{\Rightarrow}^* \varepsilon$ else
   check all possible derivations in G'. A total of $\leq k^{|x|}$ leftmost derivations need to be examined.

The CYK algorithm (Cocke-Younger-Kasami)

- dynamic programming
- $O(n^3)$ time, where $n = |x|$.
- algorithm described *p.181/182*

9 Turing machines

An "algorithm" for solving a problem is a *finite set of instructions* that lead the processor, in a *finite number of steps*, from the *input data* of the problem to the corresponding *solution*.

History lesson *p.187*

To systematically eliminate all possible recipes for solving a problem we need a model of computation:
A *model of computation* is a definition that formally characterizes the basic notions of algorithmic computation, that is, the algorithm, its environment, and the computation.

Several were proposed:

- μ-recursive functions
- general recursive functions
- λ-calculus
- Turing machine
- Post machine
- Markov algorithms

The majority of researchers accepted the Turing machine as the model which most adequately captures the basic concepts of computation.

Moreover, surprisingly, it was soon proved that the models are equivalent in the sense: What can be computed by one can also be computed by the others.

Computability thesis (Church-Turing thesis). The basic intuitive concepts of computing are perfectly formalized as follows:

- "algorithm" is formalized by Turing program
- "computation" is formalized by execution of a Turing program in a Turing machine
- "computable function" is formalized by Turing-computable function

The Computability Thesis established a bridge between our intuitive understanding of the concepts of the "algorithm," "computation," and "computability" on the one hand, and their formal counterparts defined by models of computation on the other. In this way it finally enabled a mathematical treatment of these intuitive concepts.

9.1 The Turing machine model

The basic variant of the Turing machine has the following components: a control unit containing a Turing program; a tape consisting of cells; and a movable window over the tape, which is connected to the control unit. The details are:

- The tape is used for writing and reading the input data, intermediate data, and output data (results). It is divided into equally sized cells, and is potentially infinite in one direction (i.e., it can be extended in that direction with a finite number of cells)

Each cell contains a tape symbol belonging to a tape alphabet $\Gamma = \{z_1, \ldots, z_t\}, t \geq 3$. The symbol z_t is special, for it indicates that a cell is empty; for this reason it is denoted by $\sqcup$ and called the empty space. In addition to $\sqcup$ there are at least two additional symbols: 0 and 1. We will assume that $z_1 = 0$ and $z_2 = 1$.

The input data are contained in the input word. This is a word over an input alphabet Σ, such that $\{0, 1\} \subseteq \Sigma \subseteq \Gamma - \{\sqcup\}$. Initially, all the cells are empty (each contains $\sqcup$) except for the leftmost cells, which contain the input word.

- The control unit is always in some state from a finite set of states $Q = \{q_1, \ldots, q_s\}, s \geq 1$. We call q_1 the initial state. Some states are called final; they are gathered in the set $F \subseteq Q$. All the other states are non-final. If the index of a state is of no importance, we use q_{yes} and q_{no} to refer to any final and non-final state, respectively.

  There is a Turing program (TP) in the control unit. TP directs TM's components. TP is characteristic of the particular TM, i.e., different TMs have different TPs. A TP is a partial function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$, called the transition function.
  Note: The TM is by definition deterministic, having at most one choice for a move in each situation.

  We can view δ as a table $\Delta = Q \times \Gamma$, where

  - $\Delta[q_i, z_r] = (q_j, z_w, D)$ if $\delta(q_i, z_r) = (q_j, z_w, D)$ is an instr. of δ,
  - $\Delta[q_i, z_r] = 0$ if $\delta(q_i, z_r) \uparrow$ (undefined).

  Without loss of generality, we can assume that there is always a transition from a q_{no}, and none from q_{yes}.

- The window can move over any cell. Then, the control unit can read a symbol through the window, and write a symbol through the window, substituting the previous symbol. In one step, the window can only move to the neighboring cell.

- Before the TM is started, the following must take place:

  1. an input word is written to the beginning of the tape
  2. the window is shifted to the beginning of the tape
  3. the control unit is set to the initial state

- From now on the TM operates independently, in a mechanical stepwise fashion as instructed by its TP. If the TM is in a state $q_i \in Q$ and it reads a symbol $z_r \in \Gamma$, then:

  if q_i is a final state, then TM halts

  else, if $\delta(q_i, z_r) \uparrow$ (i.e. TP has no next instruction), then the TM halts

  else, if $\delta(q_i, z_r) \downarrow = (q_j, z_w, D)$, then the TM does does the following:

  1. changes the state to q_j
  2. writes z_w through the window
  3. moves the window to the next cell in the direction $D \in \{L, R\}$ (for left and right), or leaves the window where it is ($D = S$, for stay)

Formally, a TM is a seven-tuple $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$. To fix a particular TM, we must fix $Q, \Sigma, \Gamma, \delta, F$.

An instantaneous description (ID) of a TM is the string $I = \alpha_1 q \alpha_2$, if the current configuration of the TM has the window over the first symbol of α_2 and α_2 ends at the rightmost non-blank symbol.
An ID is the "snapshot" of a current configuration (status) of TM's components between successive instructions.

An ID I can directly change to J – written $I \vdash J$ – if there is an instruction in TM's program whose execution changes I to J. The reflexive and transitive closure of $\vdash$ is $\vdash^*$; if $I \vdash^* J$, then we say that ID I can change to J.

9.2 Use of a turing machine

There are three elementary tasks where TMs are used:

- Function computation
  "Given a function φ and arguments $a_1, \ldots, a_k$, compute $\varphi(a_1, \ldots, a_k)$."
- Set recognition
  "Given a set S and an object x, decide whether or not $x \in S$."
- Set generation
  "Given a set S, generate a list $x_1, x_2, x_3, \ldots$ of exactly the members of S."

9.2.1 Function computation on TMs

Each TM T induces, for any $k \geq 1$, a function φ_T that maps k words into 1 word. We define φ_T as follows.
Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM and $k \geq 1$. The k-ary proper function of T is a partial function $\varphi_T : (\Sigma^*)^k \to \Sigma^*$, defined as follows:
If the input to T is k words $u_1, \ldots, u_k \in \Sigma^*$, then the value of φ_T at $u_1, \ldots, u_k$ is defined to be

$$\varphi(u_1, \ldots, u_k) := \begin{cases} v, & \text{if } T \text{ halts} \wedge \text{returns on the tape the word } v \wedge v \in \Sigma^*; \\ \uparrow, & \text{if } T \text{ doesn't halt} \vee \text{the tape doesn't have a word in } \Sigma^*. \end{cases}$$

The interpretation of $u_1, \ldots, u_k$ and v is arbitrary.

In practice, however, we usually face the opposite task:
"Given a k-ary function $\varphi : (\Sigma^*)^k \to \Sigma^*$, find a TM T that can compute φ's values."
That is, given φ, we must construct a TM T such that $\varphi_T = \varphi$.

Let $\varphi : (\Sigma^*)^k \to \Sigma^*$ be a function. Then:

- φ is computable if $\exists$TM that can compute φ anywhere on $dom(\varphi) \wedge dom(\varphi) = (\Sigma^*)^k$;
- φ partial computable (p.c.) if $\exists$TM that can compute φ anywhere on $dom(\varphi)$;
- φ incomputable if there is no TM that can compute φ anywhere on $dom(\varphi)$.

9.2.2 Set recognition on TMs

Each TM T induces a language $L(T)$, the language accepted by T.

Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM and $w \in \Sigma^*$ a string. We say that w is accepted by T if $q_1 w \vdash^* \alpha_1 p \alpha_2$, for some $p \in F$ and $\alpha_1 \alpha_2 \in \Gamma^*$. The language accepted by T is the set $L(T) = \{w \mid w \in \Sigma \wedge w \text{ is accepted by } T\}$.
So, a word is accepted by T if it causes T to enter a final state (if submitted as input). The language accepted by T consists of exactly such words.

The interpretation of w is arbitrary.

In reality, however, we usually face the opposite task:
"Given a set $S \subseteq \Sigma^*$, find a TM T that accepts S."
That is, given a language (set) S, we must construct a TM T such that $L(T) = S$.

Let $S \subseteq \Sigma^*$ be a language (set). Then:

- S is decidable if $\exists$TM that answers YES/NO to "Is $x \in S$?" for any $x \in \Sigma^*$.
- S is semi-decidable if $\exists$TM that answers YES to "Is $x \in S$?" whenever $x \in S$
- S in undecidable if there is no TM that answers YES/NO to "Is $x \in S$? for any $x \in \Sigma^*$"

It looks that for some sets S we cannot algorithmically decide the question "Is $x \in S$?" Why?

If $S = L(T)$ is semi-decidable and $x \in \Sigma^*$ an arbitrary word, then:

- if x is in S, then T will eventually halt on input x (and accept x).
- But if x is not in S, then T may not halt on input x (and never reject x).

For such an S, as long as T is still running on input x, we cannot tell whether

- T will eventually halt (and accept/reject x) if we let T run long enough, or
- T will run forever.

In other words:

- If, in truth, $x \in S$, then T will (halt and accept x and) answer YES
- It, in truth, $x \notin S$, then
  - T may (halt and reject x and) answer NO; or
  - T may (never halt and) never answer NO.

9.2.3 Set generation on TMs

Let $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a TM. T is called a generator if it writes to its tape, in succession and delimited by #, (some) words from Σ^*. (We assume that # is in $\Gamma - \Sigma$.) The language generated by T is defined to be the set $G(T) = \{w \mid w \in \Sigma^* \wedge T \text{ eventually writes } w \text{ to the tape}\}$.

In practice, we usually face the opposite task:
"Given a set S, generate a list $x_1, x_2, x_3, \ldots$ of exactly the members of S."
That is, given a language (set) S, we must construct TM T such that $G(T) = S$.

Questions: When can the elements of a given set S be generated, i.e. listed in a sequence so that every element of S eventually appears in the sequence? When can such a sequence be algorithmically generated, i.e., by a TM? Can every countable set be algorithmically generated?

9.2.4 C.E. languages (sets)

Suppose that the elements of a given set S can be listed in a sequence so that every element of S eventually appears in the sequence. If x is an arbitrary element of S, then x will eventually appear in the list; it will appear as nth in order, for some $n \in N$. So we can speak of the $1^{st}, 2^{nd}, 3^{rd}, \ldots n^{th}, \ldots$ element of S. Because the elements of S can be enumerated, we say that S is enumerable.

A set S is computably enumerable (c.e.) if $S = G(T)$ for some TM T; that is, if S can be generated by a TM.

A set S is c.e. *iff* S is semi-decidable.

9.3 Modifications of the Turing machine

This holds true for all following modified TMs:
Although V seems to be more powerful than the basic model T, it is not so; T can compute anything that V can compute. We prove this by describing how T can simulate V. (The other way round is obvious as T is a special case of V.)

9.3.1 TM with finite storage

This variant V has in its control unit a finite storage capable of memorizing $k \geq 1$ tape symbols and using them during the computation. The Turing program (TP) is the function $\delta_V : Q \times \Gamma \times \Gamma^k \to Q \times \Gamma \times \{L, R, S\} \times \Gamma^k$.

9.3.2 TM with multiple-track tape

This variant V has the tape divided into $tk \geq 2$ tracks. On each track there are symbols from the alphabet Γ. The window displays tk-tuples of symbols, one symbol for each track. The TP is $\delta_V : Q \times \Gamma^{tk} \to Q \times \Gamma^t k \times \{L, R, S\}$.

9.3.3 TM with two-way infinite tape

This variant V has the tape unbounded in both directions. Formally, the TP is the function $\delta_V : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$.

9.3.4 TM with multiple tapes

This variant V has $tp \geq 2$ unbounded tapes. Each tape has its own window that is independent of other windows. TP is $\delta_V : Q \times \Gamma^{tp} \to Q \times (\Gamma \times \{L, R, S\})^{tp}$.

9.3.5 TM with multidimensional tape

This variant V has a d-dimensional tape, $d \geq 2$. The window can move in d dimensions, i.e., $2d$ directions $L_1, R_1, L_2, R_2, \ldots, L_d, R_d$. The turing program is $\delta_V : Q \times \Gamma \times \{L_1, R_1, L_2, R_2, \ldots, L_d, R_d, S\}$.

9.3.6 TM with nondeterministic program

This variant V has a Turing program δ that assigns to each (q_i, z_r) a finite set of alternative transitions $\{(q_{j1}, z_{w1}, D_1), (q_{j2}, z_{w2}, D_2), \ldots\}$. The machine nondeterministically chooses a transition from the set and performs it.

How does V choose a transition out of the current alternatives?
The following is assumed: the machine *chooses* a transition that leads it to a solution (e.g., to a state q_{yes}), if such transitions exist; otherwise, the machine halts.
The nondeterministic TM is not a reasonable model of computation because it can foretell the future when choosing from alternative transitions. Nevertheless, it is a very useful tool, which makes it possible to define the minimum number of steps needed to compute the solution (when a solution exists). This is important when we investigate the computational complexity of problem solving. We will see that in the following chapters.

9.3.7 The importance of the modifications of the TM

The modifications are useful when we try to prove the existence of a TM for solving a given problem P. Usually, the construction of such a TM is easier if we choose a more versatile modification of TM.

> Sometimes, we can even avoid the complicated construction of the actual TM for solving P.
> How? We do as follows:
>
> 1. We devise an intuitive algorithm A (a "recipe", finite list of instructions) for solving P.
> 2. Then we say: "By the Computability Thesis, there is a TM T that does the same as A." Then, we can refer to this T (as the true algorithm for solving P) and treat it mathematically.

Since none of the modifications is more powerful than the basic TM, this additionally supports our belief that the Computability Thesis is true.

The computations on the modifications of TM can considerably differ in *time* (number of steps) and *space* (number of visited cells). But this will become important only in Computational Complexity Theory (where we will investigate the time and/or space complexity of problem solving).

9.4 Universal turing machine

9.4.1 Coding of TMs

Given a TM $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, we want to encode T, i.e. represent T by a word over some coding alphabet.

How will we encode TM T?

- The coding alphabet will be $\{0, 1\}$.
- We'll only encode δ, but in such a way that Q, Σ, Γ, F which determine the particular T, can be restored (obtained) from the encoded δ. How will we encode TP δ?
  1. If $\delta(q_i, z_j) = (q_k, z_l, D_m)$ is an instruction of δ, then we will encode the instruction by the word
     $$K = 0^i 10^j 10^k 10^l 10^m$$
     where $D_1 = L, D_2 = R, D_3 = S$.
  2. In this way, we will encode each instruction of δ.
  3. From the obtained codes $K_1, K_2, \ldots, K_r$ we will construct the code $\langle\delta\rangle$ of δ:
     $$\langle\delta\rangle = 111K_1 11K_2 11 \ldots 11K_r 111$$
- We will identify the code $\langle T\rangle$ of the TM T with $\langle\delta\rangle$, i.e. $\langle T\rangle := \langle\delta\rangle$.

9.4.2 Enumeration of TMs

We can interpret $\langle T\rangle$ as the binary code of some natural number. We call this number the index of T.

Notice that some natural numbers are not indexes (because their binary codes do not have the required form, which results from the encoding method).

- To avoid this we introduce the following convention:
  Any natural number whose binary code is not of the required form is an index of the empty TM.

  The δ of the empty TM is everywhere undefined; for every input, this TM immediately halts (in 0 steps).

Now we can say: every natural number is the index of exactly one TM.

Given an arbitrary $n \in N$, we can obtain from the n the components Q, Σ, Γ, F that define the particular TM (i.e. the TM whose index is n). How?

1. We inspect the binary code of n to check if it is of the required form $111K_1 11K_2 11 \ldots 11K_r 111$.
2. If it is, we partition the code into strings $K_1, K_2, \ldots, K_r$ and by analyzing these we can collect all the information needed to obtain all the components $\delta, Q, \Sigma, \Gamma, F$ of the TM T.

The restored TM can be viewed as the nth basic TM and denoted by T_n.

By letting n run through $0, 1, 2, \ldots$ we obtain the sequence of TMs

$$T_0, T_1, T_2, \ldots$$

Notice that this is an enumeration of all basic TMs.

9.4.3 The existence of a universal turing machine

There is a Turing machine U that can compute whatever is computable by any other Turing machine.

Proof idea: The idea is to construct a Turing machine U that will be capable of simulating any other TM T. To achieve this, we use the method of proving by Computability Thesis (CT):

1. first, we describe the concept of the machine U and describe the intuitive algorithm (that should be) executed by U's Turing program, and
2. then we refer to CT to prove that U exists.

Proof

- The input tape contains an input word consisting of two parts: the code $\langle T \rangle$ of an arbitrary Turing machine $T = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$, and an arbitrary word w.
- The work tape is initially empty. The machine U will use it in exactly the same way as T would use its own tape when given the input w.
- The auxiliary tape is initially empty. The machine U will use it to record the current state in which T would be at that time, and for checking whether this state is a final state of T.
- The control unit contains a Turing program that executes an algorithm.

The Turing program of U should execute the following intuitive algorithm:

1. Check if the input word is $\langle T, w \rangle$, where $\langle T \rangle$ is a code of some TM. If it is not, halt.
2. From $\langle T \rangle$ restore F and write $\langle q_1, F \rangle$ to the auxiliary tape.
3. Copy w to the work tape and shift its window to the beginning.
4. (Let the auxiliary tape have $\langle q_i, F \rangle$ and the work tape window scan z_r).
   If $q_i \in F$, halt. (T would halt in the final state q_i).
5. On the input tape, search in $\langle T \rangle$ for the code of the instruction $\delta(q_i, z_r) = \ldots$
6. If not found, halt (T would halt in the final state q_i).
7. (The instruction $\delta(q_i, z_r) = \ldots$ was found and is $\delta(q_i, z_r) = (q_j, z_w, D)$.
   On the work tape, write the symbol z_w and move the window in the direction D.
8. On the auxilliary tape, replace $\langle q_i, F \rangle$ by $\langle q_j, F \rangle$.
9. Return to step 4.

This algorithm can be executed by a human. By the Computability thesis, there exists a TM $U = (Q_U, \Sigma_U, \Gamma_U, \delta_U, q_1, \sqcup, F_U)$ whose δ_U does the same (executes the algorithm). We call U the *Universal turing machine* (UTM).

9.4.4 Construction of an UTM

What is the simplest UTM?

- We focus on UTMs with no storage and a single two-way infinite tape with one track.
- How shall we measure the 'simplicity' of such UTMs?
  - Shannon proposed the product $|Q_U| \cdot |\Gamma_U|$ (the maximal number of instructions in δ_U)
  - A more realistic measure would be the actual number of instructions in δ_U.

Researchers focused on different classes of UTMs, denoting them by $UTM(s,t)$ where $s,t \geq 2$, containing all UTMs with s states and t tape symbols.

The smallest one (for now) is UTM(4,5) = 22 instructions

9.4.5 The importance of the Universal Turing Machine

Turing's discovery of a universal TM was a theoretical proof that a general-purpose computing machine is possible, at least in principle.

Turing was certain that such a machine could be built in reality:
It is possible to construct a physical computing machine that can compute whatever is computable by any other physical computing machine.
He envisaged something that is today called the general-purpose computer.

9.5 The first basic results

Theorems
Let S, A, B be arbitrary sets. Then:

1. S is decidable $\Rightarrow$ S is semi-decidable
2. S is decidable $\Rightarrow$ $\overline{S}$ is decidable
3. S and $\overline{S}$ are semi-decidable $\Rightarrow$ S is decidable
4. A and B are semi-decidable $\Rightarrow$ $A \cap B$ and $A \cup B$ are semi-decidable
5. A and B are decidable $\Rightarrow$ $A \cap B$ and $A \cup B$ are decidable

10 Undecidability

10.1 Decision problems and other kinds of computational problems

We define the following four kinds (classes) of computational problems:

- *Decision problems* (also called yes/no problems). The solution of a decision problem is the answer YES or NO (The solution is a single bit.)

- *Search problems.* Given a set S and a property P, the solution of the search problem is an element x of S such that x has the property P. (The solution is an element of a set.)
- *Counting problems.* Given a set S and a property P, the solution of a counting problem is the number of elements of S that have the property P. (The solution is a natural number.)
- *Generation problems* (also called enumeration problems). Given a set S and a property P, the solution of a generation problem is a list of elements of S that have the property P. (The solution is a sequence of elements of a set.)

We will focus on the decision problems.
Because the decision problems ask for the simplest possible solutions, i.e. solutions representable by a single bit. (We are pragmatic and hope that this will make our study of other kinds of computational problems simpler.)

10.2 Problem solving

10.2.1 Language of a decision problem

There is a link between decision problems and sets that enables us to reduce the questions about decision problems to questions about sets. We'll uncover it in 4 steps.

1. Let D be a decision problem.
2. We are usually faced with a particular instance d of the problem D. The instance d is obtained from D by replacing the variables in the definition of D with actual data. The problem D can be viewed as the set of all the possible instances of D. We will say that an instance $d \in D$ is positive/negative if the answer to d is YES/NO, respectively.
3. In the natural-language description of d there can be various actual data (numbers, matrices, graphs, ...). However, to compute the answer on a machine—be it an abstract model such as the TM or a modern computer—we must represent these actual data in a form that is understandable to the machine. How?
   Since any machine uses some alphabet Σ (e.g. $\Sigma = \{0, 1\}$), we must choose a function that will transform (code) every instance of D into a word in Σ^*. We call such a function the *coding function* and denote it by 'code'. Thus, code: $D \to \Sigma^*$, and *code*(D) is the set of codes of all instances of D. We will write $\langle d \rangle$ instead of code(d).
4. Gather the codes of all the positive instances of D in a set L(D).
   The language of a decision problem D is the set $L(D)$ which is defined as $L(D) = \{\langle d \rangle \in \Sigma^* | d \text{ is a positive instance of } D\}$.

Now observe that the following relation holds:

$$d \in D \text{ is positive} \Leftrightarrow \langle d \rangle \in L(D)$$

This is the link between decision problems and sets (formal languages).

What we gain:
Solving a decision problem D can be reduced to recognizing the set $L(D)$ in Σ^*.

The answer to the instance d of D can be found if we determine where is $\langle d \rangle$ relative to $L(D)$
The link is important because it enables us to apply –when discussing and/or solving decision problems –all the theory that we developed to recognize sets.

What does the recognizability of $L(D)$ tell us about the solvability of D ?

- $L(D)$ is decidable $\Rightarrow$ There is an algorithm that, for any $d \in D$, answers yes or no.
  Proof: There is a TM that, for any $\langle d \rangle \in \Sigma^*$, decides whether or not $\langle d \rangle \in L(D)$
- $L(D)$ is semi-decidable $\Rightarrow$ Then there is an algorithm that,
  - for any positive $d \in D$, answers yes;
  - for a negative $d \in D$, may or may not answer no in finite time

  Proof: There is a TM that, for any $\langle d \rangle \in L(D)$, accepts $\langle d \rangle$. However, if $\langle d \rangle \notin L(D)$, the algorithm may or may not reject $\langle d \rangle$ in finite time.
- $L(D)$ is undecidable $\Rightarrow$ Then there is no algorithm that, for any $d \in D$, answers yes or no.
  Proof: There is no TM capable deciding, for any $\langle d \rangle \in \Sigma^*$, whether or not $\langle d \rangle \in L(D)$.

Let D be a decision problem. We say that the problem

- D is decidable (or computable) $L(D)$ is a decidable set;
- D is semi-decidable $L(D)$ is a semi-decidable set;
- D is undecidable (or incomputable) $L(D)$ is an undecidable set.

Terminology
Instead of a decidable/undecidable problem we can say computable/incomputable problem. But the latter notion is more general: it can be used with all kinds of computational problems, not only decision problems. The terms solvable/unsolvable is even more general: it addresses all kinds of computational and non-computational problems.

10.3 Halting problem

The halting problem D_{Halt} is defined by

$$D_{Halt} = \text{“Given a TM } T \text{ and } w \in \Sigma^*, \text{ does } T \text{ halt on } w?\text{”}$$

The halting problem D_{Halt} is undecidable.
Comment: This means that there exists no algorithm capable of answering, for arbitrary T and w, the question “Does T halt on w?” So, any algorithm whatsoever, which we might design now or in the future for answering this question, will fail to give the answer for at least one pair T, w.

The universal language, denoted by K_0, is the language of the Halting problem, that is,

$$K_0 = L(D_{Halt}) = \{\langle T, w \rangle \mid T \text{ halts on } w\}.$$

The second language is obtained from K_0 by imposing the restriction $w := \langle T \rangle$.
The diagonal language, denoted by K, is defined by

$$K = \{\langle T, T\rangle \mid T \text{ halts on } \langle T\rangle\}$$

Note:

- K is the language of the problem D_H = "Given a TM T does T halt on $\langle T\rangle$"
- D_H is a subproblem of D_{Halt} (since it is obtained from D_{Halt} by fixing w to $w = \langle T\rangle$).

Proof idea:
We suppose that K is a decidable set.
Then there must exist a TM D_K that for any T, answers $\langle T, T\rangle \in ?K$:

$$D_K(\langle T, T\rangle) = \begin{cases} \text{yes}, & \text{if } T \text{ halts on } \langle T\rangle; \\ \text{no}, & \text{if } T \text{ doesn't halt on } \langle T\rangle; \end{cases}$$

Our intention is to construct S in such a way that, when given as input its own code $\langle S\rangle$, S will expose the incapability of D_K to predict whether or not S will halt on $\langle S\rangle$

D_K is unable to correctly decide the question $\langle S, S\rangle \in ?K$, but this contradicts out supposition that K is a decidable set D_K its decider. So K is not decidable.

10.4 The basic kinds of decision problems

10.4.1 The are undecidable sets that are still semi-decidable

K_0 is a semi-decidable set

> We must find a TM that accepts K_0. Here is an idea. Given an arbitrary $\langle T, w\rangle$, the machine must simulate T on w, and if the simulation halts, the machine must return yes and halt. So, if such a machine exists, it will answer yes *iff* $\langle T, w\rangle \in ?K_0$. But we already know that such a machine exists: it is the Universal Turing machine U. Hence, K_0 is semi-decidable.
>
> *Comment:* this is why K_0 is called the universal language.

Corollary: K_0 is an undecidable (but still) semi-decidable set.
Similarly we prove for K.

10.4.2 The are undecidable sets that are not even semi-decidable

$\overline{K_0}$ is not a semi-decidable set.

> If $\overline{K_0}$ were semi-decidable, then both K_0 and $\overline{K_0}$ would be semi-decidable. But then K_0 would be decidable (Post's theorem). This would be a contradiction. So $\overline{K_0}$ is not semi-decidable.

In the same way we prove that $\overline{K}$ is not a semi-decidable set.

A decision problem D can be of one of the 3 kinds (classes):

- D is decidable
  This means that there is an algorithm that can solve any instance $d \in D$. Such an algorithm is called the decider of the problem D.

- D is semi-decidable undecidable
  This means that no algorithm can solve any instance $d \in D$. But there is an algorithm that can solve any positive $d \in D$. It is called the recognizer of D.

- D is not semi-decidable
  This means that for any algorithm there is a positive instance and a negative instance of D such that the algorithm cannot solve either of them

10.5 Some Other Incomputable Problems

There are many other incomputable problems about algebra, programs, grammars that no algorithm can completely solve.

10.5.1 Busy beaver problem

Informally, a busy beaver is the most 'productive' TM of its kind.
It does not waste time with writing symbols other than 1 or not moving the window. Let us group such TMs into classes $\tau_n, N = 1, 2, \ldots$ where τ_n contains TMs with the same number of states.

Define τ_n (for $n \geq 1$) to be the class of all TMs that have:

- the tape unbounded in both ways;
- n non-final states (including q_1) and one final state q_{n+1};
- $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, \sqcup\}$;
- δ that writes only the symbol 1 and moves the window either to L or R.

For any $n \geq 1$, there are finitely many TMs in τ_n.

We say that a TM $T \in \tau_n$ is a stopper if T halts on an empty input.
For every $n \geq 1$, there exists a stopper in τ_n. Hence there is at least one and at most finitely many (i.e. $|\tau_n|$) stoppers in τ_n.

So, there must exist in τ_n a stopper that attains, among all the stoppers in τ_n, the maximum number of 1s that are left on the tape after halting.

Such a stopper is called the n-state busy beaver and denoted n-BB.

The *busy beaver function* is $s(n) =$ 'the number of 1s attained by n-BB'
The busy beaver function is incomputable.

11 Computational complexity theory

11.1 Deterministic time and space(classes DTIME, DSPACE)

11.1.1 Deterministic time complexity & complexity classes DTIME

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a DTM with $k \geq 1$ 2-way infinite tapes. We say that DTM M has *(det.)* time complexity $T(n)$ if, for every $w \in \Sigma^*$ of length n, M makes $\leq T(n)$ steps before halting.

- It is assumed that M reads all of w; thus $T(|w|) \geq |w| + 1$, so $T(n) \geq n + 1$. So $T(n)$ is at least linear.

A TM M of time complexity $T(n)$ can decide $w \in ?L(M)$ in $\leq T(|w|)$ steps.

A language L has (det.) time complexity $T(n)$ if there is a DTM M of (det.) time complexity $T(n)$ such that $L = L(M)$. We define the class of all such languages by

$$\text{DTIME}(T(n))=\{L \mid L \text{ is a language} \wedge L \text{ has (det.) time complexity } T(n)\}$$

Informally, DTIME($T(n)$) contains all Ls for which the problem $w \in ?L$ can be det. solved in $\leq T(|w|)$ time.
This can be extended to a decision problem D and its language $L(D)$ (p.277).

11.1.2 Deterministic space complexity & complexity classes DSPACE

Let $M = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a DTM with 1 input tape and $k \geq 1$ work tapes. We say that DTM M has (det.) space complexity $S(n)$ if, for every input $w \in \Sigma^*$ of length n, M uses $\leq S(n)$ cells on each work tape before halting.

- input-tape cells do not count
- It is assumed that M uses at least the cell under the initial position of the window. So $S(n)$ is at least constant function 1.

A TM M of space complexity $S(n)$ can decide $w \in ?L(M)$ on space $\leq S(|w|)$.

A language L has (det.) space complexity $S(n)$ if there is a DTM M of (det.) space complexity $S(n)$ such that $L = L(M)$. We define the class of all such languages by

$$\text{DSPACE}(S(n))= \{L \mid L \text{ is a language} \wedge L \text{ has (det.) space complexity } S(n)\}$$

Informally, DSPACE($S(n)$) contains all Ls for which the problem $w \in ?L$ can be det. solved on $\leq S(|w|)$ space.
This can be extended to a decision problem D and its language $L(D)$ (p.279).

11.2 Nondeterministic time and space (classes NTIME, NSPACE)

11.2.1 Nondeterministic time complexity & complexity classes NTIME

$N = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a NTM. We say that NTM N is of nondet. time complexity $T(n)$ if, for every input $w \in \Sigma^*$ of length n, there exists a computation in which N makes $\leq T(n)$ steps before halting.

- Again it is assumed that N reads all of w; thus $T(|w|) \geq |w| + 1$, so $T(n) \geq n + 1$. So $T(n)$ is at least a linear function.

A NTM N of time complexity $T(n)$ can decide $w \in = L(N)$ in $\leq T(|w|)$ steps.

A language L is of nondet. time complexity $T(n)$ if there is a NTM N of nondet. time complexity $T(n)$ such that $L = L(N)$. The class of all such languages is

NTIME($T(n)$)=$\{L \mid L$ is a language $\wedge L$ has nondet. time complexity $T(n)\}$

Informally, NTIME($T(n)$) contains all Ls for which the problem $w \in ?L$ can be nondet. solved in $\leq T(|w|)$ time. This can be extended to a decision problem D an its language $L(D)$ (p.282).

11.2.2 Nondeterministic space complexity & complexity classes NSPACE

Let $N = (Q, \Sigma, \Gamma, \delta, q_1, \sqcup, F)$ be a NTM with 1 input tape and $k \geq 1$ work tapes. We say that NTM N is of nondet. space complexity $S(n)$ if, for every input $w \in \Sigma^*$ of length n, there exists a computation in which N uses, before halting, $\leq S(n)$ cells on each work tape.

- Again, the input-tape cells do not count.
- It is assumed that N uses at least the cell under the initial position of the window. So $S(n)$ is at least constant function 1.

A NTM N of nondet. space complexity $S(n)$ can decide $w \in ?L(N)$ on $\leq S(|w|)$ space.

A language L has nondet. space complexity $S(n)$ if there is a NTM N of nondet. space complexity $S(n)$ such that $L = L(N)$. The class of all languages is

NSPACE($S(n)$)=$\{L \mid L$ is a language $\wedge L$ is of nondet. space complexity $S(n)\}$

Informally, NSPACE($S(n)$) has all Ls for which $w \in ?L$ can be nondeterministically solved on $\leq S(|w|)$ space.
This can be extended to a decision problem D an its language L(D) (p.284).

11.2.3 Summary of complexity classes

In terms of formal languages:

DTIME($T(n)$) = $\{L \mid L$ is a language $\wedge$ L has time complexity $T(n)\}$

DSPACE($S(n)$) = $\{L \mid L$ is a language $\wedge$ L has space complexity $S(n)\}$

NTIME($T(n)$) = $\{L \mid L$ is a language $\wedge$ L has nondet. time complexity $T(n)\}$

NSPACE($S(n)$) = $\{L \mid L$ is a language $\wedge$ L has nondet. space complexity $S(n)\}$

In terms of decision problems:

DTIME($T(n)$) = $\{D \mid D$ is a decision problem $\wedge$ $L(D)$ has time complexity $T(n)\}$

DSPACE($S(n)$) = $\{D \mid D$ is a decision problem $\wedge$ $L(D)$ has space complexity $S(n)\}$

NTIME($T(n)$) = $\{D \mid D$ is a decision problem $\wedge$ $L(D)$ has nondet. time complexity $T(n)\}$

NSPACE($S(n)$) = $\{D \mid D$ is a decision problem $\wedge$ $L(D)$ has nondet. space complexity $S(n)\}$

Informally:

DTIME($T(n)$) = {decision problems solvable deterministically in time $T(n)$}

DSPACE($S(n)$) = {decision problems solvable deterministically on space $S(n)$}

NTIME($T(n)$) = {decision problems solvable nondeterministically in time $T(n)$}

NSPACE($S(n)$) = {decision problems solvable nondeterministically on space $S(n)$}

11.3 Tape compression, linear speedup, and reductions in the number of tapes

11.3.1 Tape compression

We can encode several symbols to one symbol from a larger alphabet.

For any $c > 0$ is DSPACE($S(n)$) = DSPACE($cS(n)$) and NSPACE($S(n)$) = NSPACE($cS(n)$)

11.3.2 Linear speedup

We can group several steps into a new, larger step.

Two conditions need to be fulfilled:

- TM must have at least 2 tapes (e.e. $k > 1$),
- $inf_{n\to\infty} T(n)/n = \infty$ must hold

  Informally: $T(n)$ must grow faster than n (at least slightly). Only then will there remain, after reading the input, some time available for computation.

If $inf_{n\to\infty} T(n)/n = \infty$, then for any $c > 0$
DTIME($T(n)$) = DTIME($cT(n)$) and NTIME($T(n)$) = NTIME($cT(n)$)

11.3.3 Summary

Under certain conditions:

DTIME($T(n)$) = DTIME($cT(n)$)

NTIME($T(n)$) = NTIME($cT(n)$)

DSPACE($S(n)$) = DSPACE($cS(n)$)

NSPACE($S(n)$) = NSPACE($cS(n)$)

Instead of saying that a decision problem D is in $DTIME(n^2)$, we can say that D has det. time complexity of the order $O(n^2)$.

11.3.4 Reductions in the number of tapes

If $L \in DTIME(T(n))$, then L is accepted in time $O(T^2(n))$ by a 1-tape TM.

If $L \in NTIME(T(n))$, then L is accepted in time $O(T^2(n))$ by a 1-tape NTM.

If $L \in DTIME(T(n))$, then L is accepted in time $O(T(n)logT(n))$ by a 2-tape TM.

If $L \in NTIME(T(n))$, then L is accepted in time $O(T(n)logT(n))$ by a 2-tape NTM.

If L is accepted by a k-work tape TM of space complexity $S(n)$, then L is accepted by a 1-work-tape TM of space complexity $S(n)$.

11.4 Relations between DTIME, DSPACE, NTIME, NSPACE

11.4.1 Relations between different complexity classes

- $DTIME(T(n)) \subseteq DSPACE(T(n))$
  i.e. What can be solved in time $O(T(n))$, can also be solved on space $O(T(n))$
- $L \in$DSPACE$(S(n)) \wedge S(n) \geq log_2 n \Rightarrow \exists c : L \in$DTIME$(c^{S(n)})$
  i.e. What can be solved on space $O(S(n))$, can also be solved in (at most) time $O(c^{S(n)})$. (Here c depends on L.)
- $L \in$NTIME $(T(n)) \Rightarrow \exists c : L \in$DTIME$(c^{T(n)})$
  i.e. What can be solved nondeterministically in time $O(T(n))$, can be solved deterministically in (at most) time $O(c^{T(n)})$.
  Consequently, the substitution of a nondeterministic algorithm with a deterministic one causes at most exponential increase in the time required to solve a problem.
- NSPACE$(S(n)) \subseteq$ DSPACE$(S^2(n))$, if $S(n) \geq log_2 n \wedge S(n)$ is "well-behaved" i.e. What can be solved nondeterministically on space $O(S(n))$, can also be solved deterministically on space $O(S^2(n))$.
  Consequently, the substitution of a nondeterministic algorithm with a deterministic one causes at most quadratic increase in the space required to solve a problem.

11.4.2 "Well-behaved" complexity

A function $S(n)$ is space constructible if there is a TM M of space complexity $S(n)$ such that for each n, there exists an input of length n on which M uses exactly $S(n)$ tape cells. If for each n, M uses exactly $S(n)$ cells on any input of length n, then we say that $S(n)$ is fully space constructible.

A function $T(n)$ is time constructible if there is a TM M of time complexity $T(n)$, such that for each n, there exista an input of length n, on which M makes exactly $T(n)$ moves. If for all n, M makes exactly $T(n)$ moves on any input of length n, then we say that $T(n)$ is fully time constructible.

The sets of space and time constructible functions are very rich and include all common functions. Moreover, most common functions are also fully space and fully time constructible.

11.5 The classes P, NP, PSPACE, NPSPACE

The requirements of a computation for a computational resource (time, space) are considered to be reasonable if they are bounded by some polynomial.

11.5.1 P, NP, PSPACE, NPSPACE

Define the complexity classes P, NP, PSPACE and NPSPACE as:

- P= $\bigcup_{i\geq 1}$DTIME(n^i)
  is the class of all decision problems deterministically solvable in polynomial time
- NP= $\bigcup_{i\geq 1}$NTIME(n^i)
  is the class of all decision problems nondeterministically solvable in polynomial time
- PSPACE= $\bigcup_{i\geq 1}$DSPACE(n^i)
  is the class of all decision problems deterministically solvable on polynomial space
- NPSPACE= $\bigcup_{i\geq 1}$NSPACE(n^i)
  is the class of all decision problems nondeterministically solvable on polynomial space

11.5.2 The basic relations (between P, NP, PSPACE, NPSPACE)

The following inclusions hold: P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ NPSPACE

- (P $\subseteq$ NP) Every deterministic TM of polynomial time complexity can be viewed as a (trivial) nondeterministic TM of the same time complexity.
- (NP $\subseteq$ PSPACE) If $L \in$NP then $\exists k$ such that $L \in$NTIME(n^k).
  So $L \in$ NSPACE(n^k), and hence (by Savitch) $L \in$DSPACE(n^{2k}). Therefore $L \in$PSPACE.
- (PSPACE = NPSPACE) Trivially, PSPACE $\subseteq$ NPSPACE. The opposite direction:
  NPSPACE = $\bigcup$ NSPACE(n^i) $\subseteq$ (by Savitch) $\subseteq$ DSPACE(n^j) $\subseteq$ PSPACE.

11.6 The question P $=^?$ NP

We have just proved PSPACE = NPSPACE
$\Rightarrow$ When space complexity is polynomial, nondeterminism adds nothing to the computational power.

We know P$\subseteq$NP, but is P=NP?
$\Rightarrow$ Is it true that when time complexity is polynomial, nondeterminism adds nothing to the computational power?

We try to prove that P$\subseteq$NP
using "the most difficult" problem in NP.

11.6.1 Problem reductions

Suppose that there existed a $D^* \in$ NP, such that we could "easily" reduce every $D \in$ NP to D^* in the following sense:

- there would exist a function $r : D \to D^*$

- that could "easily" transform any instance $d \in D$ into an instance $r(d) \in D^*$
- such that the solution s to $r(d)$ could be "easily" transformed into the solution "?" to d.

  Then, for every problem D, solving could be "easily" replaced by solving of D^*

If this were possible, then every D could be viewed as "at most as difficult as" D^* In other words, D^* could be viewed as the "most difficult" problem in NP.

11.6.2 Polynomial-time reductions

First we define "easily" = "in deterministic polynomial time"

A problem $D \in$NP is polynomial-time reducible to a problem D', i.e. $D \leq^p D'$, if there is a deterministic TM M of polynomial time complexity that, for any $d \in D$, returns a $d' \in D'$, such that d is positive $\Leftrightarrow$ d' is positive.
The relation $\leq^p$ is called polynomial-time reduction.
Intuitivelly: M replaces, in polynomial time, $d \in D$ with $d' \in D'$ that has the same answer as d. (M takes $\langle d \rangle$ and in poly. time returns a word $M(\langle d \rangle)$, where $\langle d \rangle \in L(D) \Leftrightarrow M(\langle d \rangle) \in L(D')$)

11.7 NP-complete and NP-hard problems

We have seen that the "most difficult" problem in NP could be defined as the problem D^* that has the following property:

- $D^* \in$ NP
- $D \leq^p D^*$, for $D \in$ NP

A problem D^* is said to be NP-hard if

- $D \leq^p D^*$, for every $D \in$ NP.

A problem D^* is said to be NP-complete if

- $D^* \in$ NP and
- $D \leq^p D^*$, for every $D \in$ NP

Hence, D^* is NP-complete if D^* is in NP and D^* is NP-hard.

11.7.1 An NP-complete problem: SAT

A Boolean expression is inductively defined as follows:

- Boolean variables $x_1, x_2, \ldots$ are Boolean expressions
- If E, F are Boolean expressions then so are $\neg E$, $E \vee F$, and $E \wedge F$.

A Boolean expression E is satisfiable if the variables of E can be consistently replaced with values TRUE/FALSE so that E evaluates to TRUE.

The problem SAT = "Is a Boolean expression E satisfiable?" (the Satisfiability Problem)

SAT is NP-complete

11.7.2 Proving NP-completeness of problems

Let $D \leq^p D'$. Then

- $D' \in P \Rightarrow D \in P$
- $D' \in \text{NP} \Rightarrow D \in \text{NP}$

So any problem D that can be $\leq^p$-reduced to a problem in P (or in NP), is also in P (or NP).

The relation $\leq^p$ is transitive. $D \leq^p D' \wedge D' \leq^p D'' \Rightarrow D \leq^p D''$

The following holds:

- D^* is NP-hard $\wedge$ $D^* \leq^p D^\star \Rightarrow D^\star$ is NP-hard
- D^* is NP-complete $\wedge$ $D^* \leq^p D^\star \wedge D^\star \in \text{NP} \Rightarrow D^\star$ is NP-complete

11.7.3 Summary

If P $\neq$ NP, then NP contains:

- NPC - the class of all NP-complete problems
- NPI - the class of all NP-intermediate problems
  (Ladner): if P $\neq$ NP, then there exists a problem in NP that is neither in P nor in NPC
- P

If P $\neq$ NP, then no problem in NPC or NPI has polynomial time complexity.