Finite Automata

A finite automaton has a finite number of different states that it can be in. We distinguish between accepting and non-accepting states, and define the automaton by a graph indicating the state transitions. The set of sequences which are accepted on being input defines the language accepted by the automaton $L \subseteq \Sigma^*$.

Formally we can take a deterministic automaton as $(Q, \Sigma, \delta, q_0, F)$ with Q the states, Σ the alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, $q_0 \in Q$ the start state, and $F \subseteq Q$ the set of accepting states. Where M is this tuple, $L(M) = \{w \in \Sigma^* \mid q_0 \xrightarrow{w}{}^* q \in F\}$.

We say that if $F' := Q \setminus F$ then $L(M')$ is the language complement of $L(M)$. A language is regular provided we can find an FSA that accepts it. Note that by simulating two automata simultaneously we can see that any finite language must be regular. We can tell that there must exist irregular languages just on the basis that the set of languages is uncountably infinite while there exist only countably many FSA.

Theorem 1 *Regular languages are closed under concatenation.*

To do this, we ideally want to just have an automata process the characters in the first language to determine whether they are accepted, followed by processing the characters in the second language if the first is accepted. The problem comes in noting where to split the sequence, motivating the introduction of non-deterministic finite automata.

Whereas in a deterministic finite automaton (DFA) we have for each $a \in \Sigma$ there is a unique a-transition, in a non-deterministic finite automaton (NFA) there may be multiple or no a-transitions. We say that in an NFA, a string is accepted provided there exists a possible sequence of state-transitions that reaches an accepting state, irrespective of whether other possible sequences would reject.

Formally we take an NFA as $(Q, \Sigma, \delta, q_0, F)$ again, with the exception that $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function, so for $a \in \Sigma \cup \{\varepsilon\}$, $q \xrightarrow{a} q' \Leftrightarrow q' \in \delta(q, a)$. We also write $q \Rightarrow q'$ where $q \overset{\varepsilon}{\Rightarrow} q'$ iff $q = q'$ or we can get from q to q' via some sequence of ε-transitions, and $q \overset{w}{\Rightarrow} q'$ means the sequence $w \in \Sigma^*$ interspersed with ε-transitions can reach q' from q.

Theorem 2 *Every NFA has an equivalent DFA.*

The idea behind this is that given an NFA we want to have a DFA that tracks the possible states in which the NFA might be. To do this we use a state space which is the powerset of the NFA's state space, where to be in a state is to say that it was possible in the NFA to reach any of the elements in this state. Consequently the accepting states are those which contain an accepting element.

Now having that NFAs are equivalent to DFAs, thus for two DFAs accepting certain languages, we can just introduce ε-transitions between the accepting states of the first DFA to the beginning state of the second DFA, and thus we see that the languages' concatenation is accepted.

To get a more compact notation for regular languages, we use regular expressions. Taking $\Sigma \cup \{\varepsilon\}$ as the set of characters, and $\varnothing$ a distinct regular expression with $L(\varnothing) = \varnothing$, for any regular expressions E and F, $E + F$, $E \cdot F$, and E^* are all regular expressions. $\varnothing$ is the $(+)$-identity, and $\{\varepsilon\}$ the $(\cdot)$-identity.

Theorem 3 (Kleene's theorem) *Let $L \subseteq \Sigma^*$. L is regular if and only if L is denoted by some regular expression E.*

Assume we have a regular expression E with $L = L(E)$. Do this inductively. We have that ε, $\varnothing$, and a singleton $a \in \Sigma$ are each accepted by an NFA. If $L(E)$ and $L(F)$ are both regular, we have $L(E^*) = L(E)^*$ is regular, $L(E + F) = L(E) \cup L(F)$ is regular, and $L(EF) = L(E) \cdot L(F)$ is regular. Thus each regular expression is regular.

For any NFA, we construct the regular expression $E^X_{q,q'}$ for $X \subseteq Q$ such that $L(E^X_{q,q'})$ is the set of strings whereby one can get from q to q' with intermediate states in X. Thus $E^{\varnothing}_{q,q'}$ is the sum over a_i with $q' \in \delta(q, a_i)$.

Non-deterministic Pushdown Automata

To form an analogous automaton for CFLs as NFAs are for regular languages, we introduce the notion of a pushdown automaton. Intuitively, a pushdown automaton is like an NFA, except it has a stack to record information. In each step, the NPDA pops the top symbol off the stack, and as a function of this symbol, the symbol being read, and the state of the automaton, it may either push a symbol onto the stack, move the read head, or enter a new state.

Strictly, an NPDA is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$ where

- Q is the finite set of states.
- Σ is the finite input alphabet.
- Γ is the finite stack alphabet.
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the finite transition function, so there are only ever finite options to transition to. Allowing for stack sequences of different lengths is necessary in order to allow the number of elements in the stack to change.
- $q_0 \in Q$ is the start state.
- $\bot \in \Gamma$ is the initial stack symbol.
- $F \subseteq Q$ is the set of accept states.

A configuration of M is an element of $Q \times \Sigma^* \times \Gamma^*$ describing the current state, the portion of the input yet unread, and the current stack contents. The start configuration is $(q_0, w, \bot)$, and if for $a \in \Sigma$, $(q, \gamma) \in \delta(p, a, A)$, then for any $v \in \Sigma^*$ and $\beta \in \Gamma^*$,

$$(p, av, A\beta) \to (q, v, \gamma\beta)$$

and if $(q, \gamma) \in \delta(p, \varepsilon, A)$ then we just get $(p, v, A\beta) \to (q, v, \gamma\beta)$ (no input symbol is consumed).

We define the reflexive transitive closure of $\to$, $\overset{*}{\to}$ by

$$C \xrightarrow{0} D \iff C = D$$
$$C \xrightarrow{n+1} D \iff \text{there is } E \text{ such that } C \xrightarrow{n} E \text{ and } E \to D$$

and $C \overset{*}{\to} D$ iff there is $n \geq 0$ such that $C \overset{n}{\to} D$. Then M accepts an input x if for some $q \in F$ and $\gamma \in \Gamma^*$ we have $(q_0, x, \bot) \overset{*}{\to} (q, \varepsilon, \gamma)$. An equivalent accepting convention requires $\gamma = \varepsilon$.

Note that deterministic PDAs are a special case of NPDAs, but they are strictly weaker (to prove, note that NPDAs are not closed under complements).

Theorem 4 *A language is context-free if and only if some NPDA recognises it.*

Given a CFG $(V, \Sigma, \mathcal{R}, S)$ we can relatively straightforwardly construct an NPDA by holding the constructed word in the stack, and terminating after all variables have become terminals. The reverse direction is slightly more complicated, as we must first show that any NPDA is equivalent to an NPDA with a single state. If we have this, then define a CFG by $\mathcal{R} = \{A \to cB_1 \cdots B_k \mid (q, B_1 \cdots B_k) \in \delta(q, c, A), c \in \Sigma \cup \{\varepsilon\}\}$.

To show that every NPDA can be simulated by a one-state NPDA, we need to maintain all state information on the stack. Set $\Gamma' = Q \times \Gamma \times Q$, written as $\langle pAq \rangle$. For each transition $(q_0, B_1 \cdots B_k) \in \delta(p, c, A)$, include in δ' the transition $(*, \langle q_0 B_1 q_1 \rangle \langle q_1 B_2 q_2 \rangle \cdots \langle q_{k-1} B_k q_k \rangle) \in \delta'(*, c, \langle pAq_k \rangle)$ for all choices of $q_1, \cdots, q_k$.

Theorem 5 *Context-free languages are closed under the regular operations of union, concatenation, and star.*

Kozen's axioms

For regular expressions we have the following equivalencies:

- $E + (F + G) \equiv (E + F) + G$.
- $E + F \equiv F + E$
- $E + \varnothing \equiv E$
- $E + E \equiv E$
- $(EF)G \equiv E(FG)$
- $\varepsilon E \equiv E\varepsilon \equiv E$
- $E(F + G) \equiv EF + EG$
- $(E + F)G \equiv EG + FG$
- $\varnothing E \equiv E\varnothing \equiv \varnothing$
- $\varepsilon + EE^* \equiv E^*$
- $\varepsilon + E^*E \equiv E^*$
- $F + EG \leq G \Rightarrow E^*F \leq G$
- $F + GE \leq G \Rightarrow FE^* \leq G$

Note that $E \leq F$ iff $L(E) \subseteq L(F)$. Kozen's theorem states that all valid equivalencies between regular expressions can be derived from the above axioms.

Non-regular languages

Lemma 6 (Pumping lemma) *If A is a regular language, then there is a number p such that if $s \in A$ of length at least p, then $s = xyz$ satisfying*

- *For each $i \geq 0$, $xy^iz \in A$*
- *$|y| > 0$*
- *$|xy| \leq p$*

We prove this using a DFA with $p = |Q|$. Suppose $s = a_1 \cdots a_n \in L(M)$ with $n \geq p$. With $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_p} q_p$, there must be a repeat in the sequence $q_0, \cdots, q_p$. With $q_j = q_{j'}$, we have some initial segment x before we reach q_j, a segment y which we may repeat however many times we would like to get from q_j back to itself, and finally z to get from q_j to q_n.

Theorem 7 (Myhill-Nerode) *A language is regular if and only if $\equiv_L$ has finite number of equivalence classes over Σ^* (the index of L), where $x \equiv_L y$ if for every $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$. Moreover, the index is the size of the smallest DFA recognising L.*

The backwards direction follows from considering a DFA with k states. With distinct sequences $x_1, \ldots, x_{k+1}$, $q_0 \xrightarrow{x_i}{}^* q_i$, by the pigeonhole principle there are some $i \neq j$ such that $q_i = q_j$, so for $z \in \Sigma^*$, $x_i z \in L$ iff $x_j z \in L$, so $x_i \equiv_L x_j$. Thus the index of L is at most k, so any regular language must have finite index.

The forwards direction follows by constructing the DFA

$$(\{[x] \mid x \in \Sigma^*\}, \Sigma, ([x], a) \mapsto [xa], [\varepsilon], \{[w] \mid w \in L\})$$

for which we always get $[\varepsilon] \xrightarrow{w}{}^* [w]$, and $[w] \in F$ iff $w \in L$.

Context-Free Grammars

Another way to generate strings is via a context-free grammar, formed of variables, terminals, rules, and a start symbol. We set w to be the start symbol, choose an occurrence of X in w (if none, stop), pick a rule whose left hand side is X and replace it, then repeat.

Formally, a CFG is a 4-tuple $G = (V, \Sigma, \mathcal{R}, S)$ where V is a finite set of variables (or non-terminals), Σ is a finite set of terminals, disjoint from V, $\mathcal{R} \subseteq V \times (V \cup \Sigma)^*$ is the set of rules where each left hand side is an input variable, and the right hand side a result it can be mapped to, and $S \in V$ is the start symbol. Note that if a variable does not have a rule corresponding to it, then no word can be produced and so that variable can be removed from V without change to anything.

The language generated by G is $L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$, where the relation $\Rightarrow$ is defined as $uAv \Rightarrow uwv$ for each $A \to w$ in $\mathcal{R}$, and $\Rightarrow^*$ its reflexive and transitive closure.

Theorem 8 *A language is regular if and only if it is generated by a right-linear CFG, a CFG where each rule is either of the form $R \to wT$ or $R \to w$ for $R, T \in V$, $w \in \Sigma^*$.*

Say a CFG is strongly right-linear if each rule is of the form $R \to aT$, $R \to T$, or $R \to \varepsilon$ for $a \in \Sigma$. Each right-linear CFG has an equivalent strongly right-linear CFG, which should be immediately clear. We can then construct equivalencies with variables corresponding to states, and transition functions representing rule application.

We say that a CFG is ambiguous if it has strings which have two distinct parse trees. Equivalently, a CFG is ambiguous if there is a string which has two different leftmost derivation, as all a parse tree does is identify a leftmost derivation. In general the question of whether a given CFG is ambiguous is very difficult.

Lemma 9 (Pumping lemma for CFLs) *If L is a context-free language, then there is a number p such that if $w \in L$ of length at least p, then w may be divided into five pieces, $w = uxyzv$ such that*

- *For each $i \geq 0$, $ux^iyz^iv \in L$*
- *$|xz| > 0$*
- *$|xyz| \leq p$*

The proof idea here is that we place the generating grammar into Chomsky normal form, so every rule is of the form $A \to BC$ or $A \to a$ where a is terminal, $S \to \varepsilon$ is permitted, and neither B nor C are the start variable. If we take p sufficiently large, to derive any word of length longer than p requires that a nonterminal variable be repeated in the derivation tree, meaning both that its repetition could be removed to get the case $i = 0$, and that additional repetitions can be introduced for $i > 1$.

Turing Machines

A Turing machine is defined by $(Q, \Sigma, \Gamma, \vdash, \_, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, where

- Q is a finite set of states.
- Σ is a finite set, containing the input alphabet.
- Γ is a finite set with $\Sigma \subset \Gamma$, containing the tape alphabet.
- $\vdash \in \Gamma \setminus \Sigma$ is the left endmarker.
- $\_ \in \Gamma \setminus \Sigma$ is the blank symbol.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.
- $q_0, q_{\text{acc}}, q_{\text{rej}} \in Q$ are the start, accept, and reject states, with $q_{\text{acc}} \neq q_{\text{rej}}$.

Intuitively, $\delta(q, a) = (q', b, L)$ means that when in state q scanning a, then enter state q', write b over the cell, then move the head left by one cell. We must restrict δ in relation to the left endmarker, such that for any $p \in Q$ there is q such that

$$\delta(p, \vdash) = (q, \vdash, R)$$

so $\vdash$ is never overwritten, and never moved left of. Additionally, neither q_{acc} nor q_{rej} can ever be left, so for any $a \in \Gamma$ there is $c, c' \in \Gamma$, $D, D' \in \{L, R\}$ such that

$$\delta(q_{\text{acc}}, b) = (q_{\text{acc}}, c, D)$$
$$\delta(q_{\text{rej}}, b) = (q_{\text{rej}}, c', D')$$

At any point in time we can write the state of a turing machine as $(u, q, v) \in \Gamma^* \times Q \times \Gamma^*$ where the tape contents are uv and the head location is over the first symbol of v. Thus we write for $\delta(q, b) = (q', c, D)$ if $D = L$

$$(ua, q, bv) \to (u, q', acv)$$

and if $D = R$

$$(ua, q, bv) \to (uac, q', v)$$

We say that a Turing machine accepts a sequence $w \in \Gamma^*$ if $(\varepsilon, q_0, \vdash w) \to^* (u, q_{\text{acc}}, v)$ for some $u, v \in \Gamma^*$. Given an input, a Turing machine may either be accepted, rejected, or loop. We say that a Turing machine is total if it halts on all inputs.

One useful extension of Turing machines is multi-tape Turing machines. This is similar to the original Turing machine, except we have $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$. We can simulate this fairly easily, but we have to expand the alphabet to include not just $\Sigma_M \cup \{\vdash\} \cup \Gamma^k$, but $(\Gamma' \cup \Gamma)^k$, where Γ' is the set of marked symbols, where the mark indicates that this is where the head is located on that tape currently. At each point the tape scans through to find the marks, recalling them in the state, then going back through and changing the state correctly. Then it returns to the left end of the tape and restarts.

Another useful extension is non-deterministic Turing machines (NDTM), wherein the transition function has type $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. Like with NFAs, we say that a word is accepted if there is a branch by which we may reach an accepting state.

Lemma 10 *A language is recognised by some Turing machine if and only if it is recognised by some NDTM.*

To see this transformation, we use a 3-tape Turing machine to simulate an NDTM. The first branch stores the input word, the second simulates a branch, while the third stores the position of the process within the tree. We must do breadth-first search, because otherwise we have certain branches which don't terminate.

Algorithms

We can consider any algorithm as a decision problem. That is, a problem which expects a yes or no answer. We can represent each decision problem as a language - the set of instance encodings for which "yes" ought to be returned. We say that a decision problem is decidable (or recursive) if the corresponding language is decidable by a Turing machine. We can also have recursively enumerable languages, where we have a Turing machine which accepts every word in the language, but not necessarily terminate on other words.

Note that we can encode Turing machines within Σ^* with $\Sigma \neq 0$, so the number of Turing machines are countable. The number of languages $\mathcal{P}(\Sigma^*)$ is uncountable however, indicating that there is no surjective map from languages to Turing machines, so there must be undecidable languages.

Lemma 11 *$\{\langle A \rangle \mid A$ is a DFA such that $L(A) = \varnothing\}$ is decidable.*

To show this, run a DFS starting at the start state to find the connected component containing the start state, and accept if no accepting state is within this component.

Lemma 12 *$\{\langle A, B \rangle \mid A$ and B are DFAs such that $L(A) = L(B)\}$ is decidable.*

To do this, construct and run the emptiness deciding algorithm on $(A \cap \overline{B}) \cup (B \cap \overline{A})$, noting that this is empty if and only if $A = B$.

Lemma 13 *$\{\langle G \rangle \mid G$ is a CFG such that $L(G) = \varnothing\}$ is decidable.*

To do this, mark all terminal symbols in G, and then from each marked symbol mark the rule creating them, and check ultimately if the start symbol is marked. If it is not, then accept, and otherwise reject.

Lemma 14 *$\{\langle A, B \rangle \mid A$ and B are CFGs such that $L(A) = L(B)\}$ is undecidable.*

Universal Turing Machines

A universal Turing machine is a Turing machine U that can simulate the actions of any Turing machine. For any TM M and any input x of M, U accepts $\langle M, x \rangle$ if and only if M accepts (respectively rejects of loops on) x.

To construct U, we take it as a 3-tape TM. The first holds the input Turing machine, the second contains the contents of the input M's tape, and the bottom contains the current state of M, and the current position of M's tape head. **Finish proof**.

Theorem 15 *$\{\langle M, w \rangle \mid M$ is a TM that accepts input $w\}$ is recursively enumerable, but undecidable.*

The universal Turing machine, while accepting results that ought to be accepted, and rejecting those that ought to be rejected, cannot determine if a result is going to halt, and so cannot decide. To show this, assume that we have such a Turing machine H. Then we can construct D taking $\langle M \rangle$ as input which returns the opposite of H ran on $\langle M, \langle M \rangle \rangle$, and $D(\langle D \rangle)$ returns the opposite of $H(\langle D, \langle D \rangle \rangle)$, which is a contradiction.

We say that a language is co-RE if its complement $\overline{L}$ is RE.

Theorem 16 *A language is decidable iff it is both RE and co-RE.*

Theorem 17 *The Halting Problem, $\{\langle M, w \rangle \mid M$ is a TM and M halts on input $w\}$ is undecidable.*

Theorem 18 *The Emptiness Problem, $\{\langle M \rangle \mid M$ is a TM and $L(M) = \varnothing\}$ is undecidable.*

Theorem 19 *The Equivalence Problem, $\{\langle M_1, M_2 \rangle \mid M_1$ and M_2 are TMs and $L(M_1) = L(M_2)\}$ is undecidable.*

Theorem 20 *$\{\langle M \rangle \mid M$ halts on all inputs $\}$ is neither RE nor co-RE.*

Theorem 21 (Rice's theorem) *Every non-trivial property of the RE sets is undecidable. That is, for a proper non-trivial $S \subset RE$, a decision problem $P = \{\langle M \rangle \mid L(M) \in S\}$ is undecidable.*

We have that for any such problem, there is a K such that $\langle K \rangle \in P$. With $\langle M, x \rangle$ we can construct $N_{M,x}$ as a TM which takes an input y, saves it on a track, then writes x on another track and runs M on it. If M halts, then run K on y, and return the result. Assuming without loss of generality that P does not accept any Turing machine with an empty language, and we get that M halts on x iff $\langle N_{M,x} \rangle \in P$, and so P must be undecidable.

The intuition here is that we cannot determine statements about recursively enumerable languages using a Turing machine.

The Post Correspondence Problem gives a collection of dominoes such that reading the string off the top gives the same string as reading it off the bottom. It turns out that this is undecidable, because we can correspond any match with an accepting run of a Turing machine on an input.

Complexity

One way of introducing the notion of complexity is via Turing machines. Whereas language theory classifies sets according to their structural complexity, and in terms of what can be computed in principle, complexity theory considers the amount of resources required to recognise a language.

When we consider complexity, we only consider total Turing machines, and the worst case complexity.

Definition 1 *The running time of a Turing machine M is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that M takes before acceptance or rejection of an input length n.*

Theorem 22 *If L is recognised by a k-tape TM M_1 with running time bounded by $f(n)$, then L is recognised by a k-tape TM M_2 with running time bounded by $c \cdot f(n)$ for any $c > 0$, provided $f(n) = \omega(n)$*

To prove we just expand our tape alphabet so to allow us to take multiple steps in one.

Theorem 23 *Let $f : \mathbb{N} \to \mathbb{R}_+$ such that $f(n) \geq n$ for all n. Then for every k-tape TM with running time $f(n)$, there is an equivalent $O(f^2(n))$ single-tape TM.*

To do this just analyse the construction to get a multi-tape TM. At each step we make $O(kf(n))$ steps, as the tape has maximum length $O(f(n))$ at any point.

The observation from this is that all 'reasonable' deterministic computational models are polynomially equivalent – they can simulate each other with only a polynomial increase in running time.

Definition 2 **P** *is the class of languages that are decidable in polynomial time on a deterministic single tape TM:*

$$\mathbf{P} = \bigcup_{k \geq 0} \text{TIME}(n^k)$$

where $\text{TIME}(f(n))$ is the set of languages with complexity $O(f(n))$.

Definition 3 *For N a deciding NDTM. The running time of N is the function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n.*

An NDTM is called a decider if all branches halt on all inputs.

Theorem 24 *For $f(n)$ any function where $f(n) \geq n$. Every single tape NDTM whose running time is bounded by $f(n)$ has an equivalent $2^{O(f(n))}$ time deterministic single tape TM.*

Given an NDTM we construct the 3-tape TM using breadth first search. Every branch in the tree has length bounded by $f(n)$, and if b is the maximum number of choices for any configuration, there are at most $b^{f(n)}$ leaves. Thus the number of nodes is bounded by $O(b^{f(n)})$ so the total time is $O(f(n)b^{f(n)}) = 2^{O(f(n))}$. Furthermore, converting to a single-tape TM at most squares the running time, which does not change it asymptotically.

Corresponding to non-deterministic cases, we define $\text{NTIME}(f(n))$ as the set of languages decidable by an $O(f(n))$ non-deterministic TM.

Definition 4 **NP** *is the class of languages that are decidable in polynomial time on a non-deterministic single-tape TM:*

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

An alternative definition of **NP** is in terms of polynomial time verifiers.

Definition 5 *A verifier for a language A is a TM V such that*

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

The string c is called a certificate and supplies the proof of membership in A. When V runs in a time polynomial in the length of w, then V is a polynomial time verifier.

Theorem 25 **NP** *is the class of all languages that have a polynomial time verifier.*

Assume an $O(n^k)$ time verifier V. Given a problem instance w, we can non-deterministically guess the certificate c of length at most n^k and run V on $\langle w, c \rangle$. **Finish proof**

We say that a language A is polynomial time reducible to a language B, written $A \leq_p B$, if there is a computable function f, computed by a polynomial time TM, and for every w, $w \in A$ iff $f(w) \in B$. Intuitively, A takes at worst a polynomial time addition to the time of B to be recognised.

If for every $A \in \mathbf{NP}$, $A \leq_p B$, then B is NP-hard. If additionally $B \in \mathbf{NP}$, then B is NP-complete.

Theorem 26 *Let B be an NP-complete problem. Then $B \in \mathbf{P}$ iff $\mathbf{P} = \mathbf{NP}$.*

To show that a problem B is NP-complete, we initially need $B \in \mathbf{NP}$, and then either that $A \leq_p B$ for all $A \in \mathbf{NP}$, or that $A \leq_p B$ for some known NP-complete problem A. One useful example is the SAT problem, which decides for a propositional formula φ whether there is an assignment to its variables via which it is true.