Turing-machines

Friedl Katalin
BME SZIT
friedl@cs.bme.hu

March 22, 2016

The assumption that the memory of a PDA is a stack turns out to be a pretty strong restriction. Now, we introduce a computational model that relaxes this restriction. Although this is also a theoretical model, we will see that from a certain point of view it is the most general possible. There are many equivalent variants of Turing machines, but for the sake of simplicity we only treat here one deterministic and one nondeterministic version. The definition is similar to the preceding ones, but here the stack is replaced by a tape. We can move on the tape one slot at a time, however, we can move forwards and backwards, as well.

Deterministic Turing-machine

Defintion 1 *Let $k \geq 1$ be an integer. A k-*tape Turing-machine *is described by a seven tuple $M = (Q, \Sigma, \Gamma, q_0, *, F, \delta)$, where:*

- *Q is a finite nonempty set, the set of* states *of the machine*
- *Σ is a finite nonempty set, the input* alphabet
- *Γ is a finite nonempty set,* tape alphabet*, $\Sigma \subset \Gamma$*
- *$q_0 \in Q$ the* start state
- *$* \in \Gamma \setminus \Sigma$, the* blank symbol *of the tape,*
- *$F \subseteq Q$ the set of* accept states*,*
- *δ the* transition function*, $\delta : (q, a_1, a_2, \cdots, a_k) \to (q', b_1, b_2, \cdots, b_k, D_1, D_2, \cdots, D_k)$, where $q, q' \in Q$, $a_i, b_i \in \Gamma$ and $D_i \in \{L, R, S\}$ (that is **L**eft, **R**ight or **S**tay).*

The meaning is the following. Each tape has a beginning and is one-way infinite. Each slot on the tapes can store one symbol of Γ. The machine is in state q_0 at the beginning. On the first few slots of the first tape (starting at the first slot) the input word is stored, which can only contain symbols from Σ. The rest of the first tape, and if $k > 1$, then the other tapes everywhere are

filled up with blank symbol $*$. Each tape has a read/write head that stay on the first slot.

If in a given situation the character under the read/write head on the first tape is a_1, that on the second tape is a_2, on the i^{th} tape is a_i, and the machine is in state q, then in one step according to the value of the transition function $\delta(q, a_1, a_2, \ldots, a_k) = (q', b_1, b_2, \cdots, b_k, D_1, D_2, \cdots, D_k)$ the machine moves to state q', rewrites character a_i to b_i on the i^{th} tape and the head moves Left, Right or Stays put corresponding to the value D_i.

The Turing machine performs sequence steps corresponding to its transition function during a computation. We have to take care of that if a head is at the beginning of a tape then it does not move Left from there (it must not "fall off" the tape). The computation stops when machine cannot move, that is the computation gets stuck, i.e., the transition function is not defined for the given situation. The machine accepts the input if it gets stuck in an accept state (a state in F).

It is important to note that it is not guaranteed that a Turing-machine stops on a given input. The following possibilities are there. In the second and third case the Turing-machine does not accept the input word.

- the machine sooner or later stops in an accept state, that is it accepts the input.
- the machine sooner or later stops in a non-accept state, that is it rejects the input
- the machine never stops on the given input, that is it loops. In this case the machine does not accept the input.

The definition of Turing-machines resembles to the definition of incomplete automata. Important formal restriction is that now the condition of acceptance is different. In case of finite automata and pushdown automata it was required that the input must be completely read till its end, the computation must not get stuck before that. Now, it is not necessary to read the input completely, however, acceptance is only possible when the machine gets stuck. It is possible to give acceptance conditions similar to the ones in case of finite automata, we would obtain an equivalent model, but the form introduced above is the most generally widespread and it is simpler to use.

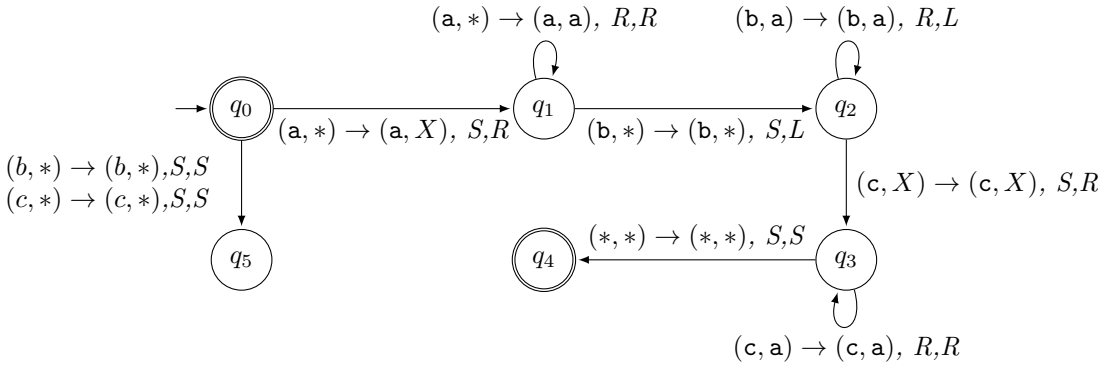
It can be shown, that despite of the differences that the Turing-machine is a generalization both of finite automata and pushdown automata: both can be easily simulated by Turing-machine. Indeed, the states of the automata are stored in the states of the Turing-machine, the stack of the pushdown automata can be stored on a tape. The Turing-machine has more possibilities than finite automate in two sense, namely the head(s) can move in two directions and that the heads can also write. It can be shown that from these two only the second one is important, since a finite automaton that can move in two direction on the tape also accepts only regular languages.

Defintion 2 *The language recognized by Turing-machine M:*

$$L(M) = \{w \in \Sigma : M \text{ accepts word } w\}$$

Turing-machines can also be represented by graphs. In this case the arrows representing transitions are labeled by what characters are written by the machine on reading given characters, furthermore in what directions the heads move.

Example 1 *The 2-tape Turing-machine shown below recognizes language* $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$. *The underlying idea is that the input characters* a*'s are copied to the second tape. Then reading the second tape from right to left we can compare the number of* b*'s to the number of* a*'s, finally, reading forward we can compare the same number to the number of* c*'s.*



In a little bit more details the working of the machine is as follows.

- *If the empty word is the input in state q_0, then the machine accepts it. If the input starts with character* b *or* c*, then the machine moves to state q_5 where the computation stops and the machine rejects the input. If the input starts with symbol* a*, then an X is written on the first slot of the second tape, so when reading from the right to left, it marks the beginning of the tape. The first tape is not changed.*

- *State q_1 is the copying state, as long as an* a *is read on tape 1, an* a *is written on tape 2. On reading the first* b*, the machine moves to state q_2. (If character* c *comes, then the computation stops since the machine gets stuck, and the input is rejected since q_1 is not an accept state.)*

- *In state q_2 the head of the first tape moves still to the right, while the head of tape 2 moves to the left as long as character* b *is read on the first tape and* a *is read on the second tape. The machine can move to state q_3 iff the first character* c *comes on the first tape exactly when the head arrives at the symbol X marking the beginning of the second tape, that is iff the numbers of characters* a *and* b *were equal.*

- *State q_3 is to compare the number of characters* c *and the number of* a*'s on tape 2. This time the head moves to right on tape 2, again.*

- *The machine moves to accept state q_4 iff the heads reach the blank symbols on both tapes at the same time.*

Thus, it is shown that language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$ can be recognized by a Turing-machine, but it is known that there is no pushdown automaton (let alone finite automaton) recognizing it.

The diagonal language and the halting language

In order to show a language that cannot be recognized by Turing-machine either, it is necessary to note that a Turing-machine can be described by a finite sequence of characters (basically, the transition function must be given). Such a sequence can be encoded by a 0/1 sequence, so a Turing-machine description (or a description of a pushdown automaton or finite automaton) can be considered as an element of $\{\mathtt{0},\mathtt{1}\}^*$. Naturally, not all words are descriptions of Turing-machines. If $w \in \{\mathtt{0},\mathtt{1}\}^*$ describes a Turing machine, then let the corresponding Turing-machine be denoted by M_w.

For the following language every 0/1 sequence is considered in two roles. Once, it is viewed as a Turing-machine description, second it is considered as an input of Turing-machines.

Defintion 3 *The* diagonal language L_d *consists of words $w \in \{\mathtt{0},\mathtt{1}\}^*$ that are Turing-machine descriptions and machine M_w does not accept word w, that is*

$$L_d = \{w \in \{\mathtt{0},\mathtt{1}\}^* : w \notin L(M_w)\}.$$

Theorem 1 *There exists no Turing-machine M that recognizes the diagonal language L_d.*

Proof: The proof is indirect. Let us assume that M is a Turing-machine that recognizes L_d and let x be its description. The question is whether x is contained in the diagonal language.

Case 1. Assume that $x \in L_d$. By the definition of the diagonal language this means that $x \notin L(M_x)$. On the other hand, by the choice of x, $M_x = M$ and by the choice of M we have that $L(M_x) = L(M) = L_d$. This is a contradiction, since both $x \in L_d$ and $x \notin L(M_x)$ should hold at the same time.

Case 2. Assume now that $x \notin L_d$. By definition this implies that $x \in L(M_x)$, which is a contradiction, again, since $L(M_x) = L(M) = L_d$.

. There are no other possible cases, so our original assumption, the existence of a Turing-machine recognizing L_d is false. □

Another "problematic" case is, when there exists a Turing-machine that recognizes the given language, however it does not stop in finite time for all inputs.

Defintion 4 *The* halting language L_h *consists of such (Turing-machine, input) pairs that the given Turing-machine stops in finite time on the given input.*

$$L_h = \{w\#x : M_w \text{ stops on input } x\}$$

Theorem 2 *There exists no Turing-machine M that stops in finite time on all inputs and recognizes the halting language L_h.*

Sketch of the proof: We show that if there were such a Turing-machine M, then there would exist another one M' that recognizes the diagonal language. However, this latter one is known not to exist. Thus, let us assume that Turing-machine M is such that it stops on every input and $L(M) = L_h$. This basically means that we can test whether machine M_w stops on input w.

If we know that it stops, then we can simulate it, as well (we need to interpret the description of the Turing-machine, basically) and after finite number of steps it turns out whether M_w accepts w. If $w \in L(M_w)$, then let M' reject the input, and if $w \notin L(M_w)$, then let M' accept it.

On the other hand, if it turns out that Turing-machine M_w does not stop on input w, then we do not need to simulate its computation. If it does not stop, then it does not accept either, so in this case let M' stop in a non-accept state.

Thus, we have constructed Turing-machine M' that stops in finite time on each input and recognizes the diagonal language, that contradicts to the previous theorem. □

Turing-machines can be viewed as programs written in some programming language, and the word fed to the machine can be considered as input of the program. Using this analogy, the previous theorem states that it is impossible to write a program that decides in finite time for any given other program and input for it whether it would stop in finite time.

We will soon see that this analogy is not an exaggeration. The power and limitations of Turing-machines in general is shown by the following.

Church–Turing-thesis

1. There exists a Turing-machine recognizing language L iff there exists a (not necessarily finite) process (algorithm) that accepts exactly the words of language L.

2. There exists a Turing-machine that stops for every input in finite time recognizing language L iff there exists a (always finite) process (algorithm) that for every input word x decides whether it is a word of language L.

We have no definition for the concepts of process, algorithm of the thesis, so we cannot consider the thesis above as a theorem. The Church–Turing-thesis states the fact based on experience that so far no computational model were introduced that could recognize more languages that Turing-machines can.

In short, the Church–Turing-thesis states that Turing-machine is a best possible and strongest computational model we have.

Nondeterministic Turing-machine

Nondeterminism is an already known concept, it works the same way here. The value of the transition function is a set in case of nondeterministic Turing-machine, the machine accepts an input word if there is a possible computation that stops in an accept state. It is worth noting that the tree of computation of a nondeterministic Turing-machine may contain infinite branches.

Similarly to finite automata, it is possible to get rid of nondeterminism in case of Turing-machines.

Theorem 3 *Every nondeterministic Turing-machine can be simulated by a deterministic Turing-machine.*

Sketch of the proof: Let M be the nondeterministic Turing-machine we want to construct deterministic machine M' for. The idea is that for an arbitrary input x M' performs a breadth first search walk on the computation tree of M (more precisely it generates from top to bottom this computation tree). If it finds an accepting leaf (that is where M would get stuck in an accept state), then M' stops in an accept state. If the BFS tree walk ends so that M' did not find an accepting leaf then it stops in a non-accept state. On the other hand, if the tree is infinite and it does not contain an accepting leaf, then M' will not stop, either, (which by definition means that it does not accept the input). □

Remark 1 *Note that depth first search cannot be used, because it does not return if there is an infinite path in the computation tree.*

Polynomial time

We study Turing-machines that stop on every input in finite time in the followings. In that case the main question is how many steps do they take before stopping. It is worth taking this number of steps as a function of the input length, since longer input naturally may require more steps.

Defintion 5 *Deterministic Turing-machine M is said to have* time complexity *(or* running time*)* $f(n)$ *if for every input x we have that M takes at most $f(|x|)$ steps on input x.*

That is $f(n)$ is an upper bound for the running time of the Turing-machine on input words of length n.

Defintion 6 *Nondeterministic Turing-machine M is said to have* time complexity *(or* running time*)* $f(n)$ *if for every input x we have that M takes at most $f(|x|)$ steps on input x.*

That is $f(n)$ is an upper bound for the running time of the Turing-machine on input words of length n independently of which branch of the computation tree we look at, that is $f(n)$ is an upper bound for the height of the computation tree.

Defintion 7 *M is of* polynomial time complexity*, if it has time complexity $f(n)$ for some polynomial $f(n)$ (that is for some constant c the running time is $O(n^c)$).*

Languages can be classified according to how fast Turing-machines can be fond for them.

The two arguably most important classes are P and NP.

Defintion 8 P *is the class languages that have polynomial time complexity deterministic Turing machines recognizing them, while* NP *is the class languages that have polynomial time complexity nondeterministic Turing machines recognizing them.*

Example 2 *For example, the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n : n \geq 0\}$ belongs to class* P*, since the Turing-machine described earlier uses not only polynomial, but linear number of steps as a function of input length.*

The language classes above are also interesting because of they are robust in the sense that which languages belong to the class is independent of what machine model is used to define the class. For example, if only 1-tape Turing-machines are considered, the same classes are obtained.

In general it is extremely tedious to rewrite an algorithm to Turing-machine formulation. In order to decide whether a language belongs to class P, typically enough to argue that there is an algorithm using polynomial number of steps to determine whether a word belongs to the language. Thus, languages that belong to effective algorithms studied before are in P.

Example 3 *The following languages belong to class* P*.*

- *Language of connected graphs. The words of the language are adjacency matrices of connected graphs. In case of an N vertex graph the size of the matrix is N^2. The connectedness of the graph can be decided using breadth first search in $O(N^2)$ running time. (So this is linear time algorithm, in fact.) The process can be implemented using Turing-machine in polynomial time, so this language is in class* P *.*
- *The language of bipartite graphs. This also has a polynomial algorithm.*
- *The language of bipartite graphs that have complete matching.*
- *Those words (G, s, t, k), where G is a graph with weighted edges, s and t are two vertices of the graph, k is a number and there exists a path between s and t in G of weight at most k.*

A polynomial time complexity nondeterministic Turing-machine has to be presented in order to show that a language belongs to class NP. If we want to translate this to more usual algorithmic ideas we need another characterization of class NP.

Theorem 4 (Verifier (witness) theorem) *It holds for a language L that $L \in$ NP iff there exists constants $c_1, c_2 > 0$ and language L_1 consisting of pairs of words such that $L_1 \in$ P and*

$$L = \{x : \text{ there exists } y, \text{ such that } |y| \leq c_1|x|^{c_2} \text{ and } (x, y) \in L_1\}.$$

According to the conditions L_1 has a polynomial time Turing-machine recognizing it. This (or the corresponding polynomial time algorithm) is called an effective (polynomial time) verifier of L, since it verifies that $x \in L$ with the help of appropriate (witness) y.

Sketch of the proof: Let us first assume that $L \in$ NP. This means that there exists a polynomial time complexity nondeterministic Turing-machine M such that $L(M) = L$. That is, there exists number k that on every input of length n the length of computation paths is $O(n^k)$. Thus, if $x \in L$, then M has a branch of computation that ends in an accept state and its length is at most $|x|^k$. Such a path can be described by specifying at each state in which branch to continue, that can be done by a constant amount of bits at each step, so the description length satisfies the requirement of the witness ($c_2 = k$).

Thus, let language L_1 consist of pairs (x, y) such that if x is considered as an input of machine M, then y describes a branch of the computation tree that ends with accept. This y satisfies the length requirement as it was shown above. Checking that this is really an accepting computation branch can be done by performing the corresponding computation steps in running time linear in the length of y. Note that if $x \notin L$, then there exists no y such that $(x, y) \in L_1$.

For the other direction, lets start from that for a given x we consider all sequences y of length $c_1 \cdot |x|^{c_2}$, and for all such y we run on pair (x, y) Turing-machine M' that recognizes language L_1. For each pair the running time is polynomial, nevertheless, the total time is exponential, because there are many y's. However, the good y can be searched for nondeterministically, that is generating y is the nondeterministic part, after that M is deterministic and accepts input x if M' accepts pair (x, y). The Turing-machine M constructed in this way recognizes language L both, the nondeterministic and the following deterministic parts are of polynomial time complexity. □

Example 4 *The following languages are in* NP*.*

- *The language* HAM *of graphs containing Hamiltonian cycles.* H∈ NP, because L_1 cab be chosen to consist of pairs of words where the first member of the pair is a description of a graph, the second member is a permutation of the vertices of the graph that forms a Hamiltonian cycle. (The sequence of vertices can be done by concatenations of bit sequences of lengths $\log n$.)

  It is clear that $L_1 \in$ P, because it is easy to decide for a graph and a sequence of numbers that the sequence forms a Hamiltonian cycle in the graph – one just have to check that each vertex occurs exactly once and there are edges between each pair of consecutive vertices, and between the first and last vertex. In case of an n vertex graph that can be done by

an algorithm of running time $O(n)$ (and by Turing-machine in polynomial time).

It is also clear that there exists a good pair for a graph description x if and only if the graph contains a Hamiltonian cycle. We need only to check that the size of the witness is polynomial in the size of the graph. This holds, since the graph description is of size n^2, while the description of the witness is of size $O(n \cdot \log n)$.

- *The language* COMPOSITE *consisting of binary forms of positive composite integers.* Let L_1 consist of pairs (m, t) of positive integers written in binary, such that $1 < t < m$ and m is divisible by t, that is a proper divisor t is the witness. Divisibility can be checked in polynomial time, so $L_1 \in P$, and naturally the length of t is at most the length of m.

- *The language* 3-COLOR *of the graphs that can be properly colored by three colors.* 3-COLOR$\in$ NP, since L_1 can be chosen consisting of pairs whose first term is a graph description and the second term is the colors of vertices.

  $L_1 \in$ P, because it is easy to decide about a given graph and a given coloring that it is a proper coloring of the graph, one just has to check for each edge whether its end vertices are of distinct colors, and that in total at most 3 colors were used. This can be done by an $O(n^2)$ running time algorithm.

  It is also clear that there exists a good pair for graph x only if the graph has a proper coloring. The size of the witness is polynomial of the size of the graph, since the color of each vertex can be described by a constant length bit sequence.

Defintion 9 *The* complement $\overline{L}$ *of language* L *consists of those words that are not in* L*, that is* $\overline{L} = \{x : x \notin L\}$.

Example 5 COMPOSITE $= \overline{\text{PRIME}}$*, where* PRIME *denotes the language consisting of prime numbers written in binary.*

Defintion 10 *Let* co NP *denote the class of complements of languages in* NP*, that is* co NP $= \{L : \overline{L} \in \text{NP}\}$.

Intuitively, while for languages in NP there are effective verifiers for belonging to the language, in case of languages in co NP the effective verifier exists for not belonging to the language. For example, this is so in case of language PRIME, since a proper divisor shows that a number is *not* a prime.

Theorem 5 P $\subseteq$ NP *and* P $\subseteq$ co NP

Proof: If $L \in$ P, then there exists a polynomial time complexity deterministic Turing-machine M such that $L(M) = L$. This M can be considered being nondeterministic, as well, so $L \in$ NP.

On the other hand, if $L \in P$, then $\overline{L} \in P$ also holds, since only accept and non-accept properties of states have to be swapped. $\overline{L} \in$ NP follows by the argument above, and so by definition $L \in$ co NP. □

Remark 2 *It can be read out from the proof of Theorem 3 that from a nondeterministic Turing-machine running in polynomial time $p(n)$, one can construct a deterministic Turing-machine running in exponential time $O(c^{p(n)})$.*

Intuitively, but little ambiguously one can say that a language belongs to P if for an arbitrary x it can be decided fast whether x belongs to the language, while a language is in NP, if by conjecturing (receiving as a present, being told by an oracle, or just seeing in a dream) a witness for that x belongs to the language, it can be verified fast.

One of the fundamental questions of computer science whether P = NP is true. This would mean that finding a proof is of same complexity as verifying it. This seems unbelievable, but there is no proof known for that P ≠ NP (neither exists proof for P = NP). Generally accepted belief is that P ≠ NP, but there are also some doubts, as well.

Karp-reduction

The following concept is useful in studying languages in NP. The definition can be applied in wider range, not only for languages in NP, but we will concentrate mainly on NP.

Defintion 11 (Karp-reduction) *Let $L_1, L_2 \subseteq \Sigma^*$ two languages. L_1 can be Karp-reduced to L_2, if there exists function $f : \Sigma^* \to \Sigma^*$ computable in polynomial time that $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

In notation $L_1 \prec L_2$.

The notation suggests, as we will see soon, that language L_2 is at least as hard as language L_1.

Remark 3 *The property that a function is computable in polynomial time should be interpreted as there exists a polynomial time algorithm that computes the value of $f(x)$ for a given x. Formally, this can also be defined by Turing-machines, such a version of Turing-machines is needed where the question is not whether the input is accepted, but what is on one of its predetermined tape at the time of halting. For example, one can require that the result of the computation is the content of the second tape.*

Example 6 *Similarly to language* 3-Color *one can define language* 4-Color. *It consists of descriptions of unoriented graphs that can be properly colored using 4 colors. We show that* 3-Color$\prec$ 4-Color. *We need to exhibit an appropriate function f.*

- *If x is not a graph description (for example its length is not a square number when we expect adjacency matrix), then let $f(x) = x$. In this case $x \notin$ 3-Color and $f(x) \notin$ 4-Color*

- *For a graph G let G' be the graph obtained by adding a new vertex to G and connecting it to every vertex of G. Let $f(G) = G'$.*

Function f can be computed in polynomial time since (the adjacency matrix of)G' can be constructed from the matrix of G. On the other hand, it is clear that G can be properly colored using 3 colors iff G' can be properly colored using 4 colors.

NP-completeness

Defintion 12 *Language L is* NP-complete*, if $L \in$ NP and for every $L' \in$ NP holds that $L' \prec L$.*

NP-complete languages can be considered being hardest in class NP, since every language in NP can be reduced to them.

Historically, the first NP-complete language consisted of Boolean formulas.

A *Boolean formula* consists of logic constants 0 and 1, logic variables (Boolean variables) $x_1, \ldots, x_n$, their negated forms $\overline{x_1}, \ldots, \overline{x_n}$ connected by operations $\wedge$ ("and") and $\vee$ ("or") and parentheses. A formula is satisfiable if there is an assignment of the variables so that the value of the formula is 1.

A Boolean formula is in *conjunctive normal form* or *CNF*, if it is in the following form.

$$(x_{i_1} \vee x_{i_2} \vee x_{i_3} \ldots) \wedge (x_{i_j} \vee x_{i_{j+1}} \vee x_{i_{j+2}} \ldots) \wedge \ldots$$

A 3CNF formula is a CNF formula, where there are at most 3 literals in each parentheses

Defintion 13 *The language of satisfiable formulas is*

$$\text{SAT} = \{\varphi(x_1, \ldots, x_n) : \exists b_1, \ldots, b_n \ \textit{evaluation such that} \ \ \varphi(b_1, \ldots, b_n) = 1\}$$

The language of satisfiable 3CNF formulas is

$$\text{3SAT} = \{\varphi(x_1, \ldots, x_n) : \varphi \in SAT \textit{ and } \varphi \textit{ is of 3CNF form}\}.$$

Remark 4 *Of course, the definition above should be understood so that formulas are coded by 0-1 sequences according to some syntax and the language consists of the codes.*

Theorem 6 (Cook, Levin) *Language* SAT *is NP-complete.*

Sketch of the proof: It is not hard to show that SAT is in NP, since an evaluation resulting in value 1 is a good witness. The length of the evaluation and the time required to check it are both polynomial in the length of the input.

The hard part of the proof is to show that every language in NP can be reduced to SAT. Let $L \in$ NP be arbitrary. Then there exists a polynomial time complexity nondeterministic Turing-machine M such that $L(M) = L$. If x is

an input of M, then the Karp-reduction assigns a formula to it such that the formula is satisfiable iff $x \in L$. The basic idea is that the formula essentially describes the computation of M on input x and is satisfiable iff there exists an accepting branch of the computation. We do not go into details of that, just for the taste of it there will be variables of type z_{iq} of meaning that after the i^{th} step machine M is in state q. These must satisfy that for each i within the number of steps there exists exactly one q such that $z_{iq} = 1$. Similarly there will be variables describing the contents of the tapes or the positions of the heads. The rules how these change from the i^{th} step to the $i + 1^{\text{st}}$ step can be derived from the transition function. □

Since the formula in the theorem can be given in 3CNF form, as well, we have

Theorem 7 *Language* 3SAT *is NP-complete.*