On Computability 2

Darius Lorek

January 26, 2024

Abstract

In the paper "On Computability 1" we demonstrated the equivalence of different computational models for the evaluation of Turing-computable functions, which form the broadest set of computable functions. Among other findings, we showed that constraint satisfaction with WHILE-loops is as powerful as Turing Machines.

This paper delves deeper into the study of Turing computability. Initially, we address the Goedelization of Turing machines and explain the renowned undecidability proofs. Then we introduce space and time complexity classes, categorizing algorithmically solvable problems into these classes. This exploration leads us to the major open problems in theoretical computer science concerning the precise containment hierarchy for these classes. In this connection we also explain the concept of nondeterminism. Finally, we discuss the exponential time hypothesis and the boundaries of computability.

While many textbooks have covered these topics, our paper aims to present the well-known proofs in a detailed, step-by-step manner, making them accessible even to non-experts. We believe these proofs rightfully stand as major intellectual achievements of the past 90 years and are worth studying.

1 Algorithms, Goedel-numbers, and undecidability problems

An algorithm is a set of instructions stipulated in a finite text. However, not every finite text qualifies as an algorithm; the instructions must be unambiguously interpretable and executable. Algorithms can either terminate or be capable of arbitrary continuation. When we have these instructional finite texts at our disposal, we may wonder about the operational properties of the algorithms they exhibit. Can we classify them all based on these properties? One of the most intriguing insights into the theory of computation has been given and proven: the answer is no![1] Before we provide evidence for this assertion and draw the implications derived from it, we need to introduce the concept of Goedelization.

Once again, we are working with unbounded Turing Machines (TM), addressing the broadest class of computable functions – the Turing-computable functions. We define a TM[2] as a 7-tuple ($Q, \Sigma, \Gamma, f, q_0, B, F$) where:

Q is a finite set of head states
Σ is a finite set of valid input symbols, a subset of Γ
Γ is a finite set of valid tape symbols
f is the transition function $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, can be undefined for some arguments
q_0 from Q is the start state
B is the blank symbol, a member of Γ
F is a set of end states, $F \subseteq Q$

[1] here we're not discussing trivial properties of the text, such as whether it begins with a 'T'
[2] see also the paper "On Computability 1"

We assume that the defined Turing Machine (TM) is a 1-tape TM. However, it can be shown that every n-tape TM can be simulated by a 1-tape TM. We enumerate the finite sets $Q = \{q_0, \ldots, q_t\}$ and $\Gamma = \{a_0, \ldots, a_r\}$. We can limit Γ to $a_0 = 0$, $a_1 = 1$, $a_2 = \#$, $a_3 = B$ without losing any information, achieved by coding every state and valid tape symbol binary. Furthermore, we assume q_0 is the start state. Each transition rule $f(q_i, a_j) = (q_k, a_l, s)$ can then be specified as:

$$f_{ijklm} = \#\#bin(i)\#bin(j)\#bin(k)\#bin(l)\#bin(m)$$

$$\text{with } bin(m) = \begin{cases} 0 \; for \; s = N \\ 1 \; for \; s = R \\ 10 \; for \; s = L \end{cases}$$

If we write all transition rules sequentially, we obtain a codification of the TM M in $\{0, 1, \#\}^*$. We then define a transformation function $h: \{0, 1, \#\}^* \rightarrow \{0, 1\}^*$ as $h(0) = 00$, $h(1) = 01$, $h(\#) = 11$ and $h(w_1 \ldots w_s) = h(w_1) \ldots h(w_s)$ with transition rules $w_1, \ldots, w_s$. This codification of the TM M in $\{0, 1\}^*$ we call Goedelization and the binary-coded number is the Goedel-number of M.

The transition rule words f_{ijklm} have a specific structure, commencing with the current state and input symbol, followed by the target state, output symbol and the instruction for the read/write-head movement. For deterministic TMs, the current state and the input symbol uniquely determine the subsequent configuration. Hence, if words w_{ijklm} and $w_{i'j'k'l'm'}$ are two distinct valid words of a TM, so $(i, j) \neq (i', j')$, which means that from a specific configuration only one subsequent configuration can be reached. We will later consider the nondeterministic case as well. Goedel-numbers are not unique for a TM; the words w_{ijklm} as elements of a Goedel-number, can be arranged in different orders and still describe the same TM. Additionally, a concrete Goedel-number doesn't necessarily describe only one TM; in that case, the TMs in question can be mutually transformed in each other.

Let u be a Goedel-number, and M_u be the Turing Machine described by u. The set $K = \{u \mid u \in \{0, 1\}^*, M_u \text{ halts when run on input } u\}$ is called the special halting problem. We will show by contradiction, that the special halting problem is not decidable.

Proof. We assume the existence of a TM Q that runs on inputs u, which can decide whether M_u applied on u halts or not. For this, Q must halt on every input u. The transition function[3] f_Q of Q is then defined as:

$$f_Q(u) = \begin{cases} 1 \text{ if } M_u \text{ halts when run on input u} \\ 0 \text{ else} \end{cases}$$

From Q, we can derive another TM R with the following transition function:

$$f_R(u) = \begin{cases} 1 \text{ if } f_Q(u) = 0 \\ \text{undefined if } f_Q(u) = 1 \end{cases}$$

so R doesn't halt in the case where $f_Q(u) = 1$. So, with the assumption of an ever-halting TM Q, we get a contradiction when running R on its own Goedel-number r:

R halts when run on r $\Leftrightarrow f_Q(r) = 0 \Leftrightarrow M_r$ doesn't halt on r $\Leftrightarrow$ R doesn't halt when run on r. ↯

[3] sometimes named characteristic function

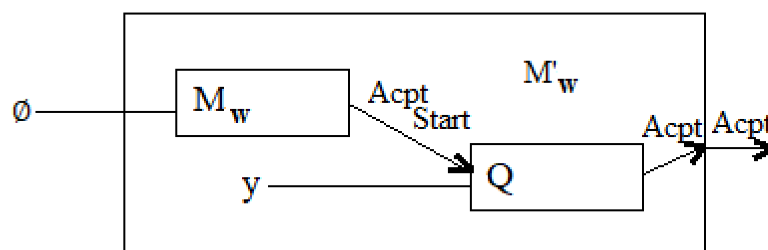
The general halting problem is a set H = {u v | u, v ∈ {0, 1}*, M_u halts when run on input v}. The undecidability for the general halting problem follows from the fact that K ⊆ H and K not decidable.

We will continue with the proof of the undecidability for TMs with an initially blank tape. This, in turn, contributes to proving the intriguing theorem of the general undecidability for any property or characteristic of the TM's transition function[4]. The halting problem for TMs with a blank tape is a set H_0 = {u | u ∈ {0, 1}*, M_u halts when run on a blank input}.

Proof. For every word u#v, where u, v ∈ {0, 1}* with a Goedel-number v and an input word u we assign a TM $M_{u\#v}$, which starts with a blank tape, initially writes the word u on the blank tape, moves to the beginning of the word and acts then like M_v on u. We can easily specify a Goedel-number for this composed TM $M_{u\#v}$. Now, let's assume H_0 is decidable. We will use the Goedel-number of $M_{u\#v}$ as input for the decision TM T for blank tape problems, which decides if $M_{u\#v}$ halts or not. But, $M_{u\#v}$ acting on the blank tape halts if and only if M_v halts on the input u. Therefore, if H_0 is decidable, the general halting problem H is also decidable, as $H \subseteq H_0$. ↯

The Rice theorem states that for a subset S of the set R of all Turing-computable functions ($S \subset R$), it holds $C(S) = \{u \mid f_{M_u} \in S\}$ is decidable only when $S = \emptyset$ or $S = R$. The idea behind the proof is to reduce this problem to one that is already known to be undecidable. Here, we do so by utilizing the H_0 problem.

Proof. We assume $S \neq \emptyset$ and $S \neq R$, and let ω be the nowhere-defined function (ω: ∅→A with an empty domain). Is ω ∈ S, there must exist a computable function q, $q \notin S$ as $S \neq R$. Let Q be the TM that computes q, and w ∈ {0, 1}* be an arbitrary word. Every binary word w describes a TM M_w. In case w has an inadequate format, so it describes a TM without movements. We assign the word w to a TM M'_w, which works as follows: with the word y ∈ {0, 1}* on its tape, it first ignores the input and acts like M_w on a blank tape. If M_w halts so M'_w acts then like Q on the input y, and in that case, the transition function $f_{M'_w} = q \notin S$, otherwise $f_{M'_w} = \omega \in S$.



Construction of the TM M'_w in the proof of Rice theorem

We have then a TM which can decide C(S) where C(S) is non-trivial ($S \neq \emptyset$ and $S \neq R$). But then the complement set $\overline{H_0} \subseteq C(S)$ and further $H_0 \subseteq C(S)$. ↯

At first glance, the proof may seem a bit contrived as we define that a TM that never stops still computes a function, specifically the nowhere-defined function. However, the crucial point lies in the assumption that we can construct a TM with a composed transition function f with $q \notin S$ and $\omega \in S$ and this hypothetical TM, if it were possible to build, would be capable of deciding the halting problem for every Goedel-number w. Reducing a problem to one that

[4] Henry Gordon Rice 1953

is already known as undecidable, is the common approach in undecidability proofs. Rice theorem states then that any non-trivial semantic property of a language recognized by a TM is undecidable. This finding has tremendous consequences for computability considerations. So, based on the code alone, we lack a general criterion to identify which function is implemented by a TM. Therefore, sets such as {u | Mu computes a constant function} or {u, v | Mu computes the same function as Mv} are undecidable.

Up to this point, without specific specification, we've been dealing with the broadest class of formal languages - the type-0 languages or recursively enumerable languages, which can be recognized by unrestricted TMs. By imposing appropriate restrictions on TMs, we can identify nested subclasses. These nested classes form the Chomsky hierarchy of formal languages or, equivalently, the hierarchy of automaton types that accept them. Rather than delving further into these formal language hierarchies, our focus will shift to the complexity classes of functions realized by TMs, where we'll examine their space and time complexities[5].

2 Space and time complexities

At first, we define some deterministic complexity classes and examine the formal languages within them.

Definition. Let's be $k \in N$ and s, t: $N \to N$ with $s(n) \geq n$ and $t(n) \geq n$ for all n.
DSPACE_k (s(n)) = {L | there exists a O(s(n))-space-bounded deterministic k-tape TM which accepts L}
DTIME_k (t(n)) = {L | there exists a O(t(n))-time-bounded deterministic k-tape TM which accepts L}
$\mathrm{DSPACE}(s(n)) = \bigcup_k \mathrm{DSPACE}_k(s(n))$
$\mathrm{DTIME}(t(n)) = \bigcup_k \mathrm{DTIME}_k(t(n))$

For any total recursive time- or space-bound t(n), s(n) it holds: there exists a recursive language L which is not in DTIME(t(n)) resp. DSPACE(s(n)). We will show it for DTIME(t(n)).

Proof. Since t(n) is total recursive, there exists a halting TM M that computes this function. We construct a multi-tape TM M' which accepts a language $L \subseteq \{0, 1\}^*$ not being in DTIME(t(n)). On one tape of M' we compute t(n) and use it as a counter. We consider x_i, the i-th string of the canonical order of $\{0, 1\}^*$: (0, 1, 00, 01, 10, 11, 000, …), as the Goedel-number of the TM M_i. The transition function is then encoded as a binary string in a special format. Again, if a string has an inadequate format, it describes a TM without movements. We consider multi-tape TMs M_i and define the language $L = \{x_i \mid x_i \in \{0, 1\}^*, M_i$ does not accept x_i in $t(|x_i|)$ steps$\}$. This language L is total recursive. The TM M', which accepts L, works as follows: on input w, M' simulates M on one tape and calculates t(|w|). Then, on that tape, it counts down every step M_i takes on $w = x_i$. M' simulates M_i for at most t(|w|) steps and accepts if M_i halts before reaching t(|w|) steps and does not accept if M_i reaches the time limit t(|w|) without reaching an accepting state.
Now, is L in DTIME(t(n))? If so, M_i would accept x_i in at most t(n) steps with $n = |x_i|$. But then, according to the definition $x_i \notin L$, a contradiction. On the other hand, if $x_i \in L$, then M_i will not accept x_i in $t(|x_i|)$ steps, contradicting L is in DTIME(t(n)). Both directions lead to contradictions. Then, M_i is not time-bounded by t(n), and $L \not\subseteq$ DTIME(t(n)). □

[5] for further exploration of formal languages, we recommend reading e. g. 'Introduction to Automata Theory, Languages, And Computation' by John E. Hopcroft and Jeffrey D. Ullman

Thus, by employing TMs defined in a way that prevents them from meeting the specified criteria required to verify the assumed assertions, we have shown through contradiction that for every function or language, it is possible to find functions or languages that are genuinely more complex in terms of DSPACE or DTIME. This establishes the existence of an infinite hierarchy of deterministic time and space complexity classes. This assertion can be extended to non-deterministic complexity classes as well.

The theorem above shows that for every recursive time or space complexity function f(n), there exists a complexity function f'(n) with a language L in that complexity class, which is not in the complexity class f(n). However, the question arises: at what point does an increase in upgrowth lead to a new complexity class? The following theorems show the magnitude of growth required to enter a new, higher deterministic complexity class.

In our subsequent considerations, we use TMs featuring a read-only input tape containing the input word, a working tape, and, if necessary, a distinct counter tape where we tally every move or each newly visited cell on the other tapes of the TM. TMs with separate input tapes are referred to as off-line TMs. In our exploration of complexity, our focus lies on the time and space consumption on the working tape. This enables us to investigate time complexities below the DTIME(n) limit. Otherwise, the mere scan of the input word of length n already takes n steps. A binary counter requires a space of $\log_2 n$ to count n steps.

Definition. We term a function s(n)-space constructible if there exists a TM M that is s(n)-space bounded and if, for any n ∈ N, there exists a word w of length n (n = |w|) on which M indeed uses s(n) tape fields. The set of space constructible functions includes, for example, log n, n^k, 2^n, n!, $s_1(n)*s_2(n)$ or $2^{s1(n)}$ if both $s_1(n)$ and $s_2(n)$ are space constructible. In this context, the TM M does not need to use s(n) tape fields for every w of length n but only for one specific w of length n. We now prove the following lemma:

Lemma. If a language L is accepted by an s(n)-space bounded TM with $s(n) \geq \log_2 n$, then L is also accepted by an s(n)-space bounded TM that halts on every input w.
Proof. Let M be an s(n)-space bounded TM with z states and t tape symbols that accepts L. If M accepts, it performs a sequence of at most $(n+2)zs(n)t^{s(n)}$ moves before the state configuration is repeated, as there are $(n+2)$ input head positions, z states, s(n) working tape positions, and $t^{s(n)}$ possible different contents on the working tape with an alphabet |Γ| = t. M can halt after counting down at most $(4zt)^{s(n)} \geq (n+2)zs(n)t^{s(n)}$ moves, as at this point a state configuration has been repeated. To see this, take the $\log_2$ on both sides of the inequation after reduction:

$$\log(4(4z)^{s(n)-1}) \geq \log((n+2)s(n))$$
$$\rightarrow 2 + 2(s(n)-1) + (s(n)-1)\log z \geq \log n + \log\left(1+\frac{2}{n}\right) + \log s(n)$$
$$\rightarrow 3\,s(n) - 1 \geq \log n + \log s(n) \text{ (as } \log z \geq 1 \text{ and } \log\left(1+\frac{2}{n}\right) \rightarrow 0 \text{ with } n \rightarrow \infty)$$
$$\rightarrow 2\,s(n) - 1 \geq \log s(n) \text{ (as we assumed that } s(n) \geq \log n)$$

Thus, $(4zt)^{s(n)} \geq (n+2)zs(n)t^{s(n)}$ is correct. We then implement a counter and count the moves from $(4zt)^{s(n)}$ downwards. Either M accepts w before the counter reaches 0, or the counter reaches 0, indicating that M has made at least $(n+2)zs(n)t^{s(n)}$ moves, and we are in a state configuration that we have encountered already before. M enters a loop, so we can halt without accepting w. Note, we need s(n) space for a counter when counting in base $4zt$. □

Hence, space bounded TMs using at least $\log_2 n$ tape space halt on every input. We will briefly show that if $s_2(n)$ is a full constructible function with $\lim_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$ and if $s_1(n)$ and $s_2(n)$ are both at least $\log_2$ n, then there exists a language in DSPACE($s_2(n)$) that is not in DSPACE($s_1(n)$). According to the definition, $s_2(n)$ is a full constructible function if a TM M uses exactly $s_2(n)$ cells on every input of length n (theorem T1).

Proof. Consider all off-line TMs with tape symbols {0, 1} and a separate working tape. We can specify these TMs as Goedel-numbers in binary order. A prefix filled with an arbitrary number of 1s guaranties that every TM can have an arbitrary long Goedel-number. Now, we construct a TM M that uses $s_2(n)$ cells but is distinct on at least one input word from any $s_1(n)$-space bounded TM. We let M work on a word w of length n.
First, we delimit the working space to $s_2(n)$ cells, by marking $s_2(n)$ cells on a separate tape, moving then the head synchronously to the working tape, and stopping if M tries to leave the marked sector. We can do it, as $s_2(n)$ is a full constructible function.
The word w will be rejected by M in the case it leaves the marked sector, ensuring that M is $s_2(n)$-space bounded. We then simulate with M the TM M_w on the word w, which is the binary Goedel-number of M_w. If M_w is $s_1(n)$-space bounded and uses t tape symbols, we need $\lceil \log_2 t \rceil \, s_1(n)$ space for this simulation, as every tape symbol needs at most $\lceil \log_2 t \rceil$ space in binary codification. M accepts w if it can conduct this simulation using at most $s_2(n)$ space, and M_w halts without accepting w. Since M is $s_2(n)$ space bounded, L(M) is in DSPACE($s_2(n)$). We will show that L(M) is not in DSPACE($s_1(n)$).
Suppose there exists an $s_1(n)$-space bounded TM with t tape symbols that accepts L(M). The Lemma above ensures that this TM halts on all input words. Furthermore, it appears infinitely many times in the list of all off-line TMs with valid tape symbols {0, 1} due to the infinite number of 1s prefixes in the Goedel-numbers of this TM. Additionally, as $\lim_{n\to\infty} \frac{s_1(n)}{s_2(n)} = 0$, we can find a sufficiently long word w with |w| = n such that $\lceil \log_2 t \rceil \, s_1(n) < s_2(n)$. Let M_w be the TM with the sufficiently long w. M has enough tape space on input w to simulate M_w and to accept if M_w rejects. Hence, L(M) ≠ L(M_w), leading to a contradiction. Therefore, L(M) is not in DSPACE($s_1(n)$). □

Note that, as in most previous proofs, a diagonalization argument was employed here. We listed all off-line TMs M_w, working on a sufficiently long own binary Goedel-numbers w in $\lceil \log_2 t \rceil \, s_1(n)$ space, and showed that an appropriately constructed TM M isn't in this list, as L(M) ≠ L(M_w) for all these sufficiently long binary Goedel-numbers w. Additionally, we assumed that $s_2(n)$ is a full constructible function, but this requirement can be relaxed to simple constructible functions.

If $s_2(n)$ is a full constructible function, a TM M uses exactly $s_2(n)$ cells on every input of length n. If $s_2(n)$ is a simple constructible function, M uses exactly $s_2(n)$ cells on at least one input of length n. Now we need to ensure that M is $s_2(n)$-space bounded. To achieve this, we employ a TM M_1, that marks $s_2(n)$ cells on some input w. Σ is the input alphabet for M_1. M operates with two traces on the input tape, utilizing the alphabet $\Sigma \times \{0, 1\}$. The first trace is processed as if it were the input for M_1, and the second trace as the code w of a TM M_w with the alphabet $\Sigma \times \{0, 1\}$. The modification, compared to the TM M in the previous proof, is that M now places $s_2(n)$ boundaries on tapes 1 and 2 through the simulation of M_1 on the first trace. This ensures that M is $s_2(n)$-space bounded, and rejects if these boundaries are violated.

Note that all common functions $s(n) \geq n$ are also full space-constructible.

3 Density of the time complexity hierarchy

In terms of time complexities, the number of tapes plays a significant role. While we diagonalize over all possible multi-tape TMs, for simulations, we need to use a TM with a concrete number of tapes. This process incurs a loss of $\log_2 n$ time when a 2-tape TM is employed for this simulation.

Consider a 2-tape TM M, where the cells of the first tape are divided into two traces, and the second tape is solely used for copying purposes. The first tape is not limited on both sides. At the beginning of the simulation, the input word is written on tape 1. The upper trace is empty, and the input word is placed in the lower trace. The r/w head cell is symbolised as H. To the left or right of H are cell blocks named B_n and B_{-n}, each of length 2^{n-1}, where $n \geq 1$:

a_{-8}	a_{-7}	a_{-6}	a_{-5}	a_{-4}	a_{-3}	a_{-2}	a_{-1}	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3		

Every move of the r/w head to the left or right is simulated by shifting the 'pushed away' tape symbols form the current block into the upper trace in the opposite direction. If then H points to an empty cell in the lower trace, symbols are drawn onto the H position. For example, after two left moves, the configuration on tape 1 would be as follows:

									a_0							
a_{-8}	a_{-7}	a_{-6}	a_{-5}	a_{-4}	a_{-3}	a_{-2}		a_{-1}	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3		

										a_0	a_1					
a_{-8}	a_{-7}	a_{-6}	a_{-5}	a_{-4}			a_{-3}	a_{-2}	a_{-1}	a_2	a_3	a_4	a_5	a_6	a_7	a_8
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3		

Please note that we retain all other symbols in their original cells until we need a new symbol in H from the next left or right 2^{n-1} block. Thus, another left move on tape 1 would result in:

									a_{-1}	a_0	a_1					
a_{-8}	a_{-7}	a_{-6}	a_{-5}	a_{-4}				a_{-3}	a_{-2}	a_2	a_3	a_4	a_5	a_6	a_7	a_8
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3		

In a further left move, block B_{-3} must be moved towards H, and block B_2 must be moved away:

												a_0	a_1	a_2	a_3	
a_{-8}					a_{-7}	a_{-6}	a_{-5}	a_{-4}	a_{-3}	a_{-2}	a_{-1}	a_4	a_5	a_6	a_7	a_8
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3		

Now we work again within the inner blocks until we need a_{-8} resp. a_0 in H for which we would need to perform the extensive shift again. So, the simulation of the next left resp. right move would result in (left):

									a_{-3}			a_0	a_1	a_2	a_3			
a_{-8}					a_{-7}	a_{-6}		a_{-5}	a_{-4}	a_{-2}	a_{-1}	a_4	a_5	a_6	a_7	a_8		
			B_{-3}			B_{-2}	B_{-1}	H	B_1	B_2				B_3				

resp. right move:

							a_{-5}					a_0	a_1	a_2	a_3			
a_{-8}					a_{-7}	a_{-6}	a_{-4}	a_{-3}		a_{-2}	a_{-1}	a_4	a_5	a_6	a_7	a_8		
			B_{-3}			B_{-2}	B_{-1}	H	B_1		B_2			B_3				

We utilize tape 2 exclusively for the block copy process. Now, let's determine the additional steps required by these head movements in comparison to the head movements of an ordinary TM M' which uses a tape without traces. The most time-consuming operation in this process is the shift of a whole block into the upper trace of the neighbouring block, along with the shift of a new block from the opposite direction onto cell H. We refer to this operation as the B_i operation. It can occur at most every 2^{i-1} moves for the block B_i because, before the B_i operation, all blocks B_1, B_2, …, B_{i-1} must be fully filled. Therefore, the first B_i operation cannot occur before the 2^{i-1}-th move of the simulated ordinary TM M'. When M' operates in t(n) time, our simulation performs B_i operations only for an i such that $i \leq \log_2 t(n) + 1$. Each such B_i operation takes up to $m2^i$ moves ($m \geq 3$), as time needed for copying a block is proportional to the block's length. When M' makes t(n) moves, the outlined simulation using a 2-trace tape incurs:

$$t_1(n) = \sum_{i=1}^{\log_2 t(n)+1} \left(m2^i \, \frac{t(n)}{2^{i-1}} \right) \qquad \text{(A)}$$

moves. From equation (A), we can derive $t_1(n) = 2m \cdot t(n) \cdot \lceil \log_2 t(n) + 1 \rceil \rightarrow t_1(n) < 4m \cdot t(n) \cdot \log_2 t(n)$. A multi-trace tape simulation of one-trace tape TMs isn't limited to single tape TMs, but it can also be applied to multi-tape TMs without any additional time loss. In this scenario, we treat all the blocks Bi on different tapes as one block Bi, all blocks B_{i+1} as one block B_{i+1}, and so on. Similarly, all cells H are treated as one cell H. We only employ a larger set of tape symbols in cells having k traces, k even and k > 2, and necessarily a more complex transition function. In time complexity considerations, we are free to enlarge both. The constant factor 4m in the above inequation is negligible. Therefore, if a language L is accepted by an m-tape TM M_m within the time complexity t(n), then it is accepted by a 2-tape TM M_2 within the time complexity $t(n) \cdot \log_2 t(n)$. We can now prove the following theorem: If $t_2(n)$ is a full time-constructible function, and $\lim_{n \to \infty} \frac{t_1(n) \log t_1(n)}{t_2(n)} = 0$, then there exists a language in DTIME($t_2(n)$) that is not in DTIME($t_1(n)$) (theorem T2).

Proof. The proof is analogous to the proof in the space case. We construct a TM M that is $t_2(n)$-time bounded but distinct on at least one input word from any $t_1(n)$-time bounded TM. M works as follows: it operates on the input word w as a Goedel-number of the TM M_w and simulates M_w on w. M has a fixed number of tapes, so for certain TMs M_w, M might have fewer tapes than M_w. However, as demonstrated earlier, every multi-tape TM can be simulated by a 2-tape TM with a logarithmic time cost ($\log_2 t_1(n)$). Moreover, M_w may use more tape symbols than the fixed number of M's tape symbols. This can incur at most a constant factor of time c, as these tape symbols of M_w can be encoded with tape symbols of M, which utilize a constant factor more space to be scanned in every head move.
To ensure that M simulates M_w for at most $t_2(n)$ steps and halts, $t_2(n)$ must be a full time-constructible function. An additional tape is then used to run simultaneously a TM that precisely requires $t_2(n)$ time on every input of length n, ensured because $t_2(n)$ is full time-constructible. M accepts w only if M_w completes processing on w and rejects it. We can specify the Goedel-number w of the TM M_w with a preceding 1s prefix so that w can be arbitrarily long. If M_w is a $t_1(n)$-time bounded TM and there exists a sufficiently long Goedel-number w for M_w (which is ensured), such that $c \cdot \log t_1(|w|) \cdot t_1(|w|) \leq t_2(|w|)$, the simulation will halt. In that case, $w \in L(M)$ if and only if $w \notin L(M_w)$. It holds that $L(M) \neq L(M_w)$ for every $t_1(n)$-time bounded M_w. Therefore L(M) is in DTIME($t_2(n)$) - DTIME($t_1(n)$). □

Example. If $t_1(n) = 2^n$ and $t_2(n) = n \cdot \log n \cdot 2^n$ then

$$\lim_{n\to\infty} \frac{t_1(n)\log t_1(n)}{t_2(n)} \to \lim_{n\to\infty} \frac{2^n \log 2^n}{n\ \log n\ 2^n} = \lim_{n\to\infty} \frac{1}{\log n} = 0$$

so $\mathrm{DTIME}(2^n) \subsetneq \mathrm{DTIME}(n \cdot \log n \cdot 2n)$.

Above theorems show that even a relatively modest growth increase in t(n) establishes a new complexity class in both space and time. However, there are lower bounds for this growth. For instance, the function $n \cdot 2^n$ versus the function 2^n does not constitute a new time-complexity class under the defined conditions, which require, among other things, the simulation of an n-tape TM with a 2-tape TM, decelerating the processing by a factor of $\log_2 t(n)$.

4 Nondeterministic Turing Machines and Savitch theorem

We need to introduce the concept of nondeterminism to understand important open problems in theoretical computer science, which have persisted without solutions for decades, including the most prominent among them – the P-NP problem.

A nondeterministic Turing Machine is a theoretical model of computation where the transition function f: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is replaced by the transition relation r: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. This means that given a state $q \in Q$ and a symbol $\alpha \in \Gamma$ on the tape, the TM may proceed in many different ways. There is not only one possible next state q' together with the written symbol α' and the head movement direction L or R. In complexity inquiries, the shortest way from the initial state q_0 to a final state h is to be considered.

There are evident relations among different complexity classes DSPACE, DTIME, NSPACE, and NTIME, and some less apparent. It is clear that $\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n) + 1)$, as within f(n) time, a TM can visit at most f(n) + 1 tape cells. Furthermore, it holds $\mathrm{NTIME}(f(n)) \subseteq \mathrm{DTIME}(c^{f(n)})$, as a f(n)-time bounded nondeterministic TM M_n with z states, t tape symbols and k tapes has at most $z(f(n) + 1)^k t^{k\,f(n)}$ state configurations on an input w of length n. This number can be bounded by $d^{\,f(n)} \geq z(f(n) + 1)^k t^{k\,f(n)}$ with $d = z(t + 1)^{3k}$ for all $f(n) \geq 1$ as:

$$\begin{gathered} z^{f(n)}(t+1)^{3k\,f(n)} \geq\ z^{f(n)}(t+1)^{2k\,f(n)}t^{k\,f(n)} \geq\ z(f(n)+1)^k t^{k\,f(n)} \\ \to z^{f(n)-1}(t+1)^{2k\,f(n)} \geq\ (f(n)+1)^k \\ \to (f(n)-1)\log z + 2kf(n)\log(t+1) \geq\ k\log(f(n)+1) \\ \to \frac{f(n)-1}{k}\log z + 2f(n)\log(t+1) \geq\ \log(f(n)+1) \end{gathered}$$

and a deterministic k-tape TM M can decide whether M_n accepts w of length n by generating a list of all possible state configurations that can be reached from the initial state configuration. This process can be executed in quadratic time relative to the total number of state configurations m, as from any state configuration $c_i \in \{c_1, c_2, ..., c_m\}$, at most all configurations can be reached $c_i \to (c_1 \vee c_2 \vee \ldots \vee c_m)$, resulting in a total of m^2 possibilities. Since the list of all reachable state configurations is no longer than $(d^2)^{f(n)}$ times a constant b for the length of the description of one state configuration, the time is limited by $c^{f(n)}$ for a constant c.

Less obvious is the theorem[6] stating that if f(n) is a full space-constructible function and $f(n) \geq \log_2 n$ then $NSPACE(f(n)) \subseteq DSPACE(f^2(n))$.

Proof. Let M_n be a f(n)-space bounded nondeterministic TM. There exists a constant c such that there are at most $c^{f(n)}$ state configurations for an input word w of length n. If M_n accepts w in the shortest path, so within a sequence of at most $c^{f(n)}$ moves, reaching the maximum of moves when M_n goes through all state configurations toward the final state. If M_n repeats a state configuration, it is not following the shortest path toward the final state. In complexity analyses of nondeterministic TMs, we mostly consider only the shortest paths to the final state.
We define $I_1 \overset{i}{\rightarrow} I_2$ as a path from the state configuration I_1 to the state configuration I_2 in at most 2^i moves. We can determine whether $I_1 \overset{i}{\rightarrow} I_2$ by evaluating if $I_1 \overset{i-1}{\longrightarrow} I'$ and $I' \overset{i-1}{\longrightarrow} I_2$ for every I'. This involves two evaluations of state configuration changes, each in at most 2^{i-1} moves. To achieve this, we use a recursive function 'det', call it with the parameters det(I_1, I', i-1) and det(I', I_2, i-1), and halt the recursive calls when i reaches 0. This function can be implemented on a deterministic TM M, where each recursive call would require space for the call parameters I_1, I_2, I' and i. I_1, I_2 and I' are state configurations, each of maximal length f(n). The parameter i can be encoded in binary representation using at most $m \cdot f(n)$ cells, taking into account that M_n makes a maximum of $c^{f(n)}$ moves. The counter for these moves requires no more than $\lceil \log_2 c^{f(n)} \rceil = f(n) \lceil \log_2 c \rceil$ cells, where $m = \lceil log_2 c \rceil$. Therefore, the parameter block requires a maximum of $(m + 3) \cdot f(n)$ cells, and is recursively used no more than i times. In total, we need space for a maximum of $(m + 3) \cdot f(n) \cdot m \cdot f(n) = (m^2 + 3m)\, f^2(n)$ cells. □

In the above proof of Savitsch's theorem, certain details have been omitted. It's worth noting that the state configuration I' doesn't necessarily represent a reachable configuration in M_n's run but rather an arbitrarily configuration of maximal length f(n). In this proof, we check any configuration that falls within the f(n) length boundary. Thus, we don't need to ensure that I' is a valid state configuration according to the transition relation of M_n. If i = 0, we only need to determine if $I_1 = I_2$ or $I_1 \overset{\mathbf{0}}{\rightarrow} I_2$ (I_2 reachable from I_1 in $2^0 = 1$ step). For this, we need to hold available the transition relation of M_n on a tape of M. However, this takes no more than $(j + 1) \cdot f(n)$ space, where j is the maximal number of subsequent configuration states in the transition relation of the nondeterministic TM M_n.
Additionally, we must keep track of the already examined I's within the recursive calls of 'det'. Here, we can use the canonical order of all state configurations of maximal length f(n) and loop over them, increasing the canonical number by 1 in each for-loop. This, too, requires not more than f(n) space. Consequently, the space requirement here is no larger than O(f(n)), thus we never exceed $O(f^2(n))$.

Finally, we assumed M_n and M to be off-line TMs, utilizing an additional tape solely as an input tape for the word w. Therefore, we were not required to incorporate the input w into each state configuration. Otherwise, in case we utilized usual TMs, we would have needed to impose the condition $f(n) \geq n$ for f(n). It is also important to note that the constant $k = m^2 + 3m$ doesn't lead to a change in complexity class, as for every k > 0, the following holds: if an f(n)-space bounded TM M accepts language L, then a $k \cdot f(n)$-space bounded TM M' also accepts L. This can be achieved by simply merging k cells of M into one cell of M', expanding the set of tape symbols Γ, and modifying the transition function accordingly by introducing new states in Q.

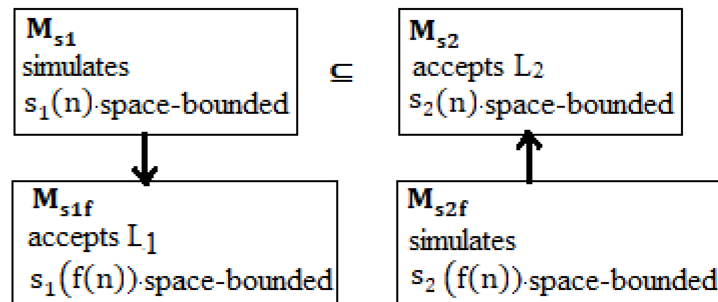
[6] Savitch theorem, formulated by Walter Savitch 1943-2021

5 Chain of complexity class inclusions

The Savitch theorem directly implies, for example, that $NSPACE(n^2) \subseteq DSPACE(n^4)$ or $NSPACE(3^n) \subseteq DSPACE(9^n)$. Before we introduce a more general chain of inclusions involving deterministic and nondeterministic space and time complexity classes, let's briefly discuss the translation lemma. The translation lemma states the following: if $s_1(n)$, $s_2(n)$, and $f(n)$ are full space-constructible functions, and furthermore, $s_2(n) \geq n$ and $f(n) \geq n$, then it holds that $NSPACE(s_1(n)) \subseteq NSPACE(s_2(n)) \rightarrow NSPACE(s_1(f(n))) \subseteq NSPACE(s_2(f(n)))$.

Proof. L_1 is accepted by a nondeterministic $s_1(f(n))$-space bounded TM M_{s1f}. Now, consider $L_2 = \{ x\$^i \mid M_{s1f}$ accepts x, space bounded by $s_1(|x| + i)\}$ with a suffix \$...\$ using i-times a new symbol '\$' not being in the alphabet of L_1. The TM M_{s1} accepting L_2 works as follows: on an input $x\$^i$ M_{s1} marks at first $s_1(|x| + i)$ cells, guaranteed by the full space-constructability of $s_1(n)$. Then M_{s1} simulates M_{s1f} on x and accepts if M_{s1f} accepts x using not more than $s_1(|x| + i)$ cells. With $n = |x| + i$ M_{s1} is thus $s_1(n)$-space bounded. The assumption $NSPACE(s_1(n)) \subseteq NSPACE(s_2(n))$ ensures that L_2 is also accepted by a nondeterministic $s_2(n)$-space bounded TM M_{s2}. Now, we need to construct a TM M_{s2f} which simulates M_{s2} and accepts the original language L_1 within the space of $s_2(f(n))$. At first, M_{s2f} marks $s_2(f(n))$ cells, which is realizable as $s_2(n)$ and $f(n)$ are full space-constructible. It holds $s_2(n) \geq n \rightarrow s_2(f(n)) \geq f(n)$ so M_{s2f} uses not more than $s_2(f(n))$ space. So, M_{s2f} simulates M_{s2} on the input word x from the input $x\$^i$ for M_{s2}. If the head of M_{s2} is within the word x, the head of M_{s2f} is at the same position. If the head of M_{s2} is within the \$...\$ zone, M_{s2f} uses a counter to record the head position of M_{s2}. The counter is at most $\log_2 i$ cells long. M_{s2f} accepts when M_{s2} accepts $x\$^i$. If M_{s2} doesn't accept, the counter of M_{s2f} will increment until it expands over $s_2(f(|x|))$ cells, then M_{s2f} halts.

If x is in L_1, then $x\$^i$ is in L_2 for an i that fulfils the equation $s_1(|x| + i) = s_1(f(|x|))$. As $f(n) \geq n$, so $i = f(|x|) - |x|$ fulfils the equation, and the counter needs not more than $\log_2(f(|x|) - |x|)$ space in such a case. $s_2(f(|x|)) \geq f(|x|) \rightarrow s_2(f(|x|)) - |x| \geq f(|x|) - |x|$, therefore, it is enough space available for the counter. Then, x is accepted by M_{s2f} if $x\$^i$ is accepted by M_{s2} for an $i \geq 0$. It holds then for L_4, the language accepted by M_{s2f}, $L_4 = L_1$ and $L_1 \subseteq NSPACE(s_2(f(n)))$. □



TMs used in the proof of the translation lemma

We outline once more the idea behind the above proof. M_{s1f} is an $s_1(f(n))$-space bounded TM working on $x \in L_1$. M_{s1} simulates M_{s1f} on $x\$^i$. With $n = |x| + i$, M_{s1} is $s_1(n)$-space bounded. Then we assume that $L(M_{s1}) \subseteq L(M_{s2})$ so the language accepted by M_{s1} is also accepted by M_{s2}, which is $s_2(n)$-space bounded. Eventually, we have shown that M_{s2} can be simulated by M_{s1f}, which is an $s_2(f(n))$-space bounded TM, and therefore $L_4 = L(M_{s2f}) = L_1$.

It doesn't matter that M_{s1} works a priori on a longer input word of length n = |x| + i than M_{s1f}, where n = |x|. M_{s1} only needs enough space to simulate M_{s1f}. Correspondingly, M_{s2f}, in its simulation of M_{s2}, should not exceed n = |x| + i space, but a binary counter needs not more space than $\log_2 i \leq i$, so this requirement is fulfilled.

Similar proofs exist for DSPACE, DTIME and NTIME. With the help of the translation lemma, we can, for example, prove that $\mathrm{DTIME}(2^n) \subsetneq \mathrm{DTIME}(n2^n)$. We assume the opposite, $\mathrm{DTIME}(n2^n) \subseteq \mathrm{DTIME}(2^n)$, and show that it leads to a contradiction. With $s_1(n) = n2^n$, $s_2(n) = 2^n$ and $f(n) = 2^n$ we can write:

$$\mathrm{DTIME}(2^n 2^{2^n}) \subseteq \mathrm{DTIME}(2^{2^n})$$

and with $f(n) = n + 2^n$:

$$\mathrm{DTIME}((n + 2^n)2^n 2^{2^n}) \subseteq \mathrm{DTIME}(2^n 2^{2^n})$$

so both together:

$$\mathrm{DTIME}((n + 2^n)2^n 2^{2^n}) \subseteq \mathrm{DTIME}(2^n 2^{2^n}) \subseteq \mathrm{DTIME}(2^{2^n})$$

But theorem T2 above states that there is a language in $\mathrm{DTIME}((n + 2^n)2^n 2^{2^n})$ that is not in $\mathrm{DTIME}(2^{2^n})$ if:

$$\lim_{n\to\infty} \frac{2^{2^n} \log 2^{2^n}}{(n+2^n)2^n 2^{2^n}} = \lim_{n\to\infty} \frac{1}{n+2^n} = 0. \quad ↯$$

If our assumption $\mathrm{DTIME}(n2^n) \subseteq \mathrm{DTIME}(2^n)$ is false, and on the other hand, $\mathrm{DTIME}(2^n) \subseteq \mathrm{DTIME}(n2^n)$, where a class of a smaller time period is always in a class of a bigger time period, then it holds: $\mathrm{DTIME}(2^n) \subsetneq \mathrm{DTIME}(n2^n)$. Note that this can't be shown immediately from theorem T2 as:

$$\lim_{n\to\infty} \frac{2^n \log 2^n}{n\, 2^n} = 1.$$

Also, with the help of the translation lemma, we prove the following theorem for nondeterministic space hierarchy in a polynomial range. For $\alpha > 0$ and $r \geq 0$ it holds:

$$\mathrm{NSPACE}(n^r) \subsetneq \mathrm{NSPACE}(n^{r+\alpha}).$$

Proof. For every non-negative real number r, we can find positive integers s and t such that it holts: $r \leq \frac{s}{t} < \frac{s+1}{t} \leq r + \alpha$. So, it is sufficient to show that for every s and t:

$$\mathrm{NSPACE}(n^{\frac{s}{t}}) \subsetneq \mathrm{NSPACE}(n^{\frac{s+1}{t}}).$$

We show that the opposite, $\mathrm{NSPACE}(n^{\frac{s+1}{t}}) \subseteq \mathrm{NSPACE}(n^{\frac{s}{t}})$, leads to a contradiction. Using the translation lemma and $f(n) = n^{(s+i)t}$, we get:

$$\mathrm{NSPACE}(n^{(s+1)(s+i)}) \subseteq \mathrm{NSPACE}(n^{s\,(s+i)}) \qquad \text{(A)}$$

For $i \geq 1$ we have $s(s + i) \leq (s + 1)(s + i - 1)$ and it holds:

$$\mathrm{NSPACE}(n^{s(s+i)}) \subseteq \mathrm{NSPACE}(n^{(s+1)(s+i-1)}) \qquad \text{(B)}$$

We use alternating (A) for i = s, s-1, …, 0 and (B) for i = s, s-1, …, 1 and get:

NSPACE($n^{(s+1)2s}$) ⊆ NSPACE($n^{(s)2s}$) ⊆ NSPACE($n^{(s+1)(2s-1)}$) ⊆ NSPACE($n^{s(2s-1)}$) ⊆ … ⊆ NSPACE($n^{(s+1)s}$) ⊆ NSPACE(n^{s^2}) → NSPACE($n^{(2s^2+2s)}$) ⊆ NSPACE(n^{s^2}).

Savitch theorem says that NSPACE(n^{s^2}) ⊆ DSPACE(n^{2s^2}), theorem T1 that DSPACE(n^{2s^2}) ⊊ DSPACE(n^{2s^2+2s}) and it holds also DSPACE(n^{2s^2+2s}) ⊆ NSPACE(n^{2s^2+2s}). Altogether then:

$$\text{NSPACE}(n^{(s+1)2s}) \subsetneq \text{NSPACE}(n^{2s^2+2s}) \quad ↯.$$

The assumption NSPACE($n^{\frac{s+1}{t}}$) ⊆ NSPACE($n^{\frac{s}{t}}$) is then false, the inclusion in the other direction is obvious, so eventually:

NSPACE($n^{\frac{s}{t}}$) ⊊ NSPACE($n^{\frac{s+1}{t}}$) for all positive integers s and t. □

Such a dense complexity hierarchy, as determined for nondeterministic space classes, has not yet been proven for time hierarchies. An analogy to the Savitch theorem, which plays a central role in the above proofs, has not been found for time complexities. We will briefly explore time complexity concepts in the next chapter. In a broader view, the following illustrates the subset relations between diverse complexity classes:

DSPACE(log n) ⊆ NSPACE(log n) ⊆ **DTIME**(n^k) **⊆ NTIME**(n^k) ⊆ DSPACE(n^k) ⊆ DTIME(2^{n^k}) ⊆ NTIME(2^{n^k}) ⊆ DSPACE(2^{n^k}) for k ∈ N

or in the usual short notation:

DLOGSPACE ⊆ NLOGSPACE ⊆ **P ⊆ NP** ⊆ PSPACE ⊆ EXPTIME ⊆ NEXPTIME ⊆ EXPSPACE

Furthermore, it holds that D$(EXP)^i$TIME ⊆ N$(EXP)^i$TIME ⊆ $(EXP)^i$SPACE where $(EXP)^i$ represents i times iterated exponential functions. Due to the Savitch theorem with NSPACE(s(n)) ⊆ DSPACE($s^2(n)$), PSPACE = $\bigcup_{k \in N}$ DSPACE(n^k) = $\bigcup_{k \in N}$ NSPACE(n^k) and EXPSPACE = $\bigcup_{k \in N}$ DSPACE(2^{n^k}) = $\bigcup_{k \in N}$ NSPACE(2^{n^k}). Additionally, complexity considerations regarding time below the limit of TIME(n), where n = |x|, have no sense as we need at least n moves to read the input word.

One of the big questions in theoretical computer science is whether any of these relations are proper subsets of the sets to the right, meaning we can replace ⊆ by ⊊. The most prominent among them is the P ≠ NP? question. However, from the time hierarchy theorem and the space hierarchy theorem, we know that P ⊊ EXPTIME, NP ⊊ NEXPTIME and PSPACE ⊊ EXPSPACE.
We do not address complexity class hardness or completeness problems here, nor do we discuss complementary sets. For further exploration of these topics, refer to the relevant literature in theoretical computer science, as exemplarily listed at the end of this paper.

6 The exponential time hypothesis

The *trivial* simulation of a nondeterministic TM by a deterministic TM, trying all possible values of each transition, yields NTIME(t(n)) ⊆ DTIME($2^{t(n)}$). For many problems, it can be

proved that they are equivalent to the satisfiability of 3-CNF (conjunctive normal form) Boolean formulas. In complexity theory, the exponential time hypothesis is an unproven assumption suggesting that the satisfiability of 3-CNF Boolean formulas cannot be solved in sub-exponential time. This implies that all equivalent problems are not solvable in sub-exponential time either.

First, let us briefly explain how a general CNF Boolean formula can be transformed into a 3-CNF Boolean formula[7]. In general CNF formulas, the OR clauses can contain more than 3 variables. Therefore, we need to convert these clauses so they are not longer than 3 variables. We provide an example for this transformation. Let C_1 be a CNF formula $C_1 := (x_1 \lor x_2 \lor x_3 \lor x_4 \lor x_5 \lor x_6 \lor x_7) \land w_1 \land w_2$, where the clauses w_1, w_2 are not longer than 3 variables, so we only need to transform the first clause – if necessary, every other clause longer than 3 can be transformed in the same way. We introduce new variables y_1, y_2, y_3, z_1 and z_2 and write:

$$(x_1 \lor x_2 \lor x_3 \lor x_4 \lor x_5 \lor x_6 \lor x_7) \xrightarrow{\text{SAT}} (x_1 \lor x_2 \lor y_1) \land (x_3 \lor x_4 \lor y_2) \land (x_5 \lor x_6 \lor y_3) \land (x_7 \lor \neg y_1 \lor \neg y_2 \lor \neg y_3) \xrightarrow{\text{SAT}} (x_1 \lor x_2 \lor y_1) \land (x_3 \lor x_4 \lor y_2) \land (x_5 \lor x_6 \lor y_3) \land (x_7 \lor \neg y_1 \lor z_1) \land (\neg y_2 \lor \neg y_3 \lor z_2) \land (\neg z_1 \lor \neg z_2)$$

Note that these formulas are not equivalent, but we do not ask for equivalence here. The second and third formulas are satisfiable only if the original formula is satisfiable. If the original formula is false with $x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = x_7 = 0$, the converted formulas cannot be made true. In this way, we pass on the original satisfiability to the derived formulas. The second derivation is necessary as the last clause is still longer than 3 after the first derivation. Note further that this transformation can be done in polynomial time, as it requires no more than $\log_2 n$ transformation iterations, where n is the number of variables in the original clause. The satisfiability problem of formulas in CNF is thus reduced to 3-CNF satisfiability.

Unfortunately, this reduction approach cannot be extended to 2-CNF formulas, as not every 3-CNF formula can be transformed into a 2-CNF formula in compliance with satisfiability at all. To clarify this, consider the following example - when we talk here about dependency between two Boolean variables, we mean that these variables occur in the same clause of a 2-CNF formula. Let's first examine how it would work if we had two variables, y_1 and y_2, each dependent only on either x_1 or x_2. In this case, we could transform $(x_1 \lor x_2) \xrightarrow{\text{SAT}} (x_1 \lor y_1) \land (x_2 \lor y_2) \land (\neg y_1 \lor \neg y_2)$, keeping the satisfiability property intact, as on $x_1 = x_2 = 0$, the new formula cannot be made true, and on the other hand, when $x_1 \neq x_2$ or $x_1 = x_2 = 1$, we can always find a value assignment that satisfies the resulting 2-CNF.

Now, let's attempt to transform a simple 3-CNF formula with only one clause into a 2-CNF formula $(x_1 \lor x_2 \lor x_3) \xrightarrow{\text{SAT}} \ldots \land (y_1 \lor y_2) \land \ldots$ Assume that the value of a clause $B = (y_1 \lor y_2)$ is dependent on the value assignment of clause $A = (x_1 \lor x_2 \lor x_3)$ because there are clauses of the form $(\neg)x_1/x_2/x_3 \lor (\neg)y_1/y_2$ in the resulting 2-CNF formula. B can't be a constant clause; if it were, always resulting in 1, it could be simply deleted from the derived formula. However, B must be 1 if one of the three variables x_1, x_2 or x_3 is 1, so it is dependent on all three variables from A. Having only two variables in B that are dependent on the values of three variables in A means that at least one variable, y_1 or y_2, is dependent on two variables from A.

[7] see, for example, the Tseytin transformation in the literature or on the internet for how any general Boolean formula can be transformed into a CNF Boolean formula

Without loss of generality, assume that y_1 is dependent on x_1 and x_2. But then, all three possible non-symmetric derivations fail to keep satisfiability:

$$\ldots(x_1 \lor x_2)\ldots \rightarrow \ldots(x_1 \lor y_1) \land (x_2 \lor y_1) \land (y_1 \lor \ldots)\ldots$$
but on $x_1 = x_2 = 0$, the new formula can be made true ↯
$$\ldots(x_1 \lor x_2)\ldots \rightarrow \ldots(x_1 \lor y_1) \land (x_2 \lor y_1) \land (\neg y_1 \lor \ldots)\ldots$$
but on $x_1 \neq x_2$, the new formula can be made false ↯
$$\ldots(x_1 \lor x_2)\ldots \rightarrow \ldots(x_1 \lor y_1) \land (x_2 \lor \neg y_1) \land (y_1 \lor \ldots)\ldots$$
but on $x_1 = 1$ and $x_2 = 0$ the new formula can be made false ↯

Therefore, a new 2-CNF would not reflect the satisfiability of the original formula when one new variable is dependent on two original variables. In the end, only trivial reductions like the following are possible when transferring 3-CNF to 2-CNF: $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor \neg x_3) \overset{\text{SAT}}{\longrightarrow} x_1 \lor x_2$. The satisfaction of 2-CNF is solvable in polynomial time, so a reduction of 3-CNF satisfiability to 2-CNF satisfiability in polynomial time would imply that P = NP. The lack of such a possibility for general transfer of 3-CNF to 2-CNF suggests that probably P ≠ NP.

Problems solvable in P are considered *manageable* by computers; they can provide a solution within a reasonable runtime, even when scaled up with an increasing input length. In our paper “On Computability 1” we considered constraint satisfaction systems and the efficient solvability of problems defined within those systems. We defined the constraints as containing relations of one variable or relations between two variables. Does this mean that problems defined in this way are inherently of polynomial complexity, similar to 2-CNF satisfiability? This question will be addressed in a future paper. For now, we briefly direct our attention to another topic: the limits of computability.

7 Non-computable functions

Shortly after algorithmic procedures began to be formalized, questions arose about the limits of resource consumption by algorithms. For a brief period until the mid-1920s, there was a belief that there was nothing beyond the so-called primitive-recursive functions. Primitive-recursive functions are those that can be constructed from simple basic functions like constant 0 function, projections onto an argument, and successor function through composition and primitive recursion. However, in 1926, Wilhelm Ackermann found a function that did not fit into this definition schema. Its space and time consumption, due to its enormous numerical growth, exceeded those of primitive-recursive functions. Ackermann provided a cumbersome definition that included another auxiliary function. Later, a simplified definition was presented by Rózsa Péter and others. In complexity analysis, the following version is often given, which has the same asymptotic runtime behavior:

$$\begin{aligned} A_1(n) &= 2n && \text{if } n \geq 1, \\ A_k(1) &= 2 && \text{if } k \geq 2, \\ A_k(n) &= A_{k-1}(A_k(n-1)) && \text{if } n \geq 2 \text{ and } k \geq 2 \end{aligned}$$

The functions A_k can be interpreted as a natural continuation of addition, multiplication, exponentiation, etc.:

$$A_1(n) = 2 + 2 + 2 \ldots + 2 = 2n$$
$$A_2(n) = 2 \cdot 2 \cdot 2 \ldots \cdot 2 = 2^n$$

$$A_3(n) = 2^{2^{2^{\cdots^{2}}}}$$

We can prove that A is not a primitive-recursive function by showing that A grows faster than any primitive-recursive function f. Thus, for any given n-variables primitive-recursive function f, there exists an integer k such that for all variables $x_1, \ldots, x_n$, it holds:

$$A_k(\max(x_1, \ldots, x_n)) > f(x_1, \ldots, x_n)$$

Proof. Firstly, we need to prove a lemma that ensures that if the functions $g_1, \ldots, g_m$ and h satisfy the above inequation, then the function f, a functional composition of $g_1, \ldots, g_m$ and h, $f = h \circ (g_1, \ldots, g_m)$, also satisfies the inequation (lemma L1). For that, assume the existence of integers $k_0, k_1, \ldots, k_m$ such that:

$$A_{ki}(\max(x_1, \ldots, x_n)) > g_i(x_1, \ldots, x_n) \text{ for } m \geq i \geq 1$$

and

$$A_{k0}(\max(y_1, \ldots, y_m)) > h(y_1, \ldots, y_m)$$

for all $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$. We define an integer $k = \max(k_0, k_1, \ldots, k_m) + 2$ so that:

$$h(y_1, \ldots, y_m) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) < A_{k0}(\max(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))) < A_{k0}(\max(A_{k1}(\max(x_1, \ldots, x_n)), A_{k2}(\max(x_1, \ldots, x_n)), \ldots, A_{km}(\max(x_1, \ldots, x_n))))$$
$$= A_{k0}(A_{k-2}(\max(x_1, \ldots, x_n))) \leq A_{k-2}(A_{k-2}(\max(x_1, \ldots, x_n))) < A_{k-2}(A_{k-1}(\max(x_1, \ldots, x_n)))$$

-- *because of* $A_k(x) > A_{k-1}(x)$ --

$$= A_{k-1}(\max(x_1, \ldots, x_n) + 1) \leq A_k(\max(x_1, \ldots, x_n))$$

-- *because of* $A_k(x) = A_{k-1}(A_k(x - 1))$ *and* $A_k(x) \geq A_{k-1}(x + 1)$ --

$$\rightarrow \quad f(x_1, \ldots, x_n) < A_k(\max(x_1, \ldots, x_n)) \text{ for all } x_1, \ldots, x_n \qquad \square$$

Secondly, we need the proof of a lemma that states that an (n + 1)-variables function f, defined by the primitive-recursive n-variables function g and the (n + 2)-variables primitive-recursive function h as:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n) \text{ and } f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y))$$

assuming the existence of integers $k_g, k_h \geq 1$such that for all $x_1, \ldots, x_n$, y and z:

$$A_{kg}(\max(x_1, \ldots, x_n)) > g(x_1, \ldots, x_n) \text{ and } A_{kh}(\max(x_1, \ldots, x_n, y, z)) > h(x_1, \ldots, x_n, y, z)$$

implies that:

$A_k(\max(x_1, \ldots, x_n, y)) > f(x_1, \ldots, x_n, y)$ for all $x_1, \ldots, x_n$, y and an integer $k \geq 1$ (lemma L2).

Let's now be $k = \max(k_g, k_h) + 3$ so that:

$$f(x_1, \ldots, x_n, y) = h(x_1, \ldots, x_n, y - 1, f(x_1, \ldots, x_n, y - 1)) = h(x_1, \ldots, x_n, y - 1, h(x_1, \ldots, x_n, y - 2, \ldots, f(x_1, \ldots, x_n, 0))\ldots)) = h(x_1, \ldots, x_n, y - 1, h(x_1, \ldots, x_n, y - 2, \ldots, g(x_1, \ldots, x_n))..)) < A_{kh}(\max(x_1, \ldots, x_n, y - 1, A_{kh}(\max(x_1, \ldots, x_n, y - 2, \ldots, A_{kg}(\max(x_1, \ldots, x_n))\ldots)))) \leq A_{k-3}(\max(x_1, \ldots, x_n, y - 1, A_{k-3}(\max(x_1, \ldots, x_n, y - 2, \ldots, A_{k-3}(\max(x_1, \ldots, x_n, 0, A_{k-3}(\max(x_1, \ldots, x_n)))\ldots)))) \leq A_{k-3}(\max(x_1, \ldots, x_n, y - 1, A_{k-3}(\max (x_1, \ldots, x_n, y - 2, \ldots, A_{k-3}(\max(x_1, \ldots, x_n, 0, A_{k-2}(\max(x_1, \ldots, x_n) - 1)\ldots)))) <$$

-- *because of* $A_k(y) \geq A_{k-1}(y + 1)$ --

$$A_{k-2}(\max(x_1, ..., x_n) + y) \leq A_{k-2}(2 \cdot \max(x_1, ..., x_n, y)) < A_k(\max(x_1, ..., x_n, y))$$

-- because of $A_k(x) = A_{k-1}(A_k(x - 1))$ and $A_k(x) > x$, the last '<' because of $A_k(y) > A_{k-2}(2y)$ --

$$\rightarrow \quad f(x_1, ..., x_n, y) < A_k(\max(x_1, ..., x_n, y)) \text{ for all } x_1, ..., x_n, y \qquad \square$$

With these two lemma results, we can prove the original statement that the Ackermann function A is not primitive-recursive by induction on the number of compositions and primitive recursions. To simplify the notation, we write $\varkappa = \max(x_1, \ldots, x_n)$.

Proof. If the derivation of f requires no compositions or primitive recursions, there are three possible elementary cases:

if f is a constant function whose value is c, we set k = c, so $f(x_1, \ldots, x_n) = c < A_c(\varkappa) = A_k(\varkappa)$, because it holds $k < A_k(y)$ for $y > 1$,

if f is a projection function whose value is x_i, we set k = 1, so $f(x_1, \ldots, x_n) = x_i < A_1(\varkappa)$, because it holds $A_1(y) = 2y$ for $y \geq 1$,

if f is a successor function, we set k = 1, so $f(x) = x + 1 < A_1(x)$ for $x > 1$, because it holds $A_1(y) = 2y$ for $y \geq 1$

The induction step is straightforward. Assuming the statement holds for all functions requiring r compositions and primitive recursions, if f requires r + 1 compositions and primitive recursions, there are two possibilities:

if f is derived through composition from $g_1, \ldots, g_m$ and h, and the hypothesis holds for each of $g_1, \ldots, g_m$, and h, then lemma L1 ensures the existence of a number k such that $f(x_1, \ldots, x_n) < A_k(\varkappa)$

if f is derived through primitive recursions from $g_1, \ldots, g_m$ and h, and the hypothesis holds for each of $g_1, \ldots, g_m$, and h, then lemma L2 ensures the existence of a number k such that $f(x_1, \ldots, x_n) < A_k(\varkappa)$ $\square$

With the finding that function A grows faster than any primitive-recursive function, we can easily prove through contradiction that A is not primitive-recursive.

Proof. Assume A is primitive-recursive. Then, according to the proof above, there must be a k such that $A_k(\max(x,y)) > A_x(y)$ for all x, y, so for k = x = y too. But then $A_k(k) > A_k(k)$. ↯.

It is not possible to determine a priori, in an efficient way, the number of recursive calls needed for a computation of $A_x(y)$. In procedural computing languages, WHILE-loops must be employed to implement the Ackermann function; FOR-loops alone would not suffice. In our paper "On Computability 1", we demonstrated that the procedural formalism with WHILE-loops is equivalent to Turing-computability, which, in fact, encompasses all computable functions. In terms of recursion theory, we use the term 'µ-recursivity' to refer to the class of functions that can be computed by Turing machines. This topic is a subject of computability theory, and we mention it parenthetically in this paper, as our focus here is on the complexity of computational processes.

The space and time consumption of the Ackermann function is tremendous; for instance, the modified Ackermann function defined above results in $2^{128} \approx 3,4028$ e+38 for $A_4(3)$ and the

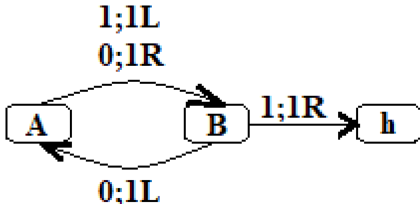
numbers higher than $A_4(4)$ cannot be feasibly computed as the result surpasses the number of atoms in the visible universe, and high numbers naturally result in high space requirements. Despite its staggering resource demands, the Ackermann function remains computable.

In the last section, we delve into functions that are well-defined but not computable. This implies that these functions are not even μ-recursive, meaning there exist no Turing Machines capable of computing them. The following are definitions for such functions, known as generalized busy beaver functions:

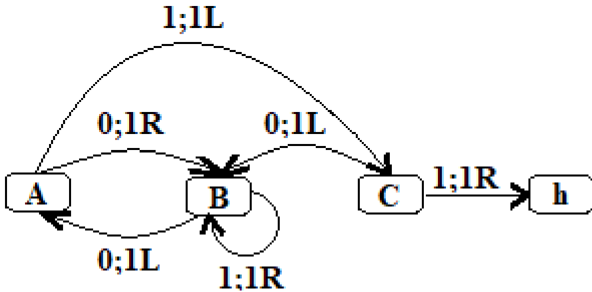
$\Sigma(n, m)$: the largest number of non-zeros printable by an n-state, m-symbol TM started on an initially blank tape before halting

$S(n, m)$: the largest number of steps taken by an n-state, m-symbol TM started on an initially blank tape before halting

Then, it is not feasible to implement universal TMs that can compute these numbers. What we can do, however, is run all implementations of n-state, m-symbol TMs until they either halt or repeat the configuration state. In the deterministic case, we can ascertain that a TM will not halt if it repeats a configuration state. We can then count the non-zeros for the TMs that have halted, determining the highest number as for Σ or the highest number of moves taken as for S. This is feasible for small numbers. For instance, the results for $\Sigma(2, 2)$ or $\Sigma(3, 2)$ are known, along with the TMs that generate them. Below are the definitions of TMs computing $\Sigma(2, 2) = 4$ and $\Sigma(3, 2) = 6$:

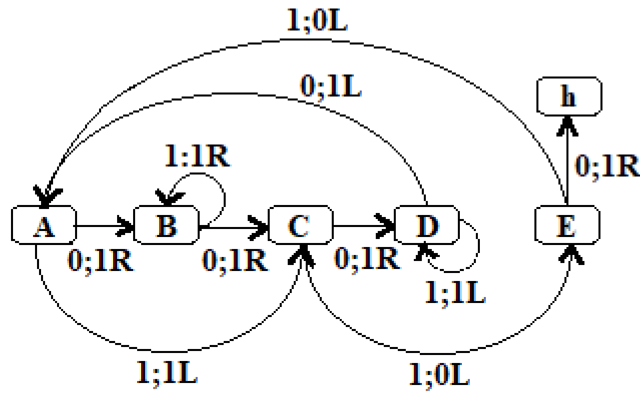


Σ(2, 2) TM



Σ(3, 2) TM

In the picture above, the symbol before ';' represents the read symbol on the tape, and after ';', the symbol to write into the cell. While $\Sigma(4, 2) = 13$, the exact value of the 5-state busy beaver $\Sigma(5, 2)$ is still unknown. The best candidate for this function as of the end of 2023 is the following TM:



Σ(5, 2) TM

We can implement this TM in a short bash script[8] and count the 1s produced by the script in the output file:

```
#!/bin/bash
# busy beaver 5-state
# count afterwards with: $ grep -o 1 bb5.txt|wc -w

declare -A tm
tm[A,0]=1B2
tm[A,1]=1C0
tm[B,0]=1C2
tm[B,1]=1B2
tm[C,0]=1D2
tm[C,1]=0E0
tm[D,0]=1A0
tm[D,1]=1D0
tm[E,0]=1H2
tm[E,1]=0A0

declare -a arr=( $(for i in {1..20000}; do echo 0; done) )

final=0
stat=A
typeset -i n=15000 cell=0 m=0 r=0
output="bb5.txt"
[ -e "$output" ] && rm $output

echo "Turing Machine: ${tm[A,0]} ${tm[A,1]} ${tm[B,0]} ${tm[B,1]} ${tm[C,0]}
${tm[C,1]} ${tm[D,0]} ${tm[D,1]} ${tm[E,0]} ${tm[E,1]}"
echo "---"

# main while loop
while !(( final ))
do

        if [[ $stat == "A" ]]; then (( r++ )); fi
        if !(( m % 100000 )); then echo $m; fi
        if [[ $stat == "E" && ( $cell == 0 ) ]]; then final=1; fi

        arr[n]=${tm[$stat,$cell]:0:1}
        move=${tm[$stat,$cell]:2:1}
        stat=${tm[$stat,$cell]:1:1}
        (( n = n + $move - 1 ))
        cell=arr[n]
        (( m++ ))

done

echo "steps: $m"
echo "visits of A: $r"
echo ${arr[*]} >> $output
exit
```

[8] runnable e. g. on Windows' Cygwin, a collection of GNU and Open Source Linux tools

We allocate here a tape of 20000 cells, which is sufficient for the calculation, and place the r/w-head on the cell n = 15000 in state ‘A’ at the beginning - the tape contains initially only 0s. In the definition of the transition function (tm[X,A] = BY0/2), where X,Y $\in$ Q (set of states) and A,B $\in \Gamma = \{0, 1\}$ (tape symbols), a rightmost ‘0’ indicates a head movement to the left and a ‘2’ a head movement to the right. So, we can use the array index n to specify a new position of the head as n = n + 0/2 - 1. We count the moves and the number of times the initial state 'A' has been visited. The last content of the tape is then saved in the output file. Here are the results:

steps: 47176870
visits of A: 16332
$ grep -o 1 bb5.txt|wc -w
4098

Hence, for the time being, it has been demonstrated in 47176870 head moves that $\Sigma(5, 2)$ is greater than or equal to 4098. Regarding the 5-state busy beaver game, I have read that out of all possible TMs, 21 holdouts are still refusing to halt. The estimation is that by around 2040, all should have either stopped or begun repeating configurations, rendering them unable to accept, and at that point, we will know the exact value of $\Sigma(5, 2)$.

However, must we wait for all of them to stop or repeat configurations until then? Let's consider the TM described above. We can analyse the head movements and the tape content. The TM operates in cycles. In each cycle, the machine adds two 1s to the left edge of the marked block for every triplet of cells in the block. To do so, the machine converts two 1s from a triplet to 0s, creating a pattern like …100100<100>10011… with two trailing 1s at the right end. Successively, it then converts all 0s within the block back to 1s. At the end of each cycle, the machine adds three additional 1s to both the left and right of the already marked block. The marked block grows from n cells to $n + 2 * \lfloor n - 2 \rfloor / 3 + 6$ cells in each cycle. Now, let's explore the break criterion for this process. The TM is in the 'A' state 16332 times during processing. In the ‘A’ state, the head cell and the left neighbour cell must contain a 1, and the second-left neighbour cell must contain a 0 to reach the halting state ‘h’ (…$01^{A}\underline{1}$00100<100>10011…). This is the only situation where the block between the leftmost 1 and the rightmost 1, which includes both, has a length divisible by 3. Therefore, a value of n that satisfies the equation n mod 3 = 0 becomes our break criterion. The following script calculates the value of n for each cycle and breaks the calculation on n mod 3 = 0. After a few initial moves of the TM, we can start the first regular cycle with a value n = 10. The 47176870 moves of the TM defined above took ~1,5 hours of runtime on my laptop, while the cycle loop calculation took only a few milliseconds:

```
#!/bin/bash

final=0
typeset -i A=0 n=10

# main while loop
while !(( final ))
do

	(( A = ( n - 2 ) % 3 ))
	(( n = ( 2 * ( n - 2 - A ) / 3 ) + n + 6 ))
	echo "cycle for n: $n, A: $A"
	echo "---"
	if !(( n % 3 )); then final=1; fi

done

(( result = ( n / 3 ) + 2 ))
echo "Number of 1's: $result"
exit
```

As mentioned before, calculating $\Sigma(n, 2)$ for TMs, where $n > 5$, is practically infeasible. For the time being, $\Sigma(6, 2)$ is estimated to be at least $10\uparrow\uparrow 15$, where $\uparrow$ is the Knuth's up-arrow[9]. $\uparrow\uparrow$ represents here a tetration, an exponentiation tower, so $10\uparrow\uparrow 15$ is:

$$\text{15-times } 10^{10^{\cdots^{10}}}$$

Finally, we provide the proof for non-computability of $S(n, m)$ and $\Sigma(n, m)$ with $m \geq 2$ for a sufficiently large n. Importantly, we use only two tape symbols in the proof, {0, 1}, which implies its applicability to cases with more symbols as well.

Proof. Let's assume that $S(n)$ resp. $\Sigma(n)$ is computable, and contradict it. This implies the existence of TMs M_S and M_Σ, which evaluate $S(n)$ and $\Sigma(n)$ respectively. Given an input of n 1s, they produce $S(n)$ resp. $\Sigma(n)$ 1s and then halt.

Now, let's construct a composite TM M_S' that works as follows: it writes n 1s on an empty tape, doubles the 1s (so on a tape with n 1s, it will produce 2n 1s), evaluates $S(n)$, and finally clears all 1s before halting. Constructing a TM that writes n 1s on an initially blank tape is trivial; it can be done with n states by writing a 1 for each 0, changing the state $q_i \rightarrow q_{i+1}$, and moving the head to the right until $i + 1 = n$, then halting. Now, consider the last three phases of M_S': doubling, evaluating $S(n)$ and cleaning of the tape. If M_S' requires n_0 states for these three phases, then for writing n_0 1s on an initially blank tape plus these three phases, it needs $n_0 + n_0 = N$ states. However, M_S' writes then N 1s on the tape in the first two phases, evaluates $S(N)$, and eventually cleans the tape of the $S(N)$ 1s. The cleaning alone takes $S(N)$ moves, so M_S' needs more than $S(N)$ moves before halting. This contradicts the assertion that M_S computes $S(n)$, the largest number of moves taken by an n-state TM started on an initially blank tape, as M_S' takes more moves for $n = N$. ↯

In analogy, we can create a TM M_Σ', which writes n 1s on an initially blank tape, doubles the 1s, computes $\Sigma(n)$, and in the last step, searches for the first 0 on the tape, replaces it with 1, and halts. If we need n_0 states for the last three phases (doubling, computing $\Sigma(n)$, and increasing the number of 1s on the tape by one), then for writing n_0 1s on an empty tape plus the three phases, $n_0 + n_0 = N$ states should be sufficient. However, M_Σ' writes $\Sigma(N) + 1$ 1s symbols on the tape. Therefore, $\Sigma(n)$ cannot be the largest number of non-zeros printable by an n-state TM started on an initially blank tape, as it's not true for $n = N$. ↯

These elegant contradictions establish that $S(n)$ and $\Sigma(n)$ are not computable for sufficiently large values of n. The presented busy beaver TMs demonstrate the ability to compute $S(n)$ or $\Sigma(n)$ for small values of n - for instance, $\Sigma(3) = 6$, $\Sigma(4) = 13$, $\Sigma(5) \geq 4098$ (with the expectation of determining the exact value in the future). However, a pertinent question emerges: at what threshold of the number of states, which we denote as N_{CL} (Number of States Computability Limit), do $S(n)$ and $\Sigma(n)$ transit into non-computability? This question extends to TMs with more than two tape symbols. Given that the computability of $S(n)$ and $\Sigma(n)$ is equivalent to the halting problem, the identification of N_{CL} would give us an upper limit for decidability as well.

These questions, along with the specifics of cycle loop calculations for busy beaver TMs, will be revisited in a further paper.

---

[9] result found by Pavel Kropitz in 2022

8 Conclusion

Of course, it is not feasible to fully encompass the evolution of theoretical computer science in a short paper, given the span of the past 90 years since the groundbreaking theses of Alan Turing and Kurt Gödel in the 1930s, as Turing, in his work 'On Computable Numbers, with an Application to the *Entscheidungsproblem'* (May, 1936), reformulated Gödel's results from 1931.

Drawing on key theorems developed over the past 90 years, we aim to provide a brief overview of important areas in theoretical computer science, including undecidability problems, complexity classes, nondeterminism, the exponential time hypothesis, and the boundaries of computability. Thanks to these theorems, we understand that constructing a complete and contradiction-free axiomatic theory is unattainable. The Rice's theorem, a cornerstone in computer science, delineates the boundaries of decidability concerning properties of programs. In essence, we are unable to decide any non-trivial property of such a program in advance. In brief, one must let the program do the computation and observe the outcome. The exponential time hypothesis[10], if true, not only implies $P \neq NP$ but also asserts a stronger statement. It implies that various computational problems have reached their optimal programs, as they are largely reducible to each other. Finally, we encounter the boundaries of information processing and, consequently, our capability for comprehension, already by exhausting the resources of time and space. When faced with tricky questions, like those related to the maximal utilization of space or time, we find ourselves surpassing the boundaries of computability.

Naturally, these findings don't merely impact theoretical computer science; they also must extend their influence to our worldview. Concepts like truth, decidability, or cognitive ability are thereby relativized, challenging any claim to their absoluteness. Such is not the nature of our world. We will continue to delve into these topics.

Literature

John E. Hopcroft, Jeffrey D. Ullman “Introduction to Automata Theory, Languages, And Computation“

Chin-Liang Chang, Richard Char-Tung Lee “Symbolic Logic and Mechanical Theorem Proving“

Larry Wos, “Automated Reasoning (33 Basic Research Problems)“

Walter Savitch, “Deterministic simulation of nondeterministic Turing Machines (Detailed Abstract)“

Wolfgang J. Paul “Komplexitätstheorie“

Rainer Klar “Digitale Rechenautomaten“

Richard Blum, Christine Bresnahan “Linux Command Line and Shell Scripting Bible“

[10] postulated by Impagliazzo and Paturi 1999