Elemente der Mathematik 77 (2022), 53–62
DOI 10.4171/EM/439

Polynomial fill-in puzzles or how to make sense of the Cook–Levin theorem

Andreas Müller

Andreas Müller studied mathematics at ETH Zurich and completed a doctorate in algebraic topology in 1990. For the last 14 years, he has been professor of mathematics at the Eastern Switzerland University of Applied Sciences. Telling compelling mathematical stories has become a passion he likes to share with undergraduate students in his mathematical seminar, where he does not shy away even from challenging topics like quantum mechanics and cosmology. He is constantly looking for new engineering applications of abstract mathematics, particularly if Lie groups are involved as well.

1 The problem

Theoretical computer science studies whether a program or more abstractly a Turing machine can decide whether a string formed from characters of a finite set Σ, also called a word, belongs to some language (defined as any subset of Σ^*, the set of all strings built from characters in Σ). Complexity theory tries to figure out how the runtime of such a computation scales with the size of the input word to be tested. It classifies languages as those that can be decided in polynomial time (complexity class P) and those about which

Die Komplexitätstheorie zeigt, dass man Probleme, die man mit einem Computer lösen will, nach dem Rechenaufwand in verschiedene, von der Computer-Hardware unabhängige Klassen einteilen kann. Für den Praktiker sind die Klassen P der effizient lösbaren Problem und NP, die zusätzlich schwierig lösbare, aber effizient verifizierbare Probleme enthält, die Wichtigsten. Der Satz von Cook–Levin hat gezeigt, dass es in der Klasse NP Probleme gibt, auf die jedes andere NP-Problem reduziert werden kann, sie sind also die schwierigsten Probleme in dieser Klasse, sie werden NP-vollständig genannt. Der Beweis verwendet eine Konstruktion, die etwas technisch und sehr speziell anmutet. In diesem Artikel wird gezeigt, dass sie ein Spezialfall eines allgemeiner nützlichen Konzeptes ist, nämlich eines polynomiellen Ausfüllrätsels. Polynomielle Ausfüllrätsel sind fast trivialerweise in der Klasse NP. Viele, wie zum Beispiel Sudoku, sind einem breiten Publikum bekannt. Sie ermöglichen ein intuitiveres Verständnis für diese grundlegenden Konzepte der Komplexitätstheorie.

we only know how to verify membership of a word in a language in polynomial time (class NP). Technically, the class NP consists of the languages decidable in polynomial time on a nondeterministic Turing machine, we will return to this aspect later. There is an extensive literature on complexity theory, [7] is a classic textbook on the subject.

Any problem that is to be solved by a computer can be reformulated as the task of accepting or rejecting some string, usually consisting of a formal problem description and solution. We are thus justified to use the terms language and problem interchangeably below and will switch between these terminologies as we see fit. We will thus use phrases like: problems in P have solution algorithms with runtime polynomial in the problem size n, the execution time is $O(n^k)$, where n is the input size.

The significance of the classes P and NP for the software engineer is that for problems in P, efficient algorithms are often readily available in libraries or can easily be built. While runtime $O(n^k)$ can still be too long for large n and especially for large k, the problem pales in comparison to problems outside of P, where only algorithms with exponential runtime are known at best. Outside of the class NP, not even verification of a solution is possible in polynomial time, these problems are for all practical purposes so complex that one can consider them unsolvable by computers.

The class NP contains all problems with solutions verifiable in polynomial time. This includes of course all problems in P, as their solution can be verified by solving the problem in polynomial time and comparing to the proposed solution, i.e., $\mathrm{P} \subset \mathrm{NP}$. But NP also contains many problems for which no polynomial time solution algorithm is known. Internet security is based on the observation that it is hard to factor products of large prime numbers, but very easy to verify a proposed factorization. This is a property many interesting puzzles published in newspapers share: they are easy to verify, but hard to solve. This, of course, is part of their appeal. The problems in $\mathrm{NP} \setminus \mathrm{P}$ are thus the software practitioner's nemesis: they are efficiently verifiable but one cannot write a scalable program to solve them. Or in business terms: never promise an efficient and cheap solution of such a problem to a customer, or he will be disappointed.

Problems in NP can be compared with respect to their difficulty. Returning to the terminology of languages, we say a mapping $f\colon \Sigma^* \to \Sigma^*$ is a *polynomial reduction* from language A to language B, if $w \in A \Leftrightarrow f(w) \in B$ and if $f(w)$ is computable in time polynomial in $|w|$. We denote the polynomial reduction as $f\colon A \leq_P B$, it is a preorder on NP problems. If language B can be decided in polynomial time, then so can language A: just use the reduction f to convert w into $f(w)$, this takes polynomial time, then decide whether $f(w) \in B$ in polynomial time using the known program for B. We are thus justified to read $A \leq_P B$ as "A is polynomially easier to solve than B". The most difficult problems in NP with respect to this preorder are called NP-complete, every other problem in NP can be reduced in polynomial time to one of these problems. And they certainly scare the hell out of the programmer, as most probably no polynomial algorithm to solve them can be found. They are good candidates for problems in $\mathrm{NP} \setminus \mathrm{P}$.

The question whether $\mathrm{P} = \mathrm{NP}$ is a famous open problem [5]. It isn't that pressing for the software engineer, though. Experience shows that the runtime of any solution algorithm for any NP-complete problem will scale exponentially and thus lead to software with performance bottlenecks and bad user experience. Should it turn out that $\mathrm{P} = \mathrm{NP}$, NP-complete

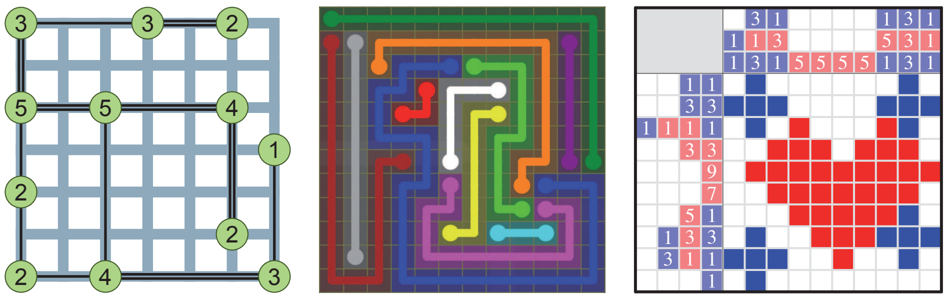


Figure 1. Various fill-in puzzles, from left to right: Hashiwokakero, Flow and Nomogram.

problems would most probably still scale with a rather high power of the input size, so customer satisfaction will not change. As the matter stands, knowing that a problem is NP-complete is as close as we can currently get to knowing that a problem is hard to solve by computer.

The practitioner will hardly ever directly prove that a problem is NP-complete. Instead, she will try to relate the problem at hand to some problem known to be NP-complete from references like [2, 4], using polynomial reduction. In most cases, she will attempt a one-to-one reduction to a reference problem. A large body of reference problems is therefore required and has been built over the years using very ingenious techniques, starting with Richard Karp's famous initial catalog [3]. Sipser [7] beautifully illustrates how many problems are equivalent to problems about graphs. This note intends to add another such device, the *fill-in puzzle*. In addition to the familiar Sudoku puzzle, Figure 1 shows some more examples of this kind of puzzle. All of them are interesting in the sense that solving them is not straightforward, no polynomial solution algorithm is known. They use an array, from a programmer's perspective an even more basic data structure than a graph, though, according to [1], very much related.

From the theoretical perspective we first need to establish that there actually is such a "most difficult" problem. The Cook–Levin theorem below states that satisfiability, i.e.,

$$SAT = \{\varphi = \varphi(x_1, \ldots, x_n) \mid \varphi \text{ is a satisfiable logical formula}\}$$

is such a problem. A formula is called satisfiable if there is a choice of logical values for the variables x_i such that the formula becomes true.

Theorem 1 (Cook–Levin)**.** *Any language that can be decided by a nondeterministic Turing machine in polynomial time can be polynomially reduced to SAT.*

A Turing machine operates on symbols written into cells of an infinite tape, initialized to contain the word w under consideration and filled with blanks symbolized as $\sqcup$ on both sides of w. A read-/write-head controlled by a finite state machine can move over the tape in both directions one cell at a time and it can change the contents of one cell in each step. It stops when it enters any of the two special states q_{accept} and q_{reject}. The word is accepted by the computation if the machine stops in state q_{accept}. This is how the Turing machine decides membership of the word w in the language.

In a deterministic machine, the rules uniquely determine the followup state and the new cell content. In a nondeterministic machine, multiple states or cell contents are consistent with the rules defining the machine. This adds an element of choice and exponentially multiplies the possible paths the computation can take. To simulate such a machine on a deterministic machine, all the possible choices have to be tried out, which usually results in exponentially increasing runtime. To say that such a machine can decide a problem in polynomial time means that the number of steps is bounded polynomially by the input size, provided the machine "knows" which choice to make whenever there is one. So the problem is to find the computation where all the "right" choices were made. This is very simular to the situation the experienced Sudoku player finds himself in. The rules may not uniquely determine the number to place in a cell, some choice needs to be made which may later turn out to be wrong, forcing the player to retrace his steps.

There are standard software solutions for the question asked by the *SAT* problem. A *constraint solver* is a program that finds values for the variables in a predicate that will result in a true value. The `constraint` module in Python [6] offers such a solver and can be used to find solutions to Sudoku problems. To a programmer, a constraint solver is some kind of a pancea: at least in principle, every finite discrete problem can be solved by a constraint solver, albeit very slowly in most cases. The idea that every problem can be reduced to *SAT* is thus not completely surprising to a programmer.

To prove Theorem 1, one has to construct a formula from the nondeterministic Turing machine mentioned in the theorem. The purpose of this note is to show that the well-known but relatively abstract construction shown, e.g., in [7] can be made more intuitive, and as Corollary 4 shows, more practically useful, by the concept of a polynomial fill-in puzzle explained in Section 3. Section 2 uses the well-known Sudoku puzzle to motivate the concept and shows how to derive a formula from such a puzzle. The proof of Theorem 1 can then be completed by showing that any nondeterministic decision problem can be reduced to a polynomial fill-in puzzle, which is done in Section 4.

2 Sudoku

The Sudoku puzzle (Figure 2) consists in filling each square of a $n^2 \times n^2$-grid with one of the numbers $[n^2] = \{1, \dots, n^2\}$ in such a way that no row, no column and no $n \times n$-subsquare contains any number more than once. Furthermore, some squares with grid coordinates $(i, j) \in I \subset [n^2] \times [n^2]$ have a number v_{ij} prescribed, $(i, j) \in I$. The case $n = 3$ is the puzzle played by many people on a regular basis. Let us call

$$\mathit{SUDOKU} = \left\{ \langle n, I, v_{ij} \rangle \;\middle|\; \begin{array}{l} \text{The } n\text{-Sudoku puzzle with squares } (i, j) \in I \text{ prefilled} \\ \text{with values } v_{ij} \text{ can be solved.} \end{array} \right\},$$

where the notation $\langle n, I, v_{ij} \rangle$ indicates a string representation of the triple (n, I, v_{ij}) to bring the problem in line with our terminology of languages and decision problems. Is it possible to convert each Sudoku puzzle into a formula that is satisfiable if and only if the puzzle has a solution? Can a Sudoku puzzle be solved by a constraint solver? Indeed:

Theorem 2. *SUDOKU can be reduced in polynomial time to SAT.*

6	8	2	5	**9**	**1**	**4**	7	3
4	9	1	**7**	3	2	8	6	5
5	7	**3**	6	8	**4**	**9**	1	2
7	3	**4**	1	6	8	2	**5**	9
9	6	5	4	**2**	3	7	8	**1**
1	**2**	8	9	5	7	**6**	3	**4**
8	1	**6**	**3**	4	9	**5**	2	**7**
3	5	9	2	**7**	**6**	1	4	8
2	4	**7**	**8**	**1**	5	3	9	6

Figure 2. The familiar Sudoku puzzle is a special case $n = 3$ of the general $n^2 \times n^2$ puzzle.

Proof. The solution of the Sudoku puzzle is an assignment of values z_{ij} to each cell (i, j) of the grid such that rules of the game are respected. In the following, we first express the rules of the game as a formula ψ in the variables z_{ij}. The second step converts ψ to formula φ with purely logical variables. The Sudoku puzzle has a solution precisely when the variables can be chosen in such a way that the formula becomes true.

Step 1: We construct a formula $\psi(z_{ij} | 1 \leq i, j \leq n^2)$ in the variables z_{ij} with values in $[n^2]$. The prescribed values lead to the constraint

$$\psi_{\text{prescribed}} = \bigwedge_{(i,j)\in I} (z_{ij} = v_{ij}).$$

The rules for rows, columns and subsquares translate to formulae as follows. For each row i, all the z_{ij} are different or

$$\boxed{\begin{array}{l}\text{each symbol exactly}\\ \text{once in each row}\end{array}} \quad \Rightarrow \quad \psi_{\text{rows}} = \bigwedge_{i=1}^{n^2} \Bigl(\bigwedge_{j\neq k} (z_{ij} \neq z_{ik}) \Bigr).$$

Similarly for columns and subsquares:

$$\boxed{\begin{array}{l}\text{each symbol exactly}\\ \text{once in each column}\end{array}} \quad \Rightarrow \quad \psi_{\text{columns}} = \bigwedge_{i=1}^{n^2} \Bigl(\bigwedge_{j\neq k} (z_{ji} \neq z_{ki}) \Bigr),$$

$$\boxed{\begin{array}{l}\text{each symbol exactly}\\ \text{once in each subsquare}\end{array}} \quad \Rightarrow \quad \psi_{\text{subsquares}} = \bigwedge_{\text{subsquares}} \Biggl(\bigwedge_{\substack{(i,j)\neq(k,l)\\ \text{within subsquare}}} (z_{ij} \neq z_{kl}) \Biggr).$$

Then $\psi = \psi_{\text{prescribed}} \wedge \psi_{\text{rows}} \wedge \psi_{\text{columns}} \wedge \psi_{\text{subsquares}}$ is a formula that is satisfiable if and only if there is a solution to the Sudoku puzzle.

Step 2: We convert the formula ψ into a formula using only logical variables. To this end, for each variable z with values in $[n^2]$ we construct n^2 boolean variables $x_1, \dots, x_{n^2}$. If the value of z is c, then x_c is the only one of those variables that is true. We call the x_i *indicator variables* for the value of z.

We have to construct a formula that ensures that precisely one of those variables is true. This is equivalent to saying that if x_c is true, then no other x_d can be true, and this must hold for all $c \in \Sigma$. As a logical formula, this is

$$\varphi_z(x_1, \dots, x_{n^2}) = \bigwedge_{c=1}^{n^2} \Bigl(x_c \Rightarrow \neg \bigvee_{d \neq c} x_d \Bigr) = \bigwedge_{c=1}^{n^2} \Bigl(\neg x_c \vee \neg \bigvee_{d \neq c} x_d \Bigr). \tag{1}$$

Thus to convert each variable z_{ij} into the indicator variables $x_{ij,1}, \dots, x_{ij,n^2}$, we have to add the additional constraint

$$\psi_{\text{logical}} = \bigwedge_{i=1}^{n^2} \bigwedge_{j=1}^{n^2} \varphi_{z_{ij}}.$$

In the first step we have built the formula ψ from terms of the form $z_{ij} = c$ and $z_{ij} = z_{kl}$. These have to be converted into expressions in the indicator variables x_{ijc} as follows:

$$\begin{aligned} z_{ij} = c \quad &\rightarrow \quad x_{ijc} \\ z_{ij} = z_{kl} \quad &\rightarrow \quad \bigvee_{c=1}^{n^2} (x_{ijc} \wedge x_{klc}). \end{aligned} \tag{2}$$

These substitutions convert the formula ψ into a formula of length on the order of at most n^2 times the size of ψ. □

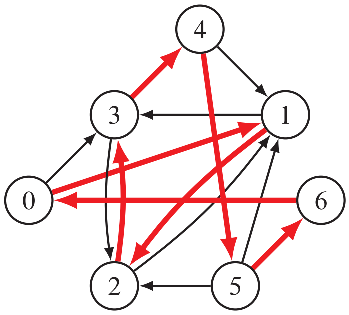
The second step is of course the standard method to get to a logical formula from a formula in character-valued variables also employed by [7]. We will call the terms on the left-hand side in (2) *comparisons*. Comparisons are easy to express in terms of indicator variables.

3 Polynomial fill-in puzzles

The Sudoku problem discussed in the previous section can be generalized to a so-called polynomial fill-in puzzle.

Definition 1. *A* fill-in puzzle *is a game played on an $n \times m$ grid by filling in symbols from an alphabet Σ subject to a set of rules.*

Obviously, Sudoku is such a fill-in puzzle. As a further example, we reformulate the problem to decide whether a directed graph has a closed Hamiltonian path as a polynomial fill-in puzzle (see Figure 3). In a directed graph $G = \{V, E\}$ with $n = |V|$ vertices and edges $E \subset V \times V$, we have to find a closed path visiting each vertex exactly once. To encode the graph, we can use a table of all the edges as in Figure 3. Cell (i, j) is left white if $(i, j) \in E$, also ignoring loops, i.e., edges $(v, v) \in E$. Placing gray dots in white cells so that each row and column contains exactly one dot selects edges in such a way that exactly one edge arrives in each vertex and one leaves. However, these vertices could still form multiple cycles. The selected pairs (v, w) define a mapping $v \mapsto w$. If by iterating this mapping starting from one vertex, all other vertices can be reached, a Hamiltonian path has been found.



	0	1	2	3	4	5	6
0	▨	●	▨		▨	▨	▨
1	▨	▨	●		▨	▨	▨
2	▨		▨	●	▨	▨	▨
3	▨	▨		▨	●	▨	▨
4	▨		▨	▨	▨	●	▨
5	▨			▨	▨	▨	●
6	●	▨	▨	▨	▨	▨	▨

Figure 3 Problem of finding a closed Hamiltonian path in a directed graph (left) encoded as polynomial fill-in puzzle (right). The path is given by the thick arrows in the graph which correspond to dots in the table.

The idea is thus to convert a fill-in puzzle into a formula just like Sudoku, but for this to work it is necessary to control the computational cost of rule checks. There are many ways how to bound rule evaluation, we choose the following definition, which leads us most directly to a proof of Theorem 3 below.

Definition 2. *A fill-in puzzle is called* polynomial*, if the size $|\Sigma|$ of the alphabet depends polynomially on $N = nm$ and the rules can be checked with a formula of length polynomial in N depending only on comparisons.*

The interesting thing about these puzzles is that they can be reduced to *SAT* in a unified way.

Theorem 3. *Any polynomial fill-in puzzle can be reduced to SAT in polynomial time.*

Proof. As in the Sudoku example we proceed in two steps. In the first step we construct a formula ψ using variables $z_{ij} \in \Sigma$ that stand for the symbols to be placed in cell (i, j) of the grid. Such a formula can be written using comparisons only. In the second step we convert ψ to a logical formula φ using indicator variables. For both steps we have to ensure that they are computable in polynomial time.

The fill-in puzzle being polynomial means that rule checking can be done by a formula depending only on comparisons and of length polynomial in N. So we can write the condition that the puzzle can be completed as a single formula $\psi(z_{ij} | i \in [n], j \in [m])$ of length polynomial in N.

For the second step we now replace the comparisons $z_{ij} = c$ and $z_{ij} = z_{kl}$ by the expressions (2) in the indicator variables x_{ijc}. To ensure that for each pair (i, j) only one of the variables x_{ijc} is true, we add the additional constraints

$$\varphi_{\text{logical},ij} = \bigwedge_{c\in\Sigma}\Bigl(\neg x_{ijc} \wedge \neg \bigvee_{d\neq c}(x_{ijd})\Bigr), \qquad \psi_{\text{logical}} = \bigwedge_{i,j} \varphi_{\text{logical},ij}, \tag{3}$$

just as in (1).

The translation of the formula $\psi(z_{ij})$ leads to a new formula $\varphi(x_{ijc}|i \in [n], j \in [m], c \in \Sigma)$ that is at most $O(|\Sigma|^2)$ times longer than $\psi(z_{ij})$, both factors depend polynomially on N. The additional constraints form a formula of length $O(|\Sigma|^2 N)$, again polynomial in N. Both cases show that the resulting formula $\varphi \wedge \varphi_{\text{logical}}$ has size polynomial in N. $\square$

Corollary 4. *Any polynomial fill-in puzzle is in NP.*

The corollary implies that many of the puzzles published in newspapers and as apps for smartphones are in NP. Being hard to solve but easy to verify is of course part of their appeal. Some of them, Sudoku is an example [8], become even more appealing, to the mathematician at least, by the fact that they are also NP-complete.

4 Proof of the Cook–Levin theorem

The proof of Theorem 1 usually uses the compute history. Our proof does the same, but differs in how we obtain it as the solution of a polynomial fill-in problem. The compute history is a list of words built from tape characters Σ and state symbols Q representing the state of the Turing machine and the content of the tape at each step of the computation. A string of the form

$$\dots \sqcup\sqcup\sqcup\sqcup\sqcup a_1a_2a_3 \dots a_{k-1} q a_k \dots a_n \sqcup\sqcup\sqcup\sqcup\sqcup \dots$$

represents a tape containing the word $a_1a_2a_3 \dots a_{k-1}a_k \dots a_n$ of the machine in state q with the read/write-head looking at the cell containing a_k.

Even if there are multiple accepting compute histories of a nondeterministic Turing machine for a word w, the difficult problem still is to find at least one of them.

Proof of Theorem 1. We show that the question whether $w \in \Sigma^*$ is in the language can be reduced to a polynomial fill-in puzzle. If a nondeterministic Turing machine accepts $w \in A$ in time $t(n)$, then the compute history fits into a rectangular array of dimensions $(t(n)+1) \times (2t(n)+n)$, which has to be filled with symbols from $Q \cup \Sigma$. The rules these symbols are subject to are as follows.

1. The first row contains the initial state followed by the input word $w = a_1a_2a_3 \dots a_n$:
$$\dots \sqcup\sqcup\sqcup\sqcup\sqcup q_0 a_1a_2a_3 \dots a_n \sqcup\sqcup\sqcup\sqcup\sqcup \dots$$
which can be described by a formula of comparisons of length $O(t(n))$.
2. The last row with $i = t(n)$ must contain the accepting state q_{accept}. This can be enforced with the formula
$$\bigvee_j (z_{ij} = q_{\text{accept}})$$
which contains $O(t(n))$ comparisons.
3. If a row contains an accepting state, then the machine stops and every subsequent row must be identical. For row i and $i+1$, this can be expressed as
$$\bigvee_j \big((z_{ij} = q_{\text{accept}}) \vee (z_{ij} = q_{\text{reject}})\big) \Rightarrow \bigwedge_j (z_{ij} = z_{i+1,j}).$$
This formula is of size $O(t(n))$ and depends only on comparisons.

4. Each row must follow from the previous row by exactly one Turing machine step. This means that most symbols are the same, just around the state symbol changes are expected.

   More precisely, we can compare length three subwords

   $$z_{ij}z_{i,j+1}z_{i,j+2} \qquad \text{and} \qquad z_{i+1,j}z_{i+1,j+1}z_{i+1,j+2}$$

   at (i, j). There are $M = |\Sigma \cup Q|^6$ such word pairs. Three characters are sufficient to capture a Turing machine transition, but only some of these pairs are consistent with Turing machine transitions, of which there are finitely many. If $p = (abc, def)$ is such a word pair, then the formula

   $$\begin{aligned}\psi_{ijp} = (z_{i,j} = a) \wedge (z_{i,j+1} = b) \wedge (z_{i,j+2} = c)\\ \wedge (z_{i+1,j} = d) \wedge (z_{i+1,j+1} = e) \wedge (z_{i+1,j+2} = f)\end{aligned}$$

   becomes true precisely if at position (i, j) we have a word pair consistent with this particular Turing machine transition. Thus

   $$\psi_{ij} = \bigvee_{\text{Turing machine transitions}} \Bigl(\bigwedge_{p \text{ consistent with transition}} \psi_{ijp}, \Bigr),$$

   a formula containing $6M$ comparisons, becomes true if and only if at position (i, j) we have some Turing machine transition. Finally,

   $$\psi_i = \bigwedge_j \psi_{ij},$$

   a formula containing $O(t(n)) \cdot 6M = O(t(n))$ comparisons, becomes true if and only if row $i + 1$ can be obtained from row i by a Turing machine transition.

   The work to be done to verify that one row follows from the other by a valid Turing machine step is of the order of $O(t(n))$.

It follows that every language in NP can be reduced to a polynomial fill-in puzzle. By Theorem 3, the language also reduces to *SAT*. □

The use of the compute history is well established as a proof technique, embedding it into the framework of fill-in puzzles shows the more general usefulness of the idea. It isn't too far from the computer engineer's intuition, though. Tracing a processor means producing a list of machine states. The chip designer's task is to ensure that subsequent machine states satisfy the constraints of the processor specification. The programmer adds additional constraints via the program executed by the processor. He can use a debugger to trace the machine state, which now includes memory. A problem is solvable in polynomial time if there is a polynomially sized sequence of machine states consistent with all these constraints. The programmer's version of the Cook–Levin theorem is thus the bland observation that debugging programs always reduces to the tedious task of looking at states and verifying constraints. The silver lining is that the size of the constraints is also polynomially bounded.

References

[1] Richard A. Brualdi. *The mutually beneficial relationship of graphs and matrices*. CBMS regional conference series in mathematics 115. American Mathematical Society, 2011. ISBN: 978-0-8218-5315-3.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability, a guide to the theory of NP-completeness*. Bell Telephone Labs, Inc. 1979. ISBN: 978-0-7167-1045-5.

[3] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations*. Ed. by R.E. Miller and J.W. Thatcher. New York: Plenum Press, 1972, pp. 85–103. ISBN: 978-1-4684-2003-6.

[4] *List of NP-complete problems*. May 6, 2019.
URL: `https://en.wikipedia.org/wiki/List_of_NP-complete_problems`.

[5] *P vs NP Problem*. Aug. 2020.
URL: `http://www.claymath.org/millennium-problems/p-vs-np-problem`.

[6] *python-constraint* `1.4.0`. Aug. 24, 2020.
URL: `https://pypi.org/project/python-constraint/`.

[7] Michael Sipser. *Introduction to the theory of computation*. Second edition. Thomson Course Technology, 2006. ISBN: 978-0-534-95097-2.

[8] Dennis Thom. *SUDOKU ist NP-vollständig*. PhD thesis. Universität Stuttgart, 2007.

Andreas Müller
Eastern Switzerland University
of Applied Sciences
Oberseestrasse 10
CH-8640 Rapperswil, Switzerland
e-mail: `andreas.mueller@ost.ch`