The top eight misconceptions about *NP*-hardness

Zoltán Ádám Mann
Budapest University of Technology and Economics
Department of Computer Science and Information Theory
Budapest, Hungary

Abstract: The notions of *NP*-completeness and *NP*-hardness are used frequently in the computer science literature, but unfortunately, often in a wrong way. The aim of this paper is to show the most wide-spread misconceptions, why they are wrong, and why we should care.

Keywords: F.1.3 Complexity Measures and Classes; F.2 Analysis of Algorithms and Problem Complexity; G.2.1.a Combinatorial algorithms; I.1.2.c Analysis of algorithms

Published in *IEEE Computer, volume 50, issue 5, pages 72-79, May 2017*

1. Introduction

Recently, I have been working on a survey about algorithmic approaches to virtual machine allocation in cloud data centers [11]. While doing so, I was shocked (again) to see how often the notions of *NP*-completeness or *NP*-hardness are used in a faulty, inadequate, or imprecise way in the literature. Here is an example from an – otherwise excellent – paper that appeared recently in IEEE Transactions on Computers, one of the most prestigious journals of the field: “Since the problem is *NP*-hard, no polynomial time optimal algorithm exists.” This claim, if it were shown to be true, would finally settle the famous *P versus NP* problem, which is one of the seven Millenium Prize Problems [10]; thus correctly solving it would earn the authors 1 million dollars. But of course, the authors do not attack this problem; the above claim is just an error in the paper.

Some of these errors are simply annoying, but others have profound impact on how algorithmic problems in computer science are approached, and consequently, how well those problems are solved.

The fact that *NP*-hardness is often mentioned in the computer science literature is no accident. Computer science is indeed full of tough algorithmic problems, like hardware verification, multi-processor scheduling, or frequency assignment, just to name a few relevant fields. It is a general experience that these problems are computationally hard: algorithms that solve them exactly tend to be rather slow. In a way, this practical experience is formalized by the notion of *NP*-hardness, and hence it has become a popular term among computer scientists. From this perspective, it is a gratifying development that the notion of *NP*-hardness is now enjoying wide-spread awareness.

On the other hand, there seem to be some common misunderstandings related to *NP*-hardness. Despite the appealing similarity to the informal notion of hardness, the term *NP*-hardness is not exactly the same. It is a well-defined mathematical notion. To claim that a given problem is *NP*-hard, one must prove it. Also, it is crucial to understand what it implies that a given problem is *NP*-hard (and what it does not imply) and most importantly, how *NP*-hard problems can be approached.

The aim of this paper is to showcase some of the bad practices that are currently unfortunately wide-spread in the computer science literature, to show why they are wrong and how these pitfalls can be avoided. By talking about these "sins," my aim is certainly not to rant about those who committed them (I must confess to be also guilty of having committed some of these sins myself), but to improve the general understanding of this important part of our profession. The goal is thus a deeper understanding of *NP*-hardness and its consequences, in particular those small but important details that are often misunderstood.

2. About *NP*-hardness

There are many good books with extensive coverage of *NP*-hardness, see e.g. [8],[12]. Here I give just a very short introduction to the main concepts, without any rigorous definitions or theorems.

One of the key assumptions underlying the theory of computational complexity is that an algorithm can be considered efficient if and only if it runs in polynomial time, that is, the number of steps it takes on inputs of size *N* is not more than $c_1 \cdot N^{c_2}$, where c_1 and c_2 are appropriate constants. For example, the well-known algorithm of Dijkstra to compute shortest paths in a graph makes at most $c \cdot n^2$ steps, where n is the number of vertices of the graph [4]; therefore, it is an efficient algorithm.

The set of decision problems (i.e., problems with yes/no output) for which a polynomial-time algorithm exists, is denoted by *P*. For example, the problem whether a number (the input) is divisible by 3 is in *P*.

The problem class *NP* has a more sophisticated definition. Basically, a decision problem is in *NP* if for all inputs for which the answer is 'yes,' this positive answer can be *verified* efficiently (i.e., in polynomial time), given some extra information, called the *witness* for the given input; the size of the witness must be polynomial with respect to the size of the input. For example, the problem whether a number is composite is in *NP*. The witness for m being composite is a pair of numbers a, b such that $m = a \cdot b$, $1 < a < m$, and $1 < b < m$. If such a witness is available, it can be efficiently verified that $m = a \cdot b$, $1 < a < m$, and $1 < b < m$ indeed hold and thus m is really composite.

As can be seen, the requirements for being in *NP* are weaker than for being in *P*: (i) the solution does not have to be found, only verified; (ii) for the verification, additional information (the witness) can be used. It can be easily proven that each problem in *P* is also in *NP*, or formally: $P \subseteq NP$. It is widely believed that *P* and *NP* cannot be the same, so that *P* is a non-trivial subset of *NP*. This, however, has not been proven yet, despite many efforts. The question whether $P = NP$ or $P \subsetneq NP$ is exactly the famous Millenium Prize Problem mentioned above. It is conjectured that $P \subsetneq NP$, but we cannot be sure.

In order to define *NP*-completeness, one more notion is necessary. A *reduction* of problem L_1 to problem L_2 is a function *f* with the following properties: (i) *f* can be computed in polynomial time, (ii) *f* maps each input of L_1 to an input of L_2, and (iii) for each input *x* of L_1, the answer of L_1 on input *x* is 'yes' if and only if the answer of L_2 on input *f*(*x*) is 'yes.'

As an example, consider the *k*-coloring problem, where *k* is a fixed positive integer. The input is a graph and the question is whether the vertices of the graph can be colored with *k* colors such that adjacent vertices have different colors. A possible reduction from the 3-coloring problem to the 4-coloring problem is the following. From an input of 3-coloring – a graph *G* – we construct an input of 4-coloring – another graph *G*' – by adding a new vertex *v* to it and connecting *v* to each vertex of *G* by an edge. Clearly, this transformation can be carried out in polynomial time. Thus it is clear that the first two properties of the reduction are satisfied. The third is also not hard to prove: if *G* is colorable with 3 colors, then *G*' can be colored with 4 colors using the 3-coloring of *G* and the fourth color for *v*. If, on the other hand, there is a 4-coloring of *G*', then in such a coloring, the vertices of *G* must all have colors that differ from the color of *v*; thus, *G* must be 3-colorable.

Intuitively, the existence of a reduction of problem L_1 to problem L_2 means that L_1 cannot be significantly harder than L_2. Hence the reducibility relation is a useful tool to identify the hardest problems within *NP*. However, finding a reduction from one problem to another and proving the correctness of the reduction is in general a non-trivial task.

Now everything is in place for the main definitions. A problem is *NP-hard*, if all problems in *NP* can be reduced to it. A problem is *NP-complete* if it is *NP*-hard and at the same time also a member of *NP*.

NP-complete problems can be thought of as the most difficult problems within *NP*, in the following sense: if an *NP*-hard problem turns out to be in *P*, then $P = NP$ would follow. In other words, if $P \neq NP$, then there can be no polynomial-time exact algorithm for any of the *NP*-hard problems. (The term 'exact algorithm' means an algorithm that always delivers the correct answer.) In yet other words, determining whether a polynomial-time exact algorithm exists for an *NP*-complete problem is equivalent to the *P* versus *NP* problem.

3. Misconceptions

Most of the misconceptions that I identified can be grouped into two categories: those relating to establishing that a problem is *NP*-hard and those related to the consequences of *NP*-hardness. And the first one is about the name *NP* itself:

Misconception 1. *NP means non-polynomial.*

The name *NP* actually stands for *non-deterministic polynomial-time*. In order to precisely define *P* and *NP*, or even the notion of "algorithm," a formal model of computation is necessary. For this purpose, theoretical computer science traditionally uses so-called Turing machines: abstract automata with formally defined syntax and semantics, named after famous scientist Alan Turing. Turing machines have several different variants, including deterministic and non-deterministic Turing machines. *P* is the class of problems solvable by deterministic Turing machines in polynomial time, while *NP* is the class of problems solvable by non-deterministic Turing machines in polynomial time.

The main message is that *NP* does not mean non-polynomial. To the contrary, it also means polynomial, but in a relaxed sense.

Establishing *NP*-hardness

Proving that a problem *L* is *NP*-hard involves showing a reduction from a known *NP*-hard problem to *L* [8]. Proving that *L* is *NP*-complete additionally involves proving that it is in *NP*. In many practical cases, the latter step is relatively easy. (It does require *L* to be a decision problem though, whereas also optimization problems can be *NP*-hard. Nevertheless, most *NP*-hard optimization problems can be turned easily into an *NP*-complete decision problem. For example, the problem of finding the longest path in a graph can be turned into the following decision problem: the input is a graph and a number *k* and the question is whether there is a path of length at least *k* in the graph.)

In computer science papers, many problems are claimed to be *NP*-hard or *NP*-complete, but few of these claims are actually proven. This is bad practice because not everything that looks to be a difficult problem at first sight is actually *NP*-hard; the only way to make sure that a problem *is NP*-hard is by proving it.

Misconception 2. *If the search space is exponential, then the problem is NP-hard.*

There is some connection between a problem's exponential search space and its being *NP*-hard, but this connection is far from being an implication. For many combinatorial problems, there is a trivial algorithm, which works by exhaustively checking all possibilities, i.e., searching the whole search space. Typically, this trivial algorithm has an exponential runtime. For example, in one possible formulation of the virtual machine allocation problem mentioned in the introduction, the input consists of a set of n virtual machines (VMs) with their sizes and a set of m physical machines (PMs) with their capacities, and the question is whether there is a mapping of VMs to PMs that does not overload any PM. The trivial approach here would be to try all the m^n possible mappings of VMs to PMs and check for each whether it leads to an overload.

For some *NP*-hard problems, we do not know of any exact algorithm that would have a significantly better worst-case runtime than this kind of trivial algorithm.

However, the size of the search space alone does not reveal much about the problem's real complexity. Problems with an exponential search space may admit a polynomial-time algorithm. As an example, consider the problem of computing the shortest path between two vertices in a complete graph with positive edge weights. This problem can be solved in polynomial time [4]. On the other hand, there are exponentially many paths between two vertices in a complete graph, so if we mean by search space the set of all paths between the two vertices, then the search space has exponential size. The trick is that by making use of the combinatorial structure of the problem, it is possible to devise an algorithm to find the optimum efficiently, without having to traverse the whole search space. If, instead of this, we used a brute-force approach to check the whole search space, leading to an exponential-time algorithm, it would just mean that we have not understood the combinatorial structure of the problem well enough, not that the problem were *NP*-hard.

It should also be noted that the search space is typically not an intrinsic property of the problem; rather, defining a search space may be a first step in solving the problem. There can be multiple ways to define the search space, and the size of the search space may also depend on this definition.

Misconception 3. *Adding more constraints to a problem makes it computationally harder. Specifically, adding more constraints to an NP-hard problem results in a problem that is also NP-hard.*

This misconception seems to stem from the erroneous belief that *NP*-hard problems are the ones that are "hard to solve," (i.e., the ones for which it is hard to find a solution) because intuitively it seems clear that adding more constraints makes it harder to solve the problem. However, *NP*-hardness is more subtle. For a decision problem, our aim is not to solve it (find a solution), but to decide solvability (prove that a solution exists or that no solution exists). This is a big difference, because adding more constraints may make it easier to prove that no solution can exist. Similarly, for an optimization problem, the aim is not to solve it as well as possible (find a solution with cost as low as possible) but to solve it optimally (find a solution with minimum cost and prove that no solution with lower cost can exist). This is again a big difference, because adding more constraints may make it easier to prove that no solution with lower cost can exist. Moreover, adding constraints may make the search space significantly smaller or more structured so that a polynomial-time exact algorithm may become possible.

Let us consider an example. In the Subset-Sum problem, the input consists of a set *S* of integers, and the question is whether there is a subset of *S* that adds up to 0. This problem is known to be *NP*-complete. But let us look at the more constrained problem in which the subset with sum 0 must consist of at most 4 numbers. It is easy to devise a polynomial-time algorithm for this latter problem by simply enumerating all subsets of *S* with at most 4 members and checking for each whether its sum equals 0. This is indeed polynomial because it can be easily seen that the number of subsets of size at most 4 is bounded by $c \cdot |S|^4$ for some constant c. Hence the more constrained problem is in *P*, so unless $P = NP$, it cannot be *NP*-complete.

On the other hand, it is also possible that adding constraints makes a problem more complex. As an example, consider the Min-Cut problem. The input is an edge-weighted graph, and the aim is to find a partition of the vertices into two non-empty subsets such that the total weight of the edges between the two subsets is minimum. This problem can be solved in polynomial time. However, if we add the constraint that the difference between the size of the two subsets must be at most 1, then we obtain the Min-Bisection problem, which is known to be *NP*-hard, and so it is much harder than the original problem (unless $P = NP$).

In summary, we must state that adding constraints to a problem affects its complexity in an unpredictable way.

Misconception 4. *Problems that are hard to solve in practice by an engineer are NP-hard.*

This is again an incarnation of the belief that *NP*-hardness is the same as being "hard to solve."

There may be many cases where this holds. Indeed, there are several problems in computer science, e.g., scheduling, resource allocation, verification, packing, and cutting problems, that are both *NP*-hard and hard to solve in practice.

However, there can be differences in both directions. A problem may be *NP*-hard but still typically easy in practice, or vice versa. An example for the first phenomenon is the bin packing problem, where the aim is to pack a set of items of different size into a minimum number of unit-capacity bins. This problem is *NP*-hard, but even simple greedy approaches like the First Fit Decreasing algorithm are known to deliver near-optimal, and in many cases optimal, results. Thus, in many practical situations, this problem can be seen as an easy one, and quite big problem instances can be solved even manually.

On the other hand, there are polynomial-time exact algorithms that are quite complex, in the everyday sense of the word. For example, deciding whether a number is a prime can be done in polynomial time, but with a quite complicated algorithm that is highly non-trivial to implement, let alone to carry out manually, and also not very efficient for large numbers [5].

Misconception 5. *If similar problems are NP-hard, then the problem at hand is also NP-hard.*

This is one the most frequent misconceptions, and a typical reason why people do not feel the need to prove that a given problem is *NP*-hard. For example, there are many different flavors of scheduling problems, and most of them are *NP*-hard. Hence, given a new version of the scheduling problem, one tends to assume that it will be also *NP*-hard. In several cases, such an assumption turns out to be correct, but there is no guarantee for it.

In some cases, small changes in the problem formulation can make the difference between a polynomially solvable problem and an *NP*-hard one. For example, deciding 3-colorability is *NP*-complete, but deciding 2-colorability is in *P*. Some subtle differences can also arise from what is fixed and what is part of the problem's input. For example, consider the problem whether there is a path of length at least k in a graph. If k is a fixed constant, then this problem can be solved in polynomial time. If, however, k is part of the input, then the problem is *NP*-complete.

A similar phenomenon also appears in the context of the previously mentioned VM allocation problem, if an initial allocation of the VMs to the PMs is given, and at most k VMs are allowed to be migrated from one PM to another. If k is fixed, then the problem is in *P*, but if k is part of the input, then the problem is *NP*-hard.

Of particular interest is the situation when the problem at hand is a *special case* of a known *NP*-hard problem, for example because the set of possible inputs is restricted. This does not prove anything because a special case of an *NP*-hard problem may or may not be *NP*-hard.

For example, consider the Boolean satisfiability problem (SAT for short), in which the aim is to decide whether a formula consisting of Boolean variables, negation, disjunction, and conjunction operators evaluates to true for some assignment of logical values to the variables. SAT is known to be *NP*-complete. If the input is restricted to formulae in conjunctive normal form (i.e., a conjunction of clauses, where each clause is a disjunction of literals, and a literal is a variable or its negation), and each clause consists of two literals, the resulting 2-SAT problem is in *P*. If, however, if the allowed clause length is three, the resulting 3-SAT problem is already *NP*-complete. Also, there are several other non-trivial subclasses of formulae for which SAT is known to be in *P* or *NP*-complete, respectively [1].

To see how non-obvious this issue can be in practice, let us consider an example from computer science: the register allocation problem. This arises when a compiler translates a program to machine code and while doing so, must map the variables of the program to registers of the processor. For each pair of variables, it can be determined whether or not they can be mapped to the same register (this depends on whether the so-called live ranges of the variables overlap). This gives rise to a graph (the conflict graph), in which the vertices correspond to the variables, and two vertices are connected by an edge if they are not allowed to be mapped to the same register. Let us focus on the following version of the register allocation problem: given a conflict graph, decide if it can be realized with k registers. It can be seen easily that this problem is equivalent to graph coloring: a set of variables can be mapped to the same register if and only if they can get the same color, thus the variables can be mapped to *k* registers if and only if the conflict graph is *k*-colorable.

Given that *k*-colorability is *NP*-complete, does this mean that the register allocation problem is also *NP*-complete? No! The above argument shows only that register allocation is equivalent to a special case of the *k*-colorability problem: register allocation is equivalent to *k*-colorability of the class of graphs that can represent the conflict graph of a program. Constraining the input of the *k*-colorability problem to a special graph class may make it polynomially solvable, so a crucial question remains before we can claim the *NP*-completeness of register allocation: whether the class of graphs representing conflict graphs of programs is general enough for *k*-colorability to be *NP*-complete on this class of input graphs. As it turns out, the answer to this question depends on the types of programs considered and the way the programs are represented at the compiler stage where register allocation is applied [2]. But for our purposes, the main observation is that casting the register allocation problem in terms of graph coloring is not enough to prove its *NP*-completeness because of the constrained set of possible inputs.

To sum up: the *NP*-completeness of a problem is often not as clear as it may seem at first sight, or may not be true at all. Even if similar problems are known to be *NP*-hard, subtle details – like what is fixed and what is part of the input or limitations on the set of possible inputs – can have significant impact on the complexity of the problem. Therefore, it is good practice to analyze carefully whether the problem at hand is really *NP*-hard and if so, prove it rigorously. In the best case, we may realize that the problem is actually in *P* and end up with a polynomial-time exact algorithm for it.

Consequences of *NP*-hardness

Assuming that $P \neq NP$, the fact that a problem is *NP*-hard implies that it is not in *P*. That is, there can be no polynomial-time exact algorithm for it.

Misconception 6. *NP-hard problems cannot be solved optimally.*

Although there are indeed problems that do not admit any algorithmic solution at all (e.g., the so-called halting problem), but this has nothing to do with *NP*-hardness. Most of the decision and optimization problems, whether *NP*-hard or not, that naturally arise in computer science can be solved by exhaustive search in finite time. This may be very time-consuming, but it is possible.

Misconception 7. *NP-hard problems cannot be solved more efficiently than by exhaustive search.*

What is true is that there are some *NP*-hard problems for which no algorithm is known that would offer a substantially better worst-case performance guarantee than exhaustive search. An example is the already mentioned SAT problem. If n is the number of variables, then the satisfiability problem can be trivially decided after checking all the 2^n possible variable assignments. No exact algorithm is known for the problem with significantly better worst-case runtime, and it is conjectured that no such algorithm exists. This is known as the exponential-time hypothesis (which, if true, would also imply that $P \neq NP$) [9].

However, even for SAT, there are exact algorithms that perform very well on many practical problem instances. As documented by the regular SAT competitions, the best satisfiability solvers can cope with problem instances with hundreds of thousands of variables (stemming from applications like hardware verification) in acceptable time. The trick here is the difference between worst-case and typical-case complexity: whereas the worst-case complexity of all known exact algorithms for SAT is exponential, their typical-case complexity may be much better.

There are also *NP*-hard problems for which we know exact algorithms whose worst-case complexity is much better than that of exhaustive search. An example is the knapsack problem, in which the input consists of n items with weight w_i and value v_i ($i = 1, \cdots, n$) and a capacity C (the capacity of the knapsack), and the aim is to select a subset of the items with total weight at most C and maximum total value. The trivial exhaustive search approach considers all 2^n subsets of the items. However, an algorithm can be easily constructed using dynamic programming that uses only time proportional to $n \cdot C$ [4]. This is a so-called pseudo-polynomial algorithm: its runtime is in general not polynomial with respect to the input's size (because C can be exponentially high), but if C is polynomially bounded, then it is a polynomial-time algorithm, and for many practical settings, it is significantly faster than checking all 2^n possible subsets.

Misconception 8. *For solving NP-hard problems, the only practical possibility is the use of heuristics. (By heuristic, an algorithm is meant that has no formal guarantee on the quality of solutions it finds, just empirical evidence of its usefulness. Popular examples include metaheuristics like simulated annealing and genetic algorithms, as well as problem-specific proprietary heuristics.)*

This misconception is the one that has the most profound impact. Whether it is explicitly stated or not, whether *NP*-hardness is formally proven or not, it is very frequently used as an excuse for resorting to heuristic algorithms with no performance guarantee nor any theoretical underpinning.

Of course, heuristics do play a very important role in computer science and for many algorithmic problems in practice, heuristics are indeed the best known way to solve them. However, the lack of formally proven bounds or other quality guarantees is a strong drawback, which may result in unexpected and unwanted behavior on new inputs on which the heuristic was not tested before. Therefore, heuristics should be used only if all else fails. Even *NP*-hard problems may be approached in several ways other than using heuristics.

Assuming $P \neq NP$, the *NP*-hardness of a problem precludes the existence of any algorithm that

- *quickly* delivers
- the *correct* result
- for *all* inputs.

It may be possible however that there is an algorithm that possesses two of these three desirable properties and also comes close to the third one:

- **Algorithms that *quickly* deliver the *correct* result for *many* inputs**. Algorithms with exponential worst-case complexity but much better typical-case complexity, like the SAT solvers mentioned previously, fall into this category. They always deliver the correct result and are fast in typical cases; thus, they represent an advantageous alternative to heuristics. Other relevant algorithm classes include pseudo-polynomial algorithms, which are fast for inputs in which all numbers are polynomially bounded, and fixed-parameter tractability results, i.e., algorithms whose runtime is polynomial if some parameter of the input is bounded [7].

- **Algorithms that *quickly* deliver *almost correct* results on *all* inputs**. In the case of optimization problems, approximation algorithms belong to this category. They work in polynomial time and their solution is guaranteed to be at most a given fraction worse than the optimum. For example, the previously mentioned First Fit Decreasing algorithm for the bin packing problem is guaranteed to use at most $11/9 \cdot OPT + 6/9$ bins, where OPT denotes the optimum [6]. In the case of decision problems, randomized algorithms with bounded error probability fall into this category. For example, the Miller-Rabin primality test always gives correct answer for primes, and for composite numbers, the probability of a wrong answer after the test has been repeated t times is less than $(1/2)^t$, which quickly converges to zero [5].
- **Algorithms that deliver the *correct* result for *all* inputs *in acceptable time***. Examples of this category include moderately exponential-time algorithms, i.e., algorithms with a runtime of c^n, where c is not much greater than 1. Such algorithms can be very useful in practice if the input instances are not prohibitively large [3].

4. Conclusion

In computer science papers, one often sees a line of thought like: “this problem is related to problem *X*, which is known to be *NP*-hard; therefore, this problem is also *NP*-hard; therefore, we devise a heuristic for it.” Now we can see that such a reasoning is wrong for two reasons. First, it may well be the case that the given problem is not *NP*-hard, e.g., it is equivalent to a polynomially solvable special case of problem *X*. Or maybe the problem formulation can be changed so that it becomes a polynomially solvable special case, for example by explicitly taking into account some additional constraints that arise from practical limitations (e.g., some numbers cannot be arbitrarily small or arbitrarily large). Second, even if the problem is indeed *NP*-hard, there may be several better alternatives than heuristics, e.g., approximation algorithms or pseudo-polynomial algorithms.

Hopefully, a better understanding of what *NP*-hardness actually means and what it does not mean, will lead to better algorithms for the challenging algorithmic problems in computer science.

Acknowledgments

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947).

References

[1] Stefan Andrei, Gheorghe Grigoras, Martin Rinard, and Roland Yap. A hierarchy of tractable subclasses for SAT and counting SAT problems. In *Proceedings of the 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC’09)*, pages 61–68, 2009.

[2] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *19th International Workshop on Languages and Compilers for Parallel Computing*, pages 283–298, 2006.

[3] Nicolas Bourgeois, Bruno Escoffier, and Vangelis Th. Paschos. Efficient approximation of combinatorial problems by moderately exponential algorithms. In *11th Algorithms and Data Structures Symposium (WADS 2009)*, pages 507–518, 2009.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[5] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2006.

[6] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is FFD(I)≤ 11/9OPT(I) + 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11. Springer, 2007.

[7] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[9] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[10] Clay Mathematics Institute. Millennium problems – the P vs. NP problem. `http://www.claymath.org/millennium-problems/p-vs-np-problem`.

[11] Zoltán Ádám Mann. Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms. *ACM Computing Surveys*, 48(1), 2015.

[12] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

Author bio

Zoltán Ádám Mann is associate professor at Budapest University of Technology and Economics. His research addresses algorithmic problems in computer science, with a current focus on optimization problems in cloud computing. He received the PhD degree in computer science from Budapest University of Technology and Economics in 2005.