The Top Eight Misconceptions about NP-Hardness

Zoltán Ádám Mann, Budapest University of Technology and Economics

References to NP-completeness and NP-hardness are common in the computer science literature, but unfortunately they are often in error or misguided. This article explains the most widespread misconceptions, why they are wrong, and why we should care.

Recently, I was working on a survey about algorithmic approaches to virtual machine (VM) allocation in cloud datacenters.[1] While doing so, I was shocked (once again) to see how often the notions of NP-completeness and 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 in 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 Millennium Prize Problems (www.claymath.org/millennium-problems/p-vs-np-problem). Thus, correctly solving it would earn the authors $1 million. But of course, the authors do not attack this problem; the claim is just an error in the paper.

Some of these errors are simply annoying, whereas others profoundly impact how we approach algorithmic problems in computer science and, consequently, how well those problems are solved.

It is no accident that NP-hardness is often mentioned in the computer science literature. Computer science is indeed full of tough algorithmic problems—hardware verification, multiprocessor scheduling, and frequency assignment, just to name a few relevant fields. 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 widespread awareness.

Nevertheless, some common misunderstandings related to NP-hardness persist. Despite the appealing similarity to the informal notion of "hardness," the term NP-hardness is not exactly the same. Rather, NP-hardness 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 (and what it doesn't imply) that a given problem is NP-hard and, most importantly, how we can approach NP-hard problems.

Showcasing some of the bad practices that are unfortunately widespread in the computer science literature, this article explains why they are wrong and how to avoid these pitfalls. By talking about these "sins," my aim is

0018-9162/17/$33.00 © 2017 IEEE

NP-COMPLETE PROBLEMS CAN BE THOUGHT OF AS THE MOST DIFFICULT PROBLEMS WITHIN NP.

not to rant about those who commit them (I confess to having committed some of these 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.

ABOUT NP-HARDNESS

Many good books provide extensive coverage of NP-hardness.[2,3] Thus, here I give just a 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 $c_1 \times N^{c_2}$, where c_1 and c_2 are appropriate constants. For example, the well-known Dijkstra algorithm computes the shortest paths in a graph in at most $c \times n^2$ steps, where n is the number of vertices in the graph.[4] Therefore, it is an efficient algorithm.

The set of decision problems (that is, problems with yes/no output) for which a polynomial-time algorithm exists, is denoted by P. For example, the problem of 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 (in polynomial time), given some extra information, called the *witness* for the given input. The witness's size must be polynomial with respect to the input's size. For example, the problem of whether a number is composite is in NP. The witness for m being composite is a pair of numbers a and b, such that $m = a \times b$, $1 < a < m$, and $1 < b < m$. If such a witness is available, we can efficiently verify that $m = a \times b$, $1 < a < m$, and $1 < b < m$ indeed hold; thus, m is really composite.

These brief definitions show that the requirements for being in NP are weaker than for being in P:

- The solution does not have to be found, only verified.
- The verification can use additional information (the witness).

It can be easily proven that each problem in P is also in NP: $P \subseteq NP$. It is widely believed that P and NP cannot be the same, so P is a proper subset of NP. This, however, has not been proven yet, despite many efforts. This is exactly the famous Millennium Prize Problem mentioned earlier: Does $P = NP$ or $P \subsetneq NP$? It is conjectured that $P \subsetneq NP$, but we cannot be sure.

One more notion is necessary to define NP-completeness. A reduction of problem L_1 to problem L_2 is a function f with the following properties:

- f can be computed in polynomial time;
- f maps each input of L_1 to an input of L_2; and
- for each input x of L_1, the answer of L_1 for input x is "yes" if and only if the answer of L_2 for 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 graph's vertices can be colored with k colors such that adjacent vertices have different colors. Here is a possible reduction from the 3-coloring problem to the 4-coloring problem. From an input of the 3-coloring (a graph G), we construct an input of the 4-coloring problem (another graph G') by adding a new vertex v and connecting v to each vertex of G with an edge. This transformation can be carried out in polynomial time, thus satisfying the first two reduction properties. The third is also not hard to prove: if G is colorable with three colors, then G' can be colored with four colors using the 3-coloring of G and the fourth color for v. On the other hand, in a 4-coloring of G', the vertices of G must all have colors that differ from the color of v; thus, G must be 3-colorable.

Intuitively, 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 for identifying the hardest problems within NP. However, finding a reduction from one problem to another and proving the correctness of the reduction is generally a nontrivial task.

Everything is now 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: if an NP-hard problem turns out to be in P, then P = NP would follow. In other words, if $P \neq NP$, 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.

MISCONCEPTIONS

Most of the misconceptions that I have 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. However, the first one is about the name NP itself.

Misconception 1. NP means nonpolynomial. The name NP actually stands for "nondeterministic polynomial time." To precisely define P and NP, or even the notion of "algorithm," a formal computational model is necessary. For this purpose, theoretical computer science traditionally uses so-called Turing machines: abstract automata with formally defined syntax and semantics, named after Alan Turing. Turing machines have several different variants, including deterministic and nondeterministic. P is the class of problems solvable by deterministic Turing machines in polynomial time, whereas NP is the class of problems solvable by nondeterministic Turing machines in polynomial time.

The main message is that NP does not mean nonpolynomial. On 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.[2] 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 requires L to be a decision problem though, whereas optimization problems can also 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, authors often claim problems 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 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, the problem is NP-hard. For many combinatorial problems, there is a trivial algorithm that works by exhaustively checking all possibilities, exploring the whole search space. Typically, this trivial algorithm has an exponential runtime. For example, in one possible formulation of the VM allocation problem I mentioned in the introduction, the input consists of a set of n VMs with their sizes and a set of m physical machines (PMs) with their capacities. 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 whether each 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. For 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, an exponential number of paths exists between two vertices in a complete graph, so if by "search space" we mean the set of all paths between the two vertices, then the search space has an exponential size. The trick is that by making use of the problem's combinatorial structure, 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 problem's combinatorial structure well enough, not that the problem would be 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 might be a first step in solving the problem. There can be multiple ways to define the search space, and the search space's size 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 those that are "hard to solve" (that is, those for which it is hard to find a solution) because intuitively it seems clear that adding more constraints makes it harder to solve a 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 (determine whether a solution exists). This is a significant difference because adding more constraints might make it easier to determine 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 might make it easier to prove that no solution with a lower cost can exist. Moreover, adding constraints might make the search space significantly smaller or more structured so that a polynomial-time exact algorithm might 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 four numbers. It is easy to devise a polynomial-time algorithm for this latter problem by simply enumerating all subsets of S with at most four members and checking each to determine whether its sum equals 0. This is indeed polynomial because we can easily see that the number of subsets of size at most four is bounded by $c \times |S|^4$ 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 in two nonempty 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, so it is much harder than the original problem (unless P = NP).

These examples show that adding constraints to a problem affects its complexity in an unpredictable way.

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 might be many cases where this holds. Indeed, several problems in computer science (including scheduling, resource allocation, verification, packing, and cutting problems) are both NP-hard and hard to solve in practice.

However, there can be differences in both directions. A problem could 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 sizes 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, we can consider this problem easy, and quite large problem instances can be solved even manually.

On the other hand, there are polynomial-time exact algorithms that are 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 complicated algorithm that is

nontrivial to implement, let alone to carry out manually, and also relatively inefficient for large numbers.[5]

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

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 of 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. However, if 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.

SMALL PROBLEM FORMULATION CHANGES CAN MAKE THE DIFFERENCE BETWEEN A POLYNOMIALLY SOLVABLE PROBLEM AND AN NP-HARD ONE.

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 might or might 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 (that is, 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. However, if the allowed clause length is three, the resulting 3-SAT problem is already NP-complete. Also, there are several other nontrivial subclasses of formulae for which SAT is known to be in P or NP-complete, respectively.[6]

To see how nonobvious 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 program variables to the processor's registers. 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. This problem is clearly 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! This argument shows only that register allocation is equivalent to a special case of the k-colorability problem. That is, register allocation is equivalent to k-colorability of the class of graphs that can represent a program's conflict graph. Constraining the input of the k-colorability problem

to a special graph class might 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 the programs' conflict graphs 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.[7] 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.

These examples show that a problem's NP-completeness is often not as clear as it might seem at first, and it might 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 significantly impact the problem's complexity. 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 might 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 ≠ 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 (such as the halting problem), 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 might be time-consuming, but it is possible.

Misconception 7. NP-hard problems cannot be solved more efficiently than by exhaustive search. 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. Given n variables, SAT 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 ≠ NP).[8]

However, even for SAT, exact algorithms exist that perform 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 might be much better.

There are also NP-hard problems for which we know exact algorithms with a worst-case complexity that 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 ($i = 1, \ldots, n$) and a capacity C (the knapsack's capacity). The aim is to select a subset of the items with a total weight of at most C and maximum total value. The trivial exhaustive search approach considers all 2^n subsets of the items. However, we can easily construct an algorithm[4] using dynamic programming that uses only time proportional to $n \times C$. This is a pseudopolynomial 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, I mean an algorithm 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 or not it is explicitly stated, whether NP-hardness is formally proven or not, it is frequently used as an excuse for resorting to heuristic algorithms with no performance guarantee nor any theoretical underpinning.

ABOUT THE AUTHOR

ZOLTÁN ÁDÁM MANN is an associate professor in the Department of Computer Science and Information Theory at the Budapest University of Technology and Economics. His research addresses algorithmic problems in computer science, with a focus on optimization problems in cloud computing. Mann received a PhD in computer science from the Budapest University of Technology and Economics. Contact him at zoltan.mann@gmail.com.

Of course, heuristics play an 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 that could 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 might be approached in several ways other than using heuristics.

Assuming P ≠ NP, a problem's NP-hardness precludes the existence of any algorithm that

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

It might be possible, however, for an algorithm to possess two of these three desirable properties and come close to the third.

First, an algorithm can *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 pseudopolynomial algorithms, which are fast for inputs in which all numbers are polynomially bounded, and fixed-parameter tractability results—that is, algorithms with a runtime that is polynomial if some parameter of the input is bounded.[9]

Second, an algorithm can *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 $11/9 \times \mathrm{OPT} + 6/9$ bins, where OPT denotes the optimum.[10] 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]

Lastly, an algorithm can deliver the *correct* result for *all* inputs in *acceptable time*. Examples of this category include moderately exponential-time algorithms—that is, algorithms with a runtime of c^n, where c is not much greater than 1. Such algorithms can be useful in practice if the input instances are not prohibitively large.[11]

Computer science papers often contain a line of thought similar to this: "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 might well be the case that the given problem is not NP-hard—for example, 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, such as by explicitly considering additional constraints that arise from practical limitations (some numbers cannot be arbitrarily small or arbitrarily large, for instance). Second, even if the problem is indeed NP-hard, there might be several better alternatives than heuristics, such as approximation algorithms or pseudopolynomial 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 OTKA 108947).

REFERENCES

1. Z.Á. Mann, "Allocation of Virtual Machines in Cloud Data Centers–A Survey of Problem Models and Optimization Algorithms," *ACM Computing Surveys*, vol. 48, no. 1, 2015, article no. 11.
2. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., 1979.
3. C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.
4. T.H. Cormen et al., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
5. R. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, Springer, 2006.
6. S. Andrei et al., "A Hierarchy of Tractable Subclasses for SAT and Counting SAT Problems," *Proc. 11th Int'l Symp. Symbolic and Numeric Algorithms for Scientific Computing* (SYNASC 09), 2009, pp. 61–68.
7. F. Bouchez et al., "Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove?" *Proc. 19th Int'l Workshop Languages and Compilers for Parallel Computing* (LCPC 06), 2006, pp. 283–298.
8. R. Impagliazzo, R. Paturi, and F. Zane, "Which Problems Have Strongly Exponential Complexity?," *J. Computer and System Sciences*, vol. 63, no. 4, 2001, pp. 512–530.
9. J. Flum and M. Grohe, *Parameterized Complexity Theory*, Springer, 2006.
10. G. Dósa, "The Tight Bound of First Fit Decreasing Bin-Packing Algorithm is $FFD(I) \leq 11/9OPT(I) + 6/9$," *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, B. Chen, M. Patterson, and G. Zhang, eds., LNCS 4614, Springer, 2007, pp. 1–11.
11. N. Bourgeois, B. Escoffier, and V.T. Paschos, "Efficient Approximation of Combinatorial Problems by Moderately Exponential Algorithms," *Proc. 11th Algorithms and Data Structures Symp.* (WADS 09), 2009, pp. 507–518.