P versus NP question

Dominik Wisniewski

June 6, 2006

1 Introduction

Many important problems like *3-SAT*, *Clique*, *Vertex cover*, *Travelling Salesman's* problem, etc. have only exponential time solutions. It means that in practice they cannot be solved efficiently except with small input sizes. The *P versus NP* question asks whether the problems can be solved in polynomial time or whether they are intractable by their nature. To outline the *P versus NP* problem more clearly, let us consider as an example the Travelling Salesman's problem (*TSP*). A generic instance of *TSP* consists of a finite set $C = \{c_1, c_2, ..., c_m\}$ of *cities* and *distances* $d(c_i, c_j) \in \mathbb{Z}^+$ for each pair of cities $c_i, c_j \in C$, where $i, j \in \{1, 2, ..., m\}$ and $i \neq j$. A solution to the problem is a sequence $(c_{\Pi(1)}, c_{\Pi(2)}, ..., c_{\Pi(m)})$ of cities from C such that it minimizes the following cost function:

$$\sum_{i=1}^{m-1} d(c_{\Pi(i)}, c_{\Pi(i+1)}) + d(c_{\Pi(m)}, c_{\Pi(1)})$$

A straightforward algorithm for solving the *TSP* problem uses the *Brute Force Search* method. Checking all the possible orderings of cities is time consuming and it is not hard to see that it takes $\Theta(m!)$ time, where m is the number of cities. Using dynamic programming techniques, Bellman [12] showed the the problem can be solved exactly in time $\Theta(m^2 2^m)$. In particular the *P versus NP* question asks whether the *TSP* problem can be solved in polynomial time or whether it is intractable by its nature.

The complexity of a computational problem is defined in a given model of computation. According to Garey and Johnson [6], the most common and widely used model of computation is the Turing machine. In this paper we assume that the Turning machine is the model of computation in which the problem under discussion will be expressed. Once the model of computation is fixed we can define the concept of time complexity which is required to measure efficiency of algorithms.

Definition 1 (Time complexity)**.** *Let* Π *be a problem and let* M *be a Turing machine which solves* Π*. Let us denote by* w *an instance of problem* Π *and by* $t_M(w)$ *the number of transitions (steps) occurring in the computation of machine* M *until a final state is entered (if for a given instance* w *computation of* M *does not terminate then* $t_M(w) = \infty$*).*

Let us denote by $T_M(n)$ *the maximum number of steps required by* M *to finish computation for an input instance* w *of length* n*:*

$$T_M(n) = max\{t_M(w) : |w| = n\}.$$

M *runs in polynomial time if there exists a polynome* $p(n)$ *such that for all* $n \in N$ *the following condition holds:*

$$T_M(n) \leq p(n).$$

Two kinds of Turing machines (deterministic and nondeterministic) will help us define and distinguish two important classes of problems. According to the interpretation given by Garey and Johnson [6], the nondeterministic Turing machine, in comparison to deterministic one, is augmented with a *guessing module*. The solely purpose of the guessing module is to guess the solution to a problem which the machine solves. The next stage of a nondeterministic Turing machine is to check whether the guessed solution is correct. The same definition of time complexity applies two both kinds of Turing machines and in the case of the nondeterministic one it takes into account both guessing and checking stages of the machine.

Now we can give a formal definition of the first important class of problems. It is defined as follows:

$$P = \{\Pi : \textit{there exists a deterministic Turing machine } M \textit{ which solves } \Pi \textit{ in polynomial time}\}.$$

The second important class of problems is defined as follows:

$$NP = \{\Pi : \textit{there exists a nondeterministic Turing machine } M \textit{ which solves } \Pi \textit{ in polynomial time}\}.$$

The P versus NP question can be stated as follows: suppose that we have a problem Π and a non-deterministic algorithm A which solves Π in polynomial time. We ask whether there exists a deterministic algorithm B which solves Π in polynomial time. The inverse question has a positive answer while every problem solvable by a polynomial time deterministic algorithm is also

solvable by a polynomial time nondeterministic algorithm. Intuitively the inclusion $NP \subseteq P$ does not hold because nondeterministic algorithms appear to be more powerful than deterministic ones and we not know general methods for converting the former into the latter. The power of nondeterministic algorithms lies in their ability to check exponential number of possibilities in polynomial time.

In this paper we investigate the *P versus NP* question and we try to answer the question by giving the pros and cons arguments and outlining a set methods which can be used to settle the problem. It is important to mention that in this paper we do not take a concrete standpoint for the *P versus NP* question, but we discuss two points of view to solve the question. In section 2 we introduce several basic definitions required to pose the *P versus NP* question formally, and we divide the *NP*-complete problems into two important categories: weakly and strongly *NP*-complete problems. Section 3 discusses the possible ways and methods to answer the question. Section 4 discusses methods and attempts which have failed in answering the question and probably cannot be used to settle the question. The section also outlines the consequences of possible answers to the *P versus NP* question and tries to give a justifiation for the common bielief that *P!=NP*.

2 NP-complete problems

In this section I will consider a very important subclass of problems of the class *NP*, the class of *NP-complete* problems. I will explain how the class of *NP-complete* problems is important and how it is related to *the P versus NP question*. Next I will give a couple of examples of *NP-complete* problems. Further we will see an important distinction between two kinds of *NP-complete* problems.

2.1 Basic definitions

Informally the class of *NP-complete* problems contains the hardest problems in the class *NP*(Garey and Johnson [6]). To define the class of *NP-complete* problems more formally we need to introduce the concepts of a decision problem and a polynomial transformation between two decisions problems.

We define a computantional problem Π as a mapping from $\Pi : D_\Pi \rightarrow S$, where D_Π is the set of instances and S is the set of solutions. If the set of instances $S = \{0, 1\}$ that the problem Π is a decision problem. A decision problem is no harder than the corresponding optimization(computantional) problem, since we can use the solution to the optimization problem to find a

solution for the decision problem. In many cases it can be shown that a decision problem is no easier than the corresponding optimization problem (for example in case of the *Traveling Salesman* problem). Once the concept of a decision problem has been defined we can give formal definitions of a polynomial transformation and a NP-complete problem based on the definitions given by Garey and Johnson [6].

Definition 2 (Polynomial transformation)**.** *Let* $\Pi_1 : D_{\Pi_1} \rightarrow \{0,1\}$ *and* $\Pi_2 : D_{\Pi_2} \rightarrow \{0,1\}$ *be decision problems.* Π_1 *is polynomially reducible to* Π_2*, denoted by* $\Pi_1 \leq^p \Pi_2$*, if there exists a function* $f : I_1 \rightarrow I_2$ *that satisfies the following conditions:*

1. *f can be computed in polynomial time;*
2. $\forall I \in D_{\Pi_1}\ \Pi_1(x) = 1 \Leftrightarrow \Pi_2(f(x)) = 1$.

Now we can give a formal definition of a *NP-complete* problem.

Definition 3 (NP-complete problem)**.** *Decision problem* Π *is NP-complete if and only if*

1. $\Pi \in NP$*, and*
2. *for each decision problem* $\Pi_1 \in NP\ \Pi_1 \leq^p \Pi$

Polynomial-time transformations give us a powerful tool for proving that a problem belongs to class P. Namely, if Π_1 and Π_2 are decisions problems, $\Pi_1 \leq^p \Pi_2$ and $\Pi_2 \in P$ then $\Pi_1 \in P$. On the other hand, proving that a problem Π is $NP - complete$ according to the definition of $NP - complete$ problems requires us to show that all problems in class NP are polynomially reducible to Π. Utilizing the concept of polynomial-time transformations, in practice we can show that a new problem is $NP - complete$, if we can reduce an existing $NP - complete$ problem to it in polynomial time. Further conclusions (discussed in details by Garey and Johnson [6]) coming out from the definition of polynomial transformations and NP-complete problems are summarized as follows.

1. if Π is *NP-complete* and polynomial-time solvable then *P=NP*;
2. if any problem in *NP* is not polynomial-time solvable, then all NP-complete problems are not polynomial-time solvable (*P!=NP*).

The conclusions stated above can be used to solve the *P versus NP* question. As it was told previously, one can use a polynomial-time reduction and an existing *NP*-complete problem to show that a new problem is *NP*-complete. The only problem which arises here is the need of known at least one *NP*-complete problem. Cook [4] has shown that the *Boolean Satisfiability* problem *(SAT)* is *NP*-complete. Next, a simplified version of *SAT*, the *3-Satisfiability* problem *(3-SAT)* was shown to be *NP*-complete by a polynomial-time reduction from *SAT*(the prove has been outlined by Garey and Johnson [6]). After this, other new problems have been shown to be *NP*-complete, proving their *NP*-completeness by using the polynomial-time reduction approach.

2.2 Weakly and strongly NP-complete problems

Now we will divide the class of *NP*-complete problems into two categories: *NP*-complete problems and strongly *NP*-complete problems, based on te definitions proposed by Garey and Johnson [6]. In the following we will see that the distinction is important for the *P versus NP* question. To show the idea lying behind the distinction let us consider an example problem: the *Partition* problem. The generic instance of the *Partition* problem consists of a finite set A and a total function $s : A \to \mathbb{Z}^+$. We ask if there is a subset $A' \subset A$ such that:

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a).$$

Garey and Johnson [6] used the dynamic programming technique and show that the time complexity of the derived algorithm is $O(nB)$, where n is the number of elements of A and $B = \sum_{a \in A} s(a)$. At first glance this appears to give us a polynomial time algorithm for solving the *Partition* problem and thus proving that $P = NP$. On the other hand they also showed that the length of a problem instance is $O(n \log_2 B)$. Thus nB cannot be bounded by any polynomial function of this quantity, what proves that this is not a polynomial time algorithm.

To give an exact definitions of strong and weak *NP*-complete problems we have to define the following two functions. For each decision problem Π there is an associated length function $Length : D_\Pi \to \mathbb{Z}^+$ and $Max : D_\Pi \to \mathbb{Z}^+$ function, where D_Π is the set of instances of Π. For a given instance $I \in D_\Pi$, *Length* function maps the corresponding string representation I_s of instance I to an integer that corresponds to the length of the instance I_s, and Max function maps I_s to an integer that corresponds to the largest number in instance I_s. To make the definitions of the two functions more clearly and

to allow us to talk about length of an instance instead of its size we assume that there is a encoding scheme e for a decision problem Π, which transforms each instance I of problem Π to its string representation I_s. For example Garey Johnsonfor [6] give the following $Length$ and Max functions for the *Partition* problem:

$$Length(I_s) = |A| + \sum_{a \in A} \lceil \log_2 s(a) \rceil$$
$$Max(I_s) = max\{s(a) : a \in A\}$$

, where $|A|$ denotes the number of elements in A.

According to definitions given by Garey and Johnson [6] an algorithm that solves a decision problem Π is called a *pseudo-polynomial time algorithm*, if its time complexity function is bounded above by a polynomial function of the two variables $Length[I]$ and $Max[I]$, where $Max[I]$ is not bounded by polynome of $Length[I]$. We say that a decision problem Π is a *number problem*, if there exists no polynome p such that $Max[I] \leq p(Length[I])$ for all $I \in D_\Pi$.

Let Π be a number problem, p a polynome, and let us denote by Π_p the subproblem of Π obtained by restricting Π to only these instances for which $Max[I] \leq p(Length[i])$. Garey and Johnson [6] showed that Π_p is not a number problem and if the problem Π is solvable by a pseudo-polynomial algorithm then Π_p is solvable by a polynomial time algorithm (by the definition of pseudo-polynomial time algorithm, the algorithm which solves Π is a polynomial time algorithm for the set Π_p of restricted instances of Π).

A decision problem Π is called *NP-complete in the strong sense*, if Π is *NP*-complete and there exists a polynome p such that Π_p is *NP*-complete. Otherwise Π is called *NP*-complete in the weak sense. It was shown by Gareyand Johnson [6] that, if $P = NP$, then a *NP*-complete problem in the strong sense can be solved by a pseudo-polynomial time algorithm. This observation gives us a tool for deciding if a problem does not have or can have a pseudo-polynomial time algorithm. It is also easy to see that, if decision problem Π is *NP*-complete and Π is not a number problem then Π is *NP*-complete in the strong sense.

Let us sum up our conclusions of how the theory of weakly and strongly *NP*-complete problems can be used to solve the *P versus NP* question. One could find a pseudo-polynomial time algorithm for a strongly *NP*-complete problem, what would prove that $N = NP$ (the consequence comes out from the observation that a pseudo-polynomial time algorithm for a strongly *NP*-complete problem is actually a polynomial time algorithm for the problem). One could also find a pseudo-polynomial time algorithm for a *NP*-complete problem which is not a number problem, what would prove that $P = NP$.

3 Attempts to solve the question

In this section we will see common approaches which can be used to solve the *P versus NP* question and we will see previous attempts which have been made to settle the problem.

3.1 Possible approaches to solve the *P versus NP* question

Let us suppose that $P = NP$ and let us think how somebody could prove it. The one of the most obvious way to do it is to find a polynomial time algorithm for one of almost 1000 known *NP*-complete problems. Then all other *NP*-complete problems can be reduced to the problem and solved in polynomial time too. The original solved problem should be strongly *NP*-complete, or weakly *NP*-complete, assuming that the presented algorithm is not pseudo-polynomial, but *truly* polynomial.

Some standard methods for developing polynomial time algorithms have been used including the greedy method, dynamic programming, reduction to linear programming etc. Programmers and researchers have been trying to find efficient algorithms for *NP*-complete problems over the past 30 years (Cook [7]). Unfortunately their attempts have not brought any success. The best proven upper bound on an algorithm for solving the *3-SAT* problem is approximately $0(1.5^n)$, where n is the number of Boolean variables in the input formula (Cook [2]).

Let us think how somebody could prove that $P! = NP$. To prove that $P! = NP$ one could show that for a given problem, no efficient (polynomial time algorithm) exists. Such methods for limiting the computational complexity of problems from below are known as *lower bounds* . In the last two decades, several powerful techniques for proving lower bounds have been used including diagonalization, discussed by Baker, Gill, and Solovay [5], Boolean circuits, discussed by Shannon [9], and natural proofs, discussed by Razborov and Rudich [13].

3.2 Diagonalization

The *diagonalization* argument was used by Cantor to show that the set of real numbers is uncountable. Next the technique has been successfully used by Turing to show that some problems are unsolvable (including the *Halting Problem*). Thus the following question arises: Can the diagonalization argument be used to show that $PN! = P$? The technique has been used to show that there are problems solvable in exponential time which are not

solvable in polynomial time. These are very hard decidable problems for which super-exponential lower bounds has been proved(Rabin [15]). There are also strong evidences that the diagonalization argument cannot solve the *P versus NP* question as discussed by Baker, Gill, and Solovay [5]. To make it more clear we need to introduce the concept of *relativized computations*.

In relativized computation as discussed by Baker, Gill, and Solovay [5] a Turing machine is provided with a set, called *oracle*, and the ability to determine a membership of an element in the set without any cost. For each oracle, there exists a set of problems which are effectively solvable in the presence of that oracle. Let us denote by P^A the class of problems solvable by a deterministic Truing machine M_A that uses oracle A in polynomial time and by NP^A the class of problems solvable by a nondeterministic Truing machine M^A that uses oracle A in polynomial time. Baker, Gill, and Solovay [5] have shown that there exists an oracle relative to which $P = NP$ and an oracle relative to which $P! = NP$. This observation (called *BGS theorem*) is a strong evidence that diagonalization cannot solve the *P versus NP* question, because otherwise it would contradict the *BGS theorem*. In 1987, Blum and Impagliazzo [11] proved one of the few really strong results about the *P versus NP* problem, namely that for most oracles, $NP! = P$.

3.3 Boolean circuits

Boolean circuits , discussed by Shannon [9], are a model of computation which has particularly been used for proving lower bounds on complexity of functions. The size of a Boolean circuit serves as a measure of complexity of functions and there is a close relationship between the size of a Boolean circuit and the number of steps performed by a Turing machine for a given function f (the relationship between the two computational models exists according to *Church-Turing thesis*).

More formally a Boolean circuit can be viewed as a finite acyclic graph having some number of input nodes, some number of output nodes and some number of inner nodes called *gates*. Each gate corresponds to a Boolean connective $\{AND, OR, NOT\}$. It is clear that a Boolean circuit having n input nodes and m output nodes computes a Boolean function $f : I_n \rightarrow I_m$, where I_k is the set of all finite binary sequences of length k. Shannon [9] showned that most Boolean functions require exponential size circuits. So far, however, we cannot prove such hardness for any explicit function f (e.g. for an *NP*-complete function like SAT). One could also show that there exists a *NP*-complete problem and a Boolean circuit family, such that it computes the problem, and has super-polynomial lower bound, proving concurrently that $P! = NP$. The best lower bound which have been proved so far for problems

in NP using the Boolean Circuit approach is equal to $0(3n)$ (discussed by Blum [10]).

3.4 Natural proofs

A natural proof is the notion introduced by Razborov and Rudich [13] to describe a class of proofs for proving lower bounds on the circuit complexity of a boolean function. The proofs they describe show, either directly or indirectly, that a boolean function has a certain natural combinatorial property. A natural combinatorial property C meets the following conditions:

- Largeness: C contains many functions. It requires that the property hold for a sufficiently large number of the set of all boolean functions.
- Constructivity: One can efficiently verify that a function f is in C. It requires that a property be decidable in polynomial time when the truth table of a boolean function is given as input.

A natural proof is defined as a proof with a natural property C.

They explain that the following common strategy taken to solve the P *versus* NP question cannot succeed. Namely, one approach to show that NP does not have polynomial-size circuits is: Firstly. Find some property C of functions such that SAT is in C. Secondly. Show, using some sort of inductive argument, that no function computable by polynomial-size circuits can have property C. Thirdly. This would imply SAT, cannot have polynomial-size circuits. They give evidence that no proof strategy along these lines can ever succeed, becuase a proof against polynomial-size circuits would break the widely believed assumption, that one-way functions exist, and in particular imply that the discrete logarithm is not hard.

4 Discussion

Scientists, mathematicians and engineers have been trying to resolve the P *versus* NP question since almost 40 years. Lots of attempts have been made and a lot of incorrect solutions have been presented. Using a large number of available methods (e.g. diagonalization and relativization as discussed by Sipser [1]) and trying to express the problem in different computational models (e.g. Boolean circuits as proposed by Shannon [9]) no solution to the question has been found so far. Especially We have already seen in the previous sections that the following methods have been proved not to be able to solve the P *versus* NP question:

- *Diagonalization* : Baker, Gill and Solovay [5] showed the existence of two oracles. According to the first one *P=NP* and according to the second one *P!=NP*. This implies that the question cannot be resolved by diagonalization techniques.

- *Natural proofs* : Razborov and Rudich [13] showed that the question cannot be solved using *natural proofs*. The main part of the problem is that no unnatural proof techniques are known, and no one has been able to find any. Thus resolving the *P versus NP* problem will require more than the use of existing conventional proof techniques.

- *Boolean circuits* - Valiant [14] suggests and explains why direct attempts to prove lower bounds on the time complexities of problems using Boolean circuits may not be an appropriate approach.

One could think that the question is unsolvable. Cook states that [7] lots of scientists and engineer, who are well familiarized with the topic, think we are closer to the solution than further from it. It is also important to notice that most of them think that *P!=NP*. Their belief is partly justified by the practical consequences of proving that *P=NP* and the current state of knowledge. We can consider the practical consequences of proving that *P=NP*. In this case firstly we need to consider the proof of *P=NP*. According to Cook [2], it is very possible that the presented proof is nonconstructive, in the sense that it does not yield an algorithm for any *NP*-complete problem. It can also yield a nonfeasible algorithm for example whose time complexity is $0(n^{100})$. In these two cases, the practical consequences of such a prove would not be striking.

On the other hand if *P=NP* is proved by finding a truly feasible algorithm for an *NP*-complete problem (e.g. the Satisfability problem). Cook [7] states and describes the following consequences of proving that *P=NP*: Firstly, all of the over 1000 already known *NP*-complete problems could be efficiently reduced to the Satisfability problem and then solved efficiently. Secondly, mathematics would be transformed, because computer would be able to find a formal proof of any theorem which has a proof of a reasonable length. Thirdly, the complexity-based cryptography would become impossible. According to Cook [7], the security of the Internet relies on the assumption that a factorization of a large integer and breaking the *DES* (Data Encryption Standard)[1] is very hard to perform. Cook in [4] gives also an example, and considers the situation in which we had an algorithm which solves the

[1] *DES* is a method for encrypting information, still widely used by financial services and other industries worldwide to protect sensitive on-line applications.

3-SAT problem in $0(n^2)$. Then we could use the algorithm to factor 200-digit numbers in a few minutes, what would mean that *DES* encryption algorithm is useless. Friedman [8] summarizes what famous scientists think about solvability of and a possible answer to the *P versus NP* question:

- Jeff Ullman: (Stanford, 2100, *P!=NP*)

  > "I think the problem is comparable to some of the great problems of mathematics that lasted hundreds of years, e.g., *The 4-color theorem*. Thus, I would guess 100 years. I would bet we do not have the techniques, or even names for techniques today."

- Donald Knuth: (retired from Stanford)

  > "It will be solved by either 2048 or 4096. I am currently somewhat pessimistic. The outcome will be truly worst case scenario: namely that someone will prove *P=NP* because there are only finitely many obstructions to the opposite hypothesis; hence there will exist a polynomial time solution to *SAT* but we will never know its complexity."

- Richard Karp: (Berkeley, unsure, *P!=NP*)

  > My intuitive belief is that *P!=NP*, but the only supporting arguments I can offer are the failure of all efforts to place specific *NP-complete* problems in *P* by constructing polynomial-time algorithms. I believe traditional proof techniques will not suffice. My hunch is that the problem will be solved by a young researcher who is not encumbered by too much conventional wisdom about how to attack the problem."

- Juris Hartmanis (Cornell, 2012, *P!=NP*)

  > "I hope that many other separation problems, such as *LOGSPACE, NLOGSPACE, P, PH; P, NP, PH, PSPACE; PSPACE, EXPTIME, NEXPTIME* will be solved once the first major separation result is obtained."

5 Conclusions

In this paper we were invastigating and trying to answer the *P versus NP*. We have also seen that methods including diagonalization, Boolean circuits, natural proofs have failed. We also discussed the common ways and methods which can be used to settle the question.

Mathematicians and engineers have been trying to solve the problem for the last three decades, what makes it one of the greatest unsolved problems of mathematics and computer science. Base on the current state of our knowledge we cannot say when the problem probably is solved or we cannot even say if it can be solved. On the other hand, it is widely believed that $P! = NP$. This belief is mainly based on the suspicion that it not very probable that all *NP*-complete problems, and moreover all the strongly *NP*-complete problems have polynomial time solutions.

References

[1] Michael Sipser: The history and status of the P versus NP question. Proceedings of the twenty-fourth annual ACM symposium on Theory of computing, 1992, pages 603–618.

[2] Stephen Cook: The importance of the P versus NP question. Journal ACM, vol. 50, num. 1, 2003, pages 27–29.

[3] Scott Aaronson: Guest Column: NP-complete problems and physical reality. SIGACT News, vol. 36, num. 1, 2005, pages 30–52.

[4] Stephen A. Cook: The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, 1971, pages 151–158.

[5] T. Baker, J. Gill and R. Solovay: Relativization of the P =?NP question. SICOMP: SIAM Journal on Computing, 4, 1975, pages 431-442.

[6] Michael R. Garey and David S. Johnson: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, 1990.

[7] Stephen A. Cook: The P versus NP Problem. Clay Mathematics, Institute Millennium Prize Problems, http://www.claymath.org/prize problems/pvsnp.pdf, accessed 10/10/2000.

[8] Harvey M. Friedman: Clay Millenium Problem: P = NP. Mathematics Colloquium, Ohio State University, 2005.

[9] C.E. Shannon: The synthesis of two-terminal switching circuits. Bell Systems Technical Journal, 1949, pages 59–98.

[10] N. Blum: A Boolean Function Requiring 3n Network Size. Theoretical Computer Science, Vol. 28, pages 337–345, 1984.

[11] Blum, M., and Impagliazzo: Generic oracles and oracle classes. Proceedings, 28th IEEE Symposium on Foundations of Computer Science, 1987, pages 118–126.

[12] Richard Bellman: Dynamic Programming Treatment of the Travelling Salesman Problem. Journal ACM, vol. 9, num. 1, 1962 , pages 61–63.

[13] Alexander A. Razborov and Steven Rudich: Natural proofs. Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, 1994, pages 204–213.

[14] Leslie G. Valiant: Why is Boolean complexity theory difficult? Poceedings of the London Mathematical Society symposium on Boolean function complexity, Cambridge University Press, 1992, pages 84–94.

[15] M. O. Rabin: Degree of dificutly of computing a function and a partial ordering of recursive sets. Technical Report Number 2, Hebrew University, 1960.