Turing and the Development of Computational Complexity

Steven Homer
Department of Computer Science
Boston University
Boston, MA 02215

Alan L. Selman
Department of Computer Science and Engineering
State University of New York at Buffalo
201 Bell Hall
Buffalo, NY 14260

August 24, 2011

Abstract

Turing's beautiful capture of the concept of computability by the "Turing machine" linked computability to a device with explicit steps of operations and use of resources. This invention led in a most natural way to build the foundations for computational complexity.

1 Introduction

Computational complexity provides mechanisms for classifying combinatorial problems and measuring the computational resources necessary to solve them. The discipline proves explanations of why certain problems have no practical solutions and provides a way of anticipating difficulties involved in solving problems of certain types. The classification is quantitative and is intended to investigate what resources are necessary, lower bounds, and what resources are sufficient, upper bounds, to solve various problems.

This classification should not depend on a particular computational model but rather should measure the intrinsic difficulty of a problem. Precisely for this reason, as we will explain, the basic model of computation for our study is the multitape Turing machine.

Computational complexity theory today addresses issues of contemporary concern, for example, parallel computation, circuit design, computations that depend on random number generators, and development of efficient algorithms. Above all, computational complexity is interested in distinguishing problems that are *efficiently* computable. Algorithms whose running times are n^2 in the size of their inputs can be implemented to execute efficiently even for fairly large values of n, but algorithms that require an exponential running time can be executed only for small values of n. It is common to identify efficiently computable problems with those that have polynomial time algorithms.

A complexity measure quantifies the use of a particular computational resource during execution of a computation. The two most important and most common measures are *time*, the time it takes a program to execute, and *space*, the amount of store used during a computation. However, other measures are considered as well, and we will introduce other resources as we proceed through this exposition.

Computational complexity relies on an expanded version of the Church–Turing thesis [Chu36, Tur36], one that is even stronger than the original thesis. This expanded version asserts that any two general and reasonable models of sequential computation are polynomial related. That is, a problem that has time complexity t on some general and reasonable model of computation has time complexity $p(t)$, for some polynomial p, in the multitape Turing machine model. The assertion has been proven for all known reasonable models of sequential computation including random access machines (RAMS). This thesis is particularly fortunate because of another assertion known as the Cobham–Edmonds thesis [Cob64, Edm65]. The Cobham–Edmonds thesis asserts that computational problems can be feasibly computed on some computational device only if they can be computed in polynomial time. (Truth be told, an n^{100} time algorithm is not a useful algorithm. It is a remarkable phenomenon though, that problems for which polynomial algorithms are found have such algorithms with small exponents and with small coefficients.) Combining these two theses, a problem can be feasibly computed only if it can be computed in polynomial time on some multitape Turing machine.

Computational complexity forms a basis for the classification and analysis of combinatorial problems. To illustrate this, consider the problem of determining whether an arbitrary graph possesses a Hamiltonian Circuit (i.e., a simple cycle that contains every vertex). Currently it is not known whether this problem has a feasible solution, and all known solutions are equivalent to a sequential search of all paths through the graph, testing each in turn for the Hamiltonian property. Recalling that input to a Turing machine is a word over some finite input alphabet, it must be possible to encode data structures such as graphs into words so that what is intuitively the size of the graph differs from the length of the input by no more than a polynomial. In fact, it is easy to do so. Then, we demand that the theory is capable of classifying the intrinsic complexity of this problem in a precise way, and is capable of elucidating the difficulty in finding an efficient solution to this problem.

Finally we should mention the seminal paper of Hartmanis and Stearns, "On the Computational Complexity of Algorithms" [HS65], from which the discipline takes its name. This paper formulated definitions of time and space complexity on multitape Turing machines and proved results demonstrating that with more time, more problems can be computed. It was a fundamental step to make complexity classes, defined by functions that bound the amount of resources use, the main subject of study. More abstract definitions of computability could not have offered natural guidance to an intuitively satisfying and practical formulation of computational complexity.

The Turing machine beautifully captured the discrete step by step nature of computation. Furthermore, this machine enabled the definition and measurement of time and space complexity in a natural and precise manner amenable to quantifying the resources used by a computation. This natural precision of Turing machines was fundamental in guiding the originators of this field in their fundamental definitions and first results which set the tenor for this research area up to the present day. Other, more abstract definitions of computability, while having advantages of brevity and elegance, could not and did not offer the needed precision or guidance toward this intuitively satisfying and practical formulation of complexity theory.

2 Modes of Computation

A computation of a Turing machine is a sequence of moves, as determined by its transition function. Ordinarily, the transition function is single-valued. Such Turing machines are called *deterministic* and their computations are sequential.

A *nondeterministic* Turing machine is one that allows for a choice of next moves; in this case the transition function is multivalued. If M is a nondeterministic Turing machine and x is an input word, then M and x specify a *computation tree.* The root of the tree is the *initial configuration* of M. The children of a node are the configurations that may follow in one move. A path from the root to a leaf is a computation, and a computation is accepting if the leaf is an accepting configuration.

Given a Turing machine M, the language $L(M)$ *accepted* by M is the set of words such that *some* computation of M on x is accepting. (Observe that this definition is meaningful for both the deterministic and the nondeterministic modes.) It is easy to see, by a breadth-first search of the computation tree, that for every nondeterministic Turing machine, there is a deterministic Turing machine that accepts the same language. The difficulty for computational complexity is that this search technique results in a deterministic machine that is exponentially slower than the original nondeterministic one.

The advantage of nondeterministic Turing machines is that they are a very useful mode for classification of computational problems. For example, whereas it is not known whether there is a deterministic polynomial time-bounded Turing machine to solve (an encoding of) the Hamiltonian Circuit problem, it is easy to design a nondeterministic polynomial time-bounded Turing machine that solves this problem. This is what makes it possible to give an exact classification of the Hamiltonian Circuit problem. Indeed, this problem is known to be NP-complete, which places it among hundreds of other important computational problems whose deterministic complexity is still open.

Nondeterminism is first considered in the classic paper of Rabin and Scott on finite automata [RS59]. We will return to nondeterminism and to a precise definition of NP-complete in a later section.

3 Complexity Classes and Complexity Measures

We assume that a multitape Turing machine has its input written on one of the work tapes, which can be rewritten and used an as an ordinary work tape. The machine may be either deterministic or nondeterministic. Let M be such a Turing machine, and let T be a function defined on the set of natural numbers. M is a $T(n)$ *time-bounded* Turing machine if for every input of length n, M makes at most $T(n)$ moves before halting. If M is nondeterministic, then every computation of M on words of length n must take at most $T(n)$ steps. The language $L(M)$ that is accepted by a deterministic $T(n)$ time-bounded M has *time complexity* $T(n)$. By convention, the time it takes to read the input is counted, and every machine is entitled to read its input.

Denote the length of a word x by $|x|$. We might be tempted to say that a nondeterministic Turing machine is $T(n)$ time-bounded if for every input word $x \in L(M)$, the number of steps of the shortest accepting computation of M on x is at most $T(|x|)$. It turns out that the formulations are equivalent for the specific time bounds that we will write about. But, they are not equivalent for arbitrary time bounds.

A complexity class is a collection of sets that can be accepted by Turing machines with the same resources. Now we define the time-bounded complexity classes: Define DTIME($T(n)$) to be the set of all languages having time-complexity $T(n)$. Define NTIME($T(n)$) to be the set of all languages accepted by nondeterministic $T(n)$ time-bounded Turing machines.

In order to define space complexity, we need to use off-line Turing machines. An *off-line* Turing machine is a multitape Turing machine with a separate read-only input tape. The Turing machine can read the input, but cannot write over the input. Let M be an off-line multitape Turing machine and let S be a function defined on the set of natural numbers. M is an $S(n)$ *space-bounded* Turing machine if for every word of length n, M scans at most $S(n)$ cells on any storage tape. If M is nondeterministic, then every computation must scan no more than $S(n)$ cells on any storage tape. The language $L(M)$ that is accepted by an $S(n)$ deterministic space-bounded Turing machine has *space-complexity* $S(n)$.

Observe that the space taken by the input is not counted. So space-complexity might be less than the length of the input or substantially more. (Also, in this manner we reflect the space required to carry out the computation.) One might be tempted to say that a nondeterministic Turing machine is $S(n)$ space-bounded if for every word of length n that belongs to $L(M)$, there is an accepting computation that uses no more than $S(n)$ work cells on any work tape. The comment made for time-bounds applies here as well.

Now, we define the space-bounded complexity classes: Define DSPACE($S(n)$) to be the set of all languages having space-complexity $S(n)$. Define NSPACE($S(n)$) to be the set of all languages accepted by nondeterministic $S(n)$ time-bounded Turing machines.

We note that the study of time complexity begins properly with the paper [HS65] of Hartmanis and Stearns and the study of space complexity begins with the paper [HLS65] of Hartmanis, Lewis and Stearns. These seminal papers introduced some of the issues that remain of concern even today. These include time/space trade-offs, inclusion relations, hierarchy results, and efficient simulation of nondeterministic computations.

We will be primarily concerned with classes defined by logarithmic, polynomial and exponential functions. As we proceed to relate and discuss various facts about complexity classes in general, we will see what impact they have on the following list of *standard* complexity classes. These classes are well-studied in the literature and each contains important computational problems. The classes are introduced with their common notations.

1. L = DSPACE($\log(n)$);
2. NL = NSPACE($\log(n)$);
3. POLYLOGSPACE = $\bigcup\{$DSPACE($\log(n)^k$) $\mid k \geq 1\}$;
4. DLBA = $\bigcup\{$DSPACE(kn) $\mid k \geq 1\}$;
5. LBA = $\bigcup\{$NSPACE(kn) $\mid k \geq 1\}$;
6. P = $\bigcup\{$DTIME(n^k) $\mid k \geq 1\}$;
7. NP = $\bigcup\{$NTIME(n^k) $\mid k \geq 1\}$;
8. PSPACE = $\bigcup\{$DSPACE(n^k) $\mid k \geq 1\}$;
9. E = $\bigcup\{$DTIME(k^n) $\mid k \geq 1\}$;

10. NE $= \bigcup\{\text{NTIME}(k^n) \mid k \geq 1\}$;

11. EXP $= \bigcup\{\text{DTIME}(2^{p(n)}) \mid\ p$ is a polynomial$\}$;

12. NEXP $= \bigcup\{\text{NTIME}(2^{p(n)}) \mid\ p$ is a polynomial$\}$;

The class NL contains the problem of determining for arbitrary directed graphs G and vertices u and v, whether there is a path from u to v. This problem is not known to belong to L. The latter class contains the restriction of this problem to the case that no vertex has more than one directed edge leading from it. [Jon73, Jon75]. The famous class P is identified with the class of feasibly computed problems. The corresponding nondeterministic class NP will be discussed in a later section.

The class denoted by LBA is so named because it is known to be identical to the class of languages accepted by *linear-bounded automata*, otherwise known as the context-sensitive languages [Myh60, Kur64]. This comment explains the notation for the corresponding deterministic class as well.

E characterizes the complexity of languages accepted by writing push-down automata [Mag69], and NE characterizes the complexity of the spectrum problem in finite model theory [JS74]. PSPACE contains many computational games, such as HEX [ET76].

4 Basic Results

Now we survey several different types of results that apply to all complexity classes. These results demonstrate that the definitions of these classes are invariant under small changes and they prove a variety of relationships between these classes. These are for the most part early results, and their proofs typically involve intricate Turing machine simulations.

The concepts of reducibility used in comparing combinatorial problems, including in definitions of completeness, and in related proof methods such as simple diagonalization and simulation common to complexity theory, arose as generalizations of fundamental concepts of computability theory. These ideas originated with the founders of computability theory prominently including Gödel, Church, Turing, Kleene, Post and others. Alan Turing in particularly defined his eponymous machines, embodying the specific mode of computation most amenable to the application of these ideas (as mentioned above), and also introduced the oracle Turing machine [Tur50], which enabled a machine-based definition of the most general notion of relative computability. While other, machine-independent concepts give rise to the same general results, the use of Turing's oracle Turing machines allow for a precise quantified measurement of time, space, nondeterminism, randomness, etc., which provides the more exact and fine-grained notions of computation most applicable to computer science.

4.1 Linear Compression and Speedup

The first results are of the form: if a language can be accepted with resource $f(n)$, then it can be accepted with resource $cf(n)$, for any $c > 0$. These results justify use of "big-oh" notation for complexity functions.

The Space Compression Theorem [HLS65] asserts that if L is accepted by a k-tape $S(n)$ space-bounded Turing machine, then for any $c > 0$, L is accepted by a k-tape $cS(n)$ space-bounded Turing machine. If the $S(n)$ space-bounded Turing machine is nondeterministic, then so is the $cS(n)$ space-bounded Turing machine. A simple machine simulation proves the result. As a corollary, it follows that DSPACE($S(n)$) = DSPACE($cS(n)$) and NSPACE($S(n)$) = NSPACE($cS(n)$) , for all $c > 0$.

Linear speedup of time is possible too, but not quite as readily as is linear compression of space. The Linear Speedup Theorem [HS65] asserts that if L is accepted by a k-tape $T(n)$ time-bounded Turing machine, $k \geq 1$, and if

$$\inf_{n\to\infty} T(n)/n = \infty, \tag{1}$$

then for any $c > 0$, L is accepted by a k-tape $cT(n)$ time-bounded Turing machine. Thus, if $\inf_{n\to\infty} T(n)/n = \infty$ and $c > 0$, then DTIME($T(n)$) = DTIME($cT(n)$)

Condition 1 stipulates that $T(n)$ grows faster than every linear function. The Linear Speedup Theorem does not apply if $T(n) = cn$, for some constant c. Instead, we have the result that for all $\epsilon \geq 0$, DTIME($O(n)$) = DTIME($(1 + \epsilon)n$), and the proof actually follows from the proof of the Linear Speedup Theorem. This result cannot be improved, for Rosenberg [Ros67] showed that DTIME(n) $\neq$ DTIME($2n$).

The two linear speedup theorems also hold for nondeterministic machines. Thus, if $\inf_{n\to\infty} T(n)/n = \infty$ and $c > 0$, then NTIME($T(n)$) = NTIME($cT(n)$) And, for all $\epsilon \geq 0$, NTIME$(O(n))$ = NTIME$((1+\epsilon)n)$.

However, a stronger result is known for nondeterministic linear-time complexity classes. A Turing machine that accepts inputs of length n in time $n+1$ (the time it takes to read the input) is called *real-time*. Nondeterministic Turing machines that accept in time $n+1$ are called *quasi-realtime*. The class of quasi-realtime languages is NTIME$(n+1)$. Book and Greibach [BG70] showed that NTIME$(n+1) = \bigcup\{\text{NTIME}(cn) \mid c \geq 1\}$.

Since DTIME(n) is a proper subset of DTIME$(2n)$, and

$$\text{DTIME}(2n) \subseteq \text{NTIME}(2n) = \text{NTIME}(n),$$

it follows that DTIME$(n) \neq$ NTIME(n). In 1983, Paul, Pippenger, Szemeredi, and Trotter [PPST83] obtained the striking and deep result that DTIME$(O(n)) \neq$ NTIME$(O(n))$.

4.2 Inclusion Relationships

Now we survey the known inclusion relationships between time-bounded and space-bounded, deterministic, and nondeterministic classes.

First of all, it is trivial that for every function f, DTIME$(f) \subseteq$ DSPACE(f).

Obviously a Turing machine might enter an infinite loop and still use only bounded space. Nevertheless, if a language L is accepted by an $S(n)$ space-bounded Turing machine, where $S(n) \geq \log(n)$, then L is accepted by an $S(n)$ space-bounded Turing machine that halts on every input. The proof depends on the observation that within space $S(n)$ a Turing machine can enter at most an exponential in $S(n)$ possible distinct configurations. A machine enters an infinite loop by repeating one of these configurations, thus making loop detection possible. Analysis of this result yields the following theorem.

Theorem 1 DSPACE$(S(n) \subseteq \bigcup\{\text{DTIME}(c^{S(n)}) \mid c \geq 1\}$*, for* $S(n) \geq \log(n)$.

Theorem 2 NTIME$(T(n)) \subseteq \bigcup\{\text{DTIME}(c^{T(n)}) \mid c \geq 1\}$

We alluded to this result already. Recall that a nondeterministic $T(n)$ time-bounded Turing machine M and an input x of length n determine a computation tree of depth at most $T(n)$, and that M accepts n only if one of the paths of the tree is an accepting path. Theorem 2 is simply a formal expression of the time it takes to execute a search of this tree.

The theorems just presented are proved by rather straightforward simulations. The next theorems involve deep recursions in their simulations. First we need a couple of technical definitions. The point is that the following results don't seem to hold for all resource bounds, but only for those that are *constructible*. This is not a problem, because it turns out that all the bounds in which we are interested are constructible.

A function $S(n)$ is *space-constructible* if there is some Turing machine M that is $S(n)$ space-bounded, and for each n, there is some input of length n which uses exactly $S(n)$ cells. A function $S(n)$ is *fully space-constructible* if every input of length n uses $S(n)$ cells.

A function $T(n)$ is *time-constructible* if there is some Turing machine M that is $T(n)$ time-bounded, and for each n, there is some input of length n on which M runs for exactly $T(n)$ steps. A function $T(n)$ is *fully time-constructible* if for every input of length n, M runs for exactly $T(n)$ steps.

It is merely an exercise to see that ordinary arithmetic functions such as $\log(n)$, n^c, and c^n, are fully time-constructible and fully space-constructible.

Theorem 3 (Savitch [Sav70]) *If S is fully space-constructible and $S(n) \geq \log(n)$, then* $\mathrm{NSPACE}(S(n)) \subseteq \mathrm{DSPACE}(S^2(n))$.

This is a very important result, from which the following corollaries follow. Observe that a standard depth-first search simulation would only provide an exponential upper-bound.

Corollary 1 $\mathrm{PSPACE} = \bigcup\{\mathrm{DSPACE}(n^c) \mid c \geq 1\} = \bigcup\{\mathrm{NSPACE}(n^c) \mid c \geq 1\}$ *and* $\mathrm{POLYLOGSPACE} = \bigcup\{\mathrm{DSPACE}(\log(n)^c) \mid c \geq 1\} = \bigcup\{\mathrm{NSPACE}(\log(n)^c) \mid c \geq 1\}$

For this reason, nondeterministic versions of PSPACE and POLYLOGSPACE were not defined as standard complexity classes.

Corollary 2 $\mathrm{NSPACE}(n) \subseteq \mathrm{DSPACE}(n^2)$ *and* $\mathrm{NL} \subseteq \mathrm{POLYLOGSPACE}$.

Theorem 4 ([Coo71a]) *If S is fully space-constructible and $S(n) \geq \log(n)$, then*

$$\mathrm{NSPACE}(S(n)) \subseteq \bigcup\{\mathrm{DTIME}(c^{S(n)}) \mid c > 1\}.$$

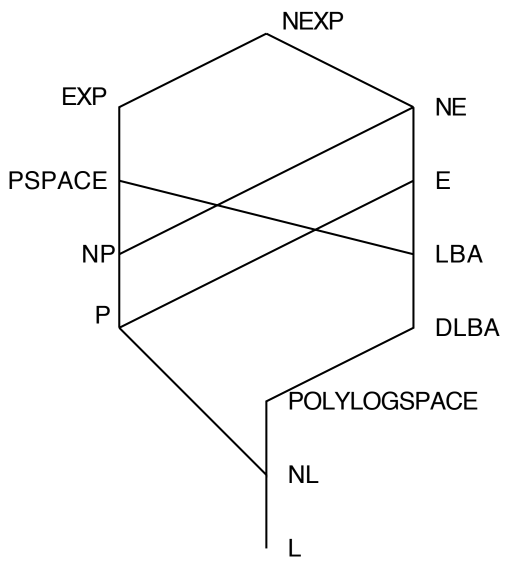

figure 1

4.3 Relations between the Standard Classes

Figure 1 shows the inclusion relations that emerge by application of the results just presented. In this figure, a complexity class C is included in complexity class D if there is a path from C to D reading upward.

From Theorem 1 we learn that L $\subseteq$ P, PSPACE $\subseteq$ EXP , and DLBA $\subseteq$ E. By Theorem 2, we know that NP $\subseteq$ PSPACE. By Savitch's Theorem, LBA $\subseteq$ PSPACE, and (Corollary 2) NL $\subseteq$ POLYLOGSPACE. Theorem 4 is used to conclude that NL $\subseteq$ P and LBA $\subseteq$ E. All other inclusions in the figure are straightforward.

4.4 Separation Results

Now we consider which of these classes are the same and which are not equal. The following theorems are proved by diagonalization.

The first theorem asserts that if two space bounds differ by even a small amount, then the corresponding complexity classes differ. The second theorem gives a separation result for time, but it requires that the time functions differ by a logarithmic factor.

There are technical reasons for this. The intuition that makes results for time harder to obtain than for space is straightforward though. Time marches relentlessly forward, but space can be reused.

Theorem 5 ([HLS65]) *If S_2 is space-constructible, and*

$$\inf_{n\to\infty} S_1(n)/S_2(n) = 0,$$

then there is a language in DSPACE($S_2(n)$) *that is not in* DSPACE($S_1(n)$).

(This theorem was originally proved with the additional assumptions that $S_1(n) \geq \log(n)$ and $S_2(n) \geq \log(n)$. Sipser [Sip78] showed that the additional assumptions are unnecessary.) As consequences of this theorem, L $\neq$ POLYLOGSPACE, POLYLOGSPACE $\neq$ DLBA, and DLBA $\neq$ PSPACE. It even follows easily from theorems we described already that LBA is properly included in PSPACE.

Theorem 6 ([HS65]) *If T_2 is fully time-constructible and*

$$\inf_{n\to\infty} T_1(n)\log(T_1(n))/T_2(n) = 0,$$

then there is a language in DTIME($T_2(n)$) *that is not in* DTIME($T_1(n)$).

Thus, P $\neq$ EXP. Equality (or inequality) of all other inclusion relationships given in the figures is unknown. Amazingly, proper inclusion of each inclusion in the chain

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE}$$

is an open question even though the ends of the chain are distinct (L $\neq$ PSPACE).

Similarly, equality of each inclusion in the chain

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}$$

is open, yet P $\neq$ EXP.

Also, it is not known whether any of the inclusions in the chain

$$\mathrm{DLBA} \subseteq \mathrm{LBA} \subseteq \mathrm{E} \subseteq \mathrm{NE}$$

are proper.

In addition to the results explicitly mentioned here, there is a nondeterministic space hierarchy theorem due to Ibarra [Iba72]. In 1987 Immerman [Imm88] and Szelepcsényi [Sze88] independently proved that nondeterministic $S(n)$ space-bounded classes are closed under complements for $S(n) \geq \log(n)$. A consequence of this important result is that there is a hierarchy theorem for nondeterministic space that is as strong as that for deterministic space. Also there is a difficult nondeterministic time hierarchy theorem due to Cook [Coo73].

Book [Boo72, Boo76] has shown that none of the complexity classes POLYLOGSPACE, DLBA, and LBA is equal to either P or NP. He does this by analyzing the closure properties of these classes, and shows that they are different from one another. To this date, it is not known which of these classes contains a language that does not belong to the other.

5 Nondeterminism and NP-Completeness

Several different additions to the basic deterministic Turing machine model are often considered. These additions add computational power to the model and allow us to classify certain problems more precisely. Often these are important problems with seemingly no efficient solution in the basic model. The question then becomes whether the efficiency provided by the additional power is really due to the new model or whether the added efficiency could have been attained without the additional resources.

The original and most important example of this type of consideration is nondeterminism. For each of the standard nondeterministic complexity classes we have been considering, it is an open question whether the class is distinct from its deterministic counterpart. (The only exception being PSPACE.)

Recall that a nondeterministic Turing machine is one with a multivalued transition function, and recall that such a machine M accepts an input word x if there some computation path of M on x that terminates in an accepting state. Let us assume, for each configuration of M, that there are exactly two possible next configurations, say c_0 and c_1. Then, M can be simulated in the following two-stage manner: (1) Write an arbitrary binary string on a work-tape of a deterministic Turing machine M'. (2) Given an input word x, M' deterministically executes the computation path of M on x that is determined by the binary-string written on its work-tape. That is, M' simultaneously simulates M and reads its binary word from left to right; if the symbol currently scanned is 0, then M' continues its simulation in configuration c_0, and if the symbol currently scanned is 1, then M' continues its simulation in configuration c_1. Clearly, there is a binary string r such that M' accepts x when r is written on its work-tape if and only if there is an accepting computation of M on x. Informally, stage (1) comprises a "guess" of a *witness* or *proof* r to the fact that $x \in L(M)$, and stage (2) comprises a deterministic *verification* that r is a correct guess.

This guessing and verifying completely characterizes nondeterminism. Consider the problem (SAT) of determining, given a formula F of propositional logic, whether F is satisfiable. If F has n variables, there are 2^n different truth assignments. All known deterministic algorithms for deciding the satifiability of F are equivalent to a sequential search of each of the assignments to see if one of them leads to satisfaction (i.e., evaluates to the truth value TRUE). Clearly, an exhaustive search algorithm for checking satisfaction takes 2^n steps for a formula of size n, thereby placing SAT into the complexity class E. However, the following nondeterministic algorithm for SAT reduces its complexity to polynomial time: Given an input formula F, (1) guess an assignment to the Boolean variables of F. This takes $O(n)$ steps. (2) Verify that the assignment evaluates to True. This takes $O(n \log(n))$ steps. Thus, SAT belongs to the class NP.

NP is the most important nondeterministic complexity class. It is the class of languages that have deterministic, polynomial-time verifiers. NP plays a central role in computational complexity as many important problems from computer science and mathematics, which are not known to be solvable deterministically in polynomial time, are in NP. These include SAT, the graph Hamiltonian Circuit problem discussed earlier, various scheduling problems, packing problems, nonlinear programming, and hundreds of others. The most central and well-known open problem in complexity theory is whether P = NP. This problem is one of the several Millenium Prize Problems and the researcher who solves it receives a $1,000,000 prize from the Clay Mathematics Institute [Ins00].

The concept of NP-completeness plays a crucial role here as it gives a method of defining the most difficult NP problems. Intuitively, a problem A in NP is NP-complete if any problem in NP could be efficiently computed using an efficient algorithm for A as a subroutine.

Formally, A is NP-complete if (1) $A \in$ NP, and (2) for every $B \in$ NP, there is a function f that can be computed in polynomial time such that $x \in B$ if and only if $f(x) \in A$. (When property (2) holds for any sets B and A, we way B is *many-one reducible to* A *in polynomial time*, and we write $B \leq_m^{\mathrm{P}} A$.) This definition captures the intuitive notion: An efficient procedure for determining whether x belongs to B is to compute $f(x)$ and then input $f(x)$ to a simple subroutine for determining membership in A.

Focus on the class NP as well as the discovery that SAT is NP-complete is due to Cook [Coo71b], and to Levin [Lev73], who independently obtained a variant of this result. Karp [Kar72] discovered a wide variety of NP-complete problems, and clearly demonstrated the importance of this new notion. These works opened the floodgates to the discovery of many more NP-complete problems in all areas of computer science and other computational disciplines.

The salient fact about NP-complete problems is that NP = P if and only if P contains an NP-complete problem. Thus, a single problem captures the complexity of the entire class.

6 Relative Computability

In this section we expand more broadly on the idea of using a subroutine for one problem in order to efficiently solve another problem. By doing so, we make precise the notion that the complexity of a problem B is related to the complexity of A—that there is an algorithm to efficiently accept B *relative to* an algorithm to efficiently accept A. Most generally, this should mean that an acceptor for B can be written as a program that contains subroutine calls of the form "$x \in A$," which returns TRUE if the Boolean test is true, and which returns FALSE, otherwise. The algorithm for accepting B is called a *reduction procedure* and the set A is called an *oracle*. The reduction procedure is *polynomial time-bounded* if the algorithm runs in polynomial time when we stipulate that only one unit of time is to be charged for the execution of each subroutine call. Obviously, placing faith in our modified Church–Turing thesis and in the Cobham–Edmonds thesis, these ideas are made precise via the oracle Turing machine.

An *oracle* Turing machine[1] is a multitape Turing machine with a distinguished work tape called the oracle tape, and three special states Q, YES, and NO. When the Turing machine enters state Q the next state is YES or NO depending on whether or not the word currently written on the oracle tape belongs to the oracle set. In this way, if the machine is provided with an oracle A, it receives an answer to a Boolean test of the form "$x \in A$" in one move. Let M be an oracle Turing machine, let A be an oracle, and let T be a time complexity function. Oracle Turing machine M with oracle A is $T(n)$ *time-bounded* if for every input of length n, M makes at most $T(n)$ moves before halting. If M is a nondeterministic oracle Turing machine, then every computation of M with A on words of length n must make at most $T(n)$ moves before halting. The language accepted by M with oracle A is denoted $L(M, A)$.

6.1 The Polynomial Hierarchy

If B is accepted by a polynomial time-bounded oracle Turing machine with oracle A, then we write $B \in \mathrm{P}^A$. P^A is the class of sets acceptable in polynomial time relative to the set A. The class NP^A is defined analogously. P^{NP} is the class of sets accepted in polynomial time using oracles in NP. Similarly, $\mathrm{NP}^{\mathrm{NP}}$ is the class of sets accepted in nondeterministic polynomial time using oracles in NP.

In analogy to the arithmetical hierarchy, there is a hierarchy of complexity classes that lies between P and PSPACE that is generated by using more and more powerful oracles. This is called the *polynomial hierarchy* (cf. [Sto76, Wra76]). The class P forms the bottom of the hierarchy and NP lies on the first level. The second level, denoted Σ_2^{P}, is defined to be $\mathrm{NP}^{\mathrm{NP}}$. The third level, Σ_3^{P}, is $\mathrm{NP}^{\Sigma_2^{\mathrm{P}}}$. In this manner, the classes of the polynomial hierarchy are defined inductively. It is not known whether the polynomial hierarchy is a strict hierarchy, just as it is not known whether $\mathrm{P} \neq \mathrm{NP}$. However, researchers believe the polynomial hierarchy forms an infinite hierarchy of distinct complexity classes, and it serves as a useful classification scheme for combinatorial problems.

If $\mathrm{P} = \mathrm{PSPACE}$, then the entire polynomial hierarchy (PH) collapses to P. However, this is not believed to be the case. Researchers believe that the hierarchy is strict, in which case it follows that PH is properly included in PSPACE. And assuming this is so,

[1]Oracle Turing machines were first discussed by Turing in his 1939 London Mathematical Society paper, "Systems of Logic Based on Ordinals" [Tur39]. They were called o-machines in the paper.

the structure of PSPACE $-$ PH is very complex [AS89].

6.2 NP-Hardness

A set A is NP-*hard* if for all sets $B \in$ NP, $B \in \mathrm{P}^A$. Note that NP-hard sets are not required to belong to NP. NP-hard problems are as "hard" as the entire class of problems in NP, for if A is NP-hard and $A \in$ P , then P $=$ NP.

Define a set B to be *Turing reducible to A in polynomial time* ($B \leq_{\mathrm{T}}^{\mathrm{P}} A$) if $B \in \mathrm{P}^A$. Then, it is interesting to compare $\leq_{\mathrm{T}}^{\mathrm{P}}$ with $\leq_m^{\mathrm{P}}$. (Recall that we needed the latter in order to define NP-complete sets.) The reducibilities "$B \leq_m^{\mathrm{P}} A$" and "$B \leq_{\mathrm{T}}^{\mathrm{P}} A$" are not the same, for they are known to differ on sets within the complexity class E [LLS75]. Define a set A to be $\leq_m^{\mathrm{P}}$-*hard* for NP if for all sets $B \in$ NP, $B \leq_m^{\mathrm{P}} A$. Under the assumption P $\neq$ NP, one can prove the existence of an NP-hard set A that is not $\leq_m^{\mathrm{P}}$-hard for NP [SG77]. Furthermore, NP-hard problems are known that (1) seem not to belong to NP, and (2) no $\leq_m^{\mathrm{P}}$-reduction from problems in NP is known [GJ79],[HS01, HS11].

6.3 Complete Problems for Other Classes

Completeness is not a phenomenon that applies only to the class NP. For each of the complexity classes in our list of standard classes, it is possible to define appropriate reducibilities and to find complete problems in the class with respect to those reducibilities. That is, there are problems that belong to the class such that every other problem in the class is reducible to them. A problem that is complete for a class belongs to a subclass (that is closed under the reducibility) if and only if the entire classes are the same. Interestingly, each of the sample problems given at the end of Section 3 are complete for the classes to which they belong. So, for example, the graph accessibility problem is complete for NL, and this problem belongs to L if and only if L $=$ NL. The canonical complete problem for PSPACE is formed by fully quantifying formulas of propositional calculus. The problem then is to determine whether the resulting quantified Boolean formula is true.

7 Nonuniform Complexity

Here we briefly digress from our description of computational complexity based on the Turing machine to describe an important finite model of computing, Boolean circuits. Since real computer are built from electronic devices, digital circuits, this is reason enough to consider their complexity. The circuits that we consider are idealizations of digital circuits just as the Turing machine is an idealization of real digital computers. Another reason to study such nonuniform complexity is because interesting connections to uniform complexity are known.

A *Boolean circuit* is a labeled, acyclic, directed graph. Nodes with in-degree 0 are called *input nodes*, and are labeled with a Boolean variable x_i or a constant 0 or 1. Nodes with output degree 0 are called *output nodes. Interior nodes*, nodes other than input and output nodes represent logical gates: they are labeled with AND, inclusive OR, or NOT. Arbitrary fan-out is allowed.

The constants 1 and 0 are the allowed inputs. If C is a circuit with exactly one output node and n input nodes, then C *realizes* a Boolean function $f : \{0,1\}^n \mapsto \{0,1\}$. When the input nodes receive their values, every interior value receives the value 0 or 1 in accordance with the logical gate that the interior node represents. If $x = x_1x_2\ldots x_n$ is a string in $\{0,1\}^n$ and C is a circuit with n input nodes, then we say C *accepts* x if C outputs 1 when x is the input. Let A be a set of strings of length n over the binary alphabet. Then A is realized by a circuit C if for all string x of length n, C accepts x if and only if x belongs to A.

As we have described computational complexity thus far, one machine is expected to serve for inputs of all lengths. The Turing machine is "uniform;" it is finitely describable, but might accept an infinite number of strings. A circuit however serves only for inputs of one length. Therefore, it is not a single circuit that corresponds to a machine, or that can accept a language, but a family of circuits, one for each length. A family of circuits is "nonuniform;" it may require an infinite description. We say that a family of circuits $\{C_n\}$ *recognizes* a set A, $A \subseteq \{0,1\}^*$, if for each n, the circuit C_n, realizes the finite set $A^n = \{x \mid |x| = n \text{ and } x \in A\}$.

It should be obvious that families of circuits are too broad to exactly classify complexity classes, because every tally language[2] is recognized by some family of circuits. So there are families of circuits that recognize undecidable languages. Nevertheless, we proceed.

The *size* of a circuit is the number of nodes and the *depth* of a circuit is the length of the longest path in the circuit. The size of a circuit is a measure of the quantity of the circuit's hardware. We concentrate our focus on polynomial size families of circuits, because these are the only families that can feasibly be constructed.

Circuit depth corresponds to parallel processing time, but we are not ready to fully justify this claim, which requires understanding the relationship between uniform and nonuniform computation. Basically, every parallel computation (on some parallel processor) can be unraveled to form a family of circuits with constant delay at each gate. So circuit depth is a lower bound on parallel time. If language A is not recognizable by a family of circuits of depth $T(n)$, then no parallel computer can compute A in time $T(n)$.

Savage [Sav72] proved that every set in DTIME($T(n)$) can be recognized by a family of circuits of size $O(T(n))^2$. A tighter result than this is proved by Pippenger and Fischer [PF79]. As an immediate consequence, every language in P has a polynomial-size family of circuits. Hence, if some language in NP does not have a polynomial-size family of circuits, then P $\neq$ NP.

Largely because of this observation, much effort has gone into obtaining lower-bounds for various classes of circuits. Thus far this effort has met with limited success: Deep results have been obtained for highly restricted classes [FSS84, Ajt83, Yao85]. There is more to this story. Shannon [Sha49], in a nonconstructive proof, showed that almost all circuits require exponential size circuits, but we have no information about which sets these are. One of the great challenges is to find lower bounds to the circuit size of families of circuits that recognize explicit, interesting, combinatorial problems.

Now we must mention the important result of Karp and Lipton [KL80] that states that if NP has a polynomial-size family of circuits, then the polynomial hierarchy collapses to the second level. Here is another important connection between nonuniform and uniform complexity, which supports the conjecture that NP does not have a polynomial-size family of circuits.

[2]A tally language is a language defined over a unary alphabet.

8 Parallelism

We consider now the theory of highly parallel computations. VLSI chip technology is making it possible to connect together large numbers of processors to operate together synchronously. The question, of course, is what can be done with the result. Several models of parallel computation have been proposed and even though each model can simulate the other without much loss, the issue of "correct" model is not quite as settled as it is with sequential computation. In this article we mention two models of parallelism, "alternating Turing machines" and "uniform families of circuits," which are important to computational complexity.

The alternating Turing machine, due to Chandra, Kozen, and Stockmeyer [CKS81] is a fascinating extension of the nondeterministic Turing machine. Informally, think of the nondeterministic Turing machine as one consisting of "existential" configurations. That is, when the Turing machine enters an existential configuration, it causes a number of processes to operate in parallel, one process for each nondeterministic choice. If one of these processes accepts, then it reports acceptance back to its parent, and in this manner the computation accepts. With the alternating Turing machine, a process that became active by this action, in turn enters a "universal" configuration that causes another large number of processes to become active. This time, the universal configuration eventually reports acceptance to its parent if and only if all of the processes it spawns accept. Consider such alternating Turing machines that operate in polynomial time, and consider the complexity class AP of languages that are accepted by these devices. It should come as no surprise that alternating Turing machines that consist of a constant $k > 0$ number of alternations accepts precisely the k^{th} level of the polynomial hierarchy. More surprising, is the theorem that AP = PSPACE. Studies have shown that it is true in general that

> parallel time is within a polynomial factor of deterministic space.

The theorem just stated suggests that presumably intractable languages, i.e., those in PSPACE, are capable of enormous speedup by using parallelism. This, however, is an impractical observation. The proof of the theorem requires an exponential number of processes. No one will ever build parallel computers with an exponential number of processes.

Before returning to the question of what are the problems that can take advantage of parallelism, we introduce our second model. Families of circuits would provide a simple and effective model were it not for the fact that small families of circuits can recognize undecidable sets. We repair this flaw by introducing uniformity. Borodin and Cook [Coo79, Bor77] define a family of circuits $\{C_n\}_n$ to be *logspace uniform* if there exists a deterministic Turing machine that on input 1^n, for each $n \geq 1$, computes (an encoding of) the circuit C_n. Then, for example, one interesting result is that P is the class of languages that have logspace uniform, polynomial-size families of circuits.

Now we can return to the question at hand and we do so with a discussion of Greenlaw *et al.* [GHR95]: We have seen already that parallelism will not make intractable problems tractable. Therefore, if we are to dramatically improve performance it must be by reducing polynomial sequential time to subpolynomial parallel time. We achieve this by trading numbers of processors for speed. The goal of practical parallel computation is to develop algorithms that use a reasonable number of processors and are exceedingly fast. What do we mean by that? We assume that a polynomial number of processors is reasonable, and more than that is unreasonable. Can fewer than a polynomial number of processors suffice? To answer this questions observe that

$$\text{(sequential time)}/\text{(number of processors)} \leq \text{(parallel time)}$$

Taking sequential time to be polynomial time, obviously, if parallel time is to be subpolynomial, then a polynomial number of processors must be used. We focus on the class of problems that have uniform families of circuits with polynomial size and polylog, i.e., $(\log n)^{O(1)}$, depth. So highly parallel problems are those for which we can develop algorithms that use a polynomial number of processors to obtain polylog parallel time bounds.

The resulting class of languages is called NC, and is named in honor of Nick Pippenger, who obtained an important characterization [Pip79]. Researchers identify the class NC with the collection of highly parallel problems, much as we identify P with the collection of feasibly computable problems.

The models we introduced are quite robust. For example, it is known that NC is identical to the class of languages accepted by alternating Turing machines in simultaneous polylog time and log space.

9 Probabilistic Complexity

In a 1950 paper in the journal Mind [Tur50], Alan Turing wrote,

> An interesting variant on the idea of a digital computer is a 'digital computer with a random element'. These have instructions involving the throwing of a die or some equivalent electronic process: one such instruction might for instance be, 'Throw a die and put the resulting number into store 1000.'

Nearly thirty years after this paper appeared the ideas put forth there became a central part of the active area of probabilistic complexity theory. The main computational model used for this study is a probabilistic Turing machine. It has been used to measure and formalize the performance of probabilistic algorithms and to define the main concepts for the theoretical study of probabilistic complexity theory.

The impetus for this work began in the mid-1970's with the work on algorithms for primality testing pioneered by Solovay and Strassen [SS77], and by Miller and Rabin [Mil76] , [Rab80]. The two algorithms invented in these papers both have the same character. They both yield methods for testing whether a given input integer is prime that are significantly more efficient than any deterministic algorithm. Instead they make use of random numbers r provided to the algorithm, and used there to decide primality. While very efficient the algorithm can, in rare instances of r, make a mistake and wrongly decide that a composite number is actually prime. If the probability of these errors happening is non-negligible, such algorithms have little worth. But in this case it is possible to ensure that the probability of error is extremely small, say smaller than the chances the computer makes a hardware error during the computation, or smaller than the number of atoms in the known universe. In such cases the algorithms can be thought to be essentially correct and useful for the actual task of finding large prime numbers for various application, most notably cryptographic applications.

From these ideas arose other examples of strong probabilistic algorithms for important interesting problems. Over the next decade these algorithms gained in impact and theorists took on the task of classifying and examining these algorithms and the strength of the assumptions and method underlying them. The model they used for this study was essentially the model proposed above by Alan Turing in 1950.

These algorithms suggested that we should revise our notion of “efficient computation”. Perhaps we should now equate the efficiently computable problems with the class of problems solvable in probabilistic polynomial time. Beginning in the 1970’s a new area of complexity theory was developed to help understand the power of probabilistic computation.

Formally, a probabilistic Turing machine is just a nondeterministic Turing machine, but acceptance is defined differently. Each nondeterministic choice is considered as a random experiment in which each outcome has equal probability. We may assume that each nondeterministic branch has exactly two possible outcomes, so that each has possibility 1/2. A probabilistic Turing machine has three kinds of final states, accepting or a-states, rejecting or r-states, and undetermined or ?-states. The outcome of the machine on an input is now a random variable whose range is $\{a, r, ?\}$. We let $\Pr[M(x) = y]$ denote the probability that machine M on input x halts in a y-state. Note that the probability of a given nondeterministic path is obtained by raising 1/2 to a power equal to the number of nondeterministic choices along it. The probability that M accepts an input x, $\Pr[M(x) = a]$, is the sum of the probabilities of all accepting paths (that is, paths which end in an a-state).

Using this model we can now define several different useful probabilistic complexity classes. These classes were originally defined by Gill [Gil77] and by Adleman and Manders [AM77]. Each consists of languages accepted by restricting the machines to polynomial time and specifying the probability needed for acceptance. The first class, PP, for probabilistic polynomial time, is the easiest to define and is the most powerful parallel class we consider, but is the least useful. Let χ_A denote the characteristic function of A. PP is the class of all languages for which there is a probabilistic, polynomial time-bounded Turing machine M such that for all x,

$$\Pr[M(x) = \chi_A(x)] > 1/2.$$

That is,

$$x \in A \rightarrow \Pr[M(x) = a] > 1/2,$$

and

$$x \notin A \rightarrow \Pr[M(x) = r] > 1/2.$$

One would like to increase the reliability of a probabilistic Turing machine by repeating its computations a large number of times and giving as output the majority result. This turns out not to be possible with PP, and this is why there is no practical interest in this class, we are at times stuck with a too high probability of error. While it is not known whether PP is contained in the polynomial hierarchy, we do have the following two quite easy facts.

Lemma 1 NP $\subseteq$ PP.

Proof Let M be a nondeterministic Turing machine that accepts an NP language L. Now consider the nondeterministic Turing machine M' whose first move is to nondeterministically either simulate M or to go into a special state where the computation, in every succeeding move, just splits into two identical computations until the length of the computation reaches the same length as the computations of M and then halts in an accepting state.

Now if $x \notin L$ then there are exactly as many accepting computations as rejecting computations. Namely, none of the computations whose first step is to simulate M accept while all of the computations whose first step is to go into the special new state accept. These two groups of computations are of the same size.

On the other hand, if $x \in L$, then as above, all computations starting with the special state accept and at least one computation of M accepts and so more computations accept than reject. So the machine M', considered as a probabilistic machine, shows that L is in PP. ∎

While stated without proof, it is quite straightforward to simulate the answer to PP computations within PSPACE, yielding,

Lemma 2 PP $\subseteq$ PSPACE.

The error probability is the probability that a probabilistic machine gives the wrong answer on a given input. The next class we define bounds the error probability away from 1/2, and that restriction makes it possible to increase reliability. The class BPP, for bounded-error probabilistic polynomial time, is the class of all languages for which there is a probabilistic, polynomial time-bounded Turing machine M and a number $\epsilon, 0 < \epsilon < 1/2$, such that for all x,

$$\Pr[M(x) \neq \chi_A(x)] < \epsilon.$$

BPP-type Turing machines are said to be of the "Monte Carlo" type. They model algorithms that are allowed to make mistakes (i.e. terminate in an r-state when $x \in L$, or terminate in an a-state, when $x \notin L$) with some small probability. On the other hand, "Las Vegas" algorithms may terminate with "?" (again with small probability), but they never err. These too can be captured by probabilistic complexity.

The primality algorithms in the examples above were Monte Carlo algorithms. It is not difficult to see that P $\subseteq$ BPP $\subseteq$ PP, and BPP is closed under complement. But where does BPP lie with respect to the polynomial hierarchy? Somewhat surprisingly, Sipser, Gacs, and Lautemann proved that BPP $\subseteq \Sigma_2^{\mathrm{P}}$ [Sip83, Lau83].

In fact something stronger can be seen from these algorithms. Namely the probabilistic primality algorithms have the property that when the input to them is prime it is always identified as being prime by the algorithms. It is only when the input in composite that there is some probability of error. To capture this property we define the class, R, an interesting, one-sided version of BPP.

The class R, for random polynomial time, is the class of all languages for which there is a probabilistic, polynomial time-bounded Turing machine M such that for all x,

$$x \in A \rightarrow \Pr[M(x) = a] > 1/2,$$

and

$$x \notin A \rightarrow \Pr[M(x) = a] = 0.$$

It has been shown that primality and also its complement, the set of all composite number, is solvable in R.

Neither R nor BPP are known to contain complete problems. Several properties of these classes suggest that such problems don't exist. And it is known that oracles exist relative to which neither class contains complete problems.

Returning to the idea of primality testing, and the complexity of the set of prime numbers, it is easy to see that primality is in co-NP. An early result of Pratt [Pra75] showed that one could also find a witness for primality which can be checked in polynomial time, putting primality into NP $\cap$ co-NP.

The early probabilistic algorithms of Strassen/Solovay and of Miller/Rabin discussed earlier show that primality is in co-R. Later, work of Adleman and Huang [AH87] improved this to obtain primality in R $\cap$ co-R, implying that primality has an efficient Las Vegas algorithm. Finally, early in this century Agrawal, Kayal and Saxena [AKS04] gave a deterministic polynomial-time algorithm for primality. If this result was known in the 70's, perhaps the study of probabilistic algorithms would not have progressed as quickly.

9.1 Derandomization

If you generate a random number on a computer, you do not get a truly random value, but a pseudorandom number computed by some complicated function on some small, hopefully random seed. In practice this usually works well so perhaps in theory the same might be true. Many of the exciting results in complexity theory in the 1980's and 90's consider this question of derandomization—how to reduce or eliminate the number of truly random bits to simulate probabilistic algorithms.

The results above add to the possibility of eliminating randomness from BPP and R problems. BPP has many similar properties to P (e.g., they are both closed under complement and $\mathrm{BPP}^{\mathrm{BPP}} = \mathrm{BPP}$) and because there are no natural problems and very few candidates for problems in BPP $-$ P, many believe that P $=$ BPP. This subject considers how to reduce or eliminate randomness from probabilistic algorithms, making them deterministic. For example P $=$ BPP is equivalent to being able to derandomize all BPP algorithms. There has been very interesting considerable progress in this area in the past decade. But it is still unknown whether P $=$ BPP, or even P $=$ R.

There have been two approaches to eliminating randomness, both of which indicate that strong, general derandomization results may be possible. The first approach arose from cryptography where creating randomness from cryptographically hard functions was shown by Blum and Micali [BM84]. Subsequently Yao [Yao82] showed how to reduce the number of random bits of any algorithm based on any cryptographically secure one-way permutation. Håstad, Impagliazzo, Levin and Luby [HILL91] building on techniques of Goldreich and Levin [GL89] and Goldreich, Krawczyk and Luby [GKL88] prove that one can get pseudorandomness from any one-way function.

Nisan and Wigderson [NW94] take a different approach. They show how to get pseudorandomness based on a language hard against nonuniform computation. Impagliazzo and Wigderson [IW97] building on this result and Babai, Fortnow, Nisan and Wigderson [BFNW93] show that BPP equals P if there exists a language in exponential time that cannot be computed by any subexponential circuit. This is a believable hypothesis.

10 Interactive Proof Systems

One can think of the class NP as a *proof system*: For example for the NP problem SAT = {satisfiable Boolean formulas} an arbitrarily powerful prover gives a proof, i.e., a string representing a truth assignment, that shows a formula is satisfiable. This proof can be verified by a polynomial-time verifier, an algorithm that is capable of checking that the string is a short (polynomial-length) proof of satisfiability. We do not put any restriction on how the verifier obtains the proof itself, it is sufficient that the proof exists. For example, the proof might be provided by a powerful prover with unrestricted computational power.

Consider a generalization of these ideas where we allow a protocol or dialogue between the verifier and a prover to decide language membership. As before there is no restriction placed on the computational power of the prover, and the verifier is restricted to be only polynomially powerful. We add one more ingredient to this mix by allowing randomization. That is, we allow the verifier, whose time is polynomially limited, to use a random sequence of bits to aid in the computation, and require only that he be convinced with high likelihood, and not with complete certainty, of the membership claim being made. Specifically the verifier is a probabilistic polynomial-time Turing machine with acceptance probability bounded away from 1/2, i.e., one implementing a Monte Carlo algorithm.

This model generalizes that of the NP verifier, it is clear that as before NP problems are those for which membership can be proved by a prover giving the verifier a polynomial length "proof" and the verifier validating that proof in deterministic polynomial time. The model also easily captures the class BPP as any BPP problem can be decided by the probabilistic polynomial time verifier without any information from the prover.

In general this simple probabilistic interactive proof where the interaction is limited to one "round" of communication, in this case the prover sending a message to the verifier, yields the complexity class of problems called MA. One can also consider extended interaction where the verifier sends messages based on her random bits back to the prover. The (fixed) bounded round version of this class is denoted AM and the unbounded polynomial round version is IP. The incredible power of these interactive proof systems has led to several of the most surprising and important recent results in computational complexity theory.

In 1985, Babai [Bab85] defined interactive proof systems to give complexity-theoretic characterizations of some problems concerning matrix groups. An alternative interactive proof system was defined by Goldwasser, Micali and Rackoff [GMR89] as a basis for the cryptographic class zero-knowledge. Zero-knowledge proof systems have themselves played a major role in cryptography.

Goldreich, Micali and Wigderson [GMW86] show that the set of pairs of nonisomorphic graphs has a bounded-round interactive proof system. Boppana, Håstad and Zachos [BHZ87] show that if the complement of any NP-complete language has bounded-round interactive proofs then the polynomial-time hierarchy collapses. As a consequence of this result it is known that the graph isomorphism problem is not NP-complete unless the polynomial-time hierarchy collapses.

In 1990, Lund et al. [LFKN90] showed that the complements of NP-complete languages have unbounded round interactive proof systems. These techniques were quickly extended by Shamir [Sha90] to show that every language in PSPACE has interactive proof system. A few years earlier, Feldman [Fel86] had proved that every language with interactive proofs lies in PSPACE. The result that IP = PSPACE is one of the central advances in complexity theory in the last two decades.

It is notable that in general proofs concerning interactive proof systems do not relativize, that is they are not true relative to every oracle. The classification of interactive proofs turned out not to be the end of the story but only the beginning of a revolution connecting complexity theory with approximation algorithms. For the continuation of this story we turn to probabilistically checkable proofs.

10.1 Probabilistically Checkable Proofs

Understandably, following on the IP = PSPACE result, there was a flurry of activity in the early 1990's examining and classifying the many variations of proof systems both stronger and weaker than IP. One natural direction of this work was to try to apply these methods to the large, central class of NP problems. After a series of results improving on the understanding of interactive proof systems with multiple provers, Babai et al. [BFLS91] scale these proof techniques down to develop "holographic" proofs for NP where, with a properly encoded input, the verifier can check the correctness of the proof in a very short amount of time.

Feige et al. [FGL$^+$96] made an amazing connection between probabilistically checkable proofs and the clique problem. By viewing possible proofs as nodes of a graph, they showed that one cannot approximate the size of a clique well without unexpected collapses in complexity classes.

In 1992, Arora et al. [ALM$^+$92] building on work of Arora and Safra [AS98] showed that every language in NP has a probabilistically checkable proof where the verifier uses only a logarithmic number of random coins and a constant number of queries to the proof. Their results have tremendous implications for the class MAXSNP of approximation problems. This class developed by Papadimitriou and Yannakakis [PY91] contains many interesting complete problems such as vertex cover, max-cut, independent set, some variations of the traveling salesman problem and maximizing the number of satisfiable clauses of a formula. Arora et al. shows that, unless P=NP, every MAXSNP-complete set does not have a polynomial-time approximation scheme. Specifically, for each of these problems there is some constant $\delta > 1$ such that they cannot be approximated within a factor of δ unless P=NP.

Since these initial results on probabilistically checkable proofs, we have seen a large number of outstanding papers improving the proof systems and getting stronger hardness of approximation results. Arora [Aro98] developed a polynomial-time approximation algorithm for the traveling salesman problem in the plane. In a series of papers Håstad [Hås99], [Hås01] obtains tight results for several central approximation problems including max-clique, max-cut, set splitting, chromatic number, and bounded-clause-length versions of SAT. A good explanation of this method and further inapproximability results can be found in Trevisan [Tre04].

11 A. M. Turing Award

The highest honor available to be bestowed on a computer scientist is the ACM's appropriately named A. M. Turing Award. The award is accompanied by a prize of \$250,000, with financial support provided by the Intel Corporation and Google Inc. Awardees who we have cited in this paper include Manuel Blum, Stephen Cook, Juris Hartmanis, Richard Karp, Michael Rabin, Dana Scott, Richard Stearns, and Andrew Yao.

12 Acknowledgment

We thanks Juris Hartmanis for his invaluable comments on Turing machines and the birth of complexity theory. We thank Lance Fortnow for his permission to use some of the discussion of the history of complexity theory, which first appeared in Fortnow and Homer [FH03].

References

[AH87] L. Adleman and M-D. Huang. Recognizing primes in random polynomial time. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 462–469, 1987.

[Ajt83] M. Ajtai. σ_1^1 formulea on finite structures. *Journal of Pure and Applied Logic*, 24:1–48, 1983.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, 160(2):781–793, September 2004.

[ALM+92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the Thirty-Third Symposium on Foundations of Computer Science*, pages 14–23. IEEE Computer Society, 1992.

[AM77] L. Adleman and K. Manders. Reducibility, randomness, and intractability. In *Proceedings of the Nineth Annual ACM Symposium on Theory of Computing*, pages 151–163, 1977.

[Aro98] S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, September 1998.

[AS89] K. Ambos-Spies. On the relative complexity of hard problems for complexity classes without complete problems. *Theoretical Computer Science*, 63:43–61, 1989.

[AS98] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, January 1998.

[Bab85] L. Babai. Trading group theory for randomness. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 421–429, 1985.

[BFLS91] L. Babai, L. Fortnow, L. Levin, and M. Szegedi. Checking computations in polylogarithmic time. In *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, pages 21–31, 1991.

[BFNW93] L. Babai, L. Fortnow, N. Nisan, and A. Wigderson. BPP has subexponential simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3:307–318, 1993.

[BG70] R. Book and S. Greibach. Quasi-realtime languages. *Mathematical Systems Theory*, 4:97–111, 1970.

[BHZ87] R. Boppana, J. Håstad, and S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13:850–864, 1984.

[Boo72] R. Book. On languages accepted in polynomial time. *SIAM Journal on Computing*, 1(4):281–287, 1972.

[Boo76] R. Book. Translational lemmas, polynomial time, and $(\log n)^j$-space. *Theoretical Computer Science*, 1:215–226, 1976.

[Bor77] A. Borodin. On relating time and space to size and depth. *SIAM J. Comput.*, 6:733–744, 1977.

[Chu36] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[Cob64] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30. North Holland, 1964.

[Coo71a] S. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 19:175–183, 1971.

[Coo71b] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo73] S. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, 1973.

[Coo79] S. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proceedings of the 11th Annual ACM Symposium on Theory of of Computing*, pages 338–345, 1979.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[ET76] S. Even and R. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the ACM*, 23:710–719, 1976.

[Fel86] P. Feldman. The optimum prover lives in PSPACE. Manuscript, 1986.

[FGL$^+$96] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43:268–292, 1996.

[FH03] L. Fortnow and S. Homer. A brief history of complexity theory. *Bulletin of the European Assocation for Theoretical Computer Science*, 80:95–133, 2003.

[FSS84] M. Furst, J. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–28, April 1984.

[GHR95] R. Greenlaw, H. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-completeness Theory.* Oxford University Press, New York, NY, 1995.

[Gil77] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, Dec. 1977.

[GJ79] M. Garey and D. Johnson. *Computers And Intractability: A Guide To The Theory of NP-Completeness.* W. H. Freeman, San Francisco, 1979.

[GKL88] O. Goldreich, H. Krawczyk, and M. Luby. On the existence of pseudorandom generators. In *Proc. 29th Ann. IEEE Symp. Found. of Comp. Sci.*, pages 12–24, 1988.

[GL89] O. Goldreich and L. Levin. Hard-core predicates for any one-way function. In *Proc. 21th Ann. ACM Symp. on Theory of Computation*, pages 25–32, 1989.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, Feb. 1989.

[GMW86] O. Goldreich, S. Michali, and A. Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 174–187, 1986.

[Hås99] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[Hås01] J. Håstad. Some optimal inapproximabiity results. *Journal of the ACM*, 48:798–849, 2001.

[HILL91] J. Håstad, R. Impagliazzo, L. Levin, and M. Luby. Construction of a pseudo-random generator from any one-way function. Technical report, ICSI, Berkely, CA, 1991.

[HLS65] J. Hartmanis, P. Lewis, and R. Stearns. Hierarchies of memory limited computations. In *Proceedings of the Sixth Annual IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 179–190, 1965.

[HS65] J. Hartmanis and R. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HS01] S. Homer and A. Selman. *Computability and Complexity Theory.* Texts in Computer Science. Springer-Verlag, New York, NY, 2001.

[HS11] S. Homer and A. Selman. *Computability and Complexity Theory, 2nd Edition.* Texts in Computer Science. Springer-Verlag, New York, NY, December 2011.

[Iba72] O. Ibarra. A note concerning nondeterministic tape complexities. *Journal of Computer and System Sciences*, 19(4):609–612, 1972.

[Imm88] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[Ins00] Clay Mathematics Institute. Millennium prize problems. `http://www.claymath.org/prizeproblems/`, 2000.

[IW97] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th ACM Symposium on the Theory of Computing*, pages 220–229. ACM, New York, 1997.

[Jon73] N. Jones. Reducibility among combinatorial problems in log n space. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems*, pages 547–551, Princeton, NJ, 1973. Department of Electrical Engineering, Princeton University.

[Jon75] N. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11:68–85, 1975.

[JS74] N. Jones and A. Selman. Turing machines and the spectra of first-order formulas. *Journal of Symbolic Logic*, 29:139–150, 1974.

[Kar72] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[KL80] R. Karp and R. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 302–309, 1980. An extended version has appeared in *L'Enseignement Mathématique*, 2nd series 28, 1982, pages 191–209.

[Kur64] S. Kuroda. Classes of languages and linear bounded automata. *Information and Control*, 7(2):207–223, 1964.

[Lau83] C. Lautemann. BPP and the polynomial hierarchy. *Information Processing Letters*, 17:215–217, November 1983.

[Lev73] L. Levin. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973. English translation of original in *Problemy Peredaci Informacii.*

[LFKN90] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Proc. 31st IEEE Symp. Foundations of Computer Science*, pages 2–10, 1990.

[LLS75] R. Ladner, N. Lynch, and A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1:103–123, 1975.

[Mag69] G. Mager. Writing pushdown acceptors. *Journal of Computer and System Sciences*, 3(3):276–319, 1969.

[Mil76] G. Miller. Reimann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13:300–317, 1976.

[Myh60] J. Myhill. Linear bounded automata. WADD 60-165, Wright Patterson AFB, Ohio, 1960.

[NW94] N. Nisan and A. Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49:149–167, 1994.

[PF79] N. Pippenger and M. Fischer. Relations among complexity measures. *Journal of the ACM*, 26:361–381, 1979.

[Pip79] N. Pippenger. On simultaneous resource bounds. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 307–311, 1979.

[PPST83] W. Paul, N. Pippenger, E. Szemerédi, and W. Trotter. On determinism and non-determinism and related problems. In *Proceedings of the Twenty fourth ACM Symposium on Theory of Computing*, pages 429–438, 1983.

[Pra75] V. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4:214–220, 1975.

[PY91] C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[Rab80] M. Rabin. Probabilistic algorithms for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[Ros67] A. Rosenberg. Real-time definable languages. *Journal of the ACM*, 14:645–662, 1967.

[RS59] M. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal*, pages 114–125, April 1959.

[Sav70] W. Savitch. Relationships between nondeterministic and deterministic time complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[Sav72] J. Savage. Computational work and time on finite machines. *Journal of the ACM*, 19:660–674, 1972.

[SG77] I. Simon and J. Gill. Polynomial reducibilities and upward diagonalizations. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 186–194, 1977.

[Sha49] C. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[Sha90] A. Shamir. IP=PSPACE. In *Proc. 31st IEEE Symp. Foundations of Computer Science*, pages 145–152, 1990.

[Sip78] M. Sipser. Halting space-bounded computations. In *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Scienc*, pages 73–74, 1978.

[Sip83] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the Fifteenth ACM Symposium on Theory of Computing*, pages 330–335, 1983.

[SS77] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977.

[Sto76] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1976.

[Sze88] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[Tre04] L. Trevisan. Inapproximability of combinatorial optimization problems. *The Electronic Colloquium in Computational Complexity*, Tech. Rep. 65:1–39, 2004.

[Tur36] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–365, 1936.

[Tur39] A. M. Turing. Systems of logic based on ordinals. *Proc. Lond. Math. Soc*, 45:161–228, 1939.

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, 49:433–460, 1950.

[Wra76] C. Wrathall. Complete sets and the polynomial hierarchy. *Theoretical Computer Science*, 3:23–33, 1976.

[Yao82] A. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

[Yao85] A. Yao. Separating the polynomial-time hierarchy by oracles. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 1–10, 1985.