The P versus NP Problem

Stephen Cook
University of Toronto

1. Statement of the Problem

The **P** versus **NP** problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time. To define the problem precisely it is necessary to give a formal model of a computer. The standard computer model in computability theory is the Turing machine, introduced by Alan Turing in 1936 [Tur37]. Although the model was introduced before physical computers were built, it nevertheless continues to be accepted as the proper computer model for the purpose of defining the notion of *computable function*.

Informally the class **P** is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input. Turing was not concerned with the efficiency of his machines, but rather his concern was whether they can simulate arbitrary algorithms given sufficient time. However it turns out Turing machines can generally simulate more efficient computer models (for example machines equipped with many tapes or an unbounded random access memory) by at most squaring or cubing the computation time. Thus **P** is a robust class, and has equivalent definitions over a large class of computer models. Here we follow standard practice and define the class **P** in terms of Turing machines.

Formally the elements of the class **P** are languages. Let Σ be a finite alphabet (that is, a finite nonempty set), and let Σ^* be the set of finite strings over Σ. Then a *language over* Σ is a subset L of Σ^*. Each Turing machine M has an associated *input alphabet* Σ. For each string w in Σ^* there is a computation associated with M with input w. (The notions of Turing machine and computation are defined formally in the appendix.) We say that M *accepts* w if this computation terminates in the accepting state. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate. The *language accepted by* M, denoted $L(M)$, has associated alphabet Σ and is defined by

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w (see the Appendix). If this computation never halts, then $t_M(w) = \infty$. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the *worst case run time* of M; that is

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n. We say that M *runs in polynomial time* if there exists k such that for all n, $T_M(n) \le n^k + k$. Now we define the class **P** of languages by

$$\mathbf{P} = \{L \mid L = L(M) \text{ for some Turing machine } M \text{ which runs in polynomial time}\}$$

The notation **NP** stands for "nondeterministic polynomial time", since originally **NP** was defined in terms of nondeterministic machines (that is, machines that have more than one possible move from a given configuration). However now it is customary to give an equivalent definition using the notion of a *checking relation*, which is simply a binary relation $R \subseteq \Sigma^* \times \Sigma_1^*$ for some finite alphabets Σ and Σ_1. We associate with each such relation R a language L_R over $\Sigma \cup \Sigma_1 \cup \{\#\}$ defined by

$$L_R = \{w\#y \mid R(w,y)\}$$

where the symbol # is not in Σ. We say that R is *polynomial-time* iff $L_R \in \mathbf{P}$.

Now we define the class **NP** of languages by the condition that a language L over Σ is in **NP** iff there is $k \in \mathbb{N}$ and a polynomial-time checking relation R such that for all $w \in \Sigma^*$,

$$w \in L \Longleftrightarrow \exists y(|y| \leq |w|^k \text{ and } R(w,y)\}$$

where $|w|$ and $|y|$ denote the lengths of w and y, respectively.

Problem Statement: Does $\mathbf{P} = \mathbf{NP}$?

It is trivial to show that $\mathbf{P} \subseteq \mathbf{NP}$, since for each language L over Σ, if $L \in \mathbf{P}$ then we can define the polynomial-time checking relation $R \subseteq \Sigma^* \cup \Sigma^*$ by

$$R(w,y) \Longleftrightarrow w \in L$$

for all $w, y \in \Sigma^*$.

Here are two simple examples, using decimal notation to code natural numbers: The set of perfect squares is in **P** and the set of composite numbers is in **NP** (and not known to be in **P**). For the latter, the associated polynomial time checking relation R is given by

$$R(\overline{a}, \overline{b}) \Longleftrightarrow 1 < b < a \text{ and } b|a \tag{1}$$

In general the decimal notation for a natural number c is denoted by $\overline{c}$.

2. History and Importance

The importance of the **P** vs **NP** question stems partly from the successful theory of **NP**-completeness, and partly from cryptography. A truly efficient algorithm for an **NP**-complete problem such as Satisfiability would on the one hand yield useful algorithms for many practical computational problems in industry, but on the other hand it would destroy the security of financial and other transactions used extensively on the internet. Here we give a brief historical treatment of **NP**-completeness, and touch on cryptography theory.

The theory of **NP**-completeness has its roots in computability theory, which originated in the work of Turing, Church, Gödel, and others in the 1930's. The computability precursors of the classes **P** and **NP** are the classes of decidable and c.e. (computably enumerable) languages, respectively. We say that a language L is c.e. (or *semi-decidable*) iff $L = L(M)$

for some Turing machine M. We say that L is *decidable* iff $L = L(M)$ for some Turing machine M which satisfies the condition that M halts on all input strings w. There is an equivalent definition of c.e. which brings out its analogy with **NP**, namely L is c.e. iff there is a computable "checking relation" $R(x, y)$ such that $L = \{x \mid \exists y R(x, y)\}$.

Using the notation $\langle M \rangle$ to denote a string describing a Turing machine M, we define the Halting Problem HP as follows:

$$HP = \{\langle M \rangle \mid M \text{ is a Turing machine which halts on input } \langle M \rangle\}$$

Turing used a simple diagonal argument to show that HP is not decidable. On the other hand, it is not hard to show that HP is c.e.

Of central importance in computability theory is the notion of reducibility, which Turing defined roughly as follows: A language L_1 is *Turing reducible* to a language L_2 iff there is an oracle Turing machine M which accepts L_1, where M is allowed to make membership queries of the form $x \in L_2$? which are correctly answered by an "oracle" for L_2. Later the more restricted notion of many-one reducibility ($\leq_m$) was introduced and defined as follows.

Definition 1: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \leq_m L_2$ iff there is a (total) computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$.

It is easy to see that if $L_1 \leq_m L_2$ and L_2 is decidable, then L_1 is decidable. This fact provides an important tool for showing undecidability; for example if $HP \leq_m L$ then L is undecidable.

The notion of **NP**-complete is based on the following notion from computability theory:

Definition 2: A language L is c.e.-*complete* iff L is c.e., and $L' \leq_m L$ for every c.e. language L'.

It is easy to show that HP is c.e.-complete. It turns out that most "natural" undecidable c.e. languages are in fact c.e.-complete. Since $\leq_m$ is transitive, to show that a c.e. language L is c.e. complete is suffices to show that $HP \leq_m L$.

The notion of polynomial-time computation was introduced in the 1960's by Cobham [Cob64] and Edmonds [Edm65] as part of the early development of computational complexity theory (although earlier von Neumann [vN] in 1953 distinguished between polynomial time and exponential-time algorithms). Edmonds called polynomial-time algorithms "good algorithms", and linked them to tractable algorithms.

It has now become standard in complexity theory to identify polynomial-time with feasible, and here we digress to discuss this point. It is of course not literally true that every polynomial-time algorithm can be feasibly executed on small inputs; for example a computer program requiring n^{100} steps could never be executed on an input even as large as $n = 10$. Here is a more defensible statement:

Feasibility Thesis: A natural problem has a feasible algorithm iff it has a polynomial-time algorithm.

Examples of natural problems that have both feasible and polynomial-time algorithms abound: Integer arithmetic, linear algebra, network flow, linear programming, many graph problems (connectivity, shortest path, minimum spanning tree, bipartite matching) etc. On the other hand the deep results of Robertson and Seymour [RS83]. provide a potential source of counterexamples to the thesis: They prove that every minor-closed family of graphs can be recognized in polynomial time (in fact in time $O(n^3)$), but the algorithms supplied by their method have such huge constants that they are not feasible. However each potential counter example can be removed by finding a feasible algorithm for it. For example a feasible recognition algorithm is known for the class of planar graphs, but none is currently known for the class of graphs embeddable in $\mathbb{R}^3$ with no two cycles linked. (Both examples are minor-closed families.) Of course the words "natural" and "feasible"in the thesis above should be explained; generally we do not consider a class with a parameter as natural, such as the set of graphs embeddable on a surface of genus k, $k > 1$. See [Coo90] for more discussion of the feasibility thesis.

Quantum computers provide a potential source of counterexamples to the "only if" direction of the thesis. This computer model incorporates the idea of superposition of states from quantum mechanics, and allows a potential exponential speed-up of some computations over Turing machines. For example, Shor [Sho97] has shown that some quantum computer algorithm is able to factor integers in polynomial time, but no polynomial-time integer factoring algorithm is known for Turing machines. However physicists have so far been unable to build a quantum computer that can handle more than a half-dozen bits, so their threat to the feasibility thesis is hypothetical at present.

Returning to the historical treatment of complexity theory, in 1971 the present author [Coo71] introduced a notion of **NP**-completeness as a polynomial-time analog of c.e. completeness, except that the reduction used was a polynomial- time analog of Turing reducibility rather than of many-one reducibility. The main results in [Coo71] are that several natural problems, including Satisfiability and 3-SAT (defined below) and subgraph isomorphism are **NP**-complete. A year later Karp [Kar72] used these completeness results to show that 20 other natural problems are **NP**-complete, thus forcefully demonstrating the importance of the subject. Karp also introduced the now standard notation **P** and **NP** and redefined **NP**-completeness using the polynomial-time analog of many-one reducibility, a definition which has become standard. Meanwhile Levin [Lev73] independently of Cook and Karp defined the notion of "universal search problem", similar to **NP**-complete problem, and gave six examples, including Satisfiability.

The standard definitions concerning **NP**-completeness are close analogs of Definitions 1 and 2 above.

Definition 3: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \leq_p L_2$ (L_1 is p-reducible to L_2) iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \Longleftrightarrow f(x) \in L_2$, for all $x \in \Sigma_1^*$.

Definition 4: A language L is **NP**-*complete* iff L is in **NP**, and $L' \leq_p L$ for every language L' in **NP**.

The following proposition is easy to prove: Part (b) uses the transitivity of $\leq_p$, and part (c) follows from part (a).

Proposition 1: (a) If $L_1 \leq_p L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.

(b) If L_1 is **NP**-complete, $L_2 \in \mathbf{NP}$, and $L_1 \leq_p L_2$ then L_2 is **NP**-complete.

(c) If L is **NP**-complete and $L \in \mathbf{P}$, then **P**=**NP**.

Notice that parts (a) and (b) have close analogs in computability theory. The analog of part (c) is simply that if L is c.e.-complete then L is undecidable. Part (b) is the basic method for showing new problems are **NP**-complete, and part (c) explains why it is probably a waste of time looking for a polynomial-time algorithm for an **NP**-complete problem.

In practice a member of **NP** is expressed as a decision problem, and the corresponding language is understood to mean the set of strings coding YES instances to the decision problem using standard coding methods. Thus the problem Satisfiability is: Given a propositional formula F, determine whether F is satisfiable. To show that this is in **NP** we define the polynomial-time checking relation $R(x, y)$, which holds iff x codes a propositional formula F and y codes a truth assignment to the variables of F which makes F true. This problem was shown to be **NP**-complete in [Coo71] essentially by showing that for each polynomial-time Turing machine M which recognizes a checking relation $R(x, y)$ for an **NP** language L, there is a polynomial-time algorithm which takes as input a string x and produces a propositional formula F_x (describing the computation of M on input (x, y), with variables representing the unknown string y) such that F_x is satisfiable iff M accepts the input (x, y) for some y with $|y| \leq |x|^k$.

An important special case of Satisfiability is 3-SAT, which was also shown to be **NP**-complete in [Coo71]. Instances of 3-SAT are restricted to formulas in conjunctive normal form with three literals per clause. For example, the formula

$$(P \vee Q \vee R) \wedge (\bar{P} \vee Q \vee \bar{R}) \wedge (P \vee \bar{Q} \vee S) \wedge (\bar{P} \vee \bar{R} \vee \bar{S}) \tag{2}$$

is a YES instance to 3-SAT since the truth assignment τ satisfies the formula, where $\tau(P) = \tau(Q) = True$ and $\tau(R) = \tau(S) = False$.

Many hundreds of **NP**-complete problems have been identified, including SubsetSum (given a set of positive integers presented in decimal notation, and a target T, is there a subset summing to T?), many graph problems (given a graph G, does G have a Hamiltonian cycle? Does G have a clique consisting of half of the vertices? Can the vertices of G be colored with three colors with distinct colors for adjacent vertices?). These problems give rise to many scheduling and routing problems with industrial importance. The book [GJ79] provides an excellent reference to the subject, with 300 **NP**-complete problems listed in the appendix.

Associated with each decision problem in **NP** there is a search problem, which is, given a string x, find a string y satisfying the checking relation $R(x, y)$ for the problem (or determine that x is a NO instance to the problem). Such a y is said to be a *certificate* for x. In the case of an **NP**-complete problem it is easy to see that the search problem can be efficiently reduced to the corresponding decision problem, so that if $\mathbf{P} = \mathbf{NP}$ then all of the associated search problems have polynomial-time algorithms. For example, an algorithm for the decision problem Satisfiability can be used to find a truth assignment τ satisfying a given satisfiable formula F by, for each variable P in F in turn, setting P to True in F or False in F, which ever case keeps F satisfiable.

There are interesting examples of **NP** problems not known to be either in **P** or **NP**-complete. One example is the set of composite numbers, mentioned in the first section, with checking relation (1). Here it is conjectured that there is no polynomial-time Turing reduction from the the search problem (integer factoring) to the decision problem. Specifically, Miller [Mil76] showed how to determine in polynomial time whether a given number is composite, assuming the Extended Riemann Hypothesis, but a polynomial-time integer factoring algorithm is thought unlikely to exist. In fact an efficient factoring algorithm would break the RSA public key encryption scheme [ARS78] commonly used to send financial information over the internet.

The complement of an **NP**-complete language is thought not to be in **NP**; otherwise the complement of every **NP** language would be in **NP**. However the complement of the set of composite numbers (essentially the set of primes) was proved to be in **NP** by an interesting argument due to Pratt [Pra75], and hence is unlikely to be **NP**-complete.

There is an **NP** decision problem with complexity equivalent to that of integer factoring, namely

$$L_{fact} = \{\langle a, b\rangle \mid \exists d(1 < d < a \text{ and } d|b)\}$$

Given an integer $b > 1$, the smallest prime divisor of b can be found with about $\log_2 b$ queries to L_{fact}, using binary search. Using Pratt's theorem, it is easy to see that the complement of L_{fact} is also in **NP**: a certificate showing $\langle a, b\rangle$ is a NO instance of L_{fact} could be the complete prime decomposition of b, together with Pratt certificates proving that each of the prime factors is indeed prime. Thus it is considered unlikely that L_{fact} is **NP**-complete.

Computational complexity theory plays an important role in modern cryptography [Gol99]. The security of the internet, including most financial transactions, depend on complexity-theoretic assumptions such as the difficulty of integer factoring or of breaking DES (the Data Encryption Standard). If **P**= **NP** these assumptions are all false. Specifically, an algorithm solving 3-SAT in n^2 steps could be used to factor 200-digit numbers in a few minutes.

In [Sma98] Smale lists the **P** vs **NP** question as problem 3 of mathematical problems for the next century. However Smale is interested not just in the classical version of the question, but also a version expressed in terms of the field of complex numbers. Here Turing machines must be replaced by a machine model that is capable of doing exact arithmetic and zero tests on arbitrary complex numbers. The **P** vs **NP** question is replaced by a question related to Hilbert's Nullstellensatz: Is there a polynomial-time algorithm which, given a set of k multivariate polynomials over $\mathbb{C}$, determines whether they have a common zero? See [BCSS98] for a development of complexity theory in this setting, including the intriguing result that the Mandelbrot set is undecidable.

3. The Conjecture and Attempts to Prove it

Most complexity theorists believe that **P**$\neq$ **NP**. Perhaps this can be explained by considering the two possibilities: **P**$=$ **NP** and **P** $\neq$ **NP**.

Suppose first that **P** $=$ **NP** and consider how one might prove it. The obvious way is to exhibit a polynomial-time algorithm for 3-SAT or one of the other 1000 or so known **NP**-complete problems, and indeed many false proofs have been presented in this form. There is a standard toolkit available [CLR90] for devising polynomial-time algorithms, including the greedy method, dynamic programming, reduction to linear programming, etc. These are the subjects of a course on algorithms, typical in undergraduate computer science curriculums. Because of their importance in industry, a vast number of programmers and engineers have attempted to find efficient algorithms for **NP**-complete problems over the past 30 years, without success. There is similar strong motivation for breaking the cryptographic schemes which assume **P** $\neq$ **NP** for their security.

Of course it is possible that some nonconstructive argument, such as the Roberton/Seymour proofs mentioned earlier in conjunction with the Feasibility Thesis, could show that **P** $=$ **NP** without yielding any feasible algorithm for the standard **NP**-complete problems. However at present the best proven upper bound on an algorithm for solving 3-SAT is approximately 1.5^n, where n is the number of variables in the input formula.

Suppose on the other hand that **P** $\neq$ **NP**, and consider how one might prove it. There are two general methods that have been tried: diagonalization/reduction and Boolean circuit lower bounds.

The method of diaonalization with reduction has been used very successfully in computability theory to prove a host of problems undecidable, beginning with the Halting Problem. It has also been used successfully in complexity theory to prove super-exponential lower bounds for very hard decidable problems. For example, Presburger arithmetic, the first-order theory of integers under addition, is a decidable theory for which Fischer and Rabin [FR74] proved that any Turing machine deciding the theory must use at least $2^{2^{cn}}$ steps in the worst case, for some $c > 0$. In general, lower bounds using diagonalization and reduction relativize; that is they continue to apply in a setting in which both the problem instance and the solving Turing machine can make membership queries to an arbitrary oracle set A. However in [BGS75] it was shown that there is an oracle set A relative to which **P** $=$ **NP**, suggesting that diagonalization/reduction cannot be used to separate these two classes. It is interesting to note that relative to a *generic* oracle, **P** $\neq$ **NP** [BI87, CIY97].

A *Boolean circuit* is a finite acyclic graph in which each non-input node, or *gate*, is labelled with a Boolean connective; typically from $\{AND, OR, NOT\}$. The input nodes are labeled with variables $x_1, ..., x_n$, and for each assignment of 0 or 1 to each variable the circuit computes a bit value at each gate, including the output gate, in the obvious way. It is not hard to see that if L is a language over $\{0,1\}$ that is in **P**, then there is a polynomial-size family of Boolean circuits $\langle B_n \rangle$ such that B_n has n inputs, and for each bit string w of length n, when w is applied to the n input nodes of B_n, then the output bit of B_n is 1 iff $w \in L$. In this case we say that $\langle B_n \rangle$ *computes* L.

Thus to prove $\mathbf{P} \neq \mathbf{NP}$ it suffices to prove a super-polynomial lower bound on the size of any family of Boolean circuits solving some specific **NP**-complete problem, such as 3-SAT. Back in 1949 Shannon [Sha49] proved that for almost all Boolean functions $f : \{0,1\}^n \to \{0,1\}$, any Boolean circuit computing f requires at least $2^n/n$ gates. Nevertheless all attempts to find super-linear lower bounds for "explicitly given" Boolean functions have met with total failure; the best such lower bound proved so far is about $4n$. Razborov and Rudich [RR97] explain this failure by pointing out that all methods used so far can be classified as "Natural Proofs", and Natural Proofs for general circuit lower bounds are doomed to failure, assuming a certain complexity-theoretic conjecture.

The failure of complexity theory to prove interesting lower bounds on a general model of computation is much more pervasive than the failure to prove $\mathbf{P} \neq \mathbf{NP}$. It is consistent with present knowledge that not only could Satisfiability have a polynomial-time algorithm, it could have a linear time algorithm on a multitape Turing machine. The same applies for all 21 problems mentioned in Karp's original paper [Kar72]. There are complexity class separations which we know exist but cannnot prove. For example, consider the sequence of complexity class inclusions

$$\mathbf{LOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

A simple diagonal arugment shows that the first is a proper subset of the last, but we cannot prove any particular adjacent inclusion is proper.

As another example, let **LINEAR-SIZE** be the class of languages over $\{0,1\}$ which can be computed by a family $\langle B_n \rangle$ of Boolean circuits of size $O(n)$. It is not known whether either **P** or **NP** is a subset of **LINEAR-SIZE**, although Kannan [Kan82] proved that there are languages in the polynomial hierarchy (a generalization of **NP**) not in **LINEAR-SIZE**. Since if $\mathbf{P} = \mathbf{NP}$ then the polynomial hierarchy collapses to **P**, we have

Proposition 2: If $\mathbf{P} \subseteq \mathbf{LINEAR\text{-}SIZE}$, then $\mathbf{P} \neq \mathbf{NP}$.

This proposition could be interpreted as a method of proving $\mathbf{P} \neq \mathbf{NP}$, but a more usual belief is that the hypothesis is false.

A fundamental question in complexity theory is whether a source of random bits can be used to substantially speed up the recognition of some languages, provided one is willing to accept a small probability of error. The class **BPP** consists of all languages L which can be recognized by a randomized polynomial-time algorithm, with at most an exponentially small error probability on every input. Of course $\mathbf{P} \subseteq \mathbf{BPP}$, but it is not known whether the inclusion is proper. The set of prime numbers is in **BPP** [SS77], but it is not known to be in **P**. A reason for thinking that $\mathbf{BPP} = \mathbf{P}$ is that randomized algorithms are often successfully executed using a deterministic pseudo-random number generator as a source of "random" bits.

There is an indirect but intriguing connection between the two questions **P** = **BPP** and **P** = **NP**. Let **E** be the class of languages recognizable in exponential time; that is the class of languages L such that $L = L(M)$ for some Turing machine M with $T_M(n) = O(2^{cn})$ for some $c > 0$. Let **A** be the assertion that some language in **E** requires exponential circuit complexity. That is

Assertion A: There is $L \in \mathbf{E}$ and $\epsilon > 0$ such that for every circuit family $\langle B_n \rangle$ computing L and for all sufficiently large n, B_n has at least $2^{\epsilon n}$ gates.

Proposition 3: If **A** then **BPP** = **P**. If **not A** then **P** $\neq$ **NP**.

The first implication is a lovely theorem by Impagliazzo and Wigderson [IW97] and the second is an intriging observation by V. Kabanets which strengthens Proposition 2. In fact Kabanets concludes **P** $\neq$ **NP** from a weaker assumption than **not A**; namely that every language in L can be computed by a Boolean circuit family $\langle B_n \rangle$ such that for infinitely many n, B_n has fewer gates than the maximum needed to compute any Boolean function $f : \{0,1\}^n \to \{0,1\}$. But there is no consensus on whether this hypothesis is true.

Appendix: Definition of Turing machine

A Turing machine M consists of a finite state control (i.e. a finite program) attached to read/write head moving on an infinite tape. The tape is divided into squares, each capable of storing one symbol from a finite alphabet Γ which includes the blank symbol b. Each machine M has a specified input alphabet Σ, which is a subset of Γ, not including the blank symbol b. At each step in a computation M is in some state q in a specified finite set Q of possible states. Initially a finite input string over Σ is written on adjacent squares of the tape, all other squares are blank (contain b), the head scans the left-most symbol of the input string, and M is in the initial state q_0. At each step M is in a some state q and the head is scanning a tape square containing some tape symbol s, and the action performed depends on the pair (q, s) and is specified by the machine's transition function (or program) δ. The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state.

Formally a Turing machine M is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ where Σ, Γ, Q are finite nonempty sets with $\Sigma \subseteq \Gamma$ and $b \in \Gamma - \Sigma$. The state set Q contains three special states $q_0, q_{accept}, q_{reject}$. The *transition function* δ satisfies

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$$

If $\delta(q, s) = (q', s', h)$ the interpretation is that if M is in state q scanning the symbol s then q' is the new state, s' is the symbol printed, and the tape head moves left or right one square depending on whether h is -1 or 1.

We assume that the sets Q and Γ are disjoint.

A *configuration* of M is a string xqy with $x, y \in \Gamma^*$, y not the empty string, and $q \in Q$.

The interpretation of the configuration xqy is that M is in state q with xy on its tape, with its head scanning the left-most symbol of y.

If C and C' are configurations, then $C \overset{M}{\to} C'$ if $C = xqsy$ and $\delta(q, s) = (q', s', h)$ and one of the following holds:

$C' = xs'q'y$ and $h = 1$ and y is nonempty.
$C' = xs'q'b$ and $h = 1$ and y is empty.
$C' = x'q'as'y$ and $h = -1$ and $x = x'a$ for some $a \in \Gamma$.
$C' = q'bs'y$ and $h = -1$ and x is empty.

A configuration xqy is *halting* if $q \in \{q_{accept}, q_{reject}\}$. Note that for each non-halting configuration C there is a unique configuration C' such that $C \overset{M}{\to} C'$.

The *computation* of M on input $w \in \Sigma^*$ is the unique sequence $C_0, C_1, ...$ of configurations such that $C_0 = q_0w$ (or $C_0 = q_0b$ if w is empty) and $C_i \overset{M}{\to} C_{i+1}$ for each i with C_{i+1} in the computation, and either the sequence is infinite or it ends in a halting configuration. If the computation is finite, then the number of steps is one less than the number of configurations; otherwise the number of steps is infinite. We say that M *accepts* w iff the computation is finite and the final configuration contains the state q_{accept}.

References

[ARS78] L. Adleman, R. L. Rivest, and A. Shamir. A method for obtaining digital signature and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and Real Computation*. Springer-Verlag, New York, 1998.

[BGS75] Baker, Gill, and Solovay. Relativizations of the P =? NP question. *SICOMP: SIAM Journal on Computing*, 1975.

[BI87] M. Blum and R. Impagliazzo. Generic oracles and oracle classes. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 118–126, Los Angeles, CA, October 1987. IEEE Computer Society Press.

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

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 3–5 1971 1971.

[Coo90] Stephen A. Cook. Computational complexity of higher type functions. In Ichiro Satake, editor, *Proceedings of the International Congress of Mathematicians, Kyoto, Japan, 1990*, pages 55–69. The Mathematical Society of Japan, Springer-Verlag, 1991.

[CIY97] Stephen Cook, Russell Impagliazzo, and Tomoyuki Yamakami. A tight relationship between generic oracles and type-2 complexity theory. *Information and Computation*, 137(2):159–170, 15 September 1997.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* McGraw Hill, New York, 1990.

[Edm65] Jack Edmonds. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B*, 69:67–72, 1965.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, volume 7, pages 27–41. American Mathematical Society, Providence, RI, 1974.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[Gol99] Oded Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Springer-Verlag, Berlin Germany, 1999. ISBN 3-540-64766-X.

[IW97] Impagliazzo and Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *ACM Symposium on Theory of Computing (STOC)*, 1997.

[Kan82] Ravi Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and Control*, 55:40–56, 1982.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[Lev73] L. Levin. Universal search problems (in Russian). *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. English translation in Trakhtenbrot, B. A.: A survey of Russian approaches to Perebor (brute-force search) algorithms. Annals of the History of Computing, 6 (1984), pages. 384-400.

[Mil76] G. L. Miller. Reimann's hypothesis and tests for primality. *J. Comput. System Sci*, 13:300–317, 1976.

[Pra75] V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, August 1997.

[RS83] N. Robertson and P. D. Seymour. Graph minors I–XIII. *Journal of Combinatorial Theory B*, 1983-1995.

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

[Sho97] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal of Computing 26:1484–1509, 1997.

[Sma98] Steve Smale. Mathematical problems for the next century. *MATHINT: The Mathematical Intelligencer*, 20, 1998.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977.

[Tur37] Alan Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc. ser. 2*, 42:230–265, 1936-7.