Chapter 3
The One Million Dollars Problem

Alessandro Panconesi

Dedicated to the memory of a gentle giant

Abstract In May 1997, Deep Blue, an IBM computer, defeated chess world champion Garry Kimovich Kasparov. In the decisive Game 6 of the series, Kasparov, one of the most talented chess players of all time, was crushed. According to Wikipedia "after the loss Kasparov said that he sometimes saw deep intelligence and creativity in the machine's moves". Can machines think? Scientists and philosophers have been debating for centuries. The question acquired powerful momentum in the beginning of the twentieth century, when the great mathematician David Hilbert launched an ambitious research program whose aim was to prove conclusively that his fellow mathematicians could be supplanted by what we now call computers. He believed that mathematics, one of the most creative human endeavours, could be done by them. In spite of the rather dramatic denouement of Hilbert's program, the question he posed is not yet settled. The ultimate answer seems to be hidden in a seemingly silly algorithmic puzzle: a traveling salesman is to visit N cities; can the cheapest tour, the one that minimizes the total distance travelled, be computed efficiently?

Our story begins in Paris on March 24, 2000, when the Clay Foundation, during a ceremony held at the Collège de France, announced an initiative intended to *"celebrate the mathematics of the new millennium."* The American foundation instituted seven prizes of one million dollars each for the resolution of seven open problems of mathematics, selected by a committee of distinguished mathematicians for their special significance. Among these seven "Millennium Problems", alongside

A. Panconesi (✉)
Dipartimento di Informatica, Sapienza Università di Roma, via Salaria 113, 00198 Roma, Italy
e-mail: ale@di.uniroma1.it

G. Ausiello and R. Petreschi (eds.), *The Power of Algorithms*,
DOI 10.1007/978-3-642-39652-6_3,

classics such as the Riemann Hypothesis and the Poincaré Conjecture, a newcomer stood out, a computer science problem known as *"P vs. NP"*. Formulated just 40 years ago in the Soviet Union by Leonid Levin, a student of the great Andrei Kolmogorov, and simultaneously and independently in the West by Stephen A. Cook of the University of Toronto in Canada, and by Richard M. Karp of the University of California at Berkeley, this algorithmic problem is considered to be of fundamental importance not only for the development of mathematics and computer science but also for its philosophical implications. Its resolution will tell us something fundamental about the intrinsic limitations of computers, regardless of the state of technology. At first blush, the problem does not appear forbidding or especially noteworthy but, as they say, looks can be deceiving. One of the many ways in which it can be formulated is the following, known as the Traveling Salesman Problem (henceforth abbreviated as TSP): suppose that we have a map with N cities that must be visited by means of the shortest possible route. That is, starting from any city, we must visit each city once and only once, and return to the city we started from. Such an itinerary is called a tour and the goal is to find the shortest possible one.

Computers seem ideally suited for such tasks. The resulting mathematical challenge is to find an algorithm (or, in the final analysis, a computer program) that is able, given any input map, to find the shortest tour. The crucial requirement that our algorithm must satisfy is to be efficient. The one million dollars problem is the following: does there exist such an algorithm?

Although it may seem surprising, no one has ever managed to find an efficient algorithm for the TSP, and it is conjectured that none exists! Note that it is completely trivial to develop an algorithm that finds the shortest tour. Already in the first year of college computer science students are typically able to come up with this algorithm:

> Enumerate all possible tours among the N cities, measuring the length of each one of them, and record the shortest one observed so far.

In other words, if we generate by computer all possible tours, measure the length of each, and keep the shortest distance observed so far, we will eventually find the shortest one. The catch, of course, is that this algorithm, as easily programmable as it may be, is not efficient: its execution time is so high to make it completely useless. To see this, note that if you have N cities the number of all possible tours is given by:

$$(N-1) \times (N-2) \ldots \times 4 \times 3 \times 2 \times 1.$$

(Start from any city, then we have $N-1$ choices for the second city, $N-2$ for the third, and so on). This number is denoted as $(N-1)!$ and, even for small values of N, it is so large as to require a certain amount of thought in order to be comprehended. For example, $52!$ is equal to:

80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,883,277,824,000,
000,000,000[1]

How big is this number? Expressed in billionths of a second it is at least 5,000 billion times greater than the age of the universe. In industrial applications it is necessary to solve the traveling salesman problem for values of N in the order of the thousands, and beyond (Fig. 3.1). Numbers like 5,000! or 33,474! would require quite some time and ingenious mathematical notation to be grasped and are immeasurably greater than a puny, so to speak, 52!. It is therefore quite clear that the algorithm described is of no help in solving realistic instances of the TSP. Moreover, and this is a fundamental point, *no technological improvement will ever make this algorithm practical.*

In summary, the difficulty with the TSP is not that of finding an algorithm that computes the shortest tour, but to find an efficient one. Surprisingly, nobody has been able to find such a method and not for lack of good-will. The problem is quite relevant from the point of view of the applications, where it is encountered often and in different guises. For decades, researchers have tried to come up with efficient algorithms for it, but without success.

In conclusion, the algorithmic problem for which one million dollars are at stake is this: find an efficient algorithm (i.e., a programmable procedure) for the TSP, or prove that such an algorithm does not exist. This deceptive-looking problem is at the heart of a series of far-reaching mathematical, technological and even philosophical issues. To understand why we return to Paris, but going back in time.

3.1 Paris, August 8, 1900

> Who of us would not be glad to lift the veil behind which the future lies hidden; to cast a glance at the next advances of our science and at the secrets of its development during future centuries? What particular goals will there be toward which the leading mathematical spirits of coming generations will strive? What new methods and new facts in the wide and rich field of mathematical thought will the new centuries disclose?

Thus began the famous keynote address that the great mathematician David Hilbert, at age 40, gave at the International Congress of Mathematicians in 1900.[2] With his speech Hilbert who, along with Henri Poincaré, was considered the most influential mathematician of the time, outlined an ambitious program to shape the future of mathematics. The centerpiece of his proposal was his now famous list of 23 open problems that he considered to be central to the development of the discipline.

[1]The calculation, by no means trivial, is taken from http://www.schuhmacher.at/weblog/52cards.html

[2]Maby Winton Newson [85].

Fig. 3.1 A TSP input instance consisting of 15,112 German cities. The calculation of the shortest tour (shown in the figure) kept two teams of researchers from Rice University and Princeton University busy for several months and required considerable ingenuity and computing resources: 110 parallel processors, equivalent to 22.6 years of calculation on a single processor. In 2006, another instance with 85,900 cities was solved, again with massive parallel resources. The computation time, translated for a single processor, was 136 years. (See "Traveling salesman" on Wikipedia). Note that this considerable deployment of resources was needed to solve just two isolated instances of the problem – the methods developed have no general applicability

What makes this speech so memorable in the history of ideas is the fact that the twentieth century has been a golden age for mathematics. With his 23 problems Hilbert succeeded in the incredible undertaking of shaping its development to a large extent for a 100 years. On the problems posed by Hilbert worked titans such as Kurt Gödel, Andrei Kolmogorov and John (Janos) von Neumann, and more than once the Fields Medal, the highest honor in mathematics, has been assigned for the resolution of some of Hilbert's problems. In fact, the event that we mentioned at the beginning, the announcement of the seven Millennium Problems at the Collège de France, was meant to celebrate and hopefully repeat the fortunate outcome of that memorable speech, exactly 100 years later.

In modern terms, one of Hilbert's main preoccupations, which was already apparent in his work on the foundations of geometry *"Grundlagen der Geometrie"* (Foundations of Geometry) and that became more explicit with the subsequent formulation of his program for the foundations of mathematics, was the pursuit of the following question: can computers do mathematics, and if so to what extent?

One of the underlying themes of Hilbert's vision was an ambitious program of "mechanization" or, in modern terms, computerization of the axiomatic method, one of the pillars of modern science that we have inherited from Greek antiquity and of which "The Elements" by Euclid, a book that remained a paragon of mathematical virtue for many centuries, is probably the most influential example. The axiomatic method, in no small measure thanks to the revitalizing work of Hilbert, now permeates modern mathematics. According to it, mathematics must proceed from simple assumptions, called axioms, by deriving logical consequences of these through the mechanical application of rules of inference. To give an idea of what it is about let us consider a toy axiomatic system whose "axioms" are the numbers 3, 7 and 8 and whose "inference rules" are the arithmetic operations of addition and subtraction. Starting from the axioms, we apply the inference rules in every possible way to generate other numbers. For instance, we can derive $5 = 8 - 3$, and $2 = 8 - (3 + 3)$, etc. The new numbers thus obtained are the "theorems" of this theory. The "theorems" obtained can be combined with each other and with the axioms to generate other "theorems" and so on. In our example, since 2 is a theorem and $+$ is an inference rule, we can generate all even numbers. Thus 4, 6, etc., are theorems whose corresponding proofs are $2 + 2$, $2 + 4$, and so on. The axiomatic systems that are used in mathematics are obviously much more expressive and complex and, in general, are meant to manipulate predicates – logical formulas that can be true or false – rather than raw numbers. Predicates are sentences that express the universe of mathematical statements, things like *"There are infinitely many prime numbers"*, *"The equation* $x^n + y^n = z^n$ *has no integer solutions for* $n > 2$*"*, and so on. But, by and large, the essence of the axiomatic method is the one described: precise inference rules are applied to predicates in a completely circumscribed and mechanical fashion, in the same way as the four arithmetic operations handle numbers. The axiomatic method offers the best possible guarantees in terms of the correctness of the deductions. If the starting axioms are propositions known to be true, and they do not contradict each other, by applying the rules of inference (which are nothing other than simple rules of logical deduction) we can only generate other true statements. For the Greeks the axioms were self-evidently true by virtue of their sheer simplicity. (In contrast, the modern point of view requires only that they do not contradict each other.) For instance, one of the axioms of Euclidean geometry is the following, arguably self-evident, statement: *"If we take two points in the plane, there is a unique straight line that goes through them"*.[3] The mechanical aspect of the axiomatic approach is clear: given that the application of an inference rule is akin to performing an arithmetical operation, and since a mathematical proof is nothing other than the repeated application of such elementary and mechanical operations, that from a set of premises lead to a conclusion, a computer should be able to reproduce such a process. And indeed, a computer is certainly able to mimic it. For instance, given

[3] Of course, as we now know with hindsight 2,000 years later, assuming that axioms are self-evident truths is a slippery slope, exemplified by the development of non-Euclidean geometry.

an alleged proof of a mathematical statement such as *"There exist infinitely many prime numbers"* a computer can rather easily determine if the given proof is correct or not. But beware! This by itself does not mean that computers can supplant mathematicians. The aspects of the matter are quite complex, but for our purposes it is sufficient to focus on a fundamental dichotomy, the difference between inventing and verifying. To check the correctness of a given mathematical proof – i.e. its verification – is a mental activity fundamentally quite different from the act of finding it. The latter is a creative act while the former is not.

To create a mathematical proof is a convoluted and complex psychological process in which logic does not play a primary role at all, necessarily supplanted by mysterious visions that suddenly emerge from the subconscious. In contrast, to verify a demonstration (i.e. to convince oneself of its correctness) is a task that can be intellectually very demanding but that does not require creativity. In order to verify a proof one must check that the conclusions follow, now logically, from the premises through the correct application of rules of inference. This is an essentially mechanical process, somewhat similar to performing a long series of complicated arithmetical operations. It is in this second phase, during which the correctness of the visions and insights of the creative process of mathematical discovery are put under rigorous scrutiny, that the task of logic, very limited in the discovery phase, becomes predominant. In the words of Alain Connes, the great French mathematician, winner of the Fields Medal,

> Discovery in mathematics takes place in two phases. [...] In the first, the intuition is not yet translated into terms that you can communicate in a rational manner: here is the vision, [...] a kind of poetic inspiration, which is almost impossible to put into words. [...] The second phase, that of demonstration, is an act of verification: it requires high concentration and a kind of extreme rationalism. But luckily there is still a vision, which activates the intuition of the first phase, does not obey certainty and is more akin to a burst of poetic nature.[4]

By transcribing the complex creative process that leads to the discovery of mathematical truths in a succession of elementary and verifiable steps, the mathematician submits voluntarily to the most exacting verification discipline: the tortuous path that leads to the inescapable conclusion is laid out carefully and thoroughly in a long sequence of logical, and hence unimaginative and ultimately mechanical, steps. In such a way not only a machine, but even a person who is completely devoid of mathematical talent can be convinced!

Unfortunately, the teaching of mathematics, even at university level, is almost exclusively confined to the hard discipline of verification. This hides its creative dimension and it is not surprising that mathematics suffers from the undeserved reputation of being a dry discipline. That one of the most imaginative human activities must suffer from such an injustice is a tragedy comparable to the destruction of the great masterpieces of art.

[4] Alain Connes [19].

3.2 *"Calculemus!"*

Let us go back to Hilbert and his influential school, who believed that it was possible to push the level of formalization of the axiomatic method to a point where it becomes entirely mechanical. It may seem surprising now, but Hilbert and his followers, including geniuses like von Neumann, not only believed that the verification process but also the discovery of mathematical truths could be made by a computer! As we discussed, to find a mathematical proof and to verify it are two completely different psychological processes. How could Hilbert think that even the former, a quintessentially creative act, could be carried out by a machine? In Science, often the power of an idea stems not so much from the fact that it ultimately turns out to be correct, but rather from its vision, its power to summon *"the leading [minds] of coming generations"* who, in their quest for truth, will produce revolutionary new knowledge, and a leap forward in the understanding of the world. Hilbert's Program for the "computerization" of mathematics is a case in point. Its great merit was to frame a set of bold, profound questions of great mathematical, and even philosophical, import in such a way that they could be fruitfully tackled in precise and concrete mathematical terms. Hilbert's promethean conjectures gave impetus to a steady flow of sensational results on the foundations of mathematics that culminated, in 1931, with the Incompleteness Theorem of Kurt Gödel, one of those achievements that, like the great masterpieces of art and music, will forever continue to inspire humankind to accomplish great and noble deeds: *"Kurt Gödel's achievement in modern logic is singular and monumental – indeed it is more than a monument, it is a landmark which will remain visible from afar, in space and time."* (John von Neumann).

One of the ways in which Hilbert's program concretely took shape is the so-called "problem of decidability". Leaving aside several important technical issues, in a nutshell Hilbert believed that one could design an algorithm (or, equivalently, write a computer program) which, on receiving as input a mathematical statement such as *"There are infinitely many prime numbers"* or *"There exist planar graphs that are not 4-colorable"*, would correctly output the correct answer – in our examples, respectively, *"Yes, this sentence is true"* and *"No, this statement is false"*. The question of whether such an algorithm exists is called the *"Entscheidungsproblem"*, the Problem of Decision. In fact, Hilbert did not consider it quite as an open problem but almost as a foregone conclusion. Hilbert believed that computers could not only verify the correctness of mathematical proofs, but find them! This is a less ambitious version, restricted to mathematics "only" of the dream cherished by the mathematician, philosopher, naturalist, diplomat and... computer scientist Gottfried Wilhelm Leibniz who thought that it was possible to settle differences of opinion through logic, by means of a computer:

> The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes

> among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right.[5]

In order to address the Entscheidungsproblem it was first of all necessary to understand what an algorithm is. In the early decades of the twentieth century, several people, including Emil Post, Alonzo Church, and Kurt Gödel captured the intuitive notion of algorithm with precise mathematical definitions. But it was Alan Turing, perhaps the first genuine computer scientist, who proposed the most satisfactory characterization. Turing devised an abstract machine, now known as the Turing Machine that, to all intents and purposes, from the conceptual point of view is a full-fledged, modern digital computer capable of executing any computer program.[6] In modern terms, for Turing an algorithm was nothing other than a computer program written in a programming language like C, Java or Python. In other words, an algorithm is any process of calculation that is programmable on a modern computer. This was a rather exceptional contribution for the simple reason that at the time computers did not exist. To develop his theory Turing invented an abstract digital computer along with a primitive but powerful programming language, a sort of assembly language with which one can express, or more precisely program, any computation. The beauty and power of the approach were such that it allowed Turing to demonstrate by means of an uncannily simple argument that the Entscheidungsproblem has a flatly negative answer: there is no algorithm that can determine whether mathematical statements given in input are true or false.[7] Now, someone might breathe a sigh of relief – human creativity is saved! But not quite: as we shall see, the computer is a tough nut to crack.

Although ultimately the research program advocated by Hilbert proved to be fundamentally flawed, to the point that, essentially, the truth turned out to be the exact opposite of what he believed, the amazing discoveries made in its wake opened grand (and in some sense humbling) new vistas and jump-started whole new disciplines. In particular, although the advent of the digital computer cannot be attributed solely to the research related to Hilbert's program, there is no doubt that it played a key role. The ENIAC, perhaps the most successful early electronic computer,[8] was developed at Princeton in the context of a project supervised by

[5] Cited by Wikipedia's article on Leibniz: http://en.wikipedia.org/wiki/Gottfried_Wilhelm_Leibniz

[6] Turing invented two abstract devices, the Turing machine, which corresponds to a computer running a specific program, and the universal Turing machine, corresponding to a general purpose computer able to execute any program. See Chap. 1.

[7] The same conclusion was reached independently by Alonzo Church at Princeton about a year earlier thanks to the invention of the λ-calculus, that in essence is a functional programming language of which LISP is perhaps the best-known example.

[8] The first electronic computer is due to Konrad Zuse, who set one up in Berlin in 1941. In a remarkable instance of scientific short-sightedness his invention was considered by the German authorities to be "strategically irrelevant" and not developed. The indomitable Zuse immediately after the war founded a company, the second ever to market the computer, trying to plant a fertile seed in the arid soil of post-war Europe. Meanwhile, overseas the first computers were triggering the computer revolution.

John von Neumann who, besides being an early believer of Hilbert's program, knew well all developments arising from it, including those due to Alan Turing. The two became acquainted in Princeton where Turing spent some time as a post-doctoral researcher. Moreover, the difference between a Turing Machine and a modern computer is "only" technological, and indeed immediately after the end of the war Turing was able to develop in Britain a modern electronic computer called ACE.

3.3 Finding Is Hard: Checking Is Easy

The dichotomy concerning mathematical proofs that we introduced, the mechanical nature of the verification process versus the complex creativity of mathematical discovery, is at the core of the one million dollars problem. To understand why, let us consider the following algorithmic problem known as Satisfiability (henceforth SAT). We shall adopt the convention normally used in college textbooks, where an algorithmic problem is described by specifying the input data and the type of result you want in output:

Satisfiability (SAT)
Input: a Boolean formula $F(x_1, x_2, \ldots, x_N)$ with N variables $x_1, x_2, \ldots, x_N$, each of which may assume either the value true or the value false.
Question: is the formula F satisfiable, i.e., is there a truth assignment to the variables that makes F true?

SAT is the propositional-logic counterpart of the familiar algebraic problem of finding the zeroes of a polynomial. Given a polynomial like $x^2 - 2xy + y^2 = 0$ or $x^2 + 3y(1 - z) + z^3 - 8 = 0$ one seeks values for the variables x, y, and z that verify the expressions. With SAT, logical connectives (and, or, implies, etc.) are used instead of arithmetic operations and the variables can assume the values true or false instead of numerical values. An assignment is sought that makes the formula true (see box "Satisfiability of Boolean expressions").

Satisfiability of Boolean expressions

This computational problem concerns so-called Boolean expressions, an example of which is the following:

$$((x_1 \vee x_3 \rightarrow \neg x_2) \rightarrow (x_2 \rightarrow x_5 \wedge x_4)) \wedge \neg(x_1 \vee (\neg x_5 \wedge x_3)).$$

(continued)

(continued)

The variables x_1, x_2, x_3, x_4, x_5 may assume either the value T (true) or F (false). The symbols $\rightarrow$, $\neg$, $\wedge$, $\vee$ are logical (also called Boolean) *connectives* whose semantics is defined in the following truth tables. These are just a few of all logical connectives. The negation inverts the truth value of the expression it operates on (an expression can be a single variable or a Boolean formula, e.g. $(\neg x_5 \wedge x_3)$, $(x_2 \rightarrow x_5 \wedge x_4)$, etc.):

E	$\neg E$
T	F
F	T

The implication forbids us to derive falsehoods from truth (A and B denote Boolean expressions):

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

And finally the disjunction (or) is true when at least one of the *operands* is true, while the conjunction (and) is true only when both operands are true:

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

The above expression is satisfiable with the truth assignment $x_1 = x_2 = x_3 = F$ and $x_4 = x_5 = T$. An example of unsatisfiable formula is the following:

$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$$

(the reader should verify this).

The natural algorithmic problem associated with SAT is the following: find an *efficient* algorithm such that, given a logical formula as input, it determines whether the formula is satisfiable or not.

SAT is an important computational problem for several reasons. From a practical standpoint, many application problems can be expressed in a natural way as a satisfiability problem. Making sure that the complex software systems that control

an Airbus or the railroad switches of a busy railway station do not contain dangerous programming bugs boils down to a satisfiability problem. More generally, model checking, the discipline whose aim is to ascertain whether a specific software system is free from design errors, is, to a large extent, the quest for algorithms as efficient as possible for SAT. An efficient algorithm for SAT therefore would have a significant industrial impact. The state of the art with respect to the algorithmics of SAT is rather fragmented. There are computer programs, so-called SAT-solvers, that work quite well for special classes of formulas. But apart from such very restricted situations, all known algorithms run in exponential time and hence perform very poorly in general.

SAT encapsulates well the dichotomy between finding and checking. Given a truth assignment, it is child's play for a computer to determine whether it satisfies the formula. In order to check whether a polynomial is zero, one just replaces the numerical values specified for the variables and performs simple algebraic operations. Similarly, with a Boolean formula one just replaces the truth values specified for the variables and performs a series of logical operations to verify if the formula is satisfied. Computers can do this very efficiently, in time proportional to the length of the formula. The verification of a given truth assignment, therefore, is a computationally easy task. An entirely different matter is to find a truth assignment satisfying the given input formula, if it exists. Note that, similarly to what we saw for TSP, in principle such a finding algorithm exists. Since each variable can take only the two values true or false, a formula with N variables has 2^N possible assignments. It is easy to write down a computer program that generates all these assignments and checks them one by one. If one of them satisfies the formula we have found a satisfying assignment, otherwise, if none does, we can conclude that it is unsatisfiable. But here too the same caveat that we discussed for TSP applies: 2^N is too huge a number, even for small values of N (see box "Exponential growth is out of reach for technology"). The enumerative method proposed is thus completely useless, and it will remain so forever since no improvement in computer technology can withstand exponential growth. To appreciate how fast the function 2^N grows the reader can try to solve the following fun exercise: if you fold a sheet of paper 50 times, thereby obtaining a "stack of paper" 2^{50} times thicker than the original sheet, how high would it be? The answer is really surprising!

Exponential Growth Is out of Reach for Technology

Imagine we have designed three different algorithms for a computational problem whose complexities (running times) grow like N, N^2 and 2^N, where N is the input size in bits. The next table compares their running times when the algorithms are run on a processor capable of performing 200 million operations per second, somewhat beyond the current state of the art. The acronym AoU stands for "Age of the Universe":

(continued)

(continued)

	$N = 10$	$N = 100$	$N = 1,000$
N	0.056 ns	0.56 ns	0.0056 μs
N^2	0.056 ns	0.056 μs	0.0056 ms
2^N	0.057 μs	> 16 AoU	$> 1.39\,10^{272}$ AoU

It is instructive to see what happens to these running times when the technology improves. Even with computers one billion times faster – quite a significant leap in technology! – the exponential algorithm would still take more than 10^{261} times the age of the universe for $N = 1,000$. Keep in mind that in realistic applications N is of the order of millions if not more. Technology will never be able to match exponential growth.

Thus satisfiability embodies the following dichotomy: while it is computationally easy to check if a given truth assignment satisfies the input formula, no one knows whether it is possible to efficiently find such an assignment. The dichotomy is reminiscent of a familiar situation for students of mathematics and mathematicians. To check whether a given proof is correct (e.g., to study the proof of a theorem) is a task that may be quite demanding, but that is relatively easy if compared with the conceptual effort needed to find, to create, a mathematical proof. This similarity with the computational tasks of finding and verifying may seem shallow and metaphorical but, in fact, it is surprisingly apt. As we shall see, the problem of devising an efficient algorithm for satisfiability is equivalent to that of finding a general procedure capable of determining whether any given mathematical statement admits a proof that is understandable by the human mind!

Another surprising fact is that SAT and TSP from a computational point of view are equivalent, different abstractions that capture the same phenomenon. There is a theorem, whose proof is beyond the scope of this book, which shows that if there were an efficient algorithm for SAT, then there would exist one for TSP too, and vice versa. Therefore, an equivalent formulation of the one million dollars problem is: find an efficient algorithm for SAT or prove that such an algorithm does not exist.

3.4 The Class NP

The dichotomy "checking is easy" vs. "finding is difficult" is shared by a large class of computational problems besides SAT. Let us see another computational problem that exhibits it:

Partitioning
Input: a sequence of N numbers $a_1, a_2, \ldots, a_N$
Question: can the sequence be partitioned, i.e. divided into two groups in such a way that the sum of the values in one group is equal to the sum of the values in the other group?

For example, the sequence 1, 2, 3, 4, 5, 6 is not partitionable (because the total sum of the numbers in the whole sequence is 21, which is not divisible by 2), while the sequence 2, 4, 5, 7, 8 can be partitioned into the two groups 2, 4, 7 and 5, 8. Indeed, $2 + 4 + 7 = 5 + 8$. PARTITIONING is a basic version of a fun problem in logistics called KNAPSACK: we are given a knapsack (which can be thought of as an abstraction for a truck, a ship or an aircraft) of maximum capacity C, and a set of N objects, each of which has a weight and a value. The goal is to select a subset of objects to carry in the knapsack in such a way that their total value is maximum, subject to the constraint that their total weight does not exceed the knapsack's capacity.

If we look for an efficient algorithm for PARTITIONING we soon realize that a brute force approach is out of the question. Here too the space of all possible partitionings, given an input sequence of N values, has 2^N candidates (because every element has two choices, whether to stay in the first or in the second group) and therefore the trivial enumerative procedure is useless. The verification procedure, however, once again is rather straightforward and computationally inexpensive: given a partitioning of the input sequence, just compute two sums and check if they are the same.

The class NP contains all computational problems that are verifiable efficiently, like SAT and PARTITIONING. This is a very important class because it contains a myriad of computational problems of great practical interest. Indeed, NP problems are encountered in all fields where computers are used: biology, economics, all areas of engineering, statistics, chemistry, physics, mathematics, the social sciences, medicine and so on and so forth. One of the reasons why NP is so rich with problems of practical importance is due to the fact that it captures a huge variety of optimization problems. We already saw an example of an optimization problem, TSP: given the input (a map with N cities) we are seeking the optimal solution (the shortest tour) among a myriad of candidates implicitly defined by the input (all possible tours). Thus, in optimization we are seeking an optimal value inside a typically huge space of candidates. In contrast, NP is a class of decision problems, computational problems whose answer is either yes or no. There is however a standard way to transform an optimization problem into a decision problem. As an example, consider the following budgeted variant of TSP:

Budgeted TSP
Input: a map with N cities and a budget B (expressed, say, in kilometers)
Question: Is there a tour whose length is no greater than B?

The problem asks if there is a way of visiting all the cities within the budget available to us, while in the original TSP we were looking for the tour of minimum length. Notice that now the answer is either yes or no. This budget trick is rather standard. With it we can turn any optimization problem into a decision problem. From a computational point of view, an optimization problem and its decision version are equivalent: if there exists an efficient algorithm to solve the optimization problem then there exists an efficient algorithm to solve the corresponding decision problem, and vice versa. In the case of TSP, for instance, if we solve the optimization problem by computing the shortest tour, then we can immediately answer the question *"Is there is a tour of length not exceeding the budget?"* thereby solving the decision problem. The converse is also true: If we can answer the question *"Is there is a tour of length not exceeding the budget?"* then we can efficiently determine the shortest tour (the slightly involved proof is beyond the scope of this book).

In order to appreciate the huge variety of contexts in which optimization problems arise it can be instructive to see one of the many examples taken from biology. DNA is a sequence of genes each of which is a set of instructions, a recipe, with which the cell can synthesize a specific protein. It is known that many species share the same gene pool, but the genes are positioned differently along the genome, i.e., genomes are anagrams of each other. For example a bonobo may have the following genome (letters represent genes):

PRIMATE.

While chimpanzees and humans might otherwise have, respectively, the genomes RPTIMEA and MIRPATE. Biologists postulate different mutation mechanisms that are responsible for the shuffling of genes during the course of evolution. One of these is the so-called "reversal". A reversal is due to a transcription error that literally inverts a segment of the genome, as in this example (the reversed part appears in bold):

PR**IMAT**E → PR**TAMI**E

Biologists believe that reversals are rare events, taking place at intervals of millions of years. In our example the distance between bonobos and humans is only one reversal, **PRIM**ATE → **MIRP**ATE, while both species turn out to be farther away from chimpanzees (the reader can try to determine the minimum number of reversals between the three different species). Biologists assume that

Fig. 3.2 From *left* to *right*: a young chimpanzee who already shows signs of aggressiveness, two bonobos in a normal loving attitude, and a lovely child

the minimum number of reversals to transform a genome into another is a good estimate of the evolutionary distance between the two species. Therefore, at least in this example, the lustful bonobo would be our closer relative than the irascible chimpanzee (Fig. 3.2). Thus, given two genomes, it is of interest to determine the minimum number of reversals between them. The corresponding decision version of this optimization problem is the following[9]:

Sorting by Reversals
Input: Two genomes G_1 and G_2 with the same set of genes, and a budget T.
Question: Is it possible to transform G_1 into G_2 with no more than T reversals?

The reader should try to convince him-/herself that this computational problem is in NP and that it is computationally equivalent to its optimization version (i.e., given two genomes find the minimum number of reversals that transforms one into the other).

In summary, the class NP captures a very large class of optimization problems of great practical importance. NP also contains other non-optimization problems that are of enormous industrial interest, such as the automatic verification of software and of robotic systems. But there is more. NP is also interesting from the point of view of the foundations of mathematics, a topic to which we now turn.

[9]The algorithmic structure of this important combinatorial problem was investigated by my late friend Alberto Caprara, who died tragically in a mountain accident. Alberto was able to establish that the problem is NP-complete (see next paragraph), thus settling an important open problem in the field of computational biology. The proof was quite a tour de force, and conveyed a sense of serene strength, quite like Alberto. This chapter is dedicated to him.

3.5 Universality

A beautiful and fundamental theorem concerning the class NP was established simultaneously and independently by Leonid Levin and Stephen A. Cook in the early 1970s. The theorem states that the class NP contains universal problems. A computational problem X in NP is universal (but the correct terminology is NP-complete) if it has the following property: if there is an efficient algorithm A for X, then, taken any other problem Y in NP (by means of a general procedure) we can transform algorithm A into an efficient algorithm B for Y. An immediate consequence of universality is: if we find an efficient algorithm for a universal problem then we can also find an efficient algorithm for every other problem in NP.

For example, satisfiability is universal: if one day you could find an efficient algorithm to decide whether a given Boolean formula is satisfiable, then it would be possible to efficiently solve any problem in NP. This result is known as Cook's Theorem (Levin had demonstrated the universality of a different problem, known as tessellation). Despite being a very strong property, universality appears to be the norm rather than the exception. Universal problems are plenty and we have encountered some of them already: satisfiability, partitioning, sorting by reversals and the budgeted versions of TSP. Other examples of universal problems are the decision versions of natural optimization problems like finding the longest path in a network, determining the smallest set of lecture halls to host university classes, packing a set of items with the smallest number of bins, and thousands more. For none of them it is known whether an efficient algorithm exists.

The concept of universality is very useful because it allows us to greatly simplify a complex picture without loss of generality. The class NP contains a bewildering variety of computational problems, how can we tell if there are efficient algorithms for all of them? Universality allows us to focus on just one problem, provided that it is universal. If we could find an efficient algorithm for a universal problem like TSP or satisfiability then we could efficiently solve any problem in NP. Conversely, if we could show that an efficient algorithm for TSP or satisfiability does not exist, then no universal problem may admit an efficient algorithm. In this way, the study of the "computational complexity", as it is called, of a myriad of problems can be reduced to the study of a single mathematical object.

3.6 The Class P

So far we have freely used the term "efficient algorithm", but what exactly do we mean by that?[10] The standard definition given in textbooks is the following: an algorithm is efficient if its computation time grows polynomially with the size of

[10]We revisit here concepts discussed in Chap. 2.

the input. It is intuitive that the larger the input the longer the calculation takes. For example, a multiplication algorithm will need more and more time to perform the product as the numbers to multiply become larger and larger. One of the missions of the theory of algorithms is to determine as precisely as possible the running time of algorithms, in a technology-independent manner (something that can be achieved thanks to appropriate mathematical models such as the Turing machine). If N is the number of bits needed to represent the input, the running time of the algorithm is "polynomial" if its computation time grows at most as a polynomial. For example, N, N^2, $N \log N$, $N^{3/2}$ are polynomial computation times ($\log N$ too is fine since it grows much slower than N). In contrast, we saw that the enumerative algorithms for SAT and TSP require exponential time, at least 2^N and $(N-1)!$ steps respectively, and we know that this kind of "complexity" is prohibitive, regardless of the technology.

NP is the class of problems whose solutions can be verified to be correct in polynomial time, while P is the class of problems for which the solution can be not only verified, but found in polynomial time. Various chapters of this book discuss important problems that are in P, such as the calculation of shortest paths in a network, the retrieval of information from the Web, and several fascinating examples taken from cryptology and computational biology. Other important computational tasks that are in P are the multiplication of matrices, solving systems of linear equations, determining whether a number is prime, the familiar "find" operation to look for the occurrence of a word inside a text, and the very general problem of linear programming (which asks us to optimize a linear function subject to linear constraints). The class P captures the set of all problems that are computationally tractable. We thus come to the most general formulation of the one million dollars problem:

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

If P = NP then every problem in NP can be solved efficiently, i.e., in polynomial time. In contrast, if P $\neq$ NP, as it is widely conjectured, then no universal problem can be in P. (Why? The reader should try to answer.) If this is indeed the case we will have to accept our lot: we live in a world where computational intractability is the rule rather than the exception because, as we have discussed, NP contains a myriad of natural occurring computational problems that are universal (Fig. 3.3).

The theory of algorithms has thus developed two main approaches to ascertain the so-called "algorithmic complexity" of a problem. Given a computational problem, we can either exhibit a polynomial-time algorithm for it, or show that it is universal for NP (i.e., NP-complete). In the latter case we do not know for sure that the problem does not admit efficient algorithms, but we do know that finding one is a very difficult mathematical challenge, worth at least one million dollars! Let us stress once again that this theory is independent of technology: the watershed between exponential and polynomial complexity is such that no technological improvement will ever make exponential-time algorithms practical.

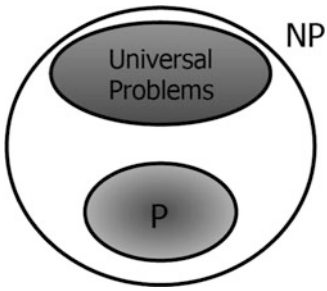

Fig. 3.3 The world as seen from NP. If P = NP then the three classes coincide, but if P ≠ NP then the rich class of Universal (NP-complete) problems is disjoint from P

After having discussed the practical implications of the one million dollars problem, let us look at the other side of this precious coin, the link with the foundations of mathematics.

3.7 A Surprising Letter

As recalled, the theory of NP-completeness was developed in the early 1970s independently in the Soviet Union and in the West. An exceptional discovery made in the basement of the Institute for Advanced Study at Princeton in the late 1980s sheds new light on the history of the P vs. NP problem: it had been prefigured by no less than Kurt Gödel. In a letter that he wrote to John von Neumann, dated 20 March 1956, he states the P vs. NP problem quite ahead of its time. Although we do not know for sure, it is unlikely that von Neumann read the letter, or at least that he gave it much thought. At the time, unfortunately, he was lying in a hospital bed fighting the disease that would soon lead him to his death. In the letter, with a few quick, clean strokes Gödel goes back to Hilbert's Entscheidungsproblem arguing that, in a sense, his approach was simplistic. Suppose, Gödel says, that we find the proof of an interesting mathematical statement, but this proof is enormously long, say, it consists of $2^{1,000}$ symbols. This number, even when expressed in picoseconds, is well beyond the age of the universe. It is clear that such a proof would be of very little use: its length must be compatible with human life! The interesting question to ask therefore, is not so much whether a particular mathematical statement admits a proof but if a proof exists that can be grasped by mere mortals. To capture this more realistic situation in mathematical terms, consider the following decision problem, a sort of Entscheidungsproblem restricted to mathematical proofs that can be grasped by human beings[11]:

[11] We are sweeping a few technical issues under the carpet for the sake of simplicity. For a rigorous treatment see [13].

Existence of a Readable Proof (ERP)
Input: a mathematical statement E and a number N.
Question: is there a proof of E of length at most N?

The question asked by Gödel in his letter is whether ERP admits an efficient algorithm: is it possible to find a readable proof, or conclude that none exists, in time polynomial in N, for example in time proportional to N or N^2? In this way Gödel identifies the true nature of the problem investigated by Hilbert and, embryonically if not naively, by Leibniz: can the creative process that leads to mathematical proofs be replaced by the work of a computer, i.e., a machine? To see why, let us compare the two approaches to this question, as proposed by Hilbert and by Gödel. Recall that Hilbert starts from the following computational problem (notice the difference with ERP):

Existence of a Proof (ED)
Input: a mathematical statement E.
Question: is there a proof of E?

Let us now consider the following three questions, concerning these computational problems:

1. (Hilbert's Entscheidungsproblem) Is there an algorithm for ED?
2. Is there is an algorithm for ERP?
3. (Gödel's letter) Is there an efficient algorithm for ERP?

Gödel, perhaps because he was living at a time when electronic computers had become a reality, frames the problem correctly in a computational light. As he notes in his letter, an affirmative answer to the third question *"would have consequences of the utmost importance [as] it would imply that, in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician with regard to yes/no questions could be replaced entirely by the work of a machine."*[12]

To clarify this point, suppose that there is an algorithm of complexity N^2 for the problem ERP and that we are interested in a particular statement E, e.g., the Riemann hypothesis. To find out if there is a proof of the Riemann hypothesis using ten million symbols, or about 350 pages, with current technology we would have to wait for a day or so. Possibly, something so deep and complex like the Riemann hypothesis might need more than 350 pages. No problem! By setting N equal to 100 million, corresponding to a proof of nearly 35,000 pages, you

[12] A translation of the letter can be found at http://blog.computationalcomplexity.org/2006/04/kurt-gdel-1906-1978.html

would have the answer within 4 years. It may seem a lot, but what about Fermat's Last Theorem, whose proof required nearly four centuries? To have the answer of what is considered the most important open problem in mathematics we might as well wait a few years! Moreover, technological improvements, providing ever more powerful hardware, would soon make manageable values of N over a billion, which is equivalent to considering arguments millions of pages long, a limit beyond which it certainly makes no sense to look for proofs.

We now note a trivial but surprising fact. The (negative) answer to question 1 required a flash of genius, in its most elegant form due to Turing. In contrast, the answer to question 2 is affirmative and rather trivial. Let us see why. Every mathematical theory can be described by means of standard mathematical formalism in which all the formulas, sentences, definitions, theorems and so on, can be written down by means of a finite set of symbols, the so-called alphabet (this should not be surprising, after all mathematics is written in books). Specifically, any mathematical theory uses the ten digits $0, 1, \ldots, 9$, uppercase and lowercase letters of the alphabet, sometimes Greek letters, special symbols like $\forall$, $\exists$, etc., parentheses (,), [,], and so on. Suppose, for the sake of simplicity but without affecting the generality of the argument, that our alphabet has 100 symbols. To determine whether a certain statement admits a proof with N symbols it is sufficient to generate all candidate proofs, i.e., all possible 100^N strings of symbols, and check them one by one. The checking part, to determine whether a given sequence of N symbols is a correct proof of a statement E, is doable by computer efficiently, in polynomial time.

The answer to question 2 therefore is affirmative: there is a simple, indeed trivial general procedure that allows a computer to tell, given any statement of mathematics whatsoever, whether there exists a proof understandable by human beings. For the proponents of human superiority over soulless machines this might sound like a death knell, but there is no need to worry. The by-now-familiar catch comes to our rescue: the algorithm that we outlined, having exponential complexity, is useless. To settle the question of human superiority once and for all we have to follow once again our lodestar – computational efficiency – and consider the third question, which asks whether machines can find understandable proofs efficiently. It should not come as a surprise by now to learn that ERP is universal (NP-complete). In other words, the problem posed by Gödel in his letter, question 3 above, which so effectively captures an old and formidable question – whether the creativity of a mathematician can be effectively replaced by a machine – is none other than the one million dollars problem:

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

If $\mathrm{P} = \mathrm{NP}$, then, given any mathematical statement, a proof or a refutation can be found efficiently by an algorithm and, therefore, by a computer. It could be argued that in this case the discoverer of this exceptional algorithm could claim not one, but six million dollars, as it could answer all Millennium Problems remained unsolved

until now![13] This seems very unlikely and is part of the circumstantial evidence that leads us to conjecture that the two classes must be different. This triumph of human creativity over the machines, however, would come at a heavy price, for we would have to resign ourselves to live in a world where, despite the impressive progress of technology, computational intractability remains the norm, even for trivial-looking tasks.

The Class P and Efficient Computations

A common, and quite sensible, objection that is raised when the class P is encountered for the first time is the following: if the running time of an algorithm is, say, proportional to $N^{1,000,000,000,000,000}$, we do have polynomial-time but this is as impractical as a running time like 2^N, and therefore P cannot be considered as a good characterization of the realm of feasible computations. To be sure, the intuitive concept of "feasible computation" is a very elusive one and P is only a raw characterization of it. It does capture however some of its fundamental aspects. A common occurrence is that, once a polynomial-time algorithm for a problem is found, however impractical, successive improvements bring about new algorithms of better and better performance. In practice, most polynomial-time algorithms have very low exponents. One could also argue that P embodies a conservative approach: if a problem does not have a polynomial-time algorithm, it must certainly be computationally unapproachable. The main point however is different and has to do with the combinatorial structure of algorithmic problems. As we have discussed, if we could afford exponential running times like 2^N, NP-complete problems like Satisfiability would admit trivial, brute-force algorithms: it would be sufficient to enumerate all possible candidate solutions (i.e., all possible truth assignments) and check them one by one, to see if any satisfies the formula. Suppose now that we could afford a running time of $N^{1,000,000,000,000,000}$. Even such a huge polynomial cannot match the exponential growth of 2^N and if we want an algorithm for SAT to cope with all possible inputs we cannot rely on a brute-force enumerative approach any more. In order to zero in on a satisfying assignment, or to conclude that none exists, a good deal of mathematical ingenuity would be needed to identify and exploit hidden organizational principles of the solution space (in the case of SAT, all possible truth assignments), provided that they exist. In fact, it is quite conceivable that no such regularities exist for NP-complete problems.

[13] Since the awards were established, the Poincaré Conjecture, one of the most coveted prey, has been answered affirmatively by Grigori Perelman. He refused, however, to collect both the prize and the Fields Medal!

3.8 The Driving Force of Scientific Discovery

"As long as a branch of science offers an abundance of problems, so long is it alive; a lack of problems foreshadows extinction" warned David Hilbert in his famous speech. Computer science certainly enjoys an extraordinary vitality in this respect. The pace of its fantastic technological achievements – a revolution that continues unabated in full swing – and the momentous social consequences that they engender, are perhaps the reason why the conceptual depth at the basis of this revolution is underappreciated. And yet, as we have seen, some of the scientific questions posed by computer science, of which P vs. NP is just one example, are of the greatest import, with ramifications deep and wide not only for mathematics but for many other disciplines. Nor could it be otherwise. Computer Science tries to understand the laws that govern two concepts that lie at the foundations of our understanding of the world: information and computation. These laws, which at present we are just beginning to glimpse, are as concrete as those of nature and, like those, define the boundaries of our universe and our ability to maneuver within it. The one million dollars problem is just the tip of a glittering iceberg.

3.9 Bibliographic Notes

The Millennium Problems and their relevance are discussed on the website of the Clay Mathematics Institute [79] (see Millennium Problems).

The theory of NP-completeness is now part of the standard university undergraduate curriculum in computer science around the world. Among the many excellent algorithms books now available I can certainly recommend "Algorithm Design" by Jon Kleinberg and Éva Tardos [67]. For a discussion from the viewpoint of computational complexity theory, a nice and agile book is "Computational Complexity" by Christos Papadimitriou [89]. The classic text, however, remains "Computers and Intractability: a Guide to the Theory of NP-Completeness" by Michael R. Garey and David S. Johnson [47]. This book played an important role in disseminating the theory, and it remains one of its best accounts.

An interesting article which describes the state of complexity theory in the Soviet Union is [106]. Kurt Gödel's letter to von Neumann is well analyzed in [13]. Both articles are available on the Web.

Turning to the popular science literature, a very nice account that explores the role of mathematical logic for the development of computers, from Leibniz to Turing, is "Engines of Logic: Mathematicians and the Origin of the Computer" by Martin Davis [24].

Biographies can offer interesting insights by placing ideas and personalities in their historical and cultural context, as well as reminding us that science, like all human activities, is done by people of flesh and bone, with their dreams, their pettiness and greatness, happiness and suffering. An excellent biography is "Alan

Turing: the Enigma" by Andrew Hodges [60], which tells the story of the tragic life of this great scientist, his role in decoding the secret code of the German U-boats during the Second World War, until the brutal persecution and breach of his fundamental human rights that he endured at the hands of the British authorities.

Finally, a wonderful booklet is "Gödel's Proof" by Nagel and Newman [83], a layman's masterly exposition of Gödel's magnificent Incompleteness Theorems.