Space-Time Assumptions Behind NP-Hardness of Propositional Satisfiability

Olga Kosheleva and Vladik Kreinovich
University of Texas at El Paso
500 W. University, El Paso, TX 79968, USA
olgak@utep.edu, vladik@utep.edu

Abstract

For some problems, we know feasible algorithms for solving them. Other computational problems (such as propositional satisfiability) are known to be NP-hard, which means that, unless P=NP (which most computer scientists believe to be impossible), no feasible algorithm is possible for solving all possible instances of the corresponding problem. Most usual proofs of NP-hardness, however, use Turing machine – a very simplified version of a computer – as a computation model. While Turing machine has been convincingly shown to be adequate to describe what can be computed *in principle*, it is much less intuitive that these oversimplified machine are adequate for describing what can be computed *effectively*; while the corresponding adequacy results are known, they are not easy to prove and are, thus, not usually included in the textbooks. To make the NP-hardness result more intuitive and more convincing, we provide a new proof in which, instead of a Turing machine, we use a generic computational device. This proof explicitly shows the assumptions about space-time physics that underlie NP-hardness: that all velocities are bounded by the speed of light, and that the volume of a sphere grows no more than polynomially with radius. If one of these assumptions is violated, the proof no longer applies; moreover, in such space-times we can potentially solve the satisfiability problem in polynomial time.

1 Formulation of the Problem

General problem. Which problems can be solved in feasible time and which cannot? To answer this question, it is necessary to formally describe which algorithms are feasible, what is a problem, and how can we know that a problem cannot be solved by a feasible algorithm. Let us recall how this is done in theory of computation; for details, see, e.g., [2, 4, 7].

Feasibility: a brief reminder. Many algorithms are feasible; for example, most algorithms whose computation time is bounded by a square or a cube of

the bit size n of the input are usually feasible.

However, some algorithms require, even for inputs of reasonable length, computation time which exceeds the lifetime of the Universe. For example, for problems for which we know that the bit size of the solution y does not exceed the bit size $\mathrm{len}(x)$ of the input x, we can find a solution by using exhaustive search, i.e., by trying all possible words y of size $\mathrm{len}(y) \le n \stackrel{\mathrm{def}}{=} \mathrm{len}(x)$. However, even for words in a binary 0-1 alphabet, this would require, in the worst case, trying $1 + 2 + \ldots + 2^n = 2^{n+1} - 1$ possible words.

Even for reasonable-size inputs, of size $n \approx 1000$, this would require $2^{1000} \approx 10^{300}$ computation steps. Even if each of $\approx 10^{90}$ elementary particles which form the Universe serves as one of the parallel processors, each of these processors would still need to perform 10^{200} computation steps: and even if we divide the lifetime of the Universe to the smallest possible time quantum (the time during which light passes through an elementary particle), we would still get no more than $\approx 10^{40}$ computation steps. Thus, such exponential-time algorithms are usually considered to be infeasible.

This observation prompts the usual definition of feasibility. For each algorithm A, let $t_A(x)$ denote the number of computation steps on input x. The worst-case number of computation steps $t^w_A(n) \stackrel{\mathrm{def}}{=} \max\{t_A(x) : \mathrm{len}(x) = n\}$ on all inputs x of size (length) n is known as the (worst-case) *computational complexity* of the algorithm A. In these terms, an algorithm is called *feasible* if and only if it is *polynomial-time*, i.e., if there exists a polynomial $P(n)$ for which $t^w_A(n) \le P(n)$ for all n.

This definition is not perfect:

- an algorithm with computational complexity $t^w_A(n) = 10^{1000} \cdot n$ is polynomial-time, but clearly not feasible;
- on the other hand, an algorithm with computational complexity $t^w_A(n) = \exp(10^{-9} \cdot n)$ is practically feasible for all inputs of size $\le 10^9$, but is is not polynomial time.

However, the above definition is the best we have :-(

What is a problem. In a precisely formulated problem, it may be difficult to solve a problem, but it should be feasible to check whether a proposed candidate for a solution is indeed a solution.

For example, in mathematics, the main problem is: given a statement x, produce a detailed proof y of either the statement x or of its negation. Coming up with a proof is often very difficult, but once a detailed step-by-step proof is produced, it is easy to check step-by-step whether each step is correct – even a computer can do it provided that the proof is detailed enough. In this case, the problem is: given x, find y such that $C(x, y)$ holds, where $C(x, y)$ is a feasibly computable predicate describing that y is a proof of x or of $\neg x$.

Of course, to be able to check the proof in reasonable time, we must also require that length of this proof is feasible. Similarly to feasible time, it is

reasonable to formalize this requirement by requesting that there exist a polynomial $P_\ell(n)$ such that $\text{len}(y) \le P_\ell(\text{len}(x))$. Thus, a problem takes the following form: given a word x, find a word y such that $C(x, y)$ and $\text{len}(y) \le P_\ell(\text{len}(x))$ – or produce a message that such a proof y is not possible.

Similarly, in physics, the main problem is: given the observation data x, find a law y that fits all this data. Once a formula y is found, it is easy to check, observation-by-observation, that all the observations x satisfy this formula; however, coming up with an appropriate formula is often very difficult. In this example, the limitation on the size of y is even more severe: namely, the length of y must not exceed the length of x – if we do not make this requirement, then we can simply take the listing of all the observations as the desired formula. In this case, $\text{len}(y) \le \text{len}(x)$, i.e., $\text{len}(y) \le P_\ell(\text{len}(x))$ for $P_\ell(n) = n$.

In engineering, we are given specifications x, e.g., about a bridge, and we need to find a design y which satisfies all the specifications. Modern software enables us to feasibly check whether a given design satisfies the desired specifications, but finding such a design is often difficult. The design must be feasible to implement, which means that we must have $\text{len}(y) \le P_\ell(\text{len}(x))$ for some polynomial $P_\ell(n)$.

In all these cases, we have a feasible algorithm $C(x, y)$ and a polynomial $P_\ell(n)$, and our task is: given a word x, find a word y for which $C(x, y)$ and $\text{len}(y) \le P_\ell(\text{len}(x))$ – or produce a message that such y is not possible. This will be our general definition of a problem.

In this definition, once we have a guess y, it is feasible (i.e., requires polynomial time) to check whether this guess is a correct solution. In theoretical computer science, computations with guesses are called *non-deterministic*. Because of this, such problems are called *non-deterministic polynomial-time*, or NP, for short.

All problems from the class NP are algorithmically solvable: e.g., by exhaustive search. For each input x, the length of possible solution y is bounded. Thus, we can, in principle, find the solution y by applying *exhaustive search*, i.e., by testing all possible words y of length $\text{len}(y) \le P_\ell(\text{len}(x))$.

Exhaustive search is not feasible. The problem with the exhaustive search algorithm is that the corresponding computation time is proportional to the number of possible words of a given length, and this number grows exponentially with the length of the input, as $S_A^{P_\ell(\text{len}(x))}$, where S_A is the number of possible symbols. We already know that such exponential-time algorithms are not practically feasible.

Are feasible algorithms possible? Is P equal to NP? For some problems from the class NP, there exists a feasible (polynomial-time) algorithm for solving the corresponding problem. The class of such feasibly solvable problems is denoted by P.

It is not known whether all the problems from the class NP can be thus solved, i.e., whether P=NP. This is a long-standing open problem. Most computer scientists believe that P≠NP.

The notion of NP-hardness. While it is not known whether P is equal to NP, it is known that some problems from the class NP are the hardest. This "hardness" is described by the notion of *reduction*: if a problem A can be reduced to problem A', this means that the problem A' is at least as hard as the problem A.

The notion of reduction can illustrated on the following simple example. A usual way to solve an equation of the type $a\cdot x^4+b\cdot x^2+c=0$ is to reduce it to the problem A' of solving the quadratic equation. For this reduction, we introduce a new variable $y=x^2$; in terms of this new variable, the original equation takes the form $a\cdot y^2+b\cdot y+c=0$. We know how to solve the corresponding quadratic equation; once we find its solution, we can find x as $\pm\sqrt{y}$. Thus, to solve a particular case of the original problem A, we:

- form the corresponding particular case of the problem A'; we will denote the corresponding algorithm by U_1;
- solve this new particular case;
- use the solution to compute the solution to the original problem; we will denote the corresponding algorithm by U_3.

Both algorithms U_1 and U_3 can be multiple-valued.

It is also important to make sure that in this manner, we can find *all* solutions to the original problem, i.e., that for every solution of the original problem, there is a solution to the problem A' from which this solution can be obtained (in our case, $y=x^2$); we will denote the corresponding algorithm by U_2.

In general, when we have two problems from the class NP, a problem A described by a feasible property $C(x,y)$ and a problem A' described by a feasible property $C'(x',y')$, then we say that A is *reducible* to A' if these exists three feasible algorithms U_1, U_2, and U_3 with the following properties:

- if $C'(U_1(x),y')$, then $C(x,U_3(y'))$;
- if $C(x,y)$, then $C'(U_1(x),U_2(y))$ and $U_3(U_2(y))=y$ (in the multiple-valued case, $y\in U_3(U_2(y))$).

The first property means that if we start with an instance x of the problem A, build the corresponding instance $x'=U_1(x)$ of the problem A', and a solution y' to this new instance, then, by applying the algorithm U_3 to this solution y', we get a solution $y=U_3(y')$ to the original problem.

The second property means that if y is any solution to the original problem, then it can be obtained by applying the above procedure, when we use an appropriate solution $y'=U_2(y)$ to the corresponding instance $x'=U_1(x)$ of the problem A'.

We say that a problem P is *NP-hard* if it is as hard or harder than every problem from the class NP, i.e., in precise terms, if every problem from the class NP can be reduced to the problem P.

Propositional satisfiability: historically first example of an NP-hard problem. The first problem for which NP-hardness was proven was the problem of *propositional satisfiability.* In this problem, the input x is a *propositional formula*, i.e., a formula which can be obtained by Boolean ("true"-"false") variables $z_1, \ldots, z_v$ by using propositional operations "or" ($\vee$), "and" (&), and "not" ($\neg$). An example of a propositional formula is $(z_1 \vee \neg z_2 \vee \neg z_3) \,\&\, (\neg z_1 \vee z_3)$. The objective is to find a tuple $y = (z_1, \ldots, z_v)$ of Boolean values for which the given formula is true.

One can easily see that this is a problem from the class NP: if we have a formula x and a Boolean tuple y, then checking whether x is true for these values of z_i takes linear (thus polynomial) time – hence the corresponding property $C(x, y)$ is feasible. The length of the tuple does not exceed the length of the original formula, so here $\text{len}(y) \le P_\ell(\text{len}(x))$ for a simple polynomial $P_\ell(n) = n$.

NP-hardness has actually been proven for a special class of propositional formulas in *Conjunctive Normal Form* (CNF), i.e., formulas of the type $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$, where each *clause* has the form $a \vee \ldots \vee b$, and $a, \ldots,$ b, are *literals*, i.e., variables z_i or their negations $\neg z_i$.

Most textbook proofs of satisfiability's NP-hardness are based on Turing machines. NP-hardness of satisfiability means that we can reduce every problem from the class NP to the satisfiability (and even to CNF-SAT). This is how NP-hardness of satisfiability is usually proven: by taking a general problem from the class NP and showing that this problem can be reduced to CNF-SAT.

These proofs are usually reasonably simple and straightforward, so at first glance, the proofs seem to be intuitively clear. However, a more detailed look shows that these proofs are not as intuitive as they may seem.

Indeed, by definition, a problem from the class NP means that we have a feasible (polynomial-time) algorithm $C(x, y)$, and the problem is: given x, find y for which the property $C(x, y)$ is satisfied. In the existing proofs, *polynomial-time* is understood as polynomial-time on a Turing machine.

Again, at first glance, this may seem reasonable. A Turing machine is what we would now call a simplified computer. A Turing machine consists of a tape (which is potentially infinite) which consists of cells. Each cell can be either empty or contain a symbol from the given list (e.g., 0 or 1). There is also a *head* which, at any given moment of time, is located near one of the cells. The head can be in one of the states from a given list. It starts at a special *start* state, with the input x written on a tape.

At each moment of time, depending on the current state h of the head and on the symbol s in the corresponding cell, the machine can do three things:

- overwrite the symbol s with a new symbol $s' = f(h, s)$ depending on h and s;
- change its state h to a new state $h' = g(h, s)$ depending on h and s; and
- depending on h and s, either stay at the same cell, or move one step to the left, or move one step to the right.

The machine stops when it reaches a special *halt* state. Once the Turing machine stops, what is written on the tape is considered to be the result of the computations. In other words, we say that a Turing Machine computes a function $y = F(x)$ if, every time we start it with the input x, it eventually halts and produces $y = F(x)$.

Why Turing machines are used in theory of computation. While Turing machine is a very primitive device, more like an old-fashioned tape recorder than a computer, it is known to be a universal computational device – in the sense that whatever complex computer can compute, a Turing machine can compute as well. This explains why Turing machines are used in theory of computation: they are much simpler than actual computers and, at the same time, they describe the exact same class of computable functions as more complex computers.

Because of this, if we want to prove that a function is not computable, there is no need to consider more complex devices: it is sufficient to prove that this function cannot be computed on a Turing machine.

Why the use of Turing machines in NP-hardness proofs is not fully satisfactory. As we have mentioned, Turing machines are perfect in describing what can be, in principle, computed. Of course, from the practical viewpoint, it makes no sense to build and use Turing machines: they are often very slow in comparison with the actual computers.

For example, if we are looking for an element e in a sorted array $a_1 \leq \ldots \leq a_n$, then on a real computer, we can use bisection and find the location i of the element e (i.e., the index for which $a_i = e$) in logarithmic time $t \leq \log_2(n)$. In the beginning, we know that i is in the interval $[\underline{i}, \overline{i}]$, with $\underline{i} = 1$ and $\overline{i} = n$. On each iteration, once we know such an interval, we compute the midpoint $m = \lfloor (\underline{i} + \overline{i})/2 \rfloor$ and compare e with a_m.

- if $e = a_m$, the problem is solved, we found the index, it is m;
- if $e < a_m$, this means that $i < m$, so we replace the original interval $[\underline{i}, \overline{i}]$ with the half-size interval $[\underline{i}, m - 1]$;
- if $e > a_m$, this means that $i > m$, so we replace the original interval $[\underline{i}, \overline{i}]$ with the half-size interval $[m + 1, \overline{i}]$.

In both cases, we get a interval which is at least twice narrower than the original one. After k iteration, the interval's width is decreased by a factor of 2^k. So,

after $k = \log_n(n)$ iterations, the original width $n - 1$ is decreased at least by a factor of $2^k = n$. The resulting interval of width ≤ 1 cannot contain two different integers and thus, consists of a single integer i.

On a Turing machine, however, we start with the head located before a_1. When the desired value is cloated as $i = n$ (i.e., when $e = a_n$), the only way to find this location is to read the word a_n and to compare it with e. This means that the machine must move from a_1 all the way to a_n, thus passing by at least n cells. But a Turing machine can move at most one cell at a time. Thus, on a Turing machine, search requires at least n computational steps – and for large n, the amount n is much larger than $\log_2(n)$:

- for $n = 10^3$, we have $\log_2(n) \approx 10$;
- for $n = 10^6$, we have $\log_2(n) \approx 20$;
- for $n = 10^9$, we have $\log_2(n) \approx 30$;
- etc.

So, when we require that $C(x, y)$ is computable in polynomial time on such a super-slow device as a Turing machine, we are unecessarily limiting ourselves, since what we really want is properties $C(x, y)$ which can be computed in feasible time on a real computer.

Mathematically, it is OK to use Turing machines, but intuitively, it is desirable to consider more realistic computational devices. From the purely mathematical viewpoint, the situation is not as bad as it may seem: it turns out that, while Turing machines are indeed slower, they preserve computability in polynomial time. Many results show that if a function can be computed in polynomial time on a more complex computational device, then it can be also computed in polynomial time on a Turing machine.

With these additional results in mind, we can conclude that even if we understand feasible time as polynomial time on a realistic complex computer, every problem from the corresponding class NP can still be reduced to CNF-SAT. However, these additional results – that polynomial time on a computer translated into polynomial time on a Turing machine – results without which we do not get the desired reduction, are much more complex and less intuitive than the textbook proofs of SAT's NP-hardness. These additional results are therefore not included in the usual textbook analysis of NP-hardness – and so, the easiness of the usual proof kind of hides the fact that the actual proof of the desired result is much less intutive than it seems at first glance.

What we do in this paper. To make the NP-hardness proof more convincing, we provide a new proof, a proof in which instead of a Turing machine we use a generic computational device.

This proof makes it clear what assumptions about space-time are needed in this derivation. We also show that these assumptions are necessary: if one

of these assumptions is violated, then we can potentially solve satisfiability problems in polynomial time.

Comment. The main results of this paper were first announced in [3].

2 A New Proof that Satisfiability Is NP-Hard – Which Makes Space-Time Assumptions Behind This Result Explicit

What we start with. We have a problem from a class NP, we want to show how to reduce this problem of CNF-SAT. By definition, a problem from the class NP can be formulated as follows:

- we have a feasible predicate $C(x, y)$ (i.e., a feasible algorithm that always returns "true" or "false"),
- we have a polynomial $P_\ell(n)$, and
- we have a word x.

The problem is to find a word y for which $C(x, y) =$"true" and whose length $\text{len}(y)$ is bounded by the polynomial of the length $\text{len}(x)$ of the input word x, i.e., $\text{len}(y) \le P_\ell(\text{len}(x))$.

The algorithm $C(x, y)$ checks, in polynomial time, whether a given "guess" y is indeed a solution to the problem with the given x.

By definition, the algorithm $C(x, y)$ is feasible, i.e., on some computational device, its running time $t_C(x, y)$ is bounded by a polynomial of the length of its input: $t_C(x, y) \le P_C(\text{len}(x) + \text{len}(y))$ for an appopriate polynomial $P_C(n)$.

Computational device: component cells and their states. Let us analyze a computational device on which this algorithm $C(x, y)$ runs. A typical computational device consists of discrete *cells*. For example, each memory bit can be viewed as an elementary cell, a piece of wire that connects several elements on a chip can be viewed as a cell, etc.

Cells can be of different volume. Let us denote the smallest volume of a cell by ΔV.

Each cell can be in different *states*. For example, a memory bit can be in two states: 0 and 1. A wire can be in three states: not sending any signal, sending 0, and sending 1; etc. In principle, a physical object can be in infinitely many different states, but since all measurements are not accurate, we can only distinguish between finitely many states.

Different cells can have different number of possible states. Let us denote the largest number of possible states by S.

We will assume that the *time quantum* – i.e., the minimal time during which the computer performs a step – for this computational device is equal to Δt;

this means that we can only consider the state of the computer at times 0, Δt, $2\Delta t$, etc.

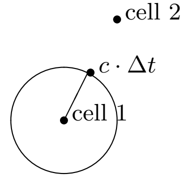
Comment. For simplicity, in the following text, these moments of time will be denoted by consequent integers: moment 0, moment 1, etc., so the next moment of time to moment t will be also denoted by $t+1$.

First physical assumption: $v \le c$. We will take into consideration the fact that, according to modern physics, the speed of every process is limited by the speed of light c.

Dynamics of states. Let us use the above physical assumption to describe how a state of each cell changes with time.

Since the speed of communication is bounded by the speed of light, the state of the cell in the next moment of time can only be influenced by the states of the cells that are at a distance $\le r = c \cdot \Delta t$ from this desired cell: indeed, if a cell is further away, then during the time quantum, its influence will not be able to reach the original cell.

So, the state of the cell at the next moment of time $t+1$ is determined only by the states of the cells inside the sphere of radius r, which is the "sphere of influence" of a given cell. We will call cells that can influence a given cell its *neighbors*.



Let us estimate the number $N_{\rm neigh}$ of neighbors.

By definition, ΔV is the smallest volume of a cell. This means that each cell occupies the volume that is greater than or equal to ΔV. Thus, $N_{\rm neigh}$ cells occupy the volume $\ge N_{\rm neigh} \cdot \Delta V$. On the other hand, all these cells are located inside the sphere of radius r. The total volume inside the sphere is $\frac{4}{3} \cdot \pi \cdot r^3$; therefore, $N_{\rm neigh} \cdot \Delta V \le \frac{4}{3} \cdot \pi \cdot r^3$, and hence,

$$N_{neigh} \le \frac{\frac{4}{3} \cdot \pi \cdot r^3}{\Delta V}.$$

Let us define the state of a cell i at moment t by $S_{i,t}$. Then, we can describe the evolution of the states as follows:

$$S_{i,t+1} = f_{i,t}(S_{i,t}, S_{j,t}, \ldots, S_{k,t}), \quad (1)$$

where the number of neighboring cells $(S_{j,t}, \ldots, S_{k,t})$ is $\le N_{\rm neigh}$.

Towards reduction to propositional satisfiability: making all variables Boolean. We want to reduce our problem to propositional satisfiability. In propositional satisfiability, all the variables are Boolean. To get closer to this problem, let us represent each state by a sequence of Boolean (0-1) values.

To do that, we will enumerate all the states of each cell, describe each state by its ordinal number, and represent this ordinal number in the same manner as this number is represented in the computer, i.e., by a sequence of its binary digits.

Since the largest possible number of states of a cell is S, we can represent these states by integers from 0 to $S-1$. Let us denote by B the total number of binary digits in the binary representation of $S-1$. Then, all numbers smaller than $S-1$ require the same or smaller number of digits. Hence, we need B bits to describe each state.

By using k bits, we can describe 2^k different numbers; thus, to represent S different states by B bits, we must have $2^B \geq S$, i.e., $B \geq \log_2(S)$. Therefore, we can take, as B, the smallest integer for which this inequality is true, i.e., $B = \lceil \log_2(S) \rceil$.

Thus, each state $S_{i,t}$ can be represented as a sequence of B bits $s_{i,1,t}, s_{i,2,t}, \ldots, s_{i,b,t}, \ldots, s_{i,B,t}$. Here, the bit number b takes values $b = 1, \ldots, B$. From the equation (1), we can now conclude that the value of each of these variables at time $t+1$ depends on the values of the variables that describe neighboring cells at the time t:

$$s_{i,b,t+1} = f_{i,b,t}(s_{i,1,t}, \ldots, s_{i,B,t}, \ldots, s_{j,1,t}, \ldots, s_{j,B,t}, \ldots, s_{k,1,t}, \ldots, s_{k,B,t}). \quad (2)$$

The total number of variables in the right-hand side is bounded by $\leq N_{\text{neigh}} \cdot B$.

Transforming the conditions into propositional form. All the variables in the expression (2) are Boolean, but the relation between these variables is not yet Boolean. To make it Boolean, let us express each formula (2) in Conjunctive Normal Form (CNF).

This can be done if we first translate a general formula F into a Disjunctive Normal Form, i.e., form of the type $D_1 \vee \ldots \vee D_m$, where each disjunction D_j is of the type $a \,\&\, \ldots \,\&\, b$, with literals $a, \ldots, b$. For that, we form a truth table for the formula F, i.e., describe its value (true or false) for all 2^k possible combinations of truth values of its k variables. A formula is true if and only if the inputs coincide with one of the tuples for which F is true. For example, if the formula F is true when x_1 and x_2 are both true and when x_1 and x_2 are both false, then F is equivalent to $(x_1 \,\&\, x_2) \vee (\neg x_1 \,\&\, \neg x_2)$.

To translate a formula F into CNF, we transform $\neg F$ into DNF, and then apply de Morgan rules $\neg(A \vee B) \equiv \neg A \,\&\, \neg B$, $\neg(A \,\&\, B) \equiv \neg A \vee \neg B$, and $\neg(\neg A) \equiv A$ to transform the negation of the DNF into a CNF. For example, if $\neg F \equiv (x_1 \,\&\, x_2) \vee (\neg x_1 \,\&\, \neg x_2)$, then

$$F \equiv \neg((x_1 \,\&\, x_2) \vee (\neg x_1 \,\&\, \neg x_2)) \equiv \neg(x_1 \,\&\, x_2) \,\&\, \neg(\neg x_1 \,\&\, \neg x_2) \equiv$$

$$(\neg x_1 \vee \neg x_2) \,\&\, (\neg(\neg x_1) \vee \neg(\neg x_2)) \equiv (\neg x_1 \vee \neg x_2) \,\&\, (x_1 \vee x_2)).$$

This translation requires 2^k computational steps, where k is the number of variables. In our case, k is bounded by a constant $\le N_{\rm neigh} \cdot B$ which does not depend on the size of the input. Thus, 2^k is also bounded by a constant: $2^k \le 2^{N_{\rm neigh} \cdot B}$.

The translation gives us a propositional formula $F_{i,b,t}$ which describes the evolution of the b-th bit $s_{i,b,t+1}$ in the description of the i-th state.

Combining these formulas by "and", we can now describe the entire computation of $C(x,y)$ by a single formula. Indeed, given algorithm $C(x,y)$ and input x, it is necessary to describe that:

- The device operates correctly, i.e., all the states are changed accordingly. This is described by the following long formula:

$$F_{1,1,1} \,\&\, F_{1,1,2} \,\&\, \ldots \,\&\, F_{i,b,t} \,\&\, \ldots \,\&\, F_{N_{\rm cells},B,T},$$

  where $1 \le i \le N_{\rm cells}$, $1 \le b \le B$, $1 \le t \le T$, and T is the computation time (= total number of computational steps) in computing $C(x,y)$.

- We also need to describe that the input is the given one $x = x_1 x_2 \ldots$:

$$s_{i_1,b_1,1} = x_1 \,\&\, s_{i_2,b_2,1} = x_2 \,\&\, \ldots,$$

  where i_k is the cell that contains the k-th bit of the input x.

- Finally, we need to describe that the result of the computation is "true" in the "final" cell i_r: $s_{i_r,b_r,T} =$"true".

So, we use "and" to combine these formulas into a "long formula" F.

This is indeed a reduction to satisfiability. We have designed the algorithm U_1 that transforms each instance x of the original NP-problem into a propositional formula $x' = F$. This long formula describes the fact that:

- we started with given input x and some y,
- we performed the computation of the property $C(x,y)$, and
- we got $C(x,y)$ to be true.

Once we have a satisfying tuple y' for this formula, we read y from the bits describing the inputs y at moment 1. This is our algorithm U_3.

If we know the solution y to the original problem, then we can run a feasible algorithm for checking $C(x,y)$ and record all the values of all the bits of all the states at all moments of time. This is our algorithm U_2. One can easily check that this is indeed the desired reduction:

- if the tuple y' makes the propositional formula $C'(U_1(x),y')$ true, this means that for the input x and for the $y = U_3(y')$ which corresponds to y', the value $C(x,y)$ is also true, i.e., that y is indeed a solution to the original problem;

- vice versa, if y is a solution to the original problem, then for the Boolean tuple $y' = U_2(y)$ which describes the process of computing $C(x,y)$, the long Boolean formula $F = x' = U_1(x)$ holds, i.e., we have $C'(x',y')$.

The reduction is feasible. To complete our proof, let us show that the designed algorithms U_i are indeed *feasible*, i.e., that their computation time is bounded by a polynomial of the input x.

This is clear for the algorithm U_3, in which we simply pick some bit values. Let us prove feasibility of the main reduction algorithm U_1. In this algorithm, we apply a constant number of computation steps to each of $N_{\rm cells}$ cells, to each of B bits, and to each of T moments of time. Thus, the computation time of this algorithm is proportional to the product $N_{\rm cells} \cdot B \cdot T$. The number of bits B is a constant that does not depend on the length of the input at all.

Since $C(x,y)$ is a feasible algorithm, its computation time T is bounded by the polynomial of the length of its input. Each polynomial can be bounded, from above, by a simple polynomial $A \cdot n^k$: indeed, for all natural numbers n, we get

$$a_0+a_1\cdot n+\ldots+a_k\cdot n^k \le |a_0|\cdot n^k+|a_1|\cdot n^k+\ldots+|a_k|\cdot n^k = (|a_0|+|a_1|+\ldots+|a_k|)\cdot n^k.$$

Thus, we can always conclude that $T \le A\cdot({\rm len}(x)+{\rm len}(y))^k$ for some A and k.

The length of y is limited by a polynomial ${\rm len}(y) \le P_\ell({\rm len}(x))$. We can similarly conclude that ${\rm len}(y) \le A' \cdot ({\rm len}(x))^{k'}$. Thus,

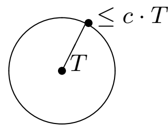
$$T \le A \cdot ({\rm len}(x) + A' \cdot ({\rm len}(x))^{k'})^k,$$

i.e., $T \le P({\rm len}(x))$, where we denoted $P(n) \stackrel{\rm def}{=} A \cdot (n + A' \cdot n^{k'})^k$. So, the computation time T is indeed bounded by a polynomial of the length of the original input x.

Let us estimate the total number of cells $N_{\rm cells}$ that participate in this computation.

In principle, many cells could be computing, but only those cells can influence the final result which are not too far away, because if the cell is at a distance $> c\cdot T$ from the final monitor, then, even if it is sending all its information with the largest possible speed – the speed of light – the final cell will still not be able to receive this information before the computations are over.

Thus, it is sufficient to consider only the cells that are located within a distance $\le c\cdot T$ from the final cell, i.e., within a sphere of radius $R = c\cdot T$:



The volume of this sphere V is $\dfrac{4}{3}\cdot\pi\cdot(c\cdot T)^3$. Therefore, the total number of cells $N_{\rm cells}$ in this sphere is bounded by the ratio $\dfrac{V}{\Delta V}$, i.e.,

$$N_{\rm cells} \le \frac{\frac{4}{3}\cdot\pi\cdot(c\cdot T)^3}{\Delta V} = \frac{4\pi\cdot c^3}{3\cdot\Delta V}\cdot T^3.$$

Since T is bounded by a polynomial $T \le P(n)$, we conclude that

$$N_{\text{cells}} \le \frac{4\pi \cdot c^3}{3 \cdot \Delta V} \cdot (P(\text{len}(x)))^3.$$

The cube of a polynomial is also a polynomial; thus, the number of cells is bounded by the polynomial of $\text{len}(x)$.

Hence, the time $t_{U_1}(x)$ needed to compute the formula F is bounded by the product of three polynomials, and hence, also by a polynomial: $t_{U_1}(x) \le P_1(\text{len}(x))$ for some polynomial $P_1(n)$.

Similarly, the algorithm U_2 finishes in polynomial time. The reduction is feasible, so NP-hardness is proven.

3 Example

Description of a toy problem. To make the above construction clearer, let us illustrate it on the example of the following toy problem. In this problem, the input x is one bit, the output y is one bit, and the condition $C(x, y)$ that we want to achieve is $x = y$.

In other words, in this toy problem, we are given a bit x, and we want to find a bit y which satisfies the property $x = y$.

Computational device for checking the desired property. In accordance with the above proof, we need to start with a computational device that, given x and y, checks whether $x = y$. In the beginning, we have two cells: an x-cell that contains the input bit x and a y-cell which contains the bit y.

We also need a wire to transmit the information. We will thus send the content of the y-cell to the x-cell, and then use the x-cell to compare its original content with what is send by wire. Once the y-signal is sent, we no longer need it, so we can simply erase it (i.e., replace it with 0).

The whole computation process takes 3 moments of time:

- at moment $t = 1$, the x-cell contains x, the y-cell contains y, and the wire is inactive;
- at moment $t = 2$, the x-cell still contains x, the y-cell now contains 0, and the wire transmits the y signal;
- at moment $t = 3$, the x-cell contains 1 if $x = y$ and 0 otherwise, the y-cell contains 0, and the wire is again inactive.

In this computations process, we have 3 cells: the x-cell, the y-cell, and the wire. The x-cell has 2 possible states: 0 and 1, so one bit is sufficient to describe its state. According to the general notation, we will denote the state of this bit at moment t by $s_{1,1,t}$. Similarly, to describe the state of the y-cell, we need one but $s_{2,1,t}$.

The wire can be in 3 possible states: inactive, sending 0, and sending 1. Thus, to describe the state of the wire, we will need 2 bits. Let the first bit describe whether the wire is active or not, and the second bit describe the signal sent via an active wire. So, the state S_3 of the wire is either 00 (inactive), or 10 (sending 0), or 11 (sending 1).

In this case, $S = 3$, and the number of bits B needed to describe the state of each of the cells is $B = 2$.

Corresponding dynamics of states. Let us describe the above computations in terms of changing states.

At the first moment of time, the wire is inactive: $s_{3,1,1} = s_{3,2,1} = 0$.

At the second moment of time, the first cell retains its state, i.e., $s_{1,1,2} = s_{1,1,1}$. The second cell becomes 0: $s_{2,1,2} = 0$. The wire becomes active: $s_{3,1,2} = 1$, and the signal it transmits is exactly the bit originally stored in the y-cell: $s_{3,2,2} = s_{2,1,1}$.

At the third moment of time, the x-cell gets the value 1 if the value that was previously stored in this cell coincides with what was sent thorugh the wire: $s_{1,1,3} = 1 \Leftrightarrow s_{1,1,2} = s_{3,2,2}$. The y-cell still contains 0: $s_{2,1,3} = 0$, and the wire is again inactive: $s_{3,1,3} = s_{3,2,3} = 0$.

Describing the dynamics in CNF terms. Many of the above formulas have the form $a = 0$, etc., for different variables a, etc. To describe theses formulas in the CNF terms, we need to translate the following general formulas into CNF: $a = 0$, $a = 1$, $a = b$, and $a = 1 \Leftrightarrow b = c$. Then, we will be able to simply substitute the appropriate variables – e.g., $s_{3,1,1}$ – instead of a etc. into the resulting CNF formula, and get the CNF form for the above formulas with variables s_{ijk}.

Let us use the above algorithm to translate the formulas $a = 0$, $a = 1$, $a = b$, and $a = 1 \Leftrightarrow b = c$ into CNF one by one.

Translating $a = 0$ into CNF. For the formula $a = 0$, the truth tables for formula F itself and for its negation $\neg F$ take the form

a	F	$\neg F$
0	1	0
1	0	1

The formula $\neg F$ is true only when $a = 1$, so its DNF form is a. Thus, its CNF form is $\neg a$. This means, e.g., that the formula $s_{3,1,1} = 0$ becomes $\neg s_{3,1,1}$.

Translating $a = 1$ into CNF. For the formula $a = 1$, the truth tables for formula F itself and for its negation $\neg F$ take the form

a	F	$\neg F$
0	0	1
1	1	0

The formula $\neg F$ is true only when $a = 0$, so its DNF form is $\neg a$. Thus, its CNF form is a. This means, e.g., that the formula $s_{3,1,2} = 1$ becomes $s_{3,1,2}$.

Translating $a = b$ into CNF. For the formula $a = b$, the truth tables for formula F itself and for its negation $\neg F$ take the form

a	b	F	$\neg F$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	1	0

The formula $\neg F$ is true either when $a = 0$ and $b = 1$, or when $a = 1$ and $b = 0$. So, its DNF form is $(\neg a \,\&\, b) \vee (a \,\&\neg b)$. According to de Morgan laws, to get a negation F, we need to change all conjunctions to disjunctions, all disjunctions to conjunctions, and each literal by its negation. Thus, the CNF form is $(a \vee \neg b) \,\&\, (\neg a \vee b)$. This means, e.g., that the formula $s_{1,1,2} = s_{1,1,1}$ becomes $(s_{1,1,2} \vee \neg s_{1,1,1}) \,\&\, (\neg s_{1,1,2} \vee s_{1,1,1})$.

Translating $a = 1 \Leftrightarrow b = c$ into CNF. Finally, for the formula $a = 1 \Leftrightarrow b = c$, the truth tables for formula F itself and for its negation $\neg F$ take the form

a	b	c	F	$\neg F$
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

The corresponding DNF form for $\neg F$ is

$$(\neg a \,\&\, \neg b \,\&\, \neg c) \vee (\neg a \,\&\, b \,\&\, c) \vee (a \,\&\, \neg b \,\&\, c) \vee (a \,\&\, b \,\&\, \neg c),$$

so its negation F takes the CNF form

$$(a \vee b \vee c) \,\&\, (a \vee \neg b \vee \neg c) \,\&\, (\neg a \vee b \vee \neg c) \,\&\, (\neg a \vee \neg b \vee c).$$

This means that the formula $s_{1,1,3} = 1 \Leftrightarrow s_{1,1,2} = s_{3,2,2}$ takes the form

$$(s_{1,1,3} \vee s_{1,1,2} \vee s_{3,2,2}) \,\&\, (s_{1,1,3} \vee \neg s_{1,1,2} \vee \neg s_{3,2,2}) \,\&$$
$$(\neg s_{1,1,3} \vee s_{1,1,2} \vee \neg s_{3,2,2}) \,\&\, (\neg s_{1,1,3} \vee \neg s_{1,1,2} \vee s_{3,2,2}).$$

The resulting long formula. The resulting formula should include:

- the CNF forms of all the formulas describing the state's dynamics,
- the fact that the initial value x is given; for example, for $x = 0$, it should be $s_{1,1,1} = 0$, i.e., $\neg s_{1,1,1}$; and
- the fact that the result of checking the property $C(x, y)$ is "true"; according to our computation scheme, this result is stored in the x-cell at moment 3, so this requirement takes the form $s_{1,1,3} = 1$, i.e., $s_{1,1,3}$.

Thus, the corresponding long formula takes the following form:

$$\neg s_{3,1,1} \mathbin{\&} \neg s_{3,2,1} \mathbin{\&}$$

$$(s_{1,1,2} \vee \neg s_{1,1,1}) \mathbin{\&} (\neg s_{1,1,2} \vee s_{1,1,1}) \mathbin{\&}$$

$$\neg s_{2,1,2} \mathbin{\&} s_{3,1,2} \mathbin{\&}$$

$$(s_{3,2,2} \vee \neg s_{2,1,1}) \mathbin{\&} (\neg s_{3,2,2} \vee s_{2,1,1}) \mathbin{\&}$$

$$(s_{1,1,3} \vee s_{1,1,2} \vee s_{3,2,2}) \mathbin{\&} (s_{1,1,3} \vee \neg s_{1,1,2} \vee \neg s_{3,2,2}) \mathbin{\&} (\neg s_{1,1,3} \vee s_{1,1,2} \vee \neg s_{3,2,2}) \mathbin{\&}$$

$$(\neg s_{1,1,3} \vee \neg s_{1,1,2} \vee s_{3,2,2}) \mathbin{\&}$$

$$\neg s_{2,1,3} \mathbin{\&} \neg s_{3,1,3} \mathbin{\&} \neg s_{3,2,3} \mathbin{\&}$$

$$\neg s_{1,1,1} \mathbin{\&} s_{1,1,3}.$$

This formula says that for given $x = 0$ and for some y, we performed the checking of the property $C(x, y) \equiv x = y$ and concluded that the result of checking is "true". Once the formula is satisfied, we can find y as the original value of the y-cell, i.e., as $y = s_{2,1,1}$.

4 Space-Time Physics Behind the NP-Hardness Result

Space-time assumptions behind the proof. The above proof used two main assumptions about space-time:

- that there is a limitation on the communication speeds, and
- that the volume of a sphere of radius R is bounded by a polynomial of R.

Both space-time assumptions are crucial for the NP-hardness result. Let us show that both space-time assumptions are necessary not just for our *proof* of NP-hardness, but also for the NP-hardness *result* itself.

Indeed, if we do not have any limitations on the communication speed, if we can set up any communication speed with want, then we can exponentially increase communication speed with the increase in the input size, and thus, transform the exponential number of computation steps for an exhaustive-search solution to any NP problem into computations which require a constant time.

Similarly, if the volume of the sphere grows exponentially with the radius r, as $\exp(k \cdot r)$, then we can place exponentially many processors into a sphere, make each processor test one of the exponentially many possible solutions y, and let the processor which finds a solution report to the center. For example, for satisfiability, we have 2^v possible combinations $y = (z_1, \ldots, z_v)$, so to fit 2^v processor, we need a radius R for which $\frac{\exp(k \cdot r)}{\Delta V} = 2^n$, i.e., for which $r = a \cdot v + b$. The resulting time is composed of linear time for testing whether y is a solution, and linear time r/c to communicate the results – so we can solve satifsiability in linear time.

Comments.

- It is worth mentioning that in some physically reasonable models of space-time, we do have such an exponential dependence of the volume on radius, so in these models, we can potentially solve NP-hard problems in polynomial time; see, e.g., [1, 4, 5, 6].

- If the volume of the sphere grows slower than exponentially but faster than polynomially with the radius r, then, by parallelizing exhaustive search, we get an algorithm which is not polynomial, but it is still faster than all parallel algorithms corresponding to Euclidean geometry (in which the volume grows as r^3).

Acknowledgments

This work was supported in part by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721, by Grants 1 T36 GM078000-01 and 1R43TR000173-01 from the National Institutes of Health, and by a grant N62909-12-1-7039 from the Office of Naval Research.

The authors are thankful to all the students from the University of Texas at El Paso graduate Theory of Computation classes, especially to Monica Nogueira, and to all participants of the 2011 International Sun Conference on Teaching and Learning, for valuable suggestions.

References

[1] S. Aaronson, "NP-complete problems and physical reality", *ACM SIGACT News*, 2005, Vol. 36, pp. 30–52.

[2] M. G. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, California, 1979.

[3] O. Kosheleva and V. Kreinovich, "NP-hardness proofs with realistic computers instead of Turing machines: Towards making Theory of Computation course more understandable and relevant", *Abstracts of the 2011 International Sun Conference on Teaching and Learning*, El Paso, Texas, March 10–11, 2011, p. 19.

[4] V. Kreinovich, A. Lakeyev, J. Rohn, and P. Kahl, *Computational Complexity and Feasibility of Data Processing and Interval Computations*, Kluwer, Dordrecht, 1997.

[5] V. Kreinovich and M. Margenstern, "In some curved spaces, one can solve NP-hard problems in polynomial time", *Notes of Mathematical Seminars of St. Petersburg Department of Steklov Institute of Mathematics*, 2008, Vol. 358, pp. 224–250; reprinted in *Journal of Mathematical Sciences*, 2009, Vol. 158, No. 5, pp. 727–740.

[6] D. Morgenstein and V. Kreinovich, "Which algorithms are feasible and which are not depends on the geometry of space-time", *Geombinatorics*, 1995, Vol. 4, No. 3, pp. 80–97.

[7] C. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, Massachusetts, 1994.